DRAFT July 8, 1999

DISSERTATION

INTEGRATION OF PARTIALLY OBSERVABLE MARKOV DECISION PROCESSES AND REINFORCEMENT LEARNING FOR SIMULATED ROBOT NAVIGATION

Submitted by
Larry D. Pyeatt
Department of Computer Science

In partial fulfillment of the requirements
for the Degree of Doctor of Philosophy
Colorado State University
Fort Collins, Colorado
Summer 1999

ABSTRACT OF DISSERTATION

INTEGRATION OF PARTIALLY OBSERVABLE MARKOV DECISION PROCESSES AND REINFORCEMENT LEARNING FOR SIMULATED ROBOT NAVIGATION

This dissertation presents a two level architecture for goal-directed robot control. The low level actions are learned on-line as the robot performs its tasks, thereby reducing the need for the system designer to program for every possible contingency. The actions are adaptive to failures in sensors and effectors, allowing the robot to perform its assigned tasks despite hardware failure. Reactivity, deliberation, and learning are an integral part of the architecture.

The architecture uses a partially observable Markov decision process (POMDP) model for planning, and reinforcement learning (RL) for low level actions. In addition to the robot architecture, this dissertation presents and evaluates a new parallel POMDP solution algorithm and a new algorithm for using decision trees to perform function approximation in RL.

New low level actions may be instantiated with no knowledge of what state transition they are supposed to accomplish. The patterns of reward and punishment cause them to each learn to perform their assigned state transitions. In the event of sensor or effector failure, the low level actions adapt so as to maximize reward even with reduced sensor information or effector availability.

Experiments are conducted in a simulated maze-like environment to compare different versions of the architecture. In the first experiment, hand coded actions are used. The remaining experiments compare the performance of the system using hand coded actions to the performance of the system using learned actions. A final experiment demonstrates that the system can learn a new action that was not pre-specified by the system designer.

The experiments demonstrate that the combination of POMDP planning and reinforcement learning provides a very reactive system that can also achieve long term goals, adapt to failures, and learn new low level actions. In order to demonstrate the robot control architecture, it was necessary to improve or modify existing approaches to reinforcement learning and POMDP planning. The approach to learning low level actions is different from any previous approach, and the experimental results indicate it performs well in the simulated maze-like environment.

Larry D. Pyeatt
Department of Computer Science
Colorado State University
Fort Collins, Colorado 80523
Summer 1999

Acknowledgements

Thanks to Adele Howe, who gave me guidance and encouragement. Thanks also to Tony Cassandra for his help with the POMDP section. Finally, thanks to Kathy for hanging in there.

In Memory of Elmer Lonzo Belk, Jr. and Zelma Routh

Table of Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.1.1 Layered Architecture . . . 3
1.1.2 Single Layer Approaches . . . 4
1.2 Approach . . . 5
1.2.1 Low level Actions . . . 7
1.2.2 POMDP Planning . . . 7
1.3 Research Goals . . . 8
1.4 Experiments . . . 9
1.5 Contributions . . . 10
1.6 Thesis Outline and Summary . . . 10

2 Review of Robot Architectures and Planning **12**
2.1 Sense-Model-Plan-Act . . . 12
2.2 Reactive Systems . . . 12
2.2.1 Subsumption Architecture . . . 13
2.2.2 CHILD . . . 14
2.3 Reactive Planning . . . 14
2.3.1 RAPs . . . 15
2.3.2 Cypress . . . 15
2.3.2.1 PRS . . . 15
2.3.2.2 SIPE . . . 16
2.3.2.3 Combining SIPE and PRS . . . 16
2.3.3 RALPH-MEA . . . 17
2.3.4 GLAIR . . . 18
2.3.5 DSSA . . . 18
2.4 New Forms of Planning . . . 19
2.4.1 Planning with Continuous Execution . . . 19
2.4.2 Planning as a Markov Decision Process . . . 19
2.4.2.1 Completely Observable Markov Decision Processes . . . 20
2.4.2.2 Planning with COMDPs . . . 21
2.4.2.3 Partially Observable Markov Decision Processes . . . 21

2.4.2.4 Planning with POMDPs 22
2.5 Relationship of The POMDP/RL Approach with Past Work 23

3 Review of Low Level Behaviors and Reinforcement Learning 26
3.1 Behavioral Cloning 27
3.2 Neural Control for Low Level Behaviors 28
3.2.1 Manipulation 28
3.2.2 Locomotion 29
3.3 Reinforcement Learning for Low Level Behaviors 29
3.4 Review of Reinforcement Learning 31
3.4.1 Sarsa 33
3.4.2 Q-learning 34
3.4.3 Other RL Methods 34
3.4.4 Improvements to Reinforcement Learning 35

4 Reinforcement Learning of Actions 40
4.1 Environment: Simulated Automobile Racing 40
4.2 RARS Agent Architecture 41
4.2.1 Low Level Behaviors 42
4.2.2 Control Mechanism 43
4.2.3 Experiments 43
4.2.3.1 Experiment 1: Monolithic vs. Decomposed 44
4.2.3.2 Experiment 2: Tuning Learning 45
4.2.3.3 Experiment 3: Adding a New Behavior 47
4.2.4 Results of Initial Experiments 50
4.2.5 Concerns and Questions 50
4.3 Decision Tree Function Approximation in Reinforcement Learning 51
4.3.1 Table Lookup 51
4.3.2 Neural Network 52
4.3.3 Decision Trees 52
4.3.4 Implementation 53
4.3.5 Experiments on Decision Tree Reinforcement Learning 55
4.3.6 Conclusions for Decision Tree Reinforcement Learning 60

5 Review of POMDP Planning 61
5.1 POMDP Basics 62
5.1.1 Problem of Continuous State Space 64
5.1.2 Finding the Value of Possible Next States 66
5.2 Finding Policies For POMDPs 66
5.2.1 Value for a Fixed Action and Observation 68
5.2.2 Value for a Fixed Action 69
5.3 Exact Solution Methods 73
5.3.1 Sondik's One-Pass Algorithm 74
5.3.2 Monahan's Enumeration Algorithm 76
5.3.3 Cheng's Linear Support Algorithm 76
5.3.4 The Witness Algorithm 78
5.3.5 Incremental Pruning 79
5.3.6 Restricted Region 80

5.4 Approximate Solution Methods 80

6 Parallel Restricted Region POMDP Algorithm 82

6.1 Solving POMDPs Using Incremental Pruning 82
6.1.1 Purging Vector Sets 84
6.1.2 Incremental Pruning 88
6.1.3 Restricted Region 88
6.2 Parallel Restricted Region 89
6.3 Experiments and Results on Parallel Restricted Region 91
6.4 Analysis of Parallel Restricted Region 93

7 Combining POMDP Planning with RL Actions 95

7.1 What is Gained by Integration of POMDP and RL 96
7.2 Description of the Robot Architecture 97
7.2.1 Control Loops 99
7.2.2 Low Level Actions 100
7.2.3 High Level Planning 102
7.2.4 Finding New Actions 106

8 Experiments and Results 109

8.1 Simulation 109
8.2 Creating the POMDP Model 111
8.3 Experiment 1: Learning Replacement Actions 113
8.4 Experiment 2: Sensor and Effector Failure 114
8.5 Experiment 3: Generalization 118
8.6 Experiment 4: Learning a New Action 121
8.7 Conclusions About the Experiments 126
8.7.1 Replacing Actions 126
8.7.2 Sensor and Effector Failure 126
8.7.3 Generalization 126
8.7.4 Learning a New Action 127
8.7.5 Execution Speed 127

9 Contributions, Questions, and Conclusion 128

9.1 Robot Architecture 129
9.1.1 Task-Directed 129
9.1.2 Reactive 130
9.1.3 Reliable and Adaptive 130
9.1.4 Generalizing 131
9.1.5 Self Extending 131
9.2 POMDP Solution 132
9.3 Reinforcement Learning 132
9.4 Limitations and Questions for Future Research 132
9.5 Conclusion 134

References 135

Index 145

List of Tables

4.1 Performance during the fourth period. 59

5.1 Immediate rewards for the sample problem. 67

6.1 Test problem parameters. 93
6.2 Results of experiments. 93

7.1 Example transition probability matrices. 106
7.2 Primary and non-primary transitions. 107
7.3 Combined uncovered non-primary transitions. 108

8.1 Number of times the robot failed. 119
8.2 Uncovered non-primary transitions. 122
8.3 Primary transitions for the new action after learning. 124

List of Figures

1.1 System overview. 5

3.1 An agent using reinforcement learning. 32

4.1 Simplified agent architecture for RARS experiments. 41
4.2 Three reinforcement learning networks. 44
4.3 Comparison of three reinforcement learning networks. 46
4.4 Learning performance for the improved learning strategy. 48
4.5 Using the passing behavior to improve performance. 49
4.6 Dividing the state space with a decision tree. 52
4.7 Algorithm for decision tree based reinforcement learning. 54
4.8 Smoothed learning performance on the mountain car problem. 56
4.9 Smoothed learning performance on the pole balance problem. 57
4.10 Smoothed learning performance for RARS. 58

5.1 A graphical representation of belief state. 63
5.2 The new belief state. 63
5.3 A graphical representation of a policy for a two state POMDP. 65
5.4 How the value function partitions belief space. 66
5.5 Value function at horizon one. 67
5.6 The value function can be transformed. 68
5.7 The transformed value function. 69
5.8 Each possible observation has a different transformation. 70
5.9 Partitions for the belief space. 71
5.10 Partitions for the belief space as intersections. 71
5.11 Partitions for the belief space given initial belief state and action. 72
5.12 Combined a_0 and a_1 value functions. 73
5.13 Sondik's first region. 74
5.14 Sondik's second region. 75
5.15 Sondik's third region. 75
5.16 The complete Sondik region. 76
5.17 First vector found by Cheng's algorithm. 77
5.18 Second vector found by Cheng's algorithm. 77

5.19 Third vector found by Cheng's algorithm. 78
5.20 Regions defined in the Witness algorithm. 79
5.21 IP creates all combinations of vectors from the source sets. 79
5.22 The set of vectors is checked to find out which ones are dominant. 80
5.23 The resulting set is combined with the $S(a, z_2)$ set. 80

6.1 White's algorithm for purging a set of vectors. 85
6.2 The two solid vectors are in the set W. 86
6.3 Linear programming test for domination. 87
6.4 The dominate function. 87
6.5 Incremental pruning method for combining the S_z^a sets. 88
6.6 Parallel algorithm for purging a set of vectors. 91
6.7 Each thread communicates with one server. 92
6.8 Server algorithm for parallel purge. 92

7.1 System overview. 98
7.2 Low level actions are implemented as object classes. 100
7.3 Rewards are assigned according to the most likely next state. 101
7.4 State map for a small section of corridor. 103

8.1 Arrangement of Khepera infrared sensors. 110
8.2 Environment and state space for the experiments. 112
8.3 Accuracy of the most likely state improves. 113
8.4 The path used for learning the actions. 114
8.5 Performance of hand coded and learned actions. 115
8.6 Response to gradual failure of all sensors. 116
8.7 Response to intermittent failure of an effector. 117
8.8 Difference between learned actions and hand coded actions. 119
8.9 Location where the robot most often became wedged. 120
8.10 Performance with the new action. 123
8.11 The state assignments in the POMDP model. 123
8.12 Paths followed by the robot. 125

Chapter 1

Introduction

This dissertation presents a two level architecture for goal-directed robot control. Reactivity, deliberation, and learning are integral parts of the architecture. The low level actions are learned on-line as the robot performs its tasks, thereby reducing the need for the system designer to program for every possible contingency. The actions are adaptive to failures in sensors and effectors, allowing the robot to perform its assigned tasks despite hardware failure. New actions can be created and learned on-line under control of the higher level planning system. In addition, the higher level planning system is reactive to failures of the low level actions because, by design, the planner expects failures and provides for contingencies.

1.1 Motivation

This dissertation is concerned with robot navigation in a simple simulated environment. Robot navigation is currently a very active research area, and there are many approaches under investigation. Reinforcement learning is one approach that provides reliability and continuous learning, but does not provide the ability to specify arbitrary goals. Planning can provide this feature, but systems based on planning tend to be brittle and do not adequately deal with failures and unexpected environmental events. The goal of this research is to combine a planning component with reinforcement learning for low level actions, resulting in a system that has the benefits of both approaches while avoiding their weaknesses.

This research relies on simulation of the robot and its environment. Simulation provides an alternative to the mechanical problems, high costs, and long turnaround times associated with currently available robot technology. Simulators historically have not represented the real world with a sufficient degree of accuracy to allow lessons learned from simulation to be used reliably on a hardware robot. However, simulators are becoming increasingly detailed and sophisticated. With increases in computing power, it is now feasible to build simulators that attempt to accurately model robotic sensors, effectors, and the environment. I have selected two simulators to use in my experiments: RARS and Khepera. The simulators were chosen for several reasons.

Execution speed: Reinforcement learning may take many trials to learn a low level action. Higher execution speed in the simulator means that the experiments take less time to run.

Stochastic simulation: In the physical world, there is always some uncertainty and stochastic simulation helps to model this uncertainty.

Level of detail: The simulator should provide the same type of sensor data and effector control as would be available on a real robot, while expending as little computational resources as possible. If the simulator does not accurately reflect the physical world, then there is no guarantee that any agent architecture developed in simulation will be useful in the real world.

The Robot Auto Racing Simulator (RARS) (Timin 1995) is designed to allow robotic cars to compete in automobile races. I performed several experiments in this domain to test ideas on reinforcement learning for low level actions and ways to combine them. Section 4.1 describes the simulator in detail.

Khepera is a small, modular robot that includes a simulator for development. The basic system has two drive wheels, a processor, and eight light sensors. Other components can be added, such as a gripper or CCD camera. The low number of sensors allows fast simulation and reduces the amount of information which must be processed by the controller. However, it also limits the ability of the robot to make subtle distinctions in its environment. This could lead to difficulty in the robot knowing where it is, so reliable navigation with this robot is difficult. Section 8.1 describes the simulator in more detail.

Considerable current research in robotics is aimed at developing reliable, generalizable robot control architectures for navigating in an office or maze environment. This dissertation presents an approach to solving this problem. The design criteria for my robot control system are that it must be:

Task-Directed: able to accept a task or goal and then work to achieve that task or goal. Changing the goal should not require learning or changing the control program.

Reactive: able to react quickly to sensor information and select an appropriate action according to the situation.

Reliable: high probability of success at achieving the goals that are assigned, even when its own hardware fails or when its knowledge of the world is inaccurate.

Adaptive: should adapt to long term changes in the environment.

Generalizing: able to apply knowledge learned in one situation to another similar situation.

Self Extending: able to add new actions to take advantage of knowledge about the environment.

The performance of the robot architecture will be measured against these criteria.

Some of the most successful robot control systems today use a multi-level approach where the lower levels contain short term reactive mechanisms, while higher levels are responsible for long term planning and goal attainment (Hasemann 1995; Brooks 1991a; Smithers and Malcolm 1987). Each level can be viewed as a layer of control that is built on top of the layer beneath it. Higher layers work at a high level of abstraction, while lower layers deal more directly with the hardware and the environment.

Quick and correct reaction is essential to maintain safety of a robot in its environment. If the robot cannot react quickly to events and situations in the world, it may damage itself

and anything else in its path. Purely reactive systems make decisions based on current sensor information. This allows for fast decision making, but does not always result in good long term decisions and goal attainment.

Deliberation is also important for goal-directed robot control. Without some form of long term planning, the robot is unlikely to accomplish anything useful and cannot readily accept new goals. Traditional planning approaches require a lot of processing time, and work best at a high level of abstraction, making them unattractive for low level reactive control. However, planning systems generally include a discrete or symbolic notion of state, which may be lacking in a purely reactive system. This notion of state allows the robot to plan sequential tasks in order to reach long term goals (Mataric 1992a). Some robot control systems that use only reactive mechanisms can provide fixed goal-seeking behavior, but *changing* the goal in these systems is difficult (Dorigo and Colombetti 1998; Mataric 1992a).

1.1.1 Layered Architecture

Taken by themselves, neither reactive control nor planning seem to be sufficient for robotic control applications. A general consensus has emerged that some kind of layering is required in a general goal-seeking robot control architecture (Zelek and Levine 1998). The motivation behind the multi-level approach is to exploit the benefits of both reaction and planning while avoiding their individual weaknesses (Pyeatt and Howe 1998). Providing adaptation, learning, reactivity, and deliberation is the goal of most robot control architectures. However, few architectures address all of these goals (Zelek and Levine 1998; Dorigo and Colombetti 1998). High reliability of low level actions is often seen as a necessity in robot control systems, but high reliability is difficult to achieve (Dorigo and Colombetti 1998). Most robot control systems with fixed actions tend to be brittle and not adaptive to hardware failures and changes in the environment (Nehmzow 1994).

It is not always possible to guarantee that an action will succeed, or to know *a priori* what low level actions the robot will need (Dorigo and Colombetti 1998; Mataric 1992a). A robot has to deal with problems both in the environment and with its own sensors and effectors. Sensors and effectors can become unreliable or cease to function altogether, making it difficult for the robot to determine the state of the world and to affect the world in a predictable manner. Approaches to robot control that rely on accurate sensor information and reliable action execution often perform poorly when the sensors and effectors malfunction. Ideally, the robot should adapt and perform well, whatever the situation (Ferrell 1994). The capability for quick reaction to unexpected environmental factors and adaptation to failed or unreliable sensors and effectors should be built in from the start, not added as an afterthought (Smithers and Malcolm 1987; Smithers and Malcolm 1988).

In addition to simply adapting to failure and reacting to the environment, the robot should learn new ways to accomplish the tasks that are assigned (Dorigo and Colombetti 1998). This ability to learn new actions should be a basic requirement of the robot control architecture. Previous research in learning low level actions has either been constrained to learning actions that are pre-specified by the system designer or has relied on a fixed set of actions and concentrated on finding ways to utilize those actions (Mahadevan and Connell 1992; Bagnell, Doty, and Arroyo 1998; Kontoravdis, Likas, and Stafylopatis 1992; Singh 1991; Singh 1992b; Ring 1991; Ring 1993a; Dorigo and Colombetti 1998).

The robot control architecture described in this dissertation uses a mapping of the environment that is inspired by the map that is strongly suspected to exist in biological

systems. There is evidence that biological systems rely on different groups of neurons to perform various tasks (Fischer and Lazerson 1994). Research in psychology and cognitive science supports the position that biological systems make a distinction between knowledge about how to affect the world and factual knowledge that describes the world (Anderson 1983; Fitts and Posner 1967; Keil 1989; Sun 1995).

The brain appears to be organized as a hierarchical system composed of interconnected structures that have very different purposes (Coren, Porac, and Ward 1984). For instance, a large body of evidence has been accumulated to support the notion that biological systems create a map of their environment. The discovery of *place cells* in areas CA3 and CA1 of the rat hippocampus (O'Keefe and Dostrovsky 1971) has led to the idea that the hippocampus acts as a *map space* (O'Keefe and Nadel 1978). Cells that fire as a function of the animal's orientation in space have also been evidenced in various other regions of the rat brain (Taube, Muller, and Ranck 1990). Some recent computational models of animal navigation consider the hippocampus as a hetero-associative network (Trullier and Meyer 1997; Schmajuk and Thieme 1992; Muller, Stead, and Pach 1996) that learns the relationships between neighboring but distinct places and learns how places are connected to one another. Section 7.1 describes the map used in this architecture.

1.1.2 Single Layer Approaches

Researchers have experimented with purely reactive systems based on reinforcement learning or genetic algorithms. They have developed systems that exhibit good reactivity and adaptation, but lack the ability to pursue dynamically specified goals (Gruau and Quatamaran 1996; Singh 1991).

A monolithic single layer approach to goal-directed robot control using reinforcement learning is not practical for several reasons. Reinforcement learning methods do not scale efficiently to large tasks, so breaking down the goal into sub-tasks and learning each task independently appears to be the most viable approach (Singh 1992b). Some of the main bottlenecks are *temporal resolution* (expanding the unit of learning from the smallest possible step in the task), *division and conquest* (finding smaller subtasks that are easier to solve), *exploration*, and *structural generalization* (generalization of the value function between different locations) (Singh 1992a). One way to extend reinforcement learning to perform multiple tasks is to use different modules to learn the solutions of the different tasks (Anderson and Hong 1994). This indicates the need for some mechanism, such as a layered architecture, to compose the tasks.

In a single layer monolithic approach using reinforcement learning, the ability to accept goals from an outside entity requires some sort of input encoding of the current goal. As more goals are added to the system, more inputs are required. The learning time for reinforcement learning does not scale well with the number of inputs (Dayan and Hinton 1993), and so far, no reinforcement learning algorithm has demonstrated the ability to dynamically add inputs, as would be necessary for a robot that can learn new goals.

Organisms with short lifespans learn (adapt) through evolution. For instance, the individuals in many species are born with all of their responses hard-wired and do not learn (adapt) much during their lifetimes. However, because they reproduce prolifically, the species is able to adapt its hard-wired reactions fast enough to keep up with environmental changes. Species with longer lifespans and lower reproduction rates must rely on learning (adaptation) in each individual (Fischer and Lazerson 1994).

Most single level genetic algorithm approaches to robot control have been limited to

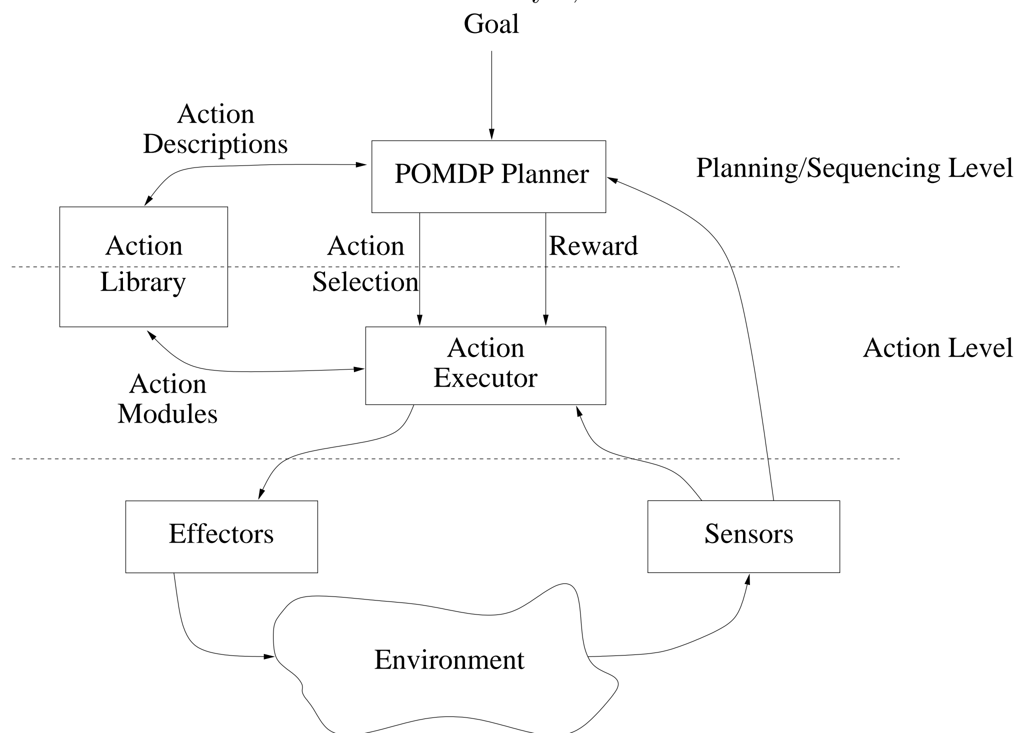


Figure 1.1: System overview.

insect-like actions. Several systems have been demonstrated that are capable of walking, chasing prey, feeding, and performing other complex actions (Gruau and Quatamaran 1996). Although limited adaptation has been demonstrated, these approaches lack the ability to perform long-term planning and goal attainment where the goal changes during the life of an individual. Genetic programming has been used to evolve simulated robots for playing soccer (Luke and Spector 1996; Luke 1998). Low-level actions were hand coded, and genetic programming was used to generate rules for selecting the actions. Although very successful for playing soccer, this approach did not result in taskable robot control. More powerful genetic encodings, which allow for the creation of layered structures of processing elements, rather than specifying each processing element, may help allow genetic algorithm approaches to incorporate learning and adaptation in an individual as well as in the species.

1.2 Approach

I have developed a two level robot control system that provides goal-driven robot navigation and the ability to learn new low level actions as needed. The lower level provides actions, while the higher level composes actions to achieve goals. Training and learning are intrinsic to both levels of the architecture.

A general architecture could conceivably require more than two levels. However, two levels are adequate to show the capabilities of this approach. What differentiates this research from previous approaches is the way that planning is performed and the way in which learning and adaptation is supported.

The robot control architecture uses *actions* for low level reactive control. The robot is

performing an action when it uses its effectors to change the world state. In many robot control architectures, an action is simply a set of control signals sent to the effectors. Actions are often seen as discrete events that cause an immediate change at the next time step, but in my architecture, an action may be executed for several time steps and may use sensory information and/or short term memory to represent the current state.

An action without short term memory has no explicit representation of state, but may have implicit state. In this architecture, the higher level planner does have explicit state. At each level, the notion of state is different. Actions are general enough to be used from a variety of initial planning level states and result in many possible final states. The important feature of an action is how it *changes* the world state. Several actions which make changes in the world state can be applied serially to an initial state in order to reach a goal state. *Planning*, in this framework, is the act of selecting and ordering which actions to activate to achieve long term goals.

An overview of the system is shown in Figure 1.1. The architecture is covered in more detail in Chapter 7. The two levels are intended to encapsulate two types of knowledge and operate within two different time frames. The major components of the system are:

POMDP Planner The Planning Level implements procedural knowledge by using a partially observable Markov decision process (POMDP) planner to generate a policy for selecting the appropriate low level action for every possible situation. POMDP planning is a form of reinforcement learning (Sutton and Barto 1997a). A review of POMDPs is given in Chapter 5. The planning level is meant to operate over a longer time period than the individual low level actions. POMDP planning was chosen primarily because re-planning is not needed if a low level action fails to perform as expected. The system simply selects another action and continues execution. In other words, plan execution is reactive. The actions are stored in a library, and the POMDP planner uses the *descriptions* of the available actions during planning.

Action Executor: This module performs low level actions that are applied over a short time period. The actions are responsible for changing the current state by sending control signals to the effectors. The action executor loads the *implementation* portion of the required action from the library and executes it. If learning occurs during execution, the action executor stores the updated action implementation back in the library for future use. The low level action has no notion of a goal, and its view of the world is based on very low level sensory data. An action simply maximizes the reward that it gets from the planner. Reinforcement learning is ideally suited for such requirements. An overview of reinforcement learning is given in Section 3.4.

The division into two layers allows each module to work at an appropriate level of abstraction and provides both reactive control and deliberation. In a sense, POMDP planning is both reactive and deliberative. Deliberation is performed to create a policy from the POMDP model of the world, and reactive execution is performed to implement the policy. The ability for POMDP planning to represent the world as a discrete state space is critical for supporting learning at the low level and provides a convenient method for the specification of goals. New potential goals can be added simply by adding states to the POMDP.

1.2.1 Low level Actions

Psychologists distinguish between innate reflexes, over which the organism has only slight control, and *actions* which are organized abilities under voluntary control that allow flexible behavior that can be affected by rewards and punishments (Fischer and Lazerson 1994; Bruner 1973). I define actions similarly in the robot control architecture. This is slightly different than the definition that is used for behavior in behavior based robotics, which does not necessarily include rewards and punishments.

One major goal of this research is to show that the robot can begin with few actions and learn actions as necessary. The system does not require the programmer to specify a set of possible actions, because that information is implicitly embedded in the POMDP model. The POMDP model is a combination of the state map that is provided by the programmer and the transition probabilities that are learned by the robot. Other systems have composed low level actions in order to reach long term goals, and some systems have been able to learn pre-specified actions. My work goes one step further so that actions are specified and learned automatically as they are needed. Chapter 7 explains how this is accomplished.

In Singh's (1991, 1992b) work, the system learned to use pre-programmed actions to perform useful functions. My work is similar in spirit to Singh's approach, but does not allow for the *creation* of new actions. All actions in Singh's approach had to be explicitly defined before learning could occur. Thus, the agent was not able to extend its set of actions. The approach of Anderson and Hong (1994) did not require that all actions be pre-defined, but did not include a planning component, so the goal state could not be arbitrarily specified. Their experiments evaluated a modular reinforcement learning approach for solving the pole balancing problem, but give no indication of how well the approach would work in robot navigation. My architecture allows the robot to have a great deal more control over what it learns at the low level while still allowing an arbitrary goal state to be specified at any time.

1.2.2 POMDP Planning

The task of the POMDP planner is to create a policy for reaching the goal while dealing with uncertainty. Thus, it is a form of *model-based* reinforcement learning (Sutton and Barto 1997a). Mataric (1992b), Dean et al. (1993), and Koenig et al. (1996) have shown that POMDPs provide a good approach for meeting the requirements for the planner level in a robot. The POMDP model explicitly represents uncertainty in the execution of actions, uncertainty in the sensors, and approximate knowledge of the environment and the location of the robot. The fact that it is based on an underlying Markov model simplifies the tasks of tracking the world state and generating reinforcement for the low level actions.

POMDP planning requires a topological map of the environment. The topological map represents each possible position and orientation for the robot. Some researchers have shown how such maps can be learned by the POMDP planner (Mataric 1992b; Zimmer and von Puttkamer 1994). A hierarchical map can be learned on-line and used to focus attention and reduce the number of states that must be considered at one time (Donnart and Meyer 1996). The ability to learn the topological map is not the focus of this dissertation, so in my experiments, a topological map is given to the robot.

POMDP planning allows only one action to be active at any time and that action is responsible for controlling the robot. It would be more difficult to allow multiple actions

to be active at the same time as is done in some other architectures such as Subsumption (Brooks 1991c; Brooks 1991b). Multiple simultaneous actions are difficult in the robot control architecture due to the problem of assigning rewards and tracking the current state. Allowing only one action per time step is not a severe limitation if each action is allowed to control several effectors simultaneously.

Each time an action is executed, the POMDP planner tracks the actual state transition that occurs. There may be no state transition in the library that could "explain" the transition that actually occurred. In this case, it is safe to assume that an action could be created specifically to cause the unexplained state transition. In this way the robot can learn more efficient low level actions and the POMDP planner can learn to use those new low level actions. The mechanism for learning actions is described in Chapter 7.

The architecture developed in this research should be general enough to be used in a wide range of robotic applications, including mobile robots and robotic arm assemblies. The only requirement is that the domain can be expressed as a discrete state space that can be modeled as a Markov decision process. Most real world domains fit this requirement; the robot designer has a great deal of flexibility in modeling the environment.

1.3 Research Goals

> "An artificial system able to learn an appropriate behavior by itself is the dream of any developer."
>
> *Marco Dorigo and Marco Colombetti*
> *"Robot Shaping," 1998*

The goal of this research is to develop and demonstrate a system which is capable of generating and using new low level actions in a small robot navigation domain. Adaptation and learning at the low level are integral to the robot architecture, and adaptation at the planning level is also supported. The central questions for the system as applied to this domain are:

- What forms of reinforcement learning are relatively insensitive to learning parameters, and can be scaled up to work as a component in the architecture? In many reinforcement learning systems, the human investigator can tune the learning parameters, but in the robot control architecture, the system will be generating its own parameters for each reinforcement learning action. It is important to have a reinforcement learning implementation that will reliably learn with a very broad range of learning parameters. Since action creation and learning is under control of the POMDP planner, there is no opportunity for the programmer to adjust the learning parameters.

- Does POMDP planning scale to a small robot domain? POMDP planning is a relatively new area of research. Previous work has shown promise, but there are problems associated with scaling POMDP solution methods to even small real-world domains (Cassandra, Littman, and Kaelbling 1996).

- Does the combination of POMDP planning and with low level actions result in reliable navigation in the small robot domain?

- Is the combined POMDP/RL system more reliable than the same system using hand coded actions? How do learned actions compare with hand coded actions when dealing with imperfect sensors, uncertainty and failures in effectors and sensors? Reinforcement learning provides a mechanism for adapting to failures and changes in the

environment, but how well will the adaptation work under the control of a higher level planner that may be uncertain about its state in the environment? The same failures that necessitate adaptation at the low level also degrade the ability of the planning level to determine where it is in the world, and therefore, its ability to assess how well a low level action has performed its function.

- How well do the learned actions generalize to work in states that they were not trained in?
- Can the system discover and learn new actions? How does learning new actions affect the efficiency and reliability of the robot? An action is useful if it allows the robot to reach goals that could not be reached without the action, or if it allows the robot to reach a goal more efficiently than it could without the action.

1.4 Experiments

The research goals articulated in the previous section motivate the experiments. The experiments described in Chapter 4 answer the first question: What forms of reinforcement learning are relatively insensitive to learning parameters, and can be scaled up to work as a component in the architecture? Experiments with reinforcement learning using the Robot Auto Racing Simulator (RARS) (Timin 1995) were used to explore these questions. At first, I used table lookup and standard backpropagation to estimate the value function. Table lookup is relatively insensitive to learning parameter settings, but does not scale well with the number of inputs. Standard backpropagation scales well, but the experiments showed that it is very sensitive to learning parameters and convergence cannot be guaranteed. Experiences with table lookup and backpropagation prompted me to explore an alternative approach using decision trees to approximate the value function. This method is much less sensitive to learning parameters than the backpropagation approach and scales much better than table lookup. The results of experimentation showed that the decision tree approach provided the right mix of reliable learning and scalability.

After finding a suitable form of reinforcement learning I began experiments on POMDP planning. The major problem associated with POMDP planning is that it is extremely computationally expensive. It was necessary to find a way to generate a policy for problems that are much more difficult than those typically found in POMDP literature. I developed a parallel algorithm, coupled with some minor improvements to the best known POMDP solution algorithm. I ran experiments using some standard POMDP problems in order to verify that the parallel POMDP solver is faster than previous algorithms. The parallel algorithm is adequate for planning in a small, but non-trivial domain. This is enough for testing the architecture. With the POMDP planner working, I ran experiments to get a baseline on performance using a fixed set of pre-programmed low level actions using the Khepera simulator. POMDP planning with hand coded low level actions was found to be very reliable and could scale to a small but non-trivial robot domain.

The first two phases of the project dealt with creating and testing the components for the two level system. In phase three, I combined them to show that the resulting system was reactive, reliable, adaptive, and generalizable. I replaced the pre-programmed actions from phase two with reinforcement learning modules to provide the same set of actions. Then, I measured the performance of both systems under a variety of conditions.

A final set of experiments was conducted to answer the remaining question: Can the system generate useful actions? The POMDP planner was extended to evaluate state transition probabilities that are learned as the system operates. The extended planner looks for transitions that are not explained by any existing low level action. When such a transition is found, a new action module is created and associated with the new transition. The new action module is trained on-line to perform the new transition, thereby extending the set of low level actions available to the POMDP planner.

1.5 Contributions

The most significant contributions are the new reinforcement learning and POMDP algorithms presented as part of this dissertation. The decision tree function approximation approach to reinforcement learning algorithm provides scaling that is superior to table lookup with convergence superior to backpropagation based approach using sigmoidal activation functions. Other researchers are currently extending my decision tree approach to value function approximation.

The POMDP solution algorithm is capable of solving larger problems than were previously possible. In addition, my implementation of an exact POMDP solver has verified the only other existing exact POMDP solver, written by Cassandra (1998a), and will provide a testbed for POMDP researchers to experiment with alternative algorithms.

Secondly, the experiments demonstrate that the robot control architecture meets its design criteria for the domain in which it was tested. The third contribution of this dissertation work is an approach to learning adaptive low level actions. It is the *combination* of reinforcement learning with POMDP planning that results in this capability. Both components provide reactivity at different time scales. This combination provides a very reactive system that can also achieve long term goals, adapt to failures, and learn new low level actions. In order to demonstrate the robot control architecture, it was necessary to improve or modify existing approaches to reinforcement learning and POMDP planning.

1.6 Thesis Outline and Summary

Section 1.2 presented a high level view of the architecture that was developed for this dissertation. Details are left for later chapters, since full understanding of the architecture requires some knowledge of topics covered in Section 3.4 and Chapter 5. The robot control architecture can be viewed as a framework for a specific implementation. Later chapters discuss the details of the current implementation, and cover the architecture in more detail. The remainder of this dissertation is organized as follows:

Chapter 2 reviews some of the robot architectures that have been developed previously and relates them to the work in this dissertation. It discusses the methods used in those architectures to implement low level actions and the methods used to combine the levels. This is no means a complete listing of all robot architectures, but covers some that have characteristics in common with the work in this dissertation. All of the architectures covered try to solve the opposing goals of reactivity and deliberation, with varying degrees of success. This chapter provides background material and motivation for the robot control architecture. Section 2.4 covers some approaches to planning under uncertainty. Again, no attempt is made to review every approach in the literature, but only to give an overview of some of the more notable planning approaches. This section provides background material

on planning as a Markov decision process and provides justification for using a Markov model approach.

Chapter 3 gives a review of some approaches to implementing low level actions. It discusses the strengths and weaknesses of these approaches and provides justification for why low level actions were implemented using reinforcement learning. Section 3.4 provides an in-depth review of reinforcement learning, and motivates the work in decision tree function approximation for reinforcement learning. This section covers basic reinforcement learning theory and presents the two most common approaches, Sarsa and Q-learning. Section 3.4.4 discusses some of the problems that can be encountered with reinforcement learning and describes some of the methods that have been proposed to alleviate these problems.

Chapter 4 presents a new decision tree based approach to reinforcement learning. This chapter presents some experiments and results comparing decision tree, table lookup, and backpropagation based reinforcement learning.

Chapter 5 provides some necessary background on Partially observable Markov decision processes. The POMDP model is explained, followed by descriptions of the known exact solution algorithms. Some approximate methods are also covered briefly. This section gives background material used to explain the robot control architecture. It also highlights the difficulties involved in finding solutions to large POMDP problems.

Chapter 6 covers a new parallel algorithm for finding exact solutions to POMDP problems. This algorithm modifies a previous algorithm to run efficiently in a coarse grained parallel environment. This algorithm enabled me to find exact solutions to POMDP problems that were too large to be solved with any previous exact algorithm. Although the problems that can be solved are still considered too small to be of any practical use in robotics, the parallel algorithm allowed me to demonstrate the architecture in a small, but non-trivial domain. This chapter presents a performance comparison between the parallel algorithm and the non-parallel version for solving several standard POMDP problems.

Details of how the robot learns low level actions are presented in Chapter 7. This chapter explains how learning is made possible by the characteristics inherent in the planner and actions. All of the background material that was covered in previous chapters is drawn together in this chapter to form a complete picture of the robot architecture.

Chapter 8 presents the experiments and results. This chapter covers experiments and results with both levels of the architecture in the Khepera robot simulator.

Significant contributions of this work and some of the issues that it addresses are examined in Chapter 9. The architecture presented in this dissertation is compared to previous approaches to robot control and the relative strengths and weaknesses are assessed. This chapter also discusses significant contributions to POMDP solution methods and to function approximation in reinforcement learning. Finally, Section 9.5 presents some of the remaining issues and gives suggestions for possible future work. The approach to learning low level actions that is presented in this dissertation is different from any previous approach, and the experiments show that it performs well. The major hurdles to be overcome are in scaling the action and planning levels to work on a larger robot in a more complex environment.

Chapter 2

Review of Robot Architectures and Planning

An architecture is a description of the basic components that make up a system and and how those components work together to form the whole (Dean and Wellman 1991). This chapter covers some extant robot architectures and compares them with the architecture presented in this dissertation.

2.1 Sense-Model-Plan-Act

The earliest robotic architectures used a sense-model-plan-act approach (Nilsson 1984; Doshi, Desai, Lam, and White 1988). In this paradigm, information from sensors was gathered and used to build a model of the current world state. Building the world model usually involved feeding the sensory data to a semantic network which, after a great deal of computation, produced a detailed symbolic description of the world state. Once the model was constructed, a planner was used to create a step by step procedure for reaching the desired goal state. Finally, the plan was executed. Planning was performed using this detailed description, which also consumed a lot of computation. Due to the amount of computation required, several minutes or even hours could elapse between the time the robot sensed its environment and when it began to act on the information it had gathered.

The sense-model-plan-act paradigm assumes a predictable or static world between sensing and acting. This assumption is usually not valid for real-world environments, which can change while the robot is modeling and planning. By the time the robot acts, the world can be very different from the one that the sensory data indicated. As a result, these early robots had to move very slowly, re-planning after every step. Most attempts to improve this method of robot control consisted of speeding up the sense-model-plan-act loop.

2.2 Reactive Systems

Robots based on the sense-model-plan-act paradigm were able to perform reasonably well only in static domains and were very slow to act. In addition, the large computational requirements precluded the use of on-board processing. Clearly, a new paradigm was needed that could react in a timely fashion to changes in the environment.

2.2.1 Subsumption Architecture

In 1986, Rodney Brooks published a landmark paper on the *subsumption architecture* (Brooks 1986), which introduced a fundamentally different approach to robot control. The subsumption approach allows modeling of tight sense-act loops as finite state machines. While early approaches to robot control started with planning, subsumption starts at the other end and attempts to create intelligence by combining basic behaviors. Brooks (1991a) reasons that the artificial intelligence community need not attempt to build "human level" intelligence into machines directly from scratch. Citing evolution as an example, he claims that we can first create simpler intelligences, and gradually build on the lessons learned from these to work our way up to more complex behaviors.

The first set of behaviors in a robot might provide the ability to avoid collisions by moving away from approaching objects but otherwise standing still. A higher level behavior might cause the robot to move in a given direction. This behavior dominates the obstacle avoidance behavior by suppressing its output unless an obstacle is detected. In this way, higher level behaviors are built up, subsuming the function of lower level behaviors. The result was a robot that can move around for hours without colliding with anything.

The design of the subsumption architecture has been influenced by a philosophy of "no global world models" and "no traditional AI planning." The underlying architecture is distributed, with no shared memory or global communication network. The subsumption architecture is based on networks of behaviors that are each implemented as an augmented finite state machine (AFSM). Each AFSM has a number of states and a set of input and output ports. Each input port has a buffer register that always contains the most recently received message on that port. The networks are built by wiring output ports of AFSMs to inputs of others. Messages are sent over these wires. Each AFSM changes its internal state according to the messages it receives and possibly generates messages, which are sent along its outgoing wires.

The subsumption approach allows a decomposition of the world model. Each behavior can model the part of the world that it needs in parallel with the other behaviors. This resulted is faster sense-act cycles, allowing robots to move much more quickly than is possible with the sense-model-plan-act paradigm. Due to the success of the subsumption architecture, the focus of most robotics research shifted to sensing and acting, and the emphasis on planning was reduced.

When designing robots with a number of behaviors, the subsumption architecture must be able to decide which behaviors are appropriate at a given time and, from these, which one should be selected. Brooks (1991b) describes a system of activation which is modeled upon animal hormone systems. In this system, some of the AFSMs of the basic subsumption architecture layers can be halted or inhibited by the presence or absence of a hormone. This computational mechanism is designed to modulate the robot's behavior by thresholding the layers of AFSMs and preventing the activity of those levels below the threshold. The hormonal mechanism leads to integration of behaviors in the subsumption architecture.

Mataric (1992a) points out that the distributed nature of the architecture allows it to scale well. Although not yet demonstrated, the number of layers in the subsumption architecture and the complexity of the resulting behavior is probably limited (Brooks 1991c), since the human designer can only manage a limited amount of complexity in the interactions between behaviors. The addition of hormones increases the size of the system linearly, but there is a limit on the intersections between hormones, effectively limiting the system's overall size.

Subsumption architecture demonstrates the importance and power of low level behaviors. Experiments with the subsumption architectures also underscore the need for some higher level coordinating mechanism. Subsumption architecture does not allow for learning of behaviors directly, so the system designer is responsible for anticipating every possible situation.

Subsumption is one type of *behavior-based* robot architecture. Though behavior-based robot architectures are varied, they share several common features, including (Arkin 1998):

- emphasis on the importance of coupling sensing and action tightly,
- avoidance of representational symbolic knowledge, and
- decomposition of behaviors or situation-action pairs into contextually meaningful units.

Some other behavior-based approaches include motor schemas (Arbib 1992), circuit architecture (Rosenschein and Kaelbling 1987), colony architecture (Connell 1989), and animate agent architecture (Firby 1995).

2.2.2 CHILD

Temporal Transition Hierarchies is a neural network architecture that learns by adding new nodes to a basic network (Ring 1993a; Ring 1993c; Ring 1993b). The new nodes dynamically adjust weights in the existing network. New nodes take their inputs from some time in the recent past, instead of the current inputs, allowing this architecture to operate in non-Markovian domains. A Markovian domain is one where the optimal decision can be made based only on the current state.

A Temporal Transition Hierarchies network learns very quickly when compared to recurrent neural networks on supervised learning benchmarks. They have no hidden units in the traditional sense, and the activation functions are linear. However, the network architecture still allows it to compute non-linear mappings as weighted sums of linear functions.

Continual, Hierarchical Incremental Learning and Development (CHILD) (Ring 1994) combines Temporal Transition Hierarchies with Q-learning. At each step, the robot's sensory input from the current state is given to the transition hierarchy network which produces a value for each possible action. The network's response at the previous time step is modified according to the maximum action value from the current time step. This system is designed to learn continually and adapt to new situations and environments. New hierarchies are added for each skill or behavior, and new behaviors can be constructed from old ones by building onto extant hierarchies. This allows the robot to encapsulate skills from early learning into a foundation for later learning.

CHILD has been shown to work well for navigating a maze-like environment where the inputs at each time step may not uniquely determine the current state (Ring 1993c; Ring 1994). The ability to use inputs from previous time steps helps determine the actual state and choose an appropriate action. CHILD demonstrates that reinforcement learning can be used to solve problems in non-Markovian domains.

2.3 Reactive Planning

Although they paved the way for several years of progress in robotics, the reactive approaches were seen as insufficient by some researchers. The main problem was the lack of

long-term planning for goal attainment. However, lessons with the sense-model-plan-act paradigm showed that planning for robotic applications needed to be re-thought to provide better performance and reaction to changes in the environment. One common approach has been to plan at different levels of abstraction. The lower levels provide short-term reactive actions and hide the details from the higher levels, which focus on achieving long term goals. This approach to combining the best features of planning and reactive systems is known as reactive planning.

2.3.1 RAPs

Reactive Action Packages (RAPs) (Firby 1989) are an approach to reactive planning for robot control. RAPS allow the high level planner to specify tasks, instead of actions, resulting in more robust plans. A RAP groups together and describes all of the known methods to carry out a single task in different situations. The higher level planner creates plans as a sequence of RAPs to be used. This allows a plan to be created without specifying what low level actions are to be performed. As each RAP in the plan is selected for execution, the situation surrounding the robot is used to index into the RAP and select the most appropriate method.

Execution monitoring and local re-planning are integral to the RAPS system. Firby argues that both capabilities are necessary primarily because planning and primitive action execution take place at different levels of abstraction. A primitive action instructs the robot hardware to perform a well-defined, concrete action in the real world and, thus, requires an appropriate initial state. On the other hand, the purpose of planning is to take very high level goals and produce sequences of simpler tasks to achieve those goals. A common mistake in early robot control systems was in asking the planner to generate task sequences at the level of primitive actions. In complex, dynamic domains, planners cannot produce accurate plans in such detail; therefore, inappropriate actions will inevitably be included. RAPs are a mechanism for reducing the probability that the plan will fail by keeping the plan at a high level of abstraction. In essence, the system performs planning at a high level and selects from compiled plans at the low level. A system that uses RAPs qualifies as a reactive planning system since it provides a mechanism for quickly selecting among different primitive actions.

2.3.2 Cypress

Wilkins et al. (1995) developed Cypress, a hierarchical planning system based on SIPE and PRS. In this system, PRS is used for reactive execution of plans generated by the SIPE deliberative planner, and can invoke SIPE to re-plan when necessary. PRS performs low level planning of actions to meet the goals specified by SIPE. The potential search space is reduced by forming abstract plans in SIPE and expanding these into more detail in PRS. Thus, SIPE avoids the detail encountered at the lowest level of planning.

2.3.2.1 PRS

The Procedural Reasoning System (PRS) (Georgeff and Ingrand 1989b; Georgeff and Ingrand 1989a) is a hybrid system for reactive planning. PRS has four major components:

1. a database called the World Model containing current beliefs and facts about the world,

2. a set of current goals to be realized,

3. a set of plans describing how certain sequences of actions and tests may be performed to achieve given goals or to react to particular situations, and

4. an intention structure containing those plans that have been chosen for eventual execution.

An interpreter selects appropriate plans based on the World Model, places those selected on the intention structure, and executes them. The system interacts with its environment through its World Model which acquires new beliefs from changes in the environment and through its own actions.

PRS is designed to be a general purpose reasoning system particularly suited for use in domains in which there are predetermined procedures for handling the situations that might arise. PRS integrates traditional goal-directed reasoning and reactive behavior. The Procedural Reasoning System continuously tests its decisions against its changing knowledge about the world and can redirect the choices of actions dynamically while still pursuing long term goals. One important aspect of PRS is the fact that it monitors its current state and detects unexpected deviations.

2.3.2.2 SIPE

System for Interactive Planning and Execution monitoring (SIPE) (Wilkins 1984) is a classical hierarchical planner. The decision to implement planning at different levels of abstraction is based upon the assumption that by avoiding the detail encountered at the lowest level of planning, the potential search space can be reduced by forming abstract plans and expanding these into more detail as needed. SIPE uses an execution monitor to verify that the plan steps execute correctly and to select new steps by re-planning. SIPE detects and corrects for interactions occurring in the plan by using *plan critics*. These critics, and their associated solvers, may use the re-planning actions provided by the execution monitor to modify plans. This allows plans to be modified without performing complete re-planning.

2.3.2.3 Combining SIPE and PRS

Cypress integrates traditional planning (SIPE) with reactive control (PRS). Cypress differs from previous approaches in that it

- allows for the generation and execution of complex plans with parallel actions,
- integrates goal-driven and event-driven activities during execution,
- uses evidential reasoning for dealing with uncertainty, and
- uses re-planning to handle run-time execution problems.

The re-planning capabilities are of particular note, as Cypress is the first system that supported asynchronous run-time re-planning with a general purpose generative planner. While Cypress does not address all aspects of activity in dynamic and uncertain environments, it does provide a powerful system for defining goal-directed, reactive robots that can operate successfully in challenging domains. Cypress gains a lot of power from the use of PRS, which allows the planner to remain at a relatively high level of abstraction.

Cypress demonstrates how a multi-level planner can be constructed from two existing AI technologies. One possible problem with this approach is that it is difficult to re-plan while plan execution is taking place. The currently executing plan may change the world state in some way that is not expected or desired in the new plan. This could lead to an immediate failure in the new plan and require re-planning again. Meanwhile, execution continues, possibly changing the world state again in a way not expected or desired by the new plan. This cycle could continue indefinitely, with the planner always trying to catch up to the current world state. The planner could spend all its effort trying to deal with changes in the environment caused by partial execution of old plans. This problem is somewhat alleviated by the fact that the planner is working at a relatively high level of abstraction and can, therefore, generate plans much more quickly than an architecture based on the sense-model-plan-act paradigm.

2.3.3 RALPH-MEA

The Rational Agent with Limited Performance Hardware – Multiple Execution Architectures (RALPH-MEA) is a control architecture designed to make intelligent decisions in real time (Ogasawara 1993). RALPH-MEA controls an autonomous underwater vehicle. RALPH-MEA relies on decision theory, a well-principled framework that explicitly considers world uncertainty and multiple objectives to compute the optimal decision to execute (Poole 1996).

RALPH-MEA uses multiple plan execution architectures (EAs) running in parallel to select actions. An execution architecture is a program module that selects an action based on sensory information. All of the EAs in the system receive the same plan to execute, but each uses a different type of knowledge to determine a best action choice. A meta-level control system monitors the action selection of the EAs and determines an overall best action. Because each EA operates on different knowledge types, they have varying cost of computation and quality of computed results. For example, one EA that uses "if <`condition`> then <`action`>" rules will probably have relatively low computation costs, but the quality of its action recommendations also may be low. In contrast, a EA based on a decision-theoretic planning system may have high relative computation costs, but correspondingly high quality results. Depending on the domain, a combination of the results of the EAs is a better strategy.

EAs come in four types. The primary one is based on decision theory and the other three provide improved execution speed by compiling knowledge into an easily accessed form. The four types of EA are:

Decision-Theoretic: The expected utility of each action is computed and the action with the maximum expected utility is selected. The Decision Theoretic execution architecture can learn by operating in its environment and recording the results of its actions.

Goal-Based: The utility of state knowledge from a Decision-Theoretic EA can be compiled into this representation which requires less computation to select an appropriate action. However, changes in the environment may make a Goal-Based EA obsolete.

Action-Utility: The expected utility of each action is stored and used to select appropriate actions. This EA works just like the Decision-Theoretic EA, but lacks the capacity to learn and does not look beyond the current time step.

Condition-Action: This execution architecture is similar to standard rule based production systems. Condition-Action EAs can be compiled from Decision-Theoretic EAs.

RALPH-MEA has been tested on a simulated underwater vehicle for the task of mapping undersea mine fields while avoiding detection. The experiments show that it performs well in most situations and can react in a timely manner to unexpected events. Much of the success is attributed to the ability of the system to compile knowledge into forms which provide quick response time. RALPH-MEA is based on the sense-model-plan-act paradigm coupled with compiled plans and is intended to provide quick response with limited computational resources. It demonstrates one way in which learning can take place in a robot architecture: compilation of knowledge into easily accessible form. The system does not learn anything new about its environment, but it does compile the knowledge needed to react quickly to a specific situation into a readily usable form so that it can use its computational resources more efficiently.

2.3.4 GLAIR

Hexmoor et al. (1993) propose a two level approach which they call GLAIR (Grounded Layered Architecture with Integrated Reasoning). This architecture has a low level layer for modeling tasks that require short times between sensing and acting and a high-level layer for modeling tasks that can tolerate delays between sensing and acting.

High-level reasoning is performed symbolically in one module and low level activities are performed at two levels; Perceptuo-Motor (PM) and Sensori-Actuator (SA). This approach allows the system to be easily parallelized. Each of the two levels may be implemented on different hardware with communication between them.

At the perceptuo-motor level, the behaviors that result in physical actions are generated by an automaton. The PM-automatons (PMA) are a family of finite state machines represented by <`Rewards`, `Goal Transitions`, `Goals`, `Action Transitions`, `Actions`, `Sensations`>. The PMAs are not required to use all elements of the representation.

GLAIR incorporates a mechanism that allows knowledge to migrate from the higher to the lower levels. A strategy that is initially formulated from explicit rule-based reasoning can be compiled into an implicit form of knowledge at the PMA. The next time circumstances make this action applicable, it can be selected for execution without having to resort to high-level processes. This mechanism allows behaviors to be remembered and re-used at a future time. However, behaviors can only be generated using rule based reasoning which means that the system designer must provide suitable rules and knowledge. Therefore, the system designer is required to include a large set of rules.

2.3.5 DSSA

Domain Specific Software Architecture (DSSA) (Hayes-Roth, Pfleger, Lalanda, Morignot, and Balabanovic 1995) is an architecture for adaptive intelligent systems. This architecture has two levels: cognitive and physical. The physical level interfaces with the environment by performing actions and receiving sensory inputs. The cognitive level interfaces with the physical level by sending plans and receiving perceptions and feedback. The communications are performed using message passing with both levels operating concurrently and sending messages when necessary.

Internally, the cognitive and physical level modules are architecturally identical. The internal components are a set of behaviors, a world model, and a meta controller. Each

behavior has a set of triggering conditions. At any point in time, several behaviors can be active. Each active behavior produces an action. A meta-controller in each level selects which action is actually performed.

This architecture is intended to be modular and to support the creation of a robot with any set of capabilities. Within each level, the behaviors can be selected from a library for inclusion in a specific robot. Moreover, the information base can be selected from a library to give the specific robot an area of expertise. This modularity should allow a new robot to be created from library components very quickly. The architecture has been tested by generating several robots and demonstrating that they meet their design criteria. However, the re-usability of library components is not very strongly supported. For each robot, additional library components were created, with very little re-use. DSSA demonstrates a multi-level architecture and introduces the notion of behavior libraries which can be included in a robot.

2.4 New Forms of Planning

Mobile robots must be able to make high level plans for achieving goals, but must also be capable of reacting in a timely fashion to the environment. While many robotics researchers were experimenting with ways to incorporate reactivity into existing approaches to symbolic planning, others began to redefine the notion of planning.

2.4.1 Planning with Continuous Execution

All of the planning approaches discussed so far are based on traditional symbolic or *plan space* planning. This form of planning is a search through the space of possible plans. Operators transform one plan into another by manipulating symbols. Plan space planning includes evolutionary methods as well as partial order planning and other more traditional symbolic planning methods. Plan space planning is difficult to apply efficiently to stochastic problems.

It is usually not possible for the robot to have complete knowledge about its environment, so planning everything in advance is not possible. With plan space planning, it is necessary to create partial plans at a high level of abstraction and take care of the details only when necessary. Saffiotti (1993) suggests that there are two common approaches: trying to extend a classical planner to handle reactive actions and trying to extend a reactive system to perform planning. He believes that both of these approaches are inherently flawed and suggests that an integrated solution should attack both sides at the outset. He redefines the notion of a plan. Instead of being a partially ordered set of operators, the plans he discusses are sets of situation→action rules; and instead of specifying a sequence of discrete steps to execute, the plan introduces biases on a continuous reactive execution. This is a form of *state space* planning. State space planning is viewed primarily as a search through a state space to find the appropriate action to take in every state.

2.4.2 Planning as a Markov Decision Process

A common state space plan execution model for robot control is to allow the robot to choose one action to perform at each time step. Choosing the best action requires thinking about more than just the immediate effects of the actions. The immediate effects are often easy to predict, but the long term effects are not always as transparent. Sometimes actions with

poor immediate effects can have better long term ramifications. Ideally, the robot should choose the action that makes the right tradeoffs between the immediate rewards and the future gains. When the robot is making decisions about what actions to take, it is following a *policy*. A policy is a mapping that tells what action to take in each possible world state. Casting the planning problem as a Markov Decision Process (MDP) provides a way to formalize the decision making process. When a reward is associated with reaching some goal state or states, planning is the search for a policy that will maximize rewards.

The five components of an MDP model are

1. a finite set of states $\mathcal{S}$,
2. a finite set of actions $\mathcal{A}$,
3. a set of actions $A(s) \subseteq \mathcal{A}$ for each state $s \in \mathcal{S}$ that can be executed in that state,
4. a set of transition probabilities $\Pr(s'|s,a) \forall s \in \mathcal{S}, s' \in \mathcal{S}, a \in A(s)$, and
5. a set of immediate rewards $r^a(s) \forall a \in A(s), s \in \mathcal{S}$ that are available after taking any legal action from any state.

The transition probabilities just specify the probability that the state is s' given that the previous state was s and action a was taken. Planning in an MDP is considered to be a form of model based reinforcement learning (Sutton and Barto 1997b). The object of this type of planning is to construct a policy which, when executed, will select the action in each state that maximizes the rewards received.

2.4.2.1 Completely Observable Markov Decision Processes

The standard MDP model is often referred to as a Completely Observable Markov Decision Process (COMDP), in order to distinguish it from other variations. In a COMDP model, it is assumed that the current state can be fully determined at each step, e.g., a robot can determine exactly what state it is in from information immediately available from sensors. The policy in a COMDP is a mapping from states to actions. When executing a plan, the robot first determines what state it is in, and then consults the policy to find out what action to take. One way to find an optimal policy for an MDP is to find the optimal value function, which can be determined by a simple iterative algorithm called *value iteration* (Sutton and Barto 1997b). The value of each state s is stored in a set $V(s)$ and each value is initialized to zero. Then value iteration is run for some number of iterations or until the change made to $V(s)$ on an iteration falls beneath some threshold. Value iteration updates the estimated value of each state s at each iteration k using the equation

$$V_{k+1}(s) = \max_a \sum_{s'} \Pr(s'|s,a) \left[r^a(s) + \gamma V_k(s') \right], \tag{2.1}$$

where $0 < \gamma \leq 1$ adjusts the learning rate.

COMDPs have previously been exploited as a paradigm for planning in robotic systems (Koenig and Simmons 1994; Koenig, Goodwin, and Simmons 1996). COMDP planning is interesting because it allows for uncertainty in the formulation of a plan (i.e., it allows for the effects of actions to be nondeterministic) and provides a mechanism for efficient re-planning by modifying the existing policy, which is often more efficient than creating a completely new policy.

2.4.2.2 Planning with COMDPs

Mataric (1992b) describes an architecture that integrates a map representation into a reactive, subsumption based mobile robot. The architecture is not simply a combination of traditional planning and reaction, it attempts to integrate reaction and planning into one system. The map representation may be useful because it allows path planning to be fast. The map is created and updated on-line. Making a local greedy choice at each map node results in the global shortest path within the graph in $O(n)$ time, where n is the number of nodes in the map. The map can be considered to be a state diagram where each state corresponds to a specific input pattern. Only one node in the state diagram is active at any time, indicating the approximate position of the robot. This differs from Saffiotti's approach because different actions are invoked to move from one state to another instead of introducing a bias on a continuous reactive execution. Mataric used deterministic state transitions based on the current map position and the heading of the robot.

Dean et al. (1993) provide a method based on Markov decision processes for efficient planning in stochastic domains. Goals are encoded as reward functions expressing the desirability of each world state. The planner must find a policy that maximizes future rewards. The method assumes that the environment can be modeled as a stochastic automaton. Such an automaton can be represented as a set of states, a set of actions, and a matrix of transition probabilities. Achieving a goal consists of performing a sequence of actions which result in a state satisfying the goal conditions. They built a planning system that attempts to minimize the expected number of actions needed to reach a given goal by adding states to an envelope as needed. The envelope contains a subset of the states in the full MDP model and defines a simplified model that can be used to generate a policy more quickly than is possible with the full model. There are two basic subroutines in their algorithm.

Envelope extension adds states to the envelope, making it approximate the complete MDP more closely.

Policy generation computes an optimal policy for the envelope. A complete policy for the complete MDP is constructed by augmenting the policy for the envelope with a set of default actions or reflexes to be executed for states outside the envelope.

Failures of low level actions can trigger the system to extend its envelope, providing a better representation of the complete MDP. This results in a more accurate assessment of the current state, which helps the system to select the correct action, so that failures are less likely to occur in the future.

Dyna (Sutton 1990; Sutton and Barto 1997b) uses model based, or *indirect* reinforcement learning to generate a policy. Dyna also incorporates model-free, or *direct*, reinforcement learning to improve the policy directly, and uses experience to improve the model. This approach has most of the features required for planning in the POMDP/RL robot architecture, but does not account for partial observability of the current state.

2.4.2.3 Partially Observable Markov Decision Processes

COMDP methods do not work well in robot planning tasks because robots can rarely determine their state from immediate sensor observations. Exact state determination is computationally expensive and generally requires expensive hardware or the presence of special, easily sensed, markers at key locations in the environment. This difficulty in determining location leads to a variation of MDP known as *Partially Observable* Markov Decision

Processes (POMDP). Instead of directly observing the current state, the robot makes sensor observations which provide evidence about what state it is in. The observations can be probabilistic; thus, an observation model must list the probability of each observation z for each state in the model. More formally, the POMDP model is an MDP with the addition of a set of observation probabilities $P(z|s,a) \forall s \in \mathcal{S}, a \in \mathcal{A}, z \in \mathcal{Z}$ which specifies the probability of making observation z given that action a is taken in state s.

The policy is a mapping that specifies what action to take in each state. For COMDPs, the representation for the policy is straightforward. Each state is mapped to exactly one action. For a robot to follow its policy, it determines what state it is in, looks up that state in the policy, and performs the specified action. In the case of a POMDP, however, the robot can never tell exactly what state it is in. The current state is represented as a probability distribution b over all possible states. This probability distribution is usually referred to as the *belief state*. The value function for a POMDP is represented as a set of vectors $\mathcal{V}$ in the belief space. Each vector $v \in \mathcal{V}$ is associated with an action. At each time step, an action is chosen by calculating the vector dot product between the current belief state b and every vector in the policy. Whichever vector has the largest dot product wins, and the action associated with that vector is chosen. Chapter 5 provides more information on planning with POMDPs.

2.4.2.4 Planning with POMDPs

Some important recent advances in robotics, machine learning, and control systems are attributable to POMDP research (Koenig, Goodwin, and Simmons 1996; Zhang and Lee 1998). A POMDP framework was used in a tour guide robot at the National Museum of American History (Roy et al. 1998). The robot uses trajectory information and coastal navigation to increase positional accuracy. Coastal navigation is inspired by traditional navigation in ships, which often use the coastline of continents for navigation in the absence of better tools such as GPS. The robot uses a map of the information content of states in its environment for planning. The planner tries to find paths that maximize information content, thereby increasing positional accuracy. Such plans tend to keep the robot close to fixed objects and away from large open areas.

A major problem with POMDPs is the expense of solving for a policy due to the continuous nature of the belief space. Belief discretization provides a principled and practical approach for solving non-trivial POMDPs. *Belief discretization*, as the name implies, involves partitioning the continuous belief space into a finite set of discrete intervals (Geffner and Bonet 1998). With a discrete belief space, value iteration is relatively easy, since the number of states is finite. Using belief space discretization, dynamic programming can be applied to find approximate policies for relatively large POMDP problems (Geffner and Bonet 1998). Resulting solutions are not necessarily optimal because there is no way to determine if the algorithm has converged, and belief discretization introduces errors.

Cassandra, Kaelbling, and Kurien (1996) used value iteration on the underlying COMDP model of the environment in robot navigation task to find a COMDP policy and then used the most likely state to select the action at each time step. To execute the policy, the robot maintains its POMDP belief state b, which is used to select an action from the COMDP policy. This approach allowed their robot to navigate in an environment that was too large for exact POMDP methods while still providing some of the benefits of the POMDP model. They experimented with three methods to choose the action:

Most Likely State: The state in b with the highest probability is taken to be the actual

state.

Voting: Each possible state s_i votes by selecting an action and the vote of each possible state s_i is weighted by the belief b_i that the robot is actually in that state. The action values are summed over all possible states, and the action with the highest value is chosen.

Q_{MDP}: This is a more refined version of the voting method. The value of each possible action at each possible state s_i is weighted by the belief b_i that the robot is actually in that state. The action values are summed over all possible states, and the action with the highest value is chosen.

They implemented these algorithms and tested the ability of the robot to reach a goal from a known starting location. They found the "Most Likely State" algorithm yielded superior performance.

One highly visible success in POMDP planning for robotics is the Xavier robot at Carnegie Mellon University (Simmons and Koenig 1995; Koenig, Goodwin, and Simmons 1996; Simmons, Goodwin, Haigh, Koenig, and O'Sullivan 1997). Xavier uses POMDP planning to navigate in an office environment. Tasks are specified by telling the robot where to go using verbal commands or via a World Wide Web interface. Xavier does not try to create a POMDP policy directly. Instead, it finds a policy for a COMDP but maintains a POMDP belief state to determine its current location. The belief state is used to select an action from the COMDP policy using a voting method similar to that used by Cassandra, Kaelbling, and Kurien (1996). Xavier is very reliable. Over a period of 81 hours of service, it successfully completed 93% of the navigation tasks that were assigned to it.

An approximate approach could be used in the POMDP/RL architecture, but for this work, I did not want to introduce another unknown quantity. The goal of this dissertation is to combine POMDP and RL in a system that can learn new actions. The decision to use exact solution methods results in simpler analysis of the results. With exact solutions, the quality of the POMDP policy is not in question, which allows for better interpretation of the results. Although optimal policies are probably not necessary for most robot control tasks (Cassandra et al. 1996; Simmons et al. 1997), they are used in this dissertation to avoid any bias that could be introduced by using approximate solutions.

2.5 Relationship of The POMDP/RL Approach with Past Work

With the exception of CHILD, none of the robot architectures that we have reviewed provide a mechanism for learning directly from the environment at the low level. The best that they can manage is a form of compiled plan, where information given to the high level planner migrates down to the reactive action level. CHILD does provide low level learning and adaptation, but does not perform long term planning and cannot pursue dynamically assigned goals. This is not adequate for most domains. A truly useful goal-directed robot must be able to perform reactive high level planning and low level learning and adaptation. Adaptive, low level, reactive control must be built in from the ground up, not added as an afterthought (Smithers and Malcolm 1987; Smithers and Malcolm 1988).

No matter how well-designed the robot, eventually effectors and sensors will fail. Ideally, the robot should adapt to these failures as well as to changes in the environment, treating

them as just one more environmental problem to be overcome. The goal of the POMDP/RL architecture is to provide the robust adaptation and learning component that will enable the robot to carry out its mission even when things go wrong.

Section 1.1 gave a description of the simulated Khepera environment. Section 1.2 gave a high level description of the POMDP/RL architecture. This section discusses some of the advantages of the POMDP/RL robot architecture when compared to other architectures. A more complete description of the POMDP/RL architecture is presented in Chapter 7.

The POMDP/RL robot control system is a two level architecture. Two benefits of a multi-level approach are that it modularizes the design and allows processing to be performed at various levels of abstraction. Each level can be implemented in a way that fits the level of abstraction at which it processes information. However, a large number of levels can become cumbersome to manage. A two level architecture is simple enough to manage while sufficiently complex to allow experimentation with the interaction between layers.

The architecture is hierarchical. At the lowest level, actions are reactive and use data from the sensors with very little abstraction. The planner level decides what action to activate to reach some goal, and also incorporates some reactivity, but with a longer response time than the lower level. When given a goal, it generates a policy that tells what action to use for every possible contingency.

The architecture is a hybrid system and benefits from the strengths of each of its components. It also gains from the synergy between its components, and provides the capability to define and learn new actions as needed and to adapt to failures in sensors and effectors. The POMDP/RL architecture fits the description for a two layer architecture, but incorporates learning at the reactive layer and adaptation at the planning layer.

The POMDP/RL approach uses some ideas from subsumption, but avoids the complexity of having multiple actions active at the same time. Instead, the POMDP/RL approach concentrates on providing complex, adaptive actions which can be switched on or off by the planner. The high level POMDP planner is given a goal and then generates a *policy* that specifies what action to execute for every belief state in order to reach the goal state. Serial actions were chosen in order to avoid the complexity inherent in coordinating the concurrent execution of several actions. Allowing only one action to be active at any time results in a system that is easier to understand and debug, and enables the planner to assign a reward to each action for performing its task. Credit assignment is critical for providing adaptation and learning at the low level in the POMDP/RL architecture.

The POMDP/RL approach divides the navigation into different levels of abstraction and has some similarity to a system based on RAPs. RAPs provide most of the functionality required for a general robot control system, but does not allow for learning. This means that every possible situation must be anticipated by the designer. Such a system can fail when faced with a situation for which there is no suitable action. As with RAPS, PRS does not incorporate learning, so the designer is responsible for anticipating every possible contingency.

In the POMDP/RL architecture, the planner monitors its current state and detects unexpected deviations, but it uses a very different mechanism than traditional symbolic planners. The planner in the POMDP/RL system generates a policy that specifies what low level action to invoke for every possible contingency. One drawback to this is that it can take hours or even days on current computer hardware to construct the initial policy. Once constructed, a policy can be efficiently stored for later use, providing the benefits of a compiled plan approach. Existing policies can be modified to change the goal to a nearby state, reducing the computational requirements. In addition, there is some evidence

that exact policies may not be necessary for reliable navigation (Cassandra, Kaelbling, and Kurien 1996; Simmons and Koenig 1995).

Like Cypress, the POMDP/RL robot architecture is also based on existing AI technologies, and their combination results in capabilities that neither possess alone. The POMDP/RL approach uses reactive actions at the low level and reactive plan execution at the upper level. Using a policy that specifies what action to take for every possible situation prevents the type of re-planning cycle that is a potential hazard for systems that do asynchronous run-time re-planning.

The POMDP/RL approach detects the potential for new low level actions based on observations of state transitions and then creates low level modules to effect those actions. The POMDP/RL approach does not need to be told what is possible, in the way a decision-theoretic planner does. Instead, the POMDP/RL system learns what is possible by observing. The natural stochasticity of the environment helps the POMDP/RL architecture to discover what actions are possible.

GLAIR requires the system designer to include a large set of rules. The POMDP/RL approach does not have this requirement. The designer must only provide a map of the environment. The system learns for itself what low level actions are possible.

The POMDP/RL approach, like DSSA, includes the notion of an action library, but library modules in the POMDP/RL architecture are learned. Some actions may be initially hand coded, but the robot is capable of adding new actions to the library, and can replace hand coded actions with adaptive, learned actions. The POMDP/RL approach provides the modularity of DSSA plus learning and adaptation at the low level.

Chapter 3

Review of Low Level Behaviors and Reinforcement Learning

In robotic control tasks, a low level control module is generally required to meet time constraints and provide robust control in a noisy environment. Several techniques have been used to implement low level behaviors, including behavioral cloning, reinforcement learning, and subsumption and other finite state automaton approaches. This chapter explains some of these techniques and discusses their relative strengths and weaknesses. Section 3.4 presents the approach to low level behaviors that is used for the research in this dissertation.

Most multi-level robot architectures can be described in terms of how behaviors are implemented. There are three basic approaches:

Monolithic Behavior: All behaviors are implemented by one module. This approach can lead to large behavior modules which are difficult to understand and debug, but it has the advantage of centralized control and coordination of behaviors. Also, all behaviors use the same information base. Thus, information is not duplicated between separate behaviors. The monolithic behavior approach provides tight coupling between behaviors, but can lead to large and unwieldy program modules.

Parallel Behaviors: Behaviors are implemented in separate modules, and more than one module may be active at any time. This approach requires some sort of coordinating mechanism to select or integrate the actions proposed by separate modules. The parallel behavior approach helps to modularize the implementation of behaviors, but requires a coordinating mechanism which, like monolithic behaviors, can be complex and difficult to debug.

Serial Behaviors: Behaviors are implemented in separate modules, and only one module is active at any time. This approach requires a coordinating mechanism for determining which behavior is active, but sidesteps the issue of coordinating several active behaviors. This approach is attractive because it is much easier to coordinate the activities of multiple behaviors in series than in parallel, but it may not be as efficient as approaches that allow multiple behaviors to execute simultaneously, and may have more complex behaviors than systems that allow parallel execution of behaviors.

The POMDP/RL approach uses serial behaviors. The following sections describe some alternate methods for creating behaviors, each of which falls into one or more of these broad

categories.

3.1 Behavioral Cloning

Sammut (1996) describes a system for automatically building rule based systems for low level behaviors. Tasks such as piloting an aircraft require sub-cognitive reactions on the part of the pilot, while the pilot may not be able to describe in detail how decisions are made. This precludes the traditional approach to building an expert system for the control system. Sammut uses an approach referred to as *behavioral cloning* to create rules for the control system. Behavioral cloning is a machine learning method for extracting situation-action rules from logs of actions taken by a human subject. This approach to low level behaviors was used to generate rules for flying an airplane through a set of maneuvers.

They created logs of human pilots using a flight simulator and then used behavioral cloning to generate a set of situation-action rules. The flight plan consisted of several distinct maneuvers:

1. Take off and fly to an altitude of 2000 feet.
2. Level out and fly to a distance of 32,000 feet from the starting point.
3. Turn right to a compass heading of approximately 330°.
4. At a North/South distance of 42,000 feet, turn left to head back towards the runway.
5. Line up on the runway.
6. Descend to the runway, keeping in line.
7. Land on the runway.

The rules for executing each of these maneuvers were learned separately, resulting in seven low level behavior modules, one for each stage of the flight. A higher level controller determined which of the behavior modules is active at any given instant. This is an example of how long term goals can be implemented in a high level module which selects serially from a set of low level reactive controllers. This system was able to fly the airplane through the required course very smoothly. However, the learned behaviors were not very generalizable. For instance, one behavior turned right to a compass heading of 330°, but it could not be used to turn to any other heading. With a much larger set of behaviors, it may be possible to build a more general system.

Behavioral cloning results in behaviors that have no representation of goals. The rules are pure situation-action rules. This can lead to a lack of robustness. When a situation occurs that is outside of the data that was used to train the system, the clone can fail entirely. This undesirable trait can be reduced somewhat by training in the presence of noise, but it is still unlikely that all possible situation action pairs will be present in the training data. The resulting system may not be able to select an appropriate action in novel situations. Also, there is no mechanism for learning from mistakes, so performance will not improve beyond what is initially learned. However, this approach could be used to develop behaviors that are difficult or impossible to learn by trial and error, particularly if errors can cause damage to the robot or its environment.

3.2 Neural Control for Low Level Behaviors

In general, neural systems tend to be robust to noise and perturbations in the environment, unlike traditional symbolic approaches. For this reason, neural learning has been used for low level robot control (Torras 1994). Neural network based error minimization systems have been used for solving inverse kinematics for robot arms and visual robot positioning. Neural network based correlational procedures have been used for hand-eye coordination and inverse dynamics of robot arms. Reinforcement learning techniques using neural networks have been used for mobile robot navigation and fine manipulation for robot arms. The following paragraphs provide some indication of the wide variety of robot applications that are possible with neural systems.

3.2.1 Manipulation

Poo, Ang Jr., Teo, and Li (1992) investigated the use of a neuro-model based controller architecture for a one link robotic arm. They show improved performance over a standard model based control algorithm in the presence of noise and disturbance. Model based robot control involves incorporating the robot dynamic model into the robot controller to transform the highly nonlinear robot dynamics into equivalent linear systems. Linear control theory can then be applied to create closed loop control. They use a backpropagation neural network to learn the robot dynamics. The neural network can then be used as a replacement module for the modeling component in a standard model based controller approach. The neural network approach allows the model to adapt to changes in the robot dynamics, resulting in a more robust system.

Krishnaswamy, Ang Jr., and Andeen (1991) present a structured approach to creating neural network robot motion controllers. This work is similar to the work by Poo, Ang Jr., Teo, and Li (1992) in that both systems use a neural network to model the inverse kinematics. However, this work goes further and puts the actual control algorithm in a neural network. The goal for this work was to develop a controller for a robotic arm with three degrees of freedom. The controller was required to efficiently move the end effector to any specified position. They divided the task into three modules: inverse kinematics, acceleration control, and linearization. Each module was implemented using neural networks with backpropagation training. The inverse kinematics network receives the desired end effector position in Cartesian coordinates and produces the desired joint coordinates. The acceleration controller receives the desired joint coordinates along with the actual joint coordinates reported by the arm joint sensors and produces the desired joint accelerations. The desired accelerations are passed to the linearizer, which compensates for non-linearities in the manipulator and sends the adjusted signals to the actuators. All of the networks were trained on actual data acquired from the manipulator and then the networks were connected together. The system performed well, providing smooth and robust control for the robot. Both of the previous approaches provide reliable control, but at the cost of a great deal of time to learn and careful tuning to the problem that is being solved.

The GeSAM system, developed by Liu, Iberall, and Beckey (1989), merges neural networks with symbolic systems. GeSAM is a robot hand control system based on human prehensile function. It uses an adaptive neural network to learn the relationship between object primitives and a set of grasp modes for picking up cylindrical objects. The four grasp modes are modeled on the ways in which humans grasp objects. The system learns to select the proper grasp mode for a specific cylinder. The inputs to the system are the diameter

and height of the cylinder to be grasped. A high level symbolic system controls overall functionality while the neural system takes care of the low level details. The high level system simply issues the command to grasp the object, without considering its dimensions. The neural network considers the dimensions of the object and selects the appropriate grasp mode. Updating and improving the neural network performance can be done online.

Bullock, Grossberg, and Guenther (1992) investigate using a neural network model for the classical motor equivalence problem, where many different joint configurations of a redundant manipulator can all be used to realize a desired trajectory. They show that the system can compensate for variable tool lengths, clamping of joints, distortions of visual input, and unexpected limb perturbations. They relate their approach to neurophysiological data. Both of the previous approaches have high computational requirements.

3.2.2 Locomotion

Dubrawski and Crowley (1994) present a neural approach to reactive navigation. The task of the system is to provide a steering angle signal letting a robot reach a goal while avoiding collisions with obstacles. They use a Fuzzy ART neural self organizing classifier to perform a perceptual space partitioning and a neural associative memory to memorize the system's experience and combine influences of different behaviors. They test the system's learning ability, starting with zero knowledge, and find that it efficiently learns to reach its goals. This approach works well considering its simplicity, but does not deal well with changes in the environment.

Tani and Fukumura (1994) used a neural network to implement a vector field approach to robot navigation. The tasks assigned to the robot were to home to a predetermined goal and to cycle through multiple locations in a predetermined sequence. The objective is to construct a vector field in the task coordinates that converges to the point representing the goal. Then the robot only needs to follow the direction of the vector field from its current location to the goal. The robot learns each task with the help of a trainer who is assumed to know the optimal paths. For each training run, the robot is placed at an arbitrary point and guided by the trainer to the goal position. Guidance to the goal is repeated from various starting points so that the robot learns a general robust homing scheme in the work space.

A major advantage of neural systems is that they can often compensate for drastic changes in the environment by learning continually. These features make neural networks an attractive alternative to symbolic processing for low level control. However, neural networks often require long training periods and large data sets, which may be difficult to acquire. Additional problems may arise with some types of neural networks when they are used in reinforcement learning. These problems are detailed in Chapter 4. Neural networks seem to have advantages for low level control, but thus far, are less useful for task-oriented planning and long term goal attainment. For this reason, they are seldom used for high level control in robot navigation.

3.3 Reinforcement Learning for Low Level Behaviors

When using reinforcement learning, the robot continually receives sensory inputs from the environment and sends control signals to the effectors. Along with the sensory inputs, it receives a reinforcement signal at each time step. The robot must maximize the average

reinforcement which it receives over time by creating an optimal control signal selection policy. It does this by learning the value of each possible control signal in each state. The central idea of reinforcement learning algorithms is Temporal Difference (TD) learning, which is a combination of Monte Carlo methods and dynamic programming (Sutton and Barto 1997b).

Reinforcement learning has been used in several ways to provide low level behaviors. Singh (1991, 1992b) has implemented a two level approach, called CQ-L, where the low level or *elemental* behaviors use reinforcement learning and a task-sensitive gating module learns to compose the appropriate elemental modules over time in order to achieve long term goals. In this approach, all elemental behaviors are Markov Decision Tasks (MDTs) that share the same state set $\mathcal{S}$, action set $\mathcal{A}$, and the same environment dynamics. Each elemental task has its own reward function based on the desired final state. A *composite* task is rewarded when one of the subtasks reaches its terminal state. This architecture was tested in a simulated robot navigation environment and was able to solve some composite tasks, such as moving from one point to another and then proceeding to a third point. This approach shows that reinforcement learning of low level behaviors can be made to work when the exact behaviors are specified, but provides no mechanism for the generation of new behaviors by the robot itself. Also, it does not address the issue of higher level planning and is not taskable.

Kontoravdis, Likas, and Stafylopatis (1992) examined the use of reinforcement learning in control applications. Their goal was to guide a simple simulated robot through a series of left and right turns using only low level behaviors. They showed that the simulated robot could learn to avoid obstacles and move around. They gave the robot no goal other than continuous movement without hitting anything. Their work indicates that reinforcement learning may be used to learn movement in the environment while simultaneously avoiding obstacles. Because reinforcement learning maximizes long term reward, and running into obstacles is unlikely to be a good way to gain reward the low level actions automatically learn to avoid obstacles.

Dorigo and Colombetti (1994) trained a robot to perform a predefined target behavior using reinforcement learning. They performed experiments with both simulated and real robots. Their goal was to create a robot capable of surviving in a real environment. They trained the robot to display five different behaviors and then defined four ways to combine these behaviors. Their approach differed slightly from most reinforcement learning strategies. Instead of forcing the robot to learn with no initial knowledge, they gave the robot some basic responses and then used reinforcement learning to improve the robot's knowledge. The basic behaviors are approaching an object, chasing an object, entering a well-defined physical state which is a function of a feature of the environment, avoiding contact with an object, and fleeing from an object. They studied four ways to combine the low level behaviors:

Independent sum Two or more independent responses are produced at the same time, with the effectors acting on a weighted sum of the commands sent to them.

Combination Two or more responses are combined into a resulting behavior, using domain knowledge to guide the combination.

Sequence A behavioral pattern is built as a sequence of basic responses, thereby removing the possibility of simultaneous conflicting actions.

Suppression A response suppresses a competing one based on a prioritization of basic responses, so that higher priority responses gain control.

The learning system architecture is designed in such a way as to favor learning by exploiting the environment's characteristics in order to make learning possible. An external trainer then makes suggestions that the robot uses to create an effective control strategy. The problem with this approach is in finding the right balance between design, learning, and training. They rely heavily on experimentation with a simulated robot and its environment to determine what the balance should be.

Mahadevan and Connell (1992) use reinforcement learning for automatic programming of behavior based robots. They find that the reinforcement learning techniques are capable of learning individual behaviors that sometimes outperform hand coded solutions. They also conclude that using a behavior based architecture speeds up reinforcement learning by converting the problem of learning a complex task into that of learning a simpler set of special purpose reactive subtasks. They have five requirements for a robot learning algorithm.

Noise immunity The technique should be able to deal with noise. State descriptions may sometimes be wrong, and the use of probabilistic smoothing techniques is often required.

Fast convergence The technique should be able to converge in a reasonable number of trials to be feasible on a real robot.

Incrementality The learning algorithm should allow the robot to improve its performance while it is learning. Since the examples are generated by the robot itself, this allows the robot to explore its environment quicker and generate better examples.

Tractability The learning algorithm should be computationally tractable. That is, every iteration of the algorithm should be done in real time.

Groundedness The technique should only depend on information that can actually be extracted from the sensors on a robot.

They train the reinforcement learning algorithm to perform the following behaviors.

- Finding a box
- Pushing a box
- Getting unwedged

They find that reinforcement learning can adequately perform the required behaviors. Their research also shows that a simulator can be used for initial training, but it is not apparent how well the approach will scale with the addition of new behaviors.

3.4 Review of Reinforcement Learning

Reinforcement learning is an approach to learning through interaction with the environment (Sutton and Barto 1997b). It is modeled on the type of learning that occurs in nature. When we do something that brings a positive reward (pleasure), we tend to do the

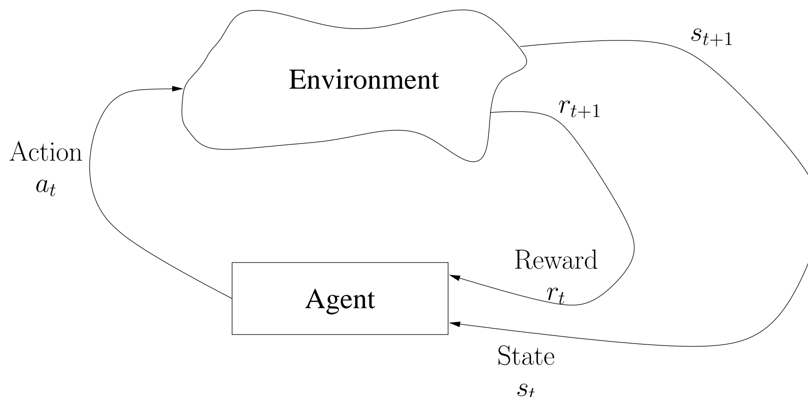


Figure 3.1: An agent using reinforcement learning interacts with its environment by receiving states and rewards from the environment, and generating actions that change the environment.

same thing again. Likewise, actions that bring negative reward are avoided. The ability to link cause and effect (sometimes incorrectly) is a basic ability shared by even the simplest animals. The central idea of reinforcement learning algorithms is Temporal Difference (TD) learning (Sutton 1988). The following section provides a brief introduction to reinforcement learning methods.

An agent using reinforcement learning receives sensory inputs from the environment and uses them to select what action to take. As shown in Figure 3.1, the agent receives a reinforcement signal from the environment along with the current state at each time step. At each time step, the reinforcement can be positive, negative, or zero. The agent must maximize the average reinforcement that it receives over time by creating an optimal action selection *policy*. A policy is a function that maps states to actions. The reinforcement learning agent finds a policy by learning an estimate of the value of taking each action from each state. Another way to say this is that the agent learns to approximate the *value function*. A value function V is a function that maps inputs representing the current state s to a real number v that is the expected long-term reward that can be earned when starting from that state.

A policy π is defined to be better than or equal to a policy π' if and only if $V^{\pi}(s) > V^{\pi'}(s)\ \forall s \in \mathcal{S}$, where $\mathcal{S}$ is the set of all possible states that the agent can be in. This simply says that if the value of a policy π is better than the value of another policy π' *at every state*, then π is said to be better than π'. The *optimal* policy $V^*(s)$ is defined as

$$V^*(s) = \max_{\pi} V^{\pi}(s)\ \forall s \in \mathcal{S}, \tag{3.1}$$

which is simply the policy that maximizes value at every possible state. Since we are interested in choosing actions given the current state, it is often more convenient to use the *optimal action-value function*, which we define as

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a)\ \forall s \in \mathcal{S}, a \in \mathcal{A}. \tag{3.2}$$

Given the state s and an action a, this function gives the expected return for taking action a in state s and thereafter following an optimal policy. Given this function, it is easy enough to choose the best action at each state by evaluating $Q^*(s,a)\ \forall a \in \mathcal{A}$ and choosing the action with the maximum value.

Q^* can be expressed in terms of V^* as follows:

$$Q^*(s,a) = E\left[r_{t+1} + \gamma V^*\left(s_{t+1}\right)|s_t = s, a_t = a\right] \tag{3.3}$$

where $E[x]$ is the expected value of x, γ is a *discount factor* that weights the importance of future rewards and r_{t+1} is the reward at the next time step. Equation 3.3 shows that the optimal action-value function is simply the expected value of the immediate reward for taking action a in state s and then following the optimal policy from the resulting state.

The Bellman optimality equation expresses the fact that the value of a state under an optimal policy is equal to the expected return for taking the best action from that state. From Equation 3.3, the Bellman optimality equation for V^* is

$$V^*(s) = \max_a E\left[r_{t+1} + \gamma V^*\left(s_{t+1}\right)|s_t = s, a_t = a\right], \tag{3.4}$$

and the Bellman optimality equation for Q^* is

$$Q^*(s,a) = E\left[r_{t+1} + \gamma \max_{a'} Q^*\left(s_{t+1}, a'\right)|s_t = s, a_t = a\right]. \tag{3.5}$$

These equations, combined with mechanisms for exploring the state space, lead directly to the Temporal Differencing (TD) methods that are the core of reinforcement learning.

Given a state s, the *greedy* action for that state is the one with the maximal expected action value from Equation 3.5. The most common method for exploring the state space is to choose the greedy action most of the time, but choose a random action with some probability ϵ. The probability of choosing the greedy action in state s is $1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|}$ and the probability of choosing some other action is $\frac{\epsilon}{|\mathcal{A}(s)|}$. This method is referred to as an ϵ-greedy policy. The following sections explain two common TD methods that use ϵ-greedy policies.

3.4.1 Sarsa

There are two main classes of TD learning algorithm: on-policy and off-policy. On-policy methods evaluate or improve the policy that is currently being used to make decisions, while off-policy methods evaluate or improve a different policy than the one that is currently being followed. In off-policy learning, the policy used to select actions is called the *behavior* policy, while the policy that is being improved is called the *estimation* policy. In on-policy learning, there is only one policy that is used to select actions while it is being improved. Both methods use some stochastic method, such as ϵ-greedy policies, to select actions. Stochastic action selection results in exploration the state space and provides data needed to improve the approximation to the policy.

The simplest on-policy TD update rule, derived from Equation 3.5, is known as Sarsa. The word Sarsa comes from State, Action, Reward, State, Action, and the algorithm uses every element of the quintuple of events $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ in the update equation (Sutton and Barto 1997b). The Sarsa update equation is

$$\Delta Q_t(s_t, a_t) = \alpha\left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right] \tag{3.6}$$

where $Q(s_t, a_t)$ is the estimated value of state s at time t given action a_t, r_{t+1} is the reinforcement at time $t+1$, α controls the learning rate, and γ controls the importance of future rewards relative to immediate reward. At every time step, the update rule adjusts the

estimated value function Q so that it more closely approximates the actual value function Q^*.

The greedy action selection policy chooses an action at time t which leads to the state s_{t+1} having the highest value at the next time step. In order to choose the state with the highest value, the action selection policy must know the mapping from the current state s_t to the next state s_{t+1} given the action a for all actions $a \in \mathcal{A}(s)$. This is not always possible, especially in the case where the outcome of taking a specific action in a specific state is non-deterministic. The best that can be done in some cases is to weigh the value of each possible next state s' by the probability of transitioning to that state from the current state s, given action a.

An ϵ-greedy action selection policy is used to select the action that will lead to the state with the highest value, with some stochastic selection to encourage the system to explore the state space. In other words, the system may occasionally perform an action that is not best according to the current policy. If the action chosen in state s_t is random or if the task ends at state s_t, then s_t is a terminal state. If s_{t+1} is a terminal state, then $Q(s_{t+1}, a_{t+1})$ is defined as zero. This can have a negative influence on learning (Sutton and Barto 1997b).

3.4.2 Q-learning

The difficulty of avoiding terminal states while taking random actions can be overcome by using off-policy learning. The best known off-policy TD control algorithm is Q-learning (Watkins 1989), which follows an ϵ-greedy policy while learning the corresponding greedy policy. Its simplest form, known as one-step Q-learning, is defined by the update equation

$$\Delta Q_t(s_t, a_t) = \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]. \tag{3.7}$$

The goal of the reinforcement learning algorithm is to learn the value of each state-action pair in the state space. In the case of Q-learning, the function Q that is learned is a direct approximation of the optimal action-value function Q^*. The approximation can be improved at each iteration, independently of the policy currently being followed. In other words, the current policy does not have to be optimal, or even near optimal, in order for the approximation Q to be improved; random actions are no longer viewed as terminal states. The approximation Q has been shown to converge with probability 1 to Q^* for table lookup methods, given that all state-action pairs are continually updated and that certain constraints on the sequence of step-size parameters are observed.

In general, reinforcement learning methods may use table lookup, neural network, or other methods to store an approximation to the value function. Many function approximation techniques are available, and evaluating function approximation techniques is an important area of RL research.

3.4.3 Other RL Methods

Sarsa and Q-learning are the most common algorithms for reinforcement learning, but there are several other approaches. For instance, the actor-critic methods divide the problem into two parts: a value function approximator and a policy representation. The policy representation is called the *actor* because it is used to select an action at each time step. The value function approximator is called the *critic* because it critiques the actions that are selected. Learning is always on-policy, since the critic must learn to evaluate whatever policy is being

followed by the actor. Thus, the critic attempts to learn V^* directly by associating observations with rewards. The TD error is output from the critic and is used to drive training in both the actor and the critic. Stochastic action selection helps the system to explore the state space by occasionally choosing actions that were not chosen by the policy network. This approach effectively decomposes the reinforcement learning task into two subtasks: learning the value function and learning the policy. Several early reinforcement learning systems based on TD methods were actor-critic systems (Barto, Sutton, and Anderson 1983; Witten 1977). The Adaptive Heuristic Critic (AHC) (Sutton 1984; Anderson 1989; Anderson 1986) is probably the most well known actor-critic system. Actor-critic systems are more biologically plausible than Q-learning, but research in actor-critic systems has lagged behind research in Q-learning and other techniques due to the lack of convergence proofs. Recently, convergence proofs for actor critic systems have been presented and this may help to accelerate actor-critic research efforts (Prokhorov and Wunsch II 1997).

Real-Time Dynamic Programming (RTDP) (Barto, Bradtke, and Singh 1993) is a reinforcement learning algorithm that is intended to allow an embedded system to improve its performance with experience. This approach uses full value iteration backups and illuminates aspects of other DP-based reinforcement learning methods, such as Q-learning. RTDP is important for its contribution to the theory behind reinforcement learning approaches, and gives some insight into how Q-learning can be embedded in a robot control system. They showed that RTDP converges to optimal solutions when applied to several types of real-time problem solving tasks, and showed how DP can be adapted to performing DP updates concurrently with problem solving and control. RTDP is a *model-based* method because it requires an explicit world model for the value iteration. Finding a policy using model-based RL methods can be viewed as state space planning with a Markov decision process (Sutton and Barto 1997b). Section 2.4.2 covered this topic. Partially observable Markov decision processes are another form of model-based reinforcement learning (Sutton and Barto 1997a). This method is used for high level planning in my robot architecture.

3.4.4 Improvements to Reinforcement Learning

The previous sections reviewed approaches to reinforcement learning, including actor-critic, Q-learning, Sarsa, and RTDP. Currently, Q-learning may well be the most popular approach, but there are problems for which other approaches yield faster and/or more reliable convergence (Sutton and Barto 1997b). Considerable research has been directed at improving the performance of reinforcement learning architectures.

Lin (1992) examined reinforcement learning frameworks based on AHC and Q-learning. He also evaluated five extensions to both learning frameworks. The extensions are experience replay, learning action models from planning, teaching, n-step prediction, and TD(λ).

Experience Replay: Reinforcement learning with neural network function approximation can be inefficient in that the information obtained by trial and error is utilized to adjust the networks only once and then thrown away. This is wasteful, especially since some experiences may not occur very often and some experiences (such as those where the robot is physically damaged) are costly to obtain.

Experience replay is a method for reusing information. An experience can be stored as a quadruple (x, a, y, r) which indicates that action a in state x results in new state y and reinforcement r. Using experience replay is simply a matter of saving a set S of

past experiences and presenting them to the reinforcement learning system as if they had just occurred. This speeds up the process of credit assignment and can result in the system learning more quickly. New experiences can be added to S as they are encountered. When neural networks are used to approximate the value function, this approach helps to prevent over training by keeping a broad range of training data available for the neural network.

Learning Action Models: Past experience can also be used to create an action model. An action model is a mapping from a state/action pair (s_t, a_t) to the next state and reinforcement (s_{t+1}, r_{t+1}). In other words, it is a model of the environment that is built from experience. The action model serves as a replacement for the real environment during reinforcement learning. In a stochastic environment, each action may have multiple possible outcomes. In this case, the action model must determine the next state and reinforcement probabilistically. An accurate action model can help reinforcement learning by allowing it to experience the consequences of actions without actually performing them. This allows the robot to learn more quickly, if the action model can be run faster than real time. It also allows the robot to avoid mistakes in the real world.

Teaching: Since it starts with no knowledge, reinforcement learning requires that the system be able to reach the goal through random actions. If it is difficult for random actions to reach the goal state, learning can take many trials. Teaching is a way to improve learning performance of reinforcement learning systems. In this approach, the teacher provides the learning system with one or more examples of appropriate actions necessary to reach the goal state. The solution provided by the teacher does not need to be optimal because it is only used to provide a starting point for the reinforcement learning system to begin its search for the optimal policy. The mechanism used is similar to the mechanism for experience replay, except that the actions come from a teacher instead of from the robot. This approach can introduce a strong bias towards what the teacher believes is the optimal policy, even when the teacher does not actually have the optimal policy.

n-Step Prediction: In the one-step RL methods covered previously, the reward from the next time step is used to update the current estimate at the current state. The estimate for the current state can also be updated using rewards from steps farther in the future. For instance, two-step prediction uses rewards from the next two time steps, and n-step prediction uses rewards from the next n time steps, averaged together.

TD(λ) This can be viewed as an extended form of n-step prediction where the relative importance of the return from each time step is weighted by a λ parameter (Sutton 1988; Watkins 1989). The one-step return is given the weight $1 - \lambda$. The two-step return is weighted by $(1 - \lambda)\lambda$. The three-step return is weighted by $(1 - \lambda)\lambda^2$, and so on. In effect, the weight of each subsequent return fades by λ. Theoretically, this method considers returns from infinitely far in the future, but in practice it is necessary to truncate the returns at some point n steps in the future.

Lin experimented with several combinations of AHC and Q-learning using replay, teaching, and modeling. This simulated environment is a 25×25 cell world containing food, obstacles, and enemies. The robot is required to find food while avoiding enemies. The

robot and each enemy can move in any of four directions on each time step. He found that the Q-learning approach was best for this application. Replay, teaching, and modeling were all found to improve performance, but no attempt was made to use all three enhancements in a single robot.

Reinforcement learning has been shown to converge when a table lookup method is used to store the function approximation. However, the size of the table grows very quickly with the number of inputs. Due to memory constraints, it is not feasible to implement a function lookup table for a large system. Various attempts have been made to use less memory intensive general function approximators such as backpropagation neural networks. There have been notable successes in several domains such as checkers (Samuels 1959), Backgammon (Tesauro 1990), and robot navigation (Lin 1992). However, reinforcement learning with general function approximators is notoriously unstable. Boyan and Moore (1995) list several simple situations where popular function approximation algorithms such as polynomial regression, backpropagation, and local weighted regression fail to converge. Gordon (1995), Thrun and Schwartz (1993), and Sabes (1993) give some reasons why these failures occur.

One reason is that, in order to guarantee convergence, any change made to the value function approximation must be local to the current state (Thrun and Schwartz 1993; Singh and Ye 1994; Sabes 1993). One way this can be accomplished is by mapping the continuous state variables into discrete intervals. For instance, if we have a state variable $x \in [-1 \ldots 1]$, we can map it to a discrete space $x_d \in (X_1, X_2, \ldots, X_N)$. Doing this for all state variables yields a set of symbols that specify the current state. The discrete space can be enumerated, resulting in an ordinal value for each possible state. The ordinal value can be used to index a table which stores the value function approximations for each state. This lookup table approach guarantees that only local changes are made to the function approximation. Sparse, coarse coding of the input space with table lookup has also been shown to converge more reliably than using direct backpropagation (Sutton 1996).

Neural reinforcement learning typically uses a backpropagation neural network with a sigmoidal activation function to learn the value function, because that is arguably the most popular and pervasive form of neural network. One problem with using this approach is that backpropagation networks are not limited to making local changes in the function approximation (Boyan and Moore 1995; Baird 1995). A change to the function for one set of inputs can have enormous effects for other sets of inputs. What this translates to in terms of reinforcement learning is that learning the value of a specific state or state action pair may cause the destruction of the learned value of some other state. Thrun and Schwartz (1993) discuss another problem with using backpropagation networks. Due to the nature of backpropagation and reinforcement learning, it is very easy for the system to consistently overestimate the utility of state-action pairs, resulting in improper credit assignment. For these and other reasons, the backpropagation approach with sigmoidal activation functions is not guaranteed to converge. This problem is alleviated somewhat by using experience replay and other techniques, but at the cost of much higher computation.

There have been several recent attempts to improve the convergence of neural reinforcement learning techniques (Tsitsiklis and Van Roy 1997; Jaakkola, Jordan, and Singh 1994; Singh 1993; Anderson 1993). Most of these attempts fall into two broad categories; recoding the input space and modifying the backpropagation algorithm. One promising approach is the use of radial basis functions (RBFs) instead of sigmoidal activation functions (Kretchmar and Anderson 1997). This approach alleviates the problems of non-local update, but tends to have problems representing the value function at the edge of the state space and

introduces some waviness in the value function. Selecting the parameters of the radial basis functions is also an active area of research.

Another approach to improving convergence when using neural networks for function approximation is to change the network equations for the backpropagation algorithm. A modification of backpropagation called *residual gradient algorithms* minimizes the residual Bellman equation for reinforcement learning instead of the standard direct gradient algorithm of traditional backpropagation (Baird 1995). It is guaranteed to converge, but may take a long time to do so. Convergence can be accelerated by using a combination of residual gradient algorithms and direct gradient algorithms.

Mahadevan and Connell (1992) developed a statistical clustering approach to function approximation with reinforcement learning on a robot. The task for their robot was to find boxes and push them against a wall. They specified two behaviors for their robot and gave suitable rewards when it achieved its task. Their robot had a limited number of sensors, and they pre-processed the sensor data to reduce it to a very small number of observations. They examined two approaches: 9 bits of sensory information with a weighted Hamming clustering approach and 18 bits of sensory information with statistical clustering. They found that reinforcement learning was a viable approach to learning individual behaviors, producing similar performance to a hand coded subsumption architecture approach.

When the state can be described by a set of Boolean or discrete-valued variables, it is possible to learn compact decision trees for representing the value function. The G-learning algorithm (Chapman and Kaelbling 1991; Kaelbling, Littman, and Moore 1996) starts by assuming that no partitioning is necessary and tries to learn Q values for the entire environment as a single state. In parallel with this process, it gathers statistics, which are used to determine whether it should modify its state space representation. It uses a T-test to determine if there is some bit b in the state description such that the Q values for states in which $b = 1$ are significantly different from the Q values for states in which $b = 0$. If such a bit is found, then it is used to split the decision tree. The process is repeated in each of the leaves. This method is able to learn very compact representations of the Q function in the presence of an overwhelming number of irrelevant, noisy state attributes. It was used by McCallum (1995) to learn behaviors in a complex driving simulator. However, G-learning cannot acquire partitions in which attributes are significant only in combination with other attributes.

The U Tree (Uther and Veloso 1998) algorithm extends G-learning to work in a continuous state space. Like G-learning, this algorithm keeps historical information. Occasionally, it does batch processing to determine what states need to be split. They evaluated the Kolmogorov-Smirnov test and sum-squared error as a means to determine where and when to split a leaf node, but did not evaluate using the T-test.

I developed an extension to G-learning that works in domains with real-valued variables. My extension is similar to U Tree, but I explored some alternative metrics for determining when and where to subdivide the space and analyzed performance in different domains. Also, the U Tree algorithm uses batched updating of the state space representation while my approach does continuous updating. With batch updating, all leaf nodes are inspected periodically. Continuous updating inspects leaf nodes whenever they are updated, which results in a more uniform distribution of evaluations over time, and avoids evaluating leaf nodes that have not been updated recently. My approach is covered in Chapter 4.

Agents need to satisfy two opposing problems regarding their internal state space. First, the agent's state space may have *too many distinctions*, meaning that an abundance of perceptual data is available but the level of detail is too finely grained to be of any practical

use. In this case, the agent needs to apply selective perception to prune away unnecessary distinctions and focus its attention only on certain features. Second, even though there are too many distinctions in the perceptual data, the agent's state space may simultaneously contain *too few distinctions*, meaning that perceptual limitations have temporarily hidden crucial features of the environment from the agent. In other words, the agent's state model is too coarse. This problem, called *hidden state*, can often be solved by using memory of features from previous views to augment the agent's perceptual inputs.

Andrew McCallum (1994, 1993, 1995) developed an approach to reinforcement learning called Utile Distinction Memory (UDM) that uses an instance-based approach to learn the state space representation. UDM uses selective perception and short-term memory to simultaneously prune and augment the state space provided by the agent's perceptual inputs. During learning, the agent selects task-relevant state distinctions with a *utile distinction* test. The utile distinction test is based on statistics, similar to G-learning, to determine when a distinction helps the agent predict reward. McCallum also advocates using instance-based learning for making efficient use of accumulated experience, and using a tree structure to hold variable-length memories. The basis of UDM is to use previous state information to disambiguate the current state. UDM keeps a long history of sensory inputs that can be applied to learn the value of new states that are discovered, thus it requires a large amount of computer memory.

Utile Distinction Memory scales much better than table lookup methods and finds good approximations to the value function. McCallum explored four main variations on Utile Distinction Memory and showed very good results in several domains. In some cases, the UDM performance was an order of magnitude better than previous algorithms. UDM may provide improved performance over my simpler decision tree based reinforcement learning, but at the cost of significantly higher computing resources.

H-learning is a model-based, average reward approach that has been shown to converge more quickly and to better policies than Q-learning in some domains (Tadepalli and Ok 1994; Tadepalli and Ok 1998). Bagnell, Doty, and Arroyo (1998) compared Q-learning with H-learning on a real robot. They used Charm, a small, two-wheeled robot with limited processing capabilities, for their experiments. The sensors and actuators on Charm are simple enough to use a table lookup method for reinforcement learning. H-learning worked better than Q-learning for Charm, but does not scale well to a more sophisticated robot. I use Q-learning to allow scaling to more complex robots.

Chapter 4

Reinforcement Learning of Actions

The goal of this dissertation research was to develop a system that can learn adaptive low level actions. This required some initial experiments to determine an appropriate approach to learning the low level actions. For testing alternative low level learning designs, I chose a domain that required only low level actions with minimal coordination and no long-term planning.

I tested reinforcement learning implementations for low level control using the Robot Auto Racing Simulator (RARS) (Timin 1995), in which different robots compete in automobile races. This simulator is complex enough to be interesting, while simple enough that all the variables can be examined and/or controlled. The purpose of the preliminary study was to gain knowledge about learning low level actions that may be applicable to more complex domains.

I ran two sets of experiments. The first set of experiments helped to select a reinforcement learning architecture that could learn to race in the RARS simulator, and that could be manually extended to learn a new behavior for passing competing cars. These experiments resulted in a system that performed adequately, but the reinforcement learning method did not converge reliably and was subject to overtraining. The second set of experiments, covered in Section 4.3, evaluated an alternative reinforcement learning method to provide more reliable convergence and immunity from overtraining. These experiments guided the design of a more general robot navigation architecture described in Chapter 7.

4.1 Environment: Simulated Automobile Racing

RARS is designed to allow different robots to compete in automobile races. The primary user interface to RARS consists of a rich set of command line options. The user controls the number of robots participating in the race and selects which robots to use, which track to use, how many laps in a race, how many races to run, and whether or not the physical model is deterministic. RARS provides the user with a real time display of race statistics, such as the maximum and average speeds for each car in the race, and the relative positions of each car. RARS can simulate from 1 to 16 different robots simultaneously and can display the race in real time, or can be called to run the race at faster than real time speed with or without the display.

Each robot is implemented as a subroutine that is compiled into RARS. To achieve real-time simulation, each robot subroutine is required to execute within a limited time. The exact time depends upon the speed of the processor and the size of the time slice during

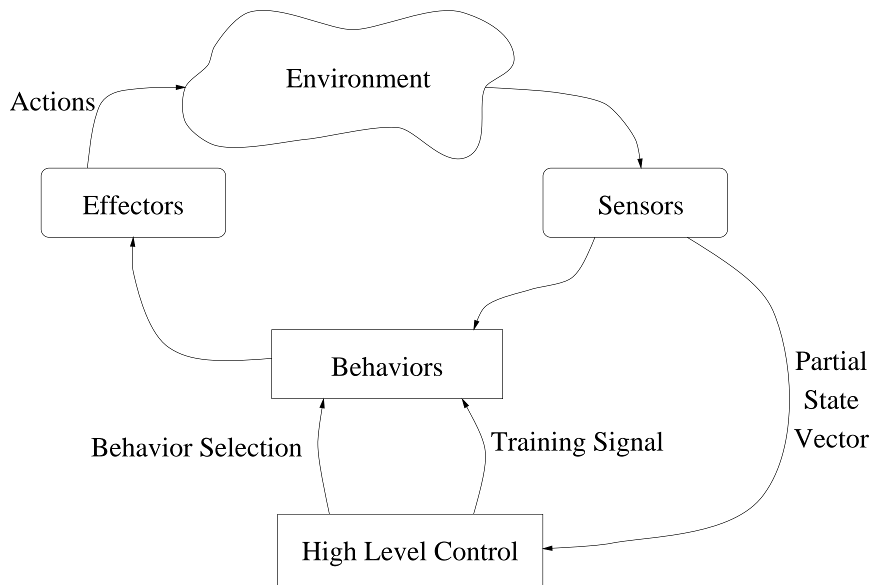


Figure 4.1: Simplified agent architecture for RARS experiments.

simulation.

RARS is a discrete time simulator. At each time step, RARS updates the position and velocity of each car. It then calls the subroutine for each robot in the race. Each robot routine receives a structure containing information about its position, current velocity, distance to the end of the current track segment, curvature of the track, and relative position and velocity of nearby cars. The robot routine calculates command signals for accelerating and steering, which are returned to RARS for updating and continuing the simulation.

Several agents are included in the standard RARS distribution. These agents were contributed by programmers from all over the world. All of the agents are heuristic, and none have the capacity to learn.

The environment, as reflected in the required control signals, requires the robot to control two interacting effectors: accelerating and steering. Thus, this environment leaves little question of what outputs to include in the architecture, but how to coordinate them and how to code the outputs and sensors are open questions.

4.2 RARS Agent Architecture

As shown in Figure 4.1, the robot racing architecture consists of two levels: low level behaviors and a simple coordinating mechanism. Each low level behavior was implemented as a reinforcement learning network.

The higher level coordinating mechanism is a simple set of heuristics responsible for ensuring that the car moves around the track. This coordinating mechanism is a placeholder for the POMDP planner in the complete robot control architecture. As such, it provides just enough functionality to test reinforcement learning design decisions in the RARS domain.

In addition to the reinforcement learning low level behaviors, one low level behavior was hard-coded. If the car leaves the track, a hard-coded low level behavior pilots the car back onto the track, returns the car to a safe state, and ensures that progress will be made in the correct direction on the next time step. The hard-coded low level behavior is also given

control whenever the car is moving in the wrong direction around the track.

To be successful, the agent is required to quickly respond to its environment and must have mastered several skills: racing, passing, and making pit-stops. Although the agent could have several behaviors available to it, only one behavior was active at any time. The active behavior receives inputs from the environment and sends control signals to the effectors. The control mechanism selects which behavior is active and assigns reinforcements. This architecture is extensible and provides a means to let the agent learn to race with no other agents in the environment. Once the agent has learned its racing behavior, new behaviors can be added for interaction with other agents.

4.2.1 Low Level Behaviors

The final RARS agent had two learned behaviors and one hand coded behavior. The core behavior for this domain is `race`, which is responsible for driving the car around the track as quickly as possible when there are no other cars nearby. Each behavior must coordinate two complex interacting activities: *steering* and *accelerating*. The learned behaviors begin with no knowledge and learn purely by trial and error. Of course, this results in a lot of crashes. While the reinforcement learning networks were being trained, the system used a hand coded heuristic behavior, `recover`, to recover from crashes. The `recover` behavior was implemented as a set of simple rules. A second learned behavior, `pass`, was added in the second experiment to direct actions needed to pass another vehicle. The `pass` behavior was critical for success when racing against other agents.

All behaviors received the same set of inputs to indicate the current state. A behavior could use a any subset of the available inputs. The inputs were

1. speed of the car,
2. orientation of the car relative to the direction of the track,
3. distance to left edge of track,
4. distance to right edge of track,
5. velocity of the car perpendicular to the direction of the track,
6. current setting of the wheel,
7. current setting of the pedal,
8. radius of the current track segment,
9. radius of the next track segment, and
10. distance to end of current track segment,
11. bearing to nearest competing car,
12. distance to nearest competing car,
13. bearing to next nearest competing car, and
14. distance to next nearest competing car.

Two distinct control signals were required for driving the car: pedal and wheel. Each behavior was responsible for selecting the appropriate settings for both of these control signals simultaneously. Each control signal was divided into three possible actions. For the pedal, the three possible actions were to accelerate by 10 ft/sec, remain at the same speed, and decelerate by 10 ft/sec. The possible wheel commands were to adjust steering left by 0.1 radians, do not adjust steering, or adjust steering right by 0.1 radians. Limiting the set to three possible actions is similar to the bang-bang control strategy used by Anderson (1989) for controlling an inverted pendulum and has the advantage that it limits the number of possible actions and, therefore, limits the control functions which must be learned.

4.2.2 Control Mechanism

The purpose of the control mechanism was to coordinate the settings for the pedal and wheel so as to optimize speed and competitive advantage while maintaining safety. Coordination was achieved through two mechanisms: behavior selection and reinforcement. The behavior selection mechanism determined which behavior was active at any given point, meaning that their control signals were passed to the simulator. The behavior selection mechanism was implemented as a pre-defined heuristic which switched control based on human intuition of when to switch. The reinforcement signal coordinated the learning of separate networks so that their actions worked together rather than clashing.

Without a good reinforcement signal, the reinforcement learning networks may not provide the correct response or may never converge. Two different reinforcement signals were used during system development. The first reinforcement signal was used to train the reinforcement network that was responsible for setting the pedal. The reinforcement signal is set to -1 if the car is off the track, or the speed of car is less than 15MPH, or if the car just completed a lap and the time for the lap is greater than the time required to complete time the previous lap. Otherwise, the reinforcement is 0. This function provided negative reinforcement for leaving the track and for going slowly. The slight bias in the network obviated the need for positive reinforcement.

The second reinforcement signal was used to train the reinforcement network that was responsible for setting the wheel. The reinforcement signal is calculated as: The reinforcement signal is set to -1 if the car is off the track or if the car just completed a lap and the time for the lap is greater than the time required to complete time the previous lap. Otherwise, the reinforcement is 0. Since the steering network was not directly responsible for the speed of the car, the negative reinforcement for going too slowly was not included in the reinforcement signal. However, the steering network could affect the time that it takes to complete a lap. For instance, by electing to steer to the inside on curves, the steering network could reduce the distance traveled on a particular lap. For this reason, the reinforcement signal for lap time was included.

4.2.3 Experiments

Three experiments were performed to test design decisions. The first experiment tested three variations on neural network architectures for reinforcement learning. The second experiment compared a fourth architecture to the best of the original three architectures. The third experiment verified that additional behaviors could be added to the system.

Each reinforcement learning network was tuned, trained, and tested on a single track in RARS. For the first experiment, performance was measured in length of time needed to

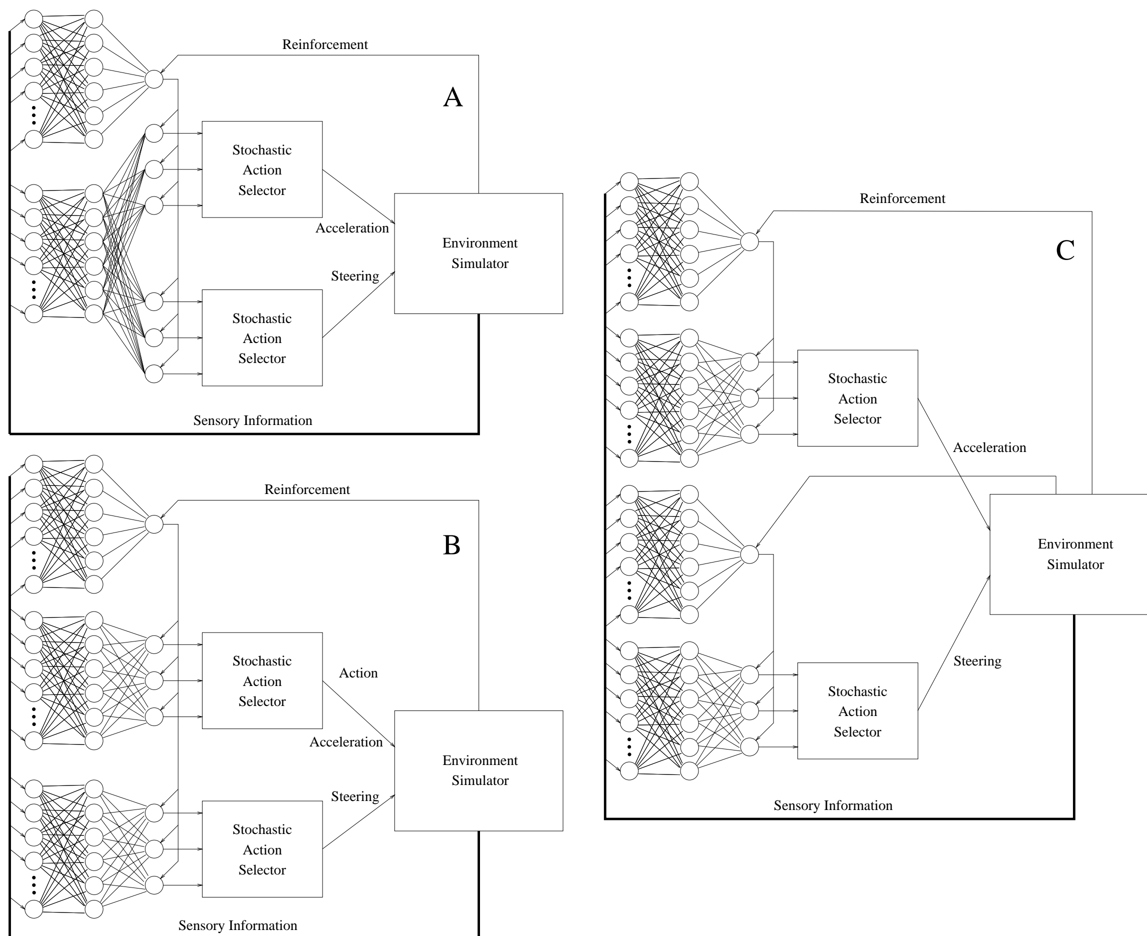


Figure 4.2: Three reinforcement learning networks to implement the race behavior. A) Single Utility Single Action, B) Single Utility Multiple Action, C) Multiple Utility Multiple Action.

complete one lap and the failure rate after a certain number of races. A failure occurred whenever the car left the track. For the second experiment, "damage" was used as the primary evaluation measure. A car sustained damage whenever it collided with another car or left the track. Thus, the damage measure includes the simpler failure measure, but is more appropriate for measuring how well the agent is doing when racing against other cars. In both experiments, the number of time steps required to complete each lap was also measured. This measure is inversely related to the average speed maintained by the car for each lap. During the course of the experiments, the reinforcement learning architecture, the input representation, and the number of behaviors were changed.

4.2.3.1 Experiment 1: Monolithic vs. Decomposed

The first experiment used actor-critic networks similar to those of Barto, Sutton, and Watkins (1989), Anderson (1989), and Lin (1992) for the learned `race` behavior. Section 3.4.3 gave an overview of actor-critic reinforcement learning. This experiment compared three organizations of actor-critic reinforcement learning for the two control signals. The monolithic network used a single utility network and a single action network (SS). The

hybrid network used a single utility network but multiple networks for choosing an action (SM). The decomposed network used multiple utility and multiple action networks for speed and steering (MM). Figure 4.2 shows the three network architectures. The reinforcement learning networks had nine inputs, six hidden nodes, and one output for each action.

The decomposed approach (MM) exhibited the best performance after training. Figure 4.3 shows the relative performance for the three approaches on the two performance measures. The two graphs are similar because fewer failures mean less time spent recovering from failure.

4.2.3.2 Experiment 2: Tuning Learning

Following the results of the first experiment, I used separate networks for each possible action. This approach required each network to learn the utility for a single action. The experiments with actor-critic networks indicated that this approach was superior to using a single network with multiple output units. The resulting agent was able to drive around the track, but still performed poorly when compared to all of the agents in the RARS distribution. This experiment investigated whether tuning of the learning networks could significantly improve performance.

The input representation, learning algorithm, and reinforcement protocol were all changed for this experiment. In the first experiment, the networks were given inputs as a continuous input signals. In the second experiment, inputs were represented by 21 bit vectors, in which one of the bits is set to one to indicate direction while all other bits are set to zero.

Since the first experiment showed that the MM approach worked best and there appeared to be no advantage to sharing the critic network between action networks, I also used Q-learning instead of the actor-critic approach to reinforcement learning. One advantage to Q-learning was that it only required a single network for each control signal, as opposed to the actor-critic approach, which needed two networks. Since there were fewer neurons to train, I expected Q-learning to converge more quickly that actor-critic. The single network in Q-learning is used both to select actions and estimate the value of those actions. The neural network has one output for each possible action, and uses the current state s as input. At each time step, $Q(s, a)$ for all a is calculated and the action with the highest value is chosen probabilistically.

The implementation of Q-learning used 15-step truncated TD(λ) instead of simple 1-step returns. TD(λ) is described in Section 3.4.4. The previous n states were stored in a queue and learning at each time step was performed on the nth previous state. The return R_{t-n} to state s_{t-n} was calculated by the recursive equation

$$R_{t-1} = \gamma\lambda R_t + r_t + \gamma(1-\lambda)Q(s_t, a_t) \tag{4.1}$$

where R_t was the truncated return to state t of future rewards, r_t was the reward given at time t, and $Q(s_t, a_t)$ was the value of the state action pair chosen at time t. When using the above equation, R_t was initially set to zero. γ is a decay term that discounted future returns. λ is a parameter that determines the weight of future returns. Once the return had been calculated, the value of the state action pair $Q(s_{t-n}, a_{t-s})$ was updated using

$$\Delta Q(s_{t-n}, a_{t-n}) = \alpha\left[R_{t-n} - Q(s_{t-n}, a_{t-n})\right]. \tag{4.2}$$

$\Delta Q(s_{t-n}, a_{t-n})$ was used to train the neural network output unit corresponding to the action a_{t-n}. The other neural network outputs were not trained for that time step. This allowed

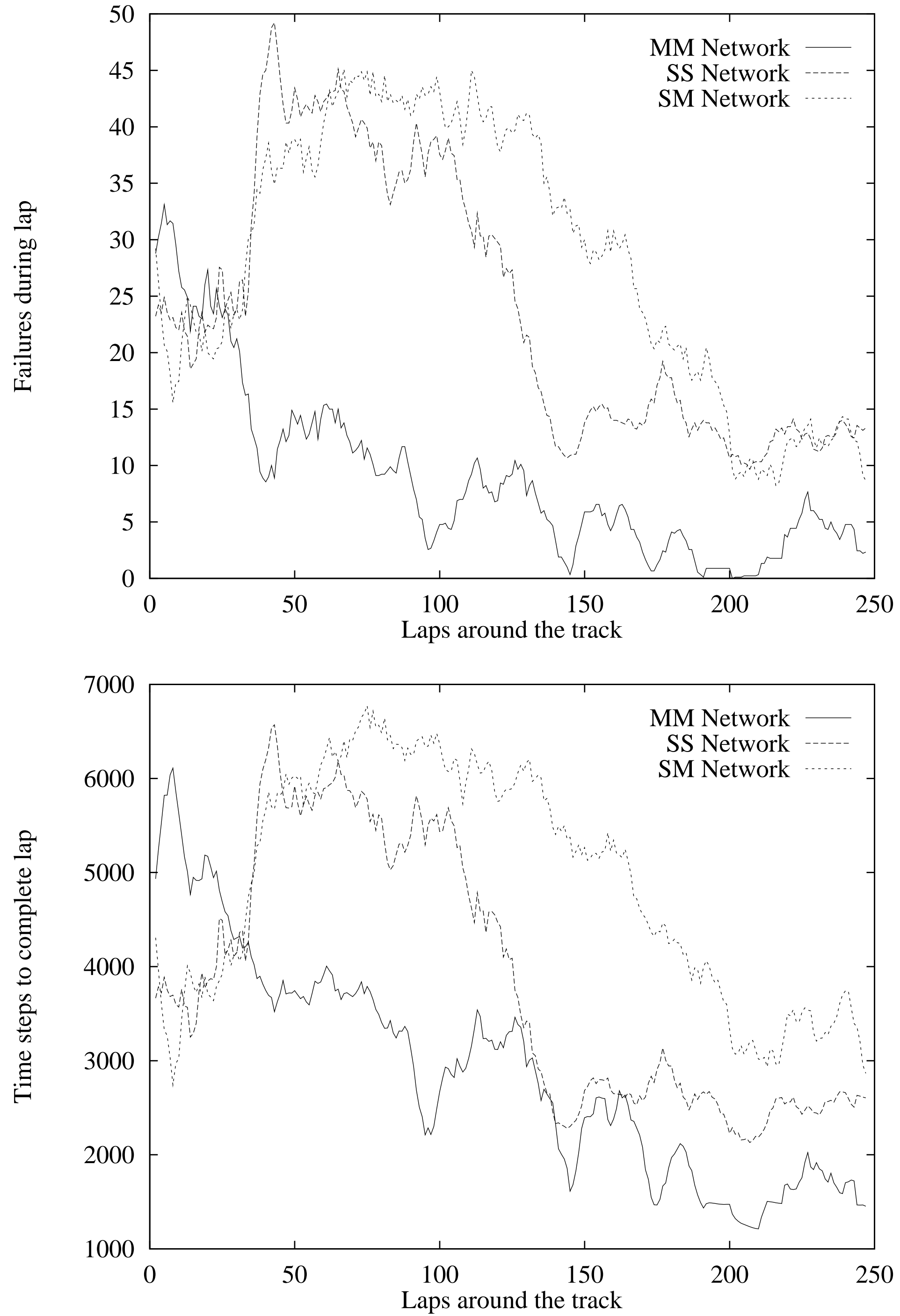


Figure 4.3: Comparison of three reinforcement learning networks to implement the race behavior.

the algorithm to compute a more accurate value for the return and causes reinforcement learning to converge more quickly.

Tuning the learning significantly improved performance. As shown in Figure 4.4, the Q-learning approach learned to avoid failure much more rapidly. The time to complete a lap was much lower with the new approach and was competitive with some of the heuristic control strategies that were supplied with RARS. Some of the best heuristic strategies could complete a lap in under 600 time steps on average while the worst heuristic strategies took more than 800 time steps. After sufficient training, the Q-learning approach could complete a lap in about 700 time steps on average.

4.2.3.3 Experiment 3: Adding a New Behavior

Although the agent performed well when it was the only vehicle on the track, it did not have any strategy for avoiding other vehicles. RARS detects collisions between cars and calculates damage when a collision occurs. When a car has accrued enough damage points, it is removed from the race. To solve this problem, I added the `pass` behavior to direct the interaction with other agents on the track and allow it to safely pass the other cars.

When the high level control mechanism detected a car ahead, it gave control to `pass`, which steered around the other vehicle. `Pass` was given 15 time steps to move around the other car. If the car left the track or ran into the other car before the 15 time steps had expired, a reward of -10 was given, and the `recover` behavior was given control. Otherwise, the `pass` behavior was given a reward of 1, and the `race` behavior was given control.

For training, I created three slow moving rule-based cars: blocker1, blocker2, and blocker3. These three cars were derived from the simple "cntrl0" car which comes with RARS. Blocker1 was adjusted to stay with 25% of the track to its left. Blocker2 was set to stay at the center of the track, and Blocker3 was set to stay with 75% of the track to its left. The blocker cars were set to run at 35 miles per hour; the average speed of a car in the RARS distribution is about 65 miles per hour. If only one blocker was used, the normal `race` behavior would learn to stay in a part of the track where the blocker was absent, reducing the need for a passing behavior. Using three blocker cars forced the agent to make passing maneuvers.

The `race` behavior was trained alone on the track for 1000 laps, and then the `pass` behavior was initialized and trained on the track for 1000 laps with the three blocker cars. Neural network weights were saved and used to run 100 races against the blocker cars. Each race ended when the slowest moving car completed two laps.

The passing behavior greatly reduced the amount of damage sustained by the car and decreased the number of time steps required to complete a lap. Figure 4.5 shows the damage accrued by the car and the time steps required to complete a lap during the test races with and without the passing behavior enabled. Both tests were started with the weights initialized to the values saved after training the passing behavior. The average speed for the car was 63.5 miles per hour, slightly slower than the 65 miles per hour average speed for the best five cars in the RARS distribution, but a considerable improvement over earlier performance. The periodicity in the graphs of Figure 4.5 is a result of the periodic alignment of the blocker cars. Since they are all running at fixed speeds, there is a periodic alignment of the blocker cars, presenting a more difficult passing situation.

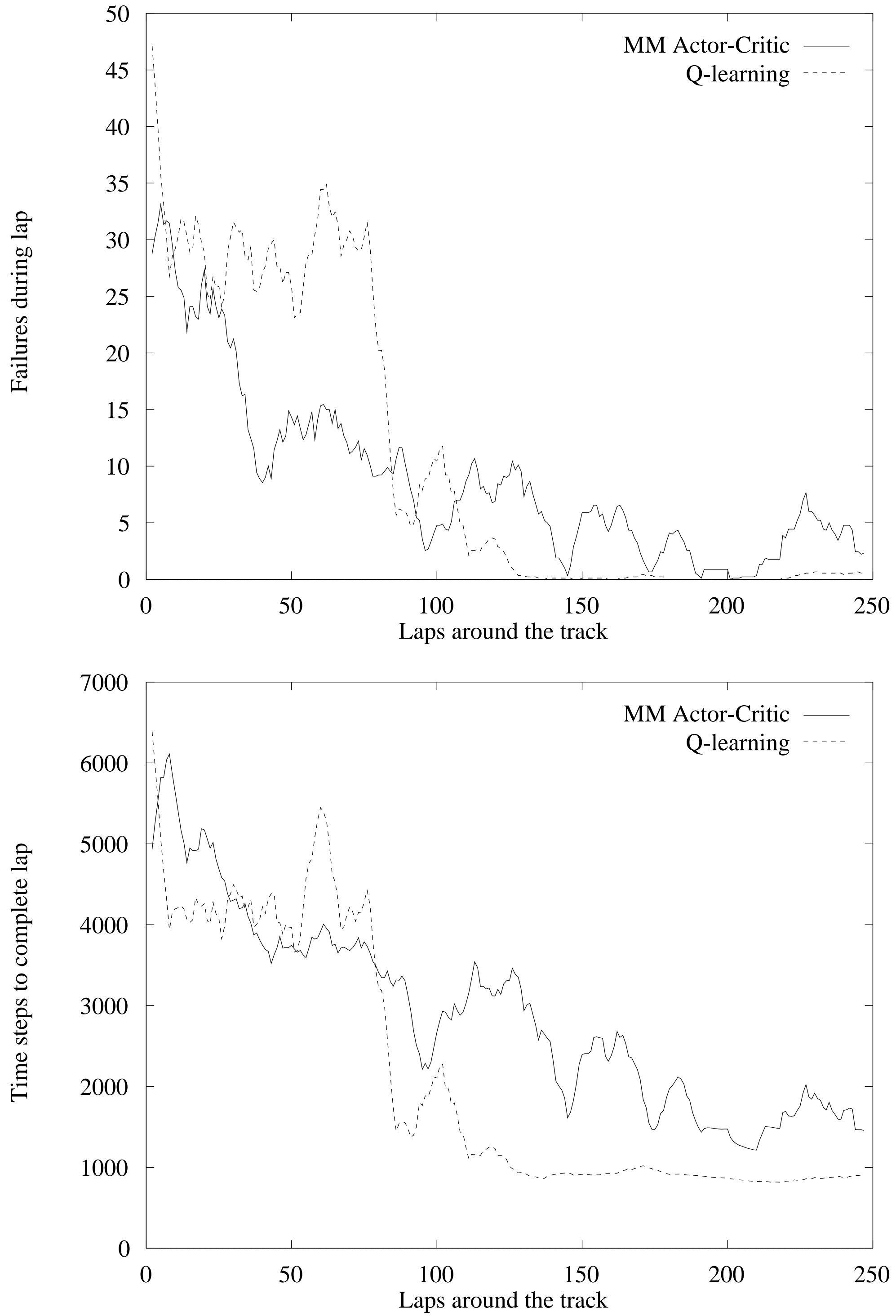


Figure 4.4: Learning performance for the improved learning strategy.

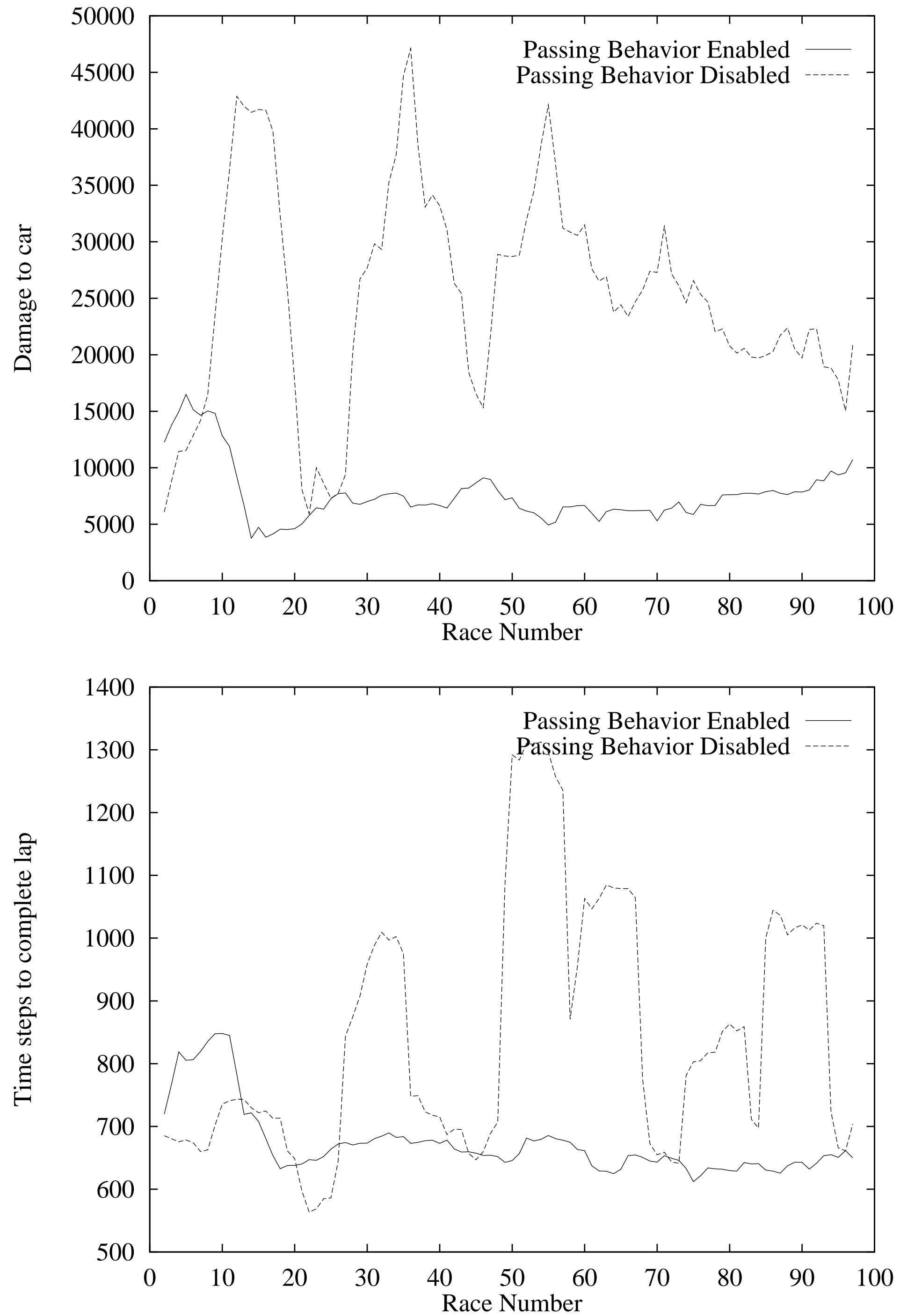


Figure 4.5: Using the passing behavior to improve performance.

4.2.4 Results of Initial Experiments

The first experiment confirmed that for the RARS environment, a monolithic architecture is not only less modular, and so harder to extend, but also learns less quickly and achieves a lower overall level of competence.

The second experiment showed that reinforcement learning with neural networks can be made to work in this domain, but requires careful tuning of parameters. As expected, tuning the learning is critical for performance. This is a disappointment because it means that for an agent to build its own low level behaviors it may need both considerable time to tune them and knowledge about how to do so.

The third experiment demonstrated that the human developer can add new behaviors easily to the decomposed behavior set. The passing behavior was incorporated by creating a new network with an identical input representation and learning algorithm to the other behaviors. A reinforcement schedule was designed for the passing behavior, and a switch added to the heuristic control selection mechanism. The passing behavior was trained together with the normal race behavior.

The final agent used Q-learning with separate networks for each of three behaviors, coarse coded input, and a coordination mechanism combining simple rule-based selection of behaviors, reinforcement signals and synchronized training. This best agent differed significantly from the first pilot agent that was built. Incremental testing of the design decisions was used to develop an agent that was well suited to its environment.

My preliminary work showed that learning and composing low level behaviors is feasible for this domain. Decomposing the behavior into separate modules for steering and acceleration resulted in better performance than using a single monolithic control module. The RARS work also resulted in a better understanding of suitable neural network reinforcement learning architectures and parameter settings.

4.2.5 Concerns and Questions

The agent could be further improved by finding a more appropriate function approximation technique. Neural networks were less than ideal because they do not perform localized learning. When the network adjusted the value of one state, the weight change could affect other, very different states. This is a very undesirable feature in reinforcement learning. Table lookup methods provide localized learning, and reinforcement learning has been proven to converge for these systems. However, table lookup does not scale well with the size of the input space. Reinforcement learning would benefit greatly from a function approximation technique which combines the scalability of neural networks with the localized learning of table lookup methods.

Although this preliminary work has shown that reinforcement learning can be used for low level control, it was not clear how higher level planning can best make use of this learning ability. In the RARS domain, I used a simple high level control strategy to select behaviors to execute. This strategy will probably not work well for a more complex agent. Also, in this simple form, there was no mechanism for learning new behaviors that are not explicitly specified by the programmer.

Although neural network based reinforcement learning worked adequately for performing in RARS, some problems remained. The main problem with neural reinforcement learning was a tendency towards over-training and forgetting. Once the network had learned an adequate policy, the system no longer visited states that were not along the optimal tra-

jectory. Thus, the information used to train the neural network came from only a small portion of the input space. The problem with using neural networks in this situation was that training in one portion of the state space caused errors to increase in other parts of the state space. In effect, the network forgot the values for states that it had not visited recently. This typically resulted in a cycle where the robot learned to perform the behavior well and then suddenly began to perform very poorly. Eventually, it re-learned what it had forgotten and its performance improved again. This problem is significant, because the behaviors in a more general architecture should be reliable so that higher level planning is not disrupted.

Another concern is why the reinforcement learning method never reached average performance compared with the heuristic cars. One possible explanation is that the neural network function approximator did not accurately learn the value function. This could be affected by the input representation, neural network learning rate, or several other factors. My next set of experiments were designed to test a relatively unexplored approach to function approximation in reinforcement learning with the goal of improving performance and reducing tuning requirements.

4.3 Decision Tree Function Approximation in Reinforcement Learning

Representation of the value function is a major concern when implementing a reinforcement learning strategy. Q-learning is guaranteed to converge as long as certain constraints are met in the implementation of the value function and the size of the step that is taken at each iteration. Of major importance is the property of *local support* in the value function approximation method (Boyan and Moore 1995). When updating the value function for a given state, the update should generalize somewhat to nearby states, but should not affect states that are distant. This section explains my decision tree function approximation technique for reinforcement learning.

4.3.1 Table Lookup

A popular approach for representing the value function in reinforcement learning is the table lookup method. This approach is guaranteed to converge, subject to some restrictions on the learning parameters (Sutton and Barto 1997b). However, table lookup does not scale well with the number of inputs. The straightforward table lookup method subdivides the input space into equal intervals. Each part of the state space has the same resolution. Some variations of this approach, such as sparse coarse coding and hashing (Sutton 1996), have been used to improve scalability somewhat.

Even the simplest behavior for the RARS robot uses seven inputs. Using table lookup with each input dimension divided into 10 regions would result in 10^7 table entries. Not only does this require a lot of memory in the computer, but it makes the algorithm converge very slowly. Each of the 10^7 states must be visited many times before an accurate approximation to the value function is learned, resulting in slow convergence. A better approach would allow high resolution only where it is needed and lower resolution everywhere else. This would result in fewer states overall and still afford high resolution where it is needed.

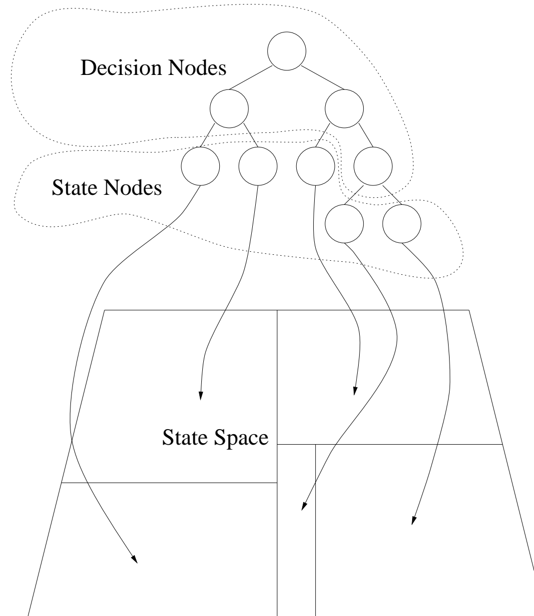


Figure 4.6: Dividing the state space with a decision tree.

4.3.2 Neural Network

The naive approach to reinforcement learning with neural networks typically uses a back-propagation neural network with a sigmoidal activation function to learn the value function. This approach can solve larger problems than table lookup, but it is not guaranteed to converge. In practice, the neural network approach often performs poorly even on relatively simple problems (Boyan and Moore 1995).

Some techniques are designed to reduce this learn-forget cycle, but the basic problem is that local update is critical for RL to work reliably (Gordon 1995). In the experiments of Section 4.2.3, two techniques were used to minimize this behavior: reducing the learning rate as learning progressed, and carefully tuning the neural network parameters. This greatly reduced the period of the learn/forget cycle, but did not totally alleviate the problem. A limited form of experience replay was implemented, but did not result in measurable reduction in the learn/forget cycle.

In the experiments of Section 4.2.3, the learn/forget cycle was not a major problem because there were only a few behaviors. Each behavior would work correctly for about 90% of the time and then perform poorly for 10% of the time. With only a few behaviors, this was acceptable because the probability of more than one behavior being in the forget part of the cycle at any time was still relatively low, but because it executes behaviors serially, this could cause poor performance when a behavior is executed while in the forget phase. However, as the number of behaviors increases, more of them will be performing poorly at a given time. This may cause severe problems as the system tries to recover from multiple failures caused by malfunctioning behaviors.

4.3.3 Decision Trees

Decision trees (Murthy 1996) divide the space with varying levels of resolution. Figure 4.6 shows an example of a decision tree that divides the state space into five regions. Each

decision node in this example makes a decision about one of the input variables. This type of decision tree is referred to as axis-parallel because decision boundaries are always parallel to the axes. The decision tree maps any input vector from the state space to one of the state nodes, which corresponds to a region in the state space.

Given a decision tree that maps input vectors onto state space regions, reinforcement learning can be used to associate a value with each region. This is just a generalization of the table lookup method, since a lookup table can be represented as a decision tree. However, the more general tree representation allows state space regions to vary in size. The decision tree has high resolution of the state space where it is needed and low resolution elsewhere, thereby reducing the number of discrete states. Reducing the number of states can greatly reduce the amount of time needed for reinforcement learning to find a good policy.

Using a decision tree to approximate the value function is not a new idea. Section 3.4.4 describes some previous approaches. My approach extends G-learning to a continuous state space, while requiring less historical information than UDM. The decision tree is learned along with the policy. My work is very similar to the U Tree algorithm, but I explored some alternative metrics for determining when and where to subdivide the space and analyze performance in different domains. Also, the U Tree algorithm uses batched updating of the state space representation while we do continuous updating. With batch updating, all leaf nodes are inspected periodically. Continuous updating inspects leaf nodes whenever they are updated, which results in a more uniform distribution of evaluations over time and avoids evaluating leaf nodes that have not been updated recently.

Variable resolution dynamic programming (Moore 1991) is one method for performing reinforcement learning in real valued state spaces where straightforward discretization would fail due to the curse of dimensionality. In this algorithm, a kd-tree (similar to a decision tree) is used to partition the state space into coarse regions. The coarse regions are refined into detailed regions, but only in parts of the state space which are predicted to be important. This notion of importance is obtained by running "mental trajectories" through the state space. This algorithm proved effective on a number of problems for which full high-resolution arrays would have been impractical. It has the disadvantage of requiring a guess at an initially valid trajectory through state-space.

4.3.4 Implementation

The decision tree contains two types of nodes: decision nodes and leaf nodes. A *decision node* represents a single decision about one input variable. This decision determines which branch to take to find the next node. Each *leaf node* stores the estimated values for its corresponding region in the state space. The algorithm uses Q-learning as described in Section 3.4.2, and each leaf node stores one value for each possible action that can be taken, along with a history of the inputs and temporal difference errors that have been received. The history list is used to decide whether the region represented by the node should be split.

The decision tree starts out with only one leaf node that represents the entire input space. As the algorithm runs, the leaf node gathers information in its history list. When it has enough information, a test is performed to determine whether the leaf node should be split (see Section 4.3.4). If a split is required, the test also determines the decision boundary. A new decision node is created to replace the leaf node, and two new leaf nodes are created and attached to the decision node. The old leaf node is then deleted. In this manner, the tree grows from the root downward, continually subdividing the input space

1. Receive input vector v and reward r_t for time t.
2. Use input vector v to find a leaf node representing state s_t.
3. Select the action a with the largest value of $Q(s_t, a)$, or select a random action with some small probability ρ.
4. If $\rho > \rho_{min}$ then $\rho \leftarrow \rho \times decay$, were $0 < decay < 1$.
5. If the action was not chosen at random, calculate $\Delta Q(s_{t-1}, a_{t-1})$ and update $Q(s_{t-1}, a_{t-1})$.
6. Add $\Delta Q(s_{t-1}, a_{t-1})$ and v to the history list for the leaf node corresponding to s_{t-1}.
7. Decide if s_{t-1} should be divided into two states by examining the history list for s_{t-1}.
   (a) if history_list_length < history_list_min_size then split := False
   (b) else
      i. calculate average μ and standard deviation σ of $\Delta Q(s_{t-1}, a_{t-1})$ in the history list
      ii. if $|\mu| < 2\sigma$ then split := True
      iii. else split := False
8. Perform split, if required
9. Save r_t, a_t and s_t so that they can be used for training on the next iteration.
10. Return a_t.

Figure 4.7: Algorithm for decision tree based reinforcement learning.

into smaller regions. Figure 4.7 summarizes the algorithm.

The decision about when a node should be split and where to place each decision boundary is crucial. I investigated three methods from the decision tree literature plus a new method that is similar to G-learning. All four of the methods use mean and standard deviation of $\Delta Q(s_t, a_t)$ (covered in Section 3.4.2) over some fixed history to determine if a node should be split (see Figure 4.7), but they use different algorithms to choose a decision boundary. The decision about where to place the boundary is made by testing the improvement in sorting the samples in the history list by $\Delta Q(s_t, a_t)$, placing a boundary at the mean between each pair of samples, and then performing the test on each input variable. The boundary/input variable pair that results in the best partitioning of the input space, as determined by the partitioning method, is used to split the node. The four methods are:

Information Gain This is the classic method used in Quinlan's ID3 (Quinlan 1986). It measures the information gained from a particular split. The boundary with the largest information gain is used to split the node.

Gini Index This metric is based on the Gini Criterion by Breiman et al. (1984), but modified as in OC1 by Murthy, Kasif, and Salzberg (1994). The Gini Index measures the probability of misclassifying a set of instances. The boundary with the lowest probability of misclassification is used to split the node.

Twoing Rule This metric, also proposed by Breiman et al. (1984) and used by Murthy, Kasif, and Salzberg (1994), compares the number of examples in each category on each side of the proposed split. The boundary that most evenly divides the samples into two groups is used to split the node.

T-statistic This approach is based on the T-statistic. The algorithm calculates the means

and variances for each input variable. If the node has not received any positive $\Delta Q(s_t, a_t)$ in its history list, then the input variable with the highest variance is chosen as the decision variable for the new decision node. Otherwise, the decision is made by calculating the T-statistic for each variable and selecting the variable with the highest T-statistic. This approach is similar to that used in G-learning, although I remove the restriction that all inputs be binary, allowing the algorithm to learn in a continuous state space.

4.3.5 Experiments on Decision Tree Reinforcement Learning

To assess the performance of the decision tree based reinforcement learning algorithms, I compared them to table lookup and neural network reinforcement learning on three problem domains. The study focused on answering the following questions:

1. How quickly does each algorithm learn?
2. Which reinforcement learning algorithm performs best after training?
3. Is the decision tree based approach less prone to the learn/forget cycle than the neural network approach?

The performance study used the RARS domain, described in Section 4.1, and two simpler domains from the reinforcement learning literature.

Mountain car is a classic reinforcement learning task where the goal is to learn the proper acceleration to get out of a valley and up a mountain (Boyan and Moore 1995). The car does not have enough power to simply climb up the mountain, so it has to rock back and forth across the valley until it gains enough momentum to carry it up the mountain.

Pole balance is another classic problem where the goal is to balance a pole that is affixed to a cart by a hinge (Barto, Sutton, and Anderson 1983). The cart moves in one dimension on a finite track. At each time step, the controller decides whether to push the cart to the left or to the right.

The neural network approach used a backpropagation network with sigmoidal activation function. The θ parameter, which controls the steepness of the activation function, was set to 1.0 for all experiments. The learning rate, or step size, parameter was set to 0.01 for the pole balance and mountain car problems, and was set to 0.1 for RARS. The momentum term was set to 0.0 for all experiments. These parameters were chosen after some trials in each domain with various parameter settings. For the pole balance problem, the network had four input nodes, six hidden nodes, and 2 output nodes. For mountain car, there were two input nodes, three hidden nodes, and three output nodes. For RARS, there were two networks, each with nine inputs, twelve hidden nodes, and three output nodes.

For each method, Q(λ) (Watkins 1989) was used. For pole balance and mountain car, λ was set to 0.75 and γ was set to 0.95. For RARS, λ was set to 0.8 and γ was set to 0.99. The random rate parameter, ρ was initially set to 0.25 and was recalculated as $\rho \leftarrow \rho \times 0.99$ on each iteration of Q(λ) where $\rho > 0.001$. The time step for simulating mountain car and pole balance was 0.02 seconds. The time step for the RARS simulation was 0.0549 seconds. For the decision tree approach, the minimum history list length before a node could be considered for splitting was set to 200. In general, long history lists result in

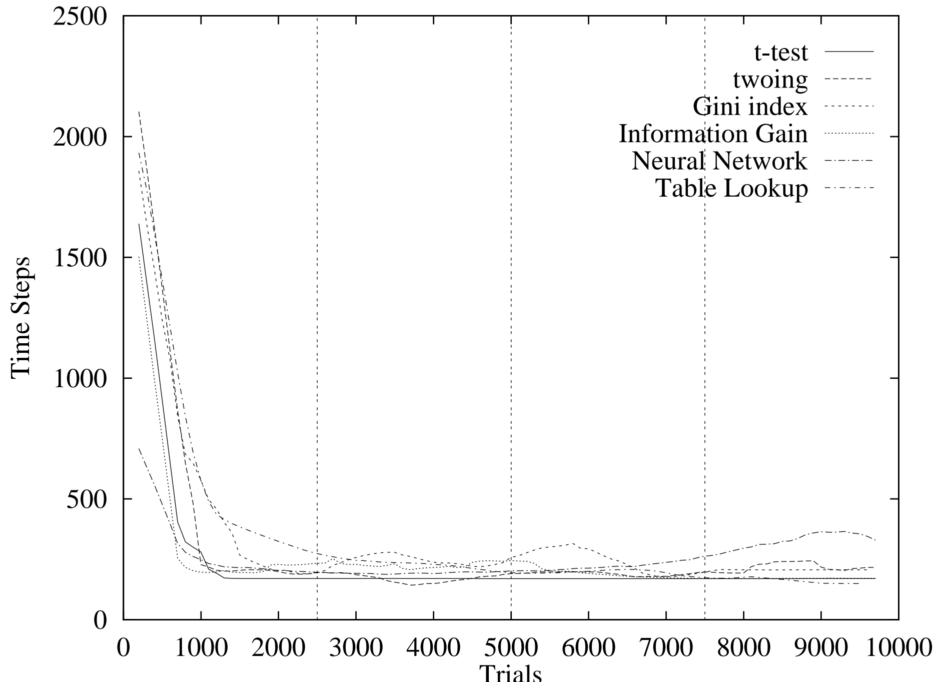


Figure 4.8: Smoothed learning performance on the mountain car problem.

slower learning, while short history lists do not provide enough information. A history list length of 200 provides enough information to make good decisions about when and where to split the nodes while allowing the system to learn at a reasonable rate.

For each problem domain, I ran the reinforcement learning algorithms with the six representations for a fixed number of trials and then divided the total number of trials for each algorithm into four periods. The number of trials in each domain was determined by how long it took the algorithms to reach a stable policy. On the mountain car problem, I ran each algorithm for a total of 10,000 trials; each trial lasted for 10,000 time steps or until the car reached the goal state. On the pole balancing problem, we ran each algorithm for 20,000 trials of 2,000 time steps or until the controller failed to maintain the pole in a balanced position within the confines of the track. Extending the trials beyond 2,000 time steps did not affect the results. In the RARS domain, each algorithm was run for 300 laps around the track, regardless of how many time steps were required.

All experiments were run 10 times in each domain, and I calculated the average performance for each trial. For the Mountain car domain, I collected the number of steps taken to reach the goal. For pole balancing, I collected the time that the pole remained balanced. For the RARS domain, I collected the time that the car took to complete each lap around the track.

How long does it take for each algorithm to reach a good, stable solution? Figures 4.8–4.10 show the performance of the various algorithms.

Mountain car: This is the easiest of the three problem domains, having only two inputs and three possible actions. As shown in Figure 4.8, all of the methods converged to about the same level of performance, with little difference in the time that they took to find a good solution.

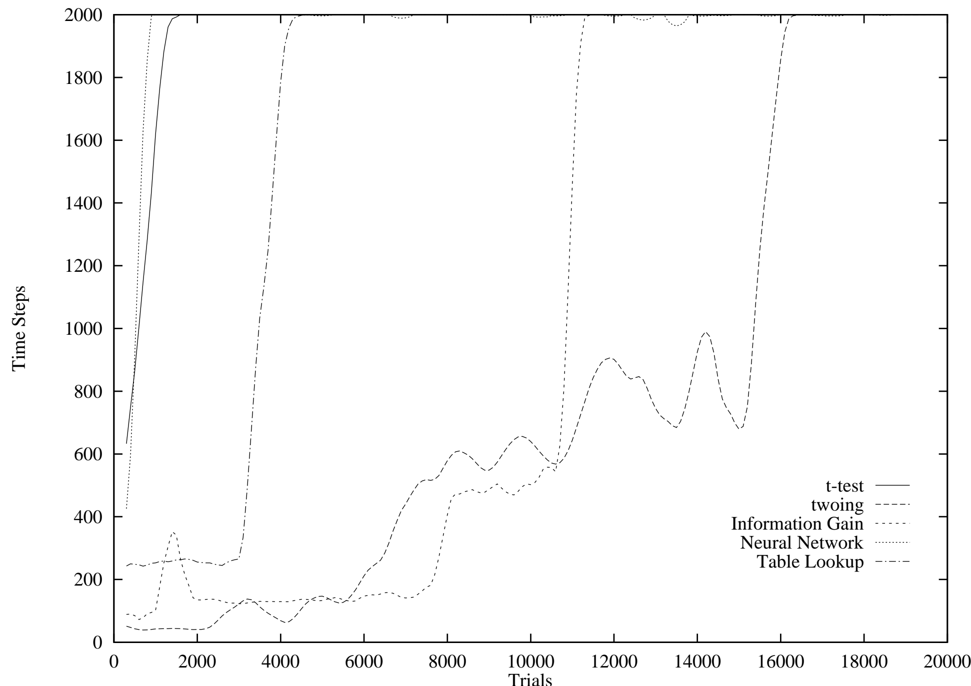


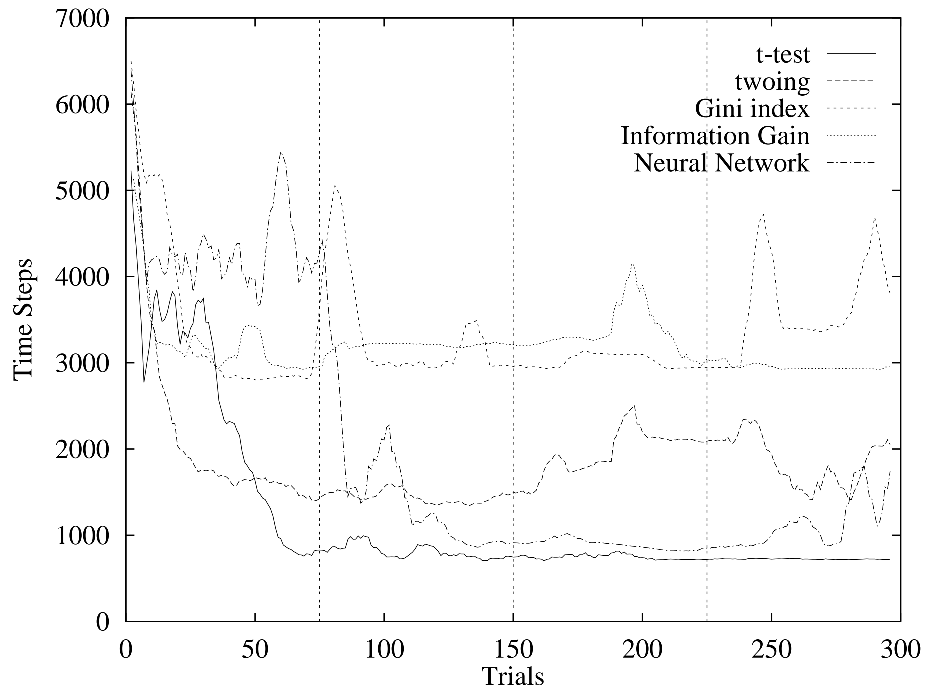
Figure 4.9: Smoothed learning performance on the pole balance problem. Gini index did not converge for this problem and is omitted from the graph.

Pole balance: This domain is a little more difficult than the Mountain car problem. Figure 4.9 shows the relative performance of the various algorithms on this problem. The T-test decision tree method converged to a stable solution after less than 2,000 trials on average and successfully balanced the pole thereafter. The table lookup method took a little longer, but still found a solution in only 5000 trials. The neural network approach found a good solution in fewer trials than any other method, but showed some instability even after 10,000 trials.

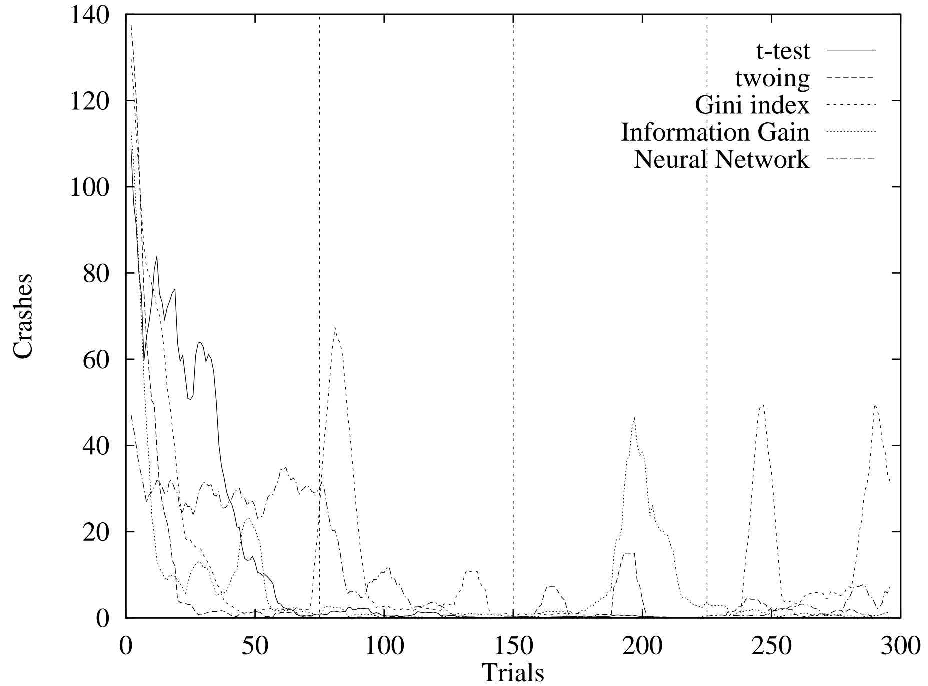
RARS: This problem is too large to be solved using the table lookup method, so I only compare the performance of the various decision tree methods and the neural network approach. Figure 4.10 shows the performance of these algorithms. The T-test decision tree method quickly finds a good state-space representation and achieves good performance. The neural network approach also finds a good policy. The other decision tree methods do not perform as well; they were still searching for suitable state-space representations at the end of the run.

Overall, it appears that the T-test decision tree method performs best. The neural network approach worked well in all three domains, but showed evidence of overtraining in all cases. Overtraining causes the upward trend in time steps after 7000 trials on the mountain car problem, the small performance dip between trials 12000 and 14000 on the pole balance problem, and the upward trend in the RARS data after 250 laps.

For each problem domain, I ran all of the reinforcement learning algorithms (neural network, table lookup, and four variations of decision tree) and then divided the total run time for each algorithm into four periods. ANOVA is a statistical technique which tests differences in mean values of a dependent variable between two or more categories of



(a) time steps per lap.



(b) number of crashes per lap.

Figure 4.10: Smoothed learning performance for RARS.

Table 4.1: Performance during the fourth period. Each column shows average and standard deviation.

	Mountain Car		Pole Balance		RARS Crashes		RARS Times	
	μ	σ	μ	σ	μ	σ	μ	σ
Gini	206	0.7	No result		15.7	22.4	3617	751.8
Info Gain	170	2.2	2000	0	1.1	1.9	2956	82.8
NN	328	45.4	1999	0.004	2.8	4.6	1172	566.0
Table	160	13.6	2000	0	No result		No result	
T-test	170	0.3	2000	0	0.01	0.1	723	14.8
Twoing	219	53.4	1848	360.2	1.4	2.7	1869	418.0

independent variables. I calculated a one way ANOVA on each period with algorithm as the independent variable and performance as the dependent variable. For all problems, $P < 0.01$, indicating statistically significant performance differences (mountain car: $F = 113.45$, pole balance: $F = 219.67$, RARS lap time: $F = 506.13$, and RARS crashes: $F = 29.43$). Table 4.1 shows the performance for each algorithm over the fourth period, where learning should be complete and performance should be stable.

Mountain car: The table lookup method performed better than any other method. However, information gain and T-test decision tree methods also performed well. I performed a one tailed T-test of information gain vs. T-test methods and found no significant difference ($t = 1.35, p < 0.18$). The table lookup method had higher standard deviation than either information gain or T-test, indicating that the state space representation of the decision tree approaches may have been better suited to the problem than the fixed grid representation used in table lookup.

Pole balance: Information gain, table lookup, and T-test all achieved perfect performance and were able to balance the pole for 2000 time steps for every trial in period four. The neural network approach found a good solution, but failed occasionally due to overtraining. The twoing decision tree converged to a stable solution during period four.

RARS: The T-test approach shows significantly better performance than the other methods. The neural network approach performed well, but crashed a great deal more often than any of the top three decision tree methods.

Overall, the T-test method performed best, providing the best lap times and the lowest number of crashes in the RARS domain, and giving perfect performance in pole balancing. On the mountain car problem, the T-test method was slightly worse on average than table lookup, but performance had far less variation from one trial to the next.

The main motivation for this work was to overcome the learn/forget cycles that I had encountered using the neural network approach. I assessed whether a method suffered from a learn/forget cycle by examining the learning curves and comparing standard deviation (shown in Table 4.1) in its ultimate (period four) performance.

Mountain car: The neural network approach initially found a good policy, but became overtrained after about 6,000 trials. The high standard deviations show that the neural network and twoing value approaches were unstable. Figure 4.8 shows that

the performance of the neural network method actually deteriorated noticeably during period four. The T-test, Gini, and information gain methods had low standard deviations, indicating that they had reached a stable solution.

Pole balance: The T-test, table lookup, and information gain methods all had zero standard deviation and perfect performance. The twoing method found a good solution during period four, as shown in Figure 4.9. The neural network approach found a good solution, but occasionally failed due to overtraining.

RARS: The T-test method had the lowest standard deviation in both performance measures indicating that it had found a stable solution. The neural network approach performed almost as well as the T-test based decision tree method for some time. However, after about 250 trials, the neural network became overtrained. Its high standard deviation indicates that it did not converge to a stable solution at that time. Figure 4.10a shows that the performance of the neural network approach was good throughout period three and actually decreased during period four.

In all three of the domains, the neural network approach showed signs of overtraining. The T-test approach performed best in two of the domains and was outperformed by table lookup in one domain. The table lookup method gave the best performance in the mountain car and second best on pole balancing, but could not be used on the RARS domain due to the number of inputs.

4.3.6 Conclusions for Decision Tree Reinforcement Learning

Decision tree function approximation for reinforcement learning provides good learning performance, meets my needs for more reliable convergence than the neural network approach, and did not exhibit overtraining in the domains tested. It also has lower memory requirements than the table lookup method and scales better to large input spaces.

Chapter 5

Review of POMDP Planning

A Partially Observable Markov Decision Process (POMDP) model can be viewed as a generalization to an MDP model in which the exact state may not be observable (Littman 1996). One example is a robot lacking enough sensors to tell exactly where it is, or is able to determine its exact state only by running a time consuming vision system.

A POMDP model, introduced by Al Drake (Platzman 1998), consists of

- a finite set of states $\mathcal{S} = \{s_0, s_1, s_2, \ldots, s_{|\mathcal{S}|-1}\}$,
- a finite set of actions $\mathcal{A} = \{a_0, a_1, a_2, \ldots, a_{|\mathcal{A}|-1}\}$,
- a set of actions $A(s) \subseteq \mathcal{A}$ for each state $s \in \mathcal{S}$ that can be executed in that state,
- a set of transition probabilities $\Pr(s'|s, a) \forall s \in \mathcal{S}, s' \in \mathcal{S}, a \in A(s)$,
- a set of observations $\mathcal{Z} = \{z_0, z_1, z_2, \ldots, z_{|\mathcal{Z}|-1}\}$,
- the observation probabilities $\Pr(z|s', a) \forall z \in \mathcal{Z}, s' \in \mathcal{S}, a \in A(s)$,
- and a set of immediate rewards $r^a(s) \forall a \in A(s), s \in \mathcal{S}$ that are available after taking any legal action from any state.

The transition probabilities simply specify the probability that the state is s' given that the previous state was s and action a was taken. The observation probabilities specify the probability of seeing observation z given that the current state is s' and action a was taken in the previous state. A more general case of POMDP problem would include stochastic rewards instead of deterministic rewards. However, this would greatly complicate the process of finding a policy.

Although the underlying dynamics of the POMDP are still assumed to be Markovian, there is no direct access to the current state. The problem now becomes one of disambiguating the current state in order to make the optimal action choice. To solve this problem, the current state is represented as a probability distribution b over $\mathcal{S}$ known as the *belief state*. In the POMDP model, after each action, a sensor observation is made, and the belief state b is updated. Updating the probability distribution just involves using the transition and observation probabilities along with the current belief state, the action chosen, and the observations received to calculate a new probability distribution b over $\mathcal{S}$. The exact method is covered in Section 5.1. At each time step, the next belief state is fully determined by the current belief state, the action chosen, and the observations received. Since

the process of maintaining the belief state is Markovian, it provides the same information as if the complete history were maintained. Although there may be many ways to arrive at the current belief state, the current decision and the next belief state do not depend on what path was taken to arrive at the current belief state. The following section explains how the action is selected at each time step and defines some terminology. Later sections in this chapter describe several algorithms for finding the optimal action selection policy for a POMDP.

5.1 POMDP Basics

This section describes how actions are selected from the POMDP policy and how the belief state is updated at each time step. Much of the material in this section is derived from Cassandra (1998a). Just as in the COMDP model, the robot selects an action at each time step by consulting its policy. For this discussion, assume that a policy been found for some POMDP problem, and that policy is being used to select an action at each time step. The policy in a POMDP is a function $\pi(b) \rightarrow a$ that maps belief states to actions. The policy is stored as a set $\mathcal{V}$ of vectors (or line segments), with each vector having an associated action. It is common in the POMDP literature to refer to $\mathcal{V}$ as the policy, although it is more properly referred to as the value function. The distinction between value function and policy is seen as unnecessary for a POMDP because the value function maps directly to the policy. Each action a is associated with a set of vectors $\mathcal{V}(a) \subseteq \mathcal{V}$ such that $\mathcal{V}(a) \cap \mathcal{V}(a') = \emptyset \, \forall \; a, a' \in \mathcal{A}, a \neq a'$. At each time step t, the action to perform is defined as

$$a_t = \operatorname*{argmax}_{a \in \mathcal{A}} \left(\max_{v \in \mathcal{V}(a)} (b \cdot v) \right). \tag{5.1}$$

In other words, the dot product is taken between the current belief state and each vector in the policy. The vector with the highest dot product wins, and its associated action is chosen. The action is then executed, and the belief state is updated for the next iteration.

Consider a very simple POMDP with two states ($\mathcal{S} = \{s_0, s_1\}$), two actions ($\mathcal{A} = \{a_0, a_1\}$), and three observations ($\mathcal{Z} = \{z_0, z_1, z_2\}$). Figure 5.1 shows how belief space for this problem can be represented.

Since there are two states and the belief space must satisfy the constraint $\sum_{s \in \mathcal{S}} b_s = 1$, the belief space can be represented as a single, thick line between s_0 and s_1. In general, the belief space of a POMDP with n states can be represented as the hyper-plane in $n-1$ dimensions for which $\sum_{s \in \mathcal{S}} b_s = 1$. The two state example is labeled by belief state $(1, 0)$ on the left and $(0, 1)$ on the right. On the far left, the probability of being in state s_1 is 0, which means that the probability of being in state s_0 is 1. On the far right, the probability of being in state s_1 is 1, and there is 0 probability that s_0 is the true state. If the current belief state is somewhere in between, as shown in the Figure 5.1, then the actual state is unknown, but the true state is s_1 with some probability b_1 or s_0 with probability $b_0 = 1 - b_1$.

Assume that the robot starts with a particular belief state b, takes action a_1, and receives observation z_1 after taking that action. There are a finite number of actions and a finite number of observations; so given any belief state, there are a finite number of possible next belief states. These correspond to each combination of action and observation. Since the example has two actions and three observations, there are exactly six possible new belief states. Note that there may be some cases where one of the six possible new belief states

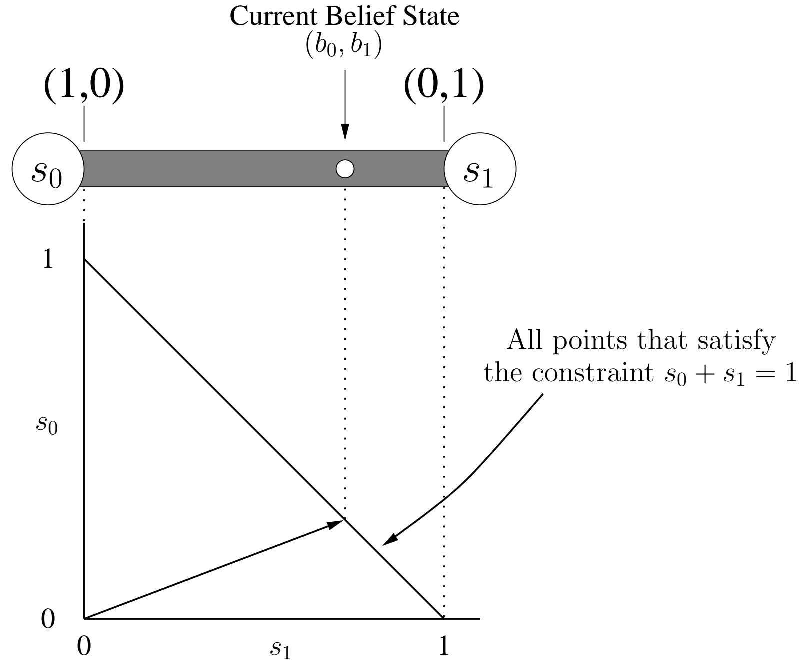


Figure 5.1: A graphical representation of belief state showing how the belief state for a two-state POMDP problem can be represented as a point on a line segment or as a dot in a bar connecting the two states.

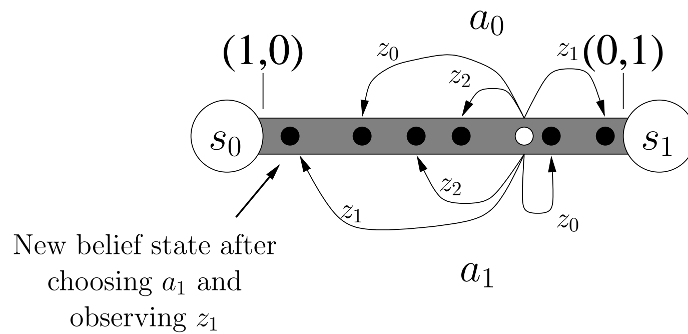


Figure 5.2: The new belief state depends on the action chosen and the observation received after executing that action. The three arcs at the top represent possible transitions when a_0 is chosen, while the arcs at the bottom represent possible transitions when a_1 is chosen.

is identical to the current belief state. Figure 5.2 shows this process graphically, with arcs representing the possible transformations of the belief state.

The arcs along the top of the graph represent possible outcomes when a_0 is chosen, while the arcs on the bottom represent what can happen when a_1 is chosen. Knowing which action was chosen and what observation was made completely determines the new belief state. Note that the new *belief state* can be determined, but this is not the same as determining the exact *state*. The new belief state $b_z^a(s')$ can be calculated with the belief state maintenance equation

$$b_z^a(s') = \frac{\Pr(z|s',a)\sum_{s\in\mathcal{S}}\Pr(s'|s,a)b(s)}{\Pr(z|b,a)} \tag{5.2}$$

where

$$\Pr(z|b,a) = \sum_{s'\in\mathcal{S}}\left[\Pr(z|s',a)\sum_{s\in\mathcal{S}}\Pr(s'|s,a)b(s)\right].$$

At each time step, Equation 5.2 is evaluated for all $s \in \mathcal{S}$. In terms of Figure 5.2, Equation 5.2 selects which arc is followed to find the new belief state.

The process of maintaining the belief state is Markovian. The next belief state depends only on the current belief state, action, and observation. This property means that a discrete space POMDP problem can be seen as a continuous space COMDP problem. Discrete space COMDP problems can be solved with value iteration, and the value iteration algorithm can be adapted to work with a continuous space belief space. The transitions of this new continuous space COMDP are easily derived from the transition and observation probabilities of the POMDP using Equation 5.2. The problem is reduced to solving a COMDP by using value iteration, but the COMDP does not have discrete states, so the value iteration algorithm must be adapted.

5.1.1 Problem of Continuous State Space

Value iteration cannot be applied directly because the belief state space is not discrete. In discrete state COMDP value iteration, a table with one entry per state is maintained. The value of each state is stored in the table, providing a finite representation of the value function. Since the POMDP has a continuous belief state space, the value function may be some arbitrary continuous function. Obviously a table representation cannot be used. Thus, the first problem is how to easily represent this value function.

The POMDP formulation imposes some restrictions on the form of the solutions to the continuous space COMDP that is derived from the POMDP. It has been shown that the finite horizon value function is piecewise linear and convex (PWLC) for every horizon length (Papadimitriou and Tsitsiklis 1987; Sondik 1971). Since the value function is PWLC, it is only necessary to find a finite number of linear segments that make up the value function at each iteration, and that set of line segments is all that needs to be stored. Remember that although the value function in the two state example is made of lines, the PWLC property is also true as lines become planes in higher dimension problems. Figure 5.3 shows a sample value function over belief space for a POMDP. The value function is the upper surface of a finite number of linear segments. In the figure, the current belief state maps to a line segment labeled a_0 and to other line segments labeled a_1. The a_0 line segment provides a higher value for the current belief state, so it is chosen as the action for this time step. If the current belief state were close to s_0, then the winning line segment would be

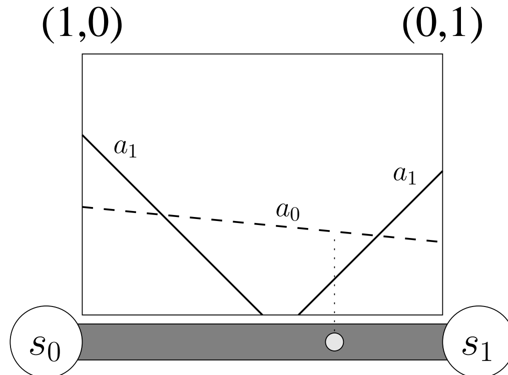


Figure 5.3: A graphical representation of a policy for a two state POMDP. The solid lines represent the value of taking action a_1, while the dashed line represents the value of taking action a_0. At the current belief state, represented by the dot, the value of action a_0 is greater than the value for action a_1, so a_0 is the best action to take.

the a_1 line segment on the left side of the figure. Likewise, if the belief state were close to s_1 then the winning line segment would be the a_1 line on the right of the figure. In either case, the action chosen would be a_1.

The following paragraphs present a quick example of how the action is chosen, given a POMDP policy. The lines in Figure 5.3 represent the policy $\mathcal{V} = \bigcup_i \mathcal{V}(a_i)$, where $\mathcal{V}(a_0) = \{v_0 = 2b_0 + 3b_1\}$ and $\mathcal{V}(a_1) = \{v_1 = 5b_0 - 5b_1, v_2 = -6b_0 + 6b_1\}$. For simplicity, the linear equations are represented as vectors, by taking only their coefficients. Thus, $\mathcal{V}(a_0) = \{\vec{v}_0 = (2,3)\}$ and $\mathcal{V}(a_1) = \{\vec{v}_1 = (5,-5), \vec{v}_2 = (-6,6)\}$. The belief state represented in Figure 5.3 can be represented by the vector $\vec{b} = (0.3, 0.7)$. To find the correct action to take in this belief state, the value of each action at the current belief state is calculated, and the action with the largest value is selected. This is expressed with the POMDP action selection equation:

$$\pi(\vec{b}) = a = \operatorname*{argmax}_{a \in \mathcal{A}} \left(\max_{v \in \mathcal{V}(a)} \vec{v} \cdot \vec{b} \right). \tag{5.3}$$

Applying this equation to the current problem yields:

$$\begin{aligned} \vec{v}_0 \cdot \vec{b} &= 2 \times 0.3 + 3 \times 0.7 = 2.7 \\ \vec{v}_1 \cdot \vec{b} &= 5 \times 0.3 - 5 \times 0.7 = -2 \\ \vec{v}_2 \cdot \vec{b} &= -6 \times 0.3 + 6 \times 0.7 = 2.4 \end{aligned}$$

The vector that maximizes the dot product is v_0, and since $v_0 \in \mathcal{V}(a_0)$, a_0 is chosen. Representing the value function for each horizon as a set of vectors provides a convenient method for choosing actions. The value of a belief state is given by the largest dot product of every vector in $\mathcal{V}$ with the belief state. From now on, the notational arrow is dropped, and it is assumed that the policy and belief state are represented as vectors.

Instead of vectors or linear segments over belief space, another way to view the value function is that it partitions belief space into a finite number of regions. Figure 5.4 shows how the linear segments are related to partitions of belief space. The value function and this partitioning representation are both used to explain the algorithms. The two representations are more or less interchangeable.

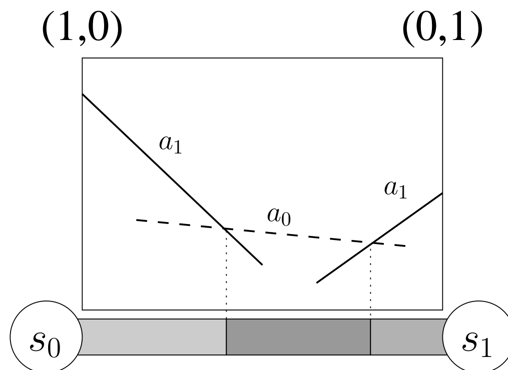


Figure 5.4: How the value function partitions belief space.

5.1.2 Finding the Value of Possible Next States

Representing the value function as a set of line segments solves the problem of the continuous belief space by allowing us to partition belief space into a finite number of regions. This partitioning is possible because the value function is always PWLC. However, the continuous belief space causes further problems when value iteration is applied.

The horizon indicates how far rewards have been propagated. For instance, a value function at horizon of 25 indicates that rewards from 25 steps in the future are incorporated in the value function. Value iteration starts at horizon one and iteratively updates the value function to include rewards from the next longer horizon. For each iteration of value iteration in a discrete state space COMDP, a state's new value at horizon $h+1$ would be found by looping over all the possible next states and summing the values at horizon h, weighted by the probability of ending up in that state and then adding in the immediate rewards to each state. The belief state representation transforms the POMDP into a COMDP with belief state replacing the true state. However, it is impossible to enumerate over all possible next belief states because the belief space is continuous and therefore the number of belief states is uncountably infinite. Using the PWLC value function, the value for any one next belief state can be found, but since the belief space is continuous, the value for *all* belief states cannot be found by enumerating over belief states.

There are several different approaches to solving the problem of value iteration over a continuous belief space, and they rely on the fact that the value function for a POMDP is PWLC so as to avoid enumerating all possible next belief states. The problem is now reduced to the following: Given a set of line segments (or vectors) representing the value function for horizon h, generate the set of line segments for the value function of horizon $h+1$. Thus, the line segments partition the belief space into a finite number of regions so that enumeration can be performed.

5.2 Finding Policies For POMDPs

This section gives an example of value iteration on a simple POMDP up to a horizon length of two. The simple POMDP problem described in Section 5.1 is used.

At horizon zero, the value of each state is assumed to be zero. Thus, when finding the value function at horizon one, no future rewards are considered; so only the immediate rewards are used to determine the new value function. The immediate rewards are part of

State	Action	Reward
s_0	a_0	1
	a_1	0
s_1	a_0	0
	a_1	1.5

Table 5.1: Immediate rewards for the sample problem.

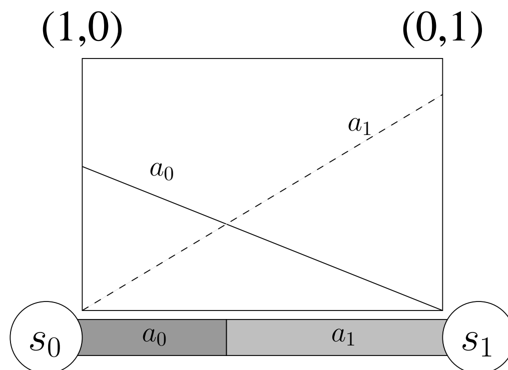


Figure 5.5: Value function at horizon one.

the POMDP model. In the example, there are two states and two actions; so the POMDP model will include four separate immediate reward values. These are the values of doing each action in each state. These values are defined over the discrete state space of the POMDP, but it is easy to get the value of doing a particular action in a particular belief state, because that is simply the probabilities in the belief state weighted by the value of each state.

As an example, suppose that the immediate rewards for the problem are described by Table 5.1. Figure 5.5 shows how these these values can be displayed over the belief space. The immediate rewards for each action specify a linear function over belief space. Since the goal is to choose the best action, whichever action gives the highest value is chosen. Thus, action selection depends on the particular belief state. Figure 5.5 also shows the partition of belief space that this value function imposes. The region on the left is defined by all belief states where a_0 is the best choice, while the region on the right represents all belief states where a_1 is best. If the belief state is [0.25 0.75], then the value of doing action a_0 in this belief state is $0.25 \times 1 + 0.75 \times 0 = 0.25$, and action a_1 has value $0.25 \times 0 + 0.75 \times 1.5 = 1.125$. Therefore, a_1 is chosen.

The horizon two value function can now be constructed from the horizon one value function. The goal in building this new value function is to find the best action (or highest value) achievable using only two consecutive actions (i.e., the horizon is two) for any initial belief state. The explanation of how this is accomplished will be presented in three stages: computing the value of the current belief state for a fixed action and observation, computing the value of a current belief state given only an action, and computing the value for a current belief state while considering all possible actions and observations. The explanation is broken down in this way to make it easier for the reader to understand.

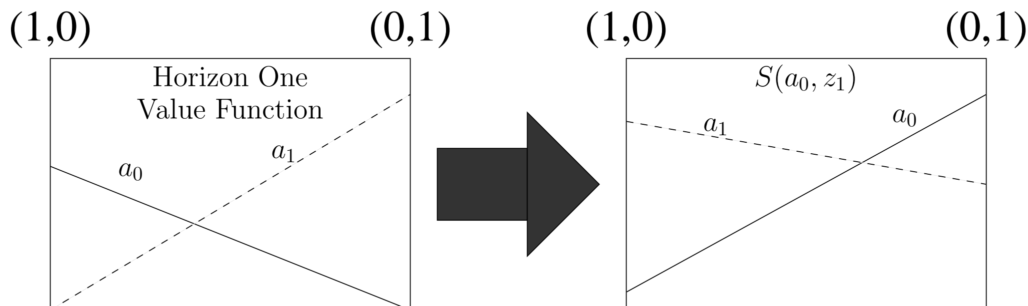


Figure 5.6: The value function can be transformed to give the value for the new belief state b' when taking action a_0 and observing z_1 from any initial belief state.

5.2.1 Value for a Fixed Action and Observation

Given a particular belief state, b, what is the value of doing action a_0 if, after the action, z_0 was observed? Using value iteration, the value of a belief state b for horizon two, given action a, is defined as the value of the best next action a' taken from the next belief state b'. In general, the best possible value would include all possible sequences of the two actions. However, the problem will be restricted to a single immediate action a_0 so that the immediate value is fully determined. This will simplify the explanation and represents the approach taken by most solution methods.

The goal is to find the best achievable value for the belief state b' that results from the initial belief state b when action a_0 is performed and z_1 is observed. Since the initial belief state, the action, and the resulting observation are all known, Equation 5.2 is applied to find a new belief state. This new belief state will be the belief state when there is one more action to perform, since the horizon length is two and one of the actions has already been determined. The best values for every belief state when there is a single action to perform is exactly what the horizon one value function specifies.

Suppose the belief state b is [0.75 0.25] and Equation 5.2 is applied to find the new belief state $b' = [0.3\ 0.7]$. As covered in the previous section, the horizon one value of doing action a_0 in belief state b is $0.75 \times 1 + 0.25 \times 0 = 0.75$. The single step value of taking action a_0 in b' is $0.3 \times 1 + 0.7 \times 0 = 0.3$ and the value of taking action a_1 in b' is $0.3 \times 0 + 0.7 \times 1.5 = 1.05$. Thus, a_1 is the best action to take in belief state b', and that action is chosen. Adding the values for taking a_0 in b and a_1 in b' results in a total value of 1.8 for starting in belief state b and performing action a_0, observing z_1 and then performing a_1. This is the best value for starting in b, performing a_0 and observing z_1, and also determines the best action to take in the next belief state b'.

This process can be repeated to find the value for a different initial belief state given the same action and observation. The same result can be obtained by simply transforming the belief state using Equation 5.2, then using the horizon one value function to find the value of the new belief state. Finding the value for all the belief points given this fixed action and observation seems a little more difficult, since the belief space is continuous, and therefore, the number of belief states to be transformed is uncountably infinite. However, it is possible to transform the value function itself, instead of transforming each belief state individually.

Since the action and observation are fixed, the horizon one value function is a function of the transformed belief state b' which is a function of the initial belief state b. It turns

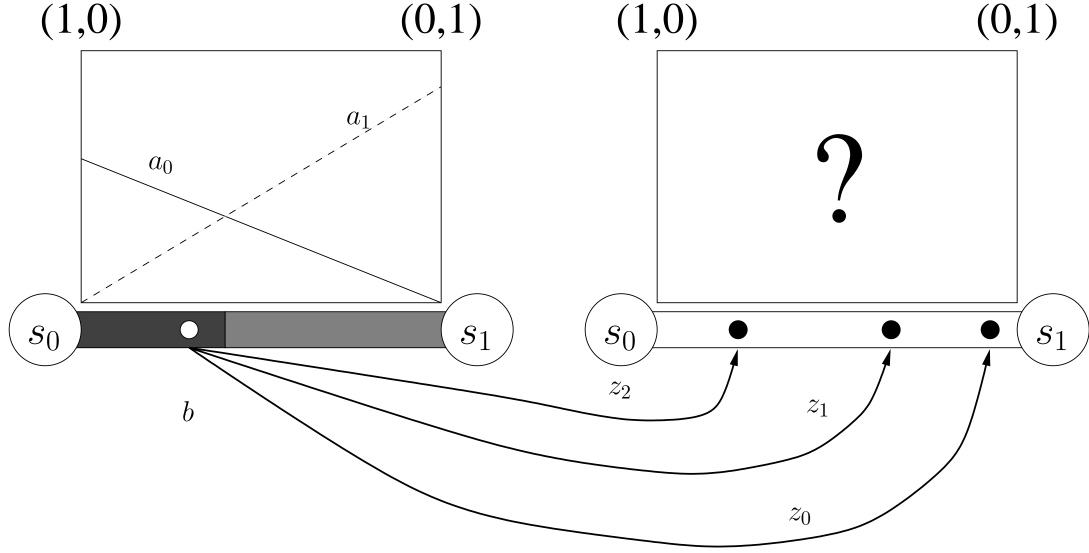


Figure 5.7: The transformed value function must account for all possible observations.

out that a function over the entire belief space can be constructed from the horizon one value function that has the belief transformation built in (Cassandra 1998a). This results in a function that maps the value of each belief state to its corresponding value after the action a_0 is taken and observation z_1 is seen. Figure 5.6 shows this transformation. The transformation preserves the number of linear segments, but changes the value of the segments over the belief space.

$S(a, z)$ represents the transformed value function for a particular action and observation. This function will be described later. The following paragraphs assume that $S(a, z)$ can be found.

5.2.2 Value for a Fixed Action

The preceding sections showed how to compute the conditional value of an individual point by getting its immediate reward value and then transforming it to find the resulting belief state value. This is the value if observation z_1 is seen. However, because the observations are probabilistic, there is no guarantee that z_1 will be seen. In this example, the three possible observations each lead to a separate resulting belief state, as in Figure 5.7.

Even though the action is known, the observation cannot be known in advance. To get the true value of the belief point b, it is necessary to account for all of the possible observations. Each observation for a given belief state has a certain probability associated with it. If the value of the resulting belief state for a given observation is known, then the value of the belief state can be found without knowing the observation. This is accomplished by simply weighting each resulting value by the probability of each observation.

Suppose that the values of the resulting belief states for belief state b, action a_0 and all three observations have been computed, and the values for each resulting belief state are $V(z_0, a_0) = 0.8$, $V(z_1, a_0) = 0.7$, and $V(z_2, a_0) = 1.2$. These are the values that were initially calculated when taking one belief point at a time. Weighting by the probabilities of each observation occurring, the probabilities of getting each of the three observations for the given belief state and action are found to be $V(z_0, a_0) = 0.6$, $V(z_1, a_0) = 0.25$, and

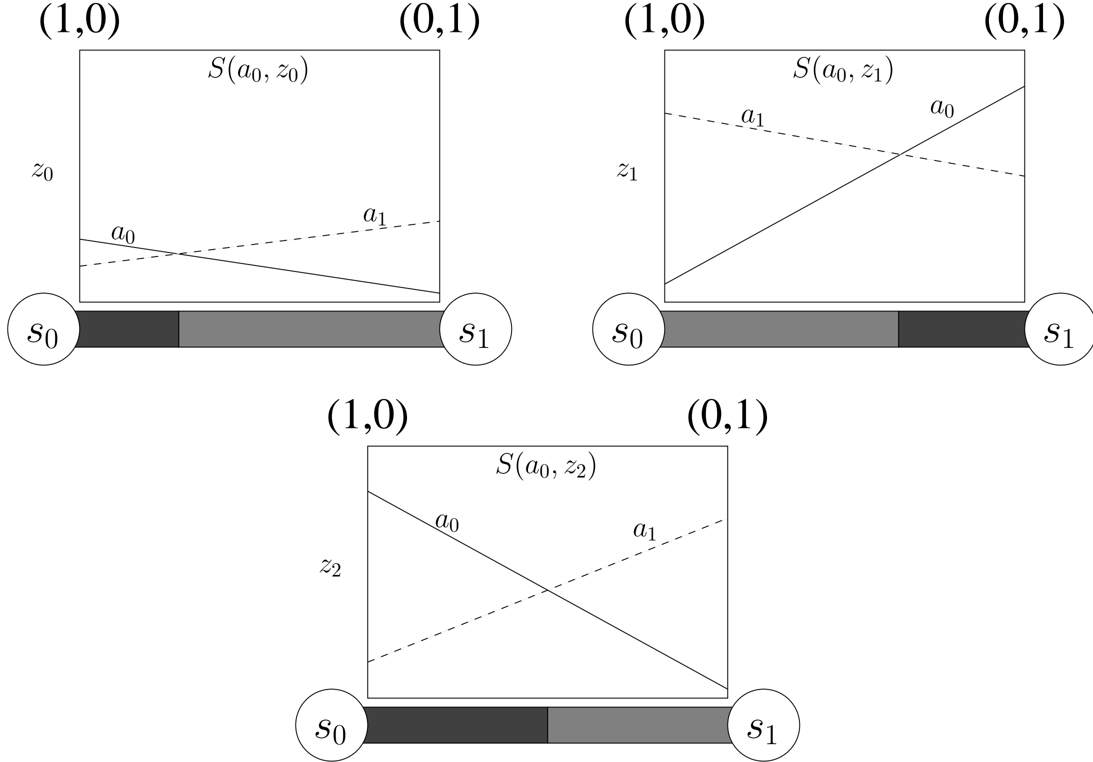


Figure 5.8: Each possible observation has a different transformation.

$V(z_2, a_0) = 0.15$. The horizon two value of the belief state b for action a_0 is $0.6 \times 0.8 + 0.25 \times 0.7 + 0.15 \times 1.2 = 0.835$ plus the immediate reward of doing action a_0 in b. This is the $S(a, z)$ function that was mentioned earlier. The transformed value function $S(a_0, z_1)$ actually factors in the probabilities of the observation. So in reality, the $S(a, z)$ function is not quite what was originally claimed. The $S(a, z)$ function already has the probability of the observation built into it.

Figure 5.8 shows the transformation of the horizon one value function for the action a_1 and all three observations. Notice that the value function is transformed differently for all three observations and that each of these transformed functions partitions the belief space differently. How the value function is transformed depends on the specific observation probabilities, state transition probabilities, and immediate rewards in the POMDP model. What all this implies is that the best next action to perform depends not only upon the initial belief state, but also upon exactly which observation is received.

The horizon two value of a belief state, given a particular action a_1 at time t depends not only on the value of doing action a_1 but also upon what action is performed at time $t+1$. However, the next action will depend upon what observation is received. For a given belief state and observation, the corresponding S function partition can be used to select the next action.

Figure 5.9 shows the S partitions from the previous figure all together. The light regions of the bars are the belief states where action a_1 is the best next action for each observation, and the dark regions are where a_0 would be best. Starting at the belief point b shown in Figure 5.9 and performing action a_0, the best next action would be a_0 if either z_0 or z_2 is observed and would be action a_1 if z_1 is observed. The best strategy after doing action a_0

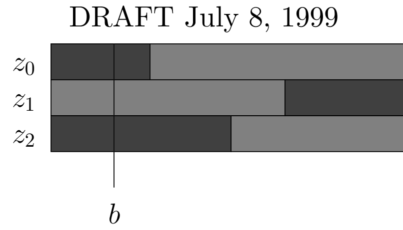


Figure 5.9: Partitions for the belief space given initial belief state b and action a_0.

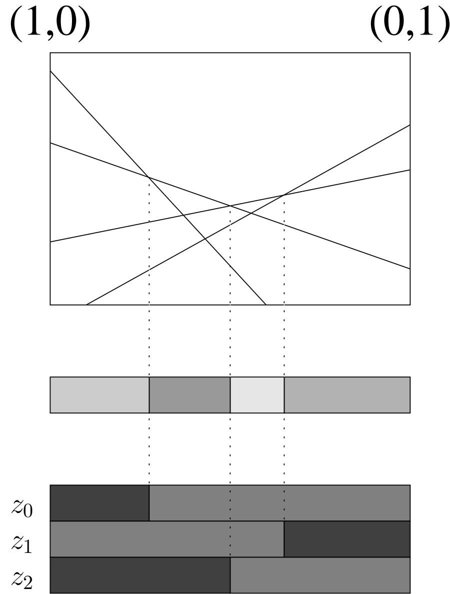


Figure 5.10: Partitions for the belief space as intersections in lines that form the value function.

from any initial belief state is apparent from the figure. The problem now is to find the best value of a belief state b given that the first action is fixed to be a_0 regardless of which observation is seen.

Since each of the partition regions actually corresponds to a line in the S function, the value of the belief point b can easily be found. This is the horizon two value, but the immediate reward from doing action a_0 must also be added to the value of the functions $S(a_0, z_0)$, $S(a_0, z_1)$, and $S(a_0, z_2)$ at belief point b. If the action is fixed to be a_0 and the future strategy is fixed to be the same as it is at point b, then the value of every single belief point for that particular strategy can be found by simply summing all of the appropriate line segments. The line segment for the immediate rewards of the a_0 action and the line segments from the S functions for each observation's future strategy are used. This results in a single line over all belief space representing the value of adopting the strategy of doing a_0 and the future strategy of ($z_0 : a_0$, $z_1 : a_1$, $z_2 : a_0$). The notation for the future strategies just indicates an action for each possible observation.

This particular future strategy was derived from the belief point b, and it is the best future strategy for that belief point. However, just because the value of this future strategy can be computed for each belief point doesn't mean it is the best strategy for *all* belief points. A line can be drawn through the partition diagram at an arbitrary belief point b' to show the best future strategy for that belief point.

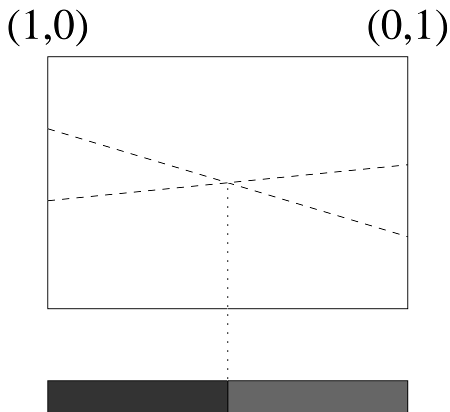


Figure 5.11: Partitions for the belief space given initial belief state b and action a_1.

Since there are three observations and two actions, there are a total of $2^3 = 8$ different future strategies. However, some of those strategies are not the best strategy for any belief points. Given the partitioning figures, all the useful future strategies are easy to pick out. For the example, there are only four useful future strategies. Figure 5.10 shows these four strategies and the regions of belief space where each is the best future strategy.

Each region in Figure 5.10 corresponds to a different line segment in the value function for the action a_0 and horizon length two. Each of these line segments is constructed by adding the immediate reward line segment to the line segments for each future strategy. The line segments for each of the four future strategies are shown at the top of Figure 5.10.

Note that each one of these line segments represents a particular two action strategy. The first action is a_0 for all of these segments and the second action depends upon the observation. If there were only the action a_0 in the model, then the value function shown in the previous figure would be the horizon two value function. However, because there is another action, its value must be compared with the value of action a_1 to find the true horizon two value function.

The whole process can be repeated for action a_1 to find the value function for that action. Figure 5.11 shows the value function and partition for action a_1. In this case there are only two useful future strategies. The value functions for each action can be drawn together to see where each action gives the highest value, as shown in Figure 5.12.

The left side of Figure 5.12 shows the combined value function for a_0 and a_1. This figure shows that there are two regions where action a_0 is preferred and two regions where a_1 is the best choice. Two of the line segments from the a_0 action value functions are not needed, since there are no belief points where it will yield a higher value than some other immediate action and future strategy. These two vectors are said to be *dominated* by the other vectors, while a vector that forms part of the upper PWLC surface is said to *dominate* over a part of the belief space. The simplified horizon two problem is shown on the right of Figure 5.12. Note that the new partitioning is made from intersections of the lines from the original a_0 and a_1 value functions. The partition shown below the value function in Figure 5.12 shows the best horizon two policy, indicating which action should be taken in each belief state.

The horizon three policy can be constructed from the horizon two policy in the same way as the horizon two policy was constructed from the horizon one policy. The general algorithm for one step of value iteration for solving POMDP problems is (Cassandra 1998a):

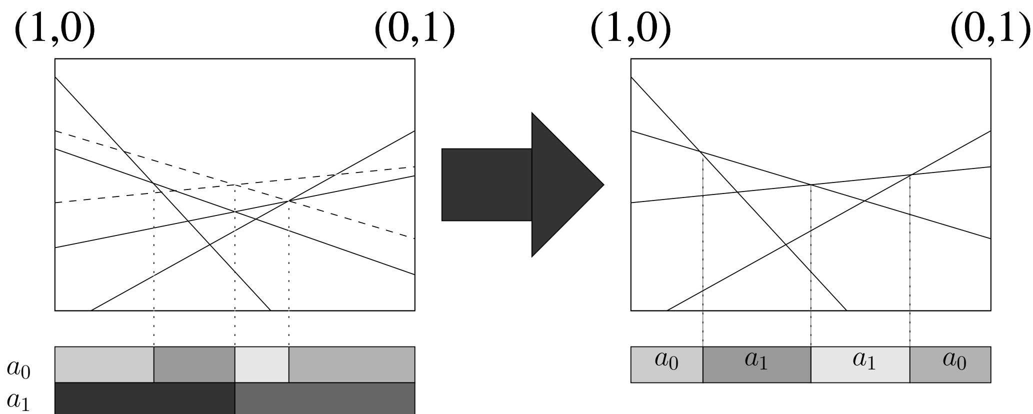


Figure 5.12: Combined a_0 and a_1 value functions.

1. Transform the current value function V for each action $a \in \mathcal{A}$ and each observation $z \in \mathcal{Z}$. This will result in $|\mathcal{A}| \times |\mathcal{Z}|$ sets of line segments (or vectors). These sets are often referred to as the S_z^a sets or the $S(a, z)$ functions.

2. Create the strategy for each action by adding the immediate rewards and the $S(a, z)$ functions for each of the useful strategies. The partition that this value function will impose is easy to construct by simply looking at the partitions of the $S(a, z)$ functions. This will result in $|\mathcal{A}|$ sets of vectors, each set represents the optimal strategy for one action. These are usually referred to as the S_a sets or $S(a)$ functions.

3. Merge the S_a sets together and see which line segments can be discarded. A line segment can be discarded if it does not form part of the upper surface. Such a line segment is said to be dominated by the other line segments. The resulting set is $\mathcal{V}'$, the new policy set.

The concepts and procedures can be applied to any horizon length. The following sections provide an overview of several algorithms for solving POMDP problems. On a computer with infinite precision, they would find the value function exactly for any horizon length. All of the exact solution methods are based on some variation of this value iteration procedure.

5.3 Exact Solution Methods

Several algorithms can compute the optimal policy for a POMDP, but these methods are very computationally expensive. All of the current exact solution methods find the optimal value function by exhaustive search.

For all of the examples, let $\mathcal{V}'$ be the set of vectors that are desired for the next stage of value iteration and $\mathcal{V}$ be the set of vectors that were obtained from the previous stage. The stages of value iteration simply correspond to succeeding horizon lengths. Each vector in $\mathcal{V}'$ is constructed by combining the vectors in $\mathcal{V}$ and the immediate rewards. More precisely, each vector is constructed from the immediate rewards and the transformation (using the S functions) of $\mathcal{V}$. The next section presents a short overview of the algorithms. Chapter 6 covers the Incremental Pruning algorithm, which is currently the most efficient POMDP algorithm, in more depth.

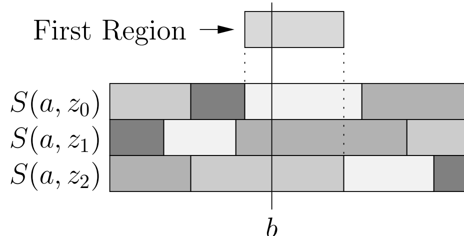


Figure 5.13: Sondik's first region.

5.3.1 Sondik's One-Pass Algorithm

Sondik (1971) proposed the first exact POMDP algorithm. This algorithm starts with an arbitrary belief point, constructs a vector for that point and then defines a set of constraints over the belief space where this vector is guaranteed to be dominant. Thus, it operates by defining a belief space region for the point.

The region defined, called the *Sondik region*, is actually the intersection of three easier to describe regions. When a vector is constructed from a belief point b, that vector represents a known strategy. This strategy is the best one for that belief point and some nearby belief points. However, it might not be the best strategy for all belief points. When moving away from b, there are two ways that this strategy might not be the best strategy for other belief points: either the immediate action doesn't change and the future strategy changes, or another action might become better. Any change is guaranteed to be discrete.

The first region that contributes to the Sondik region assures that even if the best action doesn't change, the future course of action doesn't change either. This region is derived from the $S(a, z)$ sets as shown in Figure 5.13. The endpoints of this region must be checked to see which vectors are dominant on either side.

Sondik's second region assures that the best vector for the current action is not exceeded by each of the current best vectors for each of the other actions. Even though some action a' and future course of action isn't as good as a at the point b, there could be a nearby point where that action and course of action becomes better.

It may be that a' and its current future course of action will never be better than the current best action a and its best future course of action. However, a' and a different course of action may become better than the current best action a and its course of action. The region must be restricted to incorporate this. Restriction is accomplished by defining the limits of each action's best region.

Figure 5.14 shows action a and the first region defined above. The dashed line is the best vector for this other action a' at belief point b. Note that there are belief states where a' is better than a, even though a is best at the belief point b. This second region assures that the region is restricted such that the best action does not change. It can be seen from the picture that the first region by itself would include points where the value of performing a is not the best attainable value.

The dashed line represents a particular future strategy for the non-optimal action. In other words, the dashed line represents the best strategy to do if that action must be taken first. The second region covers belief points where this strategy is not better than the strategy at belief point b.

Sondik's third region covers the case where not only does the best action change, but the best strategy for that other action changes from what is the current best strategy. The

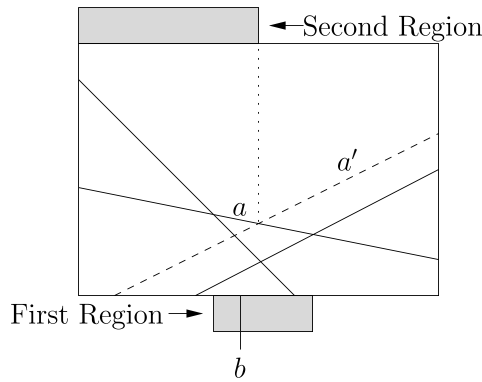


Figure 5.14: Sondik's second region.

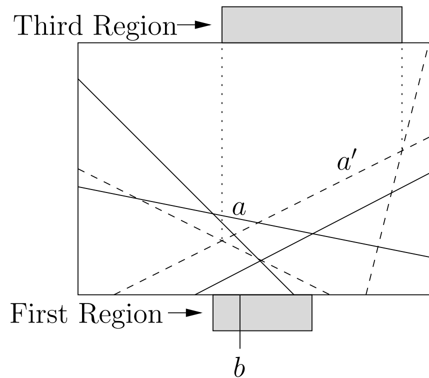


Figure 5.15: Sondik's third region.

dashed line in Figure 5.14 only shows part of the story for what is happening for the other action. The dashed line represents the best future strategy for that action at point b, but nothing is known about other future strategies for that action.

To ensure that the Sondik region covers only belief points where the action and future strategy doesn't change, the region where the dashed line is the best strategy for the other action is also included. This Figure 5.15 shows the three different strategies for the other action. The region where the original dashed line dominates the other dashed lines is the third region.

The intersection of the three regions is the region where action a is always the best one. Figure 5.16 shows the complete region that is imposed by Sondik's one-pass algorithm. However, note that the Sondik algorithm is too restrictive. Some points to the left of the Sondik region still have the a vector as the best one, but because of the third region, those are not included in the final region.

Linear programming can be used to find the maximum point on the edge of the Sondik region. That point also lies on the boundary of a region adjacent to the current region. In effect, it gives us a point in another region, making the job easier. The algorithm can start at the left edge and work across, discovering each Sondik region one at a time.

Unfortunately, the regions defined by Sondik's algorithm are extremely conservative. The Sondik region is not guaranteed to be the complete region in which the vector is dominant. Thus, it may generate the same vector for many belief points.

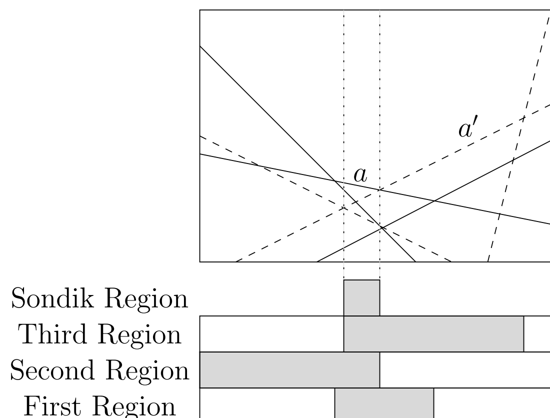


Figure 5.16: The complete Sondik region.

5.3.2 Monahan's Enumeration Algorithm

The task is to generate a finite set of points so that all of the vectors in $\mathcal{V}'$ can be constructed. The algorithm proposed by Monahan (1982) generally ignores trying to find these points. Instead, it simply generates all the possible vectors that can be constructed by combining the vectors in $\mathcal{V}$ and the immediate rewards. For each observation, a vector $v \in \mathcal{V}$ is selected, then all the vectors are transformed and combined to construct a potential vector for $\mathcal{V}$'.

This approach generates a large number of vectors. Many of the vectors are not useful, since they are completely dominated by other vectors over the entire belief space. The algorithm can eliminate the useless ones at the expense of some computing time, but enumerating the vectors takes a long time, even for small problems. It is possible to construct a problem where every one of those vectors enumerated is useful over some portion of belief space. Thus, in the worst case, the general problem of one step of value iteration on a POMDP is very hard, and this algorithm will perform as well as any other in the worst case.

5.3.3 Cheng's Linear Support Algorithm

The next algorithm, proposed by Cheng (1988), starts with Sondik's idea, but relaxes the constraints. Cheng's linear support algorithm does not focus on actions and future courses of actions. It simply picks a point, generates the vector for that point, and then checks the region of that vector to see if it is the correct one at all corners (vertices) of the region. If not, it adds the vector at that point and checks its region.

Cheng's linear support algorithm starts with an arbitrary point and adds the best vector at that point to $\mathcal{V}$. It is easy to find the best vector for this point, though the region where this vector is best is not known. The algorithm assumes that this vector represents $\mathcal{V}$ over the region that it dominates and attempts to either prove or disprove that assumption.

Given a point somewhere in the middle of the belief space, the task is to find the vector that is largest at that point. This vector may or may not be in $\mathcal{V}$. If it is not in $\mathcal{V}$, then the largest difference between this vector and $\mathcal{V}$ will occur at the endpoints, since $\mathcal{V}$ is PWLC. Therefore, it is only necessary to check the endpoints in order to determine whether or not the new vector should be added to $\mathcal{V}$. Figure 5.17 gives an example, starting with one vector in $\mathcal{V}$. The original vector is shown with dots at the endpoints that need to be

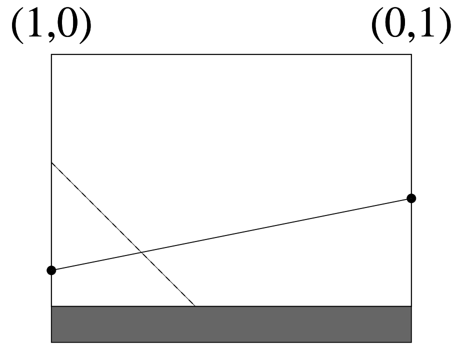


Figure 5.17: First vector found by Cheng's algorithm (solid line with dots at the endpoints) and second vector (dashed line).

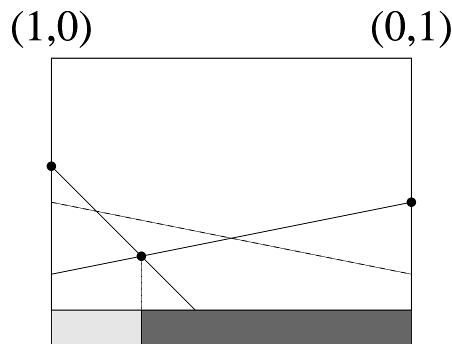


Figure 5.18: Second vector found by Cheng's algorithm.

checked. Checking the left endpoint yields a new vector that is better at that point.

The new vector is added to the current set, and the point where it intersects the original vector is added to the set of points that need to be checked. Looking for the best value at that point yields another vector. Figure 5.18 shows the new situation. Again, this vector is added to the estimate of $\mathcal{V}$ and the points where this new vector intersects the old ones are added to the set of points that must be checked. Looking for the best value at these new endpoints yields the same points, so they can be removed from the set of points that need to be checked.

The right endpoint of the original vector still has not been checked. Looking for the best value at that location yields another vector, shown in Figure 5.19. The algorithm continues in this manner until there are no points left to be checked. Since each vector's region has been checked and all the points have been processed, the current set of vectors is indeed guaranteed to be the value function, $\mathcal{V}$.

The two key points are:

1. if the value function is incorrect, the biggest difference will occur at a corner, and

2. if all possible region corners are generated, then nothing can be overlooked.

Linear support has been shown to be more efficient than Sondik's one-pass algorithm (Cheng 1988).

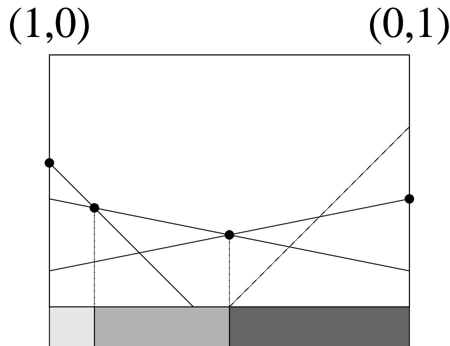


Figure 5.19: Third vector found by Cheng's algorithm.

5.3.4 The Witness Algorithm

The Witness algorithm (Littman 1994) defines regions in the belief space for a vector and looks for a point where that vector is not dominant. Such a point, if found, is a *witness* to the fact that the current estimate $\mathcal{V}$ of the value function is not optimal. The witness point is used to find another vector to add to the current set. The Witness algorithm can be viewed as a combination of Sondik's One-pass and Cheng's Linear support algorithms.

Like the previous algorithms, the Witness algorithm concentrates on finding the best value function for each of the actions separately, then combines them into the final $\mathcal{V}'$ value function for each iteration. Besides doing each action in isolation, it also finds the value function for only one observation at a time. A vector at a belief point is constructed by selecting a transformed vector from $\mathcal{V}$ for each observation. The Witness algorithm starts with an arbitrary belief point and generates its vector. It adds this vector to a set and assumes that this set is $\mathcal{V}'$; it then goes about trying to prove or disprove whether or not this set truly is $\mathcal{V}'$.

In constructing a vector, a choice is made for each observation of one of the $\mathcal{V}$ vectors representing a particular future strategy. The algorithm then looks at the individual choices made, observation by observation to see where a different choice would yield a better value. It defines the region where it is assured that the particular choice is best. If it finds a belief point where a different strategy would be better, then this serves as a witness to the fact that the current set of vectors is not yet the real $\mathcal{V}'$.

The regions defined by this algorithm are best described by the intersection of two simpler regions. One is the region over which the current vector dominates all the other vectors that have been generated up to this point. This region is shown in the top of Figure 5.20. Note that according to this figure, there are three vectors in the policy set $\mathcal{V}$. The first region simply reduces the search for a point to the region over the vector that dominates the part of belief space containing b.

The second region, also shown in Figure 5.20, defines the area over which the current future strategy will be best for the current observation. Outside of this region, a different course of action for this observation would result in a better value. If a belief point is found where another future strategy can be substituted, this new strategy cannot simply replace the existing one. Since this point is considered in isolation from the other observations, it could be that in addition to changing the choice for this observation, it is also necessary to change the choice for another observation. Finding a belief point where the current observation's choice could be changed just provides a witness to the fact that there is a

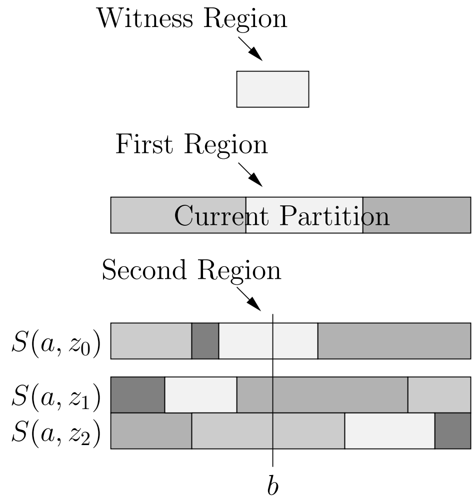


Figure 5.20: Regions defined in the Witness algorithm.

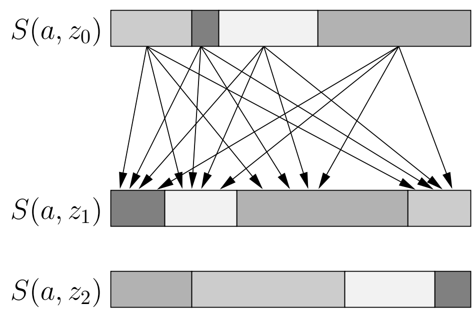


Figure 5.21: IP creates all combinations of vectors from the source sets.

point where the value function is better than the current $\mathcal{V}$ indicates. This point can be used to generate the real best vector for that point while taking into account all the observation choices.

This is repeated for every observation to get a set of vectors for a particular action and then all actions are combined. Finally, all dominated vectors are eliminated to get the set $\mathcal{V}'$ for the current iteration.

5.3.5 Incremental Pruning

Incremental Pruning (IP) (Zhang and Liu 1996) combines elements of Monahan's enumeration and Witness algorithms. Like Witness, it considers constructing sets of vectors for each action individually and then focuses on each observation, one at a time. The previous algorithms all tackled the problem indirectly by defining regions. Since the main problem is in finding all the different combinations of future strategies, IP attacks that problem directly.

For a given action, IP first constructs all of S_{az} sets. Monahan's algorithm would then try all ways of combining choices from each of these. IP does this incrementally, observation by observation. Figure 5.21 shows how IP creates a set of new vectors by combining all of the vectors in the sets $S(a, z_0)$ and $S(a, z_1)$. The vector that is dominant over the first region is combined with the vectors that are dominant in each of the four regions of $S(a, z_1)$,

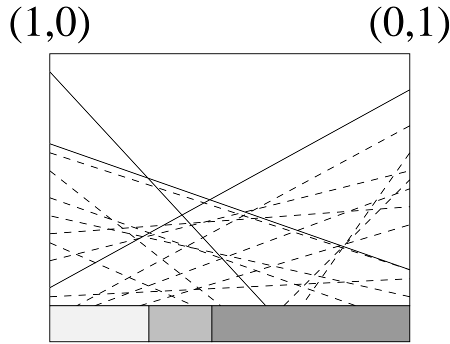


Figure 5.22: The set of vectors is checked to find out which ones are dominant.

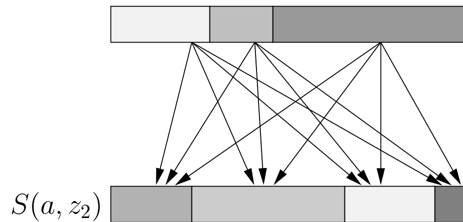


Figure 5.23: The resulting set is combined with the $S(a, z_2)$ set.

and so on, resulting in sixteen combinations. These combinations are represented by sixteen arrows in the figure.

Figure 5.22 shows the resulting set of vectors. All of the useless vectors are eliminated, leaving only the vectors that are dominant. In this case, there are only three vectors in the intermediate set. These vectors are then combined with the vectors in $S(a, z_2)$ as shown in Figure 5.23, and the resulting set is made parsimonious again. This procedure is performed for each observation. When all observations are done, the resulting intermediate set is the S_a set. This process is repeated for the other actions to create all of the S_a sets, which are then combined just as in the Witness algorithm.

5.3.6 Restricted Region

Incremental pruning uses the idea of filtering or *pruning* sets of vectors to reduce the set to only the vectors that dominate over some region. Several methods can be used for pruning. One approach, known as Restricted Region (RR) (Cassandra, Littman, and Zhang 1997), has been shown to be best in most cases. In IP, two sets of vectors are combined, and then the resulting set is purged. RR is simply a more efficient way to carry out the combination and purging operation. This method will be covered in detail in Chapter 6.

5.4 Approximate Solution Methods

All exact POMDP solution methods are very computationally expensive. It has been shown that no exact solution method can scale well with the size of the problem (Goldsmith and Mundhenk 1998). This has prompted some researchers to search for ways to find high quality approximate solutions in a reasonable amount of time. At the time of this writing,

approximate solutions are a relatively new area of research, and very few results have been presented.

One approximation technique under investigation is policy iteration with belief space quantization. The idea is to break up the belief space into bins and then lump all of the belief states within a bin into a single aggregate belief state. Hansen (1998) showed that this approach could produce high quality approximate solutions to POMDP problems, while requiring far less computation than finding an exact solution. He extended his approach with a heuristic search strategy to further reduce the amount of computation. Comparison to exact solutions show that there is a tradeoff between the quality of result that can be obtained and the size of the belief state bins. For efficient execution, it is desirable to make the bins as large as possible, but at some point, this begins to affect the quality of the resulting policy. So far, no research has been performed to characterize the effects of bin size for solving various problems.

Hauskrecht (1997) presented a method for computing bounds on the value function for control problems with infinite horizon. He used a grid-based linear interpolation method and compared it to a simple lower bound method using Sondik regions. By incrementally improving the bounds with value iteration, his method produced a good approximation to the true value function.

If partial observability is removed from a POMDP model, the result is a completely observable Markov decision process. Another approximate solution approach is to use dynamic programming or reinforcement learning on the completely observable Markov decision process and then map the belief state to a specific state in order to follow that policy. This approach avoids the problems associated with finding the policy for the POMDP problem directly. Cassandra, Kaelbling, and Kurien (1996) used value iteration to find a policy for the underlying MDP in a robot navigation task. They found that this approach performed well in a small office environment. However, they suggest that this approach may not be as good as finding an approximate solution to the true POMDP because it ignores the observation probabilities when creating the policy.

Chapter 6

Parallel Restricted Region POMDP Algorithm

Exact solution methods for POMDPs do not scale well. The largest problems that have been solved to date have only tens of states, less than ten actions, and on the order of ten to twenty observations. This is clearly not suitable for planning in a robot which may have dozens of actions and observations and operate in an environment with hundreds or even thousands of states. To address this issue, some researchers are investigating methods for finding approximate policies for large POMDP problems (Geffner 1998; Hansen 1998).

Although approximate solution methods may provide acceptable policies for large POMDP problems, there is a good reason to avoid such approaches at this stage in my research. Using approximations of the optimal policy would introduce another unknown into my research. With exact solution methods, I can be confident that any problems are not due to the quality of the policy generated by the POMDP planner. This allows me to concentrate on learning at the low level without having to worry about any possible errors introduced by suboptimal planning. Once I have verified that my ideas are sound in the case of exact POMDP planning, it may be possible to scale up by using approximate solutions.

Since approximate solutions could potentially affect performance, my goal was to find a method to find exact solutions to POMDP problems in a small but non-trivial robot domain. After investigating the state of the art in exact solution methods for POMDP problems, I implemented a POMDP solver using the Restricted Region (RR) algorithm, which is currently the best exact POMDP solution method (Cassandra, Littman, and Zhang 1997). After an initial performance evaluation, it became apparent that it was not feasible to find an exact solution for even the smallest problem that I envisioned, which had 64 states, 3 actions and 16 observations. This prompted me to develop a parallel algorithm for exact POMDP solutions. An overview of Restricted Region was given in Section 5.3.6. The following section will cover the standard form of the Witness and Incremental Pruning, and Restricted Region algorithms in depth. Section 6.2 will explain the parallel algorithm. The experiments and results are described in Section 6.3, and Section 6.4 gives some suggestions for future work.

6.1 Solving POMDPs Using Incremental Pruning

The goal for a POMDP solution method is to construct a policy π that specifies the action to take for every possible belief state $b \in \mathcal{B}$ so as to maximize discounted future rewards.

Most POMDP solution algorithms generate the policy by finding the value function, which specifies the value of taking each possible action from every belief state. The value function can always be represented as a piecewise linear and convex surface over the belief space (Smallwood and Sondik 1973; Monahan 1982). A POMDP policy is a set of labeled vectors that are the coefficients of the linear segments that make up the value function.

In value iteration algorithms, a new value function $\mathcal{V}'$ is calculated in terms of the current value function $\mathcal{V}$ at each iteration (Cassandra, Littman, and Zhang 1997). $\mathcal{V}'$ is the result of adding one more step of rewards to $\mathcal{V}$. In the case of an infinite horizon problem, $\mathcal{V}'$ represents an improvement in the approximation to the optimal value function. The function $\mathcal{V}'$ is defined by:

$$\mathcal{V}'(b) = \max_{a \in \mathcal{A}} \left(\sum_{s \in \mathcal{S}} r(s,a)b(s) + \gamma \sum_{z \in \mathcal{Z}} \Pr(z|b,a)\mathcal{V}(b_z^a) \right). \tag{6.1}$$

where $r^a(s)$ is the expected immediate reward for taking action a in state s, and $\Pr(z|b,a)$ is the probability of observing z given that action a is taken in belief state b. The term b_z^a is defined in Equation 5.2. Equation 6.1 defines the value for a belief state b as the value of the best action that can be taken from b where the best value is given by the expected immediate reward for that action plus the expected discounted value of the resulting belief state.

Equation 6.1 can be decomposed into simpler combinations of other value functions:

$$\mathcal{V}'(b) = \max_{a \in \mathcal{A}} \mathcal{V}^a(b) \tag{6.2}$$

$$\mathcal{V}^a(b) = \sum_{z \in \mathcal{Z}} \mathcal{V}_z^a(b) \tag{6.3}$$

$$\mathcal{V}_z^a(b) = \frac{\sum_{s \in \mathcal{S}} r(s,a)b(s)}{|\mathcal{Z}|} + \gamma \Pr(z|b,a)\mathcal{V}(b_z^a). \tag{6.4}$$

Breaking up the value function is the key step in deriving the incremental pruning algorithm (Cassandra, Littman, and Zhang 1997). The $\mathcal{V}'(b)$, $\mathcal{V}^a(b)$ and $\mathcal{V}_z^a(b)$ functions are piecewise linear and convex value functions. The $\mathcal{V}_z^a(b)$ functions are action and observation specific functions that are combined to form the $\mathcal{V}^a(b)$ functions. The $\mathcal{V}^a(b)$ functions, in turn, are action dependent value functions that are combined to form the new value function $\mathcal{V}'(b)$. All of the value functions are defined in terms of simple transformations of other value functions. Because these functions involve linear combinations, the transformations preserve piecewise linearity and convexity (Smallwood and Sondik 1973; Cassandra, Littman, and Kaelbling 1996).

Since the value functions $\mathcal{V}_z^a$, $\mathcal{V}^a$, and $\mathcal{V}'$ are PWL, they can be represented as sets of vectors composed of the coefficients for each PWL segment. If $\mathcal{V}$ can be expressed as a piecewise linear function and $\mathcal{V}(b) = \max_{\alpha \in S} b \cdot \alpha$ for some finite set of $|\mathcal{S}|$-vectors S, the functions can be expressed as $\mathcal{V}_z^a(b) = \max_{\alpha \in S_z^a} b \cdot \alpha$, $\mathcal{V}^a(b) = \max_{\alpha \in S^a} b \cdot \alpha$, and $V'(b) = \max_{\alpha \in S'} b \cdot \alpha$ for some finite set of $|\mathcal{S}|$-vectors S_z^a, S^a, and S' $\forall\, \alpha \in \mathcal{A}, z \in \mathcal{Z}$. The vector sets have a unique representation of minimum size (Cassandra, Littman, and Kaelbling 1996), and these minimum-sized sets are denoted by the symbols S_z^a, S^a, and S'.

The following review of vector and set operations will help with later explanations and algorithms. The lexicographical ordering of vectors is defined as $\alpha > \beta \iff \alpha(s) > \beta(s)\,\forall\, s \in \mathcal{S}$. Vector sums are also performed element by element such that $\theta_s = \alpha_s + \beta_s\,\forall\, s \in \mathcal{S}$ and vector dot products are defined as usual to be $\alpha \cdot \beta = \sum_{s \in \mathcal{S}} \alpha_s \beta_s$. The *cross sum* of

two sets of vectors is defined as $\mathcal{X} \oplus \mathcal{Y} = \{\chi + \psi | \chi \in \mathcal{X}, \psi \in \mathcal{Y}\}$ which extends to collections of vector sets as well. Set subtraction is defined by $\mathcal{X} \backslash \mathcal{Y} = \{\chi \in \mathcal{X} | \chi \notin \mathcal{Y}\}$. A vector of all ones is denoted as 1, a vector of all zeros is denoted as 0, and a vector e_s is defined for every $s \in \mathcal{S}$ where e_s is a vector of all zeros except for one element $e_s(s) = 1$. The notation $\mathsf{purge}(\mathcal{X})$ represents the operation of reducing a set $\mathcal{X}$ to its minimum size form.

With the vector set notation, the sets described earlier can be re-defined as

$$S' = \mathsf{purge}\left(\bigcup_{\alpha \in \mathcal{A}} S^a\right) \tag{6.5}$$

$$S^a = \mathsf{purge}\left(\bigoplus_{z \in \mathcal{Z}} S^a_z\right) \tag{6.6}$$

$$S^a_z = \mathsf{purge}\left(\{\tau(\alpha, a, z) | \alpha \in S\}\right) \tag{6.7}$$

where $\tau(\alpha, a, z)$ is the $|\mathcal{S}|$-vector given by

$$\tau(\alpha, a, z) = \frac{1}{|\mathcal{Z}|} r^a(s) + \gamma \sum_{s'} \alpha(s') \Pr(z|s', a) \Pr(s'|s, a). \tag{6.8}$$

Equations 6.5 and 6.6 follow directly from Equations 6.2 and 6.3 and basic properties of piecewise linear convex functions. Equation 6.7 is derived by substituting Equation 5.2 on page 64 into Equation 6.4. Calculating $\tau(\alpha, a, z)$, performing the cross sum, and performing the union are straightforward; so the problem is now reduced to purging sets of vectors. In order to get a solution quickly, the purge operation must be efficient.

6.1.1 Purging Vector Sets

Given a set of $|\mathcal{S}|$-vectors A and a single $|\mathcal{S}|$-vector α, the set of belief states in which vector α is dominant compared to all the other vectors in A is defined by:

$$R(\alpha, A) = \{b | b \geq 0, b \cdot 1 = 1, b \cdot \alpha > b \cdot \alpha', \alpha' \in A \backslash \{\alpha\}\}. \tag{6.9}$$

Note that according to this definition, α may be considered to be dominant over some belief states even if $\alpha \in A$. For any belief state b in this set,

$$\max_{\alpha' \in A \backslash \{\alpha\}} b \cdot \alpha' \neq \max_{\alpha' \in A \cup \{\alpha\}} b \cdot \alpha'.$$

The set $R(\alpha, A)$ is called the *witness region* of vector α because b can testify that α is needed to represent the piecewise linear convex function given by $A \cup \{\alpha\}$. Another way of stating this is

$$R(\alpha, A) = \emptyset \iff b \cdot \alpha <= b \cdot \alpha' \, \forall \, \alpha' \in A, b \geq 0, b \cdot 1 = 1, \tag{6.10}$$

which simply states that the witness region is empty if and only if there is no part of the belief space where α has a better value than any other vector in A.

Given this definition for the witness region R, the purge function is defined as

$$\mathsf{purge}(F, \mathcal{S}) = \{\alpha | \alpha \in A, R(\alpha, A) \neq \emptyset\}, \tag{6.11}$$

which simply returns the set of vectors in A that have non-empty witness regions. This is the minimum sized set that represents the piecewise linear convex value function.

purge $(F,\mathcal{S})$
    $W \leftarrow \emptyset$
    – *Select the obvious winners at each corner of the belief space and put them in W*
    for $s \in \mathcal{S}$
        $\max \leftarrow 0$
        for $\phi \in F$
            if $((e_s \cdot \phi > \max)$ or $((e_s \cdot \phi = \max)$ and $(\omega_s < \phi)))$ then
                $\omega_s \leftarrow \phi$
                $\max \leftarrow e_s \cdot \phi$
    for $s \in \mathcal{S}$
        if $(\omega_s \in F)$ then
            $W \leftarrow W \cup \{\omega_s\}$
            $F \leftarrow F \backslash \{\omega_s\}$
    – *Test the remaining vectors for domination*
    while $F \neq \emptyset$
        $x \leftarrow$ dominate(ϕ, W)
        if $(x = \perp)$ then $F \leftarrow F \backslash \{\phi\}$
        else
            $\max \leftarrow 0$
            for $(\phi \in F)$
                if $((x \cdot \phi > \max)$ or $((x \cdot \phi = \max)$ and $(\omega < \phi)))$ then
                    $\omega \leftarrow \phi$
                    $\max \leftarrow x \cdot \phi$
            $W \leftarrow W \cup \{\omega\}$
            $F \leftarrow F \backslash \{\omega\}$
    return W

Figure 6.1: White's algorithm for purging a set of vectors.

Figure 6.1 shows the algorithm of purge$(F, \mathcal{S})$. This operation, developed by White III (1991), returns the vectors in F that do not have empty witness regions as defined in Equation 6.10. This operation filters, or *purges* out, the unnecessary vectors. Dot product ties *must* be broken lexicographically in order for the algorithm to work properly (Cassandra, Littman, and Kaelbling 1996). The purge algorithm starts with an empty set, W, and fills it with vectors from F that have non-empty witness regions. These vectors form the upper surface of the mass of vectors in W. As such, they are components of the piecewise linear convex value function.

The algorithm then checks each vector in F to determine if it is dominant somewhere in W. If a point x is found on vector α that is above all of the lines in W, the vector α cannot simply be added to the set W, because α may be dominated by yet another vector in F at x. Figure 6.2 may help to explain what is going on here. The two solid lines are the vectors already in W, and all of the dashed lines are vectors in F. Solving for the non-dominated point on α with the highest value results in a point x that is a witness to the fact that another vector must be added to W, but does not indicate which vector should be added. To find the correct vector, the algorithm chooses the vector $v \in F$ that has the maximum dot product with x. In other words, the vector with the highest value at belief state x is the one that belongs in the set W, and it may or may not be the vector α that was used

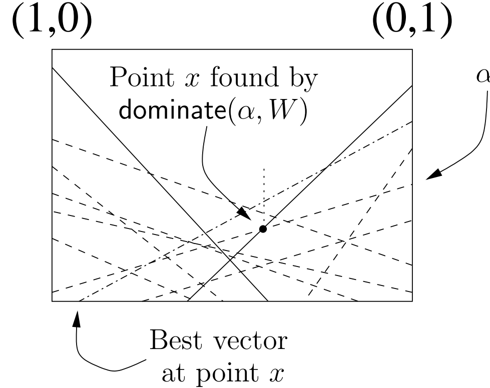


Figure 6.2: The two solid vectors are in the set W. All other vectors will be tested one at a time. For each vector α, the dominate function finds the largest point x on α that is not dominated by another vector in the set W, but this does not mean that α should be added to W. The point x is only a *witness* to the fact that W does not contain the best vector at that point.

to find the witness point x. The winning vector is shown in Figure 6.2 with the dot-dash lines. The winning vector is added to W and removed from F, and the loop repeats. When F is empty, W contains the parsimonious set of dominant vectors.

The dominate function which is called by purge finds the belief state x that gives the largest value on the line segment corresponding to α that is not dominated by another vector in W. If α is dominated everywhere, then dominate returns $\perp$. Figure 6.4 is a two state representation of the point. This problem can be expressed as finding the maximum point on the line represented by α with the constraint that the point found must be above all of the other lines in the set W. This turns out to be the definition of a linear program (LP) (Calvert and Voxman 1989). Figure 6.3 gives the algorithm for the dominate function. It creates a linear program and adds a constraint for each line in W. Two additional constraints make sure that the elements in x sum to one and that the answer is not negative. With most linear program solvers, this final constraint is not necessary. Once the linear program is set up, the solver is called. If no solution exists, or if the value of the objective function is non-positive, then a null vector is returned. Otherwise, the solution to the linear program is the desired point.

Given the purge algorithm, it is now straightforward to perform dynamic programming using Equations 6.5, 6.6, and 6.7. The straightforward application of Equation 6.7 amounts to an exhaustive enumeration of all possible vectors for each action and observation, followed by a purge operation. The resulting S_z^a sets are then cross summed and purged using Equation 6.6. The resulting S^a sets are merged and purged by Equation 6.5, resulting in S', the value function at the current horizon. This approach, known as the Witness Algorithm (Littman 1994), is very similar to Sondik's one-pass algorithm. The major problem with the Witness algorithm is the time it takes to perform the purge operation. Most of the run-time is spent solving linear programs. For this reason, it is desirable to minimize the number of linear programs that have to be solved and, more importantly, minimize the number of constraints in the linear programs themselves. For the Witness algorithm, in the best case, there are $1/2(\sum_z |S_z^a| - |\mathcal{Z}| + 1)(|S^a| + 1)(|S^a| + 2) - \sum_z |S_z^a| + |\mathcal{Z}| - 3$ total constraints, and in the worst case, there are $|S^a|(|S^a| + 1)(\sum_z |S_z^a| - |\mathcal{Z}| - 1/2)$

```
dominate (α,A)
    L ← LinearProgram(objective:max δ)
    for (α' ∈ A\{α})
        AddConstraint(L,x · α ≥ δ + x · α')
    AddConstraint(L,x · 1 = 1)
    AddConstraint(L,x ≥ 0)
    if (Infeasible(L)
        then return ⊥
        else
            (x, δ) ← SolveLP(L)
            if (δ > 0)
                then return x
    return ⊥
```

Figure 6.3: Linear programming test for domination.

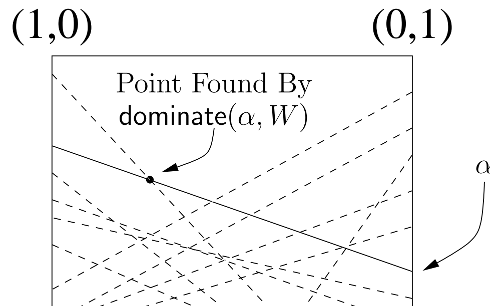


Figure 6.4: The dominate function finds the largest value on α that is not dominated by another vector in the set W.

```
incprune (S^a_{z_0} ... S^a_{z_{|Z|-1}})
    W ← purge(S^a_{z_0}, S^a_{z_1})
    for (i ← 2 to |Z| − 1)
        W ← purge(W ⊕ S^a_{z_i})
    return W
```

Figure 6.5: Incremental pruning method for combining the S^a_z sets.

total constraints (Cassandra, Littman, and Zhang 1997).

6.1.2 Incremental Pruning

Incremental pruning (Cassandra, Littman, and Zhang 1997) sequences the purge operations to reduce the number of linear programs that have to be solved and to reduce the number of constraints in the linear programs themselves. Incremental pruning can exhibit superior performance and greatly reduced complexity compared to the Witness algorithm (Zhang and Liu 1996; Cassandra, Littman, and Zhang 1997).

From the associative property of addition, $(A \oplus B) \oplus C = A \oplus (B \oplus C)$, and it also holds that $\mathsf{purge}((A \oplus B) \oplus C) = \mathsf{purge}(\mathsf{purge}(A \oplus B) \oplus C)$. Therefore, Equation 6.6 can be rewritten as

$$S^a = \mathsf{purge}(\ldots \mathsf{purge}(\mathsf{purge}(S^a_{z_0} \oplus S^a_{z_1}) \oplus S^a_{z_2}) \ldots \oplus S^a_{z_{|\mathcal{Z}|-1}}), \tag{6.12}$$

which is the approach taken by the incremental pruning algorithm. The algorithm is given in Figure 6.5. In addition to being conceptually simpler than the Witness algorithm, incremental pruning can also exhibit superior performance and greatly reduced complexity because it reduces the size of the linear programs that must be solved.

Let $A' = \mathsf{purge}(A)$ and $B' = \mathsf{purge}(B)$ so that both A' and B' are parsimonious, and then let $W = \mathsf{purge}(A' \oplus B')$, then

$$|W| \geq max(|A'|, |B'|). \tag{6.13}$$

This comes from the observation that for every $\omega \in W$, every region $R(\omega, W)$ is contained within $R(\alpha, A), \alpha \in A$ and $R(\beta, B), \beta \in B$. This means that the size of W is monotonically non-decreasing during the execution of incprune. Also, the size of W never grows explosively compared to its final size, as happens in the naive Witness algorithm. Thus, vectors are added and pruned away from W incrementally, reducing the average number of constraints in the linear programs. The worst case bounds for IP are identical to Witness, but the best case total number of constraints for IP is much better than for Witness. In practice, IP has been found to converge faster than Witness in most cases (Cassandra, Littman, and Zhang 1997).

6.1.3 Restricted Region

Restricted region is a special case of incremental pruning (Cassandra, Littman, and Zhang 1997). All of the calls to purge in incprune are of the form purge $(A \oplus B)$. Thus, the implementation of purge can be modified to take advantage of the fact that there is a great

deal of regularity in the vector set passed to it. The call to dominate in Figure 6.1 can be replaced with

$$x \leftarrow \mathsf{dominate}(\phi, D \backslash \{\phi\}) \tag{6.14}$$

where D is a set of vectors that satisfy the following set of properties:

1. $D \subseteq (A \oplus B)$
2. Let $\alpha + \beta = \phi$ for $\alpha \in A$ and $\beta \in B$. For every $\alpha' \in A$ and $\beta' \in B$, if $\alpha' + \beta' \in W$, then either $\alpha' + \beta' \in D$, or $\alpha + \beta' \in D$, or $\alpha' + \beta \in D$.

Many choices for D will satisfy the given properties (Cassandra, Littman, and Zhang 1997). Some choices are:

$$D = A \oplus B \tag{6.15}$$
$$D = (\{\alpha\} \oplus B) \cup (\{\beta\} \oplus A) \tag{6.16}$$
$$D = W \tag{6.17}$$
$$D = (\{\alpha\} \oplus B) \cup \{\alpha' + \beta | (\alpha' + \beta) \in W\} \tag{6.18}$$
$$D = (\{\beta\} \oplus A) \cup \{\beta' + \alpha | (\beta' + \alpha) \in W\} \tag{6.19}$$

Any such choice of D allows us to use the domination check of Equation 6.14 to either remove ϕ from consideration or to find a vector that has not yet been added to W.

Incremental pruning is actually a *family* of algorithms. The particular choice for D determines how efficient the algorithm is. In general, small sets make a better choice, since this reduces the number of constraints in the linear program created by the dominate function. Equation 6.15 is equivalent to Monahan's algorithm (Cassandra, Littman, and Zhang 1997). Equation 6.17 is equivalent to Lark's algorithm. Incremental pruning using a combination of Equations 6.18 and 6.19 are referred to as restricted region(RR) and is currently deemed to be the best POMDP solution algorithm (Cassandra, Littman, and Zhang 1997).

6.2 Parallel Restricted Region

I implemented the Restricted Region algorithm for the planner in my architecture and found that it required several days of computation to find a policy for my simple robot domain, which had 64 states, 3 actions, and 16 observations. Since I needed to generate several policies, it was worthwhile to spend some effort in improving the POMDP solver. This section presents the method that I developed to speed up the Restricted Region algorithm through parallelism.

Equations 6.5, 6.6 and 6.7 show how the POMDP problem can be decomposed into a set of smaller problems. Most of the computation time is spent in Equation 6.6, where the S^a_z sets are merged into the S^a sets. Since IP decomposes the problem into independent parts, the easiest way to parallelize RR would be to perform the cross sum and filter operations on the S^a_z sets in parallel. The number of threads using this approach would be limited to $|\mathcal{A}|$.

Another approach is to group the terms differently and re-write Equation 6.12 as

$$S^a = \mathsf{purge}(\ldots \mathsf{purge}(\mathsf{purge}(S^a_{z_0} \oplus S^a_{z_1}) \oplus \mathsf{purge}(S^a_{z_2} \oplus S^a_{z_3}))\ldots). \tag{6.20}$$

This formulation breaks the problem into more independent parts. For each $a \in \mathcal{A}$, $|\mathcal{Z}|/2$ threads can be started. As those threads complete, the results can be merged in parallel, and so on, until one final process merges the last two sets together. This technique is only useful for problems with $|\mathcal{Z}| \geq 4$, but would provide significant performance improvement on those problems. This approach would allow the processing to be distributed over $|\mathcal{A}||\mathcal{Z}|/2$ processors, which would work well on a symmetric multiprocessing system. However, it still does not offer a great deal of parallelism, and for this reason, I chose a different approach.

My approach is to perform the individual domination tests in parallel. Each domination test is performed by a domination server which may reside on another computer. The sequential algorithm shown in Figure 6.1 assumes that W and F will be updated at the end of each domination check and that W and F do not change during the domination check. However, this cannot be guaranteed when several domination tests occur simultaneously, and synchronizing the data between servers would be a significant problem. I changed the algorithm slightly, so that the vectors are not moved from F to W, but are simply marked as dominant. After all of the vectors in F have been checked, the ones marked as dominant are moved to W. This approach allows the initial sets to be sent to all of the servers at the beginning of each parallel merge which minimizes subsequent communication, but does double the average size of the linear programs that must be solved.

Equations 6.18 and 6.19 are the key not only to restricted region, but also to parallelizing incremental pruning. The best speedup can be achieved by distributing the linear programs over as many processors as possible and solving as many in parallel as possible. In order to do so, I must modify the `purge` function and make some other changes to the algorithm. I modified the cross sum operator $\alpha \oplus \beta$ so that the resulting vector is "tagged" to indicate α and β. I then added an operation `tagged`$\alpha(\alpha, X)$ to return the set of vectors in X that are tagged with the given α and an operation `tagged`$\beta(\beta, X)$ to return the set of vectors in X that are tagged with the given β. The complete F and W sets are sent to the servers, which use these operations to extract the subsets required for each application of Equations 6.18 and 6.19. I then modified the `purge` algorithm to perform domination checks in parallel.

The parallel purge algorithm is shown in Figure 6.6. As with RR, the algorithm first finds the dominant vectors at each corner of the belief space and moves those vectors into W. Then it starts a number of threads to perform the domination checks on the remaining vectors in F. The threads all execute the same `ThreadProc` code (shown in Figure 6.7) locally. Each thread is responsible for communicating with one server. It sends F to the remote server and then enters a loop. On each iteration, the thread takes the next available vector in F and instructs the remote server to test that vector for domination. If the remote server returns a witness point, then the thread finds the largest vector in F at that point and marks it for addition to W. When all of the vectors in F have been tested, the threads terminate, and the main program continues.

Figure 6.8 shows the server algorithm for the parallel purge algorithm. The server understands two commands. The `NewProblem` command instructs it to receive the following F and W sets. The `TestDomination` command instructs the server to test a given vector in F for domination. When a `TestDomination` command arrives, it extracts the α and β tags from the vector and extracts the sets $A = \{\alpha\} \oplus B$, $B = \{\beta\} \oplus A$, $B^W = \{\alpha' + \beta | (\alpha' + \beta) \in W\}$, and $A^W = \{\beta' + \alpha | (\beta' + \alpha) \in W\}$. D is calculated as $A \cup B^W$ or $B \cup A^W$, whichever is smaller. Finally, ϕ is tested for domination over D, and the result is returned.

```
purge (F,S)
    W ← ∅
    – Select the obvious winners at each edge of the belief space and put th em in W
    for s ∈ S
        max ← 0
        for φ ∈ F
            if ((e_s · φ > max) or ((e_s · φ = max) and (ω_s < φ))) then
                ω_s ← φ
                max ← e_s · φ
    for s ∈ S
        if (ω_s ∈ F) then
            W ← W ∪ {ω_s}
            F ← F\{ω_s}
    – Test the remaining vectors for domination
    current ← 0
    for i ∈ NUMSERVERS
        child_i ← CreateThread(ThreadProc, F, W, current, i)
    for i ∈ NUMSERVERS
        join(child_i)
    for (ω ∈ F)
        if (marked(ω)) then
            W ← W ∪ ω
    return W
```

Figure 6.6: Parallel algorithm for purging a set of vectors. The algorithm splits into NUMSERVERS threads, which mark the vectors in F that belong in W.

6.3 Experiments and Results on Parallel Restricted Region

I implemented the parallel RR algorithm in C++ and ran it on a network of Sun Ultra I workstations. For comparison, I also implemented a serial version of the RR algorithm derived from the same code base. Thus, the only difference in the two versions is in the incprune algorithm as outlined in this chapter.

I selected four problems from the literature and one of my own. The problems from the literature are representative of the problems that can be solved with current POMDP solution methods. The Tiger problem involves a three way choice between going through one of two doorways or listening for the tiger, which is behind one of the doorways. The Shuttle problem is a simple model of a space shuttle that must move back and forth between two space stations. The Painting problem involves selecting actions for painting parts and rejecting defective parts. Robot is a small robot navigation problem where the goal is to reach a specified location. My Robot problem is significantly more difficult than the other four, as evidenced by the fact that it took considerably more time to find a policy, and the advantage of my parallel algorithm is more pronounced on this problem. The problems are listed in Table 6.1.

```
ThreadProc(F, W, current, i)
    SendToServer(i, "NewProblem")
    SendToServer(i, F, W)
    while current < |F|
        lock current
        φ ← F_current
        current ← current + 1
        unlock current
        SendToServer(i, "TestDomination")
        SendToServer(i, φ)
        x ← RecieveFromServer(i)
        if (x ≠ ⊥) then
            max ← 0
            for (φ ∈ F)
                if ((x · φ > max) or ((x · φ = max) and (ω < φ))) then
                    ω ← φ
                    max ← x · φ
            lock F
            F_ω ← mark(F_ω)
            unlock F
```

Figure 6.7: Each thread communicates with one server. The thread sends F and W to the server, then begins processing vectors in F.

```
Server()
    loop
        ReceiveFromThread(command)
        if (command = "NewProblem") then
            ReceiveFromThread(F, W)
        elseif (command = "TestDomination") then
            ReceiveFromThread(φ)
            α = taga(φ)
            A = taggedα(α, F)
            A^W = taggedα(α, W)
            β = tagb(φ)
            B = taggedβ(β, F)
            B^W = taggedβ(β, W)
            if (|A| + |B^W| < |B| + |A^W|) then
                D = A ∪ B^W
            else
                D = B ∪ A^W
            x = dominate(φ, D\{φ})
            SendToThread(x)
```

Figure 6.8: Server algorithm for parallel purge.

Table 6.1: Test problem parameters.

Problem	$\lvert\mathcal{S}\rvert$	$\lvert\mathcal{A}\rvert$	$\lvert\mathcal{Z}\rvert$	Iterations	Reference
Tiger	2	3	2	273	Cassandra, Kaelbling, and Littman (1994)
Shuttle	8	3	5	283	Chrisman (1991)
Painting	4	4	2	100	Kushmerick, Hanks, and Weld (1995)
4×3	11	4	6	100	Parr and Russell (1995)
Robot	64	3	16	25	Pyeatt and Howe (1999)

Table 6.2: Results of experiments with run times given in seconds. The column headings for the Parallel RR algorithm indicate the number of servers.

Problem	RR	Parallel RR				
		1	2	3	4	5
Tiger	209	220	177	139	103	70
Shuttle	415	566	473	453	448	444
Painting	137	156	103	89	81	76
4×3	1359	1474	1156	1100	1079	1083
Robot	>86400	>86400	>86400	83465	62034	41648

Table 6.2 shows the performance of the two algorithms on the selected problems. The parallel IP algorithm was run with a varying number of servers between one and five; the algorithm was terminated when run time exceeded 24 hours. This occurred when fewer than three processors were used on the Robot problem. On the Tiger and Painting problems, each additional processor results in successively smaller performance increases. On the 4×3 problem, the performance actually degraded when a fifth processor was added. This diminishing return results from the communications overhead increasing in relationship to the amount of computation that is performed. On the Shuttle problem, the serial algorithm outperformed the parallel algorithm due to the time required to send the S_z^a sets over the network. For this problem, the S_z^a sets are relatively small and the time required to perform the purge algorithm on a single, local processor is less than the time required to send the sets over the network. Thus, the network overhead is greater than the benefit of additional processing power. On the Robot problem, which requires considerably more computation, the relationship of run-time to number of processors is more linear. Note that this relationship does not appear to correlate with the number of states, actions, or observations in the POMDP problem, but appears to be related to the size of the S_z^a sets that are created. Thus, $|\mathcal{S}|$, $|\mathcal{A}|$, and $|\mathcal{Z}|$ are not good indicators of the difficulty of POMDP problems.

6.4 Analysis of Parallel Restricted Region

My current implementation is a distributed algorithm that spreads the work across several computers connected with 10Mbit/s Ethernet. The major drawback to the current implementation is the network bottleneck created when all of the threads send the F set to the servers at once. This bottleneck could be alleviated in several ways. The main program

could broadcast the F set to all servers with a single transmission. This approach was not taken because TCP streams do not support broadcast. UDP supports broadcast, but implements unreliable communication using datagrams instead of streams. Broadcasting large vector sets using UDP would require breaking the data into packets and providing some mechanism to ensure reliability. This approach, though feasible, would require an added layer of complexity. Another way to reduce the bottleneck is to use multiple high speed network interfaces instead of a single Ethernet. Linux Beowulf and IBM SP systems both provide multiple high speed networks and would be ideal for this application. Third, and most obviously, the parallel IP algorithm could be implemented on a shared memory multiprocessor system.

This algorithm only parallelizes Equation 6.3, but could be extended to perform the incprune in Equation 6.2 and the cross sum operation in Equation 6.4 in parallel as well. Currently, these operations account for the majority of the time used by the parallel algorithm on the Tiger and Robot problems.

The parallel algorithm is most useful on problems with large S_z^a sets, where the overhead involved in sending the data to the servers is much less than the time required for computation in the incprune operation. Thus, for small problems, parallel solution does not present an advantage, while the most difficult POMDP problems can benefit greatly from parallelization.

The major computational bottleneck is in the solution of a large number of linear programs. Efficiency is a major goal in implementing a linear program solver for use in POMDP algorithms. Additionally, in order for the POMDP algorithms to work at all, the linear program solver must have a very high degree of numerical stability. The problem is exacerbated by the large number of very similar vectors that are generated when solving a POMDP. The vectors can be so similar that 64 bits of machine precision is not enough to represent the problem. I put a great deal of effort into creating a linear program solver with sufficient numerical stability and speed. My linear program solver is a sparse vector implementation based on the revised simplex method (Calvert and Voxman 1989). Simplex is the most common algorithm, but exhibits numerical instability and runs rather slowly. Newer LP algorithms based on interior point methods may be faster and provide better numerical stability. This will allow exact solution methods to scale to slightly larger problems. Although it is very likely that approximate solution methods are the only way to solve very large POMDP problems, the quality of the results obtained from approximate methods must be measured in some way. Scaling exact solution methods to solve non-trivial problems is valuable because it provides the standard against which to measure the quality of approximate solution methods.

The computational complexity of POMDP algorithms is typically expressed in terms of the number of linear programs they solve and the size of those linear programs. This metric is chosen because all of the exact solution algorithms use linear programming as a fundamental operation and the solution of linear programs is by far the most time consuming part of the algorithms. Cassandra, Littman, and Zhang (1997) gives the computational complexity of incremental pruning in terms of the number of linear programs that must be solved. Cassandra, Littman, and Kaelbling (1996) and Goldsmith and Mundhenk (1998) present more general discussions on the complexity of POMDP problems.

Chapter 7

Combining POMDP Planning with RL Actions

This chapter describes the POMDP/RL two level robot control architecture. The lower level controls the actuators that move the robot around and provides a set of actions that can be used by the higher level planning system. The higher level plans by finding a policy that specifies what action to take in every possible situation in order to move the robot from its current location to the goal. In the robot control architecture, actions in the bottom level are based on reinforcement learning (RL), while the top level is based on a partially observable Markov decision process (POMDP). The *combination* of RL Learning and POMDP planning and the ability for the system to learn new actions distinguishes this architecture from previous work. Chapter 4 covered the reinforcement learning method used for learning actions in the lower layer and Chapter 6 described the POMDP planner in the upper layer. This chapter describes the interconnection, coordination and synergy of the two layers. Section 7.1 describes the benefits of combining RL and POMDP for robot navigation and Section 7.2 describes the architecture.

Two layer control systems are common in robot architectures. Two layer control is a pragmatic solution that supports different chronological and spatial granularities of activity. The lower layer provides fast, short horizon decisions about quickly executed actions based on sensory data, while the upper layer adopts a more goal oriented view and plans over a longer scope using information abstracted from the sensory data. The lower layer is designed to allow the robot to avoid obstacles and make minor modifications to otherwise good courses of action based on sensory feedback. The upper layer ensures that the robot continually works toward its target task or goal.

The two layer control architecture offers reliability, which may be provided through adaptation, reactive control, and other mechanisms. Autonomous robots must be able to deal with failure of sensors and actuators or with changes in the environment when they are operating out of direct human control (e.g., space exploration, hazardous environments). In these situations, hardware failure can result in failure of the mission unless the control system is sufficiently adaptive. In less critical situations, reliable operation is also necessary. Even an office robot should be capable of dealing with some actuator and sensor failures or miscalibrations.

My thesis is that combining POMDP planning with reinforcement learning results in a system that is greater than the sum of its parts; namely, it has the ability to learn new actions as needed and adapt to sensor and effector failure at both levels. This ability stems

directly from two characteristics of the POMDP framework:

- In the POMDP model, actions have probabilistic outcomes. This, in part, enables the *discovery* of new actions that lie outside of the current set of actions $\mathcal{A}$. Section 7.2.4 explains how this is accomplished.
- There is a discrete-state MDP underlying the POMDP model. A discrete state transition can be used to provide reinforcement to the action that caused it to occur. Section 7.2.3 describes this mechanism in more detail.

Singh(Singh 1991) used pre-programmed low level actions with high level reinforcement learning to combine them and perform useful functions. This work is similar in spirit to the POMDP/RL approach, but does not allow for the *creation* of new actions. All actions had to be explicitly defined and hand coded, before learning could occur. Thus, the agent was not able to adapt to arbitrary changes in its environment. The POMDP/RL approach allows the robot to have a great deal more control over what it learns at the low level.

Singh (1992b) presented a different system that used reinforcement learning for low level actions. The previous system used RL at the high level and fixed actions. His new approach used RL for low level tasks as well. This approach provided a reward function for each elemental task, and specified exactly what tasks would be learned *a priori*. My work goes further by providing a more general framework for learning low level behaviors without requiring a reward function to be specified for each action or what actions will be learned. Instead, the POMDP/RL approach gives the responsibility for assigning rewards to the POMDP planner.

Successful application of POMDPs to robot control is very challenging; there is a wide gap between the current state-of-the-art in POMDP research and what is required for a successful autonomous robotic application (Cassandra 1998b). Deriving a policy for robots using POMDP models seems to be best done using models that are at a higher level of abstraction than what the robot actuators and sensors provide (Simmons and Koenig 1995; Cassandra, Kaelbling, and Kurien 1996; Nourbakhsh, Powers, and Birchfield Summer 1995), and there is some reason to believe that reinforcement learning works best at the level of sensors and actuators (Bagnell, Doty, and Arroyo 1998; Borisyuk, Wickens, and Köetter 1994; Kontoravdis, Likas, and Stafylopatis 1992; Mahadevan and Connell 1992; Millán and Torras 1992; Singh, Barto, Grupen, and Connoly 1994). POMDP planning can provide a reliable, reactive planning mechanism that works well with limited sensor information. Reinforcement learning provides a way to implement low level actions that are adaptive and work well in a dynamic environment.

7.1 What is Gained by Integration of POMDP and RL

Large changes in actuators or sensors can cause the reinforcement learning modules to fail completely. However, when enough negative rewards have accumulated, the RL modules begin searching for a better solution and learn how to use the damaged sensors and actuators, if possible. Since the only goal for the low level actions is to maximize reward, they are free to search for a better policy. In many instances, the existing policy requires only slight modification after sensor or actuator failure.

For robot navigation tasks, the states in the POMDP model correspond to positions in a topological map of the environment. Some research has shown how such maps could be learned by the POMDP planner (Mataric 1992b). Although the possibility for learning

topological maps is not precluded, the focus of the current research is on learning low level actions; so a map is provided to the robot.

The POMDP planner at the high level is given a map of the environment, but infers the relationship between the actions and the positions in the state space. This means that it can also adapt to changes in the environment and to sensor and actuator failures by adjusting the transition probabilities that it uses in planning which action to use. Thus, reliability is increased in the overall system through adaption at *both* levels. One of the main goals of this research is show how new actions can be added and learned. The simplicity of the POMDP framework makes it easy to add new actions at any time. All that is required is to add new conditional transition and observation probabilities that describe the effects of the new action. Incremental addition of actions does not affect the existing actions, but will affect the policy generated by the POMDP planner.

The high level components must be reliable enough to deal with the fact that an action may not always have the desired effect. This ability is inherent in the POMDP approach. In addition to selecting another action, the high level components must assign rewards to the low level action. A positive reward is given if the action was successful, and a negative reward is given if it was unsuccessful. Most robot control systems incorporate a mechanism for detecting when an action has failed to have the intended effect. Assigning a reward does not incur any more overhead than would be necessary just to detect action failure, but is crucial for using reinforcement learning of low level actions.

The design objectives for the POMDP/RL robot control system is that it must be:

Task-Directed: able to accept a task or goal and then work to achieve that task or goal. Changing the goal does not require learning or changing the control program.

Reactive: able to react quickly to sensor information and select an appropriate action according to the situation.

Reliable: high probability of success at achieving the goals that are assigned, even when its own hardware fails or when its knowledge of the world is inaccurate.

Adaptive: should adapt to long term changes in the environment.

Generalizing: able to apply knowledge learned in one situation to another similar situation.

Self Extending: able to add new actions to take advantage of knowledge about the environment.

The performance of the robot architecture will be measured against these criteria. The following sections describe the architecture, and Chapter 8 describes the experiments that test the architecture against these design objectives.

7.2 Description of the Robot Architecture

Figure 7.1 provides a detailed look at the robot control architecture. An overview of the system was shown in Figure 1.1. The planning layer sends action commands and reward signals to the action execution layer. The execution layer sends sensor information to the planning layer. When following a policy, the planning layer maintains a belief state over the possible states. The planning layer uses the current belief state to select a low level

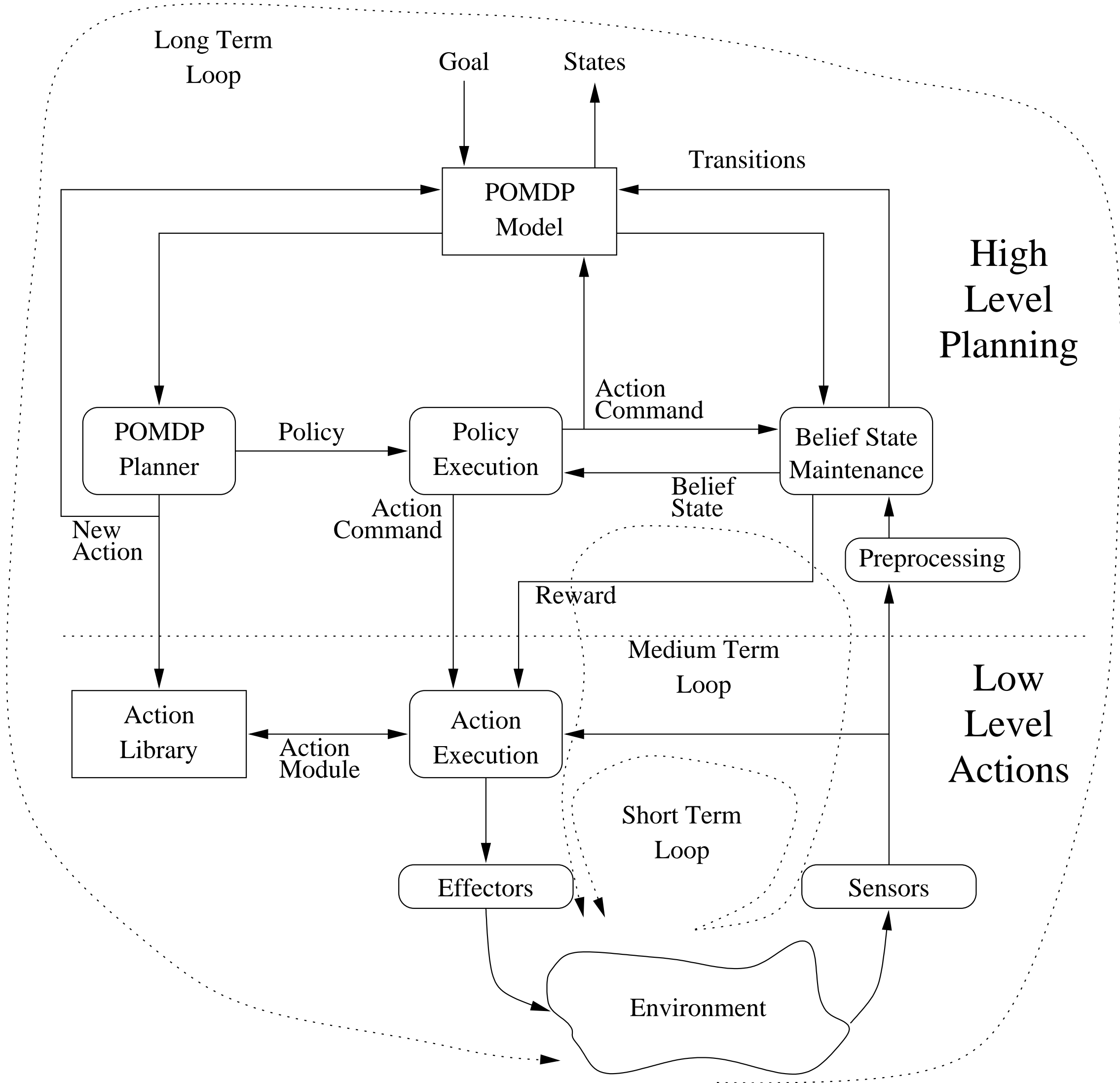


Figure 7.1: System overview.

action from the policy. The planning layer also tracks the state with the highest probability: the most likely current state. From the state transition probabilities, it is easy to find the expected next state when performing the current action in the most likely current state. When the most likely current state changes, the planning layer generates a non-zero reward for the action that is currently active. For all time steps where the most likely current state does not change, a zero reward is generated. The remainder of this section describes the components of the architecture in more detail.

7.2.1 Control Loops

The robot control architecture is based on three hierarchical control loops, each running at different time scales and different levels of abstraction.

- The short term control loop consists of the action execution module, effectors, environment, and sensors. Control decisions are made in the action execution module by the action that is currently being executed. It produces control signals that are sent directly to the effectors and makes decisions based on raw data from the sensors. This control loop provides fast response and tight coupling of the robot and its environment.

- The medium term control loop includes policy execution, action execution, effectors, environment, sensors, pre-processing, and belief state maintenance. Control decisions are made in the policy execution module according to the POMDP policy that is currently being executed. It produces control signals in the form of action commands that are executed by the action executor. The decisions made in the medium term control loop are based on the belief state, which is derived from the pre-processed sensor information combined with knowledge about the world in the form of transition and observation probabilities.

- The long term control loop includes the human investigator, POMDP model, POMDP planner, policy executor, effectors, environment, sensors, and belief state maintenance. Control decisions at this level are made by the human investigator, who may specify goals as immediate rewards in the POMDP model. For instance, the investigator may specify that the reward for state x is one and the reward for all other states is zero. In this case, the goal for the robot is to reach state x. The POMDP planner generates a policy from the POMDP model and sends the resulting control policy to the policy executor, which converts it to actions. The actions are sent to the action executor which converts them to control signals for the effectors. The sensor data is pre-processed and sent to the belief state maintenance module, which updates the POMDP model whenever transitions occur. The POMDP model can be dumped into a standard POMDP format (Cassandra 1998a) file at any time by the human investigator, who may edit the file and then load it back into the robot.

Each control loop works at a different level of abstraction. Reactive control is performed in the short term loop at the lowest level of abstraction. In the middle loop, POMDP execution provides long-term goal-directed activity. The middle loop bridges the gap between reaction and deliberation. In the middle loop, a policy is created from the POMDP model, and reactive execution is performed to follow the policy. At the outer loop, the discrete state space representation of the POMDP supports learning at the low level and provides a convenient way for the human investigator to specify goals.

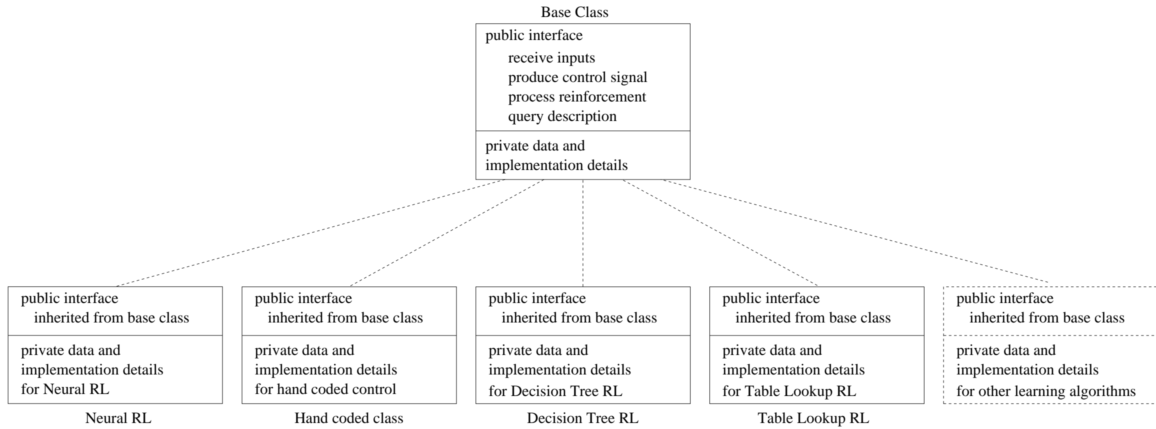


Figure 7.2: Low level actions are implemented as object classes.

7.2.2 Low Level Actions

The low level modules implement behavioral knowledge that is applied over a short time period of a few seconds. The POMDP policy executor chooses an action to execute and sends a command to the action executor. The action executor loads the required action from the library and executes it. The action modules change the world state by sending control signals to the effectors. After an action has been executed, the action executor must store the updated action back in the library for future use. The action executor can be viewed as a shell that passes information and rewards into the action module that is currently loaded and passes control signals out to the effectors. The low level action module has no notion of what it is supposed to accomplish, and its view of the world is based on very low level sensory data. Its only goal is to maximize the reward received from the POMDP planner.

During execution, the current low level action continuously runs through a cycle of sensing the current state, selecting control signals, and receiving reward. Sensors are used both for detecting external events and for detecting the internal state of the robot and its effectors. For instance, pulse encoders on the wheels give information about the robot's effectors. Thus, the robot's effectors and sensors are defined to be a part of its environment.

To achieve a high level of modularity, object oriented programming was used to implement the low level actions. As shown in Figure 7.2, a low level action is a software object with a standard interface. This approach allows the actual implementation of the software object to be hidden from the action executor and makes it very easy to stop executing one action and begin executing another. Since actions are simply software objects, they may be learned through RL or hand coded actions written by a human programmer. Hand coded actions may be included for purposes of debugging the POMDP planner, or to help bootstrap learning of other actions. The action execution module does not even know whether a particular action is hand coded or uses some form of learning. In the case of hand coded actions, the reward signals are ignored, but learned actions can change their policy if necessary to increase the long term reward that is received. All actions are identical, from the perspective of the rest of the system. The public interface to the actions is accessible by the action executor and the action library.

Each of the learned actions has its own RL module based on Q-learning with the decision tree function approximation method described in Chapter 4. Only one of the actions can

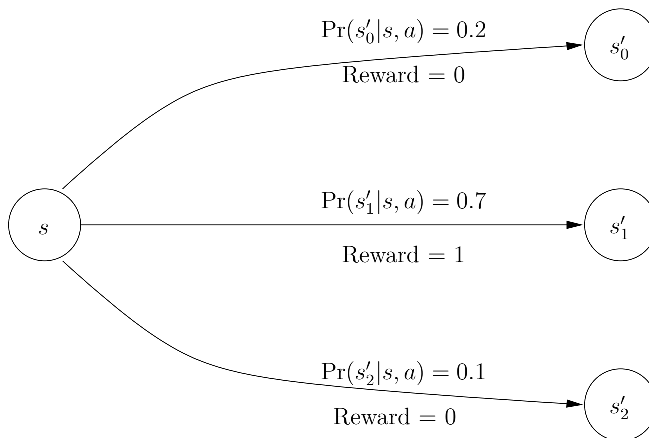


Figure 7.3: Rewards are assigned according to the most likely next state. In this case, the most likely state to follow s given action a is s'_1. When the most likely state changes, a reward of one is produced only if the new most likely state is s'_1.

be loaded into the action executor at any given time. When it is loaded into the action executor, an action module gains complete control of the actuators and has full access to sensor data. All of the RL modules are implemented identically, but each learns to perform different actions due to the rewards it receives.

The architecture in this dissertation was initially implemented with three hand coded actions: turn left, turn right, and go forward. The hand coded actions were used in debugging and experiments involving the POMDP planner. Then the hand coded actions were replaced with learned actions.

The learned low level actions are not trained to avoid obstacles, but learn to do so because it is necessary for maximizing the reward given by the POMDP planner. Suppose the policy for belief state b with most likely state s is to perform action a and the expected next state is s'_1, as shown in Figure 7.3. Equation 7.1 explains how the most likely state is calculated. Upon entering belief state b, the POMDP planner activates action a and waits for a change in the robot's most likely state. If a drives the robot into a wall, then it will not receive a positive reward. The only way it can receive a positive reward is to cause the robot's most likely state to change to s'_1. Action a will receive zero reward until the robot's most likely state changes or until it runs out of time. The time limit forces the action to avoid a periodic negative reward. The time limit is somewhat domain dependent, and should be longer than the expected time for any action to perform a state transition. When the time limit expires or a state transition occurs, the action will receive a positive reward only if the most likely state is s'_1.

Low level actions are executed by the action execution module, which retrieves specific actions from the library when needed. In the object oriented framework, this just amounts to getting a pointer to the action object. The POMDP execution module tells the action execution module which action to load, and the belief state maintenance module supplies reward signals. The reward is passed on to the action module so that it can learn to increase the long term reward that it receives. The learned actions may change some each time they are executed. The action library maintains a pointer to the currently executing action, so there is no need to store the previously executing action back in the library when a new action is chosen for execution.

At each time step, the action execution module must

- receive a signal from the POMDP executor specifying what action to execute,
- load the specified action module from the library (if it is not already executing),
- receive data from the sensors,
- receive reward from the belief state maintenance module,
- execute the action for one time step, using the sensory data and reward, and
- send the commands chosen by the action to the effectors.

The task for the action library is to manage the set of actions that are available to the robot. The action library has two interfaces: one for the POMDP planner, and one for the action execution module. The POMDP planner can request that a new action be created, while the action execution module can retrieve or store the implementation of a specified action. The POMDP model describes what each action does in terms of the action-dependent state transition probabilities $\Pr(s'|s,a)$ and observation probabilities $\Pr(z|s',a)$. These probabilities effectively represent the state transition(s) that can be expected when executing a given action a in a given state s, and therefore describe the effects of that action. The POMDP planner uses these probabilities to generate a policy. This results in a natural division between the description of an action, used by the POMDP planner, and the implementation of that action, used by the action execution module. The implementation of each action is a software object with a unique identifier. The action executor uses the action identifier to retrieve implementations from the library.

7.2.3 High Level Planning

The planning level is responsible for long term decisions and goal attainment. The human investigator can examine the states in the POMDP model and can specify a goal state. The goal state is specified as an immediate reward in the POMDP model. Planning with a POMDP can be more complex than simply specifying a goal state. Goal states are just a special case, convenient for human intuition.

The planner must

- track the current state of the robot in its world and detect when an action has resulted in an unexpected or undesired state,
- quickly and efficiently select a new action to execute at each time step,
- provide reward to the action as it executes, and
- provide a mechanism for the addition of new low level actions.

The following sections describe a planning system that meets all of these requirements.

During execution of an action, an undesired state transition can occur. Although it is undesired in the current context, it may be useful in the future. Therefore, it may be advantageous for the robot to have an action that is *expected* to cause that state transition. The belief state maintenance module notifies the POMDP model whenever a state transition occurs. The POMDP model records all state transitions during plan execution and updates

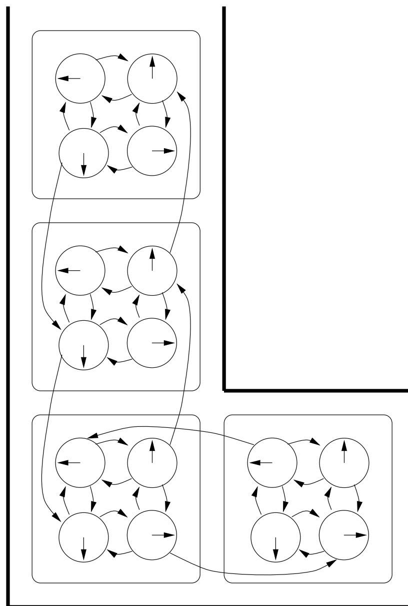


Figure 7.4: State map for a small section of corridor. All orientations within each position are shown.

the estimated state transition and observation probabilities. When it begins to generate a new policy (i.e., when the human investigator specifies a new goal), the POMDP planner searches for possible new actions in the POMDP model. Whenever the POMDP planner finds that there are observed state transitions which are not primarily associated with any action, the POMDP planner sends a request to the action library for a new action to be created and updates the POMDP model to include the new action. A more detailed explanation is given in Section 7.2.4. Planning is then performed as if the new action has always existed, and a policy is generated which may use the new action. The new action is initialized with no knowledge about what it is supposed to accomplish, but that information is present in the POMDP model. When the new action is chosen for execution, it will begin to receive rewards and will learn to perform the state transitions that are expected in the POMDP model.

The states in the POMDP model correspond to nodes in a topological map of the environment. In the topological map, each possible position and orientation for the robot is represented, along with the state transitions that are possible from that state based on environmental constraints. An example state diagram for a corridor section with a turn in it is shown in Figure 7.4. In this figure, the boxes represent a single position with four possible orientations. The POMDP state space in this example has 16 states.

The POMDP planner constructs a policy that, when followed, will cause the robot to visit a chain of states from the current state (position and orientation) to the goal state. As long as the POMDP map accurately reflects the world, the actions themselves will never be required to perform in an inappropriate situation. For instance, the value of the "move forward" action in states where the robot is facing a wall is very low. It is the responsibility of the POMDP planner to determine the value of each action in each belief state, and its ability to correctly do so depends on the accuracy of the information in the POMDP model.

The actions are represented to the POMDP planner in terms of how they affect the world state. This information is implicitly encoded in the action dependent transition probabilities in the POMDP model. The POMDP planner finds the value of each action at each point in the belief space, in the form of the value function, which maps directly to a policy for choosing the appropriate action for each belief state. This process was explained in detail in Chapters 5 and 6.

At each time step, the belief state maintenance module updates the current belief state using Equation 5.2 and then calculates the most likely state and the expected next state. When the most likely state changes from one time step to the next, the belief state maintenance module reports the state transition to the POMDP model and sends a reward to the low level action executor.

The action modules may receive more complete information from the sensors than the belief state maintenance module does. In order to simplify the POMDP and make it solvable, the sensory input is pre-processed to provide a small number of observations to the belief state maintenance module. The pre-processing is domain-specific and depends on the available sensors and features of the environment. Pre-processing converts raw sensor data into observations that are meaningful in the POMDP model. This may result in some occasional discrepancy between the most likely state derived from the belief state and the actual state, but this is generally not a problem, since discrepancies occur rarely when the sensor pre-processing is sufficient. Reinforcement learning is robust enough to converge even with occasional erroneous rewards (Sutton 1988; Dayan 1992; Watkins and Dayan 1992; Jaakkola, Jordan, and Singh 1994; Tsitsiklis 1994; Watkins 1989). Empirical tests on the architecture showed that the most likely state matches the current state more than 90% of the time. This has been sufficient to show convergence of the learned actions in the POMDP/RL experiments. The accuracy of the belief state can usually be improved by increasing the number of observations, but at the expense of a more complex POMDP model which requires more planning time to find a policy.

The remaining problem is how to determine when a state transition has occurred, given that the current state is represented as a probability distribution b. The belief space is continuous; so every time step leads to a new belief state. The POMDP world model consists of a COMDP model plus a set of observation probabilities that are dependent on the state and action. Low level actions are meant to achieve state transitions in the underlying COMDP. However, the planning level maintains only a belief state, so some mapping from belief states to the underlying COMDP states is needed. For a given belief state b, the *most likely state* is given by

$$s^b = \operatorname*{argmax}_{s \in \mathcal{S}} (b_s), \tag{7.1}$$

which simply selects the state with the highest probability in b. The belief state maintenance module detects state transitions by comparing the most likely state s^b_t from Equation 7.1 at each time step with the most likely state s^b_{t-1} on the previous time step. If $s^b_t \neq s^b_{t-1}$, then a state transition has occurred.

With this method of detecting state transitions, it is possible to generate rewards for the low level actions. Recall from Chapter 5 that one of the four parts of the POMDP model is a mapping $\Pr(s'|s,a) \forall s \in \mathcal{S}, a \in \mathcal{A}$. The *expected state* when a is executed in b is given by

$$s^e = \operatorname*{argmax}_{s' \in \mathcal{S}, s' \neq s^b} \left(\Pr(s'|s^b, a) \right). \tag{7.2}$$

Whenever $s^b_t = s^b_{t-1}$, a reward of 0 is sent to the action execution module; otherwise, the reward is calculated by comparing the current most likely state s^b_t and the expected state at the previous time step s^e_{t-1}. If $s^b_t = s^e_{t-1}$, a reward of 1 is sent to the action execution module. Otherwise, a reward of -1 is sent. A timer mechanism is also implemented so that actions must perform a state transition to avoid a periodic negative reward. Without the timer mechanism, an action may simply turn off all effectors and receive zero reward. The timer mechanism ensures that turning off all effectors will result in negative reward. From the perspective of the low level actions, the goal is to achieve a specific state transition, and it receives a positive reward only after achieving that goal.

In addition to the most likely state mechanism, the belief state maintenance module maintains a counter that is reset to zero every time a state transition is detected. Whenever the counter passes some threshold τ, a reward of -1 is sent to the behavior execution module and the timer is reset to zero. The counter mechanism forces the action to search for some way of avoiding a periodic negative reward. τ is chosen so that the action is given ample time to achieve the desired state transition. In the experiments for this dissertation, τ was set at 100; a properly functioning action typically required only 10 steps to achieve a transition. The selection of τ is not critical, as long as the action can cause a state transition within τ time steps. A minimum value for τ can be found for most domains from the speed of effectors, length of the time step, and distances between states in the POMDP model.

The POMDP planner is responsible for generating a policy that is sent to the policy executor. Finding an optimal policy is time consuming, but once the policy is found for a goal state, that policy is saved for future use. The policy consists of a set of labeled $|\mathcal{S}|$ vectors. The number of vectors in the policy depends on the probabilities and rewards in the POMDP model. In general, the size of the policy set cannot be determined *a priori*. The label on a vector tells what action to perform if that vector has the largest dot product with the current belief state.

Chapter 6 describes the parallel restricted region algorithm that is used to find optimal policies for the robot. Whenever the human investigator changes the goal state in the POMDP model, the POMDP planner must be run to find a new policy. Before calling the POMDP solver, the planner first searches for new actions that can be created. Searching for new actions could be performed at some other time, or even continuously, but it is convenient to do it whenever the POMDP planner needs to generate a new policy, since this allows the new action to be incorporated in the policy. The process of searching for and creating new actions requires very little computation, compared to the time required to find a POMDP policy; so the additional time that the planner requires is not significant. Searching for new actions every time the planner is invoked may result in some loss of efficiency, while the new action is being learned. After creating a new action or finding that no new actions are indicated, the planner invokes the POMDP solver to generate the policy and then sends the policy to the policy execution module.

Planning and execution are completely separate and can run in parallel. The robot can create a policy for its next goal while moving toward the first one. Note that this does not presuppose that the current goal will be reached, since a POMDP policy does not assume any particular starting state. The policy execution module could easily take policies from a queue that is filled by the planner. The policy executor could determine when to take the next policy from the queue simply by comparing the current most likely state with the goal state for the current policy. This mechanism was not implemented, but is a straightforward extension of the robot control architecture.

Action	State	A	B	C	D
0	A	0.2	0.8	0.0	0.0
	B	0.0	0.0	0.7	0.3
	C	0.3	0.7	0.0	0.0
	D	0.0	0.0	0.35	0.65
1	A	0.0	0.51	0.49	0.0
	B	0.5	0.25	0.0	0.25
	C	0.35	0.45	0.0	0.2
	D	0.0	0.0	1.0	0.0

Table 7.1: Example transition probability matrices.

7.2.4 Finding New Actions

Every time the POMDP planner is invoked, it queries the action library to determine if all of the actions are stable. The stability measure for each action is the time discounted average reward $\overline{r}^{a_t}$ that the action has received, defined by the following:

$$\overline{r}_t^{a_t} = \alpha \overline{r}_{t-1}^{a_t} + (1-\alpha) r_t \tag{7.3}$$

where $0 < \alpha < 1$ is a weighting factor. If all of the actions are stable, ($\overline{r}^{\,a} < \mu \forall a$, where μ is the stability threshold) the POMDP planner searches for transitions that are not associated with any action and creates a new action to learn those transitions. The planner creates a transition matrix and initializes the transition probabilities by averaging the total number of uncovered transitions that originate in each state. For instance, if state s has three transitions to other states that are not associated with any action, then the row in the new transition matrix corresponding to s will have three entries of .33 and the remaining entries will be zero. The planner adds the transition matrix to the POMDP model and instructs the action library to create a new learned action.

From the conditional state transition probabilities in the POMDP model, the set of states $\mathcal{S}^{sa}$ that are reachable from state s given action a is defined as

$$s' \in \mathcal{S}^{sa} \iff \Pr(s'|s,a) > 0 \, \forall s' \in \mathcal{S}. \tag{7.4}$$

This is simply the set of states that can be reached by following the edges leading from state s' in the directed graph that represents the underlying MDP. The *primary transition* for action a in state s is given by

$$s_p^{sa} = \operatorname*{argmax}_{s' \in \mathcal{S}^{sa}} \left(\Pr(s'|s,a) \right) \tag{7.5}$$

and the set of *non-primary transitions* is given by

$$\mathcal{S}_n^{sa} = \mathcal{S}^{sa} - s_p^{sa}. \tag{7.6}$$

Table 7.1 shows the transition probability matrices for a four state example problem with two actions: 0 and 1. When taking action 0 in state A, the most likely next state is B. The entry of 0.8 in the first row of the table represents the primary transition for action 0 in state A. The probability of a transition from A to A is also non-zero, and this is a non-primary transition.

Action	State	A	B	C	D
0	A		*		
	B			*	
	C		*		
	D				*
1	A		*		
	B	*			
	C		*		
	D			*	

(a) Primary Transitions

Action	State	A	B	C	D
0	A	*			
	B				*
	C	*			
	D			*	
1	A			*	
	B		*		*
	C	*			*
	D				

(b) Non-primary Transitions

Table 7.2: Primary and non-primary transitions for the example transition matrices.

Table 7.2 shows all of the primary and non-primary transitions for the example transition matrices in Table 7.1. Non-primary transitions may indicate the existence of a learn-able action. If there is a non-primary transition (s, a, s') for taking action a in state s that is not the primary transition for some other action a' in s, then (s, a, s') is an *uncovered* non-primary transition. The occurrence of such a transition in the POMDP model indicates that the state transition is possible, and therefore, an action can be created to perform that transition.

Uncovered non-primary transitions may occur during action learning or in normal operation. When a new action begins to be learned, it sends random control signals to the effectors, possibly causing uncovered non-primary state transitions. The belief state maintenance module updates the POMDP model to reflect the observed transitions; so the POMDP model will contain transitions that are not the primary transition for any action. Uncovered non-primary transitions can also occur in normal operation. For instance, if there are three actions available in a given state s, and at least one action has four non-zero transitions, then there is guaranteed to be at least one uncovered non-primary state transition from state s.

Random control signals are acceptable in a simulated environment, but could be dangerous on a real robot. To deal with this, a new module should be added to the action level that monitors the control signals sent to the effectors and overrides any signals that it deems dangerous. The new module is unlikely to affect learning, other than to prevent the learning of dangerous effector settings. This extension to the architecture was not implemented in this work, since all of the experiments were conducted in a simulated environment.

For each non-primary transition $s' \in \mathcal{S}_n^{sa}$, there may be some other action a' for which the primary transition $s_p^{sa'} = s'$. In this case, it is not necessary to learn a new action to cover the transition to s' because it is already covered by another action. All covered actions are removed from $\mathcal{S}_n^{sa}$ by

$$\mathcal{S}_u^{sa} = \mathcal{S}_n^{sa} - s_p^{sa'} \forall a' \in \mathcal{A}, s \in \mathcal{S} \tag{7.7}$$

leaving only transitions for which there is no action.

Table 7.3 shows the uncovered non-primary transitions for the example transition matrices in Table 7.1. This table represents transitions that have occurred in the system, but for which there is no associated action. A new action can be instantiated to learn at least some of these transitions. The new action is given the label 2 and is added to the action

Action	State	A	B	C	D
2	A			*	
	B				*
	C	*			*
	D				

Table 7.3: Combined uncovered non-primary transitions.

library. Table 7.3 is used to generate a transition probability matrix for action 2 and the matrix is added to the POMDP model.

Suppose an action is created to perform all of the uncovered non-primary state transitions. The action may be required to do something very different in state s_0 than it is required to do in s_1. If the sensory information provided to the action is adequate to discriminate between s_0 and s_1, then there may not be a problem, but what happens when the action cannot discriminate between the two states? As long as the robot visits one of the states more often than the other, then there is still no problem because this will provide a bias towards the most visited state.

The action tries to maximize the reward that it receives from the belief state maintenance module. If it visits s_0 more often than s_1, then it will receive a higher return for learning to perform the action for s_0 correctly and will learn the one action that gives the highest long term reward. Since the POMDP model is updated with every transition, eventually it will catch up to what the action is actually doing in s_1, and the belief state maintenance module will no longer give a negative reward when it does the same thing in s_1 that it does in s_0. This argument holds for any number of indistinguishable states.

Note that this leaves a non-primary transition from s_1 that still has no action. The next time the POMDP planner is run, that transition will be assigned to another action which may or may not make it a primary transition.

One caveat is that the primary transition for an action in the POMDP model must not be allowed to change while the new action has not converged to a stable policy. In general, the POMDP model can only be updated when the low level actions are stable. For some period after an action is created, it may not cause its primary transition to occur. If the transition probabilities are updated as usual, it is possible for the primary transition to change to one that is more likely to occur when the effector settings are random. For this reason, an update to the POMDP model is ignored if it would cause the primary transition for an action to change.

Chapter 8

Experiments and Results

This chapter describes the four experiments that were performed using the robot control architecture in the Khepera domain. Experiment 1 showed that hand coded actions could be replaced by learned actions using the transition probabilities that were found using hand coded actions. The hypothesis of this experiment was that learned actions would perform at least as well as, or better than, the hand coded actions under normal operating conditions. Experiment 2 tested the hypothesis that learned actions would be more reliable than hand coded actions in the presence of simulated sensor and effector failure. Experiment 3 tested the generalization of learned actions. The hypothesis was that learned actions would initially perform poorly in novel situations, but would eventually perform better than hand coded actions. Experiment 4 tested the ability of the architecture to create new actions with no pre-existing transition probabilities. The hypothesis was that adding a new action would, after learning, result in improved performance.

8.1 Simulation

This research relies on simulation of the robot and its environment. The use of simulation in robotics research is somewhat controversial. Simulation provides an alternative to the mechanical problems, high costs and long turnaround times associated with currently available robot technology. Simulators historically have not represented the real world with a sufficient degree of accuracy to allow lessons learned from simulation to be used reliably on a hardware robot. However, simulators are becoming increasingly detailed and sophisticated. With increases in computing power, it is now feasible to build simulators that attempt to accurately model robotic sensors, effectors, and the environment.

Some researchers have demonstrated the ability to test theories on a simulator and then transfer the results to hardware. Feiten et al. (1994) describes a simulator used to develop control algorithms for a real robot. They validate their algorithms in the real world using an autonomous mobile robot. They found that algorithms developed in the simulation environment could be ported to the real robot with relative ease. They divide the simulator into three levels:

1. The planner level contains the action and motion planning algorithms.
2. The pilot level deals with the commands given by the planner level and interfaces with the servo level.
3. The servo level performs low level control of the robot servos and runs the sensors.

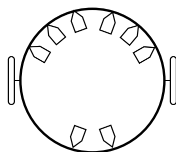


Figure 8.1: Arrangement of Khepera infrared sensors.

They specify the desired features of a simulation environment at each level. At the servo level, they use different sensor simulation techniques depending on the current stage of development in the higher level control algorithms. There is a trade off between accuracy and speed of simulation; so they sacrifice some accuracy early in the development phase in order to get the control system working and then use a more accurate low level simulation for fine tuning. They claim that their simulation architecture is general enough to be used for a wide variety of autonomous robots.

Mahadevan and Connell (1992) use reinforcement learning for automatic programming of action based robots. They use a simulator for initial training and then test the resulting system on a physical robot. They find that the time required for developing software using the simulator is much lower than the time required when using a real robot, because of hardware failures and other problems associated with hardware robots. They find that only minor adjustments are necessary to transfer from simulation to hardware environments and, overall, development is accelerated.

Dorigo and Colombetti (1994) showed that reactive actions could successfully be learned using a simulator and then transferred to a physical robot. They use two small moving robots—AutonoMouse II and AutonoMouse IV. These robots have two effectors: motors which drive wheels or tracks. AutonoMouse II has six light sensors, and a microphone for sensors. AutonoMouse IV has two light sensors, a sonar, and two "whiskers." These robots are relatively simple to simulate. They did not attempt to simulate noise in the sensors and actuators, but were still able to achieve good results when transferring the system from simulation to the real robot and only minor modifications to the system were required.

There are several major reasons to consider simulation as an alternative to building a hardware robot. One reason is that the expense of purchasing and maintaining a hardware robot can be avoided or at least delayed by using simulation during the initial development of the control architecture. Another reason to use simulation is that simulated robots are not susceptible to physical damage from collisions, vibration, and other environmental dangers. Another major advantage of simulation is the amount of control that the investigator has over the conditions of the experiment. The simulator can be put into the same initial state for repeated experimentation with more accuracy than is possible when a hardware robot is used. This higher level of control over the experiment can lead to higher confidence in the results of architectural comparisons.

The Khepera robot simulator was selected for these experiments. Khepera is a small (55mm diameter), modular robot. The basic system has two drive wheels, one on each side, that can be controlled independently, providing the ability to control speed and direction of the robot. Khepera has a small processor and eight infrared transmitter/receiver pairs arranged as shown in Figure 8.1. It also has pulse encoders on the wheels, but they are not used in this work. Other components can be added, such as a gripper or CCD camera. The low number of sensors allows fast simulation and reduces that amount of information

which must be processed by the controller. However, it also limits the ability of the robot to make subtle distinctions in its environment. This could lead to difficulty in the robot knowing where it is.

The Khepera robot is simple enough to be adequately modeled in a simulator (Michel 1995). The simulator is a discrete time simulator and can run in real-time or much faster than real time. The user-supplied robot controller is called by the simulator at each time tick. The sensor model includes stochastic modeling of noise and responds similarly to the real sensors. The simulation environment includes some stochastic modeling of wheel slippage and acceleration.

The user writes a set of subroutines that conform to a simple programming interface. The subroutines are used by the simulator to initialize, run and shutdown the user control code. The simulator provides routines that allow the controller to query sensors and control effectors.

The user can create, save and restore a simulated environment with obstacles. However, the simulator does not include any settings to simulate failures. Hooks were added into the simulator to allow simulated sensor and effector failures. The simulator was chosen for several reasons.

Execution speed: Reinforcement learning may take many trials to learn a low level action. For instance, learning the actions for Experiment 1 would require more than two hours of real-time operation. Higher execution speed in the simulator means that the experiments can be completed in under five minutes.

Stochastic simulation: In the physical world, there is always some uncertainty, and stochastic simulation helps to model this uncertainty. Also, the prevailing view in the literature is that some randomness is desirable for reinforcement and POMDP learning, because it results in better generalization. Stochastic simulation is important to model the robot in its environment more accurately than with deterministic simulation.

Level of detail: The simulator provides the same type of sensor data and effector control as would be available on a real robot, while expending as little computational resources as possible. If the simulator does not accurately reflect the physical world, then there is no guarantee that any agent architecture developed in simulation will be useful in the real world.

8.2 Creating the POMDP Model

For these tests, a small navigation area was created and divided into 16 positions. Then the robot's orientation was discretized into the four compass directions, resulting in 64 possible states for the robot. The resulting environment and state space is shown in Figure 8.2. For belief state maintenance, sensor information was pre-processed and reduced to four bits by combining the sensors in pairs and setting a threshold on the combined sensor values. This resulted in 16 observations for the POMDP model. These measures kept the state space size small so that exact solutions could be found for the POMDP policy. However, even with this simple state space, generating a policy required several days of computation.

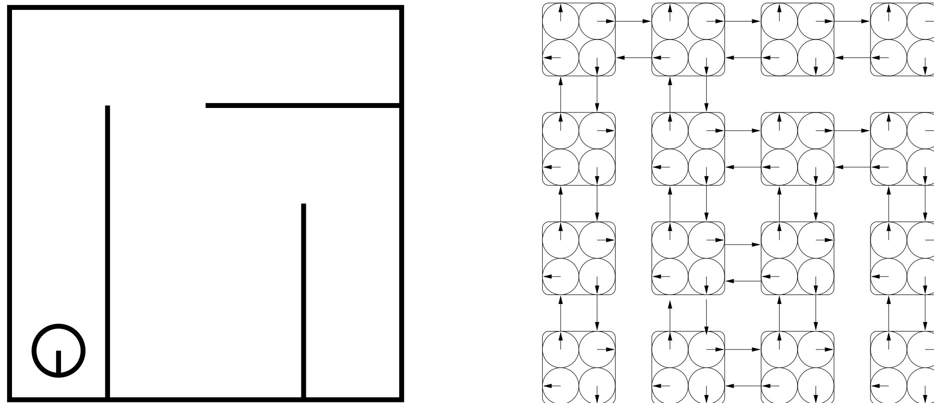


Figure 8.2: Environment and state space for the experiments.

The action modules received the sensor data directly, with no pre-processing. The commands sent to the effectors by the action modules consisted of three commands for each of the two wheels: forward, stop, and reverse.

The POMDP model was initialized using three hand coded actions. The actions were: go forward, turn left, and turn right. The hand coded actions try to avoid obstacles while performing their required function. With the hand coded actions installed in the library, all state transition and observation probabilities in the POMDP were initialized to zero.

Next, the world model was loaded into the simulator, and the robot was placed at a random position and orientation within each of the 64 possible states for 30 trials each. On ten of the trials, the robot executed the forward action. On ten of the trials, it executed the turn left action, and on the remaining ten of the trials, it executed the turn right action. At each time step during a trial, the actual state of the robot and the observation from sensors were recorded. For each set of ten trials, the observed transitions and observations were used to calculate a rough estimate of the transition and observation probabilities for that state in the POMDP model. This "seeded" the POMDP transition and observation probabilities.

A simple control policy was hand coded to cause the robot to wander about in the environment. The control policy simply chose an action at random and executed it until either a state transition was observed or for 500 time steps. Then another action was chosen and executed. As the robot wandered around in its environment, state transitions were noted, and the POMDP model was further refined. Figure 8.3 shows the improvement in belief state estimation as the robot wandered in its environment. After 300000 time steps, the most likely state matched the actual state more than 90% of the time. Most of the discrepancies were caused by the most likely state changing one or two time steps before or after the actual state change. Early or late changes in the most likely state should have little effect on the performance of the POMDP execution or on the reinforcements sent by the belief state maintenance module.

After this initialization, the POMDP model reflected the state transition and observation probabilities for the three hand coded actions. Initialization would have taken approximately 300 minutes on a real robot, but only required about 5 minutes on the simulator. Once the POMDP model was initialized, a policy could be generated and experimentation on learning actions could begin.

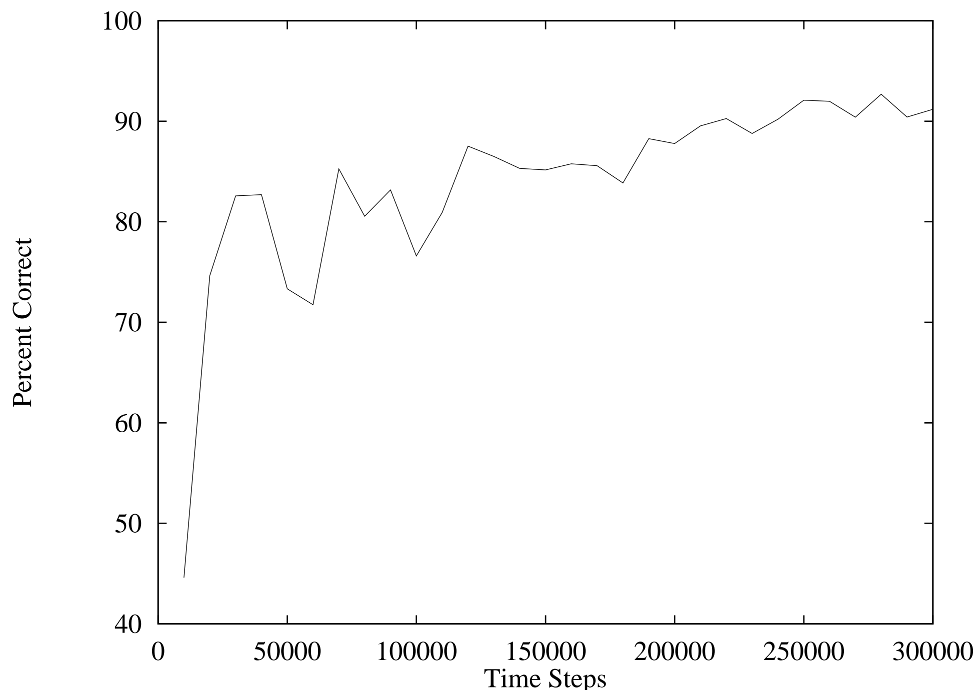


Figure 8.3: Accuracy of the most likely state improves as the robot wanders around in its environment.

For simplicity and due to computational limitations, state 13 (the robot's location in Figure 8.2) was chosen to be the goal for all of the experiments. The parallel restricted region algorithm was used to find a policy that would direct the robot to state 13 from any initial state. This is a very large POMDP problem, with 64 states, 3 actions and 16 observations. Generating a policy required several days of computation. To reduce the time required to generate the policy, a high threshold was set on the value returned by the linear program solver. This resulted in fewer vectors being included in the policy at each iteration than otherwise would have been. Cassandra (1998a) proposed this method for speeding up POMDP solution, and it is common practice in the literature. Using a high threshold probably resulted in a policy that was less than optimal. However, the policy was adequate to provide reliable performance. Because finding a policy required approximately 16 days of computation, it was not feasible to generate more than one. The policy thus obtained was used for all of the first three experiments. An additional policy was created for the fourth experiment.

8.3 Experiment 1: Learning Replacement Actions

The hypotheses for the first experiment was that the hand coded actions could be replaced with learned actions and that the learned actions would perform equally or better than the hand coded actions.

After solving the POMDP to find the policy, the hand coded actions were replaced with decision tree based reinforcement learning components. For training the RL actions,

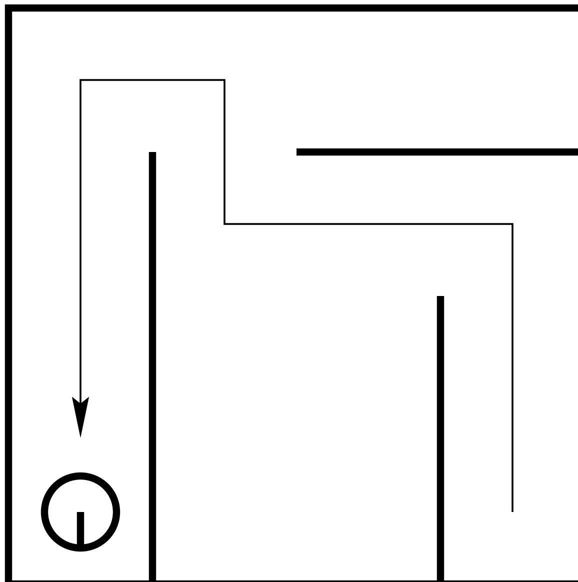


Figure 8.4: The path used for learning the actions.

the robot was always started in the southeast corner of the environment and had to reach the goal state in the southwest corner of the environment, as shown in Figure 8.4. This selection of starting state required the robot to use all three actions. Using the same task for all trials simplified analysis of the results. The new actions had no initial knowledge of what functions they were to perform and learned what to do under complete control of the POMDP planner. The robot was run for 500 trials (one trip from start to goal), with the POMDP planner assigning rewards to the actions at each state change. An additional 500 trials were run with the hand coded actions, in order to establish a performance baseline.

Figure 8.5 shows the performance of the learned actions relative to the hand coded actions. In less than 100 trials, the learned actions achieved similar performance to the hand coded actions and continued to improve over the next 100 trials, while occasionally performing worse than the hand coded actions. After 250 trials, the learned actions were performing equal to or slightly better than the hand coded actions. The hand coded actions were programmed using human intuition about what effector settings are appropriate for a given situation. The human intuition may result in good choices for effector settings, but it is very unlikely that the hand coded approach would result in actions that behave optimally. The learned actions, on the other hand, use reinforcement learning to search for an optimal control policy. It is hardly surprising, then, that the learned actions slightly outperform the hand coded actions.

8.4 Experiment 2: Sensor and Effector Failure

The primary expected advantage of the RL actions over hand coded actions is their adaptability. Thus, it is expected that the system should perform reliably even when the robot's hardware degrades, up to a point. The hypothesis is that the overall performance should degrade gracefully (i.e., roughly linearly with a shallow slope) as sensors and actuators gradually fail.

To test the hypothesis, the sensor or actuator performance was degraded in a controlled

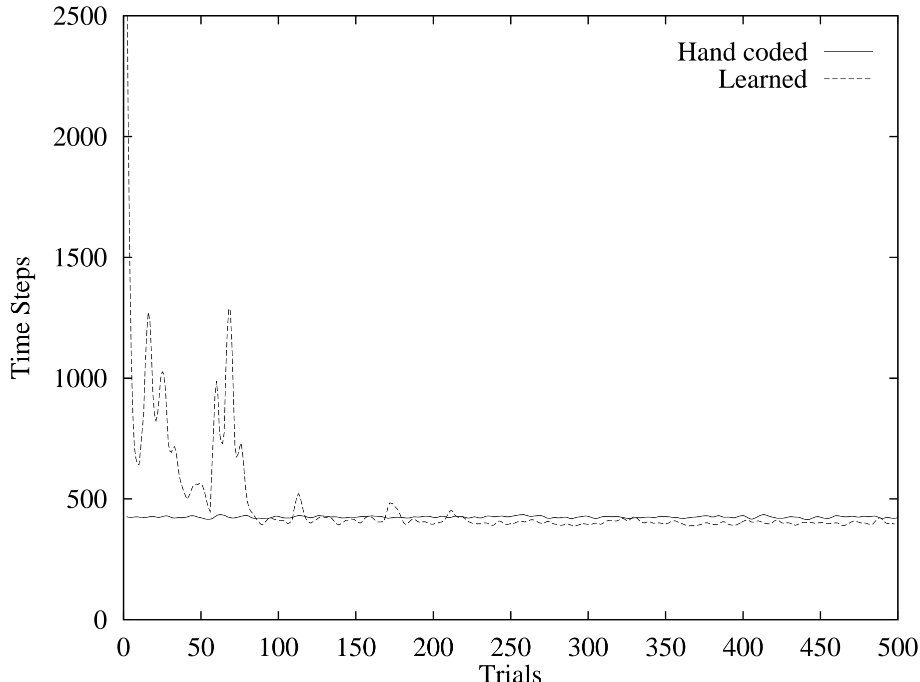


Figure 8.5: Performance of hand coded and learned actions while learning the three actions.

manner and measurements were taken on the performance of two systems: one with the learned actions and the other with hand coded actions. The hand coded actions provide a control or baseline of performance to demonstrate task difficulty. The learned actions from the previous experiment were used in this experiment.

For each trial in an experiment, the simulated sensor or actuator performance was degraded by a fixed amount. For each system configuration (learned or hand coded), the simulated robot was started from every position and orientation in the environment and the number of time steps that it took to reach the goal was recorded. On some trials, the robot became wedged and was unable to reach the goal. After a failure, the robot is re-started in the same position and orientation to try again. Thus, for each setting of degradation, data was collected from 64 instances of the robot achieving the goal.

Performance was assessed with respect to two metrics. *Failures during trial* counts the number of times that the robot was unable to reach the goal within 7500 time steps over all 64 configurations in a trial; thus, this is the estimator of reliability. *Average steps to goal* assesses efficiency: how long does it take to achieve the goal. Obviously, repeated failures in a trial causes this number to increase considerably. We first consider gradual sensor failure, then intermittent effector failure.

Sensors can fail gradually as battery power is used up, when dust accumulates on the sensors, or when calibration is lost. To assess the effect of gradual sensor failure on system performance, a scaling factor was set on all of the sensors to restrict their range. If the range of the sensors was 90% of normal, then the sensors were said to be impaired by 10%. Testing started with all sensors fully operational (0% impairment). For each unit (1%) impairment from 0% to 100%, the robot was directed to go to state 13 from state 64.

Figure 8.6 shows the results of this experiment. The curves were pre-processed with a Gaussian eleven smooth ($\sigma = 4$) (Cohen 1995). The hand coded actions failed to reach

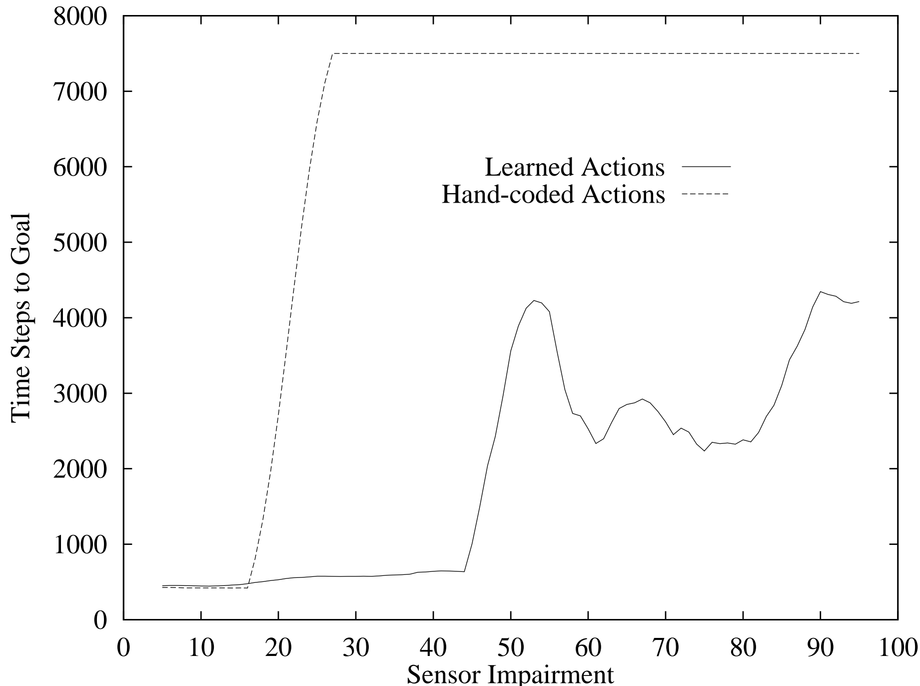


Figure 8.6: Response to gradual failure of all sensors.

the goal within 7500 time steps on every trial where the sensor degradation was greater than 21%. The hand coded actions use thresholds on the IR distance sensors to determine when the robot has collided with a wall and take appropriate action. At sensor impairments above 21%, the collision threshold is never exceeded. The result is that the robot becomes wedged against a wall and does not take the appropriate steps to become unwedged. For the hand coded actions, as expected, the time steps to reach the goal is strongly correlated with the number of failures. Due to the threshold effect, the hand coded actions begin to perform poorly with even small amounts of sensor impairment. The thresholds were chosen after careful trial-and-error to perform well in the best case, when there was no sensor degradation. Changing the thresholds could result in better performance in the presence of sensor failure, but at the cost of poorer performance in the best case.

The learned actions show better reliability. The robot failed to reach the goal within 7500 time steps on only four of the 100 trials. The time steps to reach the goal during a trial increases gradually, staying under 600, even with sensor impairment as high as 50%. At sensor impairments greater than 50%, the sensor pre-processor for the belief state maintenance module no longer reports all observations. This results in poor belief state maintenance, and consequently, poor action selection. A threshold in the sensor pre-processor is never exceeded, and the only observation that is reported to the belief state maintenance module indicates that nothing is nearby. As a result, the belief state maintenance module only has information from the transition probabilities to determine the current belief state. Even when no observations are reported, the belief state maintenance module continues to operate well enough for the robot to reach its goal state on 92% of the remaining trials, although it takes a great deal more time.

The hand coded actions used thresholds on the sensory inputs to select effector settings

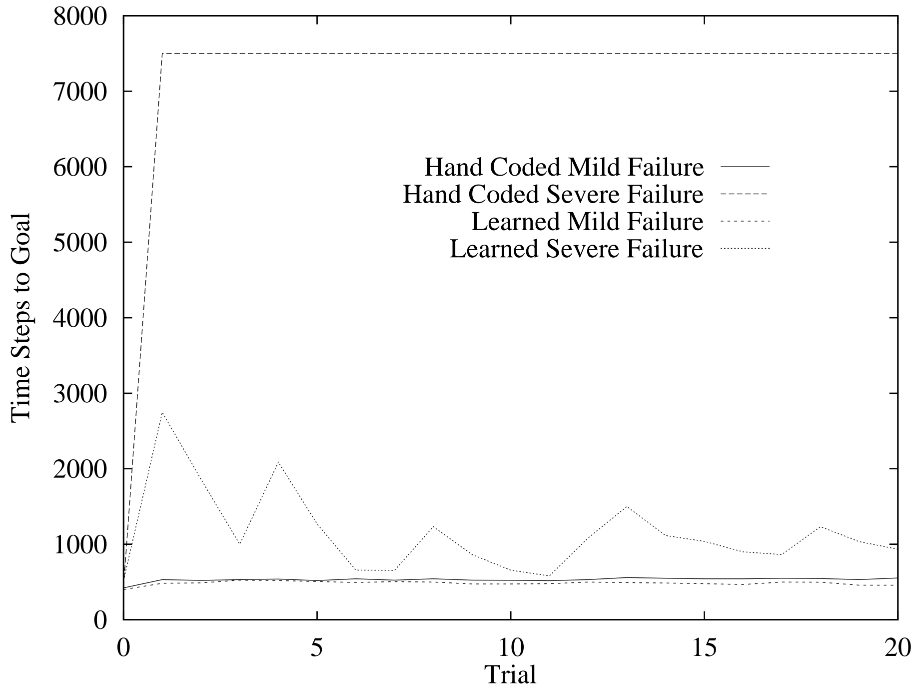


Figure 8.7: Response to intermittent failure of an effector.

at each time step. This method works well in the absence of sensor failure and allows the programmer to choose effector settings based on human intuition. However, as the sensors fail, the thresholds are no longer appropriate and the actions select inappropriate effector settings. The learned actions simply adapt as the effectors fail, resulting in higher reliability.

Effectors may partially or intermittently fail due to loose contacts, shorted motor windings, or other causes. To assess the effect of intermittent partial effector failure, a subroutine that generated the control signal for the right motor was modified to reduce the value of the command signal by a fixed amount with a 50% probability. For example, although the robot control system issued a command for the motor to run at a speed of -5, the failure simulation routine may actually send a -3 command to the right motor. If the command signal was negative, then 2 was added; if the signal was positive, then -2 was added. Mild failures were simulated with ± 2 and severe failures with ± 4 added to the right motor command signal.

The response of both hand coded and learned actions to mild and severe effector failures was tested for the same POMDP policy used in the previous experiments. For each test, one trial was run with all sensors and effectors operational to establish a baseline and then 20 more trials were run while simulating an intermittent failure in the right motor.

Figure 8.7 shows the results of this experiment on the *average steps to goal* performance metric. Both hand coded and learned actions performed very well with mild failure; the performance for both types of behaviors was only slightly degraded in the presence of intermittent effector failure. In the presence of severe effector failure, the learned actions quickly adapted to the failing effector, and reached the goal on all trials. The hand coded actions did not adapt to the effector failure, and failed to reach the goal on any of the 20 trials with severe failure.

When the robot became wedged against a wall, the hand coded actions use a "reverse and turn" strategy for getting unwedged. When a wedged situation was detected, the action sets a timer and sets the effectors to move away from the wall. The effector settings are not changed for a fixed number of time steps determined by the timer. When the timer runs out, the effectors are set to turn to robot away from the wall for one time step, and then normal operation is resumed. This approach was sufficient to provide reliable operation when effectors were working perfectly or with mild effector failure. However, with severe failure, this strategy no longer worked because the robot was unable to move far enough away from the wall and/or to change its heading sufficiently when attempting to get unwedged. The learned actions did not have a fixed strategy for dealing with this situation and could search for a better strategy when the current one no longer worked. Thus, on the first five trials with severe failure, the learned actions were adapting to the new effector response. After about five trials, the actions had adapted and performed well considering the reduced reliability of the effectors.

8.5 Experiment 3: Generalization

The robot followed the same path on each trial when learning the actions. The path took the robot to only 16 of the 64 states in the environment, and the path had more left turns than right turns. It would be reasonable to expect that the learned actions will perform well for the states that they have been trained on and that they will perform not as well in novel situations, but to what degree? The next experiment was conducted in order to determine how well the robot could generalize and whether it could continue to improve performance by learning from experience.

A random position and orientation was chosen within each of the 64 states, one at a time, and the robot was required to navigate to the goal twice from that initial position. For each new start, the time steps to the goal were recorded for one trial of each version of the robot: learned and hand coded actions. This was repeated for 2000 trials, 1000 per version.

Figure 8.8 is a plot of the relative performance of learned vs. hand coded actions. The plot shows the time taken for the hand coded actions minus the time taken for learned actions to reach the goal over 1000 trials. The plot is shown with an 11 Gaussian smooth ($\sigma = 7$) and a 99 Gaussian smooth ($\sigma = 65$). Over the first 200 trials, the learned actions did not perform as well as the hand coded actions, but performance was improving. For trials 200 to 600, performance was roughly equal for both sets of actions. After about 600 trials, the learned actions were taking slightly less time to reach the goal, on average. For trials 500 through 1000, the standard deviation for learned actions was 357.7 and for the hand coded actions it was 544.8, indicating that the time required for the hand coded actions to reach the goal was less consistent than for the learned actions.

Figure 8.8 does not tell the whole story. In some trials, the robot was unable to reach the goal within 7500 time steps. This occurred in a few of the trials where the robot became wedged against a wall, but did not detect a wedged situation due to the relatively coarse thresholding of the sensor information performed by the hand coded actions and the sparse sensor coverage of the Khepera robot. The most common location for this failure to occur is shown in Figure 8.9. At this location in the environment, the sensor coverage of Khepera is inadequate to detect the wall. Reducing the probability of this failure resulted in additional code complexity in the hand coded actions and required careful setting of the

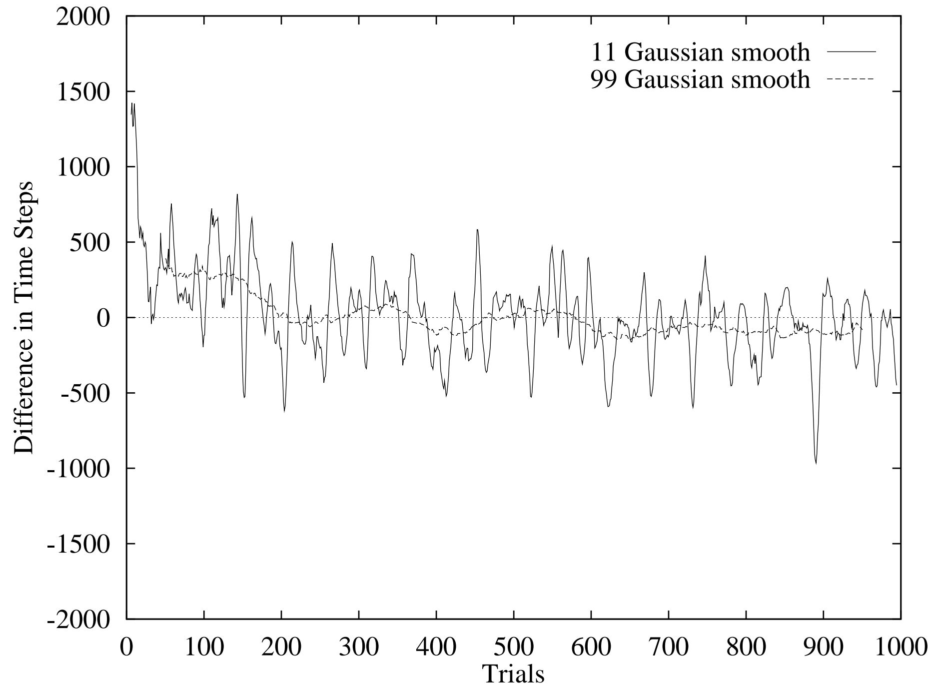


Figure 8.8: Difference between learned actions and hand coded actions in time taken to reach the goal.

Table 8.1: Number of times the robot failed to reach the goal within 7500 time steps.

Action Type	Trials			
	1-250	251-500	501-750	751-1000
Learned	1	0	0	0
Hand Coded	5	4	3	6

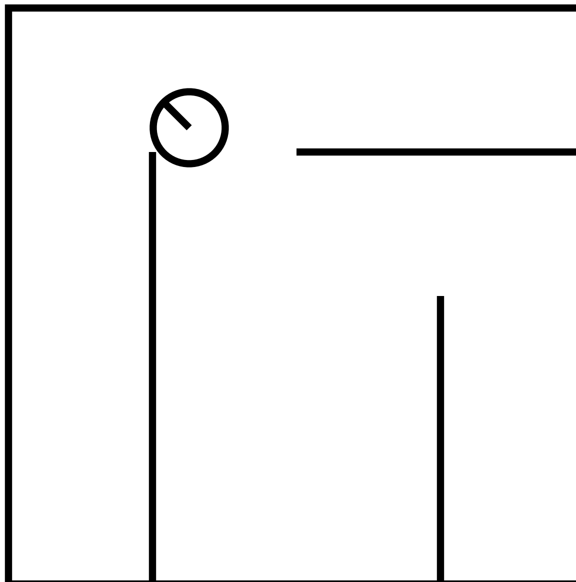


Figure 8.9: The location in the environment where the robot most often became wedged during the generalization tests.

sensor thresholds. No amount of improvement to the hand coded actions would completely eradicate this problem, because there are some situations where the robot is wedged, but the sensors do not detect a wall. The hand coded actions that were used represent a trade-off between code complexity, general performance, and the probability of the robot becoming wedged. When this occurred, the trial was halted and re-started. The failures are not reflected in Figure 8.8. So, Table 8.1 shows the number of failures that occurred for learned and hand coded actions broken into four intervals. The learned actions failed to carry the robot to the goal only once within the first 250 trials, and then adapted to provide higher reliability. The hand coded actions could not adapt, and so their performance did not improve.

Note that the learned actions found a way to deal with the situation where the robot was wedged but the sensory information did not indicate any wall, and thus overcame what appears to be a design limitation in the Khepera robot. The timer mechanism in the POMDP belief state maintenance module is responsible for this adaptation. When the robot becomes wedged and the sensors do not indicate any obstacles, the low level action continues to try to move forward. However, the robot cannot move forward, and no state transition is detected by the belief state maintenance module. The timer runs out and the action is given a negative reward. After receiving several negative rewards, the action adjusts its policy and sends a different set of commands to the effectors, freeing the robot from its wedged position, and leaving the action with a policy that is appropriate for unwedging, but not for general use. This is not a problem, since the action can adapt once again to resume its former function. Of course, the next time the action is used, it may choose the inappropriate effector settings. If so, then it will receive negative rewards until it re-learns its normal operation. Thus, the system is able to trade off some efficiency in order to improve reliability.

8.6 Experiment 4: Learning a New Action

This experiment tests the hypothesis that the system can generate a new action and use it to improve performance. The algorithm for finding uncovered non-primary transitions was implemented and added to the POMDP planner module. The POMDP model and behavior library generated in the previous experiment were loaded into the simulator and the planner was instructed to create a new action. The planner found the set of uncovered non-primary transitions, created a corresponding transition matrix for the new action, and instructed the behavior library to create a new decision tree based learned action. Training was performed, using the same goal state as was used in the previous experiments, and the performance of the system with the new action was compared to the performance of the system before the new action was added.

Table 8.2 shows the uncovered non-primary transitions that were found by the planner. The table shows considerable regularity, indicating that some of the transitions are valid from several states. Inspection of the table, with knowledge of the state mapping, reveals that there are primarily three types of transition represented. For convenience, these transitions are given the following names:

round the corner: the robot turns while moving forward,

edge over: the robot moves diagonally from the current state to the state on its left or right, and

backup: the robot moves backwards to the state behind it.

When the new action begins to learn, it may learn to round the corner from some states while backing up in others. The transition that is learned in one state does not effect the transition learned in another state unless the two states are indistinguishable based on the sensory information available to the action module. If the states are indistinguishable, then they will learn the same action. The planner identifies some uncovered transitions, and the new action may implement an amalgamation of these uncovered transitions. Exactly which transition will be chosen for each state cannot be predicted, because it depends on what states are visited most often when the policy is executed. When this action has learned a set of transitions, there will be some remaining uncovered transitions and another action can be created to cover them. Since the environment imposes physical limitations on the number of state transitions, there is a corresponding limit to the number of useful actions that can be created. As more actions are added to the system, the number of uncovered state transitions will decline, and eventually all possible actions will be learned.

After creating the new action, the POMDP planner was instructed to generate a policy to take the robot to state 13 from anywhere in the environment. Inspection of the policy revealed that it favored the new action in belief states that were on the borderline between state 17 and state 2. The new action would also be selected in belief states that bordered state 2 and state 7. The new policy was loaded into the policy execution module and used to train the new action. Figure 8.10 shows the performance of the robot while learning the new action with the new policy compared to the performance when executing the three previously learned actions using the previous policy. The new action initially resulted in reduced performance, but after about 350 trials, performance is slightly better than the system without the new action. For the final 150 trials, the three action system took an average of 359 time steps to reach the goal, while the three action system took an average of 400 time steps to reach the goal.

Table 8.2: Uncovered non-primary transitions that were used to create the new action. The numbers along the left side of the table show the current state, while the numbers along the top of the table show the next state. Asterisks denote the uncovered non-primary transitions.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
0	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
1	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
2	.	.	.	.	.	.	*	*	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
4	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
5	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
6	.	.	*	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
7	.	.	.	*	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
8	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
9	.	.	.	.	.	.	.	.	.	.	*	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
10	.	.	.	.	.	.	*	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
11	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
12	.	.	.	.	.	.	.	.	*	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
13	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
14	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
15	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
16	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
17	.	*	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
18	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
19	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
20	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
21	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	*	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
22	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
23	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
24	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
25	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	*	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
26	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
27	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
28	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
29	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
30	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
31	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
32	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
33	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
34	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
35	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
36	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
37	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
38	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.
39	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.
40	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
41	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
42	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
43	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
44	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
45	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
46	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
47	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
48	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
49	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
50	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
51	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
52	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
53	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	*	.	.	.	.	.
54	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	*	.	.	.	.
55	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	*	.	.	.	.
56	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
57	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.
58	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	*	.	.	.	.	.	.	.	*	.
59	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.
60	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
61	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
62	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.
63	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.

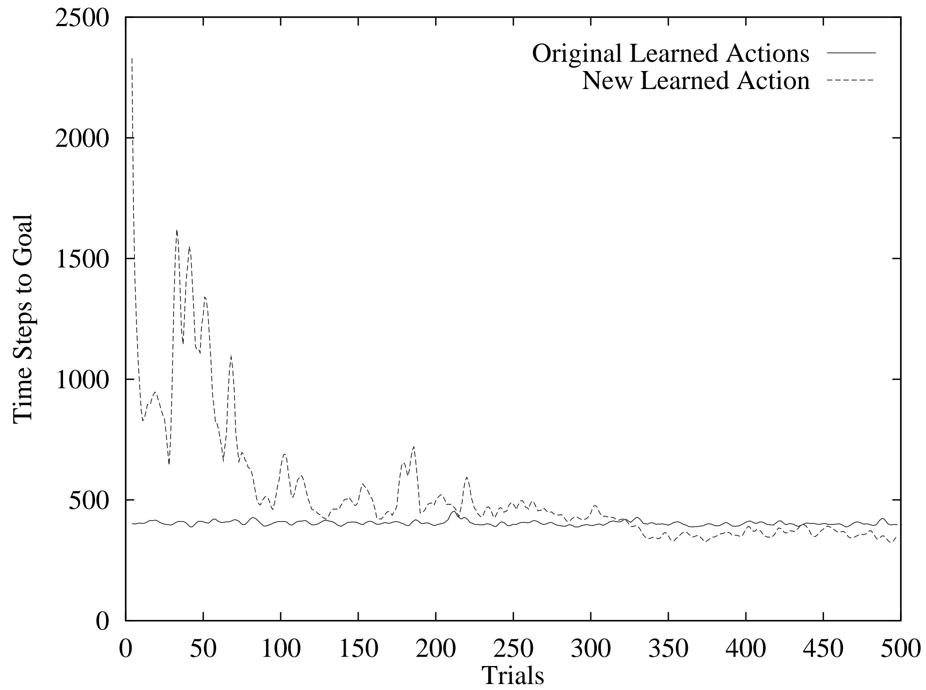


Figure 8.10: Performance with the new action.

1 17 33 49
2 0 18 16 34 32 50 48
3 19 35 51
5 21 37 53
6 4 22 20 38 36 54 52
7 23 39 55
9 25 41 57
10 8 26 24 42 40 58 56
11 27 43 59
13 29 45 61
14 12 30 28 46 44 62 60
15 31 47 63

Figure 8.11: The state assignments in the POMDP model. The environment was divided into sixteen locations with four orientations within each location, for a total of 64 states. State 1 indicates that the robot is in the Northwest corner of the environment and is facing North.

Table 8.3: Primary transitions for the new action after learning.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
0	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
1	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
2	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
3	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
4	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
5	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
6	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
7	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
8	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
9	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
10	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
11	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
12	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
13	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
14	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
15	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
16	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
17	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
18	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
19	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
20	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
21	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
22	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
23	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
24	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
25	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
26	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
27	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
28	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
29	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
30	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
31	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
32	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
33	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
34	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
35	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
36	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
37	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
38	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
39	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.
40	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
41	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
42	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
43	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
44	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
45	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
46	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
47	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
48	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
49	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
50	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
51	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
52	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
53	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.
54	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
55	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
56	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
57	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.
58	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.	.	.
59	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.	.	.	.
60	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
61	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
62	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.	.
63	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	*	.	.	.	.

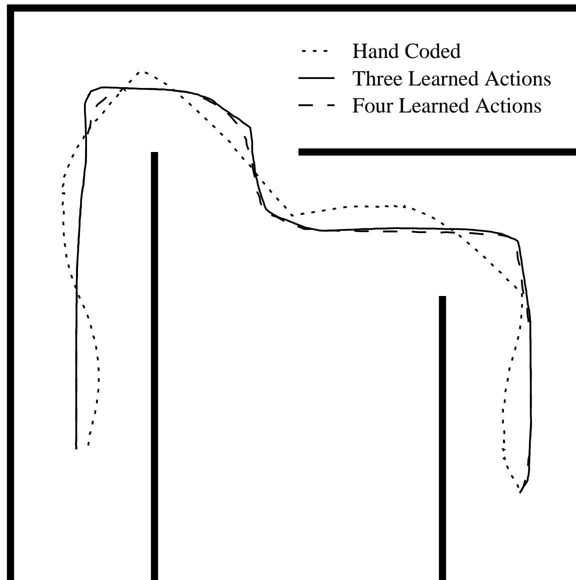


Figure 8.12: The paths followed by the robot with hand coded actions, three learned actions, and four learned actions.

After training was completed, the robot was observed with the simulator in single step mode for several trials. The observations revealed that the robot had learned to round the corner going from state 17 to state 2 and from state 2 to state 6. Figure 8.11 shows the state assignments in POMDP model. Without the new action, the robot had to use the turn left action followed by the go forward action to transition from state 17 to state 18 and then to state 2. Likewise with states 2, 3, and 6. The new action provided a more efficient way to traverse the path, by going directly from state 17 to state 2 and then from state 2 to state 6. The transitions that were learned by the new action are shown in Table 8.3. The transitions from state 17 to state 2 and from state 2 to state 6 are represented as primary transitions in this table.

Figure 8.12 shows the trajectories followed by the robot for three trials. The first trial used hand coded actions, while the other two trials used three and four learned actions, respectively. The learned actions were the ones that resulted from learning in the previous experiments. The trajectory when using hand coded actions exhibits three sharp turns and shows that the hand coded actions do not provide a smooth trajectory. On the first and last segments of the path, the hand coded actions do not maintain an even distance from the walls, and seem to over-steer slightly. The learned actions provide a smooth trajectory and tend to maintain a more even distance from the walls.

The learned actions move in a markedly rectilinear fashion compared with the hand coded actions, and this appears to be less efficient in some parts of the environment. This may be due to the nature of the environment and the fact that the learned actions try to maximize the *average* reward. The hand coded actions come very close to the wall in three places where there is a high probability that the robot will become wedged but where the sensors will not detect a wedged situation. The learned actions tend to avoid coming close to the wall in situations where the robot can become wedged. Although it appears that the learned actions could take a more efficient path, the path that they choose reduces the chance of getting wedged and results in higher average reward. Thus, the learned actions

are more successful at avoiding obstacles in this environment. This may help to explain Table 8.1, which shows that the robot is less likely to become wedged when using learned actions than with hand coded actions on the generalization experiments.

8.7 Conclusions About the Experiments

The experiments in this chapter highlight the importance of continually learning low level actions for providing reliable, adaptive locomotion. The following sections summarize the findings from the experiments.

8.7.1 Replacing Actions

The hand coded actions were implemented in 76 lines of C++ code. The left and right turn actions were only 16 lines each, while the move forward action required 44 lines. The move forward action had to sense and avoid obstacles while still moving forward. It was difficult to hand-code an action that could work appropriately in all situations. While the learned actions required substantially more code to implement, all three actions used exactly the same code, and each action required on the order of 4 Kilobytes of storage for the value function. More actions, such as moving backwards, could be added without new programming and would require on the order of 4 Kilobytes more storage per action, plus some storage for the expanded POMDP model. Although the hand coded actions performed better initially in the learning experiment, the learned actions quickly matched their performance.

8.7.2 Sensor and Effector Failure

The learned actions exhibit reliable behavior in the presence of sensor and effector degradation, compared to the hand coded actions. The learned actions were able to adapt to the changes in sensor and effector response, in contrast to the hand coded actions, which simply failed when the effectors and sensors did not provide the information as expected by the programmer. This experiment, more than any other, shows the degree of contrast between learned and hand coded low level actions.

8.7.3 Generalization

Learned actions were compared to the hand coded to factor out the difference in time needed to reach the goal. The hand coded actions were programmed to operate from any position of the maze. Because the navigation task did not appear to require adaptability, the comparison seemed fair. However, the hand coded actions were brittle. The robot, when using hand coded actions, failed to reach the goal on 1.8% of the trials. The learned actions were much more reliable, failing only one time in 1000 trials. In addition, the single observed failure in the learned actions occurred within the first 250 trials and did not occur again. This suggests that even a simple simulated controller can be more complex than the programmer expects, incurring a performance cost even in a controlled case for lack of adaptability.

Using the POMDP to guide reinforcement learning of low level actions resulted in a system that was more generalizable than the same system using hand coded actions. The

burden of implementing each low level action was transferred from the human to the machine. The hand coded actions performed better initially during generalization testing, but after some adaptation, the learned actions eventually matched and surpassed them in performance. The key to providing high reliability and generalization was the adaptability and continuous learning of the low level actions supported by the POMDP planner.

8.7.4 Learning a New Action

The POMDP planner was augmented to discover new actions and add them to the POMDP model and the action library. The new action was learned on-line in the same way that the original three actions were learned. The new action resulted in marginally better performance for the policy that was being executed. The new action learned to round the corner for two state transitions, and covered that transition for those states. The new action was useful and resulted in improved performance.

8.7.5 Execution Speed

Policy execution is very fast, and effector commands are tightly coupled with sensor data. Figure 7.1 showed the architecture in terms of control loops. The time taken to select the effector settings in the short term loop is $\mathcal{O}(\log N + |\mathcal{A}|)$ where N is the number of nodes in the in the decision tree. For these experiments $N \approx 200$, but this number is domain dependent. The time taken to select an action in the medium term loop is $\mathcal{O}(|\mathcal{V}| \times |\mathcal{S}|)$. For these experiments, $|\mathcal{V}| \approx 100$ and $|\mathcal{S}| = 64$. These numbers are also domain dependent. For the Khepera robot environment, policy execution was fast enough that the simulator could be run at more than 50 times normal speed. The time required to simulate the robot kinematics and sensors was greater than the time taken to execute the POMDP/RL controller. The limiting factor of this architecture is the time taken to generate a POMDP policy.

Chapter 9

Contributions, Questions, and Conclusion

The preceding chapters describe the research that was conducted for this dissertation. The decision tree based function approximation approach to reinforcement learning presented in Chapter 4 provides reliable low level actions. The decision tree based reinforcement learning algorithm goes beyond previous work by inducing the decision tree on line. The parallel algorithm for finding exact solutions to POMDP problems is given in Chapter 6. This algorithm divides the work among several processors to enable the solution of larger POMDP problems than previous algorithms. The POMDP solver was used in the planner to provide goal-directed behavior. Details of how these two major components work together and enable the robot to learn low level actions are presented in Chapter 7. The results from experiments presented in Chapter 8 show that the architecture is reliable, and that new actions can be learned. This chapter discusses the significant contributions of this work and some of the issues that it addresses. Section 9.5 presents some of the remaining issues and gives suggestions for possible future work.

This dissertation presents three significant contributions:

- a goal-directed robot architecture that:
  - is adaptable and reliable, and
  - can learn a new action on demand,
- a parallel method for exact POMDP solution, and
- an improved method for decision tree function approximation in reinforcement learning.

The following sections describe these contributions in more detail.

9.1 Robot Architecture

The primary contribution of this dissertation is a reliable robot control architecture that can learn low level actions. Although some attempts have been made to compile information into reactive modules or learn pre-specified actions (Ring 1994; Ogasawara 1993; Koenig, Goodwin, and Simmons 1996; Liu, Iberall, and Beckey 1989; Singh 1991; Singh 1992b; Dorigo and Colombetti 1994; Mahadevan and Connell 1992), no one has yet demonstrated a system that can generate low level actions as they are discovered. The goal of this research was to demonstrate a system that can decide for itself what low level actions it can learn and then learn them automatically.

The architecture described in this document is a two level system with POMDP planning at the high level and reinforcement learning for actions at the low level. The planning level maintains a POMDP model of the environment and generates policies for achieving goals by selecting appropriate low level actions. When executing a policy, the planning level maintains a belief state that represents the robot's belief about the state of the world. The belief state is used to select an action from the policy. The belief state is also monitored for changes in the most likely state and these changes are used to generate reward signals for the low level actions. The design objectives for this architecture from Section 1.1 were that it be task-directed, reactive, reliable, adaptive, generalizing, and self extending. The following sections describe how these design criteria are met by the architecture.

9.1.1 Task-Directed

A robot is task-directed if it can be assigned an arbitrary task or goal state and can work to achieve that task. In the case of navigation, this usually requires some environment model which the robot uses in planning its actions. The key feature is that the robot can perform planning using the information it already has and does not need to learn the task by trial-and-error.

The POMDP planner can accept goals in the form of immediate rewards. The human investigator can access the POMDP model and specify a reward for reaching a particular state, or for taking a certain action in a particular state. The investigator can even specify multiple rewards to describe complex multi-part tasks. The POMDP planner generates a policy for maximizing the rewards. During policy execution, the robot performs task directed activity to reach the goal state(s). Task-directedness can be inferred to some extent from the results of all of the experiments that were run on the architecture. In all

cases, the robot was given a specific goal state that it was required to reach. The fact that it did so reliably regardless of the initial conditions implies that the robot was actively pursuing the goal.

9.1.2 Reactive

A robotic system is reactive if there is a tight coupling between sensing and acting. This suggests minimal sensor processing, simple decision making, and fast selection of actions or effector settings. Reactive systems may have a planning component to generate plans that are at a high level, leaving the lower levels to reactively deal with the details.

The POMDP/RL architecture is reactive at both levels. The low level actions access raw sensor data and make very fast decisions to select appropriate effector settings. The planning level is also reactive. Sensor readings are used to update the belief state at each time step, and the belief state is used to select an action from the policy. The belief state typically changes more slowly than the raw sensor data. The planning level is reactive in the sense that it can quickly select an appropriate action to execute for any belief state. The actions are reactive in that they can quickly select appropriate effector settings according to sensory input.

9.1.3 Reliable and Adaptive

A system is adaptive if it can change its response to sensory input through learning. Reliability refers to the ability of the robot to achieve its goals with a high probability of success. Adaptability can lead to higher reliability.

The architecture is adaptive at both levels. The low level actions are adaptive to failure in sensors and effectors, while the planning level is adaptive to changes in the actions. Gradual sensor degradation has little effect on the ability of the actions to achieve their state transitions. When effectors fail, the low level actions adapt and try to achieve the highest reward possible given the situation. The POMDP planner updates state transition and observation probabilities to reflect the actual probabilities. When failures occur, the POMDP model changes to reflect the new transition probabilities. Adaptation is a mechanism for providing reliable execution and action generalization in this architecture.

Reliability means that the robot will reach its goal with high probability. The system is considered reliable if it can perform well even with imperfect sensors, noise, and uncertainty about its environment. Imperfections in the sensors include sensor drift and non-linearities. Noise refers to random perturbations of the sensor signals coming from the environment. The robot may be uncertain about its state in the environment. Reliable operation is important because it is not always possible to control the environment or the quality of sensor data.

The experiments in Chapter 8 showed that the architecture was much more reliable with learned actions than with hand coded actions for the simple Khepera maze environment. When effectors and sensors are working correctly, the robot reaches its goal on every trial, regardless of the starting state. When sensors or effectors malfunction, the low level actions adapt to the new conditions and maintain a very high degree of reliability. The robot was able to reach its goal even in the presence of severe sensor and effector malfunction. Although performance was poor in terms of the time that it takes for the robot to reach the goal, reliability was maintained or restored even with severe effector and sensor failure.

The architecture provides reliable action execution through continuous adaptation. The

actions receive a reward for every state transition that is detected by the planning level, and each action is an attempt to maximize the reward it receives. The actions may select any combination of effector settings in order to maximize the reward, and may change their response to sensory information if that information becomes unreliable or may change the commands sent to an effector if it becomes unreliable. This allows the actions to adapt to changes in the environment such as sensor and effector failure, thereby increasing the reliability of the robot.

POMDP navigation has previously been shown to provide reliable navigation (Cassandra, Kaelbling, and Kurien 1996; Simmons and Koenig 1995) in the absence of sensor and effector failure. The POMDP model incorporates a probabilistic representation of the effects of actions and observations. If observations are not available, the robot can use only the information about actions to maintain the belief state. In the case where an action does not perform as expected, the planning level simply updates the belief state and possibly chooses another action. The experiments in Chapter 8 showed that POMDP with hand coded actions could provide some degree of reliability in the presence of hardware failure. Furthermore, for the Khepera maze environment, the combination of POMDP and RL provides even better reliability than could be attained using POMDP planning with hand coded actions.

Both levels of the architecture are reliable even with loss of sensory information. The action level is robust to effector failure, and the planning level is robust to action failure. When combined, they produce a navigation system for the Khepera maze environment that is very reliable, even in the presence of effector and sensor failure.

9.1.4 Generalizing

Learned actions can be generalized to perform state transitions for states that they have not previously encountered. In the simplest case, the new state is indistinguishable from a state that the action has already been trained on. In this case, the action will perform correctly without further training. In states that are very different from any that the action has trained on, there may be some initial loss of performance. However, when the action is used in the new state, it will eventually learn what effector commands are appropriate. This can be inferred from Figure 8.8 which shows the time that the robot requires to reach the goal. Early in the trial, the actions had been used in only a subset of the states of the environment, and had not learned the appropriate response to all situations. As the trial progressed, the actions adapted to new situations and the average time taken to reach the goal decreased. After about 600 trials, the learned actions performed better than the hand coded actions.

9.1.5 Self Extending

Although other systems are capable of learning low level actions, they require that each action be explicitly specified by the human investigator. My approach differs in that the system can find potential actions and create them automatically. The POMDP planner examines the state transition matrices in the POMDP model and identifies any transitions that are not covered by an existing action. These transitions indicate an action that can be learned. The planner adds a new action to implement the transitions that it discovers and begins training. In this way, the robot learned a new action that improved efficiency. Since the robot has more actions to choose from, reliability may also be improved.

9.2 POMDP Solution

Another contribution of this dissertation is the Parallel Restricted Region algorithm. This algorithm can solve POMDP problems that are larger than those accomplished in the literature by distributing the work among a number of processors. The POMDP planner, based on this algorithm, can solve problems in a few days that would require weeks on a single processor.

The POMDP solver that was developed for this research is one of only two existing exact POMDP solvers and is the first to implement a parallel algorithm. The first solver, written by Cassandra (1998a), was completed about a year earlier and is used by many researchers in the POMDP community. My solver has helped to validate the accuracy of the Cassandra solver and has found exact solutions to larger problems than was previously possible.

Although the algorithm can solve larger problems than previous exact solution methods, it still does not scale to a size that is useful in a real-world robot. The most promising avenue for scaling POMDP problems is to use approximate solutions. However, it is still desirable to find exact solutions to large problems in order to verify approximate solution methods. With the parallel POMDP solution algorithm, it is now possible to verify approximate algorithms larger set of more difficult POMDP problems. Thus, the parallel POMDP solution algorithm may provide a benefit to the entire POMDP community by aiding in the development of approximate solution methods.

9.3 Reinforcement Learning

Another contribution of my dissertation research is the algorithm for reinforcement learning using a decision tree as the function approximator. This approach avoids the problems of over-training and forgetting that are common in neural network reinforcement learning systems. The decision tree approach provides learning performance and convergence similar to table lookup based reinforcement learning while avoiding the scaling problems associated with table lookup.

Some of the early experiments on the architecture used table lookup for storing the value function. An augmented POMDP sensor pre-processor was used to provide discretized state information. The decision tree reinforcement learning method was found to perform slightly better than table lookup, particularly in the sensor failure experiments, but careful comparisons were not performed.

9.4 Limitations and Questions for Future Research

As expected, the RL/POMDP combination exhibits reliable execution in the presence of sensor and actuator degradation. For the current test application, I was not inhibited by the known limitations of these methods, specifically, the amount of training required or problem size. However, future work on this project is directed at scaling up the size of the problem and giving the architecture further autonomy. In particular, the architecture must be scaled to work on a robot with more sensors in an environment with hundreds or even thousands of states. Simmons and Koenig (1995) showed that sensor pre-processing can be used to simplify the POMDP model on robots with many sensors. A similar approach

in conjunction with the decision tree based reinforcement learning may provide sufficient scaling for RL as well.

The major limitation of the architecture is the lack of scalability of POMDP policy generation. Even with the parallel restricted region algorithm, generating a policy takes several days of computation. This obviously is not practical for a real robot. To address this limitation, I will be investigating approximate solution algorithms. These algorithms operate much faster than exact solution algorithms, but possibly at the cost of lower performance. Geffner (1998) presented an approach based on real-time dynamic programming (RTDP) and discretized belief space that produced a near optimal solution to a POMDP with 59 states, 17 observations, and 5 actions in less than one second of CPU time. This POMDP problem is similar to the Khepera maze environment. His technique was also applied to a 989 state problem, and produced a policy that is thought to be near optimal, but cannot be verified with an exact solution. Hansen (1998) presented a method for approximate solution by performing heuristic search in policy space. This approach was shown to be significantly faster than restricted region for a set of common POMDP problems from the literature. Some experimentation will be required to assess the trade off between quality and speed of solution, and further work in exact solution methods is necessary to validate the policies generated by approximate solution methods.

Another avenue to faster POMDP policy generation is to use a subset of the POMDP model. Although the robot may need a very large POMDP model in order to carry out all of its tasks, each task may require only a small portion of the POMDP model. An attention focusing mechanism could be constructed to extract needed states and transition probabilities from the complete POMDP model and construct a smaller model for achieving the current task. The POMDP solver would never have to create a policy for the entire model, and planning would be greatly accelerated. Dean et al. (1993) provided one approach to selecting a subset of states from a COMDP model. Their work may shed some light on how the problem can be solved for POMDP planning. Donnart and Meyer (1996) presents a method for both learning a hierarchical COMDP map on-line and using it to focus attention and reduce the number of states that must be considered while planning.

Aside from scaling issues in the POMDP planner, there are several other problems that need to be addressed before the architecture is suitable for use on a hardware robot. One problem with implementing this architecture on a robot is that, during learning, the actions send random signals to the effectors. This is not a problem in a simulated environment, but on a real robot would be dangerous for the robot and anything nearby. One way that this problem may be overcome is by adding a safety module between the action executor and the effectors. The safety module would monitor the signals sent to the effectors, and data coming from the sensors. Whenever the effector signals are deemed dangerous, the safety module would prevent them from reaching the effectors and send a negative reward to the action executor. Thus, the action would learn not to send the dangerous signals. The safety module could be implemented using hand coded rules, a neural network, or some other method to classify the effector signals. Future work will examine some alternative implementations.

For a robot to be adaptive and still provide goal-oriented behavior, it should be able to learn a map of its environment through exploration. It is unreasonable to expect the user to provide a map for the robot, especially in situations where the robot is exploring an environment that is unknown even to the user. Topological map learning is currently an active research area and a few algorithms have been developed (Mataric 1992b; Zimmer and von Puttkamer 1994; Donnart and Meyer 1996). One of the existing algorithms may

be suitable for implementation within the current architecture.

The method used in this research for maintaining and adjusting the transition probabilities in the POMDP model is not expected to work well in larger environments. Better methods exist, such as a modified Viterbi algorithm (Simmons and Koenig 1995). This enhancement could improve the quality of state transition probability estimates, resulting in more reliable belief state maintenance, improved scalability, and better performance. Experimentation in larger environments will help assess whether an improved method for adjusting the transition probabilities in the POMDP model is needed.

In the current implementation of the architecture, actions are created whenever there are uncovered non-primary transitions in the POMDP model. This could result in the creation of many special-purpose actions. The architecture could be improved by limiting the creation of new actions using a heuristic to decide whether or not the current set of uncovered non-primary transitions should be covered by an action.

Although I have concentrated on navigation in a 2-D environment, the architecture may be applicable to manipulator control as well. Manipulators may operate in three dimensions, but the environment may still be described with a POMDP model and reinforcement learning has been shown to work in three dimensional domains (Liu, Iberall, and Beckey 1989). If the problems of scaling in two dimensions can be overcome, then this architecture should also be evaluated for manipulator control for tasks such as grasping and moving objects.

9.5 Conclusion

This dissertation presents a robot navigation architecture that can extend its set of low level actions. Although other systems have demonstrated the ability to learn low level actions, my work goes one step further and also learns the specification of the low level actions. The architecture is reliable in part because it incorporates adaptation. New actions can be learned to perform specific state transitions and generalize well to perform other state transitions.

The approach to learning low level actions is different from any previous approach, and the experimental results indicate it performs well in a small domain. The major hurdles to be overcome are in scaling the action and planning levels to work on a larger robot in a more complex environment. This work should be seen as the first step in developing a robot that can learn new actions for itself and adapt to changes in the environment.

References

Anderson, C. W. (1986). *Learning and problem solving with connectionist representations.* Ph. D. thesis, Computer and Information Science Department, University of Massachusetts, Amherst, MA.

Anderson, C. W. (1989). Strategy learning with multilayer connectionist representations. In *Proceedings of the Fourth International Workshop on Machine Learning*, pp. 103–114.

Anderson, C. W. (1993). Q-learning with hidden-unit restarting. In *Advances in Neural Information Processing Systems 5*, pp. 81–88.

Anderson, C. W. and Z. Hong (1994). Reinforcement learning with modular neural networks for control the. In *Proceedings of NNACIP'94, the IEEE International Workshop on Neural Networks Applied to Control and Image Processing.*

Anderson, J. R. (1983). *The Architecture of Cognition.* Cambridge, MA: Harvard University Press.

Arbib, M. A. (1992). Schema theory. In S. Shapiro (Ed.), *The Encyclopedia of Artificial Intelligence* (2nd ed.)., pp. 1427–1443. New York, NY: Wiley Interscience.

Arkin, R. C. (1998). *Behavior-Based Robotics.* Cambridge, Massachusetts: MIT Press.

Bagnell, J. A., K. L. Doty, and A. A. Arroyo (1998). Comparison of reinforcement learning techniques for automatic behavior programming. In *Carnegie Mellon Conference on Automated Learning and Discovery.*

Baird, L. (1995). Residual algorithms: Reinforcement learning with function approximation. In *Proceedings of the 12th International Conference on Machine Learning*, San Mateo, CA, pp. 30–37. Morgan Kaufmann.

Barto, A., R. Sutton, and C. Anderson (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics 13*, 834–846.

Barto, A., R. Sutton, and C. Watkins (1989). Learning and sequential decision making. Technical report, COINS Technical Report 89-95, Dept. of Computer and Information Science, University of Massachusetts.

Barto, A. G., S. J. Bradtke, and S. P. Singh (1993, January). Learning to act using real-time dynamic programming. *Artificial Intelligence Journal special issue on Computational Theories of Interaction and Agency 72*(1), 81–138.

Borisyuk, R., J. Wickens, and R. Köetter (1994). Reinforcement learning in a network model of the basal ganglia. In R. Trappl (Ed.), *Cybernetics and Systems '94*, Singapore, pp. 1681–1686. World Scientific Publishing.

Boyan, J. A. and A. W. Moore (1995). Generalization in reinforcement learning: Safely approximating the value function. In G. Tesauro, D. Touretsky, and T. K. Leen (Eds.), *Advances in Neural Information Processing Systems 7*, Cambridge, MA. MIT Press.

Breiman, L., J. H. Friedman, R. A. Olshen, and C. J. Stone (1984). Classification and regression trees. Technical report, Wadsworth International, Monterey, CA.

Brooks, R. A. (1986, March). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation 2*(1), 14–23.

Brooks, R. A. (1991a). How to build complete creatures rather than isolated cognitive simulators. In K. VanLehn (Ed.), *Architectures for Intelligence: The Twenty-second Carnegie Mellon Symposium on Cognition*, Chapter 8, pp. 225–239. Hillsdale, New Jersey: Lawrence Erlbaum Associates.

Brooks, R. A. (1991b, August). Integrated systems based on behaviors. *SIGART Bulletin 2*(4), 46–50.

Brooks, R. A. (1991c, January). Intelligence without representation. *AI Journal 47*(1-3), 139–160.

Bruner, J. S. (1973). Organization of early skilled action. *Child Development 44*, 1–11.

Bullock, D., S. Grossberg, and F. H. Guenther (1992). A self-organizing neural network model for redundant sensory-motor control, motor equivalence, and tool use. In *International Joint Conference on Neural Networks*, Baltimore, pp. 91–96. IEEE.

Calvert, J. and W. Voxman (1989). *Linear Programming.* Harcourt Brace Jovanovich.

Cassandra, A., M. L. Littman, and N. L. Zhang (1997). Incremental pruning: A simple, fast, exact algorithm for partially observable markov decision processes. In *Proceedings of the Thirteenth Annual Conference on Uncertainty in Artificial Intelligence.*

Cassandra, A. R. (1998a, May). *Exact and Approximate Solutions for Partially Observable Markov Decision Processes.* Ph. D. thesis, Brown University, Providence, RI.

Cassandra, A. R. (1998b, October). A survey of POMDP applications. In M. Littmann (Ed.), *Working Notes: AAAI Fall Symposium on Planning with Partially Observable Markov Decision Processes*, pp. 17–24. AAAI.

Cassandra, A. R., L. P. Kaelbling, and J. A. Kurien (1996). Acting under uncertainty: Discrete bayesian models for mobile robot navigation. In *Proceedings of the IEEE/Robotics Society of Japan Conference on Intelligent Robots and Systems.* IEEE.

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

Cassandra, A. R., M. L. Littman, and L. P. Kaelbling (1996). Efficient dynamic-programming updates in partially observable Markov decision processes. Technical Report CS-95-19, Brown University, Providence, RI.

Chapman, D. and L. P. Kaelbling (1991). Input generalization in delayed reinforcement learning: An algorithm and performance comparisons. In J. Mylopoulos and R. Reiter. (Eds.), *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, San Mateo, Ca., pp. 726–731. Morgan Kaufmann.

Cheng, H.-T. (1988). *Algorithms for Partially Observable Markov Decision Processes.* PhD dissertation, University of British Columbia, British Columbia, Canada.

Chrisman, L. (1991, July). Reinforcement learning with perceptual aliasing: The perceptual distinctions approach. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pp. 183–188. AAAI: AAAI Press/MIT Press.

Cohen, P. R. (1995). *Empirical Methods for Artificial Intelligence.* MIT Press.

Connell, J. (1989, Month). A colony architecture for an artificial creature. Technical Report 1151, MIT AI Laboratory, Cambridge, MA.

Coren, S., C. Porac, and L. M. Ward (1984). *Sensation and Perception* (Second Edition ed.). Orlando: Academic Press, Inc.

Dayan, P. (1992). The convergence of TD(λ) for general λ. *Machine Learning 8*, 341–362.

Dayan, P. and G. E. Hinton (1993). Feudal reinforcement learning. In S. J. Hanson, J. D. Cowan, and C. L. Giles (Eds.), *Advances in Neural Information Processing Systems 5*, San Mateo, CA, pp. 271–278. Morgan Kaufmann.

Dean, T., L. P. Kaelbling, J. Kerman, and A. Nicholson (1993, July). Planning with deadlines in stochastic domains. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, Washington, D.C., pp. 574–579. AAAI: AAAI Press.

Dean, T. and M. Wellman (1991). *Planning and Control.* San Mateo, CA: Morgan Kaufmann.

Donnart, J.-Y. and J.-A. Meyer (1996, Summer). Hierarchical-map building and self-positioning with monalysa. *Adaptive Behavior 5*(1), 29–74.

Dorigo, M. and M. Colombetti (1994, December). Robot shaping: developing autonomous agents through learning. *Artificial Intelligence 71*(2), 321–370.

Dorigo, M. and M. Colombetti (1998). *Robot Shaping: an Experiment in Behavior Engineering.* Cambridge, Massachusetts: MIT Press.

Doshi, R. S., R. S. Desai, R. Lam, and J. E. White (1988, December 5–7). Integration of artificial intelligence planning and robotic systems with AIROBIC. In *Proceedings of the IMACS Conference on Expert Systems for Numerical Computing*, Purdue University, West Lafayette, Indiana.

Dubrawski, A. and J. L. Crowley (1994, April). Learning locomotion reflexes: A self-supervised neural system for a mobile robot. *Robotics and Autonomous Systems 12*(3-4), 133–142.

Feiten, W., U. Wienkop, A. Huster, and G. Lawitsky (1994). Simulation in the design of an autonomous mobile robot. In R. Trappe (Ed.), *Cybernetics and Systems '94*, pp. 1499–1506. Singapore: World Scientific Publishing.

Ferrell, C. (1994). Failure recognition and fault tolerance of an autonomous robot. *Adaptive Behavior, 2*(4), 375–398.

Firby, R. J. (1989). *Adaptive Execution in Complex Dynamic Worlds*. Ph. D. thesis, Yale University.

Firby, R. J. (1995, March). Lessons learned from the animate agent project (so far). In *AAAI Spring Symposium on Lessons Learned for Implemented Software Architectures for Physical Agents*, Palo Alto, CA, pp. 92–96.

Fischer, K. W. and A. Lazerson (1994). *Human Development From Conception Through Adolescence*. New York: W. H. Freeman and Company.

Fitts, P. and M. Posner (1967). *Human Performance*. Monterey, CA: Brooks/Cole.

Geffner, H. (1998, October). Solving large POMDPs using real time dynamic programming. In M. Littmann (Ed.), *Working Notes: AAAI Fall Symposium on Planning with Partially Observable Markov Decision Processes*, pp. 61–68. AAAI.

Geffner, H. and B. Bonet (1998). High-level planning and control with incomplete information using POMDPs. In *AIPS-98 Workshop on Integrating Planning, Scheduling and Execution in Dynamic and Uncertain Environments*.

Georgeff, M. P. and F. F. Ingrand (1989a, November). Decision-making in an embedded reasoning system. Technical Report 04, Australian Artificial Intelligence Institute, Melbourne, Australia.

Georgeff, M. P. and F. F. Ingrand (1989b, November). Monitoring and control of spacecraft systems using procedural reasoning. Technical Report 03, Australian Artificial Intelligence Institute, Melbourne, Australia.

Goldsmith, J. and M. Mundhenk (1998). Complexity issues in markov decision processes. In *Proceedings of the IEEE Conference on Computational Complexity*. IEEE.

Gordon, G. J. (1995, January). Stable function approximation in dynamic programming. Technical Report CMU-CS-95-103, Carnegie Mellon University, Computer Science Department, Pittsburgh.

Gruau, F. and K. Quatamaran (1996). Cellular encoding for interactive evolutionary robotics. Technical Report CS-R9629, CWI, Amsterdam.

Hansen, E. A. (1998, July). Solving POMDPs by searching in policy space. In *Proceedings of the Fourteenth International Conference on Uncertainty in Artificial Intelligence*, Madison, Wisconsin.

Hasemann, J.-M. (1995, Oct). Robot control architectures – application requirements, approaches, and technologies. In *Proceedings of the SPIE Intelligent Robots and Computer Vision XIV: Algorithms, Techniques, Active Vision, Material Handling*, Philadelphia, pennsylvania. SPIE.

Hauskrecht, M. (1997). Incremental methods for computing bounds in partially observable markov decision processes. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence*, pp. 734–739. AAAI.

Hayes-Roth, B., K. Pfleger, P. Lalanda, P. Morignot, and M. Balabanovic (1995, April). A domain-specific software architecture for adaptive intelligent systems. *IEEE Transactions on Software Engineering 21*(4), 288–301.

Hexmoor, H., J. Lammens, G. Caicedo, and S. C. Shapiro (1993, April). Behavior based AI, cognitive processes, and emergent behaviors in autonomous agents. Technical

Report 93-15, State University of New York at Buffalo, Department of Computer Science, 226 Bell Hall, Buffalo, New York 14260.

Jaakkola, T., M. Jordan, and S. Singh (1994). On the convergence of stochastic iterative dynamic programming algorithms. *Neural Computation 6*(6), 1185–1201.

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

Keil, F. (1989). *Concepts, Kinds, and Cognitive Development.* Cambridge, MA: MIT Press.

Koenig, S., R. Goodwin, and R. G. Simmons (1996). Robot navigation with Markov models: A framework for path planning and learning with limited computational resources. *Lecture Notes in Computer Science 1093*, 322.

Koenig, S. and R. G. Simmons (1994). Risk-sensitive planning with probabilistic decision graphs. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning*, pp. 363–373.

Kontoravdis, D., A. Likas, and A. Stafylopatis (1992). Collision-free movement of an autonomous vehicle using reinforcement learning. In B. Neumann (Ed.), *Proceedings of the Tenth European Conference on Artificial Intelligence*, Chichester, UK, pp. 666–670. Wiley.

Kretchmar, R. M. and C. W. Anderson (1997, June). Comparison of cmacs and radial basis functions for local function approximators in reinforcement learning. In *Proceedings of the International Conference on Neural Networks*, IEEE Services Center 445 Hoes Lane P.O. BOX 1331 Picataway, NJ 08855-1331, pp. 834–7. IEEE: IEEE Press.

Krishnaswamy, G., M. H. Ang Jr., and G. B. Andeen (1991). Structured neural-network approach to robot motion control. In *International Joint Conference on Neural Networks*, pp. 1059–1066. IEEE.

Kushmerick, N., S. Hanks, and D. S. Weld (1995). An algorithm for probabilistic planning. *Artificial Intelligence 76*(1–2), 239–286.

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

Littman, M. L. (1994, December). The witness algorithm: Solving partially observable markov decision processes. Technical Report CS-94-40, Brown University, Department of Computer Science, Providence, RI.

Littman, M. L. (1996, March). *Algorithms for Sequential Decision Making.* Ph. D. thesis, Brown University, Department of Computer Science, Providence, RI.

Liu, H., T. Iberall, and G. A. Beckey (1989, July). Neural network architecture for robot hand control. In *Proceedings of IEEE International Conference on Neural Networks.* IEEE.

Luke, S. (1998, July). Genetic programming produced competitive soccer softbot teams for robocup97. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo (Eds.), *Proceedings of the Third Annual Genetic Programming Conference (GP98)*, San Francisco, pp. 204–222. Morgan Kaufmann.

Luke, S. and L. Spector (1996). Evolving teamwork and coordination with genetic programming. In John Koza et al (Ed.), *Genetic Programming 1996: Proceedings of the First Annual Conference*, Cambridge, pp. 141–149. MIT Press.

Mahadevan, S. and J. Connell (1992). Automatic programming of behaviour-based robots using reinforcement learning. *Artificial Intelligence 55*(2/3), 311–365.

Mataric, M. J. (1992a, May). Behavior-based systems: Main properties and implications. In *Proceedings, IEEE International Conference on Robotics and Automation, Workshop on Architectures for Intelligent Control Systems*, Nice, France, pp. 46–54. IEEE.

Mataric, M. J. (1992b, June). Integration of representation into goal-driven behavior-based robots. *IEEE Transactions on Robotics and Automation 8*(3), 304–312.

McCallum, A. K. (1994). First results with instance-based state identification for reinforcement learning. Technical Report 502, University of Rochester.

McCallum, A. K. (1995, December). *Reinforcement Learning with Selective Perception and Hidden State*. Ph. D. thesis, University of Rochester, Computer Science Department.

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

Michel, O. (1995). Khepera simulator. Available for download from `http://diwww.epfl.ch/lami/team/michel/khep-sim/`.

Millán, J. D. R. and C. Torras (1992). A reinforcement connectionist approach to robot path finding in non-maze-like environments. *Machine Learning, Special Issue on Reinforcement Learning 8*(3/4), 139–173.

Monahan, G. E. (1982). A survey of partially observable Markov decision processes. *Management Science 28*(1), 1–16.

Moore, A. W. (1991). Variable resolution dynamic programming: Efficiently learning action maps in multivariate real-valued spaces. In *Proceedings of the Eighth International Machine Learning Workshop*.

Muller, R. U., M. Stead, and J. Pach (1996). The hippocampus as a cognitive graph. *Journal of General Physiology 107*, 663–694.

Murthy, K. V. S. (1996). *On Growing Better Decision Trees from Data*. Ph. D. thesis, Johns Hopkins University, Baltimore, Maryland.

Murthy, S. K., S. Kasif, and S. Salzberg (1994). A system for induction of oblique decision trees. *JAIR 2*, 1–33.

Nehmzow, U. (1994, November). Autonomous acquisition of sensor-motor couplings in robots. Technical Report UMCS-94-11-1, Department of Computer Science, University of Manchester, Oxford Road, Manchester, UK.

Nilsson, N. J. (1984, April). Shakey the robot. Technical Report 323, AI Center, SRI International, 333 Ravenswood Ave., Menlo Park, CA 94025.

Nourbakhsh, I., R. Powers, and S. Birchfield Summer (1995). Dervish: An office-navigating robot. *AI Magazine 16*(2), 53–60.

Ogasawara, G. H. (1993). *RALPH-MEA: A Real-Time, Decision-Theoretic Agent Architecture*. Ph. D. thesis, University of California, Berkeley, California 94720.

O'Keefe, J. and J. Dostrovsky (1971). The hippocampus as a spatial map: Preliminary evidence from unit activity in the freely moving rat. *Experimental Brain Research 34*, 171–175.

O'Keefe, J. and L. Nadel (1978). *The Hippocampus as a Cognitive Map*. Oxford, UK: Clarendon Press.

Papadimitriou, C. H. and J. N. Tsitsiklis (1987, August). The complexity of Markov decision processes. *Mathematics of Operations Research 12*(3), 441–450.

Parr, R. and S. Russell (1995, August). Approximating optimal policies for partially observable stochastic domains. In *Proceedings of the 1995 International Joint Conference on Artificial Intelligence*, Montreal, Quebec.

Platzman, L. (1998, October). Notes for presentation at the AAAI symposium. In M. Littmann (Ed.), *Working Notes: AAAI Fall Symposium on Planning with Partially Observable Markov Decision Processes*, pp. 367–369. AAAI.

Poo, A., M. Ang Jr., C. Teo, and Q. Li (1992). Performance of a neuro-model-based robot controller: adaptability and noise rejection. *Intelligent Systems Engineering 1*(1), 50–62.

Poole, D. (1996, August). A framework for decision-theoretic planning i: Combining the situation calculus, conditional plans, probability and utility. In *Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence*, Portland, Oregon.

Prokhorov, D. V. and D. C. Wunsch II (1997, October). Convergence of critic-based training. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, Orlando, FL.

Pyeatt, L. D. and A. E. Howe (1998, March). Reinforcement learning for coordinated reactive control. In *Fourth World Congress on Expert Systems*.

Pyeatt, L. D. and A. E. Howe (1999, May). Integrating POMDP and reinforcement learning for a two layer simulated robot architecture. In *Third International Conference on Autonomous Agents*, Seattle, Washington.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning 1*(1), 81–106.

Ring, M. B. (1991, June). Incremental development of complex behaviors through automatic construction of sensory-motor hierarchies. In L. A. Birnbaum and G. C. Collins (Eds.), *Machine Learning: Proceedings of the Eighth International Workshop (ML91)*, pp. 343–347. Morgan Kaufmann Publishers.

Ring, M. B. (1993a). Learning sequential tasks by incrementally adding higher orders. In C. L. Giles, S. J. Hanson, and J. D. Cowan (Eds.), *Advances in Neural Information Processing Systems 5*, San Mateo, California, pp. 115–122. Morgan Kaufmann Publishers.

Ring, M. B. (1993b, January). Sequence learning with incremental higher-order neural networks. Technical Report AI 93–193, Artificial Intelligence Laboratory, University of Texas at Austin.

Ring, M. B. (1993c). Two methods for hierarchy learning in reinforcement environments. In J. A. Meyer, H. Roitblat, and S. Wilson (Eds.), *From Animals to Animats 2: Pro-*

ceedings of the Second International Conference on Simulation of Adaptive Behavior, pp. 148–155. MIT Press.

Ring, M. B. (1994, August). *Continual Learning in Reinforcement Environments*. Ph. D. thesis, University of Texas at Austin, Austin, Texas.

Rosenschein, S. and L. P. Kaelbling (1987, April). The synthesis of digital machines with provable epistemic properties. Technical Report Technical Note no. 412, SRI International, Menlo Park, CA.

Roy, N., W. Burgard, D. Fox, and S. Thrun (1998, October). Coastal navigation - robot motion with uncertainty. In M. Littman and T. Cassandra (Eds.), *AAAI 1998 Fall Symposium Series – Planning with Partially Observable Markov Decision Processes: Working Notes*, Orlando, Florida, pp. 135–140. AAAI.

Sabes, P. (1993). Approximating Q-values with basis function representations. In *Proceedings of the Fourth Connectionist Models Summer School*, Hillsdale, NJ. Lawrence Erlbaum.

Saffiotti, A. (1993). Some notes on the integration of planning and reactivity in autonomous mobile robots. In *Proceedings of the 1993 AAAI Symposium*.

Sammut, C. (1996). Automatic construction of reactive control systems using symbolic machine learning. *The Knowledge Engineering Review 11*(1), 27–42.

Samuels, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development 3*(4), 210–229.

Schmajuk, N. A. and A. D. Thieme (1992). Purposive behavior and cognitive mapping: A neural network model. *Biological Cybernetics 67*, 165–174.

Simmons, R., R. Goodwin, K. Z. Haigh, S. Koenig, and J. O'Sullivan (1997, February). A layered architecture for office delivery robots. In W. L. Johnson (Ed.), *Proceedings of First International Conference on Autonomous Agents*, Marina del Rey, CA, pp. 245–252. ACM Press, New York, NY.

Simmons, R. G. and S. Koenig (1995). Probabilistic navigation in partially observable environments. In *Fourteenth International Joint Conference on Artificial Intelligence*, Montreal, Canada, pp. 1080–1087. IJCAI: Morgan Kaufmann.

Singh, S. P. (1991). The efficient learning of multiple task sequences. In *Advances in Neural Information Processing Systems 3*, San Mateo, CA. Morgan Kaufman.

Singh, S. P. (1992a). Scaling reinforcement learning algorithms by learning variable temporal resolution models. In *Proceedings of the Ninth Machine Learning Conference*.

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

Singh, S. P. (1993). Soft dynamic programming algorithms: Convergence proofs. Technical Report CLNL-93, University of Massachusetts.

Singh, S. P., A. G. Barto, R. Grupen, and C. Connoly (1994). Robust reinforcement learning in motion planning. In J. D. Cowan, G. Tesauro, and J. Alspector (Eds.), *Advances in Neural Information Processing Systems 6*, San Mateo, CA, pp. 655–662. Morgan Kaufmann.

Singh, S. P. and R. Ye (1994). An upper bound on the loss from approximate optimal-value functions. *Machine Learning 16*, 227–233.

Smallwood, R. D. and E. J. Sondik (1973). The optimal control of partially observable Markov processes over a finite horizon. *Operations Research 21*, 1071–1088.

Smithers, T. and C. A. Malcolm (1987, May). A behavioural approach to robot task planning and offline programming. In T. Martin (Ed.), *Proceedings of the International Advanced Robotics Programme First Workshop on Manipulators, Sensors and Steps Towards Mobility*, Karlsruhe, Kernforschungszentrum Karlsruhe GmbH. Also available as DAI Research Paper No 306, Department of Artificial Intelligence, Edinburgh University.

Smithers, T. and C. A. Malcolm (1988). Programming assembly robots in terms of task achieving behavioural modules: First experimental results. DAI Research Paper 410, Edinburgh University, Department of Artificial Intelligence, Edinburgh, Scotland.

Sondik, E. J. (1971). *The Optimal Control of Partially Observable Markov Processes.* Ph. D. thesis, Stanford University.

Sun, R. (1995). Robust reasoning: integrating rule-based and similarity-based reasoning. *Artificial Intelligence 75*(2), 241–296.

Sutton, R. S. (1984). *Temporal Credit Assignment in Reinforcement Learning.* Ph. D. thesis, Dept. of Computer and Information Science, University of Massachusetts.

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

Sutton, R. S. (1990). Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, San Mateo, CA, pp. 216–224. Morgan Kaufmann.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In *Advances in Neural Information Processing Systems 8*, Cambridge, MA, pp. 1038–1044. MIT Press.

Sutton, R. S. and A. G. Barto (1997a). *An Introduction to Reinforcement Learning*, Chapter 3, pp. 258. Cambridge, MA: MIT Press.

Sutton, R. S. and A. G. Barto (1997b). *An Introduction to Reinforcement Learning.* Cambridge, MA: MIT Press.

Tadepalli, P. and D. Ok (1994). H-learning: A reinforcement learning method to optimize undiscounted reward. Technical Report 94-30-01, Oregon State University.

Tadepalli, P. and D. Ok (1998). Model-based average reward reinforcement learning. *Artificial Intelligence 100*, 177–224.

Tani, J. and N. Fukumura (1994). Learning goal-directed sensory-based navigation of a mobile robot. *Neural Networks 7*(3), 553–563.

Taube, J. S., R. U. Muller, and J. B. Ranck (1990). Head-direction cells recorded from the postsubiculum in freely moving rats: Description and quantitative analysis. *Journal of Neuroscience 10*(2), 420–435.

Tesauro, G. (1990). Neurogammon: A neural network backgammon program. In *Proceedings of the Third International Joint Conference on Neural Networks*, pp. 33–39.

Thrun, S. and A. Schwartz (1993, December). Issues in using function approximation for reinforcement learning. In *Proceedings of the Fourth Connectionist Models Summer School.* Hillsdale, NJ: Lawrence Erlbaum.

Timin, M. E. (1995). Robot automobile racing simulator (RARS). Anonymous ftp `ftp.ijs.com:/rars`.

Torras, C. (1994, August8–12). Neural learning for robot control. In A. G. Cohn (Ed.), *Proceedings of the Eleventh European Conference on Artificial Intelligence*, Chichester, pp. 814–819. ECAI: John Wiley and Sons.

Trullier, O. and J. A. Meyer (1997). Place sequence learning for navigation. In W. Gerstner, A. Germond, M. Hasler, and J. D. Nicoud (Eds.), *Proceedings of International Conference on Artificial Neural Networks*, pp. 757–762. Springer-Verlag.

Tsitsiklis, J. N. (1994). Asynchronous stochastic approximation and q-learning. *Machine Learning 16*, 185–202.

Tsitsiklis, J. N. and B. Van Roy (1997, May). An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control 42*(5), 674–690.

Uther, W. T. B. and M. M. Veloso (1998, July). Tree based discretization for continuous state space reinforcement learning. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-98)*, Madison, WI. AAAI.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards.* Ph. D. thesis, Kings College, Cambridge, UK.

Watkins, C. J. C. H. and P. Dayan (1992). Q-learning. *Machine Learning 8*(3), 279–292.

White III, C. C. (1991). A survey of solution techniques for the partially observed Markov decision process. *Annals of Operations Research 32*, 215–230.

Wilkins, D. E. (1984). Domain-independent planning: Representation and plan generation. *Artificial Intelligence 22*(3), 269–301.

Wilkins, D. E., K. L. Myers, J. D. Lowrance, and L. P. Wesley (1995). Planning and reacting in uncertain and dynamic environments. *Journal of Experimental and Theoretical AI 7*(1), 197–227.

Witten, I. H. (1977). An adaptive optimal controller for discrete-time Markov environments. *Information and Control 34*, 286–295.

Zelek, J. S. and M. D. Levine (1998). SPOTT: a predictable and scalable architecture for autonomous mobile robot control. Submitted to IEEE Transactions on Robotics and Automation.

Zhang, N. L. and S. S. Lee (1998, July). Planning with partially observable markov decision processes: Advances in exact solution methods. In *Proceedings of the Fourteenth International Conference on Uncertainty in Artificial Intelligence*, Madison, Wisconsin.

Zhang, N. L. and W. Liu (1996). Planning in stochastic domains: Problem characteristics and approximation. Technical Report HKUST-CS96-31, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong.

Zimmer, U. R. and E. von Puttkamer (1994, September). Comparing world-modeling strategies for autonomous mobile robots. In *Proceedings of IWK'94*, Ilmenau, Germany.

Index

A

action, 6
  discovering, *see* finding new actions
  execution, 101
  failure, 97
  greedy, 33
  hand coded, 100, 112, 126
  library, 102
  low level, 100
  representation for planning, 103
  stable, 105
adaptation, 120
architecture, 12
  behavior-based, 14, 31
  layered, 3
  POMDP/RL, 97
    control loops, 99
  sense-model-plan-act, 12
  single layer, 4
  subsumption, 13
average steps to goal, 115

B

behavior, 6, 26
  based robotics, *see* architecture
  reinforcement learning, 28, 31
belief state, 22, 61, 62
  discretization, 22
  maintaining, 62
  maintenance equation, 64
Bellman optimality equation, 33

C

cross sum, 83

D

decision tree
  decision node, 53
  leaf node, 53
  reinforcement learning, 52
  splitting, 54
dominate, 72, 85

E

effector failure, 117
expected state, 104

F

failures during trial, 115
finding new actions, 107
  physical limit, 121

K

Khepera, 1, 110
  design limitation, 120
  POMDP model, 111

L

linear program, 87
  solver, 93
local support, 51

M

Markov decision process, 20
  completely observable, 20
  partially observable, 22, 61
    action selection, 64
    approximate solution, 80, 82
    computational complexity, 94
    continuous state space, 64
    dynamic programming, 87

incremental pruning, 87
observations, 104
parallel restricted region, 89
planning, 102
restricted region, 88
scaling, 132
state transition, 102
planning, 20–22, 35
most likely state, 104

O

obstacle avoidance, 13, 95
through reinforcement learning, 30, 101, 126

P

parallel restricted region, *see* Markov decision process
planning
as a Markov decision process, *see* Markov decision process
plan space, 19
reactive, 15
state space, 19, 35
policy, 20
ϵ-greedy , 33
behavior, 33
estimation, 33
POMDP, 22, 62, 82
reinforcement learning, 32
optimal, 32
pruning, 80
purge, 84

Q

Q-learning, *see* reinforcement learning

R

reinforcement learning, 31
actor-critic, 34
function approximation, 36, 50, 51
decision tree, 52
neural network, 52
table lookup, 51
model-based, 7, 35
obstacle avoidance, *see* obstacle avoidance
off-policy, 33
on-policy, 33
Q-learning, 34
real-time dynamic programming, 35
Sarsa, 33
restricted region, *see* Markov decision process
reward
discounted, 32
generating, 104

S

sensor failure, 115
simulation
advantages, 109
speed, 127
Sondik region, 74
state transition, 104

T

temporal differencing, 33
n-step prediction, 35
TD(λ), 35, 45
topological map, 7, 96, 103
transition
non-primary, 106
covered, 107
uncovered, 106
primary, 106
transitions
non-primary
uncovered, 121

V

value function
POMDP, 62, 64
representation, 65
reinforcement learning, 32
optimal, 32
value iteration, 20, 83
COMDP, 20
horizon, 66

infinite horizon, 83
POMDP, 64, 72
algorithm, 72

W

wedged, 115
witness
point, 78
region, 84