Learning Adaptive Reactive Agents

A Thesis
Presented to
The Academic Faculty

by

Juan Carlos Santamaría

In Partial Fulfillment
of the Requirements for the Degree of
Doctor of Philosophy in Computer Science

Georgia Institute of Technology
October 1997

Learning Adaptive Reactive Agents

Approved:

Ashwin Ram, Chairman
Colege of Computing

Christopher G. Atkeson
Colege of Computing

Ronald C. Arkin
Colege of Computing

Alex Kirlik
Industrial and Systems Engineering

Janet L. Kolodner
Colege of Computing

Date Approved

Dedication

– To my parents Diana and Fredy –
– To my brother Alejandro –
– To my wife Carolina –

SUMMARY

An autonomous agent is an intelligent system that has an ongoing interaction with a dynamic external world. It can perceive and act on the world through a set of limited sensors and effectors. Its most important characteristic is that it is forced to make decisions sequentially, one after another, during its entire "life". The main objective of this dissertation is to study algorithms by which an autonomous agents can learn, using their own experience, to perform sequential decision-making efficiently and autonomously. The dissertation describes a framework for studying autonomous sequential decision-making consisting of three main elements: the agent, the environment, and the task. The agent attempts to control the environment by perceiving the environment and choosing actions in a sequential fashion. The environment is a dynamic system characterized by a state and its dynamics, a function that describes the evolution of the state given the agent's actions. A task is a declarative description of the desired behavior the agent should exhibit as it interacts with the environment. The ultimate goal of the agent is to learn a policy or strategy for selecting actions that maximizes its expected benefit. The dissertation focuses on sequential decision-making when the environment is characterized by continuous states and actions, and the agent has imperfect perception, incomplete knowledge, and limited computational resources.

The main characteristic of the approach proposed in this dissertation is that the agent uses its previous experiences to improve estimates of the long-term benefit associated with the execution of specific actions. The agent uses these estimates to evaluate how desirable is to execute alternative actions and select the one that best balances the short- and long-term consequences, taking special consideration of the expected benefit associated with actions that accomplish new learning while making progress on the task. The approach is based on novel methods that are specifically designed to address the problems associated with continuous domains, imperfect perception, incomplete knowledge, and limited computational resources. The approach is implemented using case-based techniques and extensively evaluated in simulated and real systems including autonomous mobile robots, pendulum swinging and balancing controllers, and other non-linear dynamic system controllers.

CONTENTS

DEDICATION iii

SUMMARY iv

LIST OF TABLES ix

LIST OF FIGURES x

ACKNOWLEDGEMENTS xii

Chapters

1 INTRODUCTION 1
- 1.1 The Research Problem . . . 2
- 1.2 Sequential Decision-Making Framework . . . 4
- 1.3 Dimensions of the Problem . . . 7
- 1.4 The Approach to the Solution . . . 11
- 1.5 Example: A Mars Rover . . . 12
  - 1.5.1 The Problem . . . 13
  - 1.5.2 Overview of the Solution . . . 15
- 1.6 Research Problem Revisited . . . 17
  - 1.6.1 Continuous State and Action Spaces . . . 17
  - 1.6.2 Imperfect Perception . . . 17
  - 1.6.3 Incomplete and Dynamic Knowledge . . . 19
  - 1.6.4 Computational Procedures . . . 20
- 1.7 Contributions . . . 22
- 1.8 Organization of the Dissertation . . . 23

2 STATEMENT OF THE PROBLEM 25
- 2.1 A Model of Behavior . . . 26
- 2.2 A Model of Performance . . . 29
- 2.3 Classification of Agents . . . 31
- 2.4 Formal Statement of the Problem . . . 34
- 2.5 Conclusions . . . 38

3 APPROACH TO THE SOLUTION **40**
3.1 Optimal Approach 41
3.1.1 Environment and Task 42
3.1.2 Information State and Optimal Perception Function 43
3.1.3 Optimal Policy and Value Function 44
3.1.4 Computing the Optimal Value Function 45
3.1.5 Computation Methods 47
3.1.5.1 Value Iteration 48
3.1.5.2 Policy Iteration 49
3.1.6 Difficulties 49
3.2 Heuristic Approach 51
3.2.1 Internal State and Perception Function 51
3.2.2 Optimal Policy and Value Function 53
3.2.3 Adaptive Policies and Function Approximators 54
3.2.4 Adaptation Methods 56
3.2.4.1 Model-Free Procedure 57
3.2.4.2 Model-Based Procedure 58
3.3 Conclusions 60

4 CONTINUOUS STATE AND ACTION SPACES **61**
4.1 Issues with Function Approximators 62
4.2 Function Approximators 65
4.2.1 One-Step Search 66
4.2.2 Adaptation 67
4.2.3 Sparse Coarse-Coded Function Approximators 69
4.2.3.1 Lookup Tables 71
4.2.3.2 Cerebellar Model Articulation Controller 72
4.2.3.3 Memory-Based Function Approximators 74
4.3 Non-uniform Preallocation of Resources 80
4.3.1 Implementation of Non-Uniform Functions Approximators . . 82
4.4 Results 82
4.4.1 Double Integrator 84
4.4.1.1 Optimal Solution 85
4.4.1.2 Uniform and Non-uniform CMAC Agent Configurations 86
4.4.1.3 Uniform and Non-uniform Instance-Based Agent Configurations 89
4.4.1.4 Uniform and Non-uniform Case-Based Agent Configurations 92
4.4.2 Pendulum Swing Up 93

4.4.2.1 Uniform and Non-uniform CMAC 98
4.4.2.2 Uniform and Non-uniform Case-Based 100
4.5 Discussion 101
4.5.1 Double Integrator 101
4.5.2 Pendulum Swing Up 107
4.5.3 Summary 109
4.6 Conclusions 110

5 IMPERFECT PERCEPTION 113
5.1 Case-Based Perception Function 115
5.1.1 Case Representation 116
5.1.2 Learning Cases 118
5.2 Parameter-Adaptive Policies 123
5.2.1 Agent Structure 124
5.3 A Case Study: SINS 126
5.3.1 The Control Module 127
5.3.2 The Reward Signal 127
5.3.3 Adaptation Module 128
5.4 Evaluation 128
5.4.1 Systematic evaluation of SINS 131
5.4.2 Study 1: Design Decisions 132
5.4.2.1 Experimental Design and Data Collection 134
5.4.2.2 Model Construction 134
5.4.2.3 Model Validation 140
5.4.2.4 Robustness Analysis 142
5.4.2.5 Learning profiles 144
5.4.3 Study 2: Experiments with a Real Robot 144
5.4.4 Discussion 148
5.5 Conclusions 148

6 INCOMPLETE KNOWLEDGE 150
6.1 Model-Based Dual Control Algorithms 152
6.1.1 Multiple Hypotheses Scenario 154
6.1.2 Unknown Parameters Scenario 155
6.2 Results 156
6.2.1 Fixed Non-Learning Policy 158
6.2.2 Random Exploration Learning Policy 160
6.2.3 Multiple Hypotheses Learning Policy 162
6.2.4 Unknown Parameters Learning Policy 166
6.3 Discussion 171

6.4 Conclusions 177

7 MARS ROVER **179**
7.1 Mars Rover 180
7.2 Details 180
7.2.1 Perception 184
7.2.2 Decision 184
7.2.3 Adaptation 185
7.3 Results 186
7.3.1 Fixed Non-Learning Agent 186
7.3.2 Passive-Learning Agent 190
7.3.3 Active-Learning Agent 197
7.4 Discussion 203
7.5 Conclusions 206

8 EVALUATING DESIGN DECISIONS **207**
8.1 Overview 208
8.2 Evaluation Methodology 209
8.2.1 Experimental Design and Data Collection 210
8.2.2 Model Construction 212
8.2.3 Model Validation 215
8.2.4 Robustness Analysis 215
8.3 Conclusions 215

9 CONCLUSIONS **217**
9.1 Contributions 218
9.2 Future Work 220
9.3 Conclusions 221

VITA **223**

BIBLIOGRAPHY **224**

LIST OF TABLES

1	Summary of the Dimensions of Sequential Decision-Making Problems.	11
2	Classification of Agents.	34
3	General Problem Statement.	35
4	Specific Problem Statement.	37
5	Design of the Uniform and Non-uniform CMAC Agents for the Double Integrator.	88
6	Design of the Uniform and Non-uniform Instance-Based Agents for the Double Integrator.	91
7	Design of the Uniform and Non-uniform Case-based Agent for the Double Integrator.	94
8	Design of the Uniform and Non-uniform CMAC Agent for the Pendulum Swing Up.	99
9	Design of the Uniform and Non-uniform Case-Based Agent for the Pendulum Swing Up.	102
10	Summary of Results for the Double Integrator at Trial 50.	105
11	Summary of Results for the Pendulum Swing Up at Trial 100.	108
12	Experimental Design Matrix.	135
13	Alternative Models for Study 1.	136
14	Best Subsets Regression Results for the Learning Phase.	137
15	Best Subsets Regression Results for the Maturity Phase.	137
16	Model Coefficients for the Learning Phase.	138
17	Model Coefficients for the Maturity Phase.	139
18	Experimental Design Matrix for the Robustness Analysis.	143
19	Model Coefficients for the Learning Phase.	143
20	Model Coefficients for the Maturity Phase.	143
21	Design of the Multiple Hypotheses Agent for the Double Integrator.	164
22	Design of the Unknown Parameters Agent for the Double Integrator.	168
23	Summary of Results for the Double Integrator at Trial 50.	175
24	Design of the Non-Learning Agent for the Mars rover.	188
25	Design of the Passive-Learning Agent for the Mars rover.	194
26	Design of the Heuristic Solution Agent for the Mars Rover.	199
27	Summary of Results for the Mars Rover Problem.	204
28	Systematic Evaluation Methodology.	209

LIST OF FIGURES

1 Sequential Decision-Making Framework 6
2 An Example: the Mars rover. 14
3 Diagram of the Research Questions. 18
4 Behavior Model 27
5 One-Step Search Algorithm. 67
6 Gradient Descent Version of the SARSA Algorithm. 69
7 CMAC Function Approximator. 72
8 Memory-based Function Approximator. 75
9 Non-uniform Resource Preallocation. 81
10 One-Step Search Algorithm with Skewing Function. 82
11 Gradient Descent Version of SARSA Algorithm with Skewing Function. 83
12 The Double Integrator Problem. 85
13 Optimal Value Function for the Double Integrator. 87
14 Optimal Trajectory for the Double Integrator. 89
15 Optimal Trajectory and Sum of Rewards for the Double Integrator. . 90
16 Skewing Functions. 92
17 Average Steps per Trial in the Double Integrator Problem. 95
18 Average Cumulative Cost in the Double Integrator Problem. 96
19 The Pendulum Swing Up Problem. 97
20 Average Steps per Trial in the Pendulum Swing Up Problem. 103
21 Average Cumulative Cost in the Pendulum Swing Up Problem. . . . 104
22 An Example of a Case. 117
23 Schematic Representation of the Matching Process. 120
24 Self-Organization of Cases in Input Space. 122
25 Agent Structure. 124
26 SINS Algorithm. 129
27 Performance of SINS vs. the Choices of Input Representation and Parameter-Selection Policy. 136
28 Residual Plots for the Learning Phase. 141
29 Residual Plots for the Maturity Phase. 141
30 Learning Profiles for SINS. 145
31 The Robot and its Environment. 146
32 Performance SINS in the Real Robot. 146
33 Actual Path Followed by the Robot. 147
34 Model-based Heuristic Solution Algorithm. 153

35 Model-based Perception Function. 157
36 Trajectories for the Optimal-Light Policy. 159
37 Trajectories for the Optimal-Heavy Policy. 159
38 Trajectory of the Car in State Space. 161
39 Belief of Hypothesis 1. 162
40 Performance of the Multiple Hypotheses Policy in the Double Integrator.165
41 Trajectory of the Car in State Space. 165
42 Belief of Hypothesis 1. 166
43 Performance of the Unknown Parameters Policy in the Double Integrator. 169
44 Trajectory of the Car in State Space. 170
45 Belief of the Mass of the Car. 170
46 Belief of the Standard Deviation of the Mass. 171
47 Trajectory of the Car in State Space. 172
48 Belief of the Mass of the Car. 173
49 Belief of the Standard Deviation of the Mass. 174
50 A Schematic of the Mars Rover Problem. 181
51 Target Detection Probability Distribution. 183
52 Test Environment. 187
53 Finite State Automata for the Non-learning Agent. 189
54 Trajectories for the Non-learning Agent. 190
55 Finite State Automata for the Passive-Learning Agent. 192
56 Trajectories for the Passive-Learning Agent. 195
57 Evolution of Belief Variables in the Passive-Learning Agent. 196
58 Trajectories for the Rover Starting with Equal Beliefs. 198
59 Evolution of Belief Variables in the Active-learning Agent. 200
60 Trajectories for the Rover Starting with 90% Belief in Lower-Left Target.201
61 Value Functions in the Internal State Space (Complete Knowledge). . 201
62 Value Functions in the Internal State Space (Incomplete Knowledge). 202

ACKNOWLEDGEMENTS

This work resulted from the interaction of many friends, colleagues, and family members. First, I would like to thank my advisor, Ashwin Ram, for his continued guidance, inspiration and support, for sharing many of his insights, and for his constant effort to make me a better scientist. I owe much to him for the freedom he gave me to pursue my own interests. Ashwin played a fundamental role during the entire development and writing of this dissertation and my education.

I am also indebted to the other members of my committee, Chris Atkeson, Ron Arkin, Janet Kolodner, and Alex Kirlik. They all contributed to improve further this dissertation and gave thoughtful criticism and comments. I would like to give special thanks to Chris Atkeson and Ron Arkin. The work described here had a strong influence from many conversations on the theory and practice of reinforcement learning, dynamic programming, and robotics. Chris introduced me to the theory of dual control and Ron gave me the opportunity to work and play with the robots in the Mobile Robotics Laboratory. I also thank Jeffrey Donnell for many key comments that helped me improve the writing of this dissertation.

I wish to thank other faculty that contributed with my education. Amihood Amir and Gil Nieger at Georgia Tech and Germán Gonzalez and Nelson Vázquez at Simón Bolívar University provided many stimulating conversations about everything. Their knowledge, scientific standards, motivation, and teaching style strongly influenced my way of thinking. Their influence extends beyond this work.

I thank Rich Sutton and Andy Barto for a very productive and intensive summer internship at the Adaptive Network Laboratory. I had the opportunity to closely interact with them and learn from many inspiring conversations. The Adaptive Network Laboratory was always a pleasant and intellectually stimulating workplace. Sergio Guzmán, Mike Duff, Any Fagg, and Nathan Sitkoff helped me make my stay so rewarding.

I wish to thank many of my friends in the IGOR group, specially Mark Devaney, Kenny Moorman, Anthony Francis, Gordon Shippey, and Andrés Gómez Da Silva. Our discussions and seminars helped me keep in touch with projects in other fields in the area and provided an invaluable source of new ideas and motivation. Eleni Stroulia, Mike Cox, Kevi Mahesh, and Sam Battha were excellent student models as I learned from them during my first years in the College. Tucker Balch, Doug

MacKenzie, and Gary Boone were the robotics pals with whom I learn all the mysteries, myths, and tricks of making mobile robots move, specially during the AAAI robot competitions.

Liliana Guédez and Augusto Op Den Bosch helped me to stand up and continue to fight when I though I had lost. Their companionship and friendship helped me regain my faith and realize I would finish some day. Juan Carlos, Suzanne, and specially Juan Carlitos Viera were always present. We shared many special moments during my stay in Atlanta.

I could not have initiated this program without the generous support of Fundación Sivensa and Fundayacucho. They provided financial support during the first two years of the program. The Army Research Lab, GTE, and Savannah River Lab provided financial support for the following years. Their scholarships helped me to concentrate more on research rather than the basic needs of life.

I will always be grateful to my parents. They have the most influence in making me the person I am today. They always reminded me that I should enjoy whichever activity I choose to perform. They taught me how to pursue and accomplish my objectives and encouraged me to seek out, explore, and learn even though it could mean leaving home. They never ceased to believe that I could finish and to hope that I would. I also thank my brother for sharing with me many special moments during the program. Our telephone conversations, e-mails, and jokes have kept me going.

Finally, I owe my deepest thanks to my wife, Carolina, for the shared sacrifices and happiness of these past few years. She helped me to keep life in perspective and made the effort so much easier.

CHAPTER I

INTRODUCTION

An autonomous agent is an intelligent system that has an ongoing interaction with a dynamic external world. It can perceive and act on the world through a set of limited sensors and effectors. Its most important characteristic is that it needs to make decisions continuously, one after another, during its entire "life". With the execution of each decision, the agent is able to modify the external world in some way towards or against its advantage. Furthermore, it is a long and on-going sequence of interdependent decisions, rather than just one, that determines the overall success or failure of the task the agent is attempting to perform. Thus, the main problem the agent confronts is that of *autonomous sequential decision-making*: at any given situation the agent must perceive its environment, decide what action to perform next, execute the action, observe the situation that results, and repeat the whole cycle at the new situation. Additionally, after executing every action, the agent should learn from the outcome so that it can make better, improved decisions as it progresses. Our main objective in this dissertation is to study algorithms by which autonomous agents can learn, using their own experience, to perform sequential decision-making efficiently and autonomously.

The next sections provide an overview of the research problem, our generic framework formulating sequential decision-making problems, the heuristic approach we developed, and the contributions of this research. Section 1.1 gives an overview of the problem and the focus of this research. Section 1.2 describes the framework we present in this dissertation to study sequential decision-making problems. The framework is particularly important because it enables the study of sequential decision-making problems at a general level in which the details of the agent, environment, and task are abstracted. Section 1.3 describes the dimensions in which sequential decision-making problems can vary according to the framework, and states the dimensions that are the focus of this research. Section 1.4 provides an overview of the solution put forward in this dissertation. The solution is derived and described abstractly in terms of the framework. This has particular importance because the solution only depends on the components of the framework and not on the details of the particular sequential decision-making problem the agent is attempting to solve.

Section 1.5 instantiates the description of the framework and an overview of the solution in a concrete example. This helps to ground the concepts and ideas described so far in a real-world example. Section 1.6 revisits in more detail the questions explored in this research and Section 1.7 provides an outline of the rest of the dissertation.

1.1 The Research Problem

Imagine a rover on Mars looking for rare rocks. As the rover executes its mission it is constantly deciding what to do next and executing actions sequentially. With every action the rover modifies the state of the environment in which it is embedded and receives new information that transforms its own knowledge about the surroundings and its role in the environment. The main question faced by the rover is: What action should I execute next? The rover must consider the fact that every action may (1) produce an immediate cost or benefit, (2) modify the state of the environment, and (3) transform its own knowledge. The rover must evaluate how beneficial or harmful these outcomes are in the context of finding rare rocks in the long-term, which is its overall goal. Furthermore, as the rover moves, turns its wheels, or grasps rocks it has no explicit teacher to tell it what actions are right or wrong. However, the rover has a sensorimotor connection with its environment that produces a vast amount of information about cause and effect and about the consequences of actions. The rover can use this information to improve the performance of its task. For example, it can learn by trial and error that moving too fast in a particular type of terrain may cause the wheels to slip or that certain types of rocks are too heavy to lift. Learning from interaction is a fundamental form of learning because the information that results from such interactions is a major contributor for developing the agent's sense of the environment and its own role in it.

Learning from interaction is a hard and interesting problem because the agent is not told which action to take, but instead must discover through trial and error which actions will accomplish its task. To be successful in this discovery the agent must interleave *action selection* and *learning from outcomes*. Each of these is a hard problem by itself. Selecting the best action to execute next is difficult because every action the agent executes may produce a change in the environment and, through the sensorimotor connection, a change in the agent itself. Thus, actions can have both immediate and long-term effects: an agent may decide to execute an action that will make it "feel good" now, but inevitably cause a change in the environment that will force it to "feel bad" later on. In addition, the agent may decide to execute an action which, although it does not produce an immediate benefit, can help it learn something that would be useful later on. Similarly, learning from the outcomes of every action is also difficult because the environment is constantly changing with the execution of each action, the agent is not told what is the right action to execute next, and the consequences the agent observes now may be the result of not only the action the agent most recently executed but other actions it executed in the past. This is the crux of the sequential decision-making problem and this research studies the question of how the agent can learn to decide what action to execute next in a given situation.

In this dissertation, we present a sequential decision-making framework to study the problem. The sequential decision-making framework provides a generic model of the behavior of autonomous agents embedded in dynamic environments as well as an objective measure to evaluate task performance. Basically, each sequential decision-making problem consists of three main elements that interact: the *agent* or decision-maker, the *environment*, and the *task*. The next section describes these elements in more detail. Sequential decision-making problems has been extensively studied from different perspectives by researchers in artificial intelligence, optimal control, and operations research communities. However, there are many topics that remain unsolved and are open for research. Some of these topics concern solving sequential decision-making problems when the environment is characterized by real-valued states and actions (i.e., continuous domains) and the agent has imperfect perception, incomplete knowledge, and limited computational resources. The research in this dissertation focuses on this type of sequential decision-making problems.

Sequential decision-making problems with the characteristics mentioned above are particularly difficult. First, it is extremely unlikely that an agent operating in environments with continuous state and action spaces would face a situation exactly identical to one encountered earlier; therefore, it must be able to generalize the outcome of every individual experience to other regions of the state and action spaces. Additionally, since the number of possible actions is infinite, the agent must use an effective procedure capable of analyzing and evaluating any possible action in order to select the best one. Second, an agent with imperfect perception is not able to completely characterize the current state of the environment (this problem is also known as the *perceptual aliasing* or *hidden state* problem). Imperfect perception heavily impairs the ability of the agent to determine the long-term consequence of its actions; therefore, the agent must be able to consider not only the recent sensation it receives but the recent history of sensations and actions to try to characterize the current environment situation. Finally, an agent that has incomplete but evolving knowledge about its environment faces the dilemma of *exploitation* vs. *exploration*. That is, actions bear a "dual" character: They must be directors to a known degree (i.e., exploit current knowledge), but also investigators to a known degree (i.e., explore to acquire knowledge). Thus, the agent must be able to select actions that are necessary not only for making progress on the task, but also for studying the environment. The sequential decision-making framework presented in this dissertation provides the mathematical foundations that are required to study this type of problems in detail, and to analyze heuristic approaches for its solution. One of the most important characteristics of the framework is that it unifies the description of sequential decision-making problems into a single model. This is particularly important because we can use a single formalism to represent, analyze, and solve generic sequential decision-making problems, specially when the environment is characterized by continuous state and action spaces and the agent has imperfect perception and incomplete knowledge. Additionally, the framework provides two dimensions by which an agent can improve its performance. One involves the acquisition of new knowledge, which we refer to as *learning*; the other involves modifying the internal self of the agent, which we refer to as *adaptation*. We describe this distinction in more detail in Chapter 2.

The next section presents the main concepts involved in the formulation and solution of sequential decision-making problems. Section 1.5 instantiates the framework in a concrete example that illustrates the use of the framework to formulate a specific problem and grounds the main concepts involved in the solution.

1.2 Sequential Decision-Making Framework

Our framework for autonomous sequential decision-making uses a formulation similar to the one used in situated automata (e.g., Agre and Chapman, 1987), reinforcement learning (e.g., Kaelbling, 1990) and optimal control problems (e.g., Bertsekas, 1995a). Our framework models the behavior of an agent acting in a particular environment as a pair of interacting automata, one corresponding to the embedded agent and the other to the physical environment. The environment delivers sensations to the agent which causes the agent to respond with actions. Each automata has a local state that changes as a function of the signals delivered from the other. The main objective of an agent designer is to implement a decision-making strategy capable of selecting and executing actions based on the local state of the agent such that their execution cause the desired effects in the environment over time.

Since agents using different decision-making strategies may achieve different degrees of success, an important issue that arises in the behavior model is the evaluation of decision-making strategies with respect to a given task. The framework provides an effective set of generic and widely applicable criteria to objectively evaluate and compare different decision-making strategies. Specifically, every strategy has a performance measure that is computed according to an evaluation criteria. An ideal strategy[1] is one that achieves the optimal performance measure. Thus, it is possible to select the best decision-making strategy from a set of strategies by comparing their performance measures. Moreover, it is possible to select an optimal decision-making strategy from the set of all possible feasible strategies.

The sequential decision-making framework consists of three main components: the agent, the environment, and the task (see Figure 1). The agent attempts to control the environment by choosing actions in a sequential fashion. The environment is a dynamic system characterized by a *state* and its *dynamics*, a function that describes the evolution of the state given the agent's actions. A task is a declarative description of the desired behavior the agent should exhibit as it interacts with the environment. A task is defined by a *reward function* and an *evaluation criteria*. The reward function associates a scalar value or *reward* to every action the agent executes and the evaluation criteria specifies how the performance measure is computed from the rewards (e.g., sum of rewards, average rewards, discounted sum of rewards, etc.). The ultimate goal of the agent is to learn a *policy*, or strategy for selecting actions, such that performance measure specified by the task description is optimized. Thus, the task defines "what" the agent should do but it is the agent's responsibility to find out "how".

Figure 1 shows the three main elements and their interaction. A description of each component follows.

- **Environment**
  The environment is the component that represents the domain in which the decision-maker or agent is embedded. The environment is an automaton characterized by a *state*, which is a compact description of the history of the environment; and its *dynamics*, which is a function that describes the evolution of the state given the agent's actions. The state contains information about the current status of the environment and, along with the agent's actions, makes the future development of the environment independent of the past. This formulation has been used to characterize dynamical systems; it is very general and applicable to any problem of interest. The environment receives as input every action delivered by the agent, undergoes a state transition as described by the dynamics, and produces as output the respective sensation, which is then received by the agent.

- **Agent**
  The agent is the component that implements the action selection mechanism. The agent is an automaton characterized by an *internal state*, a *perception*

[1]There may be situations in which there are more than one decision-making strategy that accomplish the same and optimal performance measure. In such cases, any of these strategies is considered optimal with respect to the evaluation criteria.

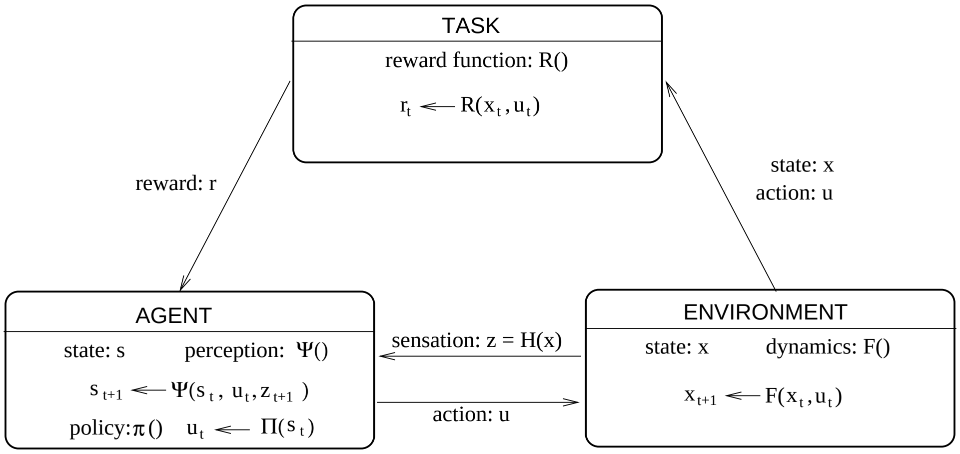

Figure 1: Sequential Decision-Making Framework
The framework consists of three main elements shown in boxes: the agent, the environment, and the task. The arrows indicate the flow of information among the elements.

function, and a *policy*. The internal state is a compact description of the sequence of sensations and actions the agent has experienced since the beginning of execution. It represents the history of the interaction of the agent with the environment. The agent uses the perception function to maintain the internal state up-to-date by producing a new internal state every time the agent executes an action and receives a new sensation. The policy is the function that implements the decision-making strategy of the agent. It maps the current internal state to the action that should be executed next. Thus, the agent operates sequentially and cyclically: every time it receives a new sensation the agent undergoes an internal state transition according to the perception function. Then it uses the policy to select and execute a new action, which causes a state transition in the environment and produces a new sensation that depends on new state.

In the simplest formulation the agent has perfect perception, which means that the internal state is exactly identical to the state of the environment. In other more complicated (and realistic) formulations, the sensations deliver only partial and/or noisy information about the state of the environment. In either case, the agent can only use the evidence available through the history of previous sensations and actions as collected through the perception function and as represented in the internal state to decide what action to execute next.

- **Task**
  The task is the component that specifies the desired behavior of the agent. The task consists of a *reward function* that associates a scalar value or *reward* to every possible combination of the state of the environment and action of the agent, and an *evaluation criterion* that specifies how the rewards are combined to produce a performance measure of the resulting behavior. Two common examples of criteria are (possibly discounted) sum of rewards and average rewards. According to these criteria the best policies are the one that accomplish the maximum sum of rewards or the maximum sum of rewards divided by number of steps. The reward function implicitly describes the behavior the agent should perform because the agent is supposed to select actions in such a way that the evaluation criterion is optimized.

Each component in the framework plays a different role in the specification of a particular sequential decision-making problem. The environment represents the object subject to control, the task represent the demands imposed on the controlled object, and the agent represents the controller per se. Each component may vary across different dimensions. For example, the state of the environment may consist of variables having finite (i.e., discrete) or infinite (i.e., continuous) values. The dimensions help to categorize the types of sequential decision making problems and approaches to their solution. The next section describes these dimensions and describe the specific type of sequential decision-making problem that are the focus of our research.

1.3 Dimensions of the Problem

Sequential decision-making problems vary in complexity along several dimensions. It is useful to classify sequential decision-making problems according to their dimensions because this may simplify the methods used for their solution. For example, sequential decision-making problems in which the environment has discrete state and action spaces and deterministic dynamics can be solved using classical planning techniques (see Russell and Norvig, 1995, Chapter 2). This section describes all the dimensions in which sequential decision-making problems may vary according to the framework. These dimension are needed to characterize precisely the sequential decision-making problems that are the focus of our research.

- **State Space**
  The state of the environment is a collection of attribute-value pairs that characterize the environment. This dimension refers to the type of description of

such attributes or variables and can be either discrete or continuous. An example of an environment with discrete state space is the game of checkers. The state consists of the current board configuration. An example of an environment with a continuous state space is a mobile robot in a flat terrain. The state consists of the position and velocity of the robot and the positions, velocities, and sizes of the obstacles.

- **Action Space**
  The agent attempts to control the environment through a set of actions that can be described using attribute-value pairs. This dimension refers to the type of description of such attributes or variables and can be either discrete or continuous. An example of an environment with discrete action space is the game of checkers. An action consists of the movement of one of the tokens. An example of an environment with a continuous action space is a mobile robot in a flat terrain. An action may consist of the acceleration vector for the robot's actuators.

- **Decision Time**
  This dimension refers to the time in which the agent is allowed to execute decisions and can be either discrete or continuous. In discrete time tasks the agent can only execute a decision at fixed time intervals. In continuous time tasks the agent can execute a decision at any time. An example of a problem using discrete time is the game of checkers. The agent must make a decision after the opponent's move. An example of a problem using continuous time is a mobile robot navigating in a flat terrain. The agent must make a decision after each infinitesimal time interval in a continuous, on-going manner.

- **Interaction Mode**
  This dimension refers to the type of interaction the agent has with the environment and can be either synchronous or asynchronous. In synchronous interaction mode the agent and the environment change their states alternatively. In other words, the environment waits for the agent while the agent is selecting its action and the agent waits for the environment while the environment is changing to the next state. In asynchronous interaction mode the agent and the environment may change their states simultaneously without one waiting for the other. An example of a problem using synchronous interaction mode is the game of checkers. Each opponent must wait without changing the board configuration until the other makes a move. An example of a problem using asynchronous interaction mode is a missile chasing a mobile target. The missile and target may change direction and speed independently of each other.

- **Perception**
  This dimension refers to the quality of the sensations received by the agent about the state of the environment and can be either perfect or imperfect. Agents using perfect perception receive sensations that contain the complete state description of the environment without noise or any other perturbation. Agents using imperfect perception receive sensations that contain partial state description of the environment; in addition, these sensations may be contaminated with noise or perturbed in some other way. An example of an agent using perfect perception is a chess player. The agent can perceive the complete board configuration without noise. An example of an agent using imperfect perception is a mobile robot navigating in a flat terrain with partially occluded obstacles. The agent cannot perceive the complete state of the environment. Also, the sensors may produce a noisy representation of the state of the environment.

- **Task Horizon**
  This dimension refers to the termination condition of the sequential decision-making task and can be either finite or infinite. Finite horizon tasks have a limited number of decision stages or bounded amount of time for task completion. Infinite horizon tasks have an unlimited number of decision stages or can continue for ever. Usually, infinite horizon tasks are created by concatenating an infinite number of finite horizon tasks. An example of a finite horizon task is an agent solving one maze. The agent executes a finite number of decisions until it finds the exit of the maze. An example of a infinite horizon task is an agent solving one maze after another. Another example of an infinite horizon task is an agent trying to minimize the energy expenditure of a building by controlling the lights, heater, and air conditioning automatically throughout the years.

- **Functions Outcome**
  This dimension refers to the description of the outcomes of the dynamics of the environment and the reward function and can be either deterministic or stochastic. Functions with deterministic outcomes produce the same result given the same inputs. On the other hand, functions with stochastic outcomes may produce different results given the same inputs, but the outcomes are governed by a specific probability distribution. An example of an agent interacting with a deterministic environment is a robot navigating in a flat terrain with perfect sensing and no slippage. The position and velocity of the robot that results after executing an specific action at a given state is always the same. An example of an agent interacting with an stochastic environment is

a backgammon player. The outcome of rolling the dice may be different even when the board configuration (i.e., state) is the same.

- **Knowledge**
This dimension refers to the types of change in the knowledge of the agent and can be either fixed or dynamic. An agent with fixed knowledge is not able to modify its understanding of the environment with experience. The knowledge remains the same regardless of what action the agent executes. The agent either knows exactly what would be the outcome of executing any action at any given state or at least knows their probability distributions. In this case, executing any action does not help the agent to reveal hidden properties in the environment. An agent with dynamic knowledge is able to modify its understanding of the environment with experience. The agent may execute "exploratory" actions to gain additional knowledge about the environment in addition to performing the task to the best of its ability. An example of an agent with fixed knowledge is a backgammon player. The agent knows what would be the board configuration after executing each action (i.e., it can evaluate every alternative board position that is possible after rolling the dice). An example of an agent with dynamic knowledge is an agent solving an unknown maze. The agent does not know if a particular branch will lead to the exit, but it can learn this by moving around and exploring the maze.

The sequential decision-making problems we will explore in depth in this dissertation are the ones with continuous state and action spaces, discrete decision times, imperfect observations, infinite horizon, deterministic outcomes, and using agents with dynamic knowledge. Table 1 shows a summary of the dimensions and their possible values. The ones shown in uppercase are the focus of this dissertation. Most of the interesting real-world problems possess these characteristics. For example, consider the Mars rover example again. The rover operates in an environment characterized by a continuous state (i.e., the position and orientation of the rover and obstacles are real-valued variables), the rover's actions are continuous (i.e., the direction in which to move is characterized by real-valued variables), the rover's perception is imperfect (i.e., the rover's sensors are noisy and do not reveal the complete state of the environment), and the rover's knowledge of the environment is incomplete and dynamic (i.e., the rover may not know the layout of the environment behind some rocks, but it can find out this information by going around such rocks).

Table 1: Summary of the Dimensions of Sequential Decision-Making Problems.

Dimension	Values	
State space	discrete	CONTINUOUS
Action space	discrete	CONTINUOUS
Decision time	DISCRETE	continuous
Interaction mode	SYNCHRONOUS	asynchronous
Perception	perfect	IMPERFECT
Task Horizon	finite	INFINITE
Functions Outcome	DETERMINISTIC	stochastic
Knowledge	fixed	DYNAMIC

1.4 The Approach to the Solution

The key element for the solution of sequential decision tasks is the concept of a *value function*, which estimates how desirable it is to be in various states. The desirability of a state is called the *value* of the state, and is an estimate of the performance measure the agent can expect to achieve when it starts from the given state and follows a given policy. The value function is key for action selection because the agent can use it to judge how pleased or displeased it might be in the long-term when it starts at a given state. Thus, when the agent is considering several possible actions, it can select the best one by using a simple lookahead procedure. First, it uses the value function to estimate the value of the states that results from the execution of each action. Then, it selects the action that leads to the state with the highest value. It is through the use of this procedure and the value function that the agent can avoid future danger or seek future pleasure even in the presence of immediate rewards or penalties.

Unfortunately, determining the value function in the general case is computationally intractable. Thus, a key problem is to develop a tractable method by which an agent can incrementally learn the optimal value function by continually exercising the current, non-optimal estimate of the value function and improving this estimate after every experience. In these methods the agent can make a decision using the current, non-optimal estimate of the value function by selecting the action leading to the highest value at the current state. Then, after observing the result of executing such action, the agent can improve the long-term estimate of the value function associated with such state.

We present a new approach to solve sequential decision-making problems in the case in which the environment is characterized by continuous state and action spaces and the agent has imperfect perception and incomplete but dynamic knowledge of the environment. This class of problems includes most realistic situations involving robotic and other physical agents. In this case, the agent is forced to select actions taking into account both their efficiency with respect to the task and their influence on the current level of knowledge their execution may produce. Additionally, the agent is forced to generalize the outcome of individual experience to other regions of the state and action spaces. Briefly, the new methods use value functions that map agent's states consisting of sensations and knowledge combined to their long-term values. The new methods also possess the distinguishing characteristic of regulating learning as required by the performance measure by considering the three consequences each action produce: the immediate reward, the next state in the environment, and the next knowledge level in the agent. At any decision point the best action is the one that maximizes the sum of the immediate reward and the value of the next state and knowledge level taken together. Thus, agents using these methods exhibit the dual characteristic of appropriately distributing the control energy for learning and control purposes. Additionally, the value function must be able to represent an infinite mapping between states and their associated long-term benefits using finite memory resources. This requires function approximators that can generalize individual experiences and reliable extrapolate values to unexplored regions of their input space.

Our approach uses case-based techniques to accomplish these goals. The solution is inspired by the optimal but computationally intractable theoretical solution to the problem. In the proposed case-based method, the agent uses previous experiences to represent the value function and estimates the value of each action at every decision point by retrieving similar past experiences and interpolating their values. Then, the agent selects, to the best of its knowledge, the action leading to the best long-term value. The agent progressively improves its decision-making process by observing the reward resulting from the execution of each action and reestimating the value function. The approach is fully implemented and evaluated empirically on problems involving robot navigation, minimum time, and continuous cost dynamic tasks. The results show that the new methods are effective and more efficient than other well-known methods in the field.

1.5 Example: A Mars Rover

This section describes an example of the types of sequential decision-making problems we will address in this dissertation. It presents the problem and gives an intuitive sense of the solution we propose to solve the problem.

1.5.1 The Problem

Consider a space mission to Mars consisting of a satellite, an immobile base station, and a mobile robot that has arrived on Mars. The main objective of the mission is to use the mobile collect rock samples near the landing site and deliver them to the immobile base station for analysis. The robot has limited sensing capabilities and incomplete knowledge of its surroundings, but it can move in any direction at any speed up to the physical limits imposed by its hardware. Additionally, images from the satellite that remained in orbit determined possible areas for the location of interesting rocks as well as dangerous zones. These images provide only initial clues about the surroundings and cannot be considered completely reliable. Imagine that the robot controller may issue a new command every fixed time interval, say, every 5 seconds. The main question for the robot controller is: what should the next command be if it wants to maximize the success of the mission, where success here means to seek and find a rock and bring it back while minimizing the energy spent during the process. Figure 2 shows the Mars rover problem.

The Mars rover problem has the three basic characteristics explored in this dissertation: continuous domains, imperfect perception, and incomplete knowledge. The domain is continuous because the state (i.e., the position and speed of the rover, obstacles, and rocks) is defined by real-valued variables. Additionally, the actions of the rover (i.e., direction and speed of movement) is continuous. The perception of the rover is imperfect because it is limited by a horizon. In other words, the rover cannot see anything beyond the horizon and even within the horizon, the sensors can only detect rocks with a reliability that is inversely proportional to its distance (i.e., the closer the rock is to the sensor the more reliable is its detection). The knowledge of the rover is incomplete because it does not know the terrain layout with certainty, although a previous belief is given by the images from the satellite. However, the knowledge level is dynamic because the rover can improve this belief of the environment layout by moving around and sensing its surroundings.

The sequential decision task in this problem is to learn to efficiently detect, retrieve, and bring back a rock while minimizing energy. This will require a trajectory that efficiently balance exploration of the possible target locations, finding a candidate rock, and bringing it back safely. A trajectory that takes the rover directly to the closest possible target is not good enough because the rover will not learn more about the environment layout. Conversely, a trajectory that takes a long path through the entire environment before reaching a possible target will enable the rover to learn more about the environment but take too much energy. Intuitively, a good trajectory should take the rover to the possible target sites in such a way that the robot can confirm the presence or absence of candidate rocks before committing to

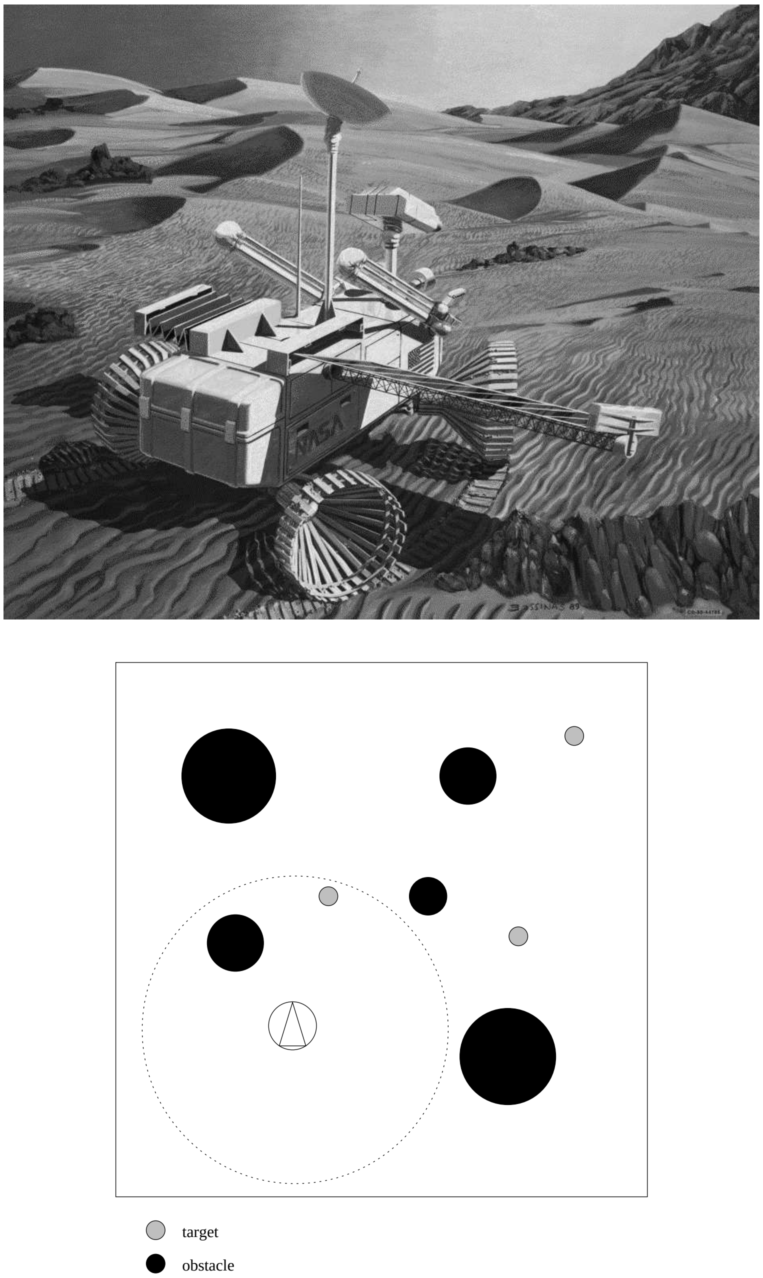

Figure 2: An Example: the Mars rover.
Top: An artistic representation of a Mars rover (courtesy of the Jet Propulsion Laboratory). Bottom: a schematic of the Mars rover problem. The black and grayed circles correspond to obstacles or other dangerous zones and possible targets respectively. The dotted circle around the rover corresponds to the perception horizon (i.e., the rover cannot perceive anything beyond that circle.)

reach that target site. In doing so it should also minimize the amount of energy spent during the process.

1.5.2 Overview of the Solution

Some special scenarios are very easy to solve. Imagine that there are only two target sites located in such a way that they form a straight segment with the location of the robot. Thus, the first target is at one extreme of the segment, the robot is at the other extreme, and the second target is somewhere in the segment between the two extremes. The trivial solution is for the robot to head out directly towards the target at the extreme because in doing so it will necessarily cross over the second target and detect the presence of any rocks there. Now imagine a series of more difficult scenarios in which the second target is moved away perpendicularly to the segment (i.e., the two targets and the robot now form a triangle). Clearly, the original trajectory is no longer the best because the robot may not reliably detect the presence of rocks at the second and closer target. The best trajectory should still take the robot to the target at the extreme but it should also move it close enough to the second target so that the robot has better chance of detecting rocks there. However, if the robot cannot detect rocks at the second target, it should continue to head out to the original destination. Finally, imagine the most difficult case in which there are non-aligned target sites as well as obstacles among them. The best trajectory still needs to take the robot to the target at the extreme and close enough to the second target, but minimizing the chance of colliding with obstacles and the energy spent during the process.

The framework presented in this dissertation allows the study and solution of this type of sequential decision-making. In this example, the rover controller corresponds to the reactive agent, the surface of Mars corresponds to the environment, and a reward function corresponds to the task such that it (1) assigns energy costs to different actions, (2) penalizes the robot for every collision, and (3) rewards the robot for grabbing a rock. The agent has three main components: an internal state, a perception function, and a value function. The internal state consists of an estimate of the rover's current position in the environment as well as the current belief about the existence of rocks at the target sites. The perception function is responsible for modifying the internal state after executing every action and observing the resulting sensation (i.e., changing the estimate of the rover's current position and beliefs based on the outcome of the previous action). The value function maps values to internal states so that the agent can judge the desirability of being at a particular internal state.

Given this structure, the procedure for selecting the best action at any decision stage is as follows. For now, let us assume that the value function provides accurate estimates for the values of internal states. Then, at any given decision stage the agent just needs to select the action leading to the highest value. More specifically, at any decision stage the agent perform the following procedure:

1. For every admissible action at the current stage:
   (a) Determine the possible internal states that may result from executing this action.
   (b) Compute the value of executing this action as the sum of:
      i. The immediate reward obtained for executing this action according to the reward function and,
      ii. The value of the internal states that may result from executing this action according to the value function.
2. Select the action that leads to the highest value.

When the value function provides accurate estimates for the values of internal states, then the action that results from the procedure above described is optimal. This is due to the Bellman's principle of optimality, which will be described in more detail in the following chapters. However, the more usual case occurs when the value function does not provide accurate estimates. The methods presented in this dissertation take this into account and provide the agent a way to continuously make the value function more accurate. The basic idea behind these methods is to change the values of the internal states that occur in a sequence. Specifically, every time the agent undergoes a state transition, the current value of the earlier state is adjusted to make it closer to the value for achieving the later state. Letting s_t and s_{t+1} respectively denote the internal state before and after the state transition at time t, $V(s_t)$ and $V(s_{t+1})$ denote the values of these states, and r_t the immediate reward that result from executing action u_t, then the update rule can be written:

$$V(s_t) \leftarrow r_t + V(s_{t+1})$$

where the expression informally means: make $V(s_t)$ more like $r_t + V(s_{t+1})$. Thus, the agent can incrementally learn a more accurate value function by continuously exercising the current, non-accurate value function and improving such the estimates after every experience. The complete algorithm can be seen as a heuristic search procedure (e.g., Hart, Nilsson, and Raphael, 1968; Hart, Nilsson, and Raphael, 1972), but with the fundamental difference that the heuristic evaluation function changes with experience. This dissertation explores the use of this type of algorithms for solving sequential decision-making problems in which the environment is characterized by continuous state and action spaces and the agent has imperfect perception and incomplete knowledge.

1.6 Research Problem Revisited

This section describes more precisely the questions that motivate the research in this dissertation. Figure 3 shows a diagram of these research questions.

1.6.1 Continuous State and Action Spaces

How can the agent represent the value function in problems having continuous state and action spaces?

Sequential decision problems with discrete state and action spaces are known as *Markov Decision Processes* (MDP) and have been extensively studied by researchers in the field of reinforcement learning and operations research (e.g., see Watkins, 1989; Ross, 1993; Bertsekas, 1995a). In problems having discrete state and action spaces the value function can be represented as a table in which each element represents one of the states and its content represent the value of such state. An agent can learn the optimal value function progressively as it interacts with the environment using a variety of methods ranging from dynamic programming (Bellman, 1957) to temporal difference learning (Sutton, 1988). However, in problems having continuous state and action spaces, the value function must operate with real-valued variables representing states and actions, which means that it should be able to represent the value for infinitely many state and action pairs. This makes the learning problem very difficult because it is very unlikely that the agent would experience exactly same situations it has experienced before. Thus, the agent must be able to generalize the value of specific state-action combinations it has actually experienced to the situation it is currently facing so that it can make a good decision about what to do next.

The main contribution of the dissertation in this area is to extend the use of function approximators to represent the value function across both the continuous state and action spaces. Additionally, it describes two new memory-based learning algorithms (instance-based and case-based) that an agent can use to represent and learn the value function with different degrees of resolution across the state and action spaces. The algorithms are implemented and extensively evaluated in two well-known classical problems in control theory: double integrator and pendulum swinging.

1.6.2 Imperfect Perception

How can the agent characterize the current ongoing situation of the environment?

When the agent has imperfect perception it cannot perceive the actual state of the environment, which is a crucial piece of information needed to perform an

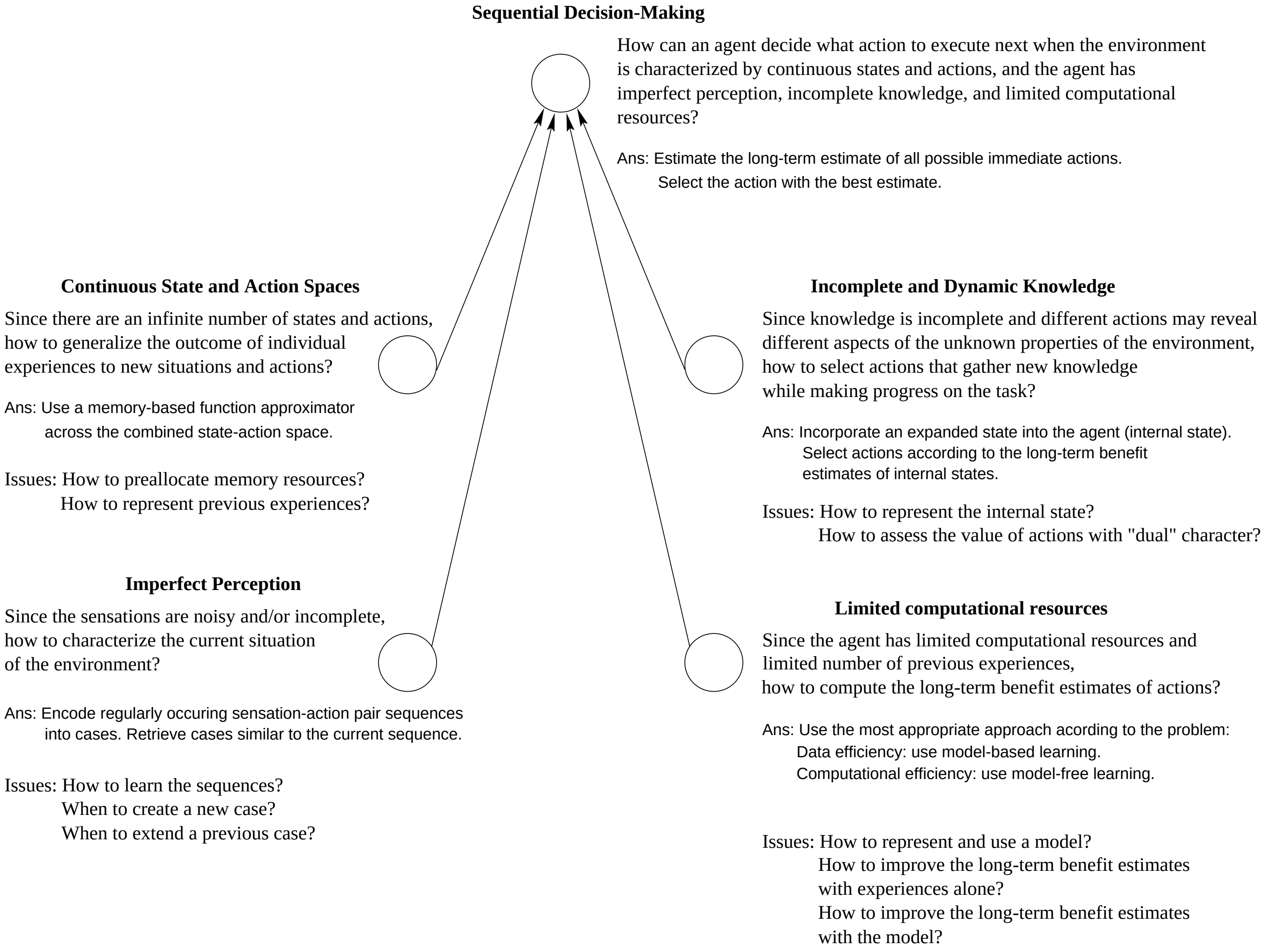

Figure 3: Diagram of the Research Questions.

accurate selection of the action to execute next. The problem of imperfect perception in continuous domains has been studied by researchers in the field of optimal control under the name of *state estimation* (e.g., see Stengel, 1994; Narendra and Annaswamy, 1989). Basically, common methods for state estimation are based on recursive estimation in which an agent combines the current state estimate with every new incoming sensation to produce a better, more accurate state estimate. The most popular and best understood recursive estimation method used in control theory is the Kalman filter (e.g., see Sorenson, 1966). The main benefit of the Kalman filter is its ability to efficiently process information sequentially, producing the best state estimate every time new information is available. However, either a linear model of the dynamics function of the environment or its local linear model expansion are required for the successful application of this or any other recursive estimation method. Another popular recursive estimation method used in the context of dynamic systems characterized by discrete states and actions is the Bayesian method. It is widely used in problems known as *Partially Observable Markov Decision Processes* (POMDP) (Sondik, 1971; Cassandra, Kaelbling, and Littman, 1994; Parr and Russell, 1995). However, in many of real-world problems a linear model of the dynamics function is not available or at best it is known imprecisely and the assumption of discrete states and actions is inappropriate. Thus, in these situations, the agent must rely on a state estimation method that does not require the use of such models. Such methods would allow the agent to determine the current situation and successfully choose the right action to execute next.

The main contribution of the dissertation in this area is a case-based perception function that uses the recent history of sensations and actions to represent situations. The perception function is able to characterize the current environment situation without explicit state identification. Similar sequences of past history of sensation-action pairs are encoded as cases that also contain estimates of the value of executing different actions in such situations. The case-based perception function is implemented on a Denning MRV-III robot and evaluated through extensive simulated and real experiments performed with the mobile robot.

1.6.3 Incomplete and Dynamic Knowledge

How can an agent with incomplete knowledge about the environment choose actions to accomplish both learning more about the environment and successfully performing its task?

The problem of choosing actions that accomplish both learn more about the environment and make progress on the task is known as the *dual control* problem and was initially put forward by Fel'dbaum (Fel'dbaum, 1965). To solve the dual control problem, the agent has to devote some effort, but not too much, to *exploring* the environment in order to *exploit* it proficiently. The exact solution of the dual control problem is computationally intractable. Approximate solutions usually focus almost entirely on action-selection mechanisms that are able to use incomplete knowledge and does not explicitly consider that the knowledge level may change as the agent execute the actions. That is, an agent uses its current level of knowledge to select and execute actions that are efficient from the perspective of such incomplete knowledge. Often, some type of random exploration (e.g., Boltzmann exploration; Watkins, 1989) or exploration bonuses (e.g., Sutton, 1990) bypasses the standard action-selection mechanism so that the agent is able to learn the effect of "unseen" actions. The agent learns to perform better each time because new incoming information is used to improve the knowledge level, which leads to better decisions. However, this is a *passive* form of learning because the agent neither explicitly selects actions to gain new information nor considers how each action will influence its level of knowledge in the short- or the long-term. This contrasts with the more efficient and robust form of *active learning* (e.g., Cohn, Atlas, and Ladner, 1994; Cohn, Ghahramani, and Jordan, 1995) or goal-driven learning (e.g., Ram and Leake, 1995) in which the agent selects actions taking into account both their efficiency with respect to the task and the influence on the current level of knowledge their execution may produce. Thus, in order to select actions that reveal useful knowledge about the environment and make progress on the task, an agent must be able to represent the value of both the current state of the environment and the current knowledge level.

The main contribution of the dissertation in this area is to extend the formulation of sequential decision-making model to include an explicit representation of the uncertainty or knowledge level into the agent's internal state. The value function of the new internal state measures the long-term consequence of both being at a given environment state and having a specific uncertainty about the environment. Thus, when an agent makes a decision based on the new internal state it will be considering both the current environment state and the knowledge level. The dynamics of the new internal state is a combination of the dynamics of the environment and the effect on the knowledge level each action produces at a given state. Thus, the value of internal states incorporate the value of modifying the state of the environment as well as the knowledge level. The approach is implemented an evaluated in a version of the double integrator in which some parameters are unknown to the agent. Additionally, the approach is demonstrated in a simulated version of the Mars rover.

1.6.4 Computational Procedures

How can an agent improve the computation of the value function and the policy?

The most generic method to compute the value function is due to Bellman and it is called *dynamic programming* (Bellman, 1957). There are two variants of dynamic programming methods: value iteration and policy iteration. In value iteration an initial (incorrect) value function is used to produce a new, more accurate value function. With each iteration, the current value function estimate is successively approximated to the optimal value function. Policy iteration is analogous to value iteration with the difference that it operates on the policy function instead of the value function. Both dynamic programming methods are able to produce an exact solution to the sequential decision-making problem but require complete knowledge of the dynamics and reward functions and are computationally very expensive. On the other hand, researchers in the field of reinforcement learning have developed stochastic approximation methods that an agent can use to compute the value function directly as it interacts with the environment. The main idea consists of using experiences to progressively learn the optimal value function. The agent incrementally learns the optimal value function by continually exercising the current, non-optimal estimate of the value function and improving such estimate after every experience. More specifically, the agent can make a decision using the current, non-optimal estimate of the value function by selecting the action leading to the best value at the current state. Then, after observing the result of executing such action, the agent can use a reinforcement learning algorithm such as Sutton's TD(λ) algorithm (Sutton, 1988) or Watkins' Q-learning algorithm (Watkins, 1989) to improve the long-term estimate of the value function associated with such state and action. Thus, an agent without knowledge of the dynamics and/or reward functions has available two general strategies for learning using its own experience: it can use its own experience to learn models of the dynamics and reward functions and then apply a dynamic programming method to compute the value function, or it can use its own experience to compute the value function directly using one of the previously mentioned reinforcement learning algorithms. The former strategy has been term *indirect* or *model-based* learning and the latter *direct* or *model-free* learning.

In this dissertation we explore model-free and model-based approaches and discuss the advantages and disadvantages of both strategies from the lessons learned regarding the implementation. We discuss further the conditions that are required for each approach to excel and support these arguments by presenting some comparison results on simulated experiments with the double integrator, pendulum swing up, and mobile robot systems (see also Atkeson and Santamaría, 1997).

We study in detail and in isolation each of the first three issues: continuous domains, imperfect perception, and incomplete knowledge in Chapters 4, 5, and 6, respectively. In each chapter we concentrate the research to the issue in hand while simplifying the other issues to facilitate the study. In Chapter 7 we evaluate the heuristic approach presented in this dissertation and applied to the Mars rover problem, which is a complex problem that combines all the three issues. We study the fourth issue, limited computational resources, across these chapters because it is present in every problem we study. In Chapters 4 and 5 we use the model-free version of the heuristic approach, while in Chapters 6 and 7 we use the model-based version.

1.7 Contributions

This dissertation presents a new approach to solve sequential decision-making problems in which the environment is characterized by continuous state and action spaces and the agent has imperfect perception and incomplete knowledge. It describes the implementation of the approach using case-based methods and evaluates the implementation empirically in various domains including pendulum swinging and robotic navigation. In summary, the main contributions of this work are the following:

1. Use of value function approximators to solve sequential decision-making tasks in problems with continuous state and action spaces.
   (a) Description and evaluation of two new memory-based algorithms: instance-based and case-based.
   (b) Description and evaluation of a modular technique for implementing value function approximators having different degrees of resolution across the state and action space.
2. Case-based perception function for situation assessment without explicit state identification.
   (a) Extension of the case-based method to continuous domains.
   (b) Description and evaluation of method for combining case-based and abstract sequential decision-making in reactive controllers for mobile robots.
3. New framework for the suboptimal solution of sequential decision-making problems with agents using incomplete and dynamic knowledge.
   (a) Method for active learning in n-hypothesis scenarios.
   (b) Method for active learning of unknown parameters.

4. A systematic evaluation methodology for evaluating design decisions. We developed the methodology to evaluate the design decisions of the robotic system we used in Chapter 5. However, the methodology can be used to analyze and evaluate design decisions in other AI learning systems as well.

1.8 Organization of the Dissertation

Chapters 2 and 3 introduce and describe the generic sequential decision-making problem and the proposed approach to the solution, respectively. The basic tools used in the description are mathematical concepts that capture the roles and relationships of the agent, environment, and task without making any commitments to the particular problem or implementation method. Chapter 2 states the generic sequential decision-making problem and introduces the formulation for studying and evaluating the behavior of an autonomous agent acting in a dynamic environment. The formulation combines concepts from ecological psychology, control theory, reinforcement learning, and decision theory and describes the mathematical formulation used throughout the dissertation. Chapter 3 describes the optimal solution to the generic sequential decision-making problem in theory and discusses the difficulties associated with its practical implementation. The chapter describes the heuristic solution approach to the problem that we developed and that shares many of the fundamental characteristics of the optimal solution and allows its practical implementation. The heuristic solution approach is the foundation of all the methods presented in the subsequent chapters.

Chapters 4, 5, and 6 focus on the continuous domains, imperfect perception, and incomplete knowledge issues of the generic sequential decision-making problem respectively. While each of these three issues is present in every problem that we are interested in, each chapter focuses on a single issues in order to explore it in detail. Chapter 4 focuses on the issue of continuous domains and using function approximators to generalize the outcomes of individual experiences. The chapter describes a new approach for using value function approximators to solve sequential decision problems characterized by environments having continuous state and action spaces. Additionally, it describes and evaluates the implementation of two types of case-based value function approximators. Case-based function approximators are used in the following two chapters as the main representational tool to study imperfect perception and incomplete knowledge. Chapter 5 focuses on the issue of imperfect perception. The chapter describes a new case-based perception method the agent can use to characterize the state of the environment when the sensations are noisy and incomplete. Additionally, it describes a new solution for learning of parameter-adaptive reactive controllers for robotic navigation. Both approaches are implemented and extensively evaluated in a reactive robotic system that performs autonomous navigation. We evaluate the design decisions involved during the design of the robotic system using a new evaluation methodology which is described in detail in Chapter 8. Chapter 6 focuses on the issue of incomplete knowledge. It describes a new approach for the solution of sequential decision problems when the agent has incomplete knowledge. The approach is implemented and evaluated in a well-known problem in control theory: the double integrator. The issue of limited computational resources is present in every problem and we study it across chapters. In Chapters 4 and 5, we use the model-free version of the heuristic approach and in Chapters 6 and 7 we use the model-based version.

The main objective of Chapter 7 is to demonstrate the application of the proposed approach in a sequential decision-making problem with continuous domains, imperfect perception, and incomplete knowledge all together. The chapter describes the application of the heuristic approach in a complex sequential decision-making problem: the Mars rover. In this problem, an autonomous mobile robot must learn how to navigate in a terrain with obstacles and collect useful rocks. The mission of the rover is to collect as many rocks as quickly as possible while minimizing collisions with obstacles. The rover faces a sequential decision-making problem having the three main characteristics studied in this dissertation: continuous state and action spaces, imperfect perception, and incomplete knowledge. We show that our methods work together effectively to solve this problem.

Chapter 8 describes in detail the evaluation methodology we developed to evaluate the design decisions of the self-improving robotic navigation system we studied in Chapter 5. The methodology enables the study of the influence of design decisions and environment characteristics in the performance of learning systems. We use this methodology to analyze the design decisions in our system but it is general enough to evaluate design decisions in other complex learning systems.

Chapter 9 draws some conclusions from this research, and outlines directions for future research.

CHAPTER II

STATEMENT OF THE PROBLEM

The general problem of sequential decision-making is to find a strategy for selecting actions an agent can use to achieve effective behavior, specifically, a strategy the agent can use to decide what action to execute next taking into account the long-term benefit its execution will bring to the task. So far, the meaning of *effective* and *behavior* has been kept vague. This chapter formalizes the meaning of these words by describing formal models of behavior and performance. Together these models constitute the *sequential decision-making framework* that we use to study the problem. The sequential decision-making framework described in this chapter results from combining several ideas from ecological psychology (e.g., Brunswik, 1956; Gibson, 1979), control theory (e.g., Stengel, 1994; Bertsekas, 1995a), reinforcement learning (e.g., Kaelbling, Littman, and Moore, 1996; Sutton, 1988), and operations research (e.g., Martin, 1967; Raiffa and Schlaifer, 1972). The framework provides a mathematical foundation to study the sequential decision-making problem as an abstract entity regardless of the particular instantiation and distracting details of any specific problem. Additionally, the framework will be used to characterize the research problem precisely and the solution methods we put forward in this dissertation. The framework serves as the base for a classification of agents with respect to the characteristics of their policies and internal states and directs the research into this area.

The main objective of this chapter is to provide a formal mathematical description of the generic sequential decision-making problem. For this purpose, we first describe the abstract concepts for behavior and performance, then discuss the classification of agents that is based on the models, and finally put them all together in a formal definition of the generic sequential decision-making problem. Sections 2.1 and 2.2 define and describe the mathematical concepts about behavior and performance, respectively. Section 2.3 presents the classification of agents with respect to the capabilities of their decision-making strategies and internal states. Section 2.4 presents the formal description of the generic sequential decision-making problem. Section 2.5 concludes the chapter.

2.1 A Model of Behavior

We adopt the *ecological* perspective to study behavior (e.g., Effken and Shaw, 1992). The ecological perspective takes its name from an approach to psychology that was advanced by Brunswik (1956) and Gibson (1979). Although differing in the details of their respective views, these researchers shared the view of studying behavior as the result of the interaction between the agent (i.e., organism) and its environment. As Brunswik remarked:

> Both organism and environment will have to be seen as systems, each with properties of its own. ... Each has surface and depth, or overt and covert regions (Brunswik, 1957, p. 5).

Moreover, Brunswik indicated that organismic and environmental systems should be described in symmetrical terms. That symmetry is represented in what Brunswik called the *lens model* of behavior. The lens model follows the principle of parallel concepts, for each concept on one side is paralleled by a symmetrical concept on the other. Following this view and the representational formalism of systems theory,[1] we model behavior as a pair of interacting dynamical systems, one corresponding to the agent and the other to the environment. In system theory, a system may be broadly defined as an aggregation of objects united by some form of interaction or interdependence. When one or more aspects of the system change with time, the system is generally referred to as a *dynamical* system. Influences that originate outside the system and act on it are called *inputs*. The quantities of interest that are affected by the action of these external influences are called *outputs* of the system. In most cases the outputs depend not only on the current inputs but also on the past history of the inputs and hence that of the system. The concept of the *state* was introduced to capture this dependence without explicitly keeping track of the history of the inputs and to predict (deterministically or probabilistically) the future of evolution of the system given future inputs.

We model behavior by representing the environment and the agent as two interconnected dynamical systems. The environment outputs to the agent's inputs (i.e., the agent's sensations) and the agent outputs to the environment's inputs (i.e., the agent's actions). The environment delivers sensations to the agent which causes the agent to respond with actions. Additionally, each system has a local state that changes as a function of the signals delivered from the other. Behavior is the sequence of changes that occurs in each of the systems' local states due to this interaction. These sequences are referred to as *trajectories*. Figure 4 describes this model graphically.

[1] The ecological approach has much in common with systems theory, as Gibson (1979, p. 2) himself pointed out.

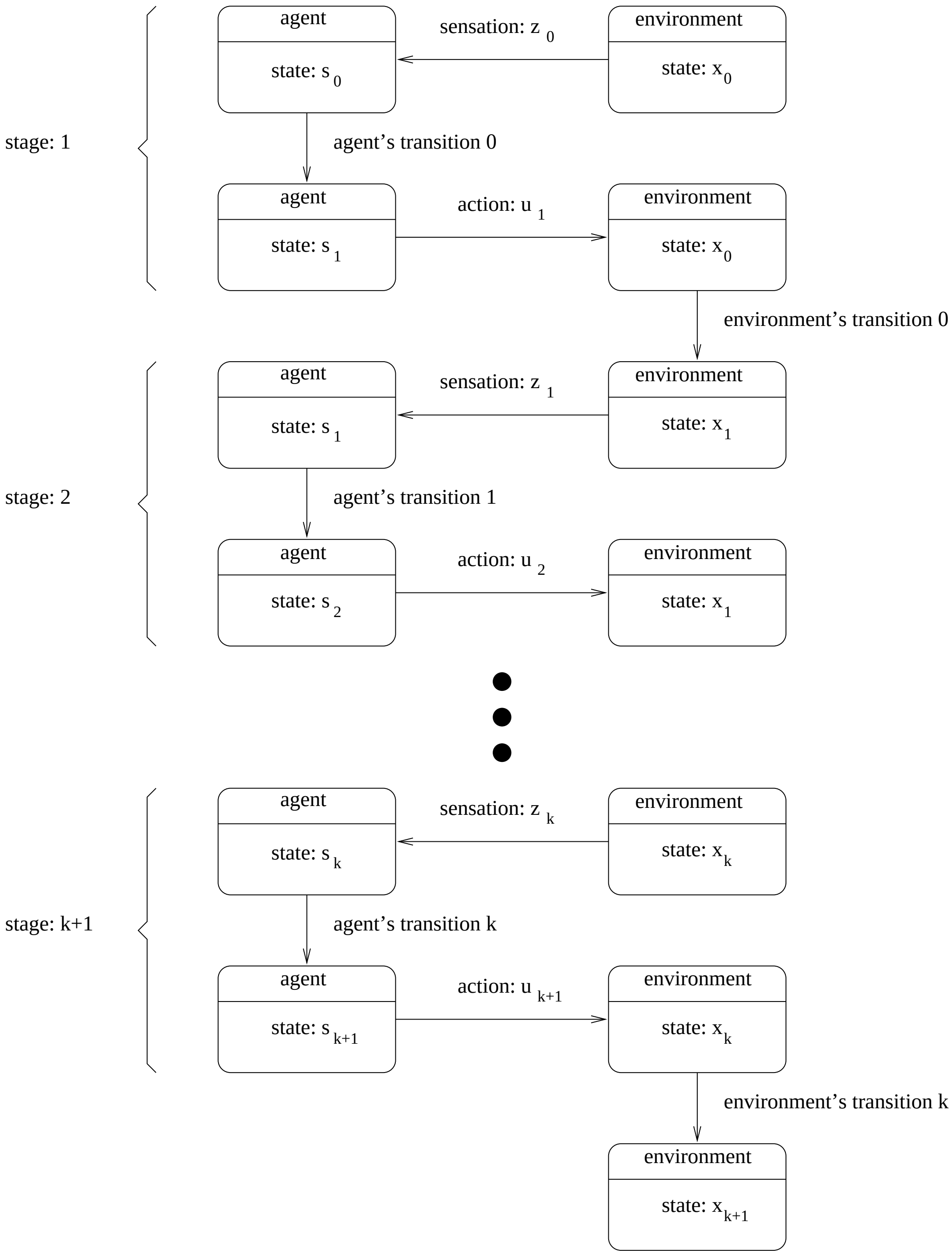

Figure 4: Behavior Model

Mathematically, the environment is modeled as a discrete-time dynamical system characterized by state x_t at time t and the dynamics function $x_{t+1} = F(x_t, u_t)$ that maps the current state of the environment x_t and agent's action u_t into the next state x_{t+1}. Thus, the input to the environment is the agent's action and the output is the state itself. The state of the environment represents the current situation of the external world given all previous actions and the function $F(\cdot)$ represents the "laws of nature" that characterizes the evolution of the state of the world.

Similarly, the agent is also modeled as a discrete-time dynamical system characterized by state s_t at time t and a dynamics function $s_t = \Psi(s_{t-1}, u_{t-1}, z_t)$ that maps previous state of the agent s_{t-1}, the previous action u_{t-1}, and the current sensation z_t into the current state s_t. The state of the agent represents the current situation of the agent given all previous sensations and actions, and the function $\Psi(\cdot)$ represents the agent's perception and reasoning mechanism that modifies the agent's state given the content of the previous state, the previous action, and current sensation (i.e., the previous action's consequence). Sensations are the output of sensor devices and are related to the current state of the environment by the *sensor function*, $z = H(x)$. The sensor function represents the operation the agent's sensors perform to extract the information from the environment. In ideal cases, the sensors deliver the exact state of the environment (i.e., $z = H(x) = x$). In most cases, however, the sensor's output only reports noisy or incomplete information about the state of the environment. Additionally, the agent is characterized by a strategy for selecting actions given its current state, called the *policy* function $u_t = \pi(s_t)$. The policy function represents the decision-making mechanism of the agent, which maps the agent's state to actions. Thus, the input to the agent is the sensation and the output is the action. Sensations enter the agent through the sensor function and modify the state of the agent through the perception function. The policy function maps the agent's state to the actions that are delivered as outputs (refer to Figure 1 in page 1).

The interaction between the agent and the environment forms a process that is best described as a sequence of discrete-time changes of their local states.[2] Each sequence forms a trajectory in its respective state space that depends (deterministically or probabilistically) on the initial states in both systems. When the outcomes of both systems are fully determined given each system's states and inputs, the process is called *deterministic*. Not all the processes, however, are deterministic, and, as a matter of fact, many of the interesting ones are not of this type. The class of processes in which the effect of the system's inputs (either sensations or actions)

[2]The mathematical models of the systems can be extended to continuous-time dynamics; however, the vast majority of hardware implementations are performed on digital machines capable of producing actions at very high, but at discrete-time intervals. Note that all the other quantities in the formulation such as states, actions, and sensations remain without constraints on the domain: they can be discrete or continuous.

is best described using probability density functions are called *stochastic*. Whereas the trajectories produced by deterministic processes are unique those produced by stochastic processes may vary across identical replications. However, both types of trajectories can be completely described using probability distributions that depend on the systems' states.

The formalism described above is central to the methods presented in this dissertation because it provides a consistent framework for analyzing systems of any degree of complexity. This formalism has been used by scientists and engineers in the field of system theory to successfully model and describe a wide range of physical, chemical, biological, or economic systems (e.g., Bhatia, 1967; Birkhoff, 1927).

2.2 A Model of Performance

In this dissertation we are concerned with the design of decision-making mechanisms that agents can use to achieve satisfactory performance. This section presents the framework used to evaluate the performance of an agent acting in its environment. Basically, throughout the dissertation we use the same performance criteria used in decision theory to evaluate multi-stage decisions: Each decision is assigned a reward, and a performance index for an entire sequence of decisions is computed using some *performance criterion* that combines the rewards into the performance index. Taken together, the reward function and the performance criterion represent a declarative description of the *task* because they described the demands made on the behavior.

The objective of the reward function is to evaluate the effectiveness of the agent's policy in bringing about the desired behavior of the agent-environment interaction. As the agent executes its policy, it modifies the state of the environment producing a trajectory that characterizes the evolution of the environment. The reward function assigns a benefit to each point in along the environment's state trajectory and provides a means to evaluate different policies. The benefit may reflect actual financial gain and be expressed in monetary units; it may indicate deviation from some ideal physical situation and be expressed in engineering units; or it may simply represent the passage of time in going from initial to final values of the state of the environment. In general, the reward function relates the *actual* behavior of the agent-environment interaction with the demands made on the *desired* behavior. A key aspect of this formulation is that decisions are not evaluated in isolation since good policies must be able to produce trajectories that balance the desire for immediate rewards with the undesirability of high future costs.

As a mathematical concept, the reward function takes the form of $r_{t+1} = R(x_t, u_t)$, where r_{t+1} is the reward the agent receives at stage $t+1$, which depends on the state of the environment, x_t, and the action, u_t, the agent performs at decision stage t.

The performance index, $V^{\pi}(s_0, x_0)$, measures the benefit of the entire trajectory that results from executing the policy, $u = \pi(s)$, when the agent and the environment start at states s_0 and x_0, respectively. The performance index has meaning in both deterministic or stochastic processes. In the deterministic case, the performance index is the return associated with the unique trajectory that results given the initial states. In the stochastic case, the performance index is the expected value of the performance index of all possible trajectories that may result given the initial states.

There are several criteria for combining rewards for individual actions into a performance index for the entire task. Three of the most common ones are finite horizon, discounted rewards, and average rewards (Kaelbling, Littman, and Moore, 1996).

The *finite-horizon* criterion computes the performance index by summing the rewards associated with each decision stage:

$$V^{\pi}(s_0, x_0) = E\left[\sum_{t=0}^{T-1} R(x_t, u_t)\right] \tag{1}$$

where $E[\cdot]$ denotes the expectation operator and T represents the total number of decision stages of the task. This criteria is commonly associated with tasks that have known duration or finite-horizon (i.e., T decision stages). An example of this type of task is the one that the Mars Pathfinder probe accomplished to reach Mars' surface. The Mars Pathfinder was launched on December 4, 1996 and during its flight to Mars the probe had 6 opportunities (including the launch itself) to correct its course and successfully approach Mars' atmosphere: three while near Earth and three as it approached Mars.[3] The time of occurrence of each opportunity was specified before the launch and during each opportunity the probe should engage its main thrusters in such a way that it corrects its course while minimizing the entire energy expenditure. The finite-horizon criterion is ideal for specifying the desired behavior on the landing of the Mars Pathfinder on Mars because the task has a known finite number of stages ($T = 6$) and the reward function can directly represent the energy expended in each TCM for $t < 6$ and the squared distance from the actual and desired landing location for $T = 6$.

The *discounted-rewards* criterion is commonly associated with tasks that have unknown or infinite duration (i.e., infinite horizon). A task with an infinite number of stages is never satisfied in practice, but constitutes a reasonable approximation for

[3]Each of the course correction opportunities is called a *trajectory correction maneuver* or *TMC* and occurred at specific points in time during the trajectory of the vehicle. These times were specified before the launch (e.g., TMC-1 should occur after 37 days of the launch and TCM-2 should occur 60 days after launch, etc.). Source provided by Jet Propulsion Lab's web pages; URL: `http://wwwmpf.jpl.nasa.gov/`.

problems involving a finite but very large number of stages. This criterion computes the performance index by summing discounted rewards:

$$V^{\pi}(s_0, x_0) = E\left[\sum_{t=0}^{\infty} \gamma^t R(x_t, u_t)\right] \tag{2}$$

The constant γ is a positive scalar with $0 < \gamma < 1$, called the *discount factor*. The meaning of γ is that future rewards matter less than the same rewards incurred at the present time. As an example, in a task where the agent invests money, the money received at the k-th period is depreciated to initial period money by a factor of $(1+r)^{-k}$, where r is the rate of interest; here $\gamma = 1/(1+r)$. This discount scheme assigns a *time-value* to the rewards just as the interest rate assigns a time-value for money in economic theory: the value of rewards received in the future (i.e., future value) is less than the value of rewards received in the present (i.e., present value).

The *average-rewards* criterion is also associated with tasks that have unknown duration or infinite horizon. The criteria computes the performance index as a long-run average of rewards:

$$V^{\pi}(s_0, x_0) = \lim_{T\to\infty} E\left[\frac{1}{T}\sum_{t=0}^{T-1} R(x_t, u_t)\right] \tag{3}$$

This criterion is used in infinite-horizon tasks where discounting is inappropriate. Such cases involve tasks with large number of stages in which the time the rewards are received does not matter as long as they are received. In this criterion there is no value associated to the time in which rewards are received, which means that a policy that achieves poor rewards at the beginning of the task followed by large rewards far away in the future has the same performance index than a policy that achieves the same rewards but in the opposite order.

It should be observed that reward functions and performance criteria can be combined to achieve multiple objectives simultaneously. In such cases, there may be a trade-off between conflicting demands, as in conducting a cost-benefit analysis. Additionally, all computations are performed under the expectation operator $E[\cdot]$. This guarantees a single value for the performance index for the trajectories associated with the initial states s_0 and x_0 for either deterministic or stochastic processes.

2.3 Classification of Agents

The previous two sections provided the general framework for describing and evaluating the behavior of an agent acting in its environment. The focus of this work is to study algorithms by which the agent can improve its behavior by making it more appropriate for a given task. There are two dimensions by which an agent can improve its performance and these two dimensions parallel the so-called "knowledge-level learning" and "symbol-level learning" introduced by Dietterich (1986). The former type of learning refers to improvement of performance that arise due to the change in the knowledge level of the agent. The latter refers to improvement of performance that arise because the internal processes of the agent are made more efficient, but no new information is gained. In this document, we will use the term *learning* to denote the knowledge-level learning and the term *adaptation* to denote symbol-level learning. Intuitively, learning involves the acquisition of new knowledge, and adaptation involves modifying the internal self of the agent. This section discusses the classification of agents according to these dimensions. The classification is useful for characterizing the research in this area.

Two main factors influence the actions the agent chooses to execute at any given stage: the agent's state and the policy. The state of the agent characterizes the current situation of the agent and the policy represents the decision-making strategy of the agent. Thus, at any given decision stage, the agent applies its policy to the current state to decide what action to execute next. The main objective of the policy is to map states to actions in such a way that produces efficient task performance when interacting with the environment. The implementation of the policy can range from a simple lookup in a table mapping the appropriate action for all possible states to a complex computation that depends exclusively on the entire state history. There are two possible dimensions that influence the selection of the action and they can be used to usefully classify agents. These dimensions corresponds to whether the policy remains fixed or changes with experience (i.e., adaptation), and whether the state is capable or not of incorporating new information about the unknown properties of the environment with new sensations (i.e., learning).

An agent using a fixed policy has a strategy for selecting actions that remains constant over time. Such an agent will respond with the same action whenever it finds itself in the same state, or, in case of an stochastic policy, with the same action probability distribution. Mathematically, a fixed policy $\pi(\cdot)$ remains constant over time and $\pi^{t_1}(s_{t_1}) = \pi^{t_2}(s_{t_2})$ whenever $s_{t_1} = s_{t_2}$. A special class of agents using fixed policies and states that do not incorporate new information are *reactive* agents (e.g., Agre and Chapman, 1987; Brooks, 1986; Arkin, 1989; Kaelbling, 1990; Maes, 1990). These agents are characterized by having their internal state identical to the current sensation, and their policies described in terms of stimulus-response type of functions. Mathematically, $s_t = z_t = H(x_t)$. Common examples of reactive agents using fixed policies are the agents based on the submsumption architecture (Brooks, 1986) and motor-schema based reactive navigation (Arkin, 1989). The internal state of these agents consists of the sensations delivered by the agent's sensors and the policies are preprogrammed by the designers using several task-achieving modules that interact to produce the final action at every stage. Pure reactive agents cannot change their knowledge level because their internal state only represents the current sensation; however, they can improve their performance by modifying their policy. Reactive agents using fixed policies achieve the same level of performance across replications.[4]

An agent using a changing policy has a strategy for selecting actions that evolves with time. Such an agent may not respond with the same action or action probability distribution at the same state in two or more different occurrences. In the context of this dissertation, agents with changing policies will be referred to as *adaptive* agents due to their capability for modifying their policy. These type of agents are common in reinforcement learning models (e.g, Kaelbling, Littman, and Moore, 1996; Sutton, 1988; Lin, 1992; Barto, Sutton, and Anderson, 1983; Tesauro, 1995). In reinforcement learning, agents use the outcome of every experience to modify their policy in such a way that the long-run sum of rewards (i.e., performance index) improves. Thus, adaptive agents improve the level of performance across replications by changing their policies. Mathematically, at two different times t_1 and t_2 the policy of the agent may change in such a way that $s_{t_1} = s_{t_2} \not\Rightarrow \pi^{t_1}(s_{t_1}) = \pi^{t_2}(s_{t_2})$, which means that the agent may not respond with the same action (or action probability distribution) at the same state. The performance of adaptive agents improve with experience; however, such improvement does not result because the agent explicitly seeks and incorporates new information about the unknown properties of the environment. In the context of this dissertation, agents capable of changing their knowledge level will be referred to as *learning* agents due to their capability to seek new information or perform knowledge-gathering actions.

A learning agent has the distinguishing characteristic of having an internal state capable of incorporating new information about the unknown properties of the environment with every incoming sensation. In general, the internal state of learning agents is more complex than the internal state of reactive agents because the former must include the same description used in the latter and, in addition, some description about the current level of knowledge of the agent. As with reactive agents, a learning agent may operate using fixed or adaptive policies. However, a learning agent with a fixed policy may achieve a higher level of performance than a reactive agent because the learning agent may collect new information as it interacts with the environment that causes a change in its knowledge level, and consequently, in its decisions. However, the decision mechanism of the agent remains the same,

[4] Fixed policies may be deterministic or stochastic. In the latter case the agent may produce different trajectory realizations across identical replications; however, the performance index is computed using the expectation operator $E[\cdot]$, which guarantees a single value for the performance index for all the trajectories associated with the initial states s_0 and x_0 in either the deterministic or stochastic case. Thus, a reactive agent using a fixed stochastic policy will achieve the same level of performance.

Table 2: Classification of Agents.

	fixed policy	**changing policy**
sensation state	reactive agent	adaptive agent
knowledge state	learning agent	learning adaptive agent

which means the agent cannot improve the way it executes actions to collect new knowledge or accomplish its task. Finally, a learning agent with an adaptive policy is able to change its decision-making mechanism and take advantage of better ways to collect knowledge, make progress on the task, or both. Table 2 summarizes the classification.

2.4 Formal Statement of the Problem

This section puts together the mathematical concepts of behavior and performance into a generic formulation of sequential decision-making problem. The statement of the problem is described in Table 3 and a diagrammatic description is shown in Figure 1 (page 1).

The problem is very general. The statement of the problem in Table 3 does not require that the states, actions, or sensations take a finite number of values or belong to a space of n-dimensional vectors. It does require, however, the assumptions made by the models of behavior and performance; namely, that the environment can be characterized as a dynamical system and that the demands of the task can be expressed as a reward function and a performance criteria.

The three main components of the framework (environment, task, and agent) capture the three main elements in the sequential decision-making problem. The environment structure describes the characteristics of the element subject to control, the task structure describes the demands made on the control, and the agent structure describes the constraints for the control. Each component brings about different dimensions to the problem:

- **Environment**
  - *State space*: refers to the domain of the environment's state X. It can be discrete or continuous.
  - *Action space*: refers to the action space U that acts upon the environment. It can be discrete or continuous.

Table 3: General Problem Statement.

Given:	
Environment	a discrete-time dynamic equation: $x_{t+1} = F(x_t, u_t)$ where x_t is an element of the state space X and u_t is an element of the action space U.
Task	a performance criterion: (e.g., finite-horizon, discounted reward, average rewards) and the reward function: $r_{t+1} = R(x_t, u_t)$ where r_t is a scalar value.
Agent	a sensor function: $z_t = H(x_t)$ where z_t is an element of the sensation space Z and a perception function: $s_t = \Psi(s_{t-1}, u_{t-1}, z_t)$ where s_t is an element of the internal state space S.
Determine:	a policy function: $u_t = \pi(s_t)$ such that the performance index is maximized.

  - *Dynamics outcome*: refers to the type of output of the environment's dynamics function $F(\cdot)$. It can be deterministic or stochastic.

- **Task**
  - *Task horizon*: refers to the number of decision stages T in the task. It can be finite or infinite.
  - *Rewards outcome*: refers to the type of output of the reward function $R(\cdot)$. It can be deterministic or stochastic.

- **Agent**
  - *Perception*: refers to the capability of the sensor function $H(\cdot)$ to capture the environment's current state. It can be perfect or imperfect.
  - *State content*: refers to the internal state space S, which represents the type of information contained in the agent's state. It can range from sensations only to environment's state estimates to any kind or form of knowledge.

The particular type of sequential decision-making problems we study in this dissertation are the ones having continuous state and action spaces, deterministic and stochastic function outcomes, infinite horizon, imperfect perception, and agent states consisting of environment's state estimates and knowledge represented using probabilities. Additionally, the approach to the solution occurs in the context of learning adaptive agents, which means that the agent starts with an initial, non-optimal policy and attempts to improve it after every experience. Table 4 presents the specific problem statement.

The problem statement in Table 4 fits the Mars rover example of Section 1.5. The state of the environment x represents the current layout of obstacles, targets, and rover in the terrain. The dynamics function $F(\cdot)$ represents the laws of motion of the rover (or any other entity) in the terrain, i.e., it determines the next layout of the terrain given the current layout and rover's action. The reward function $R(\cdot)$ penalizes the rover for the amount of energy spent at every stage and rewards the rover for retrieving an interesting rock. The discounted reward criterion may be used to evaluate the performance of the task in this case because it is desirable to retrieve rocks earlier rather than later. The rover uses a sensor function $H(\cdot)$ to collect information from the environment and the perception function $\Psi(\cdot)$ to change its current belief (i.e., internal state) based on the previous belief, previous action, and current sensation. The current policy function $\pi^k(\cdot)$ maps internal states to actions and the rover uses it to select action u_t at state s_t. Based on the outcome (r_{t+1}

Table 4: Specific Problem Statement.

Given:	
Environment	a discrete-time dynamic equation: $x_{t+1} = F(x_t, u_t)$ where x_t is an n-dimensional vector in $\Re^n$ and u_t is a p-dimensional vector in $\Re^p$.
Task	a performance criterion: (e.g., finite-horizon, discounted reward, average rewards) and the reward function: $r_{t+1} = R(x_t, u_t)$ where r_{t+1} is a scalar value.
Agent	an initial policy: $u = \pi^0(s_t)$, a sensor function: $z_t = H(x_t)$ where z_t is an m-dimensional vector in $\Re^m$, and a perception function: $s_t = \Psi(s_{t-1}, u_{t-1}, z_t)$ where s_t is of the form: $s_t = \{\hat{x}_t, \kappa_t\}$ $\hat{x}$ is an estimate of the state of the environment and κ is the knowledge level of the agent.
Use:	policy $\pi^k(\cdot)$ to select the action at the current state s_t.
Adapt:	improve $\pi^k(\cdot)$ into $\pi^{k+1}(\cdot)$ based on the outcome: r_{t+1} and s_{t+1}.

and s_{t+1}), the rover adapts the policy into $\pi^{k+1}(\cdot)$ in an attempt to improve the performance index.

The problem as stated in Table 4 contains the main characteristics discussed in Section 1.3 and summarized in Table 1 (page 11). The environment's state x and the agent's actions are real-valued variables, the sensor function $H(\cdot)$ does not reveal the complete state of the environment (i.e., $z \neq x$), and the knowledge level of the agent κ is incomplete and dynamic (i.e., the perception function $\Psi(\cdot)$ updates the internal state at every stage). Additionally, the agent has limited computational resources, which is the main reason for using a non-optimal policy $\pi^k(\cdot)$ and improving it with the outcome of every action. In the next chapter, we describe the theoretical optimal solution to the problem and discuss its intractability. We then describe the approach inspired by the optimal solution that we present in this dissertation. A surprising aspect of the approach is that its applicability depends very little on the nature of the state, control, and sensation spaces. The remaining chapters of the dissertation present implementations in autonomous mobile robots, pendulum swinging and balancing controllers, and other non-linear dynamic system controllers.

2.5 Conclusions

The behavior of an autonomous agent embedded in a dynamic environment can be formally described as a pair of two interacting dynamical systems: one corresponding to the agent and the other to the environment. The agent delivers actions to the environment and the environment delivers sensations to the agent, and the states of both dynamical systems change as a consequence of this interaction. The performance of any behavior can be evaluated using a reward function that assigns a scalar reward to every action the agent executes. The desired behavior is the one that maximizes the performance index, which is computed according to a given criterion. The ultimate goal of the agent is to learn a policy that maximizes the performance index.

The design of a decision-making agent involves the specification of the internal state, the perception function, and the policy. The representation of internal states may range from pure sensations to complex combinations of state estimates and knowledge content. In the former case the agent can only use the information contained in the current sensation to decide what to do next while in the latter case the agent can incorporate new information about the unknown properties in the environment and take advantage of it. Additionally, the agent may use a fixed policy to decide the action to execute at any given state. Designing a successful fixed policy is difficult because designers need to foresee every possible situation the agent can face as well as every possible outcome the agent may sense in response to every possible action. Accurate conceptual models and powerful programming tools may help alleviate such design process but the complexities of most real world environments quickly become overwhelming. A learning adaptive agent design, however, may overcome these difficulties because it is programmed to adapt its policy as it interacts with the environment. Designers can build a learning agent by specifying a task declaratively and providing an initial (most likely non-optimal) policy. Then, as the agent interacts with the environment, it can use its own experience to modify and improve its policy. The success of a learning adaptive agent design over a non-learning non-adaptive one depends on the mechanisms the agent uses to learn and adapt its policy with experience. As long as such mechanisms can improve the current policy more often than not the learning adaptive agent will accomplish its task better than the static one.

This chapter presented the formal mathematical description of generic sequential decision-making problems. The next chapter describes the optimal and heuristic solutions to the generic problem. Since both solutions are described mathematically in terms of the components of the framework, they are applicable to any particular problem instance. Thus, while the next chapter describes our solution in abstract terms, Chapters 4, 5, 6, and 7 describe the research and evaluation we performed by instantiating different versions of the heuristic approach on concrete problems.

CHAPTER III

APPROACH TO THE SOLUTION

In the previous chapter, we presented our formulation of the generic sequential decision-making problem. In the generic problem, the environment, the agent, the task, and their relationships are described as mathematical concepts that characterize the behavior of the agent embedded in the environment and the performance of the agent's policy with respect to the task. This chapter begins with the theoretical approach to derive the optimal agent that solves the generic problem. The derivation of the optimal agent is based solely on the mathematical description of the components of the framework. Therefore, the derivation of the solution is independent of the characteristics of the domain and the task. The optimal agent is guaranteed to select the optimal action at any stage during the sequential decision process considering both all the available information the agent has gathered until that stage and all the possible information the agent may collect in the future based on its current level of knowledge. However, in almost any application the optimal agent requires nearly infinite amount of storage and computational resources. Nevertheless, the derivation of the optimal solution provides useful insights for the development of a heuristic approach to the problem.

Next, the chapter discusses our heuristic approach to the problem. The heuristic agent is not guaranteed to select the optimal action at every stage; however, its design inherits many of the most important characteristics of the optimal agent. This chapter presents the formulation of the optimal solution to the generic sequential decision-making problem, discusses the difficulties associated with the optimal approach, and describes the new heuristic approach that we have developed. The heuristic agent design is the central object of investigation in this dissertation. The following three chapters describe the research and evaluation we performed on three central issues of the heuristic agent design in isolation: continuous domains (Chapter 4), imperfect perception (Chapter 5), and incomplete knowledge (Chapter 6), respectively. In Chapter 7, we describe the application of the complete heuristic agent design to a single problem: a Mars rover.

This chapter is organized as follows. Section 3.1 presents the formulation and optimal solution to the generic sequential decision-making problem. The optimal agent design consists of the optimal state representation, the optimal perceptual function, and the optimal policy. The optimal state representation is a quantity that completely characterizes the situation of the agent (including knowledge level) given the history of all previous sensations and actions. The optimal perceptual function is a procedure that updates the agent's state taking optimal advantage of every new information provided by the sensations the agent collects at every stage. The optimal policy is the strategy for selecting actions that maximizes the performance index such that no future observation is neglected and no past observation is forgotten. The optimal policy possesses the distinguishing characteristic of regulating its learning as required by the performance index. In practice, the implementation of the optimal policy is computationally intractable. Section 3.2 presents our heuristic solution to the sequential decision-making problem that retains the important characteristic of the optimal solution of regulating the control energy for learning and control purposes. The heuristic approach uses a finite dimensional quantity to represent agent's states and a function approximator to represent their associated long-term benefits or value function. The agent selects actions according to the current estimate of the value function and simultaneously adapts the value function to improve the estimate towards the optimal value. Section 3.3 concludes the chapter.

3.1 Optimal Approach

This section presents the formulation of the general decision-making problem, its optimal solution, and the difficulties associated with the implementation of the optimal agent. The general decision-making problem considered here consists of a generic environment, a generic task using the finite-horizon criteria, and the optimal agent. The only restriction imposed on the agent is on the sensation function $H(\cdot)$. This function defines the amount of information about the environment that enters the agent at every stage and it will be assumed as given. In this context, the optimal agent design must completely specify the agent's state (here, I_t), the perception function (here, $\Psi^*(\cdot)$), and the policy function (here, $\pi^*(\cdot)$). The optimal agent state is the quantity that best summarizes the current situation of the agent, the optimal perception function represents the procedure that best updates the agent's state given the most recent sensation at each stage, and the optimal policy is the procedure that selects the optimal action at every stage. The next subsections describe each of this components in more detail.

3.1.1 Environment and Task

Following the formulation presented in Chapter 2, the environment is a discrete-time dynamic system described by the dynamics function

$$x_{t+1} = F(x_t, u_t) \tag{4}$$

where $x_t \in \Re^n$ and $u_t \in \Re^p$ are the state of the environment and the agent's controlling action at stage t, respectively. The state consists of a set of variables that completely characterize the current situation of the environment and, together with the agent's subsequent decisions, fully determines (deterministically or stochastically) the evolution of the environment. The dynamics function is a vectorial function that maps the current state of the environment to the next state under the influence of the agent's current action.

The task is characterized by the reward function, $r_{t+1} = R(x_t, u_t)$, and by the discounted rewards with finite-horizon criteria.[1] The reward function associates a scalar to every action the agent may execute at any given state of the environment. The optimal performance index is given by

$$V^*(I_0, x_0) = E\left[\sum_{t=0}^{T-1} \gamma^t R(x_t, u_t)\right] \tag{5}$$

where I_0 and x_0 represent the starting state of the optimal agent and the environment, respectively. The ultimate goal of the optimal policy is to map the agent's state to the best possible action the agent can execute at that state. The best action at a given state is the one that together with all the subsequent actions maximizes the performance index subject to the constraints given by the dynamics of the environment.

The optimal agent perceives the current state of the environment using the sensor function

$$z_t = H(x_t) \tag{6}$$

where $z_t \in \Re^m$ is the sensation associated with state x_t. Sensations can be noisy or incomplete according to the capabilities of the agent's sensors as described by the sensor function.

[1]The derivation of the optimal solution is analogous for either finite-horizon or infinite-horizon tasks. In finite-horizon tasks, typically $\gamma = 1$ and the value function depends on the stage t. In infinite-horizon tasks, typically $0 < \gamma < 1$ and the value function is stationary (i.e., it does not depend on the stage t).

3.1.2 Information State and Optimal Perception Function

The state of the optimal agent captures the knowledge contained in the history of all the previous sensations and actions up to the current stage. This type of state is known in control theory as the *information state*, $s_t = I_t$, and it is the most general type of state the agent can use to represent its current situation. A common example of information state is the conditional probability density function of the current state of the environment given the current history of sensations and actions (Stengel, 1994; Tse, Bar-Shalom, and Meier, 1973; Bertsekas, 1995a; Bryson and Ho, 1975)

$$I_t = p(x_t \mid Z^t, U^{t-1}) \tag{7}$$

where Z^t and U^{t-1} represent the history of sensations and actions up to stage t. Mathematically,

$$\begin{aligned} Z^t &\equiv \{z_0, \ldots, z_t\} \\ U^{t-1} &\equiv \{u_0, \ldots, u_{t-1}\} \end{aligned}$$

The information state, I_t, can be viewed as a quantity that is equivalent to all a priori knowledge of the agent and the observation sequences Z^t and U^{t-1} in describing the future evolution of the environment (Stengel, 1994). The information state is important because it captures all the agent knows about the environment at a given point in time as well as all the predictions about the evolution of the environment the agent can infer from that knowledge base. As will be described in more detail next, the information state can guide an exhaustive search procedure over all possible policies to find the optimal one.

The optimal perception function is responsible for updating the information state every time the agent executes an action and receives a new sensation. Specifically, at any given stage t the perception function uses the previous information state I_{t-1}, the previous agent's action u_{t-1}, and the current incoming sensation z_t to construct the new information state I_t. For the optimal agent, the perception function represents a procedure that takes optimal advantage of every bit of new information provided by u_{t-1} and z_t given I_{t-1}. When the information state is the conditional probability density described in Equation 7, the optimal update procedure is Bayes' rule

$$\begin{aligned} I_t &= p(x_t \mid Z^t, U^{t-1}) \\ &= \frac{1}{c(z_t)}\, p(z_t \mid x_t) \int_{\Omega_{x_{t-1}}} p(x_t \mid x_{t-1}, u_{t-1})\, I_{t-1}\, dx_{t-1} \\ &= \Psi^*(I_{t-1}, u_{t-1}, z_t) \end{aligned} \tag{8}$$

where $c(z_t)$ is a normalizing constant that depends on the most recent sensation (see, e.g., Young, 1984).

3.1.3 Optimal Policy and Value Function

The optimal policy is a function of the feedback type that maps the agent's state to the best action the agent can execute at that state

$$u_t = \pi^*(I_t) \tag{9}$$

where I_t is the agent's state at stage t. The objective of the optimal policy given by Equation 9 is to maximize the performance index given by Equation 5. At every stage, the policy chooses actions according to the current value of the information state.

The key concept in the derivation of the optimal policy is the *value function.* This function maps the information state to the best expected performance index the agent can receive from this information state at any given stage. The optimal value function is important because it is used to guide the search for the optimal policy. Additionally, the agent can use an explicit representation of the optimal value function to select the best action at any given stage of the sequential decision task. The optimal value function is defined as:

$$\begin{aligned} V^*(t, I_t) &\equiv \max_{u_t, u_{t+1}, \ldots} E\left[\sum_{k=t}^{T-1} \gamma^{k-t} r_{k+1} \mid U^{k-1}, Z^k\right] \\ &= \max_{u_t, u_{t+1}, \ldots} E\left[\sum_{k=t}^{T-1} \gamma^{k-t} R(x_k, u_k) \mid I_k\right] \end{aligned} \tag{10}$$

With an explicit solution of the optimal value function the agent can solve the sequential decision-making problem by executing the following two procedures at every stage:

1. **Perception**: Update the information state based on the previous information state I_{t-1}, the previous action u_{t-1}, and the most recently received sensation z_t. The update uses Bayes' rule (Equation 8) to optimally take advantage of the new information provided by u_{t-1} and z_t given I_{t-1}.

   The updated information state captures all the a priori and collected knowledge the agent has after receiving the most recent sensation. Additionally, it serves as the basis for selecting the optimal action in the current stage. The agent must update the information state on-line at every stage.

2. **Decision**: Perform a one-stage lookahead search to find the optimal action to execute at the current stage. The search consists of finding the action that, according to the current information state, maximizes the expected sum of the immediate reward and the expected value of all the information states that may

result at the next stage from executing that action. Specifically, the optimal action satisfies:

$$u_t^* = \arg\max_{u_t} E_{z_{t+1}} \; [\, R(x_t, u_t) + \gamma V^*(t+1, I_{t+1}) \,|\, I_t \,] \tag{11}$$

where $E_{z_{t+1}} \; [\, R(x_t, u_t) \,|\, I_t \,]$ and $E_{z_{t+1}} \; [\, \gamma V^*(t+1, I_{t+1}) \,|\, I_t \,]$ represent the predictions of the immediate reward and discounted value of the resulting information state, respectively, based on the knowledge contained in I_t and action u_t.

The expectation operation in Equation 11, $E_{z_{t+1}}[\cdot]$, is over all possible sensations the agent could receive in the next stage by applying action u_t conditioned on the current information state, i.e., $z_{t+1} \,|\, I_t, u_t$; hence, the subscript z_{t+1} in $E[\cdot]$. In other words, each action results in an immediate reward and an expected information state for the next stage. The search computes the expected immediate reward and information states each action would produce based on the current level of knowledge, and selects the action that achieves the maximum sum of the expected immediate reward and value of the predicted information state. This is, by definition of the optimal value function, the optimal action to execute at stage t (see Bar-Shalom, 1990). The agent can perform the one-stage lookahead search on-line at every stage or else it can use a feedback table or built-in formula that associates every possible information state with the optimal action, $\{I_t \rightarrow u_t^*\}$. The table or formula may be constructed off-line, but the agent must use it on-line.

In principle, the sequential decision-making problem is solved once an explicit solution of the optimal value function is derived; however, in practice, the problem only begins with computing the optimal value function. The next section describes the key concepts involved in the derivation of the value function as well as a set of different methods for its computation.

3.1.4 Computing the Optimal Value Function

The principle of optimality and dynamic programming are the key concepts involved in computing the optimal value function. The complete derivation of the optimal solution is extensive and involved (see, e.g., Meier, 1965); therefore, this subsection only outlines the derivation and summarizes the results below.

The principle of optimality is a simple, but powerful concept that is the basis of the dynamic programming method (Bellman, 1957). It can be stated as follows: at any stage, whatever the present information and past decisions, the remaining decisions must constitute an optimal policy with regard to the current information state. In other words, if the set of decisions $\{u_{t+1}^*, u_{t+2}^*, \ldots\}$ taken from a given information state, say I_{t+1}, is optimal, and the decision u_t^* is the optimal action that takes the agent from information state I_t to I_{t+1}, then the set of decisions $\{u_t^*, u_{t+1}^*, \ldots\}$ is optimal with respect to the information state I_t.

The principle of optimality in the information state space yields the dynamic programming equation (also known as the Bellman's equation, Bertsekas, 1995a)

$$V^*(t, I_t) = \max_{u_t} E_{z_{t+1}} \left[R(x_t, u_t) + \gamma V^*(t+1, I_{t+1}) \mid I_t \right] \tag{12}$$

The dynamic programming equation is a powerful construct that simplifies a multi-stage optimization problem into a single-stage one. The maximum expected benefit the agent can achieve at stage t is the action that maximizes the sum of the expected immediate reward and the expected maximum benefit of the resulting information state given the action. The equation uses the principle of optimality and the "divide-and-conquer" approach to derive the value function: the optimal policy at stage t consists of the optimal action at stage t and the optimal policy at stage $t+1$. The optimal value function $V^*(t+1, \cdot)$ encapsulates all that is important for the computation of the optimal action at stage t; namely, the maximum benefit of the information state at $t+1$. Therefore, once $V^*(t+1, \cdot)$ is known, computing $V^*(t, \cdot)$ is equivalent to solving a single-stage optimization problem.

A crucial operation present in Equation 12 consists of predicting the information state of the next stage. For this purpose the agent uses the knowledge embedded in the information state at stage t and its perception function. The knowledge embedded in I_t and action u_t define the probability density function for the predicted sensation: $\hat{z}_{t+1} = z_{t+1} \mid I_t, u_t = p(z_{t+1} \mid I_t, u_t)$. The perception function defines the predicted information states based on the predicted sensations: $\hat{I}_{t+1} = I_{t+1} \mid I_t, u_t = \Psi^*(I_t, u_t, \hat{z}_{t+1})$. Thus, the expected discounted value of the resulting information state given action u_t (second term in Equation 12) is computed as

$$\begin{aligned} E\left[\gamma V^*(\hat{I}_{t+1}) \mid I_t\right] &= \gamma E_{z_{t+1}} \left[V^*(t+1, I_{t+1}) \mid I_t \right] \\ &= \gamma \int_{\Omega_{z_{t+1}}} V^*(t+1, I_{t+1})\, p(z_{t+1} \mid I_t, u_t)\, d_{z_{t+1}} \\ &= \gamma \int_{\Omega_{z_{t+1}}} V^*\left(t+1, \Psi^*(I_t, u_t, z_{t+1})\right)\, p(z_{t+1} \mid I_t, u_t)\, d_{z_{t+1}} \end{aligned} \tag{13}$$

The dynamic programming equation performs an operation similar to the "preposterior analysis" technique from the operations research literature (e.g., Raiffa and Schlaifer, 1972). This technique is usually formulated in two-step experiments where the first decision (here, u_t) is for information gathering from which a posterior information will result (here, z_{t+1}) that will be used to make the subsequent decision (here, u_{t+1} performed during the next stage based on I_{t+1}). Since by stage t the agent must select action u_t without knowing the actual outcome z_{t+1}, the agent must make its decision based on what it currently knows about the outcome's prediction. The prior (to the experiment) probability density of the (posterior) result of the experiment is called the "preposterior density" (here, $\hat{z}_{t+1} = z_{t+1} \mid I_t, u_t$).

Thus, one can say that the dynamic programming equation performs a preposterior analysis, which is "anticipation" (in a statistical sense, i.e., causal) of future information according to what the agent currently knows. The maximization operation implies that at every stage, the decision has to use the available "hard" information (I_t) and "soft" information ($\hat{z}_{t+1}$) about subsequent hard information ($\hat{I}_{t+1}$) and the associated expected discounted benefit ($\gamma V^*(t+1, \hat{I}_{t+1})$). This implies that optimal actions take into account not only what the agent has learned in the past but also what it may be able to learn in the future such that the performance index is maximized.

3.1.5 Computation Methods

The dynamic programming method exploits Bellman's equation (Equation 12) to find the value function induced by a given policy, or to find the optimal value function and the corresponding optimal policy. The key idea of the dynamic programming method is to monotonically approach the optimal value function using successive approximations, which can occur in two different functional spaces: value-function space or policy-function space. The former method is called *value iteration* and the latter *policy iteration*. Value iteration starts with an initial value function estimate $V_0(\cdot)$ and generates a sequence of new value functions $V_1(\cdot), V_2(\cdot), \ldots$, for each iteration where each value function is closer to the optimal value function. Policy iteration produces a similar sequence but in policy-function space. It starts with an initial policy $\pi_0(\cdot)$ and produces a sequence of policy functions $\pi_1(\cdot), \pi_2(\cdot), \ldots$, for each iteration. Each policy is guaranteed to perform better than the previous one. The two methods are described in more detail next. For simplicity, it will be assumed that the value function is independent of the stage t, which is the case in infinite horizon tasks because policies are assumed stationary.[2] For finite-horizon tasks, the derivation is similar but with the additional difficulty that the stage enters the procedure as another state variable.

[2] Stationary policies are those in which time does not form part of the state of the system; therefore, the decision does not depend on t.

3.1.5.1 Value Iteration

Value iteration is a method that computes successive approximations of value functions until convergence to the optimal value function is achieved (i.e., until the function satisfies Equation 12). In every iteration the current approximation $V_k(\cdot)$ is used to construct the next approximation $V_{k+1}(\cdot)$ using an update formula derived from Bellman's equation

$$V_{k+1}(I_t) = \max_{u_t} E_{z_{t+1}} \left[R(x_t, u_t) + \gamma V_k(I_{t+1}) \mid I_t \right] \tag{14}$$

The information state I_t is approximated with n variables $v_i, i = 1, \ldots, n$ and each variable is quantized to N_i levels. Each control variable $u_j, j = 1, \ldots, p$ is quantized to M_j levels. The successive approximation starts from an initial estimate and continues until the error between two consecutive iterations is less than some bound (see Bertsekas, 1995a, Section 7.2). For finite-horizon tasks the value function at the k-th iteration coincides with the optimal value function at the $T-k-1$ stage (i.e., $V_k(\cdot) = V^*(T-k-1, \cdot)$; see Larson, 1967).

Initially, $V_0(I)$ is found for all quantized variables v by evaluating $E\left[R(x,u) \mid I\right]$ for each quantized control u and choosing the maximum value by direct comparison. Next, for each quantized state I_t, each quantized control u_t is applied, and the next information state $\hat{I}_{t+1} = I_{t+1} \mid I_t, u_t = \Psi^*(I_t, u_t, \hat{z}_{t+1})$ is computed. The maximum discounted value at the next state, $E\left[\gamma V_0(\hat{I}_{t+1}) \mid I_t\right]$, is found by interpolation using the values of $V_0(\cdot)$ at quantized states. The quantity $E\left[R(x_t,u_t) \mid I_t\right]$ is computed directly. The sum of these quantities for each quantized control are then compared, and the maximum value is stored as $V_1(I_t)$. The procedure continues in this manner, with $V_{k+1}(\cdot)$ being computed in terms of $V_k(\cdot)$, for a finite number of iterations or until the difference between $V_{k+1}(\cdot)$ and $V_k(\cdot)$ is insignificant.

Value iteration generally requires an infinite number of iterations to converge,[3] although there are important special cases where it terminates finitely (see Bertsekas, 1995b, Section 2.2). Value iteration is guaranteed to converge to the optimal value function whenever the optimal value function is finite.

The total amount of updates (U) for a single iteration is $U = \left(\Pi_{i=1}^{n} N_i\right)\left(\Pi_{j=1}^{p} M_j\right)$, and the amount of storage required to keep $V(\cdot)$ is $\Pi_{i=1}^{n} N_i$. In practice, the computational and storage requirements become excessive when the value iteration method is applied to large problems. This characteristic is known as Bellman's *curse of dimensionality* (Bellman, 1957).

[3] The rate of convergence depends on the definition of convergence (see Howard, 1966).

3.1.5.2 Policy Iteration

Policy iteration is a method that computes successive approximations of policy functions until convergence to the optimal policy is achieved. Every iteration consists of two phases: policy evaluation and policy improvement. During policy evaluation, the value function $V^{\pi_k}(\cdot)$ associated with a given policy $\pi_k(\cdot)$ is computed. During policy improvement, a new policy $\pi_{k+1}(\cdot)$ is derived from $V^{\pi_k}(\cdot)$. The new policy is guaranteed to perform better than the previous policy.

The policy evaluation operation uses a recursive formula similar to the dynamic programming equation to compute the value function associated to the given policy

$$V^{\pi_k}(I_t) = E_{z_{t+1}} \left[R(x_t, \pi_k(I_t)) + \gamma V^{\pi_k}(I_{t+1}) \,|\, I_t \right] \tag{15}$$

A method similar to value iteration can be used to compute the associated value function; however, it does not require the maximization operation because the policy is responsible for supplying the actions at every stage. Once the associated value function is computed, a policy improvement operation is performed

$$\pi_{k+1}(I_t) = \arg\max_{u_t} E_{z_{t+1}} \left[R(x_t, u_t) + \gamma V^{\pi_k}(I_{t+1}) \,|\, I_t \right] \tag{16}$$

The total amount of computation of the policy iteration method is similar to the one in the value iteration method; however, policy iteration may converge very rapidly to the optimal policy.

3.1.6 Difficulties

The basic difficulty of the optimal solution occurs during the computation of the value function. The one-stage search (Equation 11) in the control procedure requires the use of the optimal value function in explicit form; however, very few problems admit an explicit (i.e., closed-form) solution to Equation 12 (see, e.g., Stengel, 1994). Additionally, the dynamic programming method, even though efficient, is subject to the "curse of dimensionality" (Bellman, 1957) because the dimension of the information state at best, increases linearly in time. This can be seen by analyzing the order of storage associated with the pair (Z^t, U^{t-1}) at a specific stage, say t. According to Equation 9, the pair consists of all the sensations and actions the agent has performed up to stage t. Since each sensation and action is a vector in $\Re^m$ and $\Re^p$ respectively, by stage t the pair would contain t sensations and $t-1$ action vectors resulting in a total of $tm + (t-1)p$ variables. Thus, any numerical computation method for the optimal value function will soon become inefficient due to the increasing number of variables it will need to consider.

Assuming an explicit solution for the value function exists, there are still additional problems in representing the information state. Using the conditional state probability density as the information state is unfeasible because, in general, such density has infinite dimensions at every stage. This forces the value function to have an infinite input domain, which makes its practical realization impossible. One may attempt to approximate the density as a table look-up type of function; however, even if this can be done, the required storage would be impractical and it would still not solve the dimensionality or the computational problems since, in general, the approximate optimal control law is non-linear in the information state.

In summary, there are three major difficulties that limit the applicability of implementing the optimal policy solution:

- **Dimensionality problem**: The information state is either infinite dimensional or finite, but grows with each stage. Thus, even with appropriate finite approximate quantizing, the number of quantization points for representing the information state will soon become too large to be handled by a computer of any size.

- **No explicit representation of the value function**: In general, Equation 12 does not have an explicit closed-form solution and a numerical method must be used to find a direct solution. However, even if the optimal value associated to a given information state can be efficiently computed, finding the optimal action at a given information state requires evaluation of an extremely large (possibly infinite) number of possible predicted information states. Thus, the procedure becomes practically impossible for any computer.

- **Computation**: Finding the optimal value function is a computationally expensive process. The dynamic programming method, even though efficient, is subject to the "curse of dimensionality" because the dimension of the information state increases linearly with each stage, which increases the number of variables the method needs to consider.

In the next section, a new heuristic approach will be presented that is based on the principle of optimality using an approximation of the information state and the value function. This heuristic approach to sequential decision making is a major contribution of this dissertation. The approach does not only take into account the information contained in past observations but also considers the future observation program and its associated statistics. In other words, the approach takes into account the information the agent has learned in the past and the new information the agent may learn in the future such that the performance index is maximized.

3.2 Heuristic Approach

In the previous section it was noted that the main difficulties in implementing the optimal policy are: (1) the information state is either infinite dimensional or finite, but grows with at each stage; (2) the optimal value function associated with the information state is generally not an explicit function; (3) the methods available to find the optimal value function are computationally expensive. Thus, a reasonable suboptimal approach would be to: (1) approximate the information state such that the dimension of its space stays constant across all stages; (2) approximate the optimal value function with a function approximator; (3) asymptotically learn the optimal value function using a computationally inexpensive stochastic approximation procedure. The following subsections describe in more detail each of these issues. Together they constitute the proposed heuristic solution approach to the sequential decision-making problem.

This section describes the derivation of the optimal control problem for sequential decision tasks in the context of non-optimal state representation and perception function. The non-optimal state representation is any finite dimensional quantity that summarizes the past and characterizes the future evolution of the environment in a wide sense. The non-optimal perception function represents the process responsible for updating the state at every stage based on the content of the state at the previous stage, the previous action, and the current sensation. The particular implementation of the state representation and perception function depends on the design of the agent; however, the heuristic solution derivation is independent of the implementation and the nature of the environment or task. The formalism attempts to find the optimal solution given the constraints of the non-optimal state and perception function. The derived solution is suboptimal but possesses the same characteristics of the optimal solution. Namely, it explicitly considers the short- and long-term effects each action produces on the current state of the environment and level of knowledge in the agent and selects actions sequentially considering the information the agent has observed in the past and will collect in the future in order to maximize the performance index.

3.2.1 Internal State and Perception Function

The information state is a quantity that completely characterizes all the knowledge the agent has at a particular stage. A reasonable approximation of the information state should consist of a finite dimensional quantity that summarizes the past and characterizes the future evolution of the environment in a wide sense. This quantity will be referred as the *internal state* of the agent. The internal state exists inside the agent and explicitly represents both the current state of the environment (or its best estimate based on past sensations) and the uncertainty or knowledge level of the unknown properties of the environment. Additionally, the internal state drives the agent's action-selection mechanism. Thus, when an agent makes a decision it considers not only the immediate reward and change in the state of the environment such decision would produce, but also the change in what it currently knows about the partially known properties of environment.

To simplify the discussion, it will be assumed that the internal state, s_t, is composed of two separate finite dimensional quantities: an estimate of the state of the environment, $\hat{x}_t$, and a suitable representation of the knowledge level for the unknown properties, κ_t.[4] In other words, the internal state of the agent is given by

$$s_t = \{\, \hat{x}_t, \kappa_t \,\} \tag{17}$$

The vector-valued variable, $\hat{x}_t$, represents the agent's best estimate of the state of the environment at stage t based on a priori knowledge and previous sensations. The variable, κ_t, represents the agent's partial information about the unknown characteristics of the environment at stage t and measures the amount of study of the environment. For example, this variable may contain probability distributions of the unknown characteristics and, as the agent collects information, the a posteriori probability distributions may gradually approach to the actual characteristics of the environment. The distinctive property of this type of sequential decision-making problems is that the rate of change in κ_t depends on the strategy of the agent.

The internal state plays the same role as the information state I_t in the optimal approach. The agent uses the internal state to select and execute an action based on a given policy $u_t = \pi(s_t)$. Then, it observes the incoming sensation (z_{t+1}) and constructs the new internal state using a recursive estimation equation based on Bayes' rule,

$$\begin{aligned} s_t &= \{\, \hat{x}_t, \kappa_t \,\} \\ &= \{\Psi_x(\hat{x}_{t-1}, \kappa_{t-1}, u_{t-1}, z_t), \Psi_\kappa(\kappa_{t-1}, \hat{x}_{t-1}, u_{t-1}, z_t)\} \\ &= \Psi(s_{t-1}, u_{t-1}, z_t) \end{aligned} \tag{18}$$

where $\Psi_x(\cdot)$ and $\Psi_\kappa(\cdot)$ denote the recursive estimation functions for the state and knowledge level, respectively. Function $\Psi_x(\cdot)$ represents the procedure the agent uses to change the current estimate of the state of the environment based on the most recent sensation. For example, a mobile robot may update the current estimate of its position on the terrain according to the previous estimate, the previous action,

[4] The commitment of representing the internal state as two separate quantities is not a requisite of the heuristic approach. However, the discussion is easier to follow when such commitment is made because the role each quantity plays can be clearly seen.

and the most recent sensation consisting of the current distances to obstacles. The function $\Psi_\kappa(\cdot)$ represents the procedure that updates the knowledge level of the agent about the unknown properties of the environment. For example, a rock-grasping mobile robot may improve the probability of finding good rocks in a certain region of the terrain based on the previous probability and the most recent noisy rock-detection sensation. The details of the implementation of the perception functions depend on the representation of the internal state and the a priori knowledge of the agent. Examples are described in Chapters 6 and 7.

3.2.2 Optimal Policy and Value Function

The heuristic approach has intrinsic restrictions that originate due to the approximation of the information state $s_t = I_t$ with the finite quantity counterpart $s_t = \{\, \hat{x}, \kappa_t \,\}$, and the use of a possible non-optimal perception function. Nevertheless, there exists an optimal policy for the agent given these restrictions. The optimal policy in this case is a function that maps the agent's state to the best action the agent can execute at that state given the restrictions on the internal state and perception function,

$$u_t = \pi_h^*(s_t) \tag{19}$$

The objective of the policy is to maximize the performance index given by Equation 5. At every stage, the policy chooses actions according to the current value of the internal state. The basic difference between the policies associated with the optimal and heuristic approaches is in their domain spaces. While the policy in the optimal approach denotes the optimal strategy given the agent's optimal state representation and perception functions, the policy in the heuristic approach denotes the optimal strategy given the restrictions imposed by the state representation and perception function.

The optimal value function in the heuristic approach maps internal states to the best expected performance measure the agent can receive from such states. The optimal value function is defined as:

$$V_h^*(s_t) \equiv \max_{u_t, u_{t+1}, \cdots} E\left[\sum_{k=t}^{N-1} \gamma^k R(x_k, u_k)\right] \tag{20}$$

The optimal value function in the heuristic approach is also different from its counterpart in the optimal approach. The former measures the best performance index the agent can achieve given the restrictions imposed by the representation of the internal state and its associated perception function, whereas the latter measures the best performance index the agent can achieve without such restrictions.

As in the optimal approach, the heuristic agent can solve the decision-making problem by following the perception and decision procedures described earlier. During the perception procedure, the agent updates the internal state based on the content of the previous state, the previous action, and the current sensation. During the decision procedure, the agent chooses the action that maximizes the expected sum of the immediate reward and the expected value of all the internal states that may result from executing that action.

3.2.3 Adaptive Policies and Function Approximators

Computing the optimal value function in the heuristic approach is still very expensive. However, the agent may start performing with an initial policy and then improve it with experience. The main idea consists of using the agent's experience to progressively learn the optimal value function. More specifically, the agent can use a function approximator containing an initial estimate of the value function (random estimates will suffice) and decide the best action to execute next by performing the one-stage lookahead search (Equation 11). Then, after observing the outcome of executing each action, the agent can use a stochastic approximation procedure to improve the estimate of the value function by adapting the function approximator. Thus, the agent can incrementally learn the optimal value function by continually exercising the current, non-optimal estimate of the value function and improving such estimate after every experience.

A function approximator for the value function is of the form $\hat{V}_w(s)$, where s is the internal state and w is a set of adjustable parameters or *weights*. The function approximator provides the agent with an initial estimate of the value associated to each internal state. Thus, the idea consists of letting the agent decide the best action to execute next by performing the one-stage lookahead search (Equation 11) with the function approximator, and then using the outcome of each action to asymptotically improve the estimates towards their optimal values. One advantage of function approximators is that they can generalize the values associated with internal states the agent actually experiences to other regions of the internal state space. In this way, the agent can estimate the values of internal states that it has never experienced before. Also, function approximators are able to represent the optimal value function even when the variables in the internal state are continuous (i.e., real-valued variables). The details in the implementation of the one-stage lookahead search depends on the class of function approximator. We provide several examples in Chapters 4, 5, 6, and 7.

Thus, an agent using a finite dimensional internal state and a function approximator to represent the value function can solve the sequential decision-making problem by performing the following three procedures at every stage:

1. **Perception**: Update the internal state based on the content of the previous internal state s_{t-1}, the previous action u_{t-1}, and the most recently received sensation z_t. The update uses a recursive estimation rule (Equation 18) to take advantage of the new information provided by u_{t-1} and z_t given s_{t-1}.

   The updated internal state captures all the a priori and collected knowledge the agent has after receiving the most recent sensation. Additionally, it serves as the basis for selecting the action in the current stage. The agent must update the internal state on-line at every stage.

2. **Decision**: Perform a one-stage lookahead search to find the best action to execute at the current stage according to the current estimate of the value function. The search consists of finding the action that, according to the current internal state, maximizes the expected sum of the immediate reward and the expected value of all the internal states that may result from executing that action. Specifically, the best action satisfies:

$$\hat{u}_t^* = \arg\max_{u_t} E_{z_{t+1}} \left[R(x_t, u_t) + \gamma \hat{V}_w(s_{t+1}) \,|\, s_t \right] \tag{21}$$

   where $E\,[\,R(x_t, u_t) \,|\, s_t\,]$ and $E\left[\hat{V}_w(s_{t+1}) \,|\, s_t\right]$ represent the predictions of the immediate reward and value of the resulting internal state based on the knowledge contained in s_t, action u_t, and current estimate of the value function $\hat{V}_w(\cdot)$, respectively.

3. **Adaptation**: Adapt the current policy by adjusting the function approximator using a stochastic approximation procedure to asymptotically improve the value estimates towards their optimal values. According to the principle of optimality, the value function must satisfy the dynamic programming equation:

$$\hat{V}_w(s_t) = \max_{u_t} E_{z_{t+1}} \left[R(x_t, u_t) + \gamma \hat{V}_w(s_{t+1}) \,|\, s_t \right] \tag{22}$$

   where s_t and u_t are the internal state and action at stage t. Stochastic approximation procedures exploit this recursive relation to create an *update formula.* A formula the agent can use to adapt the weights of the function approximator after executing each action. More specifically, every time the agent selects action u_t and observes the next sensation z_{t+1} and reward r_{t+1}, it can verify whether Equation 22 holds or not by computing the error between the two consecutive predicted performance values: before executing the action, $\hat{V}_w(s_t)$, and after executing the action, $r_{t+1} + \gamma \hat{V}_w(s_{t+1})$. When the error is different from zero, the agent uses the update formula to adapt the weights w and improve the accuracy of the estimate of the value function at s_t. The agent can adapt the function approximator independently from the decision process and may adapt the weights once or many times at every stage.

In summary, the internal state is a heuristic representation of the information state with finite dimensions. It is a quantity that exists in the agent that summarizes the past and characterizes the future evolution of the environment in a broad sense. At each stage, the agent updates the internal state based on the actions it executes and the sensations it perceives using a recursive estimation procedure. Additionally, the agent uses the internal state in conjunction with the value function to select the best action to execute at each stage by performing a one-stage lookahead search. The agent can use a function approximator to explicitly represent the value function and function approximators use finite resources to represent an estimate of the value of internal states. The initial estimates of the function approximator may not be the optimal ones, but the agent can use the outcome of each action it executes to asymptotically improve the estimates toward their optimal values. The agent is able to improve the accuracy of the estimates by adjusting the set of weights of the function approximator with the update formula.

3.2.4 Adaptation Methods

Finding the optimal value function is a computationally expensive process. A reasonable approach is to let the agent select actions using an initial estimate of the value function and in the meantime use some stochastic approximation procedure to asymptotically adapt the estimates towards their optimal values. All the methods used to update the value function are based on the dynamic programming equation (Equation 22); however, the methods differ in the nature of the source of the data they use to perform the updates. There are two general approaches: *model-based* and *model-free*.

In the model-based approach, adaptation methods use a model of the environment $\hat{F}(\cdot)$ to update the value function, either a model that is given a priori, or one that is estimated on-line using a system identification procedure. In the model-free approach, adaptation methods do not use a model of the environment and therefore do not have access to the environment's state transition function. In the field of control theory, model-based and model-free approaches are referred to as *indirect* and *direct* approaches, respectively. The question of which approach is better is a matter of some debate in reinforcement-learning and control theory communities (Kaelbling, Littman, and Moore, 1996).

We develop two different stochastic approximation procedures to adapt the estimates of the function approximator. They are inspired by the model-based and model-free approaches and borrow many of the techniques used by previous researchers. The two procedures differ in the initial assumptions, amount of computation (i.e., computational efficiency), and amount of data available from the actual system (i.e., data efficiency) they require to accurately approximate the value function. The first procedure is based on temporal difference methods (Sutton, 1988); it does not require a model of the dynamics function, uses computational efficient updates, but requires high amount of data. The second procedure is based on value iteration (Bellman, 1957); it is a dynamic programming method that requires a model of the dynamic function, uses computational expensive updates, but requires less amount of data (Atkeson and Santamaría, 1997).

3.2.4.1 Model-Free Procedure

Our model-free procedure uses temporal difference to adapt the value function. Temporal difference updates adapt the weights of a function approximator based on the error or difference between two consecutive predictions. Sutton (1988) defines a whole family of temporal difference update formulas called TD(λ), where $0 \leq \lambda \leq 1$ is a parameter used to measure the relevance of previous predictions in the current error.

An agent improves the current estimate of $\hat{V}_{w_t}(s_t)$ by adjusting the weights after every internal state transition using TD(λ) updates. For this purpose, every time the agent selects and executes action u_t at state s_t, observes the next sensation z_{t+1} and reward r_{t+1} that results from executing that action, and constructs the next internal state $s_{t+1} = \Psi(s_t, u_t, z_{t+1})$, it uses the following TD(λ) formula to update the weights of the function approximator,

$$\Delta w_t = \alpha \left(r_{t+1} + \gamma \hat{V}_{t+1} - \hat{V}_t \right) \sum_{k=0}^{t} (\lambda\gamma)^{t-k} \nabla_{w_k} \hat{V}_k \tag{23}$$

where $\hat{V}_{t+1} = \hat{V}_{w_t}(s_{t+1})$, $\hat{V}_t = \hat{V}_{w_t}(s_t)$, $\nabla_{w_k}\hat{V}_k$ is the gradient of $\hat{V}(\cdot)$ with respect w and evaluated at state s_k, and α is a learning rate. The interpretation of Equation 23 is as follows: $\hat{V}_t$ and $\hat{V}_{t+1}$ represent the current estimate of the long-term outcome (i.e., performance measure) at states s_t and s_{t+1}, respectively. The term $\left(r_{t+1} + \gamma \hat{V}_{t+1} - \hat{V}_t \right)$ represents the error incurred by $\hat{V}_t$ in predicting the future according to $\hat{V}_{t+1}$. In other words, the value of state s_t should be equal to the immediate reward r_{t+1} plus the value of the next state s_{t+1} properly discounted. In case of error, the weights are proportionally modified in the direction of the gradient ∇_{w_t} in order to maximally reduce error. The discounted sum of the previous gradients ∇_{w_k} are also credited for the current error although their influence decays exponentially with λ.

The convergence of TD(λ) has been proved under different conditions and assumptions. Watkins (1989) shows that estimates asymptotically converge to their optimal values in systems having discrete and finite state and action spaces when TD(0) is used to perform the updates. A condition for the convergence is that all states are visited and all actions are executed infinitely often. Tsitsiklis and Van Roy (1997) show that the value function associated with a given policy converges for any linear function approximator and TD(λ) updates. There is no proof showing the convergence of TD(λ) for more complex function approximators, but this has not stopped researchers for trying these methods using different classes of non-linear function approximators. Successful results have been obtained with multi-layer neural networks (e.g., Lin, 1992; Rummery and Niranjan, 1994) and sparse coarse coding methods such as Cerebellar Model Articulation Controllers (*CMACs*) (e.g., Santamaría, Sutton, and Ram, 1996; Tham, 1995) and memory-based function approximators (e.g., Santamaría, Sutton, and Ram, 1996).

An efficient on-line implementation of the update rule is possible using "eligibility traces" (for details, see Singh and Sutton, 1996). The idea is to maintain the value of the rightmost sum in Equation 23 in a variable or *eligibility trace*, which can be easily updated at every stage using a recursive relation. The agent performs the update every time it executes an action. Thus, the method is computationally efficient since each update is computationally inexpensive. On the other hand, the data efficiency of the method can be very low since the agent requires large amounts of data (i.e., the agent must execute many actions) to make the value function estimates converge to their optimal values.

3.2.4.2 Model-Based Procedure

Our model-based procedure uses an approximation of the value iteration method. In general, internal states have real-valued variables and the direct application of value iteration may be computationally intractable (Bellman, 1957). Thus, the main idea is to calculate (possibly approximate) values for the optimal value function at a finite set of internal states, and then make a "least-squares fit" of these values to compute the weights of the function approximator. The values for the finite set of states are estimated using the value iteration method and the current value function approximation.

The model of the dynamics function $\hat{F}(\cdot)$ represents an inaccurate version of the true function $F(\cdot)$ based on the knowledge level κ_t in state $s_t = \{\hat{x}_t, \kappa_t\}$. The agent can use the model of the dynamics, the sensor, and the perception functions to predict its next internal state $\hat{s}_{t+1}$ (or probability distribution of $\hat{s}_{t+1}$) given it performs action u_t. The operation consists of three steps: First, the agent predicts the next state of the environment using the model, $\hat{x}_{t+1} = \hat{F}(\kappa_t, \hat{x}_t, u_t)$. Second, the agent predicts the next sensation given the new state of the environment $\hat{z}_{t+1} = H(\hat{x}_{t+1})$. Finally, the agent simulates perceiving the predicted sensation to compute the predicted internal state $\hat{s}_{t+1} = \Psi(s_t, u_t, \hat{z}_{t+1})$. In this way, the agent can simulate the execution of any action at any stage and determine its next state. This operation is fundamental in the value iteration method.

Initially, a set $S = \{s_t^i\}$ of initial states are selected. Next, for each state s_t, each quantized control u_t is applied, and the next state $\hat{s}_{t+1}^i = s_{t+1}^i \,|s_t^i, u_t = \Psi(s_t^i, u_t, \hat{z}_{t+1})$ is computed using the dynamics function model, where $\hat{z}_{t+1}^i = H(\hat{F}(\kappa_t^i, \hat{x}_t^i, u_t))$. The maximum discounted value at the next state, $E\left[\gamma V_{w_k}(\hat{s}_{t+1}^i)\,|s_t^i\right]$, is found using the current estimate of the value function $V_{w_k}(\cdot)$. The quantity $E\left[R(x_t^i, u_t)\,|\,s_t^i\right]$ is computed directly. The sum of these quantities for each quantized control are then compared, and the maximum value is stored as V_{k+1}^i. Finally, a least-squares fit procedure (e.g., regression; see Neter, Wasserman, and Kutner, 1989) is used to compute the new set of weights w_{k+1} according to the set of values V_{k+1}^i

$$\min_{w_{k+1}} \sum_i \left(\hat{V}_{k+1}^i - \hat{V}_{w_{k+1}}(s_t^i)\right)^2 \tag{24}$$

The procedure continues in this manner for K iterations. The quality of the approximation depends on the set of internal states S, the number of iterations K, and the accuracy of the model of the dynamics $\hat{F}(\kappa, x, u)$ in modeling the true dynamics function $F(\cdot)$. The larger the number of internal states in S, the better quality of the generalization of states outside S since the numbers of points used to fit the function approximator increases. The larger the number of value iterations the better the approximation of the values associated to the states in S since every iteration approaches the estimates to their optimal values. The better the accuracy of the model the less the computed value function differs from the real one. However, the computational cost of the method is directly proportional to the number of states in S and the number of value iterations K. Thus, there is a compromise between the accuracy of the value function approximator and the computational complexity of the value iteration method. When good or perfect models of the dynamics function are available, the data efficiency of the value iteration method can be very high, as little or no data from the environment is required to compute the optimal value function. At every stage, the agent performs the final action that results from extensive mental simulations (i.e., value iterations) using the dynamics model. On the other hand, the computational cost can be very high if many iterations are required since each value iteration is expensive.

3.3 Conclusions

In this chapter, we described the approach for the solving generic sequential decision-making problems optimally, discussed the difficulties associated with the optimal solution approach, and put forward a new heuristic solution approach that attempts to avoid these difficulties. The heuristic approach inherits the important characteristic of the optimal solution for appropriately distributing the control energy for learning and control purposes. The approach is an approximation based on the principle of optimality that uses finite storage resources to represent the information state and the optimal value function. Additionally, it uses finite computational resources to adapt the value function as the agent performs.

Three basic concepts support the optimal solution of any sequential decision-making problem: the information state, the value function, and the associated computational procedures. The information state is the quantity that characterizes the current state (including knowledge) of the agent. The value function is the instrument that assigns long-term benefits to information states. The computational procedures are responsible for computing/adapting the value function and let the agent select actions. At every stage, the agent uses the value function to select the best action to execute next, which is the action that produces the best sum of the immediate reward and value of the resulting information state. Since the information state captures the knowledge level of the agent and the value function assigns long-term benefits to information states, the agent is always optimally balancing the energy it uses to make progress on its task and learn more about the environment. However, a complete representation of the information state and a perfect value function are impossible to obtain in almost any problem. The heuristic approach to the solution inherits all the characteristics of the optimal approach by providing practical approximations to the information state and value functions and incrementally improving a non-optimal value function as the agent performs.

In this chapter, we presented a heuristic solution approach to the generic sequential decision-making problem. The approach is described in terms of the mathematical concepts provided by the framework; therefore, it is independent of the particular problem instance to which is can be applied. In Chapters 4, 5, 6, and 7, we describe the research and evaluation we performed using different versions of the heuristic approach on concrete problems. In the next three chapters, we study the following three central issues in isolation: continuous domains and function approximators, imperfect perception, and incomplete knowledge, in that order. Then, in Chapter 7 we address a complex problem that combine all the results of the previous chapters.

CHAPTER IV

CONTINUOUS STATE AND ACTION SPACES

A wide class of sequential decision-making problems are those in which the states and actions of the agent must be described using real-valued variables. Common examples of these class of problems are found in robotics where agents are required to navigate around obstacles, move manipulators, or grasp tools or objects to accomplish a task. The state of the agent in these examples corresponds to a set of variables representing positions and velocities of joints or body parts and the actions are a set of forces or torques that can be applied to the actuators. In this class of problems, the agent must select an action from infinitely many alternatives after every fixed time interval while basing its decision using the currently perceived state, which is also one of infinitely many possible states.

In this chapter we focus on the issue of continuous state and action spaces in particular. As described in Chapter 3, one of the main components of the heuristic solution is the *function approximator*. A function approximator is a tool used to represent the value function; it maps estimates of the long-term benefit associated with internal states. The agent uses the function approximator to select the best action to execute next. In problems with finite state and action spaces, the function approximator has as many parameters as there are values to store. This means that the function approximator can represent the value function very accurately but is not applicable for problems with continuous state and action spaces. In the latter case, the agent needs a function approximator that uses a more compact representation technique to store the value function. A *compact* function approximator is one that has fewer independent parameters than there are input-output mappings to store. A function approximator of this type must perform some sort of generalization in order to represent all estimates of the value function. In this chapter, we study the performance of agents using three types of compact function approximators (instance-based, case-based, and CMAC) in combination with a model-free adaptive method. The instance- and case-based function approximators described in this chapter are new. They have the distinguishing characteristic of allocating new memory resources dynamically and as required by the task, which makes them more robust to handle problems in which there is not enough a priori information to design a custom-fit function approximator. The chapter presents the result of extensive experiments in the double integrator and pendulum swing up problems that demonstrate the approach.

In order to focus specifically on function approximators in sequential decision-making problems characterized by environments having continuous state and action spaces, in the experiments of this chapter, we use problems in which the agent has perfect perception and complete knowledge of the environment. The following two chapters will describe the research we performed to study these two other issues separately.

Section 4.1 introduces the issues relevant to the design of function approximators. Section 4.2 describes in detail the three types of compact function approximators used in the experiments. Section 4.3 introduces the topic of resource preallocation in function approximators and describes a simple technique to obtain non-uniform resource preallocation while keeping the simplicity and computational efficiency of function approximators with uniform preallocation. The results of experiments performed with uniform and non-uniform sparse coarse-coded function approximators in two different sequential decision-making problems is presented in section 4.4 and section 4.5 analyzes these results. Section 4.6 concludes the chapter.

4.1 Issues with Function Approximators

In the heuristic approach, the agent uses a function approximator to represent an estimate of the long-term benefit associated with every state and action. The agent selects the best action according to these estimates and uses every the outcome of every action to adapt the long-term benefit estimates. Adapting the value function in continuous state and action spaces is very difficult. This is generally the case because it is very unlikely that the agent would experience situations exactly identical to those it has experienced before. Thus, the agent must be able to generalize the value of specific state-action combinations it has actually experienced to the situation it is currently facing so that it can make a good decision about what to do next. This is the general problem of case-based reasoning (Kolodner, 1993). Unless a closed-form solution of the optimal value function is known in advance, the choice of representation of the value function is a real challenge. The representation should be able to handle real-valued variables as inputs, precisely map the state-action pairs to their assigned values, use memory resources efficiently, support learning without too much computational burden, and generalize the immediate outcome of specific state-action combinations to other regions of the state and action spaces.

There are several issues related to the design of function approximators. This chapter explores the issues related to the following two questions in the context of sequential decision-making applied to agents with continuous state and action spaces:

1. **How can the agent use function approximators to generalize the outcome of experiences involving specific state-action pairs to other regions of the state and action spaces?**

   A common approach that have been used to represent the value function is to quantize the state and action spaces into a finite number of cells and aggregate all states and actions within each cell (e.g., Mahadevan and Connell, 1991; Sutton, 1996; Barto, Sutton, and Anderson, 1983). This is one of the simplest forms of generalization in which all the states and actions within a cell have the same value. Thus, the value function is approximated as a table in which each cell has a specific value. However, there is a compromise between the efficiency and accuracy of this class of tables that is difficult to resolve at design time. In order to achieve accuracy, the cell size should be small to provide enough resolution to approximate the value function. But as the cell size gets smaller, the number of cells required to cover the entire state and action spaces grows exponentially, which causes the efficiency of the learning algorithm to deteriorate because more data is required to estimate the value for all cells.

   Another approach is to avoid the problems associated with quantizing the state space altogether by using other types of function approximators, such as neural networks, that do not rely on quantization and can been used to generalize the value function across states (e.g., Lin, 1992; Rummery and Niranjan, 1994). The approach consists of associating one function approximator to represent the value of all the states and one specific action. For this reason, this approach is useful for systems with continuous states and discrete actions. In this way, the agent can generalize the value of one specific state and the given action to other states in the neighborhood and the same action. The approach can be extended to systems with continuous states and actions by quantizing the action space into a finite number of cells and associating one function approximator to each action cell as before. In this way, all actions within an action cell are aggregated into one and assigned the same value, but keeping the state continuous. However, as with the quantized table approach, when the number of action levels increases, more function approximators are required, and the efficiency of the learning algorithm deteriorates due to the inability of the function approximator to generalize across action levels. Moreover, it is not possible to use specific state-action experiences to improve the estimate of more than one function approximator, even when the action value is near the boundary of two different action cells.

In this chapter we describe an approach that extend the use of function approximators to systems with continuous states and action spaces. The approach consists of using only one function approximator over the combined state and action spaces. Therefore, it does not require quantization of the state nor action spaces and it can be used to generalize state-action experiences to other regions of the states and actions spaces. The presents an extensive evaluation of the feasibility of the approach in two widely known systems with continuous state and action spaces: double integrator and pendulum swing up.

2. **What is the effect of designing function approximators with non-uniform resource preallocation?**

   An important decision to consider when designing function approximators deals with resource preallocation across state and action spaces. Resource preallocation refers to the distribution of adjustable memory elements across the input domain. This is an important decision because such distribution affects the way the function approximator represents the value of state-action pairs. In general, the more the adjustable memory elements are available, the more the resolution of the function approximator. A good design strategy when no prior knowledge about the system is available is to preallocate resources uniformly across the entire state and action spaces. In this way, the function approximator will be able to represent the value function with the same resolution everywhere, which is desirable since there is no reason to favor some regions of the state-action space more than others. However, it is possible to design function approximators with non-uniform distribution of resources in order to achieve different degrees of resolution in different regions of the state-action space.

   There are several reasons why designing non-uniform function approximators may be more beneficial than designing uniform ones. First, it is unlikely that there is absolutely no prior knowledge of the system available; often the designers know, up to a certain degree, what regions of the state-action space will be used more often. In such cases, the premise that leads to the design of a function approximator with uniform resources preallocation no longer applies. Presumably, it may be an advantage to design the function approximator such that it uses fewer resources in regions of the state-action space that are known to be visited rarely while use more resources in other more heavily transited regions. Second, given fixed amount of resources, a non-uniform function approximator may lead to better performance and learning efficiency than that achieved with a uniform function approximator simply because the former is able to exploit the resources more efficiently than the latter. Finally, it may be possible to design function approximators that dynamically allocate more resources in certain regions of the state-action space and increase the resolution in such regions as required to perform effectively on-line.

In this chapter we describe a modular method that can be used to implement a non-uniform function approximator given the implementation of an uniform one. Thus, the method takes advantage of the conceptually simple and computationally efficient implementation of uniform function approximators and yet produces results equivalent to the non-uniform ones. The chapter presents the feasibility of this idea by implementing uniform and non-uniform versions of three types of sparse coarse-coded function approximators, CMAC, instance-based, and case-based, and compare the results in learning efficiency and final performance on the double integrator and the pendulum swing up.

4.2 Function Approximators

For the purpose of this chapter, we assume agents with perfect sensations and with an internal state that is a replica of the environment state. In this case, the perception function is very simple because it only needs to return the most recent sensation. The agent uses a model-free method to select actions and adapt its policy. For that purpose, it uses a variant of the value function that maps state-action pairs to their associated long-term benefit. This type of value function is known as the *action-value function* or *Q-function* (Watkins, 1989; Bradtke, 1994). The advantage of the Q-function is that the agent is able to perform the one-step lookahead search without knowing the dynamics functions of the environment. The disadvantage is that the domain of the Q-function increases from the domain of states $x \in X$ to the domain of state-action pairs $(x, u) \in X \times U$. When the agent knows the optimal Q-function, Q^*, it can select the optimal action using the following one-step search

$$u_t^* = \pi^*(x_t) = \arg\max_{u_t \in U} \{Q^*(x_t, u_t)\} \tag{25}$$

The optimal Q-function is sufficient for an agent to optimally perform the given task since the agent can use the one-step lookahead search (i.e., Equation 25) to select the optimal action at any given state. However, the agent does not have the optimal Q-function readily available from the beginning and it must learn such function using its own experience. Thus, the idea is to provide the agent with an initial estimate of the Q-function and let it decide the best action based on the current estimate. Then, the agent can use the outcome of each action to asymptotically improve the estimate towards the optimal action value. One way to accomplish this is to use a function approximator to represent Q-function. The function approximator provides the agent with estimates of the expected returns of every state-action pair even when the state and action spaces are continuous. More specifically, a function approximator for the Q-function of a given policy is of the form $\hat{Q}_w(x, u)$, where x is the state, u is the action, and w is a set of adjustable parameters or *weights*. Given a current estimate of $\hat{Q}$, the agent can use the one-step search to select the best action at some state x_t according to this estimate. Additionally, the agent may adjust the current estimate by modifying the weights of the function approximator using TD(λ) updates.

Function approximators are useful because they can generalize the expected return of state-action pairs the agent actually experiences to other regions of the state-action space. In this way, the agent can estimate the expected return of state-action pairs that it has never experienced before. Also, function approximators are able to represent the Q-function even when state and action spaces are continuous. However, note that a function approximator may not be able to accurately represent the Q-function for the entire state and action space due to its finite resources.

There are many classes of function approximators, each with advantages and disadvantages. The choice of a function approximator depends mainly on how accurate it is in generalizing the values for unexplored state-action pairs, how expensive it is to store it in memory, and how well it supports the computation of the one-step search and learning by TD(λ) updates.

4.2.1 One-Step Search

A function approximator for the Q-function can indicate the best action the agent can take at any given state. Such action is determined by performing the following one-step search:

$$\hat{u}_t^* = \arg\max_{u_t \in U} \left\{ \hat{Q}_{w_t}(x_t, u_t) \right\} \tag{26}$$

The actual implementation of the one-step search depends on the function approximator. A common technique used in problems with continuous state spaces is to use one function approximator for each action. Then, the one-step search can be performed by simply evaluating each function approximator at the given state. The best action is the one associated with the function producing the maximum value (see, e.g., Lin, 1992; Rummery and Niranjan, 1994; Sutton, 1996). However, such technique becomes quickly impractical when the number of actions increases due to storage implications and computational efficiency because the number of function approximators is proportional to the number of actions. Additionally, since each function approximators operates in isolation, the set of function approximators is

Input: state x_t and weights w_t
Output: best action $\hat{u}^*$
Algorithm:

1. For every action u_i from $u_{\min}$ to $u_{\max}$ at step Δ_u:
   (a) Evaluate $\hat{Q}_{w_t}(x_t, u_i)$.
   (b) Keep the action $\hat{u}^* = u_i$ that produces the maximum value.
2. Return $\hat{u}^*$.

Figure 5: One-Step Search Algorithm.

able to generalize the estimated return only across states. Also, the set must be evaluated using only the action values from a finite predetermined set.

It is possible to overcome the difficulties of this technique at expense of more elaborate computation while performing the one-step search. The Q-function may be represented with only one function approximator and the optimal action at a given state may be found using some numerical optimization method. For example, for function approximators that provide a direct measure of $\nabla_u \hat{Q}$, the best action could be found using a gradient descent method. The details of the implementation of the one-step search depends on the class of function approximator. However, the general algorithm shown in Figure 5 is applicable for all classes of function approximators.

The computational complexity of Algorithm 5 depends on the step size Δ_u, the cost of evaluating $\hat{Q}_{w_t}(x_t, u_t)$, and the dimensionality of the action space. Additionally, since the resolution of the search in the action space is Δ_u, the resulting action value is not truly continuous throughout the action space. However, the advantage is that only one function approximator is being used to represent the Q-function instead of $O(U/\Delta_u)$. Clearly, this is not the most efficient algorithm but it is conceptually simple and widely applicable.

4.2.2 Adaptation

An agent following policy $\pi(x)$ can improve the current estimate of $\hat{Q}_{w_t}$ by adjusting the weights after every state transition using TD(λ) updates. However, it is not possible to do so when the agent always chooses the action suggested by the given policy. The agent must also perform some random actions to be able to determine their effect. For this purpose, every time the agent selects action u_t at state x_t, and observes the next state x_{t+1} and reward r_{t+1} that results from executing that action, it uses the following TD(λ) formula to update the weights of the function approximator:

$$\Delta w_t = \alpha \left(r_{t+1} + \gamma \hat{Q}_{t+1} - \hat{Q}_t \right) \sum_{k=0}^{t} (\lambda\gamma)^{t-k} \nabla_{w_k} \hat{Q}_k \tag{27}$$

where $\hat{Q}_{t+1} = \hat{Q}_{w_t}(x_{t+1}, \pi(x_{t+1}))$, $\hat{Q}_t = \hat{Q}_{w_t}(x_t, u_t)$, $\nabla_{w_k}\hat{Q}_k$ is the gradient of $\hat{Q}$ with respect w and evaluated at the state-action pair (x_k,u_k), and α is a learning rate. The interpretation of Equation 27 is as follows: $\hat{Q}_t$ and $\hat{Q}_{t+1}$ represent the long-term outcome (i.e., discounted sum of rewards) associated with the current policy at states x_t and x_{t+1} respectively. The term $\left(r_{t+1} + \gamma \hat{Q}_{t+1} - \hat{Q}_t \right)$ represents the error incurred by $\hat{Q}_t$ in predicting the future according to $\hat{Q}_{t+1}$. In other words, the Q-value of state x_t should be equal to the immediate reward r plus the Q-value of the next state x_{t+1} properly discounted. In case of error, the weights are proportionally modified in the direction of the gradient ∇_{w_t} in order to maximally reduce error. The discounted sum of the previous gradients ∇_{w_k} are also credited for the current error although their influence decays exponentially with λ.

The rightmost sum has the following recursive property:

$$\begin{aligned}
\mathbf{W}_{t+1} &= \sum_{k=0}^{t+1} (\lambda\gamma)^{t+1-k} \nabla_{w_k} \hat{Q}_k \\
&= \nabla_{w_{t+1}} \hat{Q}_{t+1} + (\lambda\gamma) \sum_{k=0}^{t} (\lambda\gamma)^{t-k} \nabla_{w_k} \hat{Q}_k \\
&= \nabla_{w_{t+1}} \hat{Q}_{t+1} + (\lambda\gamma) \mathbf{W}_t
\end{aligned} \tag{28}$$

An on-line implementation of the update rule given by Equation 27 is possible by taking advantage of this recursive property. The idea is to maintain the value of the rightmost sum in Equation 27 in a variable, $\mathbf{W}_t$, which can be easily update at every step as $\mathbf{W}_{t+1} = \nabla_{w_{t+1}} \hat{Q}_{t+1} + \lambda\gamma \mathbf{W}_t$. Another approach to perform this computation is to store each component of the multidimensional variable $\mathbf{W}_t$ separately in an *eligibility trace*, e_t. The variable e_t represents the proportion of blame or “eligibility” the associated component in $\mathbf{W}_t$ has in the current error taking into account past information (for details, see Singh and Sutton, 1996).

In order to improve the Q-values, the policy must be chosen in such a way that both improve as the agent collects information. One way to accomplish this is to choose the action leading to the best estimate $\hat{Q}(x, \hat{u}^*)$ most of the time with ties resolved randomly. However, a small fraction, ϵ, of the time, an action is chosen randomly uniformly from the operating range. This policy is called *ϵ-greedy-policy* (see Sutton, 1996).

Input: initial set of weights w_0
Algorithm:

1. Initialization:
   (a) perceive current state: x_{curr}
   (b) select action: $u_{\text{curr}} \leftarrow$ ϵ-*greedy-policy*(x_{curr})
2. Accumulate gradient:
   $\mathbf{W}_t \leftarrow \nabla_w \hat{Q}(x_{\text{curr}}, u_{\text{curr}}) + \lambda\gamma\mathbf{W}_{t-1}$
3. Perform action:
   Execute u_{curr}, observe resultant reward, r, and next state, x_{next}.
4. Select next action:
   $u_{\text{next}} \leftarrow$ ϵ-*greedy-policy*(x_{next})
5. Adapt:
   $w_{t+1} \leftarrow w_t + \alpha(r + \gamma\hat{Q}(x_{\text{next}}, u_{\text{next}}) - \hat{Q}(x_{\text{curr}}, u_{\text{curr}}))\mathbf{W}_t$
6. Loop:
   $x_{\text{curr}} \leftarrow x_{\text{next}}$
   $u_{\text{curr}} \leftarrow u_{\text{next}}$
   If x_{curr} is a goal state then terminate else goto 2.

Figure 6: Gradient Descent Version of the SARSA Algorithm.

Both performing and learning are put together in the *SARSA*[1] algorithm (Rummery and Niranjan, 1994; Sutton, 1996). The algorithm uses the ϵ-greedy-policy and the TD(λ) updates to improve the estimates of the Q-function with every experience and consequently the performance of the agent (see Figure 6).

4.2.3 Sparse Coarse-Coded Function Approximators

The one-step search and SARSA algorithms outlined in the previous section apply to any kind of function approximator. However, not all function approximators are equally efficient. The following are the main characteristics a function approximator should exhibit to be useful and efficient:

- **Generalization**: Refers to the ability of the function approximator to accurately generalize the values for unexplored state-action pairs. In systems with continuous state and action spaces, it is unlikely that the agent experiences exactly the same situation it has experienced before. Thus, it is very important

[1]State-Action, Reward, State-Action.

that the function approximator is able to extrapolate the values of experienced state-action pairs to unseen state-action pairs.

- **Resolution**: Refers to the granularity of the function approximator and its capacity to represent different values in small areas of the input space. Thus, the ability of the function approximator to be able to accurately represent the Q-function depends on the resolution.

- **Storage**: Refers to the memory resources used to implement the function approximator. The more storage the function approximator needs, the less usable it becomes, owing to the cost associated with its maintenance. Additionally, in most cases, the storage compromises with the resolution of the function approximator, because the finer resolution, the larger are the storage needs.

- **Computational efficiency**: Refers to the complexity and efficiency of the one-step search and SARSA algorithms. An efficient function approximator must provide support for simple and efficient evaluation of state-action pairs and gradient computation because the agent performs these two operations extensively during the action selection and learning.

Our research focuses on sparse coarse-coded function approximators because their properties nicely support these basic characteristics. Sparse coarse-coded function approximators are the most basic model of associative memory. They represent the content associated with some input using several physical memory locations and generalize the content by sharing these memory locations. The more two different inputs share memory locations, the similar their contents will be. The resolution, storage, and computational efficiency vary according to the specific type of function approximator and its implementation. A detailed description of sparse distributed memory and related models is available in Kanerva (1993).

The following subsections describe three types of sparse coarse-coded function approximators (lookup tables, CMAC, and memory-based) and outline their advantages and disadvantages. The memory-based function approximators are new in this research and are used in the experiments described in the following chapters. Lookup tables are widely used in problems with finite state and actions (e.g.,Sutton, 1990; Kaelbling, 1990) and CMACs have been used in problems with continuous states and discrete actions (e.g., Sutton, 1996; Tham, 1995). We extend the application of CMACs to problems with continuous state and action spaces and used it as a benchmark to evaluate the new memory-based methods.

4.2.3.1 Lookup Tables

The lookup table is one of the most common function approximators used to represent the Q-function when the states and actions spaces are discrete. It represent the extreme in the class of sparse coarse-coded memory function approximators because each input is associated with only one memory location. More specifically, each state-action pair is an element of the table that is used to store the current estimate of the value associated with that pair. The size of the lookup table is $O(NU)$, where N and U are the number of states and actions respectively. Lookup tables represent the extreme of sparse coarse-coded function approximators in which only one memory location is used to represent the Q-value associated with every state-action pair. For system characterized by continuous state and action spaces, each element of the lookup table is mapped into a cell in the state and action spaces. Thus, all states and actions within a region or *cell* are aggregated into one table element and are all assigned the same value. This is the basic form of generalization that lookup tables implement.

Lookup tables quickly become impractical for several reasons. First, state and actions spaces must be quantize into a finite number of cells. It is often difficult to determine an appropriate quantization scheme to provide enough resolution (i.e., accuracy) and low quantization error. Second, the number of cells grows exponentially with the number of variables and geometrically with the number of quantization levels. This creates storage and maintenance problems. Third, the rate of convergence of the learning algorithm becomes extremely slow as the number of states and actions increases. Additionally, the quantized state and action spaces often create convergence problems because they form a non-Markovian representation of the dynamics of the system (see Moore and Atkeson, 1995).

Lookup tables efficiently support one-step searches and TD updates. The best action at a given state is found by indexing the table holding the state constant and performing a sweep across all possible actions values (Equation 25). Since there is only one parameter associated with every state-action pair in the lookup table, the TD updates are also easy to implement. Equation 29 shows a TD(0) update.

$$\Delta Q(x,u) = \alpha(r_{t+1} + \gamma \max_a \hat{Q}(y,a) - \hat{Q}(x,u)) \tag{29}$$

Watkins showed that under certain conditions this update rule can be used to asymptotically learn the optimal Q-values in systems having discrete state and action spaces (Watkins, 1989). Also, Peng and Williams described a way to use TD(λ) in a learning algorithm they call Q(λ), which results in a learning algorithm with faster convergence rate (Peng and Williams, 1994). However, the conditions for convergence in these algorithms rarely hold in quantized versions of continuous state and action spaces.

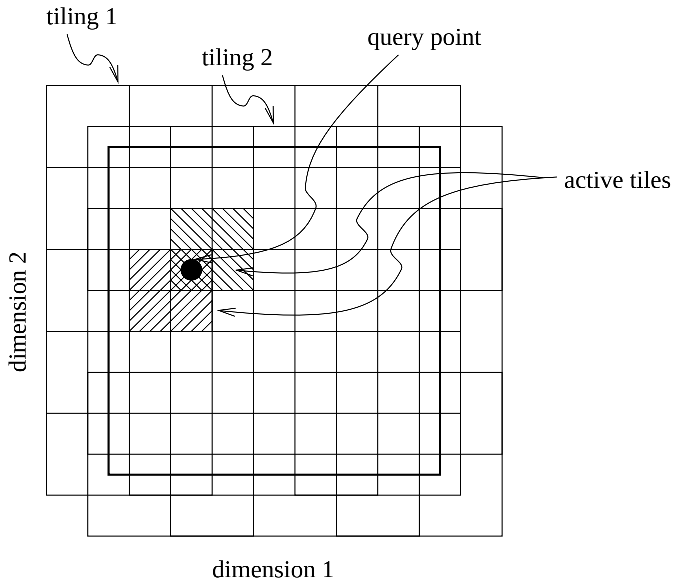

Figure 7: CMAC Function Approximator.
Every state-action pair, represented by the dot, activates one tile in each tiling. The sum of the weights of all the activated tiles represents the vale associated with that state-action pair.

4.2.3.2 Cerebellar Model Articulation Controller

Cerebellar Model Articulation Controllers, or CMACs, are a class of sparse coarse-coded memory that models cerebellar functionality (Albus, 1975). Each input or state-action pair activates a specific set of memory locations or *features*, the arithmetic sum of whose contents is the value of the stored Q-value. The CMAC takes advantage of the continuous nature of the input and is able to store the necessary data in a physical memory of practical size.

A CMAC consists of several overlapping tilings of the state-action space to produce the feature representation. A query is performed by first activating all the features that contain the state-action input and then summing the values of all the activated features. Figure 7 shows a bidimensional example of a CMAC organization. CMACs have been widely used in conjunction with reinforcement learning (Watkins, 1989; Sutton, 1996; Tham, 1995).

The size of a CMAC depends on the number of tilings and the size of each tiling. Tilings are usually large to provide enough resolution and they grow exponentially with the number of variables. A common trick to avoid large tilings is to use a consistent random hashing function to collapse a large number of tiles in a tiling into a smaller set. CMACs with uniform tilings efficiently support one-step searches since the evaluation of the Q-values is not computationally expensive. In naive CMAC implementations, TD(λ) updates are proportional to the number of tiles in the CMAC since each tile holds an eligibility trace. More efficient implementations keep a list of non-zero traces and perform the updates only on those tiles (e.g., Cishosz, 1996). A description of the structure and basic operations of the CMAC follows:

- **Tile structure**: A CMAC consists of a set of N tilings: $\{T_i, i = 1, \ldots, N\}$. Each tiling T_i consists of a set of N_i tiles or features that cover the entire input space of the tiling contiguously and without overlapping: $T_i = \{f_{ij}, j = 1, \ldots, N_i\}$. Each tile f_{ij} has a weight w_{ij} and an eligibility e_{ij}.

- **Tile selection**: For each tiling T_i, the tile f_{ij} containing the query point (x_q, u_q) is activated. All active tiles are aggregated in the set $F(x_q, u_q)$.

$$F(x_q, u_q) = \{f_{ij} \in T_i \mid (x_q, u_q) \in f_{ij}\} \tag{30}$$

- **Function evaluation**: All the weight associated with the tiles in $F(x_q, u_q)$ are summed together.

$$\hat{Q}(x, u) = \sum_{f_{ij} \in F(x,u)} w_{ij} \tag{31}$$

- **TD(λ) update**: The learning update is performed for each tile in the CMAC.

$$\Delta w_{ij} = \alpha(r_{t+1} + \gamma \hat{Q}_{t+1} - \hat{Q}_t)\, e_{ij} \quad \forall f_{ij} \in \text{CMAC} \tag{32}$$

  where $\hat{Q}_t$ and $\hat{Q}_{t+1}$ are the values of state-action pairs at (x_t, u_t) and (x_{t+1}, u_{t+1}) respectively.

- **Eligibility update**: The eligibilities of all the tiles in the CMAC are updated according to their current contribution to the value of the query point. In a query operation, every tile in $F(x_q, u_q)$ contributes equally to the value of the query point; therefore, the contribution of each tile is the fraction of the total number of tiles in $F(x_q, u_q)$:[2]

$$e_{ij} \leftarrow \begin{cases} \frac{1}{|F(x_q, u_q)|} & \text{if } f \in F(x_q, u_q) \\ \lambda\gamma\, e_{ij} & \text{otherwise} \end{cases} \tag{33}$$

[2]Equation 33 corresponds to replacement of eligibilities (see Singh and Sutton, 1996) for an explanation of the rationale behind the equation.

where $|F(x_q, u_q)|$ is a constant[3] that represents the number of tiles in the set $F(x_q, u_q)$. Note that each tile in the CMAC represents a weight of the function approximator and each active tile has a the same contribution, namely $\frac{1}{|F(x,u)|}$, in estimating the value for (x_q, u_q).

The CMAC function approximator uses computationally efficient methods for the selection of tiles when the size of each tile within each tiling is constant. The TD(λ) and the eligibilities updates are proportional to the number of tiles in the CMAC. This may deteriorate the performance of the CMAC in situations where some of the tiles are never used. The generalization and resolution of the CMAC depends on the number of tilings and the number of tiles within each tiling. The more number of tiles the better the resolution and the more number of tilings the better the generalization. However, the storage is proportional to the total number of tiles.

4.2.3.3 Memory-Based Function Approximators

Another class of sparse coarse-coded memory is memory-based. Although memory-based function approximators have not been widely used in conjunction with reinforcement learning, they are common in other tasks such as classification (e.g., Kibler and Aha, 1989) and robot control (e.g., Atkeson, 1991) (but see Ram and Santamaría, 1997; Peng, 1993; McCallum et al., 1995). In a memory-based function approximator, each memory element represents some of the state-action pairs or a *case* the agent has experienced before. A query is performed by first retrieving the nearest neighbors to the query point according to some similarity metric and then performing a weighted average of their Q-values. Additionally, new cases can be created and added to the memory when the nearest neighbors are too far away from a query point. In this way, the memory expands dynamically and on demand as new regions of the state-action space are being explored.

The size of the memory-based function approximator is dynamic. The memory starts empty and grows according to a *density threshold*, τ_d, that is used to determine when a new case should be added to the memory. Thus, when the distance of the nearest neighbor to a query point is greater than τ_d, a new case is added to the memory placed at the position of the query point. Small thresholds tend to produce function approximators with high resolution but with large amounts of

[3]In our CMAC implementation, there is only one active tile per tiling. Thus, the number of active tiles per query point is always the same and equal to the total number of tilings in the CMAC (i.e., $|F(x_q, u_q)| = N$).

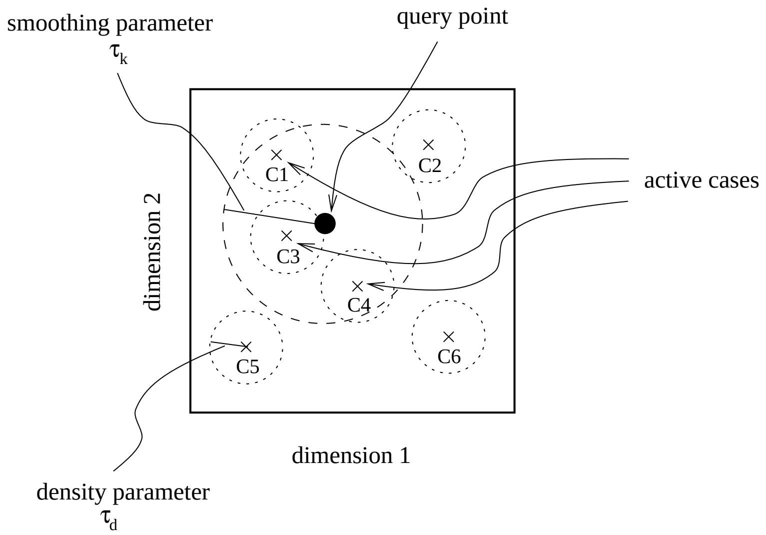

Figure 8: Memory-based Function Approximator.
Every state-action pair, represented by the dot, activates the cases that are close to it according to some similarity metric such as Euclidean distance. The weighted average of the values of all the activated cases represents the vale associated with that state-action pair. Cases that are closer contribute more to the final value than cases that are farther. New cases are created when nearest neighbors are too far away.

memory due to the high density of cases. The choice of the density threshold depends on the problem. Figure 8 shows a bidimensional example of a memory-based function approximator.

The choice of the similarity metric depends on the application. A common measure of similarity metric in continuous domains is the Euclidean distance. This metric is useful when the Q-function is expected to be continuous and smooth throughout the state-action space. To compute the value associate with some input, a weighted average among nearest neighbors is performed using a kernel function (e.g., Gaussian) with a *smoothing parameter*, τ_k, that controls the blending of values of nearest neighbors in the average. Generalization in memory-based function approximators results as a combination of the weighted average of nearest neighbors as determined by τ_k and the density of cases in memory as determined by τ_d. A good design is one in which the smoothing parameter is larger than the density parameter because in this way several cases contribute to the value of the query point while, at the same time, large portions of the domain are covered with fewer cases.

This section describes the details of two memory-based function approximators: instance-based and case-based. They differ in the structure of the memory elements, the selection of the nearest neighbors, and the procedures used to compute the weighted averages. The instance-based version uses less storage and it is useful when the value function is smooth and continuous. The case-based version uses more storage but it is able to perform better generalizations and achieve greater resolution than the instance-based version. The description of the structure and the basic operations each function approximator follows:

- **Instance-Based Function Approximator**

  The cases in the instance-based function approximator have a simple structure. Every case consists of a state and action combination the agent has used in the past and stores the approximated Q-value for such combination. Only one distance function is required to generalize the values of cases across the combined state-action space. However, this may produce over-generalization because only the cases that are similar to both the query state and the query action are used to determine the Q-value of a state-action pair.

  - **Case structure**: Every case consists of a some state-action pair (x_i, u_i) the agent has experienced in the past, the associated value Q_i, and the eligibility e_i. That is, $C_i = (x_i, u_i, Q_i, e_i)$.
  - **Nearest neighbors selection**: A similarity function is used to compute the distance d_i from the query point (x_q, u_q) to the point (x_i, u_i) associated with each case C_i in memory. Then, the set of nearest neighbors to the query point, NN_q, is determined using the following rule:

    $$NN_q = \{C_i \in \text{Memory} \mid d_i \leq \tau_k\} \tag{34}$$

  - **Function evaluation**: A kernel function, $K(\cdot)$ (e.g., Gaussian $K(d_i) = \exp(-d_i^2/\tau_k^2)$), is used to compute the relative contribution of case C_i in the value of the query point. The final value is the average of the nearest-neighbor cases weighted using their relative contributions:

    $$\hat{Q}(x_q, u_q) = \sum_{\forall\, C_i \in NN_q} \frac{K(d_i)}{\sum_j K(d_j)} Q_i \tag{35}$$

  - **TD(λ) update**: The learning update is performed to each case in memory.

    $$\Delta Q_i = \alpha(r_{t+1} + \gamma \hat{Q}_{t+1} - \hat{Q}_t)\, e_i \quad \forall C_i \in \text{Memory} \tag{36}$$

    where $\hat{Q}_t$ and $\hat{Q}_{t+1}$ are the values of state-action pairs at (x_t, u_t) and (x_{t+1}, u_{t+1}) respectively.

- **Eligibility update**: The eligibilities of all the cases in memory are updated according to their current contributions to the value of the query point. In a query operation, every neighboring case contributes according to its distance to the query point modulated by the kernel function; therefore, the contribution of each case is its weight in the weighted average computation:[4]

$$e_i \leftarrow \begin{cases} \frac{K(d_i)}{\sum_j K(d_j)} & \text{if } C_i \in NN(x,u) \\ \lambda\gamma\, e_i & \text{otherwise} \end{cases} \tag{37}$$

- **Case addition criterion**: A new case is added to the memory when the distance to the closest neighbor, $d_{\min} = \min\{d_i\}$, is larger than the threshold parameter τ_d.

- **Case-Based Function Approximator**

  The case-based function approximator use cases that contain more information than state-action pairs. Each case represents a region of the state space the agent has visited in the past, a set of actions the agent may or may have not executed in that region of the state space, and the set approximated Q-values associated to each action. Additionally, it uses two distance functions and two kernel functions to generalize and blend the values across the state and action spaces independently. For this reasons, the case-based function approximator has better resolution and generalization capabilities than the instance-based function approximator and may prove useful when the Q-function changes abruptly across actions within similar states. However, the case-based version requires more computation to perform the function evaluation and the TD(λ) updates than the instance-based version.

  - **Case structure**: Every case consists of a some state point x_i the agent has experienced in the past, an associated value Q_i that generalizes the Q-value across all the actions at that state, and the eligibility e_i. Additionally, the case stores a set of a set of N_i different actions u_{ij}, their associated Q-values Q_{ij}, and their eligibilities e_{ij} ($j = 1, \ldots, N_i$). That is, $C_i = (x_i, Q_i, e_i, \{u_{ij}\}, \{Q_{ij}\}, \{e_{ij}\} \mid j = 1, \ldots, N_i\ )$. This case structure explicitly separates the representation of the Q-value in two portions: one associated with the state alone (Q_i) that generalizes the Q-value across all actions within that state, and one associated with the actions (Q_{ij}) alone that specializes the Q-value within the state.

[4] Equation 37 corresponds to replacement of eligibilities.

- **Nearest neighbors selection**: A similarity function associated with the state space is used to compute the distance d_i^x from the query state x_q to the state x_i of each case C_i in memory. Then, the set of nearest neighbors to the query state, NN_q, is determined using the following rule:

$$NN_q = \{C_i \in \text{Memory} \mid d_i^x \leq \tau_k\} \tag{38}$$

  Thus, in the case-based version, only the similarity across the state space is considered in finding nearest neighbors. As a consequence, a case may be selected as a near neighbor even when the query action is very different than the one stored in the case. However, each case can contribute with many different values to determine the Q-value of the query action. This produces an increase in both the generalization and the resolution of the function approximator because the Q-value of an state-action pair may be estimated approximately even when the agent has tried only a few number of actions at similar states and, at the same time, there are several elements within each case to represent the Q-value with higher accuracy for different actions at the similar states.

- **Function evaluation**: The evaluation of the function approximator at query point (x_q, u_q) requires two kernel functions: $K^x(\cdot)$ and $K^u(\cdot)$. The former one is used to determine the relative contribution of each case C_i to the value of the query state x_q. The later one is used to determine the relative contribution of each action u_{ij} within each case to the value of the query action u_q.

  The evaluation is performed in two phases. During the first phase, a similarity function associated with the action space is used to compute the distance d_{ij}^u from the query action u_q to the action u_{ij} for each action in case C_i in the nearest-neighbor set. Then, the Q-value for the query action, $Q_i(u_q)$, is determined for each case using the following weighted average:

$$Q_i(u_q) = (1-\rho)Q_i + \rho \left(\sum_{\forall\, u_{ij} \in C_i} \frac{K^u(d_{ij}^u)}{\sum_j K^u(d_{ij}^u)} Q_{ij} \right) \quad \forall C_i \in NN_q \tag{39}$$

  where ρ is a parameter that controls the blending between the Q-value associated with the state, Q_i, and the Q-values associated with each action, Q_{ij}. When ρ is close to zero, the value of $Q_i(u_q)$ is close to the value associated with the state (Q_i) and less sensitive to the value of the actions (Q_{ij}) (i.e., the Q-value of the state-action pair depends more on the state rather than on the action). On the other hand, when ρ is close to one, the value of $Q_i(u_q)$ is close to the value associated with the actions and less sensitive to the value of the state (i.e., the Q-value of the state-action pair depends more on the action rather than on the state). Thus, for a fixed value of ρ, $Q_i(u_q)$ represents the Q-value of the query point (x_q, u_q) according to case C_i, which is a blend between the Q-value associated with the state and the weighted average of the Q-values associated with the actions.

During the second phase, the distances associated with state space d_i^x's and the kernel function $K^x(\cdot)$ are used to determined the relative contribution of each case C_i and merge the $Q_i(u_q)$'s into a weighted average. The following equation shows the computation.

$$\hat{Q}(x_q, u_q) = \sum_{\forall C_i \in NN_q} \frac{K^x(d_i^x)}{\sum_j K^x(d_j^x)} Q_i(u_q) \tag{40}$$

– **TD(λ) update**: The learning update is performed to each case in memory and each action within a case.

$$\begin{aligned} \Delta Q_i &= \alpha(r_{t+1} + \gamma \hat{Q}_{t+1} - \hat{Q}_t)\, e_i \ \ \forall C_i \in \text{Memory} && (41) \\ \Delta Q_{ij} &= \alpha(r_{t+1} + \gamma \hat{Q}_{t+1} - \hat{Q}_t)\, e_{ij} \ \ \forall u_{ij} \in C_i && (42) \end{aligned}$$

where $\hat{Q}_t$ and $\hat{Q}_{t+1}$ are the values of state-action pairs at (x_t, u_t) and (x_{t+1}, u_{t+1}) respectively.

– **Eligibility update**: The eligibilities of all the cases in memory and all the actions within a case are updated according to their current contributions to the value of the query point. In a query operation, every neighboring case contributes according to its distance to the query point modulated by the kernel function; therefore, the contribution of each case is its weight in the weighted average computation:[5]

$$e_i \leftarrow \begin{cases} (1-\rho)\frac{K^x(d_i^x)}{\sum_j K^x(d_j^x)} & \text{if } C_i \in NN(x,u) \\ \lambda\gamma\, e_i & \text{otherwise} \end{cases} \tag{43}$$

$$e_{ij} \leftarrow \begin{cases} \rho\frac{K^u(d_i^u)}{\sum_j K^u(d_j^u)} & \text{if } C_i \in NN(x,u) \\ \lambda\gamma\, e_{ij} & \text{otherwise} \end{cases} \tag{44}$$

– **Case addition criterion**: A new case is added to the memory when the distance to the closest neighbor, $d_{\min}^x = \min\{d_i^x\}$, is larger than the threshold parameter τ_d.

[5] Equations 43 and 44 corresponds to replacement of eligibilities.

Memory-based function approximators are more memory efficient than CMACs but require more computation to calculate the Q-value for a given state-action pair and to perform the TD(λ) updates. However, the main advantage of memory-based function approximators is their ability to dynamically allocate resources in regions of the state-action space that require them the most.

4.3 Non-uniform Preallocation of Resources

Function approximators may not have enough resolution to represent the Q-function in the entire state-action space. However, each class of function approximator provide different ways of distributing resources non-uniformly across the state-action space so that some regions will have more resolution than others. For example, CMACs may use tilings with non-uniform tile sizes, while memory-based function approximators may use different density thresholds in different regions of the state-action space. However, designing non-uniform function approximators is more complex than designing their uniform counterparts. Moreover, non-uniform designs often hinder the computational efficiency and memory storage of the function approximator. The idea of distributing the resources non-uniformly is inspired by quantization techniques widely used in digital communication theory. The objective is to approximate an analog signal with a quantized version so that the error between the two is minimized. For that purpose, more quantization levels are allocated to those regions where the amplitude of the signal is more frequently used, which reduces the average error performed in the quantization process (see, e.g., Shanmugam, 1979).

It is well-known in the field of machine learning that the representation of the input space has a large impact on learning speed and accuracy. A common technique consists of preprocessing the training examples to benefit learning and performance. We implement this technique to design function approximators that take advantage of the simplicity and efficiency of the uniform versions of function approximators and yet are equivalent to the more elaborate, but more accurate non-uniform counterparts. The implementation consists in distorting the state-action space by applying a *skewing function* and then use the *skewed state-action space* as the input to the uniform function approximator. In this way, the skewing function can be designed to expand or compress different regions of the state-action space. Expanded regions cover more area in the skewed state-action space; hence, they will use more resources of the function approximator and, consequently, will have better resolution to map the value function. Conversely, compressed regions cover less area, use fewer resources of the function approximator, and have less resolution to map the value function. There are no restrictions on the skewing function besides being able to map each point of the state-action space into some point on the skewed state-action

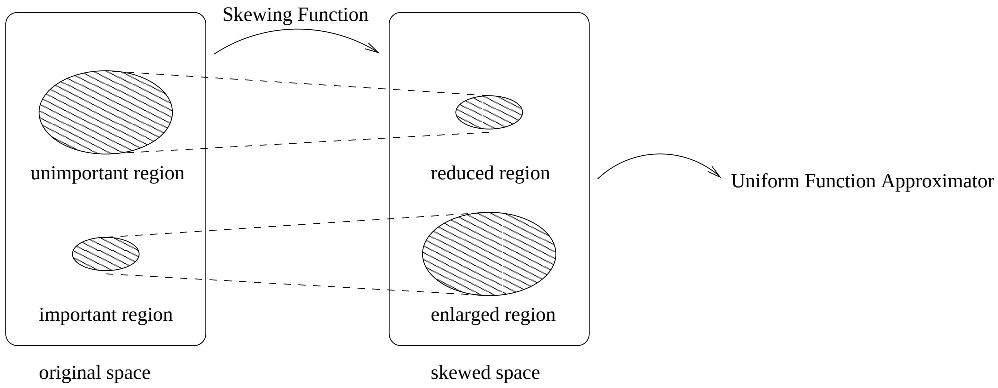

Figure 9: Non-uniform Resource Preallocation.
The skewing function distributes the resources non-uniformly. The function *skews* the space by expanding important regions and compressing unimportant ones. The skewed space is the input to the uniform function approximator.

space. Figure 9 demonstrates the graphically the process of preallocating resources non-uniformly.

The design principle followed in this research to specify the skewing function is inspired by optimization techniques used in digital communication theory. In that field, the skewing function is determined so that it optimizes some design criterion such as the mean square quantization error given the probability distribution of the input signal. The two main factors that influence the mean square quantization error are the difference between the quantized and real values at some input point and the number of times such an input point is evaluated. The mean quantization error is proportional to the product of both the actual error at some input point and the number of times that input point is used. Thus, the idea behind the optimization of the mean squared quantization error is to reduce the error more heavily on the input points that are more frequently used.

Following this principle, a skewing function can be empirically designed such that the function approximator has more resolution in the regions of the state-action space that are more frequently used. The increased resolution in those regions allows the function approximator to be closer to the real value (given enough experience) and incur to less error, which is desired. A priori knowledge of the system can guide the design of the skewing function, although the only knowledge that is required pertains to the expected frequency of use of regions in the state-action space. The skewing function should be designed in such a way that expands frequently used regions and compresses those that are used infrequently. Alternatively, the function approximator could be design to dynamically allocate more resources in those regions of the state-action space that are used more frequently. In this way, the resolution of the function approximator increases automatically without the need of prior knowledge. In Section 4.4, we describe the skewing function we used in the design of a non-uniform function approximator used by agents acting on the double integrator and the swing up pendulum problems (see Figure 16). In these two problems, it is known that the agents will perform near the origin of the state-action space most of the time. This requires a function approximator having more resolution near the origin of the state-action space; therefore, the skewing function is empirically designed to expand that region in particular.

4.3.1 Implementation of Non-Uniform Functions Approximators

The one-step search and the SARSA algorithms require little modification to accommodate the use of non-uniform function approximators. In both cases, the standard implementation of the uniform function approximator is used over the skewed state-action pair to accomplish the effect of non-uniform resource preallocation. Figures 10 and 11 show the new versions of the one-step search and the SARSA algorithms (additions shown in bold).

Input: state x_t, weights w_t, and skewing function $s(\cdot,\cdot)$
Output: best action $\hat{u}^*$
Algorithm:

1. For every action u_i from $u_{\min}$ to $u_{\max}$ at step Δ_u:
   (a) **Compute skewed state-action pair**: $(x^s, u^s) \leftarrow s(x_t, u_i)$.
   (b) Evaluate $\hat{Q}_{w_t}(x^s, u^s)$.
   (c) Keep the action $\hat{u}^* = u_i$ that produces the maximum value.
2. Return $\hat{u}^*$.

Figure 10: One-Step Search Algorithm with Skewing Function.

4.4 Results

This section describes the results of using uniform and non-uniform versions of CMAC, instance-based, and case-based function approximators in problems with continuous state and actions spaces. The problems studied are the double integrator

Input: initial set of weights w_0, and skewing function $s(\cdot, \cdot)$
Algorithm:

1. Initialization:
   (a) perceive current state: x_{curr}
   (b) select action: $u_{\text{curr}} \leftarrow \epsilon\text{-}\textit{greedy-policy}(x_{\text{curr}})$
2. **Compute skewed state-action pair**:
   $(x^s_{\text{curr}}, u^s_{\text{curr}}) \leftarrow s(x_{\text{curr}}, u_{\text{curr}})$.
3. Accumulate gradient:
   $\mathbf{W}_t \leftarrow \nabla_w \hat{Q}(x^s_{\text{curr}}, u^s_{\text{curr}}) + \lambda\gamma\mathbf{W}_{t-1}$
4. Perform action:
   Execute u_{curr}, observe resultant reward, r, and next state, x_{next}.
5. Select next action:
   $u_{\text{next}} \leftarrow \epsilon\text{-}\textit{greedy-policy}(x_{\text{next}})$
6. **Compute skewed state-action pair**:
   $(x^s_{\text{next}}, u^s_{\text{next}}) \leftarrow s(x_{\text{next}}, u_{\text{next}})$.
7. Adapt:
   $w_{t+1} \leftarrow w_t + \alpha(r + \gamma\hat{Q}(x^s_{\text{next}}, u^s_{\text{next}}) - \hat{Q}(x^s_{\text{curr}}, u^s_{\text{curr}}))\mathbf{W}_t$
8. Loop:
   $x_{\text{curr}} \leftarrow x_{\text{next}}$
   $u_{\text{curr}} \leftarrow u_{\text{next}}$
   If x_{curr} is a goal state then terminate else goto 2.

Figure 11: Gradient Descent Version of SARSA Algorithm with Skewing Function.

and the pendulum swing up. The first one is a linear dynamics system and the second one is a non-linear one, both with quadratic costs (i.e., negative quadratic rewards) that depend on the state and action values. The function approximators used in both problems are the CMAC, instance-based, and case-based. Satisfactory convergence results was obtained in all experiments except in the agent using the instance-based function approximator on the pendulum swing up. This is mainly because of the lack of resolution this type of function approximator and the complexity of the problem.

4.4.1 Double Integrator

The double integrator is a system with linear dynamics and bidimensional state. It represents a car moving in a flat terrain and subject to the application of a single force (see Figure 12). The state of the system consists of the current position, p, and velocity v of the car, or $\mathbf{x} = [v\ p]^T$ in vectorial representation. The action is the acceleration, a, applied to the system, or $\mathbf{u} = [a]$ in vectorial representation. The objective is to move the car from a given starting state to the origin (i.e., $\mathbf{x_d} = [0\ 0]^T$) such that the sum of the rewards is maximized. The reward function is a negative quadratic function of the difference between the current and desired position and the acceleration applied, $r_{t+1} = -(p_t^2 + a_t^2)$ or in vectorial representation, $r_{t+1} = -((\mathbf{x}_t - \mathbf{x_d})^T Q(\mathbf{x}_t - \mathbf{x_d}) + \mathbf{u}_t^T R \mathbf{u}_t)$, where Q and R are the positive definite 2×2 and 1×1 matrices respectively defined as $Q = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix}$ and $R = [1]$. The reward function penalizes the agent more heavily when the distance between the current and desired states is large and also when the action applied is large. This type of reward function is widely used in robotic applications because it specifies policies that drive the system to the desired state quickly while keeping the size of the driving control small. The tradeoffs between the two can be specified by appropriate selection of the Q and R matrices. This formulation is standard in optimal control theory (Stengel, 1994; Narendra and Annaswamy, 1989), in which the objective is to minimize costs instead of maximize rewards. However, both formulations are mathematically equivalent.

The dynamics of the double integrator is very simple and is described by the following equations,

$$\frac{dp}{dt} = v \tag{45}$$

$$M\frac{dv}{dt} = a \tag{46}$$

where M is the mass of the car that is unknown to the agent ($M = 1$). The acceleration is bounded to be in the range between the minimum and maximum

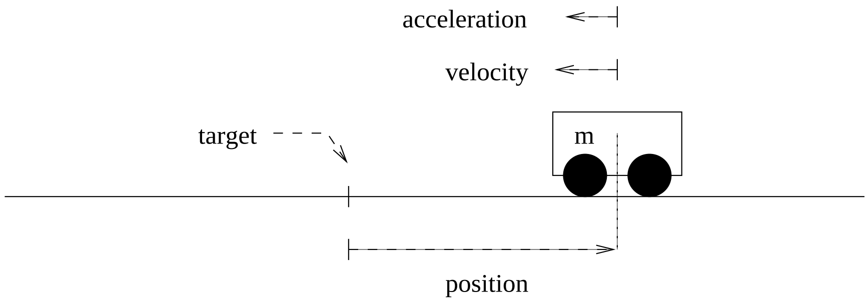

Figure 12: The Double Integrator Problem.
A car moving in a flat terrain subject to the application of a single force.

acceleration values (i.e., $a \in [a_{\min}\ a_{\max}]$, where $a_{\min} = -1$ and $a_{\max} = 1$). The double integrator is an instance of a more general class of linear dynamic systems and can be expressed using the more convenient linear matrix equation,

$$\begin{aligned} \mathbf{x}_{t+1} = \begin{bmatrix} v_{t+1} \\ p_{t+1} \end{bmatrix} &= \begin{bmatrix} v_t \\ p_t \end{bmatrix} + \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} v_t \\ p_t \end{bmatrix} \Delta t + \begin{bmatrix} 1/M \\ 0 \end{bmatrix} [a]\ \Delta t \\ &= \mathbf{x}_t + A\mathbf{x}_t\ \Delta t + B\mathbf{u}_t\ \Delta t \end{aligned} \tag{47}$$

The simulation used a time step of $\Delta t = 0.05$ seconds and new control actions were selected every four time steps. A trial consisted of starting the system at position $p = 1$ with velocity $v = 0$ and running the system until either 200 decision stages had elapsed (i.e., 40 simulated seconds) or the state got out of bounds (i.e., when $|p| > 1$ or $|v| > 1$), whichever came first. In the latter case, the agent received a negative reward (i.e., a punishment) of 50 units to discourage it from going out of bounds. An experiment consisted of 36 replications of 50 trials each, measuring the number of time steps and cumulative cost (i.e., negative sum of rewards) for each replication after each trial. The average number of time steps and cumulative cost across replications were used as measures of performance.

4.4.1.1 Optimal Solution

Systems with linear dynamics and quadratic cost functions have a simple closed-form solution for the optimal policy known as *Linear-Quadratic Regulator (LQR)*. The derivation follows from the solution to the Hamilton-Bellman-Jacobi partial differential equation (see, e.g., Stengel, 1994, or Bertsekas, 1995a, for details). Given a dynamical system with an n-dimensional state vector, $\mathbf{x}$, a m-dimensional action vector, $\mathbf{u}$, a linear dynamic function $\dot{\mathbf{x}} = A(\mathbf{x} - \mathbf{x_d}) + B(\mathbf{u} - \mathbf{u_d})$, where A and B are $n \times n$ and $n \times m$ matrices, respectively, and a one-step quadratic cost (i.e., negative reward) function of the form $r = -((\mathbf{x} - \mathbf{x_d})^T Q(\mathbf{x} - \mathbf{x_d}) + (\mathbf{u} - \mathbf{u_d})^T R(\mathbf{u} - \mathbf{u_d}))$, where Q and R are positive definite $n \times n$ and $m \times m$ matrices respectively.[6] Then, the optimal policy is given by $\mathbf{u}^* = -K(\mathbf{x} - \mathbf{x_d}) + \mathbf{u_d}$ with an associated optimal value function given by $V(\mathbf{x}) = (\mathbf{x} - \mathbf{x_d})^T P(\mathbf{x} - \mathbf{x_d})$, where K and P are $m \times n$ and $n \times n$ matrices given by Equations 48 and 49, respectively. Equation 49 is known as the *continuous-time Ricatti equation* and it is used to find the unknown matrix P, which is then used in Equation 48 to find K, also called the *gain* matrix (Bertsekas, 1995a; Richards, 1979).[7]

$$K = R^{-1}B^T P \tag{48}$$

$$\dot{P} = -Q - A^T P - PA + PBR^{-1}B^T P \tag{49}$$

In the double integrator, the values for A, B, Q, and R were described previously.[8] The solution of P in the time-invariant case is $P = \begin{bmatrix} \sqrt{2} & 1 \\ 1 & \sqrt{2} \end{bmatrix}$, which produces an equation for the optimal actions as $u^* = -Kx = -(\sqrt{2}\,v + p)$, where v and p are the instantaneous velocity and position respectively.

Figures 13 and 14 show the optimal value function and the optimal trajectory generated using the optimal policy. In the simulator the optimal controller achieves a performance of $-1.4293 \approx V(\mathbf{x_0}) = \sqrt{2}$ when the initial state is $\mathbf{x}_0 = [0\ 1]^T$. The discrepancy between the actual sum of rewards and the one predicted by the value function is due to the discretization across time and it is not considered significant. Figure 15 shows the optimal trajectory and the cumulative reward of the controller at different points of the trajectory.

4.4.1.2 Uniform and Non-uniform CMAC Agent Configurations

In these configurations the agent used a CMAC to approximate the Q-function. There are three dimensions of interest: position, p, velocity, v, and acceleration, a. The CMAC used 36 tilings. Each of the three dimensions were divided into 12 intervals. 12 tilings depended on all three variables. 12 other tilings depended on position and velocity pairs only, and the remaining 12 tilings depended on one dimension only (6 for position and 6 for velocity). The tilings sharing the same dimensions were uniformly offseted across each dimension. Other CMAC configurations were tried and compared empirically; the configuration described here is the one that produced best

[6] The constant $\mathbf{x}_\mathrm{d}$ corresponds to the desired goal state and the constant $\mathbf{u}_\mathrm{d}$ corresponds to the required force to maintain the system at the desired state (i.e., once the system is in $\mathbf{x}_\mathrm{d}$, applying $\mathbf{u}_\mathrm{d}$ results in no change).

[7] The continuous-time Ricatti equation can be simplified further for time-invariant policies. This is the case when $t \to \infty$ and $\dot{P} \to \mathbf{0}$. Then the equation simplifies to, $Q + A^T P + AP = PBR^{-1}B^T P$. The resulting optimal controller is referred to as the Linear-Quadratic Regulator of the system.

[8] In the double integrator $\mathbf{x}_\mathrm{d} = [0\ 0]^T$ and $\mathbf{u}_\mathrm{d} = [0]$.

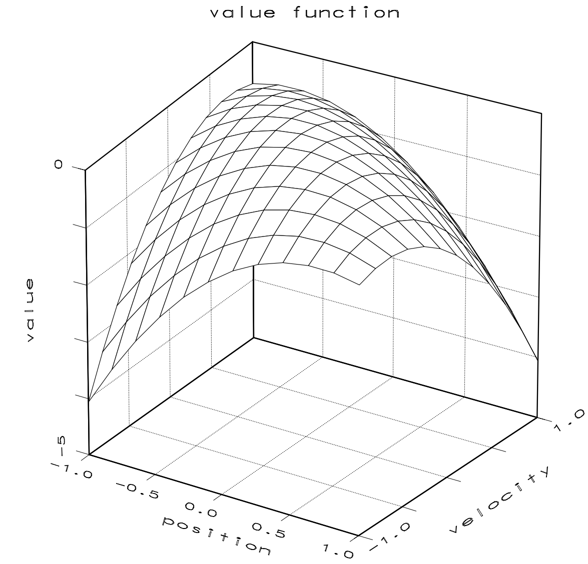

Figure 13: Optimal Value Function for the Double Integrator.
The value function (vertical axis) is plotted against position and velocity (horizontal plane). States near the origin have values closer to zero than states far from the origin because it takes more time and cost to drive the system towards the origin.

results. The SARSA algorithm in Figure 6 was used for updating the weights of the CMAC using the traces indicated by Equation 33. The values for the free constants were $\gamma = 0.99$, $\lambda = 0.7$, $\alpha = 0.1$, and $\epsilon = 0$. The one-step search was performed using 25 equally spaced values for the acceleration between the minimum value, $a_{\min} = -1$, and the maximum value, $a_{\max} = +1$ (i.e., $25\Delta_u = a_{\max} - a_{\min}$). Table 5 summarizes the agent's design for the uniform and non-uniform configurations.

The top rows of Figures 17 and 18 show the average number of time steps and accumulated cost per trial for the agent using the uniform CMAC function approximator, respectively. The error bars represent the ± 1 standard deviations of the mean across 36 replications.

For the non-uniform configuration, the agent used the same CMAC design as in the previous experiment. However, the CMAC used skewed versions of the three variables of interest: position, velocity, and acceleration. The skewing functions were as follows: $p^s = \sqrt[3]{p}$, $v^s = \sqrt{v}$, and $a^s = \sqrt{a}$. Figure 16 shows the effect of the skewing functions of the class $s(x) = \sqrt[k]{x}$ for $k = 2, 3$. The function expands the region near the origin so that the CMAC can use more tiles to represent the value for the Q-function more accurately in that region. The larger the value of k the larger

Table 5: Design of the Uniform and Non-uniform CMAC Agents for the Double Integrator.

Factor	Description	
	Uniform	Non-uniform
Variables	3: position: $p \in [-1\ 1]$; 12 intervals velocity: $v \in [-1\ 1]$; 12 intervals acceleration: $a \in [-1\ 1]$; 12 intervals	
Tilings	36: 12 based on p, v, and a; uniformly offseted. 12 based on p and v; uniformly offseted. 6 based on p; uniformly offseted. 6 based on v; uniformly offseted.	
Skewing functions	N/A	$p^s = \sqrt[3]{p}$ $v^s = \sqrt{v}$ $a^s = \sqrt{a}$
One-step search	ϵ-greedy policy (Figure 5): $\epsilon = 0$ $\Delta_u = \frac{a_{\max} - a_{\min}}{25} = 0.08$	ϵ-greedy policy (Figure 10): $\epsilon = 0$ $\Delta_u = \frac{a_{\max} - a_{\min}}{25} = 0.08$
Learning	SARSA algorithm (Figure 6): $\gamma = 0.99$ $\alpha = 0.1$ $\lambda = 0.7$	SARSA algorithm (Figure 11): $\gamma = 0.99$ $\alpha = 0.1$ $\lambda = 0.7$

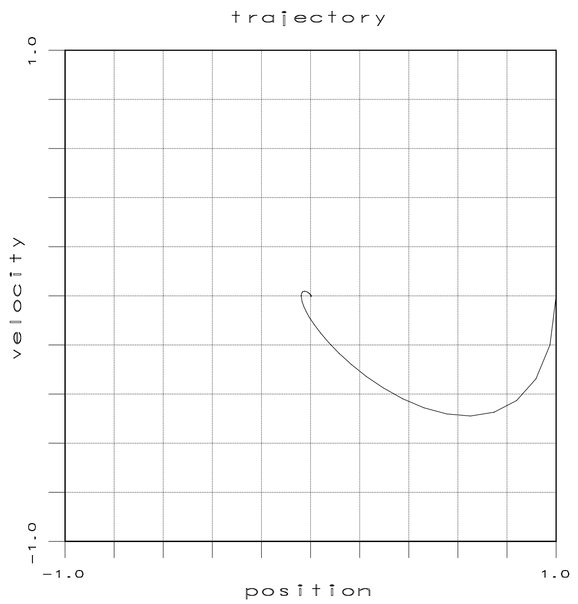

Figure 14: Optimal Trajectory for the Double Integrator.
The starting point is $\mathbf{x_0} = [0\ 1]^T$ (center right) and goal point is $\mathbf{x_d} = [0\ 0]^T$ (center).

the expansion. Thus, the dimension corresponding to the position is skewed more heavily than the velocity and acceleration. We chose this skewing function design because we knew the system will spend more time near the origin orbiting around the goal state in order to reduce punishments. Thus, we inferred that the region near the origin of the state-action space would require more resolution of the function approximator. Additionally, the one-step reward function depends only with the position, which results in a Q-function that is more sensitive to the position than to the velocity (refer to the Q matrix). The learning algorithm and the free constants were the same as with uniform CMAC.

The top rows of Figures 17 and 18 show the average number of time steps and accumulated cost per trial, for the agent using the non-uniform CMAC function approximator, respectively. The error bars represent the ± 1 standard deviations of the mean across 36 replications.

4.4.1.3 Uniform and Non-uniform Instance-Based Agent Configurations

In these configurations the agent used an instance-based representation of the Q-function. Each case represents a point in the state-action space and holds its associated value. The density threshold and the smoothing parameters were set to

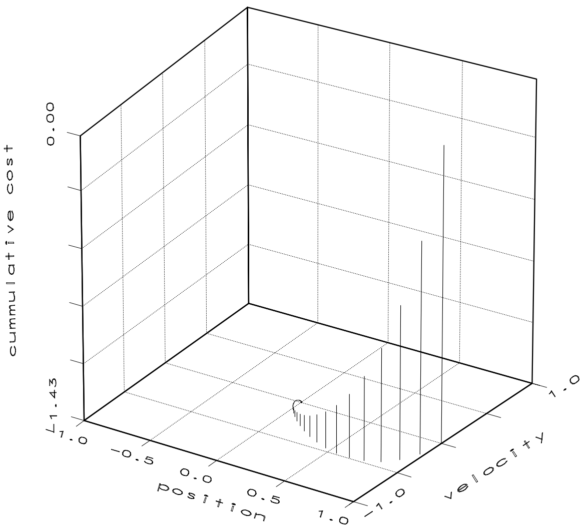

Figure 15: Optimal Trajectory and Sum of Rewards for the Double Integrator. The sum of rewards (vertical axis) decreases faster when the system is far from the origin than when it is closer to the origin.

$\tau_d = 0.06$ and $\tau_k = 0.06$, respectively. The similarity metric was the Euclidean distance and the kernel function was the Gaussian kernel. These settings produce cases with spherical receptive fields and blending of the Q-function using a small number of cases. The SARSA algorithm in Figure 6 was used to update the values of each case using the eligibility traces indicated by Equation 37. The values for the free constants were $\gamma = 0.99$, $\lambda = 0.8$, $\alpha = 0.5$, and $\epsilon = 0$. The one-step search was performed using 25 equally spaced values for the acceleration between the minimum value, $a_{\min} = -1$, and the maximum value, $a_{\max} = +1$ (i.e., $25\Delta_u = a_{\max} - a_{\min}$). Table 6 summarizes the agent's design for the uniform and non-uniform configurations.

The middle rows of Figures 17 and 18 show the average number of time steps and accumulated cost per trial for the agent using the uniform instance-based function approximator, respectively. The error bars represent the ± 1 the standard deviations of the mean across 36 replications.

For the non-uniform configuration, the agent used the same instance-based design as in the previous experiment but using the skewed version of the three variables of interest. The skewing functions were the same as the one used for the non-uniform CMAC (i.e., $p^s = \sqrt[3]{p}$, $v^s = \sqrt{v}$, and $a^s = \sqrt{a}$). The effect of using the instance-based function approximator on the skewed state-action space is similar to having variable

Table 6: Design of the Uniform and Non-uniform Instance-Based Agents for the Double Integrator.

Factor	Description	
	Uniform	Non-uniform
Variables	3: position: $p \in [-1\ 1]$ velocity: $v \in [-1\ 1]$ acceleration: $a \in [-1\ 1]$	
Distance function	Euclidean: $d(x_i, x_q) = \sqrt{(p_i - p_q)^2 + (v_i - v_q)^2 + (a_i - q_q)^2}$ density threshold: $\tau_d = 0.06$	
Kernel function	Gaussian: $K(d_i) = \exp\left(\frac{d_i}{\tau_k}\right)^2$ smoothing parameter: $\tau_k = 0.06$	
Skewing functions	N/A	$p^s = \sqrt[3]{p}$ $v^s = \sqrt{v}$ $a^s = \sqrt{a}$
One-step search	ϵ-greedy policy (Figure 5): $\epsilon = 0$ $\Delta_u = \frac{a_{\max} - a_{\min}}{25} = 0.08$	ϵ-greedy policy (Figure 10): $\epsilon = 0$ $\Delta_u = \frac{a_{\max} - a_{\min}}{25} = 0.08$
Learning	SARSA algorithm (Figure 6): $\gamma = 0.99$ $\alpha = 0.5$ $\lambda = 0.8$	SARSA algorithm (Figure 11): $\gamma = 0.99$ $\alpha = 0.5$ $\lambda = 0.8$

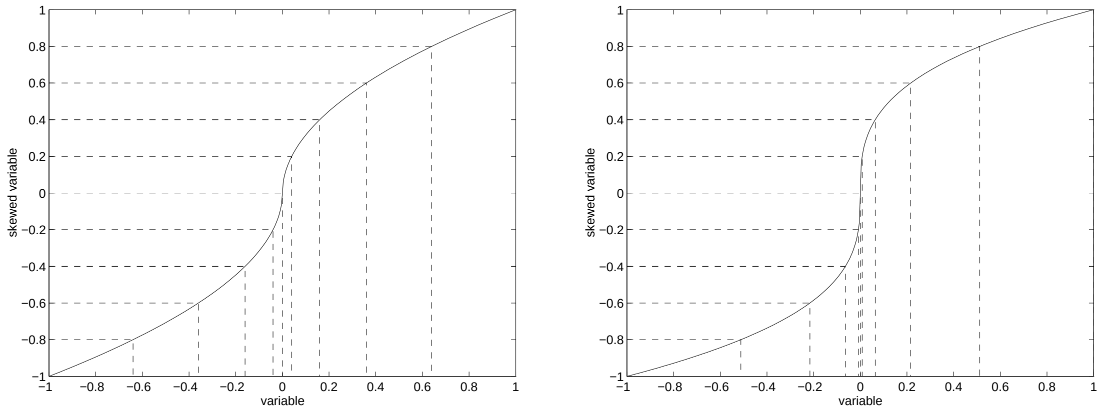

Figure 16: Skewing Functions.
Left: skewing function $s(x) = \sqrt{x}$. Right: skewing function $s(x) = \sqrt[3]{x}$. Uniform intervals of the skewed variable (vertical axis) correspond to non-uniform intervals for the variable (horizontal axis). The expansion near the origin is stronger as the value of k increases.

density and smoothing parameters. Thus, the density of cases increases near the origin because that region gets expanded by the skewing function. Similarly, the smoothing parameter adjusts accordingly so that the increased density of cases near the origin does not produce too much blending of their values.

The middle rows of Figures 17 and 18 show the average number of time steps and accumulated cost per trial for the agent using the non-uniform instance-based function approximator, respectively. The error bars represent the ± 1 the standard deviations of the mean across 36 replications.

4.4.1.4 Uniform and Non-uniform Case-Based Agent Configurations

In these configurations the agent used a case-based representation of the Q-function. The case-based function approximator uses more resources and has better resolution than the instance-based. Each case in the case-based function approximator represents a point in the state space and the value of 6 equally spaced actions in the interval $[\alpha_{\min}\ \alpha_{\max}]$. The blending factor was set to $\rho = 0.6$, which means that during the Q-value computations each case contributed 40% according to the Q-value associated to the state and 60% according to the Q-values associated with the actions. The density threshold was set to $\tau_d = 0.075$. The smoothing parameters for the state space and the action space were set to $\tau_k^x = 0.085$ and $\tau_k^u = 0.2$, respectively. The similarity metric for the input space was a weighted Euclidean distance and the kernel function was the Gaussian kernel. These settings produce cases with elliptical receptive fields with the major axis oriented diagonally in the state space, which bias the function approximator to generalize more in regions in which the position and the velocity of the car have the same sign. This kind of bias proved to be very useful once the car reached the goal position. The similarity metric for the output space was the Euclidean distance. The SARSA algorithm in Figure 6 was used to update the values of each case using the eligibility traces indicated by Equations 43 and 44. The values for the free constants were $\gamma = 0.99$, $\lambda = 0.7$, $\alpha = 0.4$, and $\epsilon = 0$. The one-step search was performed using 25 equally spaced values for the acceleration between the minimum value, $a_{\min} = -1$, and the maximum value, $a_{\max} = +1$ (i.e., $25\Delta_u = a_{\max} - a_{\min}$). Table 7 summarizes the agent's design for the uniform and non-uniform configurations.

The bottom rows of Figures 17 and 18 show the average number of time steps and accumulated cost per trial for the agent using the uniform case-based function approximator, respectively. The error bars represent the ± 1 standard deviations of the mean across 36 replications.

For the non-uniform configuration, the agent used the same case-based design as in the previous experiment but using the skewed version of the three variables of interest. The skewing functions were the same as the one used for the non-uniform CMAC (i.e., $p^s = \sqrt[3]{p}$, $v^s = \sqrt{v}$, and $a^s = \sqrt{a}$). The effect of using the case-based function approximator on the skewed state-action space is similar to having non-constant density and smoothing parameters. Thus, the density of cases increases near the origin because that region gets expanded by the skewing function. Similarly, the smoothing parameter adjusts accordingly so that the increased density of cases near the origin does not produce too much blending of their values.

The bottom rows of Figures 17 and 18 show the average number of time steps and accumulated cost per trial for the agent using the non-uniform case-based function approximator, respectively. The error bars represent the ± 1 standard deviations of the mean across 36 replications.

4.4.2 Pendulum Swing Up

The pendulum is a non-linear dynamics system with a bidimensional state. It represents a single bar held by one extremum and that can swing in a vertical plane. The bar is actuated by a motor that applies a torque at the hanging point (see Figure 19). The state of the system consists of the current angle, θ, and angular velocity ω of the pendulum, or $\mathbf{x} = [\theta\ \omega]^T$ in vectorial representation. The action is the angular acceleration, α, applied to the system, or $\mathbf{u} = [\alpha]$ in vectorial representation. The objective is to move the pendulum from its rest position (i.e., hanging down with no velocity) to the origin of the state space (i.e., bar facing up with no velocity, $\mathbf{x_d} = [0\ 0]^T$) such that the sum of the rewards is maximized. The one-step reward function is of

Table 7: Design of the Uniform and Non-uniform Case-based Agent for the Double Integrator.

<table>
<tr><td rowspan="2">Factor</td><td colspan="2">Description</td></tr>
<tr><td>Uniform</td><td>Non-uniform</td></tr>
<tr><td>Variables</td><td colspan="2">3:
position: $p \in [-1\ 1]$
velocity: $v \in [-1\ 1]$
acceleration: $a \in [-1\ 1]$</td></tr>
<tr><td>Case structure</td><td colspan="2">Number of actions: 6, equally spaced in $[-1\ 1]$
Blending factor: $\rho = 60\%$</td></tr>
<tr><td rowspan="2">Distance functions</td><td colspan="2">Input space: Weighted Euclidean.
$d_M(x_i, x_q) = \sqrt{(x_i - x_q)^T M^T M (x_i - x_q)}$
$M = \begin{bmatrix} 1 & 1 \\ -1.75 & 1.75 \end{bmatrix}$
density threshold: $\tau_d = 0.075$</td></tr>
<tr><td colspan="2">Output space: Euclidean.
$d_M(u_i, u_q) = |\alpha_i - \alpha_q|$</td></tr>
<tr><td rowspan="2">Kernel functions</td><td colspan="2">Input space: Gaussian.
$K^x(d_i^x) = \exp\left(\frac{d_i^x}{\tau_k^x}\right)^2$
smoothing parameter: $\tau_k^x = 0.085$</td></tr>
<tr><td colspan="2">Output space:
$K^u(d_i^u) = \exp\left(\frac{d_i^u}{\tau_k^u}\right)^2$
smoothing parameter: $\tau_k^u = 0.2$</td></tr>
<tr><td>Skewing functions</td><td>N/A</td><td>$p^s = \sqrt[3]{p}$
$v^s = \sqrt{v}$
$a^s = \sqrt{a}$</td></tr>
<tr><td>One-step search</td><td>ϵ-greedy policy (Figure 5):
$\epsilon = 0$
$\Delta_u = \frac{a_{\max} - a_{\min}}{25} = 0.08$</td><td>$\epsilon$-greedy policy (Figure 10):
$\epsilon = 0$
$\Delta_u = \frac{a_{\max} - a_{\min}}{25} = 0.08$</td></tr>
<tr><td>Learning</td><td>SARSA algorithm (Figure 6):
$\gamma = 0.99$
$\alpha = 0.4$
$\lambda = 0.7$</td><td>SARSA algorithm (Figure 11):
$\gamma = 0.99$
$\alpha = 0.4$
$\lambda = 0.7$</td></tr>
</table>

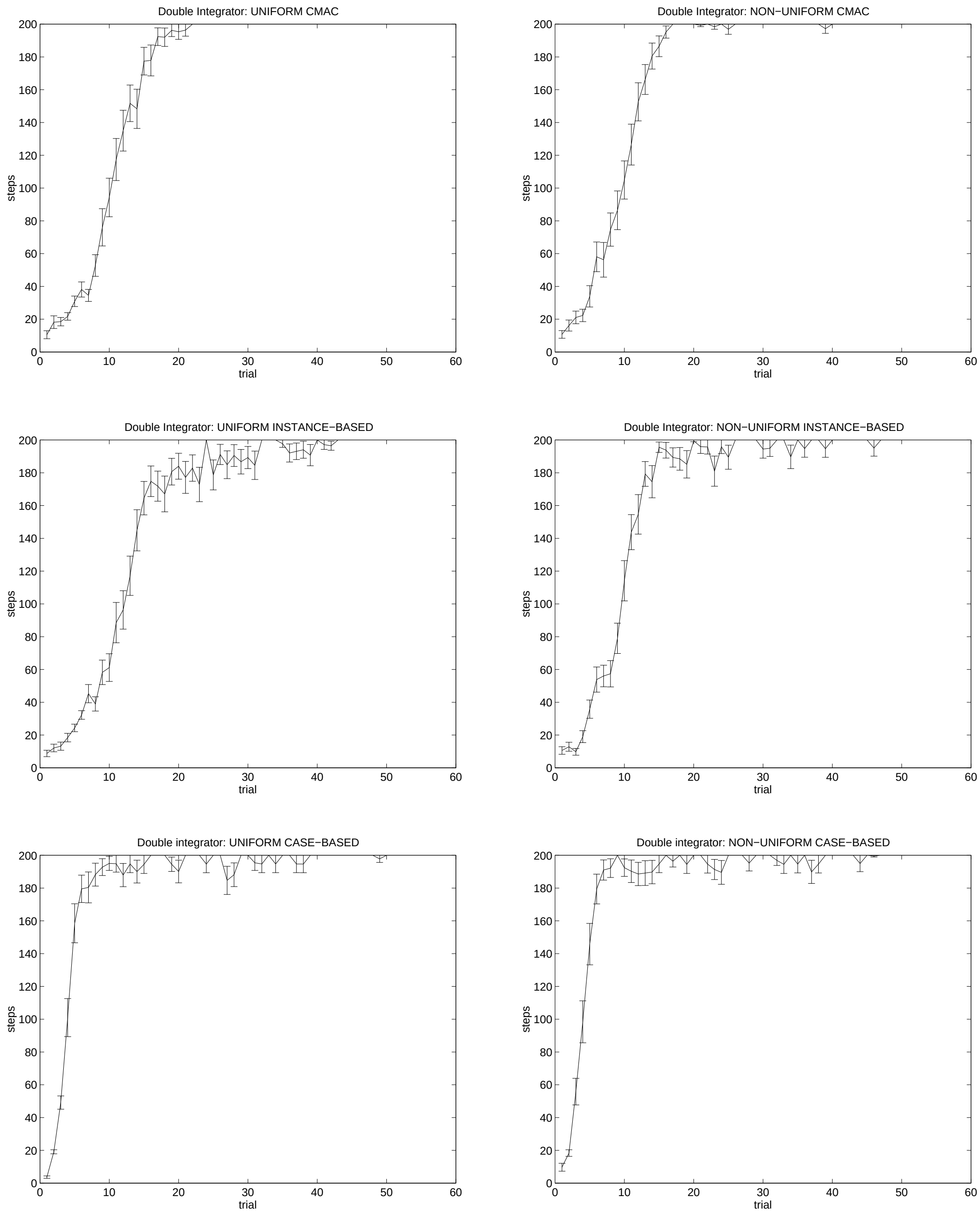

Figure 17: Average Steps per Trial in the Double Integrator Problem. The left and right columns show uniform and non-uniform configuration, respectively. Top row: CMAC. Middle row: instance-based. Bottom row: case-based.

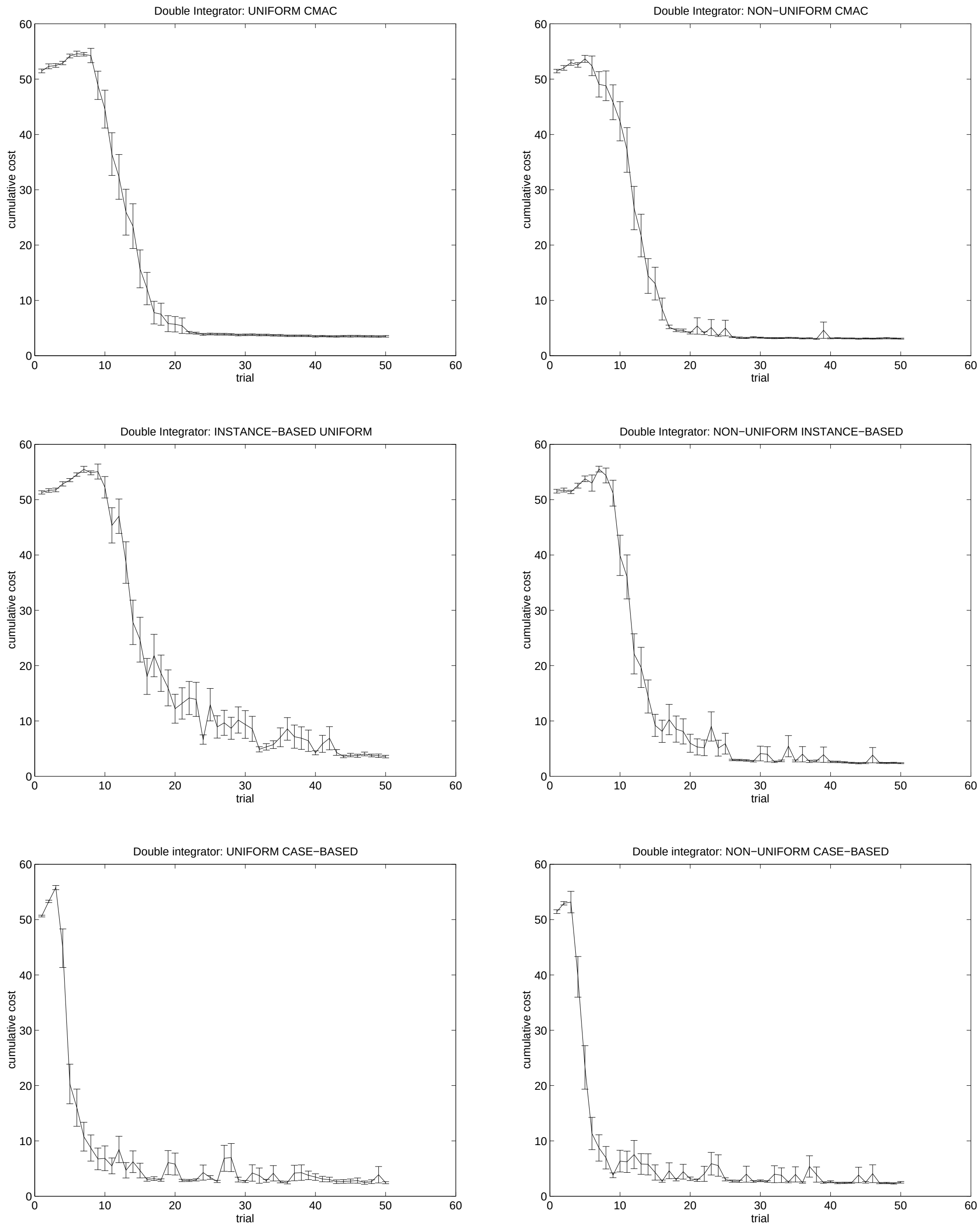

Figure 18: Average Cumulative Cost in the Double Integrator Problem. The left and right columns show uniform and non-uniform configuration, respectively. Top row: CMAC. Middle row: instance-based. Bottom row: case-based.

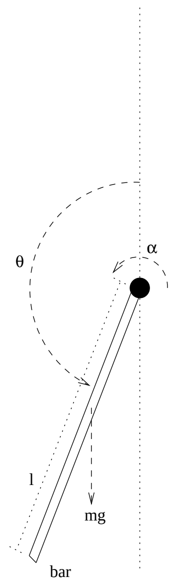

Figure 19: The Pendulum Swing Up Problem.
A bar hanging from one extremum and subject to gravity and the torque applied by a motor.

the same class used for the double integrator, that is, a negative quadratic function of the difference between the current and desired angular position and angular velocity,[9] and the angular acceleration applied, $r_{t+1} = -((\Delta_{2\pi}\theta_t)^2 + \omega^2 + \alpha_t^2)$ or in vectorial representation, $r_{t+1} = -(\Delta \mathbf{x}_t^T Q \Delta \mathbf{x}_t + \mathbf{u}_t^T R \mathbf{u}_t)$, where Q and R are the positive definite 2×2 and 1×1 matrices respectively defined as $Q = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$ and $R = [1]$.

The dynamics of the pendulum is given by the following equations,

$$\begin{aligned} \frac{d\omega}{dt} &= \frac{3}{4}\frac{1}{ml^2}(\alpha + mlg\sin(\theta)) \\ \frac{d\theta}{dt} &= \omega \end{aligned} \tag{50}$$

where $m = 1/3$ and $l = 3/2$ are the mass and length of the bar respectively, and $g = 9.8$ is the gravity. The angular acceleration is bounded to be in the range between the minimum and maximum angular acceleration values (i.e., $\alpha \in [\alpha_{\min}\ \alpha_{\max}]$, where $\alpha_{\min} = -3$ and $\alpha_{\max} = 3$). This system is more difficult to control than the

[9]The difference between actual and desired angle positions must be performed using modulo 2π and is denoted by $\Delta_{2\pi}\theta$.

ones with linear dynamics. Unlike the double integrator, no closed-form analytical solution exists for the optimal solution and complex numerical methods are required to compute it. Moreover, the maximum and minimum angular acceleration values are not strong enough to move the pendulum straight up from the starting state without first creating angular momentum. Thus, the optimal solution sometimes requires applying an action that moves the system in a direction opposite to the goal state first in order to build up enough momentum to be able to swing the bar up to the top. Additionally, the equilibrium point at the goal state is unstable, which means that the agent not only needs to manage to swing up the bar but actively balance it at the goal position as well.

As in the double integrator, the simulation used a time step of $\Delta_t = 0.05$ seconds and new control actions were selected every four time steps. A trial consisted of starting the system at position $\theta = -\pi$ with angular velocity $\omega = 0$ and running the system until 200 decision steps had elapsed (i.e. 40 simulated seconds) or the angular velocity of the pendulum got out of bounds (i.e., $\omega > 2\pi$). Thus, unlike the double integrator, the angle of the link can wrap around without causing the trial to terminate. An experiment consisted of 36 replications of 100 trials each and measuring the number of time steps and cumulative cost (i.e., negative sum of rewards) for each replication after each trial. The average number of time steps and cumulative cost across replications was used as a measure of performance.

4.4.2.1 Uniform and Non-uniform CMAC

As in the double integrator problem, the agent used a uniform and non-uniform CMACs to approximate the Q-function. The CMAC consisted of 36 tilings using the three state-action variables arranged in the same configuration as with the double integrator: 12 intervals along each dimension, 12 tiling depending on all three variables, 12 other tilings depended on position and velocity pairs only, and the remaining 12 tilings depended on one dimension only (6 for position and 6 for velocity). The tilings sharing the same dimensions were uniformly offseted across each dimension. The SARSA algorithm was used for updating the weights. The values for the free constants were $\gamma = 0.99$, $\lambda = 0.5$, $\alpha = 0.1$, and $\epsilon = 0$. The one-step search was performed using 24 equally spaced values for the angular acceleration between minimum value, $\alpha_{\min} = -3$, and the maximum value, $\alpha_{\max} = +3$ (i.e., $24\Delta_u = \alpha_{\max} - \alpha_{\min}$). Table 8 summarizes the agent's design for the uniform and non-uniform configurations.

The top rows of Figures 20 and 21 show the average number of time steps and accumulated cost per trial for the agent using the uniform CMAC function approximator, respectively. The error bars represent the ± 1 standard deviations of the mean across 36 replications.

Table 8: Design of the Uniform and Non-uniform CMAC Agent for the Pendulum Swing Up.

<table>
<tr><td rowspan="2">Factor</td><td colspan="2">Description</td></tr>
<tr><td>Uniform</td><td>Non-uniform</td></tr>
<tr><td>Variables</td><td colspan="2">3:
position: $\theta \in [-\pi\ \pi]$; 12 intervals
velocity: $\omega \in [-2\pi\ 2\pi]$; 12 intervals
acceleration: $\alpha \in [-3\ 3]$; 12 intervals</td></tr>
<tr><td>Tilings</td><td colspan="2">36:
12 based on θ, ω, and α; uniformly offseted.
12 based on θ and ω; uniformly offseted.
6 based on θ; uniformly offseted.
6 based on ω; uniformly offseted.</td></tr>
<tr><td>Skewing functions</td><td>N/A</td><td>$\theta^s = \sqrt{\theta}$
$\omega^s = \sqrt{\omega}$
$\alpha^s = \sqrt{\alpha}$</td></tr>
<tr><td>One-step search</td><td>ϵ-greedy policy (Figure 5):
$\epsilon = 0$
$\Delta_u = \frac{\alpha_{\max} - \alpha_{\min}}{24} = 0.25$</td><td>$\epsilon$-greedy policy (Figure 10):
$\epsilon = 0$
$\Delta_u = \frac{\alpha_{\max} - \alpha_{\min}}{24} = 0.25$</td></tr>
<tr><td>Learning</td><td>SARSA algorithm (Figure 6):
$\gamma = 0.99$
$\alpha = 0.1$
$\lambda = 0.5$</td><td>SARSA algorithm (Figure 11):
$\gamma = 0.99$
$\alpha = 0.1$
$\lambda = 0.5$</td></tr>
</table>

For the non-uniform configuration, the agent used the same CMAC design as in the previous experiment but using the skewed version of the three variables of interest. The skewing functions for each of the three variables were as follows: $\theta^s = \sqrt{\theta}$, $\omega^s = \sqrt{\omega}$, and $\alpha^s = \sqrt{\alpha}$. As in the double integrator, the skewing functions expands the regions near the origin so that the CMAC can use more tiles to represent the value for the Q-function with more resolution. We chose this design because we knew the system will spend more time balancing the bar once it is up and this corresponds to the region near the origin. Unlike in the double integrator, the same skewing function was used for the position and velocity because the one-step reward function depends equally on both variables (refer to the Q matrix). The learning algorithm and the free constants were the same as with the uniform CMAC.

The top rows of Figure 20 and 21 show the average number of time steps and accumulated cost per trial for the agent using the non-uniform CMAC function approximator, respectively. The error bars represent the ± 1 standard deviations of the mean across replications.

4.4.2.2 Uniform and Non-uniform Case-Based

In these configurations the agent used a case-based representation of the Q-function. Several experiments were performed using different configurations of the instance-based function approximator but the agent was not able to successfully learn to maintain the bar balanced once it reached the goal. It appears that the instance-based function approximator does not have enough resolution to effectively represent the value function due to the complexity of the non-linear dynamics of the pendulum. The case-based function approximator uses more resources and has better resolution than the instance-based. Each case in the case-based function approximator represents a point in the state space and the value of 6 equally spaced actions in the interval $[\alpha_{\min}\ \alpha_{\max}]$. The blending factor was set to $\rho = 0.6$, which means that during the Q-value computations each case contributed 40% according to the Q-value associated with the state and 60% according to the Q-values associated with the actions. The density threshold was set to $\tau_d = 0.05$. The smoothing parameters for the state space and the action space were set to $\tau_k^x = 0.05$ and $\tau_k^u = 0.2$, respectively. The similarity metric for the input space was a weighted Euclidean distance and the kernel function was the Gaussian. This produces cases with elliptical receptive fields with the major axis oriented diagonally in the state space, which bias the function approximator to generalize more in regions in which the angle and the velocity of the pendulum have the same sign. This kind of bias proved to be very useful once the bar reached the goal position and the active balancing began. The similarity metric for the output space was the Euclidean distance. The SARSA algorithm in Figure 6 was used to update the values of each case using the eligibility traces indicated by

Equations 43 and 44. The values for the free constants were $\gamma = 0.99$, $\lambda = 0.7$, $\alpha = 0.4$, and $\epsilon = 0$. The one-step search was performed using 24 equally spaced values for the acceleration between the minimum value, $\alpha_{\text{min}} = -3$, and the maximum value, $\alpha_{\text{max}} = +3$ (i.e., $24\Delta_u = \alpha_{\text{max}} - \alpha_{\text{min}}$). Table 9 summarizes the agent's design for the uniform and non-uniform configurations.

The bottom row of Figures 20 and 21 show the average number of time steps and accumulated cost per trial for the agent using the uniform case-based function approximator, respectively. The error bars represent the ± 1 standard deviations of the mean across 36 replications.

For the non-uniform configuration, the agent used the same case-based configuration as in the previous experiment but using the skewed version of the three variables of interest. The skewing functions were the same as the one used for the non-uniform CMAC (i.e., $p^s = \sqrt{p}$, $v^s = \sqrt{v}$, and $a^s = \sqrt{a}$). The effect of using the case-based function approximator on the skewed state-action space is similar to having non-constant density and smoothing parameters. Thus, the density of cases increases near the origin because that region gets expanded by the skewing function. Similarly, the smoothing parameter adjusts accordingly so that the increased density of cases near the origin does not produce too much blending of their values.

The bottom row of Figures 20 and 21 show the average number of time steps and accumulated cost per trial for the agent using the non-uniform case-based function approximator, respectively. The error bars represent the ± 1 standard deviations of the mean across 36 replications.

4.5 Discussion

We designed the previous experiments to contrast CMAC, instance-based, and case-based function approximators on the double integrator and pendulum swing up problems. This section discusses in detail the results obtained in those experiments so that we can draw some conclusions about the robustness and effectiveness of each function approximator.

4.5.1 Double Integrator

Table 10 shows the average and standard deviation of the cumulative cost and steps for CMAC, instance-based, and case-based function approximators at trial 50 on the double integrator problem.

The effect of preallocating resources through the use of skewing functions is clear with both CMAC and instance-based function approximators but not with the case-based. In the CMAC, there is statistical evidence at the 2% confidence level at trial

Table 9: Design of the Uniform and Non-uniform Case-Based Agent for the Pendulum Swing Up.

Factor	Description	
	Uniform	Non-uniform
Variables	2 inputs: position: $\theta \in [-\pi\ \pi]$ velocity: $\omega \in [-2\pi\ 2\pi]$	
	1 output: acceleration: $\alpha \in [-3\ 3]$	
Case structure	Number of actions: 6, equally spaced in $[-3\ 3]$ Blending factor: $\rho = 60\%$	
Distance functions	Input space: Weighted Euclidean. $d_M(x_i, x_q) = \sqrt{(x_i - x_q)^T M^T M (x_i - x_q)}$ $M = \begin{bmatrix} 1 & 1 \\ -1.75 & 1.75 \end{bmatrix}$ density threshold: $\tau_d = 0.06$	
	Output space: Euclidean. $d_M(u_i, u_q) = \lvert\alpha_i - \alpha_q\rvert$	
Kernel functions	Input space: Gaussian. $K^x(d_i^x) = \exp\left(\frac{d_i^x}{\tau_k^x}\right)^2$ smoothing parameter: $\tau_k^x = 0.05$	
	Output space: $K^u(d_i^u) = \exp\left(\frac{d_i^u}{\tau_k^u}\right)^2$ smoothing parameter: $\tau_k^u = 0.2$	
Skewing functions	N/A	$\theta^s = \sqrt{\theta}$ $\omega^s = \sqrt{\omega}$ $\alpha^s = \sqrt{\alpha}$
One-step search	ϵ-greedy policy (Figure 5): $\epsilon = 0$ $\Delta_u = \frac{\alpha_{\max} - \alpha_{\min}}{24} = 0.25$	ϵ-greedy policy (Figure 10): $\epsilon = 0$ $\Delta_u = \frac{\alpha_{\max} - \alpha_{\min}}{24} = 0.25$
Learning	SARSA algorithm (Figure 6): $\gamma = 0.99$ $\alpha = 0.4$ $\lambda = 0.7$	SARSA algorithm (Figure 11): $\gamma = 0.99$ $\alpha = 0.4$ $\lambda = 0.7$

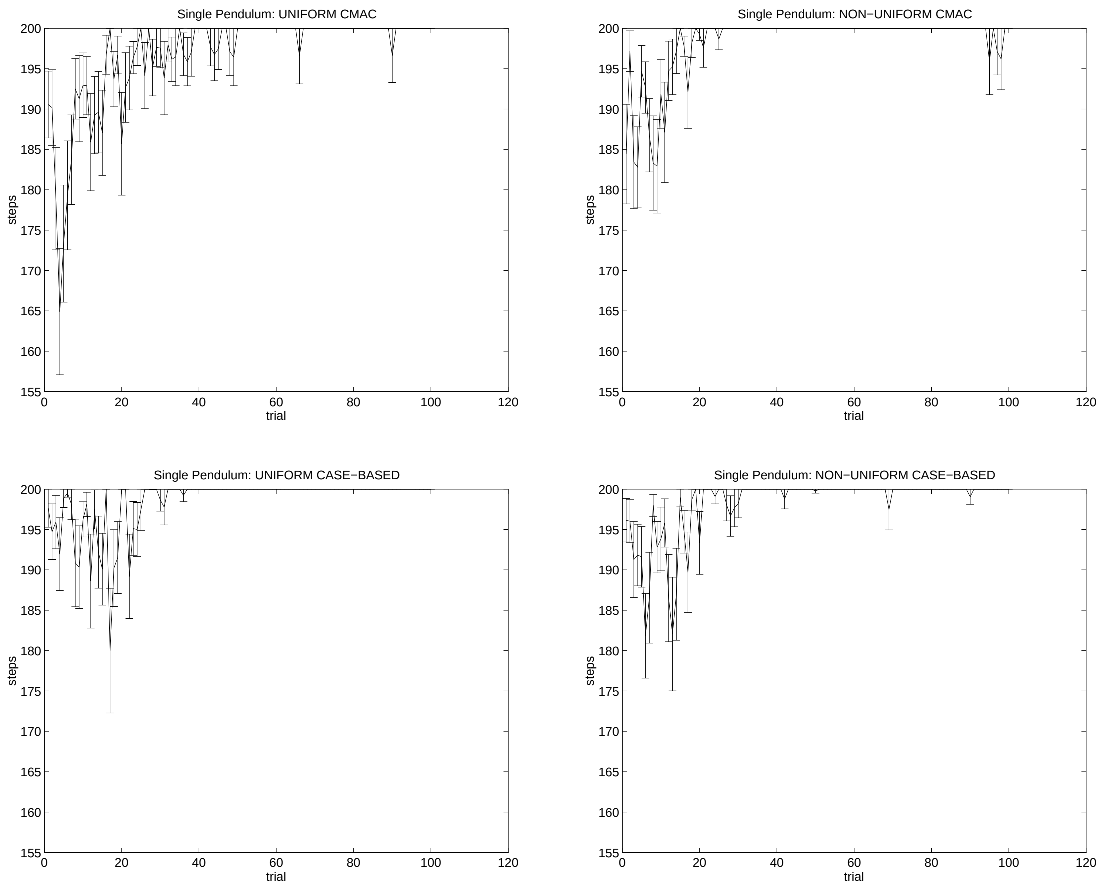

Figure 20: Average Steps per Trial in the Pendulum Swing Up Problem. The left and right columns show uniform and non-uniform configuration, respectively. Top row: CMAC. Bottom row: case-based.

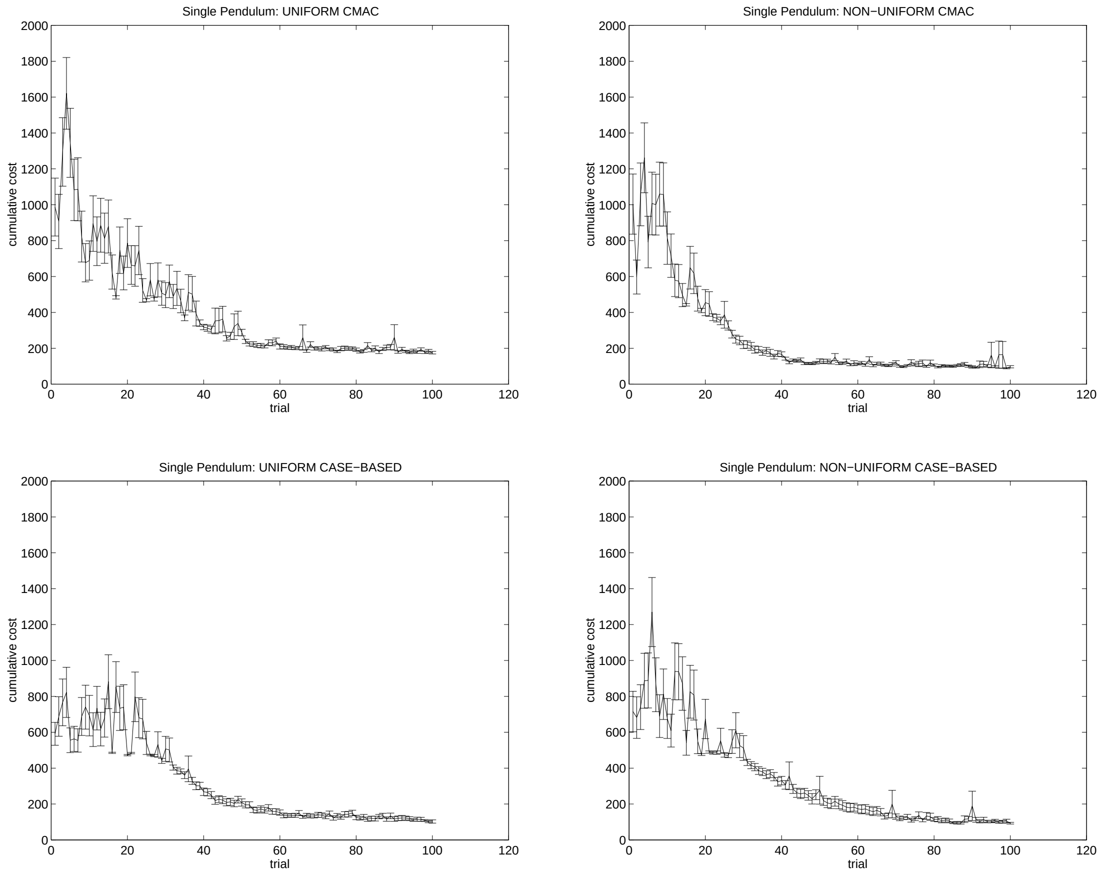

Figure 21: Average Cumulative Cost in the Pendulum Swing Up Problem. The left and right columns show uniform and non-uniform configuration, respectively. Top row: CMAC. Bottom row: case-based.

Table 10: Summary of Results for the Double Integrator at Trial 50.

Function approximator	Skewing	Cumulative cost [a]		Steps[b]	
		mean	std. dev.	mean	std. dev
Optimal	N/A	1.4293	0	200	0
CMAC	uniform	3.5100	0.1485	200	0
	non-uniform	3.0870	0.0909	200	0
Instance-Based	uniform	3.5552	0.2375	200	0
	non-uniform	2.3660	0.0944	200	0
Case-Based	uniform	2.4462	0.1854	200	0
	non-uniform	2.5178	0.1491	200	0

[a] Cumulative cost is the sum of all the immediate cost the agent received at every step during the trial. The lower values represent better solutions.

[b] Steps is the number of steps the agent took during the trial. The mean value 200 with 0 standard deviation means that the agent completed all 36 replications successfully.

50 that the cumulative cost achieved by the agent using the non-uniform version is smaller than the one achieved with the uniform version ($t_0 = 2.429$, P-value $= 0.018$).[10] Additionally, the learning performance of the agent using non-uniform CMAC stabilized faster (approximately trial 17) than one using the uniform CMAC (approximately trial 22) (see top row of Figure 18). Similarly, the first time the agent was able to stay in-bounds for the whole trial occurred at trial 17 when using the non-uniform CMAC. The same event occurred at trial 22 for the agent using the uniform CMAC (see top row of Figure 17). Also, after 50 trials, the agent using the non-uniform CMAC was more consistent than the agent using uniform CMAC because of the smaller variance on the performance (non-uniform: $\hat{\sigma} = 0.0909$, uniform: $\hat{\sigma} = 0.1485$). Summarizing, the main effects of preallocating CMAC tiles more heavily

[10] All test of hypotheses assume unknown and different variances for $\bar{X}_1$ and $\bar{X}_2$. The statistic

$$t_0^* = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}}}$$

can be used for testing $H_0 : \mu_1 = \mu_2$ in this case and it is distributed approximately as t with degrees of freedom given by

$$\nu = \frac{\left(\frac{S_1^2}{n_1} + \frac{S_2^2}{n_2}\right)^2}{\frac{(S_1^2/n_1)^2}{n_1+1} + \frac{(S_2^2/n_2)^2}{n_2+1}} - 2$$

if the null hypothesis is true.

near the origin of the state-action space are improvement in learning performance, faster learning, and better consistency. These effects are expected because the system spends most of the time around the origin trying to maximize the rewards. This requires very small adjustments using small changes in acceleration and the ability to represent the Q-function with enough resolution in that region of the state space. The skewing functions used to represent the non-uniform CMAC were designed for that purpose.

The effect of resource preallocation is also significant with the agent using the instance-based function approximator. At trial 50, there is statistical evidence at the 1% confidence level to reject the hypothesis that both versions, uniform and non-uniform, perform similarly (t_0 = 4.653, P-value = 0.000). Additionally, there is an improvement in the learning performance achieved with the agent using the non-uniform version over the agent using the uniform one because the former was able to stabilize the performance around trial 26 while the latter took until trial 43 (see middle row of Figure 18). Also, the agent using the non-uniform version learned to stay in-bounds for the whole trial at trial 21. The same event occurs at trial 26 when using the uniform version (see middle row of Figure 17). As with the CMAC version, there is a significant difference in the consistency of the performance since the variance of the non-uniform version is smaller than the variance of the uniform version (non-uniform: $\hat{\sigma} = 0.0944$, uniform: $\hat{\sigma} = 0.2375$). In summary, the effect of resource preallocation in the instance-based function approximator is similar than the one observed in the CMAC function approximator. Although the instance-based version allocates resources dynamically by creating new cases when exploring new regions of the state-action space, the effect of the skewing functions appears to show more heavily than the advantage of dynamically allocating resources in this task.

Unlike the agents using the CMAC and instance-based function approximators, the effect of resource preallocation does not appear to be significant in the agent using the case-based function approximator. At trial 50, there is not enough evidence to reject the hypothesis that both versions, uniform and non-uniform, perform similarly ($t_0 = -0.301$, P-value = 0.765). Additionally, the learning performance is similar because the agent is able to stabilize the performance around trial 16 in both versions (see bottom row of Figure 18). Also, the agent learns to stay in-bounds for the whole trial at trial 11 for both versions (see bottom row of Figure 17). Unlike the CMAC and instance-based versions, there is not a significant difference in the consistency of the performance since the variance of the uniform and non-uniform versions is about the same (non-uniform: $\hat{\sigma} = 0.1491$, uniform: $\hat{\sigma} = 0.1854$). In summary, the case-based function approximator is less sensitive than the CMAC and instance-based to the effect of resource preallocation. The case-based version allocates resources dynamically by creating new cases when exploring new regions of the state-action space. Also, it has better resolution than the instance-based version. Thus, the case-based function approximator is able to effectively take advantage of dynamically allocating resources.

There appears to be no statistical significant difference between the performance achieved with the uniform versions of the CMAC and instance-based function approximators ($t_0 = -0.161$, P-value = 0.872). However, there is a statistical significant difference between the performance achieved with the non-uniform versions; the instance-based version appears to be an improvement over the CMAC ($t_0 = 5.502$, P-value = 0.000). This difference may be dependent on the design decisions involved in both types of function approximators and the sensibility of such decisions on the learning behavior of the agent; further research is needed into this issue. Additionally, contrasting the performance curves of the agents using CMACs with the agents using the instance-based function approximator, it appears that the agents using the former are more consistent than the ones using the latter. The increased consistency achieved with the CMAC is presumably due to its capacity to smoothly generalize values to new regions of the state-action space. This does not occur with the instance-based version because there is an abrupt change or "peak" in the value function every time a new case is added to the library. Finally, there is a statistical significant difference between the performance achieved with the uniform versions of the case-based and the CMAC ($t_0 = 4.478$, P-value = 0.000), and the instance-based ($t_0 = 3.680$, P-value = 0.000). Both versions of case-based are an improvement over both versions of CMAC and the uniform instance-based.

In summary, the results show that the proposed case-based method is less sensitive to resource pre-allocation and perform better than both CMAC and instance-based on the double integrator problem. Furthermore, the skewing approach provides significant benefit to CMAC.

4.5.2 Pendulum Swing Up

Table 11 shows the average and standard deviation of the cumulative cost and steps for CMAC and case-based function approximators at trial 100 on the pendulum swing up problem. The agent using the instance-based function approximator was not able to successfully learn to maintain the bar balanced once it reached the goal.

The effect of preallocating resources through the use of skewing functions shows more significantly in the CMAC than the case-based function approximators. In the CMAC, there is statistical evidence at trial 100 that the cumulative cost achieved with the non-uniform version is better than the one achieved with the uniform version ($t_0 = 8.079$, P-value = 0.000). Additionally, the learning performance of the agent using non-uniform CMAC stabilized faster (approximately at trial 41) than

Table 11: Summary of Results for the Pendulum Swing Up at Trial 100.

Function approximator	Skewing	Cumulative cost [a]		Steps[b]	
		mean	std. dev.	mean	std. dev
CMAC	uniform	175.7926	7.3259	200	0
	non-uniform	97.3036	6.3807	200	0
Instance-Based	uniform	N/A	N/A	N/A	N/A
	non-uniform	N/A	N/A	N/A	N/A
Case-Based	uniform	102.7438	9.1446	200	0
	non-uniform	93.0292	4.7070	200	0

[a]Cumulative cost is the sum of all the immediate cost the agent received at every step during the trial. The lower values represent better solutions.

[b]Steps is the number of steps the agent took during the trial. The mean value 200 with 0 standard deviation means that the agent completed all 36 replications successfully.

the one using uniform CMAC (approximately at trial 52) (see top row of Figure 21). Similarly, the first time the agent was able to stay in-bounds for the whole trial across the 36 replications occurred at trial 35 for the agent using the non-uniform CMAC. The same event occurred at trial 50 for the agent using the uniform CMAC (see top row of Figure 17). At trial 100, the agents using the uniform and non-uniform CMAC achieved similar level of consistency across replication (uniform: $\hat{\sigma} = 7.3259$, non-uniform: $\hat{\sigma} = 6.3807$). Summarizing, the main effects of preallocating CMAC tiles more heavily near the origin of the state-action space are improvement on learning performance and faster learning. These results are similar to the ones achieved in the double integrator problem even though the pendulum swing up problem is much more difficult. The improvement in performance in the cumulative cost from 175.7926 for the agent using uniform CMAC to 97.3036 for the agent using non-uniform CMAC reveals the importance using a function approximator with higher resolution in the region near the goal state. The pendulum is highly unstable at this region and the agent is required to perform very small adjustments using small changes in acceleration in order to keep the bar balanced. Moreover, any mistake that causes the agent to lose control of the balance of the bar produces a lot of cost because it makes the bar to fall and oscillate.

The effect of resource preallocation in the case-based function approximators does not show as clearly as with the CMAC function approximators. At trial 100, there is not enough statistical evidence to reject the hypothesis that both versions, uniform and non-uniform, perform similarly ($t_0 = -0.945$, P-value = 0.349). Also, there is no noticeable difference in the learning performance in both versions since they both take around 60 trials to stabilize (see bottom row of Figure 21) and 35 trials to learn to stay in-bounds (see bottom row of Figure 20). There is, however, a significant difference in the consistency of the performance since at trial 100 the variance of the non-uniform version is smaller than the variance of the uniform version (non-uniform: $\hat{\sigma} = 4.7070$, uniform: $\hat{\sigma} = 9.1446$). However, the agent using the uniform version is able to stay in-bounds across all 36 replications after trial 37 whereas the agent using non-uniform version has some replications that achieve less than 200 steps after trial 37 (see bottom row of Figure 20). Presumably, this is because the skewing functions focus the excess resources in the region near the goal state, which produces insufficient generalization for the agent to regain control of the bar once it loses the balance and thereby forces the system to go out of bounds. In summary, the effect of resource preallocation in the case-based function approximator is less pronounced than in the CMAC function approximator. This is as expected because the case-based version allocates resources dynamically by creating a new case when exploring new regions of the state-action space. Thus, this type of function approximator is less sensitive than CMAC to the effects of skewed variables of the state-action space, which is a considerable advantage as it does not require the designer to accurately analyze the problem and design appropriate skewing functions for it.

There appears to be no strong differences in the performance at trial 100 of the agents using the non-uniform CMAC, uniform case-based, and non-uniform case-based function approximators. The cumulative cost achieved by the agents using these three types of function approximators is around 100 units whereas the one achieved by the agent using the uniform CMAC is around 175 units (see Table 11). In the context of learning rate and consistency of performance across replications, the agents using the case-based versions appear to take a few more trials to stabilize the performance and the agent using the non-uniform CMAC version appears to incur greater costs and less consistency at the earlier trials of the learning curve. However, these differences are not as marked as the differences obtained when contrasting the performance of these agents with the performance of the agent using the uniform CMAC. In summary, the learning performance achieved with the agents using the case-based versions is as good as the learning performance achieved with the one using the non-uniform CMAC. The effect of resource preallocation shows more clearly in the agents using CMACs because that type of function approximator cannot allocate resources dynamically in regions of the state-action space that require them the most.

4.5.3 Summary

Three different types of function approximators, CMAC, instance-based, and case-based, were tested in the double integrator and the pendulum swing up problems.

Both problems are characterized by dynamical systems having continuous state and action spaces. The results obtained in these experiments support the feasibility of using a function approximator to represent the Q-function in the state-action space and generalizing the value of individual experiences across states and actions.

Additionally, the results obtained demonstrate the feasibility of the modular implementation of non-uniform function approximators using skewing functions. The proposed approach appears to benefit the efficiency of function approximators that use static resource allocation such as the CMAC. On the other hand, the proposed approach does not appears to affect the efficiency of function approximators that use dynamic resource allocation such as the case-based. In particular, the effect of non-uniform resource preallocation resulted in faster learning, better performance, and improved consistency. These results showed more significantly in the pendulum swing up than in the double integrator presumably because it is more difficult to actively balance the pendulum than maintaining the car at the goal state. Therefore, the effect of using a non-uniform function approximator with high resolution at the region containing the goal state shows more significantly than using a uniform function approximator when contrasting the relative performance of the agents in the pendulum swing up and the double integrator problems.

Two memory-based function approximator were evaluated: instance-based and case-based. The instance-based function approximator uses only the state-action pairs the agent has actually experienced to represent the Q-function and new state-action pairs are incorporated when the distance to the nearest neighbor as measured by some metric function is larger than the density threshold. The case-based function approximator uses memory elements more complex than simply state-action pairs. Every case represents a state the agent has visited in the past and a set of actions along with their associated Q-values. The actions within a case are actions that the agent may or may have not experienced before. Nevertheless, this enhanced representation better supports credit assignment and generalization because the agent is able to update the associated Q-values of actions it has never experienced before based on state-action pairs that are similar according to the distance function. The case-based function approximator is able to solve problems that the instance-based function approximator may not solve as is the case of the pendulum swing up.

4.6 Conclusions

In this chapter we explored two important issues related to sequential decision-making problems applied to systems described by continuous state and action spaces. The first question asked about possible ways to generalize the outcome of experiences involving states and actions in continuous spaces. The chapter described some of the techniques researchers have used to represent the value function and proposed an alternative method to represent and generalize the value of states-actions pairs using the outcome of individual experiences. The idea consists of using only one function approximator to represent the Q-function on the combined state-action space and use the already familiar SARSA learning algorithm to adapt the parameters of the function approximator using TD(λ) backups. The disadvantage of this approach compared to other simpler ones is that the one-step search is computationally expensive. However, more efficient implementations of the one-step search may be possible by exploiting the class of function approximator being used. Additionally, there is no theoretical guarantee that this approach will converge to the correct Q-function (the function approximator may not even be able to represent the exact value function across the entire state-action space), but the results obtained on two different kinds of sparse coarse-coded function approximators on two different classes of continuous systems are encouraging. These results support the idea of using function approximators to solve reinforcement learning problems in systems with continuous state and actions. Function approximators are able to generalize the value for unseen regions of the state-action space based on the estimates of other regions of the state-action space. Also, they provide a compact representation for the Q-function and can be trained incrementally as the agent collects more data.

The second question asked about the effect of designing function approximators in which resource are preallocated across the state-action space. This is an important design decision because it directly affects the resolution capabilities of the function approximator in different regions of the state-action space. Usually, function approximators that uniformly distribute resources are more efficient and easier to implement than their non-uniform counterparts. The hypothesis is that the need for non-uniform function approximators is due to the better use of the resources that can be obtained by increasing the resolution of the function approximator in important regions of the state-action space. The results obtained tend to support this hypothesis. In addition, we used a technique common in machine learning that consists of preprocessing the input data to benefit learning performance. In our implementation, a skewing function is used to transform the original state-action space into a deformed version, which is then used as the domain of the uniform function approximator. The skewing function is empirically designed in advance to expand important regions of the state-action space so that the function approximator can use more resources to estimate the corresponding Q-values with more resolution. The use of the skewing function adds little computational complexity to procedure.

The chapter also described the feasibility of using memory-based function approximators in reinforcement learning problems with continuous state and action spaces. Memory-based function approximators belong to the class of sparse coarse-coded function approximators and can dynamically allocate resources with experiences.

The results obtained in this research showed that the memory-based function approximator was less sensible to preallocation of resources than the CMAC function approximator. We conjecture that this effect may be due to the capability of allocating more resources in those regions of the state-action space that are used more often, which allows them to dynamically adjust the resolution in those regions. This may serve as an advantage when designers do not know in advance which regions of the state-action space the function approximator should represent with higher resolution.

In the experiments presented in this chapter, we used predesigned skewing functions that remained constant throughout the experiment. However, it is also possible to use adaptive skewing functions so that the agent can dynamically increase the resolution of the function approximator as it collects more data. Future research will address this issue.

In this chapter we focused on the issue of continuous domains and function approximators. We studied problems in which the agents have perfect perception and complete knowledge, which means that the agent can perceive the complete state of the environment and uses states that do not incorporate new knowledge with the execution of each action. In Chapter 5 and 6 we focus on the problems of imperfect perception and incomplete knowledge in isolation. Because of the performance benefits shown in this chapter, we use case-based function approximators to represent the value function in all subsequent chapters in this dissertation.

CHAPTER V

IMPERFECT PERCEPTION

In the previous chapter, we focused on the issue of continuous domains and the design of function approximators. We studied the heuristic solution approach in problems in which the agent has access to the complete state of the environment (i.e., perfect perception). In this chapter, we focus on the problem of imperfect perception in sequential decision-making problems. The heuristic solution described in Chapter 3 considers a perception function, $\Psi(\cdot)$, that the agent uses to update its internal state based on the content of the previous state, the previous action, and the most recent sensation. In the context of perfect perception, each sensation delivers the complete state of the environment without noise and the role of the perception function is trivial: update the internal state with a replica of the most recent sensation. However, in the context of imperfect sensation, the perception function must extract and interpret all the relevant information contained in the most recent sensation and update the internal state accordingly. The better the internal state captures the current environment state the more accurately the function approximator can map the long-term benefit associated with that state.

In this chapter, we begin describing a new case-based implementation of the perception function the agent can use to process the incoming sensations when an a priori model of the environment is not available. Then, we formulate and solve the sequential decision-making problem in the context of imperfect perception and parameter-adaptive policies. The main problem faced by the agent when using imperfect sensors is that it cannot complete perceive the true state of the environment. In theory, all sensors are imperfect because of inherent inaccuracies associated with the sensor's detection components or overall sensor structure; however, many noise-filtering techniques often alleviate problems with sensor's signals contaminated with noise (e.g., Sorenson, 1966). A complex perception problem exists when the sensors' signal themselves represent only a small portion of the entire environment state. Consider, for example, the navigation task faced by a mobile robot equipped only with ultrasonic sensors. Ideally, the robot may accomplish its task with near-optimal performance if it had access to the relative positions and sizes of obstacles and destination with respect to itself (i.e., the complete environment's state[1]). In practice, however, the signals delivered by ultrasonic sensors represent noisy measurements about the relative distances to certain obstacles' surface; this does not directly reveal any information about positions or sizes of obstacles. The approach described in this chapter is designed to deal with this type of imperfect perception problems.

The agent uses a case-based implementation of the perception function to characterize the current environment situation (Ram and Santamaría, 1997). The case-based approach differs from other identification approaches in the sense that no explicit state estimation is attempted. Instead the perception function uses the recent history of sensations and actions directly to represent situations. Similar sequences of past history of sensation-action pairs are encoded as cases so that each case characterizes a specific environment state. The agent uses a case-based function approximator to map cases to the value (i.e., long-term benefit) of executing different actions at different situations and selects actions accordingly. In this chapter, we implement the heuristic approach to sequential decision-making at a higher level of abstraction. The sensations correspond to a set of preprocessed raw sensory inputs that provide environment descriptions and the actions represent different configurations of closed-loop control policies. Specifically, the actions represent parameters that configure low-level policies and the agent must learn how to select configuration parameters according to environment descriptions such that the performance index is maximized. We explore further the role of sequential decision-making in the adaptation of the parameters in reactive controllers in separate papers (Santamaría and Ram, 1997a; Santamaría and Ram, 1997b).

The problem studied in this chapter is autonomous robotic navigation. We implemented a robotic platform named SINS[2] to evaluate the implementation heuristic solution approach to this problem. We faced many design decisions in the implementation of the heuristic approach and we developed an evaluation methodology based on statistical tools that assess the merits of design decisions and environment characteristics (Santamaría and Ram, 1996). The evaluation methodology is applicable to a wide range of learning systems and it is described in detail in Chapter 8. We follow such methodology to evaluate the performance of the system in the context of four design decisions: two different sets of preprocessed inputs to characterize the environment and two different policies to select configuration parameters.

Section 5.1 describes the case-based approach to the perception function and the technical details for its implementation. Section 5.2 describe the technical details of

[1]Technically, the robot will also need to perceive the relative velocities of obstacles as well as its own.

[2]Self-Improving Robotic Navigation System.

the heuristic approach for parameter-adaptive control policies. Section 5.3 describes the implementation of the case-based perception function and parameter-adaptive policies in a mobile robot to perform autonomous robotic navigation. Section 5.4 describes the results of the evaluation methodology that demonstrates the approach in SINS using two different sets of preprocessed inputs to characterize environment situations and two different policies to select configuration parameters. Also, it presents the results obtained in a real robot. Section 5.5 concludes the chapter.

5.1 Case-Based Perception Function

This section describes an case-based approach to the perception function. The main role of the perception function $\Psi(\cdot)$ is characterize the current state of the environment. The function represents a procedure that updates the agent's internal state based on the content of the previous state, the previous action, and the current sensation. The ideal internal state is the one consisting of the entire sequence of previous sensations and actions $\{ z_0, \ldots, z_t, u_0, \ldots, u_{t-1} \} = \{ Z^t, U^{t-1} \}$ (i.e., an *information state*). However, the dimension of this type of state increases linearly with the number of stages, which results impractical to handle by a computer of any size. A reasonable approximation to the information state is a set of sequences of sensation-action pairs the agent commonly faces during the task. The case-based approach to the perception function consists of learning and using commonly occurring sequences or *cases* the agent faces as it interacts with the environment.

The case-based perception function uses a memory M consisting of a set of cases $C_i \in M$ containing specific sequences of sensations-actions pairs $\{ (z_0, u_0), \ldots, (z_l, u_l) \}$. Each case represents a sequence or "regularity" that occurs commonly as the agent performs the task and characterizes some region of the environment's state space. Cases are formed and refined by merging several similar past experiences. The agent uses a value function (or action-value function) mapping cases to long-term benefits to decide the action to execute at the current stage. The performance of the agent improves as a consequence of both synthesizing a representative set of sensation-action sequences and adapting the value function.

The content of the internal state of the agent at stage t is the case that best "matches" the current history of sensation-action pairs, $s_t = C^* \in M$. At every stage, the case-based perception function updates the current state of the agent by executing the following three steps: (1) Given the sensation z_t, it updates the current sequence of sensation-action pairs $\{ Z^t, U^{t-1} \}$. (2) It retrieves the case $C^* \in M$ that best "matches" the current sequence of sensation-action pairs according to some distance function. (3) It updates M based on the difference between C^* and $\{ Z^t, U^{t-1} \}$. The update may consist of (a) creating a new case in M, (b) modifying the content of the retrieved case C^*, or (c) extending the length of the sequence of sensation-action pairs encoded in C^*. The following subsections describes the details of the representational structures and algorithms involved to perform these steps.

5.1.1 Case Representation

Each case consists of a time sequence of sensation and action pairs. Sensations and actions are represented by real-valued variables in sensation and action spaces respectively. The sensation space consists of a m-tuple of m continuous sensory variables sampled at a given stage; each variable corresponds to the signal from a given sensor. Similarly, the action space consists of a p-tuple of p continuous action variables sampled at the same stage. A sequence of sensation-action pairs captures the recent history of sensations and their corresponding actions over a short window in time. Using this formulation, a case represents a portion of the trajectory the agent describes in the combined sensation-action space. The best matching case at any given stage corresponds to the case in memory that best describes the current trajectory followed by the agent at that stage. The agent can use the best matching case at a given stage to characterize the environment's state at that situation and select the most promising action according to the value function. This is discussed in more detail below. Figure 22 shows an example of a case. The memory M consists of a set of such cases.

The purpose of the case-based perception function is to use the agent's experience to create and refine cases that capture *consistent* sequences of sensation-action pairs. A pair is consistent over time when similar actions under similar environment situations produce similar results. Such sequences are advantageous because they are a reliable source of information when the agent faces new situations similar to situations experienced in the past. For example, when a robot is traversing a new situation it could use the same or similar actions as those used in previous similar situations and hope to get similar results. However, consistent associations may not always be useful. For example, if a robot detects it is near an obstacle and lowers the importance of avoiding such obstacle, it is very likely that a collision will occur. Such a sequence of associations is consistent but not useful since it deteriorates the navigation performance. However, it is the purpose of the value function to assign long-term benefit values to situations according to the reward function. The purpose of the perception function is to remember sequences of sensation-action pairs the agent can use to characterize situations.

Representing cases using real-valued variables is useful for several reasons. First, sensory information can be quickly processed using real-valued functions to produce a tuple that can be used to designate the current environmental situation. Fast information processing is a requirement because reactive agents must act in real-time.

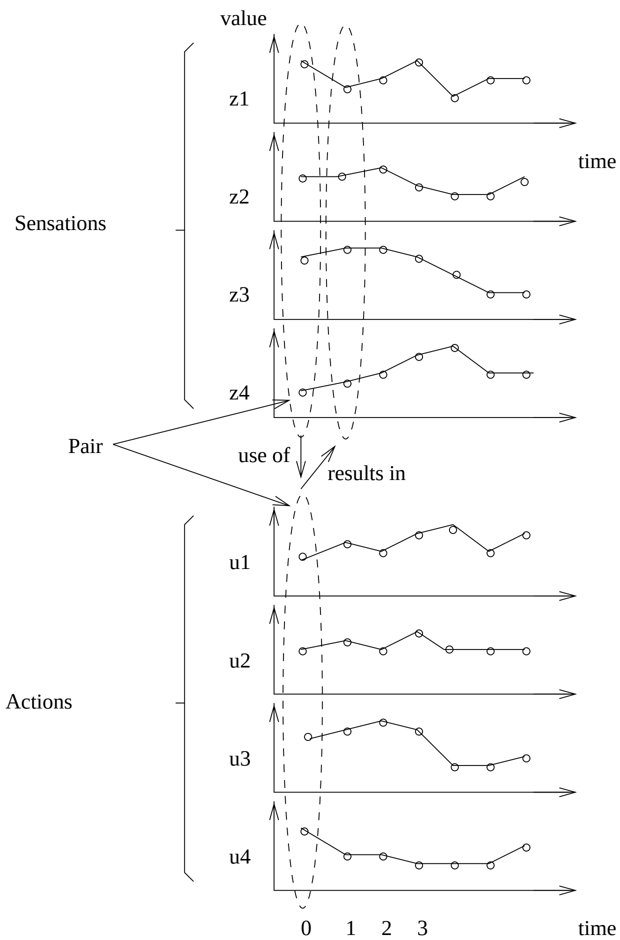

Figure 22: An Example of a Case.
The points represent the samples of sensations and actions. Sensation-action pairs arranged vertically and the sequence of pairs are arranged horizontally.

Second, real-valued variables, due to their fine grain-size, can be modified gradually with new incoming information. Gradual modification of cases ensures that the mapping from situations to long-term values is updated continuously and incrementally with every single experience. Such gradual modification of the value function helps to ensure stability in the system since the total behavior of the system is being modified in a continuous fashion. Third, real-valued variables allow extrapolation of values. This is useful when a new action is required in a region of the sensation-action space that has not been explored before. The cases of adjacent regions can be used to extrapolate the values in the required region. Finally, real-valued variables are able to represent a wide range of sensations and actions. This allow cases to capture any sequence of sensation-action pairs as long as they are proven to be consistent.

An additional innovation introduced in this approach is the time-history representation of sensations and action pairs. Keeping time-history information allows explicit representation of causal patterns between sensations and actions. Representation of causal patterns is a requisite for the detection and capture of consistent sequence of sensation-action pairs. Such representations also allow the agent to learn a broader class of policies. For example, a mobile robot might learn what to do when it is *approaching* an obstacle (a temporal concept) rather than merely what to do when it is *near* an obstacle (a static concept). Note that time-history representations are different from representations of temporal concepts (such as velocity); a time-history representation captures the successive values of the represented variable (which may in fact be a temporal variable such as velocity) over time.

5.1.2 Learning Cases

Case-based reasoning and learning deals with the issue of using past experiences to solve new problems. However, when the domain is represented using real-valued variables, remembering each single experience as a case is impractical due to bounds imposed by limits on the available memory. The alternative is to combine several similar experiences together to create an "average" or "virtual" experience or centroid in a region and let a case represent this centroid instead of each single experience (see Porter, Bareiss, and Holte, 1990, for another example of a case-based reasoning system, PROTOS, where cases represent "prototypical" experiences). In this way, each case represents all the points in the input space that are close enough to its centroid so that any information in the case can be generalized to the points inside the region that it represents. One problem with this alternative is that there is no a priori information about where such centroids should be located or how big the regions should be. The perception function must learn the centroids and the region's boundaries with each experience. Since experiences come one after another, cases must be learned gradually and incrementally. This means the content of a case depends not only on a past experience, but also on alterations introduced by subsequent similar experiences. This approach deviates from standard case-based reasoning where each case contains a previous experience or a generalization of a previous one and future similar experiences do not modify the content of an existing case (but see Ram, 1993).

The basic idea underlying learning of cases is that of competitive learning, which intuitively consists of letting all existing cases in memory compete to incorporate each new experience and allowing only the "winner" to modify itself to become more similar to the new experience. More formally, competitive learning can formulated as follows: Assume X_t represent the l most recent samples of sensation-action pairs at stage t, $X_t = \{ (z_{t-l+1}, u_{t-l+2}), \ldots, (z_t, u_{t-1}) \}$, and a set of reference cases $M = \{ C_i, i = 1, \ldots, N \}$. Assume that the C_i have been initialized in some proper way; random selection will suffice. For every given X, the best matching case C^* according to some distance metric $d(X_t, C_i)$ is updated to match even more closely to X_t. In other words, the best matching case C^*, is altered in some way to reduce the distance $d(X_t, C^*)$, and all the other cases C_i's are left intact. The result of this procedure is that the different cases tend to become specifically "tuned" to different regions of the input space of X_t in such a way that the cases C_i's tend to be located at the centers of regions of the input space according to the input probability distribution (see Kohonen, 1990). In other words, the cases self-organize themselves in order to occupy different regions of the input space according to the input probability distribution.

The distance function $d(\cdot)$ must match the samples of X_t against the samples of C_i at different relative time positions. Such a distance function can be formulated as follows: Let k denote a time index ($k = -l, \ldots, -2, -1, 0$) where 0 represents the current time and negative values represent the recent past. Let $C_i(j) = (z^i_j, u^i_{j-1})$ represent one sensation-action sample in a sequence of l_{C_i} samples of case C_i, $C_i(j) \in \{ C_i(0), \ldots, C_i(l_{C_i}) \}$, where l_{C_i} represents the total number of samples or *length* of case C_i. Then $d(X_t, C_i, k)$, the distance between X_t and C_i at relative time position k, can be defined as follows:

$$d(X_t, C_i, k) = \frac{1}{k+1} \sum_{j=0}^{k} \| X(t-k-j) - C_i(j) \| \tag{51}$$

where $k \in [0, \ldots, l_{C_i}]$ and $\|\cdot\|$ is the Euclidean norm (refer to Figure 23 for a graphical representation).

The best-matching case C^* and the position in time of the best match k^* is determined by the following equation:

$$d(X_t, C^*, k^*) = \arg\min_{C_i \in M,\, 0<k<l_{C_i}} \{ d(X_t, C_i, k) \} \tag{52}$$

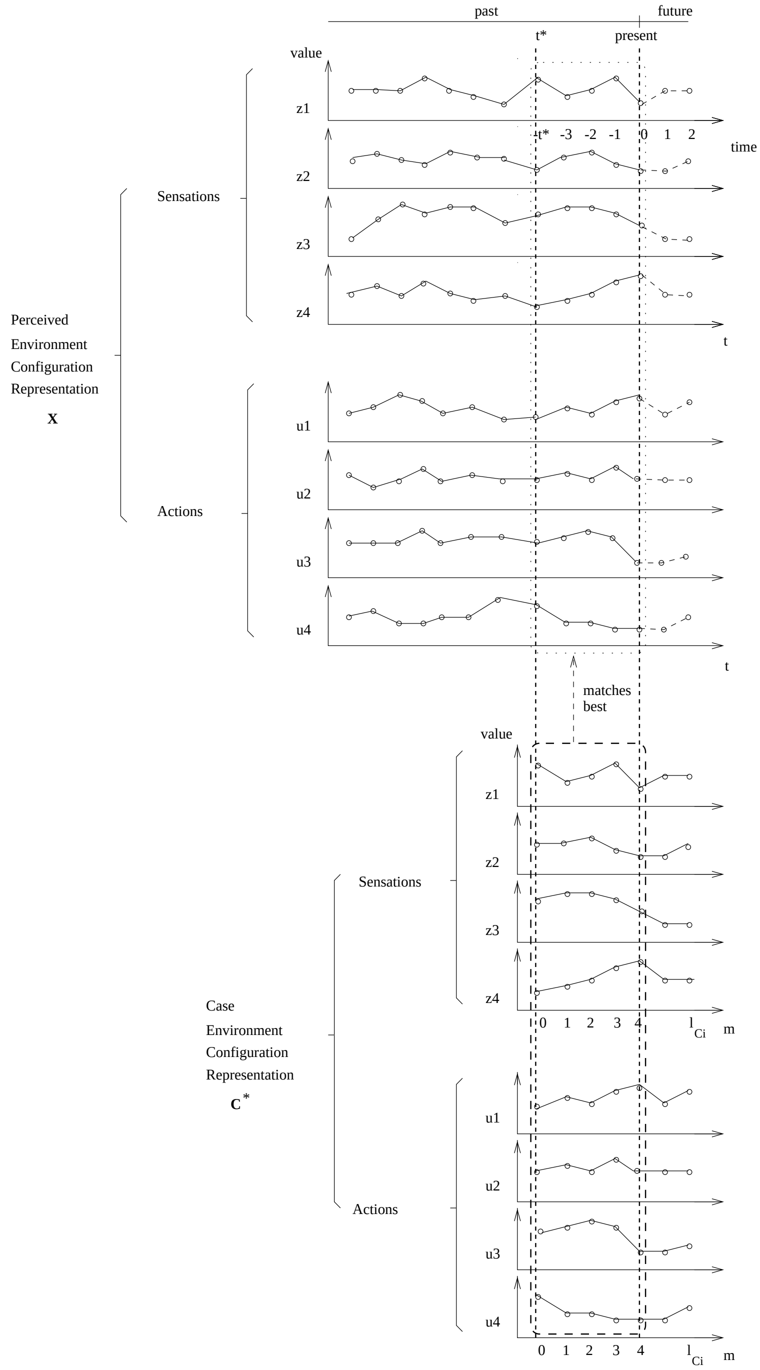

Figure 23: Schematic Representation of the Matching Process.
Each graph in the case (below) is matched against the corresponding graph in the current environment (above) to determine the best-matching case C^* and the time position k^* of the best match.

where k^* is the relative time position where the best match occurred. Equation 52 represents a process which matches all the cases in all relative time positions against the given sequence of sensation-action pairs, and selects the case having the minimum Euclidean distance per unit of time from the set of current available cases.

The update rule for the best-matching case C^* must also take into account the multiple samples along the time dimension. Equation 53 shows the version of this rule:

$$C^*_{\text{new}}(j) \;=\; C^*_{\text{old}}(j) + \tfrac{\alpha}{k^*-j+1}(X(t-j-k^*) - C^*_{\text{old}}(j)) \quad \text{for } j = 0, \ldots, k^* \tag{53}$$

Equation 53 represents a process which updates each case sample $C^*(m)$ to match its corresponding situation sample $X(j-k^*)$ with different strengths. The sample corresponding to the current time is updated more heavily than the samples taken back in time. In other words, the coefficient for updating the sample at the current time $j=0$ is α, and decreases linearly down to $\frac{\alpha}{k^*+1}$ as the position j of the sample in the time dimension goes from $t=0$ down to $t=-k^*$. This update rule has the nice effect of altering the cases evenly with their use, since the first samples in a case are used more frequently than its later samples and may be exposed to the update rule more often.

Two important decisions regarding case learning are when to create a new case (as opposed to using and modifying an existing case) and when to incorporate new samples to the end of a case. The first question addresses the issue of deciding when it is time to create a new case to map a region of the input space. Creating new cases is important because this allows the system to refine the coverage in regions of the input space that have not been considered before or that have not been covered in sufficient detail. The second question addresses the issue of capturing consistent sequences of samples into a case. This allow cases to grow in length to capture longer sequences of samples.

The decision rule to create new cases is based on the idea of minimum distance clustering, which consists of using a *density threshold*, τ, to decide whether a new experience should be incorporated into the closest case or used to create a new case. The threshold explicitly defines the border of the regions represented by each case and is updated with every experience until it converges to its best value. Given τ as the threshold value , the decision rule is defined as follows:

$$\text{decision rule}: \begin{cases} d(X_t, C^*, k^*) \leq \tau & \text{Adapt } C^* \text{ using Equation 53.} \\ d(x_t, C^*, k^*) > \tau & \text{Create a new case equal to } X_t \text{ at } k^*. \end{cases} \tag{54}$$

This means that a new case is created only when the distance between the current situation and the best-matching case is greater than the density threshold.

Intuitively, if the best available case is not as good a match to the new situation as it has been to the situations in which it has been useful in the past, then it is likely that the agent is encountering a situation for which the best available case isn't a very good match and a new case needs to be created.

The purpose of the decision rule for case learning (Equation 54) is to create different regions in the input space. Portions of the input space that are used more often are mapped with smaller regions (up to a spatial density controlled by τ) than portions of the input space used less often. In this way, as depicted in Figure 24, the mapping can be dynamically refined and specialized to those portions of the input space used more often. This results in the dynamic allocation of resources noted in the previous chapter.

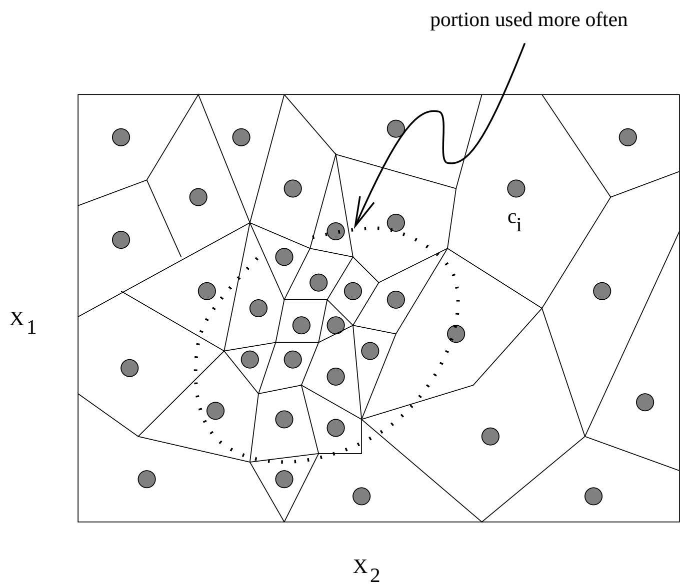

Figure 24: Self-Organization of Cases in Input Space.
The size of each region depends on the frequency of use of each portion of the input space.

The decision rule used to extend the length of the best-matching case is based on how well the case is able to predict the next sensation given the current sequence of sensations. Intuitively, if the best-matching case runs short in matching the current sequence of situations, then the case is extended to incorporate the current sensation. That is, if there is no other case that is closest to the end of the sequence, then the best decision is to extend the case that best matches the beginning of the sequence. In this way, the next time the case is retrieved under similar situations it will be longer and able to predict a longer sequence of situations. Mathematically, this can be accomplished by using Equation 52 to match cases at all relative time positions, including those that go one unit beyond the length l_{C_i} of the matching case $\mathbf{C}_i$. If the best-matching case C^* occurs at relative time position $k^* = l_{C*} + 1$, then case C^* is running short and must be extended. Equation 55 specifies the procedure:

$$\text{If } k^* = l_{C*} + 1, \quad \text{then} \begin{cases} l_{C*}^{\text{new}} = l_{C*}^{\text{old}} + 1 \\ C^*(l_{C*}^{\text{new}}) = X_t(0) \end{cases} \tag{55}$$

In summary, the above specifications describe the details of the case-based perception function. The function works by incorporating new experiences sequentially, creating new cases or updating the content of existing cases as required. Its asymptotic behavior is to partition the input space into different regions represented with cases. The cases may be thought of as *feature-sensitive detectors* that represent consistent sequences of sensation-action pairs. The sequences that are captured into cases are those that are intrinsic to the environmental situation and the dynamics of the environment. The main characteristics of this algorithm are that it creates and self-organizes cases to partition the input space based on the probability density function of the input space. Also, it works in an incremental manner since it incorporates new samples step-by-step. Finally, the algorithm can construct prototypical representations of sequences of situations that have proven to be consistent over time.

5.2 Parameter-Adaptive Policies

In most problems the agent's actions are *primitive* in that they are the low-level, general-purpose actions the agent can perform in most states, e.g., "rotate wheel by 90 degrees", or "move forward by 1 foot". This is an arbitrary and self imposed restriction; in general the set of actions can have a much more abstract relationship to the problem being solved. Specifically, actions may represent abstract configurations of goal-directed closed-loop control policies. For example, consider the following two different configurations for a reactive mobile robot: "hurried" and "patient". Both configurations are control policies able to take the robot to the destination; however, the former gives less importance to avoiding obstacles and more importance to approaching the goal and the latter does just the opposite. Instead of using always one configuration or the other, the mobile robot could use the "patient" configuration when it finds itself in highly cluttered environments and the "hurried" one in more open environments. In this case, the abstract policy would map environment situations to configurations and learning this kind of mapping is easier and more efficient than learning the actual movements the robot should take at every step considering the current low-level sensory information. The fundamental assumption is that learning the abstract policy is simpler than learning the entire control policy (see Mahadevan and Connell, 1991). This is usually the case because the learning task occurs at a higher level of abstraction than the operating level of the reactive controller.

The learning task of the agent consists of building a mapping function between the current environment descriptions and an efficient configuration of the control policy. This mapping will be referred to as the *abstract policy*, which the agent can use to continuously parameterize the control policy to the particular situation at hand in order to improve performance. The learning capability of this type of agents provides a significant advantage; designers do not need to (1) fine tune a single control policy that is applicable to all possible environment situations, and (2) specify the abstract policy since the agent can synthesize one using its own experience. The next subsections describe in detail the agent structure and the learning algorithms that implement these ideas.

5.2.1 Agent Structure

The agent performs the sequential decision-making task using two modules hierarchically: the *control* module and the *adaptation* module. The control module implements the *control policy* which is a mapping between low-level sensations and low-level actions designed to accomplish the agent's task. The adaptation module implements the *adaptive policy*, which is a mapping between environment situations and control policy configurations. Figure 25 shows the agent structure.

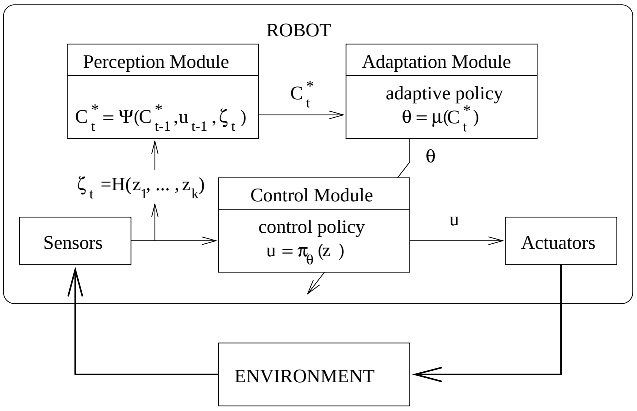

Figure 25: Agent Structure.

The control policy has two major characteristics. First, it is purely reactive because it does not use sensations to update internal models of the world that then guide action; instead, it directly uses current sensory information to select the low-level actions to execute next. Second, the control policy is usually designed as a combination of components that interact, each component being responsible for implementing a specific subtask (for example, in a mobile robot, the components implement subtasks or behaviors such as avoiding collisions or moving towards a goal). These two characteristics allow the control module to perform robustly in real-time since the mapping between sensors and actuators is implemented as quick reflexes without the reasoning necessary to update detailed world models (e.g., Arkin, 1989; Balch et al., 1995; Brooks, 1986; Langer, Rosenblatt, and Hebert, 1994; Mahadevan and Connell, 1991).

Mathematically, the control policy as a function mapping the set of inputs z delivered by the agent's sensors to the set of actions u sent to the agent's actuators:

$$u = \pi_\theta(z) \tag{56}$$

where θ represents a parameter vector that configures the control policy.

The adaptive policy is a function that maps the current environment situation C_t^* to the configuration of the control policy θ.

$$\theta = \pi(C_t^*) \tag{57}$$

where C_t^* represents the internal abstract state of the agent (i.e., current environment situation) and θ represents the abstract action (i.e., configuration of the control policy). The perception function use a set of preprocessed abstract inputs ζ delivered by the abstract sensor function $H(\cdot)$, $\zeta = H(z_1, \cdots, z_k)$. The perception function extracts the current environment situation from the last k sensations $z_1, \cdots, z_k$. Thus, at every stage t, the adaptive module uses the adaptive policy to select a new parameterization $\theta_t = \mu(C_t^*)$ for the control policy. Then, the control module executes the control policy for k consecutive steps and uses the perception function to determine the new environment situation for the next stage $C_t^* = \Psi(C_{t-1}^*, u_{t-1}, \zeta_t)$. At the next stage, the entire cycle repeats when the adaptive module uses the adaptive policy to select a new parameterization based on the new environment situation.

The adaptive policy can be hand-designed when enough knowledge about the agent, the environment, and their interaction is known in advance (Ram et al., 1992). However, an alternative approach is to incorporate the heuristic approach to sequential decision-making and let the agent learn an appropriate adaptive policy with its own experience. In this way, the agent will be capable of dealing with situations not foreseen by its designers. This is discussed next.

5.3 A Case Study: SINS

This section describes a mobile robotic system (SINS[3]) that combines the case-based perception function with parameter-adaptive policies. The system demonstrates the application of the algorithms described in the previous section for adapting the parameters of a schema-based reactive controller (Arkin, 1989). In schema-based reactive control, basic behaviors or *motor schemas* such as obstacle avoidance and movement towards a goal individually recommend specific motor actions which are then combined to produce the final action of the agent. Potential fields or forces are used to represent the recommendations of the motor schemas; thus, a final motor action can be computed as the sum of the forces of each motor schema and delivered to the robot's effectors. Each motor schema uses low-level sensations and internal control parameters to compute its potential field, which recommends the direction and speed at which the robot is to move. For example, the motor schema AVOID-STATIC-OBSTACLE directs the robot to move itself away from detected obstacles, and its control parameter **Obstacle-Gain** determines the magnitude of the repulsive potential field generated by the obstacles perceived by the system. The forces produced by all the schemas are then summed to produce a resulting force that directs the actual movement of the robot. Simple behaviors, such as wandering, obstacle avoidance, and goal following, can combine to produce complex emergent behaviors in a particular environment. Different emergent behaviors can be obtained by modifying the simple behaviors. A detailed description of schema-based reactive control methods can be found in Arkin (1989).

The mobile robotic agent consists of a control module, which uses the AuRA schema-based reactive control architecture (Arkin, 1989) implemented on a Denning MRV-III robot, and an on-line adaptation module, which performs sequential decision-making to select the controller's parameters. The control module is the parameterized motor-schema based policy, which is responsible for moving the robot through the environment from the starting location to the desired goal location while avoiding obstacles along the way. The adaptation module has two responsibilities. First, it must monitor the progress of the robot and determine the current situation of the environment. For that purpose, it uses the case-based perception function. Second, it must perform on-line adaptation of the configuration parameters being used by the control policy to get the best performance. The configuration parameters are based on the current environment situation and the action-value function the robot has learned through its experience.

[3]Self-Improving Navigation System.

5.3.1 The Control Module

The reactive controller in this implementation used three motor schemas: AVOID-STATIC-OBSTACLE, MOVE-TO-GOAL, and NOISE. AVOID-STATIC-OBSTACLE directs the robot to move itself away from detected obstacles. MOVE-TO-GOAL schema directs the robot to move towards a particular point in the terrain. The NOISE schema makes the robot move in a random direction; it is used to escape from local minima and, in conjunction with other schemas, to produce wandering behaviors. Each motor schema has a set of parameters that control the potential field generated by the motor schema. In this research, the system used the following parameters: **Obstacle-Gain**, associated with AVOID-STATIC-OBSTACLE, determines the magnitude of the repulsive potential field generated by the obstacles perceived by the robot; **Goal-Gain**, associated with MOVE-TO-GOAL, determines the magnitude of the attractive potential field generated by the goal; **Noise-Gain**, associated with NOISE, determines the magnitude of the noise; and **Noise-Persistence**, also associated with NOISE, determines the duration for which a noise value is allowed to persist.

The control module operates on a Denning MRV-III robot. The robot uses a Denning sonar ring which has twenty-four laboratory grade Polaroid Ultrasonic Rangefinders equally spaced over 360 degrees in a plane parallel to the floor. Range data is obtained by emitting ultrasonic pulses from the sensors. There are four variables configuring the control policy that represent the schema parameter values, one for each of the schema parameters (**Obstacle-Gain**, **Goal-Gain**, **Noise-Gain**, and **Noise-Persistence**) discussed earlier. The values are set periodically according to the recommendations of the best case. The new values remain constant over a four steps until the next setting period.

5.3.2 The Reward Signal

Successful navigation occurs when the robot reaches the destination point as fast as possible and without any collisions. Thus, the robot needs to minimize the time taken to reach the goal with the fewest number of collisions. This is represented in the following reward signal:

$$r = -Q_o \,\text{virtual\_collisions} + Q_s\,(\frac{\text{velocity}}{\text{max\_velocity}}) \tag{58}$$

This formula is based on the assumption that the only sensory information available to the robot consists of the 24 ultrasonic sensors and the shaft encoders. This means that the robot receives a negative signal every time a virtual collision[4] is detected and a positive signal every time the robot is moving; the faster it moves the stronger the signal. The coefficients Q_o and Q_s determine the relative importance of avoiding collisions versus maximizing velocity for a particular application. This reward signal is not computationally expensive to evaluate, uses information provided by the robot sensors, and is consistent with the performance metric defined for successful autonomous robotic navigation.

5.3.3 Adaptation Module

The ultimate goal of the adaptation module is to learn an adaptive policy for selecting configuration parameters given the environment situations that maximizes the performance index (i.e., discounted sum of rewards) subject to the constraints of the reactive controller. The ideal mapping $\mu^*(\cdot)$ is such that when the robot is in situation C_t^*, then $\theta^* = \mu^*(C_t^*)$ is the optimal parameterization of the reactive controller for that situation. The current environment situation is the output of the case-based perception function described earlier.

The adaptive module uses the action-value function approximator, $\hat{Q}_w(\cdot)$, to decide which configuration parameters to use in the current stage. This function maps every environment situation and configuration parameter combination to an estimate of the best performance index or long-term benefit the agent can receive from the situation when it applies the given parameterization. Thus, the idea consists of letting the controller perform a computationally inexpensive search to select the parameterization leading to the best long-term benefit estimate according to the function approximator, and then use the outcome of each action to asymptotically improve the estimates by adjusting the function approximator. The action-value function is implemented using by associating to each case in the M a set of N_i different actions θ_{ij}, their associated Q-values Q_{ij}, and their eligibilities e_{ij} $(j = 1, \ldots, N_i)$. Figure 26 shows the SINS algorithm.

5.4 Evaluation

SINS is fully implemented in `C++` on top of Arkin's (1989) AuRA architecture for schema-based reactive control. The system works in two modes: real and simulation. In real mode, the system controls a Denning MRV-III robot through a radio link; the

[4]Since there is a real robot involved we can not afford actual physical collisions. We use the term virtual collision to refer to the situation when the robot come so close to an obstacle that a physical collision is imminent. When this occurs, AuRA automatically diverts all its resources to focus on eliminating this imminent danger.

1. **Initially**:
   Set the initial estimate of the value function: w_0
   Set the initial configuration parameter vector: θ_0

2. **Perform**:
   At each stage $t = 1, 2, \cdots$

   (a) **Decision**:
   Select best parameterization at stage t:

   $$\hat{\theta}_t^* = \arg\max_{\theta_t} E\left[\hat{Q}_w(s_t, \theta_t)\right]$$

   Use the control policy: $u = \pi_{\hat{\theta}_t^*}(z)$ for k steps.

   (b) **Perception**:
   Observe the resulting reward and sensation for k steps:

   $$r_{t+1} = R(z_1, \ldots, z_k, \pi_{\hat{\theta}_t^*}(z_1), \ldots, \pi_{\hat{\theta}_t^*}(z_k))$$

   Compute the current environment situation:

   $$\begin{aligned} \zeta_t &= H(z_1, \ldots, z_k) \\ C_t^* &= \Psi(C_{t-1}^*, u_{t-1}, \zeta_t) \end{aligned}$$

   (c) **Adaptation**:
   Adapt the action-value function approximator using TD(λ):

   $$\Delta w_t = \alpha\left(r_{t+1} + \gamma\hat{Q}_{t+1} - \hat{Q}_t\right)\sum_{k=0}^{t}(\lambda\gamma)^{t-k}\nabla_{w_k}\hat{Q}_k$$

Figure 26: SINS Algorithm.

perceptual system of the robot consists of a Denning sonar ring which has twenty-four laboratory grade Polaroid Ultrasonic Rangefinders equally spaced over 360 degrees in a plane parallel to the floor. In simulation mode, the system simulates the values of the Rangefinders and the dynamics of the actual robot.

The methods presented above have been evaluated using extensive experiments across a variety of different types of environments, performance criteria, and system configurations. The results presented here are based largely on experiments performed with the system running in simulation mode, since that allows us to run several hundred experiments in which design parameters and environmental configurations are systematically varied. We measured the qualitative and quantitative improvement in the navigation performance of the SINS system and systematically evaluated the effects of various design decisions on the performance of the system. The results show the efficacy of the methods across a wide range of qualitative metrics, such as flexibility of the system and ability to deal with difficult environmental configurations, and quantitative metrics that measure performance, such as the number of navigational problems solved successfully and the optimality of the paths found for these problems. In addition to the simulation experiments, we also evaluated the system's performance in real mode to verify the validity of our approach.

SINS, like many other robotic navigation systems, is complex and the domain it operates in is also complex. One result of this is that the behavior of SINS has many sources for variability which cause any performance measure defined to evaluate this behavior to have variability as well. This in turn makes it difficult to assess the significance of an observed behavior of the system in a specific situation. Straightforward performance curves that show how the performance of a system improves over time are not good enough: Although these curves show that the performance improves on the specific test problems, they do not provide useful information about why the system works or how the design decisions affect the behavior of the system, nor can they be used to predict the behavior of the system under different circumstances (especially given the variability in the behavior of the system). Ablation studies can be used to analyze the impact of different system's modules in the performance of the system (Cohen and Howe, 1988; Kibler and Langley, 1988). In such studies, one or more system modules are removed or deactivated to analyze how the performance of the system changes. Although these studies do provide some information about the merit of different modules in the performance of the system, they are based on extreme operating conditions that are often impractical (i.e., one or more modules are set to be either active or inactive). Moreover, design decisions often deal with allocating certain amount of resources to different modules. Due to their nature, ablation studies can only deal with all-or-nothing resource allocation, disabling the possibility of deciding what would be the optimal amount of resources to allocate to each module.

We developed a systematic statistical evaluation methodology (described in detail in Chapter 8) to evaluate the performance of the SINS system. In this methodology, statistical tools are used to analyze the change in the performance of the system in terms of changes in design decisions and domain (or problem) characteristics. In such an analysis, the system is evaluated through systematic experiments designed to filter out undesirable sources of variability. The results of the experiments are then analyzed using statistical tools to construct a mathematical model of the behavior of the system in terms of design parameters and domain characteristics. This model is used to assess the merit of various design decisions in the algorithm and to analyze the performance of the system when it faces problem domains with different characteristics. Such an evaluation enables us to understand the behavior of the system in terms of the theory and design of the computational model, select the best system configuration for a given domain, and predict how the system will behave in case the characteristics of the domain change.

5.4.1 Systematic evaluation of SINS

The performance of SINS varies across different worlds, system configurations, and amounts of experience. Moreover, due to the nature of the task and the architecture of the system, SINS can perform differently given the same world and case library. The reason for this is that the adaptation module tunes the navigation module randomly when no appropriate case exists; this allows the system to explore and discover new regularities. This means that any performance metric used to evaluate the system must be treated as a random variable and statistical estimation techniques should be used to assess its mean value.

The next subsections describes in detail the systematic evaluation studies performed on SINS. The first set of experiments (Study 1) focuses on how the choice of input representation and parameter-selection policy influence the performance of SINS. Both of these factors represent high-level design decisions that are important in the design of the agent. The input representation refers to the type of sensations that the adaptation module of SINS uses to represent and categorize the surrounding environment (situation). The parameter-selection policy refers to the method by which this module selects the control parameters suggested by the best matching case to the current situation.

The second experiment (Study 2) verifies the performance improvement provided by the approach on an actual robot. We measure the performance of the robot performing in a indoor room with a few static obstacles over several trials. We plot the performance curves against the number of trials to verify that the navigation performance improves along different metrics as the number of trials increases. Further performance and learning results are reported in Ram and Santamaría, 1997 and Santamaría and Ram, 1996.

5.4.2 Study 1: Design Decisions

Using the evaluation methodology, we performed a systematic evaluation study to find an empirical model that describes the relationship between the choice of input representation, parameter-selection policy, and system performance as well as the conditions under the model is applicable. In this study, we evaluated two choices of input representations and two choices of parameter-selection policies. The input representation refers to the information that the learning and adaptation module of SINS uses to represent and categorize the surrounding environment (situation). The parameter-selection policy refers to the methods by which this module adapts the control parameters suggested by the best matching case to the current situation.

The reason for studying choices of input representation is to analyze the influence of the "level of information" provided as input on the performance of the system. One option is to use sensory inputs with "low-level" information. In the navigation domain, this might involve using an ultrasonic sensors to measure the distance of the closest obstacle in the direction of the goal. Another option is to use sensory inputs that encode "high-level" information; for example, using an array of ultrasonic sensors to compute a measure of the density of obstacles surrounding the robot. The former type of sensory input is very specific and may allow SINS to discover the best control parameters the robot may use in specific situations. However, we would also expect that the cases learned by the system to be very specific and not work as well in similar but not identical situations. In contrast, the latter type of sensory input is more generic and may allow SINS to discover good control parameters. However, the learned cases may be so coarse that the system may not perform near-optimally in most cases.

We studied two choices of input representations: the low-level and high-level information types introduced earlier. The low-level information type consists of the following four sensory inputs: **Obstacle-Distance-Ahead**, **Obstacle-Distance-Behind**, **Obstacle-Distance-Right**, and **Obstacle-Distance-Left**. Each of these variables provide a measure of the nearest obstacle that impede navigation in the direction towards, contrary to, right of, and left of the direction of the perceived goal respectively. The high-level information type consists of the following four sensory inputs: **Obstacle-Density** provides a measure of the occupied areas around the robot that impede navigation;[5] **Absolute-Motion** measures the activity of the system; **Relative-Motion** represents the change in motion activity over an appropriate interval; and **Motion-Towards-Goal** specifies how much progress the system has actually made towards the goal. Both types of sensory inputs are computed and constantly updated using the information received from the robot's physical sensors (in our case, 24 ultrasonic sensors arranged around the robot every 15 degrees and shaft encoders.).

Another design decision in SINS is the choice of the parameter-selection policy. Every few steps, as determined by a configuration parameter, the adaptation module may recommend new control parameters to the navigation module. To accomplish this, it retrieves the case most similar to the current environmental situation over a recent time window and suggest new control parameters from the retrieve case according to a parameter-selection policy. The best control parameters would be those that provide the best long-term benefit (as measured by the reward function, such as time spent and number of collisions with obstacles). In order to learn appropriate control parameters, SINS must "explore" the control parameter space, that is, it must try several values for each environmental situation and learn which values produce positive or negative results. This presents at least two design options for the parameter-selection policy. One option is to select the new values of the control parameters stochastically according to the Q-function. In this method, control parameters with higher values are more likely to be selected than those that have produced lower values. Another option is to select the control parameters with the best value. The former policy enforces more exploration and may discover better solutions than the latter one. On the other hand, due to the same reason, the latter method may converge faster than the former. We studied two choices of parameter-selection policies: the stochastic policy and the select-best policy. The stochastic policy selects control parameter values randomly but favoring values that lead to better long-term benefits. Specifically, parameters are selected according to a Boltzmann distribution: $P(\theta_i) = \frac{exp(\hat{Q}(C^*,\theta_i))}{\sum_j exp(\hat{Q}(C^*,\theta_j))}$, where the C^* represents the case that best matches the current situation and $\hat{Q}(C^*,\theta_i)$ represents the value associated with control parameter θ_i according to the current Q-function approximation. The select-best policy simply selects the control parameter leading to the maximum value of the Q-function, $\hat{\theta}^* = \arg\max_{\theta_i} \left[\hat{Q}(C^*,\theta_i) \right]$.

[5] Note that this sensory input does not provide any information about the distances or direction of the obstacles; it simply measures the density of occupied area around the robot.

5.4.2.1 Experimental Design and Data Collection

We performed experiments to collect data from all possible combinations of design decisions and used it to estimate the parameters of a linear model that relates the performance of the system (i.e., the median time: T) with the independent variables (i.e., the choice of input representation, the choice of parameter-selection policy, and the experience level). We used all-subsets regression analysis to determine which parameters were significant and performed model validation by analyzing the residual plots (see details in Chapter 8).

We evaluated the performance of SINS using the median value of the time it takes to solve a world. The reason for this is that the median is a robust estimator of the mean and is not too sensitive to outliers. Outliers are common in schema-based reactive control since the system can get trapped in local minima points, resulting in a significant change in the behavior of the system. An experiment consisted of measuring the time SINS takes to solve a world across five independent runs under the same conditions (i.e., same conditions but different random seeds) and reporting the median among the five runs as the response variable.

Table 12 shows the experimental design matrix for the study. It consists of a 2×2 factorial design: two choices for input representation (low-level and high-level), and two choices for parameter-selection policy (stochastic and best). Starting from no prior experience, every treatment was exposed to thirty levels of experience, each level being one complete navigation problem. We recorded the outcome of five replications for every level of experience. Thus, the total number of system runs in study was $2 \times 2 \times 30 \times 5 = 600$. The domain characteristics in this experiment were blocked by evaluating all the treatments under the same randomly generated 10% cluttered world. This allowed us to balance out the effects of the design decision choices and experience level (which were of interest) and block out the effects of other parameters such as world clutter.[6]

5.4.2.2 Model Construction

The performance of SINS was evaluated by estimating the median time to solve a world. Thus, the model that needed to be determined in this study has the median time (T) as the response variable, and would relate T with the choices of input representation and parameter-selection policy, and with amount of experience. Note that a lower value of T corresponds to better performance on the task. We used the following regressors as independent variables: input representation (I; low=0,

[6]The choice of the problem world is merely for illustrative purposes and is arbitrary as far as this chapter is concerned. The methodology can be also used to evaluate system performance on a range of problems instead of a single representative problem.

Table 12: Experimental Design Matrix.

Choice of input representation	high-level / low-level
Choice of parameter-selection policy	stochastic / best-value
Experience Levels	1-30
World Clutter	10%
Replicates	5
Response Variable	Time
Total runs	600

high=1), parameter-selection policy (P; stochastic=0, best-value=1), and experience level (E). We also considered additional regressors, such as the interaction terms IE, PE, IP, and IPE. The reason for considering all these factors is to allow for the possibility that interaction terms may explain variability in the response variable better than individual terms. Statistical analysis was used to reveal which of these terms were really significant and should be considered in the final model. Equation 59 shows the complete hypothetical model.

$$\begin{aligned} T \;=\; & \beta_0 + \beta_I I + \beta_P P + + \beta_{IP} IP + \\ & (\beta_E + \beta_{IE} I + \beta_{PE} P + \beta_{IPE} IP)E + \\ & \varepsilon \end{aligned} \tag{59}$$

Figure 27 shows the response variable T collected during the experiment. The figure shows that there are mainly two phases as experience increases: a learning phase, where performance improves; and a maturity phase, where performance remains constant (this is as intuitively expected; recall that SINS is learning to navigate a fixed world because domain characteristics have been blocked in this study). A single first-order or second-order polynomial would not be able to fit all the data with enough accuracy in both phases. As described before, there are two ways to solve this problem: either increase the number of terms in the model, or reduce the region of operability of the model. Since there are clearly two phases as experience increases, we decided to use two models, one for each phase.

Table 13 shows the alternative empirical models based on the design decision choices. The terms β_I and β_P measure the change in the intercept of the equation given a change in the choice of input representation and parameter-selection policy, respectively. The term β_{IP} measures the interaction effect of the design decisions

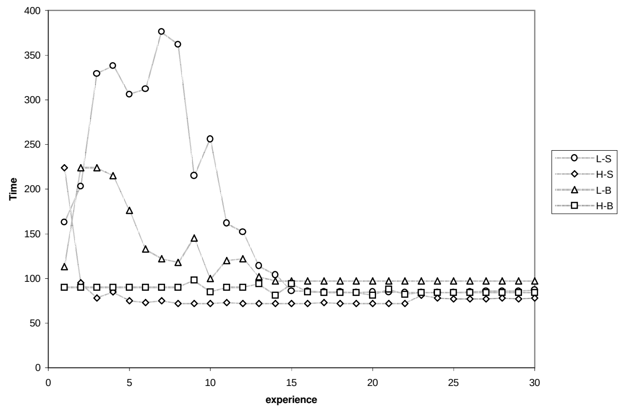

Figure 27: Performance of SINS vs. the Choices of Input Representation and Parameter-Selection Policy.
The four graphs correspond to low-level representation with stochastic policy (L-S), high-level representation with stochastic policy (H-S), low-level representation with best-value policy (L-B), and high-level representation with best-value policy (H-B).

on the intercept. The terms β_{IE}, β_{PE}, and β_{IPE} perform the same role but on the slope of the equation. In this context, the slope represents the learning rate of the system, that is, how much the system improves performance (decrease in T) per unit of experience (increase in E).

Table 13: Alternative Models for Study 1.

Input	Policy	Equation
low-level	stochastic	$\bar{T} = \beta_0 + \beta_E E$
high-level	stochastic	$\bar{T} = (\beta_0 + \beta_I) + (\beta_E + \beta_{IE})E$
low-level	best-value	$\bar{T} = (\beta_0 + \beta_P) + (\beta_E + \beta_{PE})E$
high-level	best-value	$\bar{T} = (\beta_0 + \beta_P + \beta_{IP}) + (\beta_E + \beta_{PE} + \beta_{IPE})E$

An all-subsets regression analysis was performed to determine which of the terms in the model are really significant, that is, which terms influence the response variable. In this analysis, all possible subsets of regressors are considered and a model is constructed using each subset. Tables 14 and 15 show the results of this analysis of the models for the learning and maturity phases. We measured the optimality of the model by its R^2_{adj}, the adjusted coefficient of multiple determination.[7] This coefficient measures the ability of the model to explain changes in the response variable by changes in the regressors. Its range is between 0%, which means that none of the variation in the response is explained by variation in the regressors, and 100%, which means that all of the variation in the response is explained by variation in the regressors. Thus, the larger the R^2_{adj} the more explicative is the model. This coefficient also indicates if we are overfitting the data by introducing more terms that artificially reduce the error in fit between the model and the data.

Table 14: Best Subsets Regression Results for the Learning Phase.

Number of variables	R^2_{adj}	$\hat{\sigma}$	I	P	E	IE	PE	IP	IPE
1	35.3	66.14	X						
2	45.6	60.64	X		X				
3	52.8	56.48	X				X		X
4	64.0	49.33	X	X	X				X
5	64.5	48.69	X	X	X	X		X	
6	65.9	47.99	X	X	X	X	X	X	
7	65.3	48.43	X	X	X	X	X	X	X

Table 15: Best Subsets Regression Results for the Maturity Phase.

Number of variables	R^2_{adj}	$\hat{\sigma}$	I	P	E	IE	PE	IA	IAP
1	50.9	5.61	X						
2	94.1	1.95	X	X					
3	95.0	1.79	X	X				X	
4	96.5	1.50	X	X		X			X
5	97.1	1.38	X	X		X		X	X
6	97.0	1.38	X	X	X	X		X	X
7	97.0	1.39	X	X	X	X	X	X	X

Tables 14 and 15 show the best model obtained within each subset of constant size or number of variables. In these tables, $\hat{\sigma}$ is the estimated standard deviation of the dependent variable in the model, and R^2_{adj} is the adjusted coefficient of multiple

[7]The R^2_{adj} is the R^2 value adjusted to take into account the number of parameters in the model. This allows models having different number of parameters to be compared.

determination as discussed earlier. The "X's" show which variables are included in the best model for each size.

During the learning phase, the best model obtained with the all-subset analysis corresponds to the one having six regressors as independent variables ($F = 20.04$, P-value = 0.000).[8] There is no statistical evidence that the term β_{IPE} is significant, that is, there is no evidence that the learning rate is affected by the interaction between the choice of input representation and parameter-selection policy. Table 16 shows the statistical results for each individual parameter in the model for the learning phase as well as the 90% confidence interval estimate of their real values.

Table 16: Model Coefficients for the Learning Phase.

Coefficients	Value	Std. Error	90% C.I.
β_0	334.19	23.420	(294.99,373.39)
β_E	-12.79	3.484	(-16.95,-8.63)
β_I	-209.56	28.870	(-257.89,-161.23)
β_P	-132.43	28.870	(-180.78,-84.10)
β_{IE}	7.90	2.868	(3.09,12.70)
β_{PE}	5.14	2.868	(0.34,9.94)
β_{IP}	96.00	24.780	(54.51,137.49)

The results show that the system configuration that achieves best performance is the one using the high-level representation and stochastic adaptation algorithm ($T = 51.73$ seconds at $E = 15$ experiences). However, the fastest learning rate corresponds to the system configuration with the low-level representation and stochastic adaptation algorithm ($\beta_E = -12.79\frac{\text{secs.}}{\text{exp.}}$) since changing the level of input representation or the parameter-selection policy reduces the learning rate (increases the slope). There is a big influence on the intercept (performance with no previous experience) caused by the design decisions. The interaction term (β_{IP}) shows that the amount of change in performance caused by changing one of the design decisions depends on the other design decision. For example, the average change in performance caused by switching from low-level to high-level input representation when using stochastic parameter-selection policy is $\beta_I = -209.56$ seconds. But, if the same switch is performed using the best-value parameter-selection policy, the average change in performance is only $\beta_I + \beta_{IP} = -113.56$ seconds. In both cases, it appears that

[8] The F statistic is used to determine the significance of the regression. The P-value is the probability determined by F; the lower this value the better the result, since the significance of the regression is $(1 - \text{P-value})\%$ which is 100% for P-value = 0.

using high-level input representations provides a better start in the learning process. There is no statistical evidence ($t = 0.68$, P-value $= 0.499$) to indicate that a change in the input representation improves the learning rate more than a change in the parameter-selection policy (i.e., $\beta_{IE} = \beta_{PE}$).

During the maturity phase, the best model obtained with the all-subset analysis corresponds to the one having five regressors as independent variables ($F = 389.99$, P-value $= 0.000$). There is no statistical evidence that the terms β_E and β_{PE} are significant. This means that there is no evidence that there is further learning going on after experience level 15. There is some indication that performance varies with experience in the high-level representation/stochastic adaptation system configuration, but it is positive and small ($\beta_{IE} = 0.53\frac{\text{secs.}}{\text{exp.}}$). Table 17 shows the statistical results for each individual parameter in the model for the maturity phase as well as the 90% confidence interval estimation of the real coefficient values.

Table 17: Model Coefficients for the Maturity Phase.

Coefficients	Value	Std. Error	95% C.I.
β_0	85.07	0.355	(84.47,85.66)
β_I	−14.07	1.956	(−15.45,−12.68)
β_P	11.93	0.502	(11.09,12.77)
β_{IE}	0.53	0.082	(0.39,0.66)
β_{IP}	9.27	2.766	(4.64,13.90)
β_{IPE}	−0.54	0.116	(−0.73,−0.34)

The results confirm the hypothesis that the system is in a maturity phase since two terms related with experience are not in the model and the others are statistically significant but with small values. Again, the best system configuration is the one using high-level input representations and stochastic parameter-selection policy ($T = 73$ seconds). Furthermore, there is an interaction effect in the design decisions on the final performance of the system. For example, the average change in performance caused by switching from low-level to high-level input representation when using stochastic parameter-selection policy is $\beta_I = -14.07$ seconds. But if the same switch is performed using the best-value parameter-selection policy, the average change in performance is only $\beta_I + \beta_{IP} = -12.67$ seconds. In both cases, the use of high-level input representations provides better performance after convergence. There is statistical evidence that the best-value adaptation procedure causes the final performance to deteriorate by an average of $\beta_P = 11.93$ when the system uses low-level input representations and by an average of $\beta_P + \beta_{IP} = 21.20$ when the system uses high-level input representations. This confirms the hypothesis that the stochastic parameter-selection policy takes longer to converge but arrives at better solutions.

5.4.2.3 Model Validation

The above results demonstrate the power of our systematic evaluation method: the method enables us to obtain an in-depth understanding of the performance of the system and to make design decisions in a principled manner. However, there are two assumptions that must be verified before accepting the proposed models (and associated results) as a valid: the residuals have zero mean and constant variance, and the residuals have normal distribution. The Least-Squared Error (LSE) technique relies on these assumptions; thus, since the model coefficients were calculated using this technique we must verify if these assumptions hold. Otherwise, any conclusions derived from the model could be wrong and our understanding of how the factors influence the performance of the system could be misleading. In particular, violation of the assumption about the residuals having zero mean and constant variance could introduce inaccuracy in the estimation of the model coefficients, and violations of the assumption of the residuals being normally distributed could produce underestimation of the confidence intervals (i.e., bigger confidence intervals).

A scatter plot of the residuals against the fitted response was used to diagnose changes in variance, and a normal probability plot of the residuals was used to verify the normality distribution of the residuals. The results of each of these two validation techniques are shown in Figures 28 and 29 for the learning and maturity phases, respectively. The left chart in Figure 29 shows constant band of residuals along the horizontal axis. The left chart in Figure 28 shows a systematic diagonal pattern that appears for the lower values of the fitted response. Usually, the presence of this pattern indicates that additional terms may be required, but in this study this is to be expected since the response is not linear (refer to Figure 27). For the purposes of this chapter, we decided to accept this deficiency due to the complications of introducing more terms into the model.

The right charts of Figures 28 and 29 show normal probability plots of the residuals for the learning and maturity phases. In these charts, the values of the residuals are plotted against their expected value as drawn from a normal distribution. If the residuals are indeed normal, then the plots should show a straight line that crosses the origin. In both models, the residuals have zero mean but there is some departure from linearity, especially in the model corresponding to the maturity phase. The reason for this is that the variance in the maturity phase was very small since the system responded similarly during the five replications at each experience level.

Since the two assumptions, residuals with zero mean, constant variance, and normally distributed are not fully satisfied, care must be taken in generalize this

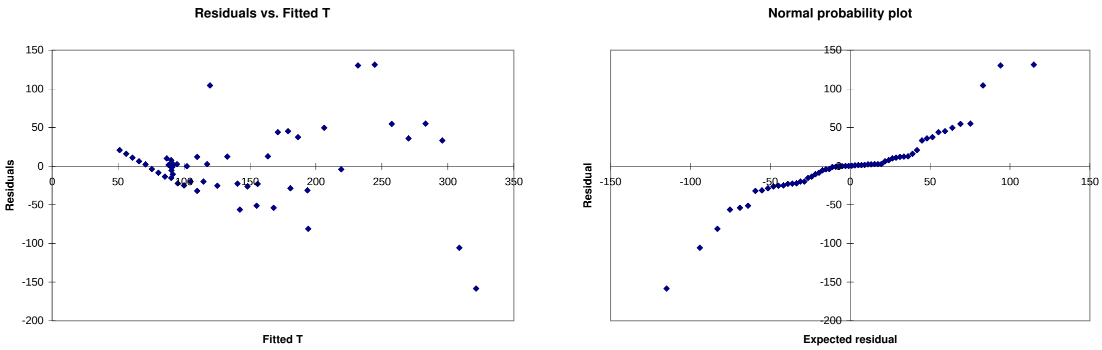

Figure 28: Residual Plots for the Learning Phase.

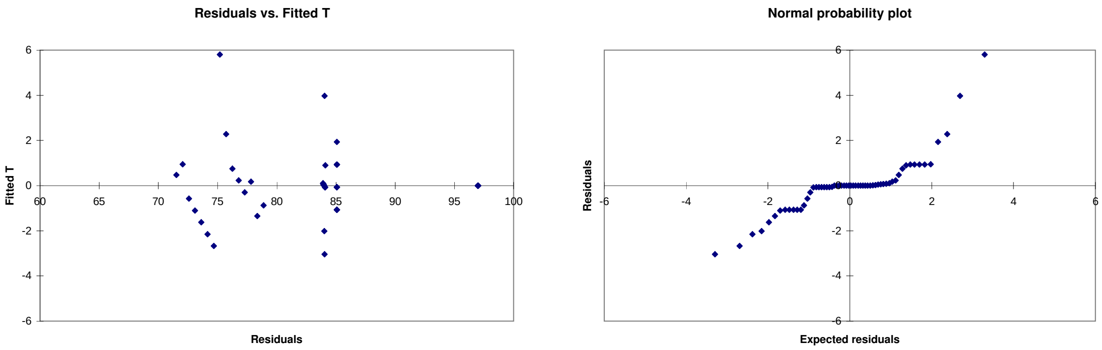

Figure 29: Residual Plots for the Maturity Phase.

result to other 10% cluttered worlds. One option at this point would be to repeat the analysis with additional terms, as discussed above. Another option is to repeat the experiments, this time systematically varying the world clutter; this is discussed next.

5.4.2.4 Robustness Analysis

A follow-up experiment were designed to evaluate the generality of the system by running experiments with a different set of problems. Since the previous experiments indicated that the best design for the system was to use high-level input representations and the stochastic adaptation algorithm, the system was configured in this manner for the new experiment. The data for the experiment was collected in the same manner as the previous experiment, but with the difference that the system solved a fixed randomly-generated 7%-cluttered world in every run.

As in the previous experiment, the model that needs to be determined has the median time (T) as the response variable. But in this case, the model relates the response variable with the amount of experience only since the other factors are constant. If such a model is found to be significant (i.e., the model shows that the amount of experience is related to the response variable), we can conclude that the system learns under the new environmental conditions. Furthermore, the coefficient derived from this model can be compared with the respective coefficient in the previous model. If a significant difference is detected, we can conclude that changing the world clutterness from 10% to 7% affects the learning performance. Table 18 shows the experimental design matrix and Equation 60 shows the complete hypothetical model for the this experiment.

$$T = \alpha_0 - \alpha_E E + \varepsilon \tag{60}$$

This model is a simplification of the model in equation 59 where the experience level (E) is the only regressor. Tables 19 and 20 show the statistical results for each individual parameter in the model as well as the 90% confidence interval estimation of its value for the learning and maturity phases, respectively.

The results show that during the learning phase, the learning rate decreased from $-(\beta_E + \beta_{IE}) = 4.89\frac{\text{secs.}}{\text{exp.}}$ to $-\alpha_E = 0.96\frac{\text{secs.}}{\text{exp.}}$. This means that the system decreases the learning rate when solving a simpler world. Additionally, the value of the intercept reduced from $\beta_0 + \beta I = 124.64$ seconds to $\alpha_0 = 102.97$ seconds. This confirms that that the system is indeed solving a simpler world and most of the learning occurs during the early experiences. During the maturity phase, the term α_E is not statistically significant. This means that the system does not improve the performance further after experience level 15, which confirms the hypothesis that is in the maturity phase.

Table 18: Experimental Design Matrix for the Robustness Analysis.

Choice of input representation	high-level
Choice of parameter-selection policy	stochastic
Experience Levels	1-30
World Clutter	7%
Replicates	5
Response Variable	Time
Total runs	150

Table 19: Model Coefficients for the Learning Phase.

Coefficients	Value	Std. Error	90% C.I.
α_0	102.97	2.76	(97.46,108.48)
α_E	-0.95	0.30	(-1.56,-0.35)

Table 20: Model Coefficients for the Maturity Phase.

Coefficients	Value	Std. Error	90% C.I.
α_0	92.71	0.56	(91.59,93.84)

5.4.2.5 Learning profiles

While the above results demonstrate the validity of the SINS approach, it is also interesting to look at the learning profile of the system, if only to provide an intuitive feel for changes in the internal behaviors and external performance of the system as it gains experience. The learning profile of the system can be determined by comparing the model's predictions with traditional learning curves based on the system's actual performance. This is shown in Figure 30. The graphs labeled "Median" show the actual time taken by the system (starting with no prior experience or hand-coded high-level knowledge) on a randomly generated 10% cluttered world; each point in the graph is the median of five replications with the system configured using the optimal parameters as derived in the previous section (i.e., high-level input representation and stochastic adaptation algorithm). The graphs labeled "Model" show the predicted performance from empirical models. The graph on the left describes the general empirical model along with the error bars for that model. The general model is the performance model derived using the data obtained using the entire range of configurations of the design parameters (Equation 59 and Tables 16 and 17). The graph on the right describes the performance model for the specific configuration used in these runs; this was obtained by redoing the statistical analysis for the data obtained from this configuration alone.

As can be observed from the figure, the performance of the system falls within the general and specific models. The general model has bigger error bars because it must fit the performance of the system over the entire range of configurations. The specific model fits the data better but can only be used to predict the system's performance with this particular system configuration. The results demonstrate that the system does indeed improve its performance with experience, and that this improvement is as predicted by the empirical models derived from the statistical analysis. Furthermore, the performance of the system approaches its optimal level around $E = 20$, which is as expected because the system reached the maturity phase around this experience level. This provides further evidence for the validity of the analysis.

5.4.3 Study 2: Experiments with a Real Robot

While simulation experiments allow us to evaluate the system's performance across a range of design decisions and domain characteristics, it is also important to verify that the system works—and learns—with a real robot. We performed an additional experiment using the actual Denning MRV-III robot to verify the validity of our approach. We used the system configured with the high-level type of input representation and the select-best policy. We ran twenty learning trials and measured the accumulated time and the number of virtual collisions. A trial consisted of placing

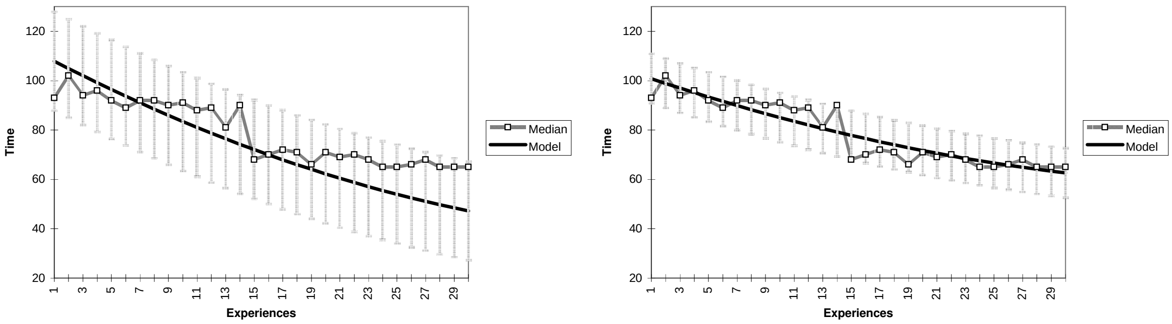

Figure 30: Learning Profiles for SINS.
Performance profiles showing median time from actual system runs and predicted time using two empirical models. On the left is the general model constructed using the data from the entire experiment on the right is the specific model constructed using only the data from the this particular system configuration.

the robot at the starting point and let it run until either it reached the goal or 300 seconds had elapsed, whichever came first. The experiment was performed in an indoor room with an rectangular free space of approximately 25×14 feet and with three circular static obstacles besides the walls. Two of these obstacles were 55-gallon drums with a diameter of 2 feet and the third obstacle was another Denning robot with a diameter of $2\frac{1}{2}$ feet. Figure 31 shows the robot and the laboratory room used in the experiment. Note that the boundaries of the navigable area are not flat; for example, there are desks and chairs against the walls so that the shape is irregular.

Figure 32 shows the performance of the robot plotted against the learning trials. The graphs show the time taken (left) and the number of virtual collisions (right) in each trial. During the first two trials the robot was not able to reach the destination point and both trials were manually terminated after five minutes. During the following trials, the robot was able to complete the task successfully and improve its performance. The robot reduced the total time taken from over 5 minutes to a final value of about 2 minutes after the first 10 trials, after which it did not improve further on this metric. Also, it reduced the number of virtual collisions from about 60 to a steady state of about 10 after the first 15 trials, which produced jerky movements along the path. The SINS system, therefore, showed a significant improvement along both metrics. Figure 33 shows the path taken by the robot in four different trials: 1, 5, 10, and 20.

The next subsection summarizes the results found in the two studies.

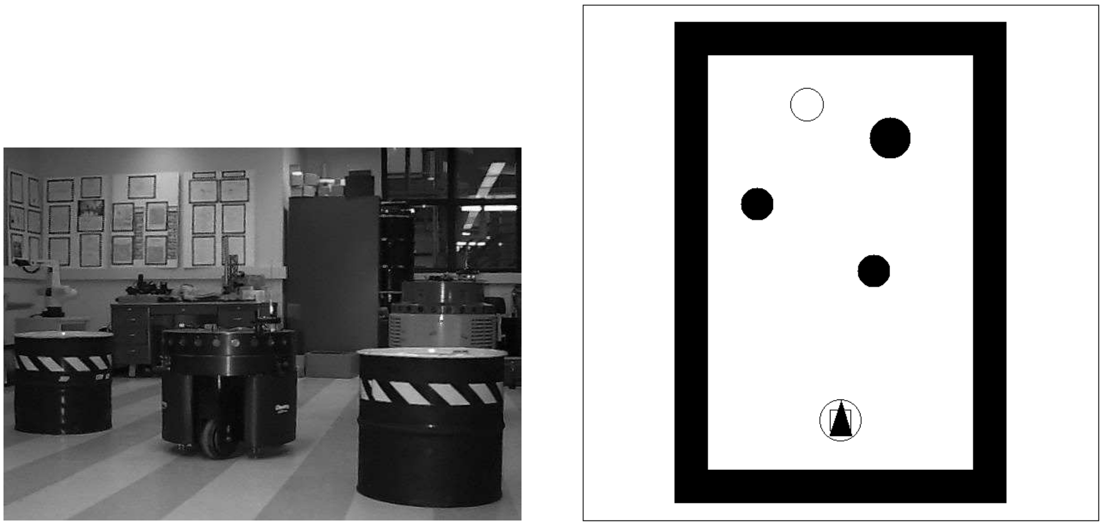

Figure 31: The Robot and its Environment.
Left: real picture. Right: schematic. In the schematic, the robot is shown at the bottom of the figure; the hollow circle near the top represents the goal; the black circles represent the static obstacles; and the black bars represent occupied areas that the robot cannot navigate through.

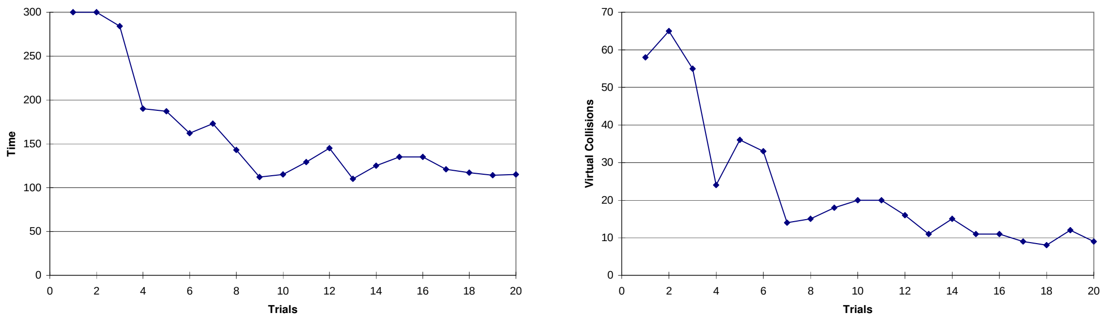

Figure 32: Performance SINS in the Real Robot.
Left: Total time per trial. Right: Number of virtual collisions per trial.

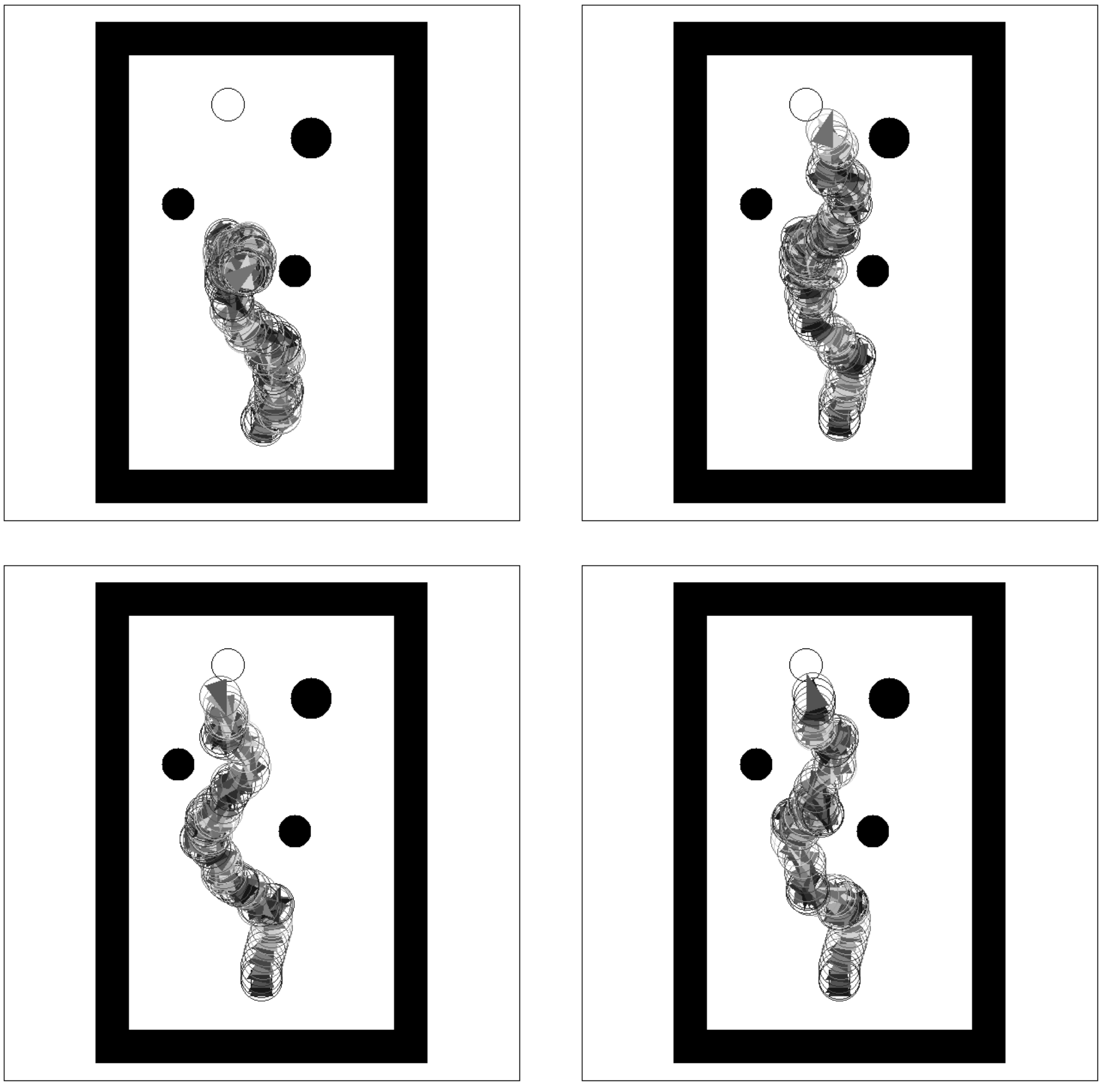

Figure 33: Actual Path Followed by the Robot.
The figure shows four different trials: trials 1 (top left), 5 (top right), 10 (bottom left), and 20 (bottom right).

5.4.4 Discussion

The performance of SINS is very complex and depends not only on simple terms but also on their interactions. Study 1 shows that the system using the stochastic policy method takes a longer time to converge than the best-value policy; however, once converged, the former arrives at better solutions than the latter. There is no interaction between the choice of input representations and the parameter-selection policy. Additionally, the results shown above help us verify and understand several aspects of SINS. In particular:

- The evaluation showed that SINS does improve its performance significantly with experiences (Table 16), thus verifying that the approach proposed in this chapter is effective.
- The regressors (I, A, and E) and their interactions the empirical model can only account for 65.3% and 97.0% of the variability in the performance of the system during the learning and maturity phases, respectively. This result is important because it captures the limitations of the derived model. Part of the remaining percentage could be explained by introducing more factors or by changing the functional forms of the terms in Equation 59; the rest of the variation in performance is due to the randomness in the system.
- The choice of input representation and the choice of parameter-selection policy independently influence the performance of SINS and there are no interactions between these decisions. The model allows us to determine the best choices for these design decisions.
- High-level input representations provide a better starting point for learning than low-level input representations. Also, the stochastic parameter-selection policy takes longer to converge than the best-value parameter-selection policy, but once converged it arrives at better solutions (Tables 16 and 17).
- When configured with high-level input representations and the select-best policy (as suggested by the simulation studies), the real robot does improve its navigation performance on an indoor navigation task with static obstacles along different performance metrics (Figure 32). This provides further verification of the approach proposed here.

5.5 Conclusions

In this chapter, we described a new case-based approach to implement the perception function and the application of the heuristic approach for sequential decision-making to configure closed-loop policies. The two approaches are combined and implemented in the SINS algorithm that performs autonomous mobile robot navigation with imperfect perception. The algorithm is able to learn a mapping between environment situations and configuration parameters and use it to configure a motor schema-based reactive controller on-line in order to improve the performance. Environment situations are represented as a set of cases that represent different regions of the environment state space. The algorithm can learn and use the mapping simultaneously, using its own experience to incrementally refine the mapping so that the performance of the system improves.

The theory supporting the algorithm consists of capturing regularities in the interaction of the agent and the environment and using them during future similar situations. Regularities are casual patterns that are consistent over time; they help to improve system performance because of the predictive power obtained through this consistency. Thus, an agent is able to perceive the environment state indirectly in terms of these regularities and detect affordances for improving its performance. Finally, the chapter presented results from systematic empirical evaluations of an implemented system that verify the validity of the approach.

Sequential decision-making problems in which the agent has imperfect perception are very difficult. When sensations do not reveal the state of the environment the agent cannot characterize the situation it is facing precisely and it is not able to assess the long-term benefits of its actions. Generally, the design of perception functions make use of recursive estimation procedure that require a priori models of the environment (for example, Kalman filters do not only require a model of the environment but also that such model is linear; Sorenson, 1966). In this chapter, we presented and extensively evaluated a new case-based perception function that does not require a priori models of the environment. The case-based method characterizes different situations as sequences of sensation-action pairs that occur regularly over time. The agent also uses the cases created by the perception function to represent the action-value function mapping long-term benefits to situation-action pairs. This result ties in function approximators and imperfect perception. The next chapter studies the issue of incomplete knowledge in isolation and Chapter 7 describes the implementation of the heuristic method in a complex problem that integrates continuous domains, imperfect perception, and incomplete knowledge.

CHAPTER VI

INCOMPLETE KNOWLEDGE

In Chapters 4 and 5 we focused on the issues of continuous domains and imperfect perception, respectively. In these chapters, we studied problems in which the internal state of the agent does not have any quantity representing the knowledge level of the agent. In this chapter we focus on the issue of incomplete and dynamic knowledge. In such sequential decision-making problems, the agent possesses an internal state that represents its knowledge level. During the execution of the task, the agent may be able to collect new information and change its knowledge level by executing actions and observing their result. In this context, the agent faces the dilemma of choosing actions to make progress on the task or to gain new information. In theory, the optimal solution requires an internal state that grows up to infinite dimensions[1] (i.e., the information state). In practice, the heuristic solution described in Chapter 3 requires an internal state with finite dimensional quantities representing the current knowledge level and an estimate of the state of the environment. In this chapter we formulate two different instantiations of the heuristic solution and describe their implementation in detail. We studied these implementations on the double integrator problem and with agents having perfect perception and case-based function approximators to represent the value function.

Typically, autonomous agents are designed so that they select actions based on their current knowledge level. However, their decision-making mechanism does not explicitly consider that their knowledge level may change as they perform the task. That is, an agent uses its current level of knowledge to select and execute actions that are efficient from the perspective of such imperfect knowledge. Often, some type of random exploration (e.g., Boltzmann exploration; Watkins, 1989) or exploration bonuses (e.g., Sutton, 1990) bypasses the standard action-selection mechanism so that the agent is able to learn the effect of "unseen" actions and incorporate new information about the environment. The agent learns to perform better each time because new incoming information is used to improve the knowledge level, which leads to better decisions. However, this is a *passive* form of learning because the

[1] Refer to Section 3.1.

agent does not explicitly select actions to gain new information nor consider how each action will influence its level of knowledge in the short- or the long-term. This contrast the more efficient and robust form of *active learning* in which the agent selects actions taking into account both their efficiency with respect to the task and the influence on the current level of knowledge their execution may produce. In this formulation, the agent's goals include both performance goals and learning goals and the agent's actions are chosen to address both types of goals (Ram and Leake, 1995).

The problem of sequential decision making under incomplete knowledge has been studied by researchers in the field of optimal control (Bertsekas, 1995a; Fel'dbaum, 1965; Bar-Shalom, 1990; Stengel, 1994). When the agent has incomplete knowledge, actions are necessary not only for making progress on the task, but also for studying the environment. Actions must have a "dual" character: They must be directors to a known degree, but also investigators to a known degree. The concept of *dual control* was first introduced by Fel'dbaum (1965) to denote the control of actions that bear this dual character. In dual control a conflict arises between the two aspects of the control action mentioned above. In fact, the agent can control successfully only when the properties of the environment are known accurately enough. Meanwhile the study of the environment requires time. Too "hurried" an agent will perform unjustified control actions, which will not by substantiated properly by the information obtained as a result of the study of the environment. Too "cautious" an agent will wait for an unnecessarily long time, accumulating information, and may not be able to accomplish the task to the required specifications. Thus, an agent must select actions in such a way that it can achieve a balance in the dual character of its actions. In other words, the agent has to devote some effort to *exploring* the environment and gather new knowledge, but only to the extend it can start *exploiting* such knowledge to proficiently perform the task. In Chapter 3, we describe the optimal, but practically unfeasible solution to the generic sequential decision-making problem and present a new heuristic approach inspired by the optimal solution method that can be implemented in agents having limited computational and storage resources. An agent using the heuristic solution is able to select actions that compromises between the progress on the task and the knowledge gain. In this chapter, we instantiate our heuristic solution to solve the double integrator problem and show that an agent using the heuristic solution performs better than agents using the best fixed and passive learning policies due to its ability to balance exploration and exploitation.

Section 6.1 describes a model-based approach derived from the heuristic solution and two different implementations of the algorithm. We describe the model-free approach and the results we obtained in the double integrator in a separate paper (Santamaría and Ram, 1997c). The two implementations are applicable when the agent has different degrees of uncertainty about the unknown properties of the environment: the multiple hypotheses and the unknown parameters scenarios. In the former, the agent knows a priori that the unknown properties can have one out of N possible values. In the latter, the agent knows which properties are unknown but it does not know their possible values. In both scenarios, the agent must select and perform actions that reveal information about the unknown properties while making progress on the task. Section 6.2 presents the results of fixed (non-learning), passive-learning, and active-learning agents on the double integrator. Section 6.3 discusses the results and Section 6.4 concludes the chapter.

6.1 Model-Based Dual Control Algorithms

In this section we present two different implementations of the heuristic solution that are applicable when the agent has different types of uncertainty about the properties of the environment: the multiple hypotheses and the unknown parameters scenarios. The basic difference between the implementations is in the representation of the knowledge level in agent's internal state. In the multiple hypotheses scenario the knowledge level consists of a set of probabilities, each denoting the likelihood of occurrence of each hypothesis. In the unknown parameters scenario the knowledge level consists of an estimate and variance of the unknown properties of the environment. Each scenario demands different perception functions: the multiple hypotheses scenario uses a Bayes estimation rule while the unknown parameters scenario uses a Kalman filter.

Figure 34 shows the general model-based implementation of the heuristic solution. Basically, the agent uses a model of the dynamics of environment to perform both action selection and value function adaptation. During action selection, the agent selects the best action to execute next in two steps. First, the agent computes a set of all possible (i.e., hypothetical) internal states and immediate rewards that may result from the execution of each action. The agent performs this computation by simulating the next action-perception cycle with the model for every given action. Second, the agent estimates the long-term benefit of each action by summing the immediate reward and value of the next internal state as suggested by the value function approximator. The agent selects and executes the action leading to the best long-term benefit. During value function adaptation, the agent improves the value function using the value iteration method we described in Chapter 3.

The following subsections describe in detail the characteristics of the representation of the internal state (Equation 17) and the perception function (Equation 18) for each of the two scenarios. Section 6.2 presents the results of the each scenario in the double integrator problem.

1. **Initially**:
   Set the initial estimate of the value function: weights w_0.
   Construct the initial internal state: $s_0 = \{\, \hat{x}_0, \kappa_0 \,\}$ based on the initial sensation z_0.

2. **Perform**:
   At each stage $t = 0, 1, \ldots$

   (a) **Decision**:
   Select and execute the best action at stage t, $\hat{u}_t^*$

   $$\hat{u}_t^* = \arg\max_{u_t \in U} E_{\hat{z}_{t+1}} \left[\hat{R}(\hat{x}_t, u_t) + \gamma \hat{V}_{w_t}(\hat{s}_{t+1}) \right]$$

   where $\hat{z}_{t+1}$ is the predicted (i.e., hypothetical) sensation:

   $$\hat{z}_{t+1} = H(\hat{F}(\hat{x}_t, u_t))$$

   and $\hat{s}_{t+1}$ is the predicted (i.e., hypothetical) internal state:

   $$\hat{s}_{t+1} = \Psi(s_t, u_t, \hat{z}_{t+1})$$

   (b) **Perception**:
   Observe the resulting reward and sensation:

   $$\begin{aligned} r_{t+1} &= R(x_t, \hat{u}_t^*) \\ z_{t+1} &= H(x_{t+1}) \end{aligned}$$

   Compute the new internal state:

   $$s_{t+1} = \Psi(s_t, u_t, z_{t+1})$$

   (c) **Adaptation**: (at the end of every trial)
   Adapt the action-value function approximator using value iteration:

   i. Compute the value for each state $s_k^i \in S$:

   $$\hat{V}_{k+1}^i = \max_{u_k \in U} E_{\hat{z}_{k+1}} \left[\hat{R}(\hat{x}_k, u_k) + \gamma \hat{V}_{w_t}(\hat{s}_{k+1}) \right]$$

   ii. Compute a new set of weights w_{t+1} by minimizing:

   $$w_{t+1} = \arg\min_{w_{k+1}} \sum_i \left(\hat{V}_{k+1}^i - \hat{V}_{w_{k+1}}(s_k^i) \right)^2$$

Figure 34: Model-based Heuristic Solution Algorithm.

6.1.1 Multiple Hypotheses Scenario

In this scenario the agent knows a priori that the unknown properties of the environment can have one out of N possible values. The agent can directly observe the complete state of the environment but cannot observe the actual value of the unknown properties; however, the agent is able to indirectly determine these values by applying specific actions at specific states. The ultimate goal of the agent is to maximize the performance index. Thus, it must select a sequence of actions that best reveal the values of the unknown properties while maximizing the sum of rewards. The internal state consists of the state of the environment, which the agent can observe perfectly, and the knowledge level, which is a set of N probability variables representing the belief or *hypothesis* that the environment dynamics is each of the N possible ones:

$$s_t = \{\, x_t, p_t^1, \ldots, p_t^N \,\} \tag{61}$$

In this scenario the state of the environment evolves according to one out of N possible dynamics functions, $F_i(\cdot)$ $i = 1, \ldots, N$ (i.e., one dynamics function for each hypothesis). The sensor function reveals the state of the environment but not the identity of the actual dynamics function; hence, the hypothesis is unknown to the agent:

$$z_t = H(x_t) = x_t \tag{62}$$

The internal state of the agent consists of the state of the environment and the set of probabilities of each hypothesis,

The perception function updates the internal state using the content of the previous state, the previous action, and the current sensation. The function updates the state of the environment directly using information delivered by the sensor function and the probabilities of each hypothesis using Bayes' rule. Thus,

$$\begin{aligned} s_t &= \{\; x_t, p_t^1, \ldots, p_t^N \,\} \\ &= \{\; \Psi_x(x_{t-1}, p_{t-1}^1, \ldots, p_{t-1}^N, u_{t-1}, z_t), \\ &\qquad \Psi_1(x_{t-1}, p_{t-1}^1, \ldots, p_{t-1}^N, u_{t-1}, z_t), \\ &\qquad \vdots \\ &\qquad \Psi_N(x_{t-1}, p_{t-1}^1, \ldots, p_{t-1}^N, u_{t-1}, z_t) \;\; \} \end{aligned} \tag{63}$$

where the state perception function merely returns the observed sensation

$$\begin{aligned} x_t &= \Psi_x(x_{t-1}, p_{t-1}^1, \ldots, p_{t-1}^N, u_{t-1}, z_t) \\ &= z_t \end{aligned} \tag{64}$$

and the hypotheses perception functions return the a posteriori probabilities of each hypothesis given the current sensation:

$$\begin{aligned} p_t^i &= \Psi_i(x_{t-1}, p_{t-1}^1, \ldots, p_{t-1}^N, u_{t-1}, z_t) \\ &= p_{t-1}^i \frac{P(\, z_t = F_i(x_{t-1}, u_{t-1})\,)}{\sum_k P(\, z_t = F_k(x_{t-1}, u_{t-1})\,) p_{t-1}^k} \end{aligned} \tag{65}$$

The quantity $P(\, z_t = F_k(x_{t-1}, u_{t-1})\,)$ represents the conditional probability of the agent receiving sensation z_t when the state of the environment is x_{t-1}, the action is u_{t-1}, and hypothesis k is assumed to be true. Bayes' rule updates each probability by generating the a posteriori probabilities p_t^i based on the a priori probabilities p_{t-1}^i and the conditional probabilities $P(z_t \mid F_k)$ $k = 1, \ldots, N$ (see, e.g., Ross, 1993).

The adaptation procedure works as describe in the model-based dual control algorithm of Figure 34. Namely, the agent keeps a function approximator for the value function $\hat{V}_w(\cdot)$ and uses it to select the most promising action at stage t. Then, the agent uses the outcome of every action to update the internal state and adapts the estimates of the value function.

6.1.2 Unknown Parameters Scenario

In this scenario the agent knows which are the unknown properties of the environment but does not know their possible values. Unlike in the previous scenario, the agent does not assume that the value of the unknown properties may be one out of a given set; however, it assumes the value, although unknown, is constant (or slowly-varying in time). Therefore, the uncertainty of the agent is greater, the perception function is more complex, and the dual control is more difficult than in the multiple hypotheses scenario. Additionally, the agent is not able to perceive the complete state of the environment without noise;[2] thus, it must use an estimate of the state and keep it up to date using a recursive estimation procedure such as a *Kalman filter*.[3] The internal state consists of an estimate of the state of the environment, an estimate of the unknown properties of the environment or *external parameters*, and the covariance matrix of the state and external parameters.

The dynamics function of the environment consists of a function that depends on the external parameter vector $p \in \Re^k$, $x_{t+1} = F(x_t, u_t, p)$. The sensor function reveals the state of the environment but not the value of the external parameter

[2] It is assumed independent additive Gaussian noise with zero mean and constant covariance matrix.

[3] Kalman filters are commonly used in control systems to estimate the state of the dynamic system when the incoming sensations are noisy (Ogata, 1990; Young, 1984; Stengel, 1994; Bertsekas, 1995a).

vector. Additionally, the sensations are contaminated with additive Gaussian noise with zero mean and constant covariance matrix R.[4]

$$z_t = H(x_t) = x_t + \epsilon_t \qquad\qquad \epsilon_t \sim \text{i.i.d.}^5\ N(0, R) \tag{66}$$

The internal state of the agent consists of an estimate of the state of the environment $\hat{x}$, an estimate of the external parameters vector $\hat{p}$, and their combined covariance matrix M:

$$s_t = \{\, \hat{x}_t, \hat{p}_t, M_t \,\} \tag{67}$$

The perception function updates the internal state using a Kalman filter. This process can be outlined in five steps (see, e.g., Stengel, 1994, Section 4.3). Figure 35 shows the algorithm.

The rationale of the Kalman filter is as follows: The first two steps compute the predicted state and covariance matrix of the quantity $\{\, \hat{x}_t, \hat{p} \,\}$. Step 3 uses this result to construct the filter gain K, which will resolve the mismatch between the predicted state estimate $\hat{x}_t^-$ and the incoming sensation z_t. Step 4 updates the state estimate based on the error between the outcome and the prediction $(z_t - \hat{x}_t^-)$. Step 5 does the same for the covariance matrix.

6.2 Results

In this section we describe and contrast the results of fixed non-learning, passive-learning, and active-learning agents acting on the double integrator problem (Figure 12). For the active-learning agent, we implement the heuristic solution using the two scenarios: multiple hypothesis and unknown parameters. We used the instance-based function approximator and the value iteration adaptation method in the experiments. A description of the instance-based function approximator is given in Chapter 4. The instance-based function approximator uses cases to represent the value function. Each case represents a point in the internal state space and holds its associated value. In these experiments, we programmed the function approximator to compute the value of query points as the multilinear interpolation of the two closest neighbors along every variable. We used this type of interpolation because it supports efficient value computation and adaptation during the action selection and the value iteration method. We initialized the value function approximator with the set of quantized points used by the value iteration method. The multiple hypotheses and unknown parameter scenarios used different configurations of the value function

[4]This assumption is required for the successful operation of the Kalman filter.

[5]Independent-identically-distributed probability density functions. In this case, Normal distributions with zero mean and constant variance.

1. State estimate prediction: $\{\hat{x}_t^-, \hat{p}_t^-\}$

$$\begin{aligned} \hat{x}_t^- &= F(\hat{x}_{t-1}, \hat{p}_{t-1}, u_{t-1}) \\ \hat{p}_t^- &= \hat{p}_{t-1} \end{aligned}$$

2. Covariance matrix prediction: M_t^-

$$M_t^- = A M_t A^T$$

where $A = \begin{bmatrix} \frac{\partial F}{\partial x} & \frac{\partial F}{\partial p} \\ 0 & I \end{bmatrix}$. The partial derivatives of $F(x, p, u)$ are evaluated at $(\hat{x}_{t-1}, \hat{p}_{t-1}, u_{t-1})$.

3. Filter gain computation: K (K is the Kalman gain)

$$K = M_t^- \left[M_t^- + R\right]^{-1}$$

4. State estimate update: $\hat{x}_t$

$$\begin{bmatrix} \hat{x}_t \\ \hat{p}_t \end{bmatrix} = \begin{bmatrix} \hat{x}_t^- \\ \hat{p}_t^- \end{bmatrix} + K\left[z_t - x_t\right]$$

5. Covariance matrix update: M_t

$$M_t = \left[{M_t^-}^{-1} + R^{-1}\right]^{-1}$$

Figure 35: Model-based Perception Function.

approximator. The next two subsections describe the particular configuration and the results we obtained in each scenario.

In these experiments the mass of the car can be either light ($M_1 = 1$) or heavy ($M_2 = 2$) and it can change from trial to trial with 50% probability. The simulation used a time step of $\Delta t = 0.05$ seconds and new control actions were selected every four time steps. A trial consists of starting the system at position $p = 1$ with velocity $v = 0$ and running the system until either 200 decision steps had elapsed (i.e. 40 simulated seconds) or the state got out of bounds (i.e., when $|p| > 1$ or $|v| > 1$), whichever came first. In the latter case, the agent received a negative reward (i.e., a punishment) of 50 units to discourage it from going out of bounds. An experiment consisted of 1 replication of 50 trials, measuring the cumulative cost (i.e., negative sum of rewards) as the measure of performance. We used only one replication because all the policies except one were deterministic and additional replications would produce the same result. The random exploration agent is the only agent using a stochastic policy. For that particular agent we chose the best performance across 36 replications.

6.2.1 Fixed Non-Learning Policy

In this subsection we describe the results of an experiment in which the agent uses a fixed policy to solve the double integrator problem. The double integrator is an instance of a linear dynamical system and possesses a closed-form solution when the mass of the car is completely known a priori. Section 4.4.1 presents a complete mathematical definition of the double integrator and its closed-form solution. The optimal policy of the agent when it knows the mass of the car is as follows,

$$u_t^* = \pi(p_t, v_t) = -(p_t + \sqrt{2M} v_t) \tag{68}$$

where p, v, and M are the current position, velocity, and mass of the car, respectively.

The optimal control for the light car is different from the optimal control of the heavy car. The light car responds quickly to moderate forces, therefore the agent will not incur large costs by applying the top acceleration at the beginning of the task. On the other hand, the heavy car responds slowly to moderate forces, therefore the agent will not incur into large costs for not moving the car fast enough at the beginning of the task. Even though cars with different mass require different optimal policies, it is possible to use, for example, the optimal policy of the light car to drive the heavy car or vice versa. The resulting trajectory is not optimal but the policy is able to accomplish the task. There are two possible configurations for an agent using a fixed non-learning policy: always use the optimal policy for driving the light car (optimal-light policy) or always use the optimal policy for driving the heavy car (optimal-heavy policy). Neither of these policies is able to incorporate information the agent gathers during the performance of the task. Figures 36 and 37 show the trajectory in state space of both cars under the control of both fixed policies.

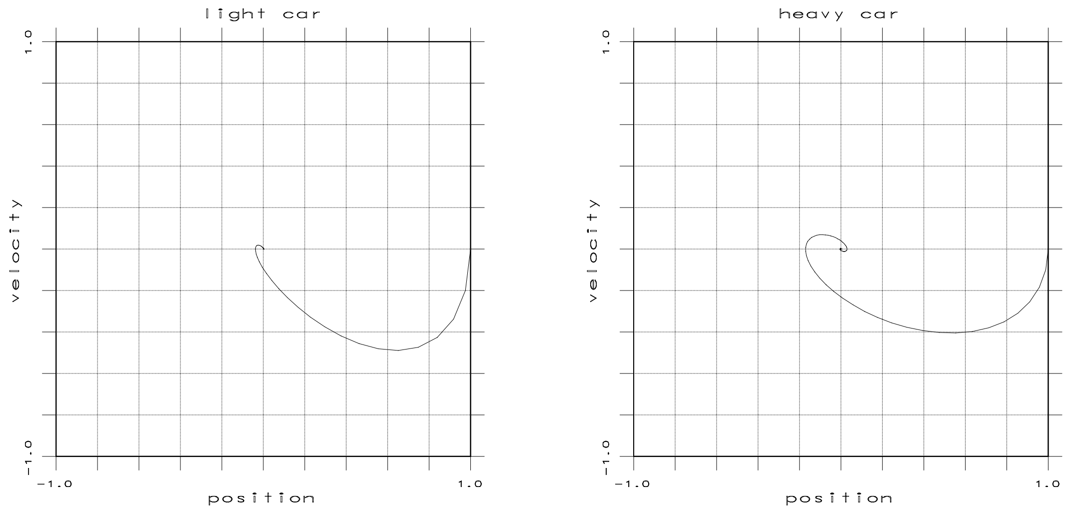

Figure 36: Trajectories for the Optimal-Light Policy.
Left: light car. Right: heavy car.

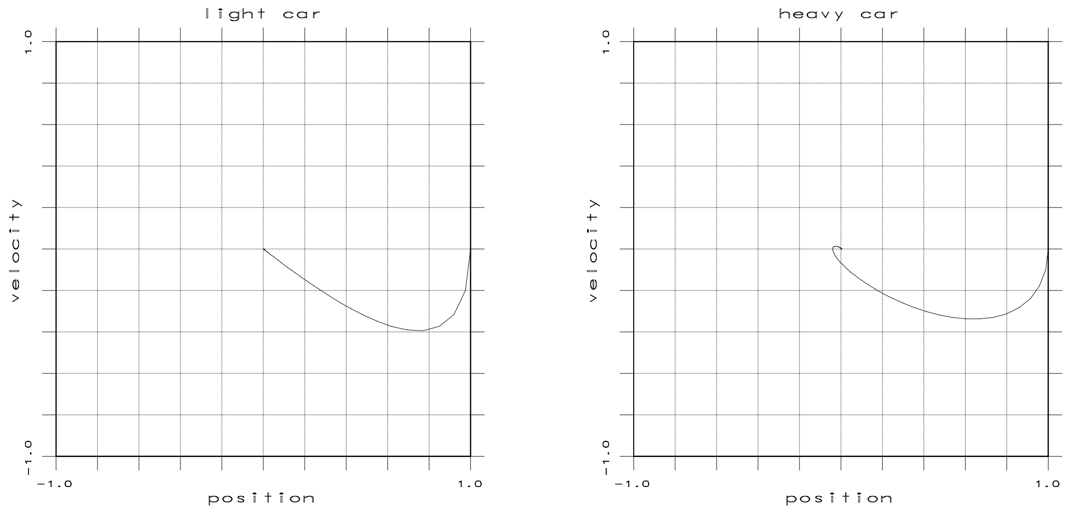

Figure 37: Trajectories for the Optimal-Heavy Policy.
Left: light car. Right: heavy car.

The agent using the optimal-light policy performs optimally when driving the light car but overshoots the target point when driving the heavy car. This occurs because the inertia of the heavy car is greater than the one of the light car and the agent cannot anticipate this fact. Conversely, the agent using the optimal-heavy policy performs optimally when driving the heavy car but undershoots the target point when driving the light car. This occurs because the agent expects the car to be less maneuverable than it actually is and acts too conservatively. A better policy would use the information collected during the performance of the task to determine whether the car is light or heavy and then use the appropriate policy thereafter. The next subsection describes the results of an agent that uses a passive-learning policy that implements this idea.

6.2.2 Random Exploration Learning Policy

In the previous subsection we showed the result of an agent using a fixed policy to solve the double integrator. The agent always acts as if it were driving the light (or heavy) car during the entire trial. However, after taking several steps the agent may use the information it has collected to infer what type of car it is driving and use the appropriate policy. In this subsection we describe the results of an experiment in which the agent uses a passive-learning policy to solve the double integrator.

In this experiment the internal state of the agent consists of the position and velocity of the car as well as the probability of belief the agent is driving the light car (i.e., hypothesis 1)[6]. The passive-learning policy uses these three variables to decide what action to execute next. The policy may operate in one of two modes: exploration or exploitation. The policy operates the exploration mode when the knowledge level of the agent is low and additional information is required to refine the knowledge. Once the agent knows which car it is driving within a given amount of certainty then the policy operates in exploitation mode. In this mode the policy outputs actions using either the optimal policy of the light or heavy car according to the value of the belief the agent. The agent uses the hypotheses perception function (Equation 65) to update the belief probability. The passive-learning policy chooses the optimal action (u_L^*) for driving the light car when the agent believes it is driving the light car with probability 75% or higher. Conversely, the policy chooses the optimal action (u_H^*) for driving the heavy car when the agent beliefs it driving the light car with probability 25% or lower. When the belief of the agent is between 25% and 75% there is not enough information to determine what car the agent is driving and the action should have a random component to force exploration. Thus, the action is the sum of two parts: (1) a weighted average of the optimal actions for driving the light and heavy car based on the current belief and (2) a random

[6]The probability of belief the agent is driving the heavy car (i.e., hypothesis 2) is the complement to hypothesis 1; therefore, the agent does not need to explicitly represent it.

magnitude for exploration. The purpose of the first part is to allow the agent to keep making progress on the task. The second part is to help the agent to collect more information and improve its current belief. The passive-learning policy is as follows,

$$\pi(p,v,q) = \begin{cases} u_L^* = -(p+\sqrt{2M_L v}) & \text{if } q \geq 0.75 \\ u_L^* * q + u_H^*(1-q) + \eta & \text{if } 0.25 < q < 0.75 \\ u_H^* = -(p+\sqrt{2M_H v}) & \text{if } q \leq 0.25 \end{cases} \tag{69}$$

where p, v, and q represent the position, velocity, and probability of belief of the agent respectively. The constants $M_L = 1$ and $M_H = 2$ represent the mass of the light and heavy car respectively and $\eta \in U[-0.1, 0.1]$ is a random quantity uniformly distributed between -0.1 and 0.1. Figures 38 and 39 show the trajectory in state space and the agent's belief of hypothesis 1 as a function of the stage in both cars.

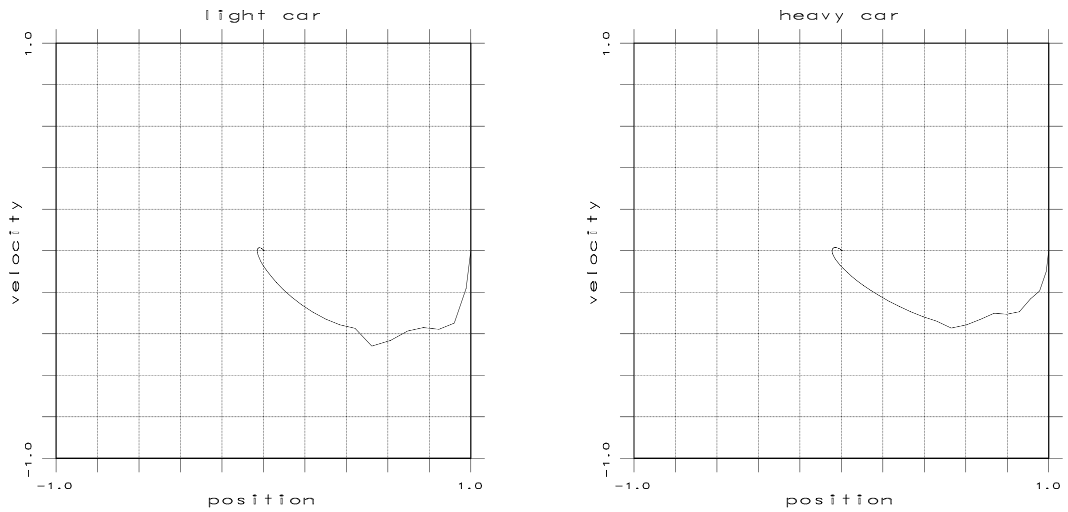

Figure 38: Trajectory of the Car in State Space.
Left: light car. Right: heavy car.

The agent using the passive-learning can take advantage of the information gathered during the performance of the task. However, the policy is fixed at design time according to the intuitions and prior knowledge of the programmer. In this experiment, the policy operates in only one of two modes at a given time: exploration or exploitation. The procedure for selecting the modes is simple: use exploration until there is enough information. Thus, even though in exploration mode the agent's use of random actions may help improve the current knowledge level, such learning is *accidental.* The agent does not select actions explicitly considering both the next state of the car and the next knowledge level their execution may produce. A better agent

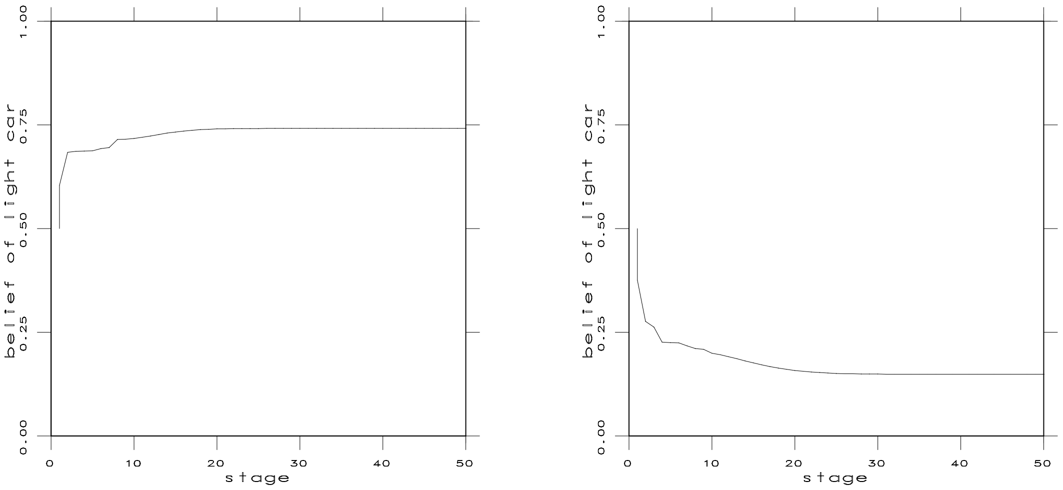

Figure 39: Belief of Hypothesis 1.
Left: light car. Right: heavy car.

design would use a policy that always balances the control energy to explore and exploit (i.e., dual control). The next section describes the results of an agent that uses the multiple hypotheses implementation of the heuristic solution to accomplish dual control in the double integrator.

6.2.3 Multiple Hypotheses Learning Policy

In the previous subsection we presented the results of an agent using a fixed learning policy. The learning agent performs better than the non-learning agent because the former uses the information gathered during the trial to improve its decision-making process. However, the agent selects information-gathering actions without considering the long-term benefit of the resulting knowledge level of the agent. In this subsection we describe the results of an experiment in which the agent uses the multiple hypotheses implementation of the heuristic solution to select actions considering both the progress in the task and the change in knowledge level of the agent.

In this experiment the agent knows that mass of the car may be light ($M_L = 1$) or heavy (M_H=2) but it does not know which one until it starts applying forces and observing the outcomes (i.e., a two-hypotheses scenario). The agent may decide to apply a strong force all the way until the car is near the goal. However, at some point it must decide to decelerate so that the car comes to a complete stop near the goal. When the agent is controlling the heavy car, it should start decelerating earlier than when it is controlling the light car. For that purpose, the agent must try to identify which car it is controlling, but to be successful it must do so as it approaches the goal. The entire sequence of actions should be such that the sum of all the rewards is maximized. Thus, an agent that spends too much time applying different forces to find out the mass of the car may be heavily penalized because it is not taking the car to the goal. On the other hand, an agent that hurries in taking the car towards the goal will not know the mass of the car with enough confidence level to stop in time and may be heavily penalized for taking too many correcting actions. The results shown below corresponds to a learning adaptive agent that uses the multiple hypotheses implementation policy to solve the double integrator problem using dual control. The internal state of the agent consists of three variables: position, velocity, and belief of the light car. The perception function updates the position and velocity directly from every sensation. Additionally, the function uses Bayes' rule to update the belief as described in Section 6.1.1. The function approximator is the instance-based and the distance function is multilinear. The adaptation method is value iteration using a perfect model of the environment and it is invoked after the conclusion of every trial. The agent is trained with 50 trials. During each trial the agent attempts to drive a car of unknown mass. Table 21 shows the design of the agent.

Figure 40 shows the performance of the agent during the 50 trials of training. In each trial the agent controls either a light or a heavy car with 50% probability for 200 stages. The figure shows the actual realization (i.e., sequence of light and heavy cars) that was used during training and separates the performance by type of car to help compare the performance within each car. The agent starts without knowing how to select actions. During the first trial the agent is not able to accomplish the task. The first estimate of the value function produces a policy that only seeks for immediate reward but keeps the position and velocity of the car within bounds. After subsequent trials, the improved value function estimates produce policies that take the car to the goal with better performance. The agent learns to improve performance regardless of the type of car it controls and the experience collected during earlier trials transfers to later trials. For example, Figure 40 shows that during trials 2, 3, and 4 the agent drove the heavy car and the experience the agent collected during these trials helped it to perform better during trial 5 even though the agent was driving the light car.

Figure 41 and 42 show the trajectory in state space and the agent's belief of hypothesis 1 as a function of the stage in both cars at trial 50.

The agent using the dual control policy can take maximum advantage of the information gathered during the performance of the task. After every trial, the agent adapts the value function such that it can represent the long-term benefit of internal states more accurately. Better long-term estimates lead to better decisions during future trials. The agent also takes advantage of the models of the environment

Table 21: Design of the Multiple Hypotheses Agent for the Double Integrator.

Factor	Description
Variables	3: position: $p \in [-1\ 1]$ velocity: $v \in [-1\ 1]$ belief: $q \in [0\ 1]$;
Cases	Uniformly distributed in a grid: $21 \times 21 \times 11$ 21 points along p. 21 points along q. 11 points along q.
Decision	One-step Search: For every action u_i from $u_{\min}$ to $u_{\max}$ at step Δ_u: $\hat{u}_t^* = \arg\max_{u_i \in U} E_{\hat{z}_{t+1}} \left[\hat{R}(\hat{x}_t, u_t) + \gamma \hat{V}_{w_t}(\hat{s}_{t+1})\right]$ $\Delta_u = \frac{a_{\max} - a_{\min}}{25} = 0.08$
Sensation	Direct observation of position and velocity: $z_t = H(x_t) = \{\, p_t, v_t \,\}$
Perception	multiple hypotheses perception (Equation 65): p_t and v_t as collected from z_t. $q_t = q_{t-1} \frac{P(\, z_t = F_i(p_{t-1}, v_{t-1}, u_{t-1})\,)}{\sum_k P(\, z_t = F_k(x_{t-1}, v_{t-1}, u_{t-1})\,) q_{t-1}}$
Adaptation	Value iteration algorithm (Figure 34): Performed after every trial. Environment dynamics $\hat{F}(\cdot)$ and reward function $\hat{R}(\cdot)$ as described in Section 4.4.1.

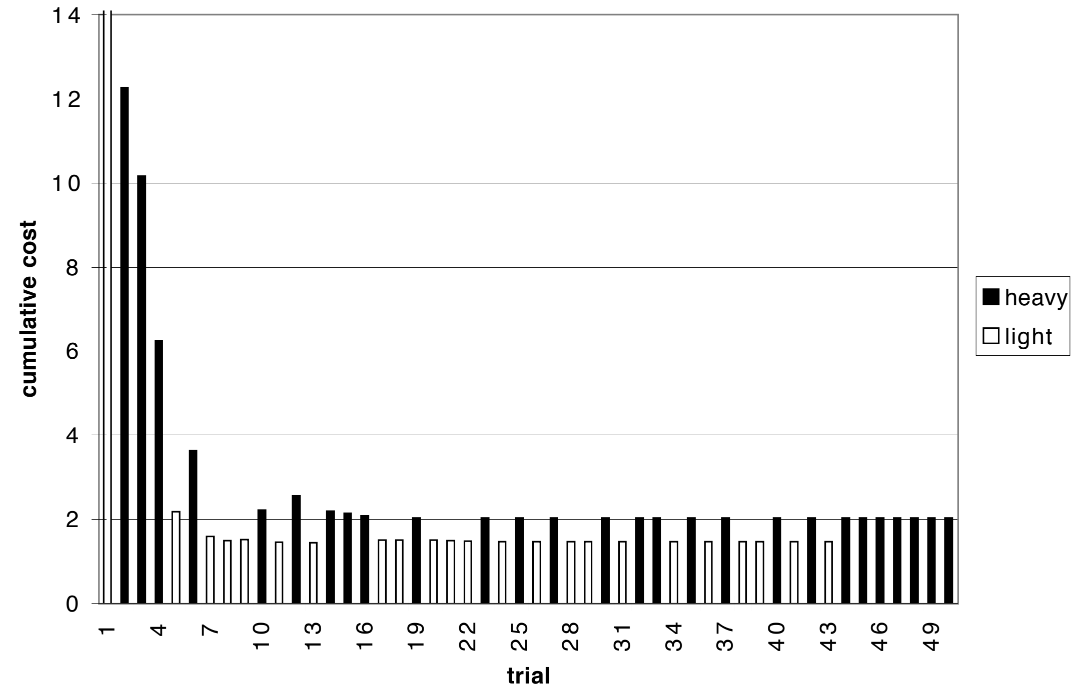

Figure 40: Performance of the Multiple Hypotheses Policy in the Double Integrator. The graph shows the cumulative cost per trial of the training sequence of 50 trials. The white and black bars correspond to the light and heavy cars respectively.

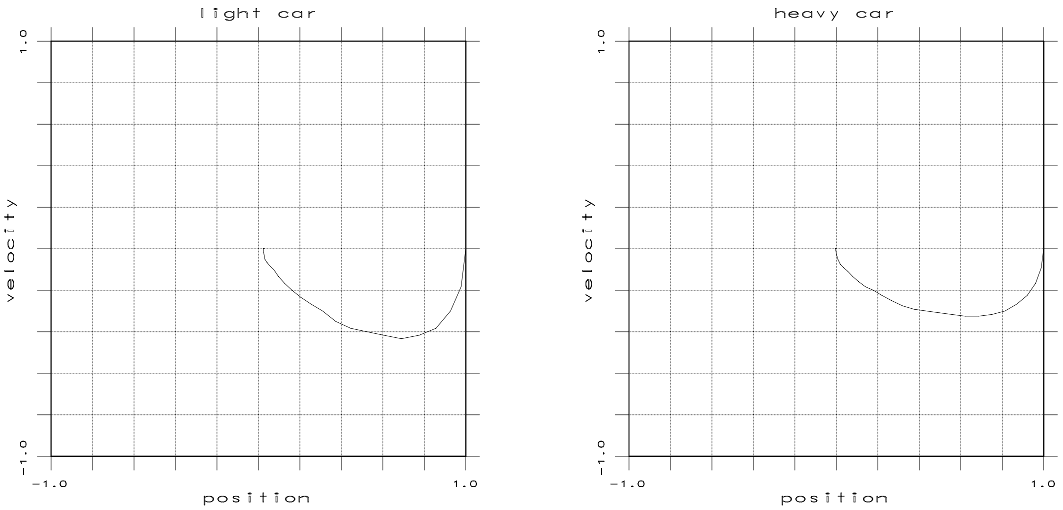

Figure 41: Trajectory of the Car in State Space.
Left: light car. Right: heavy car.

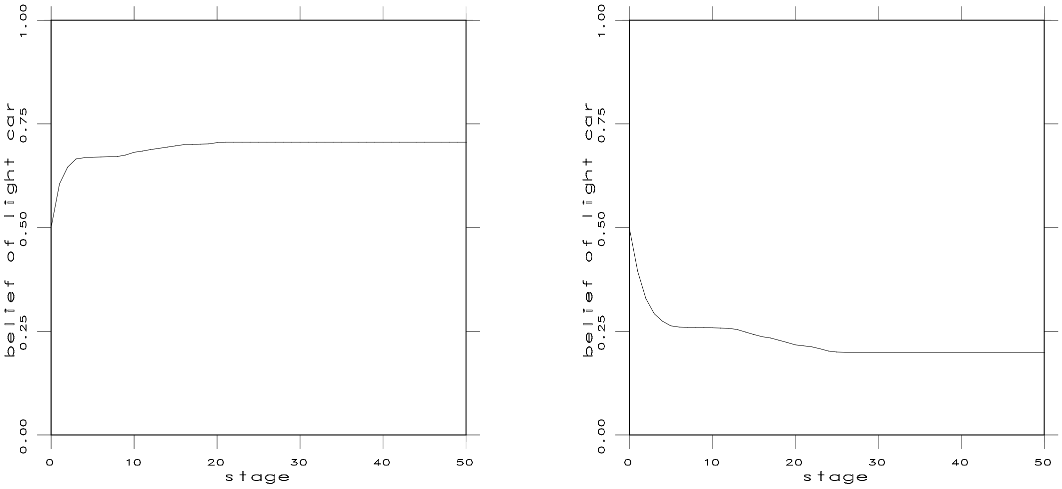

Figure 42: Belief of Hypothesis 1.
Left: light car. Right: heavy car.

and task to decide the best action to take at any stage (decision step in Figure 34) and to adapt the value function using value iteration (adaptation step in Figure 34). Additionally, the dual control policy always balances exploration and exploitation at any stage; therefore, the programmer does not need to encode "two modes" of operation as in the passive-learning policy described in the previous subsection.

The multiple hypotheses policy is only applicable when the agent knows the possible values of the unknown properties of the environment a priori. In this example, the agent knows a priori there is a light car ($M_L = 1$) and a heavy car ($M_H = 2$). The policy can be extended to incorporate more cars; however, the number of hypotheses must be always finite. The next subsection describes an agent that uses the unknown parameters implementation of the dual control policy. This agent is able to control a car with an unknown mass value that can take any value in the interval $M \in [1\ 2]$ (i.e., an infinite number of hypotheses).

6.2.4 Unknown Parameters Learning Policy

In the previous subsection we presented the results of an agent using the multiple hypotheses implementation of the dual control policy. The agent can take full advantage of the information it gathers during the performance of the task to disambiguate among the alternative hypotheses. However, the agent must known all the possible values (i.e., hypotheses) to accomplish the task. In this subsection we describe the results of an agent that uses the unknown parameters implementation of the heuristic solution to perform dual control.

In this experiment the mass of the car may vary in a continuum between light and heavy ($M \in [1\ 2]$) and it can change from trial to trial with uniform probability. Unlike in the multiple hypotheses scenario, the agent does not know a priori the possible values for the mass. The agent will need to apply different forces at different states and rely on the Kalman filter perception function to estimate the value and variance of the mass. The results shown below corresponds to a learning adaptive agent that uses the unknown parameters implementation policy to solve the double integrator problem using dual control. The internal state of the agent consists of four variables: position, velocity, mass mean, and mass standard deviation. The perception function updates the position and velocity directly form every sensation. Additionally, the function uses the Kalman filter algorithm described in Section 6.1.2 to update the knowledge level of the agent. In this particular experiment the mass of the car remains constant within every trial; therefore, the prediction steps (i.e., the first two steps) of the Kalman filter algorithm are not required. The function approximator is the instance-based and the distance function is multilinear. The adaptation method is value iteration using a perfect model of the environment and it is invoked after the conclusion of every trial. The agent is trained with 50 trials. During trial the agent attempts to drive a car of unknown mass. Table 22 shows the design of the agent.

Although the agent can control a car with an unknown mass in the interval $M \in [1\ 2]$, we trained the agent using the same experimental setting we used in the previous experiments. Specifically, the agent controls either the light or the heavy car with 50% probability for 200 stages. We chose this experimental setting in order to contrast the results of this agent with the ones we obtained in the previous experiments. Figure 43 shows the performance of the agent during the 50 trials of training. The figure shows the actual realization (i.e., sequence of light and heavy cars) that was used during training and separates the performance by type of car to help compare the performance within each car. We used the same training sequence in the training of the multiple hypotheses agent to compare the performance of both agents; however, the agent using unknown parameters agent is not constrained to use only the light or heavy cars, but any car with an unknown mass between light ($M_L = 1$) and heavy ($M_H = 2$). As in the previous experiment, the agent starts without knowing how to select actions. The first estimate of the value function produces a policy that only seeks for immediate reward but keeps the position and velocity of the car within bounds. After subsequent trials, the improved value function estimates produce policies that take the car to the goal with better performance. The agent learns to improve performance regardless of the type of car it controls and the experience collected during earlier trials transfers to later trials.

Table 22: Design of the Unknown Parameters Agent for the Double Integrator.

Factor	Description
Variables	4: position: $p \in [-1\ 1]$ velocity: $v \in [-1\ 1]$ mean: $m \in [1\ 2]$; standard deviation: $s \in [0\ 1]$;
Cases	Uniformly distributed in a grid: $21 \times 21 \times 11 \times 5$ 21 points along p. 21 points along q. 11 points along m. 5 points along s.
Decision	One-step Search: For every action u_i from $u_{\min}$ to $u_{\max}$ at step Δ_u: $\hat{u}_t^* = \arg\max_{u_i \in U} E_{\hat{z}_{t+1}} \left[\hat{R}(\hat{x}_t, u_t) + \gamma \hat{V}_{w_t}(\hat{s}_{t+1}) \right]$ $\Delta_u = \frac{a_{\max} - a_{\min}}{25} = 0.08$
Sensation	Direct observation of position and velocity: $z_t = H(x_t) = \{ p_t, v_t \}$
Perception	unknown parameters perception (Kalman filter): p_t and v_t as collected from z_t. $K = s_{t-1} a_t (1 + a_t^2 s_{t-1})^{-1}$ $m_t = m_{t-1} - K(a_t m_{t-1} - u_t)$ $s_t^{-1} = s_{t-1}^{-1} + a_t^2$ where K is the Kalman gain and and a_t is current acceleration of the car.
Adaptation	Value iteration algorithm (Figure 34): Performed after every trial. Environment dynamics $\hat{F}(\cdot)$ and reward function $\hat{R}(\cdot)$ as described in Section 4.4.1.

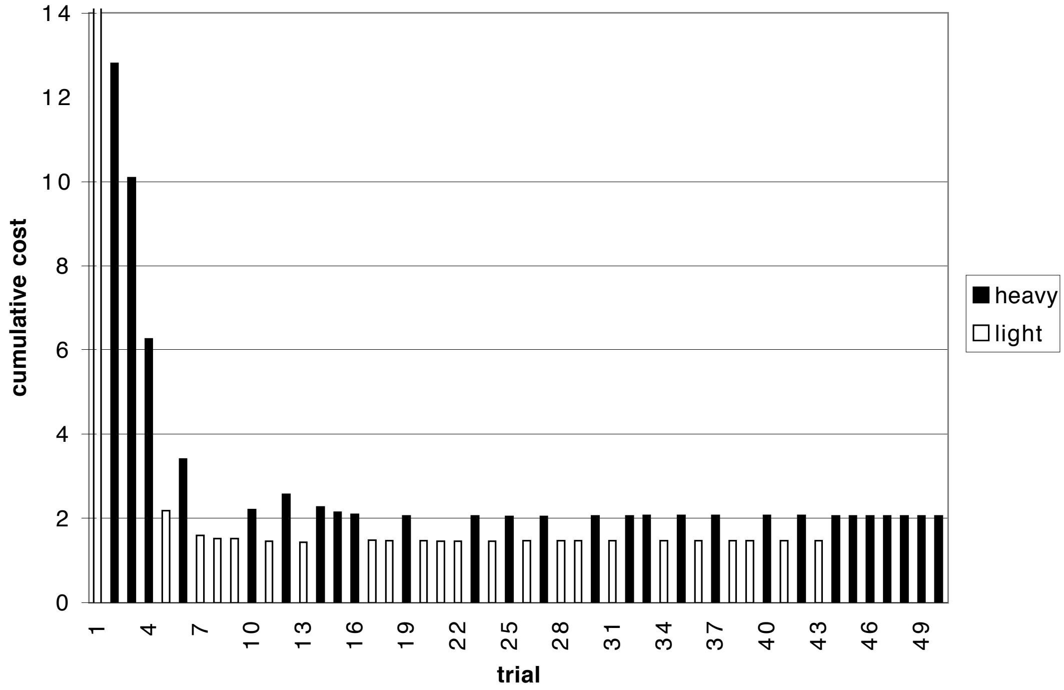

Figure 43: Performance of the Unknown Parameters Policy in the Double Integrator. The graph shows the cumulative cost per trial of the training sequence of 50 trials. The white and black bars correspond to the light and heavy cars respectively.

Figure 44 shows the trajectory in state space for the agent controlling the light and heavy car at trial 50. Figures 45 and 46 show the corresponding a plot of the agent's belief of the mass and standard deviation as a function of the stage in both cars at trial 50.

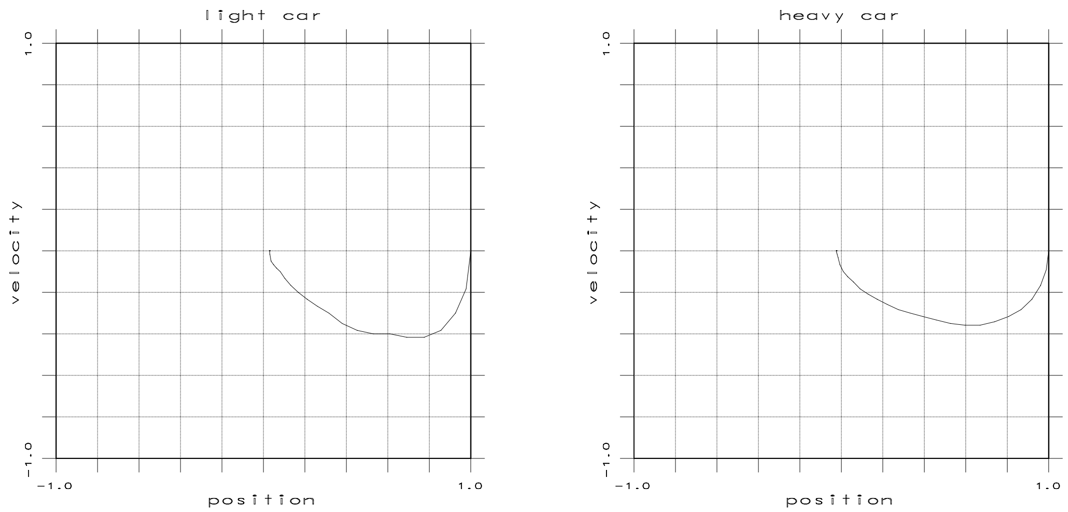

Figure 44: Trajectory of the Car in State Space.
Left: light car. Right: heavy car.

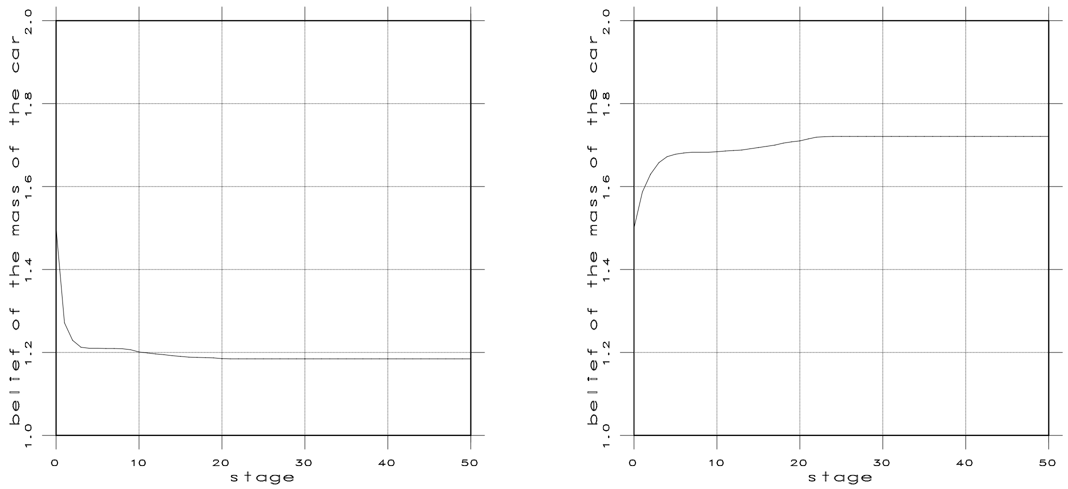

Figure 45: Belief of the Mass of the Car.
Left: light car. Right: heavy car.

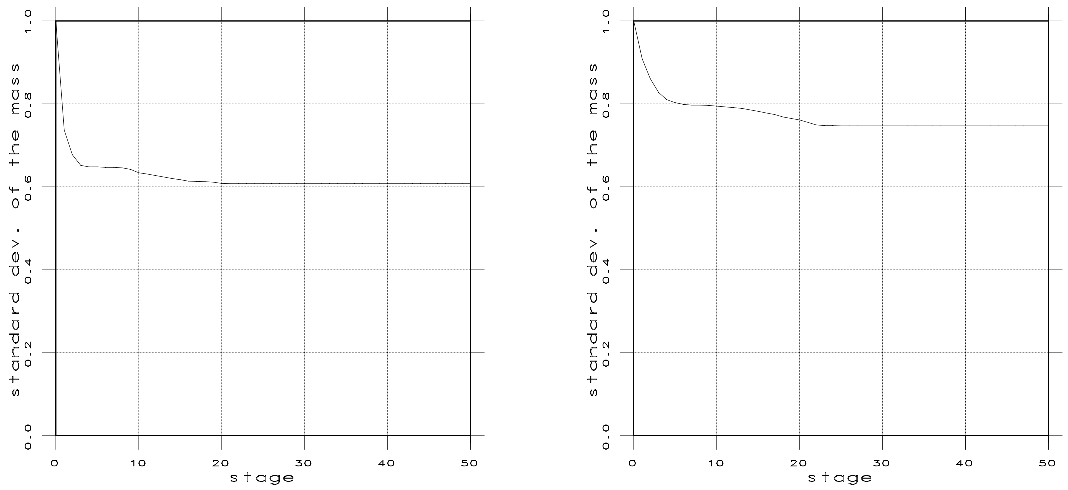

Figure 46: Belief of the Standard Deviation of the Mass.
Left: light car. Right: heavy car.

Unlike the agent using the multiple hypotheses policy, this agent can control cars with any mass values in the internal between light ($M = 1$) and heavy ($M = 2$). Figure 47, 48, and 49 show the trajectory in state space, the mass, and the standard deviation on cars with other mass values at trial 50.

As in the previous experiment, the agent using the unknown parameters policy takes maximum advantage of the information it collects during the task. Additionally, it selects actions that balance the control energy between making progress on the task, finding the mass of the car, and reducing the uncertainty. The agent also takes advantage of the models of the environment and task to decide the best action to take at any stage (decision step in Figure 34) and to adapt the value function using value iteration (adaptation step in Figure 34). The unknown parameters policy performs similarly than the multiple hypotheses policy in the light and heavy car; however, the former is applicable to control a car with an unknown mass in the interval $M \in [1\ 2]$ as demonstrated in Figure 47.

6.3 Discussion

We designed the previous experiments to contrast fixed (non-learning), passive-learning, and active-learning agents acting on several instances of the double integrator problem. In this section we discuss in detail the results obtained in those

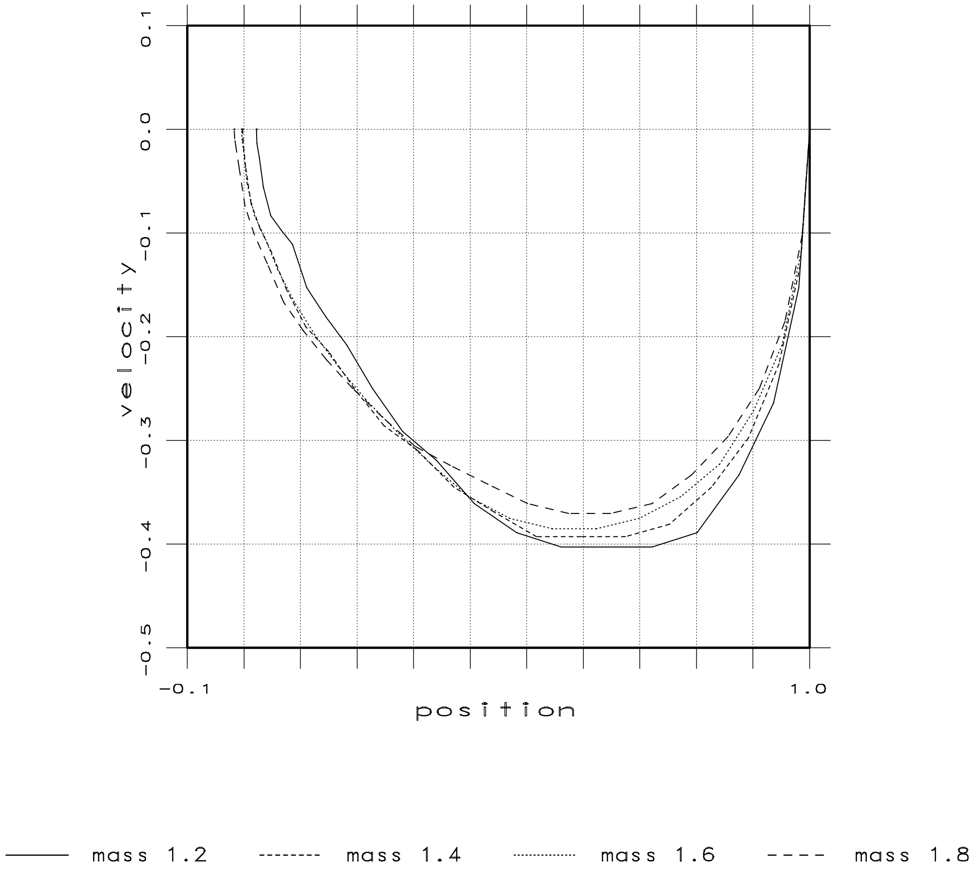

Figure 47: Trajectory of the Car in State Space.
Four different cars with masses 1.2, 1.4, 1.6, and 1.8.

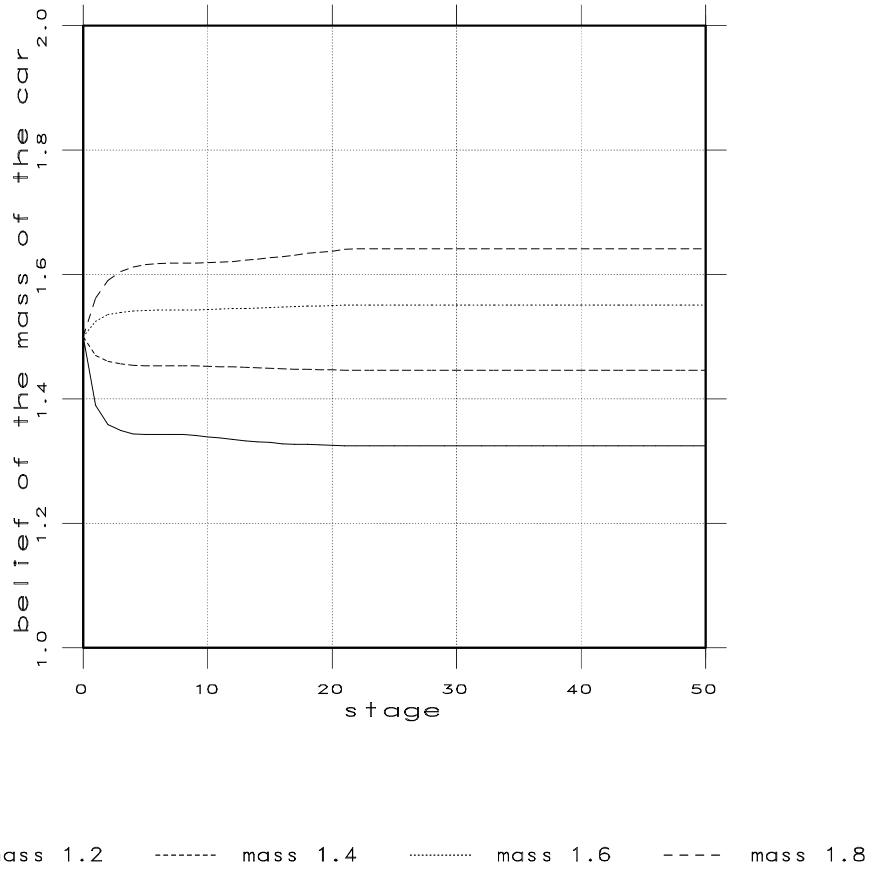

Figure 48: Belief of the Mass of the Car.
Four different cars with masses 1.2, 1.4, 1.6, and 1.8.

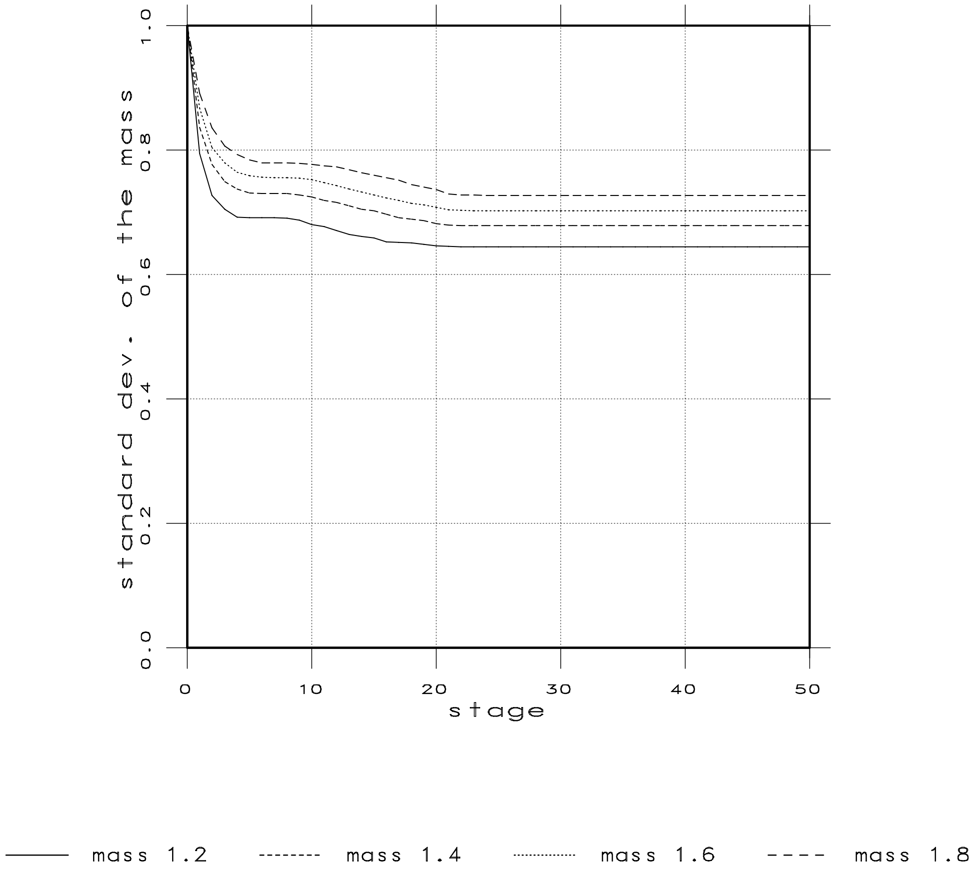

Figure 49: Belief of the Standard Deviation of the Mass.
Four different cars with masses 1.2, 1.4, 1.6, and 1.8.

Table 23: Summary of Results for the Double Integrator at Trial 50.

Agent	Cumulative cost		Overall cost [a]
	light car	heavy car	
Optimal-Light	1.4293	2.1712	1.8003
Optimal-Heavy	1.4981	2.0101	1.7541
Random Exploration	1.4673	2.0354	1.7514
Multiple Hypotheses	1.4610	2.0253	1.7432
Unknown Parameters	1.4529	2.0302	1.7416

[a] The overall performance is computed as: performance in light car $\times$ 0.5 + performance in heavy car $\times$ 0.5.

experiments so that we can draw some conclusions about the robustness and effectiveness of each type of agent.

Table 23 shows the final results of the cumulative cost of agents using four types of policies: fixed non-learning, random exploration learning, multiple hypotheses, and unknown parameters. The last two policies are dual control policies and represent two different implementations of the heuristic solution to sequential decision-making problems we present in this dissertation. Additionally, the last two agents perform adaptation of the value function; therefore, we contrast the results obtained at trial 50 to contrast the performance of these agents after the value function estimates had converged.

There are two agents using fixed non-learning policies: optimal-light and optimal-heavy. Such agents always use the optimal policy to control the light and heavy car respectively; therefore, they perform optimally only 50% of the time. The results of the optimal-light policy acting in the light car (1.4293) and optimal-heavy policy acting in the heavy car (2.0101) correspond to the optimal performance of an agent controlling the double integrator. These cumulative costs are only achievable when the agent knows the mass of the car from the beginning of the trial; therefore, they should be seen as an unreachable lower bound for agents that start without knowing the mass of the car. The overall cost of the optimal-light agent (1.8003) is worse that the one of the optimal-heavy agent (1.7541). This indicates that it is more critical to control the heavy car than the light car. This result is as expected because the reward function penalizes the agent more strongly when it uses stronger forces, which is what the agent applies when it is controlling the heavy car. Although the design of the fixed non-learning agent is very simple, this only occurs because the double integrator admits a close-form optimal solution (Equation 68). In general, fixed non-learning policies do not take advantage of any information the agent collects during a trial nor perform optimally in any of the possible scenarios (e.g., light or heavy car).

The agent using the random exploration policy can take advantage of the information it collects during the trial. Additionally, it improves its level of knowledge by taking random actions superimposed to actions that make progress on the task. Once the agent has collected "enough" information, it uses either the optimal-light or optimal-heavy policy to continue the task. The policy is easy to design because it uses the close-form optimal solution of the double integrator (Equation 68). The cumulative cost of the random exploration agent is greater than the best cumulative cost of the optimal-light and optimal-heavy agents when contrasted individually, but the overall cost is better than any of the two fixed non-learning agents. This occurs because the cost of performing random exploration and acquiring knowledge at the beginning of the trial is compensated by better performance at the end of the trial. However, the random exploration agent is not able to balance the cost of performing exploration with the long-term benefit the new information will bring to the performance of the task. Figure 38 shows a jerky trajectory at the beginning of the trial due to the execution of random actions. Figure 39 shows than the agent can successfully increase the knowledge level because the probability of belief increases or decreases rapidly when the agent is controlling the light or heavy car, respectively.

The multiple hypotheses agent is able balance the control energy between exploration and exploitation. The performance within any single car is better than the one achieved by the random exploration agent; therefore, the overall performance is also better. It is interesting to note that the agent does not require to fully identify the mass of the car before proceeding to the target (Figure 42). Additionally, although the entire sequence of actions is used to perform exploration, there is more exploration at the middle of the trial rather than the beginning or end (Figure 41 shows more jerkiness in the middle). This may occur because the reward function penalizes the agent more heavily as the car is farther away from the target. Thus, increasing the exploration pays off when the car is nearer rather that farther from the target point. Figure 42 also support this conclusion. There are two phases in which the belief increases (for the light car) or decreases (for the heavy car). The first phase is during the first 5 decision stages when the belief changes drastically from 0.5. The second phase occurs during decision stages 12 to 25, approximately when the belief drifts smoothly in the right direction. By stage 30, the agent does not incur into heavy exploratory actions because the car is very near the target point and the cost incurred for additional exploration does not pay off in the long term.

The unknown parameters agent is also able to balance the control energy between exploration and exploitation. As with the multiple hypotheses agent, the performance within any single car is better than the one achieved by the random exploration agent. The performance of the unknown parameters agent is very similar to the one of the multiple hypotheses agent; however, the former can control cars with masses other than light ($M_L = 1$) or heavy ($M_H = 2$). Figures 47, 48, and 49 show the performance of the unknown parameters agent in cars with masses $M = 1.2$, 1.4, 1.6, and 1.8. Additionally, Figure 49 shows that the pay off of improving the accuracy of the mass estimate (i.e., reducing the standard deviation of the mass estimate) is less as the mass of the car increases. This is because the heavier the car the more difficult it is to control and the less cost the agent is willing to incur to perform exploratory actions. The last two experiments demonstrate that both dual control agents, multiple hypotheses and unknown parameters, are not required to know exactly the mass of the car to successfully control it. Both agents perform better than the fixed non-learning and passive learning agents. These results demonstrate the need for selecting actions that accomplish both making progress on the task and gather new information such that the cost incurred by the agent when executing exploratory actions pays off in the long term.

6.4 Conclusions

In this chapter we focused on the issue of sequential decision-making problems in the context of incomplete and dynamic knowledge. In this context, the agent must select actions that accomplish dual control. In other words, the agent must select actions that make progress on the task and actions that help reveal new information about the unknown properties of the environment. We presented two different implementations of the heuristic solution that accomplish dual control: multiple hypotheses and unknown parameters. In the multiple hypotheses solution the agent knows a discrete set of possible values or hypothesis for the unknown properties of the environment and represents its knowledge level using probabilities that indicate the agent's belief of each hypothesis. In the unknown parameters solution the agent knows which are the unknown properties of the environment but does not know their values. The agent represents its knowledge level using an estimate and covariance matrix of the unknown values. We contrasted the performance of agents using both solutions with agents using fixed non-learning and passive learning policies in the double integrator. The dual control agents achieved the best performance.

The two instantiations of the heuristic solution exhibit the dual characteristic of appropriately distributing the control energy for learning and control purposes. As described in Chapter 3, the heuristic solution is an approximation based on the principle of optimality that uses finite storage resources to represent the information state and the value function, and finite computational resources to compute the value function. The control algorithm possesses the distinguishing characteristic of regulating its learning as required by the performance index; and it performs this by considering the three consequences each action produces: the immediate reward, the next state in the environment, and the next knowledge level in the agent. The best action is the one that maximizes the sum of the immediate reward and the discounted value of the next state and knowledge level taken together. Some actions may change the state and leave the same knowledge level. The resulting state may presumably be a good one, but if the resulting knowledge level cannot support future performance then their combined value may not be good. Similarly, other actions may change the knowledge level and leave the same state of the environment. The agent may now have better knowledge of the situation but the lack of immediate progress may cause poor task performance. The optimal decision is the one that best compromises between the progress on the task and the knowledge gain. However, the agent's only objective is to maximize the discounted sum of rewards. Thus, knowledge gathering actions are considered only with respect to this objective. This means that if the agent decides not to make much progress by collecting information at some point during the task it is only because such new knowledge will help the agent to make better, improved progress later in the task. We used the double integrator to demonstrate the computational feasibility of the algorithm and its performance level when applied to a specific problem, and to provide some insight into the dual control theory.

In the previous three chapters, we studied in isolation the issues of continuous domains, imperfect perception, and incomplete knowledge, respectively. As discussed in Section 1.6, these issues are central to realistic sequential decision-making problems. Armed with the resulting insights into these aspects of the problem, we can now explore the proposed heuristic solution to the sequential decision-making problem that exhibits all the characteristics of interest. In the next chapter we describe the application of the heuristic solution to solve a complex problem that addresses the three issues simultaneously.

CHAPTER VII

MARS ROVER

In Chapters 4, 5, and 6 we focused on the issues of continuous domains, imperfect perception, and incomplete knowledge, respectively. As discussed in Chapter 1 and summarized in Figure 3 on page 18, these issues characterize a broad class of practical sequential decision-making problems. In each of these chapters we instantiated the heuristic solution described in Chapter 3 to address each particular issue, and evaluated it on well-known problems against existing solutions to those problems. In this chapter, we instantiate the heuristic solution to solve a complex problem that combines all the three issues. The particular problem is a simulation of a Mars rover. The rover is a mobile robot capable of navigating in a terrain with obstacles and collectible rocks. The robot is equipped with sensors that can locate large obstacles reliably but not collectible rocks due to their small size. Specifically, the reliability of the sensor in detecting rocks increases inversely proportional to the distance between the robot and the rock (i.e., the closer the robot is to the rock the better the robot can determine if the rock is worth collecting). The mission of the rover is to find and collect a useful rock as quickly as possible while minimizing collisions with obstacles and the energy expended. The rover is given a set of possible locations or *targets* that may contain collectible rocks. The rover faces a sequential decision-making problem having the three main characteristics studied in this dissertation: continuous state and action spaces, imperfect perception, and incomplete knowledge. The results described in this experiment show the feasibility of the heuristic approach for solving this type of problems.

Section 7.1 presents the sequential decision-making problem faced by the rover. Section 7.2 describes the details of the implementation of the internal state representation, the sensor and perception functions, the one-stage-lookahead search, and the value function adaptation process. Section 7.3 presents the results of fixed non-learning, passive-learning, and active-learning agents on the Mars rover problem. Section 7.4 discusses the results and Section 7.5 concludes the chapter.

7.1 Mars Rover

The Mars rover is a mobile robot with limited perception. The rover knows the terrain layout (i.e., where the obstacles and possible collectible rocks are located), but it does not know if the rocks are worth collecting. The task of the rover is to maximize the probability of success in finding a rock while minimizing the time and energy in doing so and without colliding with obstacles. This will require the rover to approach each of the possible targets following a trajectory that will enable it to learn more about the rocks at the targets and finding a collectible one. Figure 50 shows a schematic of the Mars rover problem.

The Mars rover problem has the three basic characteristics in question: continuous domains, imperfect perception, and incomplete knowledge. The domain is continuous because the state (i.e., the position of the rover, obstacles, and targets) are real-valued variables. Additionally, the action of the rover (i.e., direction in which to move) is continuous. The perception of the rover is imperfect because it is limited by a horizon and its reliability is affected with distance. In other words, the rover cannot see collectible rocks that are located beyond the horizon, and when they lie within the horizon the rover can only detect collectible rocks with a probability that depends on the distance. The knowledge of the rover is incomplete because it does not know which rocks are worth collecting. However, the knowledge level is dynamic because the rover can learn more about the environment by moving around and getting closer to the rocks.

The sequential decision task in this problem is to learn to efficiently approach and find a collectible rock. The rover is given a set of targets that may contain collectible rocks and it should follow a trajectory that efficiently balances approaching one of the targets in the set and exploring other targets. A trajectory that takes the rover directly to the closer target is not good enough because the rover may miss finding a rock at that target or a target with higher probability of having a collectible rock. Conversely, a trajectory that takes a long path through all the environment before reaching a target will enable the rover to learn more about the rocks at each target but also take too much time and energy.

7.2 Details

The internal state of the rover consists of the layout of the environment (i.e., positions of the rover, obstacles, and targets) and set of probabilities associated with each target representing the belief that it has a collectible rock. The environment layout represents the environment state estimate $\hat{x}$ and the probabilities represent the knowledge level κ. With every movement the rover updates the internal state

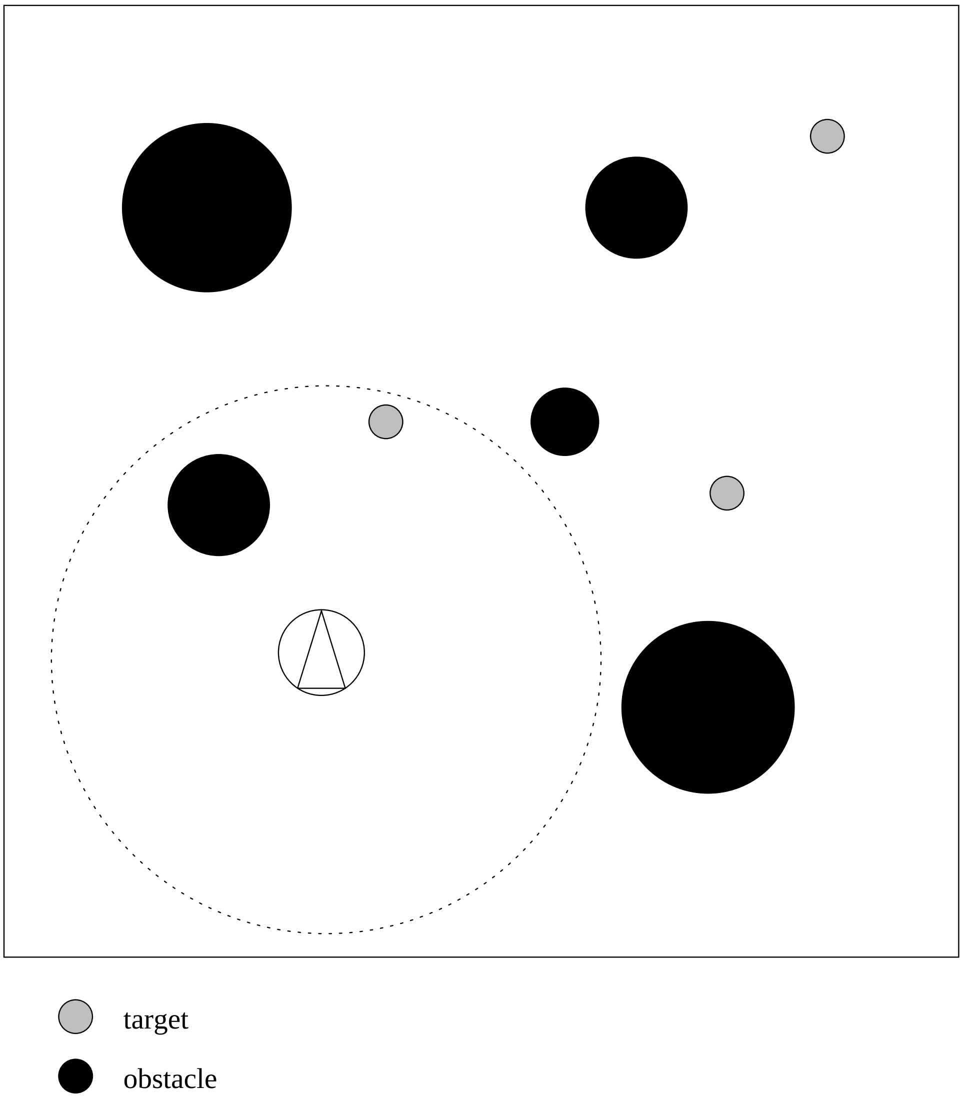

Figure 50: A Schematic of the Mars Rover Problem.
The black and grayed circles correspond to obstacles and targets, respectively. The dotted circle around the rover corresponds to the perception horizon (i.e., the rover cannot perceive collectible rocks beyond that circle.)

by adjusting the positions of the obstacles, targets, and itself, and the belief that the targets have collectible rocks according to the incoming sensations. Equation 70 shows the internal state representation of the rover.

$$\begin{array}{rc} s_t = \{ & (h_R, v_R)_t, \\ & (h_{O_1}, v_{O_1})_t, \ldots, (h_{O_n}, v_{O_n})_t \\ & (h_{G_1}, v_{G_1})_t, \ldots, (h_{G_m}, v_{G_m})_t \\ & p_t^1, \ldots, p_t^m \qquad \} \end{array} \tag{70}$$

where $(h_R, v_R)_t$, $(h_{O_i}, v_{O_i})_t$ $i = 1, \ldots, n$, $(h_{G_j}, v_{G_j})_t$ $j = 1, \ldots, m$ represents the position of the rover, obstacle i, and target j in the environment at stage t, respectively. The probabilities p_t^j represent the belief of the agent that target j has a collectible rock.

The sensor function $H(\cdot)$ delivers the position of the rover, obstacles, and targets, and a set of binary signals indicating the presence of collectible rocks at the targets at every stage. Each binary signal corresponds to one target and indicates if the associated target has a collectible rock ($z^j = 1$) or not ($z^j = 0$). The signal associated with each target is unreliable and the uncertainty depends on the distance r between the sensor and the target. The behavior of the signal is modeled using a probability function, $P(z(r)\,|\,\text{collectible}) = P_u(r)$, and the false alarm probability $P(z(r)\,|\,\neg\text{collectible}) = P_f(r)$ (a Type I error). The missed detection probability (Type II error) is given by $P(\neg z(r)\,|\,\text{collectible}) = 1 - P_u(r)$. Note that the agent cannot determine if a rock is collectible or not using a single sensor reading due to the stochastic behavior of the sensor. Moreover, the detection probabilities depend with the distance between the sensor and the target; thus, the rover needs to collect several readings and preferably from different positions to increase its belief that a given rock is collectible. The specific probability distribution of the sensor will, of course, depend on the physical properties of the rock being measured, the performance characteristics of the sensor systems, and the environment of operation of the rover. The heuristic approach does not depend on the specific distributions and its source will not be a concern here. Figure 51 describes the target detection probability distribution used in the Mars rover.

The reward function consists of four components, a constant penalty for each stage, a penalty for the amount of energy used, a penalty for getting close to obstacles,[1] and a bonus for finding a collectible rock. The constant penalty for each stage forces the rover to act quickly (i.e., accomplish the task in the minimum number of stages). The penalty for the amount of energy used consists of the quadratic magnitude of the motion vector (i.e., the higher the velocity the more energy the

[1]There is also a huge penalty $r_t = -1000$ when the rover collides with an obstacle.

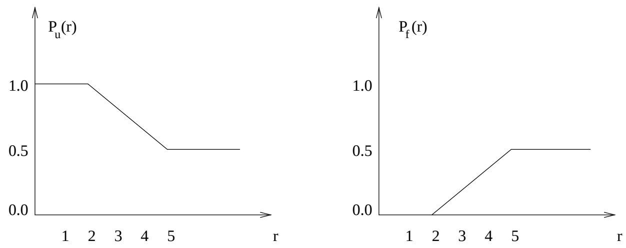

Figure 51: Target Detection Probability Distribution.
Left: $P_u(r)$. Right: $P_f(r)$. The sensor function can detect with 100% reliability whether a rock is collectible or not when the target is 2 meters or less from the sensor. The sensor function reports no information at all when the target is more than 5 meters because it has a success rate of 50% of detecting a collectible rock.

rover expends).[2] The penalty for obstacles is a quantity that is inversely proportional to the quadratic distance (up to a sphere of influence) between the rover and the obstacles. Thus, the rover will tend to stay away from obstacles. The reward function gives a unit bonus when the rover reaches a collectible rock within a sphere of influence. Maximizing the sum of rewards will require a compromise between the minimizing the cost of reaching a collectible rock while exploring all the possible targets. Equation 71 describes the reward function mathematically:

$$R(x_t, u_t) = \begin{cases} 1 & \|(h_R, v_R) - (h_{G_j}, v_{G_j})\| \leq S_G \ \vee \ G_j \text{ has a rock} \\ -\text{cost} & \text{otherwise} \end{cases} \tag{71}$$

$$\text{cost} = Q_s + Q_u \|u\|^2 + Q_o \sum_{i=1}^{n} (1 - \|(h_R, v_R) - (h_{G_j}, v_{G_j})\| / S_O)^2$$

where S_G and S_O are the sphere's of influence for obstacles and targets, respectively, $\|\cdot\|$ denotes the distance operator, and Q_s, Q_u, and Q_o denote the proportionality constants for speed, energy, and obstacles, respectively.[3]

The next subsections describe the details of the implementation of the model-based heuristic approach for the Mars rover problem.

[2]The actual amount of energy the rover expends depends on many factors including the type of terrain, the slope, the mass of the rover, etc.; however, a simple approximation of the energy consumed by the rover at each stage is its quadratic velocity, which is a measure proportional to the kinetic energy of the rover at that stage.

[3]In the experiment $Q_s = Q_u = Q_o = 1$.

7.2.1 Perception

The rover uses the sensor function to receive the new sensation and the perception function to compute the new internal state. By assumption, the sensor function delivers the new location of the rover, obstacles, and targets directly. It also delivers m binary signals indicating the usefulness of each of the targets, but the signals may be noisy according to the sensor's detection probability distributions.

The perception function (Equation 18) is implemented by the functions $\Psi_x(\cdot)$ and $\Psi_\kappa(\cdot)$. The $\Psi_x(\cdot)$ is responsible for updating the internal state representation of the environment's layout. For that purpose, it simply takes the locations of the rover, obstacles, and targets from the sensation and updates the internal state representation for the environment layout. The $\Psi_\kappa(\cdot)$ is responsible for updating the set of belief probabilities p^j's. For that purpose, it uses Bayes' rule and the detection probability distribution of the sensor:

$$p_t^j = p_{t-1}^j \frac{P(\, z_t(r) \,|\, G_j \,)}{\sum_{k=1}^m P(\, z_t(r) \,|\, G_k \,) p_{t-1}^k} \tag{72}$$

where $P(\, z_t(r) \,|\, G_k)$ is the conditional probability of receiving sensation z_t at the current location given that target j has a collectible rock. The previous belief of target j having a collectible rock, p_{t-1}^j, serves as the prior and is obtained directly from the rover's internal state. The new belief estimate p_t^j is the a posterior belief probability and is subsequently stored again in the internal state.

7.2.2 Decision

The rover uses a model of the dynamics of the environment $\hat{F}(\cdot)$,[4] the current internal state s_t, and the current estimate of the value function $\hat{V}_{w_t}(\cdot)$ to select the action $\hat{u}^*$ leading to the best long-term benefit estimate at the given internal state s_t. Actions represent velocity vectors.

The one-stage lookahead search (Equation 21) is implemented as

$$\begin{aligned} \hat{u}_t^* &= \arg\max_{u_t} E_{z_{t+1}} \left[R(x_t, u_t) + \gamma \hat{V}_w(s_{t+1}) \,|\, s_t \right] \\ &= \arg\max_{u_t} \left[\hat{r}_{t+1} + \gamma \sum_k \hat{V}_w(\hat{s}_{t+1}^k)\, P(z^k|G_j)\, p_t^j \right] \end{aligned} \tag{73}$$

where action $u_i = (\rho_i \cos(\theta_i), \rho_i \sin(\theta_i))$ is the velocity vector, $\hat{s}_{t+1}^k = \Psi(s_t, u_i, z^k)$ is the predicted internal state under action u_i and sensation z^k, $P(z^k|G_j)$ is the

[4]The model is very simple since it just updates the current position of the rover according to the action using a linear formula: $x_{t+1} = \hat{F}(x_t, u_t) = x_t + u_t\Delta_t$. This is an approximation of the real dynamics used by the rover, which is a quadratic (i.e., inertial) formula.

probability of receiving sensation z^k given that the target G_j has a collectible rock, and p_t^j is the current belief of the agent that target G_j has a collectible rock.

The interpretation of the one-stage lookahead search in the Mars rover is as follows: at stage t, when the agent has internal state s_t and performs action u_i, two events happens. First, the rover finds itself at a new position as described by $\hat{F}(\cdot)$. Second, it receives a new sensation that depends on the new location, and as a consequence of that, its internal state changes according to the perception function. Since the rover does not know what sensation it is going to receive at the new location, it must predict all possible sensations z^k and compute the corresponding internal states $\hat{s}_{t+1}^k$ based on its current belief. The predicted internal states represent what the agent would know at the next stage given action u_i and sensation z^k as predicted by the current belief. The value of being at such states is represented by $\hat{V}_w(\hat{s}_{t+1}^k)$. Thus, the best action is the one that maximizes the sum of the immediate reward ($\hat{r}_{t+1}$) the rover is going to receive and the value of the next internal state ($\hat{V}_w(\hat{s}_{t+1}^k)$) given its current knowledge. The sum must be weighted using the agent's current belief (i.e., the p_t^j).

The immediate reward $\hat{r}_{t+1}$ is computed using the current beliefs of the agent:

$$\hat{r}_{t+1} = \begin{cases} p_t^j & \|(h_R, v_R) - (h_{G_j}, v_{G_j})\| \leq S_G \\ -\text{cost} & \text{otherwise} \end{cases} \tag{74}$$

$$\text{cost} = Q_s + Q_u\|u\|^2 + Q_o \sum_{i=1}^{n}(1 - \|(h_R, v_R) - (h_{G_j}, v_{G_j})\|/S_O)^2$$

7.2.3 Adaptation

The rover represents the value function using a special configuration of the case-based function approximator (see Chapter 4). In the special configuration cases are arranged in a uniform grid across the entire internal state space. The space consists of the location variables of the rover, obstacles, and targets as well as the belief probabilities.[5] Thus, the number of parameters of the case-based function approximator is equal to the number of points in the uniform grid (i.e., a case for each point). The rover uses the value iteration method described in Chapter 3 to adapt the parameters of the value function.

[5]When the rover operates in a fixed environment, the variables associated with the position of obstacles and targets can be omitted since the only information that varies during the task is the rover's location and belief probabilities.

7.3 Results

In this section we describe and contrast the results we obtained from fixed non-learning, passive-learning, and active-learning agents acting on the Mars rover. The fixed non-learning agent use motor schema based reactive navigation as the basic control scheme. This agent moves pursuing a given target until it can reach it without collecting new information about collectible rocks as it moves. Once the rover reaches the target, if it determines that the target does not have a collectible rock, then it moves to the next target following a preprogrammed sequence of targets. The passive-learning agent uses specialized optimal-controllers to move to each target. There is one controller associated with each target and each controller can direct the rover from any location in the terrain to the associated target following a trajectory that minimizes the cost. The passive-learning agent collects additional information as it moves; therefore, the agent may decide to pursue an different target when its belief state indicates that the likelihood of finding a collectible rock at the current target is very low; however, the agent does not select actions explicitly taking into account the new information each action may produce. The active-learning agent uses the implementation of the heuristic solution we presented in the previous section. This agent move towards the most likely target of having a collectible rock according to its belief state. Additionally, the agent approaches the target using a trajectory that best balances the energy used to approach the target and collect additional information as it moves.

Figure 52 shows the environment we used to test the agents. The white and black circles represent the targets and obstacles, respectively. The black rectangle represent the location of the collectible rock and can be located in one of the three possible targets with equal probability. The rover cannot perceive the location of the collectible rock because the sensors are imperfect. The sensor behavior follows the model described in Figure 51. The rover simulator had a time step of 0.25 seconds and new control actions were selected every four time steps (i.e., every second). A trial consisted of starting the rover at position $(10, -10)$ and running the simulation until either the rover reached the collectible rock or 60 decision stages had elapsed (i.e., 1 minute). The rover rotated and moved independently following quadratic dynamics (i.e., constantly accelerated motion).

7.3.1 Fixed Non-Learning Agent

In this subsection we describe the results of an experiment in which the agent uses a fixed policy to solve the Mars rover problem. The agent uses a three-state finite state automata (FSA) (Balch et al., 1995) to specify an active target and a schema-based reactive controller (Arkin, 1989) to move towards it. Each of the states in

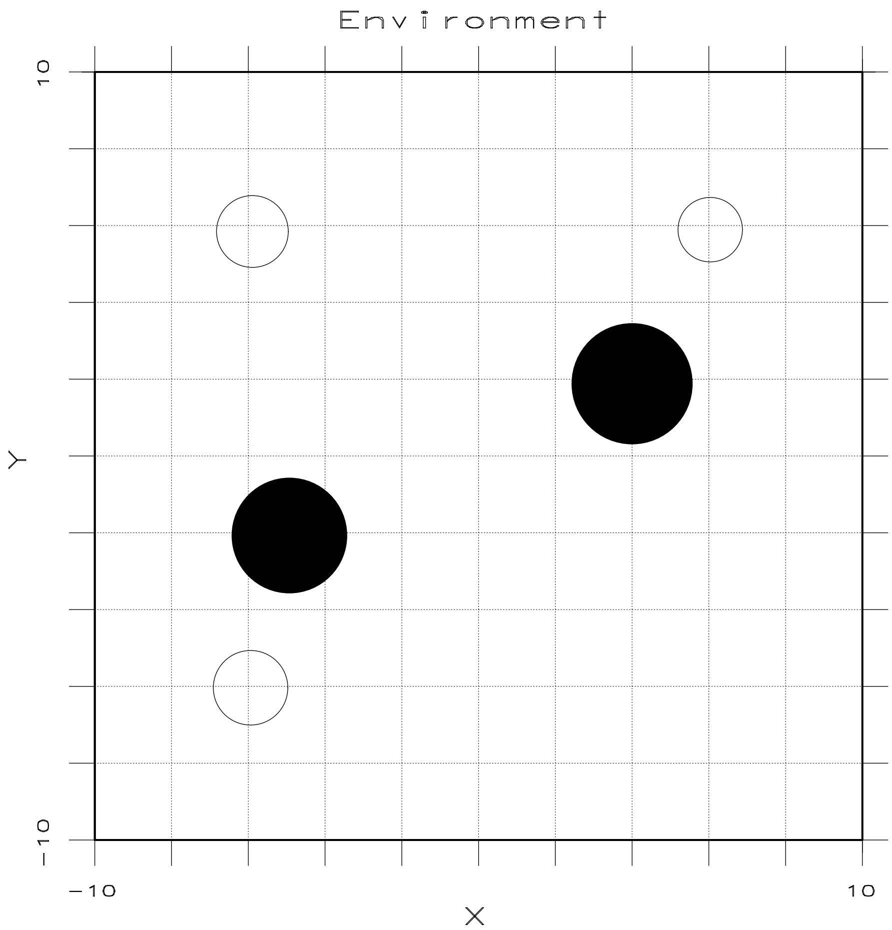

Figure 52: Test Environment.
The open and filled circles correspond to targets and obstacles, respectively. The rover starts at the lower-right corner $(10, -10)$. Distance is in meters. Every cell represents $2\text{m} \times 2\text{m}$.

the FSA specifies an *assemblage* or group of motor-schemas that run only when their corresponding state is active. Transitions among states occur when specific perceptual triggers activate. For example, the current active state may specify an assemblage the moves the rover towards a specific target while avoiding obstacles along the way. A perceptual trigger that activates when the rover is close enough to the given target causes the rover to transition to the next state in the FSA.

In the non-learning agent design, each of the states specifies an assemblage that directs the rover towards a specific target. Each assemblage consists of two motor schemas: AVOID-STATIC-OBSTACLE (ASO) and MOVE-TO-GOAL (MTG), where the goal is one of lower-right, upper-right, or upper-left targets. Figure 53 shows the FSA for the fixed non-learning agent. In this design, the agent always approaches to the lower-left target first, to the upper-left target second, and to the upper-right target last, exactly in that order. Additionally, the non-learning agent does not use a belief state nor collects additional information about the existence of a collectible rock in each target. The agent merely travels to each target in the sequence until it finds a collectible rock. Table 24 shows the design of non-learning agent.

Table 24: Design of the Non-Learning Agent for the Mars rover.

Factor	Description
FSA	Three states (Figure 53): state 1: target is Lower-Left state 2: target is Upper-Left state 3: target is Upper-Right transitions: deterministic
Decision	Motor Schemas: $u_t = G_O\mathrm{ASO} + G_G\mathrm{MTG(target)}$ G_O, G_G, and target as specified by the active state.
Sensation	Direct observation of position and velocity: $z_t = H(x_t) = \{ h_t, v_t \}$
Perception	h_t and v_t as collected from z_t.
Adaptation	None.

Figure 54 shows the trajectory of the rover in two different instances of the problem. The figure shows the trajectory when the collectible rock is in the lower-left (left) and upper-right targets (right). The former trajectory corresponds to the best possible scenario and the latter to the worse. This is because the rover always travels to the lower-left target first, upper-left target second, and upper-right target last.

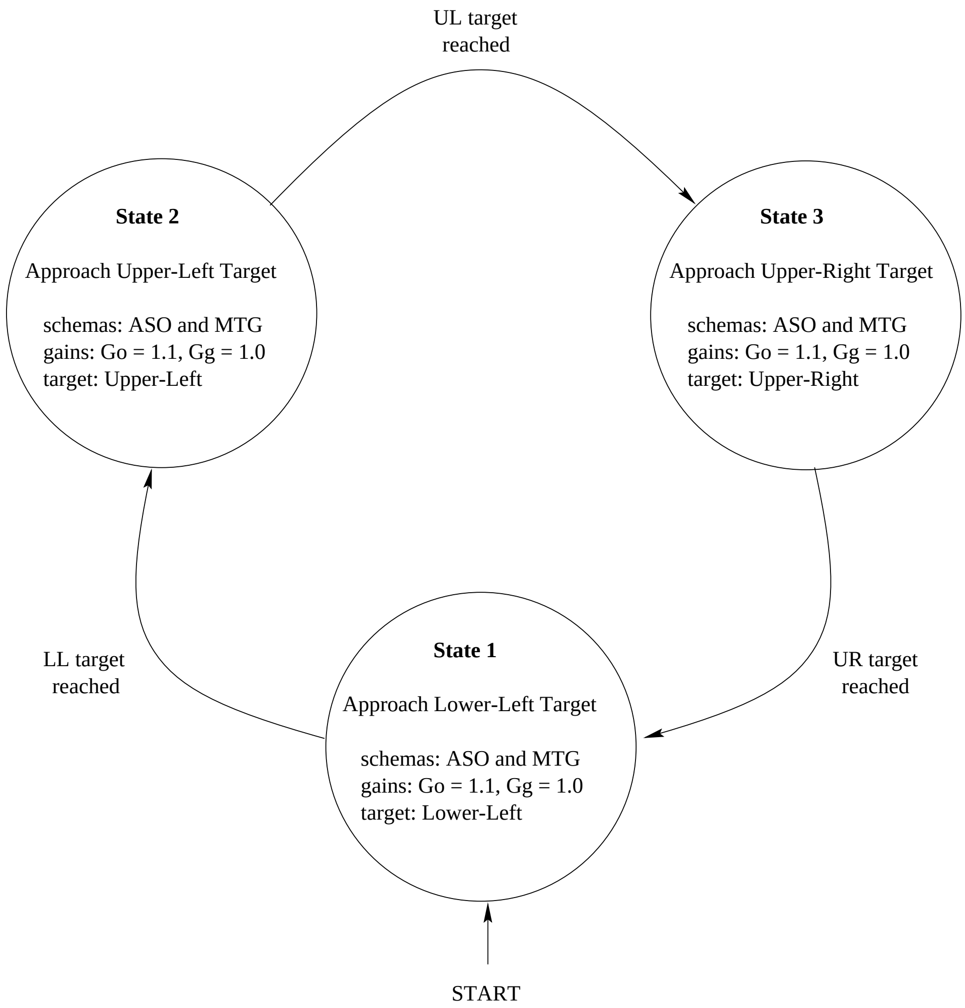

Figure 53: Finite State Automata for the Non-learning Agent.
The FSA for the passive learning agent consists of three states. Each state specifies an assemblage that moves the rover towards a specific target. Once the rover reaches the current target there is a transition to the next state which causes the rover to start moving towards the next target.

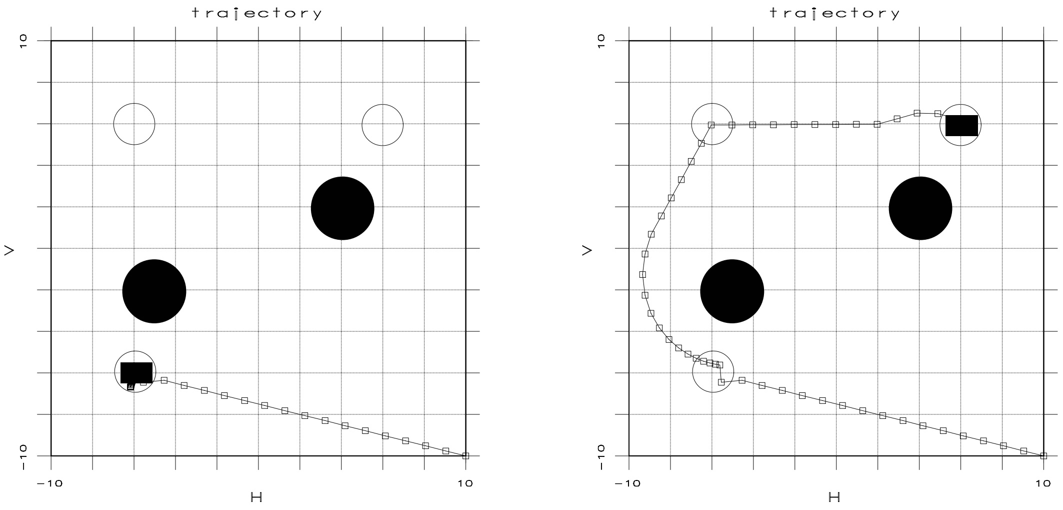

Figure 54: Trajectories for the Non-learning Agent.
Left: collectible rock in lower-left target. Right: collectible rock in upper-right target.

The non-learning agent uses a fixed non-learning policy to solve the Mars rover problem. The agent does not take advantage of additional information the agent may receive as it moves through the terrain. Additionally, most of the resulting behavior of the agent is due to the designers' programming effort because they are responsible for selecting the specific sequence of targets the rover should follow. This particular design may prove useful only when the designers know a priori that it is very likely to find a collectible rock in the lower-left target because this is the target the rover approaches first. A better agent design would use a policy that can take advantage of the information collected during the performance of the task to determine which target is the most likely to contain a collectible rock and change its course accordingly. The next subsection describes the results of an agent that uses a passive-learning policy that implements this idea.

7.3.2 Passive-Learning Agent

In the previous subsection we presented the results of an experiment in which the agent used a fixed policy to solve the Mars rover problem. The agent always follows the same sequence of targets as specified by the FSA regardless of the belief state of the agent. In this subsection we describe the results of an experiment in which the agent uses a passive-learning policy to solve the Mars rover problem. That is, the agent collects information during the task and changes it course whenever there is sufficient evidence to do so.

In this experiment the internal state of the agent consists of the horizontal and vertical position of the rover as well as the probability of belief of finding a collectible rock in each of the three targets. The passive-learning agent uses these five variables (i.e., two for position of the rover and three for the probabilities associated with each target) to decide what action to execute next. In this design, the agent uses a three state FSA and an three specialized optimal controllers to move the rover (i.e., one controller to approach to each target). The active state in the FSA indicates which of the three targets the rover should approach. The agent uses the corresponding optimal controller to direct the rover towards this target. There are two fundamental differences between this design and the non-learning agent design. First, the state transitions in the FSA are stochastic rather than deterministic. The transition probabilities are determined directly by the current values of the belief of the agent. Thus, the agent is not forced to follow a fixed and predefined sequence of targets but the target with the greater chance of containing a collectible rock according to the current belief. Furthermore, the perceptual triggers are associated to the belief state of the targets and activate a transition when the value of the belief drop below some threshold. Specifically, there is a state transition when the belief of finding a collectible rock in the current target drops lower than 5%.[6] The agent uses the perception function described in Section 7.2.1 to update its belief state; therefore, the belief of the agent changes as the rover moves through the environment. This increases the chance for the rover to find a collectible rock faster. Second, the agent uses one of three specialized optimal controllers to move to the designated target instead of a schema-based reactive controller. Each optimal controller is able to direct the rover from any point in the terrain to its associated target using the path that maximizes the sum of rewards. Each optimal controller uses a bidimensional case-based function approximator (i.e., the variables in the function approximator correspond to the location of the rover). The agent uses the value iteration method described in Chapter 3 to adapt the parameters of the function approximator. The reward functions used to train the optimal controllers follow the same form as Equation 74 except that each controller assumes the rover will find the target at the corresponding target.

Figure 55 shows the FSA for the passive-learning agent. In this design, the starting state is selected randomly according to the values of the initial belief of the agent. The state remains active until the rover finds a rock or a perceptual trigger activates. In the latter case, the next active state is selected randomly according to the current

[6] A greater value may force the rover to give up pursuing the current target when in fact it contains a collectible rock.

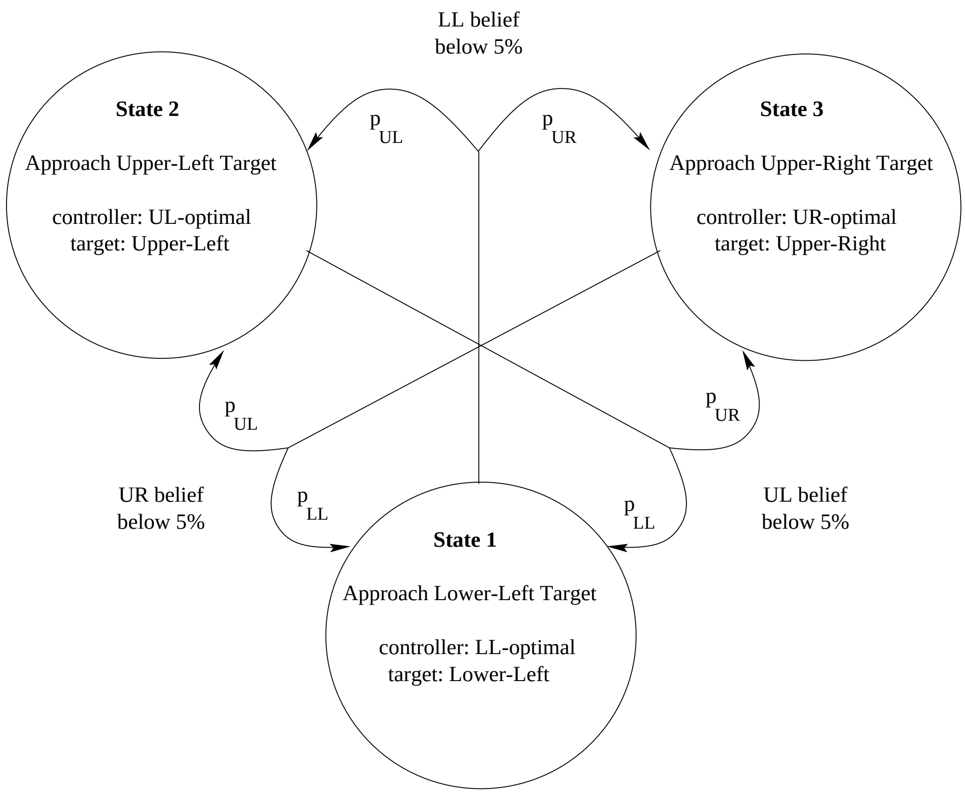

Figure 55: Finite State Automata for the Passive-Learning Agent.
The FSA for the passive learning agent consists of three states. Each state specifies a controller that moves the rover optimally towards a specific target. When the belief of finding a collectible rock drops below 5%, the FSA transitions stochastically to a new state. The transition probabilities are determined directly by the belief values; therefore, the state with target having the greatest belief is the most likely to be active next.

values of the belief. The target with greatest belief is the most likely to be active next. The passive-learning agent collects additional information about the existence of a collectible rock in each target and updates its belief accordingly; however, the agent does not select actions to explicitly acquire new information. Each controller directs the rover optimally towards the current target but independently of any information the agent may collect along its path; therefore, any new information the agent collects is passive or "accidental" although beneficial because the transitions in the FSA are biased towards the targets with larger beliefs. Table 25 shows the design of the passive-learning agent.

The agent performed 50 value iterations for each specialized optimal controller before the trials. The agent progressively refines the case-based value function estimates of each specialized optimal controller by performing a value iterations with the linear model of the rover. In each trial the environment remains constant except for the location of the collectible rock, which it is placed in one of the three target with uniform probability. Figure 56 shows the trajectory of the rover in four different instances of the problem. The figure shows the trajectory when the collectible rock (black rectangle) is in the upper-left, lower-left, and upper-right targets. The two graphs (i.e., top-right and bottom-left) show two different trajectories for the same problem instance. This happens because the agent select targets randomly and may not produce the same trajectory in every instance. The bottom-right graph shows a worse case scenario in which the rover decided to approach the wrong target twice. Figure 57 show the evolution of the belief variables for the corresponding trajectories in Figure 56.

The passive-learning agent uses a fixed-learning policy to solve the Mars rover problem. The agent takes advantage of additional information the agent may receive as it moves through the terrain, but it does not select actions to explicitly accomplish both make progress on the task and acquire new information. The specialized optimal controllers are able to direct the rover to the active targets successfully along an optimal trajectory; however, this is useful only when the rover is heading to the target having the collectible rock. This situation may only occur by chance because the rover does not know for sure which target has the collectible rock. The specialized optimal controllers performs better than the schema-based reactive controllers. This occurs because in the latter, there are some situation in which the repulsion force of the obstacle and the attraction force of the goal almost cancel each other producing very small steps and causes the rover to incur in further costs (see, for example, the trajectory of right graph in Figure 54). The passive-learning agent performs better than the non-learning agent when the designers does not know the probable location of the collectible rock a priori. The passive-learning agent does not use a predefined sequence of targets to follow and it is able to update its current belief by collecting additional information as it moves. The top-right graph in Figure 56

Table 25: Design of the Passive-Learning Agent for the Mars rover.

Factor	Description
Variables	5: horizontal position: $h \in [-10\ 10]$ vertical position: $v \in [-10\ 10]$ belief lower-left: $p_{\mathrm{LL}} \in [0\ 1]$; belief upper-left: $p_{\mathrm{UL}} \in [0\ 1]$; belief upper-right: $p_{\mathrm{UR}} \in [0\ 1]$;
FSA	Three states (Figure 55): state 1: target is Lower-Left state 2: target is Upper-Left state 3: target is Upper-Right transitions: stochastic according to beliefs
Decision	Specialized optimal controller: $j \in \{\mathrm{LL}, \mathrm{UL}, \mathrm{UR}\}$ For every action $u_i = U$ at step $(\Delta_\rho, \Delta_\theta)$ $u_i = (\rho_i, \theta_i)$, $U = \{[0, \rho_{\max}], [-\pi, \pi)\}$ $\hat{u}_t^* = \arg\max_{u_i \in U} E_{\hat{z}_{t+1}} \left[\hat{R}^j(\hat{x}_t, u_t) + \gamma \hat{V}^j{}_{w_t}(\hat{s}_{t+1})\right]$ $\Delta_\rho = \frac{\rho_{\max}}{5} = 0.25$ $\Delta_\theta = \frac{2\pi}{16} = 0.3927$ rads.
Sensation	Direct observation of position and velocity: $z_t = H(x_t) = \{h_t, v_t\}$
Perception	Multiple hypotheses perception (Section 7.2.1): h_t and v_t as collected from z_t. $p_t^j = p_{t-1}^j \frac{P(z_t(r) \mid G_j)}{\sum_k P(z_t(r) \mid G_k) p_{t-1}^k}$
Adaptation	Value iteration algorithm: Performed 50 times for each specialized controller. Environment dynamics (linear): $x_{t+1} = \hat{F}(x_t, u_t) = x_t + u_t \Delta_t$ and reward functions: $\hat{r}_{t+1}^j = \begin{cases} 1 & \|(h_R, v_R) - (h_{G_j}, v_{G_j})\| \leq S_G \\ -\mathrm{cost} & \text{otherwise} \end{cases}$

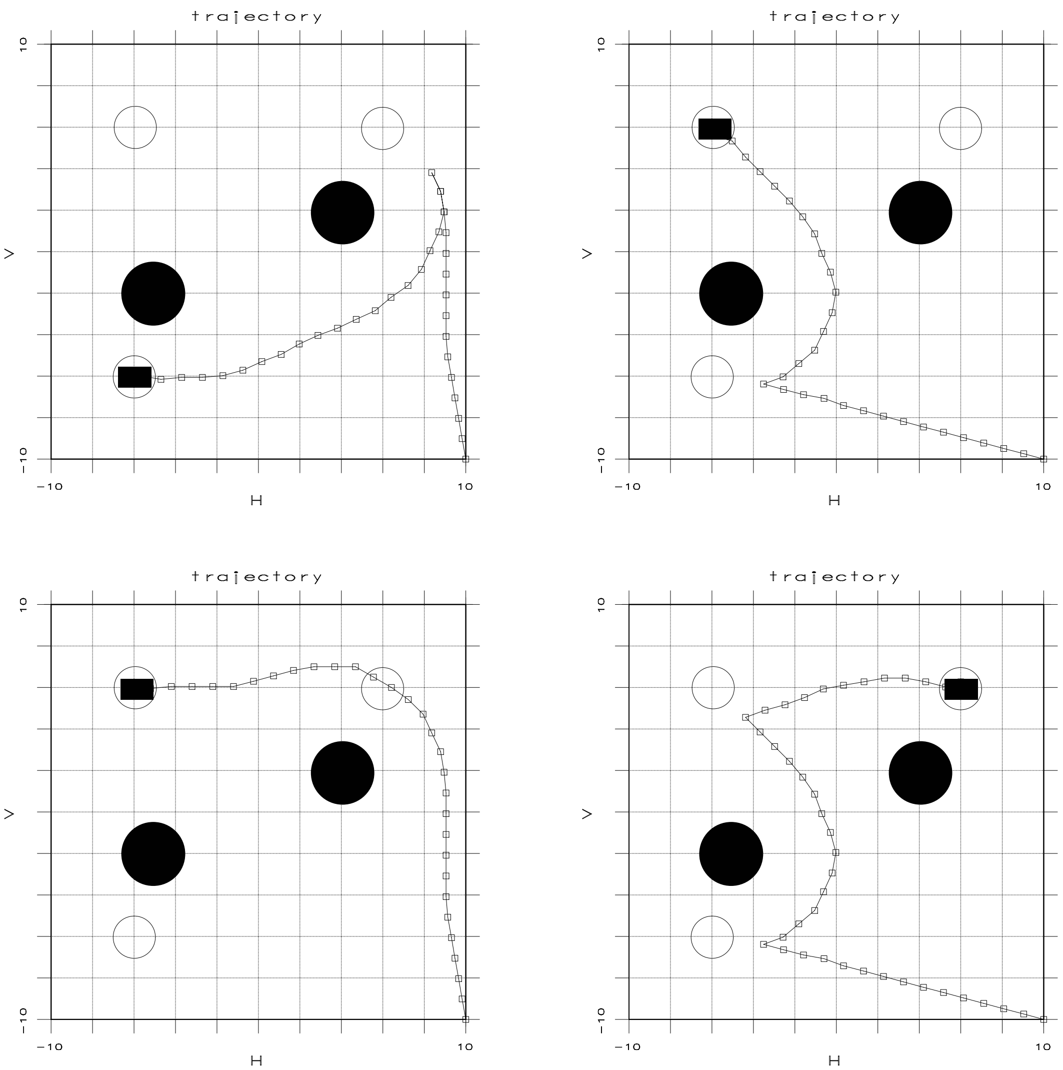

Figure 56: Trajectories for the Passive-Learning Agent.
Trajectory of the rover in four problem instances. The black rectangle indicates the location of the collectible rock. The top-right and bottom-left graphs show two trajectories in the same problem instance. The bottom-right graph shows one of the worse case scenarios in which the rover decided to approach the wrong target twice. Also, this trajectory corresponds to the path that approaches the three targets optimally.

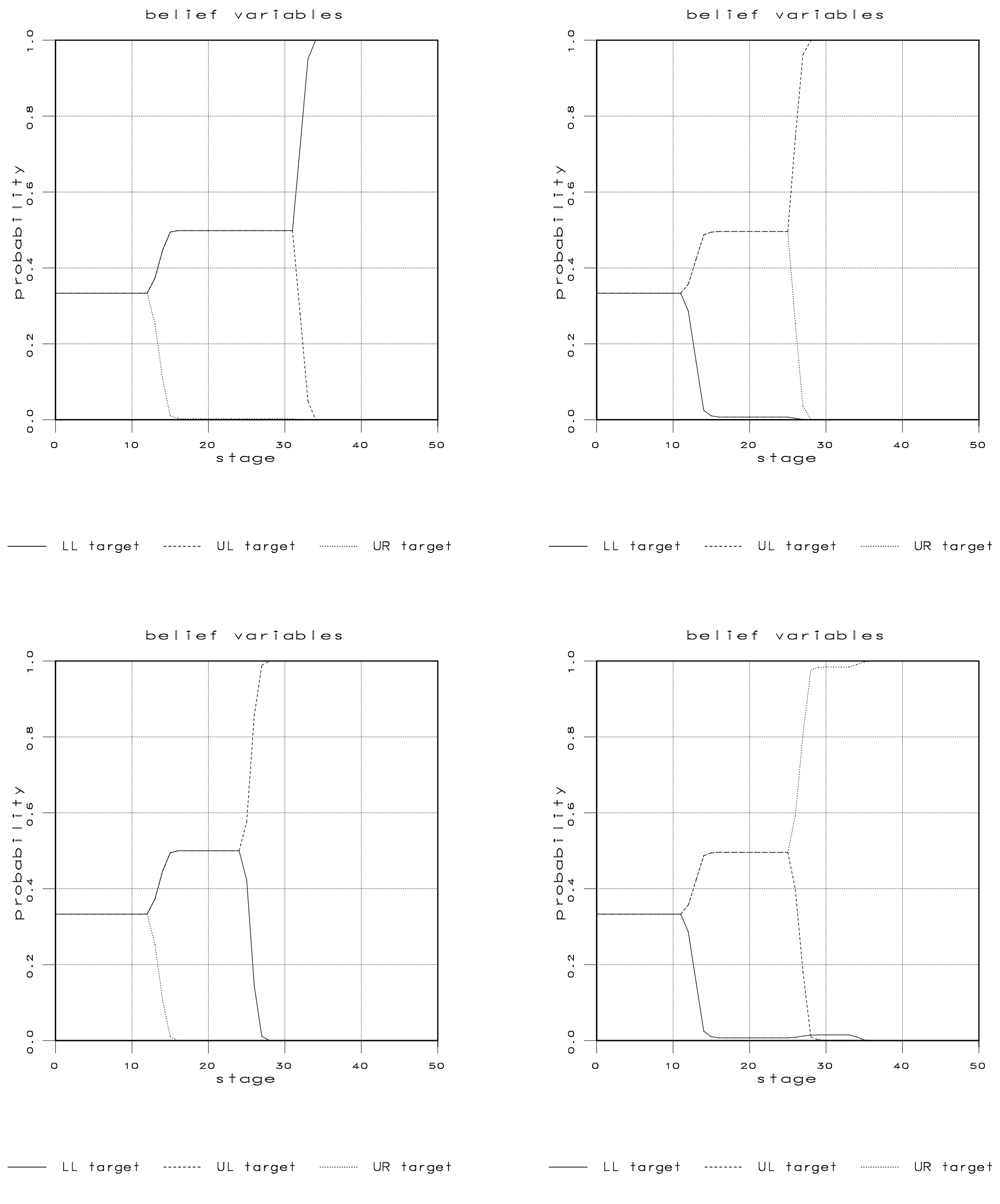

Figure 57: Evolution of Belief Variables in the Passive-Learning Agent. Every graph displays the evolution of the three belief variables as a function of the stage. The graphs correspond to each of the trajectories in Figure 56.

shows an example in which the rover is approaching the lower-left target and then decides to change its course to approach the upper-left target. The rover changed its course before actually reaching the lower-left target. This occurs because the rover is collecting sensations and, as it moves closer to the target, its belief of finding a collectible rock there falls below 5%, which activates perceptual trigger. The passive-learning agent is able to collect and incorporate new information as it moves; however, the agent is not able to select actions taking into account both the progress in the task and the new information each action would produce. The next subsection describes the results of an agent that uses active learning to accomplish the task. The active-learning agent uses the implementation of the heuristic solution described in Section 7.2 to solve the Mars rover problem. This agent is not only able to find the collectible rock but it does so following a trajectory that best balances the control energy for moving the rover closer to the targets and collecting additional information.

7.3.3 Active-Learning Agent

In the previous subsection we presented the results of an experiment in which the agent used a fixed learning policy to solve the Mars rover problem. The agent controls the rover using specialized optimal controllers and modifies its belief according to the information it collects during its trajectory; however, the agent does not select actions taking into account both the progress on the task and the additional information each action may produce. The agent changes the current target to pursue when there is enough evidence of not finding a collectible rock at that target (i.e., $p_j < 0.05$). The next target to pursue is determined randomly by the transitions in the FSA and the target having the greatest belief is the most likely to be active next. In this subsection we describe the results of an experiment in which the agent uses an active-learning policy to solve the Mars rover problem. This agent uses the implementation of the heuristic solution described in Section 7.2; therefore, it is able to select actions considering the immediate progress on the task and the new information each action may produce. The active-learning agent uses an internal state that specify the current position of the rover and the belief of finding collectible rocks at each target. At every stage, the agent select the action leading to the best long-term benefit according to the value function; therefore, the agent does not require the use of an FSA to indicate which target to approach first. This is a significant advantage over the fixed non-learning and passive-learning agents because of the simpler software architecture design.

In the active-learning agent, the internal state of the rover consists of five variables: h, v, p_{LL}, p_{UL}, and p_{UR}. Variables h and v represent the horizontal and vertical position of the rover in an absolute frame of reference. Variables p_{LL}, p_{UL}, and p_{UR} represent the belief of finding a collectible rock at the lower-left, upper-left, and upper-right targets, respectively. The perception function updates the position of the rover directly from every sensation. Additionally, the function uses Bayes' rule to update the belief as described in Section 7.2.1. The function approximator is the case-based and the distance function is multilinear. The adaptation method is value iteration using a linear model of the dynamics of the rover and the sensor model (Figure 51). Table 26 shows the design of the agent.

The agent performed 50 value iterations before the trials. In each trial the environment remains constant except for the location of the collectible rock, which it is placed in one of the three target with uniform probability. The agent starts without knowing how to select actions and progressively refines the case-based value function estimates by performing a value iterations with the linear model of the rover. Figures 58 and 60 shows the trajectory of the rover when it starts with two different sets of beliefs. The former shows the trajectory of the rover when it starts with the belief that the targets have equal probabilities of having a collectible rock. The latter shows the trajectory when the rover starts with the belief that the target at the lower-left corner has 90% probability and the other two targets have 5% probability of having a collectible rock. Figure 59 show the evolution of the belief variables for the corresponding trajectories in Figure 58.

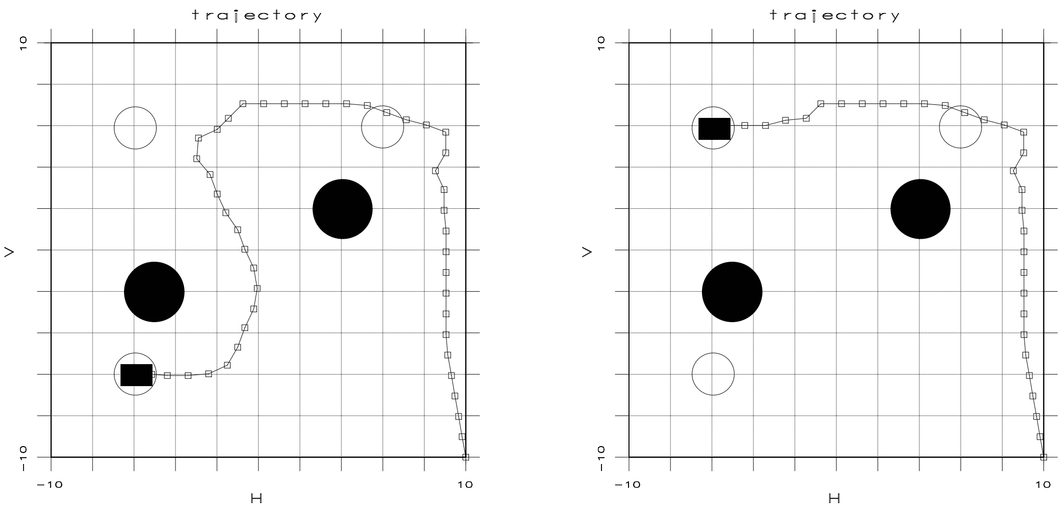

Figure 58: Trajectories for the Rover Starting with Equal Beliefs.
The collectible rock is shown as a rectangle and the markers along the trajectory represent the decision points. The rover starts in the lower-right corner of the terrain with an initial belief of $(1/3, 1/3, 1/3)$, that is, the rover starts with the belief that the targets have equal probabilities of having a collectible rock.

Table 26: Design of the Heuristic Solution Agent for the Mars Rover.

Factor	Description
Variables	5: horizontal position: $h \in [-10\ 10]$ vertical position: $v \in [-10\ 10]$ belief lower-left: $p_{\text{LL}} \in [0\ 1]$; belief upper-left: $p_{\text{UL}} \in [0\ 1]$; belief upper-right: $p_{\text{UR}} \in [0\ 1]$;
Cases	Uniformly distributed in a grid: [7] $21 \times 21 \times 11 \times 11$ 21 points along h. 21 points along v. 11 points along p_{LL}. 11 points along p_{UL}.
Decision	One-step Search: For every action $u_i = U$ at step $(\Delta_\rho, \Delta_\theta)$ $u_i = (\rho_i, \theta_i)$, $U = \{[0, \rho_{\max}], [-\pi, \pi)\}$ $\hat{u}_t^* = \arg\max_{u_i \in U} E_{\hat{z}_{t+1}} \left[\hat{R}(\hat{x}_t, u_t) + \gamma \hat{V}_{w_t}(\hat{s}_{t+1})\right]$ $\Delta_\rho = \frac{\rho_{\max}}{5} = 0.25$ $\Delta_\theta = \frac{2\pi}{16} = 0.3927$ rads.
Sensation	Direct observation of position and noisy sensors (Figure 51): $z_t = H(x_t) = \{h_t, v_t\}$
Perception	Multiple hypotheses perception (Section 7.2.1): h_t and v_t as collected from z_t. $p_t^j = p_{t-1}^j \frac{P(z_t(r) \mid G_j)}{\sum_k P(z_t(r) \mid G_k) p_{t-1}^k}$
Adaptation	Value iteration algorithm (Section 7.2.3): Performed 50 times. Environment dynamics (linear): $x_{t+1} = \hat{F}(x_t, u_t) = x_t + u_t \Delta_t$ and reward function $\hat{R}(\cdot)$ as described in Section 7.2.2 (Equation 74).

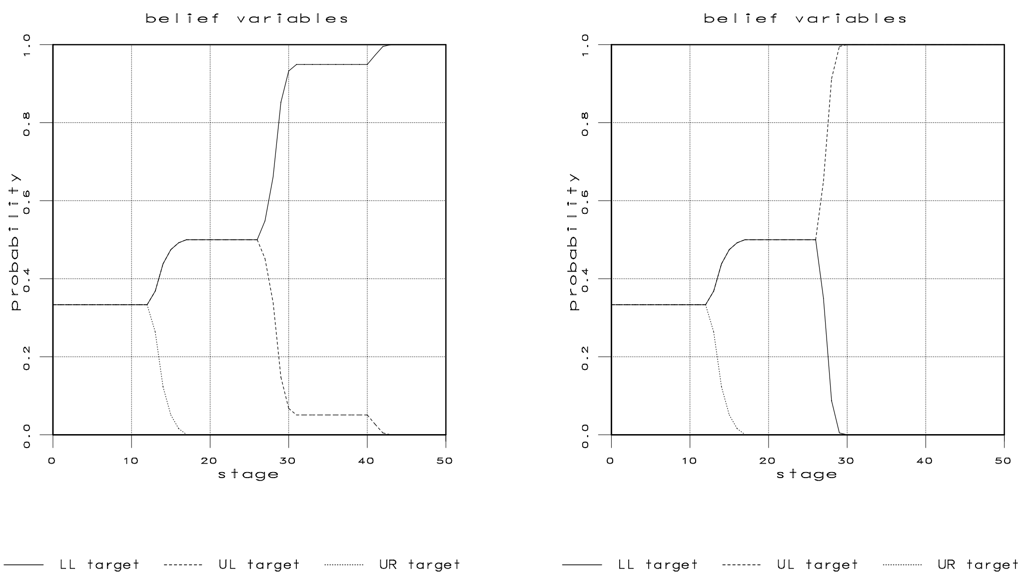

Figure 59: Evolution of Belief Variables in the Active-learning Agent. Every graph displays the evolution of the three belief variables as a function of the stage. The graphs correspond to each of the trajectories in Figure 58.

When the rover starts with equal beliefs, the best strategy is to approach to the upper-right target first. This appears to be counterintuitive to the more natural path of approaching the lower-left target first and then continue to the upper targets (left then right). The reason for this is that the rover is penalized every time it navigates between the obstacles. Additionally, there is 2/3 chance of finding a collectible rock in the upper targets. Therefore, the trajectory that results in the best long-term benefit is the one that heads towards the upper-right target first and then travels between the obstacles only after there is enough evidence that the collectible rock is in the lower-left target. The situation changes when rover starts with different beliefs. Figure 60 shows that the rover does head for the lower-right target first when the initial belief of finding a collectible rock at that target is 90%. As the rover approaches the target, the noisy sensory information provides more and more evidence to decrease the belief up to the point in which the rover decides to head to the upper targets.

Figures 61 and 62 show different cuts of the case-based value function estimates after 50 trials. The value function has four variables as input (h, v, p_{LL}, and p_{UL}) to represent the current position of the rover and the beliefs. The graphs show different cuts of the value function to help visualize the long-term benefit estimates that each input point has in the input space. Figure 61 shows the value function

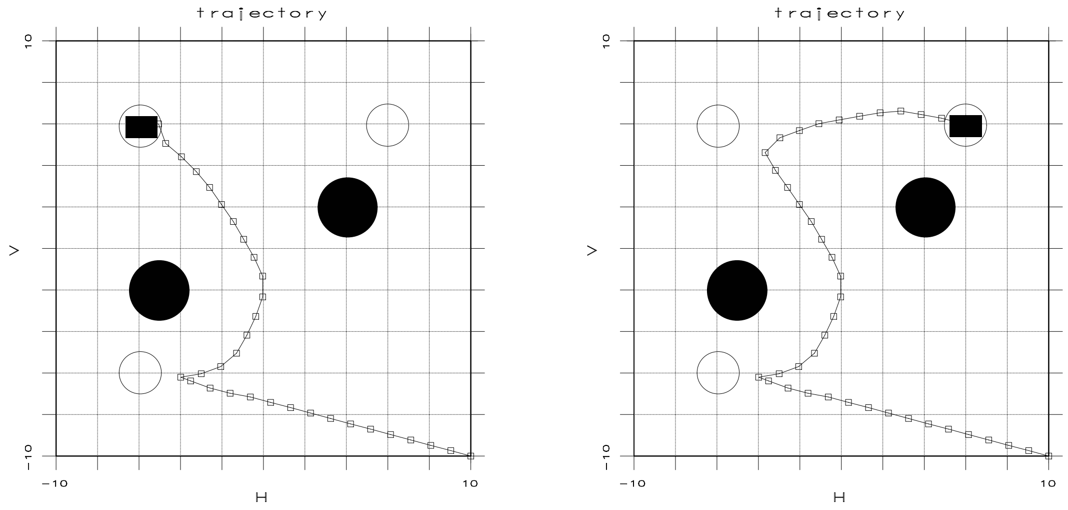

Figure 60: Trajectories for the Rover Starting with 90% Belief in Lower-Left Target. The collectible rock is shown as a rectangle and the markers along the trajectory represent the decision points. The rover starts in the lower-right corner of the terrain with an initial belief of $(0.9, 0.05, 0.05)$, that is, the rover starts with the belief that the target at the lower-left corner has 90% probability and the other two targets have 5% probability of having a collectible rock.

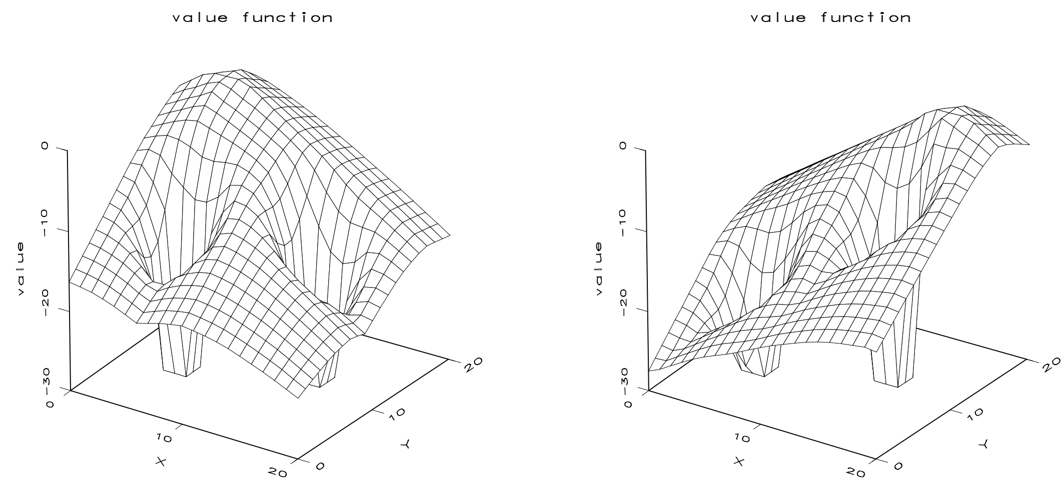

Figure 61: Value Functions in the Internal State Space (Complete Knowledge). Belief probabilities are: (lower-left target, upper-left target, upper-right target). Left: $(0, 1, 0)$, collectible rock in the upper-left target. Right: $(0, 0, 1)$, collectible rock in the upper-right target.

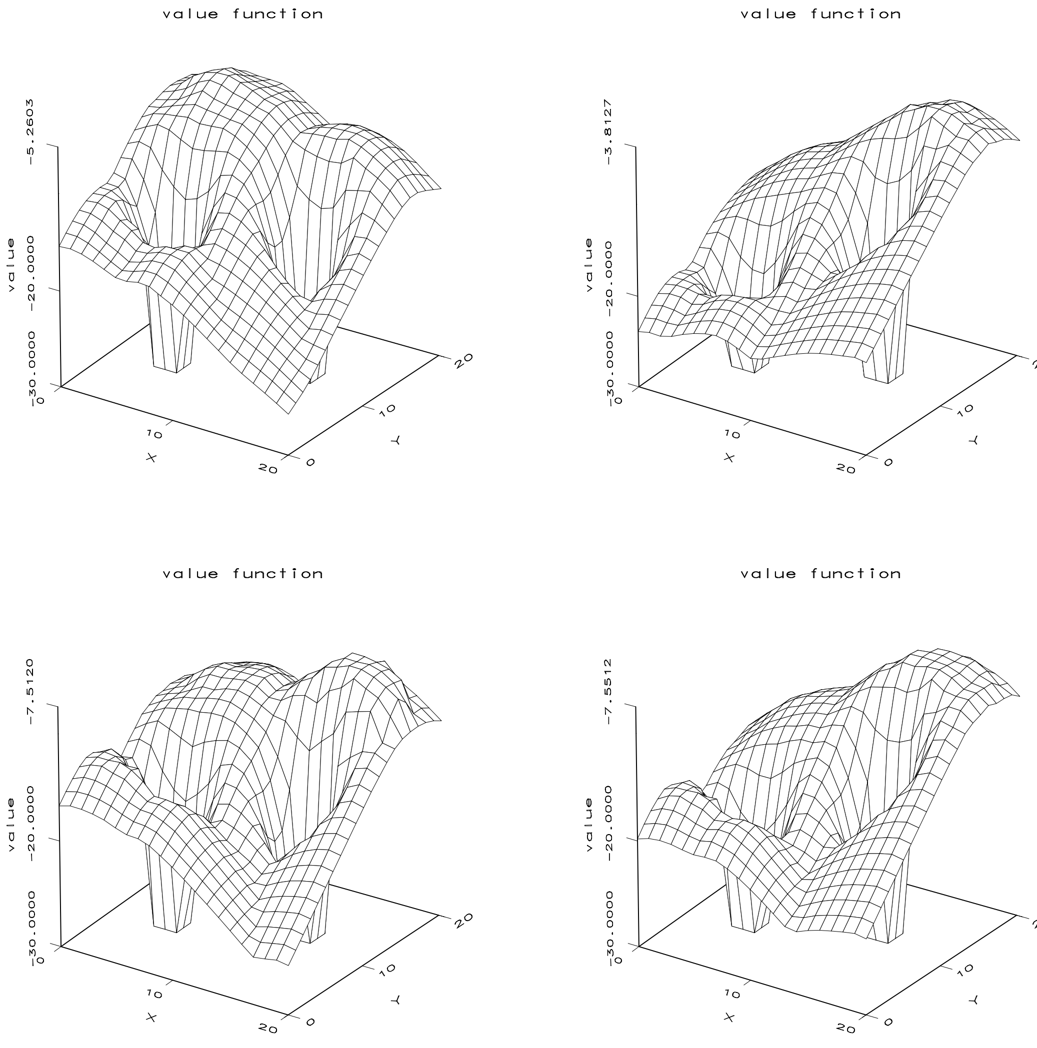

Figure 62: Value Functions in the Internal State Space (Incomplete Knowledge). Belief probabilities are: (lower-left target, upper-left target, upper-right target). Upper left: (.125, .75, .125). Upper right: (.125, .125, .75) Bottom left: (.25, .5, .25). Bottom right: (.25, .25, .5)

when the rover has 100% certainty that the collectible rock is at the upper-left or upper-right target. These cuts of the value function optimally direct the rover to a specific target without exploratory actions because the knowledge level of the rover is already complete. Figure 62 shows other cuts of the value function associated with incomplete knowledge levels. The cuts have different global maxima because there are different long-term benefit estimates associated with each point in the state space. The global maxima coincides with the targets having greater belief. The rover uses the value function to evaluate the outcome of every action at any decision stage. The value function provides an estimate of the long-term benefit associated with every hypothetical internal state that results with the execution of every action. At every stage, the rover selects the action leading to the best sum of immediate reward and long-term benefit estimate. This balances the control energy expended in making progress on the task and collecting new information.

7.4 Discussion

We designed the previous experiments to contrast the performance of the fixed non-learning, passive-learning, and active-learning agents acting on the Mars rover problem. This section discusses in detail the results obtained in those experiments so that we can draw some conclusions about the robustness and effectiveness of each type of agent.

Table 27 shows the final results of the cumulative cost of three agents: fixed non-learning, passive learning, and active learning. The last agent uses all the elements of the heuristic solution to sequential decision-making problems we have proposed in this dissertation. The table shows the cumulative cost each agent incurs when the collectible rock is placed at each of the three targets. The last column shows the mean cost of the task. The lower the value the better the performance.

The fixed non-learning agent is the one that performs worst but has a very simple design. The agent uses a three-state FSA to specify an active assemblage and a schema-based reactive controller to move towards a specific target. Each assemblage consists of a group of motor schemas that run only when their corresponding state is active. The motor schemas in each assemblage are able to move the rover towards a specific target while avoiding obstacles along the way. The agent is able to accomplish the task but it does so without paying any attention to the immediate reward or new information that results from the executing each action. Additionally, the designer must specify a preprogrammed sequence for traversing through every target until a collectible rock is found. In this experiment, the preprogrammed sequence takes the rover to the lower-left target first, the upper-left target second, and the upper-right target last with cumulative costs of 15.8717, 44.9587, 57.2625, respectively.

Table 27: Summary of Results for the Mars Rover Problem.

Agent	Cumulative cost			Mean cost [a]
	LL target	UL target	UR target	
Fixed Non-Learning	15.8717	44.9587	57.2625	39.3643
Passive Learning[b]	32.0218	32.3346	31.8987	32.0850
Active Learning	45.7740	30.7500	16.5003	31.0081

[a] The mean cost is computed as: cost to reach LL target $\times 1/3$ + cost to reach UL target $\times 1/3$ + cost to reach UR target $\times 1/3$

[b] The cumulative cost of the passive-learning agent is stochastic. The mean cumulative cost for each target is computed as a conditional mean, i.e.,

$$\text{cost}_i = E[\,\text{cost}\,|\,i\,] = \text{cost}_i + \sum_{j \neq i} \text{cost}_{ji}\, p_{ji}$$

where i and j represent the targets (i.e., $i, j \in \{\text{LL}, \text{UL}, \text{UR}\}$), cost_i and cost_{ji} represent the cumulative cost to reach target i from the starting point and from target j, respectively. $p_i = 1/3$, and $p_{ji} = 1/6$.

The mean cost incurred by this agent is 39.3743. This agent achieves the best cumulative performance for reaching the lower-left target, which is the closer target to the starting location; however, this target contains a collectible rock only 1/3 of the time. This demonstrates that may not be useful to design an agent that always approaches the closer target first.

The passive-learning agent performs much better that the fixed non-learning agent. It has an internal state consisting of the horizontal and vertical position of the rover as well as the probability of belief of finding a collectible rock in each of the targets. As in the fixed non-learning agent, the passive-learning agent uses a three-state FSA to specify an active target; however, each state in the FSA consists of a specialized optimal controller instead of an assemblage. Each specialized controller is able to direct the rover towards its associated target along a path that minimizes the cumulative cost. Additionally, the transitions in the FSA are stochastic rather than deterministic. The transition probabilities are determined directly by the current values of the belief of the agent. Thus, the agent is not forced to follow a fixed and predefined sequence of targets but the target with the greater chance of containing a collectible rock according to the current belief. The passive-learning agent performs better that than the fixed non-learning agent mainly due to three reasons. First, the agent is able to incorporate new information about the presence or absence of collectible rocks at the targets as it moves. This allows the agent to change its course before actually reaching the active target and avoid further costs caused by approaching an unuseful target. Figure 56 shows examples of this behavior. Second, the transitions in the FSA are stochastic and determined by the current values of the belief of the agent. This increases the chance for the rover to find a collectible rock faster. Third, the specialized controllers take into account the reward the agent receives with the execution of each action. This means that the agent is able to minimize the cumulative cost while it is approaching the active target. The cumulative cost of the passive-learning agent to reach the lower-left, upper-left, and upper-right targets are 32.0218, 32.3346, and 32.0850, respectively. These values are similar because the rover approaches the target having the collectible rock 1/3 of the time, on average. The mean cost is 31.0081, which is close to the performance of the active-learning agent. This demonstrates that approaching targets along optimal paths is not good enough. The rover must take actions that take into account new information as well as making progress on the task.

The active-learning agent is the one that performs best. This agent has the same internal state as the passive-learning agent; however, at every stage, the agent selects the action that, according to the current internal state, maximizes the expected sum of the immediate reward and the expected value of all the internal state that may result from executing that action. Therefore, the agent takes into account both the progress on the task and the new information each action produces. The agent is able to compromise the cost incurred for taking exploratory actions because the new information will lower the cost in the future. Figure 58 shows two examples in which the agent incurs in greater costs initially by collect additional information as compared with the passive-learning agent; however, the new information pays off later because the agent has greater chance of approaching the target having the collectible rock. For example, the left graph in Figure 58 shows the trajectory of the rover when the collectible rock in at the lower-left target. When the agent realizes that the upper-right target is not useful, it decides to approach the upper-left target before heading towards the lower-left one. The agent incurs in further cost initially following this trajectory (i.e., the top-left graph in Figure 56 shows an optimal path that moves the rover from the upper-right target to the lower-left target taking the rover down rather than up); however, the trajectory enables the rover to explore further the environment and achieve the minimum mean cost. Additionally, the left graph in Figure 59 shows the effect of additional gain of information: at stage 30, the active-learning agent has greater belief of finding a collectible rock at the lower-left target than the passive-learning agent at the same stage. The cumulative costs of the active-learning agent to reach the lower-left, upper-left, and upper-right targets are 45.7740, 30.7500, and 16.5003, respectively. Even though, the cumulative cost of active-learning agent to reach the lower-left target is greater than the ones of the other two agents, this target has a collectible rock only 1/3 of the time; therefore, it pays off to approach the upper targets first because the rover will find a collectible rock at those targets 2/3 of the time.

7.5 Conclusions

In this chapter we described an application of the heuristic solution to a complex problem and demonstrated the feasibility of the solution through an experiment with a simulated Mars rover. The rover mission is to find a collectible rock while minimizing time, energy, and collision with obstacles. The decision at any given stage depends on the progress the rover can achieve on its task while learning more about the rocks at possible targets. We contrasted the performance achieved by an agent using the heuristic solution with the performance achieved by a fixed non-learning agent and a passive-learning agent. In the Mars rover problem, is not always a good strategy to approach the closer target first or move from target to target using trajectories that minimize the cost. The rover must take actions that take into account new information as well as making progress on the task. The agent using the implementation of the heuristic solution accomplishes this objective and contributes to support the results we presented in the previous chapters, namely, the use of our heuristic solution to solve a complex sequential decision-making problem that combines continuous domains, imperfect perception, and incomplete knowledge simultaneously.

CHAPTER VIII

EVALUATING DESIGN DECISIONS

We used a systematic methodology to evaluate the design decisions of the SINS system in Chapter 5. In this chapter we describe the foundations behind the methodology, and discuss each step in detail, so that it can be used in general to evaluate other complex artificial intelligent and machine learning systems. We explore provide further details of this methodology and describe additional applications of the methodology in Santamaría and Ram (1996).

Two important goals in the evaluation of artificial intelligence (AI) systems is to assess the merit of alternative design decisions in the performance of an implemented computer system and to analyze the impact in the performance when the system faces problem domains with different characteristics (Aha, 1992; Cohen and Howe, 1989; Kibler and Langley, 1988). Achieving these objectives enables us to understand the behavior of the system in terms of the theory and design of the computational model, to select the best system configuration for a given domain, and to predict how the system will behave when the characteristics of the domain or problem changes. In addition, for machine learning systems, it is important to evaluate the improvement in the performance of the system with experience (or with learning), to show that this improvement is statistically significant, to show that the variability in performance decreases with experience (convergence), and to analyze the impact of the design decisions on this improvement in performance. This is particularly difficult in implementations of AI learning systems because such systems are typically very complex, as are the tasks and domains in which they operate. However, such an evaluation is essential if one is to go beyond mere empirical observation to the prediction and explanation of empirical results.

In this chapter we describe the methodology we used to evaluate SINS in general so that it can be applied to evaluate other AI systems. The methodology performs the evaluation of an AI system through systematic empirical experimentation over a range of system configurations and environmental conditions, coupled with rigorous statistical analysis of the results of the experiments. This methodology enables us to understand the behavior of the system in terms of the design decisions incorporated into the computational model, to select the best system configuration for a given domain, and to predict how the system will behave in response to changing domain and problem characteristics as it gains experience. Additional experiments performed on SINS in Santamaría and Ram (1996).

The chapter is organized as follows. Section 8.1 gives a brief overview of evaluation in AI learning systems. Section 8.2 describes the evaluation methodology. Section 8.3 concludes by summarizing the implications of the methodology for the evaluation other AI learning systems.

8.1 Overview

Two important characteristics of AI systems are that they are complex and the task domains in which they operate are also complex and often ill-structured. As a result, the behavior of an AI system has many sources for variability which causes any performance measure defined to evaluate this behavior to have variability as well. This in turn makes it difficult to assess the significance of an observed behavior of the system in a specific situation. Similarly, due to the complexity of the system and problem domains, theoretical analysis of the system performance given alternative decisions and domain characteristics, although desirable, is difficult in many cases (Francis and Ram, 1995; Kibler and Langley, 1988). However, straightforward performance curves that show how the performance of a system improves over time are not good enough. Although these curves show that the performance improves on specific test problems, they do not provide useful information about why the system works or how specific design decisions affect the behavior of the system, nor can they be used to predict the behavior of the system under different circumstances.

Ablation or lesion studies can be used to analyze the impact of different system's modules in the performance of the system (Cohen and Howe, 1988; Kibler and Langley, 1988). In such studies, one or more system modules are removed or deactivated to analyze how the performance of the system changes. Although these studies do provide some information about the contribution of different modules to the overall performance of the system, they are based on extreme operating conditions that are often impractical (i.e., one or more modules are set to be either active or inactive). Moreover, design decisions often deal with allocating certain amount of resources to different modules. Due to their nature, ablation studies can only deal with all-or-nothing resource allocation, and cannot be used to investigate resource or other interactions between modules.

As an alternative to these approaches, empirical model fitting and statistical tools can be used to analyze the change in the performance of the system in terms of alternative decisions and domain characteristics. In such analysis, the system is evaluated through systematic experiments designed to test the impact of various design decisions while filtering out undesirable sources of variability. The results of the experiments are then analyzed using statistical tools. Another important characteristic of many AI systems is their ability to learn from experience. In such systems, it is important to verify not only that the performance of the system improves with experience, but also that the variability in the performance of the system when solving new similar problems decreases as the number of problems solved in the past increases. Systematic empirical analysis based on statistical tools can also be used to verify the significance of these behaviors and to identify the sources of variability in the behavior of the system.

This chapter presents an evaluation methodology based on well-known statistical tools that can be used to explicitly analyze the merit of a range of design decisions in the performance of a system. The methodology consists of designing experiments to carefully control the variability in the behavior of the system and to obtain empirical performance data over a range of alternative design parameters, representational choices, algorithmic choices, and domain characteristics. This data is used to construct a mathematical model that relates the change in the performance of the system to design decisions and domain characteristics. The model can be used to select the best system configuration for a given domain, to predict the behavior of the system when the domain characteristics change, and to show how (or whether) performance improves with experience.

8.2 Evaluation Methodology

The proposed evaluation method is shown in Table 28. It consists of five phases: experimental design, data collection, model construction, model validation, and robustness analysis.

Table 28: Systematic Evaluation Methodology.

1.	Experimental Design
2.	Data Collection.
3.	Model Construction.
4.	Model Validation.
5.	Robustness Analysis (including Learning Profiles).

During the experimental design phase, the *factors* that may influence the performance of the system are identified. These factors are classified in two broad groups: *design decisions* and *domain characteristics*. Experiments are designed to measure the performance of the system while systematically varying the factors. In the data collection phase, the experiments are executed and data is collected. During the model construction phase, empirical *models* that relate the design decisions, domain characteristics, and the performance of the system are constructed. In the model validation phase, the assumptions identified during the model construction phase are verified. In this manner, the models can be used to state valid conclusions about the relationship between the system's performance and the factors (i.e., design decisions and domain characteristics). During the robustness analysis phase, the system is tested under different alternatives for the factors to assess the generality of the results. In addition, for a case-based or machine learning system, learning profile curves are constructed to compare the empirical models with traditional learning curves obtained with the system. These profiles further verify the results and also provide an intuitive feel for the changes in the external performance of the system as it gains experience. Finally, the data collection and analysis at each step may raise new questions and new ideas for continuing the evaluation, perhaps with a different focus or using a different set of factors; this results in iterative use of the methodology until the issues of interest are resolved. The following sections discuss each of these phases in more detail.

8.2.1 Experimental Design and Data Collection

Learning AI systems are typically complex in nature and their performance depends on several factors. These factors can broadly be classified into two categories: design decisions and domain characteristics. **Design decision factors** are related to the configuration of the system and often deal with allocating resources to different modules within the system. In learning AI systems, several factors are of interest: *system configuration parameters*, such as the amount of memory resources, thresholds for quality of match or distance metrics for retrieval, and other numerical parameters (such as learning rates); *knowledge representation factors*, such as the choice of representational primitives or the content theory of the domain; and *algorithmic factors*, such as the specific algorithms used for retrieval or adaptation. **Domain characteristic factors** are related to problem descriptions and are used to categorize problems in the domain: for example, the difficulty of the problems or the variability in the kinds of problems that the system might face. An additional factor that is relevant to system performance on a given problem is the amount of prior experience that the system has had. If it is feasible to train the system before it is deployed or to hand-code the appropriate experiences, the amount of prior experience can be viewed as another design decision factor for the system designer to consider.

To understand and optimize the performance of the system, it is necessary to assess the role of each factor in the system's overall behavior. During the first phase of our evaluation methodology, factors of interest are identified and experiments are designed to measure the system's performance for different alternatives or *levels*[1] for each factor. A representative sample of system configurations and problem instances is selected, each one with a different set of alternatives for each factor. Several versions of the system are built, each with a different set of choices for the design parameters, representations, and algorithms.

An *experiment* consists of measuring the performance of each version of the system executing on each set of problem instances. Thus, an experiment requires more than a single run; it requires several runs carried out under different conditions (i.e., different configurations of the system, different representational vocabularies, different algorithms for each module, different environmental configurations, different levels of problem difficulty, different order of problems, etc.). Data is gathered from all runs and analyzed using statistical techniques. This allows us to determine not only which factors influence the behavior of the system but under which circumstances and to what extent. (These relationships are captured mathematically in the model construction phase, discussed later.) This information can be used both to understand why the system worked, as well as to select the best system configuration for a given problem domain.

While designing the experiments, it is important to reduce unwanted sources of variability in the system's performance across runs. This ensures that the empirical model attributes differences in system performance solely to differences between alternative chosen factors to the extent possible. If there are other sources of variability in the performance that were not considered beforehand, the model will lose its accuracy and will not be useful in estimating the best design alternatives for optimum system performance. Unwanted sources of variability usually originate by not considering relevant factors in the model or by having a poor experimental design. For example, if the selected sample of problem instances have different levels of difficulty and if the level of difficulty is not considered as a factor, then a poor experimental design might be to measure the performance of two systems configured with a different set of alternatives on only one set of problem instances. If one version of the system solves the easier problems while the other solves the more difficult ones, it will be difficult to assess whether the difference in system performance is due to the different system configurations or to different difficulty levels. To avoid this problem, the experiments should either *balance* out the runs along the factors (i.e., run all system configurations on problems instances with all levels of difficulty), or *block* out the runs along a specific factor (i.e., run all system configurations on problem instances with only one level of difficulty).

[1] The term *treatment* is also used to denote a level.

The choice about when to balance or to block a specific factor is made by trading off the cost of running experiments against the range of applicability of the results of the empirical model to be constructed. A model is applicable only to the range of problem instances from which it was constructed. Increasing the range of problem instances increases the range of applicability of the model but also increases the number of experiments needed because each version of the system must be run with problem instances that represent the entire range. Due to the fact that factors are often grouped by design decisions and domain characteristics, one practical way to design the experiments is to balance out all the factors related to design decisions and to block out all the factors related to domain characteristics. In this way, a detailed analysis of the merits of the design decisions under specific but representative problem instances can be obtained. Such an analysis allows the selection of the best alternatives for the design decisions considered so that the system's performance is optimum when working under the representative problem instances. Next, during the robustness analysis phase, the generality of the selected "best" system configuration can be studied across different levels of domain characteristic factors.

An approach similar this use of the robustness analysis phase is described by Aha (1992). He proposes an evaluation methodology designed to understand the effect of different domain characteristics in the performance of learning systems and to derive rules that designers can use to decide when to generalize the results obtained from case studies. In contrast, our methodology is designed to understand the effects of the design decisions on the performance of the system, to determine if the results are significant, and furthermore, to analyze the domain characteristics under which the evaluation study is valid. Thus, our methodology lends insight into the theory underlying the system and provides a means to select appropriate design parameters for a given application.

At the end of the experimental design phase, the data collection phase is executed. This simply involves running the different versions of the systems on the chosen problem sets and gathering performance data using appropriate performance metrics.

8.2.2 Model Construction

After the experiments are run and the data collected, a mathematical model is constructed to fit and explain the results. Models that relate system performance and relevant factors (i.e., design decisions and domain characteristics) are useful because they provide information about how each factor influences the performance of the system. Such models can serve many purposes, such as predicting what the performance of the system would be under certain conditions of interest, and selecting appropriate system parameters, representations, and algorithms to configure a system for specific situations. The model can also help explain the behavior of the system and thereby provide insight into the theory underlying the design of the system.

Due to the complexity of learning AI systems and the often ill-structured task domains that they operate in, theoretical models that relate system performance and relevant factors are difficult to construct. Instead, the data collected during experiments can be used to infer an empirical model. Empirical models are mathematical expressions based on experimental data and can be constructed using statistical estimation techniques. The basic idea in constructing a model is to assume that there exists a functional relationship between system performance and the relevant factors. The model is a mathematical expression of this relationship.

An example of a linear empirical model is shown in Equation 75.

$$Y_i = \beta_0 + \beta_1 X_{1i} + \cdots + \beta_k X_{ki} + \varepsilon_i \tag{75}$$

In equation 75, the results of $i = 1, \ldots, n$ experimental runs are assumed to follow the additive relationship expressed above. In the model, Y_i represents the dependent variable or observed performance of the system for each of the n runs, and the X_{ji} represent the independent variable or alternative values of each of the $j = 1, \ldots, k$ factors for each of the n runs. Note that dependent variables (also known as response variables) may be used to measure not only the external performance of the system according to some metric but also internal behaviors or effects that are of interest. The value ε_i represent the *residual* or error incurred by the model in estimating the observed value Y_i given the values of the X_{ji} for each of the runs. Inferential statistical techniques are used to estimate the values of the β_j coefficients for the given sample.

This empirical model is applicable to a wide range of situations. The response variable Y_i must be a numerical quantity representing either a quantitative or qualitative variable that measures the performance of the system. For example, for a robot that navigates autonomously, a quantitative response variable may be the time the robot takes to reach the destination point, whereas a qualitative response variable may be the "smoothness" of the trajectory of the robot on a scale from 1 to 10 as determined by some judge. The independent variables X_{ji} are numerical quantities representing the alternative values of the factors under consideration. These variables may be continuous or categorical. *Continuous variables* represent factors that can be varied continuously, such as the amount of memory given to the system under study (e.g., the maximum size of memory resources) or a numerical threshold parameter (e.g., the minimum value of a parameter that is necessary to allow some operation to be performed). *Categorical variables* represent discrete alternatives, such as different representational vocabularies, different domain theories, or different adaptation algorithms that can be used in various modules of the system.

A widely used inferential statistical technique is regression analysis using least-squared error (LSE) (see, e.g., Neter, Wasserman, and Kutner, 1989). LSE estimates the model coefficients (the β_j's in equation 75) by minimizing the squared sum of the residuals (ε_i) across the sample. The model coefficients represent the influence of each factor ($X_j i$) on the performance of the system (Y_i). LSE estimation is widely used because of its properties; in particular, the Gauss-Markov theorem establishes that LSE estimators for regression models are the best linear unbiased estimators possible (Weld, 1916). Also, there are several statistical software packages available for LSE estimation.

The linear regression model in Equation 75 is very general and can incorporate a wide range of smooth functional relations. For example, ablation studies analyze the partial increment/decrement in the system's performance with the addition/elimination of a system component (Cohen and Howe, 1988). Such analyses can be performed using a regression model in which categorical variables can take on the values 1 or 0 to indicate whether a system component is present or not. A linear regression model can also be used with continuous valued parameters, such as amount of memory. Finally, quadratic terms or other functional forms can be incorporated into the model because the only restriction is that it must be linear in the β_j coefficients. Thus, the model can capture mathematically the impact of individual design decisions on the performance of the system as well as that of interactions between design decisions. Once the model is created, the best set of parameter values can be selected to optimize the performance of the system. When smooth functional relations do not apply, either the domain of the model is too ample and must be reduced or more terms (e.g., quadratic, cubic, etc.) must be added into the model. This follows from the fact that any functional relationship can be approximated by a series of polynomials with any degree of accuracy as the number of terms of the polynomials increases (Munroe, 1963).

A common problem when constructing a model is selecting appropriate independent variables to use from the set of all the possible variables that might be considered. Selecting all possible independent variables in the model may artificially reduce the error between the data and model due to overfitting. One solution to this problem is to consider all the possible subsets of independent variables and select the best model according to a specific criterion. One criterion that is often used is to select the model with the best adjusted multiple coefficient of determination (R^2_{adj}). This coefficient measures the ability of the model to explain the variability of the response variable (Y_i). The greater the R^2_{adj}, the better the model explains the variability of the response variable in terms of the variability of the independent variables. The adjusted multiple coefficient of variation takes into consideration the number of terms (estimated parameters) in the model in such a way that models that include terms to artificially reduce the error have lower R^2_{adj} values.

8.2.3 Model Validation

Any model estimated using an inferential statistical technique relies on a set of assumptions. The validity of the model constructed depends on the extent to which these assumptions hold for a given sample of data. For example, there are two assumptions implied in the additive model of Equation 75. First, the residuals are assumed to have zero mean and constant variance across samples. Second, they are assumed to be independent and normally distributed. When these assumptions do not hold, conclusions derived from the model may not be valid. Deviations from the assumption of the residuals having constant variance might lead to overestimates in the ranges of parameter values. This in turn would cause the model to be inaccurate. Small deviations from the assumption of the residuals being normally distributed usually does not create serious problems, but major departures are of concern since the conclusions derived using the model might be incorrect.

To verify qualitatively that the residuals have constant variance, a plot of the residuals against each independent variable and against the fitted response variable is used. A normal probability plot is commonly used to verify the normality distribution assumption of the residuals.

8.2.4 Robustness Analysis

In the final phase of the methodology, alternative levels for the factors are tried and verified against the model. As suggested in the experimental design phase, the experiments in this phase should focus on the factors that are associated with the domain characteristics. In this way it is possible to analyze the sensitivity of the best system configuration as obtained from the model across different domain characteristics, and to verify the generality of the system across a range of problems.

In addition, for a AI learning system, learning profiles are constructed to further verify the empirical model and to provide an intuitive look at the internal behaviors and external performance of the system as it gains experience. A learning profile is similar to a traditional learning curve, except that it also plots the predicted performance of the system based on the model in order to compare that with the actual performance.

8.3 Conclusions

AI learning systems are typically very complex, and the behavior and performance characteristics of such systems are the result of many interacting factors that originate from the many design decisions that go into building them. These design decisions range from numerical system parameters to the knowledge representation scheme chosen to represent the domain and the algorithms and metrics used for matching, adaptation, and other functional modules of the system. Additionally, the tasks, domains, and problems that these systems have typically addressed are also very complex and have a significant influence on the behavior and performance of the system. A good evaluation must show not only that a system is performing well; it should also inform us about the performance of the system under various conditions and about learning or improvement in performance with experience. Furthermore, the evaluation must provide insight about how various design decisions influence both performance and learning. Such an evaluation provides an in-depth understanding of the behavior of the system. It allows the researcher to analyze the theory or computational model based on empirical experiments with the computer program, and the system designer and end user to optimize the configuration of the computer program for a given situation of interest. A better understanding of the behavior of the system across domain characteristics also allows the designers to predict the performance of the system under different situations and to determine the conditions under which the system will perform adequately.

In this chapter we described a systematic evaluation methodology which enables us to obtain these evaluation objectives for implemented computer systems. The methodology involves identifying the design decisions to be analyzed and designing system experiments which systematically exercise the computer program over the range of design options and problem domains of interest. Based on the experimental results, a mathematical model is constructed which can be used to derive statistically significant conclusions and provide the desired insights into the behavior of the system. We used this methodology to evaluate the design decision of SINS in Chapter 5; and further details are available in Santamaría and Ram (1996). However, the methodology can be used to evaluate other AI learning systems as well.

CHAPTER IX

CONCLUSIONS

Research studying decision-making in autonomous agents is at an exciting point in its history. Approaches from several different fields has been combined into a solid mathematical foundation to study decision-making. Researchers from the fields of reinforcement learning, artificial intelligence, optimal control, psychology, machine learning, and operations research have contributed with several different approaches and understanding of the problem. The theoretical research and heuristic approach presented in the first part of this dissertation has contributed to this understanding by providing a uniform framework that combined several of the main ideas in decision-making, specifically, in the context of environments having continuous state and action spaces, and agents having imperfect perception, incomplete knowledge, and limited computational resources. These properties characterize many (if not most) problems of practical interest.

A better understanding and a strong theoretical foundation in the field of decision-making contributes to gaining acceptance of not only among AI researchers interested in building systems that work in real environments, but also among researchers interested in the applications of tomorrow in general, such as, adaptive household robots, multi-purpose software agents residing inside computer networks, adaptive multi-task space exploration robots,[1] etc. The research presented in this dissertation is motivated by considering autonomous agents that are able to evolve as they interact with their environment and perform their task with improved efficiency, better than that they could achieve initially at design time. This research has made a commitment to not predefining the agent's decision-making strategies but letting them evolve progressively and improve with the agent's own experience. The architectures presented in Chapters 4, 5, and 6 represent a first step in tackling the complex issues of achieving sequential decision-making in environments with continuous state and action spaces, and agents with imperfect perception, incomplete knowledge, and limited computational resources.

[1]A better generation of autonomous Mars rovers that can surpass the already astonishing performance of the Sojourner rover.

9.1 Contributions

This section presents a brief summary of the main contributions of this dissertation.

Function Approximators in Continuous Domains:
Agents acting in environments characterized by continuous state and action spaces must be able to generalize the outcomes of individual experiences to other regions of the state-action space. Previous approaches quantized both the state and action spaces or used function approximators with continuous state but quantized actions. In this dissertation, we developed a new approach that uses one function approximator across the combined state-action space, which enables the agent to generalize the outcome of single experiences to other states and actions simultaneously. Two new memory-based function approximators (instance-based and case-based) were described and extensively evaluated on two well-known classical control problems (double integrator and pendulum swing up). The case-based function approximator was also used and evaluated in all the other systems presented throughout the dissertation, including SINS and the Mars rover. Additionally, in this dissertation we developed a new modular technique for implementing function approximators having different degrees of resolution across the state and action spaces.

Implicit State Identification in the Context of Imperfect Perception:
Agents with sensor functions that do not reveal the complete state of the environment face the problem of hidden state. Sensations are noisy and incomplete and limit the ability of the agent to determine the environment's state, which is of paramount importance in decision making. Previous approaches use well-known recursive estimation techniques in control theory; however, such techniques rely on strong assumptions about the nature of the noise and sensor behavior. In this dissertation, we developed a new case-based perception function that the agent can use to implicitly characterize the state of the environment. The perception function uses cases to represent environment situations that consists of regularly occurring sequences of sensation-action pairs. Each case represents a different region of the combined sensation-action spaces and is used to construct a mapping between environment situations and actions. The agent uses and incrementally adapts this mapping to improve task performance. The approach was implemented and extensively evaluated in SINS, an agent that learns to control a mobile robot and self-improve the performance of autonomous robotic reactive navigation.

Learning of Parameter-Adaptive Policies:
Learning autonomous agents are not always required to learn policies mapping states to low-level actions. In general, the set of actions can have a much more abstract relationship to the problem being solved. In this dissertation, we developed and

evaluated a new approach to configure parameter-adaptive reactive control policies. The approach was implemented and evaluated on SINS, in which the agent learned to map environment situations to appropriate configurations of motor-schema based policies.

Incomplete and Dynamic Knowledge:

Agents having incomplete and dynamic knowledge about some unknown properties of the environment face the problem of dual control. In this problem the agent faces the exploration vs. exploitation dilemma: actions must be investigators to a known degree, but also directors to a known degree. Previous approaches to dual control used some type of random exploration (e.g., Boltzmann or exploration bonuses) or structured heuristics that can regulate the learning of the unknown properties (e.g., goal-driven learning approaches). In this dissertation, we developed a new dual control algorithm that has the distinguishing property of efficiently distributing the control energy for learning and control purposes; therefore, able to solve the exploration vs. exploitation dilemma. The control algorithm performs this by selecting actions that accomplish learning but only as required by the performance index. Each action produces an immediate reward, a new state in the environment, and new knowledge level in the agent. At any given stage, the best action is the one that maximizes the sum of the immediate reward and the discounted value of the next state and knowledge level taken together. The approach was implemented in two different scenarios (multiple hypotheses and unknown parameters) and evaluated on the double integrator and Mars rover.

Systematic Evaluation Methodology:

Evaluating design decisions in a learning AI system is difficult because its behavior has many sources of variability. Performance measures defined to evaluate this behavior has variability as well and makes it difficult to assess the significance of a design decision in the performance of the system. Previous approaches consisted of ablation studies or so-called "happy" curves. In this dissertation, we developed a systematic evaluation methodology based on well-known statistical techniques that enables researchers to (1) analyze the theory or computational model based on empirical experiments with the computer program, (2) better understand the behavior of the system across domain characteristics and design decisions, (3) optimize the performance of the system by selecting the most appropriate system configuration. The methodology was used to analyzed the design decisions in SINS, the self-improving autonomous robotic navigation system.

9.2 Future Work

While the heuristic approach we put forward in this dissertation for solving autonomous sequential decision-making in continuous environments has been successful, the research possibilities of this work are by no means at an end. As the range of real-world problems to which sequential decision-making is extended, both the theory and practice of the approach will also have to be extended to deal with the following:

Complex Multi-Task Perception Functions:
Agents equipped with many sensor devices face the problem of *focus-of-attention*: to select and use the most efficient subset of sensors to extract relevant information at a given time such that the agent's computational resources are not overloaded. Research in this area will need to explore and combine more elaborate perception methods that the agent could use to better characterize the state of the environment. Moreover, the heuristic approach must be extended to allow the decision-making mechanisms of the agent to additionally consider as actions, the method of perception the agent should use at each stage. In this way, the agent will have the opportunity to not only modify the external world through its actuators but also decide what perception method it should use to focus its attention and change its internal state.

Better and Faster Adaptation Procedures:
In the heuristic approach, the agent is able to incrementally improve its policy with experience. Additional research will be required to explore alternative computational methods for improving the accuracy of the numerical models used to represent the dynamics functions and adapt the value functions. In the case of model-based implementation, agents would be able to improve the value function in their "spare" time and also perform mental simulations about the outcome of alternative actions. Both factors can help the agent to find better policies. Better and faster adaptation procedures will be required for agents having a high number of degrees of freedom in the perception and action space. The current adaptation methods (e.g., value and policy iteration, temporal difference learning, etc.) are effective only when the agent has 10 or less degrees of freedom. Radically different techniques will be required to handle agents with large (30 or more) degrees of freedom. Research in this area will need to combine and integrate different learning methods into a multistrategy learning decision-making mechanism.

Multiple Agents:
The formulation of sequential decision-making put forward in this dissertation considers the environment as a generic, but unwilled dynamic system. The heuristic approach can be easily extended to incorporate explicit models for "dynamic entities" that compose the environment and correspond to other autonomous agents.

Unknown properties in these models may correspond to unknown styles of behavior associated with other agents. Dual control in this context corresponds to performing actions that better reveal these styles of behavior while making progress on the task. Research in this area will need to explore sequential decision-making in this context and study behavior of cooperative, counteractive, and neutral societies. The formulation of sequential decision-making in multiple agents can provide a mathematical foundation for the meaning of communication among agents and a better understanding of social behavior.

9.3 Conclusions

In this thesis, we set out to study the issues involved in general sequential decision-making problems. We developed a generic mathematical framework that enable us to describe the behavior of an autonomous agent acting in a dynamic environment and to compute the performance according to a task description. The framework provides an unified view to study decision-making problems characterized by environments having continuous domains and agents having imperfect perception and incomplete knowledge. In addition, the framework serves as the basis for the derivation of our heuristic solution, which efficiently solves two of the most important problems in sequential decision-making: accessing hidden state and balancing exploration vs. exploitation. We used the framework to present the theoretically optimal but practically unfeasible solution to the generic problem. We discussed the main characteristics of the optimal solution and developed the heuristic solution, which keeps the main characteristics of the optimal solution but can be implemented in practice. In the heuristic solution the agent uses an internal state to represent the current situation of the environment and the knowledge level. Additionally, the agent uses a value function approximator to represent the long-term benefit estimates associated points in the internal state space. At every decision stage the agent (1) selects the action leading to the best sum of immediate reward and long-term estimate of the internal states believed to result in the next stage, and (2) improves the long-term estimates of the value function approximator based on the outcome. With experience, the agent learns to select actions better and improves it is able to improve its own performance.

Designing autonomous decision-making agents that learn is a fundamental goal. Such agents are able to evolve as they interact with their environment and perform their task with an efficiency that is beyond their initial design. In this work, we have made a commitment for not fixing the agent's decision-making mechanisms but let them progressively evolve and improve with the agent's own experience. In this way, the agent may decide what action to perform next to affect the evolution of the external world, to better characterize its current situation, and to learn more about it. Furthermore, its decision may also change with experience to better adapt to the particulars of the environment and improve task performance. The theory outlined in this work has not only contributed to autonomous sequential-decision making, but also provides several interesting research possibilities to pursue in either physically embodied agents or pure computational (soft) agents.

This work represents some steps towards the objective of designing an autonomous agents capable of learning and adapting themselves to the environment in which they are operating. There is a long way to go down this path, but we hope that these steps will provide a sound start.

VITA

Juan Carlos Santamaría was born on December 10, 1965, in Caracas, Venezuela (father's name Fredy Jesús Santamaría, mother's maiden name Diana Mercedez Hernández). He received the B.E. degree in Electronic Engineering (Cum Laude) from the Simón Bolívar University in Venezuela in 1989. He co-founded Infotrol, C.A., with Natalia Benatuil in 1988 and work as a development engineer in the design, implementation, and testing of several SCADA (Supervisory Control And Data Acquisition) systems in Houston and Tulsa in the U.S., and in Ciudad Guayana and Caracas in Venezuela. He left Infotrol in 1991 and attended the graduate program at the College of Computing of the Georgia Institute of Technology from September 1991 to September 1997. During the Summer of 1996, he worked as a summer intern for Richard S. Sutton at the Adaptive Network Laboratory of the University of Massachusetts. He received the M.S. in Computer Science in March 1993, the M.S. degree in Industrial Engineering in December 1995, and the Ph.D. in Computer Science, with specialization in Artificial Intelligence and Cognitive Science, in December 1997 from the Georgia Institute of Technology. His research interests include learning and adaptation in autonomous agents, intelligent control, and theories of perception and decision making.

BIBLIOGRAPHY

Agre, P. and Chapman, D. (1987). Pengi: An implementation of a theory of activity. In *Proceedings of the American Association of Artificial Intelligence (AAAI-87)*, Vol. 1, pp 268–272. Morgan Kaufmann.

Aha, D. (1992). Generalizing form case studies: A case study. In *Proceedings of the Ninth International Conference of Machine Learning*, pp 1–10, Aberdeen, Scotland.

Albus, J. S. (1975). A new approach to manipulator control: The cerebellar model articulation controller (cmac). *Journal of Dynamic Systems, Measurement, and Control*, 97(3):220–227.

Arkin, R. C. (1989). Motor schema-based mobile robot navigation. *International Journal of Robotics Research*, 8(4):92–112.

Atkeson, C. G. (1991). Memory-based learning control. In *Proceedings of the 1991 American Control Conference*, Vol. 3, pp 2131–2136, Boston, MA.

Atkeson, C. G. and Santamaría, J. C. (1997). A comparison between direct and model-based reinforcement learning. In *Proceedings of the International Conference on Robotics and Automation*, Albuquerque, NM. accepted.

Balch, T., Boone, G., Collins, T., Forbes, H., MacKenzie, D., and Santamaría, J. C. (1995). Io, ganymede, and callisto - a multiagent robot janitorial team. *AI Magazine*, 16(2):39–51.

Bar-Shalom, Y. (1990). Stochastic dynamic programming: Caution and probing. *IEEE Transactions on Automatic Control*, AC-26(5):216–224.

Barto, A. G., Sutton, R. S., and Anderson, C. W. (1983). Neuronlike elements that can solve difficult learning control problems. *IEEE Transactions on Systems, Man, and Cybernetics*, 13:835–846.

Bellman, R. (1957). *Dynamic Programming.* Princeton University Press, Princeton, NJ.

Bertsekas, D. P. (1995a). *Dynamic Programming and Optimal Control*, Vol. 1. Athena Scientific, Belmont, MA.

Bertsekas, D. P. (1995b). *Dynamic Programming and Optimal Control*, Vol. 2. Athena Scientific, Belmont, MA.

Bhatia, N. P. (1967). *Dynamical systems: stability theory and applications.* Springer-Verlag, New York, NY.

Birkhoff, G. D. (1927). *Dynamical systems.* American Mathematical Society, New York, NY.

Bradtke, S. J. (1994). *Incremental Dynamic Programming for On-Line Adaptive Control. Ph.D. thesis*, Department of Computer Science, University of Massachusetts, Amherst, MA. CMPSCI Technical Report TR-94-62.

Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, RA-2(1):14–23.

Brunswik, E. (1956). *Perception and the representative design of experiments.* University of California Press, Berkely, CA, second edition.

Brunswik, E. (1957). Scope and aspects of the cognitive problem. In Gruber, H., Jessor, R., and Hammond, K. (eds.), *Cognition: The Colorado Symposium.* Harvard University Press, Cambridge, MA.

Bryson, A. E. and Ho, T. O. (1975). *Applied optimal control: optimization, estimation, and control.* Halsted Press, New York, NY.

Cassandra, A. R., Kaelbling, L. P., and Littman, M. L. (1994). Acting optimally in partially observable stochastic domains. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pp 1023–1028, Seattle, WA. AAAI Press.

Cishosz, P. (1996). Truncating temporal differences: on the efficient implementation of td(λ) for reinforcement learning. *Journal of Artificial Intelligence Research*, 2:287–318.

Cohen, P. R. and Howe, A. E. (1988). How evaluation guides ai research. *AI Magazine*, 9(4):35–43.

Cohen, P. R. and Howe, A. E. (1989). Evaluation and case-based reasoning. In *Proceedings of the Case-Based Reasoning Workshop*, pp 162–167, Clearwater, FL.

Cohn, D., Atlas, L., and Ladner, R. (1994). Improveing generalization with active learning. *Machine Learning*, 15(2):201–221.

Cohn, D. A., Ghahramani, Z., and Jordan, M. I. (1995). Active learning with statistical models. In Tesauro, G., Touretzky, D., and Leen, T. (eds.), *Advances in Neural Information Processing Systems 7*, pp 705–712. MIT Press, Cambridge, MA.

Dietterich, T. G. (1986). Learning at the knowledge level. *Machine Learning*, 1:287–316.

Effken, J. A. and Shaw, R. E. (1992). Ecological perspectives on the new artificial intelligence. *Ecological Psychology*, 4(4):247–270.

Fel'dbaum, A. A. (1965). *Optimal Control Systems*. Academic Press, New York, NY.

Francis, A. G. and Ram, A. (1995). A comparative utility analysis of case-based reasoning and control-rule systems. In *Eight European Conference on Machine Learning (ECML-95)*, Crete, Greece.

Gibson, J. J. (1979). *The ecological approach to visual perception*. Houghton Mifflin, Boston, MA.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1968). A formal basis for the determination of minimum cost paths. *IEEE Transactions on Systems, Man, and Cybernetics*, SSC-4(2):100–107.

Hart, P. E., Nilsson, N. J., and Raphael, B. (1972). Correction to "a formal basis for the determination of minimum cost paths". *SIGART Newsletter*, 37:28–29.

Howard, R. A. (1966). *Dynamic Programming and Markov Processes*. MIT press, Cambridge, MA.

Kaelbling, L. P. (1990). *Learning in embedded Systems. Ph.D. thesis*, Department of Computer Science, Stanford University, Stanford, CA. Technical Report TR-90-04.

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

Kanerva, P. (1993). Sparse distributed memory and related models. In Hassoun, M. H. (ed.), *Associative Neural Memories: Theory and Implementation*, Chapt. 3. Oxford University Press, New York, NY.

Kibler, D. and Aha, D. W. (1989). Instance-based prediction of real-valued attributes. *Computational Intelligence*, 5(2):51–57.

Kibler, D. and Langley, P. (1988). Machine learning as an experimental science. In *Proceedings of the Third European Working Session on Learning*, pp 81–92, Glasgow, UK.

Kohonen, T. (1990). The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480.

Kolodner, J. L. (1993). *Case-Based Reasoning.* Morgan Kaufmann, San Mateo, CA.

Langer, D., Rosenblatt, J. K., and Hebert, M. (1994). A behavior-based system for off-road navigation. *IEEE Journal on Robotics and Automation*, 10(6):776–783.

Larson, R. E. (1967). *State increment dynamic programming.* American Elsevier, New York, NY.

Lin, L. J. (1992). Self-improving reactive agents based on reinforcement learning. *Machine Learning*, 8(3-4):293–321.

Maes, P. (1990). Situated agents can have goals. *Robotics and Autonomous Systems*, 6:49–70.

Mahadevan, S. and Connell, J. (1991). Scaling reinforcement learning to robotics by exploiting the subsumption architecture. In *Proceedings of the Eight International Workshop on Machine Learning*, Vol. 1, pp 328–332. Morgan Kaufmann.

Martin, J. J. (1967). *Bayesian decision problems and markov chains.* Wiley, New York, NY.

McCallum, R. A., Tesauro, G., Touretzky, D., and Leen, T. (1995). Instance-based state identification for reinforcement learning. In Tesauro, G., Touretzky, D., and Leen, T. (eds.), *Advances in Neural Information Processing Systems 7*, pp 377–384. MIT Press, Cambridge, MA.

Meier, L. (1965). Combined optimal control and estimation. In *Proceedings of the 3rd Annual Allerton Conference on Circuit and System Theory.*

Moore, A. W. and Atkeson, C. G. (1995). The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, 21(3):199–233.

Munroe, M. E. (1963). *Modern Multidimensional Calculus.* Addison-Wesley, Massachusetts.

Narendra, K. S. and Annaswamy, A. M. (1989). *Stable Adaptive Systems*. Prentice Hall, Englewood Cliffs, NJ.

Neter, J., Wasserman, W., and Kutner, M. H. (1989). *Applied Regression Models*. Prentice Hall, Englewood Cliffs, NJ.

Ogata, K. (1990). *Modern Control Engineering*. Pretince Hall, Englewood Cliffs, NJ.

Parr, R. and Russell, S. (1995). Approximationg optimal policies for partially observable stochastic domains. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pp 1088–1094, Montreal, Quebec, Canada. Morgan Kaufmann.

Peng, J. (1993). *Efficient Dynamic Programming-Based Learning for Control. Ph.D. thesis*, Department of Computer Science, Northeastern University.

Peng, J. and Williams, R. J. (1994). Incremental multi-step q-learning. In Cohen, W. W. and Hirsh, H. (eds.), *Machine Learning: Proceedings of the Eleventh International Conference*, pp 189–195, Aberdeen, Scotland. Morgan Kaufmann.

Porter, B., Bareiss, R., and Holte, R. C. (1990). Concept learning and heuristic classification in weak-theory domains. *Artificial Intelligence*, 45(1-2):229–263.

Raiffa, H. and Schlaifer, R. (1972). *Applied Statistical Decision Theory*. MIT Press, Cambridge, MA.

Ram, A. (1993). Indexing, elaboration, and refinement: Incremental learning of explanatory cases. *Machine Learning*, 10(3):201–248.

Ram, A., Arkin, R. C., Moorman, K., and Clark, R. J. (1992). Case-based reactive navigation: A case-based method for on-line selection and adaptation or reactive control parameters in autonomous robotic systems. Technical Report GIT-CC-92/57, College of Computing, Georgia Institute of Technology, Atlanta, GA.

Ram, A. and Leake, D. B. (1995). Learning, goals, and learning goals. In Ram, A. and Leake, D. B. (eds.), *Goal-Driven Learning*, Chapt. 1. MIT Press, Cambridge, MA.

Ram, A. and Santamaría, J. C. (1997). Continuous case-based reasoning. *Artificial Intelligence*, 90(1-2):25–77.

Richards, R. J. (1979). *An Introduction to Dynamics and Control*. Longman, New York, NY.

Ross, S. M. (1993). *Introduction to Probability Models.* Academic Press, Boston, MA.

Rummery, G. A. and Niranjan, M. (1994). On-line q-learning using connectionist systems. Technical Report CUED/F-INFEG/TR66, Cambridge University Department.

Russell, S. J. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach.* Prentice Hall, Englewood Cliffs, NJ.

Santamaría, J. C. and Ram, A. (1996). Systematic evaluation of design decision in case-based reasoning systems. In Leake, D. (ed.), *Case-Based Reasoning: Experiences, and Future Directions*, Chapt. 12. AAAI Press, Menlo Park, CA.

Santamaría, J. C. and Ram, A. (1997a). Learning adaptive reactive controllers. Technical Report GIT-CC-97-05, College of Computing, Georgia Institute of Technology, Atlanta, GA.

Santamaría, J. C. and Ram, A. (1997b). Learning of parameter-adaptive reactive controllers for robotic navigation. In *Proceedings of the World Multiconference on Systemics, Cybernetics, and Informatics*, Caracas, Venezuela.

Santamaría, J. C. and Ram, A. (1997c). A new heuristic approach for dual control. In *Proceedings of the AAAI-97 Workshop on On-line Search*, Providence, RI.

Santamaría, J. C., Sutton, R. S., and Ram, A. (1996). Experiments with reinforcement learning in problems with continuous state and actions spaces. Technical Report UM-CS-1996-088, Department of Computer Science, University of Massachusetts, Amherst, MA.

Shanmugam, K. S. (1979). *Digital and Analog Communication Systems.* John Wiley and Sons, New York, NY.

Singh, S. P. and Sutton, R. S. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22:123–158.

Sondik, E. J. (1971). *The Optimal Control of Partially Observable Markov Decision Processes. Ph.D. thesis*, Department of Computer Science, Stanford University, Stanford, CA.

Sorenson, H. W. (1966). Kalman filetering techniques. In Leondes, C. (ed.), *Advances in Control Systems*, Vol. 3. Academic, New York, NY.

Stengel, R. F. (1994). *Optimal Control and Estimation.* Dover Publications, Mineola, NY.

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 *Machine Learning: Proceedings of the Seventh International Conference*, pp 216–224, New Brunswick, NJ. Morgan Kaufmann.

Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In Tesauro, G., Touretzky, D., and Leen, T. (eds.), *Advances in Neural Information Processing Systems 8*, pp 1038–1044. MIT Press, Cambridge, MA.

Tesauro, G. (1995). Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–67.

Tham, C. L. (1995). Reinforcement learning of multiple tasks using a hierarchical cmac architecture. *Robotics and Autonomous Systems*, 15(4):247–274.

Tse, E., Bar-Shalom, Y., and Meier, L. (1973). Wide-sense adaptive dual control for nonlinear stochastic systems. *IEEE Transactions on Automatic Control*, AC-18(2):98–108.

Tsitsiklis, J. N. and Van Roy, B. (1997). An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690.

Watkins, C. J. C. H. (1989). *Learning from Delayed Rewards. Ph.D. thesis*, Univeristy of Cambridge, England.

Weld, L. R. D. (1916). *Theory of Errors and Least Squares.* Macmillan, New York, NY.

Young, P. (1984). *Recursive Estimation and Time-Series Analysis.* Springer-Verlag, New York, NY.