

REINFORCEMENT LEARNING ALGORITHMS FOR
ROBOTIC NAVIGATION IN DYNAMIC
ENVIRONMENTS

By

TRAVIS W. HICKEY

Bachelor of Science

Oklahoma State University

Stillwater, Oklahoma

1999

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2001

REINFORCEMENT LEARNING ALGORITHMS FOR
ROBOTIC NAVIGATION IN DYNAMIC
ENVIRONMENTS

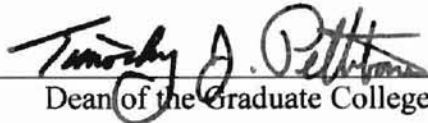
Thesis Approved:



Thesis Advisor

Carl W. Fatsis





Dean of the Graduate College

PREFACE

The purpose of this study was to examine improvements to reinforcement learning (RL) algorithms in order to successfully interact with dynamic environments. The scope of the research was that of RL algorithms as applied to robotic navigation. Proposed improvements were: addition of a forgetting mechanism, feature based state inputs, and hierarchical structuring of an RL agent. Experiments were performed to evaluate the individual merits and flaws of each proposal, to compare proposed methods to prior established methods, and to compare proposed methods to theoretically optimal solutions.

Addition of a forgetting mechanism did improve the learning times of RL agents in a dynamic environment. Direct implementation of a feature-based RL agent did not result in any performance enhancements, as pure feature-based navigation results in a lack of positional awareness, and the inability of the agent to determine the location of the goal state. Inclusion of a hierarchical structure in an RL agent resulted in improved performance, specifically when one layer of the hierarchy included a feature-based agent for obstacle avoidance, and a standard RL agent for global navigation. In summary, the inclusion of a forgetting mechanism, and the use of a hierarchically structured RL agent offer substantially increased performance when compared to traditional RL agents navigating in a dynamic environment.

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation to my academic advisor, Dr. Gary Yen for his guidance and supervision during this work. My appreciation extends to my other committee members Dr. Carl Latino and Dr. Louis Johnson. I would like to thank Dr. Gary Yen and the School of Electrical and Computer Engineering for providing me with this research opportunity.

I would especially like to thank my wife, Tammy, for her love, support, and understanding during times of difficulty. Without her encouragement, this work would not have been completed. Thanks go also to my parents, Jeffrey and Carla Hickey for their support.

TABLE OF CONTENTS

I. INTRODUCTION.....	1
1.1 BACKGROUND.....	1
1.1.1 Reinforcement Learning.....	1
1.1.2 Robotic Navigation.....	3
1.2 RESEARCH GOALS.....	4
1.3 SIGNIFICANCE OF STUDY.....	5
1.4 OUTLINE OF WORK.....	5
II. OVERVIEW OF REINFORCEMENT LEARNING.....	7
2.1 ORIGINS.....	7
2.2 ELEMENTS OF REINFORCEMENT LEARNING.....	7
2.2.1 The RL Agent.....	8
2.2.2 Environment.....	9
2.2.3 Reward Function.....	10
2.2.4 Value Function.....	11
2.2.5 Policy.....	12
2.2.6 Eligibility Traces.....	12
2.2.7 Environment Model.....	13
2.3 APPROACHES.....	13
2.3.1 Dynamic Programming.....	13
2.3.2 Monte Carlo Methods.....	15
2.3.3 Temporal Difference Learning.....	15
2.4 DISADVANTAGES OF RL METHODS.....	16
III. OVERVIEW OF ROBOTIC NAVIGATION.....	18
3.1 ORIGINS.....	18
3.2 APPROACHES AND TECHNIQUES.....	18
3.2.1 Deliberative Navigation Methods.....	19
3.2.2 Reactive Navigation Methods.....	21
3.2.3 Combined Deliberative and Reactive Navigation.....	24
3.3 SUMMARY.....	25
IV. PROPOSED SOLUTIONS.....	26
4.1 INCORPORATING A FORGETTING MECHANISM INTO Q-LEARNING.....	26
4.1.1 Penalty Based Value Function.....	26
4.1.2 Action Selection Policy.....	28

4.1.3 Forgetting Mechanism	28
4.1.4 Summary	29
4.2 FEATURE-BASED REINFORCEMENT LEARNING	29
4.2.1 Motivation	30
4.2.2 Implementation	30
4.2.3 Applications of Feature-Based RL	31
4.3 HIERARCHICAL REINFORCEMENT LEARNING	32
4.3.1 Motivation	32
4.3.2 Implementation	33
4.3.3 Advantages of Hierarchical RL	35
4.4 HIERARCHICAL, FEATURE-BASED REINFORCEMENT LEARNING	36
V. METHODOLOGY.....	37
5.1 IMPLEMENTATION OF REINFORCEMENT LEARNING AGENTS	37
5.2 SUMMARY OF EXPERIMENTS	37
5.2.1 Variation of Parameters	38
5.2.2 Comparison to Established Methods	39
5.2.3 Comparison to Optimal Solutions	39
5.3 KNOWLEDGE TRANSFER EXPERIMENTS	40
5.4 SUMMARY	40
VI. FINDINGS.....	41
6.1 STANDARD Q-LEARNING	41
6.1.1 Variation of Parameters	41
6.2 INCORPORATION OF A FORGETTING MECHANISM	44
6.2.1 Variation of Parameters	44
6.2.2 Comparison to Established Methods	45
6.2.3 Comparison to Optimal Solutions	46
6.3 HIERARCHICAL Q-LEARNING.....	47
6.3.1 Variation of Parameters	49
6.3.2 Comparison to Established Methods	51
6.3.3 Comparison to Optimal Solutions.....	51
6.4 SUMMARY.....	53
VII. SUMMARY AND CONCLUSIONS	54
7.1 SUMMARY.....	54
7.2 DISCUSSION OF RESEARCH FINDINGS	56
7.2.1 Forgetting Q-Learning.....	56
7.2.2 Hierarchical Q-Learning.....	56
7.3 CONCLUSIONS.....	58
7.4 CONTRIBUTIONS TO THE FIELD	60
7.5 FUTURE WORK	60
BIBLIOGRAPHY	63

A. SOURCE CODE	67
A.1 SUMMARY	67
A.2 SOURCE LISTINGS	67
A.2.1 <i>InitEnvironment</i>	68
A.2.2 <i>Random Map</i>	70
A.2.3 <i>OptimalDist</i>	72
A.2.4 <i>InitAgent</i>	74
A.2.5 <i>StartTrial</i>	77
A.2.6 <i>FindMax</i>	79
A.2.7 <i>EGreedy, FGreedy, and HGreedy</i>	80
A.2.8 <i>StepEnv</i>	81
A.2.9 <i>DecodeAction</i>	82
A.2.10 <i>StepAgent</i>	83
A.2.11 <i>ReduceState</i>	86
A.2.12 <i>DecodeState</i>	87
A.2.13 <i>GetArea</i>	88
A.2.14 <i>ShowMap</i>	90
A.2.15 <i>GetGoal</i>	91
A.3 EXAMPLE SCRIPTS	92
A.3.1 <i>Standard Q-Learning Example</i>	93
A.3.2 <i>Forgetting Q-Learning Example</i>	94
A.3.3 <i>Hierarchical Q-Learning Example</i>	95

LIST OF TABLES

Table	Page
6.1. Variation of Parameters for a Standard Q-Learning Agent	42
6.2. Variation of Parameters for Forgetting Q-Learning	45
6.3. Variation of Parameters for Hierarchical Q-Learning	49

LIST OF FIGURES

Figure	Page
2.1. A Simple RL Algorithm.....	8
2.2. A Gridworld Environment	10
4.1 Architecture of a Feature-Based RL Agent	31
4.2 Architecture of a Simple Hierarchical RL Agent	33
6.1 Time to Completion vs. Obstacle Density for Standard Q-Learning Agent.....	43
6.2 Time to Completion vs. Environment Size for Standard Q-Learning Agent	43
6.3 Time to Completion vs. Dynamic Period for Standard Q-Learning Agent	44
6.4 Time to Completion vs. Dynamic Period for Forgetting Q-Learning	46
6.5 Comparison of Standard Q-Learning and Forgetting Q-Learning in a Dynamic Environment.....	47
6.6 Illustration for Hierarchical RL Example	48
6.7 Time to Completion vs. Environment Size for Hierarchical Q-Learning Agent.....	50
6.8 Time to Completion vs. Obstacle Density for Hierarchical Q-Learning Agent	50
6.9 Time to Completion vs. Dynamic Period for Hierarchical Q-Learning Agent.....	51
6.10 Comparison of Hierarchical Q-Learning to Standard Q-Learning with Respect to Environment Size.....	52

6.11 Comparison of Hierarchical Q-Learning to Standard Q-Learning with Respect to Obstacle Density	52
6.12 Comparison of Hierarchical Q-Learning to Standard Q-Learning with Respect to Dynamic Period	53
7.1 Vision-Based RL Robotic Navigation System	61
A.1 Calling Tree for RL Simulations.....	67

NOMENCLATURE

S_t	The current state of the RL agent.
r_t	The reward received by the agent at time t .
a_t	The action taken by the agent at time t .
$V(S)$	The value of being in state S , called the <i>state value function</i> .
$V(S, a)$	The value of taking action a from S , called the <i>action value function</i> .
$\pi(S, a)$	The probability of taking action a from S . Called the <i>policy function</i> .
R_t	The total reward expected beginning from time t . Called the <i>return</i> .
ϵ	Probability of taking a non-greedy action when using a ϵ -soft policy.
α	Learning rate for value function updates.
γ	Weighting factor for the estimated next state in value function updates.
p	State-to-state transition penalty.
$P(S, a)$	The penalty associated with taking action a from S .
μ	State value decay factor used in forgetting.
$O(S)$	Binary function indicating whether an obstacle is present in state S .

CHAPTER I

INTRODUCTION

1.1 Background

This thesis addresses the subject of reinforcement learning (RL) as applied to robotic navigation in a dynamic environment. Prior to discussion of the proposed work, detailed overviews of the fields of RL and robotic navigation will be presented. Following the introductory material, proposals for the enhancement of traditional RL algorithms will be presented and examined in depth. An experimental methodology will be presented that is suitable for analysis of the proposed solutions, and the results of performed experiments will be detailed and analyzed.

1.1.1 Reinforcement Learning

Reinforcement learning (RL) refers to a class of unsupervised machine learning algorithms that seek to maximize a numerical reward signal. Instead of utilizing examples of correct action, as in supervised learning methods, RL methods achieve learning by trying many actions and learning which of those actions produce the most reward. A discussion of supervised vs. unsupervised learning methods can be found in [Huang, 1994]. In the most general case, the reward signal may be delayed, or even time-varying, making credit assignment to actions very difficult. The remainder of this subsection discusses motivations behind the use of RL and typical implementations and applications of RL algorithms. More detailed information will be presented in Chapter II.

As mentioned above, RL refers to a large group of learning algorithms. The common thread between these algorithms is that they all attempt to solve a particular class of problems. These problems are defined as those problems which involve an agent learning to achieve a goal through interaction with its environment [Watkins, 1989]. It can be seen that this definition is very broad; however, certain necessary elements can be extracted from the definition. The elements necessary to any RL algorithm include an agent, its environment, a method of selecting actions, a method of determining the immediate utility of each action, and a method of estimating the long term utility of actions taken [Sutton and Barto, 1998]. In addition, an RL algorithm may include a model of its environment [Whitehead, 1990]. The form and implementation of each element will be discussed in detail in Chapter II of this thesis.

The strength of reinforcement learning is that it does not require explicit examples of correct action, and can thus be applied to systems for which such examples are not readily available. RL methods have been used to train neural networks [Tesauro, 1995], to control dynamic channel assignment in communications networks [Nie, 1999] and to construct fuzzy logic rule bases for fuzzy control systems [Beom, 1995]. It can be seen that RL methods do lend themselves to the solution of many diverse problems. However, there are some limitations to the use of RL. One primary difficulty faced by application designers is that RL methods tend to learn very slowly. This can lead to poor performance in dynamic environments [Coelho, 1998]. Another weakness of RL methods is the tradeoff between exploration and exploitation. Although RL agents are trying to reach a goal as quickly as possible (exploitation), they must also seek to learn more information about their environment in order to enhance future performance

(exploration). The exploration/exploitation dilemma is analogous to the tradeoff between system control and system identification in the field of optimal control [Witten, 1976]. Transference of knowledge from one agent to another is another difficulty when considering RL systems [Malak, 2001]. This is due to the fact that RL is a global learning method that contains all of the information learned about the environment in a single value function. Knowledge gained by an RL agent is very specific to the environment the agent was operating in, and cannot be easily transferred to another agent, even if the environments are very similar. For example, the knowledge gained by an RL agent that learned to drive from Stillwater to Oklahoma City could not be transferred to an agent that was to drive from Stillwater to Tulsa. Even though the problem domains are very similar, and it seems that much general knowledge obtained from one agent could benefit the other, there is no efficient method for transference of that knowledge.

Due to the weaknesses of RL methods, little work has been done to date on the use of RL to directly control navigation of an autonomous robot. RL methods have been used in the development of robotic control systems, as was mentioned earlier, but they have not been used as a method of directly controlling a robot. Typically, RL systems are used to determine the parameters of another type of control system, such as a fuzzy controller [Beom, 1995]. The following subsection discusses methods that have been used in robotic navigation, and the dilemmas faced by these methods.

1.1.2 Robotic Navigation

One of the dominant topics in current mobile robotics research is that of autonomous navigation. As the sensory capabilities of mobile robots expand, devising control systems to efficiently utilize the large amount of sensory data available will become an

increasingly difficult task. Establishing useful relationships between a robot's perception space and control space will be increasingly complex [Davesne, 1999]. Much research has been done towards automatic analysis of sensory features. This work has touched upon many machine-learning techniques, including fuzzy logic [Buschka, 2000], neural networks [Song, 1999], and reinforcement learning [Gaskett, 2000]. Hybrid learning techniques such as neuro-fuzzy control have also been examined [Ng, 1998]. Although there are difficulties associated with the use of RL in a robotic environment, if these difficulties can be overcome RL can provide several benefits. For example, fuzzy logic control systems face the difficulty of determining the fuzzy sets used in the system [Russell and Norvig, 1995]. RL methods require no expert knowledge of the problem domain to implement, and as such are less prone to the difficulties that beset fuzzy logic systems. In addition, as RL methods are unsupervised, they require no examples of correct action to be successful, unlike most neural network training algorithms.

1.2 Research Goals

This thesis proposes methods to allow RL to be used in the direct control of a robot navigating in a dynamic environment. Two primary sub-goals will be considered: to extend RL algorithms so that they are more effective in dealing with a dynamic environment, and to develop methods of transferring knowledge from one RL agent to another.

One key to adaptation in a dynamic environment with a non-stationary fitness landscape is the maintenance of a diversity of possible solutions [Kirley, 2000]. In order to achieve the flexibility to cope with the difficulties a dynamic environment presents, RL methods will be considered that favor exploration over exploitation. This is crucial in dynamic

environments, as exploitation based methods respond very slowly to environmental changes. This reflects upon the first goal above of achieving flexibility in dealing with dynamic environments.

The second goal, of effectively transferring knowledge among RL agents, was chosen as an effective method of combating the slow learning experienced by RL agents. If even partial domain knowledge can be transferred from an experienced RL agent to a less-experienced agent, the second agent would experience the benefit of having to obtain much less information to achieve functionality in the target environment.

When considered together, reaching both of these goals will enable RL methods to be used in direct control of robotic navigation in dynamic environments.

1.3 Significance of Study

To date, reinforcement learning has not been demonstrated as a learning method suitable for handling the intricacies of navigation in a complex dynamic environment. The rate at which RL methods learn is slow enough that RL methods are incapable of dealing with moving obstacles, or even of dealing efficiently with terrain that changes over time. The presented work will show that the proposed extensions to RL methods allow for effective performance in a dynamic environment. The knowledge sharing ability could be applied to multi-agent systems to facilitate collaboration and cooperation in a multi-robot system, such as the unmanned aerial vehicle (UAV) systems used by [Godbole, et. al., 2000].

1.4 Outline of Work

The remainder of this thesis is organized as follows. Chapter II will present an overview of the reinforcement-learning paradigm, as well as an examination of the current related

literature. Following that is an examination of recent work in the field of robotic navigation in Chapter III. Chapter IV proposes extensions to the traditional RL methods. These extensions are designed to meet the research goals presented above. Chapter V details the methods of implementation, data collection, and analysis that will be used to gauge the results of the proposed algorithms. Following this, Chapter VI presents the results of the research performed. Chapter VII contains an analysis of the findings, and a summary of the work performed.

CHAPTER II

OVERVIEW OF REINFORCEMENT LEARNING

2.1 Origins

Reinforcement learning has three distinct roots. The idea of learning from experience originates in behavioral psychology [Thorndike, 1911], [Pavlov, 1927]. The method of dynamic programming, stemming from the field of optimal control [Bellman, 1957], provides a mathematical formalism suitable for learning algorithms. Dynamic programming is discussed in Subsection 2.3.1. Temporal difference (TD) methods [Samuel, 1959], [Klopf, 1975] are often used to solve the temporal credit assignment problem typically associated with RL methods. These methods are discussed in Subsection 2.3.3. Elements of all three of these threads combine to create the modern field of reinforcement learning.

2.2 Elements of Reinforcement Learning

As discussed in Chapter I, the elements common to any RL algorithm are an agent, its environment, an action selection policy, a value function, and a reward function. The RL agent is a discrete-time control system that outputs an action at each time step, and observes feedback from the environment. The environment includes all elements of the RL problem that are external to the agent. The action selection policy, hereafter simply referred to as the policy, is used to determine what action the agent takes from each state. A reward function, considered part of the environment, provides feedback to the agent about the level of success of the agent's actions. A value function is maintained

internally by the agent, and is an estimation of the long-term total reward for a set of actions starting at a specific state. Each of these elements will be discussed in Subsections 2.2.1 through 2.2.5. In addition, two optional features of RL algorithms – eligibility traces and environment models – will be discussed in Subsections 2.2.6 and 2.2.7. Figure 2.1 [Sutton, 1998] shows the architecture of a simple RL problem.

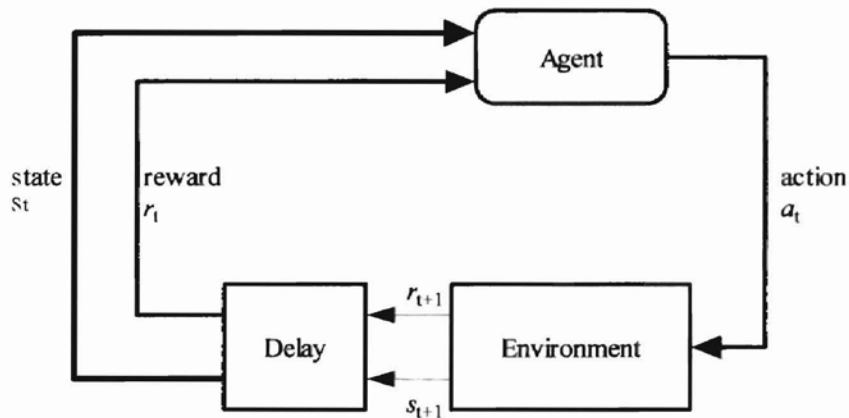


Figure 2.1: A Simple RL Algorithm

2.2.1 The RL Agent

An RL agent is an instantiation of the policy and value functions of an RL algorithm. The agent can be as simple as an element in a simulation, or as complex as a completely autonomous robot. Referring to Figure 2.1, the agent takes as inputs from the environment the state, S_t , and reward, r_t , resulting from the agent's previous action, a_{t-1} . The agent uses these inputs to maintain internally a value function, V , and an action selection policy, π . Using the value function and policy, the agent determines the action to take, which results in another reward and a new state. The goal of the agent is

to update the policy and value function so that the total reward obtained is the maximum possible.

2.2.2 Environment

In the case of a computer simulation, the environment is a representation of the world outside the agent. In this case, the environment includes a reward function, r_t , that provides the agent with information concerning its rate of success. In the case of an agent implemented in hardware, the environment is the actual environment where the agent is located. In this case, the reward function must be implemented as part of the hardware. Formally, the reward function is considered to be part of the environment in either case.

The environment in an RL problem can be classified as one of two types based on the nature of the task to be performed by an agent. An *episodic* environment is an environment in which the agent attempts to achieve a specific goal. When the agent achieves this goal, the episode is over, and a new episode can begin. A *continuing* environment is one in which the agent continuously explores, without a specific goal to achieve. Episodic tasks are similar in nature to indefinite-horizon tasks from the theory of Markov Decision Processes (MDP's). Continuing tasks are related to infinite-horizon tasks. Further discussion of RL in the context of MDP's will be presented in Subsection 2.3.1 of this chapter.

One environment commonly used in RL research is the gridworld [Ono, 1995]. In its simplest form, the gridworld is simply a two-dimensional grid where each location can either be empty or contain an obstacle. The task is usually to traverse the gridworld from a starting location to a goal location in the shortest amount of time while avoiding

obstacles. An RL agent in a gridworld typically has a very simple action set consisting of one action each for moving in the four cardinal directions. A 16×16 gridworld problem is pictured in Figure 2.2. The square marked S indicates the agent's starting state, and the square marked G indicates the location of the goal state. The black squares indicate obstacles that the agent cannot pass through.

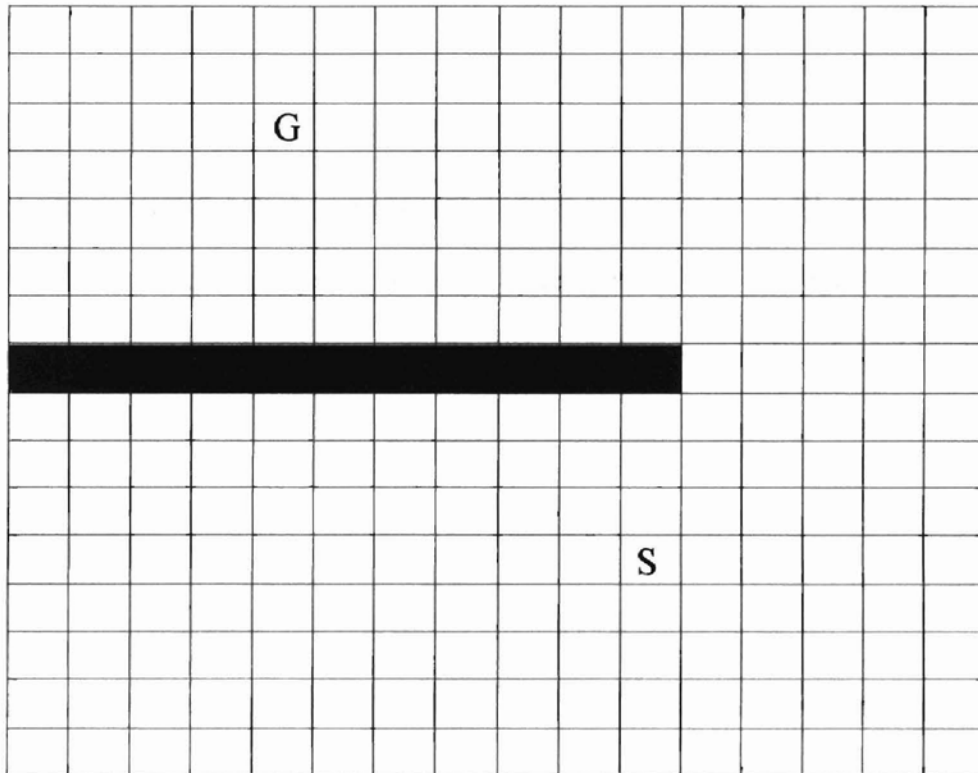


Figure 2.2: A Grid World Environment

2.2.3 Reward Function

The reward function in an RL problem provides feedback concerning the performance of the agent. This function determines the individual rewards r_t given to the agent at each time step. The reward function is designed to provide positive feedback when the agent is performing successfully, and negative feedback when the agent's performance is unsatisfactory. The exact definition of the reward function is domain specific, and must

be tailored to the application being developed. An example reward function for a gridworld environment would be to provide a value of -10 for each action that results in collision with an obstacle, $+10$ for an action that reaches the goal state, and -1 for each other action. This would encourage the agent to find the goal while avoiding obstacles. The -1 penalty on each other action encourages the agent to reach the goal as quickly as possible.

2.2.4 Value Function

Although the reward function provides immediate feedback to an RL agent, the goal of the agent is not simply achieving reward. The goal of an RL agent is to earn the maximum possible cumulative reward. This cumulative reward is referred to as return, R . A discounted return is usually used, in which the values of future rewards are reduced, as in Equation (2.1).

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \quad (2.1)$$

The value γ is a positive constant less than one, and is called the discount factor. The discounted return R_t is used in order to place greater importance on immediate rewards. In order to achieve the maximum possible return, an agent maintains a value function. The value function is an estimate of the return an agent can expect to receive in the future. The value function typically takes one of two forms: a state value function, $V(S)$, or an action value function $V(S,a)$. A state value function estimates the return that an agent can expect to receive starting from the state S . An action value function estimates the return that an agent can expect to receive if starting from state S and taking action a . Most modern RL algorithms learn by updating the value function, although there are

methods that update the policy as well [Santharam, 1997]. Section 2.3 discusses several RL algorithms, and the method of learning for each.

2.2.5 Policy

Policy refers to the method used by an RL agent to determine the most effective action to take based on the value function and its current state. A policy $\pi(S,a)$ is defined as the probability of taking action a from state S . The most commonly used policies are the greedy policy, which simply selects the action that has the highest value function, and the ϵ -soft policy, which selects the greedy action with probability $1-\epsilon$, and selects a random non-greedy action with probability ϵ . The advantage of the ϵ -soft policy is that it is less susceptible to being trapped in local minima. The greedy policy, on the other hand, is optimal if the agent has perfect information about the environment.

2.2.6 Eligibility Traces

One difficulty experienced by RL methods is the problem of assigning credit for rewards to the actions that generated those rewards. The concept of an eligibility trace is used to distribute reward to multiple actions preceding the reward, thereby improving the distribution of reward to the generating actions. In its most simplistic implementation, an eligibility trace is simply a record of the history of actions taken by an RL agent. This is typically implemented by maintaining an array of the same size as the value function, with each element in the eligibility trace corresponding to the same element in the value function. Each time an action is taken from a state, the corresponding state action pair in the eligibility trace is incremented to represent that action being taken. Following each action, the entire eligibility trace is scaled by a factor less than one. When the value

function is updated, the discounted reward value is multiplied by the eligibility trace and added to the value function, rather than only being applied to the most recently visited state or state-action pair. This has the desired effect of distributing rewards (and penalties) to the actions that preceded those rewards.

2.2.7 Environment Model

Some approaches to RL incorporate a model of the environment [Atkeson, 1997] that the RL agent is operating in. The environment model is used to speed up the learning process and to allow for planning of future actions. Learning speed is increased by allowing simulated actions to be taken by the agent before (or between) actual movements [Boone, 1997]. For example, in a path finding application, an agent could evaluate the results of taking a sequence of actions based on the current environmental model. The results of these simulated actions can be used to train the RL agent exactly as if the actions were actually taken. If the environment model is accurate, training times can be dramatically reduced through the use of simulated reinforcement.

2.3 Approaches

There are three main classes of RL algorithms: dynamic programming (DP) methods, Monte Carlo (MC) methods, and temporal difference (TD) methods. Each method has its own benefits and drawbacks, which will be discussed in Subsections 2.3.1 through 2.3.3.

2.3.1 Dynamic Programming

The problem-solving methods referred to collectively as dynamic programming are algorithms that produce optimal policies if given a perfect model of the environment [Bertsekas, 1995]. If the environment can be modeled as a finite MDP, then DP

techniques may be used to generate a policy producing the best-possible actions for the environment. The primary methods used in DP algorithms are: policy evaluation, policy improvement, policy iteration, and value iteration.

Policy evaluation refers to the act of computing the state value function $V(S)$ based on following a policy π . This value function is referred to as $V^\pi(S)$. Policy evaluation is very computation intensive, especially in large environments [Bertsekas, 1998]. Policy improvement is an iterative process of determining a new policy, π' , that provides better performance using the same value function. This is accomplished by evaluation the action value function for each possible action from a given state. The process of evaluating the action value function in this manner is referred to as state backup. Policy improvement has been proven in [Bellman, 1957] to converge to an optimal policy for the current value function. However, there is no guarantee on the speed of the convergence.

By alternating steps of policy evaluation and policy improvement, an optimal policy, π^* , and its corresponding value function, V^* , may be obtained [Howard, 1960]. This process is referred to as policy iteration. A simplified version of policy iteration, called value iteration, works in the same manner, but steps of policy improvement are stopped after a single step, rather than iterating until optimality is reached. Value iteration has substantial computational advantages over policy iteration [Puterman, 1978], and can be shown to converge to an optimum value function if one exists.

The methods of dynamic programming that have been discussed provide a very important theoretical background for the study of modern RL techniques. However, due to the requirement of a perfect model of the environment as a MDP, the use of pure DP

algorithms is very limited. The ideas presented in the discussion of DP will be returned to when temporal difference learning is discussed in Subsection 2.3.3.

2.3.2 Monte Carlo Methods

Monte Carlo (MC) methods approach the RL problem by estimating value functions through experience with the environment. The actual methods used are similar to DP methods, and are referred to in the same manner – policy evaluation, policy improvement, and policy iteration. The difference between DP and MC methods lies in the technique of value estimation, but not in the results or use of the estimated value functions. Where dynamic programming methods use calculations based on MDP theory, Monte Carlo methods rely on averaging returns from following a policy [Barto, 1994]. One weakness of this method is that if a policy does not visit a certain state, the value of that state will never be updated. In order to deal with this disadvantage, the method of off-policy evaluation is often used. In off-policy evaluation, one policy is used for action selection, but the results are used to improve a different policy. In this case, the control policy is typically ϵ -soft, while the policy estimation used for policy iteration is greedy. Although Monte Carlo methods are rarely used in modern RL schemes, the concept of off-policy evaluation is used quite often.

2.3.3 Temporal Difference Learning

Temporal difference (TD) learning combines many of the best features of both dynamic programming and Monte Carlo methods. Like dynamic programming, TD methods are iterative, updating value estimates based on previous estimates. Like Monte Carlo methods, TD methods learn from experience. TD methods possess many advantages

over either dynamic programming or Monte Carlo methods [Holland, 1975]. TD methods do not require an environment model in order to learn, as does DP; they can also be implemented in an on-line manner, unlike MC methods. The most common TD methods in use are Q-Learning [Watkins, 1989] and Sarsa [Sutton, 1996]. The primary difference between the two algorithms is that Q-Learning is an off-policy whereas Sarsa is an on-policy method. Q-Learning will be discussed below. Sarsa is not discussed, as it is very similar to Q-Learning with the exception that Sarsa uses actual values from the resultant state of an action where Q-Learning uses an estimated value of the next state when performing action value updates.

Watkin's Q-Learning algorithm estimates the optimal action value function Q^* regardless of the policy being followed. The only policy requirement for convergence to optimality is that all state-action pairs continue to be visited. The action-value function Q is learned by updating following each action according to Equation (2.2) below. α is the learning rate, which controls how quickly action values are modified. γ is the discounting factor used for determination of return.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha [r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \quad (2.2)$$

2.4 Disadvantages of RL Methods

Although RL methods do provide a framework for efficient unsupervised learning, they are not without drawbacks. One prominent drawback is the amount of training that can be required to achieve efficient behavior. Especially in cases where *a priori* knowledge is unavailable for tuning the design of the agent, learning times can become prohibitively long. This is due to the large number of actions that must be taken in order to learn about the problem domain. Although some methods, such as Q-Learning are proven to

converge to optimal solutions when given enough time to learn, the amount of time necessary is not guaranteed. In fact, the time required to learn correct behavior in an environment is in general exponential in the size of the domain [Bellman, 1957].

Another disadvantage of RL methods, associated with the slow learning times, is difficulty in acting successfully in a dynamic environment. As the environment around the agent changes, the agent must perform further exploration in order to determine the correct actions for interacting with the environment. If the rate of environmental change is faster than the ability of the agent to learn, then the agent will become incapable of finding a strategy that allows correct action.

Finally, RL methods also experience difficulty when attempting to transfer knowledge from one agent to another. This is due to the fact that RL agents incorporate all of their domain knowledge in the value function, typically with very little localization of information. It is very difficult to extract useful information from an agent's value function, as that value function is implicitly dependent on the agent's policy, the internal architecture of the agent, and the details of the surrounding environment.

Although these weaknesses make it difficult to apply RL methods to dynamic environments, the benefits of RL – model free, unsupervised learning – make it worth the effort to counteract the effects of these disadvantages.

CHAPTER III

OVERVIEW OF ROBOTIC NAVIGATION

3.1 Origins

The study of autonomous navigation is a fairly recent field, beginning with several examples of vision based navigation in the late 1980's [Kuan, 1988], [Turk, 1988], [Sridhar, 1988]. These studies primarily focused on the vision system itself, and very little attention was given to the actual navigation systems. Other early work included design of hierarchical robotic control systems [Isik, 1988], development of knowledge-based reasoning systems for navigation [Le Moigne, 1988], and work in sensor fusion for robotics [Mann, 1988]. Many of the concepts developed in these pioneering works have been incorporated into current robotic navigation research.

3.2 Approaches and Techniques

Although many different approaches to autonomous robotic navigation have been developed, in general robotic navigation methods fall into two categories [Ryu, 1999]. *Deliberative*, or planning, methods focus on determining a navigation path before attempting to traverse it, while *reactive* methods rely on reacting to environmental stimuli. Some approaches [Brooks, 1986], [Arkin, 1987] rely entirely on reactive methods, integrating a large number of reactive mechanisms to implement a coherent behavior method for the entire robot. Other efforts [Mitchell, 1987] focus entirely on the planning aspect of behavior. Although these works provide important theoretical

backgrounds that can be built upon to achieve meaningful results, most current research relies on integrating both deliberative and reactive behaviors. Subsections 3.2.1 and 3.2.2 provide individual studies of deliberative and reactive behaviors, respectively, while Subsection 3.2.3 presents methods that have been used to integrate the two types of behavior. The reason for integration of the two types of navigation is to provide the ability to perform large scale tasks that is afforded by deliberative control, yet to also deal robustly with unexpected situations, which is the hallmark of reactive control.

3.2.1 Deliberative Navigation Methods

Deliberative methods have been used extensively for path planning in robotic applications. Early path planning experiments [Klarer, 1990] operated under the assumption of a structured static environment that was completely known prior to undertaking planning. Planning was also performed in an off-line mode, where the entire plan was determined prior to beginning navigation. These simplifications allowed for easy development of basic navigation strategies that could be expanded upon to deal with dynamic and uncertain environments, and to algorithms that could be used in an on-line planning mode.

One navigation method that has received much attention is the method of artificial potential fields (APF). This method is implemented by generating a model of the environment as a potential energy field, and then following the path that represents the steepest descent along that energy field. Most artificial potential field research has focused on reactive, rather than deliberative, behavior, but some applications of APF theory to path planning have been investigated [Warren, 1990], [Wang, 2000], [Ibrahim, 2001]. Warren's approach relies on using another method to generate an initial path, and

then using repulsive forces from obstacles to modify the path. These repulsive forces push the path as far as possible from obstacles while still maintaining the initial origin and destination points of the path. Wang uses a novel approach by modeling the APF after heat transfer equations, rather than the traditionally used electrostatic potential fields. Ibrahim simplifies the problem by using attractive forces towards open spaces, rather than repulsive forces away from obstacles. Although the final result is similar, this approach simplifies the calculations that must be performed, especially in environments containing large numbers of obstacles. One difficulty typically experienced by APF methods is that a large number of local minima exist in the generated energy field. Dynamic programming methods have been applied to APF [Kwok, 1999] in order to eliminate this difficulty. The drawback to this approach is that dynamic programming methods are very computation intensive.

Important recent work focuses on path planning in dynamic or uncertain environments. Two approaches to this problem have been developed. The first approach [Kim, 2001] generates an initial plan, and adapts that plan as necessary to the environment as navigation proceeds. The second approach [Fiorini, 1996] involves planning for the motions of dynamic obstacles in the environment. Although this approach does allow for definitive analysis of behavior with respect to obstacles, it is only applicable to cases where the environment dynamics can be completely specified.

An example of adaptive planning is the re-planning method [Kim, 2001] that uses localization methods to limit the scope of modification to the current plan. By only modifying the smallest possible section of the current plan, subsequent modifications to the remainder of the plan are minimized. Another adaptive planning method [Ferrari,

1997] considers both temporal and spatial modification to the path plan. Ferrari also introduces the use of two planning metrics – plan quality and plan robustness – that can be used to characterize planned navigation paths, and to analyze the impact of variations to a given plan.

Exact planning in a dynamic environment involves the calculation of obstacle paths as well as the path of the robot. The concept of velocity obstacles [Fiorini, 1996] allows for simplification of the required planning by defining a class of dynamic obstacles whose behavior can be analyzed in general terms prior to planning. At the time of actual planning, the precalculated obstacle dynamics can be used, simplifying the actual path planning process. Recent work [Shiller, 2001] extends velocity obstacles to include obstacles with non-linear velocities and arbitrary trajectories.

Deliberative approaches to navigation are capable of dealing with very complex circumstances. However, their weakness is in dealing with unknowns in the environment. These unknowns can take the form of an unexpected obstacle, a malfunction in the robot itself, or some other event that the planning engine was incapable of foreseeing. The next subsection discusses reactive navigation methods, which are very robust when dealing with unexpected events.

3.2.2 Reactive Navigation Methods

Many approaches have been applied to reactive navigation. Artificial potential fields, neural networks, reinforcement learning, and neuro-fuzzy controllers have all been used to generate reactive behavior in autonomous robots. This subsection will explore various methods of reactive navigation.

Artificial potential fields, which were discussed in the context of deliberative behavior in Subsection 3.2.1, have also been used to generate reactive behavior. An APF algorithm based on electrostatic potential fields [Valavanis, 2000], has been demonstrated to generate an approximately optimal path through a general static environment without the use of prior knowledge. This APF approach was implemented in real-time using a mobile robot and was demonstrated to successfully generate collision-free paths.

An extension of the APF method is to use more advanced methods of sensory processing to generate the APF, and to then use traditional APF methods to generate a path. A neural network system capable of integrating inputs from several sensors [Song, 1999] has been used to detect and track moving obstacles, and to predict future states of those obstacles. This multisensor predictor method was integrated with an APF navigation method to generate real-time obstacle avoidance behavior in the presence of moving obstacles.

Reinforcement learning has been used in a limited form to train a neural network designed for reactive navigation [Millan, 1996]. Initially, Millan's controller generates actions based only on a set of "basic reflexes" that are determined *a priori* based on domain knowledge. As the robot navigates the environment, each time a new situation is encountered, the controller generates a new action based on those basic reflexes. When a familiar situation is encountered (i.e. the neural network successfully maps the situation to one of the generated actions), that action is used. Over time, an RL algorithm is used to determine the suitability of each of the generated actions, and to tune the operation of the neural network. Although this controller architecture does learn very quickly, it does have some severe limitations. The primary difficulty is that the size of the neural

network can grow very large when dealing with complex environments. In the extreme, this leads to slowed reaction times as the robot must perform increasingly more complex calculations each time a new situation is encountered. Another difficulty is that new actions may be generated for familiar situations due to incorrect classification because of sensor noise.

Neuro-fuzzy controllers have also been used in reactive navigation research [Ng, 1998]. Ng proposed a three-level neuro-fuzzy controller called Nif-T (Neural Integrated Fuzzy Controller). Nif-T consists of: fuzzy logic membership functions (FMF), a rule neural network (RNN), and an output-refinement neural network (ORNN). The FMF fuzzifies the sensory input data. This fuzzy data is operated on by the RNN, with the output of the RNN being defuzzified. The defuzzified data is then used to train the ORNN level of the controller. Using only a few rules to train the RNN, both wall-following (5 rules) and multi-robot convoying (9 rules) behaviors were implemented. Neuro-fuzzy control has been shown to be very robust; however, like all fuzzy logic based algorithms, it suffers from the requirement that *a priori* expert knowledge be used to generate the FMF.

It can be seen that reactive navigation schemes have been implemented using many different algorithms and architectures. The main weakness that defines reactive navigation schemes is a very poor ability to scale to large problem domains. Although purely reactive schemes may perform very well in small environments, the number of reactive behaviors required to interact successfully with a very large or complex environment is prohibitive to actual implementation. The next subsection will discuss integration of reactive and deliberative navigation methods to implement a comprehensive navigational controller.

3.2.3 Combined Deliberative and Reactive Navigation

There are several approaches to coordinating plan-based and reaction-based behaviors into a coherent control system. For sake of explanation, these approaches will be divided into two basic categories: those methods where reactive behaviors may override planned behavior, and those methods where planning is used to control reactive behaviors.

The first category is typified by Payton's autonomous land vehicle (ALV) control system [Payton, 1990]. In this control system, navigation routes are planned using a map-based planner. Then, the plans are executed using a set of planned behaviors. However, if at any time a possible collision with an obstacle was sensed, reactive behaviors were allowed to override the planned behaviors until the obstacle was cleared. This approach provided acceptable paths in most situations. In some cases, though, unacceptable behaviors were generated due to the interaction of planned and reactive behaviors. For example, when being used to control a car driving on a road, the controller could occasionally cause the car to veer completely off the road in order to avoid an obstacle, even if it were possible to stay on the road and still avoid it. Payton concluded that some level of interaction between the deliberative and reactive behaviors was necessary, but did not specify the exact nature of that interaction.

An example of the second type of integrated navigation is that presented by [Ryu, 1999]. In this case, a topological map based planning engine implements the deliberative navigation element. In addition to route planning, the planner also determines what reactive behaviors should be used based on the robot's current situation. As the robot navigates the planned path, the plan based controller either activates or inhibits each of a set of reactive navigation tools. This approach avoids the difficulties experienced by

Payton, but brings about its own set of difficulties. One possible problem is if a situation arises that the planning engine had no conception of. Even if a reactive behavior exists that is appropriate to deal with the incident, it is possible that at the time that behavior was inhibited. This inability to deal with unplanned events is one prominent weakness of deliberative navigation. In the case of Ryu's work, the incorporation of reactive control would combat that effect to some degree, but it is still possible to suffer from the weakness.

3.3 Summary

This chapter has presented examples of the two main types of robotic navigation, and examples of methods used to combine those two approaches. Examples of both reactive and deliberative navigation were presented, and the strengths and weaknesses of each were discussed. In summary, deliberative, or plan-based, navigation is capable of solving large navigation problems that would be very difficult for reactive approaches to deal with. However, purely deliberative control schemes have no ability to respond to unplanned events or to unknown obstacles. On the other hand, reactive navigation is very competent at obstacle avoidance and at dealing with unexpected occurrences, but is very weak when trying to solve a large problem that planning approaches would have little difficulty with. The combined strengths of the two methods allow for a very robust navigation system, which is why a large amount of research into robotic navigation has focused on integrating the two approaches into a single controller.

CHAPTER IV

PROPOSED SOLUTIONS

This chapter will present several proposed solutions to deal with the difficulties of applying RL based learning systems to autonomous navigation problems in dynamic environments. The three main ideas are: incorporation of a forgetting mechanism into RL, use of feature-based state information in an RL system, and hierarchical structuring of an RL system. In addition, the combination of feature-based and hierarchical RL will be considered.

4.1 Incorporating a Forgetting Mechanism into Q-Learning

One difficulty that stems from interacting with a dynamic environment is that an agent may attempt to make use of knowledge that has become outdated due to the dynamics of the environment. This difficulty is one facet of the exploration vs. exploitation dilemma discussed in Chapter I. In order to mitigate the effects of using outdated knowledge, it is proposed that a forgetting mechanism be incorporated into a penalty-based Q-Learning algorithm. Subsections 4.1.1 through 4.1.3 below detail this proposed algorithm.

4.1.1 Penalty Based Value Function

The proposed learning algorithm is an adaptation of Q-Learning to a deterministic environment. In a deterministic environment, the subsequent state following an action is known, allowing a simplification of the Q-Learning process by storing only values associated with each state, rather than with each state-action pair. This modification reduces the number of state values that must be maintained, consequently resulting in a

more efficient learning algorithm. The state value function that is maintained is a penalty function, which tracks the expected total cost associated with being in a given state.

As with most RL strategies, prior knowledge may be used to effectively initialize the state value function. The method used here is to simply initialize the state-value function to the distance from that state to the goal state. As the agent explores the environment, it learns the penalty associated with each state, which is approximated by the value function for that state. After each time step, the value function for the visited state is updated as per Equation (4.1) presented below.

$$V_s(i+1) = (1 - \alpha) \cdot V_s(i) + \alpha(p + \gamma \cdot V_s(i)). \quad (4.1)$$

The parameter α is the learning rate, which controls how large the updates to the value function are. Typical values of α for this algorithm are 0.05 to 0.2. The parameter γ controls the extent to which the value function update is based on the value of the next state. Large values of γ mean that the value function is very dependent on the value of the following state, while small values of γ mean that the value function update is more dependent on the state to state transition penalty, p .

As the agent explores, the value of states arbitrarily far from the goal will approach a maximum value V_{MAX} , while the value of states arbitrarily near the goal will approach a value V_{MIN} . The value of V_{MAX} can be derived from Equation (4.1) by assuming that, after an infinite period of exploration, all states arbitrarily far from the goal will achieve the same value, V_{MAX} . Substituting into Equation (4.1), we obtain

$$V_{MAX} = (1 - \alpha) \cdot V_{MAX} + \alpha(p + \gamma \cdot V_{MAX}).$$

This equation can be solved to yield $V_{MAX} = \frac{P}{1-\gamma}$. Likewise, assuming that the values of all states arbitrarily near the goal approach the value V_{MIN} , Equation (4.1) can be rewritten

$$V_{MIN} = (1-\alpha) \cdot V_{MIN} + \alpha(p + \gamma \cdot 0).$$

This can be solved for $V_{MIN} = p$.

4.1.2 Action Selection Policy

The action selection policy implemented for this research is very simple, selecting the action a that minimizes the penalty associated with selecting that action. The penalty is defined by a function that evaluates the penalty of taking action a from state S :

$$P(S,a) = 100 \cdot O(S') + V_{S'(i)}, \quad (4.2)$$

$O(S')$ is a binary function, with a value of 1 indicating that an obstacle is present in the resultant state. The scale factor on the obstacle function is chosen to be larger than V_{MAX} , so an action will never be chosen that results in collision with an adjacent obstacle.

As this action selection policy is a greedy policy, it favors exploitation rather than exploration. The forgetting mechanism presented in Subsection 4.1.3 will add exploratory behavior to the algorithm.

4.1.3 Forgetting Mechanism

The forgetting mechanism is implemented as a decay of the state value function:

$$V(S) = \mu \cdot V(S), \quad (4.3)$$

where μ is a positive scalar between zero and one. This is applied to the value function of each state after the conclusion of each episode. For values of μ close to 1, very little forgetting will take place, resulting in an agent very similar to traditional Q-Learning. For values close to zero, almost all penalty will be forgotten between episodes, effectively causing the agent to explore the environment each episode, without any reliance on previously learned knowledge.

4.1.4 Summary

A modified Q-Learning algorithm incorporating a forgetting mechanism has been proposed. The algorithm differs from Q-Learning in that it maintains a state based value function rather than a state-action pair based value function. The algorithm is characterized by a value function update rule (4.1), an action evaluation function (4.2), and a forgetting mechanism (4.3). The goal of this algorithm is to provide enhanced performance in a dynamic environment by utilizing exploratory behavior that maintains a larger set of possible solutions than is kept by traditional Q-Learning.

4.2 Feature-Based Reinforcement Learning

One difficulty involved in the use of RL in large environments is that the number of state values that must be maintained may increase to unmanageable sizes [Bellman, 1957]. In order to reduce the number of state values that must be maintained, a modification to the typical RL structure is proposed. Rather than storing a value for each individual state in the environment, it is proposed that the value function be used to store the value of each of a set of features.

4.2.1 Motivation

The primary motivation for using a feature-based RL algorithm is to reduce the number of state values that must be maintained. This will allow for use of RL agents in much larger environments. Another possible benefit of feature based RL is an increased ability to transfer knowledge from one agent to another. As the state values will be based on some feature of the surrounding environment, rather than raw state data, information about how to deal with certain features could be transferred from one agent to another. In Chapter V, methods of evaluating performance both in terms of number of state values maintained and in terms of ability to transfer knowledge will be discussed.

4.2.2 Implementation

Two approaches to the implementation of a feature-based RL algorithm will be discussed. The first approach is a simple encoding of the agent's immediate environment, while the second approach involves more sophisticated methods.

The simplest approach to implementing a feature-based RL algorithm is to directly encode the area of the environment surrounding the agent. For example, consider the gridworld environment discussed in Chapter II. If the 3×3 square area surrounding the agent's location was considered the current environmental feature, then the number of possible features would be 256 – eight squares, each of which could either contain an obstacle or be empty. Although this encoding method is very rudimentary, it could conceivably be used in simple environments.

A more advanced approach would be to use a feature recognition tool, such as a neural network, to process feature information. The neural network suggested is a multi-layer

perceptron (MLP) trained to recognize environmental features prior to inclusion in the RL agent. The input to the neural network would be the area surrounding the agent, and the output would be an index representing the current state to be used by the RL agent. An example would be to train a MLP to recognize vertical walls, horizontal walls, open areas, and the goal. A block diagram illustrating the operation of this simple feature based RL agent is shown below in Figure 4.1.

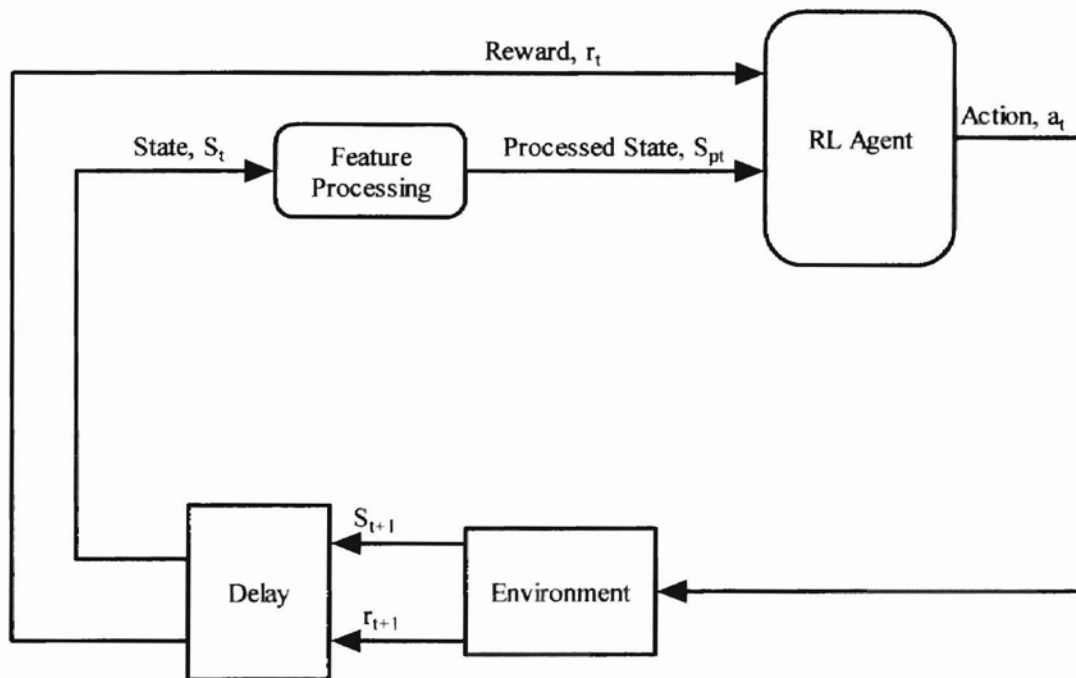


Figure 4.1: Architecture of a Feature-Based RL Agent

4.2.3 Applications of Feature-Based RL

One possible application of feature-based RL is as the obstacle avoidance component of a robotic navigation system. By using a feature-based RL system designed to recognize potential obstacles, an agent could learn through interaction the most successful way to deal with each obstacle present in the environment. Another related application would be for landmark recognition in a navigation system. As the RL agent learned the best path,

the feature-based RL would enable learning of landmarks that were on or near the best path.

One difficulty of feature-based RL is that, without modification, it provides no way of allowing the agent to know its current position in the environment. As such, it is unsuitable for direct use in most RL applications. However, when it is incorporated into a system that utilizes other methods for knowledge of position, feature-based RL can add the benefits of knowledge transference and reduced state complexity. This topic will be covered in greater detail in Section 4.3, in the discussion of hierarchical RL. As the applicability of purely feature-based RL is very limited, feature-based RL will not be considered in experiments except as a component of a hierarchical RL system.

4.3 Hierarchical Reinforcement Learning

In order to further increase the ability of RL agents to deal with a dynamic environment, a two-level hierarchical reinforcement-learning scheme is proposed. This is postulated to increase performance through two specific effects. First, by reducing problem complexity, in the spirit of divide and conquer. Second, by allowing increased knowledge transference from one agent to another through the separation of specific elements of the problem domain. Hierarchical control schemes have been used to reduce complexity in many applications [Pappas, 2000]. This section will present a method for implementing a hierarchical structure in an RL agent.

4.3.1 Motivation

As discussed in Subsection 4.2.1, the performance of an RL agent may be enhanced by reducing the complexity of the environment that the agent must deal with. This can be

accomplished by reducing the amount of state information that must be maintained by the agent. Also, although it does not directly result in performance increases, an ability to transfer knowledge from one agent to another is a desirable characteristic of an RL algorithm.

4.3.2 Implementation

A block diagram of a hierarchical RL agent is presented in Figure 4.2. The two layers of this architecture each consist of a reinforcement-learning agent. By utilizing each level of RL to handle a subset of the desired task, the complexity of the problem is reduced. In this type of hierarchical scheme, the low-level agent generates the actions to be taken by the system as a whole, using the output of the high-level RL agent as an additional input. In the example of robotic navigation, the high-level agent could be used to choose a general course of action, while the low-level agent would choose the specific action.

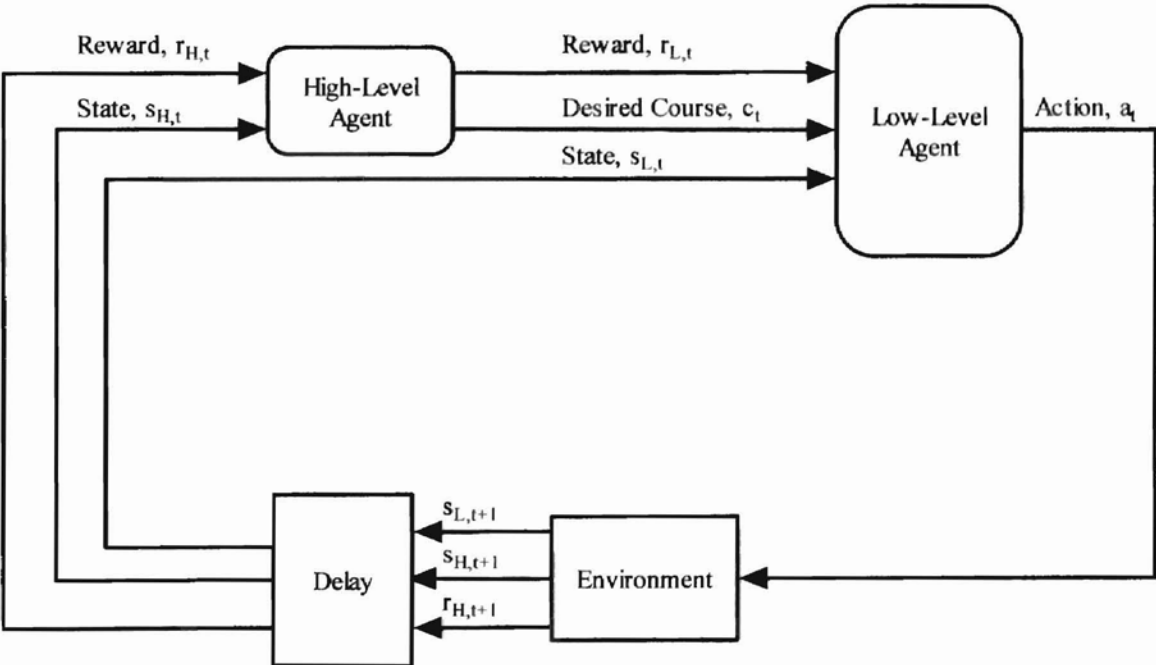


Figure 4.2: Architecture of a Simple Hierarchical RL Agent

There are several issues to address when considering this type of hierarchical scheme. First, a decision must be made about the method of presenting state information and reward information to the two different agents.

Although the decision as to how to present state information is specific to the application, in general the high-level agent should be presented with information that is global, pertaining to the largest scope of the problem to be solved. The low-level agent should be presented with local information pertaining to the specific type of sub-problem that the low-level agent is supposed to handle. For example, in a robotic navigation system, the high-level agent could use for its state information the current location of the robot; while the low-level agent could use the state of the nearby environment. The high level agent would generate a desired course to reach the goal as quickly as possible, while the low-level agent would attempt to follow that course while avoiding any nearby obstacles.

The separation of reward values for the high- and low-level agents is a much more difficult problem than the separation of state information, and is much more specific to the problem domain. As such, we will discuss the reward values specifically in the context of robotic navigation. As the high-level agent is designed to choose the overall course to follow, the reward signal used for it should be the same as that used for standard reinforcement learning agents: a small penalty for each step taken, a large penalty for collisions with obstacles, and a large reward for reaching the goal. The exact magnitudes of each reward and penalty should be determined based on the specific environment.

The reward given to the low-level agent should be designed so as to provide accurate feedback as to the suitability of that agent's actions. In order for this to happen, the reward

function must be designed specifically to fit the task designated to the low-level agent. Continuing with the example of robotic navigation, it is suggested that the low-level agent be given the task of obstacle avoidance. The output of the high-level agent would indicate the direction of travel desired, and the low-level agent would attempt to move in that direction unless obstacles were in the way. The reward to the low-level agent should be generated by the high-level agent and should reflect whether or not the low-level agent achieved travel in the desired direction, and if so, the speed with which the low-level agent achieved the goal. The low-level agent should be heavily penalized for travel in the wrong direction or for impacting obstacles, and slightly penalized for each step taken, in order to assure that it attempts to attain results as quickly as possible. It should be rewarded for travel in the direction chosen by the high-level agent.

4.3.3 Advantages of Hierarchical RL

The primary advantage of using multiple agents is to allow separation of different elements of the problem, allowing for faster learning and for transference of knowledge. This is best illustrated through example. Considering the autonomous navigation problem, an agent could be designed to drive a car from Stillwater to Oklahoma City. The high-level agent would choose the desired direction of travel with the goal of reaching Oklahoma City. The low-level agent would implement obstacle avoidance behavior. This would allow faster learning as the high-level agent would learn exclusively about the route from Stillwater to Oklahoma City, while the low-level agent would learn about avoiding obstacles, without consideration of the current location of the car. The separation of navigation and obstacle avoidance would also allow some transference of knowledge from one agent to another. Although the high-level

knowledge gained would be specific to a particular problem, the low-level knowledge could be directly transferred to another agent, as long as the structure of the agent remained the same. For example, a second agent designed to drive from Stillwater to Tulsa could use the obstacle avoidance information from the low-level agent, and have the benefits of only having to learn route information.

4.4 Hierarchical, Feature-Based Reinforcement Learning

An extension of the hierarchical RL scheme discussed above is to use the output of a feature recognition algorithm as the state input to the low-level RL agent. Considering the example of robotic navigation, the “state” perceived by the low-level RL agent could be the output of a neural network that is connected to a video camera. This would allow the low-level agent to focus specifically on obstacle avoidance behavior, and would reduce the complexity of the problem dealt with by the low-level agent.

CHAPTER V

METHODOLOGY

5.1 Implementation of Reinforcement Learning Agents

All data presented in this thesis was obtained by simulation of RL agents using Matlab. Functions were developed to implement each type of RL agent discussed: forgetting Q-Learning, feature based Q-Learning, and hierarchical Q-Learning. Functions were also developed to implement both static and dynamic gridworld environments. Each experiment was performed using a Matlab script file containing instructions for initializing the agent(s) and environment, collecting the data, and storing the data to a file. Many experiments were performed in a batch mode, so that a large number of trials could be performed without intervention. Section 5.2 below presents an overview of the experiments performed, as well as the methodology used to analyze the results. Source code used in experimentation is presented in Appendix A.

5.2 Summary of Experiments

This section presents a summary of the experiments performed, and an explanation of the methods used to analyze the collected data. This chapter presents only the methodology used for experimentation; actual results, and the analysis of those results, are presented in Chapter VI.

Four main types of experiments were performed, the details of these are presented in Subsections 5.2.1 – 5.2.4 below.

5.2.1 Variation of Parameters

Each type of learning algorithm has several parameters that must be adjusted for optimal performance. Prior to any comparative testing of algorithms, the behavior of each algorithm was analyzed with respect to variation of its learning parameters, and with respect to variation of environmental parameters. The results of these experiments were used to determine agent parameters used in all further experiments.

For a standard Q-Learning agent, the parameters that were considered are: α , λ , and γ .

The forgetting Q-Learning agent is dependent on the same set of parameters as the standard Q-Learning agent. In addition, it is dependent on the decay value for the forgetting mechanism, μ .

For the hierarchical agent incorporating feature-based RL, the parameters to be considered are: the parameters of both the low-level and high-level RL agents, which are standard Q-Learning agents.

Environmental parameters were considered as well as agent parameters. For a static environment, the parameters considered were the size of the gridworld, and the density of obstacles in the gridworld. For dynamic environments, the parameter used to characterize the environment was the dynamic period of environmental change.

In order to reduce the amount of data that must be analyzed, agent parameters were optimized one at a time. For example, considering the standard Q-Learning agent, the values of the agent parameters were initially fixed at nominal values of $\alpha = 0.1$, $\lambda = 0.9$, and $\gamma = 0.9$. For each environmental configuration that was examined, α was initially varied from 0 to 1, and the value that produced the best performance selected. The same

procedure was repeated for λ , and then for γ . Although this approach may lose some generality, it is believed to be acceptable for the purpose of clarity.

In addition to establishing appropriate parameters for each RL agent in each type of environment, the data obtained in this step allows for analysis of the behavior of each proposed algorithm. By observing how each agent performs based on its internal parameters, and the features of the environment it is operating in, some information concerning the function of that algorithm can be deduced.

5.2.2 Comparison to Established Methods

In order to establish a baseline for analysis of the proposed RL algorithms, the performance of each algorithm was compared to that of established algorithms in each environment of interest. The established method used for comparison was traditional Q-Learning, incorporating an eligibility trace. The method of comparison consisted of two parts: first, comparison of average performance over a large number of samples; and second, comparison of the speed of learning of each method. The algorithms to be compared were implemented using parameters determined from the first set of experiments performed.

5.2.3 Comparison to Optimal Solutions

If the simulation environment allowed for exact calculation of the optimal solution, the results of each RL algorithm were compared to the optimal solution. This comparison provides two pieces of information: first, if the best solution that the agent obtains is optimum or near-optimum, and second, how quickly the agent converges to a near-optimum solution. In the case of dynamic environments where a true optimal solution

could not be calculated, the optimal solution was calculated as if for a static trial, using the state of the environment at the start of the trial.

5.3 Knowledge Transfer Experiments

Knowledge transfer experiments were performed using the hierarchical RL methods described in Chapter IV. The method of evaluation was as follows:

- 1) Train two hierarchical RL agents in different gridworld environments.
- 2) Exchange the value functions of the low-level RL algorithm in each agent.
- 3) Evaluate the performance of the agents immediately, without allowing any training time.

The difference in performance following the exchange of low-level agent data will be indicative of the suitability of knowledge transference. In the ideal case, knowledge is completely transferable, and agent performance will be unchanged. In the case that complete transference is not obtained, the magnitude of the performance difference must be analyzed. A small change in performance indicates a mostly successful transfer, while a large decrease in performance indicates an unsuccessful transfer. In the case that no knowledge was successfully transferred, it would be expected that the performance of the agent following the data exchange would be similar to or worse than the agent's performance at the beginning of training.

5.4 Summary

This Chapter presented an overview of the experiments that were performed, and the methods to be used to analyze the data obtained. Chapter VI will present all experimental results, and an analysis will follow in Chapter VII.

CHAPTER VI

FINDINGS

6.1 Standard Q-Learning

In order to provide a baseline for analysis of the proposed methods, simulations were first performed for a standard $Q(\lambda)$ agent. Subsection 6.1.1 presents the data obtained.

6.1.1 Variation of Parameters

As discussed in Subsection 5.2.1, the parameters were optimized in the order: α , λ , then γ . The parameters were first optimized in a randomly generated 32×32 static gridworld environment with an obstacle density of 0.2 (20% of all states contained obstacles). Data was averaged over 100 trials in differing environments, with 300 episodes being performed per trial. The data collected was: average time to completion, in actions performed by the agent, over 300 episodes, best time to completion (as compared to the optimal solution), and the number of episodes required to reach the best time to completion. Table 6.1 contains the results of the optimization, presented in the order that the experiments were performed. In the columns containing the parameters of optimization, the parameter that was being modified is in boldface text. In the results column, the best result is highlighted for each parameter that is being varied. The first entry in the Best Performance column is the actual best time to completion in that group of trials. The second number, in parentheses, is the optimal solution for those trials. When the three performance measures used were optimal for a different set of parameters - i.e. one set of parameters produced the best time to completion, but another had a better

speed of learning – the set of parameters producing the best average time to completion was used.

Table 6.1: Variation of Parameters for a Standard Q-Learning Agent

α	λ	γ	Average Time to Completion	Best Performance	Number of Episodes to Reach Best
0.1	0.9	0.9	140.5	37 (27)	238
0.3	0.9	0.9	101.6	34 (27)	327
0.5	0.9	0.9	96.6	34 (27)	68
0.7	0.9	0.9	73.5	28 (27)	156
0.9	0.9	0.9	77.2	30 (27)	99
0.7	0.1	0.9	97.3	33 (30)	134
0.7	0.3	0.9	81.3	35 (30)	100
0.7	0.5	0.9	69.5	33 (30)	209
0.7	0.7	0.9	71.8	39 (30)	86
0.7	0.9	0.9	73.6	39 (30)	132
0.7	0.5	0.1	128.9	29 (29)	190
0.7	0.5	0.3	89.5	29 (29)	154
0.7	0.5	0.5	68.9	29 (29)	94
0.7	0.5	0.7	51.8	29 (29)	45
0.7	0.5	0.9	49.9	29 (29)	66

For the remaining experiments using standard Q-Learning, the parameters determined from this experiment were used: $\alpha = 0.7$, $\lambda = 0.5$, and $\gamma = 0.9$.

After parameter values were determined, the performance of a standard Q-Learning agent was analyzed with respect to obstacle density and to environment size. Performance with respect to obstacle density was analyzed in a static environment of size 16 by 16. Figure 6.1 is a chart showing the average time to completion over 100 trials of 200 episodes each. The obstacle density was varied from zero to 0.3 in steps of 0.05. Figure 6.2 presents a graph of time to completion vs. environment size for a standard Q-Learning agent. In this experiment, obstacle density was fixed at 0.2. The time to completion is averaged over 100 trials of 200 episodes each. The environment size was varied from 8×8 to 64×64 in steps of 8 units. The environment shape remained square.

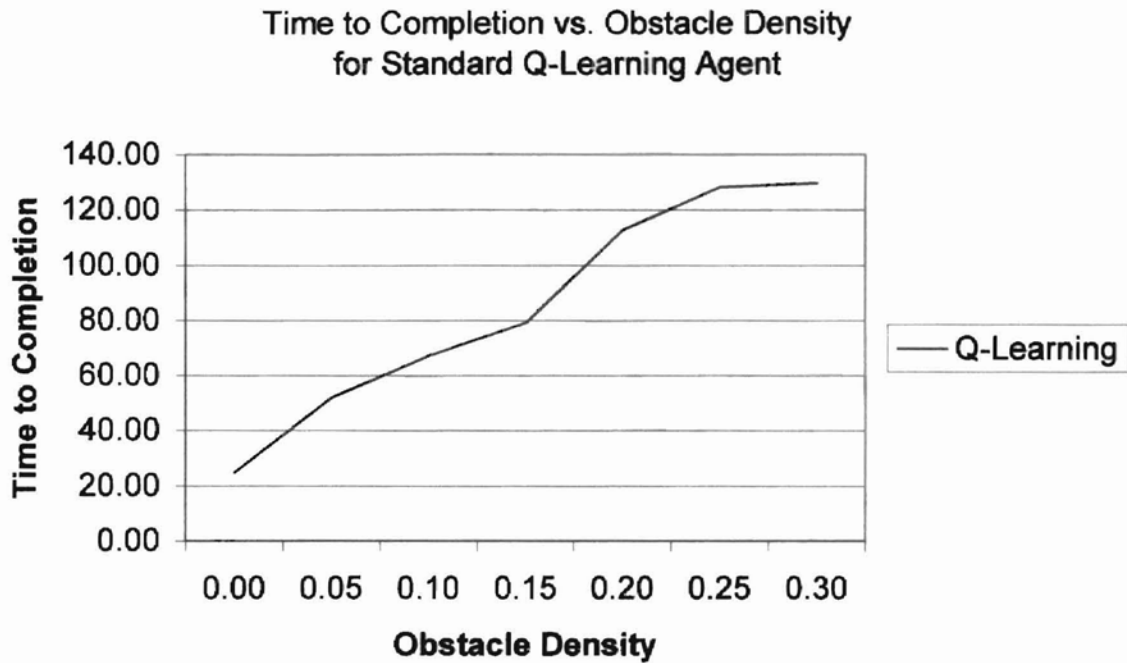


Figure 6.1: Time to Completion vs. Obstacle Density for Standard Q-Learning Agent

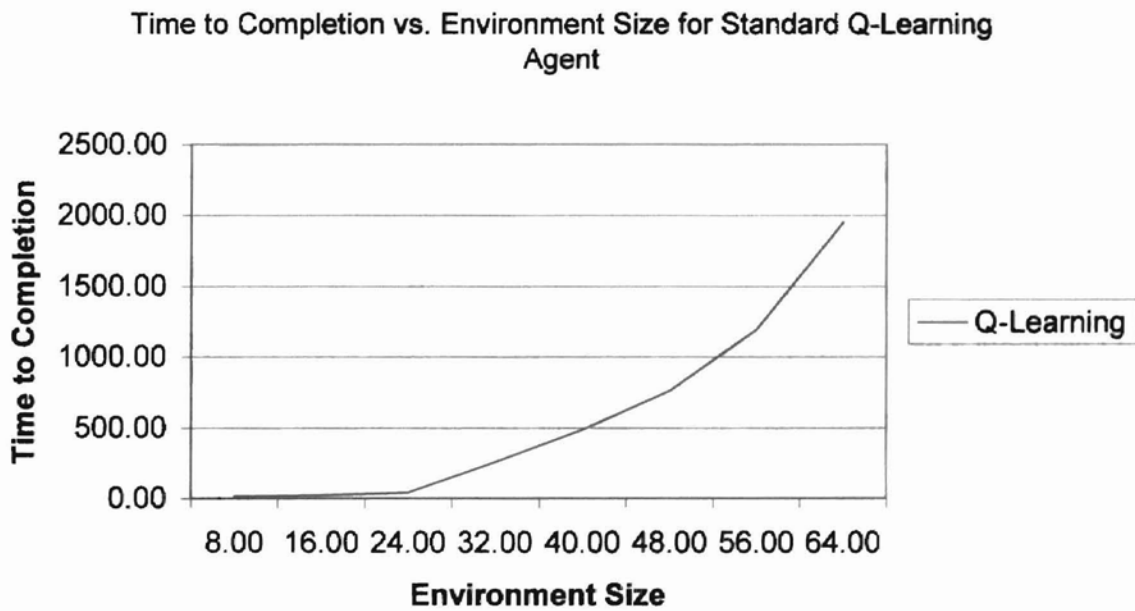


Figure 6.2: Time to Completion vs. Environment Size for Standard Q-Learning Agent

Following analysis in a static environment, the performance of standard Q-Learning in dynamic environments was investigated. The performance of a Q-Learning agent was analyzed with respect to the dynamic period of the environment. Figure 6.3 shows a graph of average time to completion vs. the dynamic period of the environment. Data was averaged over 100 trials consisting of 200 episodes each.

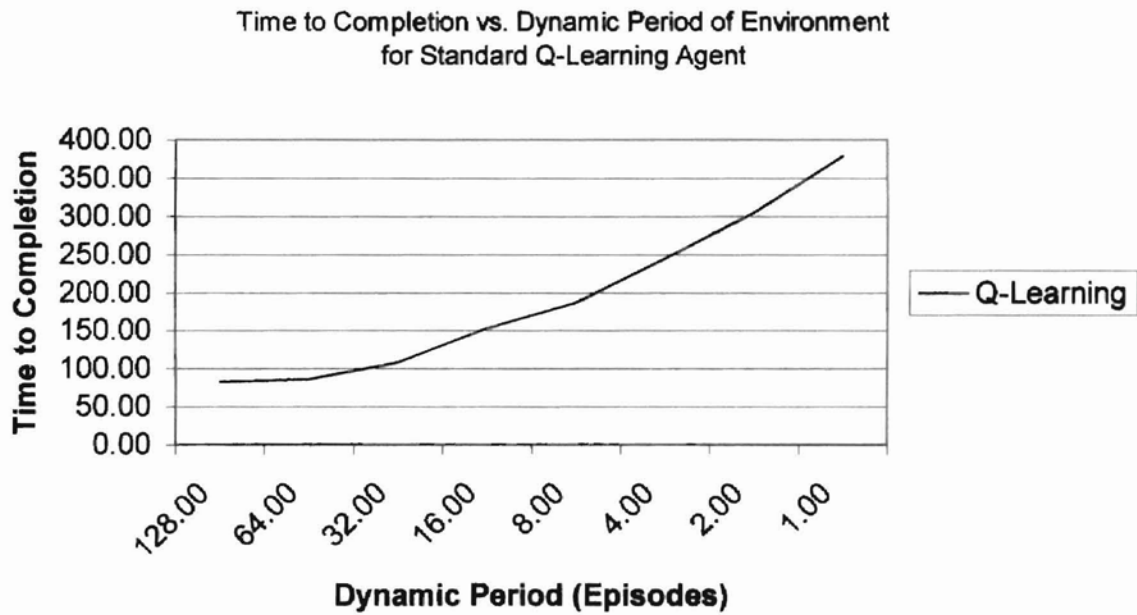


Figure 6.3: Time to Completion vs. Dynamic Period for Standard Q-Learning Agent

6.2 Incorporation of a Forgetting Mechanism

As discussed in Section 4.1, a forgetting mechanism was incorporated into a modified Q-Learning agent. This section presents the results of experiments incorporating a forgetting mechanism.

6.2.1 Variation of Parameters

As the Forgetting Q-Learning agent is very similar to the standard Q-Learning agent, the optimal parameters determined for standard Q-Learning were used, and analysis was performed only over variation of the forgetting constant. Table 6.2 shows the results of

varying the forgetting parameter. Only values from 0.9 to 1.0 were used, as values below 0.9 resulted in the agent being incapable of any learning.

Table 6.2: Variation of Parameters for Forgetting Q-Learning

μ	Average Time to Completion	Best Performance	Number of Episodes to Reach Best
0.90	140.5	1932 (31)	194
0.92	101.6	1467 (31)	198
0.94	96.6	873 (31)	193
0.96	73.5	287 (31)	189
0.98	77.2	77 (31)	120
1.00	51.3	31(31)	72

No further data is presented in the context of static environments for forgetting Q-Learning, as in a static environment the forgetting mechanism causes a decrease in performance. Performance vs. the dynamic period in a dynamic environment was analyzed, and the results are presented in Figure 6.4. The experimental parameters were the same as those used in Subsection 6.1.1 for a Standard Q-Learning Agent, with the exception that the forgetting constant was set to a value of 0.99.

6.2.2 Comparison to Established Methods

As the data obtained from Subsection 6.2.1 indicates, the forgetting parameter is best left at, or very near, one. As such performance data for the forgetting Q-Learning agent in a static environment will be identical to that for a standard Q-Learning agent.

In a dynamic environment, a difference between Q-Learning and Forgetting Q-Learning can be seen. Figure 6.5 presents a comparative chart of Q-Learning and Forgetting Q-Learning performance vs. the environment's dynamic period. For clarity, the optimal time to completion is also presented in Figure 6.5

Performance of Forgetting Q-Learning Agent vs. Dynamic Period of Environment

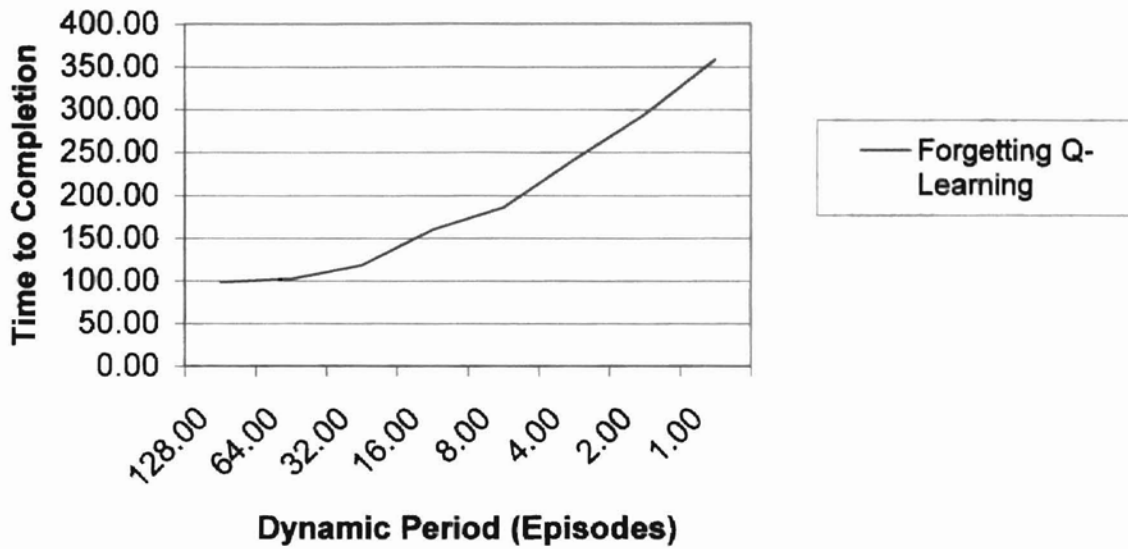


Figure 6.4: Time to Completion vs. Dynamic Period for Forgetting Q-Learning

6.2.3 Comparison to Optimal Solutions

As discussed in Subsection 6.2.4, performance of Forgetting Q-Learning in a static environment is arbitrarily close to the performance of Standard Q-Learning. As such, only data concerning dynamic environments is presented here. Figure 6.5 includes the optimal time to completion data for the experiment.

Comparison of Standard and Forgetting Q-Learning in a Dynamic Environment

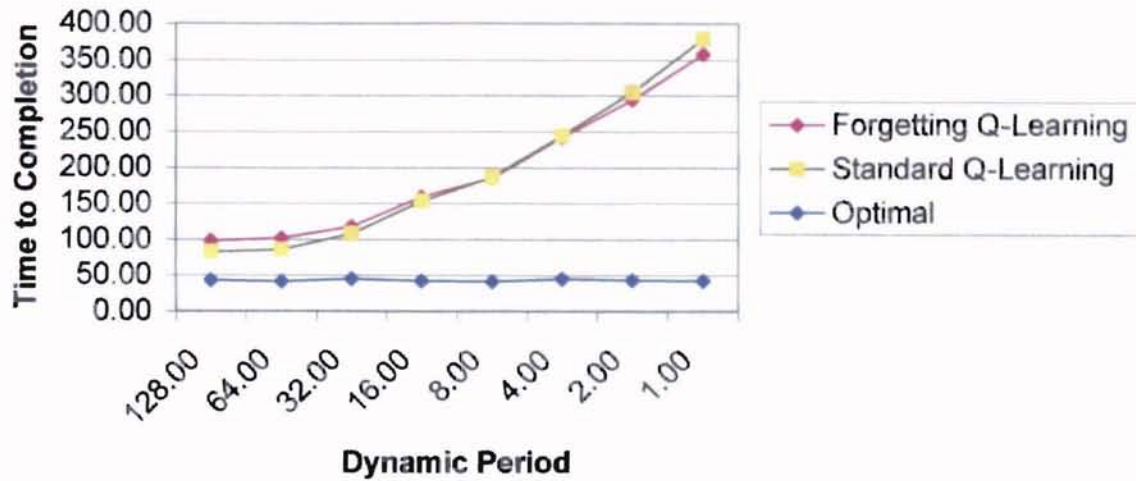


Figure 6.5: Comparison of Standard Q-Learning and Forgetting Q-Learning in a Dynamic Environment

6.3 Hierarchical Q-Learning

Hierarchical Q-Learning was discussed in Sections 3.3 and 3.4. The implementation that used for experimentation included a feature-based RL agent for the low-level agent. Simplification of the problem domain was accomplished by using a factor of 5 reduction in the size of the values stored by the high-level agent. For example, in a 25×25 element gridworld, the high-level agent would store values for a 5×5 problem. The input of the high-level agent is the agents coordinate in the reduced state space, and it receives reward and state information only upon transition into a new state in the reduced state space. The output of the high-level agent is the desired direction of travel. The input to the low-level agent is the 3×3 state area (in the full sized environment) immediately in front of the agent in the desired direction of travel. This results in a total of 512 states being maintained by the low-level agent, which roughly corresponds to the

number of states maintained by a standard Q-Learning agent in a 22×22 gridworld. Figure 6.6 illustrates the inputs to the high and low-level RL agent. The arrow represents the current location of the agent, and the area in the 3×3 square represents the input to the feature-based low-level agent. In this case, the high-level agent would be in location (1, 2), assuming the origin starts at the lower left hand corner.

The reward provided to the low-level agent after each action is one of three possibilities: a large (-100) penalty if a collision with an obstacle results, a small bonus (+5) if the high-level agent detects a transition to a state (in the reduced state space) with a higher value function, and a small penalty (-1) on each action that produces no other results.

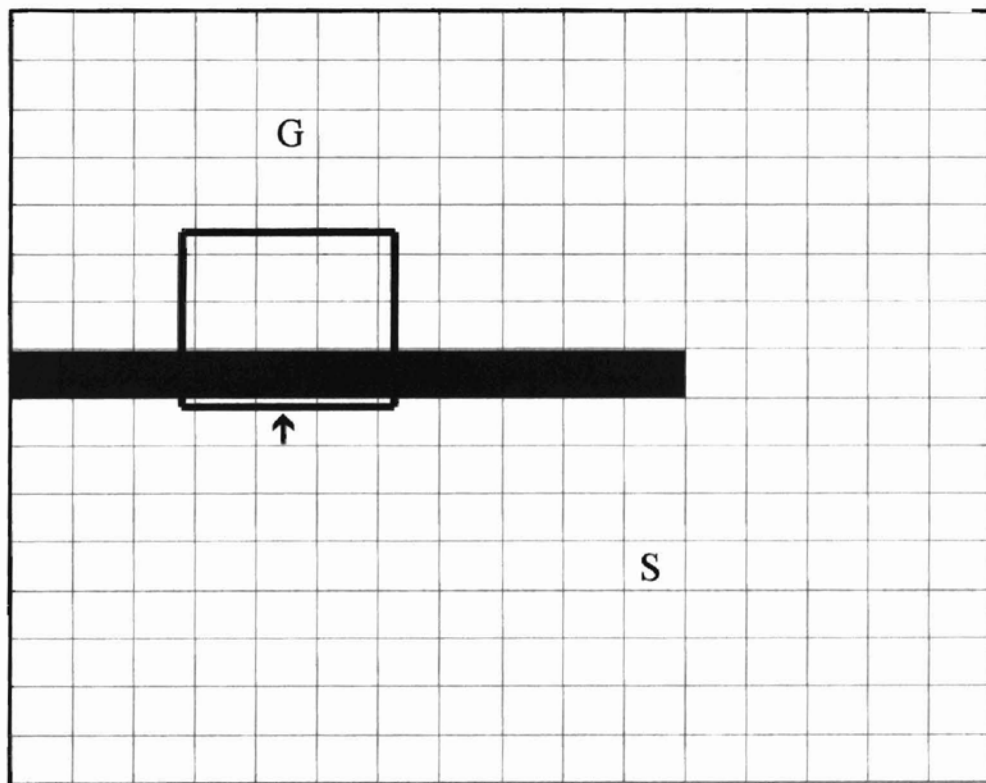


Figure 6.6: Illustration for Hierarchical RL Example

6.3.1 Variation of Parameters

As the learning rate, α , has the largest effect on the performance of an RL agent, the other parameters for both of the agents were fixed at the values established in Subsection 6.1.1. Then, the effects of varying the learning rates of both the high-level and low-level agents was examined. Table 6.3 presents the results of varying the learning rates of the two agents in a hierarchical scheme. The data was obtained using the same experimental setup as was used for variation of parameters in Subsection 6.1.1.

Table 6.3: Variation of Parameters for Hierarchical Q-Learning

α_h	α_l	Average Time to Completion	Best Performance	Number of Episodes to Reach Best
0.1	0.9	81.9	52 (31)	35
0.3	0.9	68.3	44 (31)	41
0.5	0.9	73.4	48 (31)	37
0.7	0.9	80.7	54 (31)	43
0.9	0.9	94.6	65 (31)	51
0.3	0.1	99.5	57 (29)	39
0.3	0.3	93.9	53 (29)	43
0.3	0.5	77.9	48 (29)	45
0.3	0.7	72.3	45 (29)	38
0.3	0.9	67.9	42 (29)	40

Following the establishment of the parameters for the agent, performance was tested vs. environment size and obstacle density in a static gridworld environment. Figures 6.7 and 6.8 present the results of these experiments.

In a dynamic gridworld environment, the performance of a Hierarchical Q-Learning agent was analyzed with respect to the dynamic period of the environment. Figure 6.9 presents the results of this experiment.

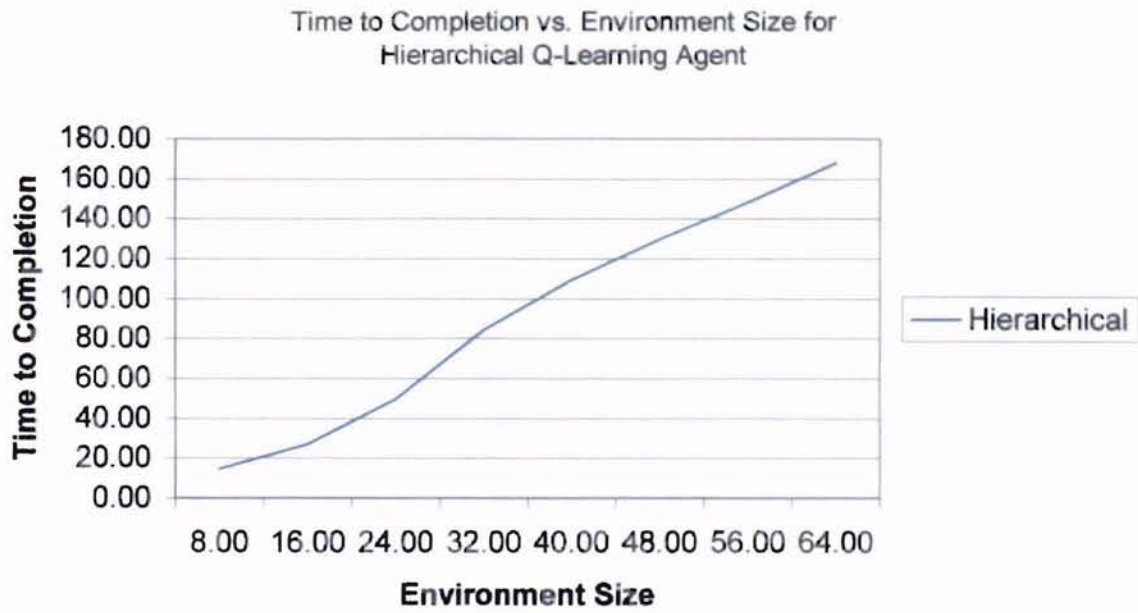


Figure 6.7: Time to Completion vs. Environment Size for Hierarchical Q-Learning Agent

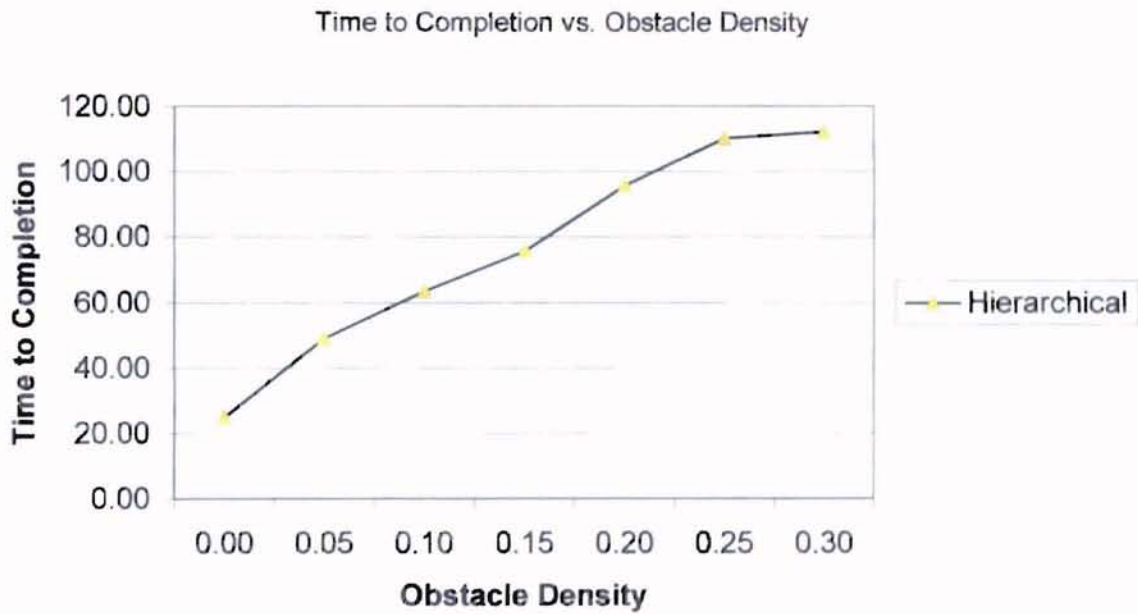


Figure 6.8: Time to Completion vs. Obstacle Density for Hierarchical Q-Learning Agent

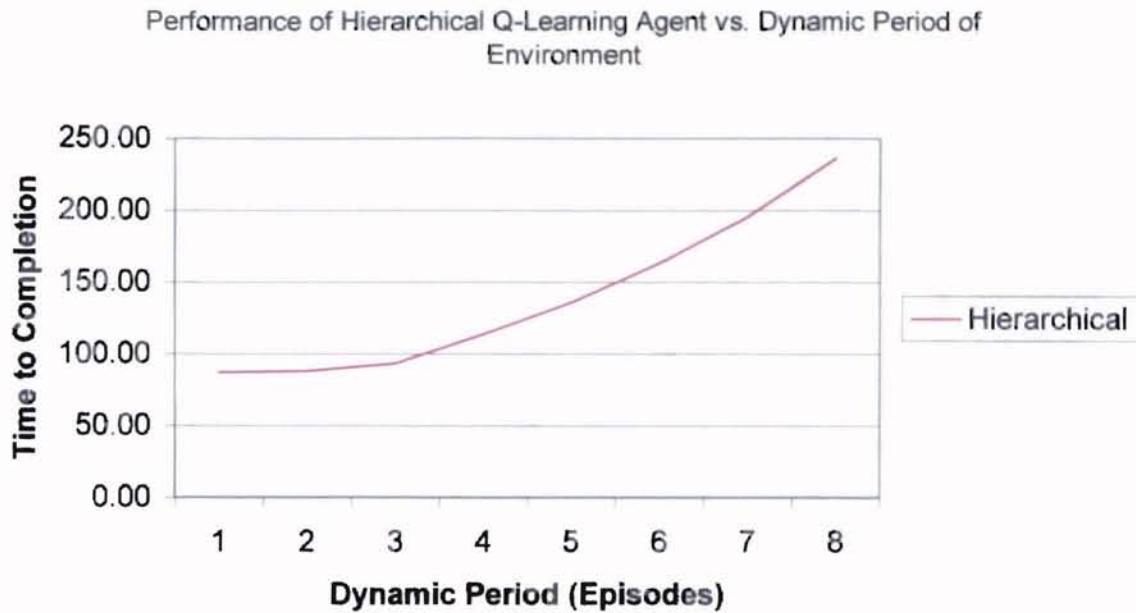


Figure 6.9: Time to Completion vs. Dynamic Period for Hierarchical Q-Learning Agent

6.3.2 Comparison to Established Methods

The performance of Hierarchical Q-Learning is compared to that of Standard Q-Learning and to the optimal solutions in this and the following Subsection. In order to reduce the number of graphs, optimal solutions have been included on the same charts as Standard Q-Learning. Figures 6.10 – 6.12 present the comparisons.

6.3.3 Comparison to Optimal Solutions

As discussed in the preceding section, data concerning optimal solutions has been included with data for Standard Q-Learning in Figures 6.10 – 6.12.

Comparison of Learning Methods vs. Environment Size

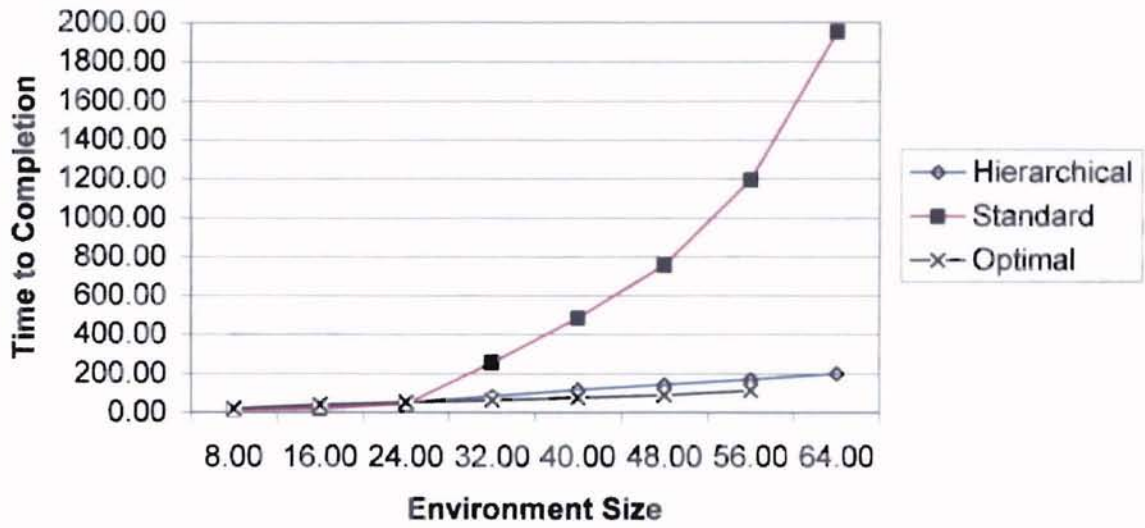


Figure 6.10: Comparison of Hierarchical Q-Learning to Standard Q-Learning and Optimal Values with Respect to Environment Size

Time to Completion vs. Obstacle Density

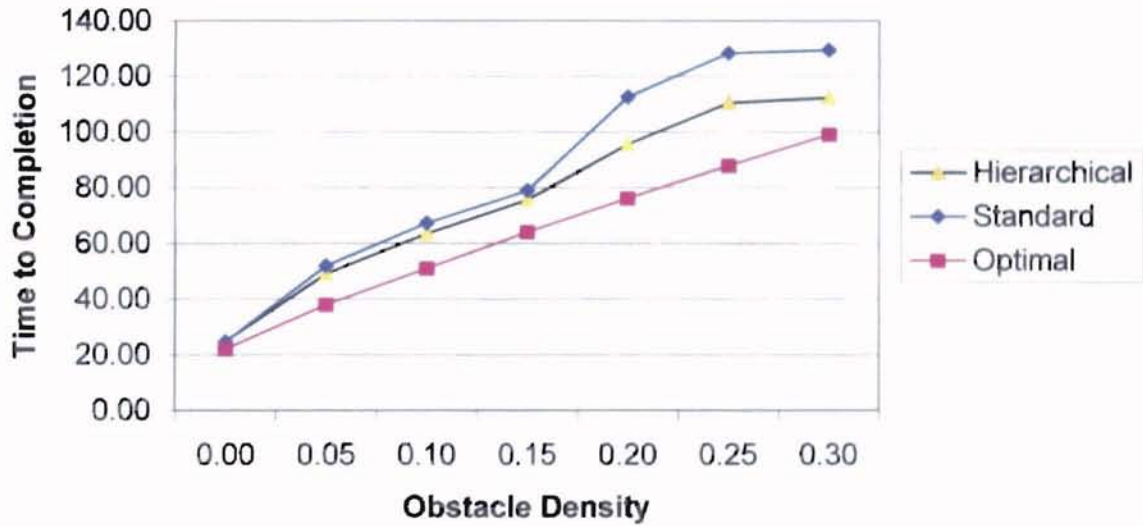


Figure 6.11: Comparison of Hierarchical Q-Learning to Standard Q-Learning and Optimal Values with Respect to Obstacle Density

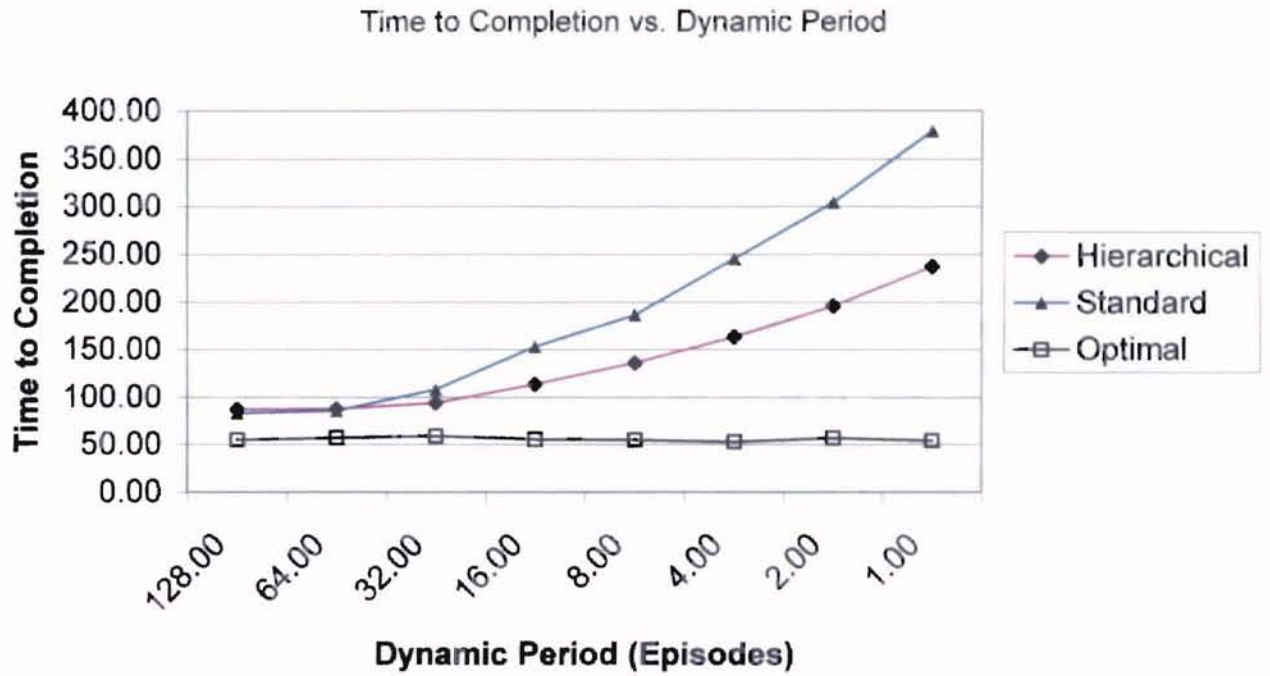


Figure 6.12: Comparison of Hierarchical Q-Learning to Standard Q-Learning and Optimal Values with Respect to Dynamic Period of Environment

6.4 Summary

This chapter has presented data obtained via Matlab simulation for the analysis of the proposed RL methods. Chapter VII presents a discussion of the results, and draws conclusions concerning the behavior of the proposed algorithms.

CHAPTER VII

SUMMARY AND CONCLUSIONS

7.1 Summary

This thesis has presented an overview of the fields of reinforcement learning and of robotic navigation, and has proposed solutions for the use of RL methods to perform navigation in dynamic environments. Two major goals have been presented: improving performance of RL agents in dynamic environments, and increasing the ability to transfer knowledge among multiple agents. Three proposed solutions have been presented and analyzed: Forgetting Q-Learning, Feature Based Q-Learning, and Hierarchical Q-Learning.

Forgetting Q-Learning is proposed to improve performance in a dynamic environment by maintaining possible navigation paths that would be considered unacceptable by traditional Q-Learning. The forgetting mechanism is implemented as a decay of unexplored state values. This leads to a higher tendency towards exploration, which should allow for increased performance in a dynamic environment.

Hierarchical Q-Learning is proposed as a method of subdividing the problem domain into a set of more manageable problems. This is accomplished by using two interacting RL agents. One agent generates desired goal-seeking behavior, while the other implements direct actions. This architecture could be considered similar to a robotic navigation scheme incorporating both deliberative and reactive elements. The high-level RL agent would represent deliberative behavior, while the low-level agent exhibits reactive behavior. Hierarchical RL algorithms are proposed to lead towards both goals of

navigation in a dynamic environment and knowledge transference among multiple agents.

Feature Based Q-Learning is proposed as a method of enhancing Hierarchical Q-Learning. Feature based RL is accomplished through the use of a feature identification method to process state inputs to the RL agent. Using a feature processing and identification scheme allows for a reduction of the state space that must be searched. However, a purely feature based RL agent would be overly simplified, and incapable of solving problems that depended on more information than just local features. When considered in the context of a hierarchical RL algorithm, a feature based RL agent makes more sense. As a low-level agent, a feature based algorithm could implement obstacle-avoidance behavior, easing the work load on the high-level agent, and allowing for faster reaction to changes in a dynamic environment.

Three primary methods were used to analyze the performance of the proposed algorithms. All three methods were implemented in Matlab simulation. The first method is to vary the parameters of the algorithm being examined and the environment it is in, and to observe the performance of the agent. This provides a baseline for comparison to other methods, and gives basic data about the behavior of the algorithm. The second analysis method used is comparison to an established RL method, Q-Learning with eligibility traces. The final method of analysis is to compare the performance of the proposed algorithms to the optimal performance in the given environment. Section 7.2 presents a discussion of the results given in Chapter VI.

7.2 Discussion of Research Findings

7.2.1 Forgetting Q-Learning

As can be seen in Table 6.2, incorporation of a forgetting mechanism into a Q-Learning agent offers no performance improvements when dealing with a static environment. We suggest that the reason for the performance decrease is that the agent is forgetting the optimal path to the goal. In a static environment, further exploration is not necessary once an optimal path has been discovered, therefore the forgetting mechanism is degrading performance by causing the agent's behavior to become too exploratory.

Figures 6.4 and 6.5 illustrate that the forgetting mechanism affords a slight performance increase (approximately 5% at the best case) when dealing with a dynamic environment. This performance increase is likely due to the same effect as the performance decrease seen in a static environment – a higher degree of exploration benefits the robot when dealing with a rapidly changing environment.

Overall, the performance gains associated with Forgetting Q-Learning are limited.

7.2.2 Hierarchical Q-Learning

Hierarchical Q-Learning has been proposed as an approach to dealing with dynamic environments that are normally not handled well by traditional RL methods. This subsection will discuss the results of experiments performed using a Hierarchical Q-Learning algorithm. Before drawing any conclusion about Hierarchical Q-Learning as a whole, the results of each experiment will be discussed.

When examining the data for variation of parameters of a Hierarchical Q-Learning agent (Table 6.3), it can be seen that the high-level and low-level RL agents reach their best

performance when using differing learning rates. The learning rate that produced the best results for the high-level agent was 0.3, while for the low-level agent it was 0.9.

The reason for the low learning rate associated with the high-level agent is likely due to the fact that the agent is operating in a very small state-space. As such, when using an eligibility trace to assign credit to previous actions, overgeneralization occurs. From the perspective of the high-level agent, very few actions are taken to traverse from start to goal. The use of an eligibility trace, couple with a high learning rate, results in assigning some credit for reaching the goal to all actions taken by the high-level agent. Without the ability to distinguish between more and less successful actions, the high-level agent learns more slowly.

The high learning rate for the low-level agent is hypothesized to be due to the complexity of the state space that it operates in. Although the actual number of states is not large, the transitions from one state to the next are non-Markovian, and in a dynamic environment are also non-stationary. In addition, the rewards seen by the low-level agent only occur at intervals of several actions. It is important for correct learning that the low-level agent associates the rewards seen with all actions taken to achieve that reward.

When examining the behavior of the Hierarchical RL Agent in a static environment (Figures 6.7 – 6.12), it is immediately apparent that large performance gains are to be had by simplification of the state space seen by the controlling agent. As the environment size grows very large, the Hierarchical Q-Learning Agent achieves an improvement of nearly an order of magnitude over the results seen by the Standard Q-Learner. We believe that this is due to the reduction in size of the state space of the high-level agent. As is seen with the integration of deliberative and reactive navigation schemes, great

performance enhancements are to be had by separating path planning tasks from obstacle avoidance tasks. This is, in essence, what is done by the Hierarchical Q-Learning algorithm.

In addition to the large performance increase seen with respect to the size of the environment, the Hierarchical RL agent also sees a small performance increase with respect to the complexity of the environment (Figure 6.11). Although the benefit is small (14% improvement at an obstacle density of 0.30), it is noticeable. Again, this is believed to be due to the effect of separating the problem domain into two sub-problems that are each much smaller than the original problem.

When dealing with a dynamic environment, the structure of the Hierarchical RL algorithm allows the agent to deal effectively with unexpected obstacles. As the dynamic period of the environment decreases – indicating a more rapidly changing environment – the Hierarchical Q-Learning agent pulls steadily away from the traditional Q-Learning agent. Although the performance of the Hierarchical Agent is fairly poor compared to the optimal path, it is far superior to the performance of Standard Q-Learning – the performance improvement is approximately 35% at a dynamic period of one episode.

Section 7.3 presents a summary of the thesis, and attempts to project the direction of future work in this area.

7.3 Conclusions

This thesis presented three new approaches to the application of reinforcement learning methods to robotic navigation. In specific, the area of navigation in a dynamic environment was explored. The dilemma between exploration and exploitation as it relates to RL methods was discussed, leading to the conclusion that a greater degree of

exploration is required in a dynamic environment. From the perspective of robotic navigation, the contrast between deliberative vs. reactive behavior was examined, and the benefits of combining the two approaches were presented.

In Chapter IV, three approaches to the RL based navigation problem in a dynamic environment were discussed. The concept of forgetting as a method of furthering exploration was presented, and a modified Q-Learning technique based on that concept was proposed. Inspired by the hierarchical structure of robotic navigation schemes incorporating both reactive and deliberative behavior, a hierarchical RL method was suggested. Another proposed improvement to traditional RL methods was to incorporate a feature processing technique into the state sensor of an RL agent. Through analysis, it was determined that a purely feature-based RL method would be incapable of true navigation, and it was proposed that a feature-based agent be incorporated as one part of a hierarchical control scheme.

The proposed solutions were implemented in simulation, and results were presented for analysis. It was determined that Forgetting Q-Learning, although offering small performance improvements, was not applicable to the majority of the problems seen in RL. The largest part of the analysis was devoted to the behavior of the Hierarchical RL scheme.

Hierarchical RL has been demonstrated to offer improved performance not only in dynamic environments, but also in static environments. This is due to the reduction in the number of stored states that is made possible by separation of the problem domain into multiple smaller problems. Although the feature processing algorithm incorporated

into the agent studied was a very simple direct pattern-matching method, it provided improved performance in both static and dynamic environments.

7.4 Contributions to the Field

The work presented in this thesis provides a strong correlation between the fields of robotic navigation and reinforcement learning. Parallels have been drawn between deliberative and reactive navigation from the robotic navigation world and different aspects of the reinforcement learning paradigm. Further, a hierarchical RL approach based on integrated deliberative and reactive control, has been implemented and demonstrated to provide large performance increases over traditional RL in a variety of environments.

7.5 Future Work

This thesis has only scratched the surface of possible RL based autonomous navigation schemes. The work on feature-based RL could be extended through the incorporation of a neural network based feature recognition system into the agent. Extension of the hierarchical RL agent to multiple levels, possibly including multiple agents per level could easily be implemented. The effects of forgetting on RL agents could be studied in greater depth, perhaps implementing an eligibility trace based method to determine which states have not been visited in a long time, and thus which states the forgetting mechanism should be applied to. Another possible direction of research is to examine current work in robotic navigation, and draw further ideas from the parallelism between robotic navigation and reinforcement learning research. The field of robotic navigation is

very exciting right now, and reinforcement learning methods have a lot to offer to this field.

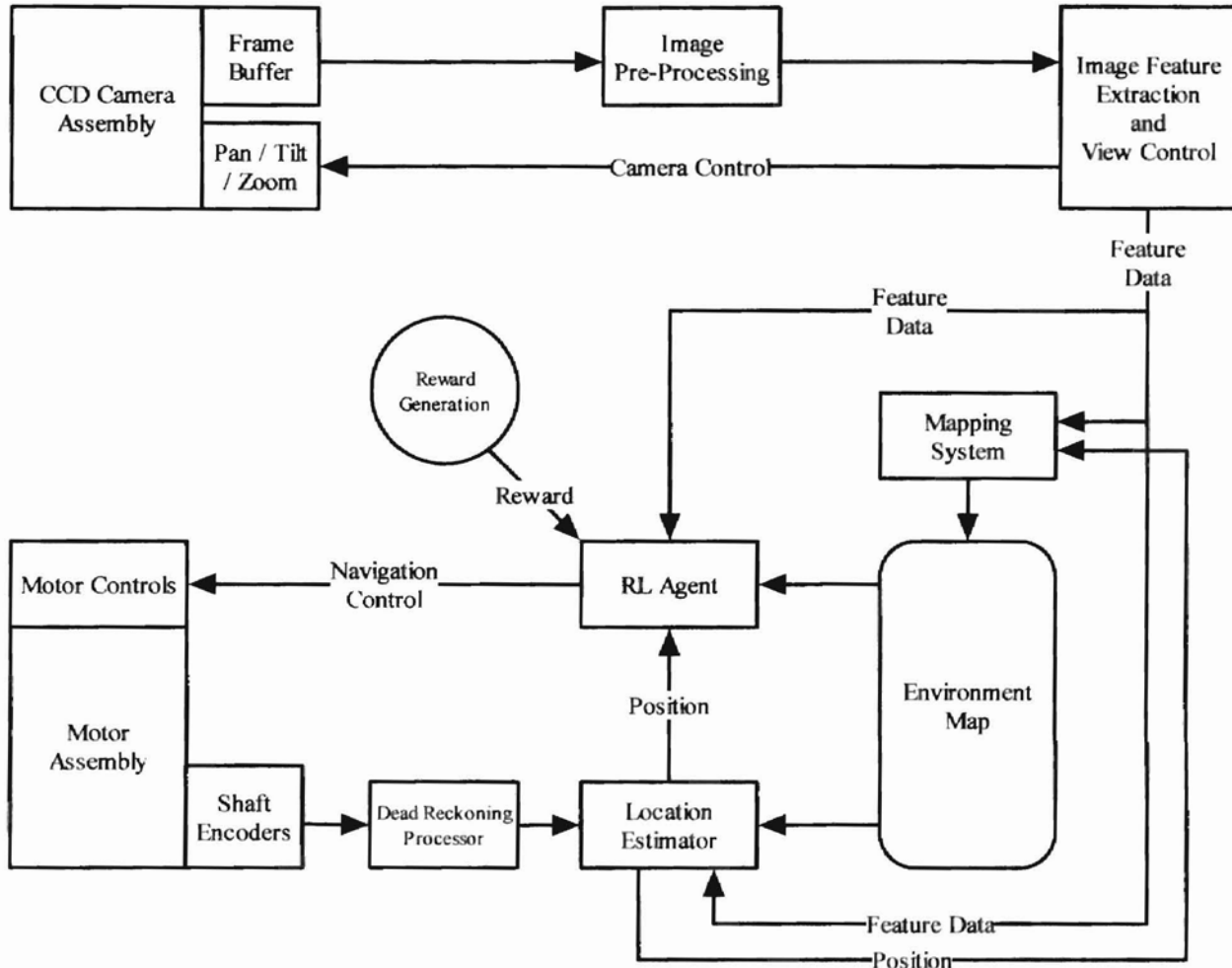


Figure 7.1: Vision-Based RL Robotic Navigation System

As an example of implementation in a robotic system, consider Figure 7.1. This figure shows a robotic navigation system incorporating a feature-based, hierarchical RL agent. The RL agent utilizes an environment model, in the form of a map, in order to increase learning speed, as discussed in Section 2.2.7. The navigation systems includes both dead reckoning and feature based methods of position estimation, and uses both feature data

and estimated location to update the environment map used by the RL agent. The reward used in training the agent is generated externally to the agent; this reward could be determined prior to implementation by the designer, or could be incorporated as part of a more complex goal-based navigation system. The exact method of implementation of each block of the pictured system is yet to be completely specified, but this figure provides a framework for implementation in an actual physical system.

BIBLIOGRAPHY

- Arkin, R.C. (1987). Motor schema based navigation for a mobile robot: An approach to programming by behavior. Proc. 1987 IEEE Conf. Robot. Auto., pp. 264-271.
- Atkeson, C.G.; Santamaria, J.C. (1997). A comparison of direct and model-based reinforcement learning. Proc. 1997 IEEE ICRA., vol. 4, pp. 3557-3564.
- Barto, A. G.; Duff, M. (1994). Monte carlo matrix inversion and reinforcement learning. In Cohen, J. D., Tesauro, G., and Alspector, J., eds., Advances in Neural Information Processing Systems: Proceedings of the 1993 Conference, pp. 687-694, San Francisco, CA. Morgan Kaufmann.
- Brooks, R.A. (1986). A robust layered control system for a mobile robot. IEEE J. Robot. Auto., vol. 2, pp. 14-23.
- Buschka, P.; Saffiotti, A.; Wasik, Z. (2000). Fuzzy Landmark-Based Localization for a Legged Robot. Proceedings of Intelligent Robots and Systems, IROS 2000, vol. 2, pp. 1205-1210.
- Bellman, R. E. (1957). Dynamic Programming. Princeton University Press, Princeton, NJ
- Beom, Hee Rak; Cho, Hyung Suck. (1995). A sensor-based navigation for a mobile robot using fuzzy logic and reinforcement learning. IEEE Trans. Syst. Man Cyber., vol. 25 (3), pp. 464-477.
- Bertsekas, D. P. (1995). Dynamic Programming and Optimal Control. Athena, Belmont, MA.
- Bertsekas, D.P. (1998). New value iteration and Q-learning methods for the average cost dynamic programming problem. Proc. IEEE Conf. Dec. Control, vol. 3, pp. 16-18.
- Boone, G. (1997). Efficient reinforcement learning: model-based Acrobot control. Proc. 1997 IEEE Intl. Conf. Rob. Auto., vol. 1, pp. 229-234.
- Coelho, J.A., Jr.; Araujo, E.G.; Huber, M.; Grupen, R.A. (1998). Dynamical categories and control policy selection. Intelligent Control 1998, pp. 459-464.
- Davesne, F.; Barret, C. (1999). Reactive navigation of a mobile robot using a hierarchical set of learning agents. Proceedings of Intelligent Robots and Systems, IROS '99, vol. 1, pp. 482 - 487
- Ferrari, C; Pagello, E.; Voltolina, M.; Ota, J; Arai, T. (1997). Multirobot motion coordination using a deliberative approach. Proc. 1997 IEEE Euromicro Workshop in Advanced Mobile Robots., pp. 96-103.

- Fiorini, P; Shiller, Z. (1996). Time optimal trajectory planning in dynamic environments. Proc. 1996 IEEE Conf. Robot. Auto., vol. 2, pp. 1553-1558.
- Godbole, D.; Samad, T.; Gopal, V. (2000). Active multi-model control for dynamic maneuver optimization in unmanned air vehicles. Proceedings of IEEE ICRA 2000, vol. 2, pp. 1257-1262
- Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor.
- Howard, R. (1960). *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA
- Huang, Yih-Fang. (1994). Artificial neural networks – learning and generalization. IEEE Asia-Pacific Conference on Circuits and Systems, 1994, pp. 162
- Ibrahim, M.Y.; McFetridge, L. (2001). The agoraphilic algorithm: a new optimistic approach for robot navigation. Proc. 2001 IEEE/ASME Intl. Conf. Adv. Int. Mecha., vol. 2, pp. 1334-1339.
- Isik, C.; Meystel, A.M. (1988). Pilo level of a hierarchical controller for an unmanned mobile robot. IEEE J. Robot. Auto., vol. 4 (3), pp. 241-255.
- Kim, Jinsuck; Amato, N.M.; Lee, Sooyong. (2001). An integrated mobile robot path (re)planner and localizer for personal robots. Proc. 2001 IEEE ICRA., vol. 4, pp. 3789-3794.
- Kirley, M.; Green, D. G. (2000). An Empirical Investigation of Optimisation In Dynamic Environments Using the Cellular Genetic Algorithm. Proceedings of the Genetic and Evolutionary Computation Conference, pp. 11-18.
- Klarer, P.R. (1990). Autonomous land navigation in a structured environment. IEEE Aero. and Elec. Syst. Mag., vol. 5 (3), part 1, pp. 9–12.
- Klopf, A. H. (1975). A comparison of natural and artificial intelligence. SIGART Newsletter, vol. 53, pp.11-13.
- Kuan, D.; Phipps, G.; Hsueh, A.C. (1988). Autonomous robotic vehicle road following. IEEE Trans. Patt. Analysis and Mach. Int., vol. 10 (5), pp. 648-658.
- Kwok, K.S.; Driessen, B.J. (1999). Path planning for complex navigation via dynamic programming. Proc. Am. Contr. Conf., vol. 4, pp. 2941-2944.
- Le Moigne, J. (1988). Domain-dependent reasoning for visual navigation of roadways. IEEE J. Robot. Auto., vol. 4 (4), pp. 419-427.

- Malak, R.J., Jr.; Khosla, P.K. (2001). A framework for the adaptive transfer of robot skill knowledge using reinforcement learning agents. Proceedings of IEEE ICRA 2001, vol. 2, pp. 1994-2001.
- Mann, R. C.; Jones, J.P.; Beckerman, M.; Glove, C.W.; Farkas, L.; Han, J.; Wacholder, E.; Einstein, J.R. (1988). An intelligent integrated sensor system for the ORNL mobile robot. Proc. IEEE Symp. Int. Control, pp. 170-173.
- Millan, J.del.R. (1996). Rapid, safe, and incremental learning of navigation strategies. IEEE Trans. Syst. Man Cyber., vol. 26 (3), pp. 408-420.
- Mitchell, J.S.B.; Payton, D.W.; Keirse, D.M. (1987). Planning and reasoning for autonomous vehicle control. Int. J. Intell. Syst., vol. 2.
- Ng, K.C.; Trivedi, M.M. (1998). A neuro-fuzzy controller for mobile robot navigation and multirobot convoying. IEEE Trans. Syst. Man Cyber., vol. 28 (6), pp. 829-840.
- Nie, Junhong; Haykin, S. (1999). A dynamic channel assignment policy through Q-learning. IEEE Trans. Neur. Net., vol. 10 (6), pp. 1443-1455.
- Ono, S; Inagaki, Y.; Aisu, H; Sugie, H; Unemi, T. (1995). Fast and feasible reinforcement learning algorithm. Proc. 1995 IEEE Int. Conf. Fuzzy Syst., vol. 3, pp. 1713-1718.
- Pavlov, P. I. (1927). Conditioned Reflexes. Oxford, London.
- Payton, D.W.; Rosenblatt, J.K.; Keirse, D.M. (1990). Plan guided reaction. IEEE Trans. Syst. Man Cyber., vol. 20 (6), pp. 1370-1382.
- Puterman, M. L.; Shin, M. C. (1978). Modified policy iteration algorithms for discounted Markov decision problems. Management Science, vol. 24 pp. 1127-1137.
- Russell, S.; Norvig, P. (1995). Artificial Intelligence: A Modern Approach. Prentice Hall, Upper Saddle River, NJ.
- Ryu, Byeong-Soon; Yahng, Hyun Seung. (1999). Integrating of reactive behaviors and enhanced topological map for robust mobile robot navigation. IEEE Trans. Syst. Man Cyber., vol. 29 (5), pp. 474-485.
- Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. IBM Journal on Research and Development, pages 210--229
- Santharam, G.; Sastry, P.S. (1997). A reinforcement learning neural network for adaptive control of Markov chains. IEEE Trans. Syst. Man Cyber., vol. 27 (5), pp. 588-600.

- Shiller, Z.; Large, F.; Sekhavat, S. (2001). Motion planning in dynamic environments: obstacles moving along arbitrary trajectories. Proc. 2001 IEEE ICRA., vol. 4, pp. 3716-3721.
- Sutton, R.S.; Barto, A.G. (1998). Reinforcement Learning: An Introduction. MIT Press. Cambridge, MA.
- Song, Kai-Tai; Chang, C.C. (1999). Reactive navigation in dynamic environment using a multisensor predictor. IEEE Trans. Syst. Man Cyber., vol. 29 (6), pp. 870-880.
- Sridhar, B; Cheng, V.H.L. (1988). Computer vision techniques for rotorcraft low-altitude flight. IEEE Contr. Syst. Mag., vol. 8 (3), pp. 59-61.
- Sutton, R. S. (1996). Generalization in reinforcement learning: Successful examples using sparse coarse coding. In Touretzky, D. S., Mozer, M. C., and Hasselmo, M. E., eds., Advances in Neural Information Processing Systems: Proceedings of the 1995 Conference. pps. 1038-1044, Cambridge, MA. MIT Press.
- Tesauro, G. J. (1995). Temporal difference learning and TD-Gammon. Communications of the ACM, vol. 38, pp. 58-68.
- Thorndike, E. L. (1911). Animal Intelligence. Hafner, Darien, Conn.
- Turk, M.A.; Morgenthaler, D.G.; Gremban, K.D.; Marra, M. (1988). VITS – a vision system for autonomous land vehicle navigation. IEEE Trans. Patt. Analysis and Mach. Int., vol. 10 (3), pp. 342 – 361.
- Valavanis, K.P.; Hebert, T.; Koluru, R.; Tsourveloudis, N. (2000). Mobile robot navigation in 2-D dynamic environments using an electrostatic potential field. IEEE Trans. Syst. Man Cyber., vol. 30 (2), pp. 187-196.
- Wang, Yunfeng; Chirikjian, G.S. (2000). A new potential field method for robot path planning. Proc. 2000 IEEE ICRA., vol. 2, pp. 977-982.
- Warren, C.W. (1990). A technique for autonomous underwater vehicle route planning. IEEE J. Oc. Eng., vol 15 (3), pp. 199-204.
- Watkins, C. J. C. H. (1989). Learning from Delayed Rewards. PhD thesis, Cambridge University, Cambridge, England.
- Whitehead, S.D.; Sutton, R.S.; Ballard, D.H. (1990). Advances in reinforcement learning and their implications for intelligent control. Proceedings of 1990 IEEE International Symposium on Intelligent Control, vol. 2, pp. 1289-1297.
- Witten, I. H. (1976). The apparent conflict between estimation and control--A survey of the two-armed problem. Journal of the Franklin Institute, vol. 301, pp. 161-189.

Appendix A

Source Code

A.1 Summary

This Appendix contains the source code used to generate the experimental data in this thesis. Figure A.1 below presents a calling tree for the software generated. Each function in the tree utilizes functions connected below it. In Figure A.1, the Experimental Script block does not represent a specific file, instead it represents a Matlab script used to perform an experiment. Section A.2 contains the listings of all functions used, and Section A.3 contains example experimental scripts.

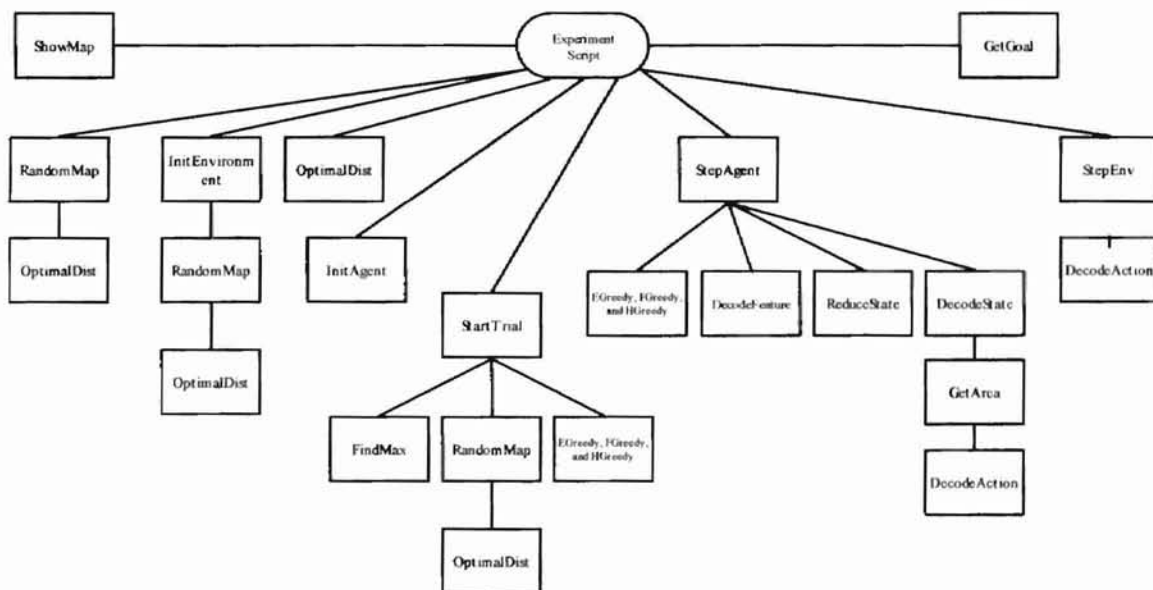


Figure A.1: Calling Tree for RL Simulations

A.2 Source Listings

The text of all source files in the calling tree above is presented in this section. Each file is presented in a separate subsection.

A.2.1 InitEnvironment

```
function E = InitEnvironment(Type, M, Parameters)
% E = InitEnvironment(Type, Map, Parameters)
% Type = 'S' for Static Environment
%       = 'D' for Dynamic Environment
% Map - 2-dimensional array containing values 0 through 3:
%       0 = Empty Square
%       1 = Obstacle
%       2 = Goal Location (Only one in map)
%       3 = Start Location
% Note - if there are no start locations specified, at
% the beginning of each episode a start location will be
% chosen randomly from all empty states. If there is
% more than one start location, the start location will
% be chosen randomly from those locations at the
% beginning of each episode.
% Parameters = Column vector containing environment specific
% information:
% For Static Environments: Parameters =
% [ObstaclePenalty, StepPenalty, GoalReward]'
% - ObstaclePenalty - Value of Reward function if an obstacle is hit
% - StepPenalty - Value of Reward function for a normal move
% - GoalReward - Value of Reward function if goal is reached
% For Dynamic Environments:
% Parameters =
% [ObstaclePenalty, StepPenalty, GoalReward, DynamicPeriod, Density]'
% - First 3 parameters as per static environments
% - Dynamic Period - Number of episodes between environment change
% - Density - Fraction of environment occupied by obstacles

if(Type == 'S') % Static Environment
    E.Type = 'S';
    E.Size.X = size(M,1);
    E.Size.Y = size(M,2);

    E.Map = M;

    E.ObsPen = Parameters(1);
    E.StepPen = Parameters(2);
    E.GoalRew = Parameters(3);

    E.State.X = 1;
    E.State.Y = 1;
end

if(Type == 'D') % Dynamic Environment
    E.Type = 'D';
    E.Size.X = size(M,1);
    E.Size.Y = size(M,2);

    E.ObsPen = Parameters(1);
    E.StepPen = Parameters(2);
    E.GoalRew = Parameters(3);
    E.DynPeriod = Parameters(4);
    E.Density = Parameters(5);
end
```

```
E.Start.X = floor(rand(1)*E.Size.X + 1);
E.Start.Y = floor(rand(1)*E.Size.Y + 1);

E.Goal.X = floor(rand(1)*E.Size.X + 1);
E.Goal.Y = floor(rand(1)*E.Size.Y + 1);

E.Map = RandomMap(E.Size, E.Density, E.Start, E.Goal);

E.State.X = 1;
E.State.Y = 1;
E.EpisodeCount = 0;
End
```


A.2.2 Random Map

```
function M = RandomMap(Size, Density, Start, Goal)
%Generates a random gridworld environment
% M = RandomMap(Size, Density)
%     Size - Structure containing two elements: Size.X and Size.Y
%           Determines horizontal and vertical size of gridworld
%     Density - Fraction of environment occupied by obstacles
% M = RandomMap(Size, Density, Start)
%     As above, but agent's start location is specified.
%     Start is a structure containing Start.X and Start.Y
%     If Start is not specified, it is determined randomly.
% M = RandomMap(Size, Density, Start, Goal)
%     As above, but agent's start and goal are specified.
%     Goal is a two-element structure containing Goal.X and Goal.Y
%     If Goal is not specified, it is determined randomly.

NumFilled = Size.X * Size.Y * Density;    % Number of obstacle locations
MaxBlobSize = ceil(NumFilled / 16);      % Max. size of single obstacle
Done = 0;    % Flag indicating that generation of env. is finished

while(Done == 0)
    M = zeros(Size.X, Size.Y);           % Start with empty gridworld
    cfill = 0;    % Count of obstacle squares
    while(cfill < NumFilled)
        BlobX = floor(rand(1)*Size.X + 1);
        BlobY = floor(rand(1)*Size.Y + 1);
        BlobSize = floor(rand(1)*MaxBlobSize + 1);
        width = floor(sqrt(BlobSize));
        for x = BlobX:(BlobX+width)
            for y = BlobY:(BlobY + width)
                if((x <= Size.X) & (y <= Size.Y))
                    if(M(x,y) == 0)
                        cfill = cfill + 1;
                        M(x,y) = 1;
                    end
                end
            end
        end
    end
    end
    if(nargin <= 2)
        SX = floor(rand(1)*Size.X + 1);
        SY = floor(rand(1)*Size.Y + 1);
        M(SX, SY) = 3; % Start Location
    else
        M(Start.X, Start.Y) = 3;
    end
    if(nargin <= 3)
        SX = floor(rand(1)*Size.X + 1);
        SY = floor(rand(1)*Size.Y + 1);
        M(SX, SY) = 2; % Goal Location
    else
        M(Goal.X, Goal.Y) = 2;
    end
    end
    test.Map = M;
end
```

```
% Generated environment must have an open path from start to goal  
    if(OptimalDist(test) > -1)      Done = 1;  
    end  
end
```

A.2.3 OptimalDist

```
function OD = OptimalDist(E)
% OD = OptimalDist(E)
% Determines optimal distance from start to goal for a gridworld.
% E = Input Environment
% OD = Optimal Distance. A return value -1 indicates that there is no
% open path from the start state to the goal state

SL = (E.Map == 3);      % Start Location
GL = (E.Map == 2);      % Goal Location
OL = (E.Map == 1);      % Obstacle Location

MaxX = size(SL,1);      % Horizontal size of map
MaxY = size(SL,2);      % Vertical size of map

MaxDist = MaxX * MaxY; % Maximum Distance (assumes every state is
visited)

% This function maintains the distance of each environmental state from
% the start state.
% The map is initialized to zero at the start location, and to the
% maximum possible distance plus one everywhere else.

SM = ones(size(E.Map)) * MaxDist;
SM = SM + 1;
SM = SM - (SM .*SL);

% The map containing the distance from each state to the start state is
% updated in a series of passes through the map. On each pass, any
% state that has not yet been updated is checked to see if it is
% adjacent to any state that has already been updated. If it is, the
% distance from the new state to the start state is set to one higher
% than that of it's neighboring state that is closest to the start. The
% process is finished when the distance to the goal state has been
% determined.

Done = 0;                % Flag to indicate completion
f = zeros(4,1);         % Stores temporary data
Pass = 0;                % Number of passes completed

while(Done == 0)
    Pass = Pass + 1;
    for x = 1:MaxX
        for y = 1:MaxY

            % Only update if there is no obstacle present

            if((OL(x,y) == 0) & (SM(x,y) > MaxDist))
                if(x > 1)
                    f(1) = SM(x-1,y);
                else % x = 1, so no valid neighbor to the left
                    f(1) = MaxDist + 1;
                end
                if(x < MaxX)
                    f(2) = SM(x+1,y);
```

```

else % x = MaxX, so no valid neighbor to the right
    f(2) = MaxDist + 1;
end
if(y > 1)
    f(3) = SM(x,y-1);
else % y = 1, so no valid neighbor above
    f(3) = MaxDist + 1;
end
if(y < MaxY)
    f(4) = SM(x,y+1);
else % y = MaxY, so no valid neighbor below
    f(4) = MaxDist + 1;
end
SM(x,y) = min(f) + 1;
End

% Goal distance has been determined

if((SM(x,y) < MaxDist) & (GL(x,y) == 1))
    Done = 1;
    OD = SM(x,y);
end
if(Pass > MaxDist) % No open path to the goal
    Done = 1;
    OD = -1;
end
end
end
end
end

```

A.2.4 InitAgent

```
function P = InitAgent(Type, Size, Parameters, Parameters2)
% P = InitAgent(Type, Size, Parameters, Parameters2)
%     - Parameters2 argument ONLY used for hierarchical agents
% P = Agent's InitialState
% Type = 'QL' - Q-Learning
% Type = 'FQ' - Forgetting Q-Learning
% Type = 'FB' - Feature based Q-Learning
% Type = 'HQ' - Hierarchical, Feature Based Q-Learning
% Size = Size of environment - structure containing Size.X and Size.Y
% Parameters = Column vector containing Agent's internal parameters
% For Type 'QL' : Parameters = [Alpha, Lambda, Gamma, Epsilon]'
%     - Alpha = Learning rate
%     - Lambda = Decay constant for eligibility trace
%     - Gamma = Weighting of expected-best outcome in backup
%     - Epsilon = Greediness of action selection policy
%           (0 = Pure Greedy, 1 = Random)
% For Type 'FQ' : Parameters = [Alpha, Lambda, Gamma, Epsilon, Mu]'
%     - Alpha = Learning rate
%     - Lambda = Decay constant for eligibility trace
%     - Gamma = Weighting of expected-best outcome in backup
%     - Epsilon = Greediness of action selection policy
%     - Mu = forgetting factor.
%           (1 = No forgetting, 0 = forget everything)
% For Type 'FB' : Parameters = [Alpha, Lambda, Gamma, Epsilon, F]'
%     - Alpha = Learning rate
%     - Lambda = Decay constant for eligibility trace
%     - Gamma = Weighting of expected-best outcome in backup
%     - Epsilon = Greediness of action selection policy
%     - F = Size of feature detector (odd integer).
%           Suggested values are 1 or 3
% For Type 'HQ' : Parameters = [Alpha, Lambda, Gamma, Epsilon, R]'
%     These parameters are for the high-level agent
%     - Alpha = Learning rate
%     - Lambda = Decay constant for eligibility trace
%     - Gamma = Weighting of expected-best outcome in backup
%     - Epsilon = Greediness of action selection policy
%     - R = Magnitude of state reduction for high-level agent.
%           - e.g. if Size.X = 25 and Size.Y = 25 and R = 5,
%             the high level agent would view the environment
%             as a 5 x 5 area.
% For Type 'HQ' : Parameters2 = [Alpha, Lambda, Gamma, Epsilon, F]'
%     These parameters are for the low-level agent
%     - Alpha = Learning rate
%     - Lambda = Decay constant for eligibility trace
%     - Gamma = Weighting of expected-best outcome in backup
%     - Epsilon = Greediness of action selection policy
%     - F = size of feature detector for low-level agent.

if(Type == 'QL')
    P.Type = 'QL';
    P.Size = Size;
    P.State.X = 1;
    P.State.Y = 1;
    P.OldState = P.State;
```

```

        P.V = zeros(Size.X, Size.Y, 4);      % Initialize Value Function
        P.e = zeros(Size.X, Size.Y, 4);      % Initialize Eligibility Trace
    P.Alpha = Parameters(1);
    P.Lambda = Parameters(2);
    P.Gamma = Parameters(3);
    P.Epsilon = Parameters(4);
    P.Action = 0;
    P.OldAction = 0;
    P.Reward = 0;
end

if(Type == 'FQ')
    P.Type = 'FQ';
    P.Size = Size;
    P.State.X = 1;
    P.State.Y = 1;
    P.OldState = P.State;
        P.V = zeros(Size.X, Size.Y, 4);      % Initialize Value Function
        P.e = zeros(Size.X, Size.Y, 4);      % Initialize Eligibility Trace
    P.f = ones(Size.X, Size.Y, 4); % Forgetting trace
    P.Alpha = Parameters(1);
    P.Lambda = Parameters(2);
    P.Gamma = Parameters(3);
    P.Epsilon = Parameters(4);
    P.Mu = Parameters(5);
    P.Action = 0;
    P.OldAction = 0;
    P.Reward = 0;
end

if(Type == 'FB')
    P.Type = 'FB';
    P.Size = Size;

    P.Alpha = Parameters(1);
    P.Lambda = Parameters(2);
    P.Gamma = Parameters(3);
    P.Epsilon = Parameters(4);
    P.FeatureSize = Parameters(5);

    P.State = zeros(P.FeatureSize);
    P.OldState = P.State;

    P.NumStates = 2^(P.FeatureSize^2);      % Number of states
    P.F = zeros(P.NumStates, 4);          % Value function
    P.Fe = zeros(P.NumStates, 4); % Eligibility trace

    P.Action = 0; % Action
    P.OldAction = 0;
    P.Reward = 0; % Reward

    % Direction from agent that feature map is obtained from
    % 0 = centered on agent

    P.Direction = 0;
end

```

```

if(Type == 'HQ')
    P.Type = 'HQ';
    P.Size = Size;
    P.State.X = 1;
    P.State.Y = 1;
    P.Reduction = Parameters(5);

    % Calculate Size of State space for high-level agent

    RSize.X = ceil(P.Size.X/P.Reduction);
    RSize.Y = ceil(P.Size.Y/P.Reduction);

    % Initialize High-Level Agent

    HP = Parameters(1:4);
    P.High = InitAgent('QL', RSize, HP);

    % Initialize Low-Level Agent

    P.Low = InitAgent('FB', Size, Parameters2);

    P.Action = P.Low.Action; % Movement (output of low-level agent)
    P.Low.Direction = P.High.Action; % Desired direction of travel
    P.OldAction = P.Action;
    P.Reward = 0; % Reward to high-level agent
    P.High.Reward = P.Reward;
end

```

A.2.5 StartTrial

```
function [P, E] = StartTrial(P, E)
% function [P, E] = StartTrial(P, E)
% P = Agent data structure      (from InitAgent)
% E = Environment data structure (from InitEnvironment)
% This function initializes one trial of an RL agent
% First, the environment is initialized, then the agent.

if(E.Type == 'S') % Static Environment
    M = E.Map;
    SL = (M == 3); % Start location
    NumSL = sum(sum(SL));

% Multiple possible start locations, choose one at random

    if(NumSL >= 1)
        [XC, YC, Temp] = FindMax(SL);
        choose = floor(size(XC,1)*rand(1) + 1);
        XS = XC(choose);
        YS = YC(choose);
    else
% No start location specified, choose one anywhere without an obstacle
        SL = (M == 0);
        NumSL = sum(sum(SL));
        [XC, YC, Temp] = FindMax(SL);
        choose = floor(size(XC,1)*rand(1) + 1);
        XS = XC(choose);
        YS = YC(choose);
    end

    E.State.X = XS;
    E.State.Y = YS;
end

if(E.Type == 'D') % Dynamic Environment
    E.EpisodeCount = E.EpisodeCount + 1;
    EC = E.EpisodeCount;
    DP = E.DynPeriod;

% Dynamic period has elapsed, change environment

    if((EC/DP) == floor(EC/DP))
        E.Map = RandomMap(E.Size, E.Density, E.Start, E.Goal);
    end

    M = E.Map;
    SL = (M == 3);
    NumSL = sum(sum(SL));
    if(NumSL >= 1)
        [XC, YC, Temp] = FindMax(SL);
        choose = floor(size(XC,1)*rand(1) + 1);
        XS = XC(choose);
        YS = YC(choose);
    else
        SL = (M == 0);
    end
end
```



```

    NumSL = sum(sum(SL));
    [XC, YC, Temp] = FindMax(SL);
    choose = floor(size(XC,1)*rand(1) + 1);
    XS = XC(choose);
    YS = YC(choose);
end
E.State.X = XS;
E.State.Y = YS;
end

% Initialize Agent to be at start location
% Also, choose agent's first action

if(P.Type == 'QL')
    P.State = E.State;
    P.OldState = E.State;
    P.Action = EGreedy(P.V, P.State, P.Epsilon);
end

if(P.Type == 'FQ')
    P.State = E.State;
    P.OldState = E.State;
    P.Action = FGreedy(P.V, P.State, P.Epsilon);
    P.V = P.V .* P.f;    % Forgetting
    P.f = ones(size(P.V)) * P.Mu;
end

if(P.Type == 'FB')
    P.Direction = 0;
    P.State = DecodeState(E.State, E, P);
    P.OldState = P.State;
    P.Action = EGreedy(P.F, P.State, P.Epsilon);
end

if(P.Type == 'HQ')
    P.State = E.State;
    P.OldState = E.State;
    P.High.State = ReduceState(E.State, P.Reduction);
    P.High.OldState = P.High.State;
    P.High.Action = EGreedy(P.High.V, P.High.State, P.High.Epsilon);
    P.Low.Direction = P.High.Action;
    P.Low.State = DecodeState(P.State, E, P.Low);
    TempState.X = DecodeFeature(P.Low.State);
    TempState.Y = 1;
    P.Low.Action = EGreedy(P.Low.F, TempState, P.Low.Epsilon);
    P.Action = P.Low.Action;
end

```

A.2.6 FindMax

```
function [I, J, Val] = FindMax(A);  
% I and J are vectors containing Row and Column indices of the location  
% of the maximal elements of A  
% Val is the value of the maximal element(s)  
Val = max(max(A));  
[I,J] = find(A == Val);
```

A.2.7 EGreedy, FGreedy, and HGreedy

These three functions all calculate the epsilon-greedy action based on a value function.

The three versions are necessary due to differences in representation of the value function.

```
function A = EGreedy(V, State, E)
%   A = EGreedy(Q, State, E)
%   A = Action selected by Epsilon-Greedy Policy over Q
%   V = State/Action Value Function
%   State = Current State
%   E = Greediness (0 = Greedy, 1 = Random)

NumActions = size(V,3);
Prob = rand(1);
if(Prob < E)      % Choose Random Action
    A = floor(NumActions*rand(1) + 1);
else
    [M, A] = max(V(State.X,State.Y, :));
end

function A = FGreedy(V, State, E)
%   A = EGreedy(Q, State, E)
%   A = Action selected by Epsilon-Greedy Policy over Q
%   V = State/Action Value Function
%   State = Current State
%   E = Greediness (0 = Greedy, 1 = Random)

NumActions = size(V,3);
Prob = rand(1);
if(Prob < E)      % Choose Random Action
    A = floor(NumActions*rand(1) + 1);
else
    [M, A] = min(V(State.X,State.Y, :));
end

function A = HGreedy(V, State, E)
%   A = EGreedy(Q, State, E)
%   A = Action selected by Epsilon-Greedy Policy over Q
%   V = State/Action Value Function
%   State = Current State
%   E = Greediness (0 = Greedy, 1 = Random)

NumActions = size(V,3);
Prob = rand(1);
if(Prob < E)      % Choose Random Action
    A = floor(NumActions*rand(1) + 1);
else
    [M, A] = max(V(State.X, :));
end
```

A.2.8 StepEnv

```
function [E, Reward, NewState] = StepEnv(E, Action)
% [E, Reward, NewState] = StepEnv(E,Action)
% Single step RL environment based on agent's action
% Inputs:
%   E = Current Environment
%   Action = Agent's action
% Outputs:
%   E = Environment following action
%   Reward = Agent's reward for action
%   NewState = agent's state resulting from action

[I,J] = DecodeAction(Action);

M = E.Map;

XP = E.State.X; %Current X Location
YP = E.State.Y; %Current Y Location

XN = XP + I; % New X Position
YN = YP + J; % New Y Position

if((XN < 1) | (YN < 1) | (XN > E.Size.X) | (YN > E.Size.Y)) % Tried to
leave edge of map
    Reward = E.ObsPen;
    XN = XP;
    YN = YP;
else
    if(M(XN,YN) == 1) % Ran into obstacle
        Reward = E.ObsPen;
        XN = XP;
        YN = YP;
    else
        if(M(XN,YN) == 2) % Reached Goal
            Reward = E.GoalRew;
        else % Normal Movement
            Reward = E.StepPen;
        end
    end
end

E.State.X = XN;
E.State.Y = YN;
NewState = E.State;
```

A.2.9 DecodeAction

```
function [I,J] = DecodeAction(Action)
% [I,J] = DecodeAction(Action)
%   I = North/South movement
%   J = East/West movement

I = 0;
J = 0;

if(Action == 1)    % Move Up / North
    I = -1;
    J = 0;
end

if(Action == 2)    % Move Right / East
    I = 0;
    J = 1;
end

if(Action == 3)    % Move Down / South
    I = 1;
    J = 0;
end

if(Action == 4)    % Move Left / West
    I = 0;
    J = -1;
end

if(Action == 0)    % No Action
    I = 0;
    J = 0;
end
```

A.2.10 StepAgent

```
function P = StepAgent(P, NewState, Reward, E)
% P = StepAgent(P, NewState, Reward, E)
% Inputs:
%   P = current state of agent
%   NewState = State resulting from agent's previous action
%   Reward = Reward resulting from agent's previous action
%   E = Environment data structure
% Outputs:
%   P = resulting state of agent

if(P.Type == 'QL')

    % Epsilon-Greedy Action
    ANew = EGreedy(P.V, NewState, P.Epsilon);

    % Pure Greedy Action
    [MaxVal, AStar] = max(P.V(NewState.X, NewState.Y, :));

    % If the greedy action has the same value as the chosen action,
    % update AStar to reflect that
    if(P.V(NewState.X, NewState.Y, AStar) == P.V(NewState.X, NewState.Y,
        ANew))
        AStar = ANew;
    end

    % Calculate change to value function based on reward
    Delta = Reward + (P.Gamma * P.V(NewState.X, NewState.Y, AStar)) -
        P.V(P.State.X, P.State.Y, P.Action);

    % Update eligibility trace for most recent action
    P.e(P.State.X, P.State.Y, P.Action) = P.e(P.State.X, P.State.Y,
        P.Action) + 1;

    % Update value function
    P.V = P.V + (P.Alpha)*Delta*(P.e);

    % If action was greedy, update eligibility trace, otherwise clear it
    if (ANew == AStar)
        P.e = (P.Gamma)*(P.Lambda)*(P.e);
    else
        P.e = zeros(size(P.V));
    end

    P.OldState = P.State;
    P.State = NewState;
    P.Action = ANew;
end

if(P.Type == 'FQ')

    % Epsilon-Greedy Action
    ANew = FGreedy(P.V, NewState, P.Epsilon);
```

```

% Pure Greedy Action
[MaxVal, AStar] = min(P.V(NewState.X, NewState.Y, :));
if(P.V(NewState.X, NewState.Y, AStar) == P.V(NewState.X, NewState.Y,
ANew))
    AStar = ANew;
end

Delta = Reward + (P.Gamma * P.V(NewState.X, NewState.Y, AStar)) -
P.V(P.State.X, P.State.Y, P.Action);
P.e(P.State.X, P.State.Y, P.Action) = P.e(P.State.X, P.State.Y,
P.Action) + 1;
P.f(P.State.X, P.State.Y, P.Action) = 1;

P.V = P.V + (P.Alpha)*Delta*(P.e);

if (ANew == AStar)
    P.e = (P.Gamma)*(P.Lambda)*(P.e);
else
    P.e = zeros(size(P.V));
end

P.OldState = P.State;
P.State = NewState;
P.Action = ANew;
end

if(P.Type == 'FB')
    FeatureNumber = DecodeFeature(NewState);
    OldFN = DecodeFeature(P.State);
    TState.X = FeatureNumber;
    TState.Y = 1;

% Epsilon-Greedy Action
ANew = HGreedy(P.F, TState, P.Epsilon);

% Pure Greedy Action
[MaxVal, AStar] = max(P.F(FeatureNumber, :));
if(P.F(FeatureNumber, AStar) == P.F(FeatureNumber, ANew))
    AStar = ANew;
end

Delta = Reward + (P.Gamma * P.F(FeatureNumber, AStar)) - P.F(OldFN,
P.Action);
P.Fe(OldFN, P.Action) = P.Fe(OldFN, P.Action) + 1;

P.F = P.F + (P.Alpha)*Delta*(P.Fe);

if(ANew == AStar)
    P.Fe = (P.Gamma)*(P.Lambda)*(P.Fe);
else
    P.Fe = zeros(size(P.F));
end

P.OldState = P.State;
P.State = NewState;
P.Action = ANew;
end
end

```

```

if(P.Type == 'HQ')
    NewHigh = ReduceState(NewState, P.Reduction);

    % Execute learning for high-level agent only if high-level state has
    % changed

    if((NewHigh.X ~= P.High.State.X) | (NewHigh.Y ~= P.High.State.Y))
        G = GetGoal(E);
        RedGoal = ReduceState(G, P.Reduction);
        if((NewHigh.X == RedGoal.X) & (NewHigh.Y == RedGoal.Y))
            Reward = E.GoalRew;
        end

        P.High = StepAgent(P.High, NewHigh, Reward, E);
        P.Low.Direction = P.High.Action;
        % Bonus given to low-level agent if new high level state is more
        % desirable than old one
        LowRewardMod = max(P.High.V(NewHigh.X, NewHigh.Y, :)) -
            max(P.High.V(P.High.State.X, P.High.State.Y, :));
    else
        LowRewardMod = 0;
    end

    % Calculate Reward Given to Low-Level Agent

    LowReward = 5*Reward + (P.Low.Direction == P.Low.Action) +
        LowRewardMod;

    % Execute Learning for Low-Level agent

    NewLow = DecodeState(NewState, E, P.Low);
    P.Low = StepAgent(P.Low, NewLow, LowReward, E);

    P.OldState = P.State;
    P.State = NewState;
    P.Action = P.Low.Action + P.Low.Direction - 1;
end

```


A.2.11 ReduceState

```
function RS = ReduceState(State, R)

% Returns reduced state value for use in hierarchical
% RL Agent

RS.X = ceil(State.X/R);
RS.Y = ceil(State.Y/R);
```

A.2.12 DecodeState

```
function S = DecodeState(NewState, E, P)

% S = DecodeState(NewState, E, P)
% Returns decoded state perceptions for an RL agent
% The exact form of the return value S depends on the type of Agent
% contained in the structure P.
% If P is a standard Q-Learning agent (P.Type = 'QL'), or a
% forgetting Q-Learning agent (P.Type = 'FQ'), then the state
% merely contains the X and Y position of the agent :
%         S.X = Horizontal position
%         S.Y = Vertical position
% If P is a feature based Q-Learning agent (P.Type = 'FB'), then
% the state is the area around the agent of the size determined
% by the agent's FeatureSize parameter

if((P.Type == 'QL') | (P.Type == 'FQ'))
    S = NewState;
end

if(P.Type == 'FB')
    S = GetArea(E, NewState, P.Direction, P.FeatureSize);
end
```

A.2.13 GetArea

```
function A = GetArea(E, State, Direction, FSize)

% A = GetArea(Environment, State, Direction, FeatureSize)

% Insure that FSize is an odd integer

FSize = 1 + 2 * floor(FSize/2);

% Determine direction for feature map
[I,J] = DecodeAction(Direction);

xs = size(E.Map, 1);
ys = size(E.Map, 2);

% Create a temporary environment map that contains a border large enough
% to encompass the entire feature map
% Any states outside the border of the actual map are padded with ones.

TempMap = ones(xs + (2*FSize), ys + (2*FSize));

% These four numbers are the x and y borders of the actual map within
% the larger temporary map

xbl = FSize + 1;
xbh = FSize + xs;
ybl = FSize + 1;
ybh = FSize + ys;

TempMap (xbl:xbh, ybl:ybh) = E.Map;

% Remove Start and Goal Locations from temporary map

SL = (TempMap == 3); % Start Location
TempMap = TempMap - SL*3;

GL = (TempMap == 2); % Goal Location
TempMap = TempMap - GL*2;

% Determine area within temporary map that corresponds to desired
% feature map

if(I == 0)
    xbl = State.X - floor(FSize/2) + FSize;
    xbh = State.X + floor(FSize/2) + FSize;
end
if(I == 1)
    xbl = State.X + 1 + FSize;
    xbh = State.X + FSize + FSize;
end
if(I == -1)
    xbl = State.X - FSize + FSize;
    xbh = State.X - 1 + FSize;
end
if(J == 0)
```

```

    ybl = State.Y - floor(FSize/2) + FSize;
    ybh = State.Y + floor(FSize/2) + FSize;
end
if(J == 1)
    ybl = State.Y + 1 + FSize;
    ybh = State.Y + FSize + FSize;
end
if(J == -1)
    ybl = State.Y - FSize + FSize;
    ybh = State.Y - 1 + FSize;
end

A = TempMap(xbl:xbh,ybl:ybh);

% Rotate selection so it is aligned with Direction
NumRots = Direction - 1;
for i = 1:NumRots
    A = ROT90(A);
end

```

A.2.14 ShowMap

```
function H = ShowMap(E);

M = E.Map;

% First, Remove Start Positions from Map (Value == 3)

SP = (M == 3);
M = M - 3*SP;

% Next, Set Value of Current Agent Location to 3

XP = E.State.X; %Current X Location
YP = E.State.Y; %Current Y Location

M(XP,YP) = 3;
M = M + 1; % Offset, so colormap works ocrrectly;

% Colormap

CM = [[1 1 1];[0 0 0];[0 1 0];[0 0 1]];
image(M);
colormap(CM);
axis off;
H = GCF;
```

A.2.15 GetGoal

```
function G = GetGoal(E)

% G = GetGoal(Environment)
% Returns the goal location in a structure containing G.X and G.Y

Map = E.Map;
GL = (Map == 2);
[G.X,G.Y] = find(GL == max(max(GL)));
```

A.3 Example Scripts

This section presents three experimental scripts that show how to use the functions presented above to perform experiments. One example is presented for each type of RL agent.

A.3.1 Standard Q-Learning Example

```
NumEpisodes = 200;      % Number of Episodes
MaxSteps = 5000;       % Maximum time per episode
EpisodeNum = 1;        % Starting Episode Number

AvgSteps = zeros(NumEpisodes,1); % Data Log
NumTrials = 1000; % Number of Trials to perform

NumSteps = zeros(NumTrials,NumEpisodes,1);

density = 0.20; % Environment obstacle density
S.X = 32; % Width of environment
S.Y = 32; % Height of environment
Map = RandomMap(S, density);

for trial = 1:NumTrials
    disp(sprintf('Trial %d', trial));
    EP = [-5, -1, 1000]'; % Environment Parameters
    QP = [0.7 0.5 0.9 0.05]'; % Agent Parameters

    E = InitEnvironment('S', Map, EP);
    OptLength = OptimalDist(E);
    Q = InitAgent('QL', E.Size, QP);

    for EpisodeNum = 1:NumEpisodes
        [Q, E] = StartTrial(Q, E);
        A = Q.Action;
        EndEpisode = 0;
        while(EndEpisode == 0)
            [E, Reward, NewState] = StepEnv(E, A);
            Q = StepAgent(Q, NewState, Reward, E);
            A = Q.Action;
            G = GetGoal(E);
            if((Q.State.X == G.X) & (Q.State.Y == G.Y))
                EndEpisode = 1;
            end
            NumSteps(trial, EpisodeNum) = NumSteps(trial,EpisodeNum) + 1;
            if (NumSteps(trial,EpisodeNum) > MaxSteps)
                EndEpisode = 1;
            end
        end %while
        disp(sprintf('Episode %d: %d Steps', EpisodeNum, NumSteps(trial,
EpisodeNum)));
    end %for episode
end %for trial
for j=1:NumEpisodes
    AvgSteps(j) = sum(NumSteps(:,j))/NumTrials;
end %for j
```


A.3.2 Forgetting Q-Learning Example

```
NumEpisodes = 200;      % Number of Episodes
MaxSteps = 5000;       % Maximum time per episode
EpisodeNum = 1;        % Starting Episode Number

AvgSteps = zeros(NumEpisodes,1); % Data Log
NumTrials = 1000;      % Number of trials to perform

NumSteps = zeros(NumTrials,NumEpisodes,1); % Data Log

density = 0.20; % Environment Obstacle Density
S.X = 32; % Environment Size
S.Y = 32;
Map = RandomMap(S, density); % Generate environment map

for trial = 1:NumTrials
    disp(sprintf('Trial %d', trial));
    EP = [5, 1, -1000]'; % Environment parameters
    QP = [0.7 0.5 0.9 0.05 0.95]'; % Agent parameters

    E = InitEnvironment('S', Map, EP);
    OptLength = OptimalDist(E);
    Q = InitAgent('FQ', E.Size, QP);

    for EpisodeNum = 1:NumEpisodes
        [Q, E] = StartTrial(Q, E);
        A = Q.Action;
        EndEpisode = 0;
        while(EndEpisode == 0)
            [E, Reward, NewState] = StepEnv(E, A);
            Q = StepAgent(Q, NewState, Reward, E);
            A = Q.Action;
            G = GetGoal(E);
            if((Q.State.X == G.X) & (Q.State.Y == G.Y))
                EndEpisode = 1;
            end
            NumSteps(trial, EpisodeNum) = NumSteps(trial,EpisodeNum) + 1;
            if (NumSteps(trial,EpisodeNum) > MaxSteps)
                EndEpisode = 1;
            end
        end %while
        disp(sprintf('Episode %d: %d Steps', EpisodeNum, NumSteps(trial,
EpisodeNum)));
    end %for episode
end %for trial
for j=1:NumEpisodes
    AvgSteps(j) = sum(NumSteps(:,j))/NumTrials;
end %for j
```

A.3.3 Hierarchical Q-Learning Example

```
NumEpisodes = 200;      % Number of episodes
MaxSteps = 5000;       % Maximum time per episode
EpisodeNum = 1;        % Starting episode number

AvgSteps = zeros(NumEpisodes,1); % Data log
NumTrials = 1000;      % Number of trials to perform

NumSteps = zeros(NumTrials,NumEpisodes,1); % Data Log

density = 0.20; % Environment obstacle density
S.X = 32; % Environment size
S.Y = 32;
Map = RandomMap(S, density); % Generate environment map

for trial = 1:NumTrials
    disp(sprintf('Trial %d', trial));
    EP = [-5, -1, 1000]'; % Environment parameters
    QP1 = [0.7 0.5 0.9 0.05 5]'; % High-Level Agent Parameters
    QP2 = [0.7 0.5 0.9 0.05 3]'; % Low-Level Agent Parameters

    E = InitEnvironment('S', Map, EP);
    OptLength = OptimalDist(E);
    Q = InitAgent('HQ', E.Size, QP1, QP2);

    for EpisodeNum = 1:NumEpisodes
        [Q, E] = StartTrial(Q, E);
        A = Q.Action;
        EndEpisode = 0;
        while(EndEpisode == 0)
            [E, Reward, NewState] = StepEnv(E, A);
            Q = StepAgent(Q, NewState, Reward, E);
            A = Q.Action;
            G = GetGoal(E);
            RedGoal = ReduceState(G, Q.Reduction);
            RedAct = ReduceState(Q.State, Q.Reduction);
            if((RedAct.X == RedGoal.X) & (RedAct.Y == RedGoal.Y))
                EndEpisode = 1;
            end
            NumSteps(trial, EpisodeNum) = NumSteps(trial,EpisodeNum) + 1;
            if (NumSteps(trial,EpisodeNum) > MaxSteps)
                EndEpisode = 1;
            end
            %ShowMap(E);
            %pause(0.05);
        end %while
        disp(sprintf('Episode %d: %d Steps', EpisodeNum, NumSteps(trial,
        EpisodeNum)));
    end %for episode
end %for trial
for j=1:NumEpisodes
    AvgSteps(j) = sum(NumSteps(:,j))/NumTrials;
end %for j
```

VITA ²

Travis W. Hickey

Candidate for the Degree of

Master of Science

Thesis: REINFORCEMENT LEARNING ALGORITHMS FOR ROBOTIC
NAVIGATION IN DYNAMIC ENVIRONMENTS

Major Field: Electrical Engineering

Biographical:

Personal Data: Born in Independence, Kansas, On January 13, 1976, the son of
Carla and Jeffrey Hickey.

Education: Graduated from Enid High School, Enid, Oklahoma with honors in
May 1994; received Bachelor of Science degree in Computer Engineering
from Oklahoma State University, Stillwater, Oklahoma in May 1999.
Completed requirements for Masters of Science degree with a major in
Electrical Engineering at Oklahoma State University in December 2001.

Experience: Employed as a software engineer at Frontier Electronic Systems,
Stillwater, Oklahoma, 1998 to present.

Professional Memberships: Institute of Electrical and Electronics Engineers
(IEEE)

