

**OPTIMAL MOVEMENT THROUGH A MAZE
BY AN APPLICATION OF Q LEARNING**

Lu Han

Bachelor of Engineering
Computer Engineering



Department of Electronic Engineering
Macquarie University

June 06, 2016

Supervisor: Associate Professor Dr. Sam Reisenfeld


ACKNOWLEDGMENTS

I would like to express my greatest gratitude to Associate Professor Dr. Sam Reisenfeld for his guidance and understanding throughout my thesis project process at Macquarie University.

STATEMENT OF CANDIDATE

I, Lu Han, declare that this report, submitted as part of the requirement for the award of Bachelor of Engineering in the Department of Computer Engineering, Macquarie University, is entirely my own work unless otherwise referenced or acknowledged. This document has not been submitted for qualification or assessment at any academic institution.

Student's Name: Lu Han

Student's Signature: 

Date: 06/06/2016

ABSTRACT

Machine learning has become a very important aspect of the development of artificial intelligence (AI) in the world of computing. As a part of the machine learning techniques, reinforcement learning enables an agent to explore and study the unknown environment and through trial and error, the agent learns how to perform desired tasks in an optimum way. In this project, one of the methods to solve reinforcement problems, namely Q-learning, is studied and analysed. A maze problem is setup to count the learning time of an agent to travel from a fixed starting point A to destination B using Q-learning algorithm on Matlab platform. The learning time will be studied and then compared with different sets of maze of different size and complexity. Finally, a thorough discussion on the differences in the learning time and methods to further optimise the Q-learning algorithm will be provided. This document reports the outcome of this project and highlights the problems and challenges encountered and solutions devised.

Contents

Acknowledgments	iii
Abstract	vii
Table of Contents	ix
List of Figures	xi
List of Tables	xiv
1 Introduction	1
1.1 Motivations	2
1.1.1 Challenges	2
1.2 Reinforcement Learning	3
1.2.1 Q-learning	4
1.3 Contributions	4
1.3.1 Project Outcomes and Impact	4
1.4 Thesis Overview	5
1.5 Project Plan	6
1.5.1 Project Objectives	9
1.6 Project Baseline Review	9
1.6.1 Time Budget Review	9
1.6.2 Financial Budget Review	9
2 Background theory	11
2.1 Markov Decision Process	11
2.2 Reinforcement Learning	16
2.2.1 Model-based Learning	18
2.2.2 Model-free Learning	18
2.3 Chapter Summary	23
3 Q-learning Algorithm	25
3.1 Action Selection	25
3.1.1 ε -Greedy	25

3.1.2	Softmax	26
3.2	Exploration vs Exploitation	26
3.2.1	Decay ϵ	26
3.2.2	Exploration function	27
3.2.3	Speedy Q-Learning	28
3.3	The Algorithm	29
3.4	Chapter Summary	30
4	Experiment Setup	31
4.1	Maze Design	31
4.2	Experiment Design	33
4.3	Mathematical QL Example	34
4.4	Matlab Coding	40
4.4.1	Simulation Run	40
4.4.2	GUI	49
4.5	Chapter Summary	50
5	Results and Analysis	51
5.1	Experiment 1	51
5.1.1	α	51
5.1.2	γ	56
5.1.3	ϵ	60
5.2	Experiment 2	64
5.2.1	Simulation Results	64
5.2.2	Discussion	66
5.3	Chapter Summary	68
6	Conclusions and Future Work	69
6.1	Conclusions	69
6.1.1	Q learning Theory	69
6.1.2	Matlab Program Design	69
6.1.3	Simulation Result Analysis	70
6.2	Future work	70
6.2.1	Optimisation on QL Algorithm	70
6.2.2	QL in Machine Learning	71
7	Abbreviations	73
A	Matlab Coding	75
A.1	Overview	75
A.2	Run file	75
A.3	GUI file	81

B Project Plan and Attendance Form	89
B.1 Overview	89
B.2 Consultation Meetings Attendance Form	89
Bibliography	91

List of Figures

2.1	A simple illustration of MDP [24]	12
2.2	Exponential decay of γ [12]	13
2.3	Discounting process [12]	13
2.4	A single transition in a MDP [12]	15
2.5	Reinforcement learning components [13]	17
2.6	Demonstration of the Q value update in a single transition [17]	21
4.1	A 3 x 3 Maze problem	31
4.2	A 10 x 10 Difficult Maze problem	32
4.3	The 3 sets of 5 x 5 maze for demonstration	32
4.4	A 2 x 2 Plain Maze	35
4.5	Graph representation of the 2 x 2 plain Maze	35
4.6	A 2 x 2 Hard Maze	38
4.7	Graph representation of the 2 x 2 Hard Maze	39
4.8	Matrix representation of a 2x2 Maze	42
4.9	Updated Transition Rules	43
4.10	Color Map	48
4.11	Sample simulation run	48
4.12	GUI design	49
4.13	Final GUI demo	50
5.1	$\alpha = 0.1$	52
5.2	Time vs α	53
5.3	Sample run when $\alpha = 0.1$	54
5.4	Sample run when $\alpha = 0.9$	55
5.5	$\gamma = 0.1$	56
5.6	Time vs γ	57
5.7	Sample run when $\gamma = 0.1$	58
5.8	Sample run when $\gamma = 0.9$	59
5.9	$\epsilon = 0.1$	60
5.10	Time vs ϵ	61
5.11	Sample run when $\epsilon = 0.1$	62
5.12	Sample run when $\epsilon = 0.9$	63

5.13	Sample run on maze 1	64
5.14	Sample run on maze 2	65
5.15	Sample run on maze 3	65
5.16	Color map Parula [15]	66
5.17	Comparison on maze 1	67
5.18	Comparison on maze 2	67
5.19	Comparison on maze 3	67

List of Tables

1.1	Time budget review summary	9
5.1	Time taken with varying α	52
5.2	Time taken with varying γ	57
5.3	Time taken with varying ϵ	61
5.4	Simulation results	66

Chapter 1

Introduction

Over the years, we have witnessed the dramatic changes in the development of computer. In the modern days, computers are not only just programmed to perform simple task such as doing calculations or word processing, but also designed to respond to its user in a more intuitive way that could assist or even play a very crucial role in the decision making process. This has become the biggest challenge in the design of Artificial Intelligence (AI) that is to make computer to perform tasks like a human being. This begins a new era of AI and Machine Learning (ML).

In the recent “Human vs Machine Go tournament” held in South Korea, the AI “AlphaGo” developed by the DeepMind team from Google has beaten one of the top Go player in the world with a final score of 4 games to 1. The board game Go [20] is an ancient game that requires either of the two players to use black or white stones to secure as much space as possible on a 19x19 board in order to win the game. The complexity of the board game Go is well known by its billions of move combinations and has generated huge interests among the mathematicians and computer scientists around the world. Due to the sophisticated playing strategy, it has been computational impossible to create an AI that can explicitly calculate each move in the game until the birth of the “AlphaGo”. The AI “AlphaGo” is designed to mimic our human brain by implementing algorithms from ML to estimate the possible best moves to play the game. It is an AI that not only can “think” which move to take, but also “learn” from each game it has played in the past to “gain experience” to improve itself.

Although the development of “AlphaGo” also involves the combination of technologies such as pattern recognition and neural network, the fundamental idea still lays in the ML [6] which has its emphasis in exploring all possible learning situations and by evaluating the different learning methods, the most effective method will be adopted into the machine design to allow machine to learn from experience rather than from explicit programming. [17, 27] .

The approach to an effective ML is called Reinforcement Learning (RL). It teaches

an agent to learn what to do and map the situations to actions in order to maximize the rewards accumulation [27]. However there are many techniques can be used to solve RL problems, such as dynamic programming (DP) and Q-Learning (QL). The purpose of this paper is to study the QL method and using Matlab to implement this method to solve a classic RL problem - the maze.

Section 2.2 and 2.3 of this document discusses in depth of the fundamental concepts of RL and QL and the most widely used algorithms to tackle learning problems in order to achieve an effective ML.

1.1 Motivations

The success of “AlphaGo” marks a huge breakthrough in AI development in human history [3, 21]. It also gives us an idea of the degree of impact to our life, if the AI can be programmed to learn to improve itself automatically with experience from every task it performed. In today’s world, everything is moving in a fast steady pace. Efficiency is the goal to every task, ranging from medical treatments to navigation, in order to keep ourselves with the flow. The key to efficiency is to optimise the way of doing things and this is often quite challenging due to the changes and uncertainties in each task we perform in our daily life.

This makes the research and development in AI with the emphasis on ML more meaningful, because computers are no longer just a tool for us to do calculations. It has proved its value that it could indeed become our most reliable assistant or even “friends”. Imagine computers can learn from bio-medical records which combinations of substances are most viable for fast synthesis to create medicines for new diseases treatment, or control system in ships or aircraft can learn from experience based on the weather or accident records to perform the most suitable manoeuvres in the case of emergency. Moreover, the process of developing an AI is also a process of thorough and better understanding of human’s learning abilities. This will help us to learn more about ourselves and unleash the potential within to bring us to the next level in terms of the development in science and technology.

1.1.1 Challenges

Designing a flawless and efficient AI proves to be a very challenging task. The complexity and uncertainty that are involved to model an event in real life requires more than just a simple learning algorithm. It needs a combination of technology in many areas, such as neural network, pattern recognition and data mining. The technology that we have in such disciplines are still at its infancy. Although we have seen some breakthroughs in the design of the leaning algorithms in ML, there are apparently a few challenges that require further research from the scientists in order to improve and maximise the efficiency in the

current algorithm.

Learning Time

The speed of the learning process is the top problem presents in the ML [10, 11]. It is like human being, some people can learning new things quickly, while some exhibit a slow progress in study. There are many variables that determines the learning time. But for machine, the more trials it takes to explore the unknown environment, the higher the chance it can produce a more accurate outcome (eg. Finding optimum path to solve a maze problem). The trade-off between exploration and exploitation is the main focus in the ML. The balance of each parameter needs further research so that the learning speed can be optimised.

Robotics Learning

In the area of software development, the ML algorithm has proven to be very mature, as we can see from the birth of the super game AI “AlphaGo” to smart voice recognition technology in our smart phone, such as “Siri” from Apple. However, the implementation of the algorithm into robotics is very challenging [33]. This is because, most of the ML algorithms focus on the discrete representation of the unknown environment, whereas robots exhibit a continuous feedback and output from their sensors and actuators. Hence, to build a flexible and intuitive autonomous robot needs a solution to the problem of how to effectively discretising the continuous world so that the learning algorithms can be fit into the ML context.

1.2 Reinforcement Learning

The fundamental principle of RL addresses the question of how an intelligent agent that senses and seeks accumulated rewards in an unknown environment can learn through experience from the exploration to find the optimal actions to achieve its goals. It also requires the agent to make use of the knowledge obtained from the past actions and rewards (both positive and negative) for the choice of action taken in the future. The balance between exploration and exploitation can be the critical factor in designing an accurate and efficient AI. As RL is an un-supervised learning method, it gives an advantage to tackle problems in a more stochastic world. In RL, the agent is not required to strictly follow any pre-set training examples, but instead, it learns everything through trial and error actively, and is able to continuously learn and update its knowledge while performing the task. This characteristic is very precious in solving many problems where the precise data is hard or even not possible to obtain in some circumstances.

1.2.1 Q-learning

QL is the most significant breakthrough in reinforcement learning. It is a model-free and off-policy technique widely used to solve RL problems. All the techniques in RL are with the purpose of finding the optimal policy in a bounded or an arbitrary environment, where the policy is a rule tells the learning agent which action to take at its current state. The classic methods, such as DP and Monte Carlo (MC) methods, usually computes the all existing states' values by averaging the sample data from each trial and the optimal policy is generated through learning and following the state with the highest value.

QL, on the other hand, takes a different approach by learning the action-value function which gives an expected values of action in each state. It then perform and favour an action with the highest value of the action over learning and comparing each individual state value. Eventually, an optimal policy will be produced by choosing to perform an action with the highest action value at each state.

The advantage of the QL is that the optimal policy is constructed by studying the action-value along the way rather than studying and predict the state value to pre-construct the optimal policy. Besides, QL can compare the expected utility of each possible action without using a fixed model. This give QL the advantage to solve problems relates to stochastic transition and rewards in an arbitrary environment.

1.3 Contributions

In this section, the project outcomes are outlined without much technical details. The section will also discuss the short and long-term impact of the thesis undertaken.

1.3.1 Project Outcomes and Impact

- A thorough literature review and explanation of the background theory is given.
- Mathematical examples and analysis of the QL algorithm are provided.
- Two of the existing Matlab code for the QL have been studied in order to familiarize with the Matlab coding environment.
- Customisation and modification have been made to the code to better suit the purpose of the project, which is to use Matlab to demonstrate the standard QL algorithm.
- A detailed explanation of the Matlab code for this project in the context of algorithm amendment, maze and experiment design is given.
- A Graphic User Interface (GUI) for generating experiment results and demonstration of the learning process is created. It has features such as, user prompt for setting

the values for all variables and numbers of samples to be taken for the experiment. It also allows user to load different sets of pre-designed maze with different size and complexity.

- The time taken for the learning agent to complete and work out the best route to the maze is simulated and analysed with different combinations of the variables α , γ and ϵ .
- Finally, a thorough discussion is made on the time variables relationship and how this can affect the learning time and ways to optimize the current QL algorithm are suggested.

All the works done in this thesis project contribute directly or indirectly to the efforts for studying and exploring possible ways to improve the current QL algorithm. In addition, this paper can serve as a viable and reliable detail literature with working GUI Matlab program for anyone who is interested in ML specifically in the area of QL.

1.4 Thesis Overview

In this section, a brief description of the whole project will be outlined. The project plan, specification and deliverables are listed in the section 1.5 of this document. This project is supervised by Associate Professor Dr. Sam Reisenfeld, in the Department of Engineering, Macquarie University. A regular meeting has been setup each week with my academic supervisor to discuss the progress and any problems encountered during the process, so that the project is in-line with the schedule and deliverables set in the project plan. The consultation meetings attendance form is listed in the Appendix B.2 of this document.

The organisation of the thesis are as follow:

Chapter 2 provides a thorough literature review and related background theory to this project. Topics discussed include Markov Decision Process (MDP), RL. In the RL section, the fundamental principle and application of the three of the most popular methods in reinforcement learning, DP, MC method and QL will be explained. In addition, mathematical examples relate to QL are also provided to illustrate the changes and convergence on the Q (state/action) matrix.

In chapter 3, an introduction is given to the biggest challenge of exploration versus exploitation in the RL and ways to balance these parameters. It follows by an explanation of the proposed QL algorithm that will be used to program in the Matlab for this project.

Chapter 4 provides a detailed explanation of the structure of the script in Matlab for the simulation and the GUI for this project. At the same time, the rationales behind the construction of the maze problem and experiments for studying the relationship among

the variables corresponds to the learning time are explained.

Results and discussions are provided in Chapter 5. This chapter discusses in detail the simulation results obtained using Matlab and comparison between the size, complexity of the maze and the learning time are also discussed.

Finally, Chapter 6 concludes and summarizes the thesis and follow by a discussion of the potential future research.

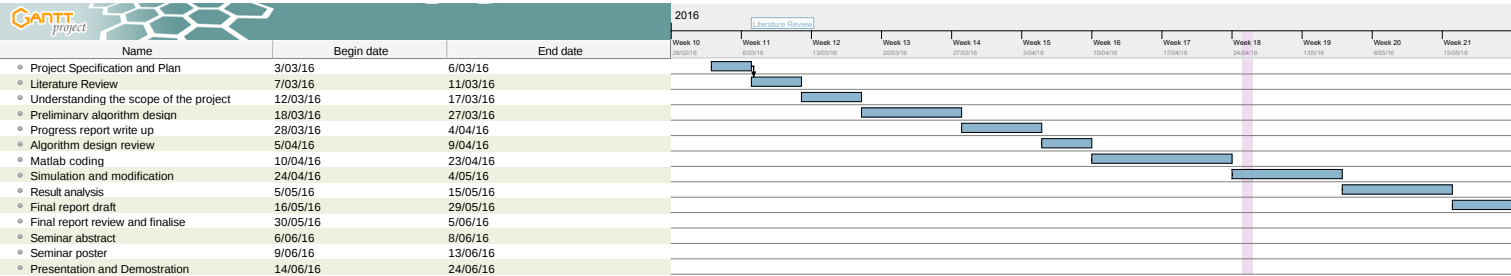
1.5 Project Plan

This section contains the project plan (Gantt Chart) depicting project specification and milestones.

Tasks

Name	Begin date	End date
Project Specification and Plan	3/03/16	6/03/16
Literature Review	7/03/16	11/03/16
Understanding the scope of the project	12/03/16	17/03/16
Preliminary algorithm design	18/03/16	27/03/16
Progress report write up	28/03/16	4/04/16
Algorithm design review	5/04/16	9/04/16
Matlab coding	10/04/16	23/04/16
Simulation and modification	24/04/16	4/05/16
Result analysis	5/05/16	15/05/16
Final report draft	16/05/16	29/05/16
Final report review and finalise	30/05/16	5/06/16
Seminar abstract	6/06/16	8/06/16
Seminar poster	9/06/16	13/06/16
Presentation and Demonstration	14/06/16	24/06/16

Gantt Chart



1.5.1 Project Objectives

The following section lists the project objectives briefly:

1. Familiarise with the MDP and RL fundamental theories
2. Familiarise with the Matlab programming platform
3. Customize and modify the existing QL algorithm to solve the proposed RL maze problems
4. Compare the learning time of the agent to complete the maze with different sets of maze of different complexity
5. Analyse and discuss the results with relevant plots in Matlab
6. Create a GUI for demonstration of simulation results

1.6 Project Baseline Review

The project was scheduled to be executed from 3rd March to 24th June, 2016. Baseline plan was generated with the intention of utilising all available days throughout the semester, including weekends and mid-semester break, for the project activities.

1.6.1 Time Budget Review

Table 1.1 summarizes the number of days spend on the project. It also indicates the percentage of the project has been completed so far. No amendments to the time schedule was required. Although the allocated time for each activity was followed closely, the order of accomplishment varied. Overall the progress of the project is in alignment with the project schedule planed in the beginning of the semester. Hence, there is no amendment necessary to the project.

Estimated work	114 days
Relised work	105 days
Completion	93 %

Table 1.1: Time budget review summary

1.6.2 Financial Budget Review

The project was initially allocated \$300, financial budget. However,the project does not require any hardware except Matlab software which a free student license is obtained from Macquarie University. Therefore, no further purchasing is needed and the budget surplus may not be utilised.

Chapter 2

Background theory

This chapter provides a thorough background on some of the topics related to this thesis and the existing research works that are relevant to this topic are also acknowledged. It is impractical to provide a comprehensive review on the current literature. The approach in this chapter mainly focused on the studies that have a strong connection to the thesis and relevant examples are given to illustrate an easier understanding of each learning process or algorithm and the importance of the role play with the variables. The concepts that are discussed in this chapter are mainly based on the literature from Sutton and Barto (1988) [27] and Tom M. Mitchell(1997). [17]

2.1 Markov Decision Process

Before we begin the journey to explore RL, a very important concept, namely the Markov Decision Process (MDP) is studied. MDP provides a fundamental understanding of RL. It is a convenient way to model a stochastic control process for decision making, so that the optimal way can be worked out. This is also the principle idea of solving any of the RL problem.

An MDP is defined by the following properties:

$\mathbf{s} \in \mathbf{S}$ a set of states;

$\mathbf{a} \in \mathbf{A}$ a set of actions;

$\mathbf{T}(\mathbf{s}, \mathbf{a}, \mathbf{s}')$ a state transition function, which specifies the probability of if starting from state \mathbf{s} took action \mathbf{a} and then land at states \mathbf{s}' ;

$\mathbf{R}(\mathbf{s}, \mathbf{a}, \mathbf{s}')$ a reward function, which specifies the immediate rewards obtained after transition from state \mathbf{s} to \mathbf{s}' .

γ a discount factor, which determines the importance of the future rewards and to assist agent to decide to choose current rewards or strive for rewards at later stages.

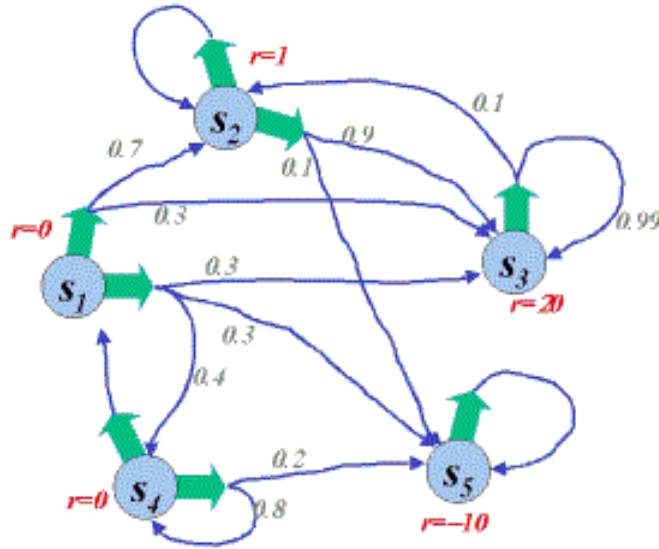


Figure 2.1: A simple illustration of MDP [24]

[4, 31]

As shown in figure 2.1, each circle represents the available state in the MDP and the green arrows indicates the actions can be performed at each state. One thing to highlight here is that, the green arrow is also a state called Q-state which is a state that is in a holding pattern where action is chosen, but the stochastic transition has not been made. This is a very important state in solving the MDP and will be discussed in detail in RL section of this report.

The values along the long arrows represents the transition function, which is the probability of landing into state s' by taking an action a . For example, if the current state is S1, the probability to go from S1 to S3, S4, and S5 by taking an action to move towards right is 0.3, 0.4, and 0.3 respectively.

The r with the values in red represents the immediate rewards received at the current state after a transition from the initial state. For example, at state S5, a transition to state S5 will incur a -20 rewards, whereas at state S3, the rewards to for going back to S3 is 20.

One parameter to highlight here is γ , the discount factor. This is one of the factor that varies during the process of accumulating the sum of rewards and it is further studied and analysed in Chapter 4 and 5 of this report. In the lecture given by Pieter Abbeel, from UC Berkeley, in 2013 on the topic of MDP, a very concise definition of the discounting factor γ is given and how this parameter is used in the process of accumulating the sum of rewards in solving a particular MDP problem.

The discount factor γ is a value ranging from 0 to 1. The assumption behind is that,



Figure 2.2: Exponential decay of γ [12]

the rewards decay exponentially. For instance, the value of the diamonds in figure 2.2 will not be the same in the future than now. As the diamond worth less, the value will get discounted exponentially as it does have an exponential growth. As illustrated in the figure, the first diamond is worth 1 at the present time. When it get visited the second time, the value will no longer be 1, but a discount factor γ , and for the next round γ^2 . The value of the γ is between 0 and 1, any degree of γ will lead to an exponential decay in the rewards.

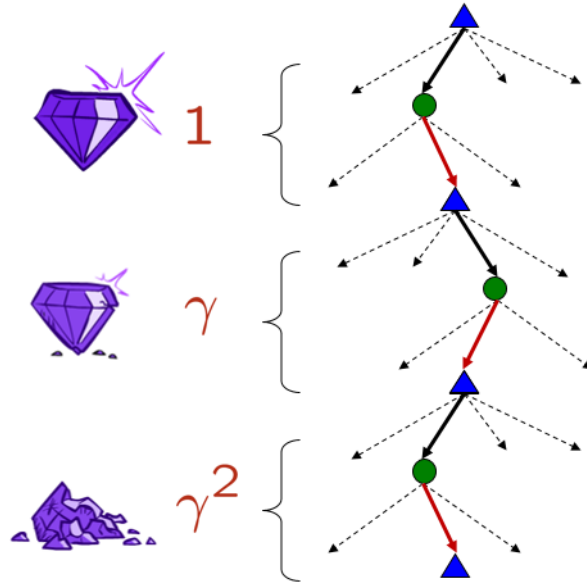


Figure 2.3: Discounting process [12]

Figure 2.3 shows the role played by γ in accumulating the rewards. The figure shows a 3-step transition tree. When we add the rewards step by step, the rewards are weighted by the discount factor as shown in the following equation:

$$\sum r = r_0 + r_1 * \gamma + r_2 * \gamma^2 + \quad (2.1)$$

For example, there are two policy π_1 and π_2 and both policy has 3 steps in the transition function. The first policy π_1 has a transition with the corresponding rewards 1, 2, 3, denoted by $U [1, 2, 3]$, whereas policy π_2 has a transition with the corresponding rewards 3, 2, 1, denoted by $U [3, 2, 1]$. The sum of the discounted rewards can be obtained by putting the values of the rewards at each state into equation 2.1. We have

$$\sum R_1 = 1 * 1 + 0.5 * 2 + 0.25 * 3 = 2.75$$

for policy π_1 and

$$\sum R_2 = 1 * 3 + 0.5 * 2 + 0.25 * 1 = 4.25$$

for policy π_2

Based on the sum of the rewards $\sum R_1 < \sum R_2$, we can say that policy π_2 is the preferred policy as it results in a higher accumulated rewards.

Since the aim is to maximize the sum of rewards over a sequence of actions, it is reasonable to prefer rewards now to rewards later. Hence, with high γ value, the agent is expected to favour rewards later and vice versa a low γ value will lead to take action that comes with higher rewards at the present state.

The Markov chain and Markov process were first introduced by Russian mathematician Andrey Markov in 1877, followed by the further development into MDP by American mathematician Richard Bellman in 1957 [2]. The principle idea of MDP is that, given a sequence of things and every indices is associated with each other corresponding to the time, so that whatever happen in the past is independent to what is going to happen in the future if the state at the current time is given. In the other words, each state is independent of any past environment states, rewards or actions took. This is represented by the following equation:

$$P(S_{t+1} = S' | S_t = s_t, A_t = a_t, S_{t-1} = s_{t-1}, \dots, S_0 = s_0) \quad (2.2)$$

for all the future state s' , r and all possible values of the past events $s_t, a_t, s_{t-1}, \dots, s_0$. [12]

MDPs are non-deterministic problems, because it models a stochastic process. The outcome of each action to take is unknown, hence to compute a sequence of actions is not possible. The only way is to find out the right action to take for every possible future state might be visited and a prescription of what are the right actions to take in each state. This prescription is known as the policy π . Hence the ultimate goal to solve MDP problems is to find the optimal policy, denoted by π^* , that maximizes the expected sum of rewards, which is represented as a mapping $\pi^* : S \rightarrow A$, where $\pi(s)$ is the action the agent takes in the state s .

Solving MDP

In order to solve a MDP, the following quantities are of our interest as shown in figure 2.2:

$V^*(s)$ is the expected utility starting in s and acting optimally, this is also the value function to calculate the state value which will be needed to construct the optimal policy;

$Q^*(s)$ is the expected utility starting out having taken action a from state s and acting optimally;

$\pi^*(s)$ is the optimal action from state s ;

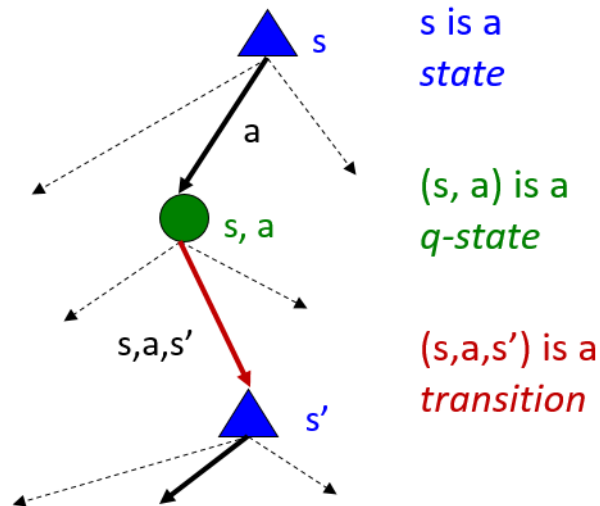


Figure 2.4: A single transition in a MDP [12]

$V^*(s)$ is a vector with an entry and a value for every possible state. It can be formulated as:

$$V^*(s) = \max_a Q^*(s, a) \quad (2.3)$$

[12]

where $\max_a Q^*(s, a)$ gives out the best sum of rewards can be obtained from the Q-state

$Q^*(s)$ is the expected sum of rewards that will be accumulate from here onwards by taking an action from state s , assuming it acts optimally from here onwards. This is can be formulated as:

$$Q^*(s, a) = \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')] \quad (2.4)$$

[27]

where we are at a chance node and do not know where we are going to land. We sum over all possible next S' as shown in the equation 2.3 and then wait for the probability to branch to the S' . While waiting for the branching, the rewards is sitting on the transition and there is a S' associated with it and we have $\gamma V^*(s')$ as it is a time step later before accumulating.

Since we have the equation for both V^* and Q^* , by substitute equation 2.3 into equation 2.2, we have an expression of V^* without utilising the value from Q^* :

$$V^*(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')] \quad (2.5)$$

[27]

This equation can be solved by using a method called value iteration. First, we start by setting the vector corresponds each state to be 0, $V_0(s) = 0$. Then we compute the next state value

$$V_{k+1} \leftarrow \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')] \quad (2.6)$$

[12]

It maximizes for the state s over different actions. Once an action is taken, it is weighted by $T(s, a, s')$ and at the same time count in the reward obtain in the transition $R(s, a, s')$ plus the value in the future discounted by a factor γ .

This function is also called the Bellman equation (Bellman, 1957). It is recursively defining V_{k+1} as a function of V_k in MDP. We can calculate V_k for any k and as k goes to infinity, the value of V_k converges to a fixed value, which is the value for each state. The optimal policy π^* can be found by following the highest state's value $V^*(s)$ at each state s .

2.2 Reinforcement Learning

Reinforcement learning is an approach to model an arbitrary environment and by trial and error, the learning agent learns to perform required tasks from experiences. The core idea is that the learning agent seeks rewards and aims to accumulate rewards by exploring the environment. At the same time, the agent also exploits the knowledge gained from the past and eventually take the optimal actions by finding the appropriate equilibrium between exploration and exploitation [28]. In RL, agent learns online and continually update its knowledge while performing the assigned tasks. The study of RL is very useful

in many circumstances as there are too many uncertainty in an arbitrary environment and it is very difficult or even impossible to find exact learning data to program a machine to react perfectly in such environment. [1] [25]

The history of RL can be traced back in the early 1950s [32]. It consists of many principles such as computer science, statistics and neuroscience. The basic idea of RL is very intuitive similar to animal learning method and it has been applied in many areas ranging from creating ultra-smart AI to play a chess game to maintaining the stability control in the power systems. [5]

The fundamental components of RL is shown in figure 2.3. In RL, the learning agent, who make the decision in the process, tries to perform a possible action in the arbitrary environment. The information of the state in the environment is also given. While taking the action, the rewards or punishment information is feedback to the agent and by processing these information, the agent learns to perform an action that can accumulate the maximum rewards in order to meet the goal corresponds to certain state in the environment [23].

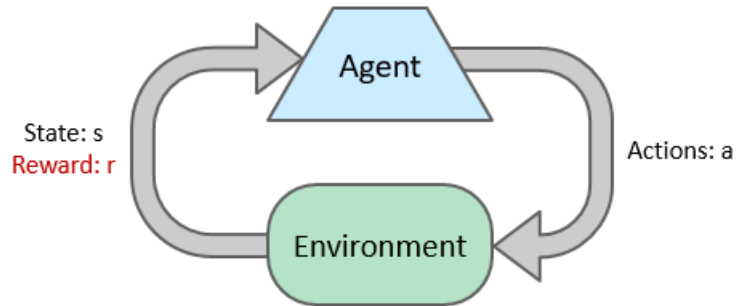


Figure 2.5: Reinforcement learning components [13]

RL model has very similar properties as MDP. It also has a set of state $s \in S$, a set of actions $a \in A$ per state and a goal to find the optimal policy π^* . However, the difference to MDP is that, the transition function $T(s, a, s')$, which provides information on the available actions in each state, and the rewards function $R(s, a, s')$, which provides information on the rewards of taking each action, are unknown in the start of the process. The idea is to let the agent to try all possible actions in each state and work out the rewards of each action in order for it to perform optimally.

Any arbitrary environment can be modelled as either a finite MDP or an infinite MDP. As we discussed in the previous section, in order to solve for a particular MDP problem, we need to know the five properties, state S , action A , transition function T , rewards function R and the discount factor γ . Since in RL, the transition function T and rewards

function are unknown, the general approach to solve RL problems is through policy estimation, in which we either estimate the state value function $V^\pi(s)$ or action value function $Q^\pi(s)$ for a given policy π to estimate the optimal policy. These two approaches are also known as Model-based (MB) and Model-free (MF).

2.2.1 Model-based Learning

In MB learning, we are assuming that there is a model with unknown parameters and from experience, we are going to estimate those parameters to build an estimated model. Then this estimated model can be used to work out an optimal policy to solve the RL problems. The learned model in this context is to learn the transition function $\hat{T}(s,a,s')$ and rewards function $\hat{R}(s,a,s')$. One of the very useful methods for the estimation used in MB learning is Dynamic Programming (DP).

Dynamic Programming

The term DP is a combination of algorithms to work out the optimum policy with a perfect model in a known or finite MDP. As a MB approach, DP utilises the sample of rewards received at each state and the probability of move from one state to another to approximate the rewards function \hat{R} and transition function \hat{T} by visiting each state again and again in the MDP [26]. The approximation process is illustrate below:

Given a set of samples v we collected after many iterations $[v, v_1, \dots, v_n]$, we can estimate the distribution for v by taking the number of times we see v in the samples divided by the total number of samples,

$$\hat{P}(v) = \frac{\text{num}(v)}{N}$$

, and once we have the approximated distribution, we can then estimate the expected v ,

$$E[V] \approx \sum_v \hat{P}(v) \cdot v$$

Along with this estimation, DP also uses a technique called bootstrapping to update the estimated values on the basis of other estimated values. One advantage of DP is that, each value estimates are constantly updated and stored are each iteration, which can be used later to estimate the transition function. However, one of the drawback is that DP requires learning taking place at every step over all states. Hence, it is impractical to model a MDP with large state spaces or an infinite MDP.

2.2.2 Model-free Learning

A more intuitive and flexible approach is to eliminate the need to learn from a model. The basic idea in MF learning is to estimate the state values v by directly averaging the

samples rather than from an estimation of a probability distribution of v .

Monte Carlo Methods

Unlike DP, MC methods does not require complete knowledge of the environment. It utilises the experience from the interaction with the environment, such as the sequence of states, actions and the rewards accumulated at each transition to estimate the state values based on the average sample taken. [14]

The process of MC methods can be devised using the same example in the previous section, where we have a sample of v values, the estimate of v value can be calculated by averaging these samples

$$E[V] \approx \frac{1}{N} \sum_i v_i \quad (2.7)$$

[13]

The rationale behind this approach is that given set of samples, value v in this case, that more likely to have a higher probability will show up more often in this sample set. Through sampling process, we automatically get more frequent occurrences of the more likely ones. Hence it eliminates the need to explicitly weight each sample data with the probability distribution.

From the Bellman equation 2.5 in the MDP section, we take samples of the outcome at state s by taking a action from a policy $\pi(s)$ after the transition from state s

$$sample_1 = R(s, \pi(s), s'_1) + \gamma V_k^\pi(s'_1)$$

$$sample_2 = R(s, \pi(s), s'_2) + \gamma V_k^\pi(s'_2)$$

....

$$sample_n = R(s, \pi(s), s'_n) + \gamma V_k^\pi(s'_n)$$

and by incorporating with the equation 2.6, the Bellman equation can be rewritten as

$$V_{k+1}^\pi(s) \leftarrow \frac{1}{n} \sum_a sample_i \quad (2.8)$$

[12]

As we can see that the transition function is eliminated in this equation which means the need to estimate a transition model is now unnecessary. This allows MC methods to perform in very highly dimensional spaces. However, the main problem MC methods is facing is the amount of exploration, that is, sufficient to cover the entire space. The trade-off between exploration and exploitation is a big issue with the techniques utilising sampling.

Temporal difference learning

With the knowledge of averaging the sample in MF approach, a new way of learning called temporal difference (TD) is introduced. The main idea is that the value function $V(s)$ is updated when a transition $T(s, a, s', r)$ occurs. By averaging the sample, the more frequent occurring s' will contribute more to this update. This will be in the sense that the agent is learning from very experience and might take several steps or actions before an action that incurs any rewards can be taken. [22]

Using equation 2.7, the process of the $V(s)$ update can be written as follow:

$$\text{Samples of } V(s): \text{sample} = R(s, \pi(s), s') + \gamma V^\pi(s')$$

$$\text{Update to } V(s): V^\pi(s) \leftarrow (1 - \alpha)V^\pi(s) + (\alpha)\text{sample}$$

$$\text{OR : } V^\pi(s) \leftarrow V^\pi(s) + \alpha(\text{sample} - V^\pi(s))$$

[12]

When a new experience comes in, the sample estimate of this new experience is illustrated by the first equation. This updates the current estimate V^π by averaging the sum of the sample estimate and current estimate with a weight α , shown in the second equation. The α is the factor ranging, from 0 to 1, that determines the importance of the sample estimates in updating the current estimate. For instance, when α is close to 1, the sample estimate contributes more to the current estimate update, whereas when α is small, the sample estimate becomes insignificant. This can be also shown in a different form in the third equation by rearranging the terms in the equation.

This $V(s)$ function update is similar to the bellman equation 2.5 described in the MDP section and can be solved using value iteration. However, this is a RL problem where a transition model is not specified. The method using here for sample averaging does not result in finding out the maximum value in the transition function $\max_a \sum_{s'} T(s, a, s')$, which give difficulties if we need to solve the state value $V(s)$ implicitly. One solution to this problem is to solve for the action-value $Q(s)$.

Q-Learning

The process of studying the state action function $Q(s)$ to solve for the given MDP is called Q-learning (QL). QL is a MF and off-policy method in solving RL problems. The big idea of QL is that the optimal policy can be estimated by performing the actions that have the highest value is each state. The essence of QL is that, it eliminates the need to estimate a transition policy for taking an action and gives the agent to chance to make choices for which action it is going to take.

From the MDP section we have equations 2.2 and 2.3:

$$V^*(s) = \max_a Q^*(s, a)$$

$$Q^*(s, a) = \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma V^*(s')]$$

As we are only interested in the Q values in QL, by substituting $V^*(s)$ in equation 2.3 with equation 2.2, we have

$$Q^*(s, a) = \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma \max_a Q^*(s, a)] \quad (2.9)$$

Since this is a MF approach, the $Q(s)$ estimates can be updated using sample averaging as follow:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + (\alpha)sample \quad (2.10)$$

[12]

This equation can also be written as:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(R(s, a, s') + \gamma \max_a Q^*(s, a)) \quad (2.11)$$

[12]

OR

$$Q(s, a) \leftarrow Q(s, a) + \alpha[R(s, a, s') + \gamma \max_a Q^*(s, a) - Q(s, a)] \quad (2.12)$$

This is beauty of the QL that it only needs a table with Q values and observe the state actions and rewards associate with it. Then constantly updates the entries of Q value in the table as we go.

In order to demonstrate how Q value is updated, a small example is given below to illustrate a single transition made by an agent, and the corresponding updates to the Q table.

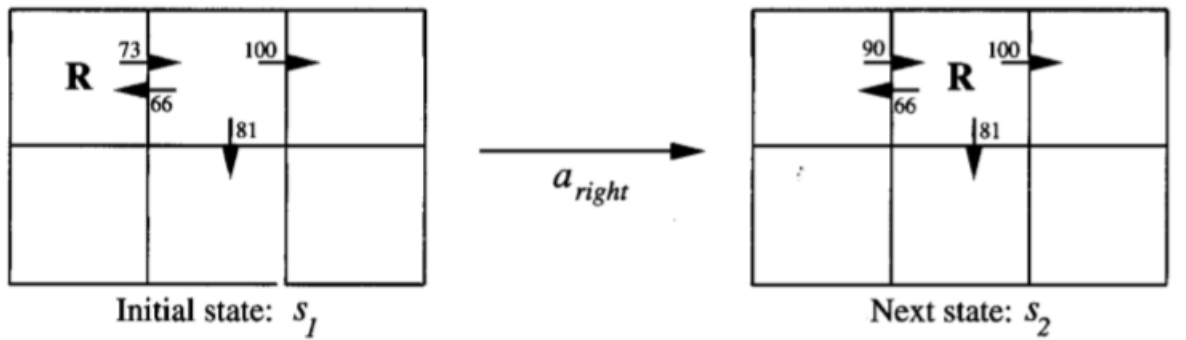


Figure 2.6: Demonstration of the Q value update in a single transition [17]

As shown in figure 2.6, the agent takes a move from the top left grid to the right top middle grid. In this particular case, the current state-action Q value corresponding to

the action taken are shown as the number next to the arrow. For example, the current action Q value for an agent to move from initial state s_1 to the next state s_2 is 73. The immediate rewards for this transition is set to be 0 and is weighted by a discount factor γ of 0.9. During the transition, the Q estimate Q^* for this action is updated by applying equation 2.9.

$$\begin{aligned} Q^*(s_1, a_r) &\leftarrow r + \gamma \max_a Q^*(s_2, a') \\ Q^*(s_1, a_r) &\leftarrow 0 + 0.9 \max_a [66, 81, 100] \\ Q^*(s_1, a_r) &\leftarrow 0 + 0.9 * 100 \\ Q^*(s_1, a_r) &\leftarrow 90 \end{aligned}$$

Hence the new Q estimate Q^* for this action is 90. This estimate is propagated backwards whenever the agent moves forward from the current state to the next state and the Q value is updated to a degree based on the learning rate α set for the particular MDP problem.

The Q update table will be converged when sufficient amount of sample has been taken. In the book “Machine Learning” published by Tom Mitchell (1997), a proof of the Q convergence is given. The assumption of this proof is based on an equal amount of visits of all states in a deterministic MDP with a bounded immediate rewards function. The fundamental idea behind is that the largest error in the Q updates table is reduced by a factor of γ each time the Q values are updated. This is because the new Q value depends partially on the possible error-prone Q estimates, while partially depends on the error-free immediate rewards. [17]

Theorem 1. *In a deterministic MDP, if the rewards $r(s, a)$ is bounded and each state action pair is visited infinitely often, then $Q_t^*(s, a)$ converges to $Q(s, a)$ as $n \rightarrow \infty$, for all s, a .*

Proof. Let Δ be the maximum error in Q_t

$$\Delta \equiv \max_{s,a} |Q_t^*(s, a) - Q(s, a)|$$

The magnitude of the error in the next Q estimate is

$$\begin{aligned} |Q_{t+1}^*(s, a) - Q(s, a)| &= |(r + \gamma \max_a Q_t^*(s', a')) - (r + \gamma \max_a Q(s', a'))| \\ &= \gamma |\max_a Q_t^*(s', a') - \max_a Q(s', a')| \end{aligned}$$

Since, the inequality holds

$$|\max f_1(x) - \max f_2(x)| \leq \max |f_1(x) - f_2(x)|$$

Hence,

$$\begin{aligned} |Q_t^*(s' a') - Q(s' a')| &\leq \gamma \max |Q_t^*(s' a') - Q(s' a')| \\ &\leq \gamma \Delta \end{aligned}$$

Each state is assumed to be visited infinitely often, as $t \rightarrow \infty$, the error $\Delta \rightarrow 0$, thus the convergence of Q is proved. □

One parameter to highlight here is the α , or also called the learning rate, which plays a very important role in converging the running averages in the $Q(s,a)$ function updates. As we mentioned in the TD learning, a decreasing α will lead to a converged Q values and hence more precise estimate of $Q(s,a)$ function. The relationship between the learning time and α will be further studied and discussed in the chapter 4 and 5 of this report.

2.3 Chapter Summary

In the introduction, we discussed about the role of RL in the development of the AI in the modern world and the various research that are going on in applying the RL methods into ML.

In this chapter, a thorough literature review and back ground theory explanation have been given. We firstly looked at the fundamentals of all learning method by explaining the concept of MDP and how various situation can be modelled by this framework.

Then we go on to explain the theory of RL by comparing different techniques such as DP and MC method in solving the MDP problems. In addition, we discussed the advantage and the drawbacks of these two methods before illustrates how the QL method is devised by combining DP and MC methods.

Finally, some numerical examples are given to illustrate the Q table updates and a simple proof of the Q table convergence is provided at the end of the chapter.

Chapter 3

Q-learning Algorithm

In this chapter, we are going to discuss the current dilemma in QL and some of the accomplished research work that focus on balancing the parameters in order to optimise the original QL algorithm. The detail QL algorithm that is used in the Matlab coding part of this project is provided at the end of this chapter.

3.1 Action Selection

As we discussed in the previous chapter, in QL, an agent needs to choose an action based on the state-action value Q corresponds its current state, before it can move to the next state. The question is, which action should it take? The biggest challenge here is how to the balance between exploration and exploitation. Whether to choose an action that associated with the highest Q value or explore more states to strive for long-term values? In order to achieve an effective QL, the agent not only needs to explore enough states for more options, but also make use of the values already in place.

Generally there are two types of methods for choosing an action:

3.1.1 ϵ -Greedy

This method favours the action associated with the highest estimate rewards, this is also known as the greedy action. The idea of this method is with a small probability of ϵ , a random action is chosen uniformly. This selected action is independent of the Q value estimate. In other words, the best action is chosen with probability ϵ and random uniform action is chosen with probability $1-\epsilon$. After a sufficient number of trials are executed, each action will be covered and the optimal actions can be found.

However, one of the drawback of this method is that the random actions are chosen uniformly. This might lead to a case that the worst action is selected as the second best action, which results in a longer learning time. [30]

3.1.2 Softmax

This method solves the problem in ε -greedy method by rank each action based on their corresponding Q value estimate. The random action is chosen based on the weight assigned to each action so that the worst actions are unlikely to be selected.

One of the Softmax techniques is Gibbs distribution. The probability distribution is shown as follow:

$$P(a|t) = \frac{e^{\frac{Q_{t-1}(a)}{\tau}}}{\sum_b e^{\frac{Q_{t-1}(b)}{\tau}}} \quad (3.1)$$

where τ is called the computational temperature. A higher τ will leads to a equal probability of choosing any action, whereas as τ goes to 0, the Softmax approach will have the same performance as the ε -Greedy policy.

[9]

3.2 Exploration vs Exploitation

As discussed in the previous section, there are both advantages and drawbacks on the two most common used method for action selection process in QL. In this section, three proposed methods from current research works are given to solve the problems in the action selection policy.

3.2.1 Decay ϵ

As the ϵ - Greedy method suggests, when ϵ is large, the selection of the action is random and when it is small, the action chosen will be mainly based on the best Q value. We have seen the drawbacks with this method, that is, it might be time consuming to work out an optimal policy even though all the states will be eventually explored. One simple way to solve this problem is to slowly let ϵ decay over time. The rationale behind is, in the beginning of the learning, the agent needs to explore as much states in the MDP as possible, so that no states with good rewards will be missed. However, as the learning progresses, the state action Q values slowly converge. This means that, there is no need for extensive exploration now. The smart way is to make use of the current Q values for the next action selection as the convergence of Q values is an indication of the better accuracy in the Q value estimates.

This method can be carried out in many forms. One of most straight forward ways, is to let $\epsilon_t = \frac{1}{1-t}$, where t is number of episodes completed in getting the agent to the goal state. The main idea behind is, ϵ is set to be a big number close to 1 in the beginning, so that the action chosen will be almost random. Once there are at least two trials completed, the action selection will change to favour the action based on the highest Q value. This will result in a fast convergence of Q values and hence a faster learning time.

3.2.2 Exploration function

The use of ϵ is still not the ideal way for balancing the exploration and exploitation. This is because, even by lowering ϵ over time, there may be one part of the state space has already been explored, but in the other part of the state space you have not explored much yet. Hence, the main focus should be to tune in how much exploration you do to where you are in the state space, rather than just have a global parameter ϵ just sending how much exploration anywhere you might be. Any state action the agent has barely seen needed to explore to find out what happens there. The theory is, when the agent have access to a state action pair that they have a lot of uncertainty about, it will assume is going to be good outcome and act according to that assumption, explore the uncertainty. This introduces the exploration function

$$f(u, n) = u + \frac{k}{n}$$

This function is going to modulate how much we are going to explore by keeping track of the visit count n , where n is the number of time that we have been in a particular state action pair which can be also denoted as $n(s, a)$ and k is a constant for the off-set. u is the utility function and in the case of Q updates, u will be the state action function $Q(s', a')$. As we can see from the exploration function, $\frac{k}{n}$ is an exploration bonus where a higher occurrence of $n(s, a)$ will lead to a small bonus for further exploration of this state action pair, where a less frequent $n(s, a)$ will encourage the agent to explore more on the uncertainty.

By applying this concept in the Q update from equation 2.11, we have a modified Q update function as follow:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(R(s, a, s') + \gamma \max_a f(Q(s', a'), N(s', a'))) \quad (3.2)$$

[12]

This equation is very similar to the normal Q update function in 2.11. However, it has the feature to not only allow the continuous Q estimate contribution to the update, but also encourage exploration on the uncertainty with the exploration bonus. For the action selection process, the agent will use the bonus in the face of uncertainty and compute the optimal action with respect to the exploration function and take the action accordingly. One advantage is that, the exploration bonus will be propagated throughout the whole Q value table, which will not only encourage the agent to take an action that has a lot of uncertainty associated with it, but also encourage the agent to take actions that lead to parts of the state space where there is still a lot of uncertainty so that exploration is maximised with the minimum number of moves.

3.2.3 Speedy Q-Learning

The learning rate α and principle of single Q value estimate in the Q table update equation contribute to the problem of slow convergence in the standard form of the QL algorithm. A new algorithm to speed up the Q convergence, called Speedy Q-Learning (SQL), is proposed by G.A Mohammad (2011) in a journal of ML research. [19] [18]

SQL is an update version of the standard QL. The main difference are as follow:

Dynamically decay α

In the standard QL, the learning rate α is set with a fixed value. This results in an inflexible study routine to either favour spending more time in learning or in performing action. Unlike QL, the SQL algorithm has a learning rate α that decays exponentially with the time.

$$\alpha_t = \frac{1}{1 - t}$$

It has the same principle as the method of decay ϵ that, in the beginning of the learning process, more time should be spent on learning from the sample estimate with a large α . As time goes by, the knowledge has become more solid and time should be allocated for performing actions based on the current state action value in the current Q table with a small α .

Multiple sample estimates

As we can see from the standard QL update equation, only one sample average is used in the state action pair Q value updates

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(R(s, a, s') + \gamma \max_a Q^*(s, a))$$

where

$$sample = R(s, a, s') + \gamma \max_a Q^*(s, a)$$

The magnitude of α in the equation determines how much the sample estimate contributes to the final Q update. This may result in a large varying sample estimate that leads to a slow convergence of the Q update and hence a slow learning time.

In SQL, the two sample averages, the previous sample estimate and the next sample estimate, are taken

$$\begin{aligned} sample_{t-1} &= R(s', a, s) + \gamma \max_a Q^*(s', a) \\ sample_{t+1} &= R(s, a, s'') + \gamma \max_a Q^*(s, a) \end{aligned}$$

Instead of using these sample estimates for the update, the absolute difference between the current state action Q value and the two sample estimates are calculated.

$$D_{t-1} = |Q(s, a) - sample_{t-1}|$$

$$D_{t+1} = |Q(s, a) - sample_{t+1}|$$

This is a better approach to update the Q table as this value of the difference is small but also a good reflection of the convergence of the Q table that can speed up the convergence and hence a faster learning time. By incorporating a dynamic learning rate α , the final update equation in the SQL is as follow:

$$Q(s, a) \leftarrow Q(s, a) + \alpha_t(D_{t-1} + D_{t+1}) \quad (3.3)$$

[7]

3.3 The Algorithm

With the knowledge of the QL equation 2.8 from previous chapter and the exploration-exploitation trade-off function, in this project the ε -Greedy, the QL algorithm for this project can be generalised as follow:

```

initialize  $\gamma$  and R;
initialize Q matrix = 0;
for each episode do
    while not reach goal state do
        select a from s ( $\varepsilon$ -greedy)
        Take action a, observe rewards r and state  $s'$ 
         $Q[s, a] \leftarrow (1 - \alpha)Q[s, a] + \alpha(R(s, a) + \gamma \max_{a'} Q[s', a'])$ 
         $s \leftarrow s'$ 
    end
end

```

Algorithm 1: Q-Learning algorithm [29]

In the project, the QL algorithm will be implemented in Matlab, where the programming environment will allow easy matrix manipulations. The rewards function R and action-value function Q will be represented by a n x n matrix respectively, where n denotes the number of states in the MDP.

This algorithm can be explained as the following steps:

1. Set the value for the discount factor γ and reward matrix R
2. Create an action-value matrix Q with the same dimension as R and initializes the Q matrix to all zeroes
3. Observe the current state s
4. Choose an action from the state s according to the ε -greedy policy
5. Perform the action and observe the reward r and the new state s'
6. Update the Q value for the state based on the maximum rewards can be obtained in the next state using the QL equation 2.8
7. Set the next state as the current state

3.4 Chapter Summary

In this chapter, we explained the preliminaries of the QL algorithm design such as the ϵ - Greedy and Softmax methods in the action selection and we also discussed the problems that arise from these method that lead to the paradigm of exploration vs exploitation.

Suggestions, such as decay ϵ , exploration function and SQL are given to balance the parameters in the exploration and exploitation, so that the QL can be optimised.

Finally, a feasible QL algorithm is proposed and each step in the algorithm is explained in details.

Chapter 4

Experiment Setup

In this chapter, the QL algorithm proposed in the previous chapter is used to solve a maze problem by finding the optimum path from a fix point A to B. The design of the experiment, the maze and the variables that are in our interest of study is explained and finally a detail explanation of the Matlab coding structure and simulation design is given.

4.1 Maze Design

The problem that is going to be solved and studied using QL is a set of $n \times n$ square maze, where n denotes the size of the maze. Since the maze is bounded by the size n , we can say that the maze can be modelled as a finite MDP. The goal, also the solution, to this maze will be finding the optimal path from a starting point A to the destination point B as shown in figure 4.1

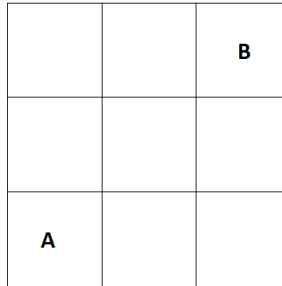


Figure 4.1: A 3 x 3 Maze problem

As the main objective of this thesis is to study the relationship among the 3 variables α , γ and ϵ in the QL algorithm corresponding to the time taken to complete the maze, 3 sets of maze with 3 different size, 2 x 2 ,3 x 3 and 5 x 5 respectively will be used for testing and demonstration purpose. The size of the maze is one of the factor to determine how quick the proposed MDP can be solved. Hence, due to time constraint of the thesis project, only the 2 x 2 maze will be used for testing of the relationship among

the variables with the time. The rest of the maze will be used in the demonstration in the project symposium at the end of this semester.

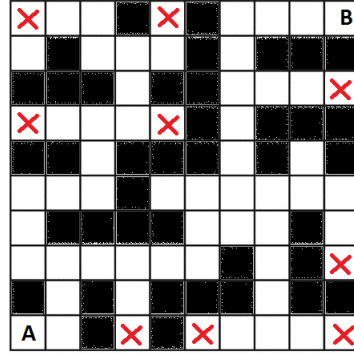


Figure 4.2: A 10 x 10 Difficult Maze problem

The complexity of the maze is often an interesting factor to study. A maze shown in figure 4.2 represents a possible scenario, where the black colour represents the obstacles and the red cross represents the dead-ends. A maze like such is hard to solve because, it is assumed that, the more obstacles that are present in the maze, the harder it will get to find the best route to the exit of the maze. It is like putting a human being blind folded into an unknown maze, it will first take the person to identified the obstacles by perhaps hitting the walls and the many possible dead-ends and routes to the exit in the maze a multiple times before the shortest route can be discovered. In the context of QL, bumping into the walls and dead-ends will be equivalent to receiving a negative rewards. Then the agent can learn from its experience (state action Q table) to work out the best route. In order to illustrate the essence of QL process, three 5 x 5 maze shown in figure 4.3 with different complexity are designed for demonstration in the project symposium.

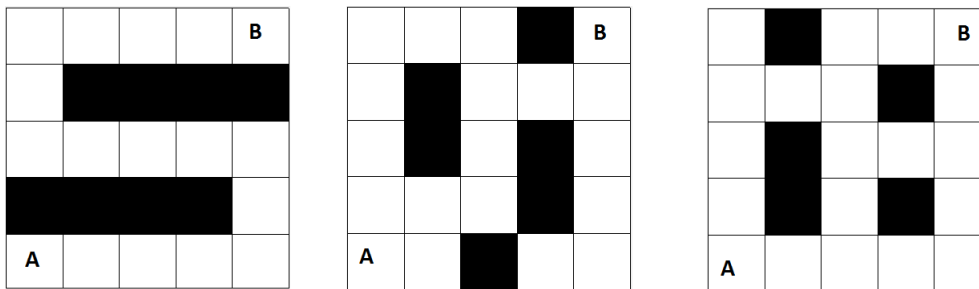


Figure 4.3: The 3 sets of 5 x 5 maze for demonstration

4.2 Experiment Design

Before the experiment is designed, it is necessary to revisit the objectives for this thesis we set in chapter 1 of this report. The two main objectives that are required experimentation and demonstrations are as follow:

- Study of the relationship between the 3 variables (α , γ and ϵ) in QL and the time taken to complete the maze.
- Graphical demonstration of the QL process in solving a complex maze using Matlab.

Learning Time vs Variables

In the QL section of the report, we learn the Q value update equation as follow:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(R(s, a, s') + \gamma \max_a Q^*(s, a))$$

As we can see from the equation, the Q updates are determined by the degree of learning rate α chosen for a particular run. The speed of the Q convergence is directly linked to importance of the contribution that the sample estimate, $R(s, a, s') + \gamma \max_a Q^*(s, a)$, has made in this equation.

The discount factor γ determines the values of the sample estimate with the value either favour taking the rewards at the present or in the future. The values of the sample estimate may also play a role in the Q convergence and this will be analysed through simulation.

Lastly, as the action selection method we have chosen for the QL algorithm is ϵ - Greedy, which suggest that the next action will be selected either random uniformly across all actions at a particular state or with the best state action value from the Q table based on the setting of ϵ in the algorithm. The randomness arises from this method will have a certain impact on the learning time as well.

The combination of these three variables are the key to determine the learning time of the agent using QL approach to solve the maze. The study of this relationship will help us to find the best combination to optimise the current QL algorithm. Hence, the design of the experiment procedures are as follow:

1. As the three variables are of value from 0 to 1, the first experiment will be examining the time taken for the completion of the 2 x 2 maze against the α value from 0.1 to 0.9 with the other two values set to the medium mark of 0.5.
2. Since the transition model in the maze is assumed to be stochastic, for each value of α , there will be 10 samples of the time taken and the average of the sample time will be calculated.

3. Plot a graph of the average time of completing the maze against the value of α from 0.1 to 0.9 for result discussion and analysis.
4. Repeat step 1 to 3, with varying the γ from 0.1 to 0.9 and fixing α and ϵ to be 0.5.
5. Repeat step 1 to 3, with varying the ϵ from 0.1 to 0.9 and fixing α and γ to be 0.5.

Graphical Demonstration

After studying the relationship between the learning time and the variables, we need to implement the QL algorithm into a bigger and more complicate problem. In the Matlab demonstration, the three 5 x 5 maze of different complexity mentioned in the previous section will be used. The aim of this experiment is to demonstrate the QL process and validate the QL algorithm by looking at the outcome of the best route generated based of the Q table. The design of the experiment procedures are as follow:

1. Run the Matlab QL simulation for the three 5 x 5 maze.
2. Compare the outcome of each maze run with a manual run (work out the best route by human visualisation).
3. Result discussion and analysis.

4.3 Mathematical QL Example

So far, we have identified the three variables α , γ and ϵ in QL that are required to the Matlab program design. However, there are still some elements missing such as the transition rules, rewards function and Q table. These are the basis for building a QL simulation. What roles these parameters are playing in the QL process and what types of data structure they can be represented in the Matlab programming environment? These are the questions need to be answered. In this section, a comprehensive mathematical illustration of using QL to solve a 2 x 2 plain maze is provided as a basic explanation of the structure that will be used in the Matlab coding process.

Plain Maze

Figure 4.4 shows the 2 x 2 plain maze that will be used in this example. As we can see from the figure, the starting state in 1 in blue and the finishing state is 4 in red. The transition rule for this maze can be represented in the graph 4.5 below. The black and red arrows represents the link between each state. In this case, a move can only be made to the adjacent state, not diagonally. For example, if the current state is 1, the actions are available in this state are moving to state 2 or state 3. State 4 is the goal state or absorption state. No actions can be made at this state as it marks the completion of the

2	4
1	3

Figure 4.4: A 2 x 2 Plain Maze

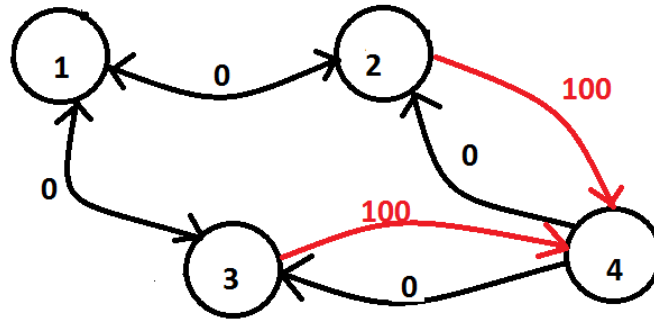


Figure 4.5: Graph representation of the 2 x 2 plain Maze

maze.

From this graph, a matrix corresponds to this transition rule, called *Linked* can be generated as follow:

$$Linked = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Again, from graph 4.5, we see that the values along the black arrows are 0, which denotes the rewards receiving when transition made among state 1, 2, and 3 based on the transition model defined in the *Linked* matrix. The values along the red arrows are 100, which denotes the positive rewards when transition has been made to the goal state. In this case, the transition from state 2 or 3 to state 4.

As a result, another matrix that represents the rewards function in the QL, called *R* can be written as follow:

$$R = \begin{pmatrix} - & 0 & 0 & - \\ 0 & - & - & 100 \\ 0 & - & - & 100 \\ - & - & - & - \end{pmatrix}$$

At this point, this maze has been modelled in terms of the rewards matrix and transition matrix. A Q matrix is required to represent Q table for the updates in QL process. It has the same dimension as the rewards matrix R and each row represents the current state, while each column represents the actions that the current state is pointing to go to the next state.

Now apply the QL algorithm described in the previous chapter to manually update the Q matrix through iterations over the maze. For this example, the discount factor γ will be set to 0.8 and the starting state is 1 and goal state is 4.

Firstly, the Q matrix is initialised to be zero

$$Q = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

and we have R matrix as

$$R = \begin{pmatrix} - & 0 & 0 & - \\ 0 & - & - & 100 \\ 0 & - & - & 100 \\ - & - & - & - \end{pmatrix}$$

Starting from state 1, the first row of R matrix, we see two possible actions for the current state 1. Either it can go to state 2 or state 3. Assuming a random action is selected, the transition is made to move from 1 to 2. Since all the Q values are zeros and the immediate rewards for this action is 0, the Q matrix remains the same.

Now we are at state 2, the possible action is to move to state 1 or state 4. We choose state 4 this time. Since the immediate rewards to go to state 4 is 100. Now the Q value for this state action pair can be calculated as

$$Q(state, action) = R(state, action) + \gamma \text{Max}[Q(nextstate, allactions)]$$

$$Q(2, 4) = R(2, 4) + 0.8 * \text{Max}[0] = 100 + 0.8 * 0 = 100$$

The current state becomes 4 and it is the goal state, which means that one episode of learning is finished and the final Q matrix is updated as

$$Q = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

For the next episode, we start from state 1 as usually. We then take action to move to state 3 and from state 3 we move to state 4. The Q value update corresponds to these state action pair can be calculated in a similar way as in the first episode. We now have the Q matrix as the following after 2 training episodes.

$$Q = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

The next episode, we take the same route as in the first episode. The difference is that after we move to state 2, the state action pair value Q is changed with the updated Q values as shown in the calculation below

$$Q(1, 2) = R(1, 2) + 0.8 * Max[0, 100] = 0 + 0.8 * 100 = 80$$

The updated Q table at the end of this episode can be written as

$$Q = \begin{pmatrix} 0 & 80 & 0 & 0 \\ 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

The learning continues and as the agent is trained with enough episodes, the Q matrix will converge as

$$Q = \begin{pmatrix} 0 & 80 & 80 & 0 \\ 64 & 0 & 0 & 100 \\ 64 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

With this final Q matrix, the best route can be found by taking the action that has the best Q values for each current state.

In order to better demonstrate the result, a matrix called bestQ is introduced. It has the same dimension as the maze and each value in the matrix represents whether or not landing at that state can lead to a maximum state action Q value. For instance, in this case the bestQ can be written as

$$bestQ = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

Where 1 represents the state contains the maximum Q values (Note the starting and goal state are also mark as 1). Based on this matrix, the best route from state 1 to state 4 is 1 - 2 - 4 or 1 - 3 - 4.

Complex Maze

For the design of the complex maze, obstacles are introduced. These obstacles are state that are inaccessible or negative rewards state. This means that, when transition from a normal state to the obstacle state, the immediate rewards will be a large negative number depends on the characteristic of the obstacle.

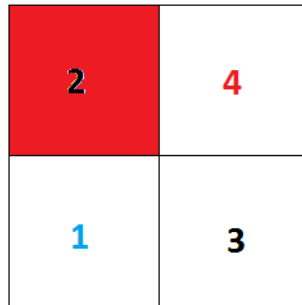


Figure 4.6: A 2 x 2 Hard Maze

Figure 4.6 is a 2 x 2 maze with one obstacle in represented by the red block in the maze and figure 4.7 is a graphic representation of the 2 x 2 maze.

In figure 4.7, the green arrow shows the transition rule into and out from the obstacle. We can see the value along the arrow now becomes -100. This value will propagate through the Q matrix update and help the agent to identify the state as obstacle, so it should try to avoid it.

Using the same mathematical example in the plain maze, the rewards matrix R is now changed to

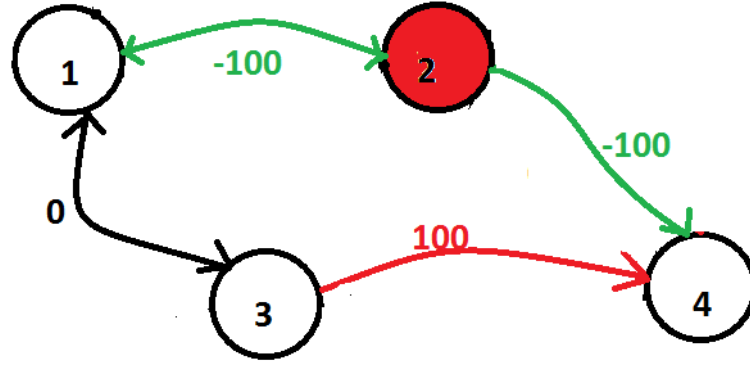


Figure 4.7: Graph representation of the 2 x 2 Hard Maze

$$R = \begin{pmatrix} - & -100 & 0 & - \\ 0 & - & - & -100 \\ 0 & - & - & 100 \\ - & - & - & - \end{pmatrix}$$

The link matrix stays the same, which suggest that the agent can still move into and out from the obstacles, but with a substantial amount of negative rewards. As the number of trainings increases, the agent will realise state 2 is an obstacle based on its poor Q value. Using the same mathematical iteration used in the plain Maze, the Q matrix will eventually converge to something like this

$$Q = \begin{pmatrix} 0 & -100 & 80 & 0 \\ -36 & 0 & 0 & -100 \\ 64 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

As a result, the bestQ matrix produce based on this Q matrix will be

$$bestQ = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

Hence, optimal path to this maze is 1 - 3 - 4.

4.4 Matlab Coding

In this section, the Matlab coding design and code explanation is given based on the understanding of the variables required and QL process from the numerical examples from the previous section. There are two Matlab scripts written for this thesis project:

Simulation This file consists of the Matlab implementation of the QL algorithm and the graphical representation of the QL process in multiple plots.

GUI This file consists of the Matlab GUI designed based on the specifications given in the objectives from chapter 1.

The QL Matlab implementation designed for this project is based on a script written by an applied artificial intelligence research group from frog.ai blog [8]. However, changes are made in the script to best suit the objectives of the simulation and demonstration for this thesis project. Changes as such will be highlight and explained in detail in the next section. The GUI Matlab script is solely designed and written by the author of this thesis, Lu Han. The two complete scripts are provided in the Appendix for study purpose. The scripts are provided as open resources, any changes and redistribution of the code are welcomed with acknowledgement.

4.4.1 Simulation Run

Variables

Based on the mathematical illustration of QL, we have the follow Matlab script for variables initialisation:

```

1 numStateRows = 2;
2 numStateCols = 2;
3 numStates = numStateRows*numStateCols;
4 goalState = numStates;
5
6 alpha = 0.5;
7 gamma = 0.5;
8 epsilon = 0.5;
9
10 mLinked = zeros(numStates);
11
12 rewardM = zeros(numStates);
13 rewardM(:,goalState) = 100;
14
15 QCurrent = zeros(numStates);

```

The numStateRows and numStateCols define the dimension of the maze, the numStates is the numbers of state in the maze and the goalState is the numerical representation of the absorption state in maze and it is initialised to be the last state number in

the matrix.

Alpha, gamma and epsilon are the three main variables in the Q value update equation. The Linked matrix is the matrix defined in the previous section for the transition rules and it has a dimension of numState x numState where each row in the matrix represents a particular current state and each column represents the state that the current state either have a link or no link to this state. The rewards matrix takes the same dimension as the mLinked. It is initialised to a zero matrix and then the immediate rewards that receive at a state that leads to a transition to the goal state (in this case, the last numerical state) is set to be 100 as shown in line 13 from the script.

Lastly, the QCurrent is a matrix that represents the Q table that needs to be updated in the QL process. It has the same dimension of the rewards matrix rewardM and is initialised to a zero matrix in the beginning of the learning process.

Complex Maze

The complex maze is designed by putting obstacles into the maze graphically. This is equivalent to assigning negative rewards to a particular state in the state rewards matrix, so that when the agent make a move into and out from such state, a negative reward is collected. Once the state action value in the obstacle state is updated as negative value after some trials and errors, the agent can then learn that this state is an obstacle and then try to avoid it by making better choices in the action selection process in the next episode. The following Matlab code illustrate such idea using the 2 x 2 maze scenario in figure 4.6, by setting the rewards to all state 2 and out from state 2 to be -100.

```
1 rewardM(:,2) = -100;
2 rewardM(2,:) = -100;
```

State to Rows and Columns conversion

The states in the graphical demonstration can be represented by a matrix corresponds to the dimension of the maze.

Matrix iteration in Matlab requires coordinates of the states correspond to the rows and columns in the matrix. For instance, as shown in figure 4.8, state 1 is at row 1 and column 1, whereas state 3 has a coordinates of row 1 and column 2. In order to manipulate such state matrix, a conversion needs to be made to instantly report the coordinates of a particular state in a state matrix. The following Matlab code is directly adopt from the frog.ai group [8].

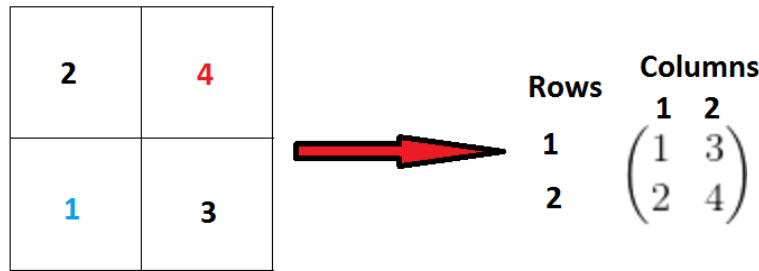


Figure 4.8: Matrix representation of a 2x2 Maze

```

1 function [row,col] = stateToRowCols(stateNum,numStateRows,numStateCols)
2 col = ceil(stateNum/numStateCols);
3 row = (stateNum+numStateRows)-(numStateRows*col);

```

This function takes in the state number and dimension of the maze and returns the row and column coordinates to that state by using the two equation devised in line 2 and 3.

Transition Rules

Once the coordinates of the state is calculated through conversion, these values can be used to update the mLinked matrix to specify the rules of transition. The following script shows how the mLinked matrix is updated.

```

1 for i = 1:numStates
2
3 [rowSi,colSi] = stateToRowCols(i,numStateRows,numStateCols);
4
5 for j = 1:numStates
6
7 [rowSj,colSj] = stateToRowCols(j,numStateRows,numStateCols);
8
9 if(colSj == colSi && (rowSj - rowSi == 1 || rowSj - rowSi == -1))
10     mLinked(i,j) = 1;
11 end
12 if(rowSj == rowSi && (colSj - colSi == 1 || colSj - colSi == -1))
13     mLinked(i,j) = 1;
14 end
15 end
16 end

```

The original code from frog.ai [8] allows transition to be made to both adjacent and diagonal states as shown in figure 4.9. This has been modified to allow only adjacent transition for the maze problem in this project.

The main idea behind the making of the transition rule is to iteratively check a particular state with all the possible state in the maze with the condition that, if the state

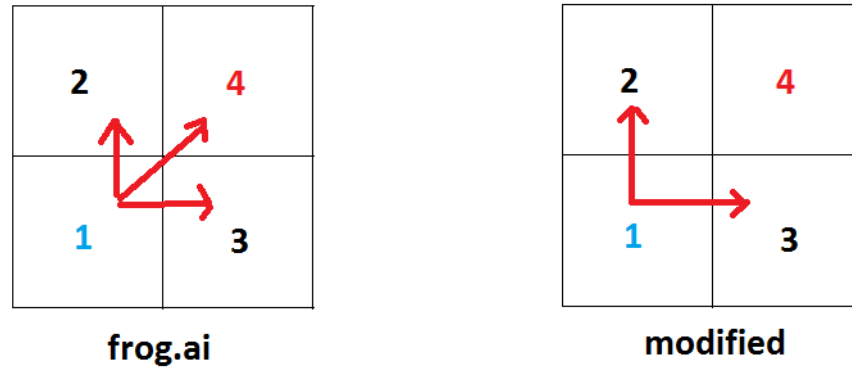


Figure 4.9: Updated Transition Rules

is in the same row as the observed state, the state is linked to the observed state only if the difference of the column coordinate of each state is either 1 or -1. Same condition for state that is in the same column as the observed state, the link is confirmed only if the difference of the row coordinate of each state is either 1 or -1.

This is illustrated in the script above that the iteration is taking place inside a nest loop and once a link is found based on the condition in line 9 and 12, the corresponding values to the coordinates in the mLinked matrix is set to 1.

ϵ - Greedy

The action selection method chosen for the QL algorithm is ϵ - Greedy which uses the terminology of choose a random action uniformly with big ϵ or best action based on the best Q value with a small ϵ .

For uniform random action selection,

```

1
2 optionVector = mLinked(CurrentState,:);
3 nextPt = 0;
4
5 while nextPt == 0
6   nextStateRandom = randi([1, numStates]);
7   nextPt = optionVector(nextStateRandom);
8 end;
```

The idea for this design is that, we take the rows of the current state off from the mLinked matrix and find the first random state from this row that appears to have a link to this current state as the next state we are going to move to. This is shown in the above code, a variable nextPt is set to 0 initially and by generating a random number from 1

to the last state number, this random number is assessing whether or not has this state number is linked to the current state in line 6 and 8. The loop will end when the first random number is found and the current state is set to the next state.

For action selection based on the best Q values, we have the following code:

```

1 nextStateVector = QCurrent(CurrentState,:);
2 possibleQ = mLinked(CurrentState,:).*nextStateVector;
3 nextPt = 0;
4 nextStateBestQ = 0;
5 nextStateBest = 0;
6
7 for i = 1:numStates
8     if possibleQ(i)>nextStateBest
9         nextStateBestQ = possibleQ(i);
10        nextStateBest = i;
11    end
12 end;
```

The key idea behind writing this code is very similar to the random action selection, but with the focus on the Q matrix QCurrent. In line 1, the rows of the current state from the QCurrent matrix is extracted. The state action value in this vector is transverse and the best Q value is updated each iteration until the end of the loop as shown in line 7 to line 12. The current state is set to the state that has the largest Q value from the iteration.

The action selection is completed with examining the ϵ value from set by the user. The condition for whether to go for random or best Q is shown below:

```

1 if (rand < (1-epsilon))
2     nextState = nextStateBest;
3 else
4     nextState = nextStateRandom;
5 end;
```

Where a random number between 0 to 1 is generated and if this number is smaller than $1 - \epsilon$, best Q policy is favoured, or in the other case, random selection will be chosen.

Q table updates

In chapter 2, we have studied the Q table update function as

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(R(s, a, s') + \gamma \max_a Q^*(s, a))$$

It consists of two parts. The first part is the current Q table weighted by $1 - \alpha$ and the second part is the sample Q estimates weighted by α . These two parts are written separately in the Matlab script below:

```

1 part1 = ((1-alpha)*QCurrent(currentState,nextState));
2 part2 = alpha*((rewardM(currentState,nextState)+
3   (gamma*(max( mLinked(nextState,:).*QCurrent(nextState,:))))));
4 QCurrent(initialState,nextState) = part1 + part2;

```

Then the Q table is updated by adding these two parts together in line 4.

Q table convergence

As the QL algorithm suggested in chapter 2, the process is stopped after 1 episode of study. However, the importance of the QL is through multiple episodes of study, before the convergence of the Q table is taking place. An additional script is written below to automatically stop the learning process once the Q table is converged.

```

1 if sum(sum(abs(Qprevious-QCurrent)))<0.0001)
2     if count>10
3         keepGoing = 0;
4     else
5         count=count+1;
6     end
7 else
8     Qprevious = QCurrent;
9     count=0;
10 end

```

The idea of constructing this code is that since each single values in the Q matrix is changing over each move, it is hard to use these values to determine the convergence of the Q matrix as a whole. The better parameter will be the sum of the while Q matrix. We can confirm the convergence of the Q by examining the difference of the sum of the previous Q matrix to the sum of the current Q matrix.

The things we are looking at here is a reasonable number of small difference that occurring consecutively. In the code, the difference we are interested is 0.001 in a 10 consecutive run. If the condition of small difference and number of consecutive occurrence is satisfied, the keepGoing parameter is set to 0 to finish the learning. Keepgoing is a parameter initially set to 1 to start the learning in the loop.

Best Q value representation

We obtain a converged Q matrix at the end of QL process. But how to use this matrix to best illustrate the optimum path in the maze? A matrix with the same dimension of the maze is needed to indicate whether a particular state is on the best route or not. The following code illustrates how this matrix is made.

```

1 bestQ = zeros(numStateRows,numStateCols);
2 current = 1;
3 bestQ(current) = 1;
4 bestQ(numStates) = 1;
5 maxQ = 0;
6 while (current ≠ goalState)
7     bestVector = QCurrent(current,:);
8     for i = 1:numStates
9         if bestVector(i)>maxQ
10             maxQ = bestVector(i);
11             current = i;
12             bestQ(current) = 1;
13         end
14     end;
15 end

```

The idea is that, starting from the initial state, the next possible states are assessed with the state action pair values associated with them. The maxQ value is an indicator that gets constant updates with the latest largest Q values in the search and if the any of the next possible states that has a value that are equal or large than the maxQ, this or these states will be put a 1 in the bestQ matrix, where states have a value 1 will be on the optimum path.

Graphic representation

The graphic illustration consists of four elements in a single figure.

- Q learning demonstration in a maze
- Optimal Path
- A plot of the sum of Q matrix against the moves taken in the QL process
- Illustration of state action Q table updates

The plotting methods are directly adopted from frog.ai [8] as follow:

```

1 %Q learning demo
2 mWatch(rowSns,colSns) = 1;
3 padded=padarray(mWatch,[1,1],'post');
4 fig1 = subplot(4,4,[1,2,5,6]);
5 pcolor(padded)
6 my_map = [1,1,1;
7           1,0,0;
8           0,0,0];
9 colormap(fig1,my_map);
10 title('Learning agent demo - move from bottom left to top right')
11 axis off;

```

```

12 axis square;
13 pause(0.001)
14 mWatch(rowSns,colSns) = 0;
15
16 %Q table updates
17 padded2=padarray(Qprevious,[1,1],'post');
18 fig2 = subplot(4,4,[11,12,15,16]);
19 pcolor(padded2)
20 colormap(fig2);
21 title('Current Table of State/Action Q Values')
22 axis square;
23 xlabel('Next State (if action to move there is taken)')
24 ylabel('Current State')
25
26 % Sum of Q vs Number of moves
27 Qconverge(episodes) = sum(sum(Qprevious));
28 subplot(4,4,[9,10,13,14]);
29 plot(totMoves,Qconverge)
30 title('total value of Q matrix')
31 xlabel('number of moves')
32 ylabel('sum of Q matrix')
33
34 %Optimal Path
35 padded=padarray(bestQ,[1,1],'post');
36 fig3 = subplot(4,4,[3,4,7,8]);
37 pcolor(padded)
38 my_map = [1,1,1;
39           1,0,0;
40           0,1,1];
41 colormap(fig3,my_map);
42 axis off;
43 axis square;
44 title('Optimun path based on the best state/action Q values')

```

The plotting strategy utilises subplot in Matlab to support multiple plots in one figure. It also uses the padarray method [16] in Matlab for real time display of the matrix iteration as grid world iteration. The biggest change I have made in the original scripts are the uses of the colormap in Matlab. The modified colormap [15] for my demonstration are shown in the figure 10, which the smallest value is the denoted by the left most colour on the colour scale. For my demonstration, the states are in white, the obstacles are in black, the learning agent is in red and the optimal path is in blue.

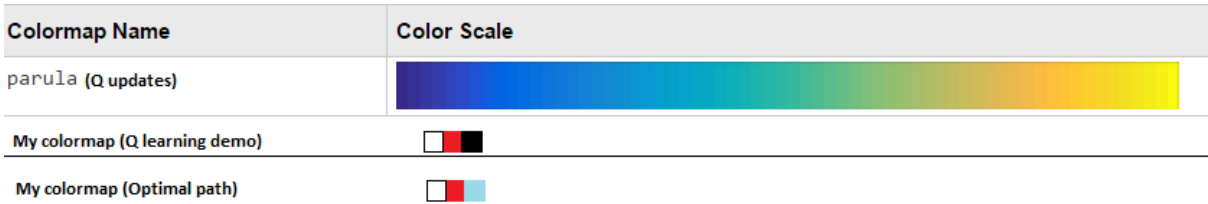


Figure 4.10: Color Map

Figure 4.11 illustrates how the figure looks like in a test run.

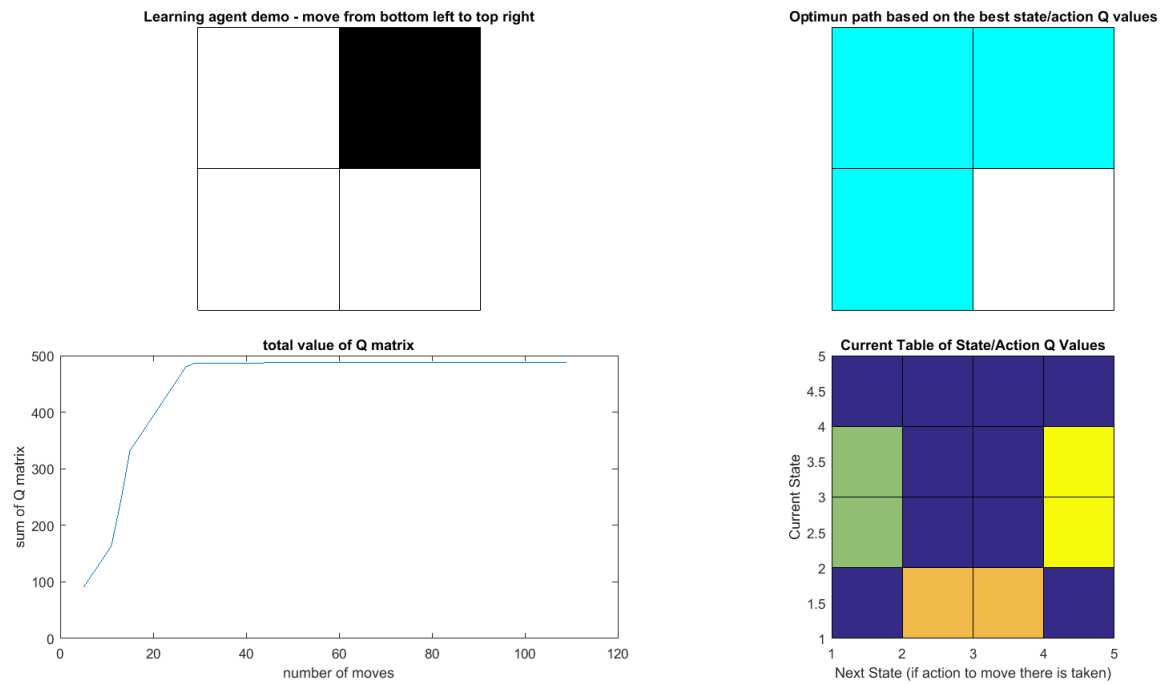


Figure 4.11: Sample simulation run

4.4.2 GUI

As part of the thesis project objectives, a GUI is designed to allow user to view and run the Q learning with great flexibility. The design requirement for the GUI are as follow:

- Allow user to set the values of each variables that includes, the size of the maze, α , γ , ϵ , the display speed for the demo and the number of samples for result analysis.
- Allow user to load a different sets of pre-set maze and run the simulation.
- Display the results from each sample run and of the average of these sample.

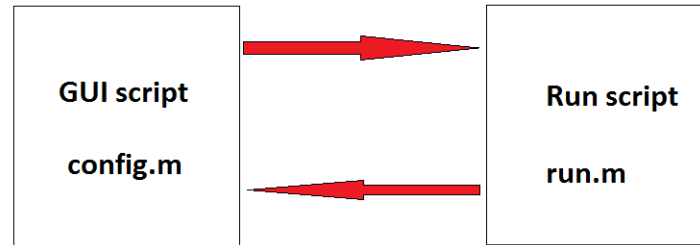


Figure 4.12: GUI design

As shown in figure 4.12, the key rationale behind the GUI design is that the GUI script takes the user input as string in the GUI and these string are converted to values before they are pass to the Run script to process. The Run script is responsible for starting the simulation and results are converted back to strings once it is finished. Then these strings are displayed back on the GUI.

For this project, the GUI is designed with the GUI toolbox from Matlab. The toolbox allows user to add and configure buttons or text box in figure display. The corresponding fig file contains a Matlab file that uses the principle of Object-Oriented-Programming (OOP) principle on the Matlab platform. Figure 4.13 shows the final design of the GUI for this project and the Matlab script associate with this GUI is provide in the Appedix as an open resource.

The screenshot shows a MATLAB GUI window titled 'config'. It is divided into several sections:

- Variables:** Contains input fields for 'Alpha', 'Gamma', 'Epsilon', and 'Display speed'.
- Maze size:** Contains input fields for 'Number of rows' and 'Number of columns'.
- Sample size:** Contains an input field for the number of samples.
- Results:** A large section containing:
 - 10 sample result tables (Sample 1 to Sample 10), each with columns for 'Episodes', 'Steps', 'Total Moves', and 'Time'.
 - A 'Sum of sample attributes' section with four input fields for the sum of episodes, steps, total moves, and sample time.
 - An 'Average of sample attributes' section with four input fields for the average of episodes, steps, total moves, and sample time.
- Maze difficulty:** A dropdown menu currently showing 'plain maze', with other options being '5 x 5 maze set 1' and '5 x 5 maze set 2'.
- Run:** A large button at the bottom right to execute the simulation.

Figure 4.13: Final GUI demo

4.5 Chapter Summary

In this chapter, the design concept of the maze problems, such as the states transition, size and the complexity of the maze, are explained. The graphically illustration of the 5 x 5 maze that are going to be used in the experiments and demonstration are also given.

Then we go on to talk about the two proposed experiments for this project, time variable relationship and validation of the QL algorithm in the Matlab simulation. The rationales behind and the experiment procedures are also explained in detail.

Before we start coding in Matlab, a complete numerical example of the Q table update in the QL algorithm on a 2 X 2 plain maze is provided to illustrate the basic idea behind the design of the structure of the Matlab code, in terms of the data type (matrix) and variables that are going to be present in the code.

Lastly, the important parts of the Matlab code correspond to the QL algorithm and the design concept behind the Matlab GUI design are explained in detail.

Chapter 5

Results and Analysis

In this chapter, we are going to look at the results from the two experiments proposed in the previous chapter for time variables relationship study and validation of the QL process. A thorough discussion is given at the end of each experiment on the simulation outcome and our expectation.

5.1 Experiment 1

The first experiment involves the study of how the three variables α , γ , ϵ in the QL updates equation affect the learning time of the agent to find the optimal path to the maze. According to the experiment procedures described in the previous chapter, a 2 x 2 plain maze will be used for the simulation.

5.1.1 α

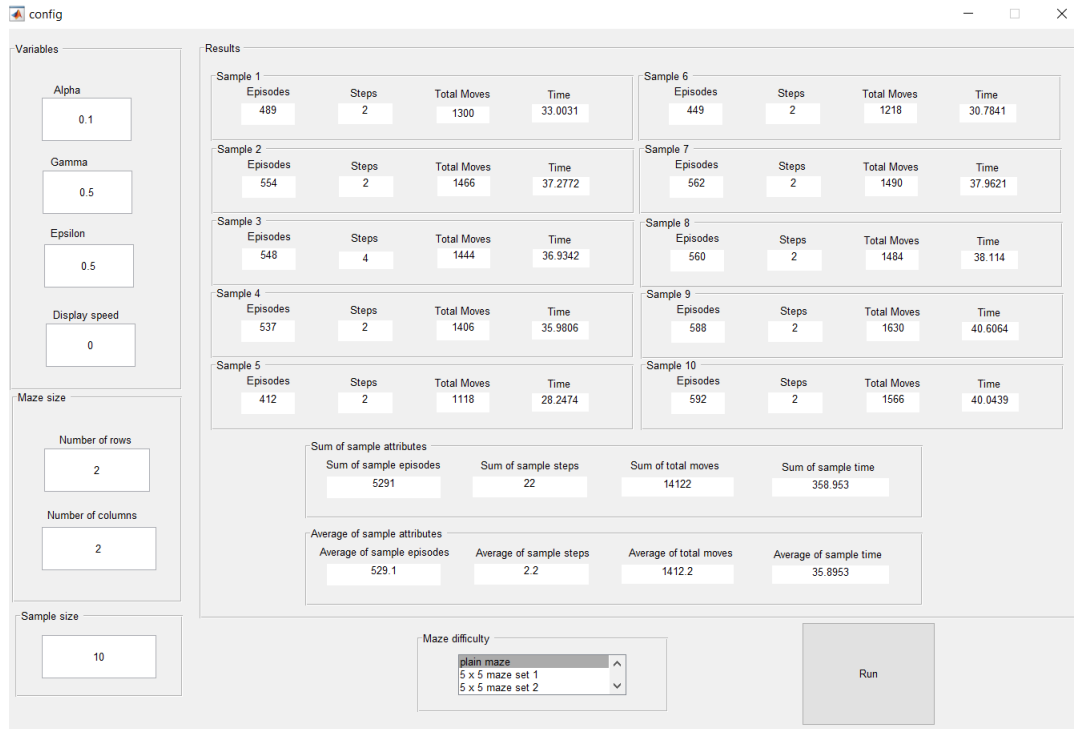
The first variable we are going to analyse is α . The setting for the simulation are as followed:

- α varies between 0.1 to 0.9
- γ and ϵ are set to 0.5
- Sample size taken for averaging is 10

Simulation results

Figure 5.1 shows the setting in the GUI for simulation. All the results from each sample taken includes the average of these results are displayed.

As we can see from the 10 samples taken, the average time taken for finding the optimal path in this maze is 38.8953s. This process repeats for $\alpha = 0.2, 0.3, 0.9$, the average

Figure 5.1: $\alpha = 0.1$

time taken for each α is record in the table shown below:

α	time /s
0.1	35.8953
0.2	18.6397
0.3	13.3165
0.4	10.504
0.5	8.08877
0.6	7.7427
0.7	5.82082
0.8	5.35505
0.9	4.266

Table 5.1: Time taken with varying α

A graph of the time taken against the different sets of α is plot as shown in figure 5.2.

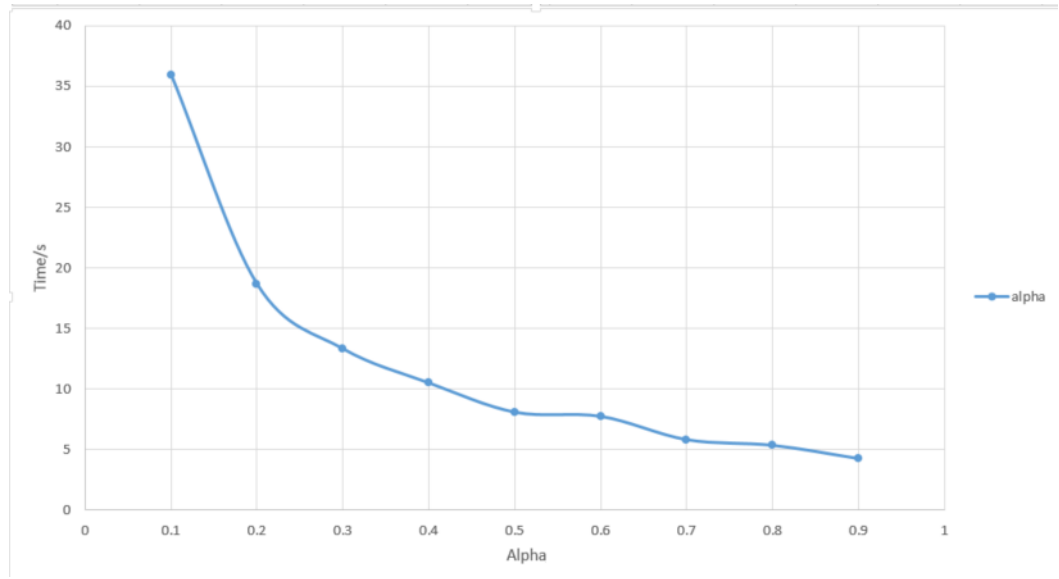


Figure 5.2: Time vs α

Discussion

As we can see from figure 5.2, the average time taken to converge the Q table is very high when α is small. The time decreases exponentially with a linear increase in the α value. We look at the Q update equation again

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(R(s, a, s') + \gamma \max_a Q^*(s, a))$$

When α is small, for instance 0.1, only 10% of the sample Q estimate is contributed to the Q updates, this results in a slow Q updates that leads to a slow Q table convergence, hence a slower time to finish learning. This is supported by the graph of sum of the Q matrix against the number of moves taken in the QL learning in figure 5.3.

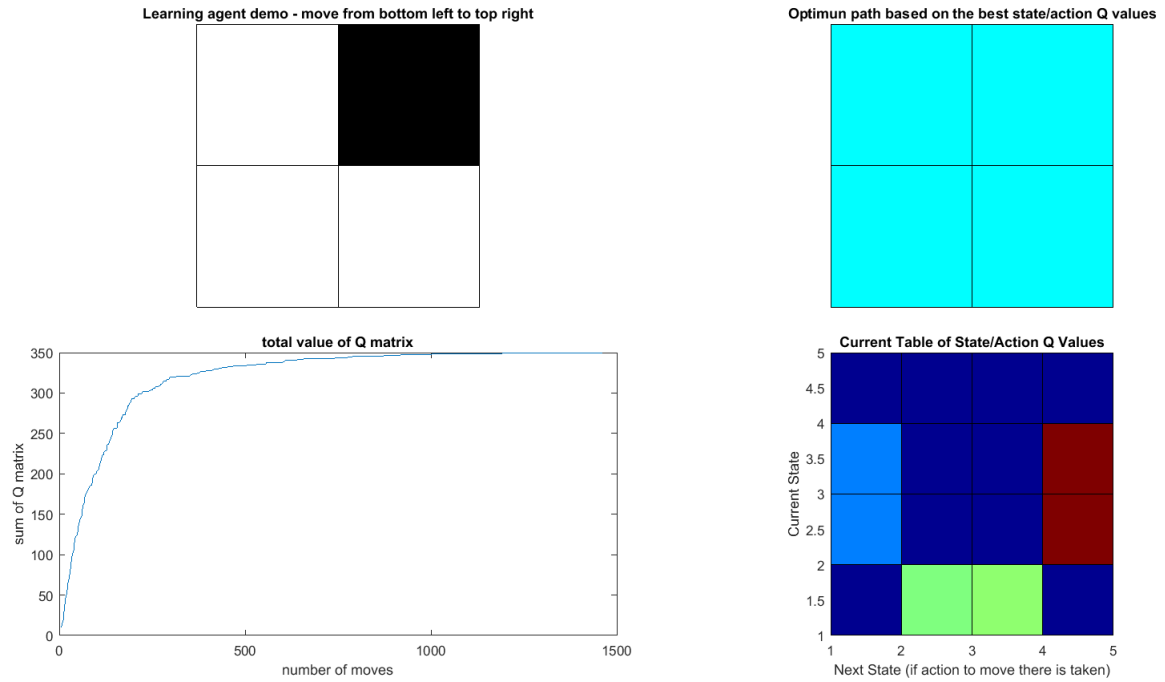


Figure 5.3: Sample run when $\alpha = 0.1$

As we can see from the figure that the Q total value of the Q matrix converges after approximately 1500 moves are made. This explains the high time value back in figure 5.2.

On the other hand, when α is large, say 0.1, 90% of the sample Q estimate is brought to the Q updates, this results in a rapid Q updates that leads to a quick Q table convergence, hence a faster time to finish learning. This can also be supported by the graph of sum of the Q matrix against the number of moves taken in the QL learning in figure 5.4.

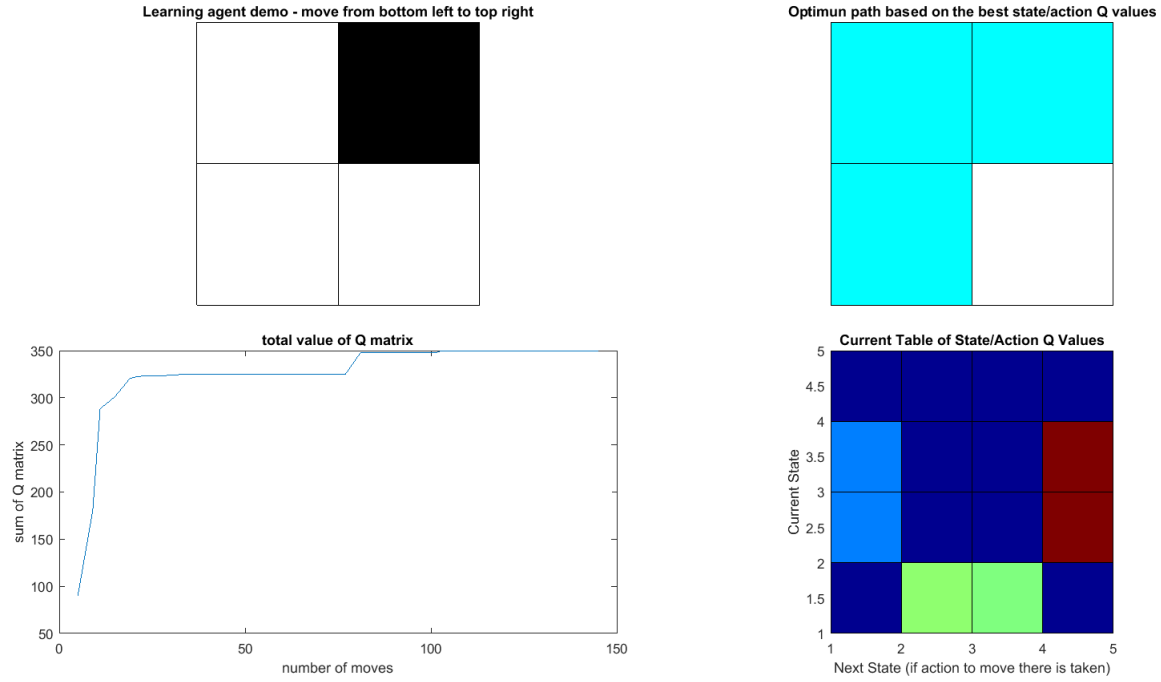


Figure 5.4: Sample run when $\alpha = 0.9$

In figure 5.4, the convergence of Q matrix happens quickly in just about 150 moves, this supports the small time value we see in figure 5.2.

One interesting to note is that, there is only one best route found when $\alpha = 0.9$, whereas when $\alpha = 0.1$, two best routes are estimated. These two extreme cases indeed illustrate the characteristics of the learning rate α explained in chapter two, where it says the more the agent learn from experience, the more accurate the learning outcome will be (in this case, finding the optimal path). However, the drawback is a slower learning time. Hence, the balance between learning more and applying the knowledge more needs to be found to further optimise the learning process.

5.1.2 γ

We go on to examine γ . The settings for the simulation are as followed:

- γ varies between 0.1 to 0.9
- α and ϵ are set to 0.5
- Sample size taken for averaging is 10

Simulation results

Figure 5.5 shows the setting in the GUI for simulation. All the results from each sample taken includes the average of these results are displayed.

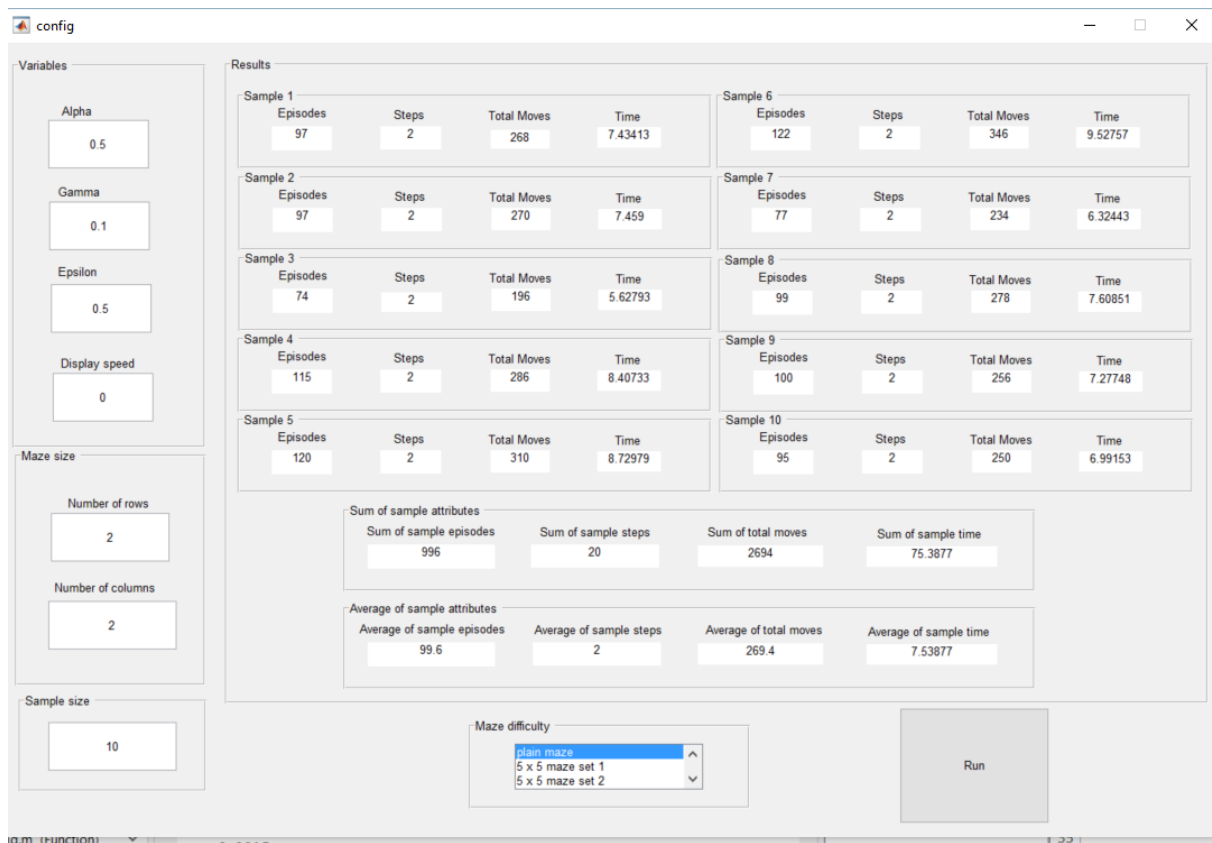


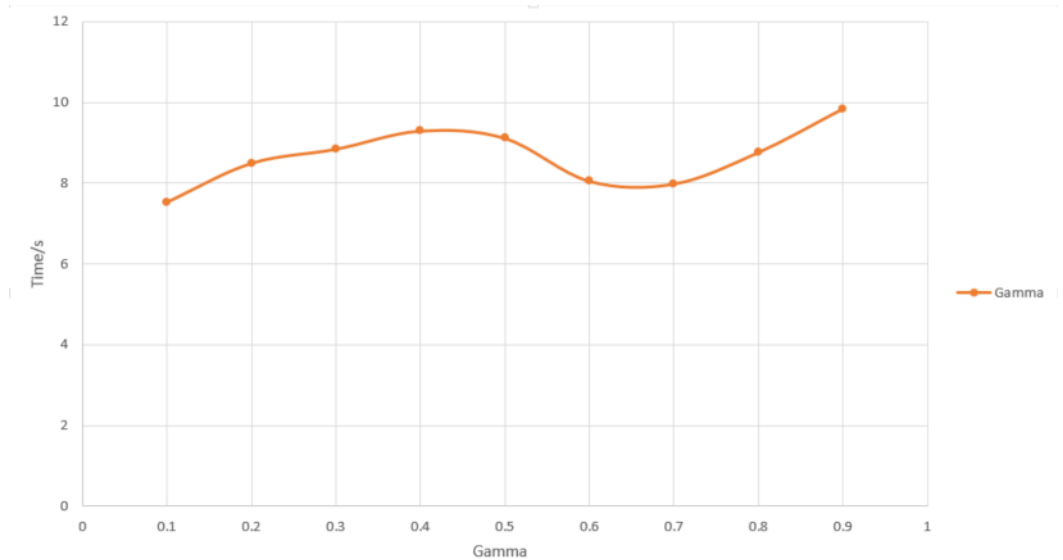
Figure 5.5: $\gamma = 0.1$

As we can see from the 10 samples taken, the average time taken for finding the optimal path in this maze is 7.53877s. This process repeats for $\gamma = 0.2, 0.3, 0.9$, the average time taken for each γ is record in the table shown below:

γ	time /s
0.1	7.53877
0.2	8.50105
0.3	8.8504
0.4	9.301
0.5	9.12023
0.6	8.05814
0.7	7.98894
0.8	8.76828
0.9	9.84848

Table 5.2: Time taken with varying γ

A graph of the time taken against the different sets of γ is plot as shown in figure 5.6.

**Figure 5.6:** Time vs γ

Discussion

In figure 5.6, we can see a fluctuating trend in the time taken to complete the learning with varying γ . However, the time difference between the time taken when γ is small and when γ is large, is not very significant. We examine this with the Q update equation

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(R(s, a, s') + \gamma \max_a Q^*(s, a))$$

γ is the factor that determines how much value of the rewards is discounted in the sample estimate. Although the sample Q estimate can be affected by different values of γ , the effect on the final Q table updates by γ is less significant than it does by α . This

explains the insignificant changes in the time taken we see in figure 5.6 when we change the value of the γ .

This is further supported by the two graphs of the total value of Q matrix against the number of moves taken when γ is set to 0.1 and 0.9 respectively.

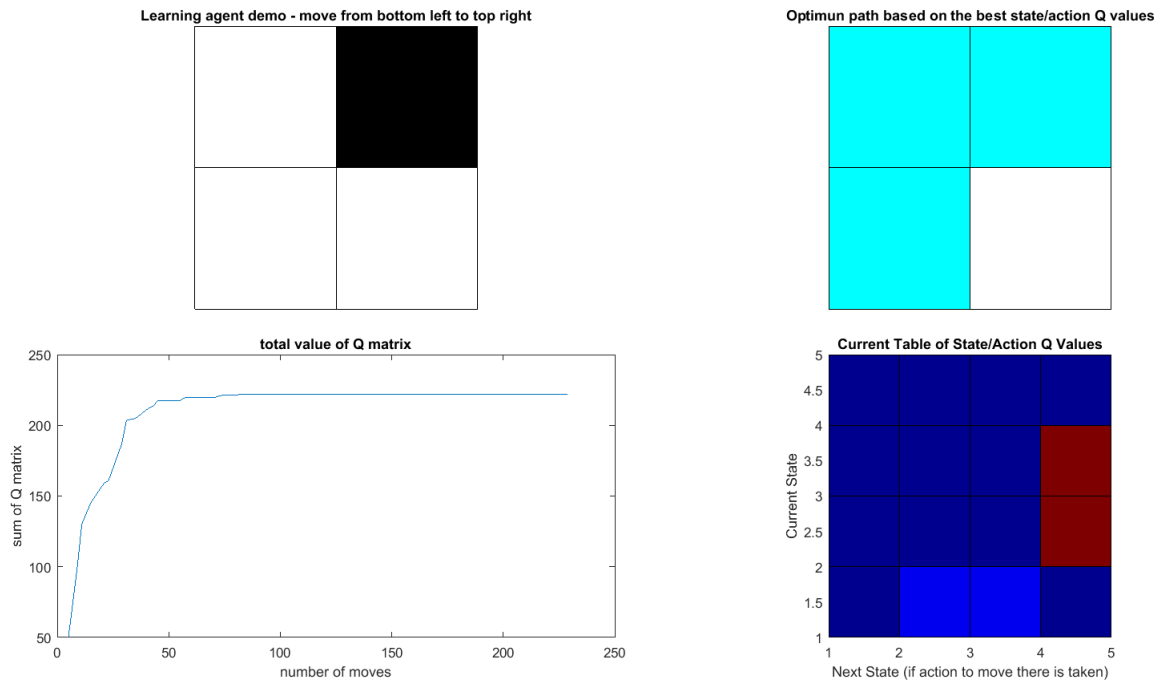


Figure 5.7: Sample run when $\gamma = 0.1$

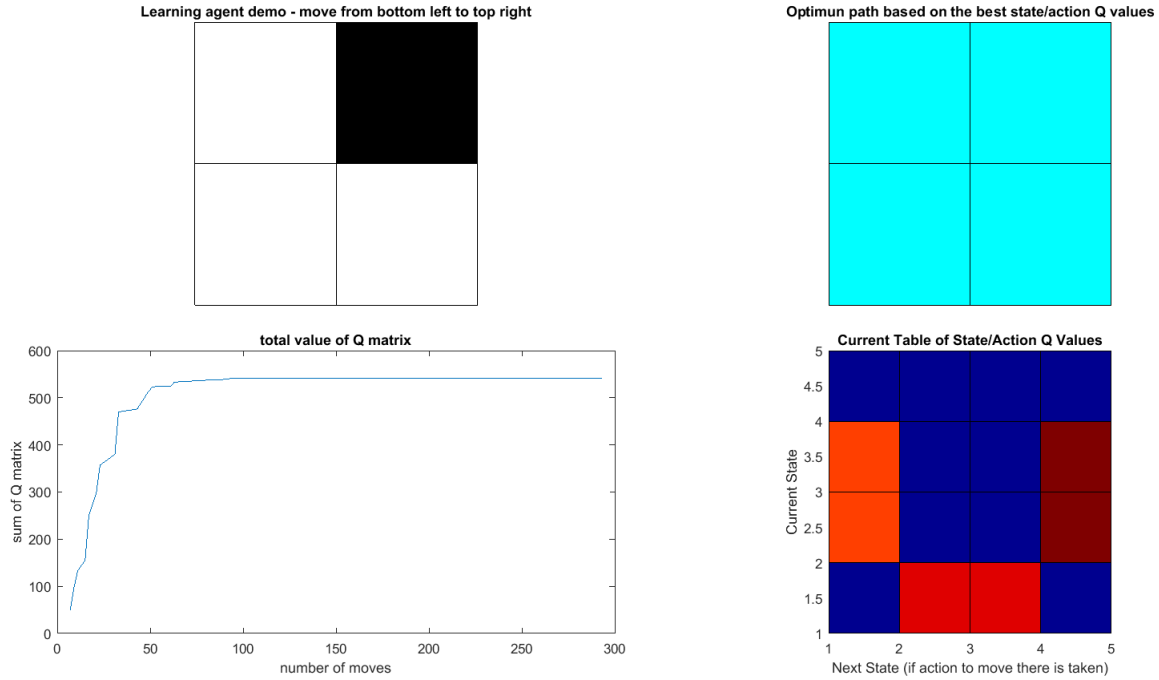


Figure 5.8: Sample run when $\gamma = 0.9$

As we can see from figure 5.7 and 5.8, the convergence of the Q table happens at approximately 220 to 250 moves with both γ value. This implies that the change in γ does not really affect the time for Q table convergence.

Similarly, we see a very interesting outcome of the optimal path detected based on the best Q values in the converged Q table with different γ . When $\gamma = 0.1$, only one path is found, whereas when $\gamma = 0.9$, two paths are discovered. A concise conclusion cannot be drawn here due to the randomness in every action taken in the experiment, the limitation on size of the maze and the inadequate samples taken. However, we can still see that by favouring current rewards (when γ is small), the chances of finding the other optimal paths are limited. Hence, a median selection of γ value of 0.5 to 0.6 as shown in figure 5.6 is a fair value for the optimisation of the QL process.

5.1.3 ϵ

Finally, We examine ϵ . The settings for the simulation are as followed:

- ϵ varies between 0.1 to 0.9
- α and γ are set to 0.5
- Sample size taken for averaging is 10

Simulation results

Figure 5.10 shows the setting in the GUI for simulation. All the results from each sample taken includes the average of these results are displayed.

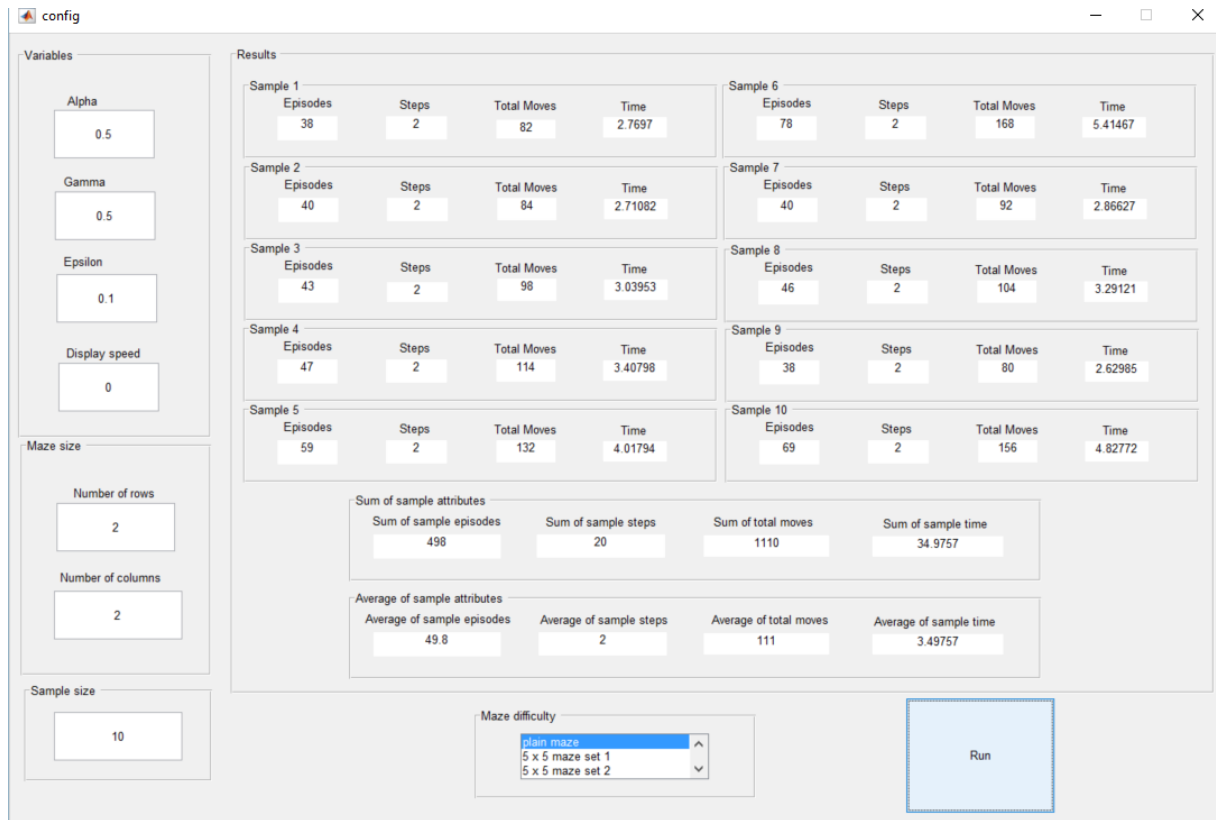


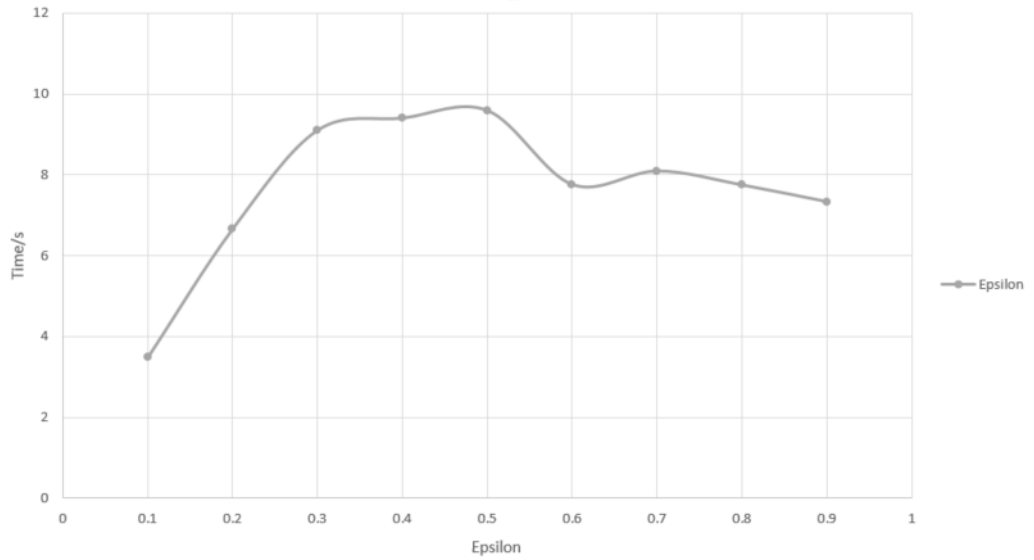
Figure 5.9: $\epsilon = 0.1$

As we can see from the 10 samples taken, the average time taken for finding the optimal path in this maze is 3.49757s. This process repeats for $\epsilon = 0.2, 0.3, 0.9$, the average time taken for each ϵ is record in the table shown below:

ϵ	time /s
0.1	3.49757
0.2	6.66684
0.3	9.09657
0.4	9.39391
0.5	9.57835
0.6	7.75411
0.7	8.08939
0.8	7.74475
0.9	7.33241

Table 5.3: Time taken with varying ϵ

A graph of the time taken against the different sets of ϵ is plot as shown in figure 5.9.

**Figure 5.10:** Time vs ϵ

Discussion

From the line graph in figure 5.10, we can see that time taken to finish the learning increases when ϵ value rise from 0.1 to 0.5. Then the time taken drops and fluctuates at the 0.6 to 0.7 marks before it finally settles at approximately 8s from ϵ value of 0.8. Lets take a look of the ϵ - Greedy method again:

- When ϵ is large, the action is mostly selected random uniformly.
- When ϵ is small, the action is mostly selected based on the best state action Q values.

This explains the trend in figure 5.10. When ϵ is small, each next action to take is chosen according to the largest Q value in the Q table updates. Hence, it takes less time before an optimal path is discovered. On the other hand, when ϵ is large, actions are taken in a random manner that most of the time is spending on exploring the states, rather than looking at the Q values. This leads to an increase in the learning time.

This explanation can also be supported by the two graphs of the total value of Q matrix against the number of moves taken when ϵ is set to 0.1 and 0.9 respectively.

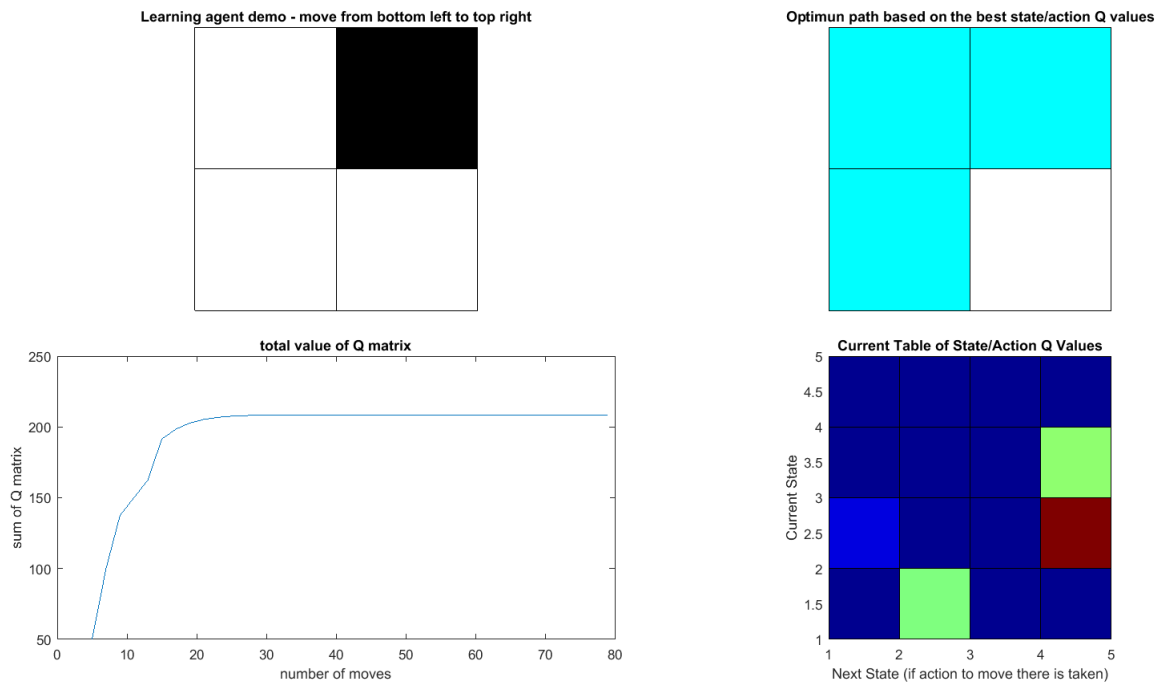


Figure 5.11: Sample run when $\epsilon = 0.1$

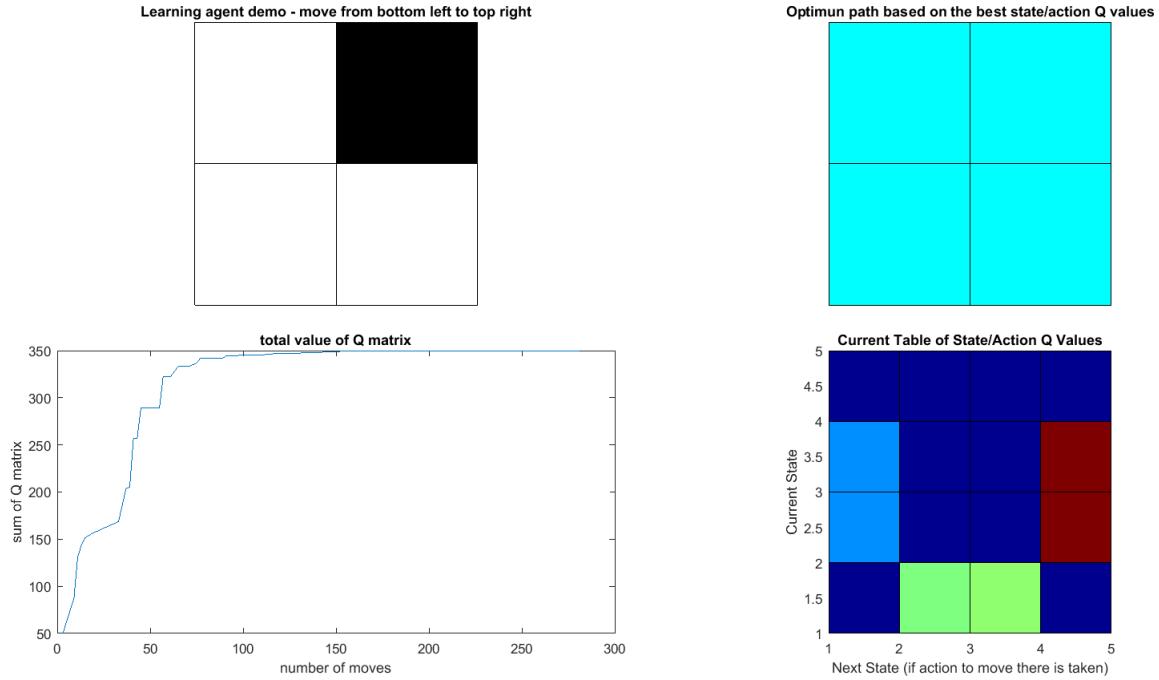


Figure 5.12: Sample run when $\epsilon = 0.9$

From figure 5.11 and 5.12, the time taken for the convergence of the Q to happen is at approximately 80 movements when ϵ is 0.1 and 280 movements when ϵ is 0.9. This indicates that indeed the learning time can be shortened by lowering the learning rate. However, the trade-off of the fast learning time can leads to an ineffective learning outcome.

As we can see from figure 5.11, one best route is detected with $\epsilon = 0.1$. But in figure 5.12, two best routes are found with $\epsilon = 0.9$. This further supports the argument that, fast learning does not necessarily result in an effective outcome. This is true in human learning. When we try to learn thing fast, often we make more mistakes, whereas when we take our time to discover more and learn thing thoroughly, making mistakes can be largely reduced from experience. This also emphasize the dilemma QL is facing to balance the trade-off between exploration and exploitation. Hence, a more effective way for ϵ value selection will be choose a high ϵ in the beginning at slowly decay ϵ over time as shown in figure 5.10 to maximise the QL outcome.

5.2 Experiment 2

From the discussion made in experiment 1, we have studied the relationship between the learning time and three variables α , γ and ϵ . With the suggestion given regards to the values selection for these variables, we are going to analyse the QL in a bigger and more complex context, a 5 x 5 maze.

In this experiment, the three proposed maze from chapter 4 are loaded into the simulation and the rest of the settings for the simulation are as follow:

- $\alpha = 0.8$
- $\gamma = 0.9$
- $\epsilon = 0.5$
- Sample size = 1

5.2.1 Simulation Results

The following three figure shows the results at the end of the three simulation run for the maze:

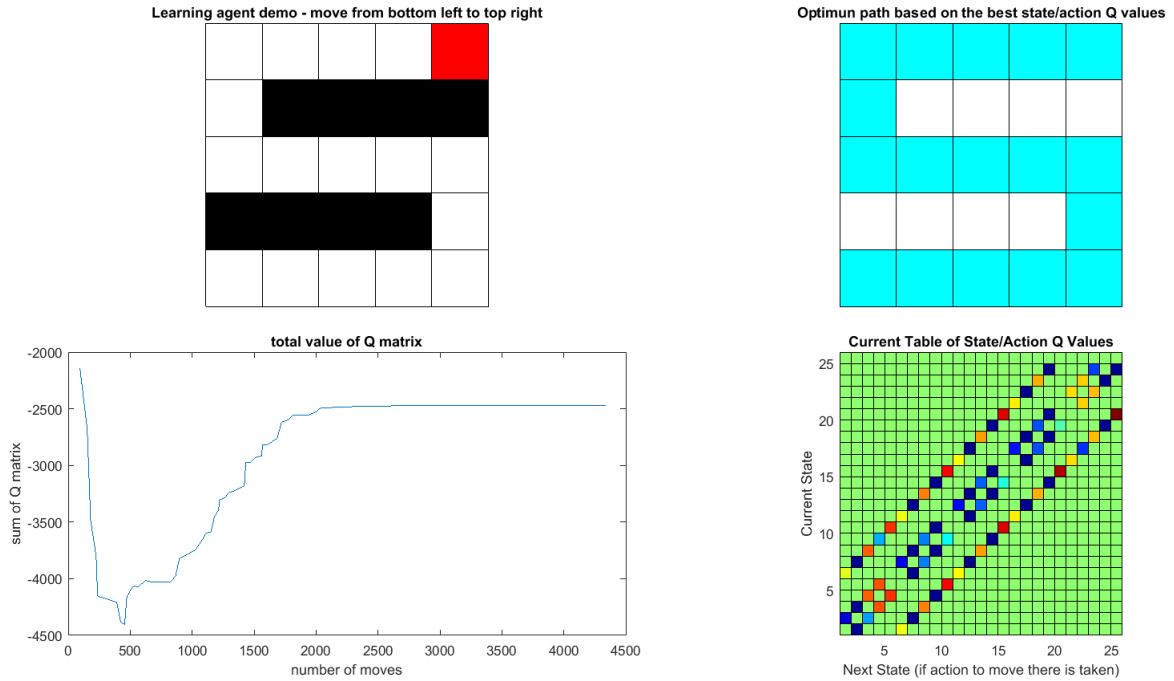


Figure 5.13: Sample run on maze 1

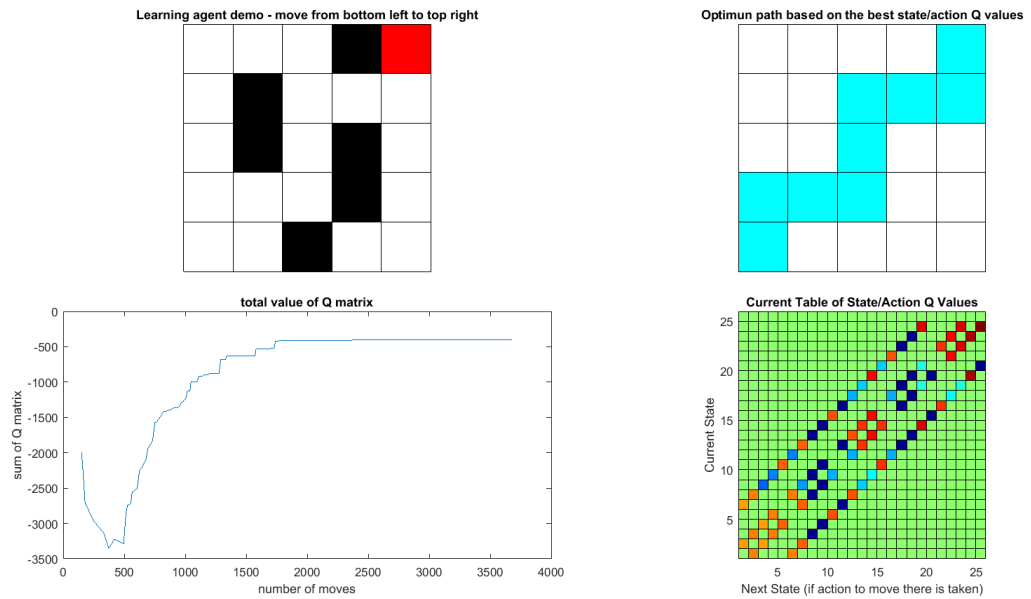


Figure 5.14: Sample run on maze 2

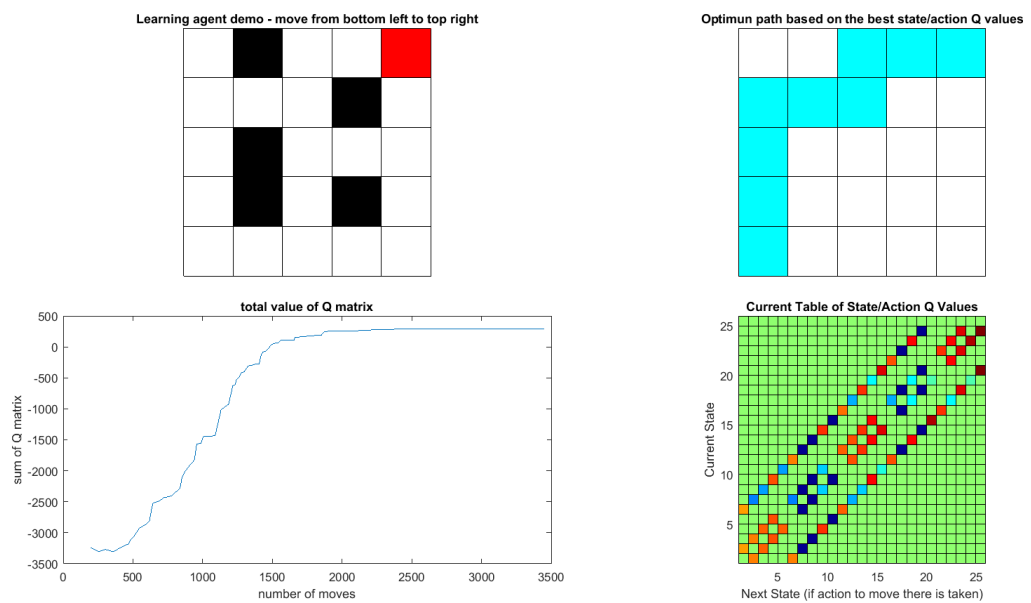


Figure 5.15: Sample run on maze 3

The table below summarizes the time taken and the number of steps in the best route for the three simulation:

Maze	Time /s	Steps
Maze 1	67.9327	15
Maze 2	60.172	7
Maze 3	55.8452	7

Table 5.4: Simulation results

5.2.2 Discussion

As we can see in the simulation results, all the three simulations have been able to provide the optimal path to the maze problem. The time taken for each simulation run are very similar with a difference of approximately 2 to 7 seconds. The time taken is within an acceptable range and it reflects the setting for simulation is reasonable that it does balance the parameter of fast learning vs thorough learning and also exploration vs exploitation we discuss in the experiment 1.

One thing to highlight here is the common characteristics of the three graphs and Q matrix updates demonstration from the simulation. As we can see, the sum of the Q values drops dramatically before regaining a rise and final convergence in all three simulation. This is because, comparing to the plain maze we used experiment 1, these 5 x 5 maze are introduced with some complexity. The common complexity in all three maze are the obstacles. These obstacles are the states that when an agent move in or out from the states will receive an immediate negative rewards.

This explains the drops of the sum of the Q values in the first half of the learning process where the agent are still constantly bumping into the obstacles and accumulate negative rewards. When the learning progress with more experience, the agent is able to identify whether or not the state is an obstacle and try to avoid moving into it. This is shown in the rise in the sum of the Q matrix and reflected in the Q matrix update demo. Recall from the previous chapter, the colour that we use in the plotting is the default colour map in Matlab Parula shown in figure 5.16.

**Figure 5.16:** Color map Parula [15]

The green value in the Q table demo represents the 0s in the matrix, whereas the blue ones represents the negative values. As the values increase, the colour will slowly change to orange, then to yellow and yellow colours are the best Q values in the matrix. The reason we did not see the green values in experiment 1 is because the 2 x 2 maze we used are without obstacle, which means there are no negative rewards, so smallest Q values in the matrix will be 0s which are represented by the blue colour in the colour map.

Although the QL is able to work out the best route in all three maze, the effectiveness and correctness need more analysis by comparing the result from the simulation with the result obtained by observation.



Figure 5.17: Comparison on maze 1



Figure 5.18: Comparison on maze 2

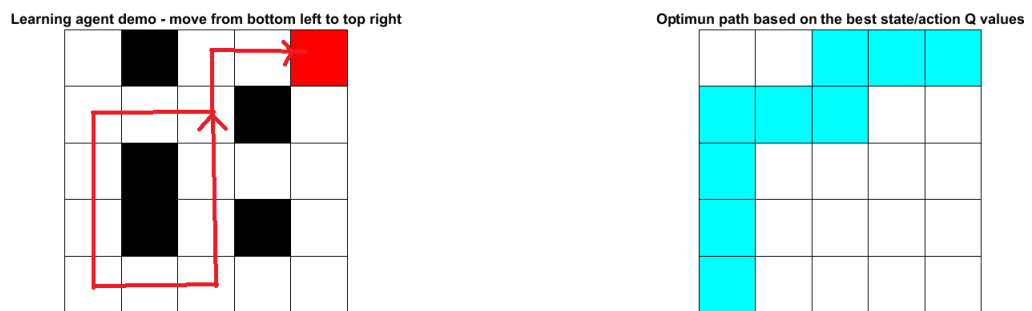


Figure 5.19: Comparison on maze 3

Figure 5.17 illustrate a comparison made for the best route calculated by observation with from simulation. As we can see that there is not one route to move from the starting

state to the goal state, and the simulation is able to identify this route.

However, the comparisons made in maze 2 and maze 3 do not fully prove the correctness of the value selection for the variables. In figure 5.18 and 5.19, we can see that there are two best routes that takes 7 steps to complete the two maze. However, the simulation only identify one of the route in each case. We can say that the simulation does work out the best route that meets the objectives and the effectiveness is not addressed in the design of simulation. This can be explained with the analysis drawn from experiment 1 regards to the value selection for α , γ and ϵ . A fixed value for the variable throughout the QL process proves to be not practical to achieve a high effectiveness in learning. Hence, methods like ϵ decay or exploration function that were suggested chapter 2 needs to be implemented to dynamically change the values of the variable, and with more samples from simulation, a more clear conclusion can be made.

5.3 Chapter Summary

In this chapter, we conducted the two experiments that are proposed in the previous chapter for the analysis of the time variables relationship and the validation of the QL algorithm in practice.

From experiment 1, we discovered that the time taken to complete the 2 X 2 maze is changed exponentially with a step change in the learning rate α and the time follows a linear change with the action selection variable ϵ . The discount factor γ , however, does not affect the learning time much in this 2 x 2 maze problem.

In experiment 2, we validated the correctness of the Matlab program by showing positive results in finding the best route to get to the exit point in the three sets of different 5 x 5 complex maze. By comparing these results with our observations of the solutions to the maze, we discovered some drawbacks with the fixed setting for the three variables. Hence, we concluded that methods that can dynamically change the variables, especially α and ϵ , needs to be devised in order to optimised the current QL algorithm for it to be implemented in a larger scale environment to solve more complex problems.

Chapter 6

Conclusions and Future Work

In this chapter, a general conclusion is given to summarise the works have been completed for this research thesis project and followed by a general discussion on how this research project can be extended to a larger context and applied into the field of AI development in the future.

6.1 Conclusions

The primary objectives of this research thesis was to design a Matlab executable program to aid the study of the Q learning method in reinforcement learning. The project looked into the fundamental background theories that contribute to the creation of the QL algorithm. In addition, two experiments are designed for firstly validation of the Matlab program built and secondly illustration of the QL theories in practice. The general conclusions of the project are listed as follow:

6.1.1 Q learning Theory

Through a thorough review of the literature on RL, we have summarized the key contents, such as MDP and RL, to illustrate the key concept of solving optimisation problems in the real world. We also looked at how the QL equation is devised by combining the theory of DP and MC through analysis of these two methods. With a mathematical demonstration of QL, we came out with an algorithm that can be implemented into the design of the Matlab program.

6.1.2 Matlab Program Design

We have designed an executable Matlab program that consists of a run script that implements the QL algorithm devised and a GUI for the convenience of demonstration and experiment analysis. The explanation of the important Matlab scripts, such as the conversion of the QL algorithm into Matlab programming language and design of the graphical

demonstration, are provided in detail. The completed Matlab scripts are provided as open resource for welcomed improvement and study.

6.1.3 Simulation Result Analysis

Two experiments were designed for the study of the relationship between the learning time and variables (α , γ and ϵ) and validation of the QL algorithm implementation in the program.

From the experiment on the relationship, we have discovered that the change in learning rate α and ϵ can have a certain degree of effects on the learning time, whereas the learning time does not vary much with the discount factor γ .

In the other experiment, we firstly validate the functional correctness of the program by comparing the results of the best route generated from the simulation and observation. Then we come to a conclusion that, the current QL algorithm needs to be further optimised so that it can be implemented into a bigger context to solve more complicated problems.

6.2 Future work

In the previous chapter, we discussed the relationship between the learning time and variables and the problems that exists in the current QL algorithm proposed in this project. This chapter briefly outlines the possible further research on the topic. The suggestions discussed herein are beyond the scope of this project, and may be implemented in the other research projects in the future. The thesis focused on the study of QL with analytical and simulation results. A more thorough analysis is required to further validate the relationship of between the learning time and the variables in the QL algorithm so that the existing QL algorithm can be optimised by devising methods to dynamically change the learning rate or other feasible action selection protocols. Work presented in the thesis can be extended in several directions. The following section will briefly cover some research avenues which could be explored in the future.

6.2.1 Optimisation on QL Algorithm

From the discussion we made in the previous chapter, we can see that the fixed variables scheme for the QL algorithm, is working but not effective. This results in an impractical experiment design for a more accurate analysis and conclusion drawn on the time variables relationship. This is because, the current time taken for the QL algorithm in the Matlab program to complete the 5 x 5 maze is approximate 1 min. The trade-off between the reasonable amounts of experiment samples taken for a more thorough analysis and the time taken for gathering one sample is too costly in terms of effectiveness of the experiment procedure. Hence, a better scheme to optimise the current QL algorithm might

be the next step to extend this project.

In chapter 2, four methods were introduced to tackle problems in exploration and exploitation:

- SoftMax
- Dynamic ϵ Decay
- Exploration Function
- Speedy Q-Learning

These four methods have the same terminology to dynamically change the agents learning process from exploration to exploitation based on either the ϵ , α value or the action selection schemes.

The proposed methods are to be implemented into the current QL algorithm from this project and tested on the 5 x 5 and 10 x 10 maze. An analysis on the time variables relationship will be drawn by comparing the outcome of the experiment results from these methods before the final optimisation scheme is devised.

6.2.2 QL in Machine Learning

In chapter 1, we outlined the current researches on ML and the importance of QL in the building of a smart AI. The next possible extension on this topic will be implementing the optimised QL algorithms in the following area:

Software AI will be designed perform tasks such as playing a chess game and optimisation in the design procedures of a satellite network.

Hardware Robots will be designed to perform tasks in automation industry or just simply mimic human body manoeuvre, such as walk or climb up a table.

Chapter 7

Abbreviations

MDP	Markov Decision Process
RL	Reinforcement learning
DP	Dynamic Programming
QL	Q-Learning
SQL	Speedy Q-Learning
TD	Temporal difference
AI	Artificial Intelligence
ML	Machine Learning
MC	Monte Carlo
MB	Model-based
MF	Model-free

Appendix A

Matlab Coding

A.1 Overview

The two Matlab files written for the simulation and GUI during the project will be provided here for reference of the readers.

A.2 Run file

```
1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %
3  % Q Learning of a single agent move in a N X N maze
4  %
5  % Author: Lu Han
6  % Original code by frog.ai http://frog.ai/blog/?p=39
7  %
8  % Modifications have been made to best suit the final year project
9  % objectives from Department of Engineering Department, Macquarie
10 % Univeristy, Jun 2016
11 %
12 % Version 1.0
13 % This code is provided as a reference to the interested reader
14 % Reuse of this code is welcome, with acknowledgement to Lu Han
15 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
16
17 %function that gets the user input from the GUI to start the simulation
18 function [episodes,steps,totalMoves,time] = ...
    run(numStateRows,numStateCols,alpha,gamma,epsilon,disSpeed,mazeLvl)
19
20 numStates = numStateRows*numStateCols;
21 % goal state
22 goalState = numStates;
23 % set linked states matrix (1 for linked; 0 for unlinked)
24 mLinked = zeros(numStates);
```

```

25
26 % allow linked states to be all adjacent neighbours excluding diagonal
27 for i = 1:numStates
28     % work out point of state i on mStates
29     [rowSi,colSi] = stateToRowCols(i,numStateRows,numStateCols);
30     % go through all options that are ok for state i for every next ...
        state j
31     for j = 1:numStates
32         % find points can move to and then convert to state number
33         % work out point of state j on mStates
34         [rowSj,colSj] = stateToRowCols(j,numStateRows,numStateCols);
35         if(colSj == colSi && (rowSj - rowSi == 1 || rowSj - rowSi == -1))
36             mLinked(i,j) = 1;
37         end
38         if(rowSj == rowSi && (colSj - colSi == 1 || colSj - colSi == -1))
39             mLinked(i,j) = 1;
40         end
41     end
42 end
43
44 % remove any links starting at goal state
45 mLinked(goalState,:) = 0;
46
47 % create q state/action matrices
48 QCurrent = zeros(numStates);
49 % create previous q state/action matrices with value set to be an ...
        arbitrary
50 % large number
51 Qprevious = ones(numStates) * inf;
52
53 %set the counter for Q matrix convergence
54 count = 0;
55
56 % set reward matrix
57 rewardM = zeros(numStates);
58 % anything leading to final state gets 100 rewards, anything leading ...
        to or
59 % out from the obstacle states gets a -100 rewards.
60 rewardM(:,goalState) = 100;
61 %pre-designed 5 x 5 maze
62 switch mazeLvl
63     case 1,
64     case 2,
65         rewardM(:,2) = -100;
66         rewardM(2,:) = -100;
67         rewardM(:,7) = -100;
68         rewardM(7,:) = -100;
69         rewardM(:,9) = -100;
70         rewardM(9,:) = -100;
71         rewardM(:,12) = -100;
72         rewardM(12,:) = -100;
73         rewardM(:,14) = -100;

```

```

74         rewardM(14,:) = -100;
75         rewardM(:,17) = -100;
76         rewardM(17,:) = -100;
77         rewardM(:,19) = -100;
78         rewardM(19,:) = -100;
79         rewardM(:,24) = -100;
80         rewardM(24,:) = -100;
81     case 3,
82         rewardM(:,8) = -100;
83         rewardM(8,:) = -100;
84         rewardM(:,9) = -100;
85         rewardM(9,:) = -100;
86         rewardM(:,11) = -100;
87         rewardM(11,:) = -100;
88         rewardM(:,17) = -100;
89         rewardM(17,:) = -100;
90         rewardM(:,18) = -100;
91         rewardM(18,:) = -100;
92         rewardM(:,20) = -100;
93         rewardM(20,:) = -100;
94
95     case 4,
96         rewardM(:,7) = -100;
97         rewardM(7,:) = -100;
98         rewardM(:,8) = -100;
99         rewardM(8,:) = -100;
100        rewardM(:,10) = -100;
101        rewardM(10,:) = -100;
102        rewardM(:,17) = -100;
103        rewardM(17,:) = -100;
104        rewardM(:,19) = -100;
105        rewardM(19,:) = -100;
106 end
107
108 %variables that tracks the total number of moves made and trials took
109 %before Q matrix is converged
110 totalMoves = 0;
111 episodes = 0;
112
113 %set condition that if Q is not converged, keep running
114 keepGoing = 1;
115
116 %start the timing for the simulation
117 tic
118 while keepGoing == 1;
119     episodes = episodes+1;
120     steps = 0;
121     mWatch = zeros(numStateRows,numStateCols);
122     initialState = 1;
123     % find initial state point on board
124     [rowSns,colSns] = ...
        stateToRowCols(initialState,numStateRows,numStateCols);

```

```

125
126 while (initialState ≠ goalState)
127     totMoves(epochs) = totalMoves;
128     totalMoves = totalMoves + 1;
129     steps = steps + 1;
130
131     % Q learning algorithm
132     optionVector = mLinked(initialState,:);
133     nextPt = 0;
134
135     % randomly pick one of the options until one is >0
136     while nextPt == 0
137         nextStateRandom = randi([1, numStates]);
138         if(optionVector(nextStateRandom) ≠ 2)
139             nextPt = optionVector(nextStateRandom);
140         end
141     end;
142
143     % pick the best next state from qvalues
144     nextStateVector = QCurrent(initialState,:);
145     possibleQ = mLinked(initialState,:).*nextStateVector;
146     nextPt = 0;
147     nextStateBestQ = 0;
148     nextStateBest = 0;
149     for ii = 1:numStates
150         if possibleQ(ii)>nextStateBestQ
151             nextStateBestQ = possibleQ(ii);
152             nextStateBest = ii;
153         end
154     end;
155
156     % pick best option 1-epsilon of time-1 and random other times
157     % if epsilon = 1 always picks random next state - is e-greedy
158     if (rand < (1-epsilon))&&(nextStateBest>0)
159         nextState = nextStateBest;
160     else
161         nextState = nextStateRandom;
162     end;
163
164     % split Q update into two parts
165     part1 = ((1-alpha)*QCurrent(initialState,nextState));
166     part2 = alpha*((rewardM(initialState,nextState)+(gamma*(max( ...
167         mLinked(nextState, :).*QCurrent(nextState, :))))));
168     QCurrent(initialState,nextState) = part1 + part2;
169
170     % find next state point on board
171     [rowSns,colSns] = ...
172         stateToRowCols(nextState,numStateRows,numStateCols);
173
174     % Q learning agent movement demo
175     mWatch(rowSns,colSns) = 1;
176     padded=padarray(mWatch,[1,1],'post');

```

```

175         %set the obstacle state color to be black
176         switch mazeLvl
177             case 1,
178             case 2,
179                 padded(2)=2;
180                 padded(8)=2;
181                 padded(10)=2;
182                 padded(14)=2;
183                 padded(16)=2;
184                 padded(20)=2;
185                 padded(22)=2;
186                 padded(28)=2;
187             case 3,
188                 padded(9)=2;
189                 padded(10)=2;
190                 padded(13)=2;
191                 padded(20)=2;
192                 padded(21)=2;
193                 padded(23)=2;
194             case 4,
195                 padded(8)=2;
196                 padded(9)=2;
197                 padded(11)=2;
198                 padded(20)=2;
199                 padded(22)=2;
200         end
201
202         fig1 = subplot(4,4,[1,2,5,6]);
203         pcolor(padded)
204         my_map = [1,1,1;
205                 1,0,0;
206                 0,0,0];
207         colormap(fig1,my_map);
208         title('Learning agent demo – move from bottom left to top right')
209         axis off;
210         axis square;
211         pause(disSpeed)
212         mWatch(rowSns,colSns) = 0;
213         initialState = nextState;
214     end
215
216     % draw value Q table
217     padded2=padarray(Qprevious,[1,1],'post');
218     fig2 = subplot(4,4,[11,12,15,16]);
219     pcolor(padded2)
220     colormap(fig2,jet);
221     title('Current Table of State/Action Q Values')
222     axis square;
223     xlabel('Next State (if action to move there is taken)')
224     ylabel('Current State')
225
226     % draw the sum of the total Q matrix against

```

```

227     Qconverge(episodes) = sum(sum(Qprevious));
228     subplot(4,4,[9,10,13,14]);
229     plot(totMoves,Qconverge)
230     title('total value of Q matrix')
231     xlabel('number of moves')
232     ylabel('sum of Q matrix')
233
234     % break if convergence: small deviation on q for 10 consecutive
235     if sum(sum(abs(Qprevious-QCurrent)))<0.0001 && sum(sum(QCurrent >0))
236         if count>10
237             keepGoing = 0;
238         else
239             count=count+1; % set counter if deviation of q is small
240         end
241     else
242         Qprevious = QCurrent;
243         count=0; % reset counter when deviation of q from previous q ...
                is large
244     end
245
246 end
247 %stop the timing
248 time = toc;
249 time
250
251 %function that generates best route based on the best Q values from ...
    the Q
252 %matrix
253 function bestQ = ...
    optimunPath(numStateRows,numStateCols,numStates,goalState,QCurrent)
254 bestQ = zeros(numStateRows,numStateCols);
255 current = 1;
256 bestQ(current) = 1;
257 bestQ(numStates) = 1;
258 maxQ = 0;
259 while (current ≠ goalState)
260     bestVector = QCurrent(current,:);
261     for i = 1:numStates
262         if bestVector(i)>maxQ
263             maxQ = bestVector(i);
264             current = i;
265             bestQ(current) = 1;
266         end
267     end;
268 end
269
270 padded=padarray(bestQ,[1,1],'post');
271 fig3 = subplot(4,4,[3,4,7,8]);
272 pcolor(padded)
273 my_map = [1,1,1;
274           1,0,0;
275           0,1,1];

```



```

276 colormap(fig3,my_map);
277 axis off;
278 axis square;
279 title('Optimum path based on the best state/action Q values')
280
281
282 % function gets the column and row numbers of a given state number
283 function [row,col] = stateToRowCols(stateNum,numStateRows,numStateCols)
284 col = ceil(stateNum/numStateCols);
285 row = (stateNum+numStateRows)-(numStateRows*col);

```

A.3 GUI file

```

1  %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
2  %
3  % Q Learning of a single agent move in a N X N maze
4  %
5  % Author: Lu Han
6  %
7  % Final year projec from Department of Engineering Department, Macquarie
8  % Univeristy, Jun 2016
9  %
10 % Version 1.0
11
12 % This code is provided as a reference to the interested reader
13 % Reuse of this code is welcome, with acknowledgement to Lu Han
14 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
15
16 %GUI file that links to the run.m file
17 function varargout = config(varargin)
18 gui.Singleton = 1;
19 gui.State = struct('gui_Name',      mfilename, ...
20                   'gui_Singleton',  gui.Singleton, ...
21                   'gui_OpeningFcn', @config_OpeningFcn, ...
22                   'gui_OutputFcn',  @config_OutputFcn, ...
23                   'gui_LayoutFcn',  [] , ...
24                   'gui_Callback',   []);
25 if nargin && ischar(varargin{1})
26     gui.State.gui_Callback = str2func(varargin{1});
27 end
28
29 if nargout
30     [varargout{1:nargout}] = gui_mainfcn(gui.State, varargin{:});
31 else
32     gui_mainfcn(gui.State, varargin{:});
33 end
34 % End initialization code — DO NOT EDIT
35

```

```

36
37 % — Executes just before config is made visible.
38 function config_OpeningFcn(hObject, eventdata, handles, varargin)
39 % This function has no output args, see OutputFcn.
40 % hObject    handle to figure
41 % eventdata  reserved — to be defined in a future version of MATLAB
42 % handles    structure with handles and user data (see GUIDATA)
43 % varargin   command line arguments to config (see VARARGIN)
44
45 % Choose default command line output for config
46 handles.output = hObject;
47
48 % Update handles structure
49 guidata(hObject, handles);
50
51 % UIWAIT makes config wait for user response (see UIRESUME)
52 % uiwait(handles.figure1);
53
54
55 % — Outputs from this function are returned to the command line.
56 function varargout = config_OutputFcn(hObject, eventdata, handles)
57 % varargout  cell array for returning output args (see VARARGOUT);
58 % hObject    handle to figure
59 % eventdata  reserved — to be defined in a future version of MATLAB
60 % handles    structure with handles and user data (see GUIDATA)
61
62 % Get default command line output from handles structure
63 varargout{1} = handles.output;
64
65
66 % — Executes on button press in Run.
67 function Run_Callback(hObject, eventdata, handles)
68 % hObject    handle to Run (see GCBO)
69 % eventdata  reserved — to be defined in a future version of MATLAB
70 % handles    structure with handles and user data (see GUIDATA)
71
72 %get the value of input from GUI
73 numStateRows = str2double(get(handles.numStateRows, 'string'));
74 numStateCols = str2double(get(handles.numStateCols, 'string'));
75 alpha = str2double(get(handles.alpha, 'string'));
76 gamma = str2double(get(handles.gamma, 'string'));
77 epsilon =str2double(get(handles.epsilon, 'string'));
78 disSpeed =str2double(get(handles.disSpeed, 'string'));
79 sampleSize =str2double(get(handles.sampleSize, 'string'));
80 sumEpisodes = 0;
81 sumSteps = 0;
82 sumTotalMoves = 0;
83 sumTime = 0;
84 mazeLvl = get(handles.listbox, 'Value');
85
86 %sampling process
87 for sampleRun = 1 : sampleSize

```

```
88 figure
89 %put the input from the GUI into run.m file to start simulation
90 [episodes,steps,totalMoves,time] = ...
    run(numStateRows,numStateCols,alpha,gamma,epsilon,disSpeed,mazeLvl);
91 switch sampleRun
92     case 1,
93         set(handles.episodes_1,'string',episodes)
94         set(handles.steps_1,'string',steps)
95         set(handles.totalMoves_1,'string',totalMoves)
96         set(handles.time_1,'string',time)
97     case 2,
98         set(handles.episodes_2,'string',episodes)
99         set(handles.steps_2,'string',steps)
100        set(handles.totalMoves_2,'string',totalMoves)
101        set(handles.time_2,'string',time)
102    case 3,
103        set(handles.episodes_3,'string',episodes)
104        set(handles.steps_3,'string',steps)
105        set(handles.totalMoves_3,'string',totalMoves)
106        set(handles.time_3,'string',time)
107    case 4,
108        set(handles.episodes_4,'string',episodes)
109        set(handles.steps_4,'string',steps)
110        set(handles.totalMoves_4,'string',totalMoves)
111        set(handles.time_4,'string',time)
112    case 5,
113        set(handles.episodes_5,'string',episodes)
114        set(handles.steps_5,'string',steps)
115        set(handles.totalMoves_5,'string',totalMoves)
116        set(handles.time_5,'string',time)
117    case 6,
118        set(handles.episodes_6,'string',episodes)
119        set(handles.steps_6,'string',steps)
120        set(handles.totalMoves_6,'string',totalMoves)
121        set(handles.time_6,'string',time)
122    case 7,
123        set(handles.episodes_7,'string',episodes)
124        set(handles.steps_7,'string',steps)
125        set(handles.totalMoves_7,'string',totalMoves)
126        set(handles.time_7,'string',time)
127    case 8,
128        set(handles.episodes_8,'string',episodes)
129        set(handles.steps_8,'string',steps)
130        set(handles.totalMoves_8,'string',totalMoves)
131        set(handles.time_8,'string',time)
132    case 9,
133        set(handles.episodes_9,'string',episodes)
134        set(handles.steps_9,'string',steps)
135        set(handles.totalMoves_9,'string',totalMoves)
136        set(handles.time_9,'string',time)
137    case 10,
138        set(handles.episodes_10,'string',episodes)
```

```

139         set(handles.steps_10,'string',steps)
140         set(handles.totalMoves_10,'string',totalMoves)
141         set(handles.time_10,'string',time)
142     end
143
144     sumEpisodes = sumEpisodes+ episodes;
145     sumSteps = sumSteps + steps;
146     sumTotalMoves = sumTotalMoves + totalMoves;
147     sumTime = sumTime + time;
148 end
149
150 %calculate the sample average
151 aveEpisodes = sumEpisodes / sampleRun;
152 aveSteps = sumSteps / sampleRun;
153 aveTotalMoves = sumTotalMoves / sampleRun;
154 aveTime = sumTime / sampleRun;
155
156 %display the sample average in the GUI
157 set(handles.sumEpisodes,'string',sumEpisodes)
158 set(handles.sumSteps,'string',sumSteps)
159 set(handles.sumTotalMoves,'string',sumTotalMoves)
160 set(handles.sumTime,'string',sumTime)
161
162 set(handles.aveEpisodes,'string',aveEpisodes)
163 set(handles.aveSteps,'string',aveSteps)
164 set(handles.aveTotalMoves,'string',aveTotalMoves)
165 set(handles.aveTime,'string',aveTime)
166
167
168 function alpha_Callback(hObject, eventdata, handles)
169 % hObject      handle to alpha (see GCBO)
170 % eventdata    reserved – to be defined in a future version of MATLAB
171 % handles      structure with handles and user data (see GUIDATA)
172
173 % Hints: get(hObject,'String') returns contents of alpha as text
174 %         str2double(get(hObject,'String')) returns contents of alpha ...
175 %         as a double
176
177 % — Executes during object creation, after setting all properties.
178 function alpha_CreateFcn(hObject, eventdata, handles)
179 % hObject      handle to alpha (see GCBO)
180 % eventdata    reserved – to be defined in a future version of MATLAB
181 % handles      empty – handles not created until after all CreateFcns ...
182 %             called
183
184 % Hint: edit controls usually have a white background on Windows.
185 %       See ISPC and COMPUTER.
186 if ispc && isequal(get(hObject,'BackgroundColor'), ...
187     get(0,'defaultUiControlBackgroundColor'))
188     set(hObject,'BackgroundColor','white');
189 end

```

```
188
189
190
191 function gamma.Callback(hObject, eventdata, handles)
192 % hObject      handle to gamma (see GCBO)
193 % eventdata    reserved – to be defined in a future version of MATLAB
194 % handles      structure with handles and user data (see GUIDATA)
195
196 % Hints: get(hObject,'String') returns contents of gamma as text
197 %          str2double(get(hObject,'String')) returns contents of gamma ...
           as a double
198
199
200 % — Executes during object creation, after setting all properties.
201 function gamma.CreateFcn(hObject, eventdata, handles)
202 % hObject      handle to gamma (see GCBO)
203 % eventdata    reserved – to be defined in a future version of MATLAB
204 % handles      empty – handles not created until after all CreateFcns ...
           called
205
206 % Hint: edit controls usually have a white background on Windows.
207 %          See ISPC and COMPUTER.
208 if ispc && isequal(get(hObject,'BackgroundColor'), ...
           get(0,'defaultUicontrolBackgroundColor'))
209     set(hObject,'BackgroundColor','white');
210 end
211
212
213
214 function epsilon.Callback(hObject, eventdata, handles)
215 % hObject      handle to epsilon (see GCBO)
216 % eventdata    reserved – to be defined in a future version of MATLAB
217 % handles      structure with handles and user data (see GUIDATA)
218
219 % Hints: get(hObject,'String') returns contents of epsilon as text
220 %          str2double(get(hObject,'String')) returns contents of ...
           epsilon as a double
221
222
223 % — Executes during object creation, after setting all properties.
224 function epsilon.CreateFcn(hObject, eventdata, handles)
225 % hObject      handle to epsilon (see GCBO)
226 % eventdata    reserved – to be defined in a future version of MATLAB
227 % handles      empty – handles not created until after all CreateFcns ...
           called
228
229 % Hint: edit controls usually have a white background on Windows.
230 %          See ISPC and COMPUTER.
231 if ispc && isequal(get(hObject,'BackgroundColor'), ...
           get(0,'defaultUicontrolBackgroundColor'))
232     set(hObject,'BackgroundColor','white');
233 end
```

```

234
235 function numStateRows_Callback(hObject, eventdata, handles)
236 % hObject      handle to numStateRows (see GCBO)
237 % eventdata    reserved – to be defined in a future version of MATLAB
238 % handles      structure with handles and user data (see GUIDATA)
239
240 % Hints: get(hObject,'String') returns contents of numStateRows as text
241 %           str2double(get(hObject,'String')) returns contents of ...
           numStateRows as a double
242
243
244 % — Executes during object creation, after setting all properties.
245 function numStateRows_CreateFcn(hObject, eventdata, handles)
246 % hObject      handle to numStateRows (see GCBO)
247 % eventdata    reserved – to be defined in a future version of MATLAB
248 % handles      empty – handles not created until after all CreateFcns ...
           called
249
250 % Hint: edit controls usually have a white background on Windows.
251 %           See ISPC and COMPUTER.
252 if ispc && isequal(get(hObject,'BackgroundColor'), ...
           get(0,'defaultUicontrolBackgroundColor'))
253     set(hObject,'BackgroundColor','white');
254 end
255
256
257
258 function numStateCols_Callback(hObject, eventdata, handles)
259 % hObject      handle to numStateCols (see GCBO)
260 % eventdata    reserved – to be defined in a future version of MATLAB
261 % handles      structure with handles and user data (see GUIDATA)
262
263 % Hints: get(hObject,'String') returns contents of numStateCols as text
264 %           str2double(get(hObject,'String')) returns contents of ...
           numStateCols as a double
265
266
267 % — Executes during object creation, after setting all properties.
268 function numStateCols_CreateFcn(hObject, eventdata, handles)
269 % hObject      handle to numStateCols (see GCBO)
270 % eventdata    reserved – to be defined in a future version of MATLAB
271 % handles      empty – handles not created until after all CreateFcns ...
           called
272
273 % Hint: edit controls usually have a white background on Windows.
274 %           See ISPC and COMPUTER.
275 if ispc && isequal(get(hObject,'BackgroundColor'), ...
           get(0,'defaultUicontrolBackgroundColor'))
276     set(hObject,'BackgroundColor','white');
277 end
278
279

```

```

280 function disSpeed.Callback(hObject, eventdata, handles)
281 % hObject      handle to disSpeed (see GCBO)
282 % eventdata    reserved – to be defined in a future version of MATLAB
283 % handles      structure with handles and user data (see GUIDATA)
284
285 % Hints: get(hObject,'String') returns contents of disSpeed as text
286 %           str2double(get(hObject,'String')) returns contents of ...
           disSpeed as a double
287
288
289 % — Executes during object creation, after setting all properties.
290 function disSpeed.CreateFcn(hObject, eventdata, handles)
291 % hObject      handle to disSpeed (see GCBO)
292 % eventdata    reserved – to be defined in a future version of MATLAB
293 % handles      empty – handles not created until after all CreateFcns ...
           called
294
295 % Hint: edit controls usually have a white background on Windows.
296 %           See ISPC and COMPUTER.
297 if ispc && isequal(get(hObject,'BackgroundColor'), ...
           get(0,'defaultUicontrolBackgroundColor'))
298     set(hObject,'BackgroundColor','white');
299 end
300
301
302 function sampleSize.Callback(hObject, eventdata, handles)
303 % hObject      handle to sampleSize (see GCBO)
304 % eventdata    reserved – to be defined in a future version of MATLAB
305 % handles      structure with handles and user data (see GUIDATA)
306
307 % Hints: get(hObject,'String') returns contents of sampleSize as text
308 %           str2double(get(hObject,'String')) returns contents of ...
           sampleSize as a double
309
310
311 % — Executes during object creation, after setting all properties.
312 function sampleSize.CreateFcn(hObject, eventdata, handles)
313 % hObject      handle to sampleSize (see GCBO)
314 % eventdata    reserved – to be defined in a future version of MATLAB
315 % handles      empty – handles not created until after all CreateFcns ...
           called
316
317 % Hint: edit controls usually have a white background on Windows.
318 %           See ISPC and COMPUTER.
319 if ispc && isequal(get(hObject,'BackgroundColor'), ...
           get(0,'defaultUicontrolBackgroundColor'))
320     set(hObject,'BackgroundColor','white');
321 end
322
323
324 % — Executes on selection change in listbox.
325 function listBox.Callback(hObject, eventdata, handles)

```

```
326 % hObject      handle to listbox (see GCBO)
327 % eventdata    reserved — to be defined in a future version of MATLAB
328 % handles      structure with handles and user data (see GUIDATA)
329
330 % Hints: contents = cellstr(get(hObject,'String')) returns listbox ...
      contents as cell array
331 %      contents{get(hObject,'Value')} returns selected item from ...
      listbox
332
333
334 % — Executes during object creation, after setting all properties.
335 function listbox_CreateFcn(hObject, eventdata, handles)
336 % hObject      handle to listbox (see GCBO)
337 % eventdata    reserved — to be defined in a future version of MATLAB
338 % handles      empty — handles not created until after all CreateFcns ...
      called
339
340 % Hint: listbox controls usually have a white background on Windows.
341 %      See ISPC and COMPUTER.
342 if ispc && isequal(get(hObject,'BackgroundColor'), ...
      get(0,'defaultUicontrolBackgroundColor'))
343     set(hObject,'BackgroundColor','white');
344 end
```


Appendix B

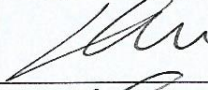
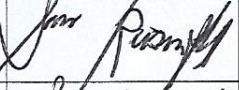
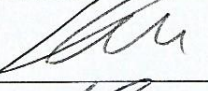
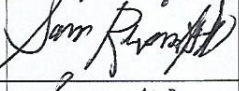
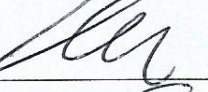
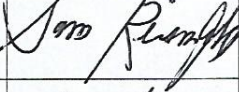
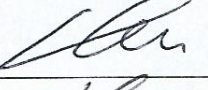
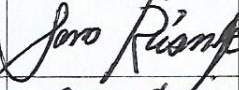
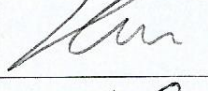
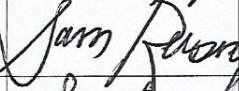
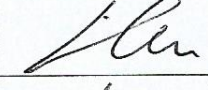
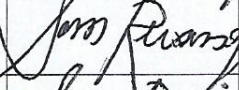
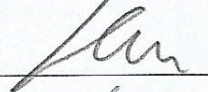
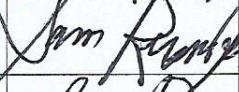
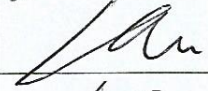
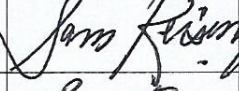
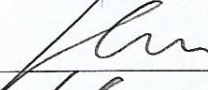
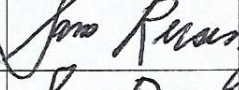
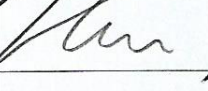
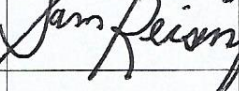
Project Plan and Attendance Form

B.1 Overview

This section contains the consultation meetings attendance form as required by the department. Both the supervisor and the student had to sign off the consultation meetings form for the official record of the meetings.

B.2 Consultation Meetings Attendance Form

Consultation Meetings Attendance Form

Week	Date	Comments (if applicable)	Student's Signature	Supervisor's Signature
2	10/03/16	Q-learning Concept project design		
3	18/03/16	Project specification		
4	23/03/16	Q learning Progress		
5	31/03/16	Q learning Progress		
7	29/04/16	Matlab coding		
8	06/05/16	Matlab coding		
9	13/05/16	Matlab code demo		
10	20/05/16	Results demo		
11	27/05/16	Final report		
12	03/06/16	Final report		

Bibliography

- [1] *Generalization in reinforcement learning: Successful examples using sparse coarse coding.* The MIT Press, 1996, vol. In David S. Touretzky, Michael C. Mozer, and Michael E. Hasselmo, editors, *Advances in Neural Information Processing Systems*, volume 8.
- [2] R. Bellman, "A markovian decision process," *Journal of Mathematics and Mechanics*, vol. 6, pp. 427,435, 1957.
- [3] G. R. Blog. (2016, jan) Alphago: Mastering the ancient game of go with machine learning. [Online]. Available: <https://research.googleblog.com/2016/01/alphago-mastering-ancient-game-of-go.html>
- [4] J. Blynel, "Reinforcement learning on real robots," Phd Thesis, University of Aarhus, may 2000.
- [5] L. W. D. Ernst, M. Glavic, "Power systems stability control: reinforcement learning framework," *IEEE*, pp. 427,435, 2004.
- [6] D. H. David Silver, Aja Huang, "Mastering the game of go with deep neural networks and tree search," *Nature* 529, jan 2016.
- [7] Y. M. E Even Dar, "Learning rates for q-learning," *Journal of Machine Learning Research*, 2003.
- [8] frog.ai. (2015, mar) Q learning example in matlab. [Online]. Available: <http://frog.ai/blog/?p=39>
- [9] C. Gaskett, "Q-learning for robot control," Phd Thesis, The Australian National University, jun 2002.
- [10] D. C. Inman Harvey, Phil Husbands, "Evolutionary robotics: the sussex approach," *Robotics and Autonomous Systems*, jan 1997.
- [11] R. J. W. Jing Peng, "Incremental multi-step qlearning. machine learning," jan 1996.
- [12] D. Klein and P. Abbeel, "Cs 188: Artificial intelligence markov decision process," *University of California, Berkeley*, vol. I.

- [13] —, “Cs 188: Artificial intelligence reinforcement learning,” *University of California, Berkeley*, vol. I.
- [14] J. S. Liu, “Monte carlo strategies in scientific computing,” Springer Series in Statistics, jan 2001.
- [15] MathWorks. colormap. [Online]. Available: <http://au.mathworks.com/help/matlab/ref/colormap.html>
- [16] —. padarray. [Online]. Available: <http://au.mathworks.com/help/images/ref/padarray.html>
- [17] T. M. Mitchell, *Machine Learning*, 2nd ed. McGraw-Hill Science/Engineering/Math, mar 1997.
- [18] M. G. Mohammad Gheshlaghi Azar, Remi Munos, “Reinforcement learning with a near optimal rate of convergence,” Technical Report inria-00636615, 2011.
- [19] —, “Speedy q-learning,” Radboud University Nijmegen Geert Grooteplein 21N, 6525 EZ Nijmegen, Netherlands, 2011.
- [20] B. News. (2016, mar) Artificial intelligence: Google’s alphago beats go master lee se-dol. [Online]. Available: <http://www.bbc.com/news/technology-35785875>
- [21] —. (2016, jan) Google achieves ai ‘breakthrough’ by beating go champion. [Online]. Available: <http://www.bbc.com/news/technology-35420579>
- [22] S. S. Richard Sutton, Doina Precup, “Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning,” *Artificial Intelligence*, 1999.
- [23] M. Ricordeau, “Q-concept-learning: generalization with concept lattice representation in reinforcement learning,” 15th IEEE International Conference on Tools with Artificial Intelligence, jan 2003.
- [24] SCS. Probabilistic robotics tutorial. [Online]. Available: <https://www.cs.cmu.edu/~thrun/tutorial/img104.GIF>
- [25] L.-J. L. Steven D. Whitehead, “Reinforcement learning of non-markov decision processes,” *Artificial Intelligence*, jan 1995.
- [26] R. S. Sutton, “Integrated architectures for learning, planning, and reacting based on approximating dynamic programming.” In Proceedings of the Seventh International Conference on Machine Learning, Morgan Kaufmann, nov 1990.
- [27] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. The MIT Press Cambridge,Massachusetts, 2012.

- [28] G. W. Taylor, "Reinforcement learning for parameter control of image-based applications," Phd Thesis, University of Waterloo, may 2004.
- [29] K. Teknomo, "Q-learning tutorial," Online: Revoledu.com, 2013.
- [30] R. v. U. Tim Eden, Anthony Knittel. Cs9417 reinforcement learning - q-learning. [Online]. Available: <http://www.cse.unsw.edu.au/~cs9417ml/RL1/index.html>
- [31] C. G. D. Wasser, "An object-oriented representation for efficient reinforcement learning," Phd Thesis, The State University of New Jersey, oct 2010.
- [32] N. Wiener, *Cybernetics, or Control and Communication in the Animal and the Machine*. Cambridge: MIT Press, 1948.
- [33] J. Wyatt, "Issues in putting reinforcement learning onto robots," MobileRoboticsWorkshop,10thBiennialConferenceof the AISB, jan 1995.