NUI Galway
OÉ Gaillimh

# An Empirical Analysis of the Parameters of the Q-Learning Algorithm and their Impact on Agent Performance in Reinforcement Learning

Darren King

Supervisor: Dr. Enda Howley

March 2017

# Abstract

The development of the field known as Potential Based Reward Shaping by Andrew Ng et al in 1999 has proved to be one of the most significant modern advancements in the field of Reinforcement Learning. By the relatively straightforward introduction of an additional shaping reward function to a learning agents training function, Ng et al showed that Agent performance could be drastically improved as the agent could now make use of valuable and accurate heuristics that were previously unavailable to it. This project aims to evaluate the effects the parameters of the Q-Learning algorithm, one of the most important developments in the field of temporal difference learning, have on an agent's performance. Due to the popularity of Potential Based Reward Shaping, this project will also evaluate the effects of these parameters for Q-Learning agents that implement Potential Based Reward Shaping. In this project a Reinforcement Learning simulator is built using Java that will measure learning agents' performance as they attempt to navigate a gridworld maze.



# Acknowledgements

I would like to take this opportunity to thank my project supervisor Dr. Enda Howley for all his support, advice and patience he showed me with regards to this project. His wealth of knowledge in the area of Reinforcement Learning was extremely beneficial to me throughout the course of this project, without which I have no doubt I would have struggled to complete this project.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Overview

The field of reward shaping allows for a learning agent to make use of prior knowledge of its environment. The underlying idea of Reward Shaping is to modify a reward function that provides the learning agent with an additional reward for exhibiting desirable behaviour. Probably the most significant advancement in this field is potential based reward shaping proposed by Andrew Ng et all [2] in 1999. It effectively tackles the issue of the temporal credit assignment problem outlined by Randlov and Alstrom [6] through use of a potential function that rewards the agent for desirable behaviour but also penalises it for undesirable behaviour.

The main focus of this research is to examine the effects of changing parameters in a Reinforcement learning algorithm. By performing this analysis we will have a better understanding of how to tweak the parameters of an RL algorithm in order to get the learning agent to perform efficiently and to train itself in its environment in as few steps as possible. A decrease in the training time is desirable in almost all Reinforcement Learning problems and this research will allow for future work in the field to achieve the maximum performance from their RL implementations without having to extensively test and tweak the parameters of their algorithm.

The research will focus on the Q-Learning algorithm, a temporal difference learning method developed by Watkins in 1989 [10]. The Q-Learning algorithm contains a number of different parameters, several of which are variables that can be altered by the user depending on the type of learning task the algorithm is being used to evaluate. These variables have a significant impact on the performance of the learning agent and so the focus of this research will be to examine these variables individually and measure their impact on that performance. In order to remove as much uncertainty from testing as possible, the experiments will be conducted in a deterministic gridworld environment. It will be examined in conjunction with Potential Based Reward Shaping (PBRS) developed by Ng et all in 1999 [2]. Each parameter will be examined for a learning agent that uses the Q-Learning algorithm with and without Potential Based Reward Shaping.

## 1.2 Research Goals

As part of this research I hope to discover the optimal parameter values to use in the Q-Learning formula in order to maximise a learning agent's performance. I also hope that the findings of my research can and will be used by other reseachers in the area of Reinforcement learning in order for them to achieve maximum performance with their Reinforcement Learning implementations.

# Chapter 2

# Background Research

## 2.1 An Introduction to Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning that was first developed in the late 1980's. It is an unsupervised machine learning algorithm that uses trial and error methods to maximise the "reward" it receives from the decisions it makes. It does this by computing an action value function for each possible choice in its environment and then uses these action value functions when deciding what action to take. In some RL implementations, a numerical reward is given after each choice it makes, in other implementations a reward is only given when the goal state is reached. Initially it chooses randomly as it has no previous knowledge/estimate of the reward it will receive if it makes a given choice. After it has made a few choices it may be able to estimate the reward it will receive if it makes a given choice. However it can only estimate potential reward if it has chosen that "choice"/option before. This gives rise to the core concept of reinforcement learning; Exploration vs Exploitation. An "agent" must decide, for any given choice it makes, whether to exploit the information it already has on the choices available to it (estimated reward for a given choice) or to explore new choices. Exploitation may be beneficial in the short term but in the long run exploration may be the most beneficial as it may result in the agent finding the optimal solution.

There are 4 key terms that a person must understand if they are to grasp the idea behind any Reinforcement Learning System. These are: Policy, Reward Function, Value Function and Model. However before we explain these 4 key ideas let us first define the concept of an "Agent".

### 2.1.1 An Agent

In the context of Reinforcement Learning, an agent is any system, person or learner that interacts with the surrounding environment but makes decisions/choices independent of that environment. An example of this would be a chess player in a game of chess. The player or "agent" can make any move it wants provided it is legal within the environment (The agent can only move one chess piece per move). The agent's choice will be informed by the environment (The current state of the game). Ultimately however the agent can move any piece it likes and makes its own move.

Eg: It may sacrifice a pawn even though it could have taken out an opposing knight.

### 2.1.2 4 elements of a Reinforcement Learning System

Now that we have defined the agent in the context of RL, let us define the 4 main elements of any reinforcement learning system.

#### Policy: $\pi$

In a RL system the "policy" defines the learning agent's way of behaving at any given time. The learning agent may change its policy based on the results of its experience within an environment/system. Every RL algorithm attempts to compute an optimal policy $\pi^*$

#### Reward Function: R

The "Reward Function" is the numerical reward received by a learning agent after each choice it makes in a RL system. The agent's sole goal is to maximise the total reward it receives in the long run based on the choices it makes. The rewards are defined by the system and independent of the agent (The agent cannot alter the reward it receives). However the agent can alter its policy in order to maximise its reward.

#### Value Function: V

The Reward function specifies immediate reward received by the system. The value function is a function of future rewards that the agent can expect to accumulate in the long run given the current policy. It takes into account the states that are likely to follow and the expected rewards available from those states. Eg: A state may have a low immediate Reward Function but the states following it are known to have a high value function resulting in a greater reward in the long run.

#### Model of the Environment

This is simply a model of the system in which the learning agent is acting within. Given a state and an action the model may be able to predict the next reward function. Advanced Reinforcement learning solutions can, given enough data, build a model of their surrounding environment and use it to plan their next move. The model can be used to alter the learning agent's policy. For large environments, computing a model of the environment becomes increasingly difficult and computationally expensive.

## 2.2 Current State of the Art

The field of Reinforcement Learning is an exciting one and it has come a long way since its conception in the late 1980's. The field is a combination of three separate threads of study: Trial and Error, Optimal Control and Temporal Difference

Learning. Since the 1980's the field has been expanded and some current real world examples of RL applications are:

### 2.2.1 Robot Navigation

Robots can be taught how to navigate their way around paces using reinforcement learning methods. Agents can be programmed to build an accurate model of their environment, find its way out of the maze or collect objects scattered throughout the search space. RL agents can easily be extended to map out a given real world area which the agent can operate in such as a room in a building. A popular example of this would be "Robby The Robot". Robby the Robot [9] is an RL agent that navigates around a physical space collecting rubbish. Robby can move move up, down, left or right and is rewarded for each piece of rubbish he collects but is penalised if he hits a wall or tries to pick up a piece of rubbish that isn't there. Robby the robot is often used as an example when introducing the field of RL.

### 2.2.2 Game Theory

Reinforcement Learning is also used in gaming scenarios where an agent is pited against another agent or sometimes against a human. RL is a popular technique for teaching an agent to play some games as it does not require large amounts of computation at each turn due to MDPs and because some games can have such a large number of possible states that it would be impossible for an agent to compute the correct action to take using other machine learning techniques. One of the most famous examples of RL applied to game theory would be TD Gammon; a backgammon program designed by Gerald Tesauro in 1992 [3] that used Temporal Difference Learning (a branch of RL) methods to teach an agent to play the board game backgammon to a level just below the top human backgammon players at the time. More recently, AlphaGo (program developed by Google Deepmind) defeated one of the worlds best Go (a Chinese strategy board game) players.

### 2.2.3 Multi-Agent Systems

A lot of the most exciting developments in the field of RL involve multi-agent systems. As the name suggests these are simply multiple agents working together to solve a problem. Eg: Multiple agents controlling traffic lights at intersections. Agents can pass information they have collected about their environment to other agents who can then use this new information to make improved decisions. Some of the most exciting developments in RL have been made in this area as agents have much more information about their environment made available to them by other agents.

Academics such as Professor Peter Stone [4] and Dr. Sam Devlin [5] among others have programmed robots to play soccer using Reinforcement Learning. Robocup, a tournament that pits these robots against each other was set up in 1997. It's goal is "by the middle of the 21st century, a team of fully autonomous humanoid robot soccer players shall win a soccer game, complying with the official rules of FIFA, against the winner of the most recent World Cup".

Some other applications of RL include:

- Air traffic control
- Elevator Dispatching
- Job-shop scheduling
- Power systems stability control

## 2.3 Evaluative Feedback Methods

As RL methods work based on feedback received from its environment I will now discuss some of the key evaluative feedback methods associated with the field of RL.

### 2.3.1 Reinforcement Learning vs Supervised Learning

Reinforcement Learning is a sub-field of Machine Learning. It uses trial and error methods to learn optimal solutions to problems. A lot of people misinterpret RL as a Supervised Learning algorithm however there are some key fundamental differences between the two that must be understood in order to avoid confusion. "A supervised learning system cannot be said to learn to control its environment because it follows, rather than influences, the instructive information it receives" [1]. A supervised learning algorithm "tries to make itself behave as instructed by its environment" [1]. In supervised learning after the agent makes a selection it is directly told what selection should have been made (Instructive feedback). This is in sharp contrast to RL which uses information that "evaluates the actions taken rather than instructs by giving correct actions" [1]. After a selection is made the agent receives a numerical reward for its choice; it is not told what the optimal action actually was. This is why there is a need for exploration in a RL environment. Exploration allows for increased chance of finding the optimal solution within the environment.

### 2.3.2 The N-armed Bandit Problem

The N-armed bandit problem is probably the most famous example that effectively illustrates the RL problem. Imagine you are playing a slot machine game. However instead of having to pull just one leaver you are given a choice from N number of leavers. Each leaver is assigned a numerical reward that you will receive if you pull that leaver. You do not know the value of the reward until after you pull the leaver although you may have estimates. Your job, as the learning agent, is to maximise the reward you receive from 1000 pulls of the leavers.

If you maintain a list of estimates of the reward functions then you have knowledge of the selection with the highest estimated reward. This is known as the "greedy" action. If you were to always select the greedy action then you are said to be "exploiting" your knowledge of the environment. If you were to choose one of the non-greedy actions you are said to be "exploring" the environment. Exploration will more than likely lead to discovery of an action with a higher reward function in the long run than the initial greedy action.

### 2.3.3 $\epsilon$-Greedy Methods and Softmax Action Selection

As discussed in the n-armed bandit section, if an agent always chooses the greedy action then they are said to be exploiting their knowledge of the system. In the long run this method is not beneficial as there is most likely a more optimal solution that the agent has not found. Thus the agent is encouraged to explore the system. However, depending on the nature of the task at hand, it may not be beneficial for the system to always explore and never exploit its knowledge of the system. This introduces the concept of $\epsilon$-greedy methods. In a $\epsilon$-greedy method the agent will exploit its knowledge of the system most of the time but some of the time it would explore new selections. Thus we say that some of the time, with a small probability $\epsilon$, the agent will make a choice at random. The value for $\epsilon$ is completely arbitrary and is set at the beginning of a RL task. Intuitively, the higher the value for $\epsilon$ then the more the agent will explore new selection choices.

Softmax Action Selection is an extension of the $\epsilon$-greedy method. While $\epsilon$-greedy methods are an effective way of exploring other choices in a RL system it is not a very efficient way. When the exploration choice is made it is a random choice; it chooses equally among all available choices. In systems where the worst choices are very bad this may not be the best way to explore the system. Softmax Action Selection provides a weighted estimate of the potential reward each choice will produce. The agent then chooses the next best option available to it based on these estimates.

## 2.4 Tackling the Reinforcement Learning Problem

There are three main types of methods currently used to solve the reinforcement learning problem; these are Dynamic Programming, Monte Carlo Methods and Temporal Difference Learning. Each method has its own strengths and weaknesses. However before discussion of these methods can begin, we must first define one of the most crucial properties when tackling RL problems: **A Markov Decision Process**

### 2.4.1 Markov Decison Process

A Markov decision process is a framework for modelling decision making in a mathematical problem. A problem can be defined as a Markov decision process if and only if it contains the Markov property. A stochastic process is said to have the Markov property if its next state depends only on its current state and not any other states preceding its current state.
Example: A game of chess can be considered a MDP because for any given state we can determine a new state. We do not need to know the previous moves/states that occurred before the current state because the current state provides all the information we need to move to a new state.

$$M = (S, A, T, \gamma, R) \tag{2.1}$$

As defined by Ng et all [2] the MDP is a tuple made up:

- S: The finite set of states
- A: The finite set of actions
- T: Transition probabilities when moving between states. ie: Probability of transfering into state s' from state s given action a.
- $\gamma$ The discount factor
- R: The reward function

Markov Decision Processes are fundamental for solving finite RL problems as RL algorithms do not have to maintain a record of past states encountered in order to move to a new state. A Markov Decision process is evaluated every time a learning agent moves to a new state.

### 2.4.2 On-Policy Learning VS Off-Policy Learning

It is imporatant to understand the difference between a learning agent using On-Policy Learning compared to an agent using Off-Policy learning. An Off-policy learner learns the value of the optimal policy independently of the agent's actions (eg:Q-Learning) whereas an On-policy learner learns the value of the policy being carried out by the agent including the exploration steps (eg:Sarsa).

### 2.4.3 Dynamic Programming

Dynamic Programming (DP) refers to algorithms that compute optimal policies for RL tasks. However these algorithms can only compute optimal policies when given a perfect model of the environment. The main idea of DP to break a large complex problem down into smaller sub-problems that can be solved individually (eg: each state change). DP allows for the computation of value functions that are used by the algorithm when changing states. The main limitation of DP in reinforcement learning tasks is that acquiring a perfect model of the environment can sometimes be either extremely computationally expensive or even impossible. Although impractical for large problems, DP methods are sometimes useful; they are guaranteed to find an optimal solution and have been shown to perform better than other methods such as Direct search or linear programming.

### 2.4.4 Monte Carlo Methods

Monte Carlo methods (MCM) are ways of solving the reinforcement learning problem based on averaging sample returns [1]. They essentially run a very large number of simulations and averages the results (the law of large numbers). MCM are mostly used in episodic tasks and it is at the end of each episode that any value estimates or policies are changed. Unlike Dynamic Programming, MCM do not require complete knowledge of the environment. "Monte Carlo methods require only experience—sample sequences of states, actions, and rewards from on-line or simulated

interaction with an environment" [1]. Although MCM need a model of the environment, the model does not need to provide the a complete probability distribution table for all transitions, only some sample transitions. MCM are most popular in situations where other methods are very difficult or impossible to implement.

### 2.4.5 Temporal Difference Learning

Temporal Difference (TD) learning is a combination of both dynamic programming ideas and Monte Carlo ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (bootstrapping) [1]. The TD learning method has advantages over both DP and MCM. First, unlike DP, TD doesn't require a full model of its environment. Second, unlike MCM, DP does not hace to wait until the end of an episode to update its value functions; it can update them after each time step or state change. TD learning methods have also been proven to converge to the optimal solution however it still remains unproven which set of methods TD or MCM converge to the correct solution the fastest.

## 2.5 Reward Shaping

Reward Shaping is a method for improving a learning agent's performance in a reinforcement learning environment. It is a learning process that has existed in the real world for centuries. It is in essence learning through a trial and error method. Humans and animals have learned over time to adapt to their surrounding environment by associating behaviours with certain situations they find themselves in. For example a dog may learn to fetch a ball in order to receive a reward of food. The dog did not develop this behaviour instantaneously. The dog was trained by first being rewarded for approaching the ball, second for picking the ball up and thirdly for releasing the ball at the feet of its owner. This training reinforces the good behaviour of fetching the ball and in future the dog will do so based on its previous experiences of the same situation. In the context of artificial intelligence, the field of reinforcement learning is built around this cause and effect relationship. A learning agent learns the optimal way to interact with its environment through trial and error. However, just as the dog was initially rewarded for just approaching the ball, the learning agent can be taught to incorporate knowledge of its environment in order to accelerate learning. This is done through the field of reward shaping.

Reward shaping works by providing the agent with heuristics, usually generated by the user (programmer), that the agent evaluates when choosing a new action to perform. The heuristics provide a **shaping reward function** $F$.

$$F(s, a, s') \tag{2.2}$$

The current state $s$, the action taken $a$ and the next state $s'$ are all used to compute the shaping reward function $F$. This shaping reward function is then

evaluated as part of the original MDP which now becomes:

$$M = (S, A, \gamma, R + F) \tag{2.3}$$

### 2.5.1 Reward Shaping Methods

#### Flag Heuristic Reward Shaping

This is a relatively straightforward reward shaping technique in which the user will place 'mini rewards' or reward flags throughout the environment to help guide the agent in the right direction. As the goal of the agent is to maximise its reward in the long run, these reward flags will encourage the agent to follow a certain path which will bring its policy closer to the optimum policy $\pi^*$. For example if the agent is in a 10x10 2D gridworld environment with starting coordinates of [0,0] and goal coordinates of [9,8] then the user could place a reward flag at [6,6] in order to encourage the agent to reach the reward flag's state more quickly rather than waste time exploring states close to [0,0]. It is important to note that, just as a human would struggle to complete a task if given incorrect information, a learning agent may struggle to find an optimal policy to solve a RL problem if the heuristic given to it is very poor or blatantly incorrect; In some cases the agent may never find a solution and will embark on infinite trajectories.

#### Potential Based Reward Shaping

Potential Based Reward Shaping is a technique developed by Ng et all [2] to combat the **Temporal Credit Assignment Problem**. The reward an agent receives navigating around states is usually very small or even negative. When navigating large state spaces, the reward for reaching the goal state normally takes so many steps that the reward is very weakly propagated back through the environments states. In essence, the influence of the reward is strongly diluted over the finite state space. If a reward shaping method like the flag based heuristic mentioned above and the heuristics supplied by the user are poor, ie: Reward Flag placement did not reflect the optimal policy, then the learning agent may never converge to a solution at all; it may travel in circles around the state space collecting the rewards from the reward flags. This problem was encountered by Randlov and Alstrom [6] when using Reinforcement Learning to train an agent to drive a bicycle towards a goal; the agent drove in circles near the start point.

Ng et al proposed the use of a potential function $\Phi$

$$F(s, a, s') = \gamma\Phi(s') - \Phi(s) \tag{2.4}$$

The shaping shaping reward function 2.2 is equal to the discounted potential of the new state $s'$ minus the potential of the old state. The potential of a state $\Phi$ is a function defined by the user that rewards the agent for desirable behaviour and punishes undesirable behaviour. It combats the temporal credit assignment problem by giving a small reward for an agent moving from state $s$ to state $s'$ where $s'$ is closer to the goal state. However if the agent moves from $s'$ back to $s$ then the agent is penalised by the same amount it had just been awarded ie: The difference between

the two potential functions. This prevents the formulation of a cycle of states within S that would yield a net positive reward if the agent were to travel through those states repeatedly. Ng et al [2] proved that the MDP using the potential function (2.3) converged to the optimal policy thereby validating the use of PBRS as a legitimate reward transformation for the Temporal Difference learning method.

# Chapter 3

# Testing Simulator

Reinforcement Learning Tasks are commonly evaluated in a gridworld environment as it allows for accurate measurement of agent performance. A gridworld environment is a simple 2 dimensional grid space. The environment allows for an agent to navigate its space by moving to adjacent squares, either all 8 adjacent squares or the 4 to the North, South, East or West. The learning agent begins at a starting state in the environment and, depending on the task at hand, navigates to desires goal states Gridworld environments are popular learning domains for RL solutions because the allow for a mapping of a number of different real world problems: Maze Navigation, Room to Room navigation (robots/drones), ordered navigation of state spaces.

## 3.1 Q-Learning Algorithm

The Q-Learning algorithm is an off policy temporal difference learning algorithm developed by Watkins in 1989 [10]. The method computes an action-value function for every state it encounter that gives the expected utility of choosing that action when in the given state. All state-action pairs are given an initial **Q-value**:

$$Q(s_t, a_t) \tag{3.1}$$

The algorithm updates these action values as it encounters them based on the reward it received from choosing action $a$ given that it was in state $s$. It then evaluates all action-value function to generate its policy.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma(max_a(s_{t+1}, a) - Q(s_t, a_t))) \tag{3.2}$$

The Q-value of the state $s$ and action $a$ at time step $t$ is updated to equal itself plus the discounted value of the reward received for moving into the new state plus the discounted value of the difference between the maximum Q-value in state $s_t + 1$ and current Q-Value. Here we can see what it means when we say that the Q-Learning algorithm is an off-policy learner; it is checking the maximum Q-value in the next state for all possible actions without the agent having to choose any of these Q-values as its new Q-value. In comparison, another popular temporal difference learning algorithm is the Sarsa algorithm:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma((s_{t+1}, a_{t+1}) - Q(s_t, a_t))) \tag{3.3}$$

Sarsa is an off policy algorithm and we can see this in equation 3.3 as the algorithm does not evaluate all Q-Values in the next state, only $Q(s_{t+1}, a_{t+1})$. Note here that action $a_{t+1}$ is determined by the policy eg: $\epsilon$-greedy, Softmax action selection, etc.

## 3.2 Simulator used in this experiment

The simulator used in this experiment, developed using Java, places the agent in a 50x50 gridworld maze where it is expected to navigate to a goal state at the opposite side of the maze. The agent will be permitted to perform 4 actions at any given time: North, South, East or West. The agent's starting position will be at the 0,0 coordinate in the upper left corner of the maze and the exit point will be at the 49,49 coordinate in the bottom right corner of the maze. If the agent hits the boundary of the maze when performing an action then the action will be evaluated by the RL algorithm but the agent will remain in the state it was in before the action was selected ie: it will be penalised for making an invalid move.

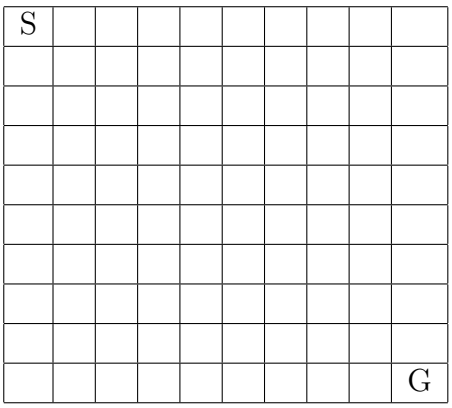


Figure 3.1: An example 10x10 gridworld environment. Agent begins at start state S and navigates to goal state G

When the agent reaches the goal state it is said to have completed 1 iteration/episode of the maze. The number of moves the agent made and the total reward accumulated by the agent are saved for later evaluation. The number of iterations the agent performs is different for each parameter examined in the testing. Figure 3.1 illustrates a 10x10 example gridworld environment that the agent has to navigate. The maze used in the simulator has no reward flags or blocked states anywhere in its state space.

Some of the implementation for the Q-Learning logic in the simulator was taken from an open source online resource [12]. I built on top of this code to make the simulator more extendable and added all logic to implement PBRS.

### 3.2.1 How the simulator works

As mentioned above the simulator is developed using the Java programming language. It uses a 3 dimensional array to store the Q-Values for all state action pairs in the environment; all of which are initially set to zero. When the simulator is run the agent passes through the environment updating its Q-Values as it goes. The agent's starting position is 0,0 and the absorbing state is located at the bottom right corner of the maze (49,49). Once the agent reaches the exit point of the maze a new iteration begins with the Q-values updated from the previous iteration. At the end of each iteration, results from the agents performance are written to a local text file for later analysis. This text file is read in by an R script which plots the graphs used in the results section of this report.

The simulator does this for implementations of the Q-Learning Algorithm on its own and Q-Learning with potential based reward shaping (PBRS). It runs this processes sequentially, reinitialising the Q-Values table and the testing parameters between the two processes. For the PBRS agent, the potential function uses the Manhattan distance of the old state $s$ and the new state $s'$.

### 3.2.2 Agent.java

This class contains all the logic for the RL agent object. The agent is initialized with a maze and a policy. Values for all of the Q-Learning parameters are set here too. The agent has a method for both the Q-Learning algorithm with and without PBRS: runWithPotential() and runWithoutPotential(). Figures 3.2 and 3.3 show the logic both for calculating the value of the potential function and updating the Q-Values with the Q-Learning formula.

```
124             oldManhattenDist = thisWorld.calcManhatten(state);
125             newManhattenDist = thisWorld.calcManhatten(newstate);
126
127             if(newManhattenDist < oldManhattenDist){
128                 //Agent moved toward goal
129                 newPotentialFunc += 1;
130             }
131             else {
132                 //Agent moved away from goal
133                 newPotentialFunc -= 1;
134             }
```

Figure 3.2: Code Extract that calculates $\Phi(s')$ and $\Phi(s)$ for use in the potential function

```
140             // Q LEARNING FORMULA HERE
141             if(newManhattenDist>= 0){
142                 ShapingReward = gamma*newPotentialFunc - potentialFunc;
143                 new_Q = this_Q + alpha * ( reward + tau*ShapingReward + gamma * max_Q - this_Q);
144 
145             }
146             //need else for when s' = s0 (goal state)
147             else{
148                 new_Q = this_Q + alpha * ( reward + gamma * max_Q - this_Q);
149             }
```

Figure 3.3: The Q-Learning Formula with Reward Shaping that updates the Q Value of the previous state in runWithPotential()

```
221     private int selectAction( int[] state ) {
222         double[] qValues = policy.getQValuesAt( state );
223         int selectedAction = -1;
224 
225         //E_GREEDY
226 
227         random = false;
228         double maxQ = -Double.MAX_VALUE;    //initially set max = lowest possible double value
229         int[] doubleValues = new int[qValues.length];
230         int maxDV = 0;
231 
232         //Explore
233         if ( Math.random() < epsilon ) {
234             selectedAction = -1;
235             random = true;
236         }
237         else {
238             for( int action = 0 ; action < qValues.length ; action++ ) {
239                 if( qValues[action] > maxQ ) {
240                     selectedAction = action;
241                     maxQ = qValues[action];
242                     maxDV = 0;
243                     doubleValues[maxDV] = selectedAction;
244                 }
245                 //more than one Q values are the same
246                 else if( qValues[action] == maxQ ) {
247                     maxDV++;
248                     doubleValues[maxDV] = action;
249                 }
250             }
251             //Choose randomly out of the equal Q values
252             if( maxDV > 0 ) {
253                 int randomIndex = (int) ( Math.random() * ( maxDV + 1 ) );
254                 selectedAction = doubleValues[randomIndex];
255             }
256         }
257 
258         if ( selectedAction == -1 ) {
259             selectedAction = (int) (Math.random() * qValues.length);
260         }
261         return selectedAction;
262     }
```

Figure 3.4: Code that chooses the action be performed by the agent

Lines 225 → 235 in figure 3.4 show the logic for implementing the $\epsilon$-greedy policy. Lines 238 → 255 show how the agent chooses the action with the highest Q-Value in the current state, cycling through all values until it finds the maximum.

### 3.2.3 Policy.java

This class contains all logic to do with the policy of the agent. It initializes a 3-dimensional array with all Q-Values for the environment and performs all operations where a Q-Value or set of Q-Values need to be retrieved or updated. Figure 3.5 shows a code extract from the policy class that returns all the Q-Values for a given state.

```
84      public double[] getQValuesAt( int[] state ) {
85          int i;
86          Object curTable = qValuesTable;
87          double[] returnValues;
88
89          for( i = 0 ; i < dimSize.length - 2 ; i++ ) {
90              //descend in each dimension
91              curTable = Array.get( curTable, state[i] );
92          }
93
94          //at last dimension of Array get QValues.
95          qValues = (double[]) Array.get( curTable, state[i] );
96          returnValues = new double[ qValues.length ];
97          System.arraycopy( qValues, 0, returnValues, 0, qValues.length );
98          return returnValues;
99      }
```

Figure 3.5: Code extract that returns the Q-Values of the passed state

### 3.2.4 Maze.java

This class contains all logic for the maze that the RL agent will be navigating in. It contains information such as maze dimensions, actions that can be performed and methods for calculating agent reward and whether an action the agent wishes to execute is permitted in the given state. Figures 3.6 and 3.7 show code extracts from Maze.java that show some of its operations.

```
97      public double getReward( int[] newstate) {
98
99          if(newstate[0] == mazeExit[0] && newstate[1] == mazeExit[1]){
100             return 100;
101         }
102         else if(rewardArray.size()>0){
103             //check if new state contains any reward
104             for(Reward r:rewardArray){
105                 if(newstate[0] == r.getxPos() && newstate[1] == r.getyPos()){
106                     break;
107                 }
108                 return r.getRewardVal();
109             }
110             return -1;
111         }
112         else return -1;
113
114     }
```

Figure 3.6: Code extract that returns the reward to be returned to the agent. Lines 102 → 108 contain code that can be handle situations where there are reward flags throughout the environment (Flag Heuristic Reward Shaping)

```
42      //Calculates Manhatten Distance from Goal State to agents current State
43      public int calcManhatten(int[] curState){
44          manDist = Math.abs(mazeExit[0] - curState[0]) + Math.abs(mazeExit[1] - curState[1]);
45          return manDist;
46      }
```

Figure 3.7: Code extract that calculates the Manhattan distance of the passed state to the maze exit

### 3.2.5 Simulator Structure

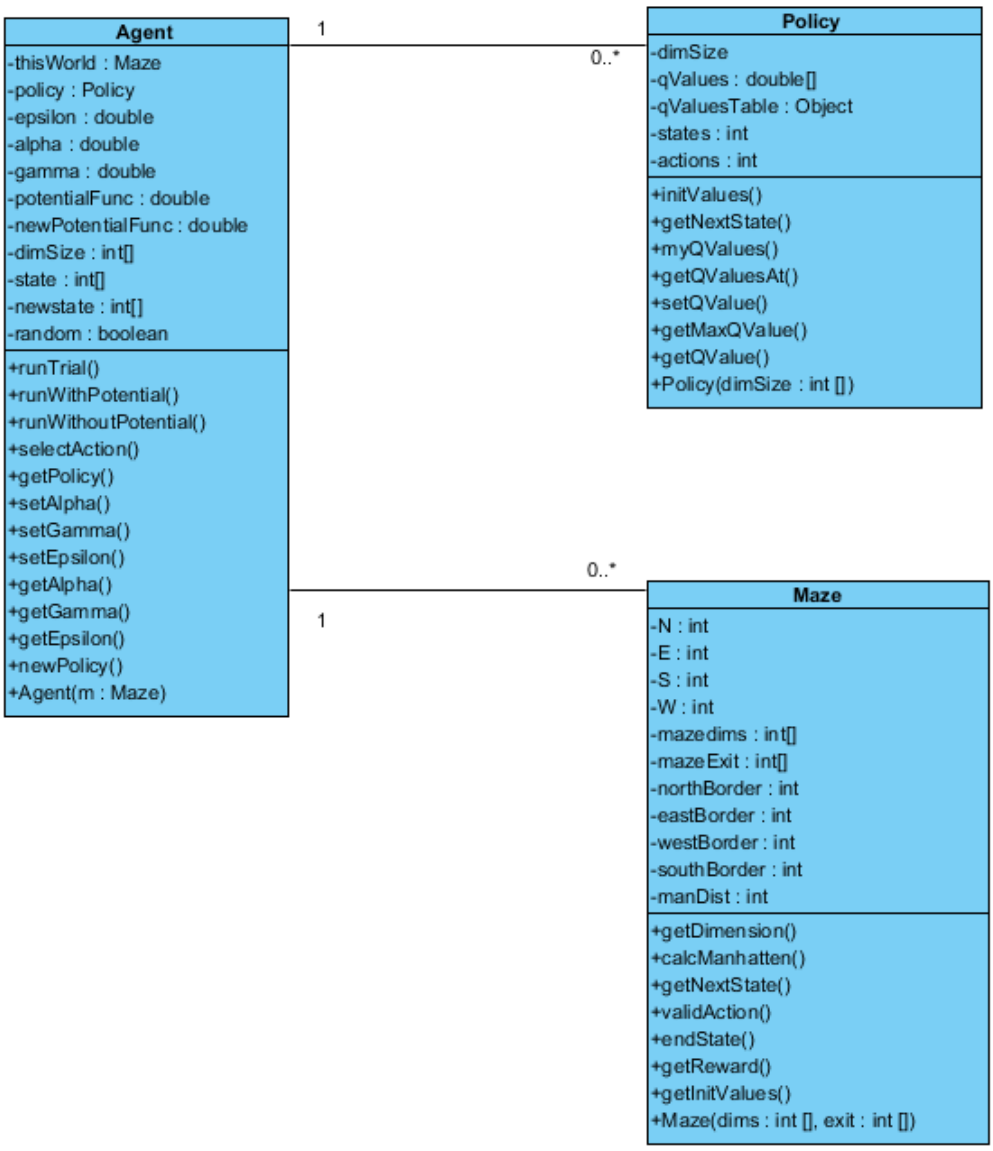


Figure 3.8: A class diagram showing the relationships between the three classes used in the simulator

# Chapter 4

# Exploration Rate: $\epsilon$

## 4.1 Introduction to the Parameter

The exploration rate is the variable that controls how often the learning agent explores new actions rather than exploiting its current knowledge of the environment. The simulator generates a random number between 0-1 and if the number generated is less than the exploration rate's value then the agent will choose its next action randomly.

## 4.2 Experiment Setup

The exploration rate $\epsilon$ is one of the most important parameters in the simulator as it will control how much exploration is done by the agent. If the environment the agent is operating in is large then we want the agent to explore as much as possible as this increases the probability of finding the optimal policy $\pi^*$. Therefore for this chapter we will examine how the agent performs as $\epsilon$ is varied.

### 4.2.1 Experimental Parameters

Table 4.1 shows the initial values of all other parameters before altering of the exploration rate.

| | |
|---|---|
| Alpha | 0.9 |
| Gamma | 1 |
| Tau | 1 |
| Step Reward | -1 |
| Goal reward | 100 |

Table 4.1: Initial Parameter Values before testing of the exploration rate

For Q-Learning agents with and without shaping, it is desirable to have the agent explore its environment initially to increase the chances of it finding the optimal policy. For these tests I will provide two agents with a significantly large initial value for epsilon and decrease them (at different rates) throughout the training

phase. The purpose of this is to discover if the increased exploration of the agent at the beginning of testing will result in the agent finding the optimal policy faster than an agent with a low fixed exploration rate. It is important to note that the exploration rate should not be allowed to reach zero as if it does so the agent will no longer learn any new information about the environment and will only reinforce with most optimal path it has found from that point onwards.

The graphs in the following results section have been passed through a **Savitzky-Golay filter** for smoothing purposes. It works by fitting a low degree polynomial by method of linear least squares. Unless otherwise specified, results are smoothed with a 5th degree polynomial using a subset of 15 adjacent data points to fit a new data point. The use of the filter allows for legible interpretation of the results output by the simulator.

For the Q-Learning agents with PBRS the following three varying exploration rates were used:

- $\epsilon$ is decreased linearly by 0.05 every 25 iterations and remains fixed once it reaches 0.05
- $\epsilon$ is halved after every 25 iterations
- $\epsilon$ is fixed at 0.1 for all 500 iterations

The conversion times for agents that do not use PBRS are much larger than those that do so for the Q-Learning algorithm without PBRS the number of iterations that will be examined is 1500. This allows for at least some of the algorithms to converge to the optimal solution. $\epsilon$ was altered after every 50 iterations. Due to the much larger number of data points this increase in iterations brings, a Savitzky-Golay filter using a 5th polynomial and a subset of 25 data points is utilized for smoothing purposes. For the Q-Learning agents without PBRS the following three varying exploration rates were used:

- $\epsilon$ is decreased linearly by 0.05 every 25 iterations and remains fixed once it reaches 0.05
- $\epsilon$ is halved after every 50 iterations
- $\epsilon$ is fixed at 0.1 for all 1500 iterations

For the purposes of simplicity these will be distinguished as 3 separate agents; Agent 1, Agent 2 and Agent 3 in each results chapter.

## 4.3 Results

### 4.3.1 PBRS Agents

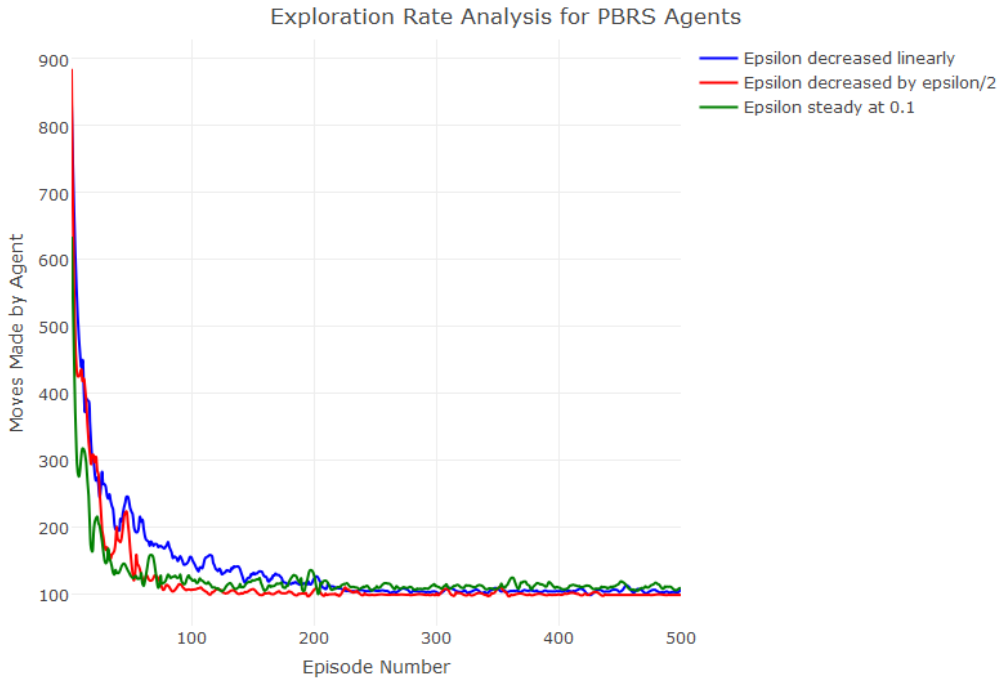


Figure 4.1: Steps taken by the PBRS agent per iteration over 500 iterations

Figure 4.1 shows that agent 3 has much fewer steps per episode initially compared to Agent 1 and 2. This is expected as the much higher initial value for exploration results in Agents 1 and 2 exploring five times as much as agent 3. However we can also see that agent 2 learns rapidly and actually outperforms agent 3 after approximately 75 episodes. This can be explained by the fact that at this point (75 iterations), the agents learning rate has been decreased to 0.125. From this point onwards Agent 2 outperforms the other two agents as it converges to the optimal policy after approximately 140 iterations. Agent 3 learns quickly but cannot converge to the optimal policy because of its fixed exploration rate. In comparison, agent 2 converges to the optimal policy after 140 iterations as the exploration rate becomes negligibly small. Although Agent 1 eventually outperformed Agent 3, its learning rate was to slow and it could never reach the optimal policy due to its fixed exploration rate.

From the results shown by figure 4.1 we cannot conclude that an initially large exploration rate leads to faster agent learning. Although Agent 2 outperformed Agent 3 after approximately 75 iterations, their learning rates at this point were very similar. The only reason Agent 2 outperformed Agent 3 for the remainder of the experiment was because its learning rate was lowered to a negligibly small value. In another situation where an agent had not yet found the optimal policy it could be observed that Agent 2's extremely low exploration rate would actually hinder its performance.

### 4.3.2 Agents without PBRS

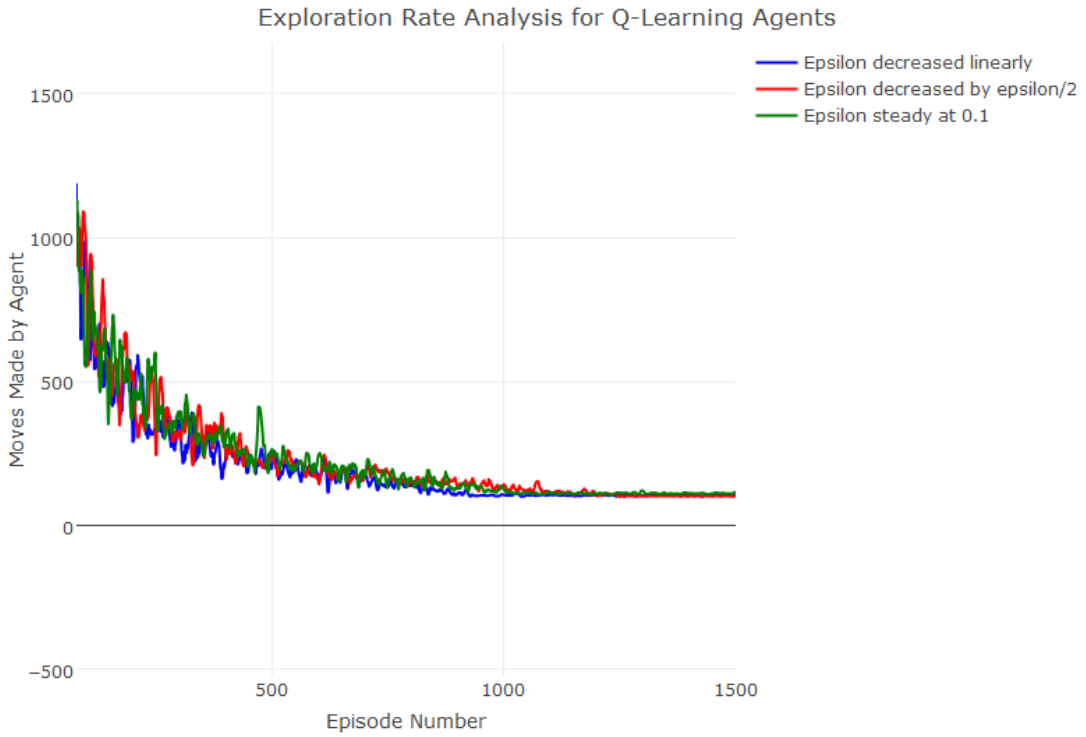


Figure 4.2: Scaled view of steps taken by the agent per episode over 1500 iterations

Figure 4.2 does not provide a clear view of which agent performed best however results showed that none of the three agents converged to the optimal policy. The closest of the three was agent 2 which converged after 1241 iterations with a step count of 102 (optimal = 98). It did not explore any other paths after this point. As outlined in the experimental parameters section, the reduction of the learning rate to effectively 0 is a very bad practice as the learning agent may never learn optimal behaviour. In some test runs (not included in this report) Agent 2 did actually converge to the optimal policy but in others it did not and as a result never improved.

Agent 1 and 3's performance was constant after approximately 1120 and 1200 respectively with their differing exploration rates resulting in slightly better results for Agent 1. As results are not overwhelmingly clear I decided to perform a paired t test for all three pairs of agents.

#### Paired T Test

If the p value returned by the test is less than 0.05 then we can say that there is a significant statistical difference between the two agents scores. These three paired t tests will use each agents steps per iteration over 150 iterations to compare their performance.

| | Agent 1 | Agent 2 |
|---|---|---|
| Mean | 324.19 | 339.06 |
| St. Dev | 852.34 | 904.88 |

Table 4.2: Paired t test for Agent 1 VS Agent 2 over 1500 iterations

The two tailed P value for these pair of agents = 0.7018. Therefore it can be concluded that there is no statistically significant difference between these pair of agents.

| | Agent 1 | Agent 3 |
|---|---|---|
| Mean | 324.19 | 315.52 |
| St. Dev | 852.34 | 668.86 |

Table 4.3: Paired t test for Agent 1 VS Agent 3 over 1500 iterations

The two tailed P value for these pair of agents = 0.5010. Therefore it can be concluded that there is no statistically significant difference between these pair of agents.

| | Agent 2 | Agent 3 |
|---|---|---|
| Mean | 339.06 | 315.52 |
| St. Dev | 904.88 | 668.86 |

Table 4.4: Paired t test for Agent 2 VS Agent 3 over 1500 iterations

The two tailed P value for these pair of agents = 0.1202. Therefore it can be concluded that there is no statistically significant difference between these pair of agents.

It must be stressed that the exploration rate should be explicitly tailored to the learning task at hand. For example the purpose of this task is to achieve the optimal policy as fast as possible and in order to achieve maximum agent performance, a low exploration rate exploration rate must be used during training and then lowered to zero once the agent has achieved the optimal policy. However in a different learning task this approach may be undesirable. In cases where the number of optimal solutions is greater than one we may want the agent to continue exploring to reinforce the Q values of *all* optimal paths. With respect to this learning task, the problem of the agent being unaware of whether or not it has found the optimal path is an one which is beyond the scope of this project; one could write a research paper on this topic alone.

# Chapter 5

# Learning Rate: $\alpha$

## 5.1 Introduction to the Parameter

The main purpose of a learning agent in an RL environment is to maximise the reward it receives in the long run. In the Q-Learning formula, the learning rate $\alpha$ is used as a means of discounting the value of future information received by the learning agent.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma(max_a(s_{t+1}, a) - Q(s_t, a_t))) \tag{5.1}$$

$\alpha$ essentially controls show quickly the newly required information (reward for moving into the new state) will override the old information. Equation 6.1 shows that a large value for alpha, such as 0.8 up to 1.0, assigns future rewards a large value close to 100% of their total value. Thus our estimates of the value of future states will be high. A low learning rate will place higher importance on the value of immediate reward received by the agent.

## 5.2 Experiment Setup

In this chapter I will analyze the performance of the Q-Learning algorithm when it is assigned different values for $\alpha$. I will build on the research of Even-Dar and Mansour [7] who examined the relationship between the convergence rate and learning rate of the Q-Learning algorithm.

Again a Savitzky-Golay filter is used to smooth the data. It uses a 5th degree polynomial and a subset of 15 data points.

### 5.2.1 Experimental Parameters

Table 5.1 show the initial values for all other parameters before altering of the learning rate.

| | |
|---|---|
| Epsilon | 0.1 |
| Gamma | 1 |
| Tau | 1 |
| Step Reward | -1 |
| Goal reward | 100 |

Table 5.1: Initial Parameter Values before testing of the learning rate

Even-Dar and Mansour [7] have shown that a high learning rate (0.8 - 1.0) produced the best precision for the Q-Learning formula for both a linear and polynomial reduction of $\alpha$ over time.

PBRS and Q-Learning agents will be tested over 150 and 1500 iterations respectively with the following fixed learning rates:

- $\alpha = 1$
- $\alpha = 0.85$
- $\alpha = 0.7$

## 5.3 Results

### 5.3.1 PBRS Agents

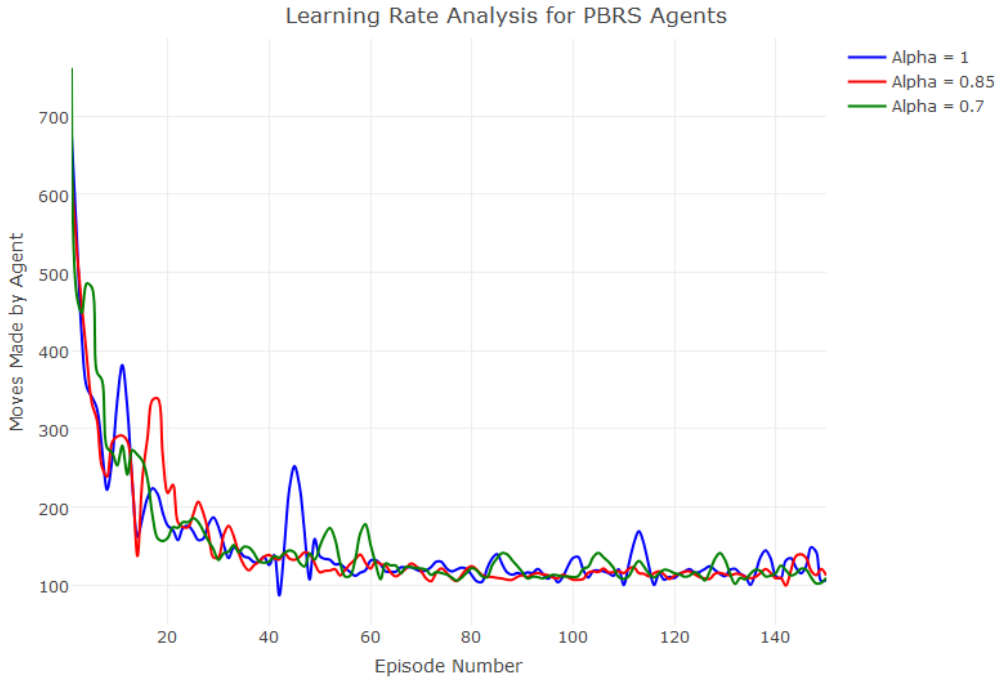


Figure 5.1: Total steps taken by the PBRS agent per iteration over 150 iterations.

The results from Figure 5.1 do not overwhelmingly prove one way or the other which of the three agents performs the best.

#### Paired T Testing

These three paired t tests will use each agents steps per iteration over 150 iterations to compare their performance.

| | Agent 1 | Agent 2 |
|---|---|---|
| Mean | 152.25 | 149.65 |
| St. Dev | 83.32 | 85.78 |

Table 5.2: Paired t test for Agent 1 VS Agent 2 over 150 iterations

The p value for these pair of agents is 0.5131. This shows that there is no significant statistical difference between Agents 1 and 2.

| | Agent 1 | Agent 3 |
|---|---|---|
| Mean | 152.25 | 152.72 |
| St. Dev | 83.35 | 92.10 |

Table 5.3: Paired t test for Agent 1 VS Agent 3 over 150 iterations

The p value for these pair of agents is 0.9149. This shows that there is no significant statistical difference between Agents 1 and 3.

| | Agent 2 | Agent 3 |
|---|---|---|
| Mean | 149.65 | 152.72 |
| St. Dev | 85.78 | 92.10 |

Table 5.4: Paired t test for Agent 2 VS Agent 3 over 150 iterations

The p value for these pair of agents is 0.5587. This shows that there is no significant statistical difference between Agents 1 and 3. Testing of all three agents shows that there is little to separate the performance of the three agents. Agent 1 had the lowest standard deviation while Agent 2 had the lowest mean. However these findings on their own are not enough to prove that one method is better than the other. I was therefore unable to add to the findings of Even-Dar and Mansour [7] to determine what is the optimal learning rate for the Q-Learning agent. However it must be noted that Even-Dar and Mansour did not examine the Q-Learning algorithm that encorporating reward shaping.

### 5.3.2 Agents without PBRS

Due to the increased number of datapoints to be plotted a Savitsky Golay filter using 5th order polynomial and a subset of 25 data points were used to smooth the data.

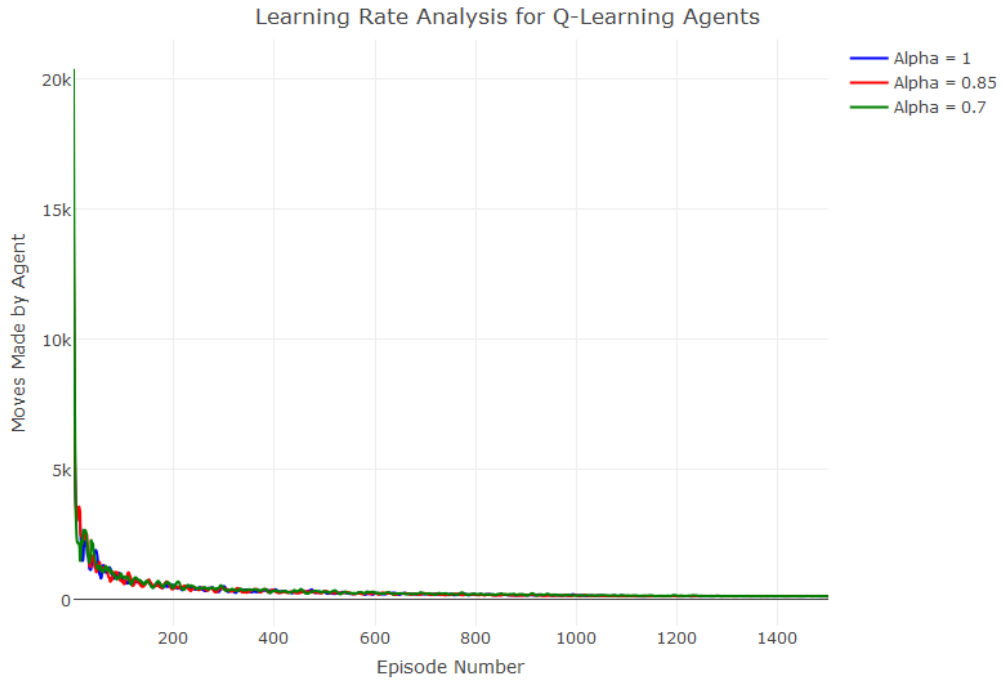


Figure 5.2: Total steps taken by the agent per iteration over 1500 iterations

Although it is impossible to compare agent performance from this graph, the text file of results showed that all three agents converged to the optimal policy (as allowed by the exploration rate of 0.1) within the 1500 iterations. Results also showed that Agent 2 converged first after approximately 1020 iterations with Agent 1 following closely, converging after 1100 iterations. Agent 3's lower learning rate resulted in convergence taking approximately 1400 iterations.

Thus I can confirm Even-Dar and Mansour's conclusion that a high learning rate of 0.8 - 1 is an better range to use for maximum agent performance with a slightly discounted value of 0.85 providing marginally better precision.

# Chapter 6

# Discount Factor: $\gamma$

## 6.1 Introduction to the Parameter

In the Q-learning algorithm the discount factor $\gamma$ is used to discount to the value of the Q-values of future states. The algorithm fetches the maximum Q-value in the new state it enters. This Q-Value is then evaluated by the formula to update the current Q-value: $Q(s, a)$.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma(max_a(s_{t+1}, a) - Q(s_t, a_t))) \tag{6.1}$$

The algorithm calculates the difference between the maximum Q-Value in $s_t + 1$ and the current Q-Value. The higher the score of the max Q-value in the state $s'$ the higher the value assigned to the transition $Q(s_t, a_t)$. The discount factor $\gamma$ determines **how much of an effect the new Q-value has** when updating the current Q value defined by equation 6.1 (It discounts it).

The discount factor is also used as part of Potential Based Reward Shaping. It is used to discount the value of the potential functions used to provide shaping rewards. the reward function F, evaluated as part of the updated MDP, is given by:

$$F(s, a, s') = \gamma\Phi(s') - \Phi(s) \tag{6.2}$$

## 6.2 Experiment Setup

In this chapter I will analyze the performance of the Q-Learning algorithm with three different values for discount factor. A Savitsky Golay Filter is used for smoothing purposes for all graphs in this chapter. Unless otherwise specified they implement a 5th order polynomial using a subset of 9 data points to smooth the data.

### 6.2.1 Experimental Parameters

Table 7.1 shows the initial values of all other parameters before altering of the discount factor.

| | |
|---|---|
| Alpha | 0.9 |
| Epsilon | 0.1 |
| Tau | 1 |
| Step Reward | -1 |
| Goal reward | 100 |

Table 6.1: Initial Parameter Values before testing of the discount factor

## 6.3 Results

### 6.3.1 PBRS Agents

Upon testing of the Q-Learning algorithm with PBRS implemented and with values for $\gamma$ that were $< 1$, the simulator was taking along time to finish as agents were taking extremely long trajectory paths to complete the maze and some cases it encountered infinite trajectories. Grześ and Kudenko [11] show that this happens because of the way the potential function is set up. They define the 3 equations that underpin the use of a potential function:

$$F(s, s') = \gamma\Phi(s') - \Phi(s) >= 0, \tag{6.3}$$

$$F(s', s) = \gamma\Phi(s) - \Phi(s') <= 0, \tag{6.4}$$

$$F(s, s) = \gamma\Phi(s) - \Phi(s) <= 0, \tag{6.5}$$

Grześ and Kudenko showed that in the case **where $n$ represents the potential function of a state** $s$, the potential for state $s'$ is n+1. thus from equations 6.3, 6.4 and 6.5 they obtained the following:

$$\gamma >= \frac{n}{n+1}, n <= \frac{\gamma}{1-\gamma}, \tag{6.6}$$

$$\gamma <= \frac{n+1}{n}, n >= \frac{1}{\gamma - 1}, \tag{6.7}$$

$$\gamma <= 1, n >= 0. \tag{6.8}$$

Thus we can conclude that $n$ increases when moving towards the goal. Therefore, for the maze size used in this experiment the maximum value of n = 98. By equation 6.6, this implies that $\gamma >= 0.\overline{98}$. Grzes and Kudenko stated:

*"If these conditions are violated, the negative shaping reward will be given for those transitions (s to s′) which should be positively reward according to the potential function Φ".*

Due to the constraints placed on $\gamma$ by equation 6.6 it is unfeasible to measure the performance of PBRS Agents with significantly different discount factors. Thus

this results section will compare the performance of only two Agents with discount factors of 0.99 and 1 respectively.

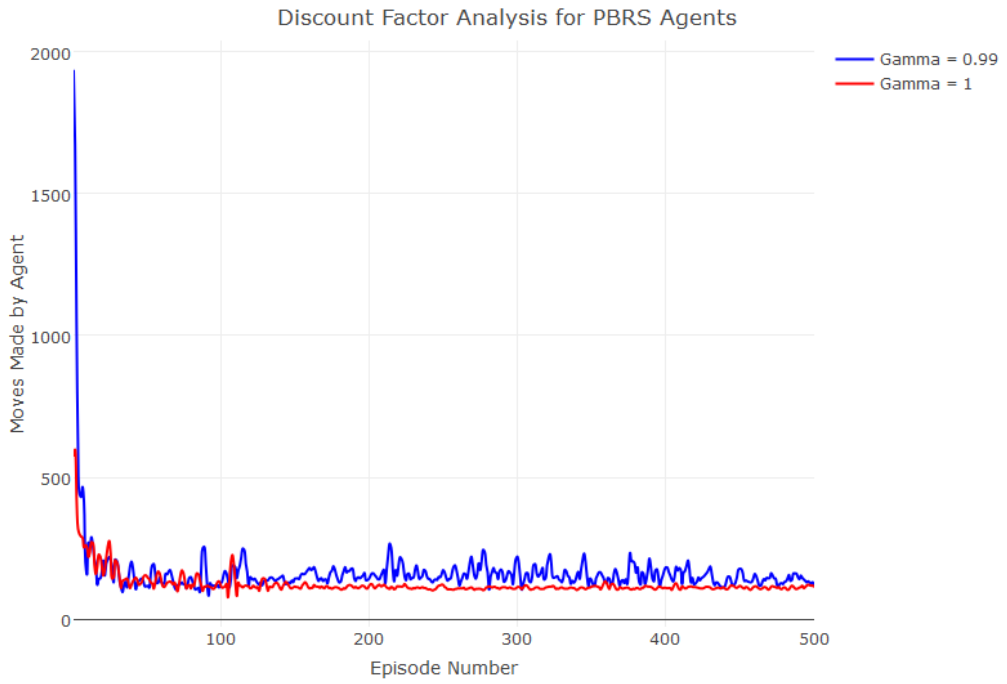


Figure 6.1: Total steps taken by the PBRS agent per iteration over 500 iterations.

Figure 6.1 shows that although the Agents' initial learning is very similar, Agent 1 does not converge to the optimum solution within the 500 iterations whereas Agent 2 converges after approximately 120 iterations. The deviation of scores for Agent 1 is also much larger than those of Agent 2. This shows that even a 0.01 of a discount of future rewards has a significant decrease in agent performance. In this particular case the reduction of $\gamma$ discounts both the value of future states Q-Values and also the value of the shaping reward received.

### 6.3.2 Agents without PBRS

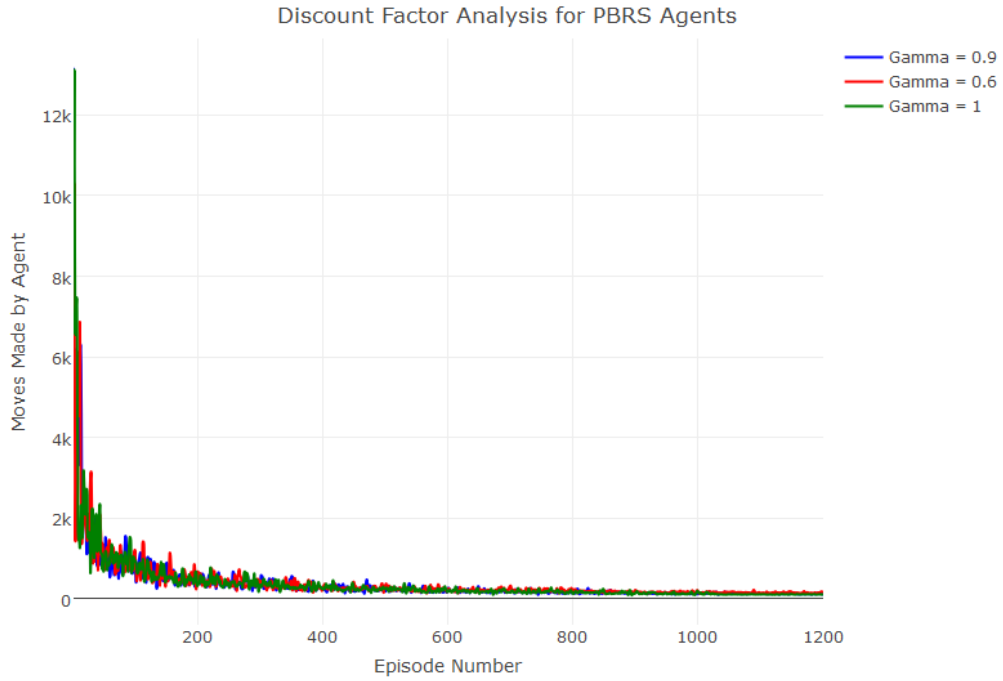


Figure 6.2: Total steps taken by the agent per iteration over 1200 iterations.

From figure 6.2 it isn't clear which agent performed the best. Results of the simulator output show that Agent 2 had much shorter initial episode lengths however it did not converge to the optimal solution within the 1200 iterations. Agent 1 converged to the optiaml solution after approximately 1180 iterations but Agent 3 performed the best, converging after approximately 1030 iterations. This shows that an undiscounted value of 1 for $\gamma$ produces maximum agent performance. A paired t test was also performed to compare the performances of the three agents.

#### Paired T Test

| | Agent 1 | Agent 2 |
|---|---|---|
| Mean | 371.07 | 395.39 |
| St. Dev | 752.13 | 734.51 |

Table 6.2: Paired t test for Agent 1 VS Agent 3 over 1200 iterations

The p value for these pair of agents is 0.1174. This shows that there is no significant statistical difference between Agents 1 and 2.

| | Agent 1 | Agent 3 |
|---|---|---|
| Mean | 371.07 | 376.07 |
| St. Dev | 752.13 | 796.63 |

Table 6.3: Paired t test for Agent 1 VS Agent 3 over 1200 iterations

The p value for these pair of agents is 0.7707. This shows that there is no significant statistical difference between Agents 1 and 3.

| | Agent 2 | Agent 3 |
|---|---|---|
| Mean | 395.39 | 376.07 |
| St. Dev | 734.51 | 796.63 |

Table 6.4: Paired t test for Agent 2 VS Agent 3 over 1200 iterations

The p value for these pair of agents is 0.1299. This shows that there is no significant statistical difference between Agents 2 and 3. All three paired t tests showed that there was no statistical difference between the agents. However the large initial episodes of Agent 3 may be the reason for this as results files showed that Agent 3 significantly outperformed the other two agents.

Results for Q-Learning agents with and without PBRS implemented show that an undiscouted value of 1 for $\gamma$ produced maximum agent performance.

# Chapter 7

# Scaling Factor: $\tau$

## 7.1 Introduction to the Parameter

The purpose of the scaling factor in the Q-Learning algorithm that uses PBRS is simply to scale the value of the shaping reward function F.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \tau(\gamma\Phi(s') - \Phi(s)) + \gamma(max_a(s_{t+1}, a) - Q(s_t, a_t))) \quad (7.1)$$

A large value for $\tau$ will give the shaping reward a higher value and vice versa.

## 7.2 Experiment Setup

In their research [11], Grześ and Kudenko showed that a scaling factor of 2 produced the best performance for a PBRS agent. They noted that

*"higher values of the shaping reward (scaled up with $\tau > 1$) had positive influence on exploration...for very large value for $\tau$ ($> 50$), The shaping reward overshadows the reward received from the environment".*

### 7.2.1 Experimental Parameters

Table 7.1 shows the initial values of all other parameters before altering of the scaling factor.

| | |
|---|---|
| Alpha | 0.9 |
| Epsilon | 0.1 |
| Gamma | 1 |
| Step Reward | -1 |
| Goal reward | 100 |

Table 7.1: Initial Parameter Values before testing of the scaling factor

In order to build on the results obtained by Grześ and Kudenko this experiment will examine the performance of the PBRS agent with these three scaling factors:

- $\tau = 2$
- $\tau = 1$
- $\tau = 5$

A Savitzky-Golay filter is used to smooth the data. It uses a 5th degree polynomial and a subset of 9 data points.

## 7.3 Results

### 7.3.1 PBRS Agents

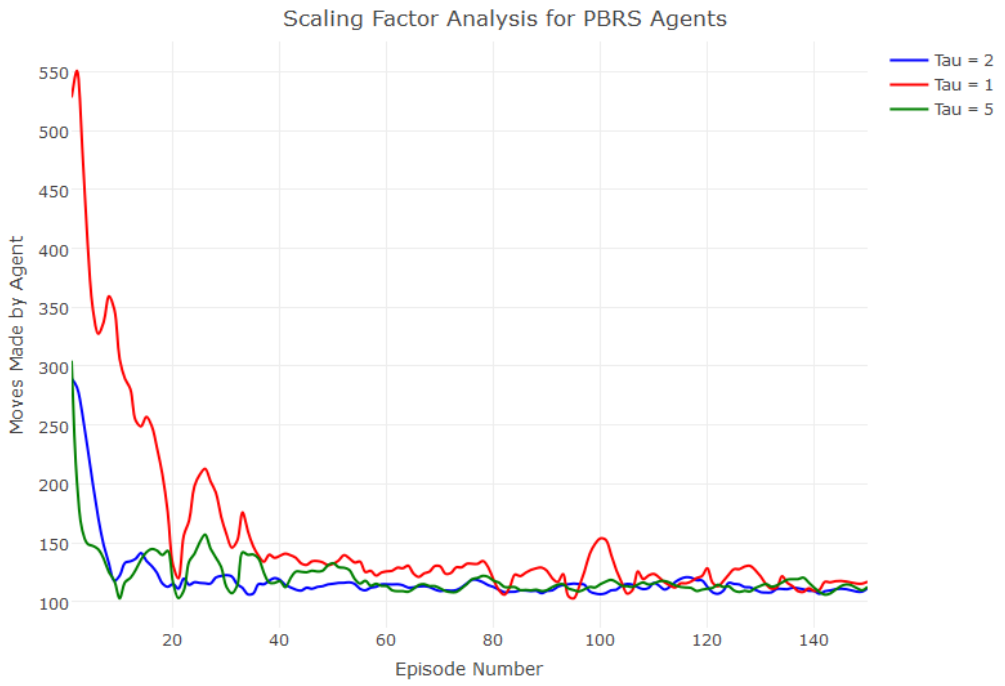


Figure 7.1: Total steps taken by the PBRS agent per iteration over 150 iterations

It is clear from figure 7.1 that Agent 2 is outperformed by the other two agents. The scaled value for $\tau$ has a positive effect on the Agent 1 and 2's performance with both agents approaching the optimal policy approximately 100 episodes before Agent 2. Figure 7.1 also shows that the much higher scaling factor of agent 3 compared to that of agent 1 did not necessarily lead to better agent performance. Agent 3's initial training episodes (0-17) were better than agent 1's. However after this point Agent 3's large scaling factor lead to significant fluctuations in episode lengths, which in turn lead to decreased cumulative reward. After approximately 60 iterations these fluctuation ceased and both agents performance was very similar for the remaining iterations. Thus it can be concluded that a scaling factor of $\tau = 2$ does indeeed produce optimal agent performance as stated by Grześ and Kudenko.

# Chapter 8

# Step Reward

## 8.1 Introduction to the Parameter

The step reward is the reward the agent receives after each move it makes in the gridworld maze. The step reward is the only parameter examined in this report that is outside of the agent's control. The Q-Learning algorithm cannot alter its value as it is provided to the agent by the environment. However, oftentimes the environment to be navigated by the agent is designed with the agent in mind or its dynamics can be changed by the user/programmer. Therefore we can alter the step reward the environment provides to the agent to see if different step rewards produce better performance.

## 8.2 Experiment Setup

Due to the fact that step reward is controlled by the agents environment and in order not to violate the Markov property, the step reward must be kept constant throughout the agents training.

### 8.2.1 Experimental Parameters

Table 7.1 shows the initial values of all other parameters before altering of the scaling factor.

| | |
|---|---|
| Alpha | 0.9 |
| Epsilon | 0.1 |
| Gamma | 1 |
| Tau | 1 |
| Goal reward | 100 |

Table 8.1: Initial Parameter Values before testing of the step reward

This experiment examines the performance of the Q-Learning agent with and without PBRS.The PBRS agents will be tested over 200 iterations while Q-Learning agents without PBRS will be evaluated over 1200 iterations. These are the three step rewards that will be examined in testing:

- Step Reward = -10

- Step Reward = 0

- Step Reward = -1

## 8.3 Results

### 8.3.1 PBRS Agents

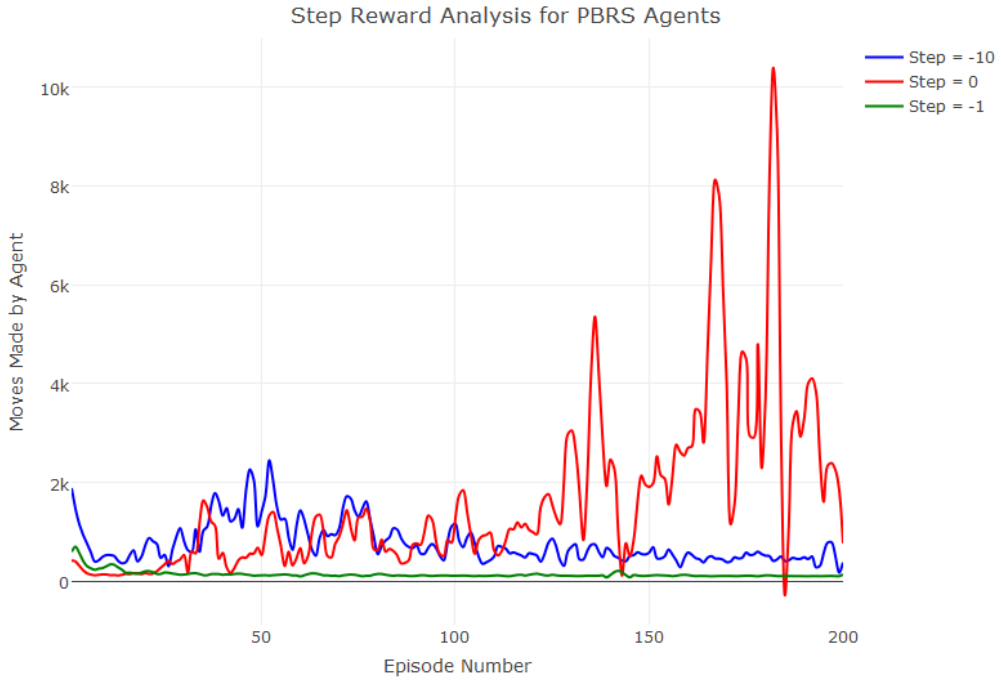


Figure 8.1: Steps taken by the PBRS agent per iteration over 200 iterations

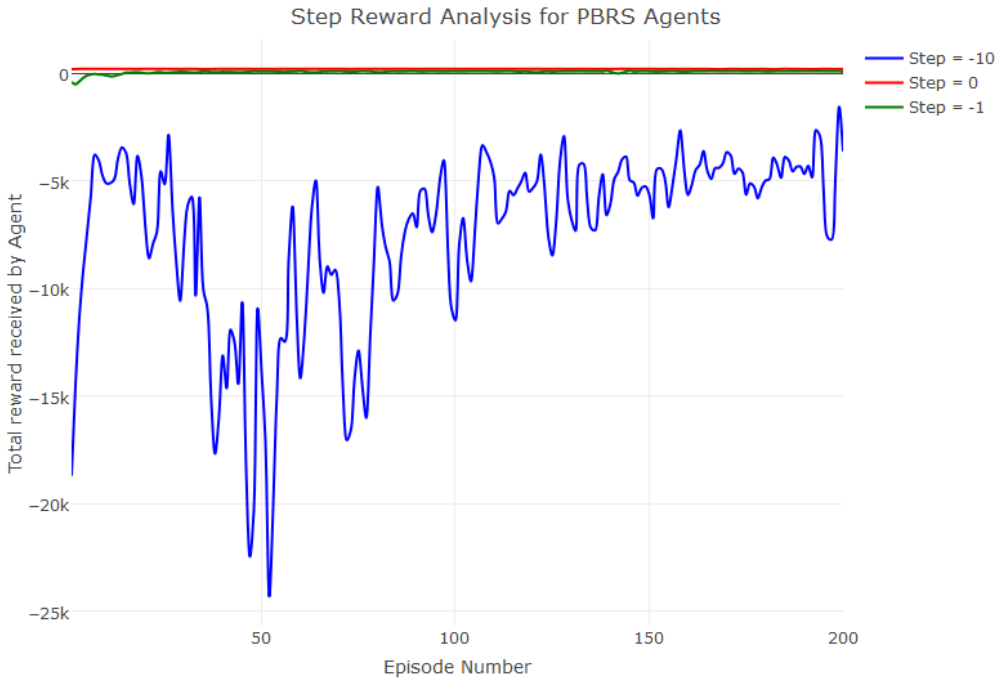


Figure 8.2: Total reward received by the PBRS agent per iteration over 200 iterations

Agent 2, although initially promising, did not continue to improve and its performance decreased dramatically over the course of the task. This can be attributed to the fact that the PBRS agent needs a negative step reward in order to reinforce good behaviour. Potential Based reward shaping provided the agent with the almost exactly the same reward total for every trajectory length. Thus the agent never really learned optimal behaviour and that is why its behaviour is very erratic in figure 8.1.

Figure 8.1 shows that a step reward of -10 proved to be to high to promote a fast training of the learning agent. Initial episodes were very long (>1500) and exploration of the environment sent the agent on extremely large trajectories as can be seen around episode 50. Figure 8.2 shows the massive difference these long episodes had on reward received by the agent. However, Figure 8.1 does show that the agent, although very slowly, does begin to improve its performance and begins to converge to the optimal solution. This is slow convergence is because the large step reward of -10 massively outweighs the heuristic provided by the potential function, which is only +1 for positive behaviour.

Figure 8.1 shows that a step reward of -1 produced the best performance for a PBRS agent. The initial training period is very quick and although it is initially outperformed by Agent 2, it converges to the optimal solution faster (approx 90 episodes) and does not deviate from it compared to Agents 1 and 2.

### 8.3.2 Agents without PBRS

With no value for step reward, Agent 2 has no method of reinforcing good behaviour as the only reward now present in the environment is the goal reward of 100. So as long as the agent completes the maze it will be rewarded the same amount regardless

of trajectory length. Thus the agent never learns and trajectory lengths are lengthy and random. For this reason I choose not to graph Agent 2's performance as the scaling of the graph to fit agent 2' s enormous values would distort the results of the other 2 agents.

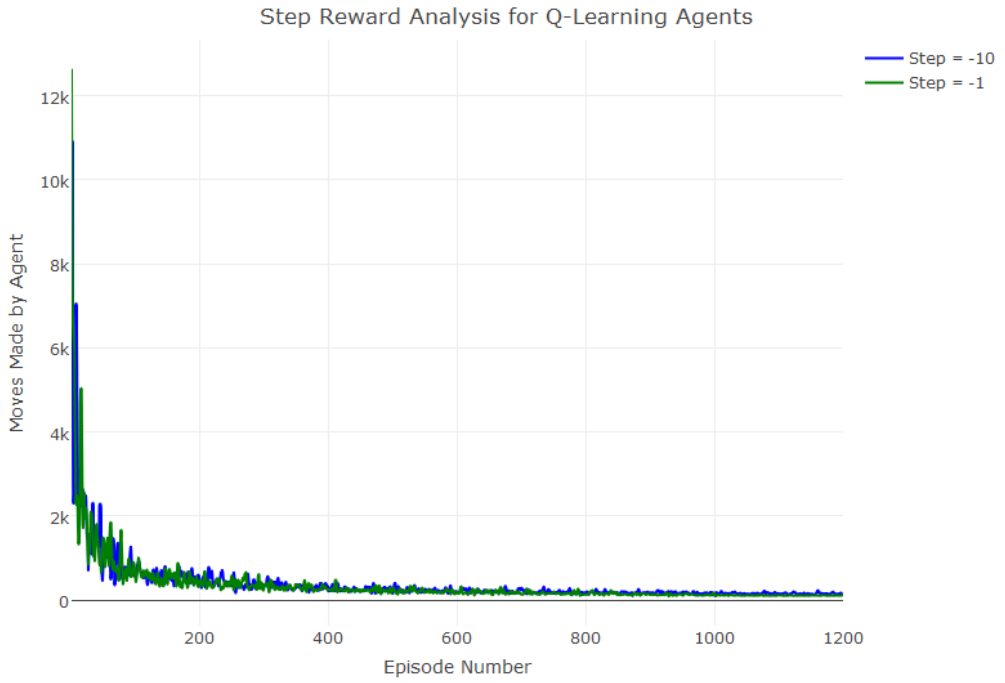


Figure 8.3: Total steps taken by the agent per iteration over 1200 iterations


Figure 8.4: Total reward received by the agent per iteration over 1200 iterations

Figures 8.3 and 8.4 does not present any overwhelming evidence as to which agents performance is better (The decreased reward received by agent 1 is expected because of the large difference in step reward). Therefore I performed a paired t test between agent 1 and agent 3.

**Paired T Test**

| | Agent 1 | Agent 3 |
|---|---|---|
| Mean | 385.15 | 357.06 |
| St. Dev | 691.51 | 699.95 |

Table 8.2: Paired t test for Agent 1 VS Agent 3 over 1200 iterations

The p value for these pair of agents is 0.0445. This shows that there is a significant statistical difference between the agents. The lower value for step reward leads to a decreased mean number of steps taken by the agent. Another important point is that, although it is not clear from figure 8.3, Agent 3 reaches the optimal policy (as allowed by the exploration rate) after approximately 1000 episodes whereas Agent 3 never reaches it; however it is still improving after 1200 iterations. These results show that a step reward of -1 is the ideal step reward to produce maximum agent performance in the environment.

# Chapter 9

# Conclusion

This research project provides a keen insight into the intricacies of the Q-learning algorithm and provides a detailed analysis of a learning agents performance when using Q-Learning on its own and in conjunction with Potential Based Reward Shaping. The contents of this paper can be summarised as follows:

**Exploration Rate**

- For both PBRS and Q-Learning without PBRS a large initial exploration rate (0.5) did not result in the agent finding an optimal policy faster compared to an agent with a lower exploration rate of 0.1. As expected, the agents with a high learning rate of 0.5 had large initial trajectories compared to the agent with the lower exploration rate.

- Although a decreasing exploration rate lead to optimal agent performance for this learning task it is ultimately better to have a low fixed exploration rate of 0.1 in order for the agent to have the agent continuously learning.

**Learning Rate**

- For agents using PBRS, results did not conclusively prove which learning rate performed best for the learning task with all agents performing very similarly.

- For agents without PBRS, a large value for the learning rate lead to better agent performance with a learning rate of 0.85 performing slightly better than a learning rate of 1.

- A learning rate of 0.7 resulted in convergence taking approximately 300 iterations longer than the other two agents

**Discount Factor**

- The equations that maintain the viability of PBRS as an effective reward shaping method severely limited the amount of testing that could be performed on the discount factor.

- PBRS agent results showed that reduction of the discount factor from 1 to 0.99 significantly reduced agent performance.
- The reduction resulted in the agent failing to converge to the optimal solution within the 500 iterations whereas a discount factor of 1 led to convergence after approximately 120 iterations.

**Scaling Factor**

- A scaling factor of 2 produced maximum agent performance for the RL task; the agent converged to the optimal policy (as allowed by the exploration rate) after approximately 45 episodes.
- The learning rate of the agent with a scaling factor of 1 was much slower than those of the other two agents and initial episode lengths were also much larger ($\approx$ 250 moves per episode larger)
- Scaling of $\tau$ to values greater than 2 did not lead to an increase in performance; initial learning was faster but the episode lengths tended to fluctuate a lot more leading to decreased cumulative reward.

**Step Reward**

- A step reward of zero initially led to faster training both PBRS agents and Q-Learning agents without reward shaping, however with no negative step reward to reinforce good behaviour the agents proceeded to embark on long episode paths as cumulative reward became independent of trajectory length. As a result performance decreased dramatically.
- A step reward of -10 resulted in larger training times for both methods and the agent didn't converge to the optimal policy for either method within the iteration limits examined in testing.
- -1 proved to be the optimal value for the step reward for both examined methods. For the PBRS method, the agent converged after 90 episodes while the Q-Learning agent without shaping converged after 1000 episodes. A paired t test also confirmed the superiority of the lower step reward.

In conclusion I believe that this research project has achieved the goals it set out to accomplish. The in-depth analysis of the Q-Learning algorithm and its inner workings this research provides will allow researchers to optimize their future Q-Learning implementations to achieve maximum agent performance without having to test various different parameter values.

# Bibliography


[1] Richard S. Sutton, Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 1998.

[2] Andrew Y. Ng, Daishi Haranda and Stewart Russell. *Policy invariance under reward transformations: Theory and application to reward shaping*. Proceedings of the 16th International Conference on Machine Learning, 1999 pp. 278-287

[3] G. J. Tesauro *TD-gammon, a self-teaching backgammon program, achieves mastr-level play*. vol 6, no. 2, pp. 215-219, 1994

[4] Peter Stone. *Layered Learning in multiagent systems: A winning approach to robotic soccer*. MIT Press, 1998

[5] Sam Devlin, Marek Grzes, Daniel Kudenko *Multi-agent, reward shaping for RoboCup KeepAway* Proc. AAMAS '11 The 10th International Conference on Autonomous Agents and Multiagent Systems, Vol 3, pp 1227-1228

[6] J. Randlov and P.Alstrom *Learning to drive a bicycle using reinforcement learning and shaping* Proc. of the 15th international conference on Machine Learning, 1998, pp.463-471

[7] Eyal Even-Dar and Yishay Mansour *Learning Rates for Q-Learning* Journal of Machine Learning Research, Volume 5 (2003) pp.1-25

[8] www.robocup.org *Robocup*

[9] Melaine Mitchell *Complexity: A Guided Tour* Oxford University Press

[10] Christopher Watkins *Learning from delayed rewards* Cambridge, 1989.

[11] Marek Grzes and Danial Kudenko *Theoretical and Empirical Analysis of Reward Shaping in Reinforcement Learning* Proc. of the 8th International Conference on Machine Learning and Applications, 2009, pp. 337-344

[12] www.cse.unsw.edu.au/ cs9417ml/RL1/source/ *GridWorld.java*