Final Year Project Report

**Full Unit – Final Report**

Reinforcement Learning

Max Anderson

A report submitted in part fulfilment of the degree of

**BSc (Hons) in Computer Science**

**Supervisor:** C.J. Watkins

Department of Computer Science

Royal Holloway, University of London

March 07, 2017

**Declaration**

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count:

Student Name: Max Anderson

Date of Submission:

Signature:

Table of Contents

[Abstract 3](#_Toc367967467)

[Project Specification 4](#_Toc367967468)

[Chapter 1: Introduction 5](#_Toc367967469)

[1.1 How to use this template 5](#_Toc367967470)

[Chapter 2: Styles in Word 6](#_Toc367967471)

[Chapter 3: Page Layout & Size 8](#_Toc367967472)

[Chapter 4: Headings 9](#_Toc367967473)

[4.1 Second Level Headings 9](#_Toc367967474)

[4.2 A Word on Numbering 9](#_Toc367967475)

[Chapter 5: Presentation issues 10](#_Toc367967476)

[5.1 Figures, Charts and Tables 10](#_Toc367967477)

[5.2 Source Code 10](#_Toc367967478)

[Chapter 6: Table of Contents and References 11](#_Toc367967479)

[Chapter 7: Project Information and Rules 12](#_Toc367967480)

[Bibliography 13](#_Toc367967481)

Abstract

Reinforcement learning is a branch of machine learning that is focused on an agent learning what to do in an environment it does not know. This is achieved by the notion of rewards. The agent receives rewards when it accomplishes certain tasks, or receives a punishment when it does something wrong. It is then the goal of the agent to learn from these, and understand what it should do in order to receive the most reward. There are various different methods for achieving this.

The goal of this project is to implement reinforcement learning methods in simulated environments, starting with simple cases such as a small 2d environment. From these simulations, experiments on various parameters and different methods will be done to show variances in learning capability, efficiency and accuracy. The simulations will be viewable in real time, to demonstrate how an agent can learn with these methods.

The work complete by the end of the first term has resulted in an MVC-designed program, with a menu screen that allows for editing graphical elements, the size of the environment, and the speed at which the simulation should run. A system for creating custom environments has also been partly implemented, which allows the user to define transitions from states, and then use this environment in the simulations. Q-learning has been implemented for a 2d, grid world environment with a user-defined amount of states. An agent is visible as it takes actions and learns. The speed the simulation is running at is modifiable during run-time, as is the exploration value of the agent (See the report on Q-learning for information). The simulation is run in a separate thread from the model, so it is not slowed down by the rendering of the environment on the screen.

The program can be found under the ‘software’ folder of this submission. To run, navigate to the src folder. Compile the source files with:

javac gridWorld/\*.java

and then run with:

java gridworld.GridWorldLauncher

The GUI for this program was written in JavaFX8, and therefore requires java 8 to run and compile.

The following git hub link is the repository for the project, including the IntelliJ IDEA software project files and the reports.

https://github.com/MaxAndersonRHUL/FinalYearProject

Project Specification

Your project specification goes here.

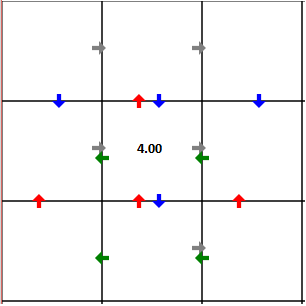
# MDP

A Markov Decision Process (MDP) describes an environment that is used in a wide range of fields and situations, including reinforcement learning. An MDP is a general, formalized structure of states and actions. An MDP can have infinite number of states *s*, and each state can have actions *a* that lead to other states (or the same state). We call this the transition function. An agent can move through this environment, taking certain actions, observing states and therefore navigate its way through the MDP, typically in search of some goal or reward.

The environment can reward the agent for taking certain actions, which is defined as the reward function, *r(s,a)*. A ‘reward’ is simply a numerical value of any size that an agent can see. It may represent a completed task, a successful execution, to tell the agent it’s doing the right thing, or something similar. Rewards can also be negative, to punish the agent for taking certain actions. This idea of rewards is at the heart of reinforcement learning – it’s about the agent noticing it has received a reward, and learning over time how it can get as much positive reward as it can by making optimal action choices.

In a deterministic MDP, an action always leads to the same state, and the reward for choosing an action remains the same throughout the life of the MDP. Conversely, in a non-deterministic MDP, an action may lead to a different state than it had before, and the reward for an action may change, as determined by a probability distribution based on the current state and action. This probability distribution depends only on the current state and action - the history of visited states and chosen actions bear no influence on the distribution. This often provides an environment that is more similar to the real world than a deterministic MDP. For example, the case of a robot moving through the physical world. In some circumstances, the robot’s actions will not always lead to the same state – when moving forward, it may get caught on a carpet, or pushed by human, resulting in arriving in an unexpected state. The randomness in a non-deterministic MDP simulates this.

In the case of this project, we do not let the agent know the reward function or the transition function. The agent will only see the state it is in, any reward it received from taking an action, and the actions it can take within a state. This is a common way to model an agent in reinforcement learning, as one of the goals of this branch of machine learning is for an agent to learn knowing little information about the environment. In the grid world example for this project, our states will be represented on a simple 2d grid, with transitions going north, south, east and/or west of a state. Below is an example of such an environment. Arrows represent transitions into other states. The centre state has a reward of 4.



# Optimal Policies and Value Function

If an MDP rewards an agent for taking certain actions in the environment, then the agents goal is typically to maximise the reward it gains throughout its execution. It can do this by formulating a list of rules or a mapping from states to actions that it believes, if followed, will result in the most reward. This is called a policy, which is denoted by π : S → A, where s is the set of states and A is the set of actions for a state. The agent we are considering for this project knows nothing about the environment except the state it is in and the actions it can take. Therefore, the agent must explore the environment, receive rewards, and figure out a good policy. An optimal policy, π\*, is the best possible choice of actions; or the actions that lead to the highest reward possible in an MDP.

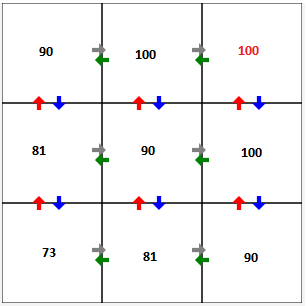
An agent is not required in order to find an optimal policy – there are other methods that will guarantee to find an optimal policy for non-deterministic or deterministic MDP’s. However, these methods require more information about the environment than we are constraining our agent to. We are interested in calculating the optimal policy in these ways, as it can be used to test against the agent’s performance. We therefore define the value function, Vπ(st), as the cumulative discounted reward that can be gained, by starting in state s­t and following policy π. The rewards are ‘discounted’ by a constant γ, between 0 and 1, to give more value to states that are closer to rewards. We end up then with the formula:

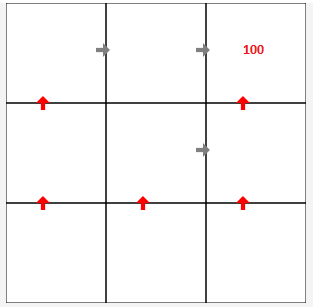
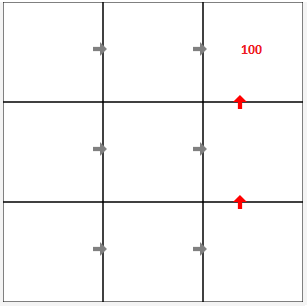
An optimal policy should then maximise this value function. The value function for an optimal policy is denoted by V\*(s). In a non-deterministic MDP, this is an estimation of the value function but can still be considered optimal if the probability distribution for a state is known. We can implement an algorithm that finds the value function for a policy by starting at the end or reward state and calculating the discounted values for each state that has actions into the reward state, and then doing the same for each state we add values to. A state’s value is the discounted estimated future reward + any immediate rewards gained from the state. The value iteration update equation is found by turning the Bellman optimality equation into an update rule.

Value iteration will allow us to know the value function, which can help us find the optimal policy. Policy iteration will find us the actual optimal policy – the exact mapping of states and actions that yields the largest estimated reward. It works by first having a policy in place that we can iteratively improve. This initial policy can be random, or we can use some other method for generating a policy that may be closer to the optimal policy. Then, following the policy, the value of each state is calculated (as explained above), and given these new values, we see if we can change the policy to increase its total estimated reward. If so, we make that change and iterate again until the policy does not change.

The two algorithms can be used to guarantee us an optimal policy – but in many environments, especially real-world environments, we cannot get the information necessary to use them. A robot cannot know the probability of being pushed by a human, or its motor moving a few degrees to far, and so cannot know the state-transition function. In our grid world environment however, we do know these functions, and so we can use them to test against the agents learning speed and accuracy, while still simulating an environment where the functions are unknown to the agent.

A 3x3 grid world example is shown in figure 1. Here, each state has a transition to adjacent states as shown by the coloured arrows. The red ‘100’ indicates a reward state, granting a reward of 100. The black values indicate the values from V\*(s) – the value function for the optimal policy. In this case, the discount variable γ was chosen to equal 0.9. 2 different examples of a possible optimal policy are shown in figure 2. While the policies are different, they are both optimal and the value function will be the same for both of them.





# Q-Learning

We have now seen how to calculate a guaranteed optimal policy for an environment. These calculations have required us to know the transition function and the reward function, and so the next task is to learn the optimal policy without this information.

As the agent can only see the state it’s in and the actions it can take, we want to learn some function that will tell the agent the action that is expected to result in the greatest reward. Previously, this was achieved with the value function. Since we do not have the information required to learn this, we introduce a new function, named the Q function

Q(s, a). The Q function is the maximum discounted cumulative reward that can be gained when in state s, taking action a, and following the optimal policy thereafter. The optimal policy is then simply found by taking the action with the highest Q value in any state. This allows the agent to choose optimal actions without having to look ahead at resulting states, and therefore does not required knowledge of the transition and reward functions.

If we learn the Q function, the agent can know the optimal policy. To learn the Q function, the agent will move around the environment, receive rewards, and update its estimation of the Q values after it takes actions. Hopefully, after some amount of iterations, this estimation of values will converge to their true values. The agent updates Q values in a deterministic MDP as follows: First it observes the current state, and chooses an action. The agent observes the resulting state, observes any rewards given, then changes the Q value for the action it took. It estimates Q values as: [1]

In other words, it changes the Q value of the action it took, to equal any immediate reward it received when entering the new state, plus the largest Q value that an action has in the new state, discounted by γ. Every time an action is taken, the agent updates the corresponding Q value.

Unlike policy iteration, we have no easy way of knowing when or if the agent has learned an optimal policy. One requirement we place on this learning algorithm to try and ensure it will converge on correct Q values, is to say the agent must visit every state and take every action, infinitely often. If the agent only takes a select few actions, or if it only takes some action once or twice at the start of its execution, it may end up with a non-optimal policy, as the actions that are not visited are not properly factored into the remaining Q values. To solve this in a grid world environment, the agent should always have a chance to take any action in every state.

As the agent moves through the environment and updates Q values, it should start to build up a better and better policy. In some situations, the policy is not required to be optimal for it to be useful. It may be the case that the agent should try and be effective at receiving rewards as soon as possible. We achieve this by deciding how the agent should choose actions. A simple way of choosing actions is to pick purely random action choices until the Q values are no longer changing, therefore meaning we have an optimal policy. Then, the agent can always choose the highest Q value to follow the optimal policy. This may take a long time, and the agent is not very useful when choosing random choices. A good method of choosing actions is to balance exploration of the environment, and exploitation of the current policy. We can do this by assigning probabilities to actions when the agent enters a state. Actions with higher Q values should have a higher probability, but all actions should have a non-zero probability. A value k > 0 is used to control how much weight Q values have on the probability. The formula for calculating such a probability when in state s, and calculating the probability of choosing action ai is shown in figure 3. [1]

Figure 3.

Larger values of k will result in the agent choosing actions with high Q values more often, and so will exploit the current policy, whereas lower values will assign high probabilities to low Q values, meaning the agent will explore actions with low Q values.

## Q-Learning in non-deterministic MDPs

As explained in the chapter ‘MDP’ of this report, a non-deterministic environment is a more realistic model for testing our agent. In this case, when an agent takes an action, the resulting state and reward is determined by a probability distribution. Because of this, we redefine our Q function for state s and action a as follows.[1]

Instead of the immediate reward for taking action a in state s, we take the expected reward, and we include the probability of the action resulting in the expected state s.

The agent’s method for updating Q values in a deterministic environment does not converge to the true values in a non-deterministic environment. If the reward for a state is constantly changing, then the agent will be stuck forever updating Q values, and will never find an optimal policy. This can be solved by using a decaying weighted average of the current Q value estimation. It requires the agent remember the amount of times it has taken an action in a state, and is defined as: [1]

Where

Visitsn(s,a) is the number of times the agent has taken the action a in state s at the nth iteration. Similarly, is the prior estimation of the Q value.

The changes made from the deterministic method mean that, as the number of times an action is taken increases, the changes made to the corresponding Q value become smaller, until eventually they are small enough changes to disregard and we can assume the Q value estimation has converged to the true value.

# Software Design

A goal of this project is to design and create a piece of software that implements the reinforcement learning techniques in a way that can be viewed and changed. The software created for this project aimed to implement proper software design techniques to reduce coupling of objects and to increase the cohesion and readability of the code. This is important, as speed is a notable factor within this program, and clear code makes it easier to understand what the program is doing during a simulation, and the parts of the code that are run most often.

## Generic Reinforcement Learning System

The software for this project was designed to work for any environment, with any user viewable graphics, and any reinforcement learning techniques that use MDPs. This requirement led to careful planning and decisions. Environments can vary wildly, such as a grid world where states have actions that can only go north, east, south, west, or a noughts and crosses game board where actions are places to put a cross, and states are a whole board.

The base classes ‘Controller’, ‘Model’, ‘View’, ‘State’ and ‘StateIdentity’ handle a lot of the work when it comes to running the simulations. They provide a basis for complex environments and controllers by hiding as much of the complexity as they can, without excluding environments and learning algorithms from being implemented.

The model contains the states, and states contain actions. The model also keeps track of the agent, and provides functionality for getting actions to a particular state, moving the agent in a random direction and calculating policy accuracy against a given policy. The model will do nothing on its own – the view may use the model to get the current environment information to display to the user, and the controller will tell the model when the state of the environment has been changed. This is useful, as some models may need to act upon the agent’s position. For example, environments can be procedurally generated as the agent finds them. This means, instead of creating every state and action possible when the simulation is started, the model can create states and actions as the agent discovers them. In the grid world model that is supplied with the software, the ‘GridWorldModel’ class inherits from ‘Model’ and adds functionality to setup grids that conform to the environment constraints – in this case, actions going north east south and or west.

The controller class is synonymous with the reinforcement learning algorithm being implemented. It provides a controlled thread that calls functions in the controller class in a time frame specified by a variable that is set by the user on the launch panel when the program is started. The functions the thread calls should be overridden by a child class, and these functions will implement the necessary algorithms to iteratively decide which action the agent should take, and then update any learning values and variables with the information gained by taking that action. The controller’s update thread calculates the average speed it is running at, and provides functionality to be paused, resumed and stopped.

The view needs to be more general than the other classes stated above. Different environments will look very different to one another. The view class does, however, provide UI elements that can be added into a child view such as showing the amount of iterations the controller has gone through, showing the average speed the controller is running at, a canvas to draw graphical elements onto, buttons to pause and play the current simulation and sliders to set the exploration value, if it is used. The view will also handle the setting up and executing of the ‘WorldViewUpdater’, a class that will call ‘redraw’ on the view every frame, up to 60 frames per second.

The state class contains a list of actions that are available from that state. The actions have a resulting state – or list of resulting states along with the probability of the resulting state to be the result of taking that action in a stochastic environment.

### State Identity

The State Identity class was created to encapsulate what a state in the simulation is. For example, in grid world, state identity is a 2 axis coordinate that corresponds to the location of the state on the grid. In the noughts and crosses environment, state identity is a 2 dimensional array of values that determine whether the position on the board contains a naught, a cross, or is empty. The state identity provides a quick, clean way of describing a state, without having to find or already have a reference to a state.

State identities form the basis for data structures in the program. Since the speed of the simulation is a concern, we need a data structure that can retrieve data fast and will remain fast even if there are thousands of entries. One of the requirements for a working state identity is to override the hashcode and equals java functions. Generating a good hashcode that successfully separates different states, and can be calculated in as short a time as is possible can be a difficult task for state identities that are capturing complex environmental states. The grid world state identity is simple.

public int hashCode() {

int hash = 17;

hash = ((hash + x) << 5) - (hash + x);

hash = ((hash + y) << 5) - (hash + y);

return hash;

}

This particular hashcode function was written by Boris Pavlović, and can be found at the following webpage: http://stackoverflow.com/questions/3934100/good-hash-function-for-list-of-2-d-positions it uses bit shift operators instead of multiplication to increase performance.

The noughts and crosses state identity is a trickier endeavour. It is shown below.

int hc=board[0].length \* board.length;

for (int j = 0; j<board.length; j++) {

for (int i = 0; i < board[0].length; i++) {

if(board[j][i] == LocationState.CROSS) {

hc = (hc\*31);

} else if(board[j][i] == LocationState.NAUGHT) {

hc = (hc\*31) + 1;

} else if(board[j][i] == LocationState.EMPTY) {

hc = (hc\*31) + 2;

}

}

}

return hc;

This generates hashcodes by iterating over all the locations on the board. For smaller board sizes, this should not be a major hindrance to performance, but it is certainly slower than the grid world’s identity. A fast and effective hashcode is very important in the software, as many data structures rely on these codes to quickly access states or other information. In particular, many hashmaps exist throughout the code that map state identities to some other data. The model contains a hashmap which maps state identities to states, for example. This allows quick access to a state, just by constructing the identity of the state and retrieving it from the hashmap. Consider, for example, the following code snipet.

Action foundAction = model.getStateTransitionTo(state, states.get(

new GridWorldCoordinate(state.getStateIdentity().x - 1,

state.getStateIdentity().y)));

This line is found in the GridWorldView class. It wants to know whether to draw an arrow from the state named ‘state’ to a state that is adjacent at x – 1. It needs some way to reference the adjacent state without having an actual reference – a new ‘GridWorldCoordinate’ is created. GridWorldCoordinate inherts from State Identity. It is created with the x and y values of the state, and can be used by the model to find a transition from ‘state’ to the state beside it, if one exists. Using hashcodes, the model can retrieve a reference to the actual state in O(1) time. The view was able to tell the model which state it was interested in by creating a grid world coordinate and supplying the coordinates of the state it wants. The model will then use the state identity as a key in the hashmap to retrieve the actual state. It can then look through the 2 states actions, and return an action if there is one transitioning between them.

Another example showing the use of state identities is in the value iteration controller.

public HashMap<StateIdentity, Double> stateValues;

The value iteration controller wants to run independently from any other part of the system. It therefore stores all the data it generates and uses itself. Here, state identities are being mapped to a double value which represents the value that the iteration controller has calculated. If the model wants to use this to calculate what the current policy accuracy is, it can do so using the identities. Furthermore, state identities are valid forever within the system. If experiments on the same environment are stopped, started, etc. and states are destroyed and changed – the state identity provides a constant way to reference these states. It is especially useful when running an amount of simulations and averaging the results of all the simulations when finished. When a simulation ends, the model is given a new set of states. The previous set of states is no longer used by the model, so any references to such states are useless and may affect the running of the program. State identities allow a persistent way to reference states.

## MVC

A model-view-controller design pattern works well for this software, as there is a logical separation of duties. The model represents the information for the environment – which is simply an MDP. The controller is the agent and the learning methods it uses to traverse the model. The view will take the information from the model and display it in a particular way that bests represents the environment and the agents’ goals. In the case of a grid world, a 2d grid of states, logically displayed such that transitions are going north, east south and west to adjacent states.

## Learning Speed

One of the potential is the amount of iterations it may take to learn in environments with many states and actions. Modern hardware is fast, but the faster the agent can learn, the easier it is to experiment and test. Drawing graphical elements on the screen is not computationally easy, and if we want the simulation to iterate thousands of times a second, it may become an issue if we are trying to draw the agent, the states, the transitions and the values to the screen at the same time. It takes a lot more time for the computer to display the live simulation on the screen then it does to calculate 1 iteration of a Q-learning algorithm. Because of this, the view is updated at most 60 times a second. This is controlled by the WorldViewUpdater. The controller, and therefore the iteration speed for the learning algorithm, is run in a separate thread. The speed this runs at can be controlled by the user before and during a simulation.

## GUI

The GUI for the software is written in JavaFX. This was chosen due to its excellent looking standard UI elements, its ability to run on Windows, Linux and MacOS, and the canvas which allows for drawing shapes and text. The ability to display graphs using JavaFX was also a useful feature.

## Experimentation in Grid World

There are many variables that can be changed on the launch panel when the program is first started for grid world. The position for the reward state and the amount of reward it gives, the start position of the agent, the discount variable, initial Q-values between a range, episodic environment or not. There is also the facility to create new grid worlds, with user defined actions, states and rewards. These environments can be saved and opened at any time. Whilst the simulation is running, the exploration value can also be set, and the policy generated by value iteration can be viewed. Most of these are simple variables that the GridWorldLauncher will extract from text fields, and assign to the necessary classes. The CurrentSimulationReference is a convenient way to get references to the current controller, model and view that are currently being used.

The graph view displays data in a line graph to the user. There are various different metrics that can be plotted onto the graph view. A list of metrics is displayed showing the name, the current value, and the amount of records of the value the simulation has stored. By selecting a metric, it will be plotted on the graph’s y axis, with iterations (agent actions) on the x axis. By selecting 2 metrics, they will be plotted against each other on the graph and related by iterations. The graph view provides a real-time view at the data of the current simulation being run.

Data can be added to the list of metrics on the graph view very easily, with the ExperimentableValue class. ExperimentableValue objects have a value and a name. When such a value is created, it will register itself to the experiment controller. Every few iterations (defined in the UI on the grid world launcher panel) the experiment controller will iterate through all the experimentable values that are in existence and record their values, along with the iteration the value was recorded on. The graph view will look at these records and values and do all the necessary work to display the metric in the list and allow it to be plotted on the graph. All that is required to add a new metric to be plotted is to create the experimentable value and update the value. An example is given below.

transient public ExperimentableValue currentConvergencePercent = new ExperimentableValue(0.0, "Policy Accuracy (%)");

The model updates this value every 5 iterations by recalculating the convergence percentage. This value exists in the model class as it serves a purpose there – the view references the value stored to display to the user. Some metrics have no reason to exist anywhere in the model, as they are only used to display graphs. In this case, the values are created and updated in the experimentation controller.

# Grid World Experiments

## Metrics

**Policy Accuracy** is a percentage that defines how close the learning policy is to the optimal policy. It is calculated by comparing the current actions with the highest Q-values in each state with the actions chosen by the optimal policy. For example, if a state has 3 actions and the optimal policy says the second and third actions are equally optimal, then the Q-values for those second and third actions must be equal, and higher than the first for the state to be considered accurate and increase the policy accuracy percentage. Therefore, the policy accuracy metric can increase and decrease as the agent is learning, and is not considering the convergence to the true Q-values for an action.

**Average reward per 1000 actions** records the rewards the agent received within the last 1000 actions, or the total amount of actions taken if the total is less than 1000, and calculates the average reward received per action taken. This value can be expected to increase when exploiting the current policy, and to decrease when exploring the environment. Average reward per 100 actions is also available.

**Exploration Value** is the value that determines whether the agent should exploit the policy or explore the environment. Values of less than 1 will favour actions with low Q-values, whereas values greater than 1 will favour actions with higher Q-values.

**Total Reward** is a simple summation of all the reward the agent has gained by taking actions that yielded a reward.

**Iterations** is the amount of actions the agent has taken.

## Experimental Method

Experiments can be run in the Grid World program. Many features are included to help with this, including random initialization of Q-values from a specified range, episodic or non-episodic, agent exploration value or pure random action choice, changing the discount variable, viewing value iteration on the environment, the metrics listed above calculated in real-time, graph view displaying the metrics over iterations and viewable in real-time, ability to set the amount of actions need to be taken before a variable is recorded for the graph view, a display of the current grid world environment with Q-values shown on actions and the current agent position, the ability to run many simulations one after the other with a set amount of actions per simulation and average the results at the end.

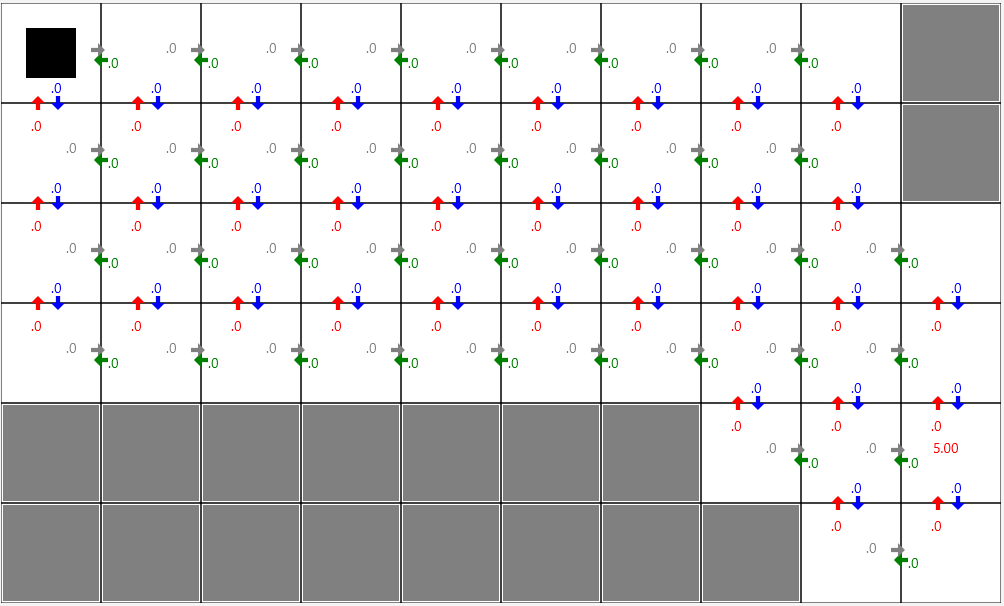
In the following section of this report, many experiments will be run to determine how effective Q-learning is, what effects Q-learnings performance, and to better understand how Q-learning learns from an environment. All experiments have been run at least 200 times over, and the results averaged and shown in graphs displaying the above metrics against iterations.

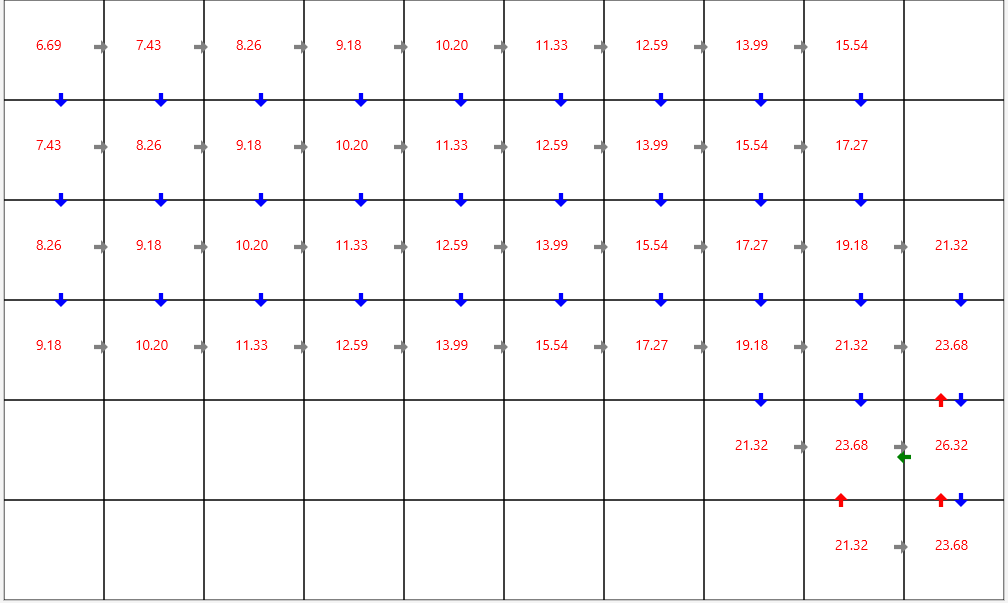
## Experiment I – Agent Learning Vs Random Choice

### Experiment I - Details

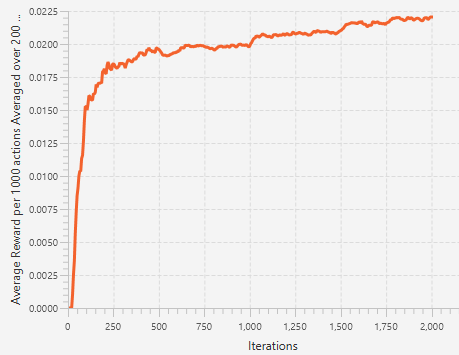
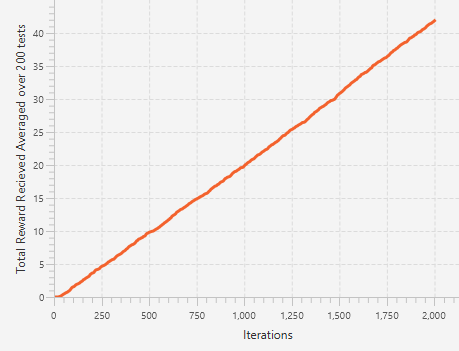
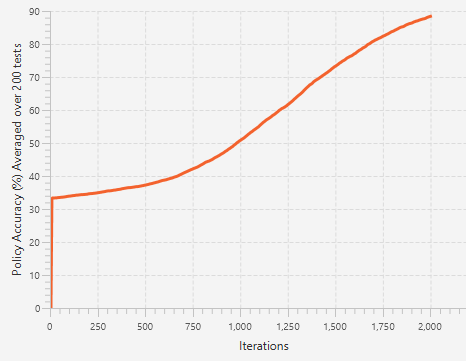
The aim of this experiment is to see how much of an effect the agent learning has when attempting to gain as much reward as possible, when compared to an agent that chooses purely random choices. The result will allow us to decide whether the learning is helping the agent to make good decisions, or if the learning is detrimental to the agent, or if the learning has no effect when compared to an agent making random choices.

This experiment will be run in the deterministic environment shown in figure 1. The agent, shown as the black square, starts in the top left state, and upon reaching the reward state shown in red text in the lower right, will be placed back to the start state. The agent will learn via Q-learning methods. The text on each arrow represents the current Q-value estimation for that action. Initially, all Q-values will be set to 0, and the discount variable will be set to 0.9. The simulation will run 200 times, each time resetting after the agent has taken 2000 actions. The results of each experiment will be averaged at each recorded time interval (1 per 5 actions taken). The random choice experiment will have an agent that has an equal chance to take any action when in any state. The Q-value estimations will still be calculated based on the agent’s action choice, so we can see the policy it is learning, even if it has no intention of using that policy. The learning agent will be run in the exact same environment, but instead will take action choices based upon the exploration value. The exploration value will be set to 1.5, to favour actions with a higher Q-value than others within a state, but still has the chance to explore. Figure 2 shows the optimal policy for the environment shown in figure 1, calculated via value iteration. Every state has an arrow leading out from it, which represents the action that is expected to result in the highest reward, when the rest of the policy is followed. The red text represents the value for each state. A state can have more than 1 arrow leading out, if there are actions which expect to result in the same reward.





### Experiment I - Results

**Random Choice**

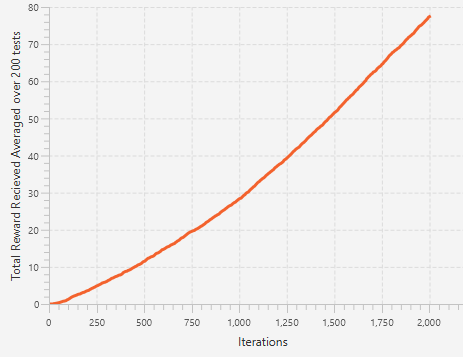
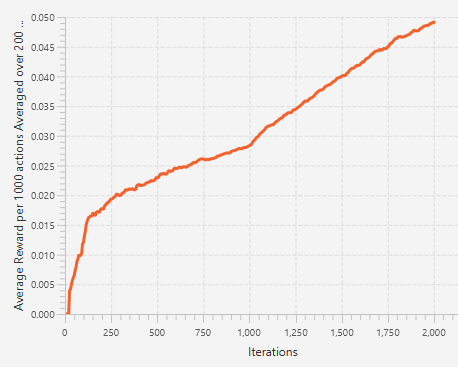
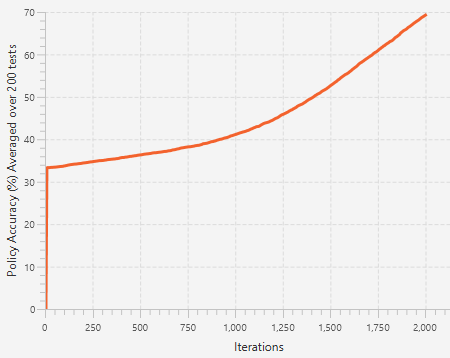
The initial bump in policy accuracy is due to all variables being set to 0, whilst the policy accuracy for this environment begins at 33.3%. It can therefore be ignored.

The results above show the average total reward received at the end of 2000 actions was 42.5, the average policy accuracy was 89%, and the average reward over 1000 actions was in the region of 0.0175 to 0.0225.

The policy accuracy shows an interesting curve as the iterations increase. This curve is likely being caused by the time it takes for the agent to initially find the reward state. Since all Q-values are set to 0 and the reward is fairly hidden in the environment, sometimes the agent may take many actions before it finds the reward, and starts to propagate the Q-values back. The accuracy change over iterations then begins to decrease towards the final 500 or so iterations. As more and more states become accurate, there will be less states for the agent to learn which action to take. This, on average, will reduce the amount of learning an agent will do per action, as it will be harder to find states that are not considered accurate.

= 0.0445% policy accuracy increase per action taken by the agent.

By inspection, the total reward received is an almost perfect linear trend, and shows the agent on average gained = 0.02125 reward per action taken. The optimal policy for this environment shows that the minimum amount of actions possible to get to the reward state, when starting at the start state is 13. The reward is 5, so the maximum reward per action taken possible is 0.3846. In this simulation, the reward gained was only 18.1% of the total it could have achieved.

**Probability weighted choice – 2.0**

The results for the agent making decisions based upon the Q-values experiment as displayed above show an average policy accuracy after 2000 actions of 69.8%. The average total reward after 2000 actions was 78.

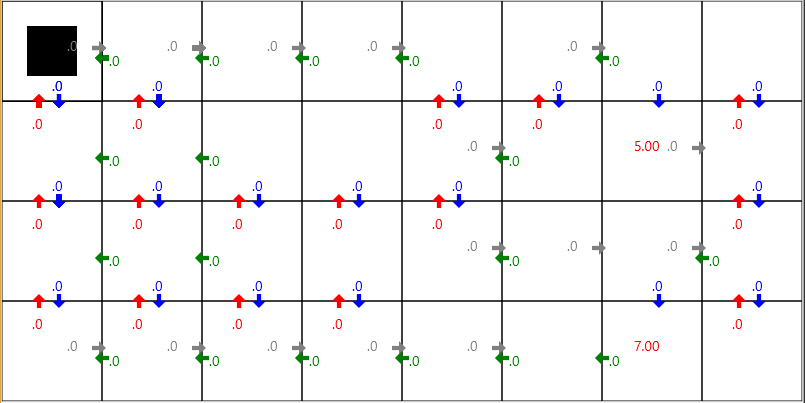
Comparing the results from the agent probability-weighted decisions with the randomly chosen actions agent show that the agent taking random actions resulted in a more accurate policy than the probability weighted choices. This can be surmised to be due to the exploration value of the probability weighted decision agent. At 1.5, the agent will favour actions with a higher Q-value, and so will more often take actions it has already taken. This leads to slower learning, as a requirement for Q-learning convergence is that every state action pair must be visited infinitely often. The line is arched in the centre – it can be theorized that, if the experiment was ran for a few more thousand iterations, the curve of the line would look very similar to that of random choice, except more stretched out on the x axis as it takes longer to learn. = 0.0349 percent accuracy increase per action taken.

The total reward received is 54% higher than random choice. The average reward per 1000 actions shows a steady increase in reward gained. Random choice shows a slight increase over iterations, whereas probability weighted choice shows a steeper increase. The difference is probability weighted choice is increasing the reward it receives as it explores and learns the environment. The maximum amount of reward per action is 0.3846. This agent achieved, on average, = 0.039.

### Conclusion

From this data, the conclusion to be drawn is that an agent favouring actions with higher Q-values will lead to a higher reward than that of a random agent, but at the cost of the speed at which it learns. The probability weighted agent was able to exploit the policy as it learns it. In situations where the agent primary goal is to learn a good policy for the environment, the random agent will perform better. In a situation where the agent needs to learn the environment whilst also receiving a high amount of reward, the probability weighted agent would be better suited.

**Increased Action Environment**

The environment used in the two experiments above has been modified, to increase the amount of actions in the environment without affecting the optimal policy for the existing states. The changes appear on the far left of the grid – the states that were empty have been filled with actions.

The same parameters as the ‘probability weighted choice – 2.0’ experiment will be chosen. The results are shown below.

The results here are very different than the probability 2.0 weighted choice experiment. Most notably, the average reward and total reward show the agent found no reward at all until 1000 actions. From the policy accuracy graph, we can see the accuracy jumping around randomly until 1,000 iterations, where the agent then finds the reward and begins to learn the optimal policy. As there is inherent randomness with the agent learning this environment, the same experiment was run 3 times, and the average amount of iterations it took to find the initial reward was consistent with the graphs shown above within a reasonable margin of error.

The agent is finding it hard to find a reward state, as the path leading to the reward states is only accessible via a single action in a single state, and that path also allows the agent to move back on itself. Changing the initial Q-values or the exploration value will not help solve this problem. The agent must randomly move about until it finds the reward. One way to prevent this issue from occurring, would be to record the agents action choices and start by always taking unique actions until a reward is found in a sort of exploration-only behaviour.

# Project Diary

September 29th 2016 – Compiled a document of notes relating to Q-learning, MDPs and general reinforcement learning techniques after reading various resources online. Found the Tom Mitchell – Reinforcement Learning book to be the most useful.

October 10th 2016 – First meeting with supervisor of term 1. Received helpful information on how to model an MDP programmatically. I did not have any work to show.

October 12th 2016 – Started programming the grid world software. Found representing the MDP fairly simple. Created a very basic GUI to show the states. I decided to start work on the program early, as I feel it will give me a better sense of how the background theory works.

October 15th 2016 – Attempting to draw arrows on the grid world display to represent transitions. JavaFX has no way to draw these for me, so have to create it from 2d points, as a custom polygon. These have to also be rotatable 90 degrees. May have to create an arrow object to handle all this.

October 16th 2016 – Found a function in javas awt library to rotate a list of 2d locations around a point. This can help with drawing the arrows. They have to be in a single array of the form [x1, y1, x2, y2, x3, y3…] however, whereas javafx requires 2 separate arrays, 1 for the x coordinates and the other for y coordinates in order to draw the polygon. I’ve therefore defined a basic upward facing arrow in a single array, which the awt function rotates. It’s then split into 2 different arrays for java to render. It’s not very efficient, so if performance problems arise this will likely be the first thing to fix. Arrows are now being properly drawn to represent transitions between states.

October 21st 2016 – Started writing the section of the report on MDP’s. From programming the grid world, I have a good idea of how they work, but I had to re-read many of the resources I found online to accurately describe them.

October 28th 2016 – Implemented an agent that can take actions in the grid world program. I’m not fully sure how the agent should interact with the model to make actions. Currently, a state object has a list of action objects. Action objects simply have a reference to a resulting state. The agent has a reference to the current state it is in, so it could look at the actions and change its state to the resulting state for the action it took. I don’t believe this will work for a non-deterministic MDP however, as an action can lead to a random state.

November 3rd 2016 – Implemented Q-learning into the grid world program. I found the theory fairly easily to implement. Q-values are represented on the screen next to their actions, and are updated in real-time by the agent. Currently, the agent chooses purely random actions. Since this was implemented in a separate thread from the view, I expected to have concurrency issues – however, after running many different environments at many different speeds, I’ve encountered no crashes or faults. The Q-values are converging on some values seemingly instant time when not limiting the iteration speed. I have yet to validate whether the values are the true Q-values.

November 6th 2016 – The agent can now choose actions based of Q-values and an exploration value. I used the method described in ‘Machine Learning’ by Tom Mitchell (13.3.5) in which actions are assigned a probability based on their Q values. Probabilities for each action are accurately calculated, however I had trouble finding a method for selecting one of these based on a randomly generated number efficiently. The method I ended up implementing may a bug when the probabilities are 0 or 1.

November 7th 2016 – Finished the GridWorldCreator class that allows for creating custom grid worlds, with user defined transitions. Specifying rewards is currently not supported.

November 10th 2016 – Second meeting with supervisor. I do not have a laptop, and did not send a git hub link, so I had no work to show. Rearranged for the following Monday (14th).

November 11th 2016 – Wrote the report on value function, value iteration and optimal policies. As with the MDP report, I had to re-read many resources to feel comfortable with explaining the theory.

November 14th 2016 – Received helpful feedback on value function and value iteration, as well as better structuring the report. Currently, information is too spread out and interweaved. Need to define strict sub-titles and describe separately individual parts of the theory.

November 22nd 2016 – Started writing report on Q-learning. As per feedback from supervisor meetings, mathematical notation and equations have been added to relevant sections of the report.

November 30th 2016 – Finished writing the report on software engineering design choices.

Bibliography

[1] Tom M. Mitchell, *Machine Learning* 1997.

Relevant Resources

http://web.mst.edu/~gosavia/tutorial.pdf

This tutorial by Abhijit Gosavi, Missouri University of Science and Technology describes an overview for reinforcement learning in a general way, without making assumptions on implementation. It contains useful information on reinforcement learning key terms, MDPs and formal notation expressing the process of reinforcement learning. Some information was not relevant to this project, such as SMDPs, transition probabilities, R-SMART algorithm.

http://www.cs.ubc.ca/~murphyk/Bayes/pomdp.html#MDP

This introduction to reinforcement learning by Kevin Murphy has a useful explanation of MDPs, and another general explanation of reinforcement learning.

Machine Learning – Tom Mitchell

I found the book on Machine Learning by Tom Mitchell to be the most useful resource. The chapter on reinforcement learning went into detail with Q-learning, and explained the process in a logical, easy to follow way. It explained concepts that the above resources did not cover, such as convergence and the difference between reinforcement learning in non-deterministic and deterministic environments. The explanations often used a grid world as an example, and as that is what I plan to implement reinforcement learning using Q-learning techniques in, it helped me form an idea of what the implementation of grid world will look like.

http://cs.stanford.edu/people/karpathy/reinforcejs/gridworld\_td.html

An interactive grid world demo using temporal difference learning. It gives an example of an agent moving through a grid world, however was not a Q-learning implementation. It was still useful in showing how an agent learns a policy iteratively in a grid world scenario.

http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node7.html

Another great explanation of reinforcement learning, this time by Mark Lee. Use of examples to explain reinforcement learning and useful explanation of reinforcement learning key concepts.

http://www2.econ.iastate.edu/tesfatsi/RLUsersGuide.ICAC2005.pdf

A presentation on reinforcement learning by Bill Smart. I found this helpful as it ‘started at the top and worked down’. Explained MDPs, then policies and value functions, dynamic programming, then the different methods of learning value functions, including Q-learning.