Software Agents: Q-learning
Training Agents for Portfolio Stocks Selection

Aishwarya Chandra Shekhar
180017578

1 Introduction

In this study, Reinforcement Learning is applied to optimise stocks selection to build an optimal investor portfolio. The agent (artificial intelligence used in decision making) will be designed to support an investor's decision-making while selecting the stocks that should be included or excluded from the portfolio, so as to maximise the stock returns/profits.

The paper begins with posing a simple example to demonstrate that the agent can be used for effective stock selection. In the second section, the parameters such as learning rate (alpha), discounting rate (gamma) and epsilon-greedy rate (e) are tweaked to find the optimal parameter values for our study. In the third section, the scope of the study is increased to demonstrate how the agent can be applied to a more complex, real life scenario. And finally, the paper concludes with establishing parallelisms between the Q-learning algorithm and error correction models in psychology.

The algorithm used is Q-learning. Q learning was first introduced by Watkins, C.J.C.H. in 1989 [1] to 'give a systematic analysis of possible computational methods of learning efficient behaviour' in animals. Basically, an agent, much like an animal, learns how to complete a task optimally with repeated attempts and given rewards based on correct decisions made (treats in animal training). In this paper's domain, the agent's decisions (or actions) are to sell/purchase stocks in a portfolio, based on the rewards (portfolio return).

In his thesis, Watkins defined an optimal learning strategy to be one that fulfils all three of the following:
1) The capacity for maximally efficient performance is valuable.
2) Exploration is cheap.
3) The time taken to learn the behaviour is short compared to the period of time during which the behaviour will be used.

The second requirement is less important in current scenario than it was for Watkins as technology has advanced greatly in the past 15 years. In this study, the second requirement is modified slightly to 'Exploration is manageable'. This is to reflect the increased complexity when applying to a real-life scenario (i.e. when the number of states becomes exponentially large).

The third is particularly important in this study's context as the agent is being trained to help an investor make decisions in mere seconds. Q-learning is used over other methods as it has proven to converge in all cases [2] and does not cause problems when the number of states are increased.
Thus, in this study, the aim is to develop a Reinforcement Learning model, such that:

1) It is helpful to an equity investor in selecting the right stocks for her portfolio to increase the portfolio returns.
2) It can be applied to any number of stocks.
3) It takes minimal amount of time as stock markets are very volatile and require quick decision-making.

In this study, Q-learning is applied to the real-life scenario using the following hypothesis:
H1) Q-learning application is valuable in stock selection.
H2) Varying parameters (discount factor and learning rate) improves results.

H3) Changing the learning policy (ε-greedy to random) improves results.
H4) The number of stocks and hence the number of states can be increased to represent a real-time stocks portfolio that allows exploration that is manageable.
H5) The reward matrix can be updated to reflect a real-life scenario.
The quality of results is based on Watkin's optimal learning strategy with emphasis on time taken. The final step is the greatest challenge in this domain. Defining real-life rewards has a large impact on the Q learning algorithm.

1.1 Domain description

Many factors directly or indirectly influence stock markets and make movements of asset prices very uncertain and unpredictable. Selection of portfolio may include two stages [3]. Firstly, performance of different securities is observed with beliefs about their future performances. Secondly, with relevant beliefs about future performances a proper choice of portfolio is made. In modern portfolio theory (MPT) of investment, the main focus is given toward the maximization of expected return of portfolio for a given amount of portfolio risk, or equivalently minimizing the portfolio risk for a given level of expected return, by carefully choosing the investment proportions of various securities.

In this study, we consider the future performance of stocks to choose them as one of the portfolio stocks. The stocks selection is done so as to maximise the portfolio return. S&P 500, an American stock market index based on the market capitalizations of 500 large companies, is considered in this study. It consists of 500 global stocks categorised into 11 sectors, such as Finance, Health Care, Consumer Discretionary and so on. Selecting the right stocks for the portfolio, from a pool of 500 dynamic stocks, is often challenging, as it requires a lot of stocks movement tracking and analysis. Applying Reinforcement learning to stock selection is a relatively new field and not a lot of material is available online. In this study, I have tried to use Q-learning in finding the right portfolio stocks.

To begin with, a portfolio of four stocks is considered. In order to make the portfolio diverse, four stocks from four different sectors are chosen. The sectors considered are Financials, Information Technology (IT), Consumer Discretionary (CD) and Industrials. The task is to select four stocks (one from each sector) from a pool of hundreds of stocks. At any point of time, suppose the portfolio has 4 stocks, viz. A, B, C, D. The price of any one of these stocks may go down and price of any stock from the pool of hundreds stocks may go up. In such a scenario, the investor would want to sell the underperforming stock in the portfolio and purchase the good performing stock from the pool.

At the beginning of any period, a portfolio of four stocks defines the initial state. Initially, the model is based on the idea that an end state is known. In other words, the investor knows which stocks need to be in the portfolio at the end of that period so that those start the next. The investor can sell and purchase only one stock at a time. Therefore, to reach the end state from the initial state, multiple transactions might be required. Traders spend years analysing and familiarising themselves with the stocks and their movements. These are hidden features that an agent simply cannot capture fully. Therefore, we do not design the agent as a means to take decision-making away from the traders, but rather support them in taking optimal decisions in a highly dynamic and volatile environment.

2 Simple example

To demonstrate how this works, we started with a simple example. Initially, the rewards were based only on achieving the desired end state. The number of stocks available to choose from for each sector is shown in the table below.

Finance	IT	CD	Industrials
Stock 1	Stock 4	Stock 5	Stock 7
Stock 2		Stock 6	
Stock 3			

In this example, there are a total of 6 states (3 x 1 x 2 x 1). It is important to note that although the transitions and their rewards are probabilistic. The states depend only on the current state and the current actions and there is no further dependence on previous states, actions or rewards. [4]

It can be seen from the table above that 4 stocks (one for each sector) need to be selected from a pool of 7 stocks.
For simplicity, the initial state (state 0) is defined by the choice of stocks: 1, 4, 5 and 7. This is represented below where each node (circle) represents a stock.

State 0
Finance IT CD Industrial

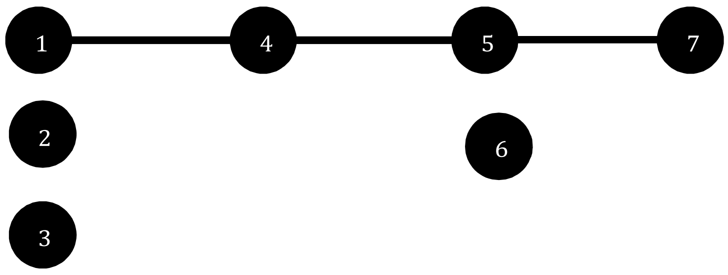


Figure 1: State 0 for simple example

As mentioned previously, the trader may sell and purchase only one stock at a time. Therefore, to have stocks: 2, 4, 6, and 7 in the portfolio, she must sell stock 1 and purchase stock 2, and sell stock 5 to buy stock 6, both transactions need to be made individually. This composition of stocks is denoted as state 5 and the aim in this example is to achieve this in an optimal order from the initial state (state 0).
All states are denoted as:
State 0: 1, 4, 5, 7; **State 1:** 2, 4, 5, 7; **State 2**: 3, 4, 5, 7; **State 3:** 1, 4, 6, 7; **State 4:** 3, 4, 6, 7; **State 5:** 2, 4, 6, 7
The desired goal state is the composition of stocks: 2, 4, 6 and 7. This is represented visually below. (States 1, 2, 3 and 4 are represented in appendix A).

State 5
Finance IT CD Industrial

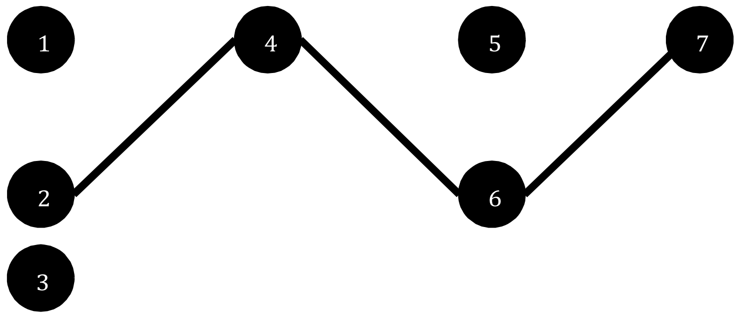


Figure 2: Goal state

Software Agents: Q-learning
Training Agents for Portfolio Stocks Selection
Aishwarya Chandra Shekhar
180017578

2.1 State Transition Function

This problem is posed using a state transition function where the agent starts from state 0 (stocks 1, 4, 5 and 7) and end at state 5 (players 2, 4, 6 and 7). The agent may only move one state at a time and, due to real life constraints, the next state must be performed by only changing one stock at a time. The below table shows the possible next states (S+1) for each current state (S).

Current state (S)	Next state (S+1)
0	1, 2, 3
1	0, 2, 4
2	0, 1, 5
3	0, 4, 5
4	1, 3, 5
5	2, 3, 4

2.2 Reward Matrix

Each state transition function is then labeled with an associated reward. In this example, a reward of 100 is applied when the agent takes an action that reaches the goal state (state 5). Any non- goal state action is associated to a reward of 0. A loop is required at the goal state to ensure that the agent remains within this state and this is applied by giving the transition from state 5 to state 5 a reward. If a state transition is not possible in one move (e.g. state 1 to state 3) a negative reward of -1 is applied. This creates the following reward matrix, R, where the rows correspond to the current state and the columns correspond to the action to the next state.

Action ➔
State 🡻

R	0	1	2	3	4	5
0	-1	0	0	0	-1	-1
1	0	-1	0	-1	0	-1
2	0	0	-1	-1	-1	100
3	0	-1	-1	-1	0	100
4	-1	0	-1	0	-1	100
5	-1	-1	0	0	0	100

Table 1: Initial R-matrix

2.3 Graphical Representation

The agent may move freely between states given that there is a possible action between states. Initial state is 0 and goal state (5) is represented in green.

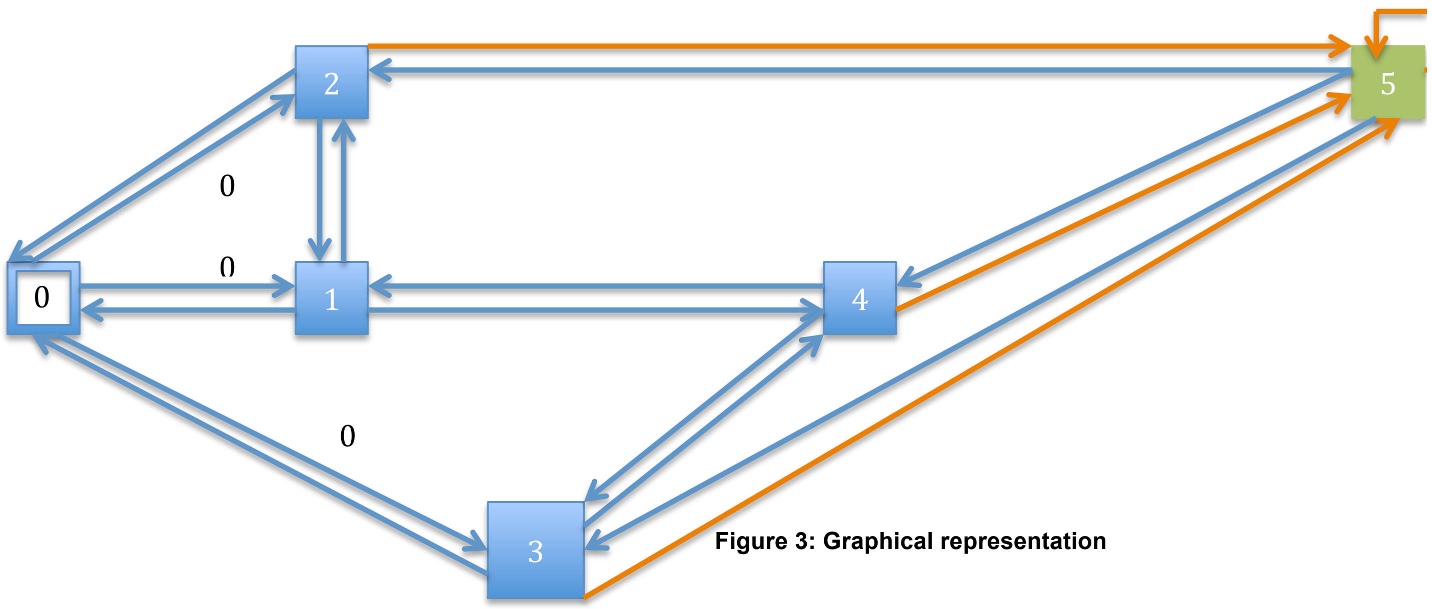


Figure 3: Graphical representation

2.4 Implementing Q-learning

Q-learning algorithm was introduced earlier and now the procedure required in obtaining the optimal path from initial state to the goal state will be outlined. This scenario is applied based on a similar example where an optimal path to reach outside through rooms is desired [4].

The scenario is defined as having an associated state transition function and reward matrix. Now a similar matrix, "Q" is added to represent the memory of what the agent has learned through experience. The rows of matrix Q represent the current state of the agent, and the columns represent the possible actions leading to the next state (the links between the nodes in graphical representation).

The agent starts out knowing nothing and is represented by the matrix Q initialized to zero. The agent then explores from state to state until the goal state is reached. Each exploration made by the agent is defined as an episode. Each episode consists of the agent moving from the initial state to the goal state. Each time the agent arrives at the goal state, the program goes to the next episode.

The agent to learn from experience uses the algorithm. Each episode is equivalent to one training session. In each training session, the agent explores the environment (represented by R matrix) and receives a reward until it reaches the goal state. The purpose of the training is to add information to agent's knowledge (represented by Q-matrix). More training/episodes help in optimizing the Q-matrix. If the Q-matrix has been optimised, the agent will converge (remain in one state) to find the fastest route to the goal state. As mentioned earlier, Q learning has proven to converge in all cases [2].

The transition rule is defined by the following formula,

$$\mathbf{Q(S, a) \leftarrow Q(S, a) + \alpha^*\{ R(S, a) + \gamma^*Max[Q(S+1, a+1)] - Q(S, a) \}} \text{ ------ Equation[1]}$$

Where S is the current state, a is the current action, S+1 is the next state and a+1 is all possible actions associated with the next state. The value of Q at state S and action '**a**' updates based on parameters alpha, α, and gamma, γ.

2.5 Parameters

The learning rate alpha, α, and discount factor Gamma, γ, are the parameters that allow the algorithm to converge. Looking at equation (1) for the transition rule, it is observed that gamma reduces the effect future rewards have on updating Q values and alpha scales the amount of new information the algorithm learns at each episode.

If Gamma is closer to zero, the agent will tend to consider only immediate rewards. If Gamma is closer to one, the agent will consider future rewards with greater weight, willing to delay the reward.

Initially, each parameter is set to a particular value then they are varied to observe the variations in results obtained. The initial parameter values are:

Alpha, $\alpha = 0.7$, Gamma, $\gamma = 0.8$

The effect of applying a discount factor or learning rate on the updated Q matrix is demonstrated first. These values are then changed to compare the results.

2.6 Finding the optimal path with the Q matrix using greedy policy

For an epsilon-greedy policy, once the Q matrix has been computed, the algorithm uses the path with the actions that have the highest reward values recorded in matrix Q for current state. In other words: 1) Set current state = initial state, 2) From current state, find the action with the highest Q value, 3) Set current state = next state, 4) Repeat Steps 2 and 3 until current state = goal state.

The algorithm returns the sequence of states from the initial state to the goal state. The policy is varied (from greedy to random) and the resulting paths obtained are compared. The parameters associated are then varied with the epsilon greedy policy.

3. Application and analysis of agent on the simple example

In this example, the rewards matrix considered is shown in section 2.2 and therefore a similar Q matrix of zeros is initialised.

Action ➔

State ⬇	Q(0)	**0**	**1**	**2**	**3**	**4**	**5**
	0	0	0	0	0	0	0
	1	0	0	0	0	0	0
	2	0	0	0	0	0	0
	3	0	0	0	0	0	0
	4	0	0	0	0	0	0
	5	0	0	0	0	0	0

Table 2: Initial Q-matrix

The previously stated rules are applied to update the Q matrix with each episode. All the calculations for first 2 episodes were performed by hand using simple Excel formulas that correspond directly to the transition rule (equation 1). In this case, 'Select a random state' rule has been decided by human to update the Q matrix initially rather than allowing the algorithm to perform randomly. Code performing random initialisation will be applied and results will be compared to see the difference.

3.1 Calculating Q-matrix by hand

<u>Episode 1</u>

For episode 1, the agent starts at state 0, travels to state 1 then state 4 where it reaches the goal. Simply, the path is 0->2->5. The results are first shown by hand calculations and then updated Q matrix is updated.

$0 \rightarrow 2$

$Q(0,2) \leftarrow Q(0,2) + 1^* \{ R(0,2) + 1 * \max [Q(2,0), Q(2,1), Q(2,5)] - Q(0,2) \}$

At the current episode: Q(0,2) = 0, R(0,2) = 0, Q(2,0), Q(2,1) and Q(2,5) all equal to 0
Therefore, $Q(0,2) \leftarrow 0 + 0.7^* \{ 0 + 0.8 * \max [0, 0, 0] - 0\} = 0$.

Similarly, $2 \rightarrow 5$

$Q(2,5) \leftarrow Q(2,5) + 0.8^* \{ R(2,5) + 0.8 * \max [Q(5,2), Q(5,3), Q(5,4), Q(5,5)] - Q(2,5) \}$
$Q(2,5) \leftarrow 0 + 0.7^* \{100 + 0.8^* \max [0, 0, 0, 0] - 0\} = 70$.

The updated Q-matrix for first episode, Q(1) is represented below, with updated values in green.

Action ➔

State 🡻

Q(1)	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	70
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

Episode 2

For episode 2, we start again at state 0 and travel to states 2, 1, 4 and finally the goal state, 5. Simply, the path 0 → 2 → 1 → 4 → 5. Our previously introduced Q values, Q(0,2) and Q(2,5) will not continue to update. Again, calculating the path by hand, following results are obtained.

0 → 2 → 1 → 4 → 5

Q(0,2) ← 0 + 0.7 * { 0 + 0.8 * max [0, 0, 70] – 0} = 39.2

Q(2,1) ← 0 + 0.7 * { 0 + 0.8 * max [0, 0, 0] – 0} = 0

Q(1,4) ← 0 + 0.7 * {0 + 0.8 * max [0, 0, 0] – 0} = 0

Q(4,5) ← 0 + 0.7 * {100 + 0.8 * max [0, 0, 0] – 0} = 70

The updated Q matrix for the second episode, Q(2), is shown below with updated values in green and values not updated in this episode in yellow:

Action ➔

State 🡻

Q(2)	0	1	2	3	4	5
0	0	0	39.2	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	70
3	0	0	0	0	0	0
4	0	0	0	0	0	70
5	0	0	0	0	0	0

The algorithm is iterated over many episodes, until the Q-matrix converges. For this particular example, the algorithm is iterated over 50 episodes, with initial parameters as **alpha = 0.7** and **gamma = 0.8**. The following figure was obtained that demonstrates that the Return (total reward) from the Q matrix does indeed converge. It is seen that the Q matrix has stopped learning new information after about 20 episodes.

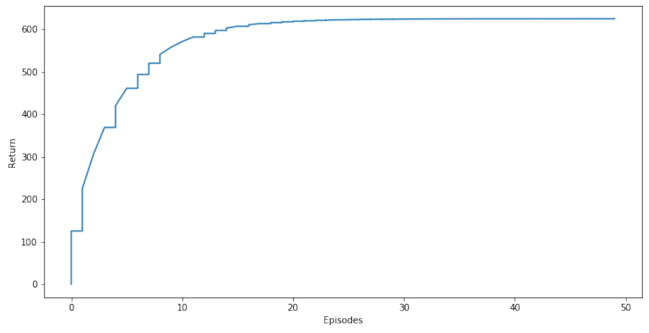
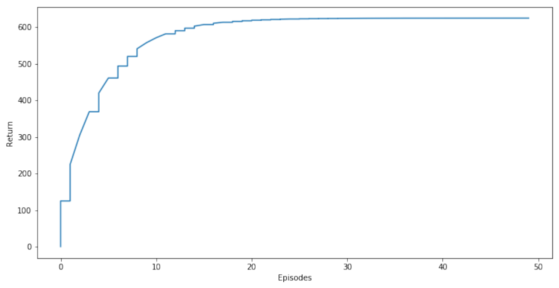
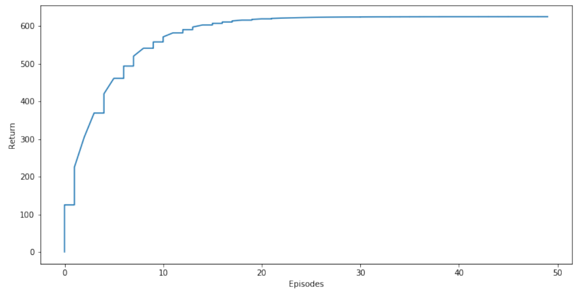


Figure 4: Performance vs. Episodes for alpha = 0.7, gamma = 0.8

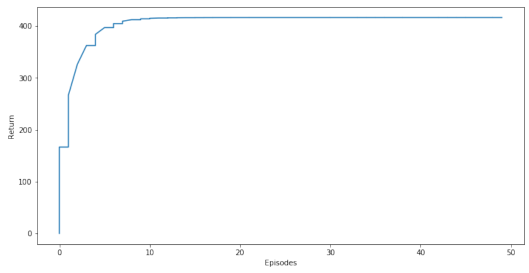
3.2 Varying the parameters

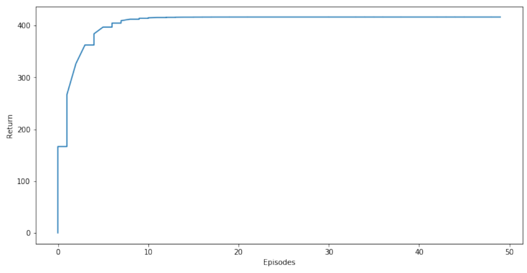
The parameters are now varied to see their impact on the results obtained. In doing so, it is attempted to further prove that Q learning can indeed be valuable to stock selection. The algorithm was run with different values of alpha and gamma. The range considered for alpha was [0.9, 0.7, 0.5, 0.3, 0.1] and for gamma was [0.8, 0.6, 0.4, 0.2]. Due to space constraints, only the graphs for alpha = [0.9, 0.7, 0.5] and gamma = [0.8, 0.6] are shown.



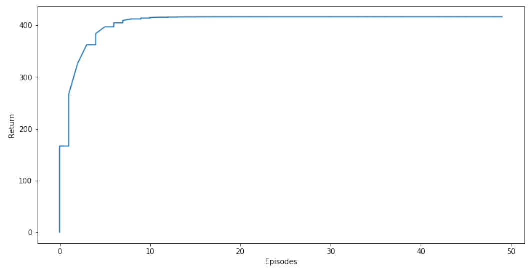


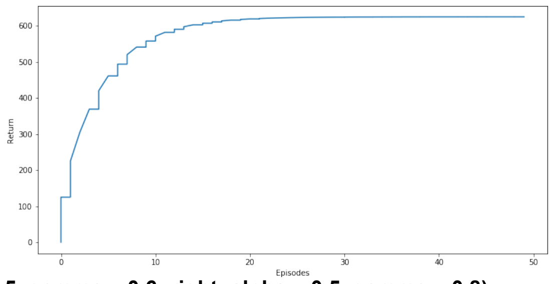
Figures 5 & 6: Performance vs. Episodes (left: alpha = 0.9, gamma = 0.8; right: alpha = 0.7 gamma = 0.8)





Figures 7 & 8: Performance vs. Episodes (left: alpha = 0.9, gamma = 0.6; right: alpha = 0.7, gamma = 0.6)





Figures 9 & 10: Performance vs. Episodes (left: alpha = 0.5, gamma = 0.6; right: alpha = 0.5, gamma = 0.8)

3.2.1 Analysis of results: From figures 5-10, it can be seen that Return and the number of episodes change with changes in alpha and gamma values. The endeavour is to find alpha and gamma values in a way that number of episodes required for convergence is reduced and the return is not sacrificed. Return of an episode is represented as:

$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^{2} * r_{t+3} + \ldots$ [R – Return (total reward), r – reward)

A high gamma value gives higher Return but also increases the number of episodes. On the other hand, a high learning rate reduces the number of episodes as it requires fewer episodes to learn.

From the graphs shown above, it can be seen that for gamma = 0.6, the Return is high and high value of alpha = 0.9 helps reduce the number of episodes to 12. Thus, the optimal values for alpha and gamma are chosen to be: **alpha = 0.9** and **gamma = 0.6.** This helps prove our hypothesis, **H2 True** that varying alpha and gamma helps improve results (in terms of reduced number of episodes).

3.2.2 Changing the decision policy from epsilon-greedy to random:

After selecting the optimal values for alpha and gamma, it is now studied whether changing the policy from epsilon-greedy (e=0) to random (e=0.5) improves the results. For this, updated Q-matrices for **alpha = 0.9** and **gamma = 0.6** after 12 episodes are compared with epsilon-greedy (e=0) and random (e=0.5) policies.

Q(12)	0	1	2	3	4	5
0	0.00	0.00	0.00	1.00	0.00	0.00
1	0.00	0.00	0.00	0.00	0.00	0.00
2	0.00	0.00	0.00	0.00	0.00	1.00
3	0.00	0.00	0.00	0.00	1.00	0.00
4	0.00	0.00	0.00	0.00	0.00	1.00
5	0.00	0.00	0.00	0.00	0.00	1.00

Tables 3: alpha = 0.9, gamma = 0.6, epsilon-greedy (e=0)

Q(12)	0	1	2	3	4	5
0	0.00	0.00	1.00	0.00	0.00	0.00
1	0.00	0.00	1.00	0.00	0.00	0.00
2	0.004	0.00	0.00	0.00	0.00	0.999
3	0.124	0.00	0.00	0.00	1.00	0.00
4	0.00	4.30e-07	0.00	0.00	0.00	0.999
5	0.00	0.00	0.00	0.00	0.00	1.00

Table 4: alpha = 0.9, gamma = 0.6, epsilon random (e=0.5)

In tables 3 and 4 above, cells highlighted in blue are the actions with the highest values for each state. The agent is given the initial state as 0 and it needs to reach the given goal state 5, in minimum number of steps. From the matrix, it can be seen that using the greedy decision policy, agent starts from state 0,travels to state 3 then to state 4 and finally to the goal state5: states 0→3→4→5→5. On the other hand, using the random policy, agent starts from state 0,travels to state 2 and from state 2, it directly reaches the goal state 5: states 0→2→5→5. It can be seen that the agent reaches the goal state in lesser number of steps when using the random policy. This can be explained as, in greedy exploitation process, agent might get trapped in an incorrect path and find it difficult to come out. In such a scenario, selecting an action at random would help agent to find a new path and hence explore other better paths. This proves our hypothesis **H3 True** that varying the policy from greedy to random improves the results. (The traversal paths for each of 6 starting states are given in the appendix B).

By this simple example, it is demonstrated that the results obtained are applicable to the selection of stocks to build an optimal portfolio and hence proves the first hypothesis, **H1 True**.

4. Application to a real life scenario

So far it has been demonstrated with simple example that valuable results can be obtained with parameter optimisation. Now it is tried to show that the learning techniques can be applied to represent a real-life scenario of portfolio selection. In the above simple example, it was shown that 4 stocks for portfolio needed to be selected from a pool of 7 stocks. In real life, there could be hundreds of stocks to select from. In such a scenario, selecting the stocks such that portfolio returns are maximised could be challenging. However, using Q-learning for such decision-making could prove to be very helpful. We consider that 4 stocks need to be selected from a pool of 10 stocks, as shown below:

Finance	IT	CD	Industrials
Stock 1	Stock 4	Stock 6	Stock 9
Stock 2	Stock 5	Stock 7	Stock 10
Stock 3		Stock 8	

In this scenario there are a total of **36 states** (3 x 2 x 3 x 2). This is 6 times the size of our simple example. In real life, the rewards depend upon the performance of a particular stock. So far, the initial and goal states were fixed but in real life this would not be the case. The trader will vary the starting selection of stocks regularly to always keep the best performing stocks in the portfolio, and the algorithm needs to accommodate this. By increasing the scope of the problem, we have shown that our fourth hypothesis, **H4**, holds **True** that we can indeed increase the number of stocks to represent a real-time portfolio selection.

In this scenario, the agent is able to start in any state and also any state could be the goal state. Therefore, rewards need to be based on stock performance. The obvious way to do this would be to predict each stock's performance in the near future. The trader can use this prediction to update the reward matrix regularly. The method for predicting the rewards is outside the scope of this study. We would rather introduce some random values, normalised between 0 and 9 (0-lowest, 9-highest), to initialise the reward matrix. (The initial reward matrix and converged Q-matrix are shown in appendix C).

4.1 Updating policy from greedy to random: Simply applying the algorithm with no end state and previously computed parameters (alpha = 0.9 and gamma = 0.6) and the greedy policy we compute the optimal paths from each initial state:
Greedy traversal for starting state 0: 0 -> 7 -> 11 -> 2 -> 22 -> 25 -> 33 -> 16 -> 17 -> 26 -> 31 -> 16 -> 17 -> 26 -> 31 -> 16 -> 17 -> 26 -> 31 -> 16 -> 17
Greedy traversal for starting state 1: 1 -> 10 -> 11 -> 2 -> 22 -> 25 -> 33 -> 16 -> 17 -> 26 -> 31 -> 16 -> 17 -> 26 -> 31 -> 16 -> 17 -> 26 -> 31 -> 16 -> 17
Greedy traversal for starting state 2: 2 -> 22 -> 25 -> 33 -> 16 -> 17 -> 26 -> 31 -> 16 -> 1
7 -> 26 -> 31 -> 16 -> 17 -> 26 -> 31 -> 16 -> 17 -> 26 -> 31 -> 16, and so on.

By updating the learning policy from greedy to one that take random states, i.e. by setting epsilon = 0.5, the following optimal paths for each initial state are computed:
Greedy traversal for starting state 0: 0 -> 9 -> 6 -> 0 -> 9 -> 6 -> 0 -> 9 -> 6 -> 0 -
> 9 -> 6 -> 0 -> 9 -> 6 -> 0 -> 9 -> 6 -> 0 -> 9 -> 6
Greedy traversal for starting state 1: 1 -> 0 -> 9 -> 6 -> 0 -> 9 -> 6 -> 0 -> 9 -> 6 -
> 0 -> 9 -> 6 -> 0 -> 9 -> 6 -> 0 -> 9 -> 6 -> 0 -> 9
Greedy traversal for starting state 2: 2 -> 26 -> 0 -> 9 -> 6 -> 0 -> 9 -> 6 -> 0 -> 9 -> 6 -> 0 -> 9 -> 6 -> 0 -> 9 -> 6 -> 0 -> 9 -> 6 -> 0, and so on.

It can be seen that by applying **epsilon = 0.5**, the number of steps converge sooner, as compared to greedy policy. Updating policy to 0.5 means that the highest reward is picked with 50% probability and therefore results become probabilistic. The reason that this is important is that with so many states, there is a large possibility of the agent getting trapped in the wrong states. By helping it to move away from this trap, agent is more likely to reach the goal state given enough time. From section 3.2.2 above, and from the traversal paths shown above, it can be said with conviction that varying the policy from greedy to random helps improve agent's performance, by reducing the number of steps needed to reach the goal state.

4.2 Updating the reward matrix in real life scenario: Currently, the reward matrix has no goal state. If the trader knows the 4 stocks that she wishes to keep in the portfolio, then the reward matrix would update accordingly. Every action that gets us to the desired end state will have reward updated to the highest reward value. If a stock starts performing bad, due to some sudden market crisis, then the trader would provide this feedback to the system and it would update the reward values accordingly to remove any actions that transition into a state with the poor performing stock.

Suppose we wish to select the portfolio that is at state 30 as per our model processing, we would update the reward matrix accordingly and the agent would calculate the optimal paths from any current state to goal state 30. Thus, in this way, the reward matrix can be updated to not only handle the stock's performance scores, but also accommodate any real-time changes in case of market crisis. This proves our fifth hypothesis, **H5** to hold **True**. Thus, we have proved all 5 of our hypothesis.

In future, more robust multi-agent algorithms could be employed, such that stock's performance could also be predicted and optimal paths could be found accordingly.

5. Establishing parallelisms between Q-learning algorithm and error correction models in psychology

An **e**rror **c**orrection **m**odel belongs to a category of multiple time series models most commonly used for data where the underlying variables have a long-run stochastic trend, also known as cointegration. ECMs are a theoretically driven approach useful for estimating both short-term and long-term effects of one time series on another. The term error-correction relates to the fact that last-period's deviation from a long-run equilibrium, the *error*, influences its short-run dynamics. Thus ECMs directly estimate the speed at which a dependent variable returns to equilibrium after a change in other variables. [5]

According to [6], Pavlov, in his famous experiment on the salivating dog, observed that if one rings a bell and follows that bell with food, dogs become conditioned to salivate after the bell is rung. This is because they related ring of the bell to the reward they received in the near past (food, in this case). This idea was mathematically formalized by Bush and Mosteller when they proposed that the probability of Pavlov's dog expressing the salivary response on sequential trials could be computed through an iterative equation where,

$$A_{next\_trial} = A_{last\_trial} + \alpha(R_{current\_trial} - A_{last\_trial}) \text{........Equation [2]}$$

In equation (2), Anext_trial is the probability that the salivation will occur on the next trial (ringing of bell). Anext_trial is computed based on the value of A on the previous trial and adds to it a correction based on the animal's experience during the most recent trial. This correction, or error term, is the difference between what the animal actually experienced (in this case, the reward of the meat powder expressed as Rcurrent_trial) and what he expected (simply, what A was on the previous trial). The difference between what was obtained and what was expected is multiplied by α, a number ranging from 0 to 1, which is known as the learning rate. The learning rate defines the rate at which the error is corrected. A high α, immediately updates R from the last trial. When the value of α is small, A is incremented very slowly to the value of R.
Bush and Mosteller equation computes an average of previous rewards across previous trials. In this average, the most recent rewards have the greatest impact, whereas rewards far in the past have only a weak impact. The iterative equation reflects a weighted sum of previous rewards.

$$A_{now} = 0.5R_{now} + 0.25\, R_{t-1} + 0.125R_{t-2} + 0.625R_{t-3} \text{ .... Equation [3]}$$

This represents an exponential series and the rate at which the weight declines is controlled by α. The Rescorla–Wagner model is an important extension of the Bush and Mosteller approach to the study of what happens to associative strength when two cues predict the same event. Their findings were so influential that the basic Bush and Mosteller rule is now often mistakenly attributed to Rescorla and Wagner.

The same concept of error correction is applied in Q-learning algorithm of Reinforcement learning. In Q-learning algorithm, the Q-value is an action value that represents expected discounted reward for executing action 'a' at state 'S'. The Q-values are updated using the following formula, equation [1] from above:

$$\mathbf{Q(S, a) \leftarrow Q(S, a) + \alpha^*\{ R(S, a) + \gamma^*Max[Q(S+1, a+1)] - Q(S, a) \}} \text{ ...Equation [1]}$$

Where, the part highlighted in green is the new Q value, the part highlighted in yellow is old Q value and grey part is the error value that is multiplied by **α** for correction. If we compare this equation with equation 2 above, we can see that green part represents A_{next_trial}, yellow part represents A_{last_trial} and grey part represents $(R_{current\_trial} - A_{last\_trial})$.

6. How error correction models could be implemented as (advanced) reinforcement learning architectures

To demonstrate how error correction models can be implemented in advanced reinforcement learning architectures, we represent Deep Q Networks that combine Q algorithm with neural networks to produce more efficient results. We can treat neural networks as the agent that learns to map state-action pairs to rewards. Like all neural networks, they use coefficients to approximate the function relating the inputs to outputs, and their learning aims to find the right coefficients, or weights, by iteratively adjusting those weights along gradients to promise less error. [7]

In reinforcement learning, convolutional networks can be used to recognize an agent's state. At the beginning of reinforcement learning, the neural network coefficients may be initialized stochastically. Using feedback from the environment, the neural net can use the difference between its expected reward and the ground-truth reward to adjust its weights and improve its interpretation of state-action pairs. This feedback loop is analogous to error correction model explained in section 5.
Reinforcement learning relies on the environment to send it a scalar number in response to each new action. The rewards returned by the environment can be varied, delayed or affected by unknown variables, introducing noise to the feedback loop.
This leads us to a more complete expression of the Q function, which takes into account not only the immediate rewards produced by an action, but also the delayed rewards that may be returned several time steps deeper in the sequence. This is analogous to error correction model proposed by Bush and Mosteller in equation 3 above.
Thus, it could be seen that error correction models could be implemented in advanced Reinforcement learning algorithms to improve the accuracy of results, as these models aim to reduce the losses that deviate the predicted value from the expected value.

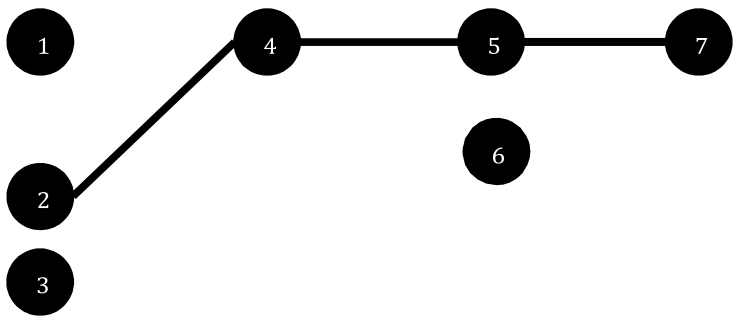
References

[1] Watkins, C.J.C.H., (1989), Learning from Delayed Rewards. Ph.D. thesis, Cambridge University
[2] Watkins and Dayan, C.J.C.H., (1992), 'Q-learning.Machine Learning'
[3] Markowitz, H. (1952). PORTFOLIO SELECTION*. *The Journal of Finance*, 7(1), pp.77-91.
[4] Tutorial on Q learning: http://mnemstudio.org/path-finding-q-learning-tutorial.htm
[5] En.wikipedia.org. (2019). *Error correction model*. [online] Available at: https://en.wikipedia.org/wiki/Error_correction_model [Accessed 4 Apr. 2019].
[6] Glimcher, P. (2011). Understanding dopamine and reinforcement learning: The dopamine reward prediction error hypothesis. *Proceedings of the National Academy of Sciences*, 108(Supplement_3), pp.15647-15654.
[7] Skymind. (2019). *A Beginner's Guide to Deep Reinforcement Learning*. [online] Available at: https://skymind.ai/wiki/deep-reinforcement-learning [Accessed 4 Apr. 2019].
[8] Python code used:
https://gist.github.com/kastnerkyle/d127197dcfdd8fb888c2

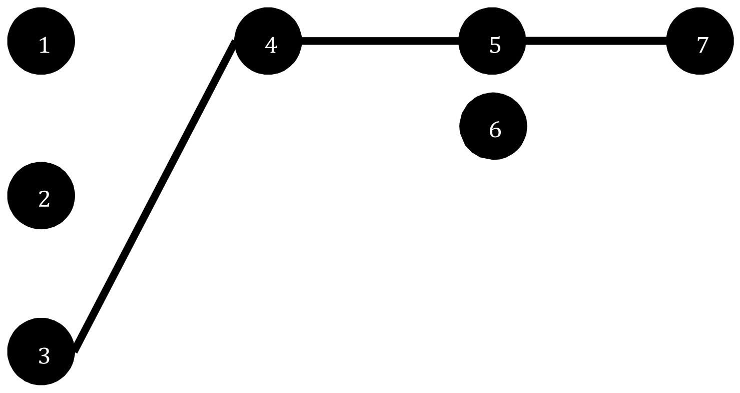
Software Agents: Q-learning
Training Agents for Portfolio Stocks Selection
Aishwarya Chandra Shekhar
180017578

Appendix A (States 1 to 4 of simple example)

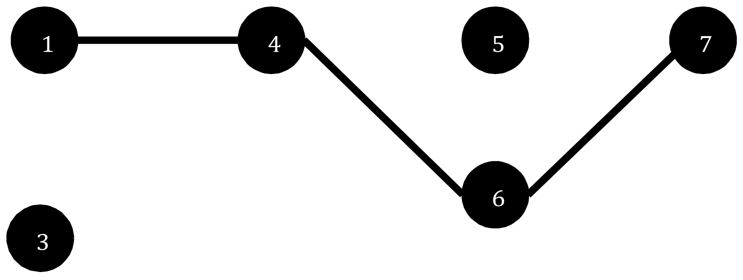
<u>State 1:</u> Stocks 2, 4, 5 and 7



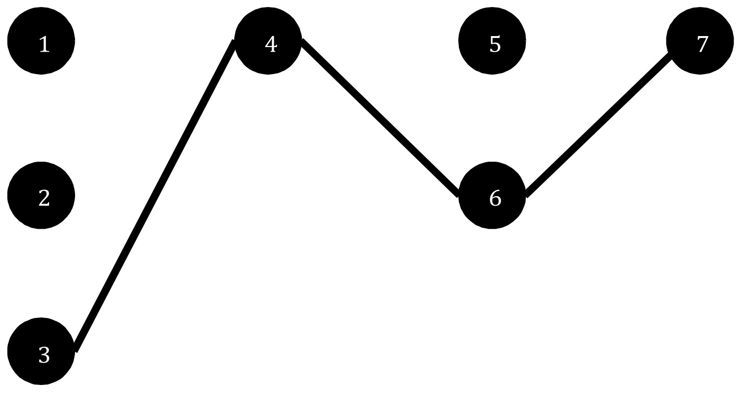
<u>State 2:</u> Stocks 3, 4, 5, and 7



<u>State 3:</u> Stocks 1, 4, 6, and 7



<u>State 4</u> Players 3, 4, 6, and 7



Appendix B

Traversal paths for all states for simple example with alpha = 0.9, gamma = 0.6, e = 0.5

```
Greedy traversal for starting state 0 0 -> 2 -> 5
Greedy traversal for starting state 1 1 -> 2 -> 5
Greedy traversal for starting state 2 2 -> 5
Greedy traversal for starting state 3 3 -> 4 -> 5
Greedy traversal for starting state 4 4 -> 5
Greedy traversal for starting state 5 5
```

Traversal paths for all states for simple example with alpha = 0.9, gamma = 0.6, e = 0

```
Greedy traversal for starting state 0 0 -> 3 -> 4 -> 5
Greedy traversal for starting state 1 1 -> 0 -> 3 -> 4 -> 5
Greedy traversal for starting state 2 2 -> 5
Greedy traversal for starting state 3 3 -> 4 -> 5
Greedy traversal for starting state 4 4 -> 5
Greedy traversal for starting state 5 5
```

Appendix C (Initial R-matrix for real life scenario, 36 x 36)

```
[5 2 3 6 1 5 0 6 4 8 3 2 5 5 5 0 2 2 8 4 0 3 2 7 9 0 2 5 7 2 5 8 5 1 8 9]
[8 5 9 3 1 8 0 1 8 3 0 5 6 8 0 1 7 2 1 3 0 3 0 1 7 2 5 5 1 3 6 2 9 0 1 7]
[0 2 3 4 7 3 6 0 8 2 3 6 3 2 7 5 0 5 4 2 4 0 1 9 9 4 1 8 6 9 6 0 3 6 2 6]
[3 3 3 2 5 5 0 0 7 1 1 7 3 8 5 3 7 3 3 3 3 4 7 1 3 7 4 7 2 1 1 2 0 4 1 1]
[5 2 2 0 6 9 1 7 5 1 3 4 5 1 5 1 0 5 8 3 0 7 9 5 1 4 8 7 7 2 2 9 6 1 2 0]
[2 8 3 8 7 7 3 5 2 1 7 4 8 0 6 6 2 0 1 6 6 2 8 3 2 8 2 3 6 5 6 4 8 0 2 3]
[2 7 2 3 3 2 1 6 2 5 0 6 6 2 3 6 2 1 1 8 8 8 6 3 7 0 4 3 0 6 2 8 7 0 7 3]
[0 9 3 4 2 4 8 7 8 9 6 4 9 1 4 4 8 4 0 6 9 7 9 9 6 9 5 9 2 1 4 1 9 8 4 0]
[1 5 9 5 4 3 7 7 8 4 6 6 1 2 0 1 0 8 6 6 8 4 5 0 9 7 8 8 3 0 9 8 7 1 2 9]
[7 7 9 1 9 9 5 0 1 7 9 7 2 5 0 6 8 3 0 0 2 8 0 3 9 2 9 3 2 6 5 5 1 0 1 5]
[4 4 8 4 3 7 2 8 2 7 9 1 4 6 7 6 9 2 0 5 6 1 5 8 1 8 9 5 5 7 7 2 8 0 7 0]
[1 6 1 4 4 0 4 7 3 5 5 4 0 3 3 7 6 2 2 0 3 5 7 2 1 3 1 1 7 4 0 6 1 4 4 0]
[2 7 2 7 8 2 2 4 4 9 3 8 6 4 4 8 7 0 3 7 2 9 8 9 4 5 8 9 7 1 5 6 7 6 0 0]
[4 4 6 2 8 9 7 2 0 1 1 0 0 2 3 8 4 9 1 9 0 4 6 0 9 3 0 4 6 2 9 4 7 2 0 2]
[3 4 5 1 5 4 3 4 5 3 3 0 4 8 8 5 9 9 7 1 8 3 0 8 4 9 6 4 5 4 7 0 7 5 8 4]
[5 9 3 3 3 5 1 3 7 5 0 5 7 2 9 8 4 2 2 7 8 7 2 4 8 3 0 7 9 7 5 4 3 1 0 7]
[8 4 7 1 4 3 7 1 3 1 8 8 6 2 0 7 9 4 6 6 1 3 6 0 5 5 7 6 9 5 7 4 9 9 0 6]
[3 7 9 2 3 0 8 3 1 7 3 9 6 3 8 3 3 3 5 6 8 1 3 4 2 2 3 2 8 1 8 0 9 0 9 2]
[7 0 2 1 3 7 5 2 0 8 6 0 1 3 0 7 7 0 5 3 1 3 1 6 8 8 7 5 9 5 7 5 5 0 5 7]
[4 6 5 7 1 9 2 5 0 1 3 8 9 3 2 4 7 5 3 7 3 2 7 3 0 9 3 0 2 5 1 9 9 3 9 0]
[6 0 9 0 4 6 0 0 3 1 9 0 6 8 1 2 4 5 2 3 7 1 6 2 1 6 5 4 6 2 2 0 9 6 3 7]
[2 1 5 2 0 8 5 6 3 4 2 7 6 4 4 1 3 1 0 1 5 1 6 7 5 4 7 4 1 1 5 5 1 7 0 8]
[6 8 1 2 4 9 0 9 5 4 7 8 8 8 2 3 1 8 3 2 0 2 8 0 3 5 9 4 2 3 7 5 7 4 8 7]
[9 3 6 7 5 6 7 7 6 4 5 0 2 8 5 3 5 2 3 5 6 7 2 4 2 8 2 3 6 2 2 4 2 0 7 6]
[7 3 0 5 1 6 1 0 9 0 1 0 6 4 3 5 3 5 7 7 7 1 6 3 2 7 4 5 5 3 4 5 0 6 0 0]
[6 0 6 1 5 0 7 1 3 5 1 3 3 9 3 3 2 4 5 0 9 0 1 2 2 6 4 1 8 1 6 5 1 9 0 7]
[1 2 7 1 4 0 8 7 4 0 9 5 2 4 9 5 1 7 2 8 2 2 9 2 7 2 4 7 4 1 9 9 0 5 9 8]
[6 0 1 2 5 0 8 5 6 7 5 5 8 2 8 1 2 1 0 0 1 1 3 8 5 4 4 2 0 4 8 2 7 0 9 0]
[0 2 4 2 3 3 7 7 3 7 3 5 5 1 9 3 6 7 8 9 5 7 5 8 5 9 8 4 3 7 9 9 9 4 5 3]
[5 2 2 9 8 6 6 7 0 5 0 2 0 4 3 9 6 8 0 9 7 8 9 6 3 3 6 7 8 9 9 6 6 0 0 6]
[4 1 8 6 8 5 9 7 2 3 4 8 6 7 7 4 1 9 3 7 4 0 5 6 1 8 6 4 0 8 0 5 7 9 0 4]
[1 8 9 2 7 3 7 0 3 6 2 2 8 4 6 9 2 3 3 2 4 0 2 1 6 3 9 4 3 1 9 9 3 4 1 0]
[3 7 9 2 7 1 1 3 5 7 4 8 8 4 6 8 6 3 3 9 7 3 5 4 6 1 4 5 6 6 1 3 6 0 5 2]
[0 1 0 2 5 2 9 0 6 3 8 3 8 5 4 5 2 9 9 9 3 8 1 4 9 5 6 8 4 8 4 6 9 5 0 1]
[2 8 9 7 4 5 0 3 7 6 0 0 3 3 7 4 0 1 7 8 9 4 2 9 0 9 8 5 2 9 2 0 7 5 7 7]
[9 2 6 9 4 2 2 8 1 1 8 3 1 5 3 6 9 7 4 3 5 2 2 6 6 8 1 0 0 2 1 4 3 2 6 0]
```

Appendix D (Converged Q-matrix for real life scenario, 36 x 36)

```
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```