A picture containing text

Description automatically generated

**CZ3005 ARTIFICIAL INTELLIGENCE**

**LAB EXERCISE 2**

**LAB REPORT**

Shearman Chua Wei Jie (U1820058D)

LAB GROUP: TSP1

Contents

[Treasure Hunting in a Cube 3](#_Toc53946322)

[Solving the MDP using Reinforcement Learning 3](#_Toc53946323)

[Q-Learning Reinforcement Learning Algorithm 3](#_Toc53946324)

[Implementation of Q-Learning algorithm in Python 4](#_Toc53946325)

[Q Agent Class 4](#_Toc53946326)

[Test Cube Function 6](#_Toc53946327)

[Results of the Q-Learning algorithm (500 episodes, 500 steps) 7](#_Toc53946328)

[Final Q-table 9](#_Toc53946329)

[Quality of the Final Solution 13](#_Toc53946330)

[Effects of Changing Q-Learning Agent Parameters 16](#_Toc53946331)

[Tuning of Individual Parameters 16](#_Toc53946332)

[Learning Rate 16](#_Toc53946333)

[Discount Factor 18](#_Toc53946334)

[Exploration Rate 20](#_Toc53946335)

[Training and Testing of Q-Learning Agent with Optimal Parameters 21](#_Toc53946336)

[Decaying Learning Rate and Exploration Rate 23](#_Toc53946337)

[Test Results 24](#_Toc53946338)

[Q-Learning Agent with no Exploration Rate and a Decaying Learning Rate 24](#_Toc53946339)

[Q-table 26](#_Toc53946340)

[Conclusion 27](#_Toc53946341)

# Treasure Hunting in a Cube

The environment is a 3D grid world. The MDP formulation is described as follows:

* State: a 3D coordinate, which indicates the current position where the agent is. The initial state is (0, 0, 0) and there is only one terminal state: (3,3,3).
* Action: The action space is (forward, backward, left, right, up, down). The agent needs to select one of them to navigate in the environment.
* Reward: The agent will receive 1 reward when it arrives at the terminal states, or otherwise receive -0.1 reward.
* Transition: The intended movement happens with probability 0.6. With probability 0.1, the agent ends up in one of the state’s perpendicular to the intended direction. If a collision with a wall happens, the agent stays in the same state.

# Solving the MDP using Reinforcement Learning

## Q-Learning Reinforcement Learning Algorithm

For this “Treasure Hunting in a Cube” MDP, we will be using the Q-Learning algorithm to solve the problem. Q-learning is a model-free reinforcement learning algorithm to learn quality of actions telling an agent what action to take under what circumstances. For any finite Markov decision process, Q-learning finds an optimal policy in the sense of maximizing the expected value of the total reward over any and all successive steps, starting from the current state. Q-learning can identify an optimal action-selection policy for any given finite Markov decision process, given infinite exploration time and a partly random policy.

When Q-learning is performed, we create what is called a Q-table that follows the shape of [state, action] and we initialize our values to zero. We then update and store our Q-values after an episode. This Q-table becomes a reference table for our agent to select the best action based on the Q-value.

An agent interacts with the environment in 1 of 2 ways. The first is to use the Q-table as a reference and view all possible actions for a given state. The agent then selects the action based on the max value of those actions. This is known as **exploiting** since we use the information we have available to us to make a decision.

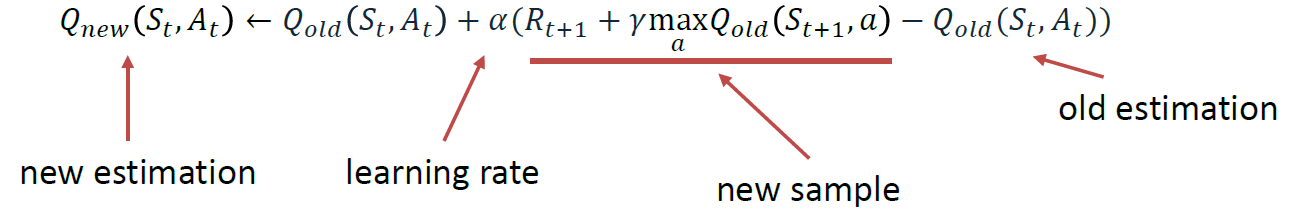
The second way to take action is to act randomly. This is called **exploring**. Instead of selecting actions based on the max future reward we select an action at random. Acting randomly is important because it allows the agent to explore and discover new states that otherwise may not be selected during the exploitation process. We can balance exploration/exploitation using exploration rate (ε) and setting the value based on how often we want the agent to explore vs exploit.

Here are the 3 basic steps of the Q-Learning algorithm:

1. Agent starts in a state (s1) takes an action (a1) and receives a reward (r1)
2. Agent selects action by referencing Q-table with highest value (max) OR by random (epsilon, ε)
3. Update Q-values

The Q-values updates occur after each step or action and ends when an episode is done. Done in this case means reaching some terminal point by the agent. The agent will not learn much after a single episode, but eventually with enough exploring (steps and episodes) it will converge and learn the optimal Q-values.

The core of the algorithm is an equation as a simple value iteration update, using the weighted average of the old value and the new information:



At every step, we adjust our Q-values based on the difference between the discounted new values and the old values. We discount the new values using discount factor (γ) and we adjust our step size using learning rate (α).

**Learning Rate**: **lr** or learning rate, often referred to as alpha or α, can simply be defined as how much you accept the new value vs the old value. Above we are taking the difference between new and old and then multiplying that value by the learning rate. This value then gets added to our previous Q-value which essentially moves it in the direction of our latest update.

**Discount factor**: gamma or γ is a discount factor. It is used to balance immediate and future reward. From our update rule above you can see that we apply the discount to the future reward. Typically, this value can range anywhere from 0.8 to 0.99.

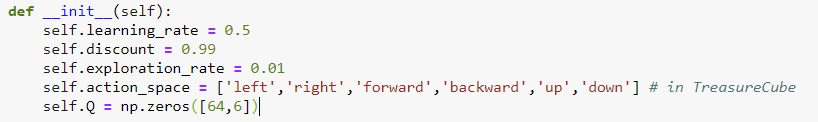
**Reward**: reward is the value received after completing a certain action at a given state. A reward can happen at any given time step or only at the terminal time step.

## Implementation of Q-Learning algorithm in Python

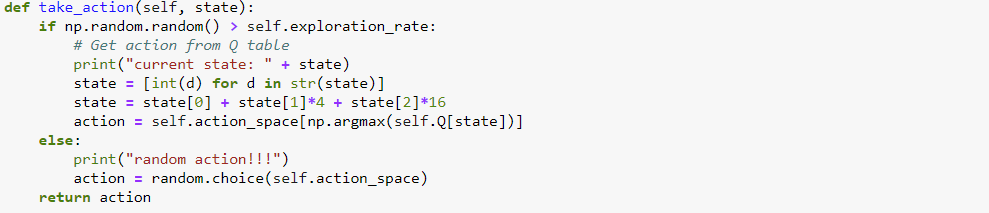
### Q Agent Class



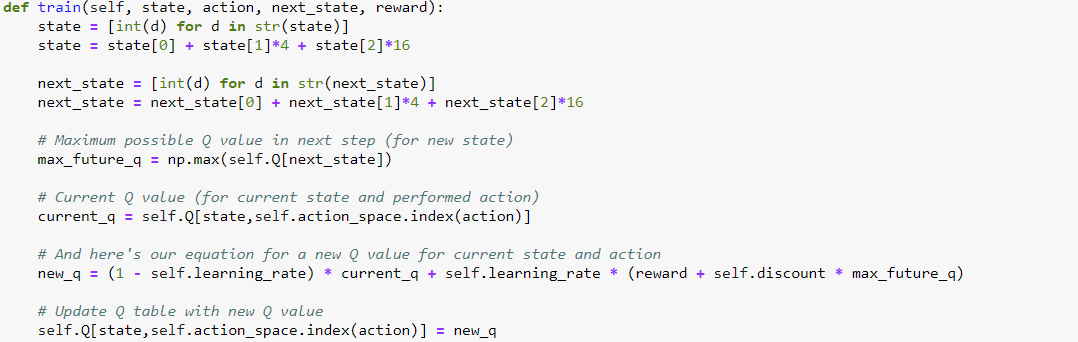
In the code, we first define the class and its functions for our Q-Learning agent. When the Agent is instantiated, its Learning Rate, Discount Factor, Exploration Rate are set based on the given parameters given in the assignment instruction, discount factor = 0.99, learning rate = 0.5, exploration rate = 0.01. We also initialize a 64 by 6 numpy array with all its values set to zeros.



Next, the **take\_action()** function of the Agent is defined so that it takes in the state the agent is currently in as an argument, and takes the action with the highest Q-value in the Q-table with reference to the state the agent is in. However, as can be seen from the function, there is a chance that the Agent will take a random action if the random number between 0 to 1 generated by the np.random.random() function generates a value smaller than the Agent’s exploration rate given by 0.01.



Lastly, the most important function for the Agent class is the **train()** function. This function takes in the state the agent was in, the selected action taken by the Agent based on the **take\_action()** function, the next state the Agent is in by taking the selected action, as well as the reward that was received by moving to the next state. Based on the parameters that was passed into the function, we make use of the Q-Learning update equation to update the Q-value of the state the Agent took the agent from. The new Q-value is (1 – learning rate) of the current Q-value of the state, added with the learning rate multiplied with the sum of the reward for moving to the next state and the maximum Q-value of the next state multiplied by the Discount Factor of the Agent given by 0.99.



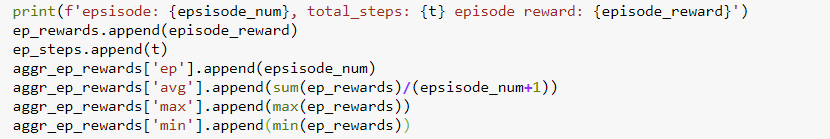
### Test Cube Function



The **test\_cube**() function is used to allow the Q-Learning Agent to interact with the “Treasure Hunting in a Cube” environment that is implemented in the **environment.py** python file. The **test\_cube**() function takes in 2 arguments, max\_episode and max\_step, where max\_episode is the maximum number of episodes you want the Q-Learning Agent to run and learn from the environment and max\_step is the maximum number of steps in each episode you want the Agent to take before terminating, in case for an episode, the Agent does not find the terminal state and is not able to automatically terminate. In the **test\_cube**() function, we instantiate a Q-Learning Agent object to learn from the “Treasure Hunting Cube” environment.

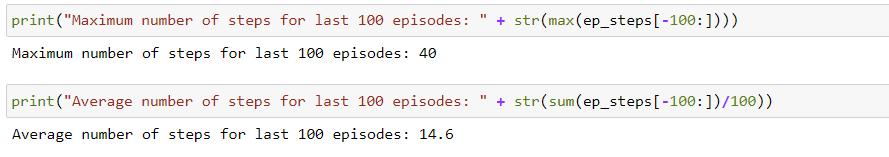
The for loop in the function runs for the number of episodes that was set by max\_episode, and for each episode, the environment is first reset to the initial state and the **state** variable is set as the environment reset state (state 0,0,0), with the **terminate** variable set to false. Then, while the episode is not terminated, the function will set the **action** variable to the action returned by the Q-Learning Agent’s **take\_action()** function, passing in the **state** variable to it. The function then passes the action to the **env.step()** function which will return the reward for taking the action given by the Q-Learning Agent, update the **terminate** variable, as well as the next state in the environment after taking the action. The previous state, action taken by the Q-Learning Agent, the new state in the environment after taking the action, as well as the reward for taking the action is then passed into the Q-Learning Agent’s **train()** function to update the Q-table values of the Agent, so that the Q-Learning Agent can learn based on the reward it was given from the action that it took. The while loop will keep running until either the maximum number of steps stated by max\_step is reached or the Agent has reached the terminal state, ending the episode. The for loop will run until the maximum number of episodes is reached.

In order to track how well the Q-Learning Agent learn throughout the episodes that is was being run, the **ep\_rewards** array, the **aggr\_ep\_rewards** array, as well as the **ep\_steps** array were created to keep track of the history of the accumulative rewards received in each episode, the average reward received by taking the average of the current episode’s reward as well as the previous episode rewards, the maximum reward received for the episodes that was ran, the minimum reward received for the episodes that was ran, as well as the number of steps taken by the Q-Learning Agent to find the terminal state in each episode.

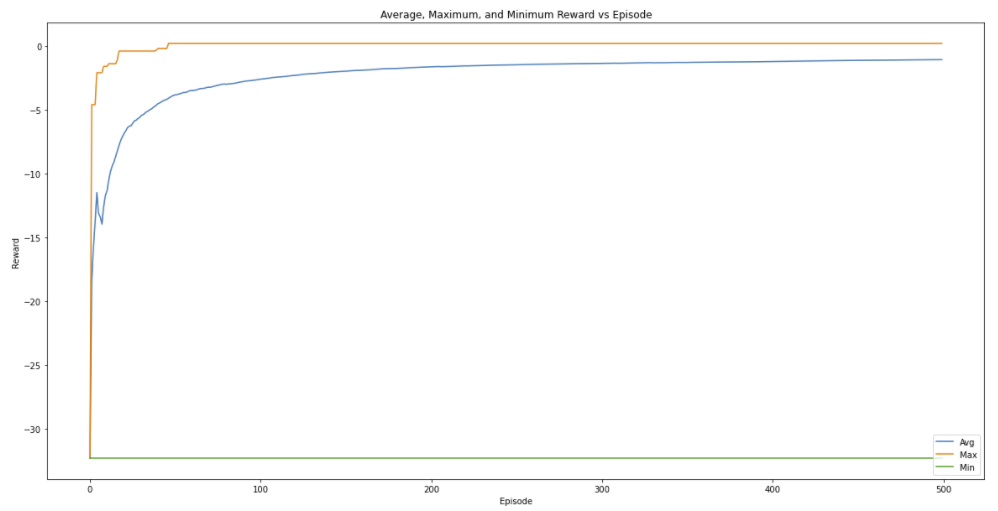


## Results of the Q-Learning algorithm (500 episodes, 500 steps)

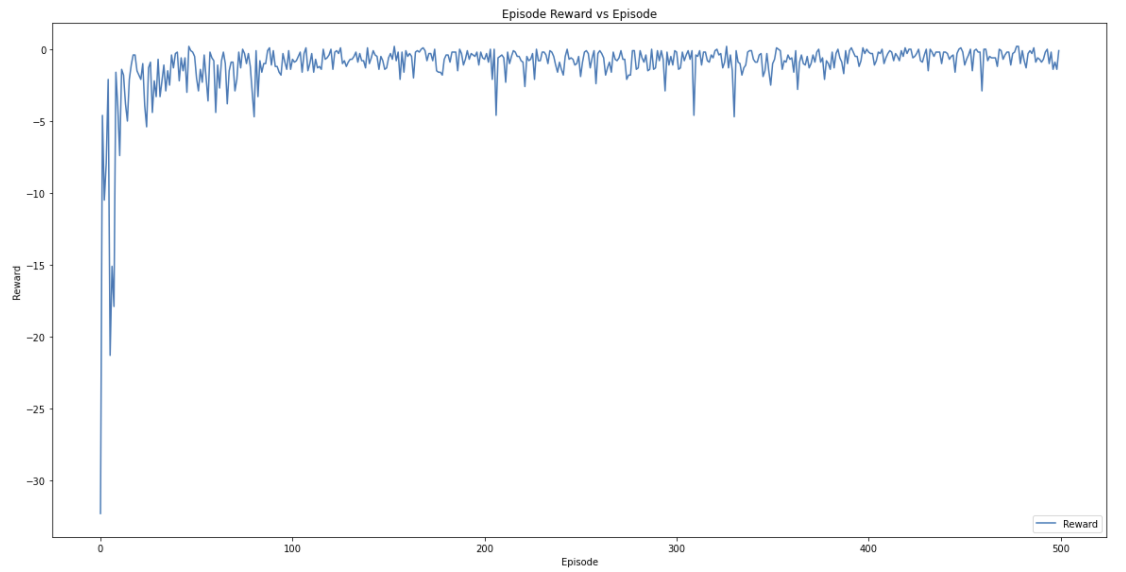
The Q-Learning algorithm was ran for 500 episodes with the maximum number of steps set to 500 as well. In order to determine how well the Q-Learning Agent was performing for the last 100 episodes, we displayed the maximum number of steps taken for the Q-Learning Agent for the last 100 episodes that was ran, as well as taking the average of the number of steps that the Q-Learning Agent took for the last 100 episodes. The best average number of steps taken for the Q-Learning Agent for the last 100 that was being run was around 14 steps.



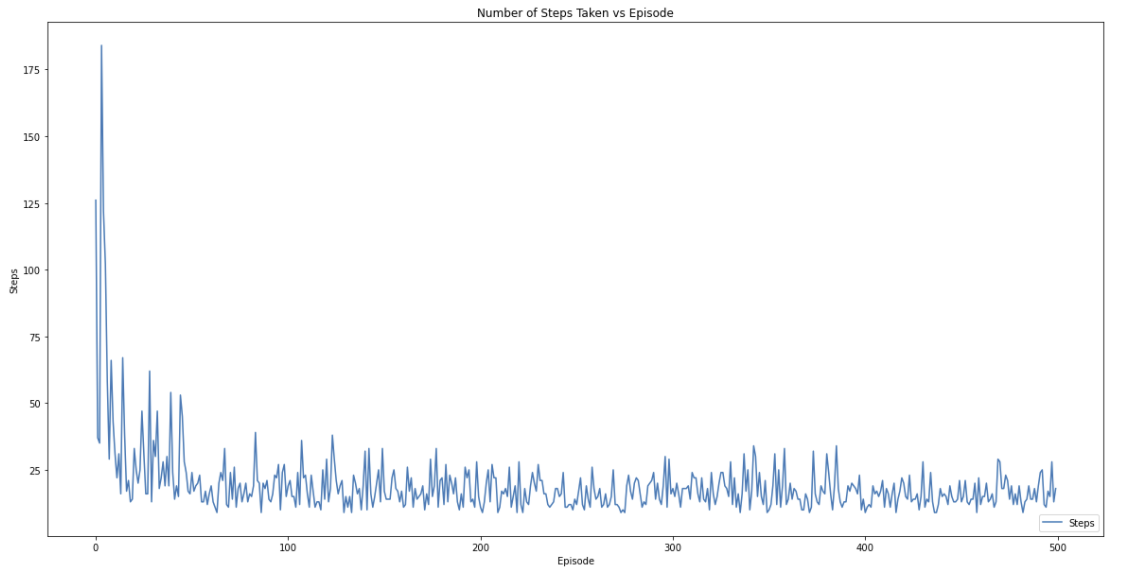
We then plot the graph of the Average Reward received, Maximum Reward received, as well as the Minimum Reward received for each episode for all of the 500 episodes. We can see that a Q-Learning Agent learns very quickly, with the Maximum reward converging at about after 40 episodes and the Average reward converging at about 300 episodes. The Minimum reward converges the fastest, which means the Q-Learning Agent adapt very quickly to avoid taking actions which results in the least reward.



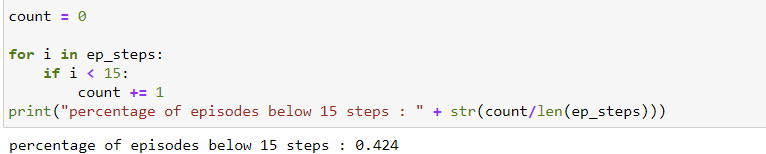
Next, we plot the graph of the Episode reward against the Episode for each of the 500 episodes. From the graph, we can see that the Episode reward received by each Episode converges very fast, with the rewards being fairly consistent after training the Q-Learning Agent for about 100 episodes. This means that we do not actually have to train the Q-Learning Agent for too many episodes beyond 100 episodes as any further training of the Q-Learning Agent will not yield much improvement to the Q-table values.

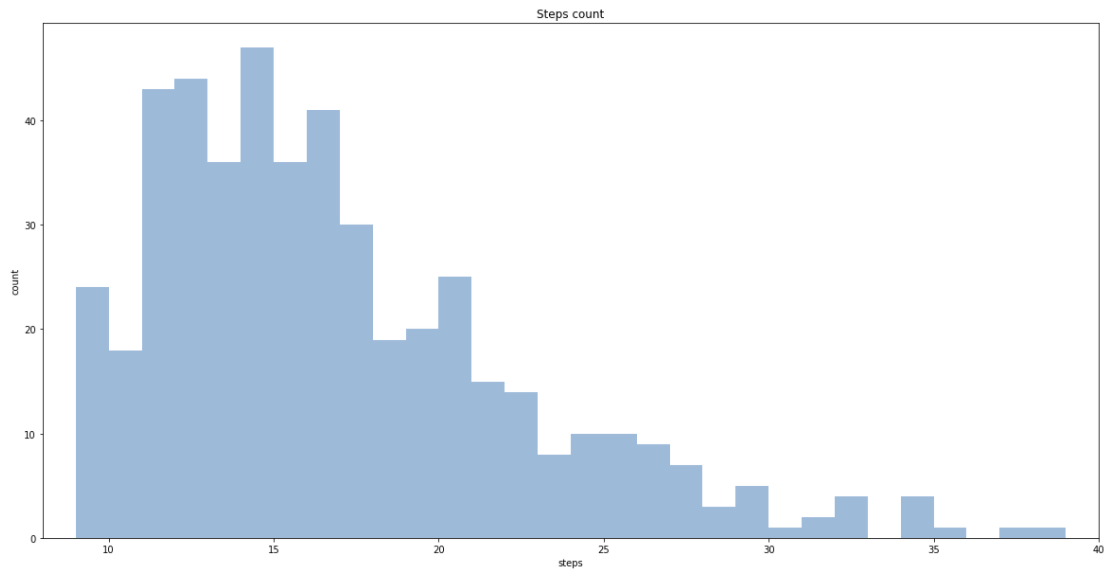


The graph below shows the Number of Steps Taken for each episode for all the 500 episodes of training the Q-Learning Agent. Similar to the Episode reward against the Episode graph, it can be seen that the Q-Learning Agent learns very quickly, with the Number of Steps Taken for each episode also converging after 100 episodes. Any increase in the number of episodes to train the Q-Learning Agent after 100 episodes will not give rise to a significantly better result, as the Number of Steps Taken for each episode is already convergent.



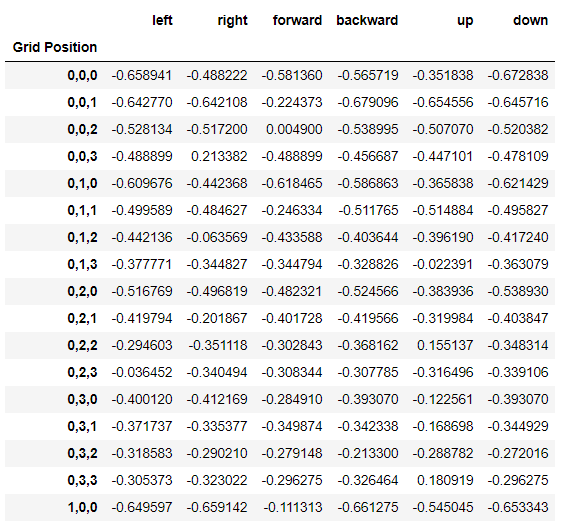
In order to determine the quality of the solution after 500 episodes of training the Q-Learning Agent, we decided to find the percentage of the number of episodes where the Q-Learning Agent took less than 15 steps for the 500 episodes that was run. We also plot the histogram of the count of the number of episodes for each step count for steps between 9 and 40. The percentage of the number of episodes where the Q-Learning Agent took less than 15 steps was around 40 percent and as can be seen from the histogram, most of the episodes took between 11 to 16 steps to find the terminal state for the “Treasure Hunting in a Cube” environment.



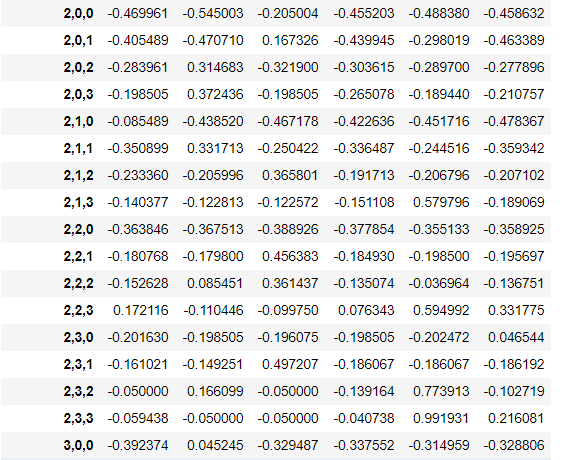


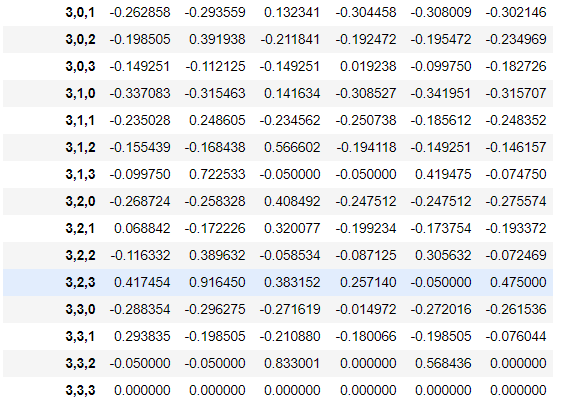
## Final Q-table

After training the Q-Learning Agent for 500 episodes, we have the final updated Q-table that was learnt by the Q-Learning Agent. Although running the Q-Learning Agent for more than 500 episodes will still cause some changes to the Q-values in the Q-table, but the changes will not be very significant and there will not be significant improvement in the number of steps taken by the Q-Learning Agent.



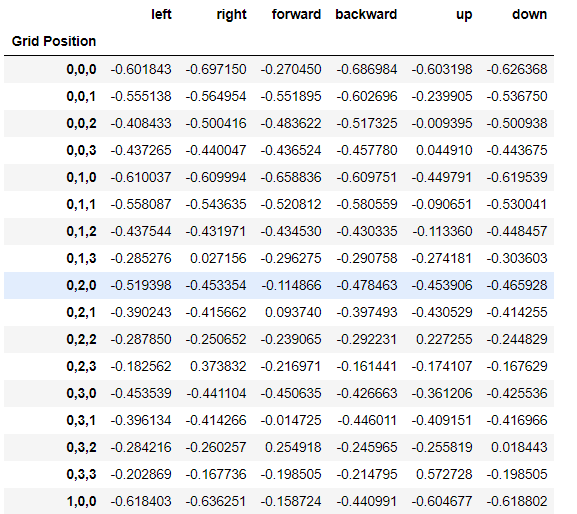






**Q-table after 500 episodes**

Below we have the Q-table after training the Q-Learning Agent for 1000 episodes. As can be seen from the table, there is not many changes in terms of getting better Q-values for each state when compared to the final Q-table of training the Q-Learning Agent for 500 episodes.





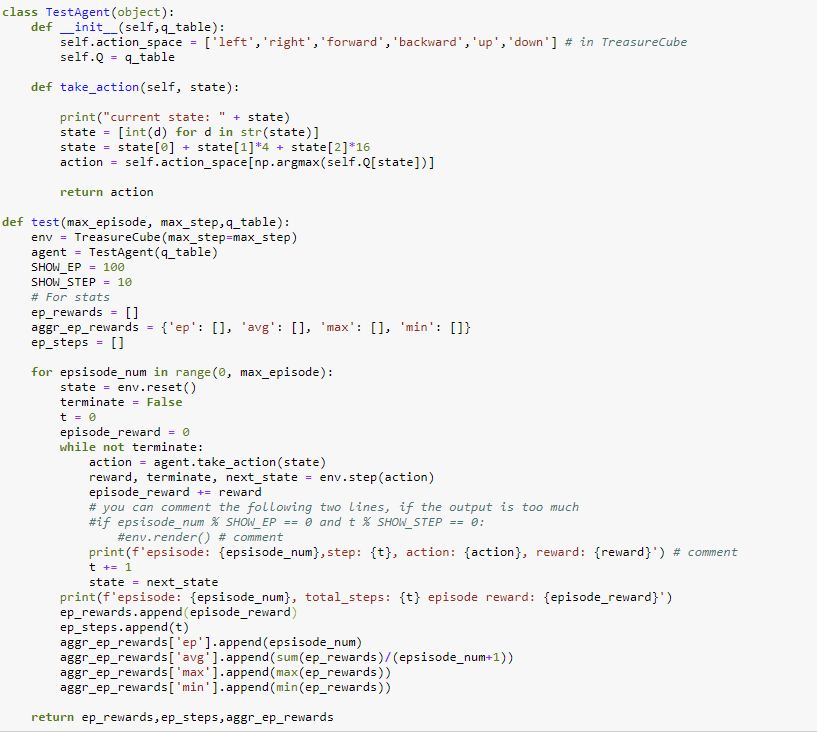




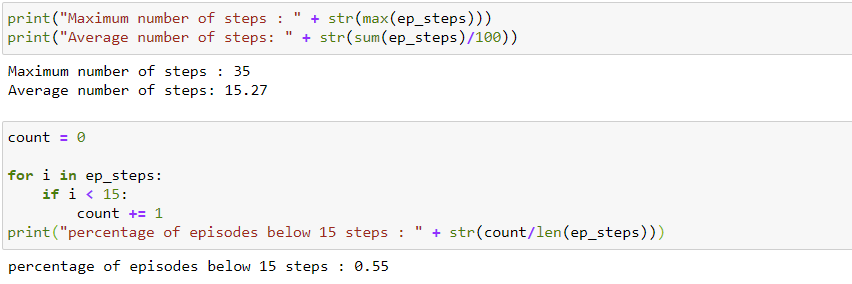
**Q-table after 1000 episodes**

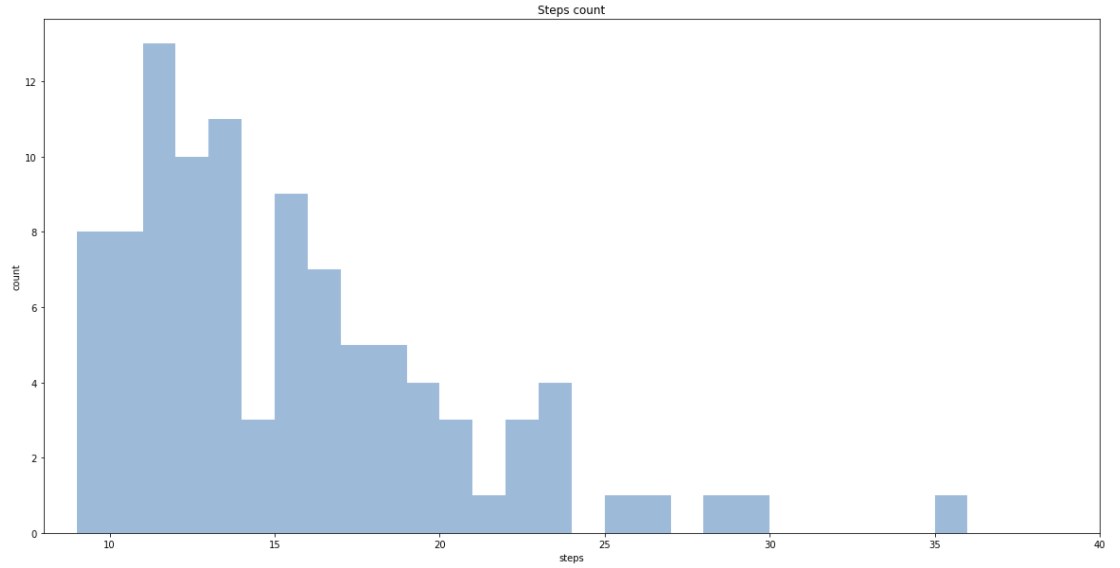
## Quality of the Final Solution

In order to test the quality of the Q-table that was learnt by the Q-Learning Agent, a **TestAgent()** class and a **test()** function is created to run the Q-Learning Agent. The TestAgent() is similar to the **QAgent()** class that we have implemented above, but it does not have a training function and only take actions based on the Q-table that was already learnt by the Q-Learning Agent, that is passed to it as an argument. The **test()** function is similar to the **test\_cube()** function above, except that there is no calling of the train function to train the **TestAgent()**.

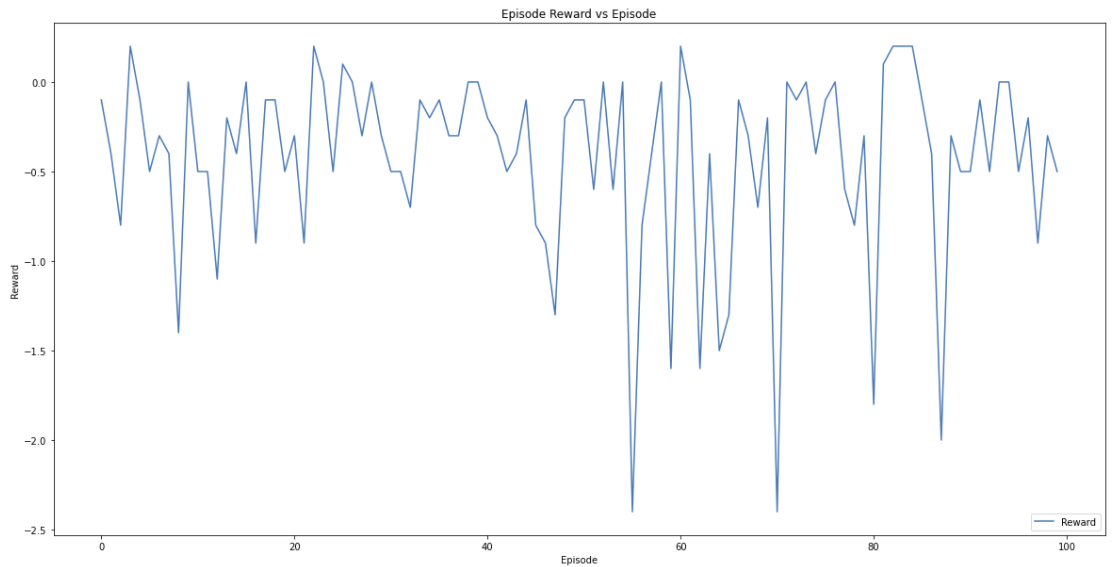


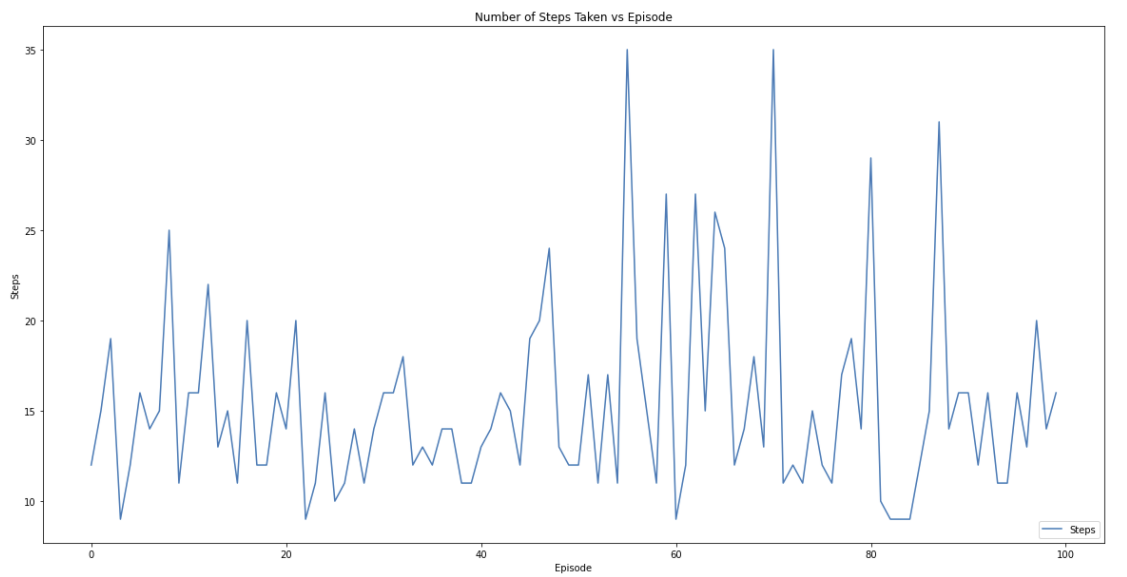
Using the code above, we are able to test the solution quality of the learnt Q-table. We run the **test()** function for 100 episodes and see how well the **TestAgent()** performed with the Q-table. From the results below, we can see that the learnt Q-table yield an average of 15 steps for 100 test episodes and have 55 percent of the episodes that was tested having below 15 steps to find the terminal state.





From the two graphs shown below, we can see that there is quite a significant amount of fluctuations for the Episode Reward and Number of Steps Taken to reach the terminal state for each episode even we a learnt Q-table. A reason for that can be the way the “Treasure Hunting in a Cube” environment was defined in the Assignment. The Assignment instruction states that “The intended movement happens with probability 0.6. With probability 0.1, the agent ends up in one of the states perpendicular to the intended direction.” What this implies is that every time the Agent decide on the best action to take for a particular state, there is a 40 percent chance that the Agent will take a completely different action from the intended action. This will cause the final Q-table to be not the most optimal for the environment, as even during learning, the Q-Learning Agent is not able learn a “true reward” for its decided action as there is a significant chance that the final action taken is not the intended action that the Q-Learning Agent wanted to execute. Therefore, even in the final Q-table, there are some states where the best Q-value for that state does not correspond to a logical best action for that particular state. Therefore, even with a learnt Q-table, we can only get a relatively “good” solution (below 15 steps to reach terminal state) only 55 percent of the time, with significant amount of fluctuations for the Episode Reward and Number of Steps Taken to reach the terminal state for each episode.





# Effects of Changing Q-Learning Agent Parameters

As can be seen from the results obtained above, the Q-table values of the Q-Learning Agent converges quickly at about 100 episodes and there are no significant improvement to the quality of the solution even when we increase the number of episodes that the Q-Learning Agent is trained, with the average number of steps taken each episode at the best minimum of 14 steps, and 55 percent of the episodes having below 15 steps to reach terminal state when tested on 100 episodes using the **test()** function.

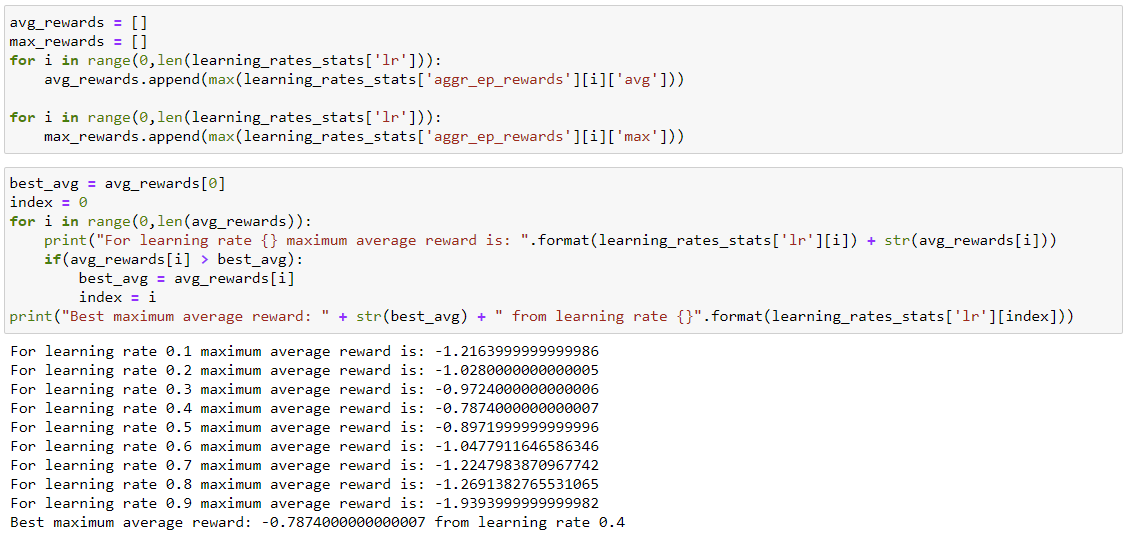
Therefore, in an effort to find out if we can improve the quality of the solution of the Q-Learning algorithm, we conducted experiments to vary the parameters of the Q-Learning Agent, namely, the Learning Rate, the Discount Factor, and the Exploration Rate, to find out if variation of these parameters can yield a better quality solution in terms of average number of steps taken each episode and percent of the episodes having below 15 steps to reach terminal state.

## Tuning of Individual Parameters

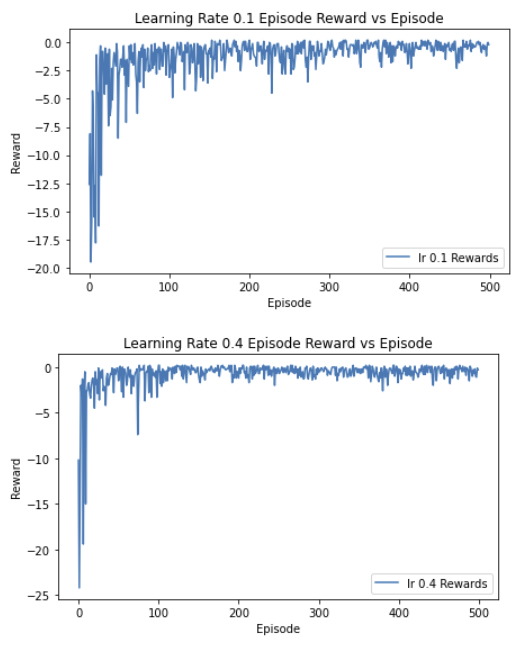
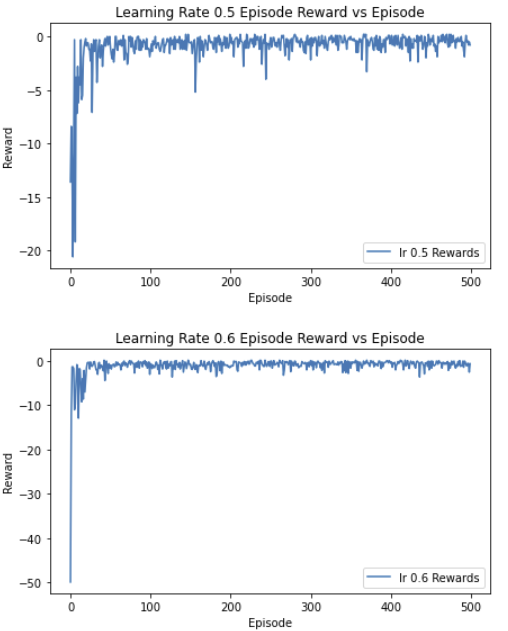
### Learning Rate

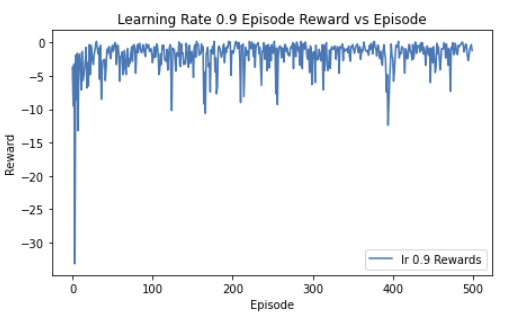
The Learning Rate of a Q-Learning Agent determines to what extent newly acquired information overrides old information. A factor of 0 makes the agent learn nothing (exclusively exploiting prior knowledge), while a factor of 1 makes the agent consider only the most recent information (ignoring prior knowledge to explore possibilities).

For the first set of experiments, we will run the Q-Learning Agent using different Learning Rates to determine what is the most optimal Learning Rate to use for our MDP. The Learning Rates that were used in the experiments are [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]. The same Q-Learning Agent was used as well as the same **test\_cube()** function, with the other Q-Learning parameters fixed the same as the Assignment instructions ( Discount Factor = 0.99, Exploration Rate = 0.01). We run the **test\_cube()** function in a for loop where in each iteration, a different Learning Rate is passed to the Q-Learning Agent. Each Q-Learning Agent with a different Learning Rate is ran for 500 episodes.



After running the Q-Learning Agent with the different Learning Rates, we then find the maximum Average Reward among the 500 episodes for each of the Learning Rate and determine which of the Learning Rate produce the best maximum Average Reward. From the experiments conducted, most of the time, the Learning Rate of 0.4 gives the best results for our MDP problem.



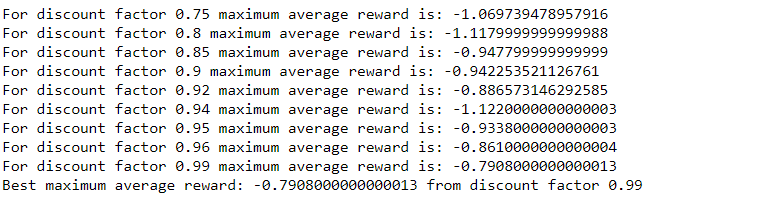


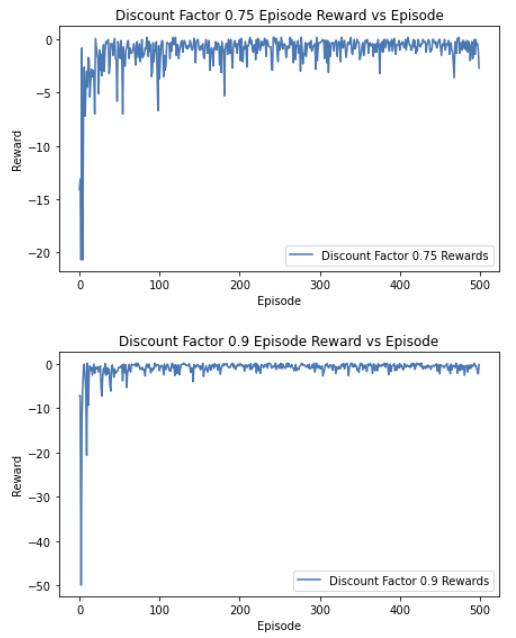
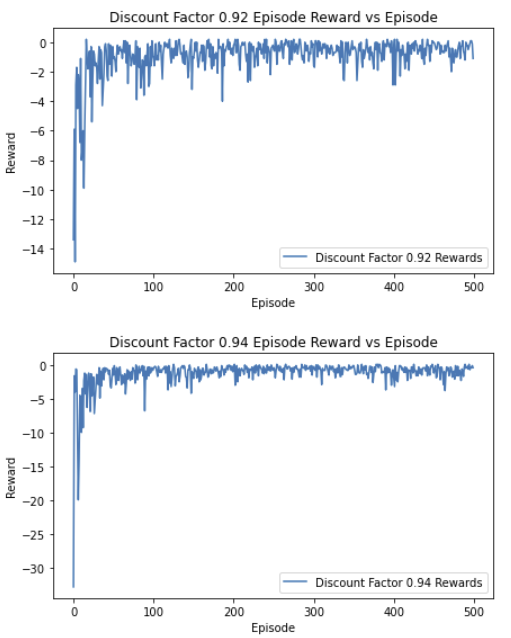
From the graphs above, where we plot the Reward receive for each episode for some of the selected Learning Rates, we find that the 0.4 Learning Rate Rewards are more stable than the other Learning Rates after it converges. We can also see that for a Learning Rate of 0.1, the Reward vs. Episode graph converges the slowest and that for a Learning Rate of 0.9, the Rewards are unstable even after initial convergence, this is due to the fact that the agent consider mainly the most recent information for a high Learning Rate.

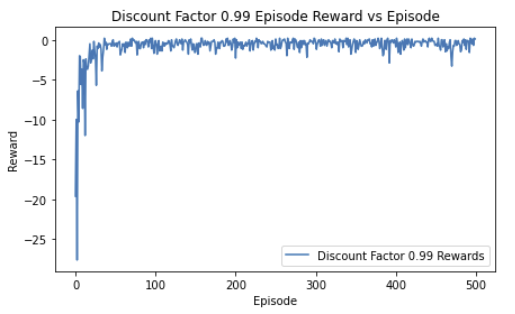
### Discount Factor

The Discount Factor determines the importance of future rewards. A factor of 0 will make the agent "myopic" (or short-sighted) by only considering current rewards, while a factor approaching 1 will make it strive for a long-term high reward.

For the next set of experiments, we will run the Q-Learning Agent using different Discount Factors to determine what is the most optimal Discount Factor to use for our MDP. The Discount Factors that were used in the experiments are [0.75,0.8,0.85,0.9,0.92,0.94,0.95,0.96,0.99]. The same Q-Learning Agent was used as well as the same **test\_cube()** function, with the other Q-Learning parameters fixed (Learning Rate = 0.4, Exploration Rate = 0.01). We run the **test\_cube()** function in a for loop where in each iteration, a different Discount Factor is passed to the Q-Learning Agent. Each Q-Learning Agent with a different Discount Factor is ran for 500 episodes.



After running the Q-Learning Agent with the different Discount Factors, we then find the maximum Average Reward among the 500 episodes for each of the Discount Factor and determine which of the Discount Factor produce the best maximum Average Reward. From the experiments conducted, most of the time, the Discount Factor of 0.99 gives the best results for our MDP problem.

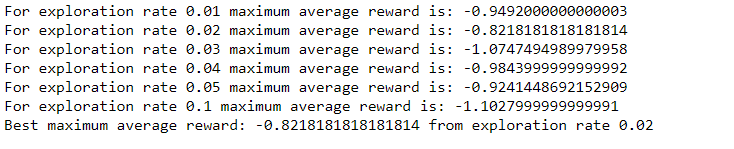


From the graphs above, where we plot the Reward receive for each episode for some of the selected Discount Factors, we find that the 0.99 Discount Factor Rewards are more stable than the other Discount Factors after it converges, with its closest match being the Discount Factor of 0.94. Discount Factors of below 0.9 gives rise to more unstable Rewards after convergence, as compared to Discount Factors of above 0.9.

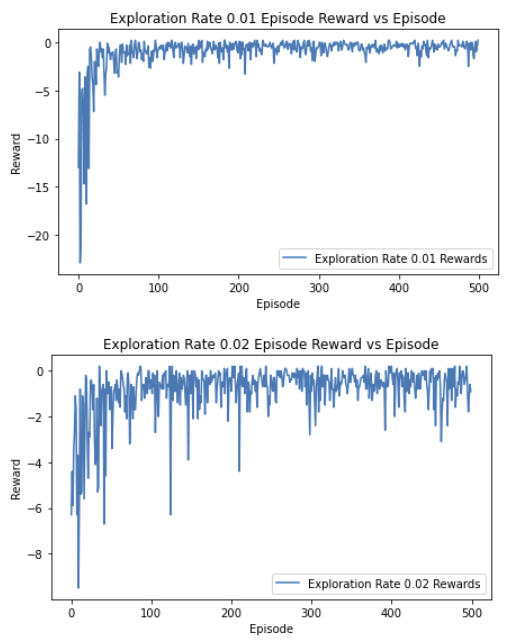
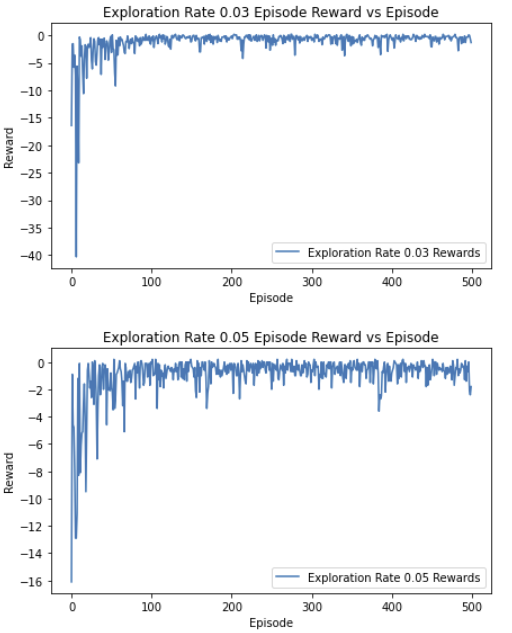
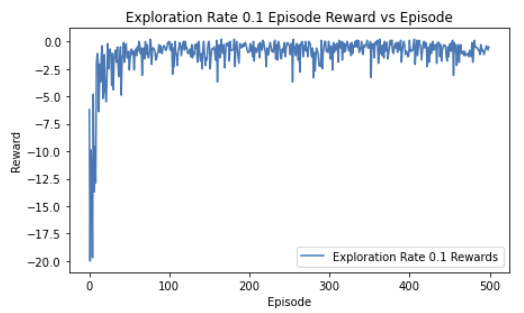
### Exploration Rate

The Exploration Rate of the Q-Learning Agent determines how often the Q-Learning Agent will take a random action, rather than the action determined by the best Q-value in the Q-table for that particular state. Acting randomly is important because it allows the agent to explore and discover new states that otherwise may not be selected during the exploitation process. Therefore, by varying the Exploration Rate, we can determine how often we want the Q-Learning Agent to explore during the learning process.

For the next set of experiments, we will run the Q-Learning Agent using different Exploration Rates to determine what is the most optimal Exploration Rate to use for our MDP. The Exploration Rates that were used in the experiments are [0.01,0.02,0.03,0.04,0.05,0.1]. The same Q-Learning Agent was used as well as the same **test\_cube()** function, with the other Q-Learning parameters fixed (Learning Rate = 0.4, Discount Factor = 0.99). We run the **test\_cube()** function in a for loop where in each iteration, a different Exploration Rate is passed to the Q-Learning Agent. Each Q-Learning Agent with a different Exploration Rate is ran for 500 episodes.



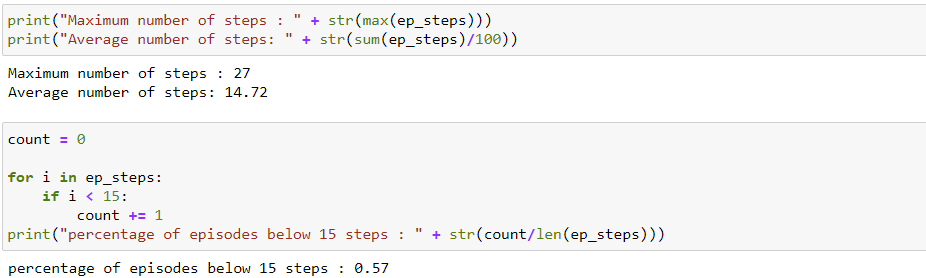
After running the Q-Learning Agent with the different Exploration Rate, we then find the maximum Average Reward among the 500 episodes for each of the Exploration Rate and determine which of the Exploration Rate produce the best maximum Average Reward. From the experiments conducted, most of the time, the Exploration Rate of 0.02 gives the best results for our MDP problem.

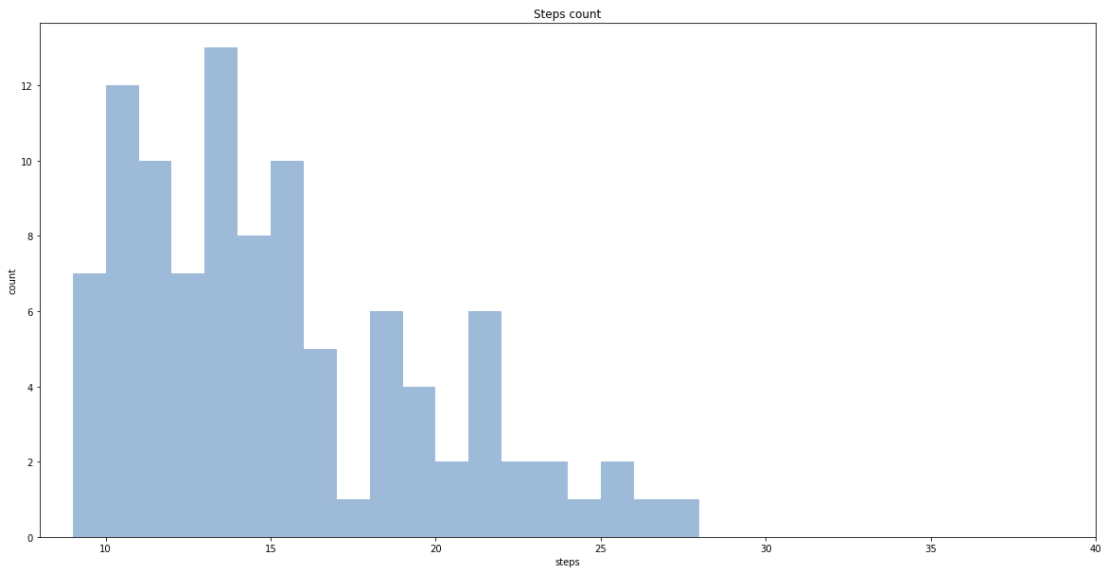
From the graphs above, where we plot the Reward receive for each episode for some of the selected Exploration Rates, we can see that for an Exploration Rate of 0.02, the lowest Reward is significantly lower than the lowest Reward of other Exploration Rates. Although we can see fluctuations of the Rewards for each episode, the fluctuations are mostly contained 0 to -2.

## Training and Testing of Q-Learning Agent with Optimal Parameters

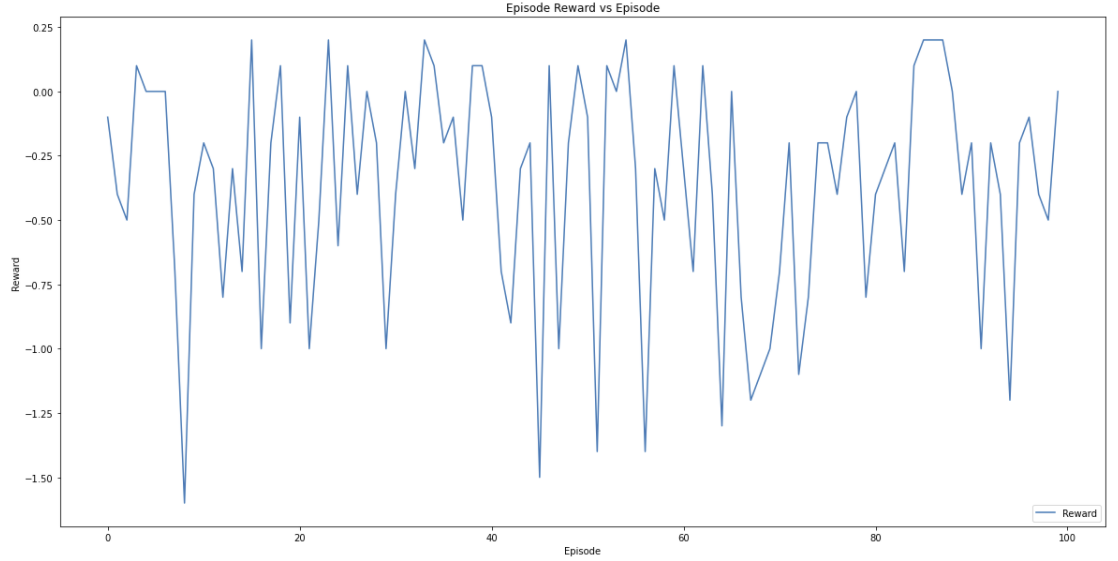
After finding the optimal parameters for the Q-Learning Agent, we then train a Q-Learning Agent using the parameters: Learning Rate = 0.4, Discount Factor = 0.99, and Exploration rate = 0.02. We train the Q-Learning Agent for 500 episodes to get the final trained Q-table. Then, using the Q-table, we pass it to the class **TestAgent()** and run the **test()** function to test the Q-table for 100 episodes on the environment.

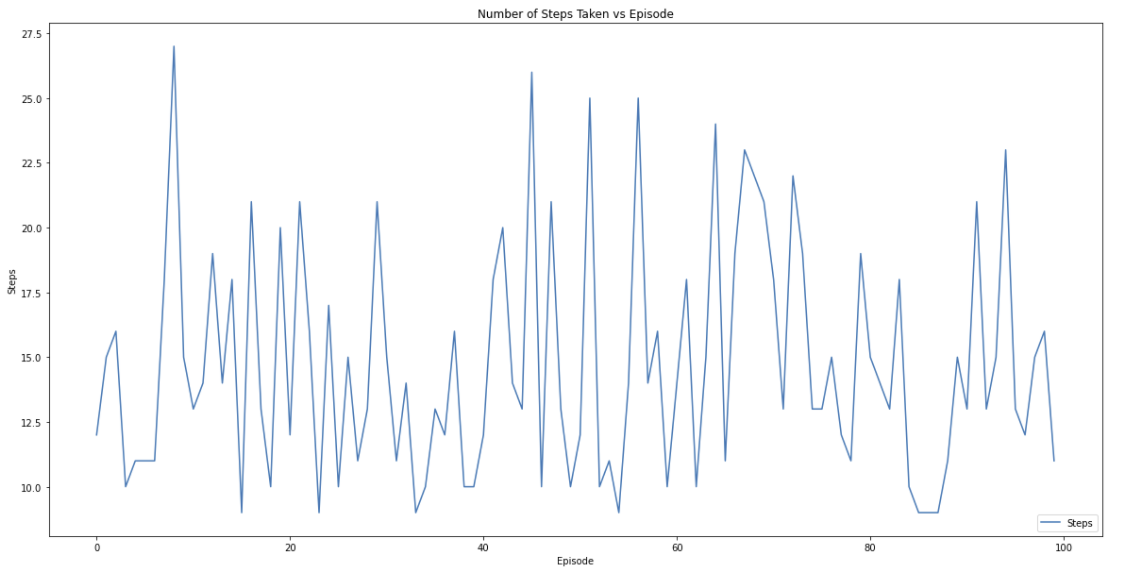


From the figure above, we can see that the Q-Learning Agent with optimized parameters fared slightly better than the Q-Learning Agent with original parameters with an average number of steps for each episode at 14.72 as compared to 15.27, and a percentage of 57 percent of episodes having lesser than 15 steps to reach terminal state as compared to the 55 percent of the original Q-Learning Agent.



The histogram above shows that with a Q-Learning Agent with optimized parameters, most of the episodes are able to reach the terminal state in less than 15 steps.

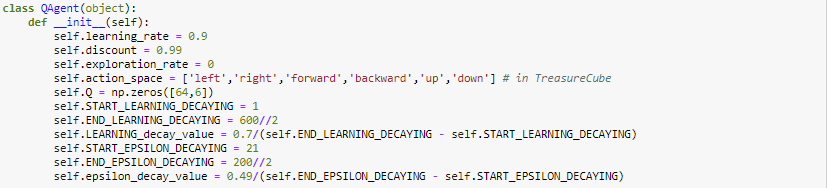


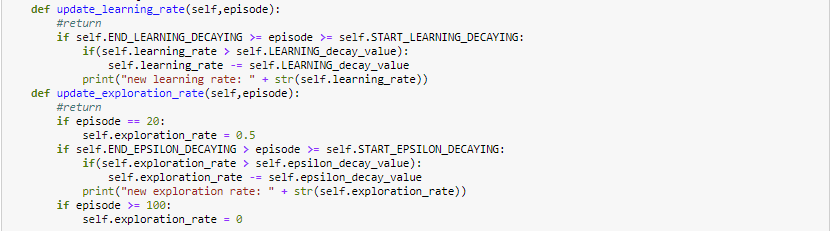


The graphs above shows that there are significantly more episodes with a positive reward received at the end of the episode as well as most of the episodes having steps below the 17.5 steps line. Although we are not able to improve the average number of steps per episodes to below 14 steps, there are improvements to the learnt Q-table.

## Decaying Learning Rate and Exploration Rate

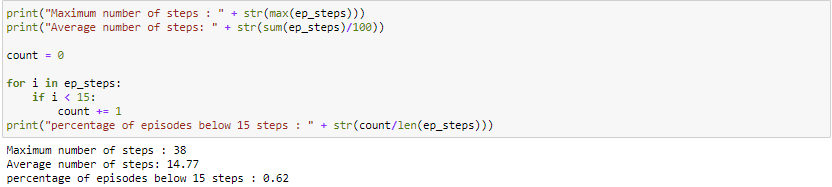
A high Learning Rate causes the Q-Learning Agent to learn faster but causes significant fluctuations in the reward receive in each episode even after convergence. A high Exploration Rate causes the agent to explore and discover new states that otherwise may not be selected during the exploitation process but can be detrimental to the learning process at the later episodes. What if there is a way to take advantage of a having a high Learning and Exploration Rate and eliminate the disadvantages that they bring? Therefore, we introduced a Decaying Learning Rate and Exploration Rate for the Q-Learning Agent, so that at the earlier episodes, the Q-Learning Agent can learn faster and explore the environment more, and have a lower Learning Rate and Exploration Rate for later episodes to limit the fluctuations in the rewards received in the later episodes.



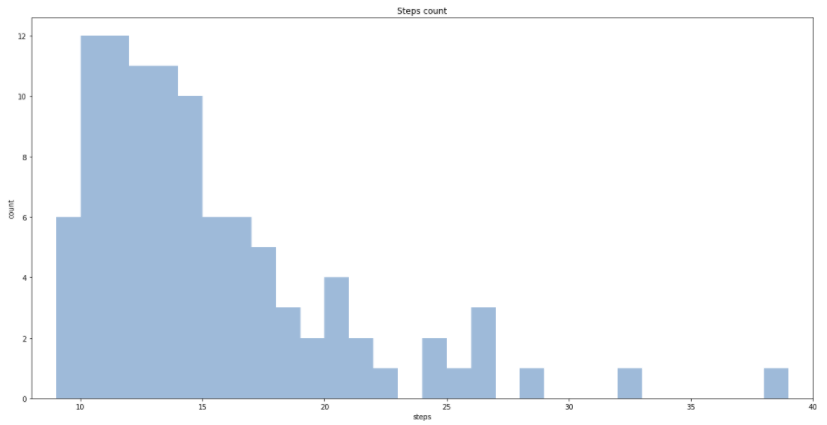


We made modifications for the QAgent class so that we set the initial Learning Rate to 0.9 and Exploration Rate to 0. We then introduced an Exploration Rate of 0.5 at episode 20 as can be seen in the **update\_exploration\_rate()** function after the agent has learned for 20 episodes. The Learning Rate starts decaying from the first episode all the way till the 300th episode until a Learning Rate of 0.2 and the Exploration Rate starts decaying at the 21st episode till the 100th episode until an Exploration Rate of 0.01. The **update\_exploration\_rate()** function and **update\_learning\_rate()** functions are used to update the Learning Rate and Exploration Rate and is called in the **test\_cube()** function. The Q-Learning Agent is trained for 500 episodes and the Q-table is then tested for 100 episodes.

### Test Results



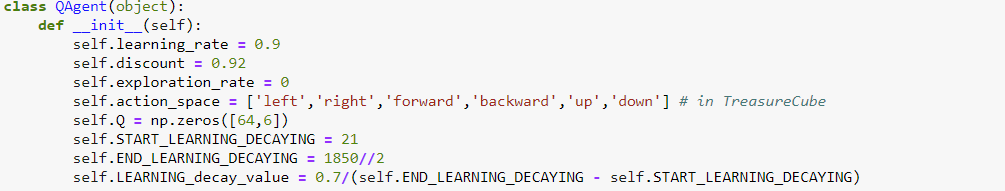
After running the **test()** on the Q-table learnt by the Q-Learning Agent with Decaying Learning and Exploration Rate, the results for testing on 100 episodes is shown in the figure above. We are still only able to get an average number of 14 steps taken to reach terminal state for each episode, but the percentage of episodes below 15 steps taken to reach terminal state is significantly improved to 62 percent.



The histogram above shows us a huge concentration of episodes that are able to reach the terminal state below 15 steps, doing significantly than the original Q-Learning Agent and the Q-Learning Agent with optimized parameters.

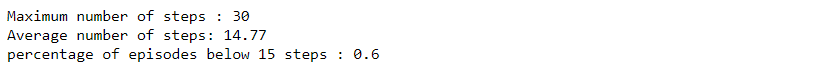
## Q-Learning Agent with no Exploration Rate and a Decaying Learning Rate

After conducting the various experiments, one thing that was realized is that we do not actually need an Exploration Rate for our environment. For our Assignment environment, it is made known that “The intended movement happens with probability 0.6. With probability 0.1, the agent ends up in one of the state’s perpendicular to the intended direction.” Which tells us that there is actually a 40 percent chance each step that the Q-Learning Agent takes a random action. Therefore, with that in mind, we have conducted an experiment with a Q-Learning Agent with 0 Exploration Rate and a decaying Learning Rate.

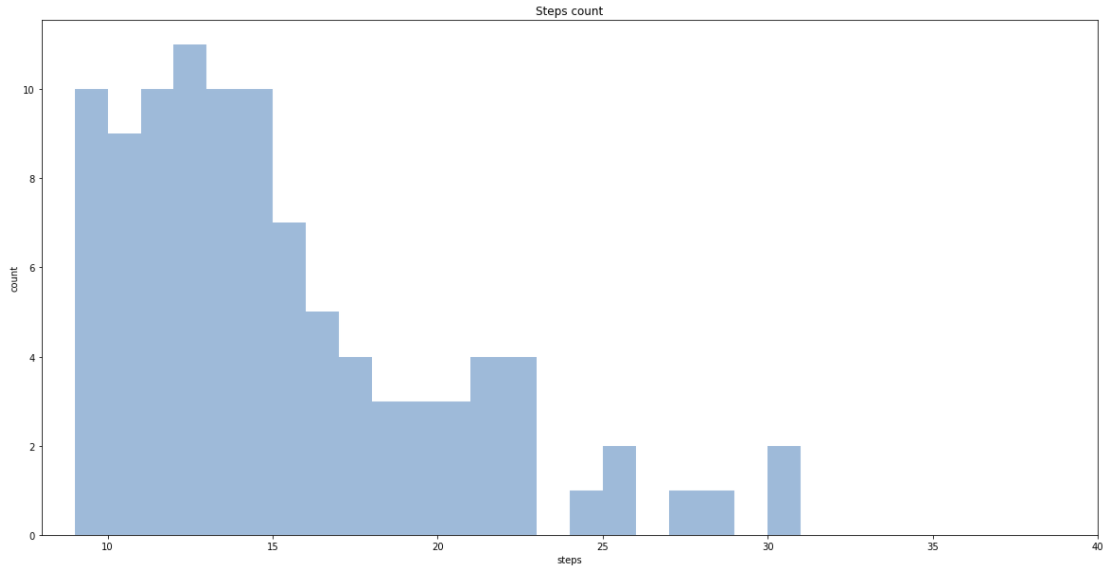




For this experiment, we set the initial Learning Rate to 0.9 and start the decay of the Learning Rate after 20 episodes. Also, taking to ensure that the Agent is trained sufficiently at lower Learning Rates at later episodes, we increase the number of training episodes to 2000 episodes. The Learning Rate is decayed to 0.2 and the decay runs from the 21st episode to the 925th episode. The Q-Learning Agent is trained for 2000 episodes and the final Q-table is then tested for 100 episodes.

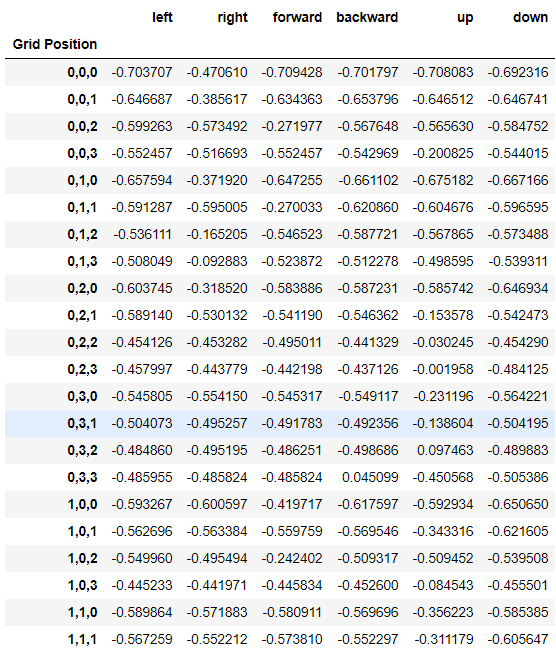


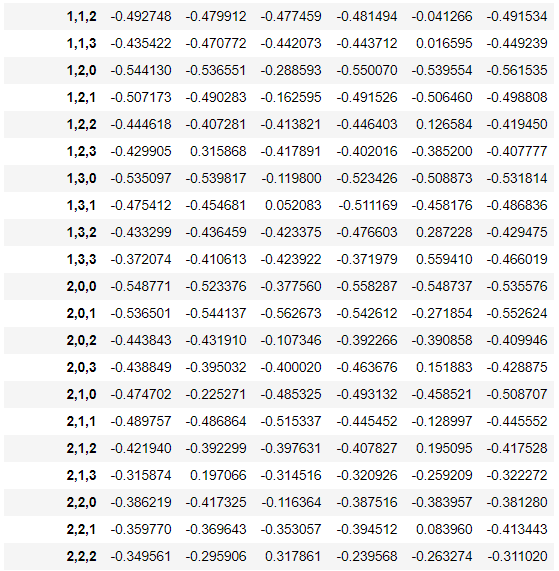
Testing the finalized Q-table for 100 episodes, we got an average number of 14 steps taken to reach terminal state for each episode, and percentage of episodes below 15 steps taken to reach terminal state at 60 percent. We can see from the results that even with a 0 Exploration Rate, we are able to get an average number of 14 steps taken to reach terminal state for each episode, and percentage of episodes below 15 steps taken to reach terminal state higher than that of the original Q-Learning Agent and Q-Learning Agent with optimized parameters.

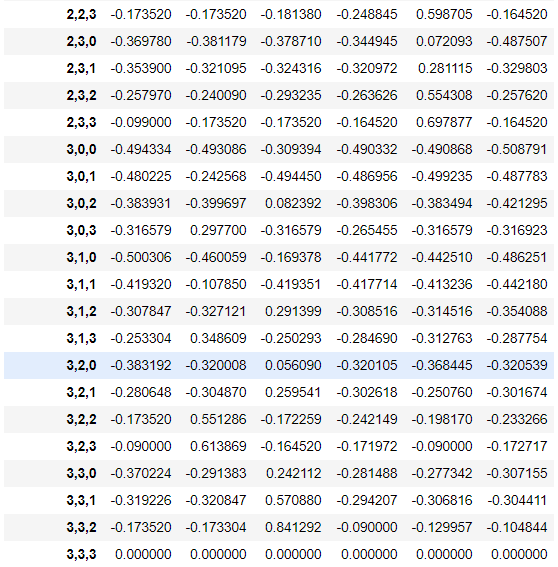


From the histogram above, we can see a high concentration of episodes having below 15 steps taken to reach terminal state and also, the number of episodes for reaching the terminal state in 9 steps (best solution) is 10 out of 100 episodes, which is the best amongst all the Q-Learning Agents tested.

### Q-table







# Conclusion

In conclusion, from all the experiments conducted on the “Treasure Hunting in a Cube” MDP, we can observe that learning of an environment by a Q-Learning Agent is very fast, with all of the Q-Learning Agents having the number of steps taken for each episode as well as the amount of reward received each episode converging at around 100 episodes, with little to no improvements to the quality of the solution after that. The best average number of steps taken for each episode to reach the terminal state is 14 steps no matter how many episodes we train the Agent on or tune the parameters of the Q-Learning Agent.

However, we learnt that by having a decaying Learning Rate and Exploration Rate, we are able to get a better-quality solution with a higher percentage of episodes having less than 15 steps to reach the terminal state when tested on 100 episodes, with 62% of the number of episodes tested reaching the terminal state in less than 15 steps, as compared to the 55% of the baseline Q-Learning Agent. We also discovered that due to the random nature of the given environment, where there is a 40% probability that the Q-Learning Agent does not take the intended action, we can do away with the Exploration Rate and still get a good quality solution for the MDP problem.

For the “Treasure Hunting in a Cube” MDP, although the best solution to reach the terminal state is 9 steps, the average number of steps taken for each episode to reach terminal state is 14 steps, which is quite far from the optimal number of steps. This can be attributed to the high randomness of the given environment. For the given environment, the intended movement happens with probability 0.6, with probability 0.1, the agent ends up in one of the state’s perpendicular to the intended direction, meaning, for every action taken in the environment, there is a 40 percent chance of taking a totally random action which causes the Agent to not be able to learn the optimal action for every state of the environment. Even when the Agent has learnt an optimal path to the terminal state from the start state, the Q-values may be overwritten in the Q-table if the Q-Learning Agent stray from the optimal path due to it not being able to take an intended action.

Therefore, even when we are able to reach the terminal state with the most optimal number of steps taken and the maximum reward achieved for some of the episodes, for most of the other episodes, we are not able to reach the terminal state with the most optimal number of steps taken, which leads to the average number of steps taken for each episode to reach terminal state being 14 steps.