A picture containing text

Description automatically generated

**CZ3005 ARTIFICIAL INTELLIGENCE**

**LAB EXERCISE 2**

**LAB REPORT**

Shearman Chua Wei Jie (U1820058D)

LAB GROUP: TSP1

# Treasure Hunting in a Cube

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

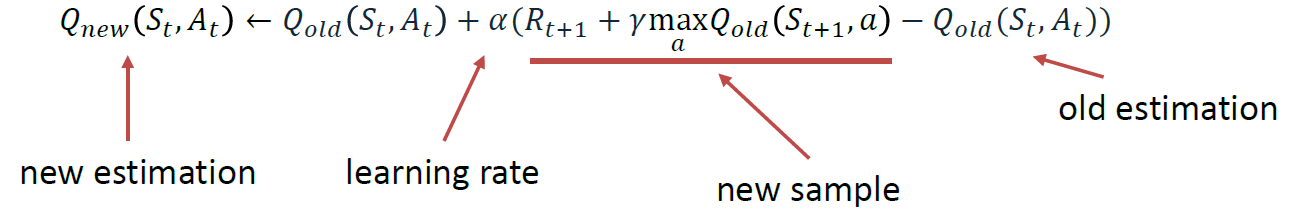
* State: a 3D coordinate, which indicates the current position where the agent is. The initial state is (0, 0, 0) and there is only one terminal state: (3,3,3).
* Action: The action space is (forward, backward, left, right, up, down). The agent needs to select one of them to navigate in the environment.
* 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.

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:



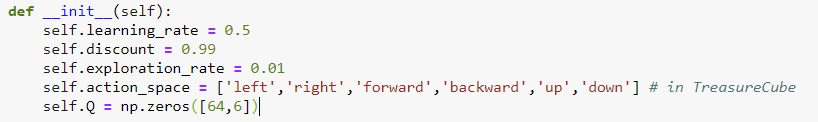
At every step, we adjust our Q-values based on the difference between the discounted new values and the old values. We discount the new values using discount factor (γ) and we adjust our step size using learning rate (α).

## Implementation of Q-Learning algorithm in Python

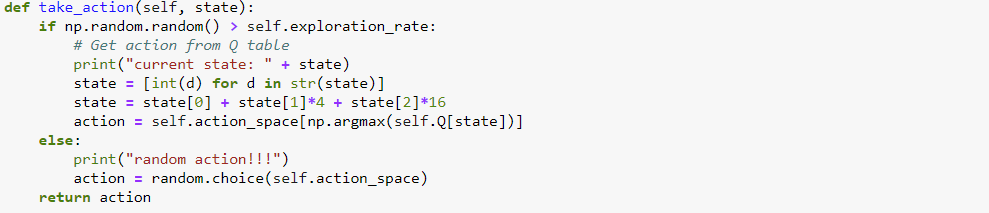
### Q Agent Class



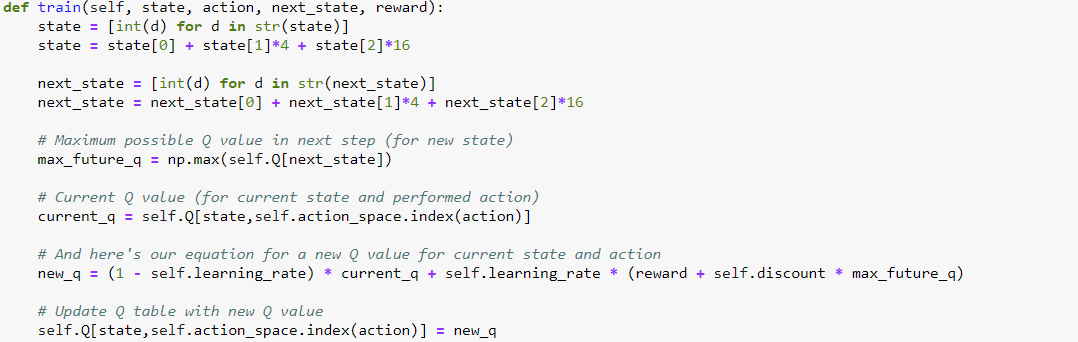
In the code, we first define the class and its functions for our Q-Learning agent. When the Agent is instantiated, its Learning Rate, Discount Factor, Exploration Rate are set based on the given parameters given in the assignment instruction, discount factor = 0.99, learning rate = 0.5, exploration rate = 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.



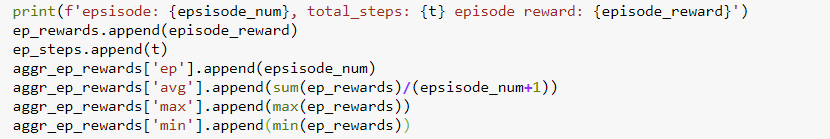
### Test Cube Function



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 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.

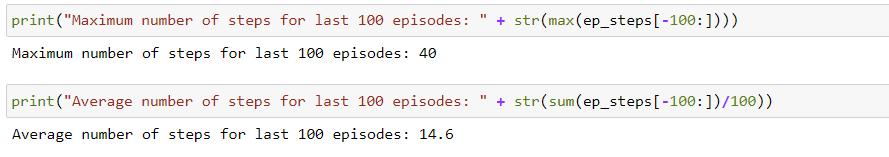
The for loop in the function runs for the number of episodes that was set by max\_episode, and for each episode, the environment is first reset to the initial state and the **state** variable is set as the environment reset state, with the **terminate** variable set to false. Then, while the episode 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.

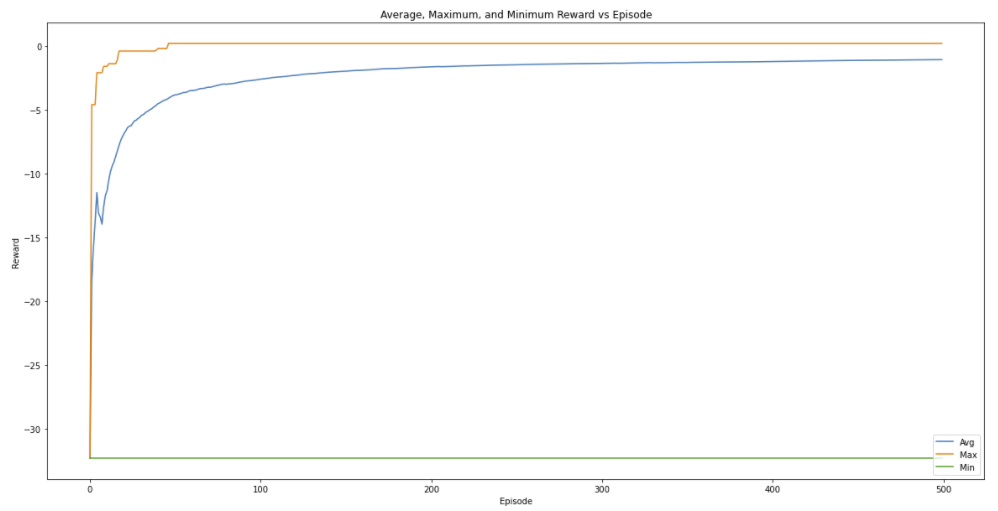


## Results of the Q-Learning algorithm (500 episodes, 500 steps)

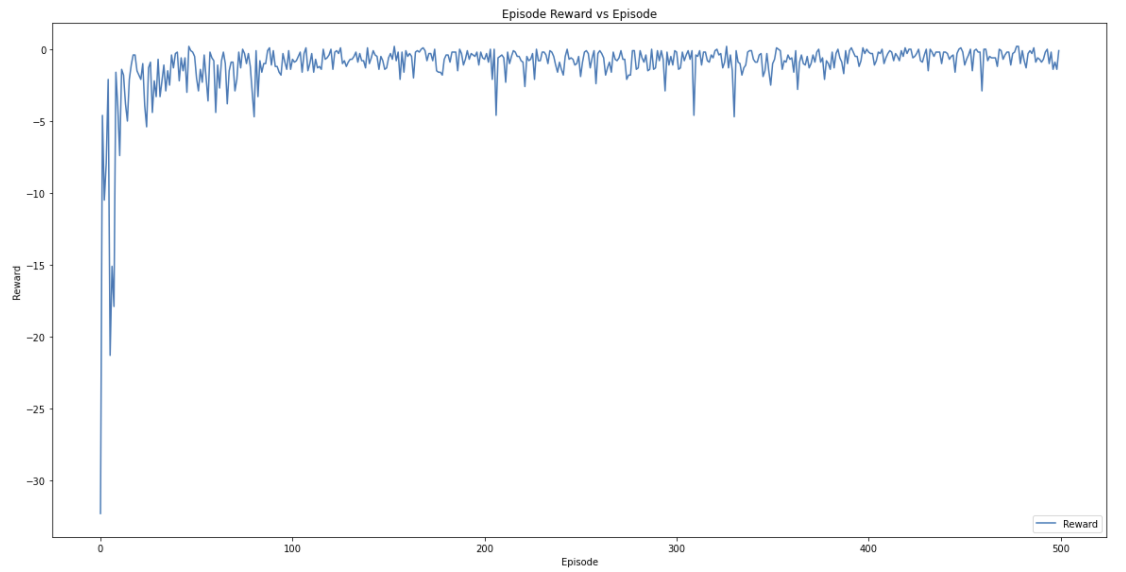
The Q-Learning algorithm was ran for 500 episodes with the maximum number of steps set to 500 as well. In order to determine how well the Q-Learning Agent was performing for the last 100 episodes, we displayed the maximum number of steps taken for the Q-Learning Agent for the last 100 episodes that was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



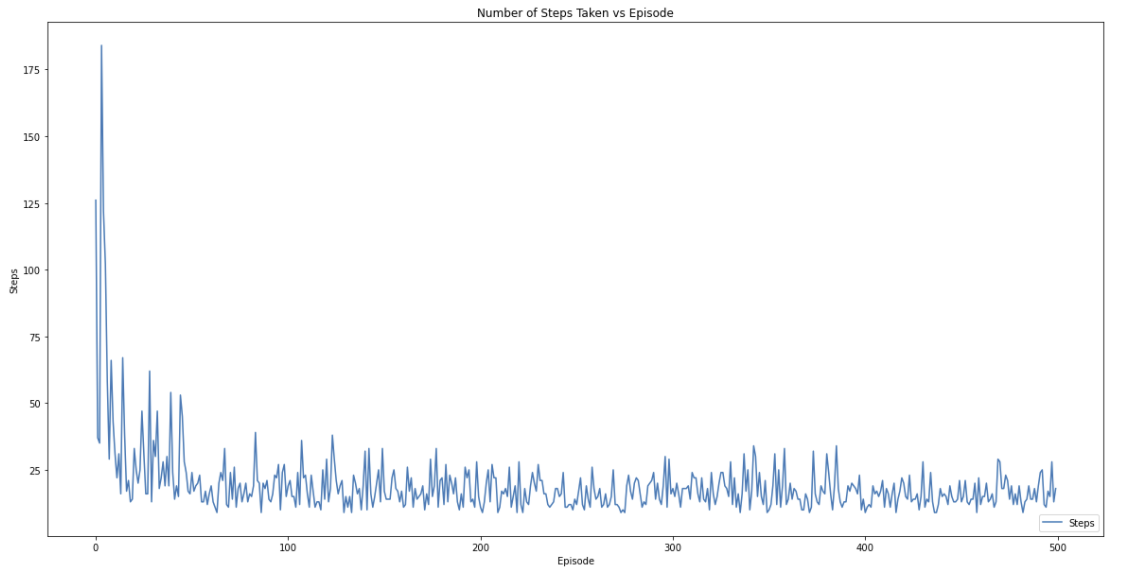
We then plot the graph of the Average Reward received, Maximum Reward received, as well as the Minimum Reward received for each episode for all of the 500 episodes. We can see that a Q-Learning Agent learns very quickly, with the Maximum reward converging at about after 40 episodes and the Average reward 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.



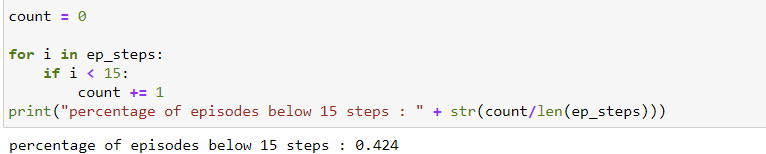
Next, we plot the graph of the Episode reward against the Episode for each of the 500 episodes. From the graph, we can see that the Episode reward received by each Episode converges very fast, with the rewards being fairly consistent after training the Q-Learning Agent for about 100 episodes. This means 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.

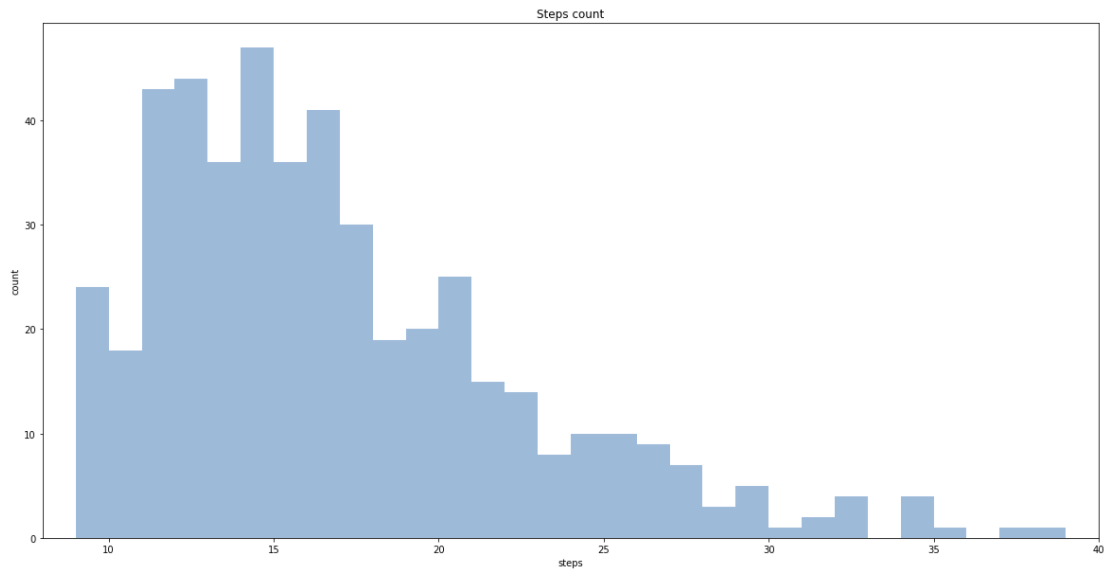


The graph below shows the Number of Steps Taken for each episode for all the 500 episodes of training the Q-Learning Agent. Similar to the Episode reward against the Episode graph, it can be seen that the Q-Learning Agent learns very quickly, with the Number of Steps Taken for each episode also converging 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.



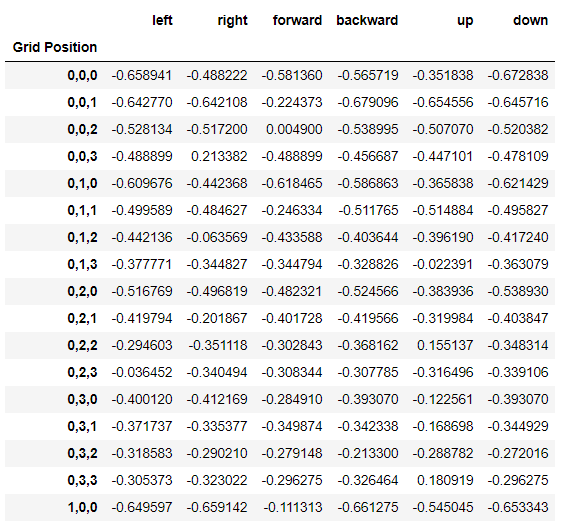
In order to determine the quality of the solution after 500 episodes of training the Q-Learning Agent, we decided to find the percentage of the number of episodes where the Q-Learning Agent took less than 15 steps for the 500 episodes that was run. We also plot the histogram of the count of the number 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.



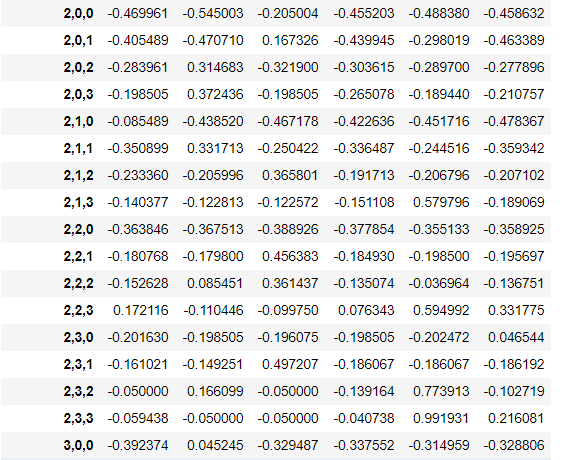


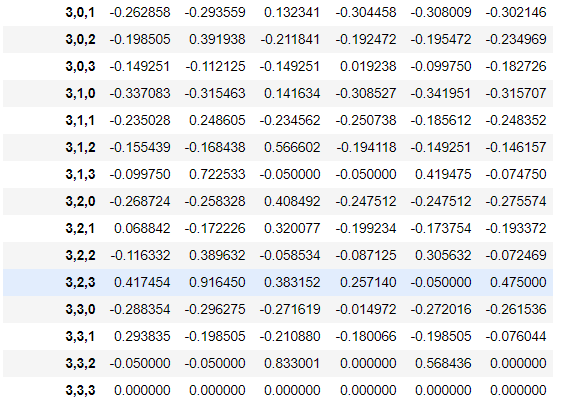
## Final Q-table

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.



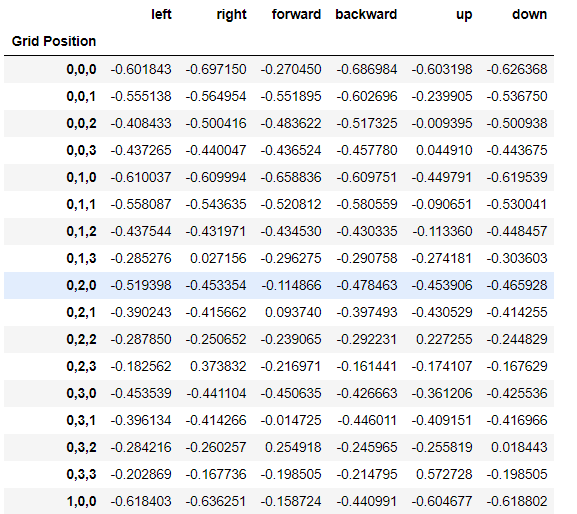






**Q-table after 500 episodes**

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.





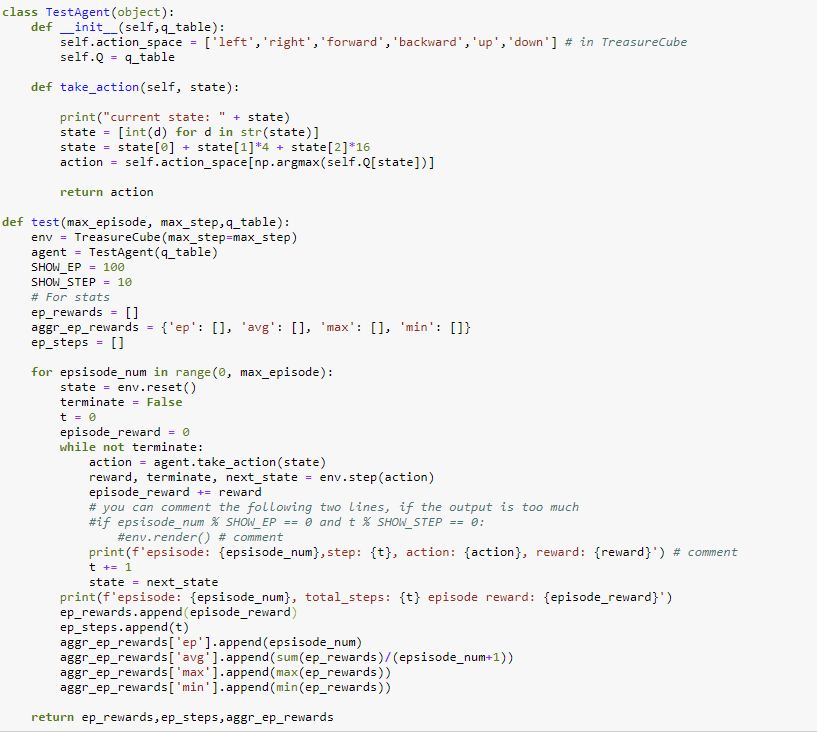




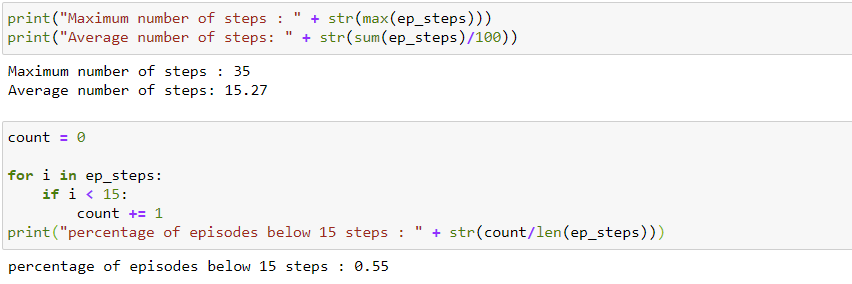
**Q-table after 1000 episodes**

## Quality of the Final Solution

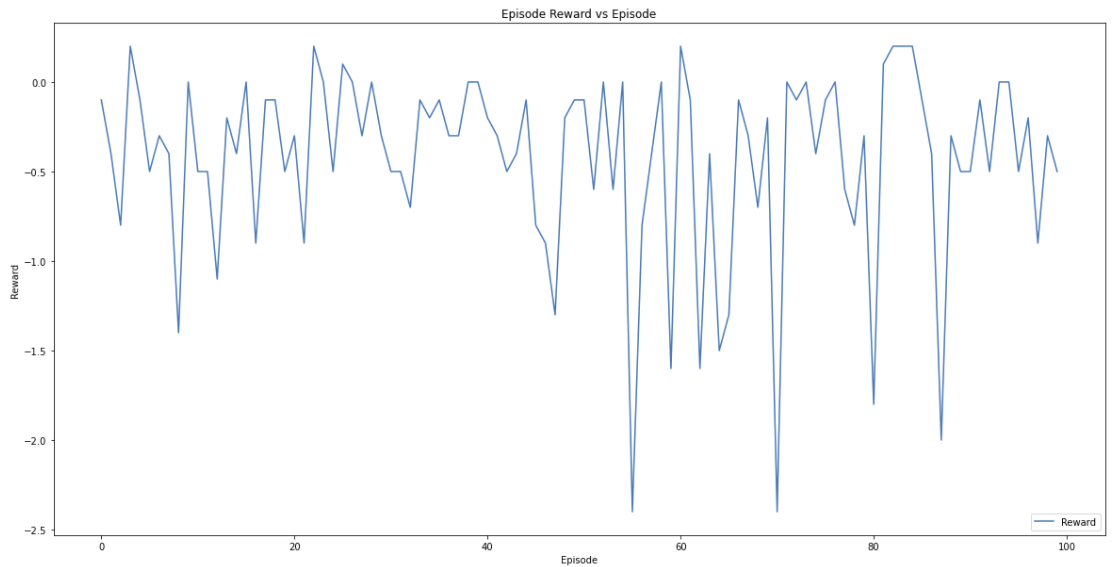
In order to test the quality of the Q-table that was learnt by the Q-Learning Agent, a **TestAgent()** class and a **test()** function is created to run the Q-Learning Agent. The TestAgent() is similar to the **QAgent()** class that we have implemented above, but it does not have a training function 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()**.

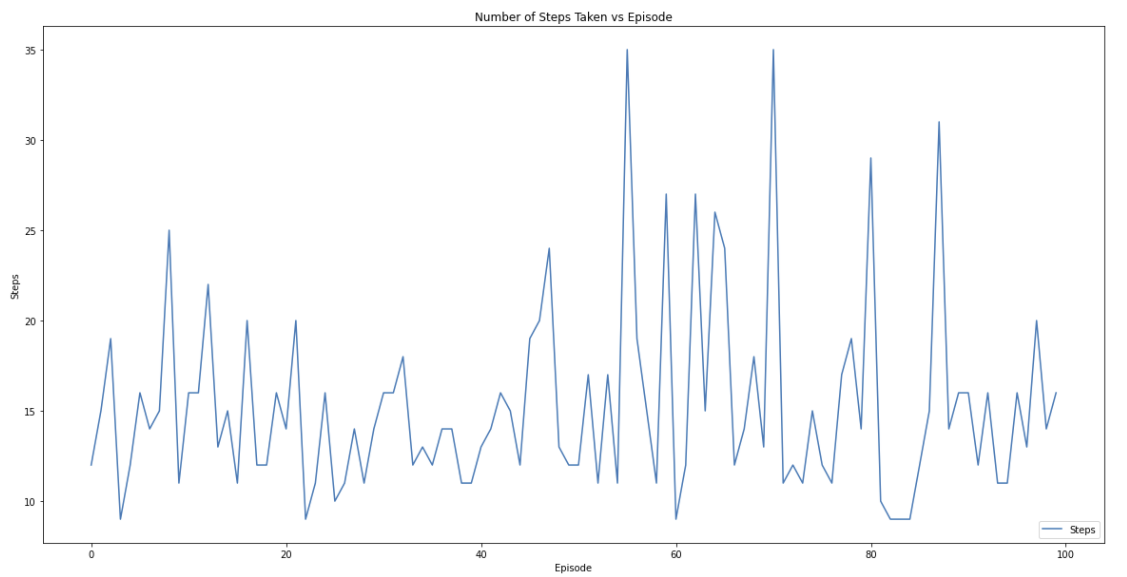


Using the code above, we are able to test the solution quality of the learnt Q-table. We run the **test()** function for 100 episodes and see how well the **TestAgent()** performed with the Q-table. From the results below, we can see that the learnt Q-table yield an average of 15 steps for 100 test episodes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





# Conclusion

In conclusion, from all the experiments conducted on the “Treasure Hunting in a Cube” MDP, we can see 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. The best average number of steps taken for each episode to reach terminal state is 14 steps no matter how many episodes we train the Agent on or tune the parameters of the Q-Learning Agent.

For the “Treasure Hunting in a Cube” MDP, although the best solution to reach 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.