Roger Wienaah

ROB 537 Homework 3

Part A:

For this assignment, I implemented a Q-learning algorithm to train an agent to navigate to a goal location on a 10x5 grid environment. The agent is spawned at random starting locations in the environment and receives a reward of +20 if it gets to the goal(door); otherwise, it receives a reward of -1 anytime it is not at the goal location.

In Q-learning, an agent learns a policy by iteratively updating a Q-value table as it explores the environment to find which states and actions yield the highest cumulative rewards. The Q-learning algorithm implemented is summarized in the following steps:

1) Initialization:

- Initialize the Q-value table $Q(s, a)$ arbitrarily for all state-action pairs. I set them to zero.
- Set the learning rate α, discount factor γ, and exploration rate ε

2) For each episode:

- Initialize the starting state, s
- Set a flag for completion to *False*.

3) For each step within the episode:

- Select an action a using the ε-greedy policy:
  - With probability ε, select a random action (i.e., exploration).
  - With probability 1 - ε, select the action $a = argmax_a\ Q(s, a)$ (i.e., exploitation).
- Take action a, observe the reward r, and the new state s'.
- Update the Q-value using the update rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha (r + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

  Where:

  - Q(s, a): Current Q-value for state s and action a.
  - α: Learning rate
  - r: Immediate reward received after taking action a from state s.
  - γ: Discount factor that balances immediate and future rewards.
  - s': Next state after taking action a.
  - $\max_{a'} Q(s', a')$: Maximum Q-value for the next state s'

- Update the current state: $s \leftarrow s'$.

4) Decay Epsilon:

  - After the episode finishes, reduce the exploration rate

5) End of Episode:

- If the terminal state is reached or a maximum number of steps is exceeded, end the episode.

6) Repeat until a stopping criterion is met (e.g., a set number of episodes)

For my experiments, I ran the algorithm **10 times** and plotted the rewards for each of the trials, and included a standard error plot.

Parameters used in my implementation:

Time steps, t = 2000

Episodes = 300

Learning rate, α = 0.25

Discount factor, γ = 0.9

Epsilon, ε = 0.9

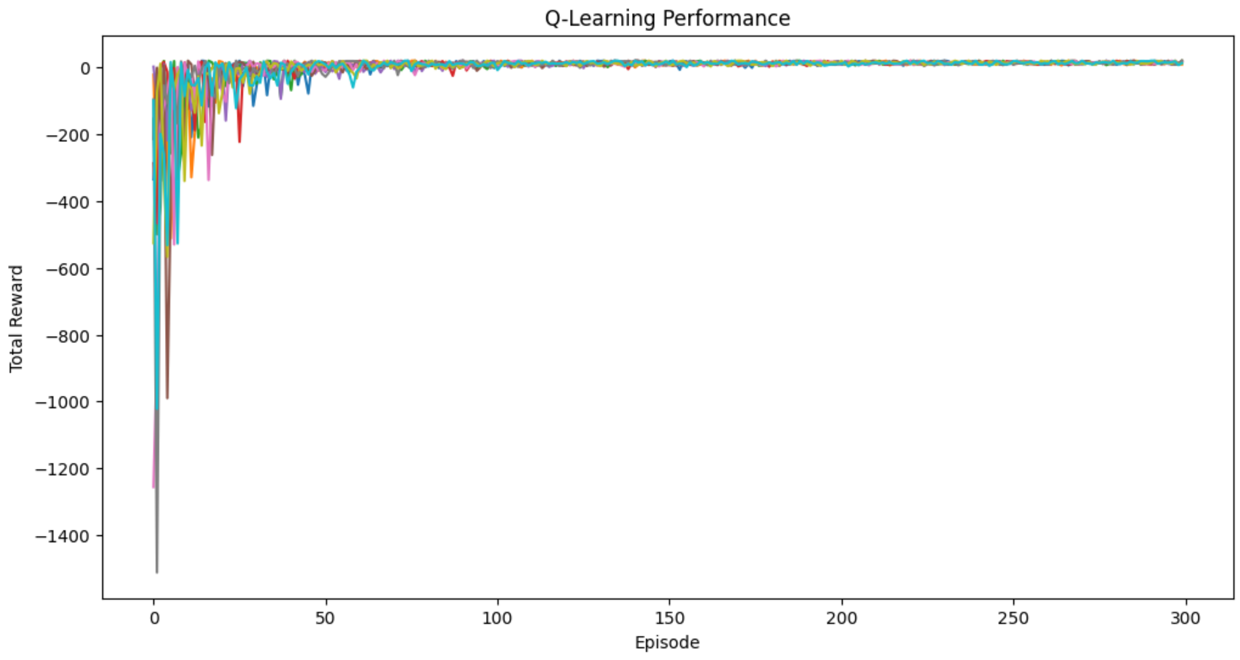
Epsilon decay rate, β = 0.9

How the algorithm performed:

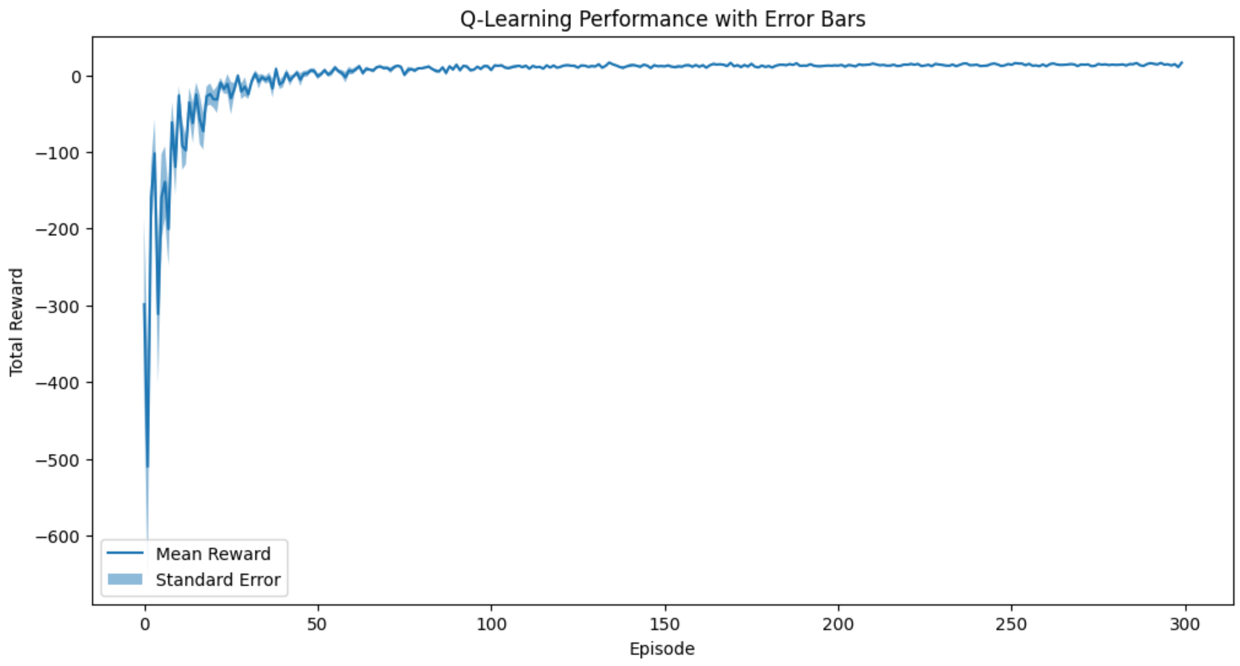
The algorithm performed well, and the agent was able to learn an optimal policy to get to the goal position. This is evident in the reward curves, the standard error plot, and Q-table heatmaps below. The reward curves converged after approximately 70 episodes. Also, the standard error was a bit large at the beginning but shrank as the training progressed. This shows that all 10 trials, despite different random starts, converged to a good policy.

10 experiments plots:

Learning curves:

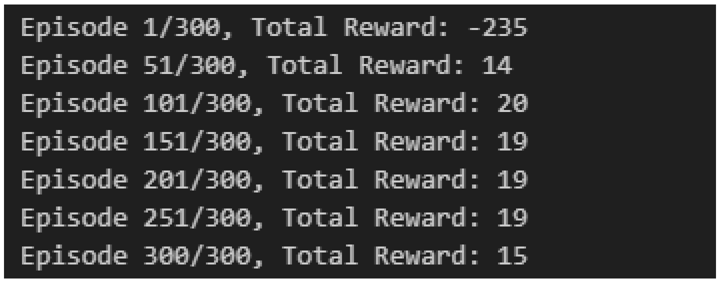


Standard Error:



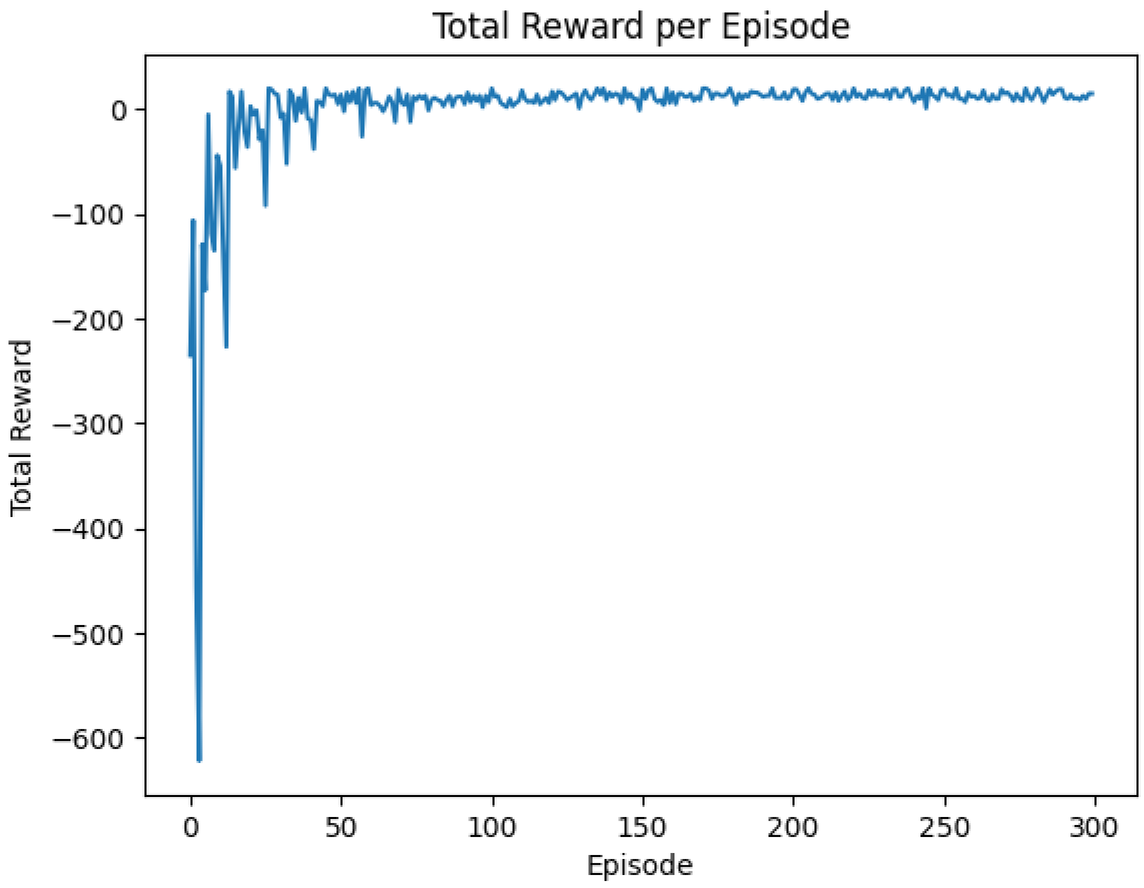
Single run experiment:

Reward outputs:

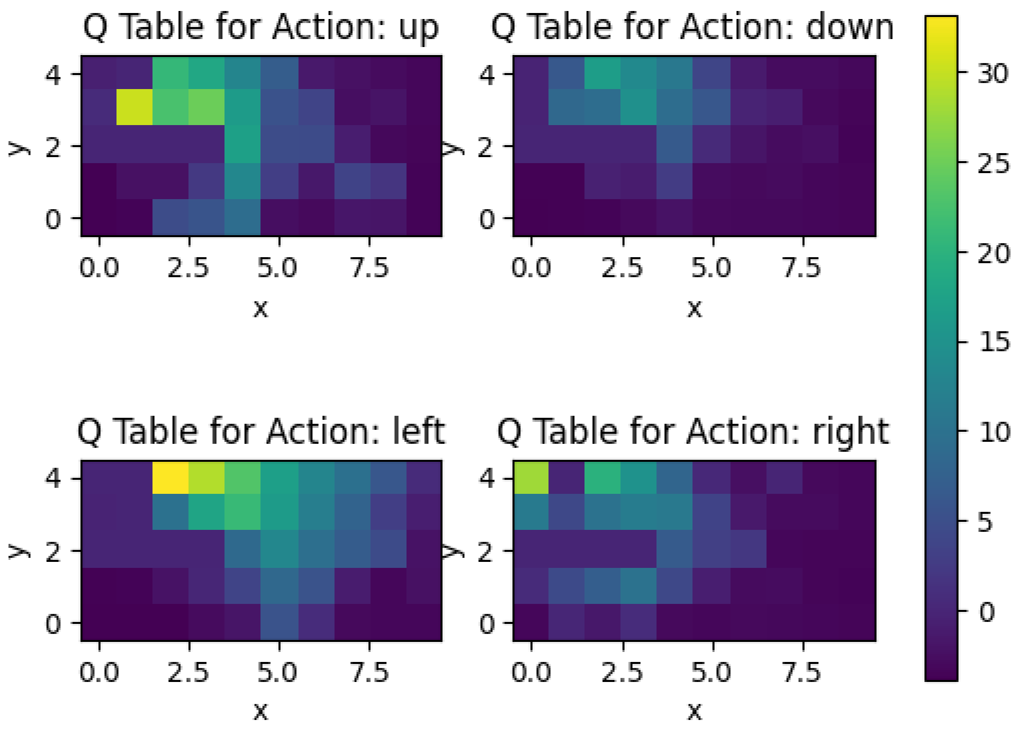


```
Episode 1/300, Total Reward: -235
Episode 51/300, Total Reward: 14
Episode 101/300, Total Reward: 20
Episode 151/300, Total Reward: 19
Episode 201/300, Total Reward: 19
Episode 251/300, Total Reward: 19
Episode 300/300, Total Reward: 15
```

Learning Curve:



Q-tables for each action except the ‘stay’ action:



Why it performed well:

I think the algorithm performed well because the environment was stationary, allowing the agent to explore sufficiently and the algorithm to converge to an optimal policy. Also, the epsilon-greedy policy with a high initial epsilon and the steady decay allowed the agent to discover high-reward states at the start of the algorithm and then use this knowledge to exploit the path that maximized its cumulative reward.

Part B:

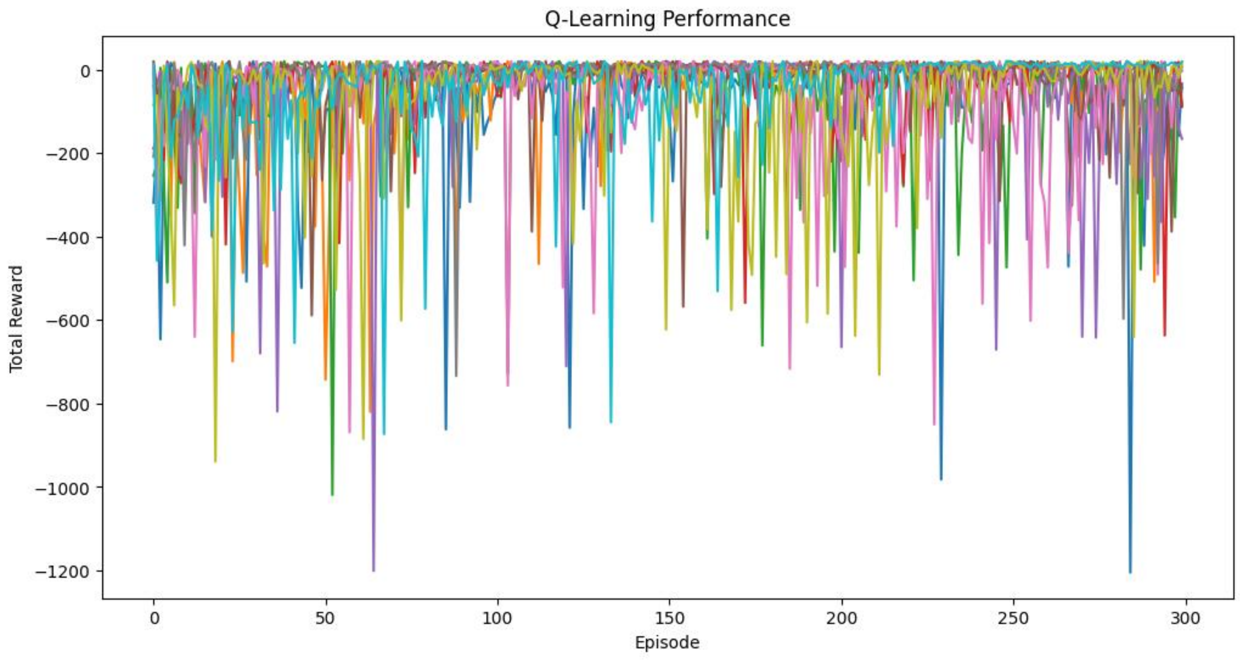
For part B, I chose to do option 1 (i.e., changing the goal position for every even timestep).

I did not modify the *move_door()* function in the code and just allowed the door to move randomly. Also, the parameters used in part A remained unchanged in this implementation.

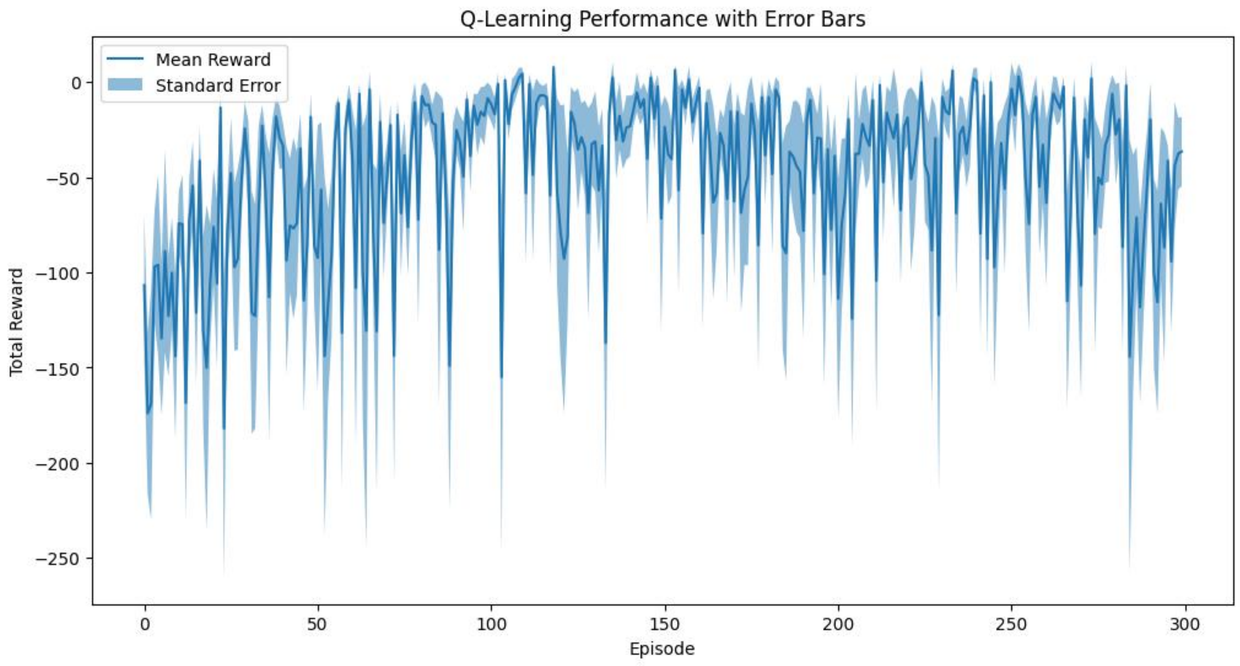
The algorithm performed significantly worse in the dynamic environment compared to part A. This is clearly shown in the plots below. The learning curves do not converge, and the mean rewards are lower with a larger standard error compared to the plots in part A. There is no concentration around a single goal in the heatmaps, either.

10 experiments plots:

Learning curves:

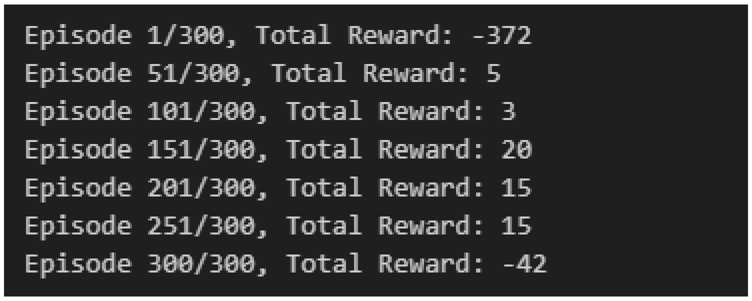


Standard Error:



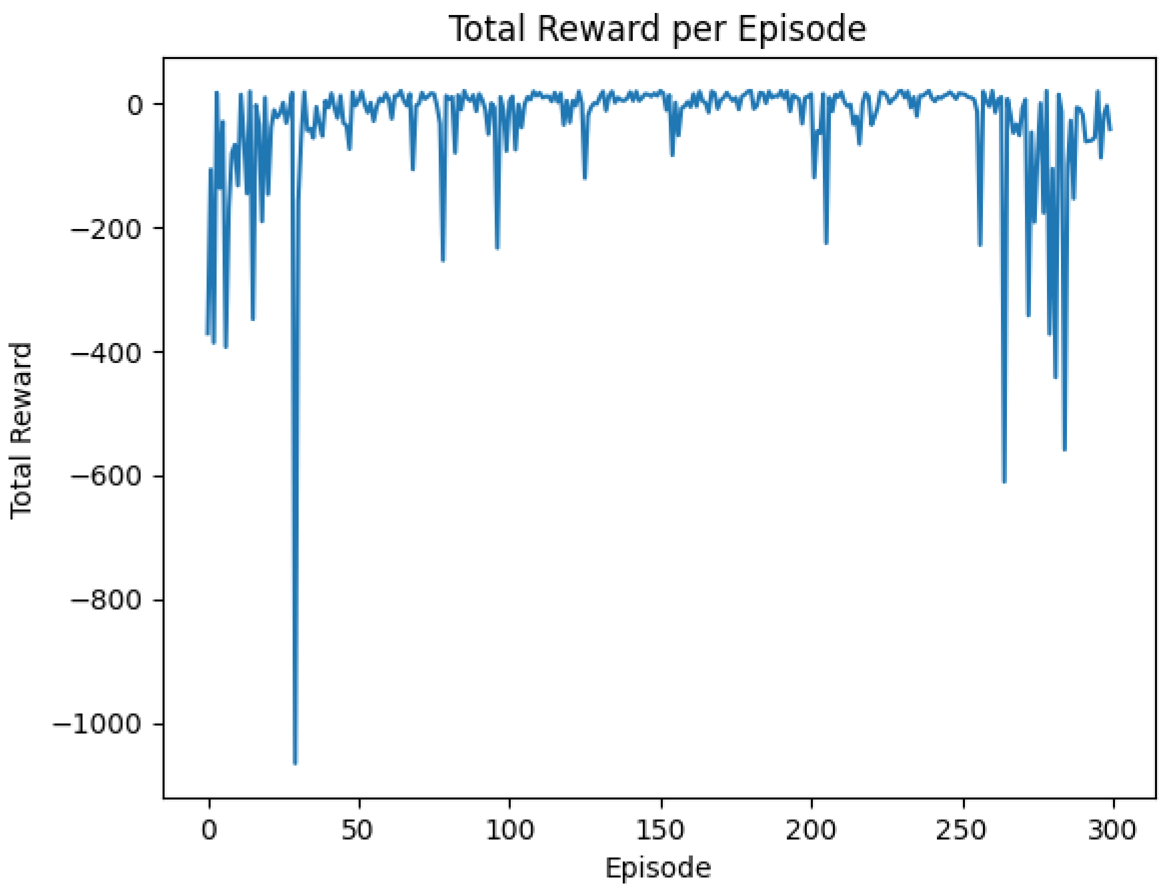
Single experiment:

Reward output:

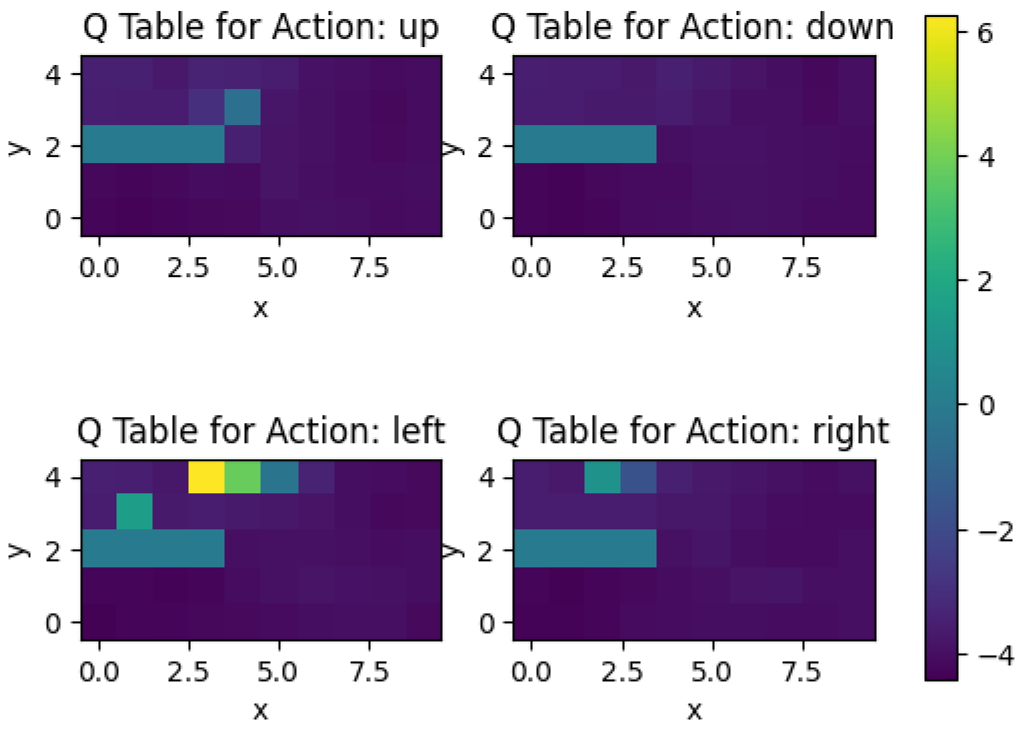


```
Episode 1/300, Total Reward: -372
Episode 51/300, Total Reward: 5
Episode 101/300, Total Reward: 3
Episode 151/300, Total Reward: 20
Episode 201/300, Total Reward: 15
Episode 251/300, Total Reward: 15
Episode 300/300, Total Reward: -42
```

Learning Curve:



Q-table for each action except 'stay':



Why it performed worse:

Moving the goal position at every even timestep introduces randomness into the environment, making it harder for the agent to learn a stable policy. The reward function constantly changes, a state-action pair that leads to a +20 reward in one episode might lead to a -1 reward in the next episode because the door has moved away. This hinders the agent ability to learn the true value of any state. My implementation of Q-learning in this homework does not handle dynamic environments well, as it is designed for a stationary MDP, where the rewards and transition probabilities are fixed. Therefore, does not learn a good policy as it fails to predict the behaviour of the door.