Originality Statement: I confirm that all work submitted as part of this document is my own. I used sources such as GeeksForGeeks, Python documentation, and ChatGPT for things like zip(), sort(), plotting and other Python nitpicky things

Parameters Used
Learning rate: 0.1
Discount factor: 0.9
Epsilon: 0.1
Episodes: 500
Time steps: 20

1 - Implement SARSA to solve this problem. How did the algorithms perform? Include learning curves and plots of the learned value tables.

Plots are included at the end of the writeup. SARSA appears to be much slower and takes a lot of missteps to reach the final solution. This is probably because it uses the q value and action of the next state instead of doing a greedy selection like Q-learning does. The heatmap shows this as well where it's not clear where the path it took is.

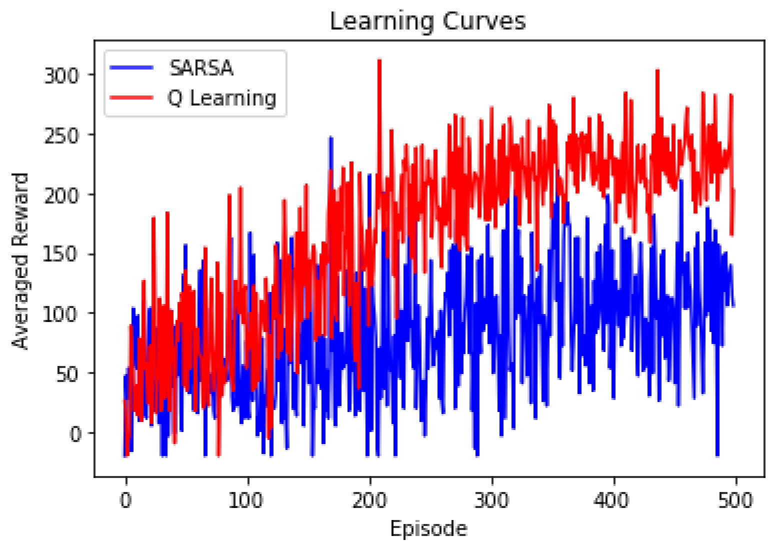
2 - Implement a Q-learning algorithm to solve this problem. How did the algorithms perform? How did solution compare to the SARSA solution? Discuss the implications of your results.

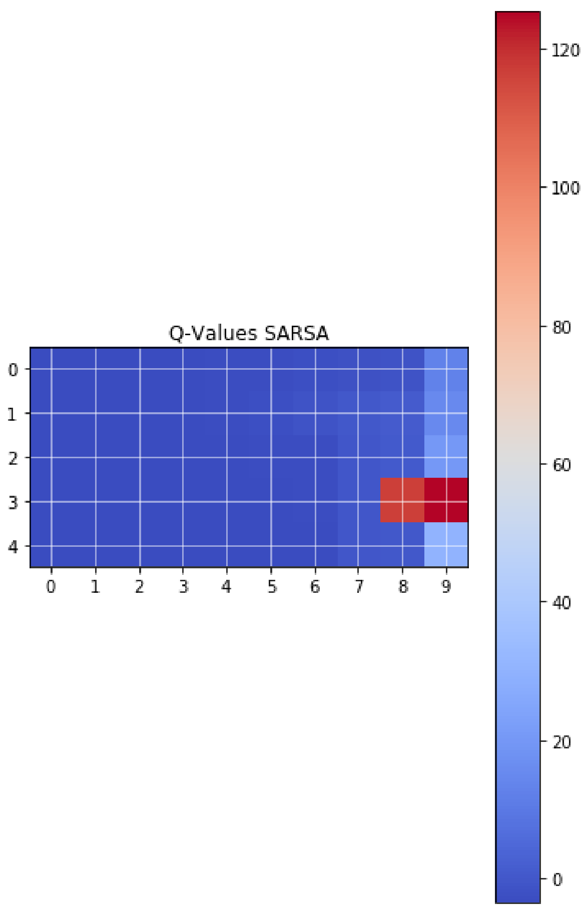
Q-learning outperformed the SARSA solution. Q-Learning appears to learn the policy much faster and as a result converges to a solution much faster than SARSA does. This is probably because of the greedy action selection which picks the best q value to follow. This is also visible in the heatmap where you can clearly see the path taken by the robot.

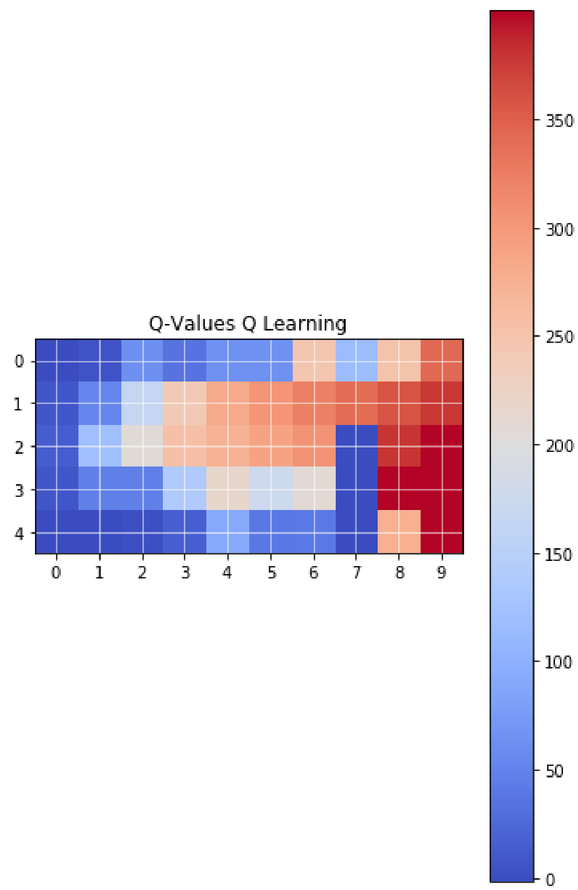
3 - Now consider the environment where the red door moves randomly by 1 cell every time step. Keep the initial starting location of the door the same as before. Use the EXACT same algorithms from problems 1 and 2 to solve this problem. How does the performance of the agent compare to problems 1 and 2? Does the agent learn a good policy? Describe your results and hypothesize why your agent performs the way it does. Speculate on how you may improve the performance of the agent. Again, plot learning curves and value tables.

Both the SARSA and Q-Learning algorithms performed significantly worse when the goal moves. This is possibly because each episode assumes that the goal is stationary so it tries going the same way. SARSA doesn't appear to learn a good policy with its averaged reward being negative more often than not. Q-learning does learn a good policy. Q-learning seems to outperform slightly and this is possibly because its greedy nature is more likely to try random actions. You can see this in the heat map where Q-learning seems to move more erratically but SARSA takes the same-ish path every time.

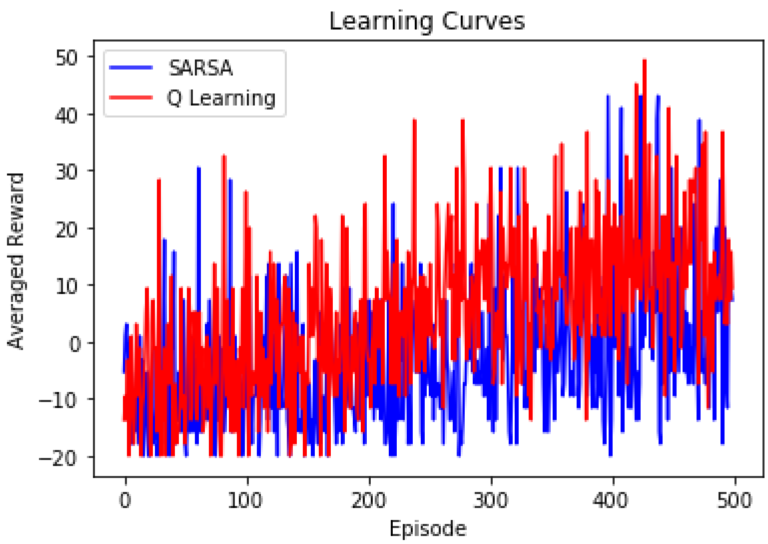
Learning Curves and Heatmaps for STATIONARY Goal

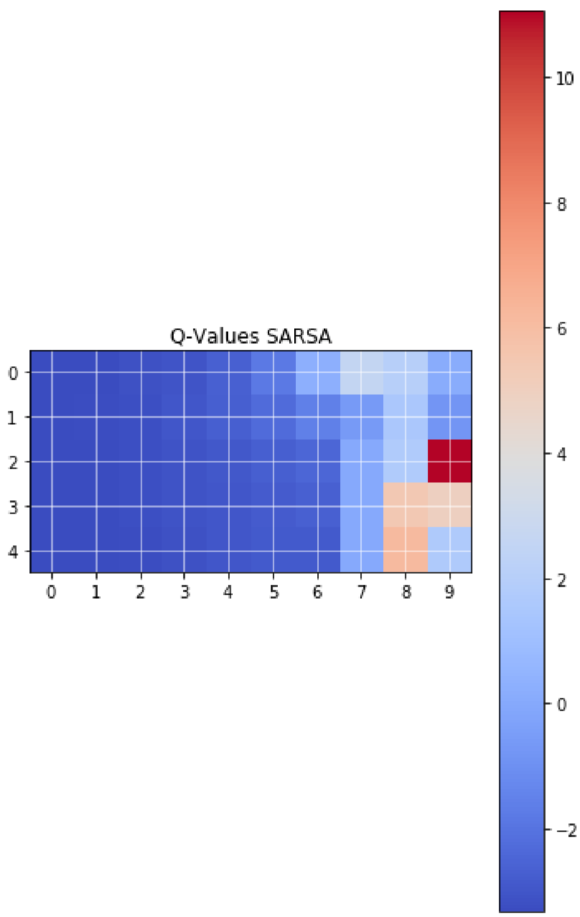


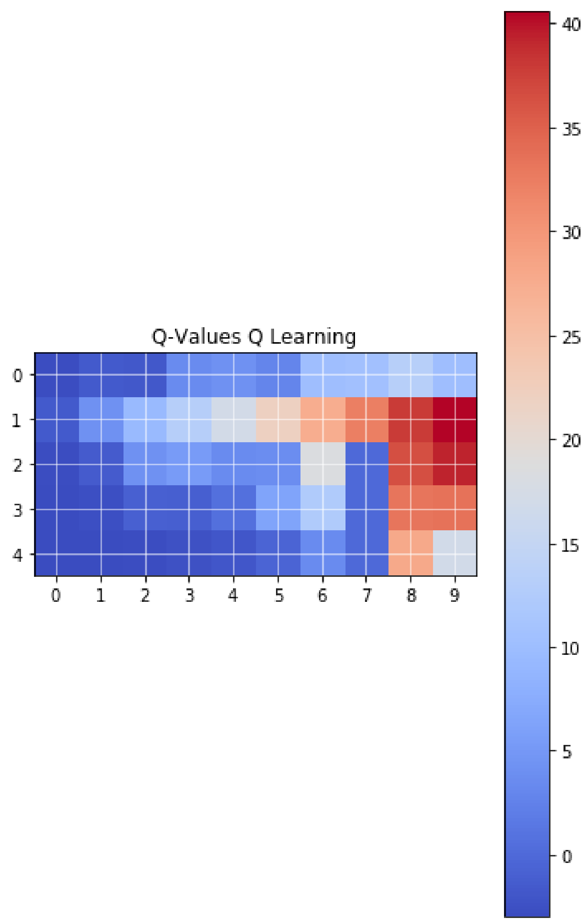




Learning Curves and Heatmaps for RANDOMLY MOVING Goal







In [58]:
```
from gridWorld import gridWorld
import random
import numpy as np
import matplotlib.pyplot as plt

STATIONARY = False

class SARSASolver:
    def __init__(self, env):
        self.q_table = {}
        self.alpha = 0.1
        self.gamma = 0.9
        self.epsilon = 0.1
        self.env = env
        self.action_options = ["up", "down", "left", "right", "stay"]


    def choose_action(self, state):
        # if the random is more than epsilon
        if random.random() < self.epsilon:
            # pick random action
            return random.choice(self.action_options)
        else:
        # else pick the best based on Q
            # create state_action pairs
            state_actions = [(tuple(state), action) for action in self.actic
            # determine the best q_value for all the pairs
            q_values = [self.q_table.get(pair, 0) for pair in state_actions]
            best_q = max(q_values)
            # pull the actions with the best q_value
            actions = []
            for i in range(len(state_actions)):
                if best_q == q_values[i]:
                    actions.append(self.action_options[i])
            # pick a random from that list if multiple best_q
            return random.choice(actions)

    def learn(self, num_episodes):
        total_rewards = []
        for learning_epoch in range(num_episodes):
            state = env.reset()
            total_reward = 0                 #every episode, reset the envir
            for time_step in range(20):
                action = self.choose_action(state)

                if (STATIONARY):
                    next_state,reward=env.step(action)  #the action is taker
                else:
                    next_state,reward=env.step(action,rng_door=True)  #the a
                next_action = self.choose_action(state) #learner chooses one

                # learning
                # pull current and next q_values
                current_q = self.q_table.get((tuple(state), action), 0)
                next_q = self.q_table.get((tuple(next_state), next_action),
                # update q_table
                self.q_table[(tuple(state), action)] = current_q + self.alph
```

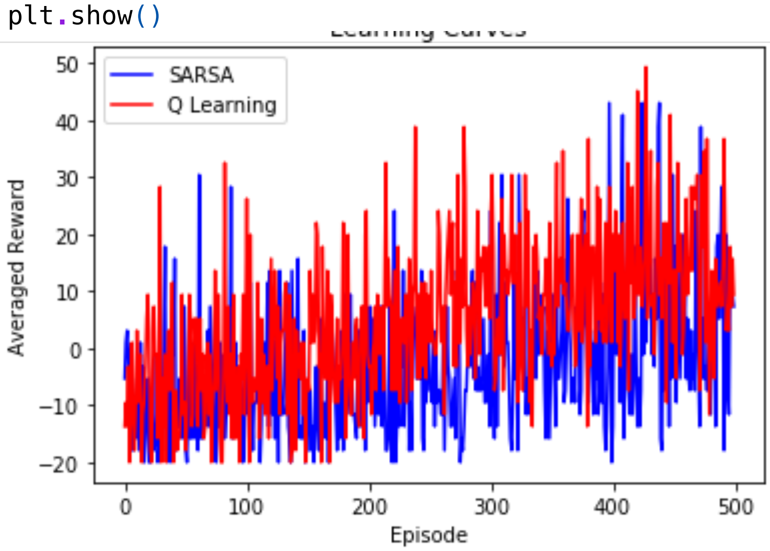
```
                # move to next state
                state = next_state
                action = next_action

                # update total reward
                total_reward = total_reward + reward
            total_rewards.append(total_reward)
        return total_rewards




class QLearnerSolver:
    def __init__(self,env):
        self.q_table = {}
        self.alpha = 0.1
        self.gamma = 0.95
        self.epsilon = 0.1
        self.env = env
        self.action_options = ["up", "down", "left", "right", "stay"]


    def choose_action(self, state):
        # if the random is more than epsilon
        if random.random() < self.epsilon:
            # pick random action
            return random.choice(self.action_options)
        else:
        # else pick the best based on Q
            # create state_action pairs
            state_actions = [(tuple(state), action) for action in self.actio
            # determine the best q_value for all the pairs
            q_values = [self.q_table.get(pair, 0) for pair in state_actions]
            best_q = max(q_values)
            # pull the actions with the best q_value
            actions = []
            for i in range(len(state_actions)):
                if best_q == q_values[i]:
                    actions.append(self.action_options[i])
            # pick a random from that list if multiple best_q
            return random.choice(actions)

    def learn(self, num_episodes):
        total_rewards = []
        for learning_epoch in range(num_episodes):
            state = env.reset()
            total_reward = 0                 #every episode, reset the envir
            for time_step in range(20):
                action = self.choose_action(state) #learner chooses one of t
                if (STATIONARY):
                    next_state,reward=env.step(action)  #the action is taken
                else:
                    next_state,reward=env.step(action,rng_door=True)  #the a
                next_action = self.choose_action(state) #learner chooses one

                # learning
                # pull current and next q_values
                current_q = self.q_table.get((tuple(state), action), 0)
                max_q_next = max(self.q_table.get((tuple(next_state), a), 0)
```

```
                # update q_table
                self.q_table[(tuple(state), action)] = current_q + self.alph
                # move to next state
                state = next_state
                action = next_action

                # update total reward
                total_reward = total_reward + reward
            total_rewards.append(total_reward)
        return total_rewards


if __name__=="__main__":
    num_episodes = 500
    num_trials = 10
    #example usage for a gym-like environment
    #state: [x,y] coordinate of the agent
    #actions: ["up","down","left","right"] directions the agent can move
    env=gridWorld()
    sarsa_all_rewards =[]
    q_all_rewards = []
    for trial in range(num_trials):
        learner1=SARSASolver(env)
        learner2=QLearnerSolver(env)
        sarsa_total_rewards = learner1.learn(num_episodes)
        q_total_rewards = learner2.learn(num_episodes)

        sarsa_all_rewards.append(sarsa_total_rewards)
        q_all_rewards.append(q_total_rewards)

    # caluclate averaged rewards
    sarsa_avg_rewards = []
    for reward_ep in zip(*sarsa_all_rewards):
        sarsa_avg_rewards.append(sum(reward_ep)/num_trials)

    q_avg_rewards = []
    for reward_ep in zip(*q_all_rewards):
        q_avg_rewards.append(sum(reward_ep)/num_trials)

    # calculate q-values for heat maps
    sarsa_q_values = np.zeros((10, 5, 5))
    qlearning_q_values = np.zeros((10, 5, 5))

    for x in range(10):
        for y in range(5):
            for index, action in enumerate(["up", "down", "left", "right","s
                sarsa_q_values[x, y, index] = learner1.q_table.get(((x, y),
                qlearning_q_values[x, y, index] = learner2.q_table.get(((x,
```

In [59]:

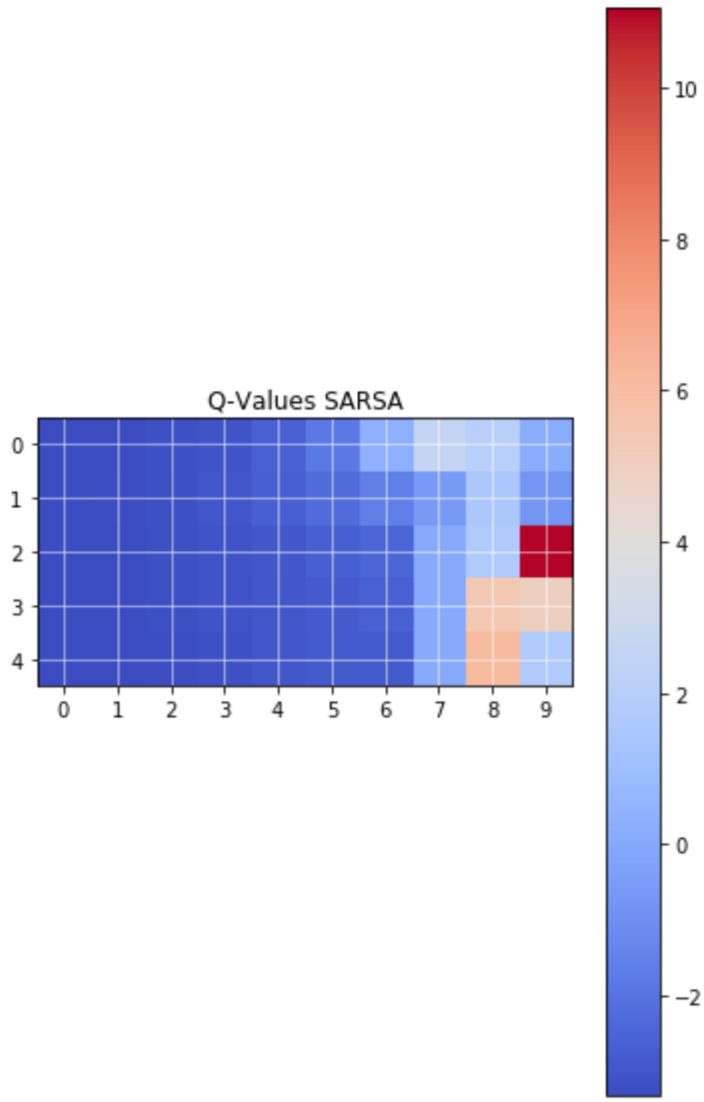
```
plt.plot(range(num_episodes), sarsa_avg_rewards, c='b', label="SARSA")
plt.plot(range(num_episodes), q_avg_rewards, c='r', label="Q Learning")
plt.legend()
plt.xlabel("Episode")
plt.ylabel("Averaged Reward")
plt.title("Learning Curves")
```

```
plt.show()
```

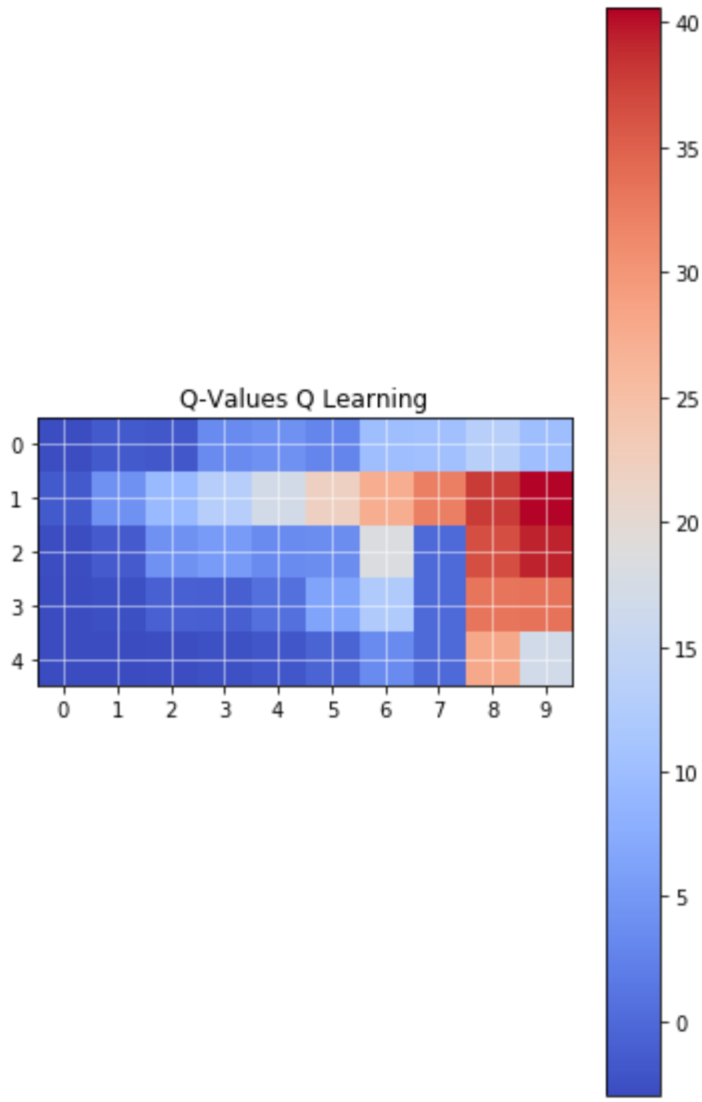


In [60]:

```
plt.figure(figsize=(6, 10))
plt.imshow(np.flip(np.transpose(sarsa_q_values.max(axis=2)),0), cmap='coolwa
plt.colorbar()
plt.title("Q-Values SARSA")
plt.xticks(range(10))
plt.yticks(range(5))
plt.grid(color='w', linestyle='-', linewidth=0.5)
plt.show()
```



In [61]:
```
plt.figure(figsize=(6, 10))
plt.imshow(np.flip(np.transpose(qlearning_q_values.max(axis=2)),0), cmap='co
plt.colorbar()
plt.title("Q-Values Q Learning")
plt.xticks(range(10))
plt.yticks(range(5))
plt.grid(color='w', linestyle='-', linewidth=0.5)
plt.show()
```


Q-Values Q Learning
0
1
2
3
4
0 1 2 3 4 5 6 7 8 9
40
35
30
25
20
15
10
5
0