CS6700: Reinforcement Learning - Tutorial 4 (Q-Learning and SARSA)

ME20B072

Your tasks are as follows:

1. Complete code for ϵ-greedy and softmax action selection policy
2. Complete update equation for SARSA - train and visualize an agent
3. Analyze performance of SARSA - Plot total reward & steps taken per episode (averaged across 5 runs)
4. Complete update equation for Q-Learning - train and visualize an agent
5. Analyze performance of Q-Learning - Plot total reward & steps taken per episode (averaged across 5 runs)

```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from IPython.display import clear_output
%matplotlib inline
```

Problem Statement

In this section we will implement tabular SARSA and Q-learning algorithms for a grid world navigation task.

Environment details

The agent can move from one grid coordinate to one of its adjacent grids using one of the four actions: UP, DOWN, LEFT and RIGHT. The goal is to go from a randomly assigned starting position to goal position.

Actions that can result in taking the agent off the grid will not yield any effect. Lets look at the environment.

```
DOWN = 0
UP = 1
LEFT = 2
RIGHT = 3
actions = [DOWN, UP, LEFT, RIGHT]
```

Let us construct a grid in a text file.

```
!cat grid_world2.txt
```

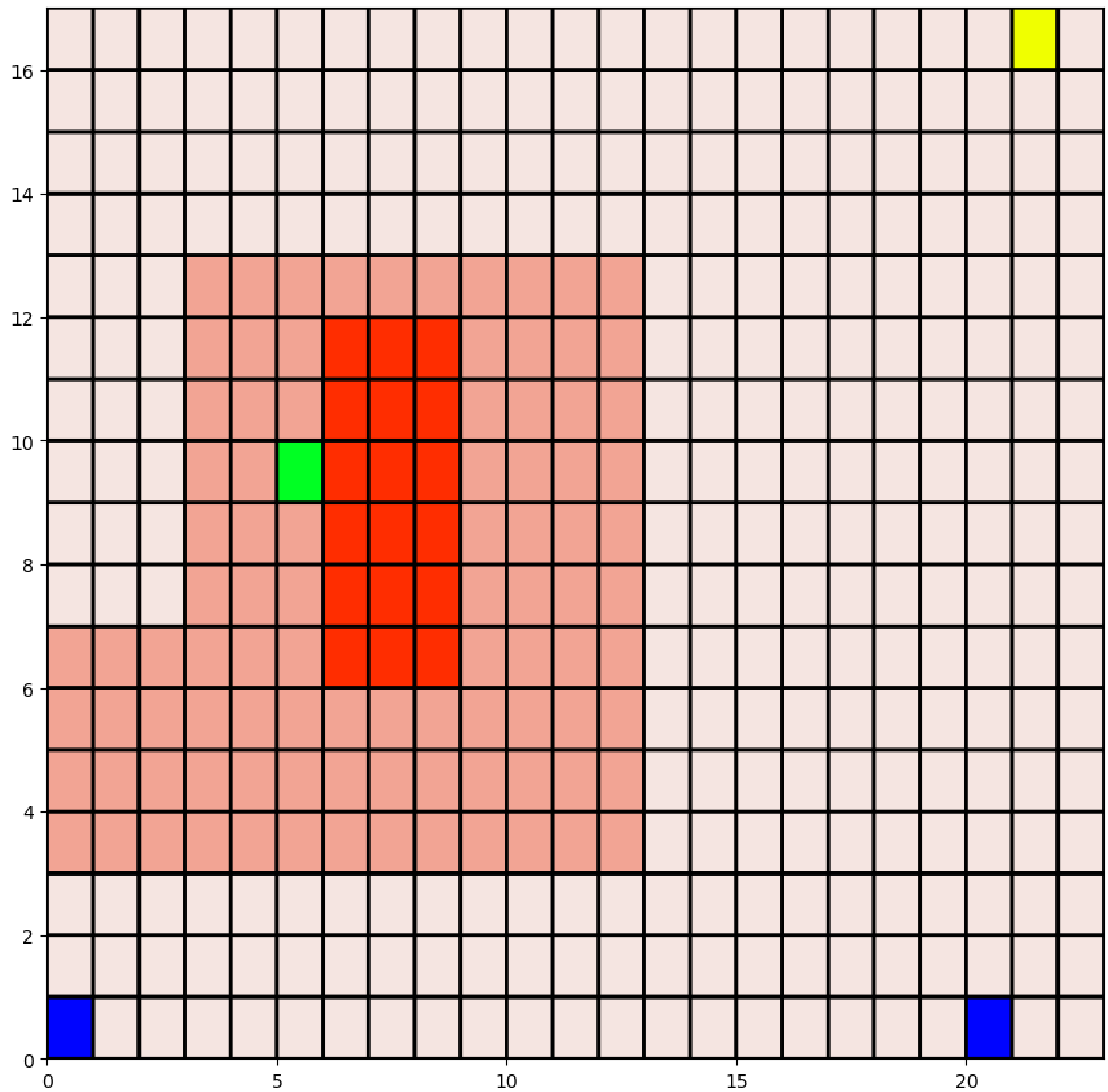
```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 2 2 2 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 2 2 2 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 2 2 2 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 2 2 2 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 2 2 2 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 2 2 2 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 1 1 1 1 1 1 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
```

This is a 17×23 grid. The reward when an agent goes to a cell is negative of the value in that position in the text file (except if it is the goal cell). We will define the goal reward as 100. We will also fix the maximum episode length to 10000.

Now let's make it more difficult. We add stochasticity to the environment: with probability 0.2 agent takes a random action (which can be other than the chosen action). There is also a westerly wind blowing (to the right). Hence, after every time-step, with probability 0.5 the agent also moves an extra step to the right.

Now let's plot the grid world.

```python
world = 'grid_world2.txt'
goal_reward = 100
start_states = [(0,0), (0,20), (16,21)]
goal_states=[(9,5)]
max_steps=10000

from grid_world import GridWorldEnv, GridWorldWindyEnv

env = GridWorldEnv(world, goal_reward=goal_reward,
start_states=start_states, goal_states=goal_states,
                max_steps=max_steps, action_fail_prob=0.2)
plt.figure(figsize=(10, 10))
# Go UP
env.step(UP)
env.render(ax=plt, render_agent=True)
```



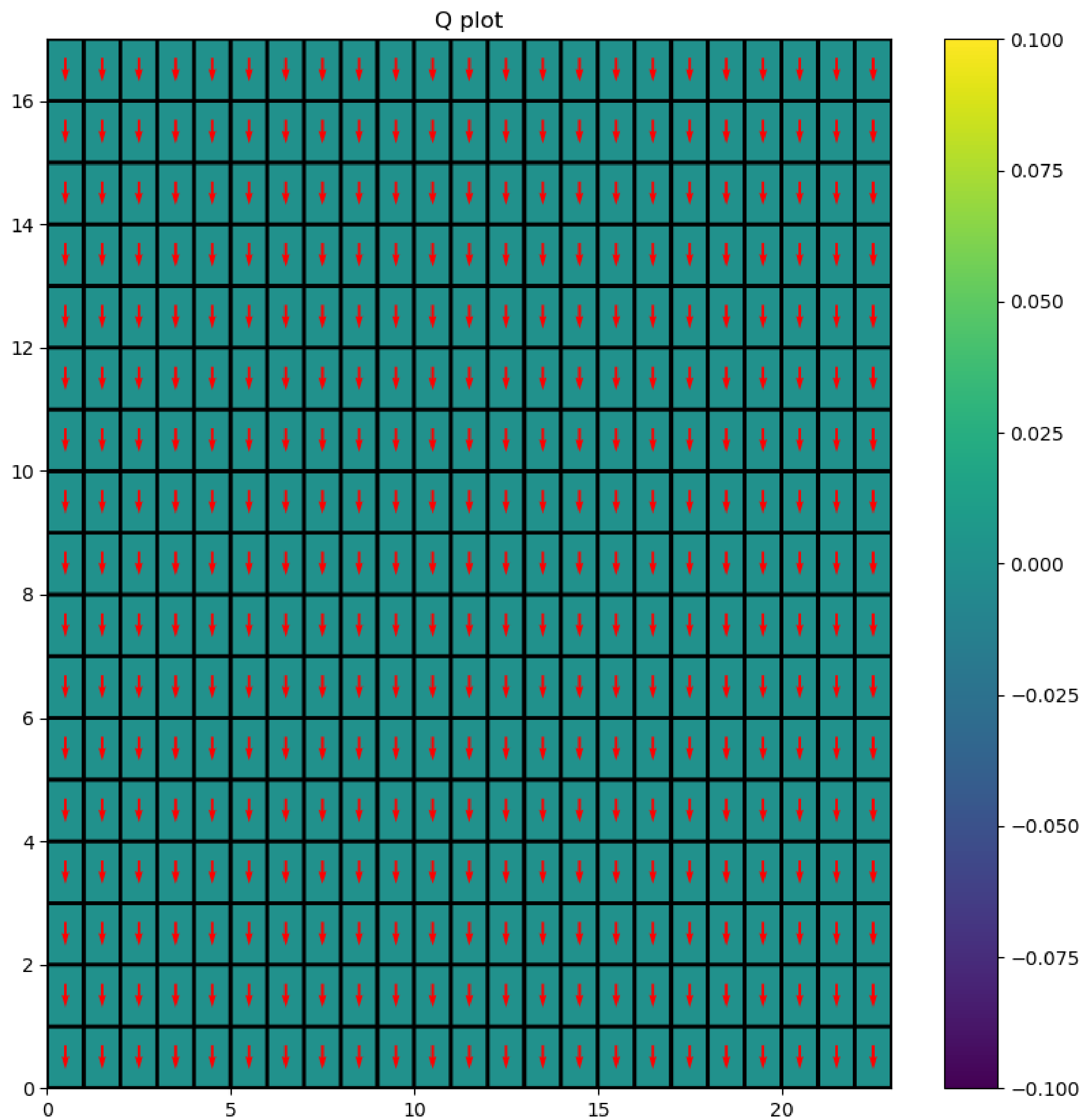
Legend

- *Blue* is the **start state**.
- *Green* is the **goal state**.
- *Yellow* is current **state of the agent**.
- *Redness* denotes the extent of **negative reward**.

Q values

We can use a 3D array to represent Q values. The first two indices are X, Y coordinates and last index is the action.

```python
from grid_world import plot_Q

Q = np.zeros((env.grid.shape[0], env.grid.shape[1],
len(env.action_space)))

plot_Q(Q)

Q.shape
```




```
(17, 23, 4)
```

Exploration strategies

1. Epsilon-greedy
2. Softmax

```
from scipy.special import softmax

seed = 42
rg = np.random.RandomState(seed)

# Epsilon greedy
def choose_action_epsilon(Q, state, epsilon, rg=rg):
    if not Q[state[0], state[1]].any() or rg.rand() < epsilon : #
TODO: eps greedy condition
        return rg.randint(0, len(Q[state[0], state[1]])) # TODO:
return random action
    else:
        return np.argmax(Q[state[0], state[1]])  # TODO: return best
action

# Softmax
def choose_action_softmax(Q, state, rg=rg):
    return rg.choice(len(Q[state[0], state[1]]), p=softmax(Q[state[0],
state[1]])) # TODO: return random action with selection probability
```

SARSA

Now we implement the SARSA algorithm.

Recall the update rule for SARSA: \begin{equation} Q(s_t,a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \end{equation}

Hyperparameters

So we have som hyperparameters for the algorithm:

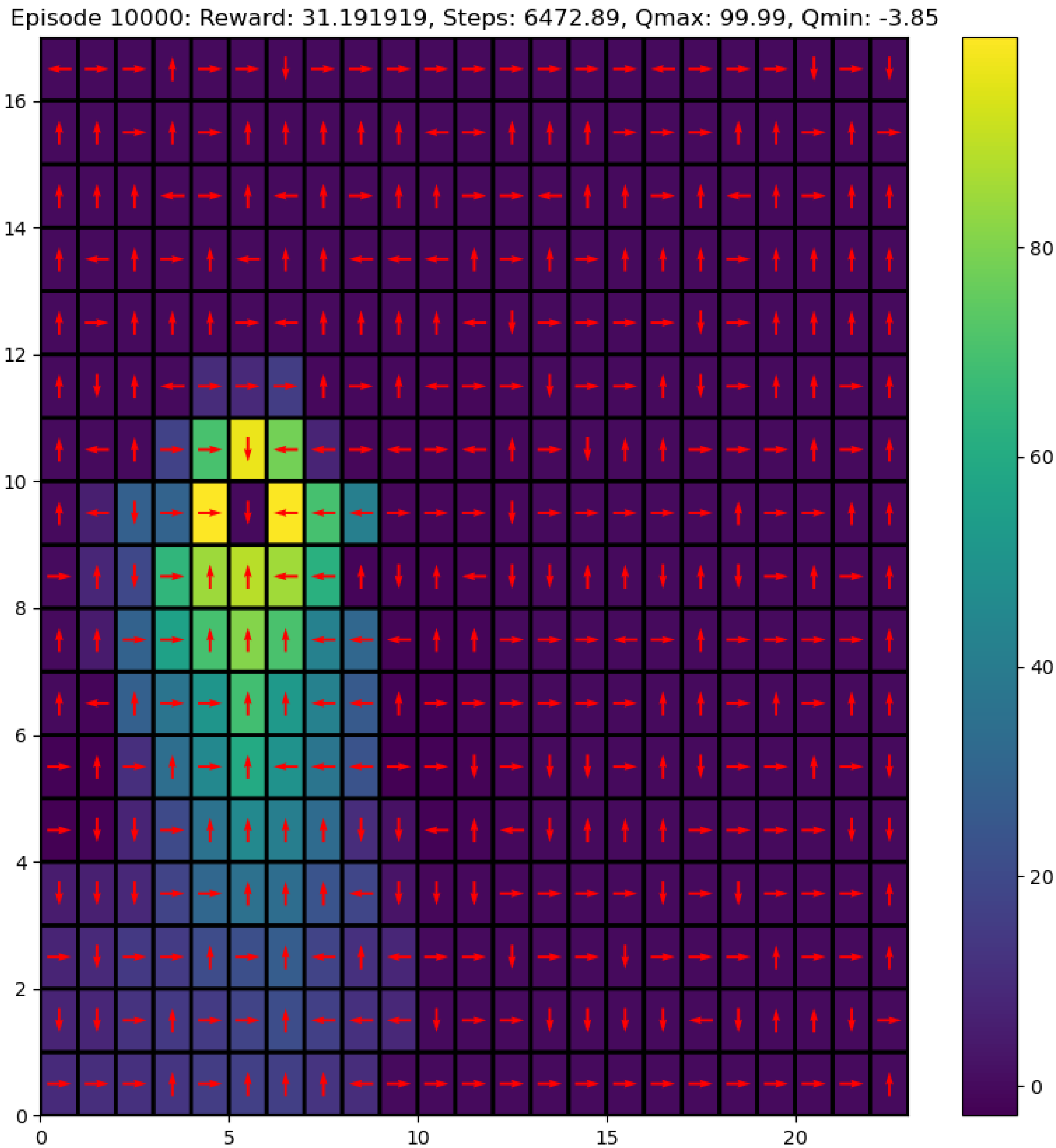
- α
- number of *episodes*.
- ϵ: For epsilon greedy exploration

```
# initialize Q-value
Q = np.zeros((env.grid.shape[0], env.grid.shape[1],
len(env.action_space)))

alpha0 = 0.4
gamma = 0.9
episodes = 10000
epsilon0 = 0.1
```

Let's implement SARSA

```python
print_freq = 100

def sarsa(env, Q, gamma = 0.9, plot_heat = False,
choose_action=choose_action_softmax):

    episode_rewards = np.zeros(episodes)
    steps_to_completion = np.zeros(episodes)
    if plot_heat:
        clear_output(wait=True)
        plot_Q(Q)
    epsilon = epsilon0
    alpha = alpha0
    for ep in tqdm(range(episodes)):
        tot_reward, steps = 0, 0

        # Reset environment
        state = env.reset()
        action = choose_action(Q, state)
        done = False
        while not done:
            state_next, reward, done = env.step(action)
            action_next = choose_action(Q, state_next)

            # TODO: update equation
            Q[state[0], state[1], action] += alpha * (reward + gamma *
Q[state_next[0], state_next[1], action_next] - Q[state[0], state[1],
action])

            tot_reward += reward
            steps += 1

            state, action = state_next, action_next

        episode_rewards[ep] = tot_reward
        steps_to_completion[ep] = steps

        if (ep+1)%print_freq == 0 and plot_heat:
            clear_output(wait=True)
            plot_Q(Q, message = "Episode %d: Reward: %f, Steps: %.2f,
Qmax: %.2f, Qmin: %.2f"%(ep+1, np.mean(episode_rewards[ep-
print_freq+1:ep]),

np.mean(steps_to_completion[ep-print_freq+1:ep]),

Q.max(), Q.min()))

    return Q, episode_rewards, steps_to_completion
```

```
Q, rewards, steps = sarsa(env, Q, gamma = gamma, plot_heat=True,
choose_action=lambda Q,state:
choose_action_epsilon(Q,state,epsilon0,rg))
```

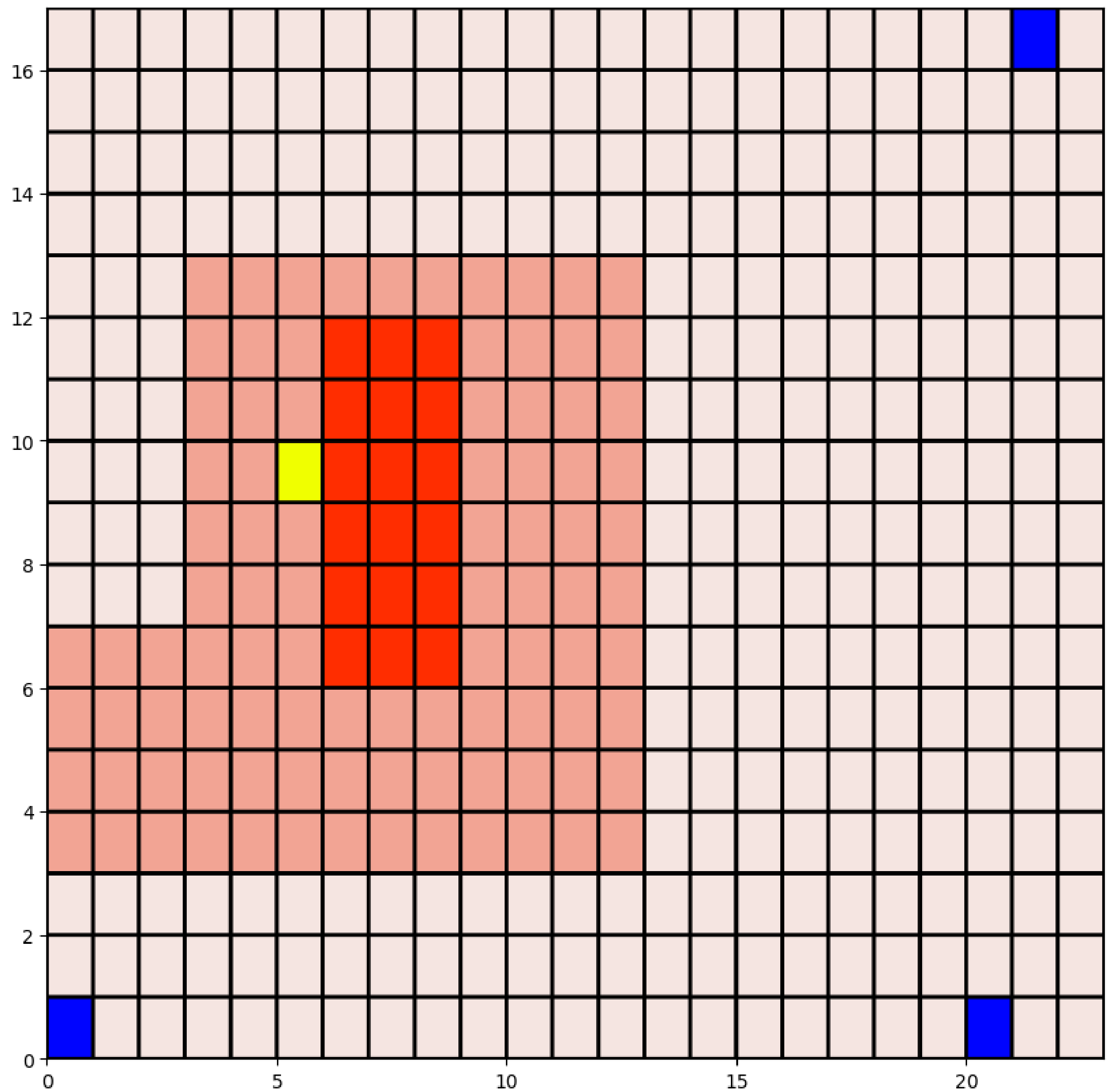



```
100%|██████████| 10000/10000 [10:34<00:00, 15.75it/s]
```

Visualizing the policy

Now let's see the agent in action. Run the below cell (as many times) to render the policy;

```python
from time import sleep

state = env.reset()
done = False
steps = 0
tot_reward = 0
while not done:
    clear_output(wait=True)
    state, reward, done = env.step(Q[state[0], state[1]].argmax())
    plt.figure(figsize=(10, 10))
    env.render(ax=plt, render_agent=True)
    plt.show()
    steps += 1
    tot_reward += reward
    sleep(0.2)
print("Steps: %d, Total Reward: %d"%(steps, tot_reward))
```




```
Steps: 28, Total Reward: 94
```

Analyzing performance of the policy

We use two metrics to analyze the policies:

1. Average steps to reach the goal
2. Total rewards from the episode

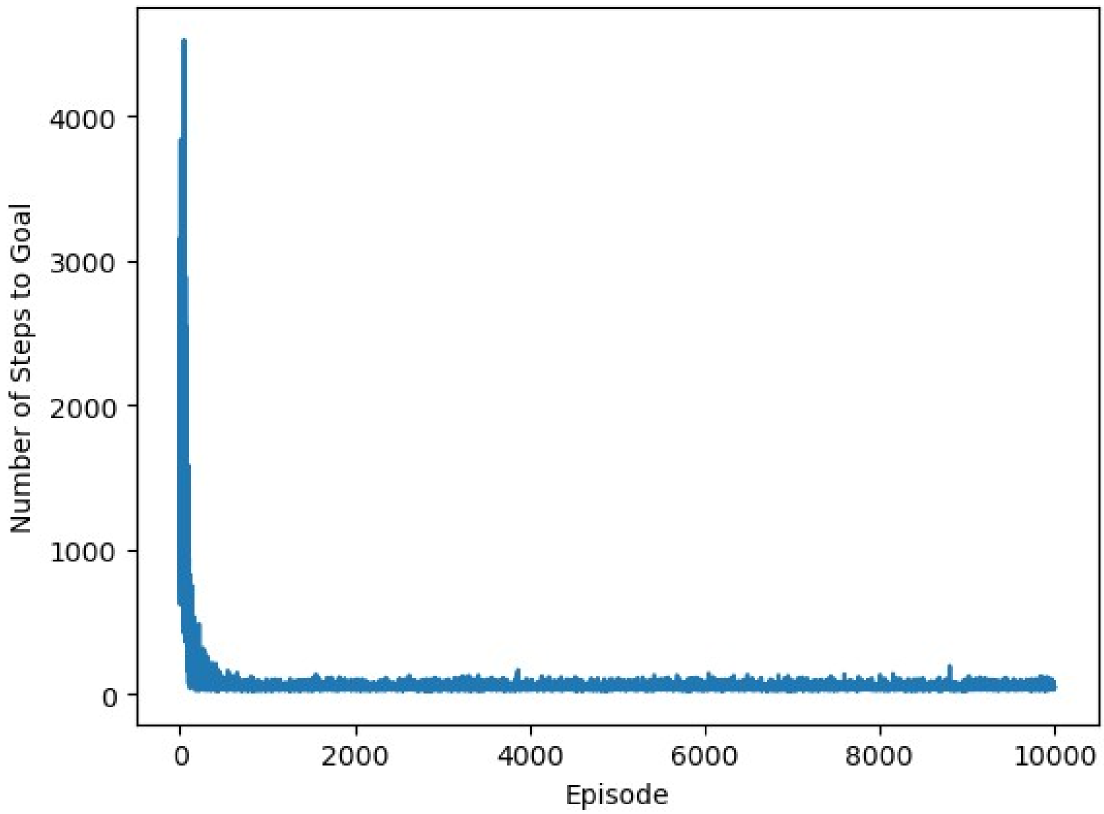
To ensure, we account for randomness in environment and algorithm (say when using epsilon-greedy exploration), we run the algorithm for multiple times and use the average of values over all runs.

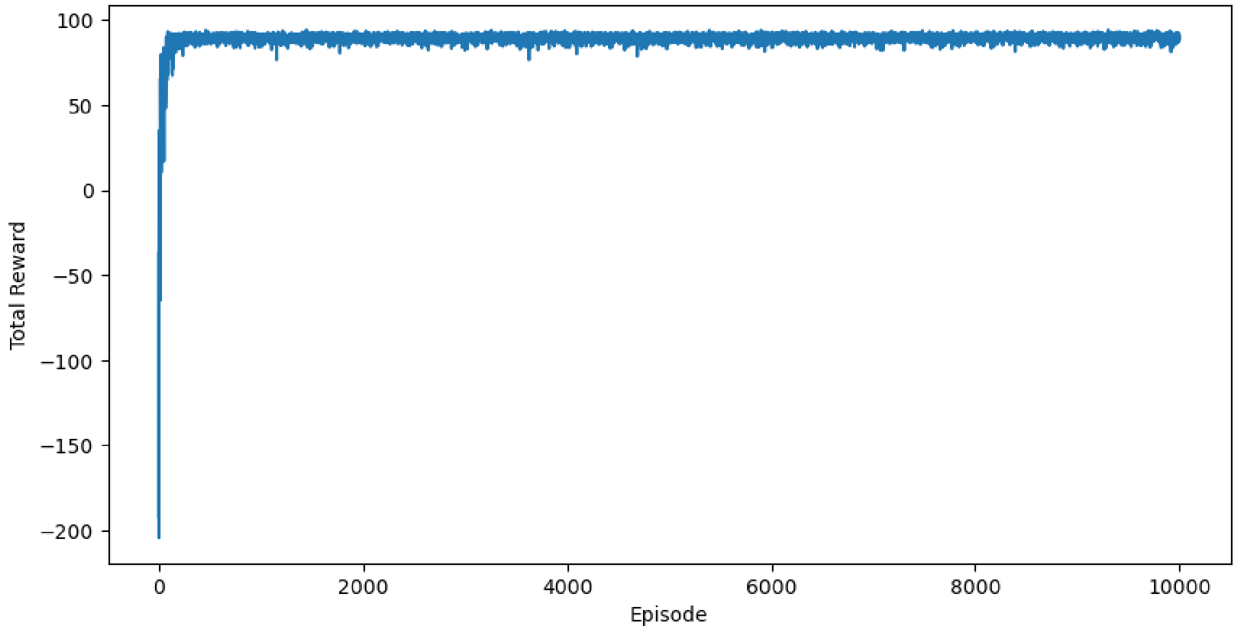
```
num_expts = 5
reward_avgs, steps_avgs = [], []

for i in range(num_expts):
    print("Experiment: %d"%(i+1))
    Q = np.zeros((env.grid.shape[0], env.grid.shape[1],
len(env.action_space)))
    rg = np.random.RandomState(i)

    # TODO: run sarsa, store metrics
    Q, episode_rewards, steps_to_completion = sarsa(env, Q, gamma=0.9,
plot_heat=False, choose_action=choose_action_softmax)

    reward_avgs.append(episode_rewards)
    steps_avgs.append(steps_to_completion)
```

```
Experiment: 1

100%|██████████| 10000/10000 [00:51<00:00, 192.96it/s]

Experiment: 2

100%|██████████| 10000/10000 [00:22<00:00, 446.02it/s]

Experiment: 3

100%|██████████| 10000/10000 [00:30<00:00, 330.44it/s]

Experiment: 4

100%|██████████| 10000/10000 [00:22<00:00, 454.25it/s]

Experiment: 5

100%|██████████| 10000/10000 [00:33<00:00, 300.67it/s]
```

```
# TODO: visualize individual metrics vs episode count (averaged across
multiple run(s))

# plt.figure()
# plt.xlabel('Episode')
# plt.ylabel('Number of steps to Goal')
# plt.show()

# plt.figure()
# plt.xlabel('Episode')
# plt.ylabel('Total Reward')
# plt.show()

plt.figure()
plt.plot(np.mean(steps_avgs,axis=0))
plt.xlabel('Episode')
```

```
plt.ylabel('Number of Steps to Goal')
plt.show()

plt.figure()
plt.plot(np.mean(reward_avgs,axis=0))
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.show()
```



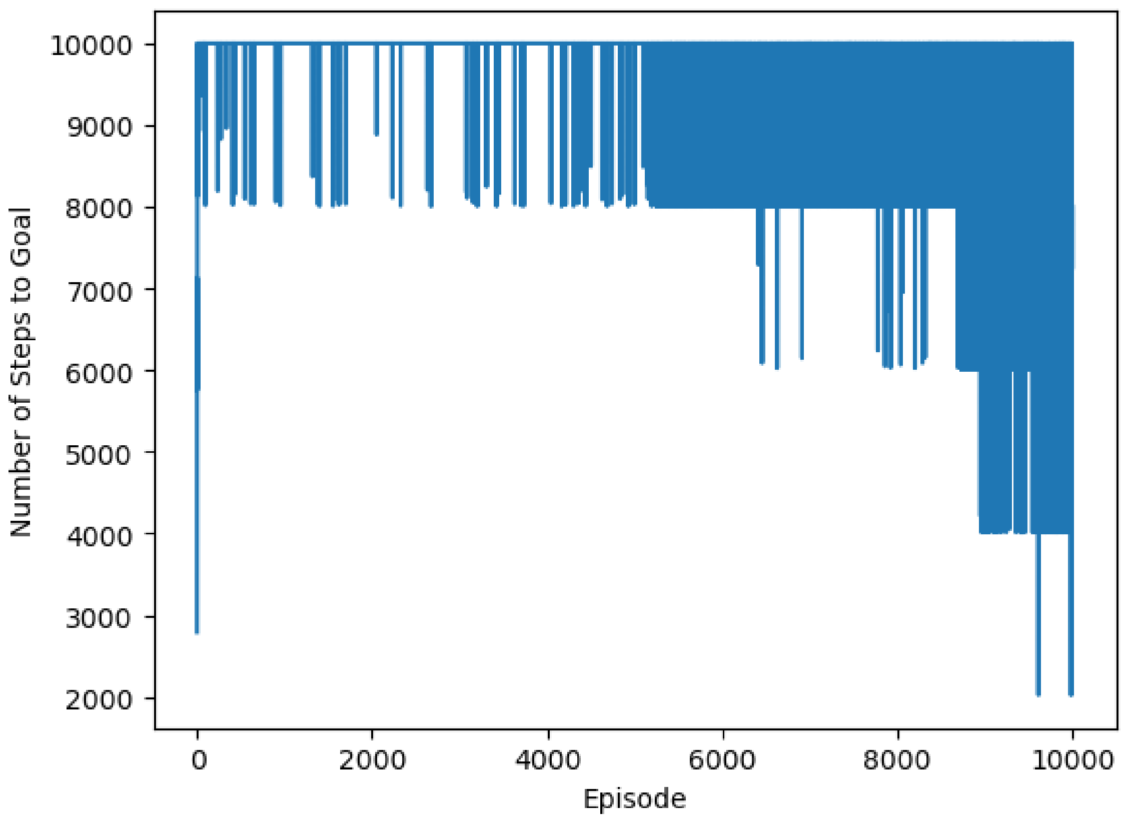


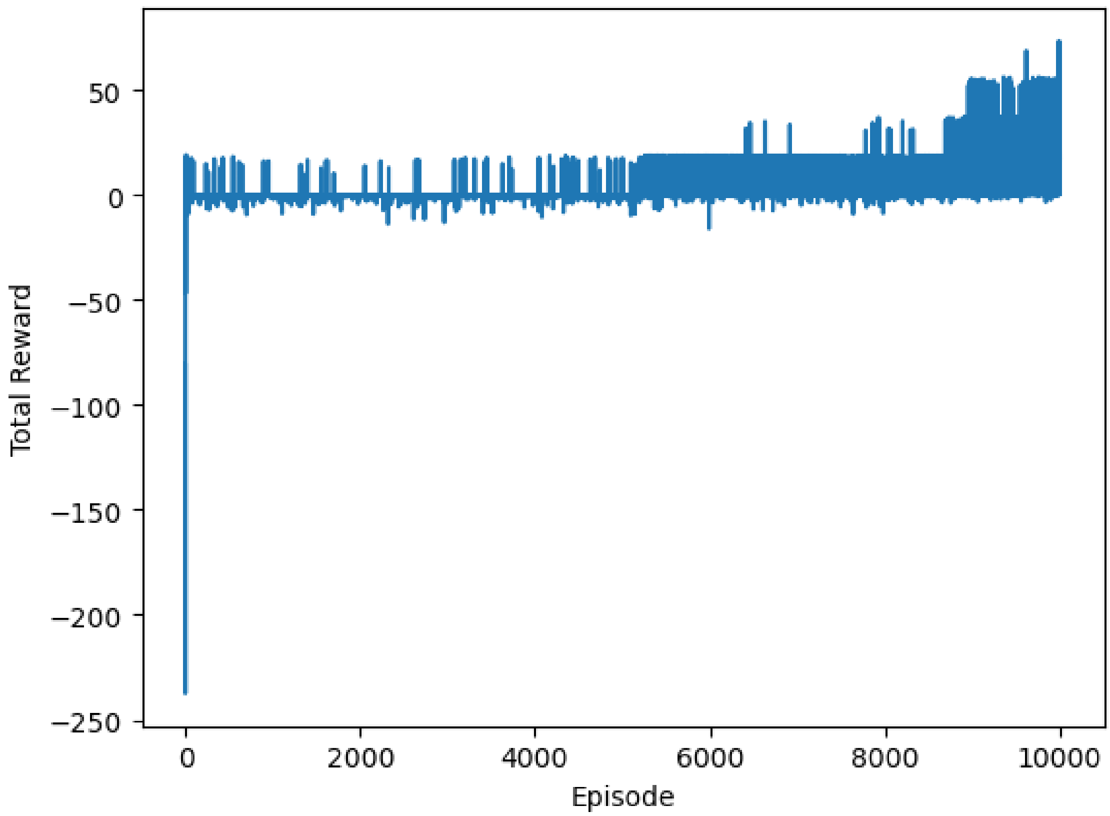

```python
num_expts = 5
reward_avgs, steps_avgs = [], []

for i in range(num_expts):
    print("Experiment: %d"%(i+1))
    Q = np.zeros((env.grid.shape[0], env.grid.shape[1],
len(env.action_space)))
    rg = np.random.RandomState(i)

    # TODO: run sarsa, store metrics
    Q, episode_rewards, steps_to_completion = sarsa(env, Q, gamma=0.9,
plot_heat=False, choose_action=lambda Q,state:
choose_action_epsilon(Q,state,epsilon0,rg))

    reward_avgs.append(episode_rewards)
    steps_avgs.append(steps_to_completion)
```

```
Experiment: 1

100%|██████████| 10000/10000 [16:34<00:00, 10.06it/s]

Experiment: 2

100%|██████████| 10000/10000 [13:10<00:00, 12.65it/s]

Experiment: 3

100%|██████████| 10000/10000 [14:15<00:00, 11.69it/s]

Experiment: 4

100%|██████████| 10000/10000 [24:23<00:00,  6.83it/s]
```

```
Experiment: 5

100%|██████████| 10000/10000 [14:26<00:00, 11.55it/s]
```

```python
plt.figure()
plt.plot(np.mean(steps_avgs,axis=0))
plt.xlabel('Episode')
plt.ylabel('Number of Steps to Goal')
plt.show()

plt.figure()
plt.plot(np.mean(reward_avgs,axis=0))
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.show()
```





Q-Learning

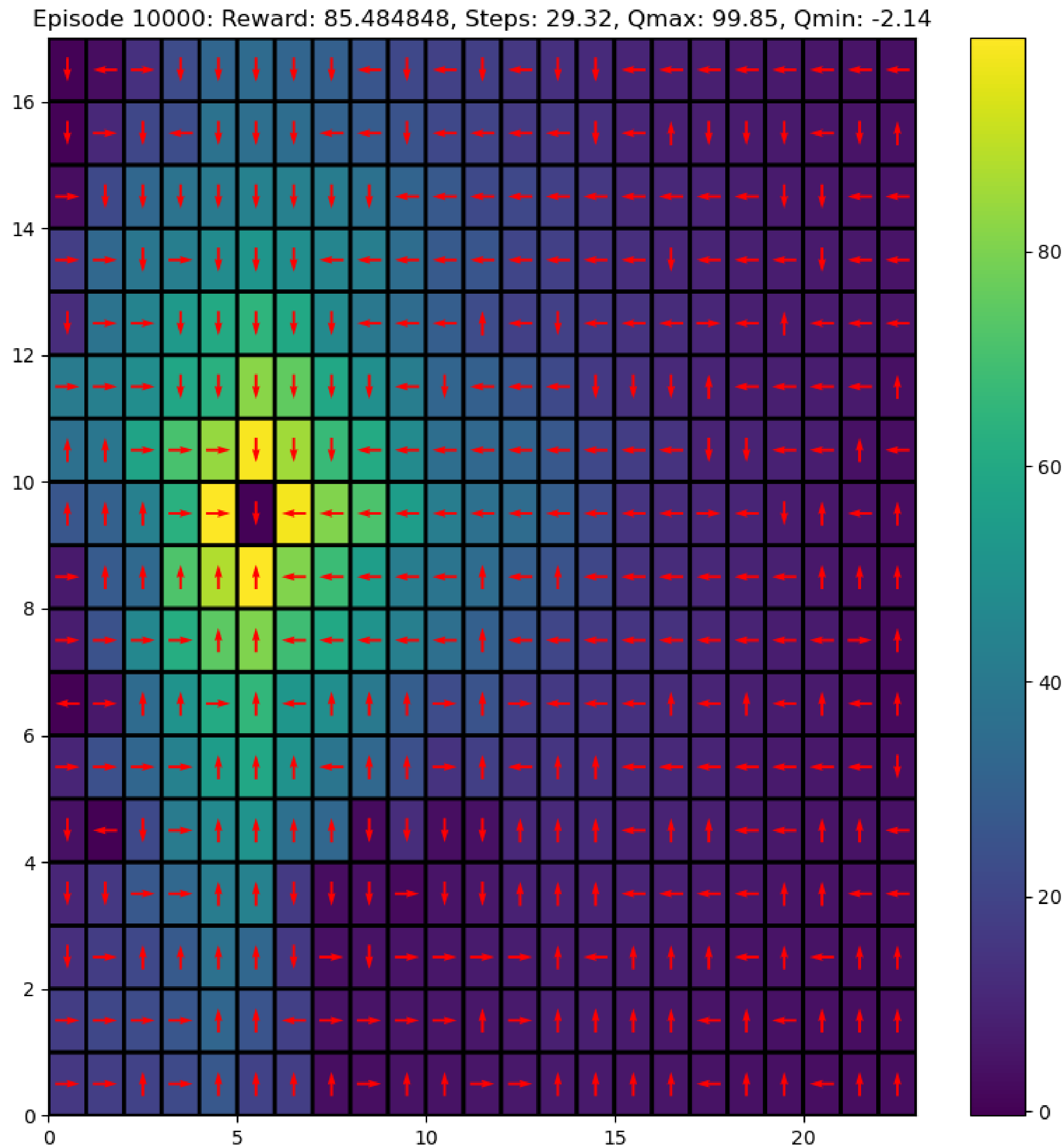
Now, implement the Q-Learning algorithm as an exercise.

Recall the update rule for Q-Learning: \begin{equation} Q(s_t,a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \end{equation}

Visualize and compare results with SARSA.

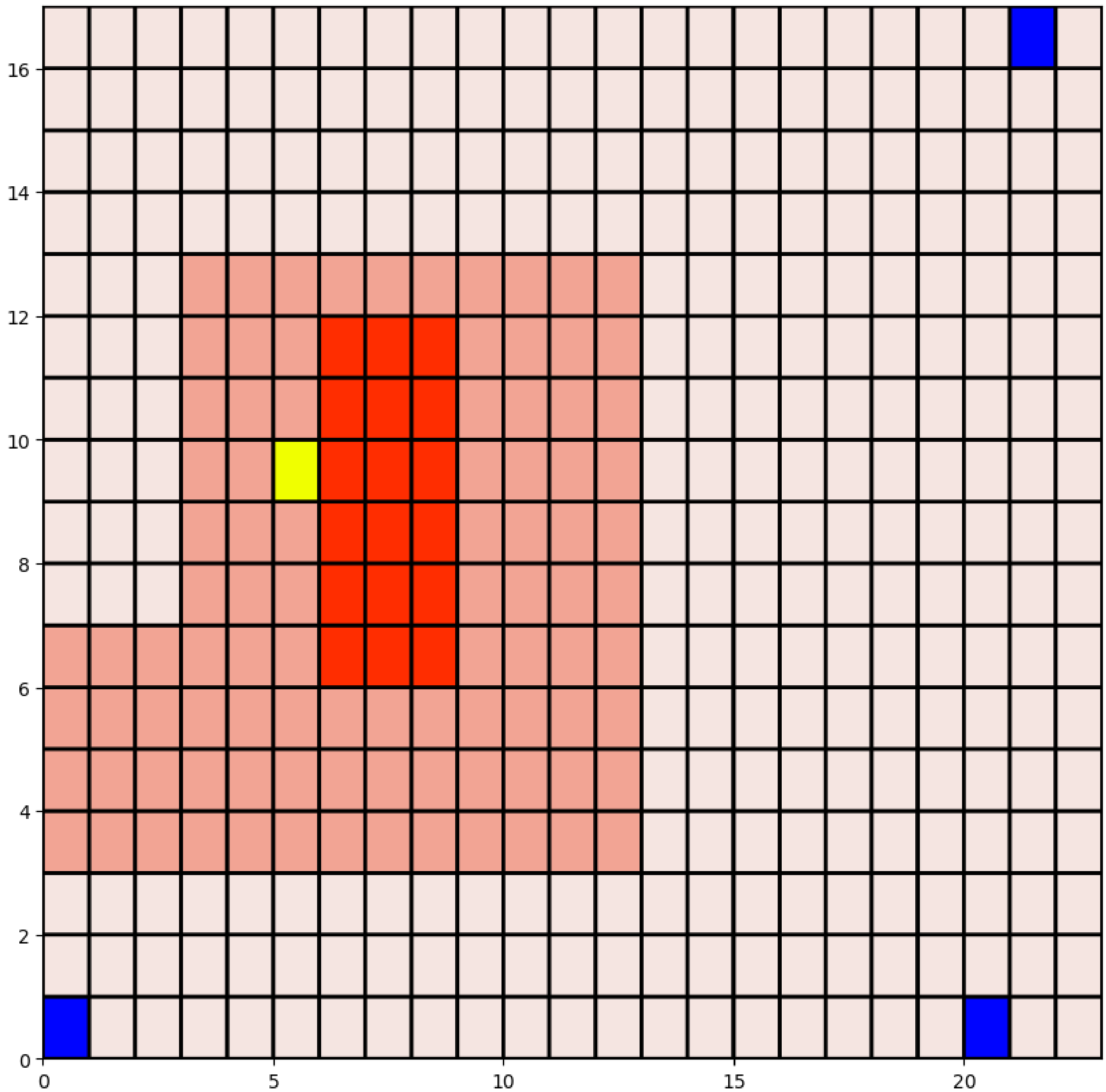
```
# initialize Q-value
Q = np.zeros((env.grid.shape[0], env.grid.shape[1],
len(env.action_space)))

alpha0 = 0.4
gamma = 0.9
episodes = 10000
epsilon0 = 0.1

print_freq = 100

def qlearning(env, Q, gamma = 0.9, plot_heat = False, choose_action =
choose_action_softmax):

    episode_rewards = np.zeros(episodes)
    steps_to_completion = np.zeros(episodes)
```

```
    if plot_heat:
        clear_output(wait=True)
        plot_Q(Q)
    epsilon = epsilon0
    alpha = alpha0
    for ep in tqdm(range(episodes)):
        tot_reward, steps = 0, 0

        # Reset environment
        state = env.reset()
        action = choose_action(Q, state)
        done = False
        while not done:
            state_next, reward, done = env.step(action)
            action_next = choose_action(Q, state_next)

            # TODO: update equation
            Q[state[0], state[1], action] += alpha * (reward + gamma *
np.max(Q[state_next[0], state_next[1]]) - Q[state[0], state[1],
action])

            tot_reward += reward
            steps += 1

            state, action = state_next, action_next

        episode_rewards[ep] = tot_reward
        steps_to_completion[ep] = steps

        if (ep+1)%print_freq == 0 and plot_heat:
            clear_output(wait=True)
            plot_Q(Q, message = "Episode %d: Reward: %f, Steps: %.2f,
Qmax: %.2f, Qmin: %.2f"%(ep+1, np.mean(episode_rewards[ep-
print_freq+1:ep]),

np.mean(steps_to_completion[ep-print_freq+1:ep]),

Q.max(), Q.min()))

    return Q, episode_rewards, steps_to_completion

Q, rewards, steps = qlearning(env, Q, gamma = gamma, plot_heat=True,
choose_action=lambda Q,state:
choose_action_epsilon(Q,state,epsilon0,rg))
```




```
100%|██████████| 10000/10000 [00:31<00:00, 319.61it/s]

from time import sleep

state = env.reset()
done = False
steps = 0
tot_reward = 0
while not done:
```

```
    clear_output(wait=True)
    state, reward, done = env.step(Q[state[0], state[1]].argmax())
    plt.figure(figsize=(10, 10))
    env.render(ax=plt, render_agent=True)
    plt.show()
    steps += 1
    tot_reward += reward
    sleep(0.2)
print("Steps: %d, Total Reward: %d"%(steps, tot_reward))
```

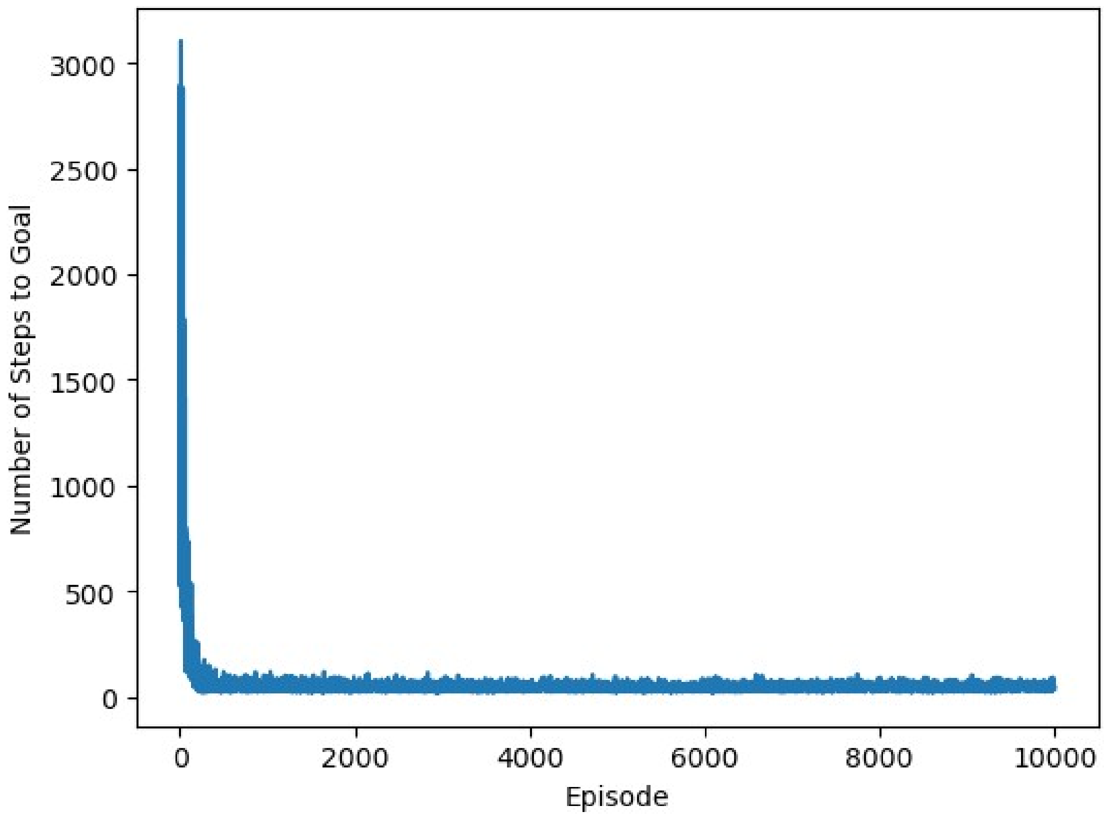



```
Steps: 38, Total Reward: 81
```

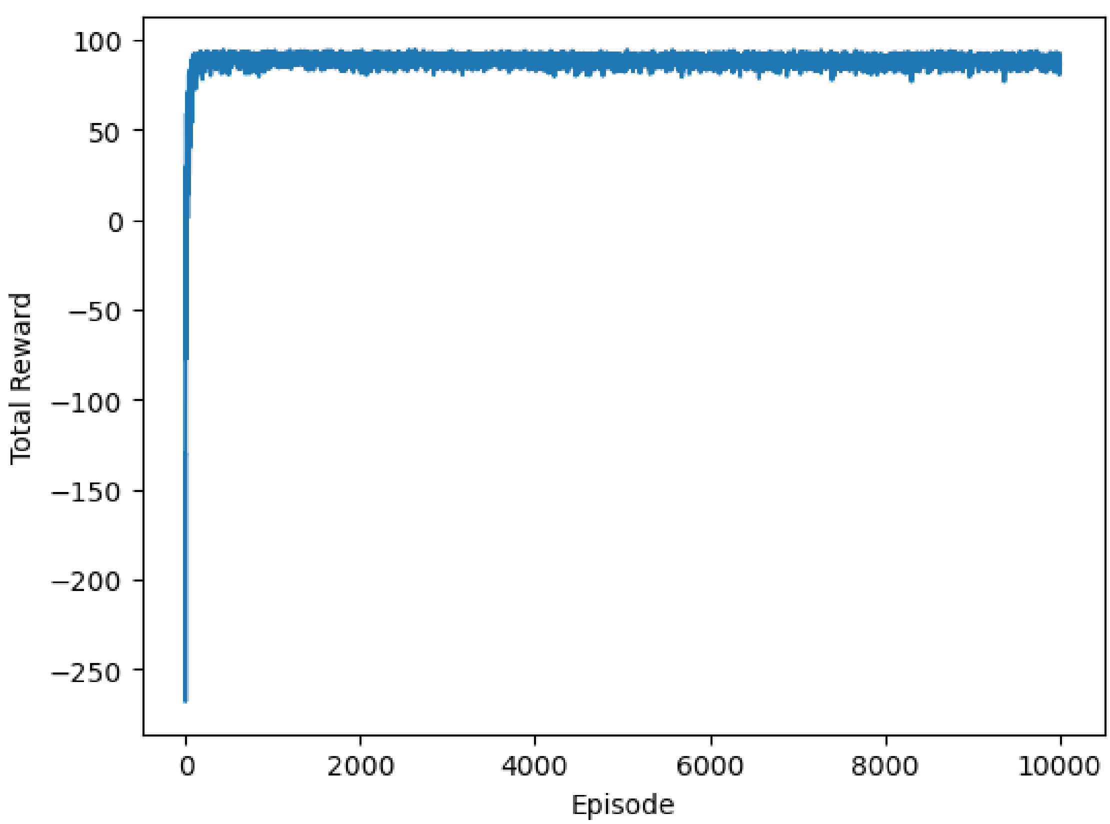
```
num_expts = 5
reward_avgs, steps_avgs = [], []

for i in range(num_expts):
    print("Experiment: %d"%(i+1))
    Q = np.zeros((env.grid.shape[0], env.grid.shape[1],
len(env.action_space)))
    rg = np.random.RandomState(i)

    # TODO: run qlearning, store metrics
    Q, episode_rewards, steps_to_completion = qlearning(env, Q,
gamma=0.9, plot_heat=False, choose_action=choose_action_softmax)

    reward_avgs.append(episode_rewards)
    steps_avgs.append(steps_to_completion)
```

```
Experiment: 1

100%|██████████| 10000/10000 [00:26<00:00, 378.49it/s]

Experiment: 2

100%|██████████| 10000/10000 [00:26<00:00, 373.85it/s]

Experiment: 3

100%|██████████| 10000/10000 [00:28<00:00, 347.48it/s]

Experiment: 4

100%|██████████| 10000/10000 [00:31<00:00, 317.00it/s]

Experiment: 5

100%|██████████| 10000/10000 [00:30<00:00, 325.51it/s]
```

```
# TODO: visualize individual metrics vs episode count (averaged across
multiple run(s))

plt.figure()
plt.plot(np.mean(steps_avgs, axis=0))
plt.xlabel('Episode')
plt.ylabel('Number of Steps to Goal')
plt.show()

# Visualization for Total Reward
plt.figure()
plt.plot(np.mean(reward_avgs, axis=0))
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.show()
```



Number of Steps to Goal
3000
2500
2000
1500
1000
500
0
0
2000
4000
6000
8000
10000
Episode

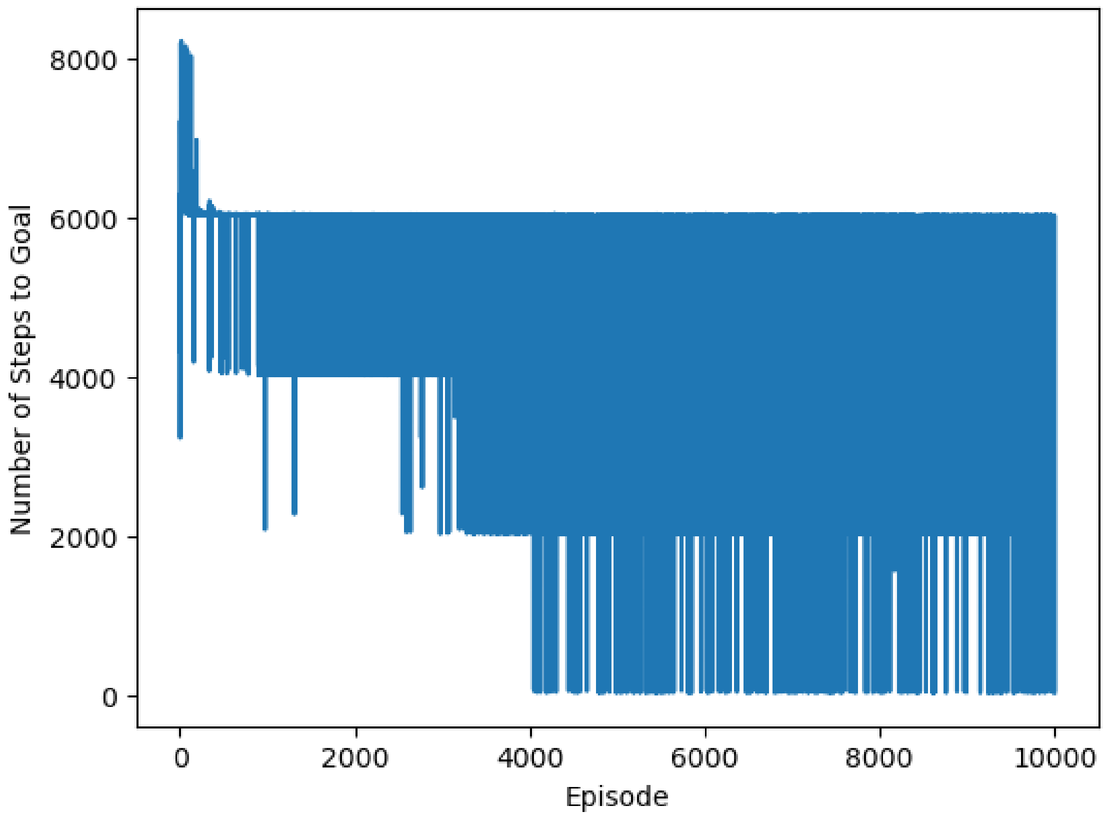


Total Reward
100
50
0
−50
−100
−150
−200
−250
0
2000
4000
6000
8000
10000
Episode

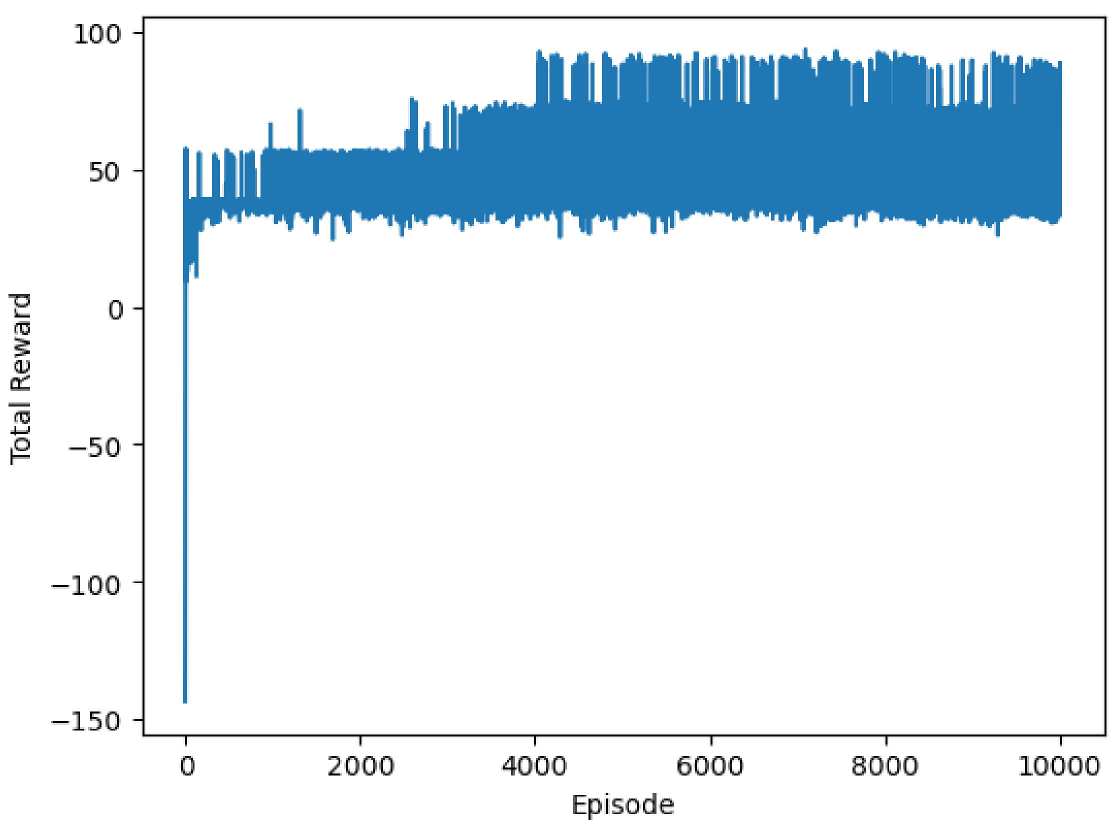
```
num_expts = 5
reward_avgs, steps_avgs = [], []

for i in range(num_expts):
    print("Experiment: %d"%(i+1))
    Q = np.zeros((env.grid.shape[0], env.grid.shape[1],
len(env.action_space)))
    rg = np.random.RandomState(i)

    # TODO: run qlearning, store metrics
    Q, episode_rewards, steps_to_completion = qlearning(env, Q,
gamma=0.9, plot_heat=False, choose_action=lambda Q,state:
choose_action_epsilon(Q,state,epsilon0,rg))

    reward_avgs.append(episode_rewards)
    steps_avgs.append(steps_to_completion)
```

```
Experiment: 1

100%|██████████| 10000/10000 [17:52<00:00,  9.32it/s]

Experiment: 2

100%|██████████| 10000/10000 [00:19<00:00, 525.46it/s]

Experiment: 3

100%|██████████| 10000/10000 [00:09<00:00, 1106.30it/s]

Experiment: 4

100%|██████████| 10000/10000 [17:29<00:00,  9.53it/s]

Experiment: 5

100%|██████████| 10000/10000 [15:09<00:00, 10.99it/s]
```

```
plt.figure()
plt.plot(np.mean(steps_avgs, axis=0))
plt.xlabel('Episode')
plt.ylabel('Number of Steps to Goal')
plt.show()

# Visualization for Total Reward
plt.figure()
plt.plot(np.mean(reward_avgs, axis=0))
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.show()
```



8000
6000
4000
2000
0
Number of Steps to Goal
0
2000
4000
6000
8000
10000
Episode



100
50
0
−50
−100
−150
Total Reward
0
2000
4000
6000
8000
10000
Episode

TODO: What differences do you observe between the policies learnt by Q Learning and SARSA (if any).

- When softmax is the selection policy there is no notable difference between Q learning and SARSA
- When Epislon Greedy is used, Q learning seems to offer higher rewards on average and it seems to find the optimal path faster than SARSA
- This could be due Epsilon Greedy algorithm being more explorative as compared to softmax, thereby traversing many states before arriving at the optimal one

```
!pip install nbconvert
!sudo apt-get install texlive-xetex texlive-fonts-recommended texlive-plain-generic

!jupyter nbconvert --to html "/content/drive/MyDrive/Colab Notebooks/CS6700_Tutorial_4_QLearning_SARSA_ROLLNUMBER.ipynb"
```