```
# Run this only if you are using Google Colab
from google.colab import drive
import os

drive.mount('/content/drive')

# change path here as per your directory structure
os.chdir('drive/My Drive/Tutorial_3')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True)
```

```
# Install relevant libraries
!pip install numpy matplotlib tqdm scipy
```

```
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
```

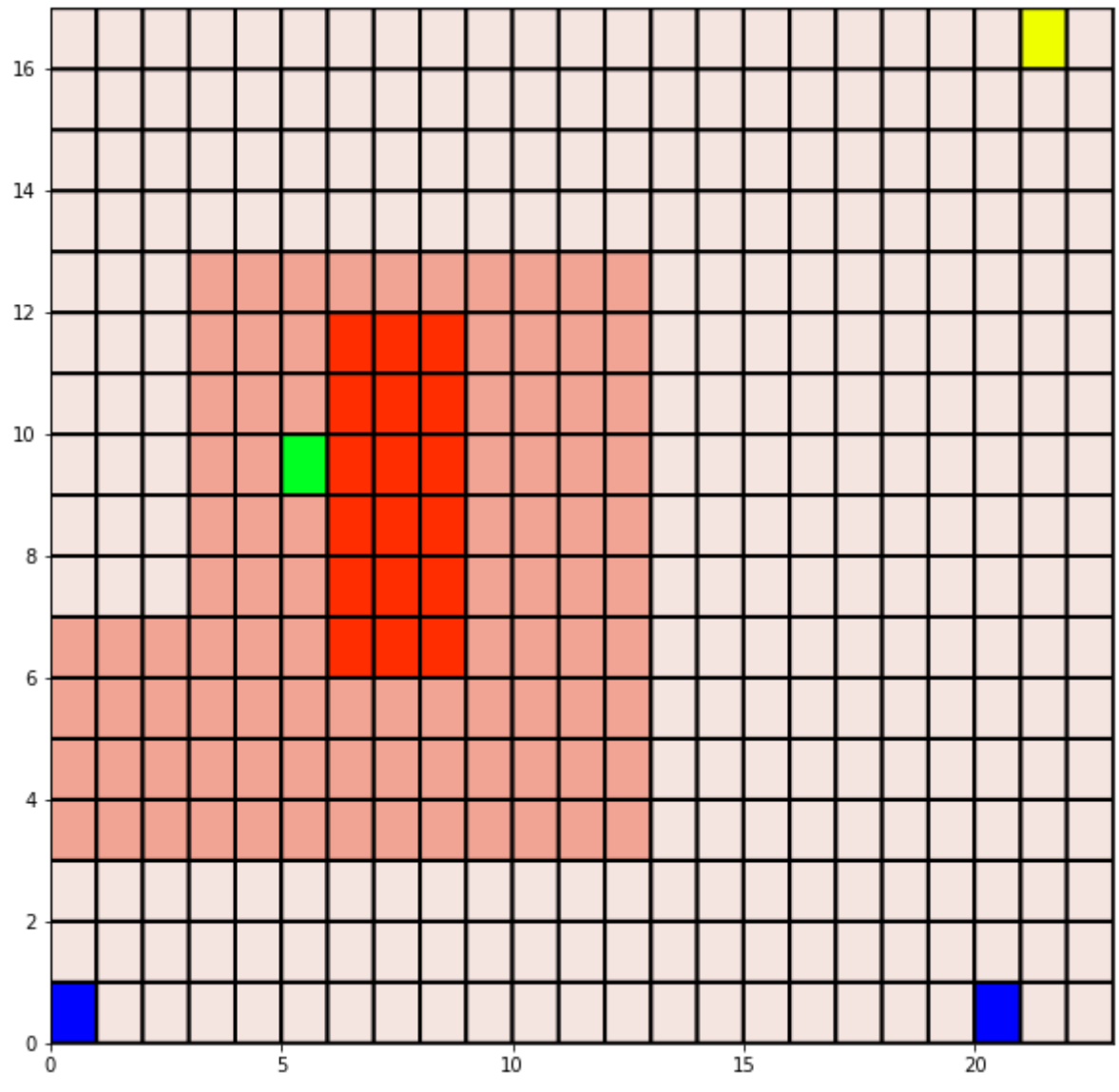
```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 2 2 2 1 1 1 1 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 2 2 2 1 1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

```
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)
```



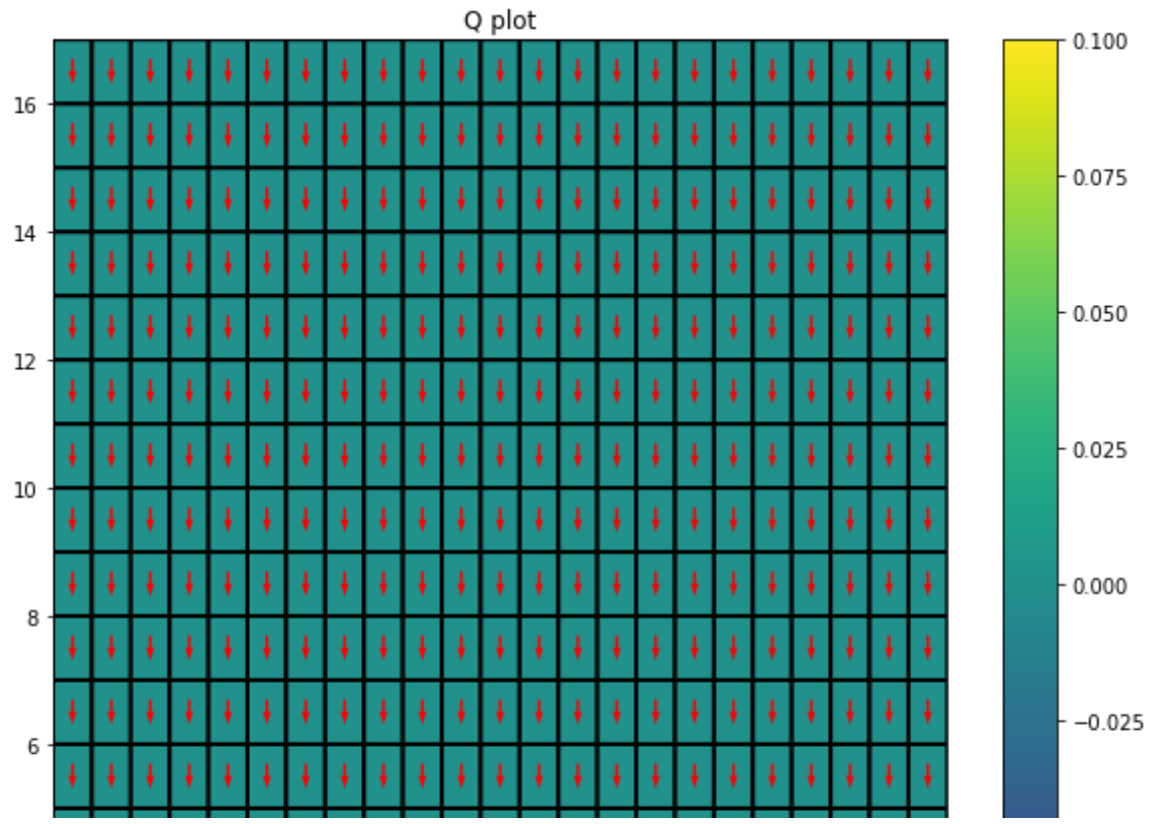
Legend

- *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.

```
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
```



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:
        return rg.choice(Q.shape[-1])
    else:
        return np.argmax(Q[state[0], state[1]])

# Softmax
def choose_action_softmax(Q, state, rg=rg):
    return rg.choice(Q.shape[-1], p = softmax(Q[state[0], state[1]]))
```

SARSA

Now we implement the SARSA algorithm.

Recall the update rule for SARSA:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Hyperparameters

So we have som hyperparameters for the algorithm:

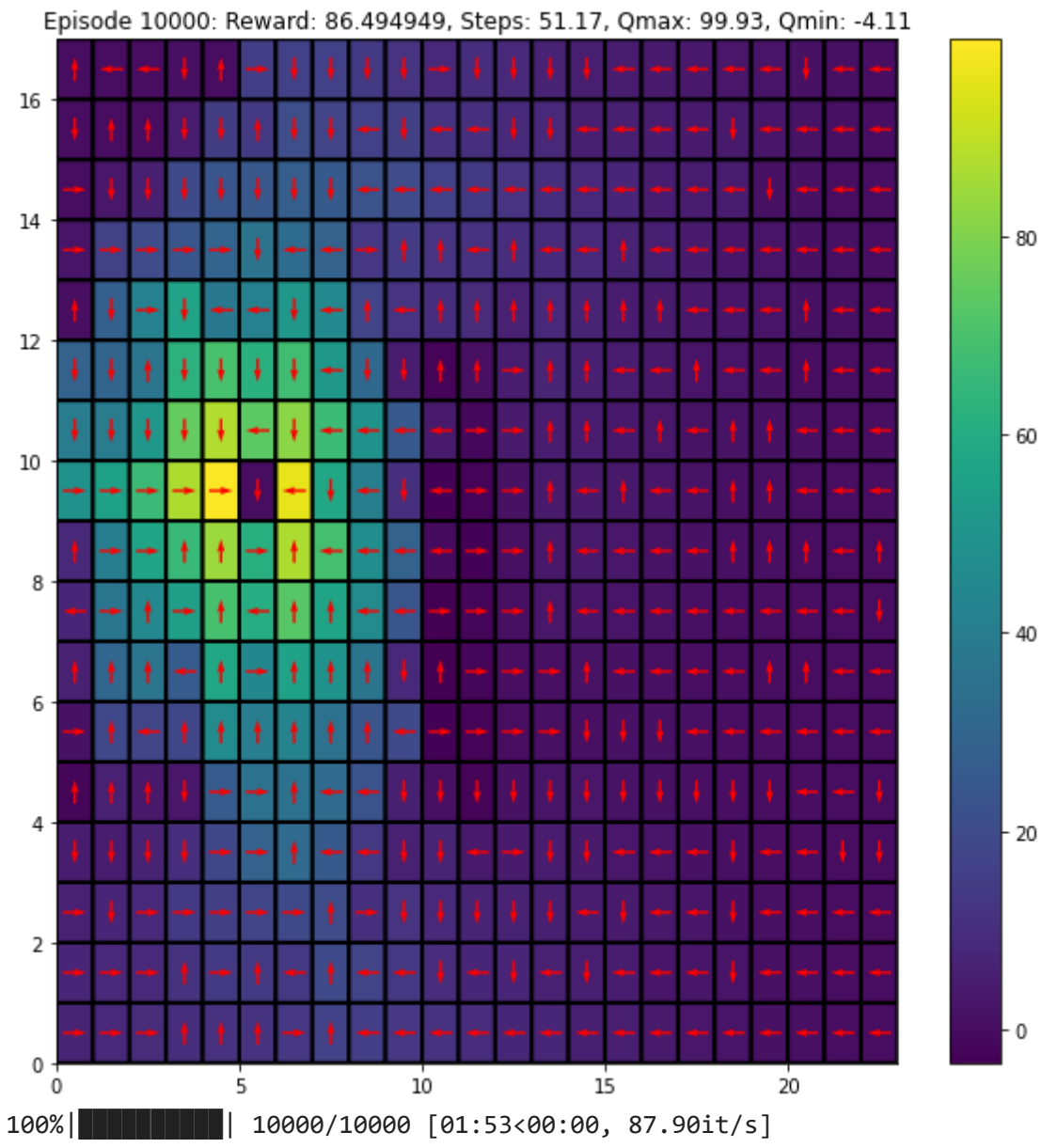
- α
- number of *episodes*.
- ϵ: For epsilon greedy exploration

```
# 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

```
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)

            # 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

            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
                                                                                                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= choose_action_softmax)
```

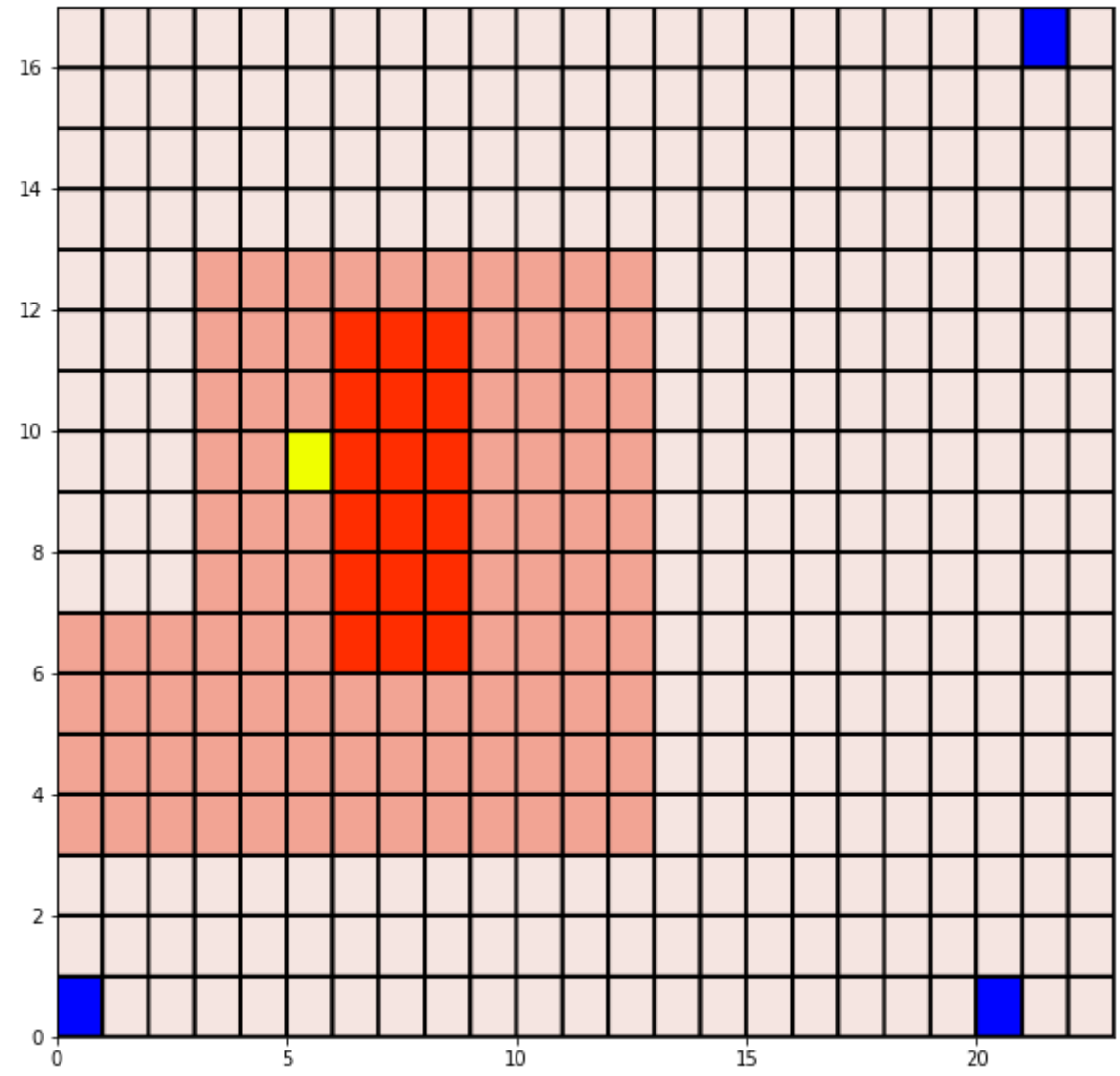



```
100%|██████████| 10000/10000 [01:53<00:00, 87.90it/s]
```

Visualizing the policy

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

```
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: 37, Total Reward: 92
```

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

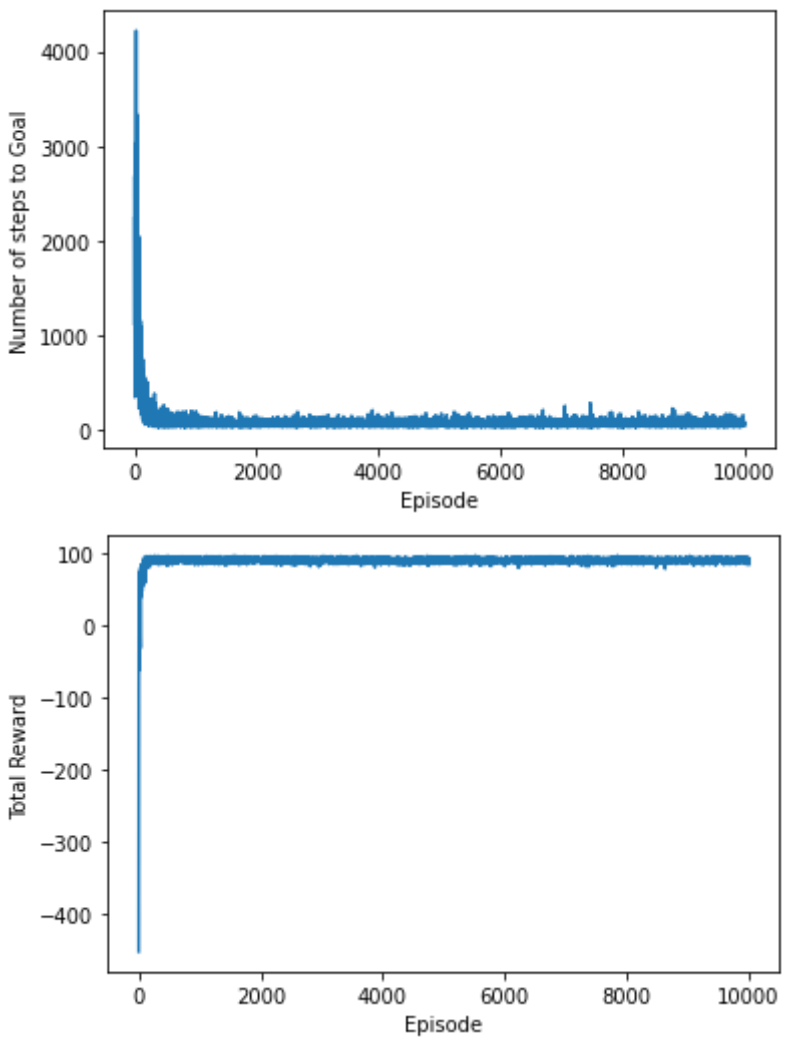
To ensure, we account for randomness in environment and algorithm (say when using epsilon-greedy exploration), we run the algorithm for multiple times and use the average of values over all runs.

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

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)
    Q, rewards, steps = sarsa(env, Q)
    Q_avgs.append(Q.copy())
    reward_avgs.append(rewards)
    steps_avgs.append(steps)
```

```
 Experiment: 1
 100%|██████████| 10000/10000 [01:08<00:00, 145.17it/s]
 Experiment: 2
 100%|██████████| 10000/10000 [01:24<00:00, 118.94it/s]
 Experiment: 3
 100%|██████████| 10000/10000 [01:19<00:00, 125.29it/s]
```

```
Experiment: 4
100%|██████████| 10000/10000 [00:59<00:00, 167.53it/s]
Experiment: 5
100%|██████████| 10000/10000 [02:21<00:00, 70.53it/s]
```

```
plt.xlabel('Episode')
plt.ylabel('Number of steps to Goal')
plt.plot(np.arange(episodes),np.average(steps_avgs, 0))
plt.show()
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.plot(np.arange(episodes),np.average(reward_avgs, 0))
plt.show()
```



Q-Learning

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

Recall the update rule for Q-Learning:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Visualize and compare results with SARSA.

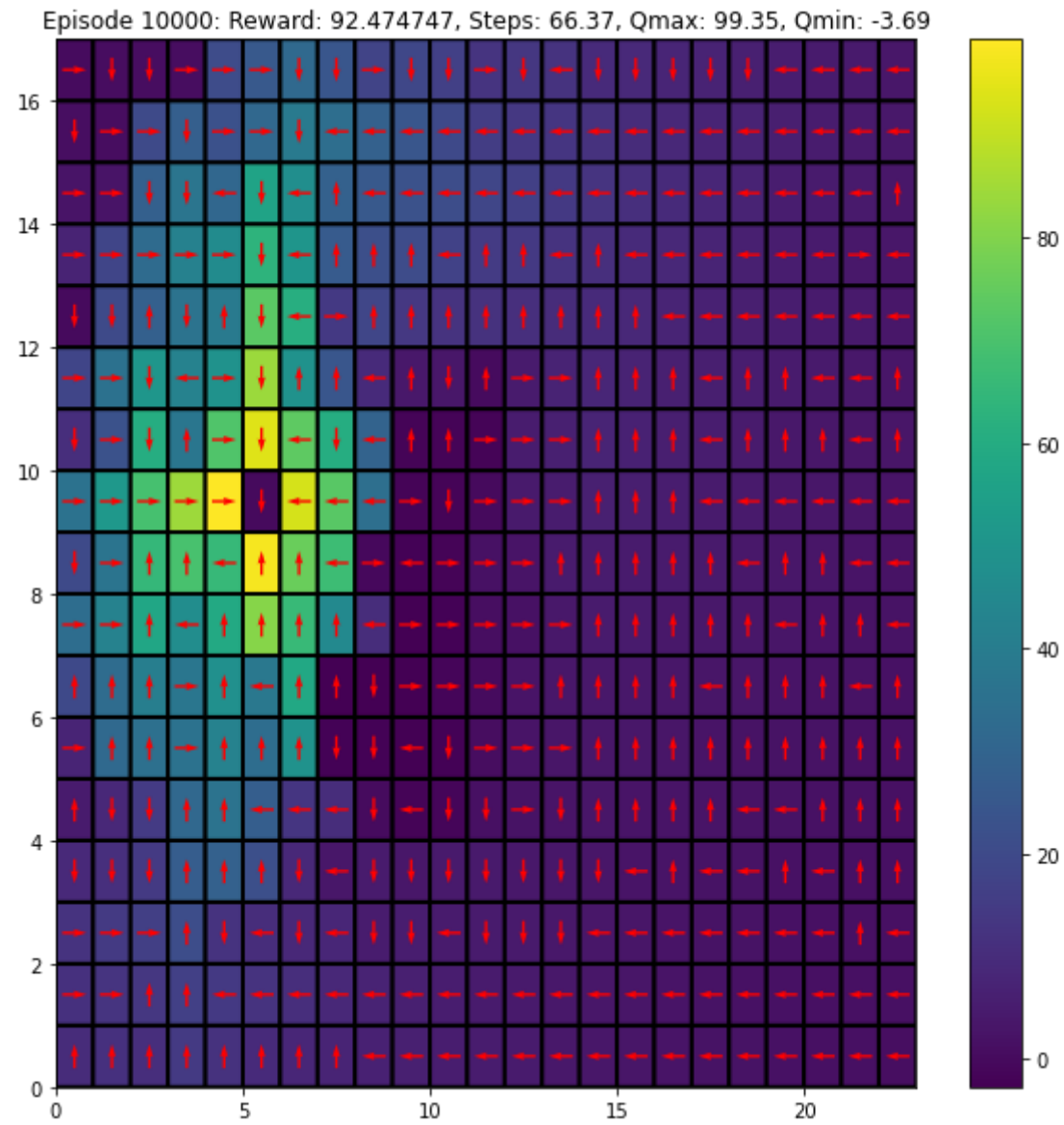
```
# initialize Q-value
Q_q = np.zeros((env.grid.shape[0], env.grid.shape[1], len(env.action_space)))

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

Let's implement Q learning

```
print_freq = 100

def Q_learning(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)

    # 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], ac

    tot_reward += reward
    steps += 1

    state = state_next
    action = choose_action(Q, state)

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
                                                                                        np.mean(steps_to_completion[ep-print_freq+1:ep]),
                                                                                        Q.max(), Q.min()))
return Q, episode_rewards, steps_to_completion
```

```
Q_q, rewards_q, steps_q = Q_learning(env, Q_q, gamma = gamma, plot_heat=True, choose_action= choose_action_softmax)
```



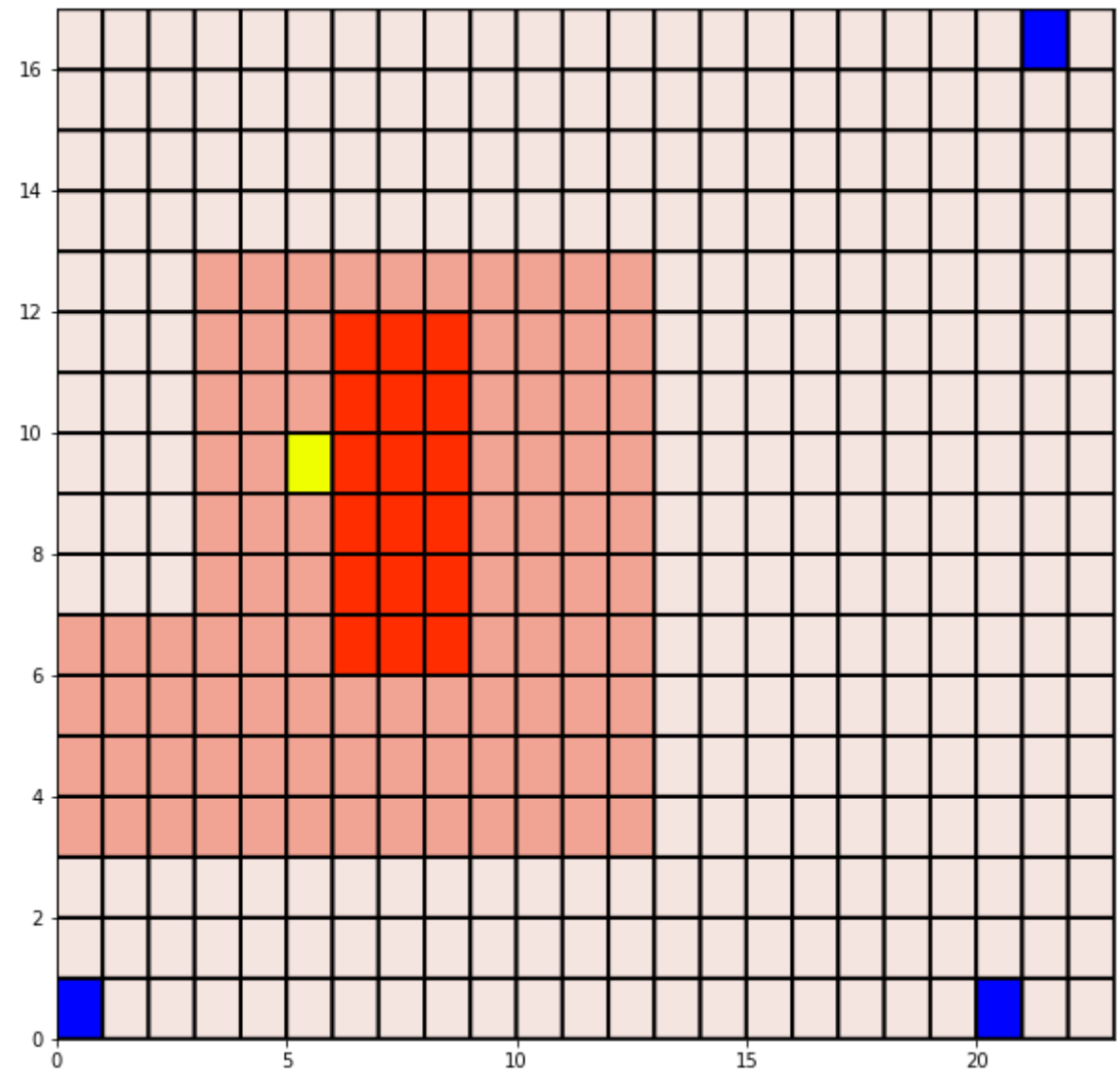

```
100%|██████████| 10000/10000 [02:01<00:00, 82.39it/s]
```

Visualizing the policy

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

```
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_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: 31, Total Reward: 97
```

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.

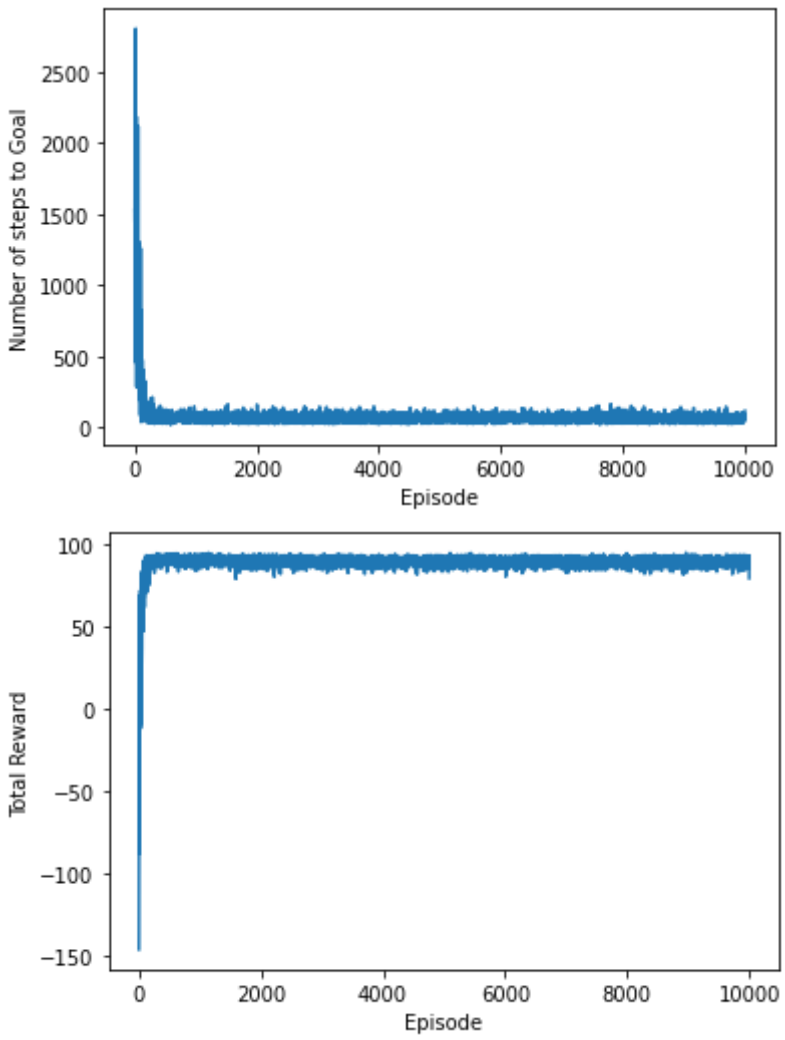
```
Q_avgs_q, reward_avgs_q, steps_avgs_q = [], [], []
num_expts = 5
```

```
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)
    Q, rewards, steps = Q_learning(env, Q)
    Q_avgs_q.append(Q.copy())
    reward_avgs_q.append(rewards)
    steps_avgs_q.append(steps)
```

```
 Experiment: 1
 100%|██████████| 10000/10000 [01:08<00:00, 145.49it/s]
 Experiment: 2
 100%|██████████| 10000/10000 [01:15<00:00, 132.81it/s]
 Experiment: 3
 100%|██████████| 10000/10000 [01:48<00:00, 91.78it/s]
 Experiment: 4
 100%|██████████| 10000/10000 [01:43<00:00, 96.78it/s]
 Experiment: 5
 100%|██████████| 10000/10000 [01:35<00:00, 105.00it/s]
```

```
plt.xlabel('Episode')
plt.ylabel('Number of steps to Goal')
plt.plot(np.arange(episodes),np.average(steps_avgs_q, 0))
plt.show()
plt.xlabel('Episode')
```

```
plt.ylabel('Total Reward')
plt.plot(np.arange(episodes),np.average(reward_avgs_q, 0))
plt.show()
```

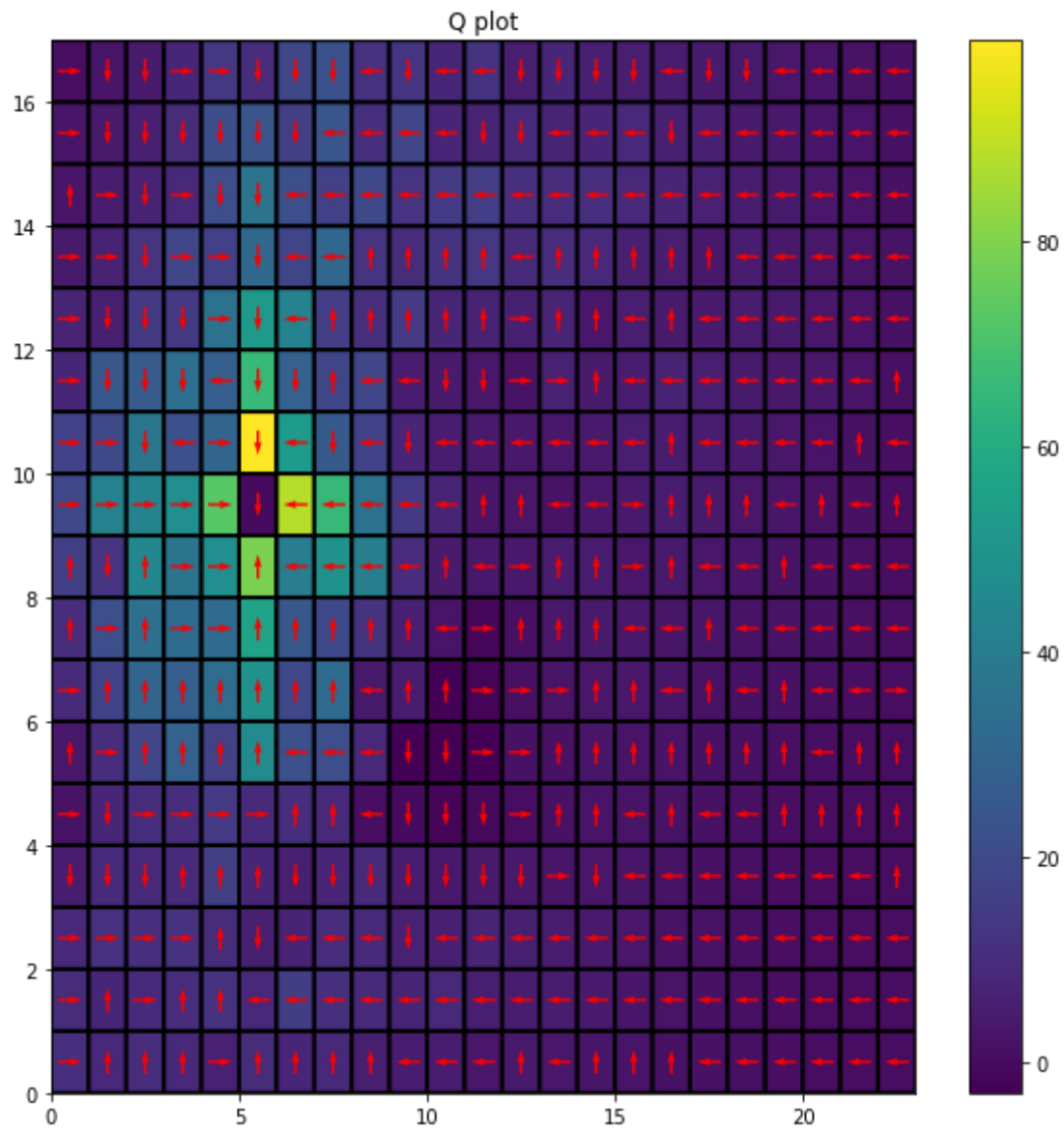


Comparing SARSA and Q learning

SARSA

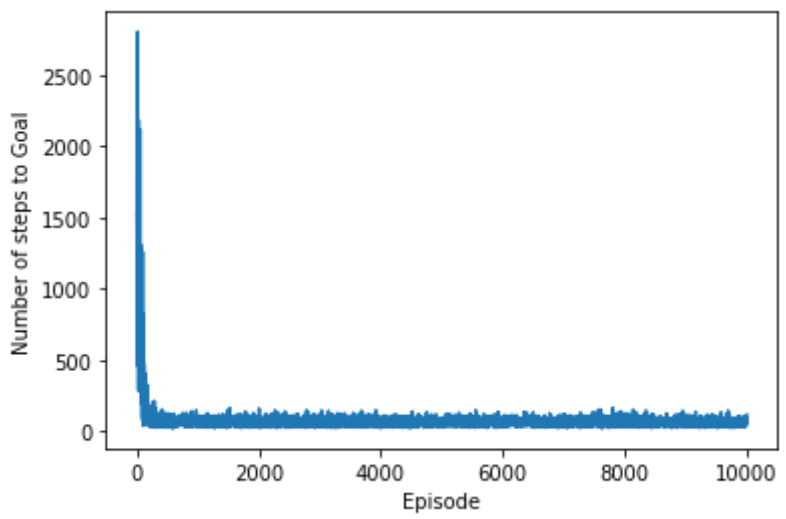
```
plt.xlabel('Episode')
plt.ylabel('Number of steps to Goal')
plt.plot(np.arange(episodes),np.average(steps_avgs, 0))
plt.show()
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.plot(np.arange(episodes),np.average(reward_avgs, 0))
plt.show()
```

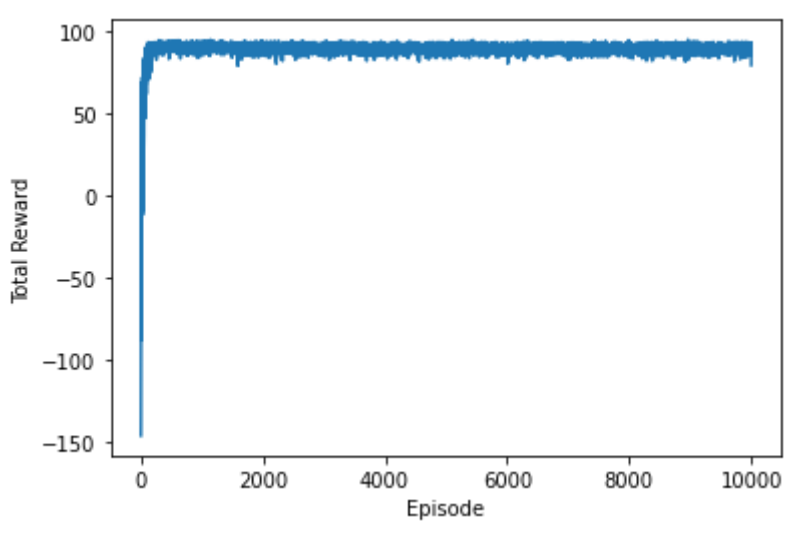
```
plot_Q(np.average(Q_avgs, 0))
```



Q-learning

```
plt.xlabel('Episode')
plt.ylabel('Number of steps to Goal')
plt.plot(np.arange(episodes),np.average(steps_avgs_q, 0))
plt.show()
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.plot(np.arange(episodes),np.average(reward_avgs_q, 0))
plt.show()
```






```
plot_Q(np.average(Q_avgs_q, 0))
```