In [1]:

```
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.

In [2]:

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

Let us construct a grid in a text file.

In [3]:

```
!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
```

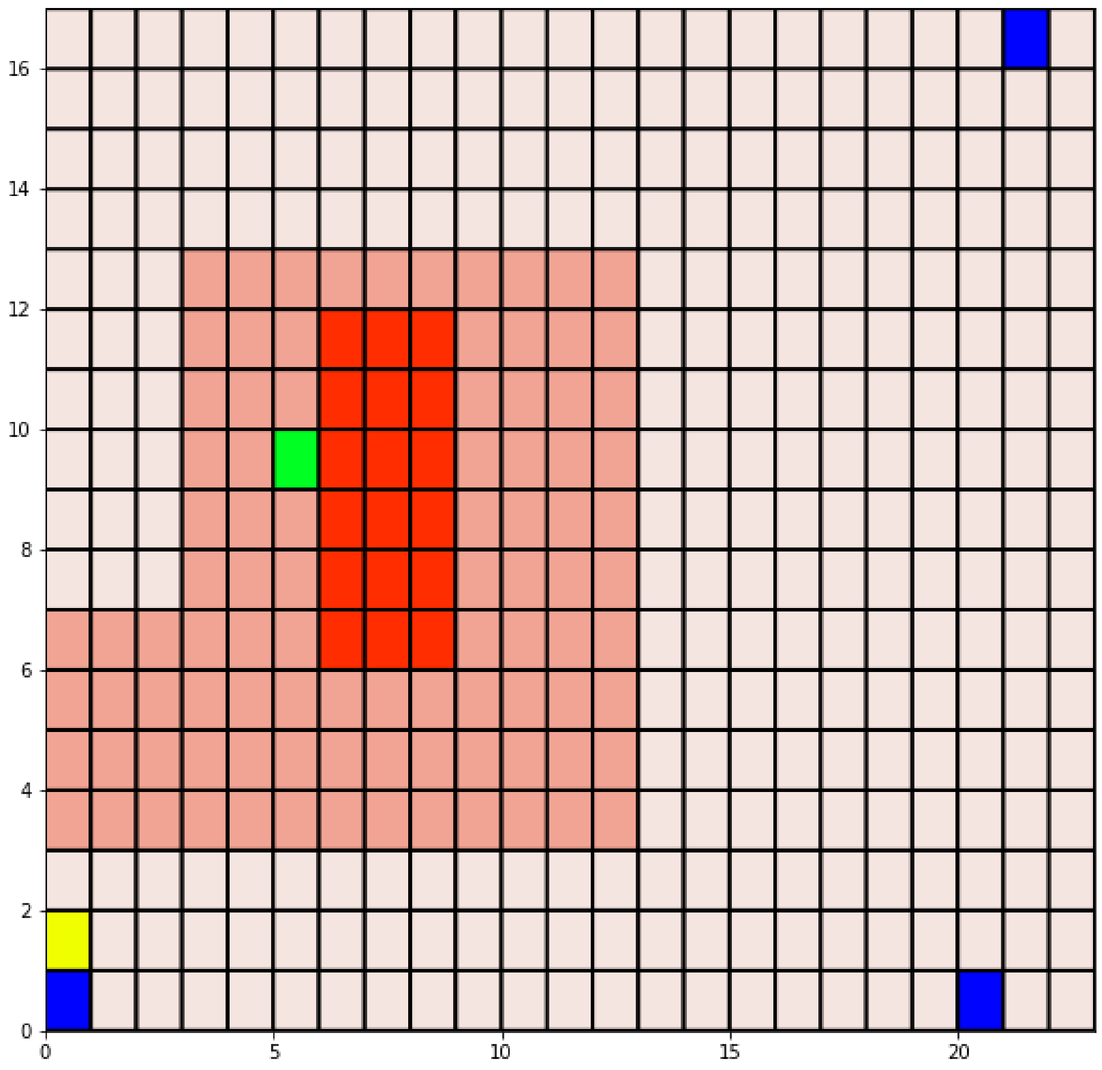
This is a 17×23 grid. The reward when an agent goes to a cell is negative of the value in that position in the text file (except if it is the goal cell). We will define the goal reward as 100. We will also fix the maximum episode length to 10000.

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.

In [4]:

```
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)
```


0
2
4
6
8
10
12
14
16
0
5
10
15
20

Legend

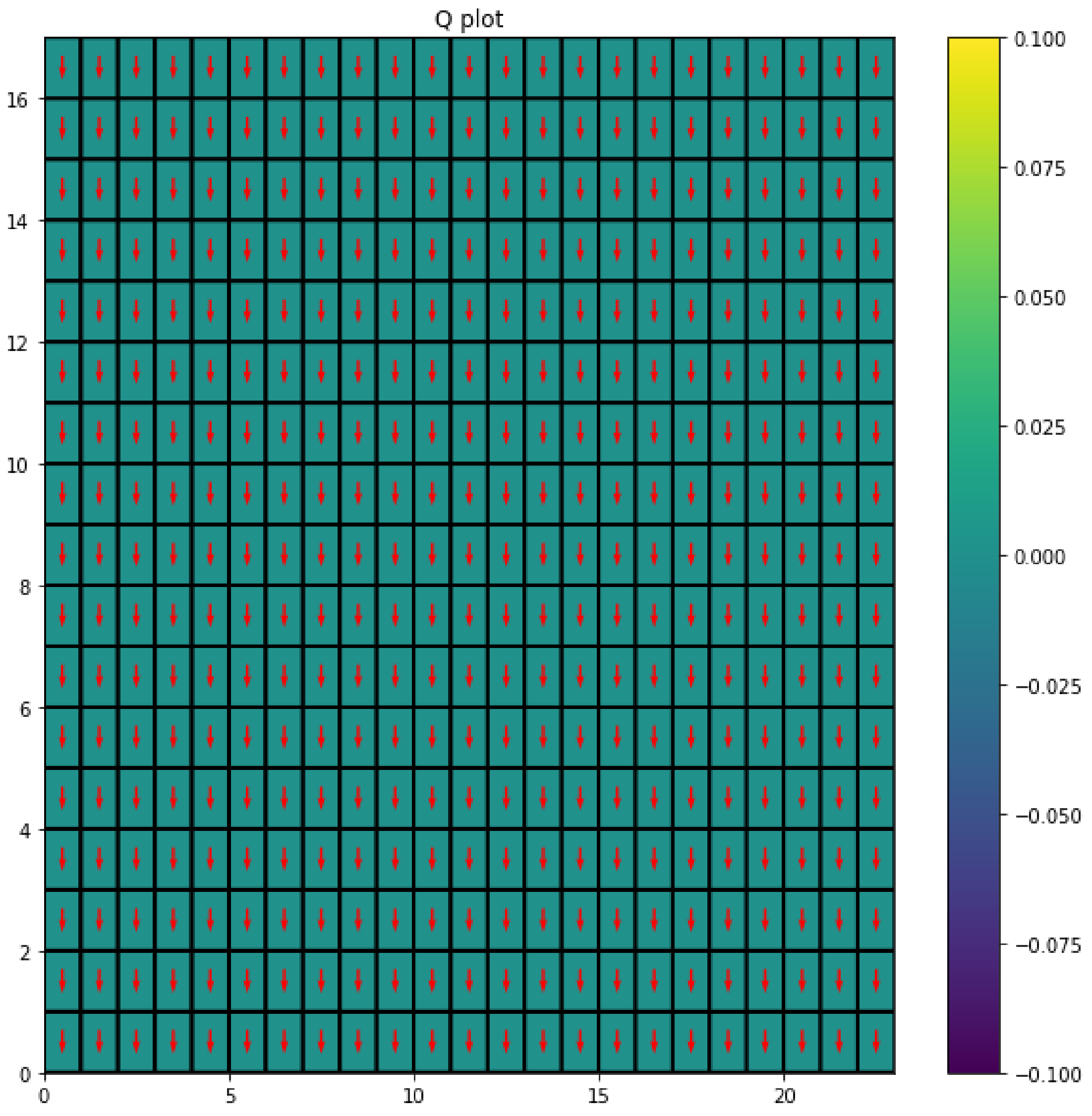
- *Blue* is the **start state**.
- *Green* is the **goal state**.
- *Yellow* is current **state of the agent**.
- *Redness* denotes the extent of **negative reward**.

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.

In [5]:

```
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
```


Q plot
0
2
4
6
8
10
12
14
16
0
5
10
15
20
0.100
0.075
0.050
0.025
0.000
−0.025
−0.050
−0.075
−0.100

```
Out[5]:

(17, 23, 4)
```

Exploration strategies

1. Epsilon-greedy
2. Softmax

```
In [6]:
```

```python
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(): # TODO: eps greedy condition
    if rg.uniform(0,1) < epsilon:
        return np.random.choice(actions) # TODO: return random action
    else:
        return actions[np.argmax(Q[state])]# TODO: return best action

# Softmax
def choose_action_softmax(Q, state, rg=rg):

    return np.random.choice(actions, p=softmax(Q[state]))# 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:

- $\alpha$
- number of *episodes*.
- $\epsilon$: For epsilon greedy exploration

In [7]:

```
# 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
```

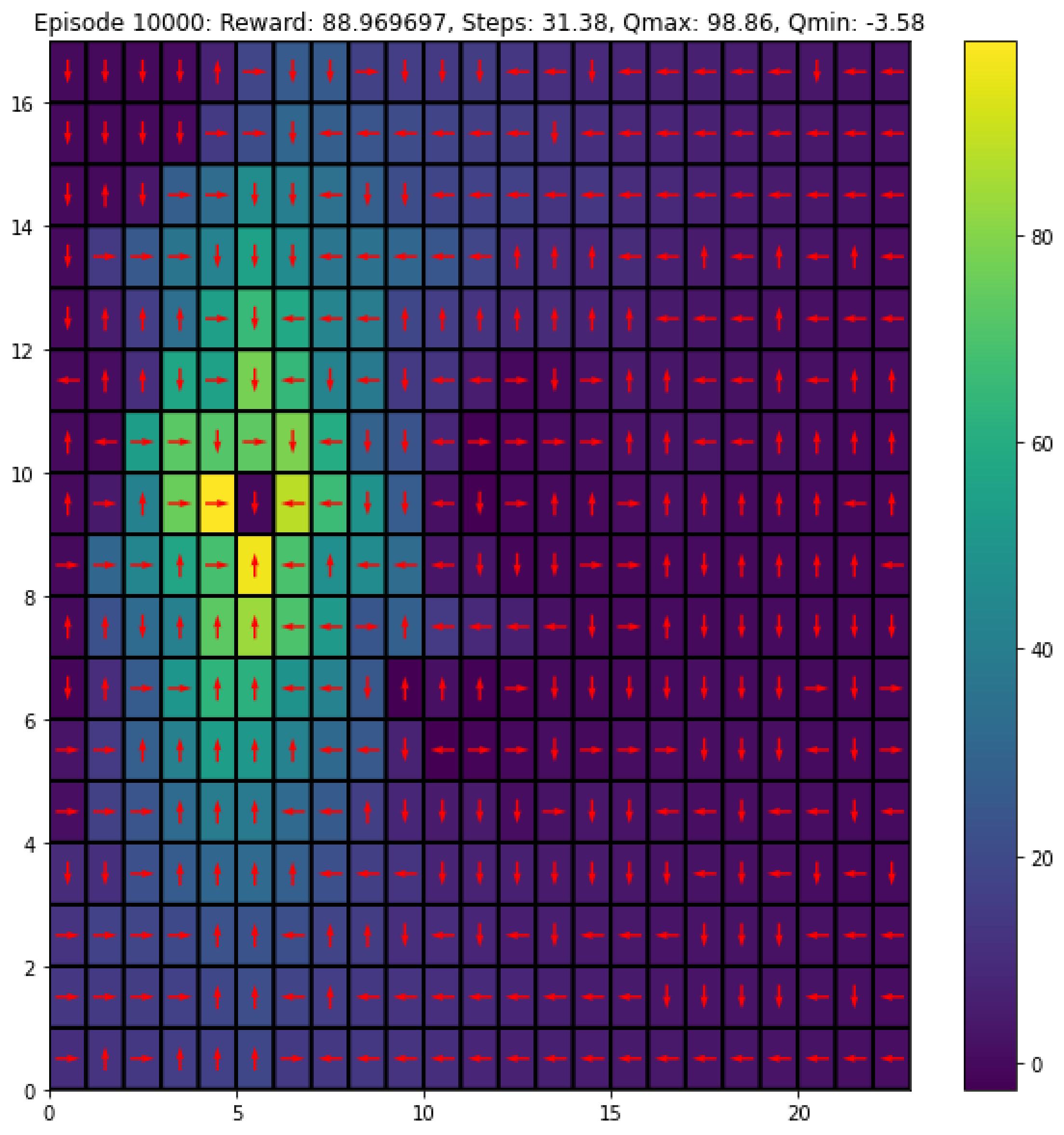
Let's implement SARSA

In [15]:

```
print_freq = 100

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

    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][action] += alpha*(reward + gamma*(Q[state_next][action_ne
xt])-Q[state][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.m
ax(), Q.min()))

    return Q, episode_rewards, steps_to_completion
```

```
In [10]:
```

```
Q, rewards, steps = sarsa(env, Q, gamma = gamma, plot_heat=True, choose_action
= choose_action_epsilon)
```



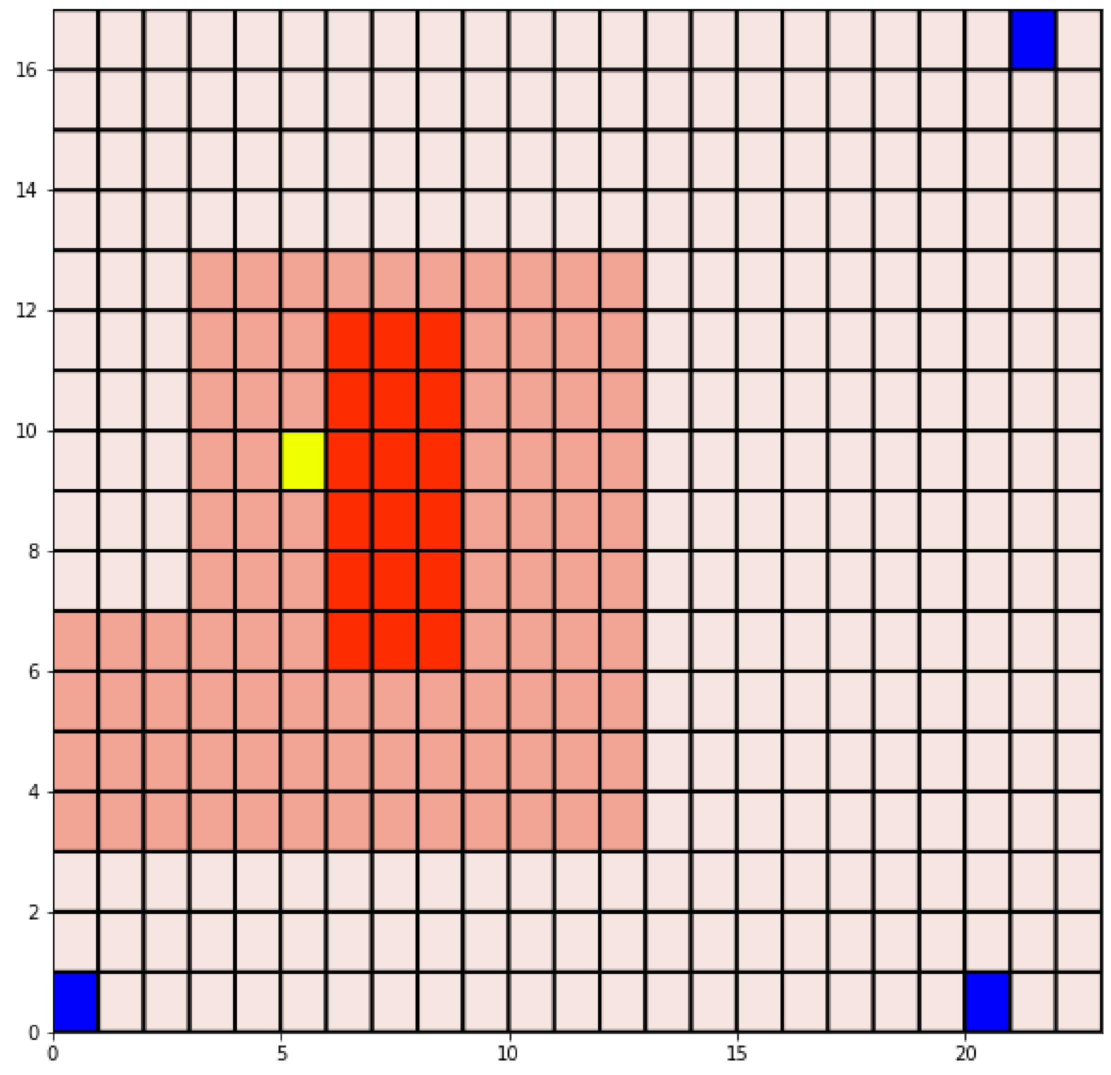
```
100%|██████████| 10000/10000 [00:46<00:00, 214.33it/s]
```

Visualizing the policy

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

In [11]:

```
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: 35, 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.

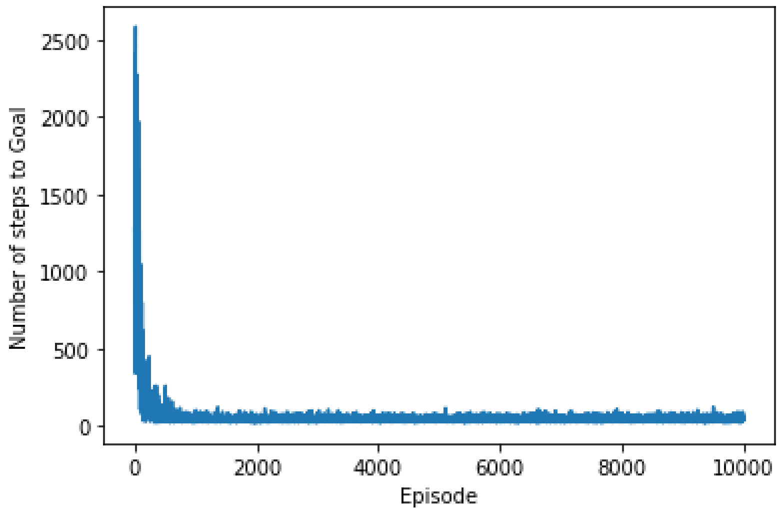
In [16]:

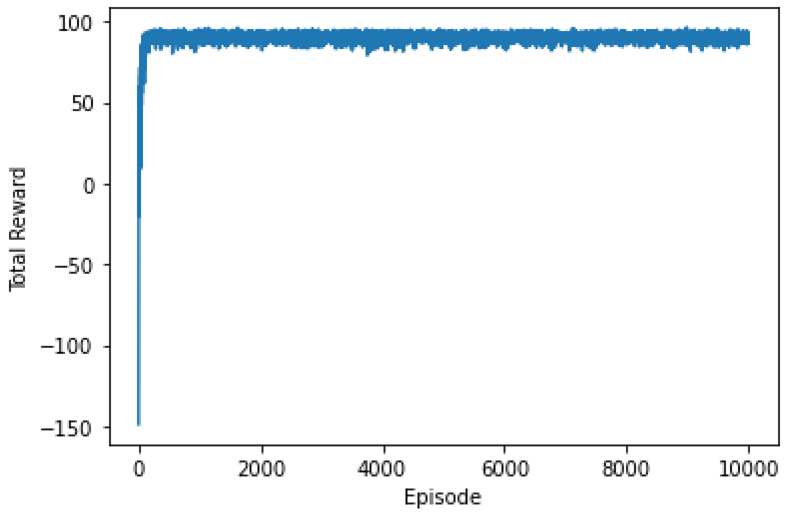
```
num_expts = 5
reward_avgs, steps_avgs = np.empty([5,10000]), np.empty([5,10000])

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 , rewards , steps =  sarsa(env, Q, gamma = gamma, plot_heat=False, choos
e_action= choose_action_softmax)
    reward_avgs[i] = rewards
    steps_avgs[i] = steps
reward_avgs = np.mean(reward_avgs,axis=0)
steps_avgs = np.mean(steps_avgs,axis=0)
```

```
Experiment: 1

100%|██████████| 10000/10000 [00:50<00:00, 198.17it/s]

Experiment: 2

100%|██████████| 10000/10000 [01:01<00:00, 163.64it/s]

Experiment: 3

100%|██████████| 10000/10000 [01:03<00:00, 158.53it/s]

Experiment: 4

100%|██████████| 10000/10000 [01:11<00:00, 139.79it/s]

Experiment: 5

100%|██████████| 10000/10000 [01:01<00:00, 163.91it/s]
```

In [ ]:

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

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

plt.figure()
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.plot(reward_avgs)
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.

In [ ]:

```
# 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
```

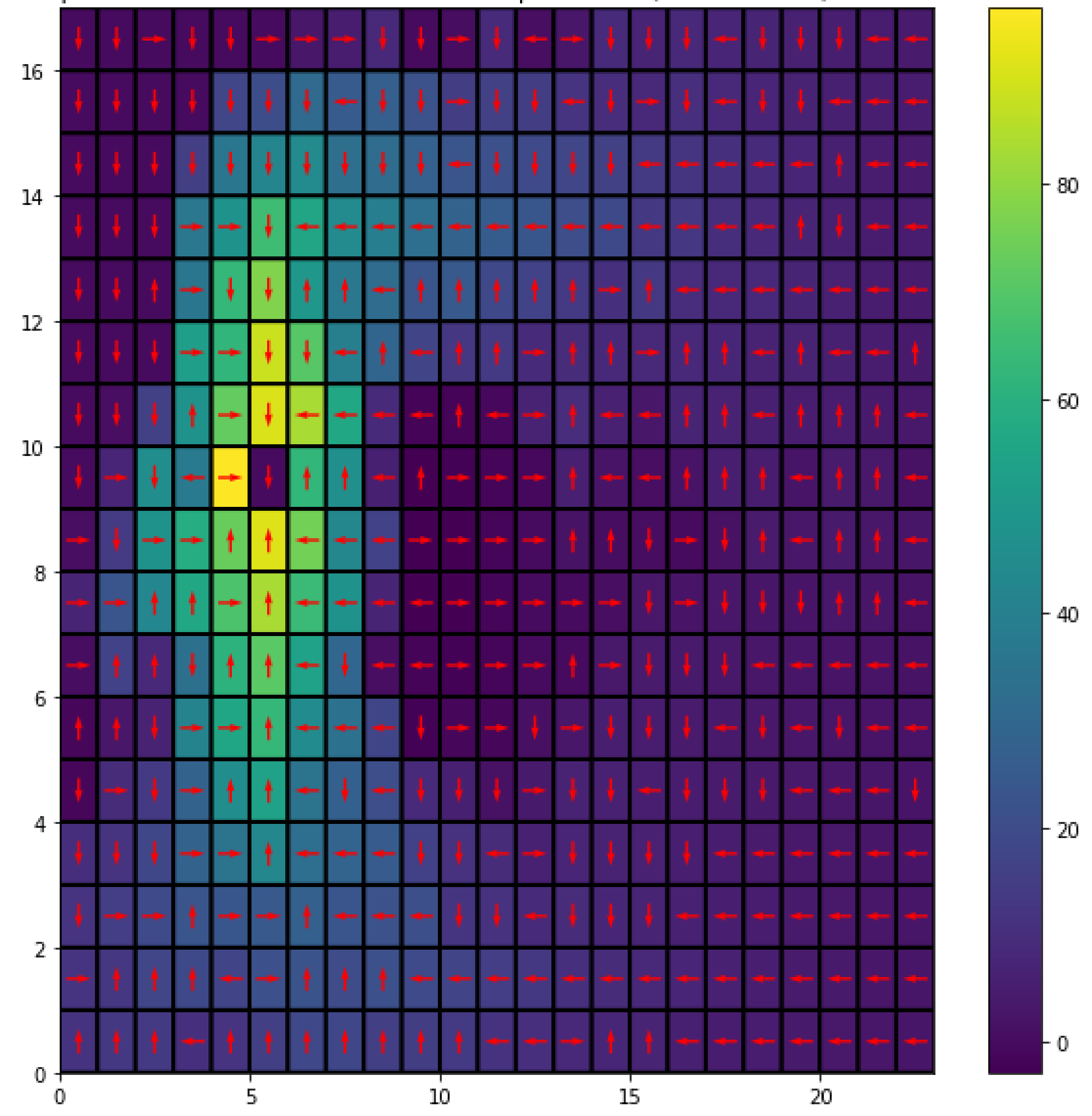
In [ ]:

```
print_freq = 100

def qlearning(env, Q, gamma = 0.9, plot_heat = False, choose_action = choose_a
ction_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][action] += alpha*(reward + gamma*(Q[state_next][np.argmax
(Q[state_next])])-Q[state][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.m
ax(), Q.min()))

    return Q, episode_rewards, steps_to_completion
```

In [ ]:

```
Q, rewards, steps = qlearning(env, Q, gamma = gamma, plot_heat=True, choose_ac
tion= choose_action_softmax)
```

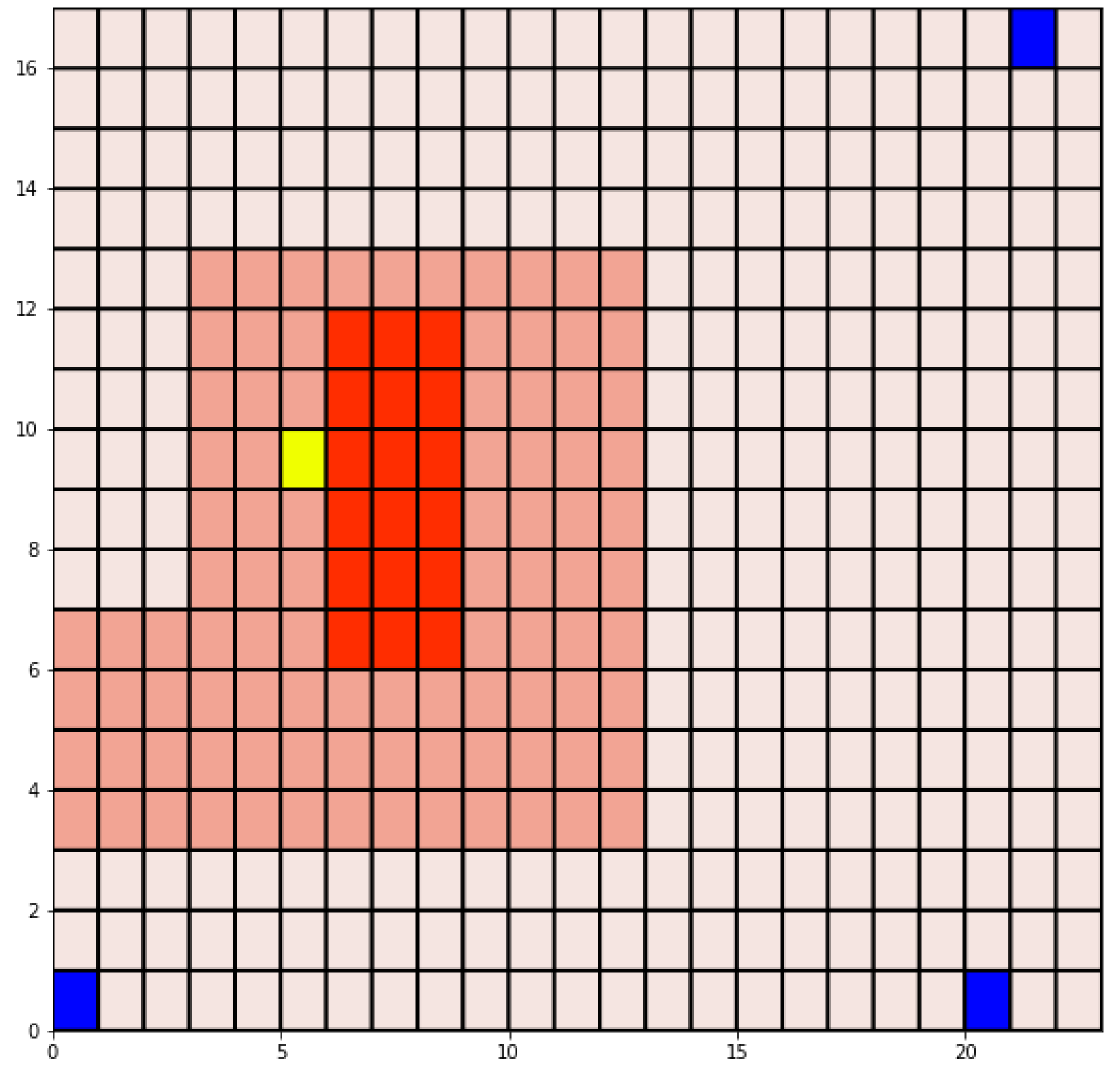
Episode 10000: Reward: 91.101010, Steps: 39.80, Qmax: 96.39, Qmin: -3.23



```
100%|██████████| 10000/10000 [01:37<00:00, 102.90it/s]
```

In [ ]:

```
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: 34, Total Reward: 91
```

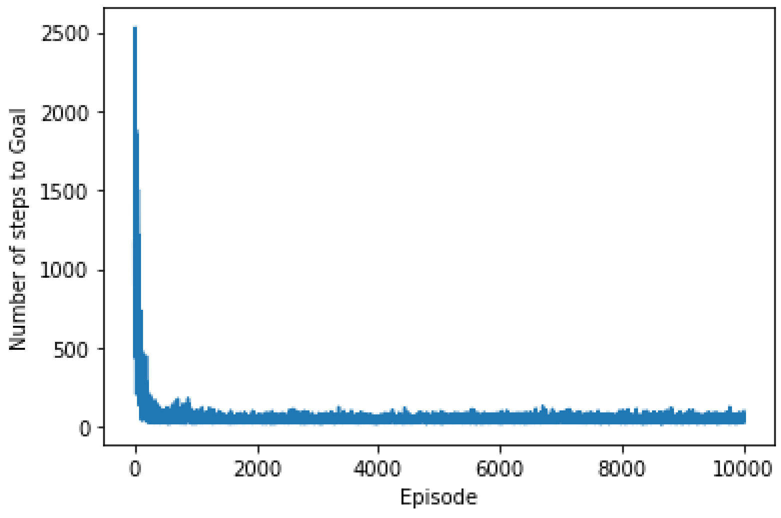
In [ ]:

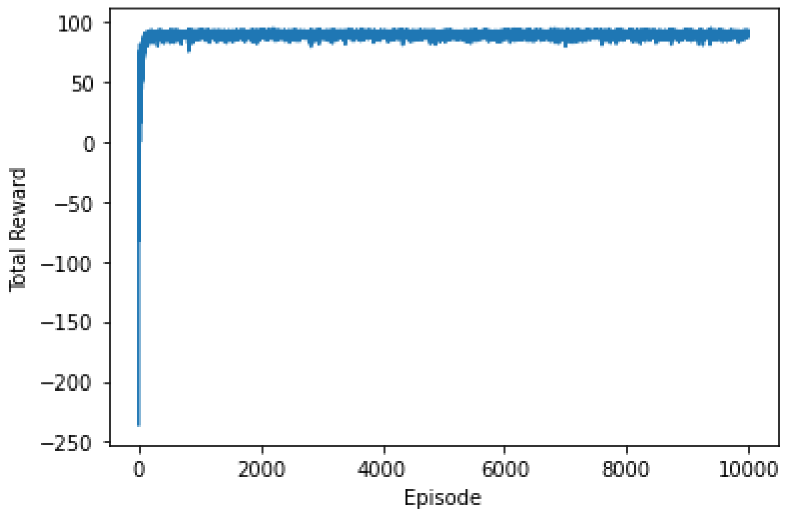
```python
num_expts = 5
reward_avgs, steps_avgs = np.empty([5,10000]), np.empty([5,10000])

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, rewards, steps = qlearning(env, Q, gamma = gamma, plot_heat=False, choo
se_action= choose_action_softmax)
    reward_avgs[i] = rewards
    steps_avgs[i] = steps
reward_avgs = np.mean(reward_avgs,axis=0)
steps_avgs = np.mean(steps_avgs,axis=0)
```

```
Experiment: 1

100%|██████████| 10000/10000 [01:01<00:00, 163.15it/s]

Experiment: 2

100%|██████████| 10000/10000 [00:53<00:00, 185.27it/s]

Experiment: 3

100%|██████████| 10000/10000 [01:02<00:00, 160.69it/s]

Experiment: 4

100%|██████████| 10000/10000 [01:18<00:00, 126.92it/s]

Experiment: 5

100%|██████████| 10000/10000 [01:26<00:00, 115.63it/s]
```

In [ ]:

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

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

plt.figure()
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.plot(reward_avgs)
plt.show()
```





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

1. SARSA is a more conservative policy as compared to Q-Learning which is a more greedy/agressive policy.
2. This is observed as often Q-Learning algorithm tends to take lesser number of steps as compared to SARSA , specifically when starting from bottom-left corner it is observed SARSA tends to explore the white cells trying to avoid the light red cells while Q-Learning spends lesser time in white cells and moves into the light red cells to reach the goal.
3. Although there is not much difference in the number of steps or rewards as difference in number of steps and total reward lies well with range of 4-5 units.

In [ ]:

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

```
Setting up texlive-base (2019.20200218-1) ...
mktexlsr: Updating /var/lib/texmf/ls-R-TEXLIVEDIST...
mktexlsr: Updating /var/lib/texmf/ls-R-TEXMFMAIN...
mktexlsr: Updating /var/lib/texmf/ls-R...
mktexlsr: Done.
tl-paper: setting paper size for dvips to a4: /var/lib/texmf/dvip
s/config/config-paper.ps
tl-paper: setting paper size for dvipdfmx to a4: /var/lib/texmf/dv
ipdfmx/dvipdfmx-paper.cfg
```

In [ ]:

```
!jupyter nbconvert --to html "/content/drive/MyDrive/Colab Notebooks/CS6700_Tu
torial_4_QLearning_SARSA_EE20B101.ipynb"
```

In [17]:

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```