In [2]:

```
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 [3]:

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

Let us construct a grid in a text file.

In [4]:

```
!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
```

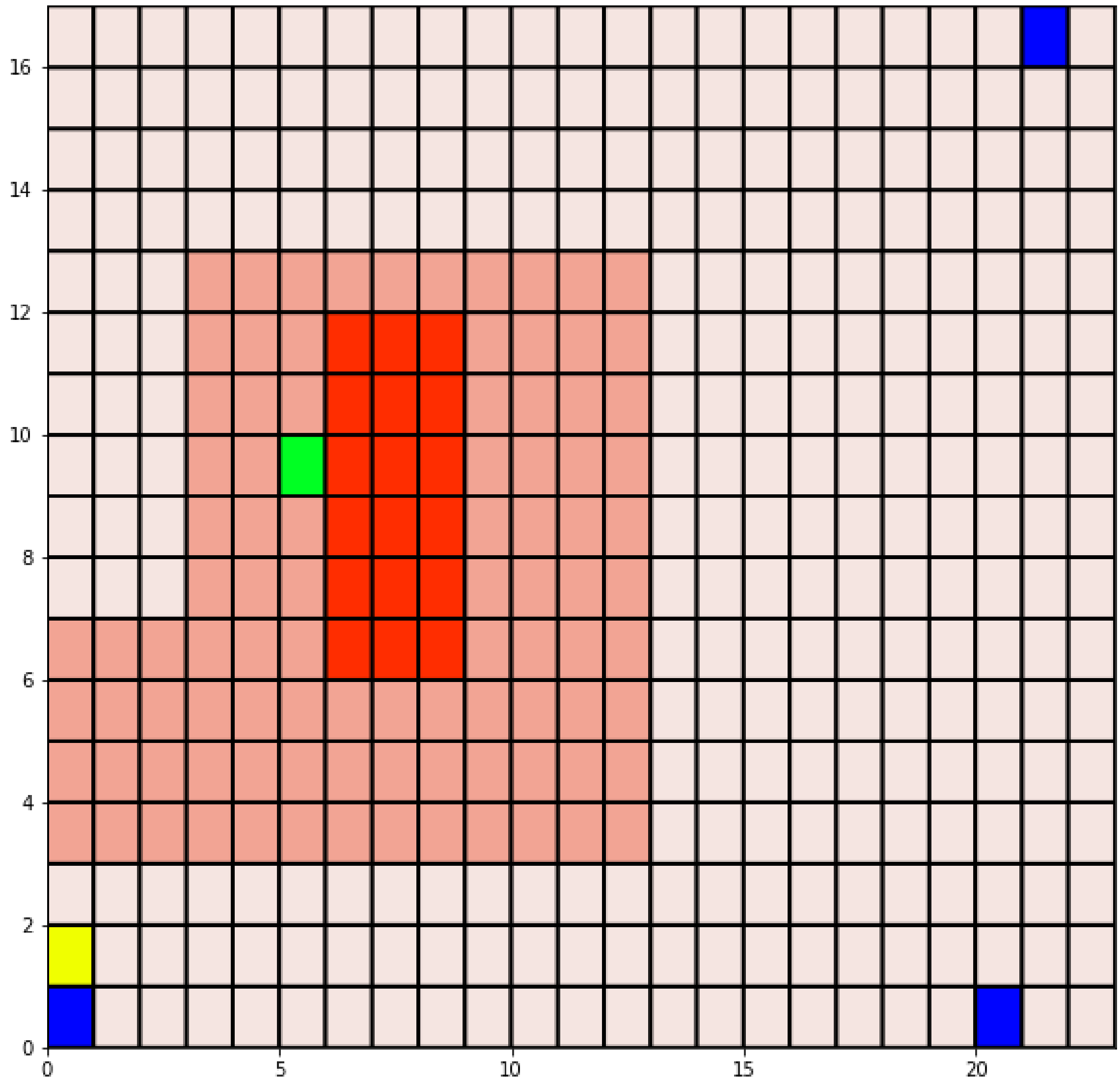
This is a 17×23 grid. The reward when an agent goes to a cell is negative of the value in that position in the text file (except if it is the goal cell). We will define the goal reward as 100. We will also fix the maximum episode length to 10000.

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 [5]:

```
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_stat
es=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

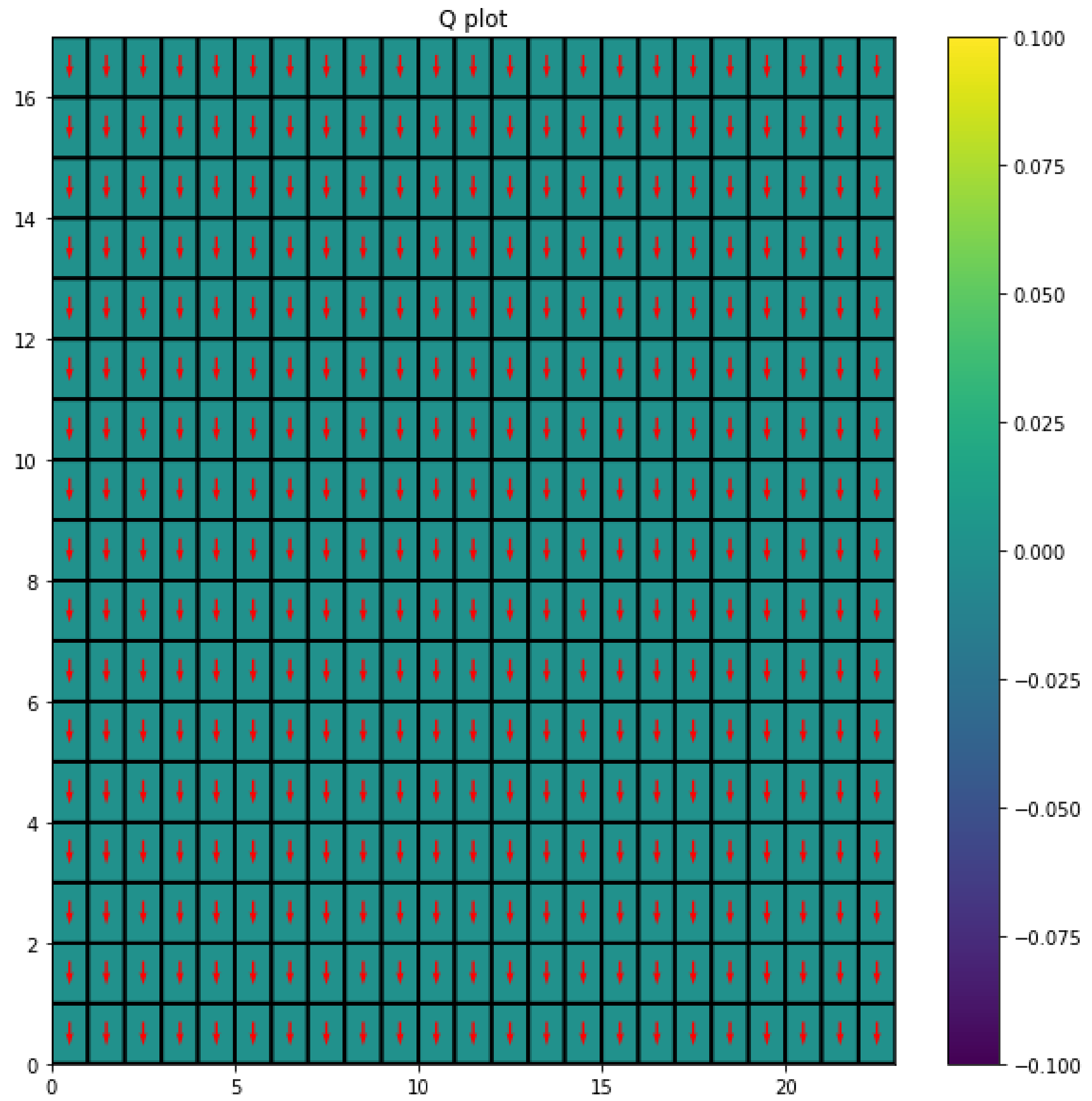
- *Blue* is the **start state**.
- *Green* is the **goal state**.
- *Yellow* is current **state of the agent**.
- *Redness* denotes the extent of **negative reward**.

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.

In [6]:

```
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
```



Out[6]:

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

Exploration strategies

1. Epsilon-greedy
2. Softmax

In [7]:

```
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
        return rg.randint(4) # TODO: return random action
    else:
      if rg.rand() < epsilon:
          return rg.randint(4) # select a random action with probability epsilon
      else:
          return np.argmax(Q[state[0], state[1]]) # select the action with the highest
Q value
      # TODO: return best action

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

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

In [8]:

```
# 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
```

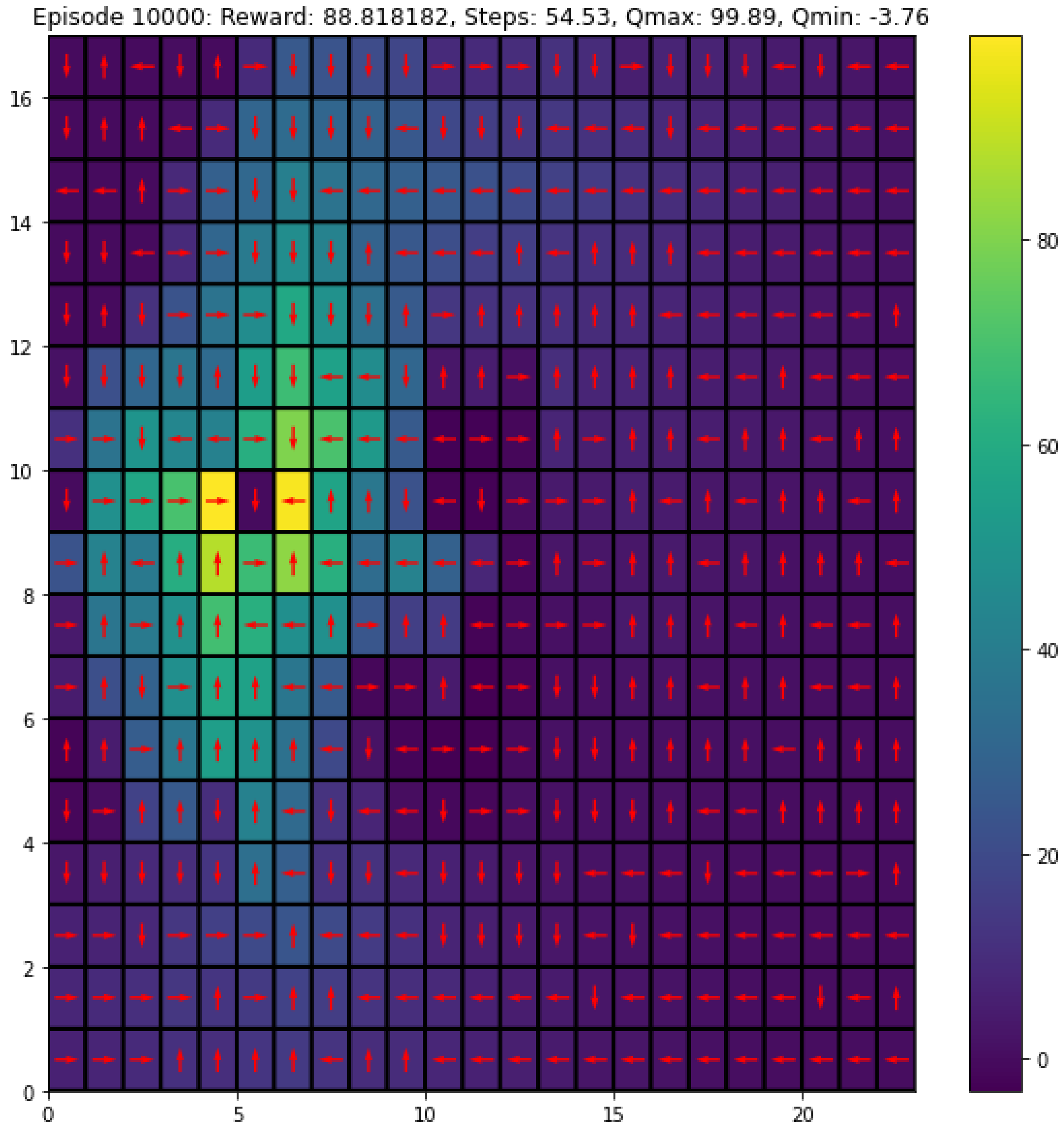
Let's implement SARSA

In [9]:

```
print_freq = 100

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

    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(step
s_to_completion[ep-print_freq+1:ep]),
                                                                           Q.max(), Q.m
in()))

    return Q, episode_rewards, steps_to_completion
```

In [10]:

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



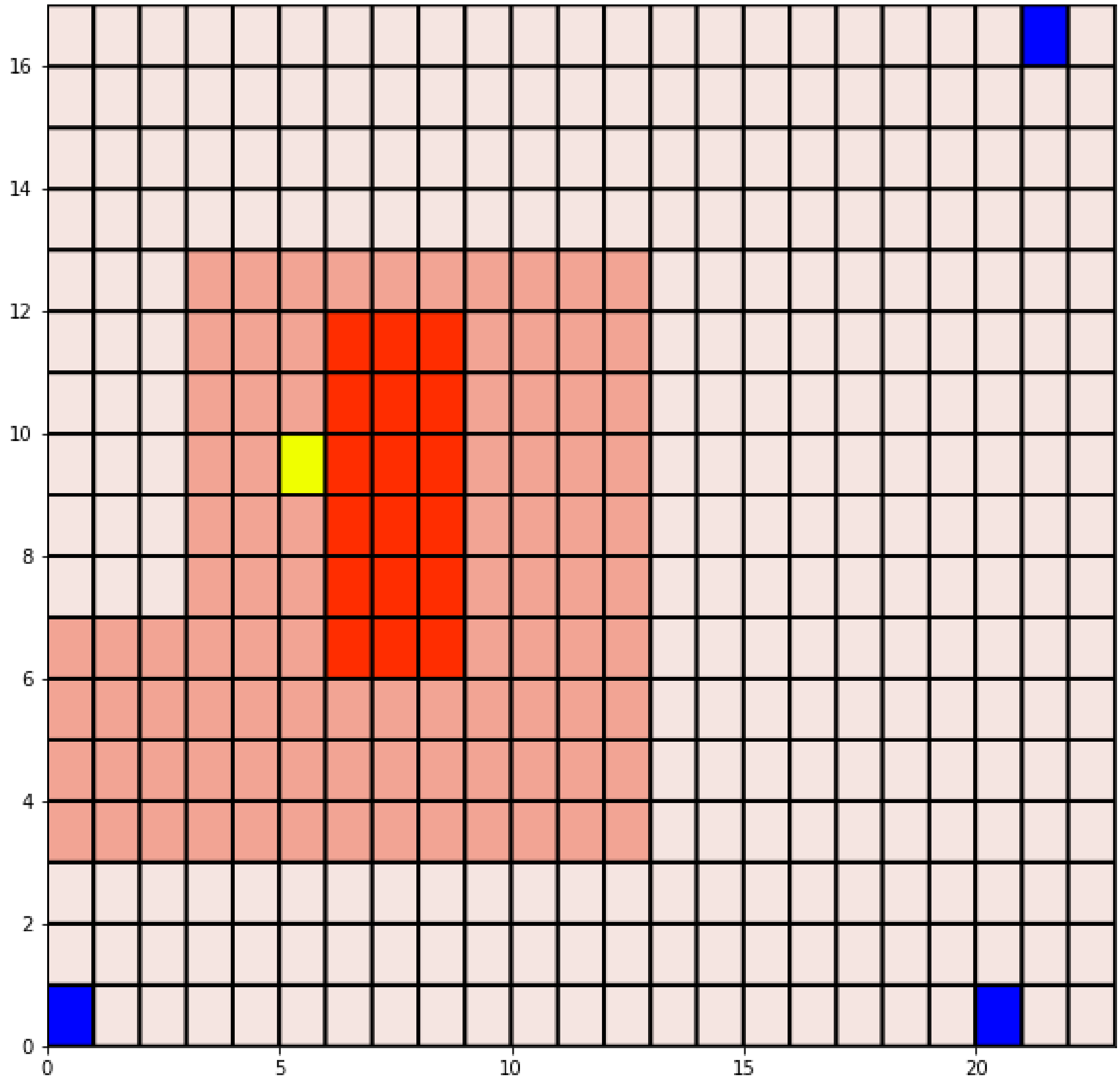

```
100%|██████████| 10000/10000 [01:32<00:00, 107.67it/s]
```

Visualizing the policy

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

In [35]:

```
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: 50, Total Reward: 95
```

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.

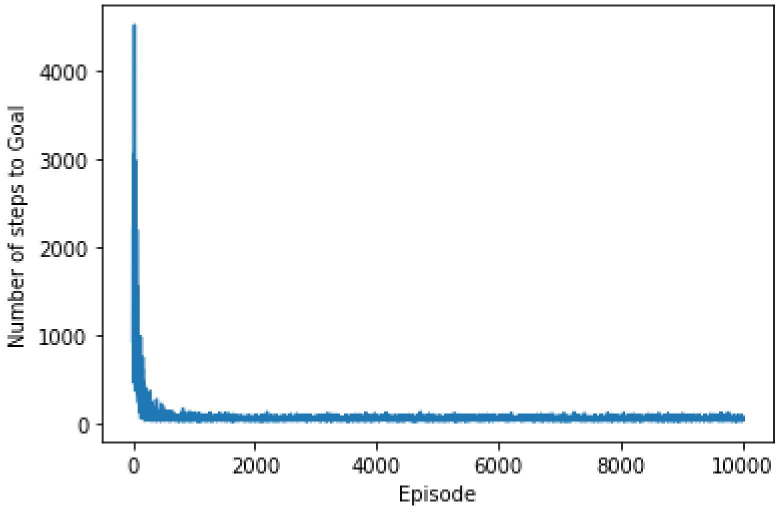
In [12]:

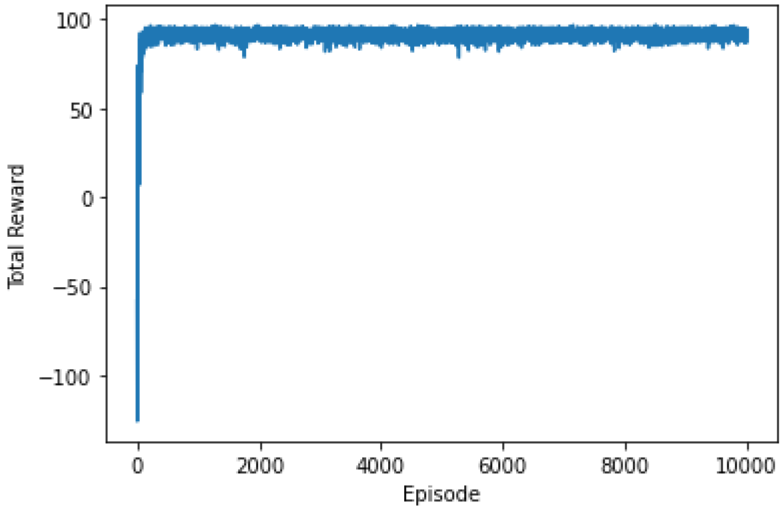
```
num_expts = 5
reward_avgs, steps_avgs = [], []

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

    # Run SARSA and store metrics
    Q, rewards, steps = sarsa(env, Q, gamma = gamma, plot_heat=False, choose_action= ch
oose_action_softmax)
    reward_avgs.append(rewards)
    steps_avgs.append(steps)

# Compute average metrics across experiments
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, 199.25it/s]

Experiment: 2

100%|██████████| 10000/10000 [01:05<00:00, 153.07it/s]

Experiment: 3

100%|██████████| 10000/10000 [00:52<00:00, 191.69it/s]

Experiment: 4

100%|██████████| 10000/10000 [01:00<00:00, 165.76it/s]

Experiment: 5

100%|██████████| 10000/10000 [00:52<00:00, 191.03it/s]
```

In [13]:

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

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

plt.figure()
plt.plot(np.arange(episodes), reward_avgs)
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:

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

Visualize and compare results with SARSA.

In [14]:

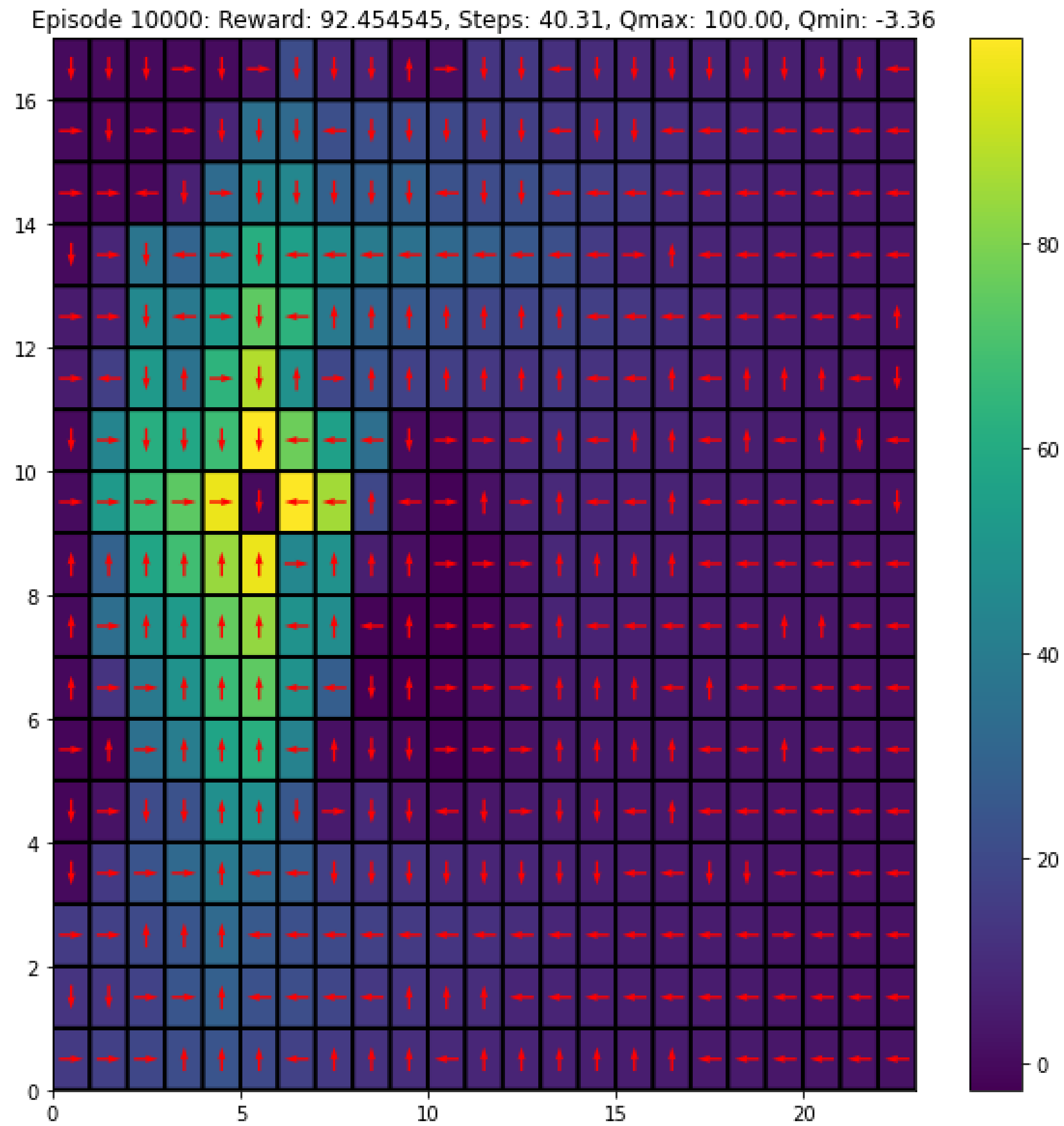
```
# initialize Q-value
Q = np.zeros((env.grid.shape[0], env.grid.shape[1], len(env.action_space)))

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

In [15]:

```
print_freq = 100

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

    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
            max_next_Q = np.max(Q[state_next[0], state_next[1], :])
            Q[state[0], state[1], action] += alpha * (reward + gamma * max_next_Q - Q[s
tate[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(step
s_to_completion[ep-print_freq+1:ep]),
                                                                           Q.max(), Q.m
in()))

    return Q, episode_rewards, steps_to_completion
```

In [16]:

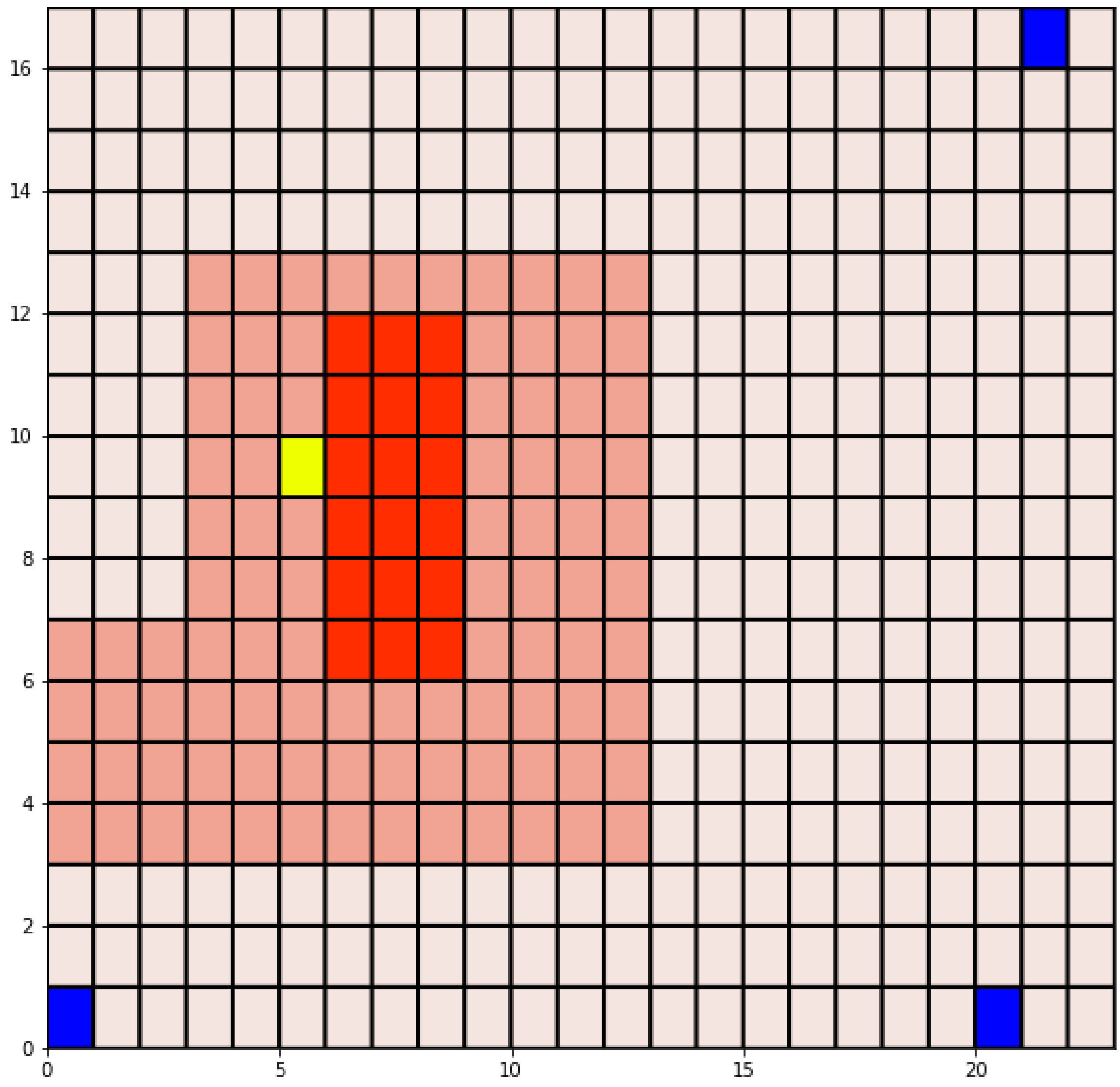
```
Q, rewards, steps = qlearning(env, Q, gamma = gamma, plot_heat=True, choose_action= cho
ose_action_softmax)
```




```
100%|██████████| 10000/10000 [01:23<00:00, 120.43it/s]
```

In [33]:

```
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: 27, Total Reward: 95
```

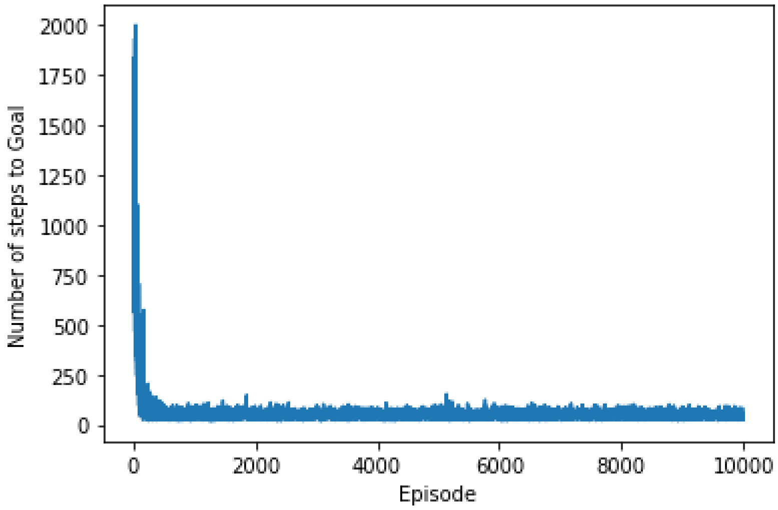
In [18]:

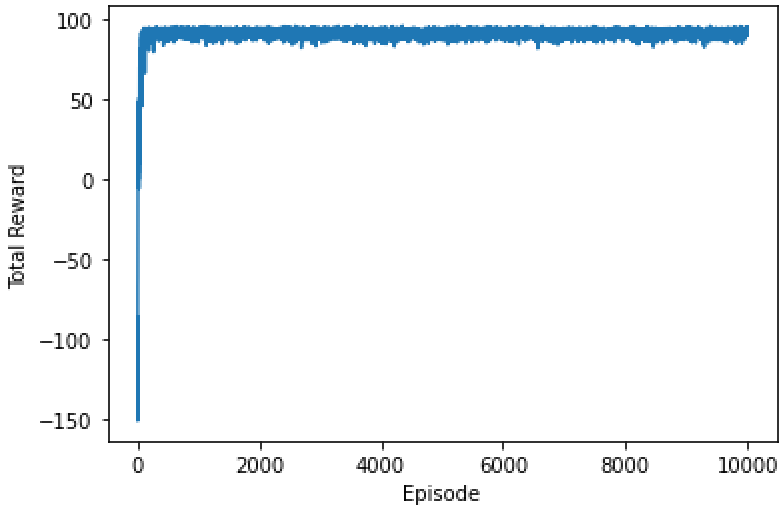
```
num_expts = 5
reward_avgs, steps_avgs = [], []

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 = gamma, plot_heat=False, choose_action= choose_action_softmax)

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

reward_avgs = np.mean(reward_avgs, axis=0)
steps_avgs = np.mean(steps_avgs, axis=0)
```

```
Experiment: 1

100%|██████████| 10000/10000 [00:57<00:00, 174.26it/s]

Experiment: 2

100%|██████████| 10000/10000 [00:57<00:00, 175.20it/s]

Experiment: 3

100%|██████████| 10000/10000 [00:51<00:00, 194.49it/s]

Experiment: 4

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

Experiment: 5

100%|██████████| 10000/10000 [00:55<00:00, 178.83it/s]
```

In [19]:

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

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

plt.figure()
plt.plot(np.arange(episodes), reward_avgs)
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.show()
```





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

The difference between SARSA and Q-learning policies are due to the fact that SARSA is an On-Policy Algorithm i.e. we need to know the next action our policy takes in order to perform an update step, where as Q-learning is an Off-Policy Algorithm i.e the policy we are updating differs in behavior from the policy we use to explore the world. Q-learning uses policy based on the optimal policy which always chooses the action with the highest Q-value.

The on-policy SARSA agent views the dark red zone (-2 reward) as riskier because it chooses and updates actions subject to its stochastic policy. That means it has learned that it has a high likelihood of receiving a high negative reward and thus takes a longer route most of the time moving on white squares rather than taking shorter route through light red squares (-1 reward).

In contrast, the Q-learning agent has learned its policy based on the optimal policy which always chooses the action with the highest Q-value. It is more confident in its ability to travel along the dark red zone (-2 reward) without taking a greater penalty.

We can observe this as SARSA took 50 steps to get the reward of 95 while Q-learning took only 27 steps to get the same reward from the same starting position on the grid.

In [38]:

```
!jupyter nbconvert --to html "/content/drive/MyDrive/Colab Notebooks/CS6700_Tutorial_4_
QLearning_SARSA_EE21D411.ipynb"
```

```
[NbConvertApp] Converting notebook /content/drive/MyDrive/Colab Notebooks/
CS6700_Tutorial_4_QLearning_SARSA_EE21D411.ipynb to html
[NbConvertApp] Writing 788993 bytes to /content/drive/MyDrive/Colab Notebo
oks/CS6700_Tutorial_4_QLearning_SARSA_EE21D411.html
```

In [ ]:

```
```