In [1]:
```
# Install relevant libraries
!pip install numpy matplotlib tqdm scipy
```

```
Requirement already satisfied: numpy in c:\users\student\.conda\envs\dsai\lib\site-packages (1.24.3)
Requirement already satisfied: matplotlib in c:\users\student\.conda\envs\dsai\lib\site-packages (3.9.2)
Requirement already satisfied: tqdm in c:\users\student\.conda\envs\dsai\lib\site-packages (4.66.5)
Requirement already satisfied: scipy in c:\users\student\.conda\envs\dsai\lib\site-packages (1.14.1)
Requirement already satisfied: contourpy>=1.0.1 in c:\users\student\.conda\envs\dsai\lib\site-packages (from matplotli
b) (1.2.0)
Requirement already satisfied: cycler>=0.10 in c:\users\student\.conda\envs\dsai\lib\site-packages (from matplotlib)
(0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in c:\users\student\.conda\envs\dsai\lib\site-packages (from matplotli
b) (4.51.0)
Requirement already satisfied: kiwisolver>=1.3.1 in c:\users\student\.conda\envs\dsai\lib\site-packages (from matplotli
b) (1.4.4)
Requirement already satisfied: packaging>=20.0 in c:\users\student\.conda\envs\dsai\lib\site-packages (from matplotlib)
(24.1)
Requirement already satisfied: pillow>=8 in c:\users\student\.conda\envs\dsai\lib\site-packages (from matplotlib) (10.
4.0)
Requirement already satisfied: pyparsing>=2.3.1 in c:\users\student\.conda\envs\dsai\lib\site-packages (from matplotli
b) (3.1.2)
Requirement already satisfied: python-dateutil>=2.7 in c:\users\student\.conda\envs\dsai\lib\site-packages (from matplo
tlib) (2.9.0.post0)
Requirement already satisfied: colorama in c:\users\student\.conda\envs\dsai\lib\site-packages (from tqdm) (0.4.6)
Requirement already satisfied: six>=1.5 in c:\users\student\.conda\envs\dsai\lib\site-packages (from python-dateutil>=
2.7->matplotlib) (1.16.0)
```

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]:
```
!type grid_world2.txt
```

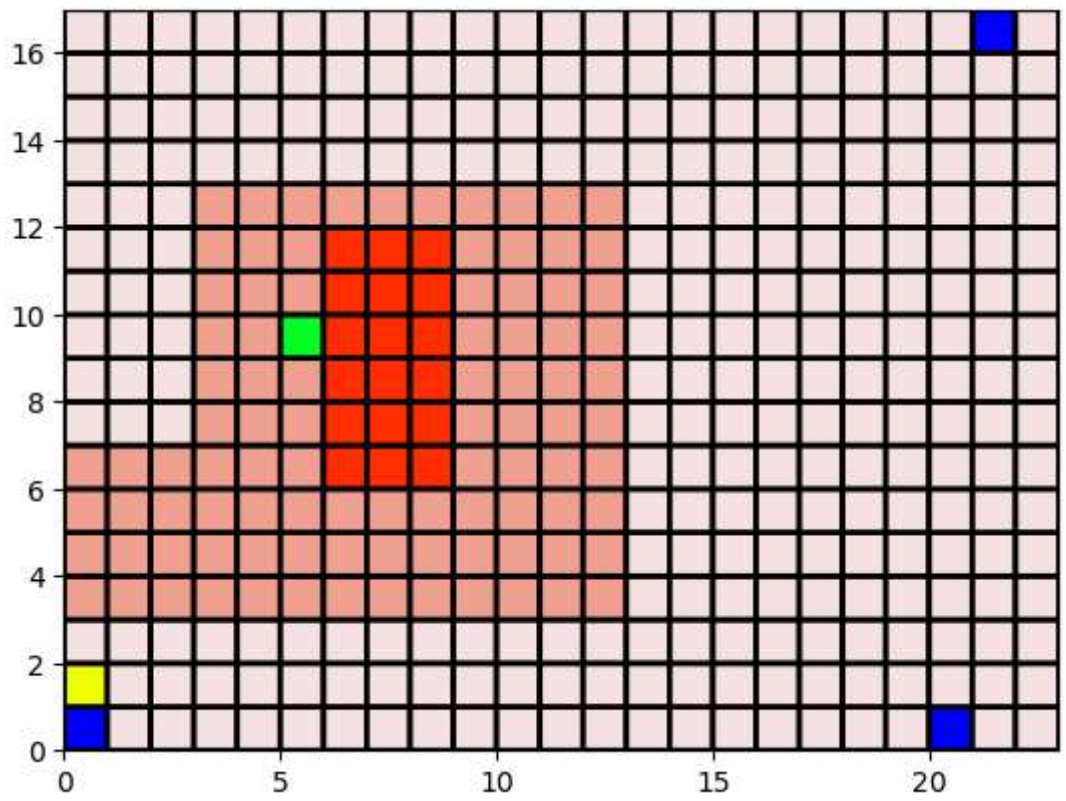
```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
1 1 1 1 1 1 2 2 2 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 2 2 2 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 2 2 2 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 2 2 2 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 2 2 2 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 1 1 2 2 2 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 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 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 [8]:
```
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))
plt.show()
# Go UP
env.step(UP)
env.render(ax=plt, render_agent=True)
```




```
<Figure size 1000x1000 with 0 Axes>
```

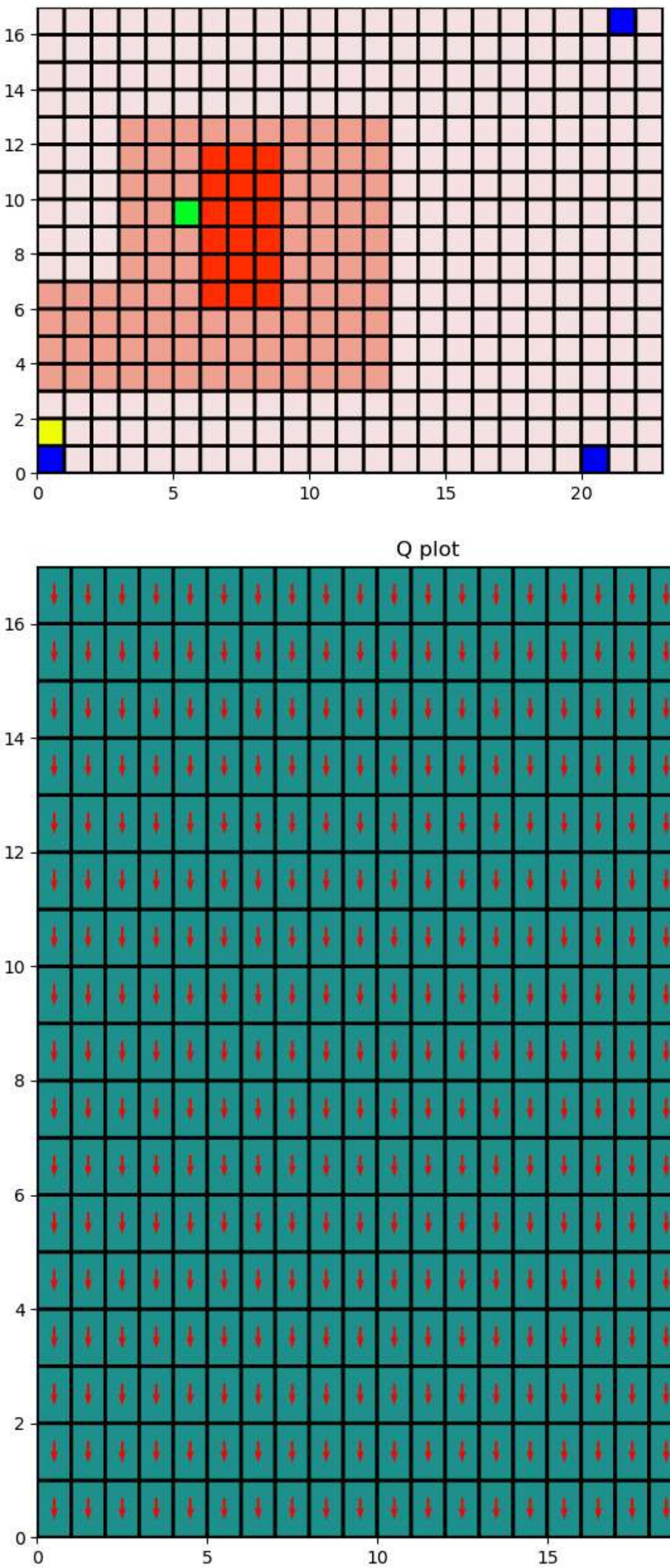
Legend

- *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 [9]:
```
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[9]: (17, 23, 4)

Exploration strategies

1. Epsilon-greedy
2. Softmax

In [10]:
```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() 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

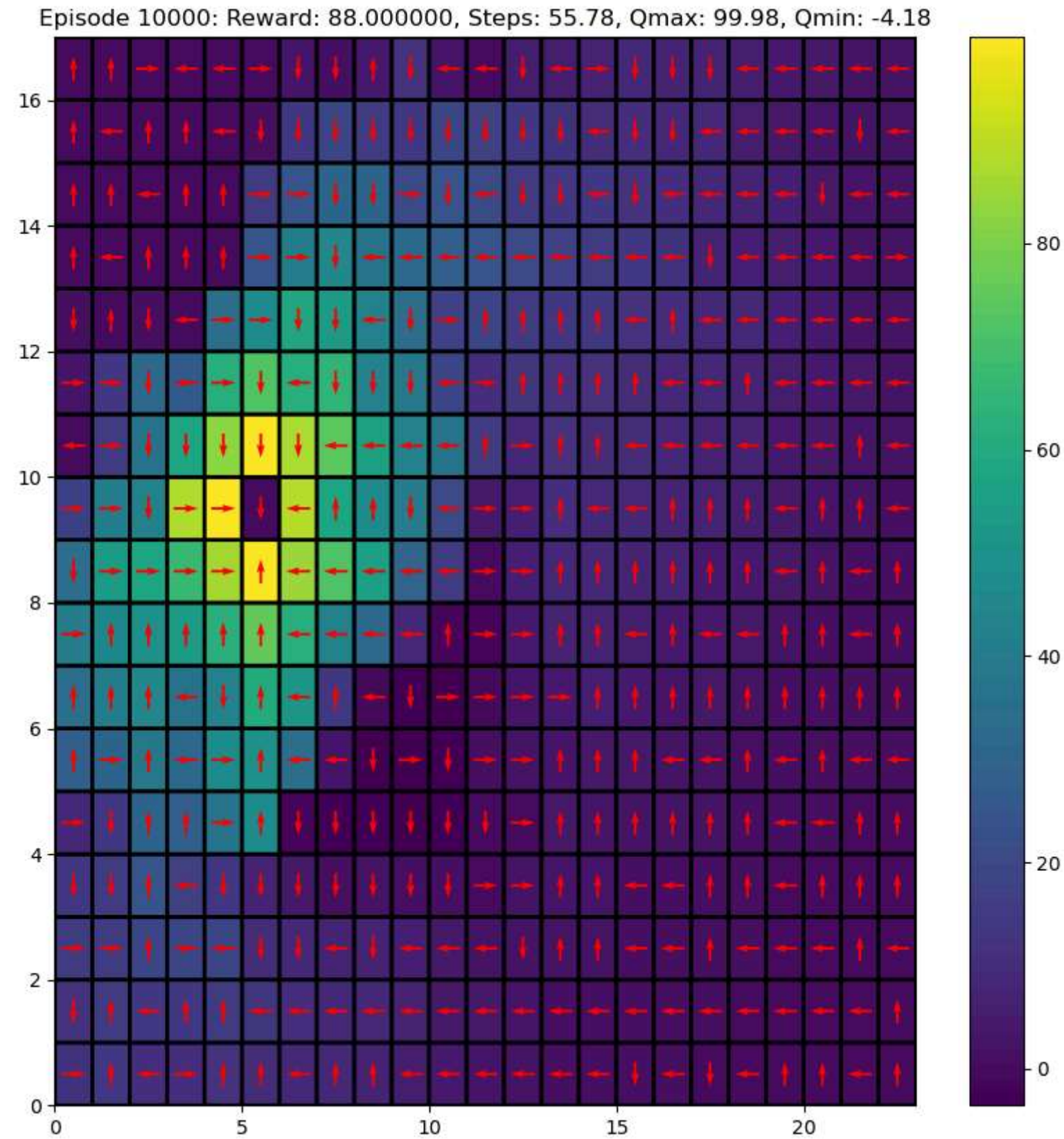
In [11]:
```python
# initialize Q-value
Q = np.zeros((env.grid.shape[0], env.grid.shape[1], len(env.action_space)))

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

Let's implement SARSA

In [12]:

```
print_freq = 10

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], 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
```

In [13]:
```
Q, rewards, steps = sarsa(env, Q, gamma = gamma, plot_heat=True, choose_action= choose_action_softmax)
```

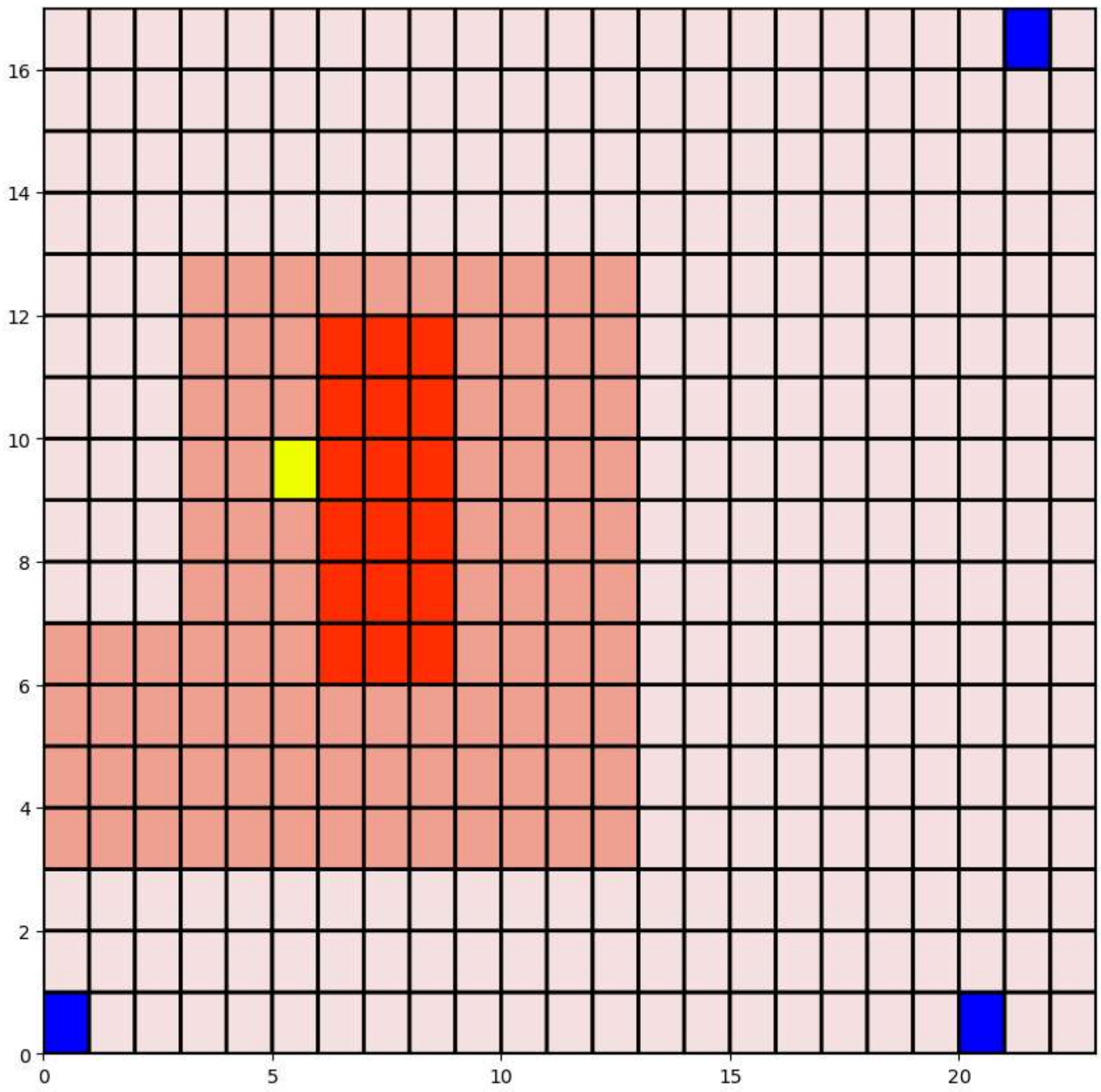


100%|██████████| 10000/10000 [01:56<00:00, 85.97it/s]

Visualizing the policy

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

In [14]:
```
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: 20, Total Reward: 89
```

Analyzing performance of the policy

We use two metrics to analyze the policies:

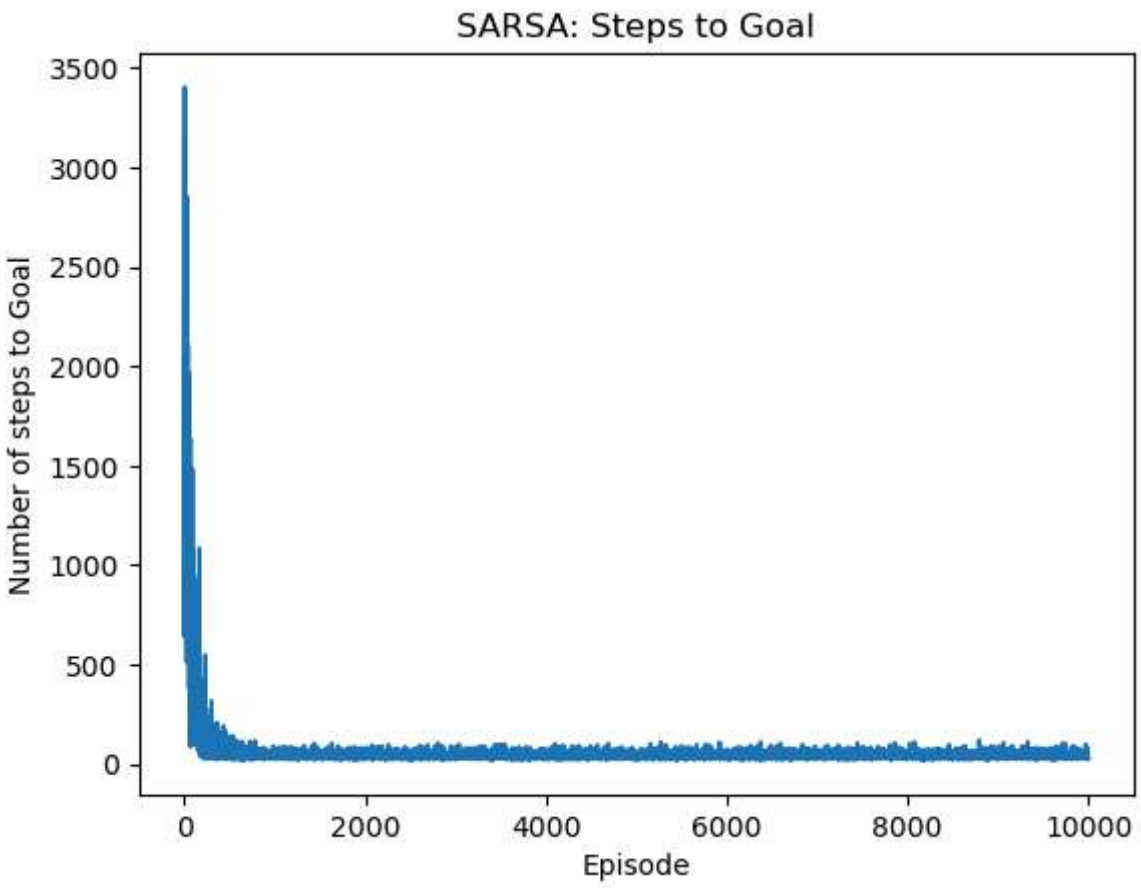
1. Average steps to reach the goal
2. Total rewards from the episode

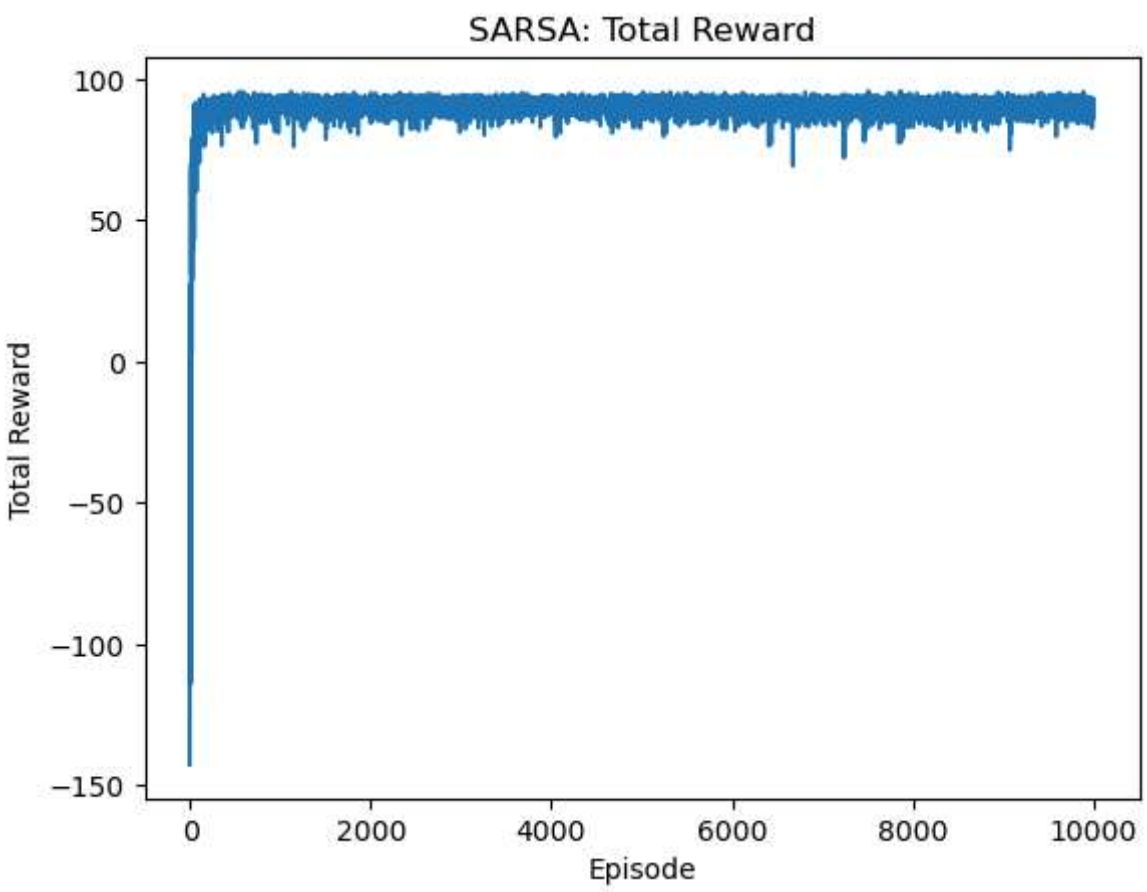
To ensure, we account for randomness in environment and algorithm (say when using epsilon-greedy exploration), we run the algorithm for multiple times and use the average of values over all runs.

In [15]:
```
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 [00:09<00:00, 1025.89it/s]

Experiment: 2

100%|██████████| 10000/10000 [00:09<00:00, 1068.88it/s]

Experiment: 3

100%|██████████| 10000/10000 [00:14<00:00, 702.39it/s]

Experiment: 4

100%|██████████| 10000/10000 [00:10<00:00, 943.97it/s]

Experiment: 5

100%|██████████| 10000/10000 [00:10<00:00, 979.99it/s]
```

In [16]:
```
plt.xlabel('Episode')
plt.ylabel('Number of steps to Goal')
plt.title('SARSA: Steps to Goal')
plt.plot(np.arange(episodes),np.average(steps_avgs, 0))
plt.show()
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.title('SARSA: Total Reward')
plt.plot(np.arange(episodes),np.average(reward_avgs, 0))
plt.show()
```





TODO: Q-Learning

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

Recall the update rule for Q-Learning:

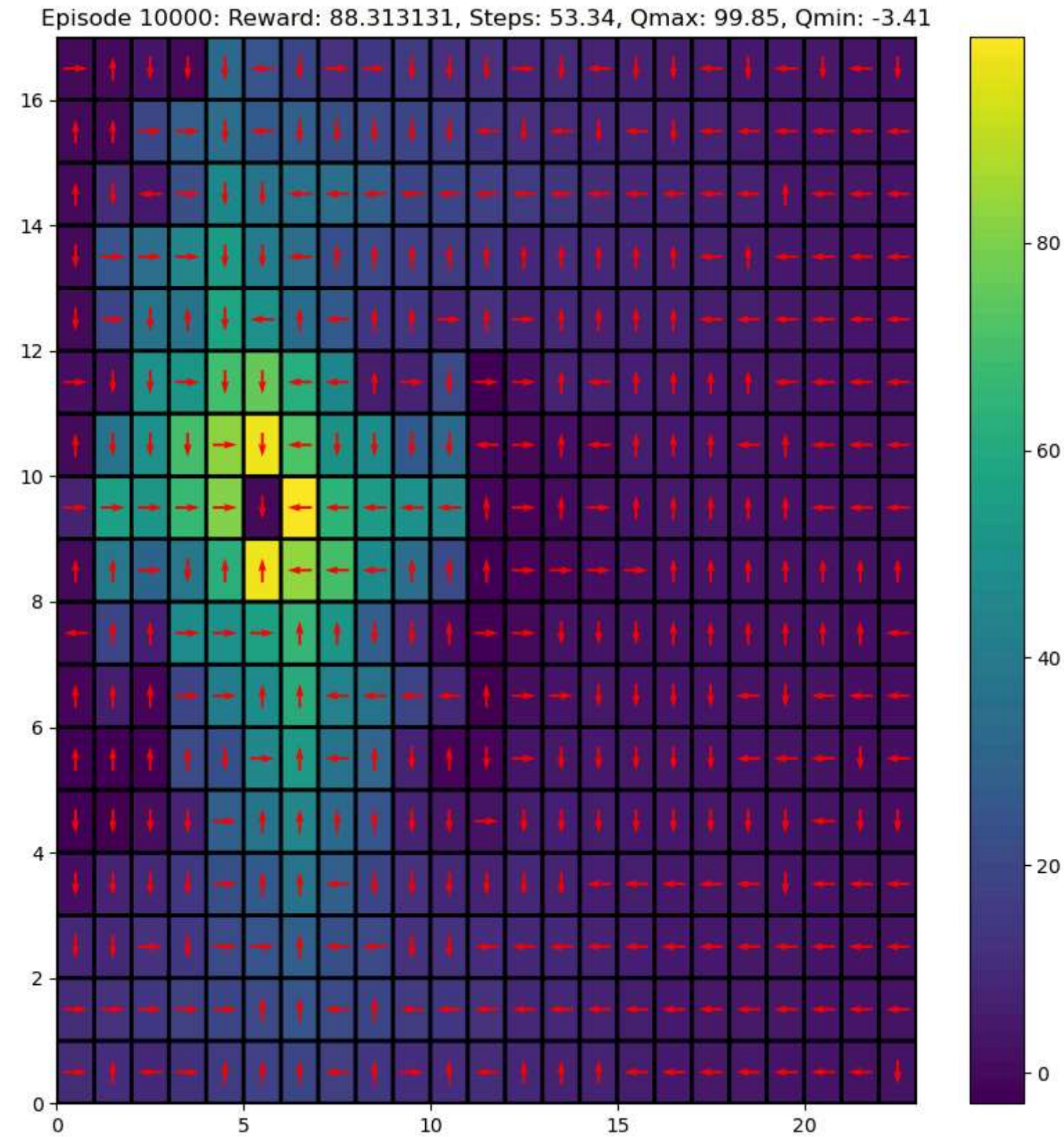
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Visualize and compare results with SARSA. Repeat experiments 5 times (as in SARSA) and report average number of steps to goal and average episodic reward. You may use the same hyperparameters as SARSA.

In [17]:

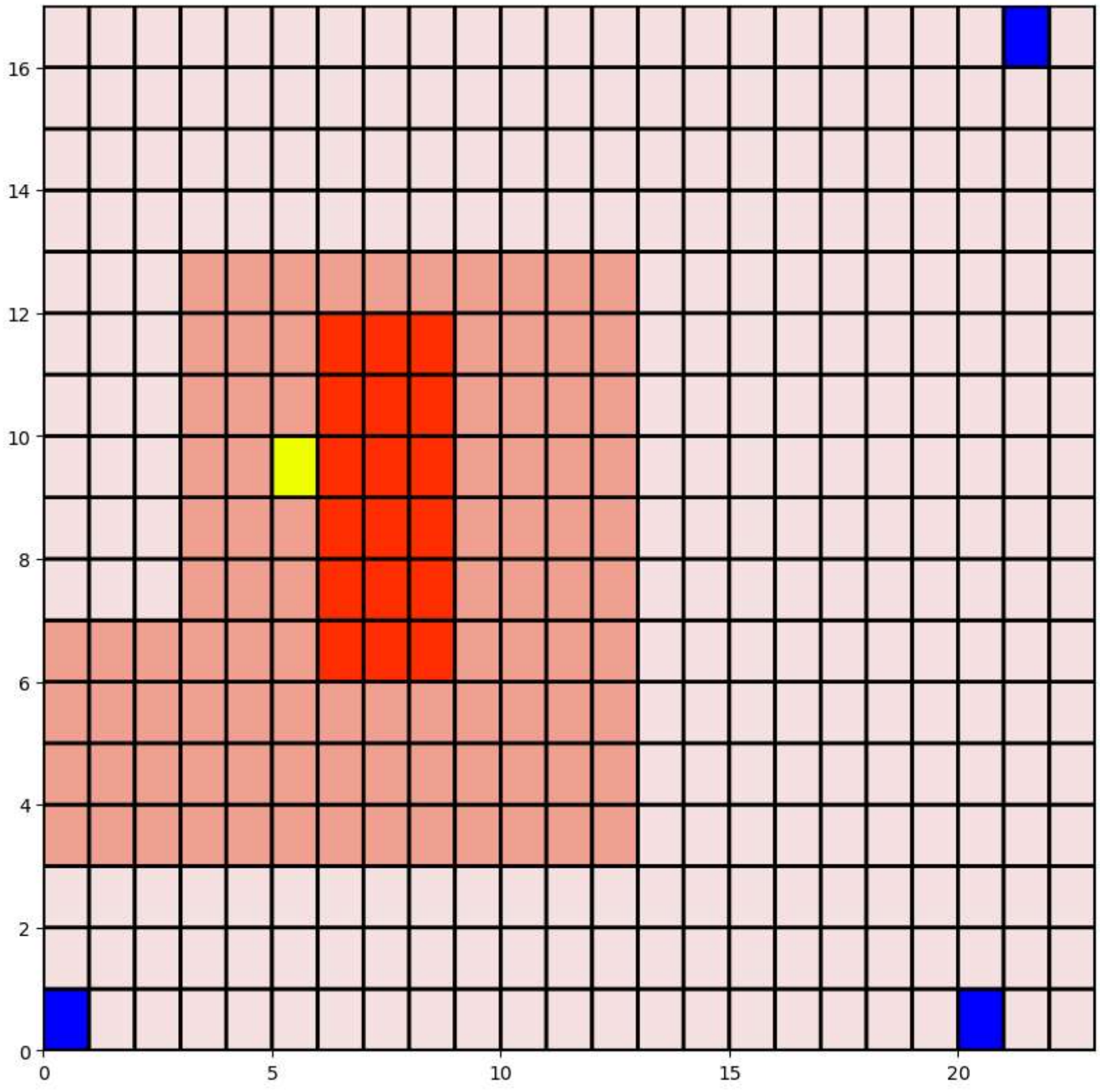
```
### WRITE CODE HERE ### (Q-Learning)
print_freq = 100

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

    state = env.reset()
    done = False
    while not done:
      action = choose_action(Q, state) #- > softmax
      #action = choose_action(Q, state, epsilon) # -> Epsilon greedy
      state_next, reward, done = env.step(action)

      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 = state_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


```

In [18]:
```
Q, rewards, steps = QLearning(env, Q, gamma = gamma, plot_heat=True, choose_action= choose_action_softmax)
```



100%|██████████| 10000/10000 [00:19<00:00, 503.23it/s]

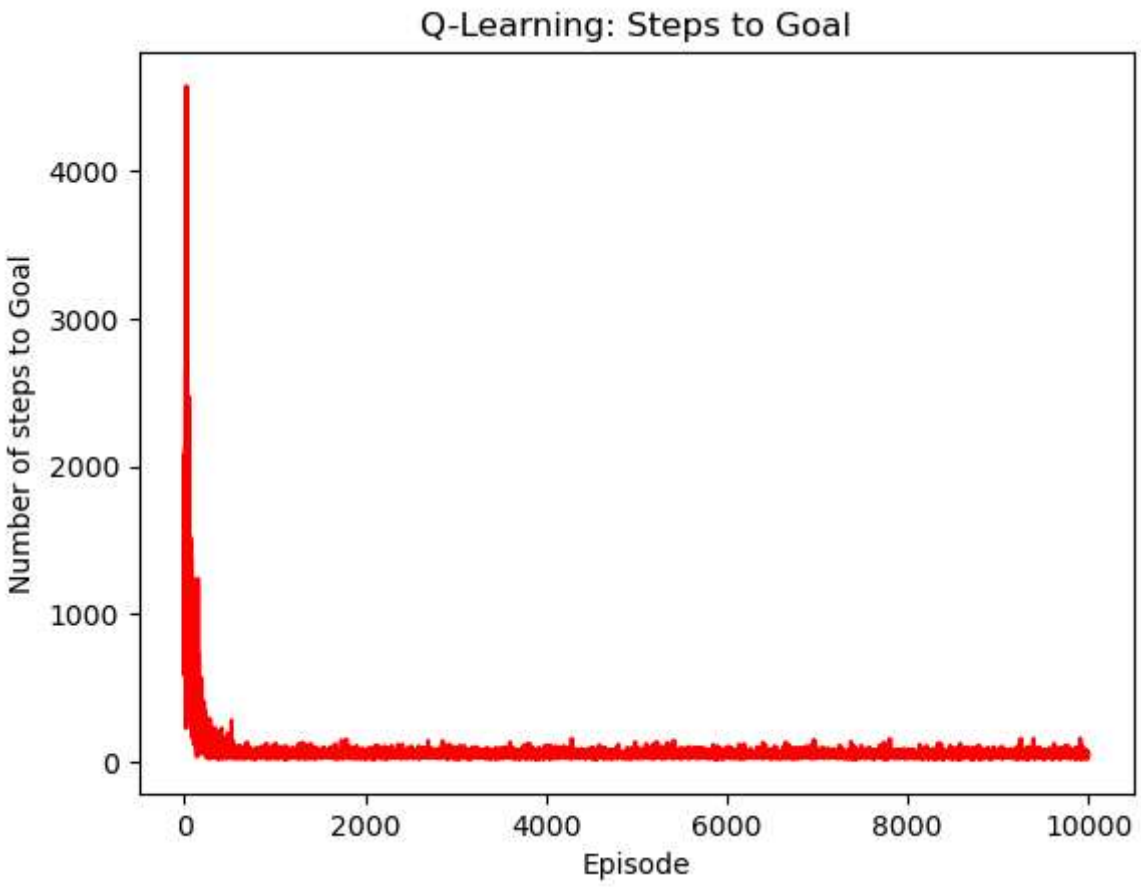
In [19]:
```
### WRITE CODE HERE ### (visualize policy)
from time import sleep

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))

```

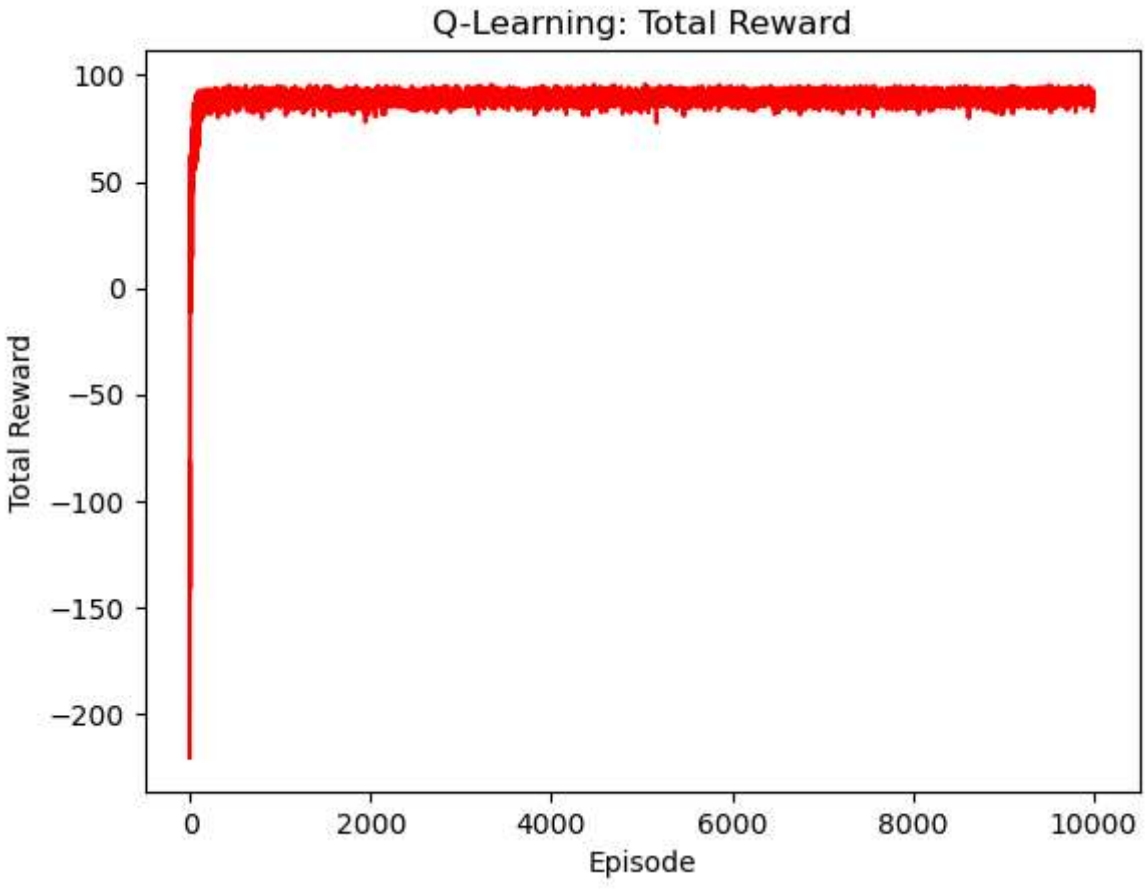



```
Steps: 31, Total Reward: 88
```

In [20]:
```
### WRITE CODE HERE ### (plot metrics avged over 5 runs)

Q_avgs_Qlearning, reward_avgs_Qlearning, steps_avgs_Qlearning = [], [], []
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_Qlearning.append(Q.copy())
    reward_avgs_Qlearning.append(rewards)
    steps_avgs_Qlearning.append(steps)

plt.xlabel('Episode')
plt.ylabel('Number of steps to Goal')
plt.title('Q-Learning: Steps to Goal')
plt.plot(np.arange(episodes), np.average(steps_avgs_Qlearning, 0), color='red')
plt.show()

plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.title('Q-Learning: Total Reward')
plt.plot(np.arange(episodes), np.average(reward_avgs_Qlearning, 0), color='red')
plt.show()
```

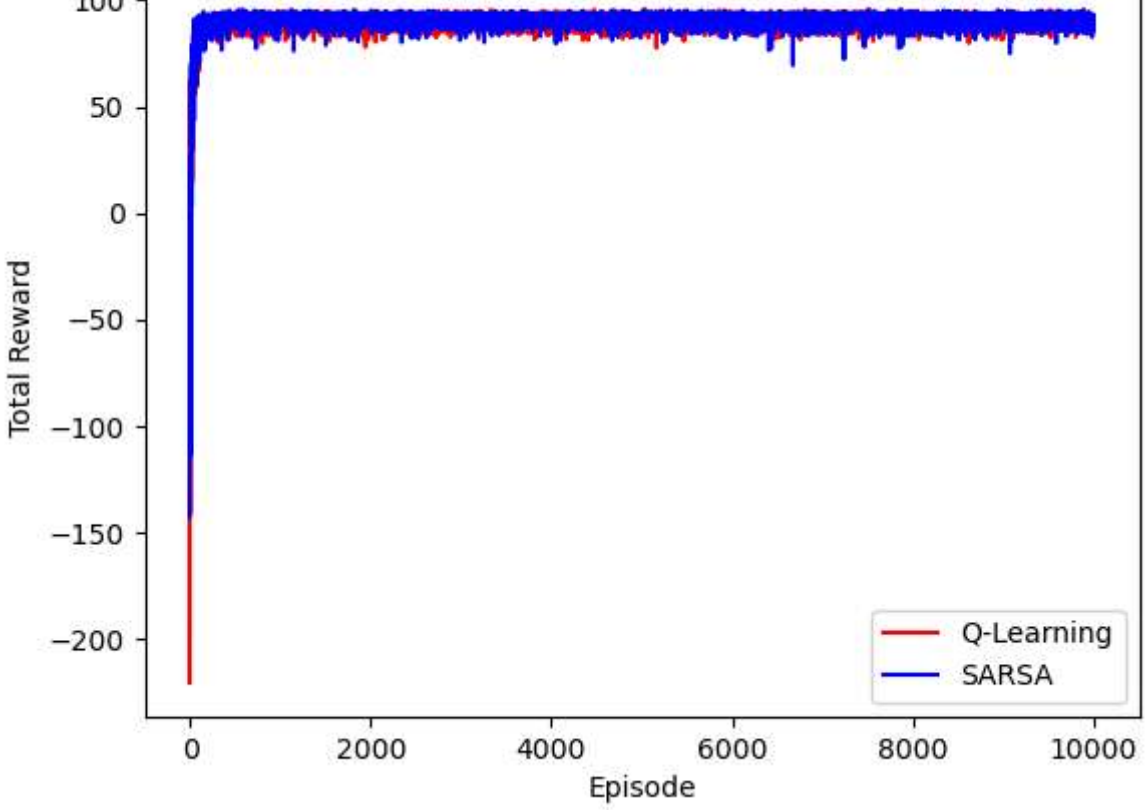
```
Experiment: 1
100%|██████████| 10000/10000 [00:16<00:00, 599.78it/s]
Experiment: 2
100%|██████████| 10000/10000 [00:10<00:00, 916.98it/s]
Experiment: 3
100%|██████████| 10000/10000 [00:10<00:00, 920.84it/s]
Experiment: 4
100%|██████████| 10000/10000 [00:09<00:00, 1013.53it/s]
Experiment: 5
100%|██████████| 10000/10000 [00:13<00:00, 763.99it/s]
```




Q-Learning: Total Reward
Total Reward
100
50
0
−50
−100
−150
−200
0
2000
4000
6000
8000
10000
Episode

TODO: Compare the policies learnt by Q Learning and SARSA and note differences (if any)

In [21]:
```
### WRITE CODE HERE ### (compare Q-Learning and SARSA performance)

plt.xlabel('Episode')
plt.ylabel('Number of steps to Goal')
plt.title('Steps to Goal: SARSA vs Q-Learning')
plt.plot(np.arange(episodes), np.average(steps_avgs_Qlearning, 0), color='red', label='Q-Learning')
plt.plot(np.arange(episodes), np.average(steps_avgs, 0), color='blue', label='SARSA')
plt.legend()
plt.show()


plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.title('Total Reward: SARSA vs Q-Learning')
plt.plot(np.arange(episodes), np.average(reward_avgs_Qlearning, 0), color='red', label='Q-Learning')
plt.plot(np.arange(episodes), np.average(reward_avgs, 0), color='blue', label='SARSA')
plt.legend()
plt.show()

```



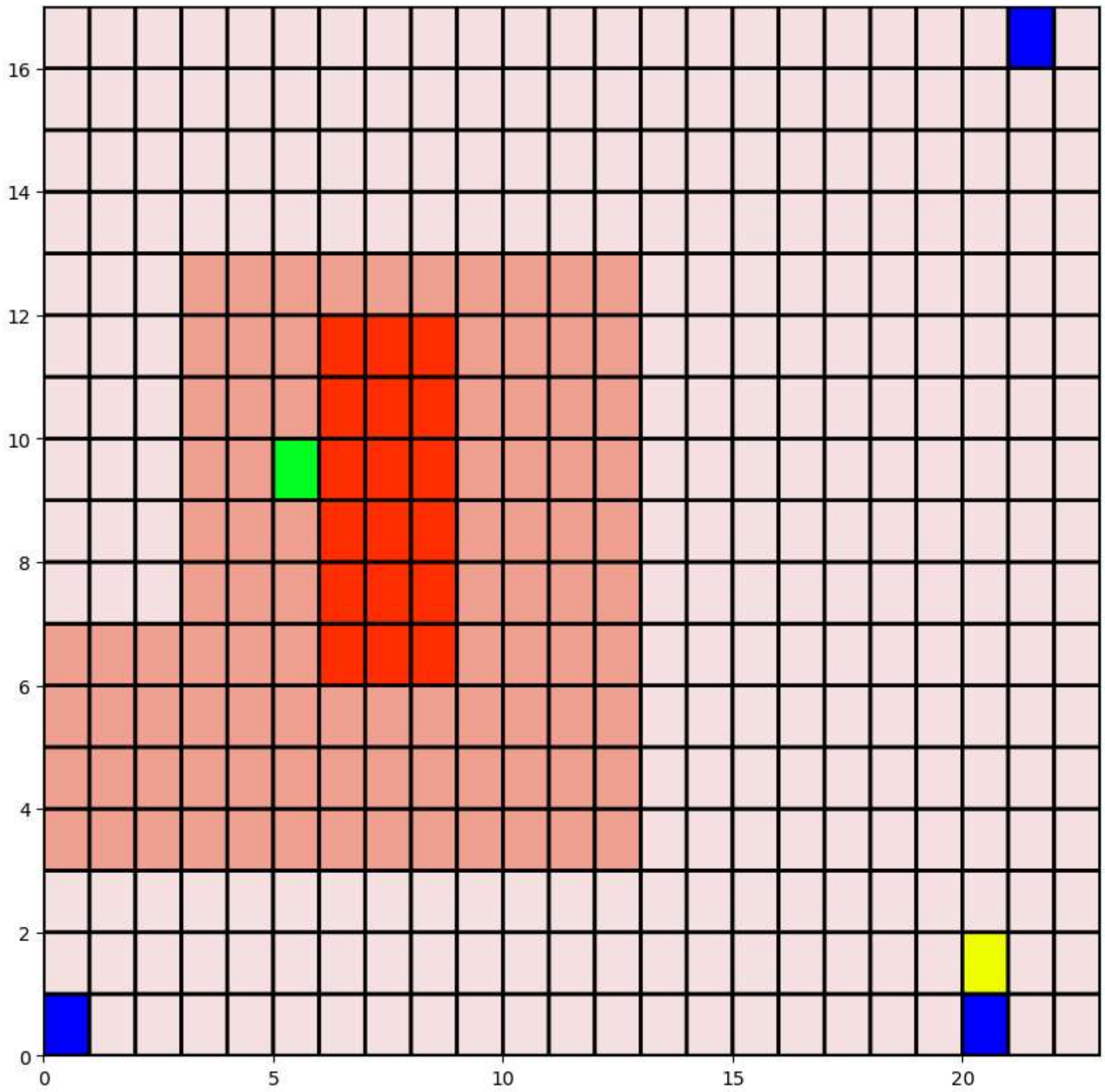
Observations:

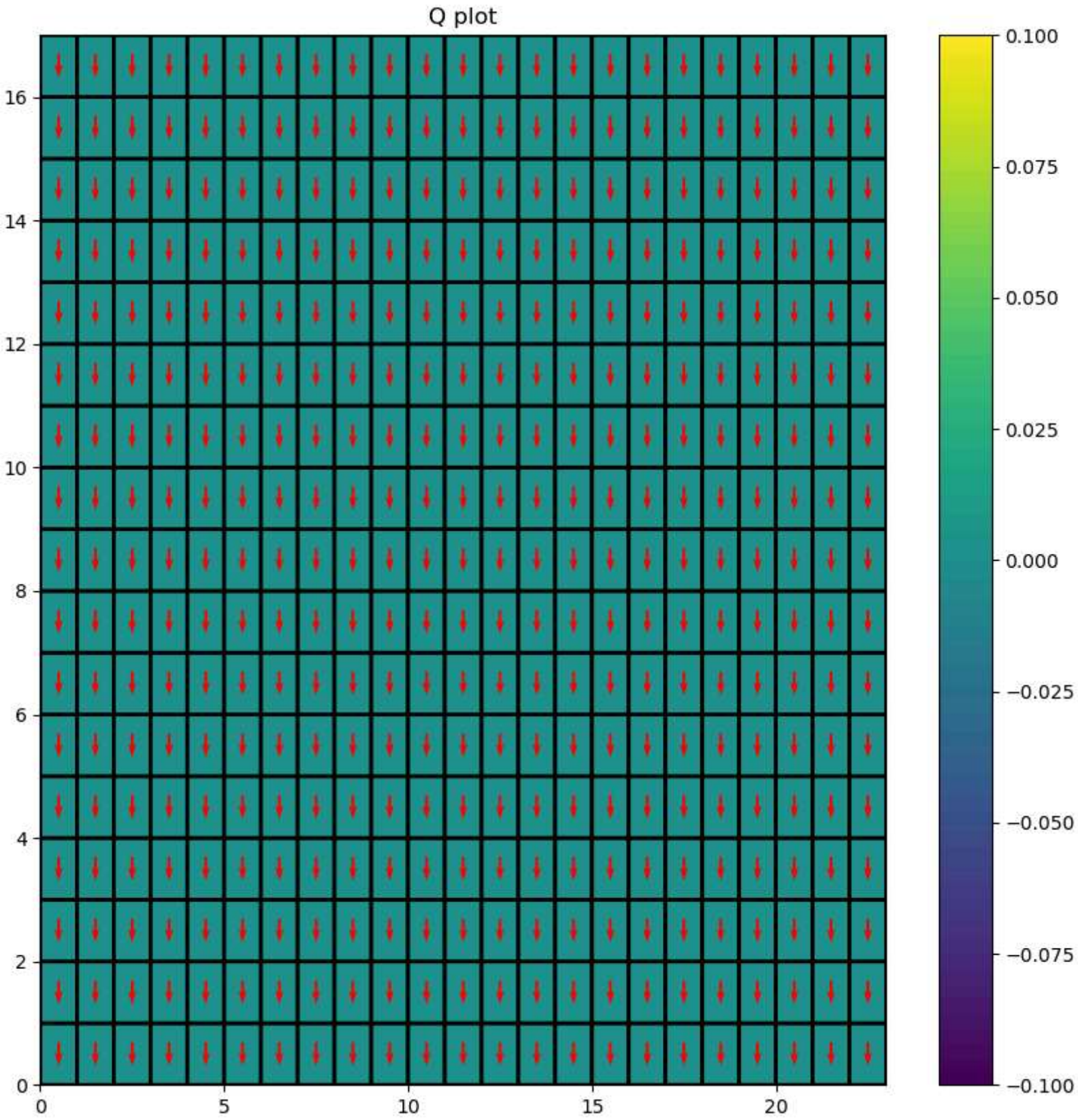
1. SARSA and Qlearning algorithms achieve similar long term rewards indicating that the algorithms converge to near optimal polices.
2. Q-Learning, tends to take slightly more steps on average as it aggressively updates based on the maximum Q values, leading to riskier exploration. In contrast, SARSA's on policy update results in a more conservative approach, often leading to convergence with fewer steps as observed in the plot.

TODO: Repeat your experiments on the windy GridWorld

In [51]:
```
### WRITE CODE HERE ### (set env to GridWorldWindyEnv)
world = 'grid_world2.txt'
goal_reward = 100
start_states = [(0,0), (0,20), (16,21)]
goal_states=[(9,5)]
max_steps=1000

from grid_world import GridWorldEnv, GridWorldWindyEnv
env = GridWorldWindyEnv(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)

```

In [52]:
```
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
```



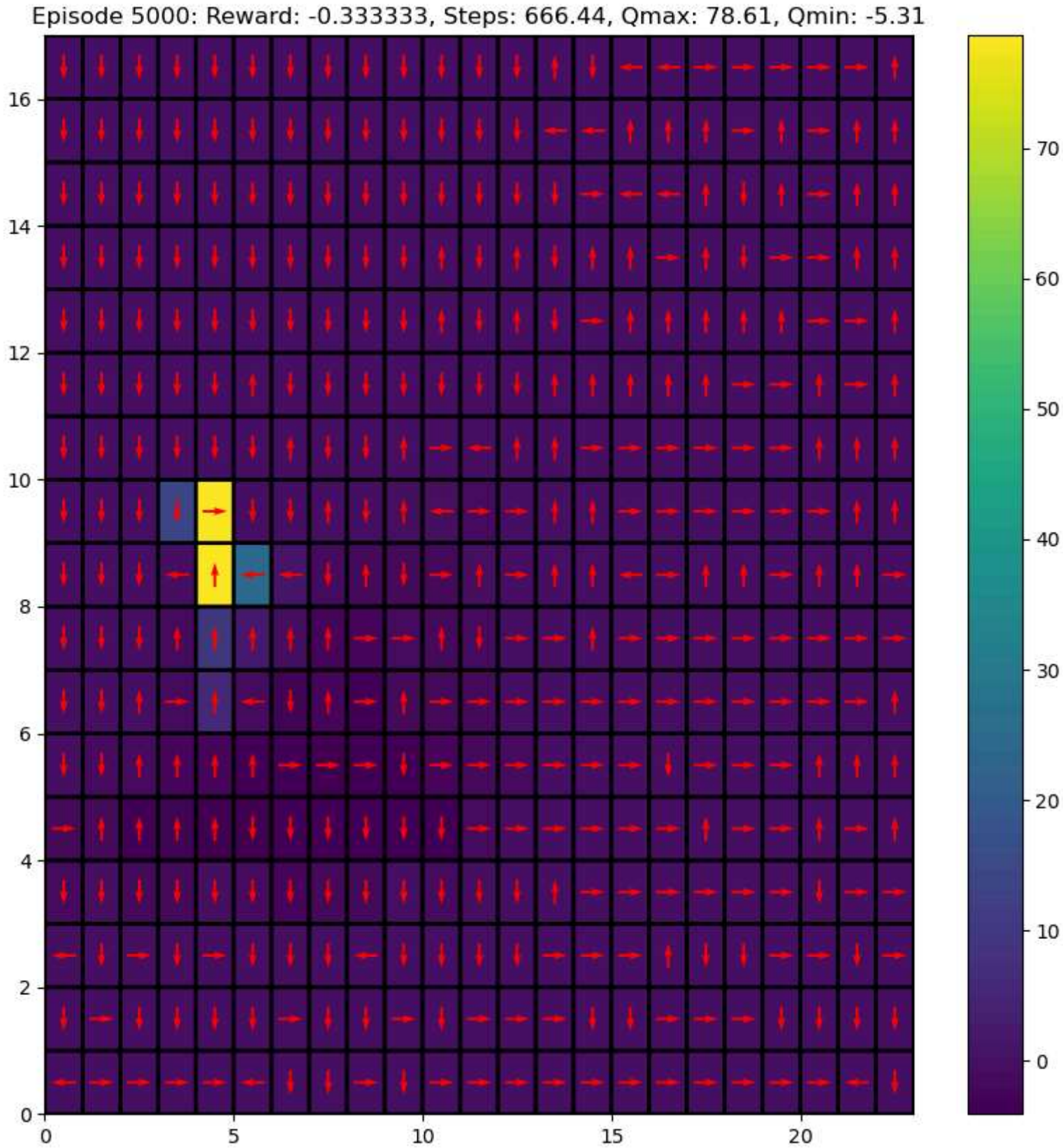


Out[52]: (17, 23, 4)

In [53]:
```
# 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 = 5000
epsilon0 = 0.1
```

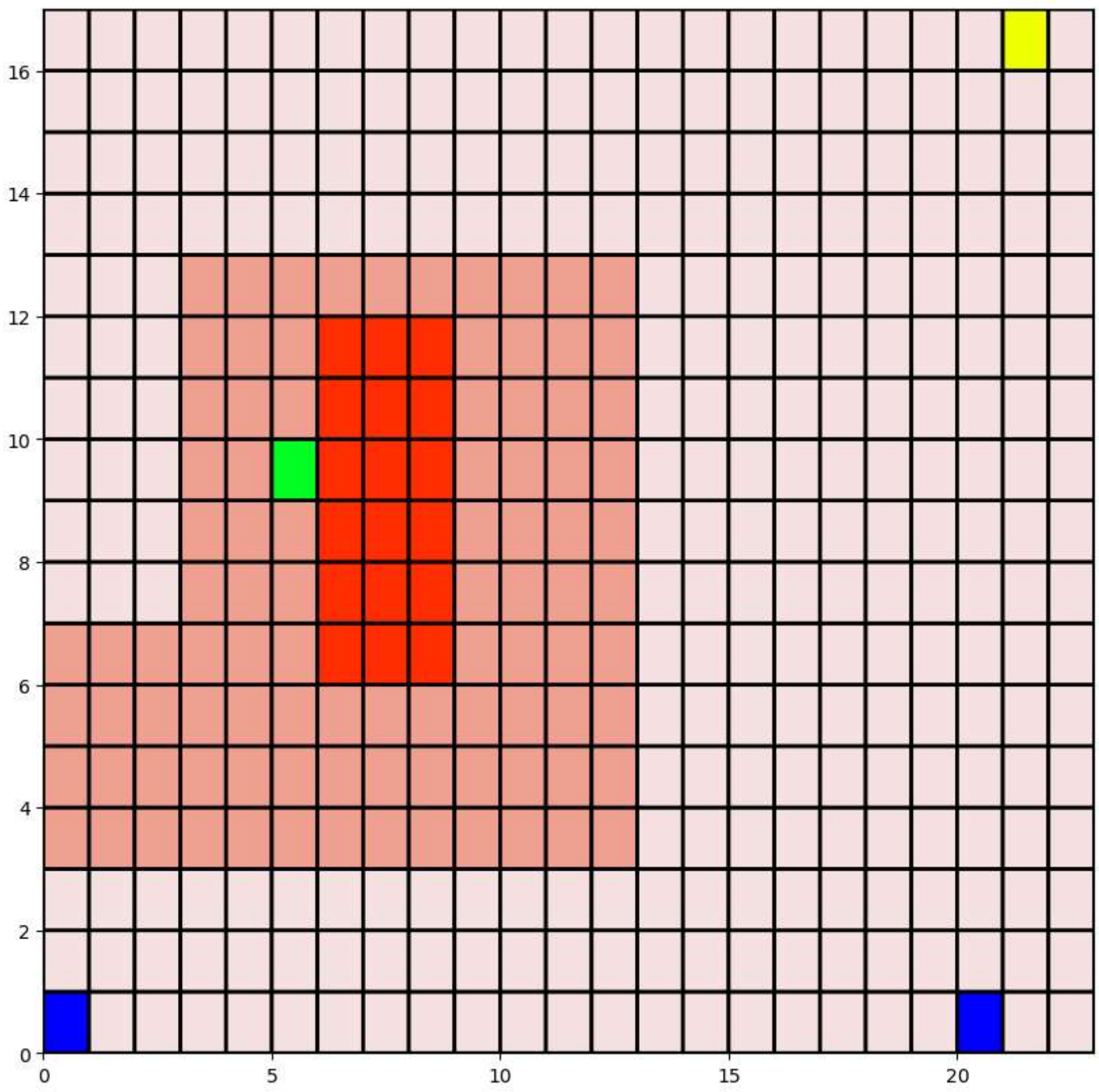
In [54]:

```
### WRITE CODE HERE ### (repeat experiments - SARSA)
print_freq = 10

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], 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= choose_action_softmax)

```



100%|██████████| 5000/5000 [02:02<00:00, 40.93it/s]

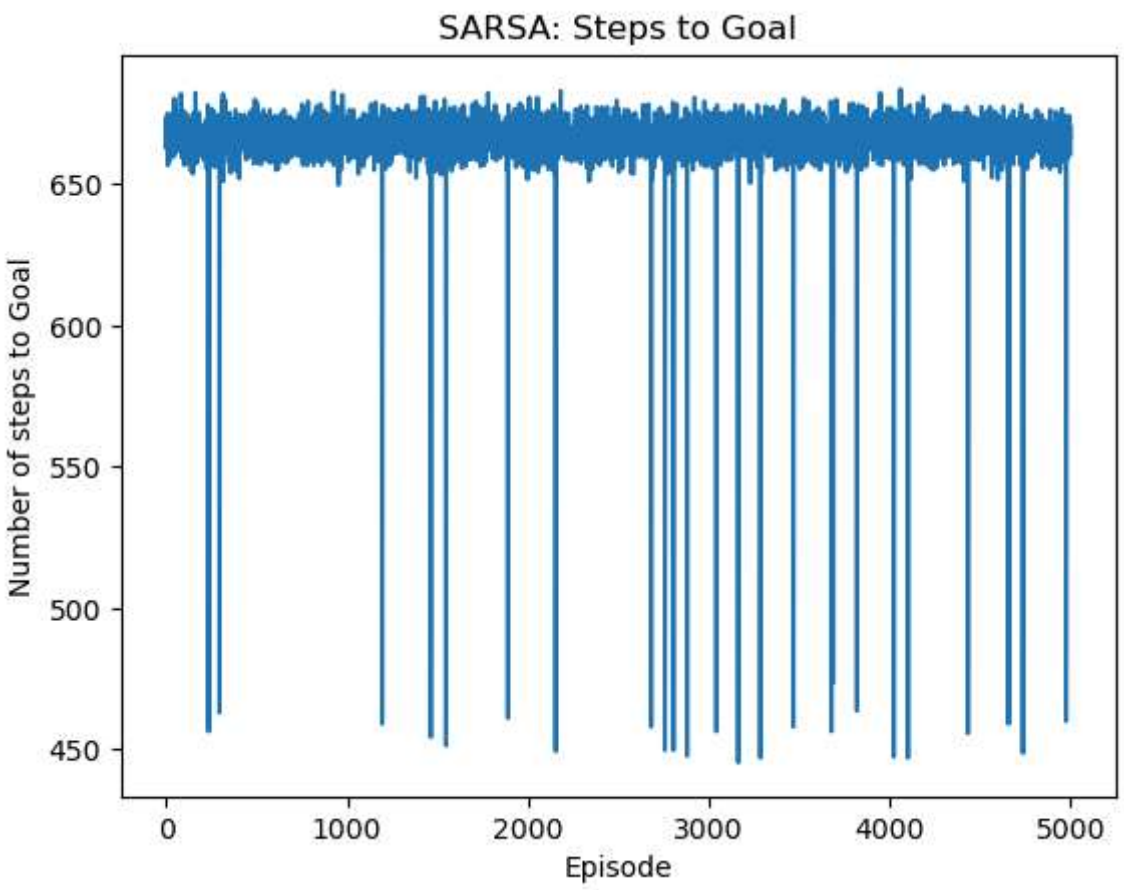
In [55]:
```
# Visualizing the policy learnt by SARSA
from time import sleep

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))
```

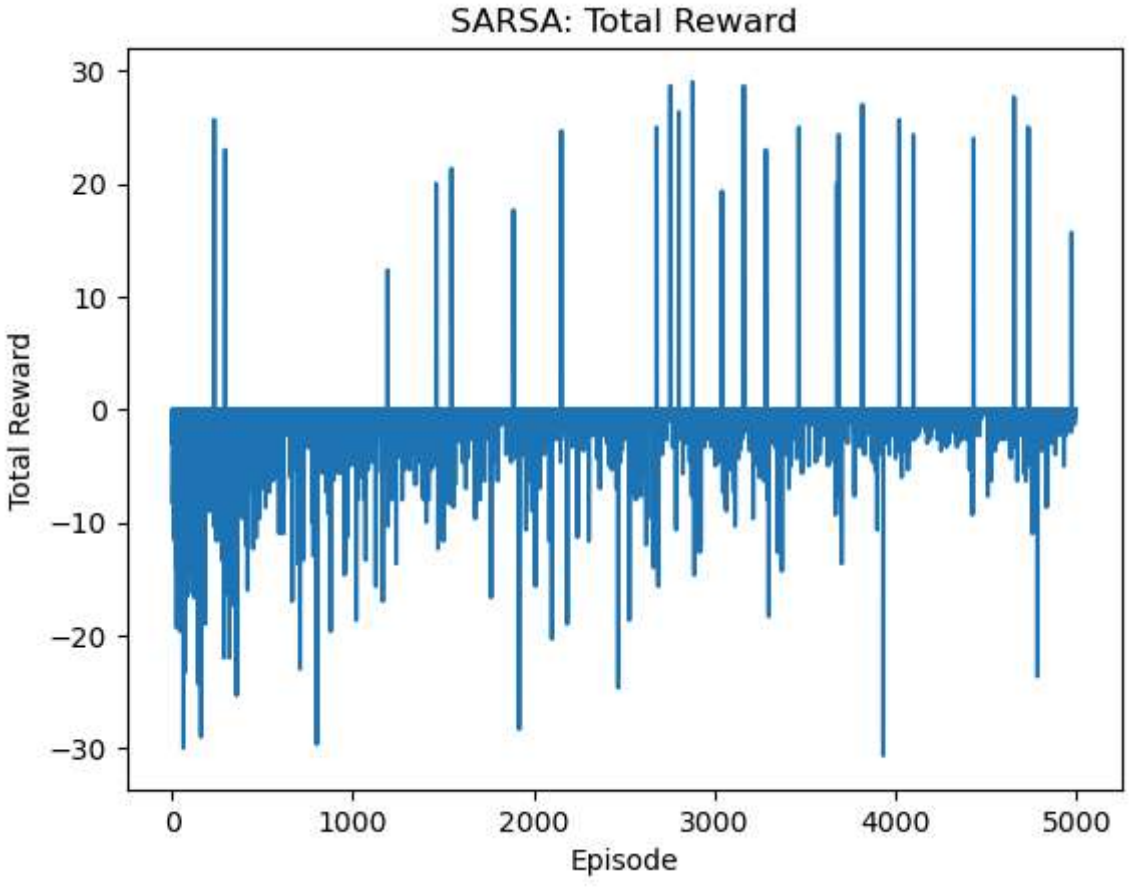



```
Steps: 659, Total Reward: 0
```

In [56]:
```
# Analyzing performance of the policy learnt by SARSA
Q_avgs_windy, reward_avgs_windy, steps_avgs_windy = [], [], []
num_expts = 3

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_windy.append(Q.copy())
    reward_avgs_windy.append(rewards)
    steps_avgs_windy.append(steps)

plt.xlabel('Episode')
plt.ylabel('Number of steps to Goal')
plt.title('SARSA: Steps to Goal')
plt.plot(np.arange(episodes),np.average(steps_avgs_windy, 0))
plt.show()
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.title('SARSA: Total Reward')
plt.plot(np.arange(episodes),np.average(reward_avgs_windy, 0))
plt.show()
```

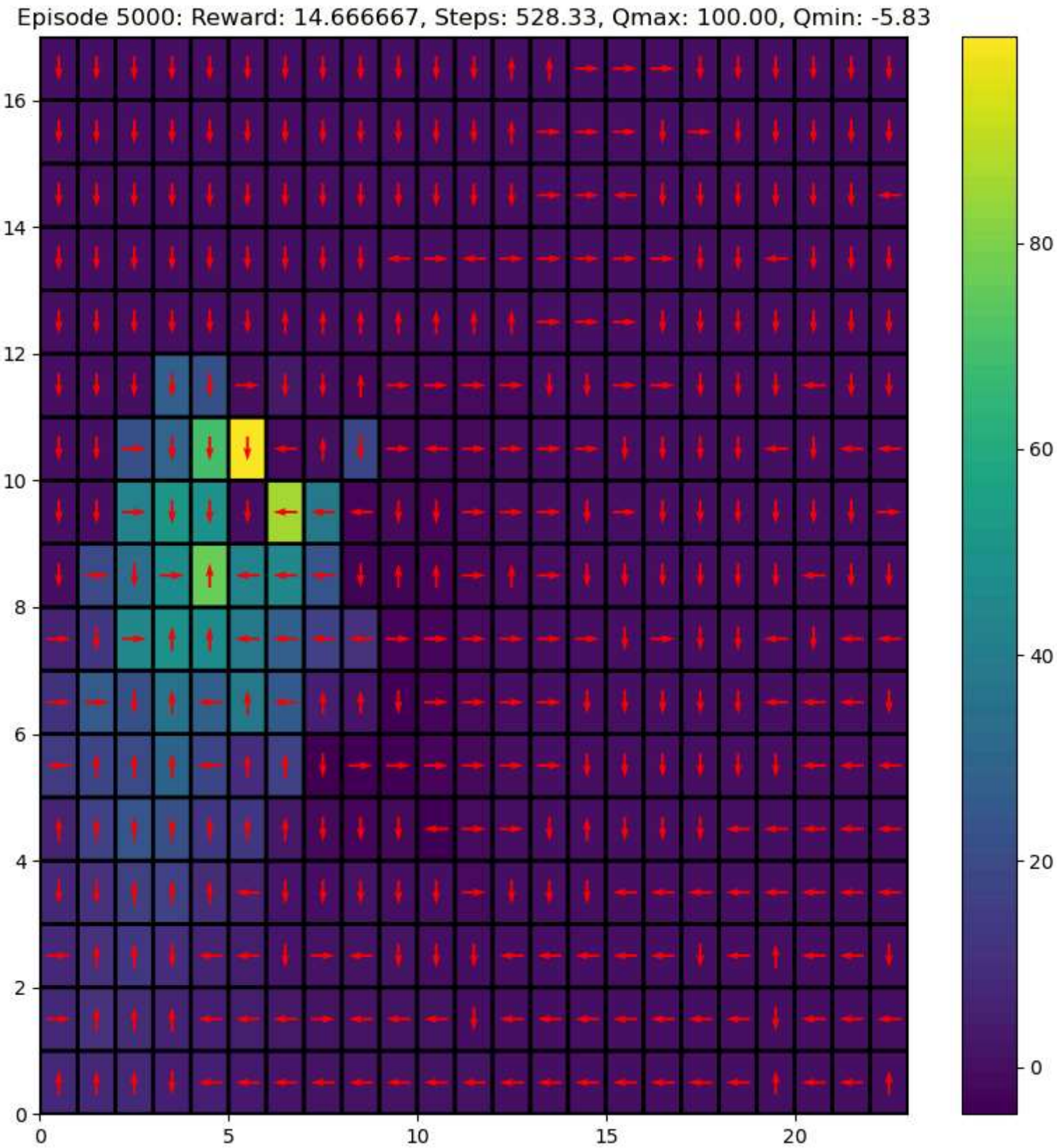
```
Experiment: 1

100%|██████████| 5000/5000 [01:06<00:00, 75.49it/s]

Experiment: 2

100%|██████████| 5000/5000 [01:06<00:00, 75.29it/s]

Experiment: 3

100%|██████████| 5000/5000 [01:07<00:00, 74.54it/s]
```




SARSA: Total Reward
Total Reward
Episode
30
20
10
0
−10
−20
−30
0
1000
2000
3000
4000
5000

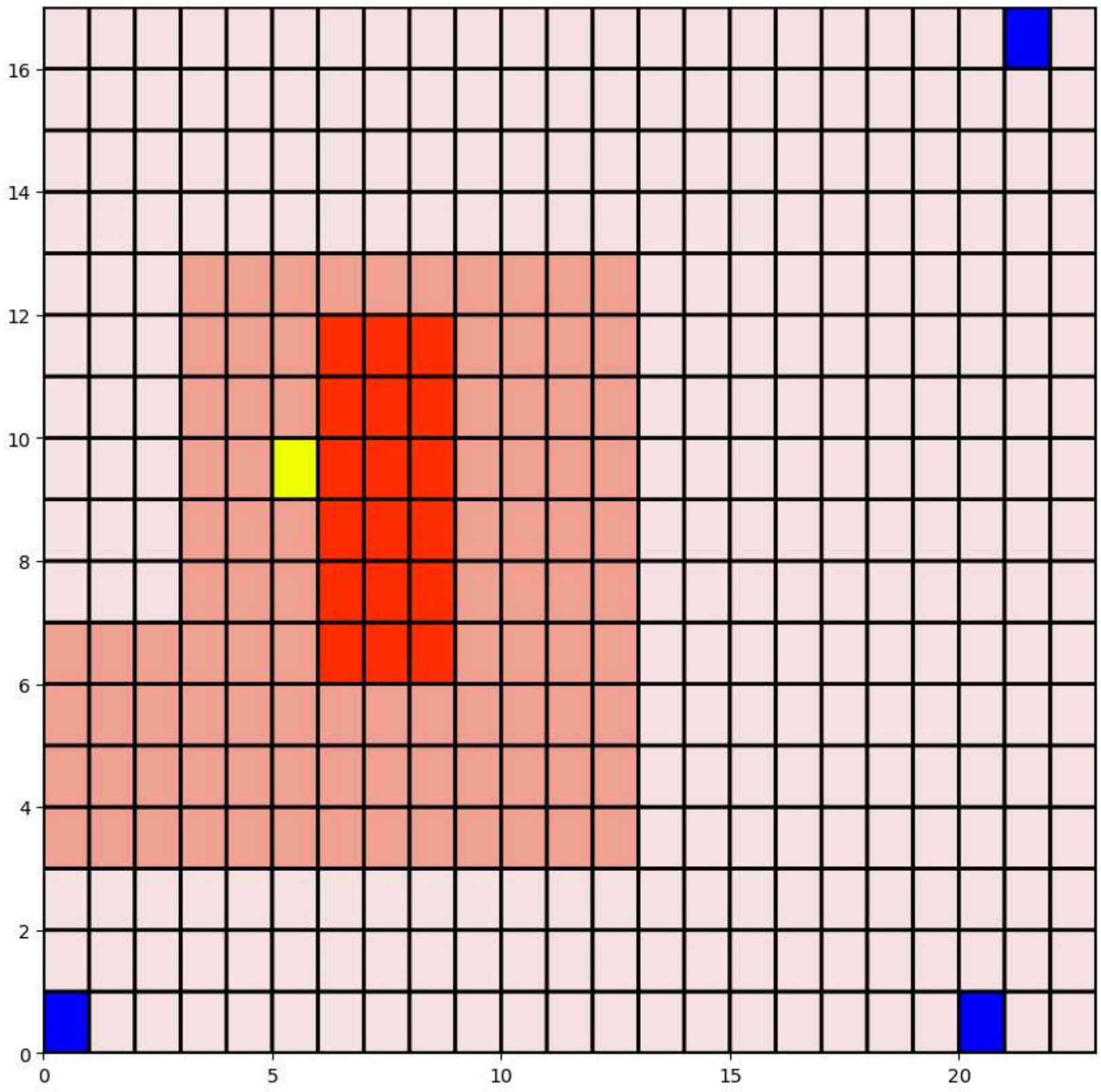
In [66]:

```
### WRITE CODE HERE ### (repeat experiments - Q-Learning)
print_freq = 10

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

    state = env.reset()
    done = False
    while not done:
      action = choose_action(Q, state) #- > softmax
      #action = choose_action(Q, state, epsilon) # -> Epsilon greedy
      state_next, reward, done = env.step(action)

      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 = state_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= choose_action_softmax)
```



100%|██████████| 5000/5000 [01:56<00:00, 42.94it/s]

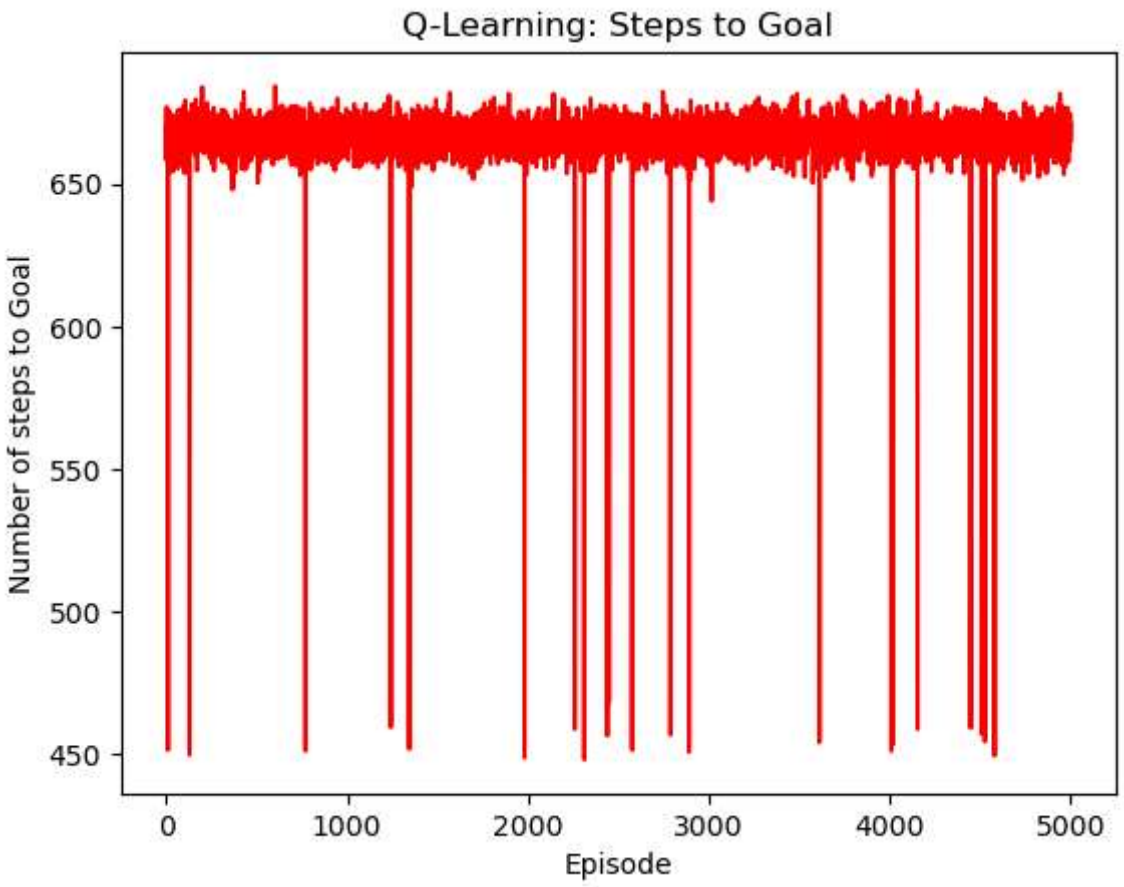
In [69]:
```
# Visualizing the policy learnt by QLearning
from time import sleep

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))
```

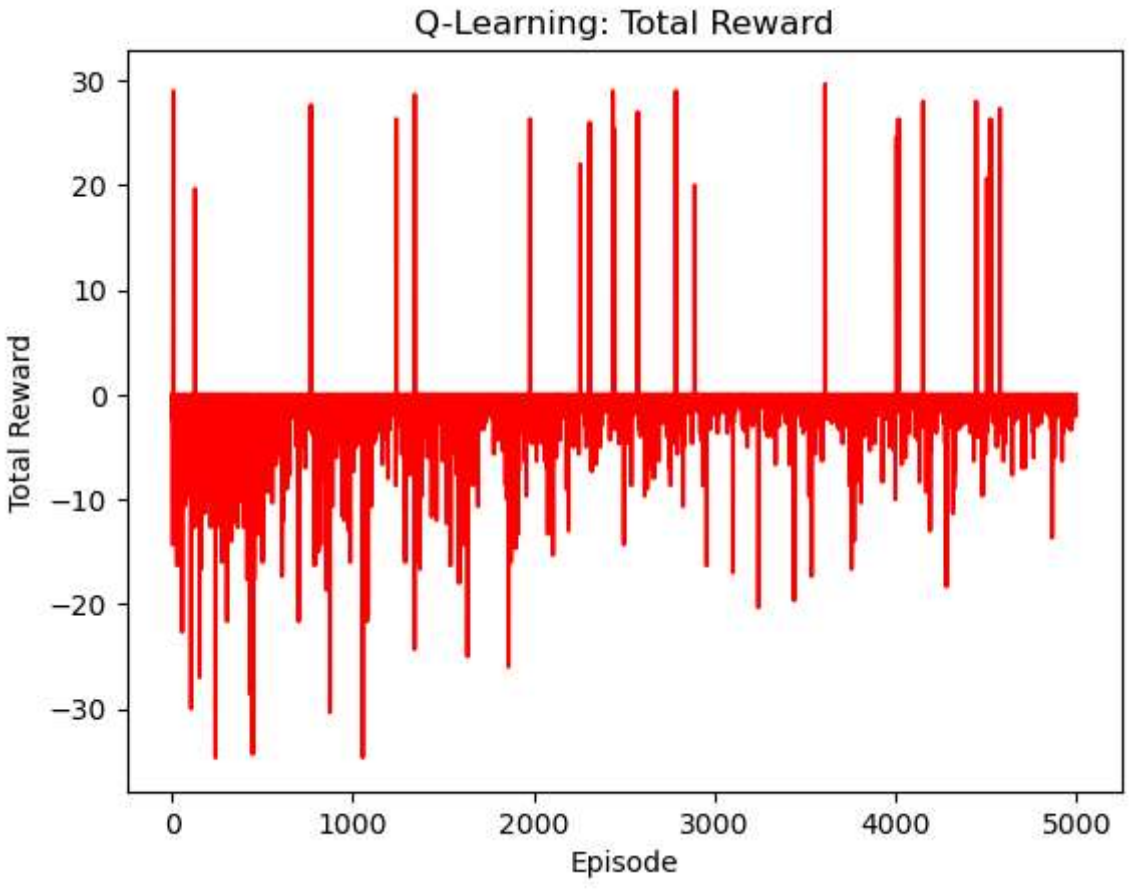



```
Steps: 215, Total Reward: 88
```

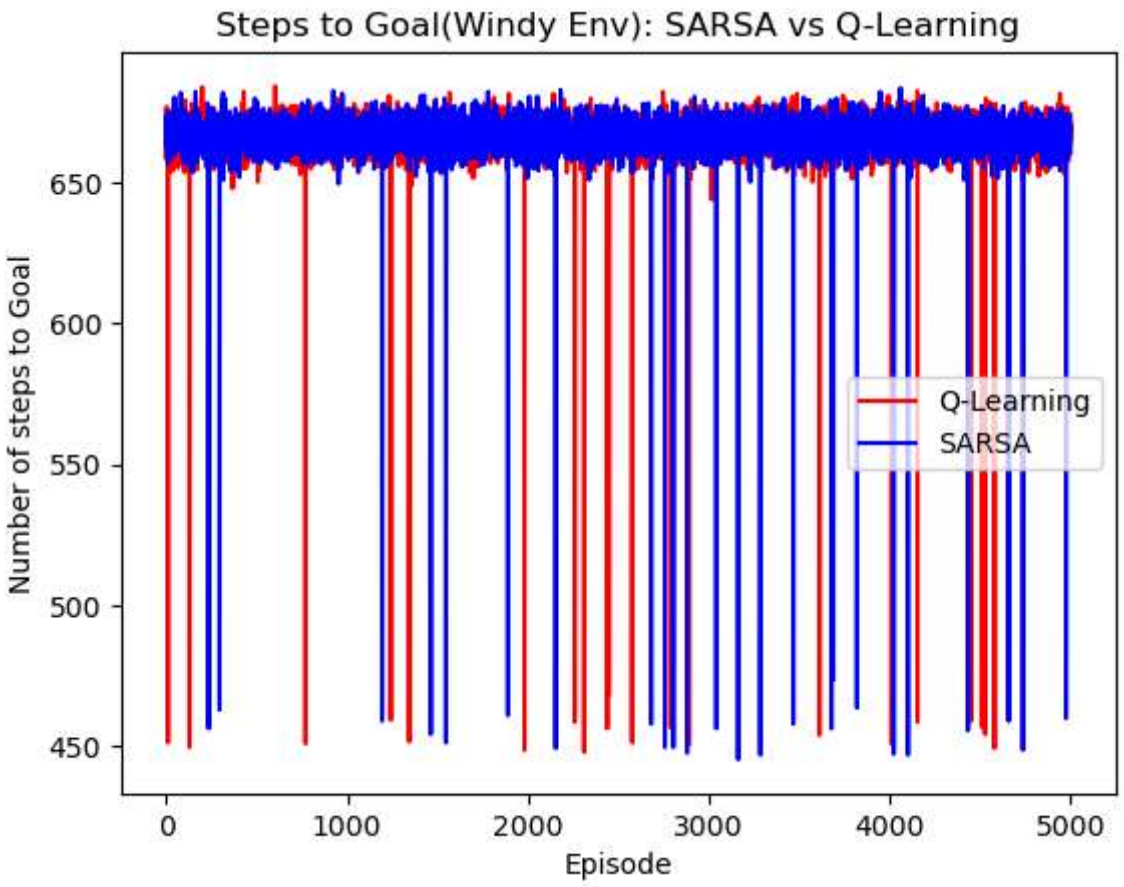
In [59]:
```
# Analyzing performance of the policy learnt by QLearning
Q_avgs_Qlearning_windy, reward_avgs_Qlearning_windy, steps_avgs_Qlearning_windy = [], [], []
num_expts = 3

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_Qlearning_windy.append(Q.copy())
    reward_avgs_Qlearning_windy.append(rewards)
    steps_avgs_Qlearning_windy.append(steps)

plt.xlabel('Episode')
plt.ylabel('Number of steps to Goal')
plt.title('Q-Learning: Steps to Goal')
plt.plot(np.arange(episodes), np.average(steps_avgs_Qlearning_windy, 0), color='red')
plt.show()

plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.title('Q-Learning: Total Reward')
plt.plot(np.arange(episodes), np.average(reward_avgs_Qlearning_windy, 0), color='red')
plt.show()
```

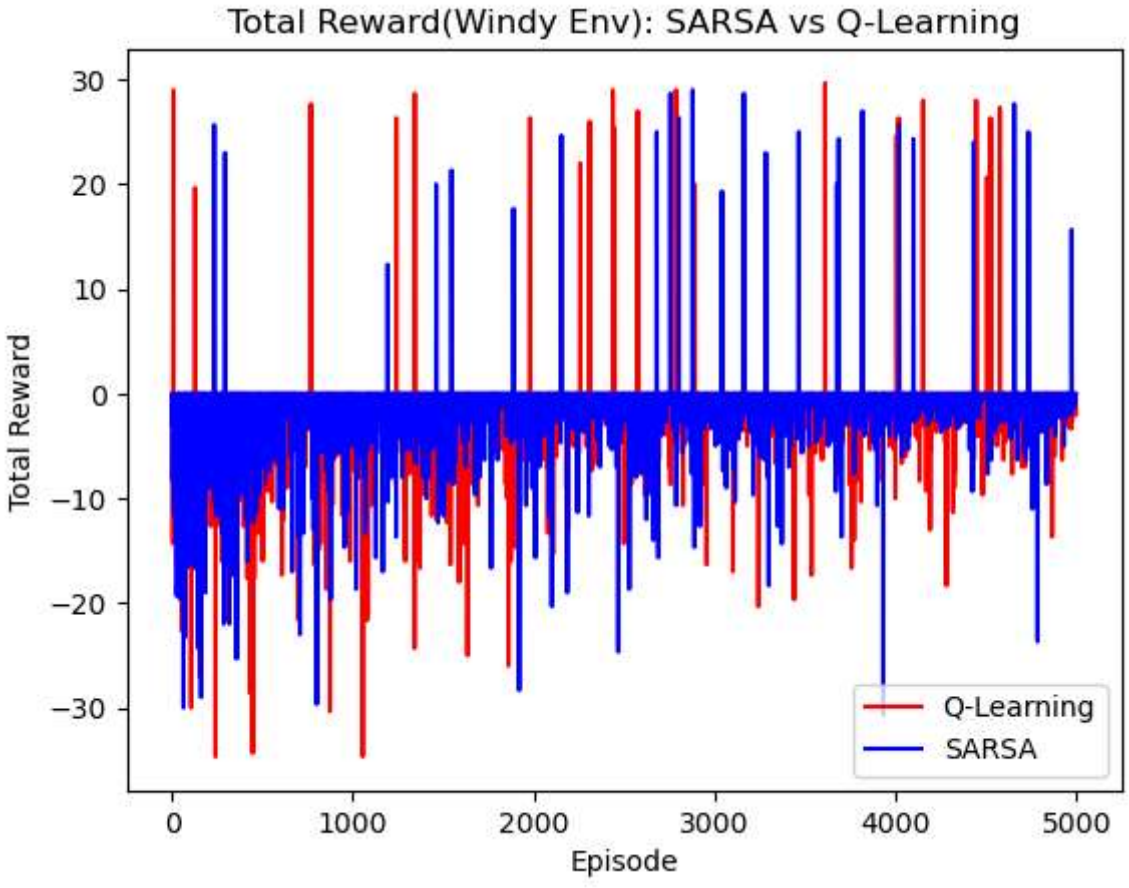
```
Experiment: 1

100%|██████████| 5000/5000 [01:05<00:00, 75.99it/s]

Experiment: 2

100%|██████████| 5000/5000 [01:08<00:00, 72.74it/s]

Experiment: 3

100%|██████████| 5000/5000 [01:06<00:00, 75.26it/s]
```




Q-Learning: Total Reward
Total Reward
Episode
30
20
10
0
−10
−20
−30
0
1000
2000
3000
4000
5000

In [60]:
```
### WRITE CODE HERE ### (compare Q-Learning and SARSA performance)

plt.xlabel('Episode')
plt.ylabel('Number of steps to Goal')
plt.title('Steps to Goal(Windy Env): SARSA vs Q-Learning')
plt.plot(np.arange(episodes), np.average(steps_avgs_Qlearning_windy, 0), color='red', label='Q-Learning')
plt.plot(np.arange(episodes), np.average(steps_avgs_windy, 0), color='blue', label='SARSA')
plt.legend()
plt.show()


plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.title('Total Reward(Windy Env): SARSA vs Q-Learning')
plt.plot(np.arange(episodes), np.average(reward_avgs_Qlearning_windy, 0), color='red', label='Q-Learning')
plt.plot(np.arange(episodes), np.average(reward_avgs_windy, 0), color='blue', label='SARSA')
plt.legend()
plt.show()

```





For the Windy Environment:

1. Sarsa Algorithm doesn't learn a policy to reach the goal state as it tend to optimze the policy according to the behaviour policy(softmax/ epsilon greedy).
2. Q learning algorithm learns a policy to reach the goal state as it always updates Q-values using the action that maximizes future rewards, regardless of the behavior policy. Due to this Off policy nature the QLearning algo is able to find a path to the goal state with considerable rewards value in presense of the Windy nature.

The Average Reward of Q learning is Higher than the SARSA algorithm. The Q learning algorithm has learnt a policy to reach the goal state from all the three start positions with considerable reward, whereas the agent following the SARSA policy gets stuck at the right most cells due the windy nature of the environment and poor policy learnt.

```
1
```