In [1]:
```
from math import floor
import numpy as np

def row_col_to_seq(row_col, num_cols):  #Converts state number to row_column format
    return row_col[:,0] * num_cols + row_col[:,1]

def seq_to_col_row(seq, num_cols): #Converts row_column format to state number
    r = floor(seq / num_cols)
    c = seq - r * num_cols
    return np.array([[r, c]])
class GridWorld:
    """
    Creates a gridworld object to pass to an RL algorithm.
    Parameters
    ----------
    num_rows : int
        The number of rows in the gridworld.
    num_cols : int
        The number of cols in the gridworld.
    start_state : numpy array of shape (1, 2), np.array([[row, col]])
        The start state of the gridworld (can only be one start state)
    goal_states : numpy arrany of shape (n, 2)
        The goal states for the gridworld where n is the number of goal
        states.
    """
    def __init__(self, num_rows, num_cols, start_state, goal_states, wind = False):
        self.num_rows = num_rows
        self.num_cols = num_cols
        self.start_state = start_state
        self.goal_states = goal_states
        self.obs_states = None
        self.bad_states = None
        self.num_bad_states = 0
        self.p_good_trans = None
        self.bias = None
        self.r_step = None
        self.r_goal = None
        self.r_dead = None
        self.gamma = 1 # default is no discounting
        self.wind = wind

    def add_obstructions(self, obstructed_states=None, bad_states=None, restart_states=None):

        self.obs_states = obstructed_states
        self.bad_states = bad_states
        if bad_states is not None:
            self.num_bad_states = bad_states.shape[0]
        else:
            self.num_bad_states = 0
        self.restart_states = restart_states
        if restart_states is not None:
            self.num_restart_states = restart_states.shape[0]
        else:
            self.num_restart_states = 0

    def add_transition_probability(self, p_good_transition, bias):

        self.p_good_trans = p_good_transition
        self.bias = bias

    def add_rewards(self, step_reward, goal_reward, bad_state_reward=None, restart_state_reward = Non

        self.r_step = step_reward
        self.r_goal = goal_reward
        self.r_bad = bad_state_reward
        self.r_restart = restart_state_reward


    def create_gridworld(self):

        self.num_actions = 4
        self.num_states = self.num_cols * self.num_rows# +1
        self.start_state_seq = row_col_to_seq(self.start_state, self.num_cols)
        self.goal_states_seq = row_col_to_seq(self.goal_states, self.num_cols)

        # rewards structure
```

```
        self.R = self.r_step * np.ones((self.num_states, 1))
        #self.R[self.num_states-1] = 0
        self.R[self.goal_states_seq] = self.r_goal

        for i in range(self.num_bad_states):
            if self.r_bad is None:
                raise Exception("Bad state specified but no reward is given")
            bad_state = row_col_to_seq(self.bad_states[i,:].reshape(1,-1), self.num_cols)
            #print("bad states", bad_state)
            self.R[bad_state, :] = self.r_bad
        for i in range(self.num_restart_states):
            if self.r_restart is None:
                raise Exception("Restart state specified but no reward is given")
            restart_state = row_col_to_seq(self.restart_states[i,:].reshape(1,-1), self.num_cols)
            #print("restart_state", restart_state)
            self.R[restart_state, :] = self.r_restart

        # probability model
        if self.p_good_trans == None:
            raise Exception("Must assign probability and bias terms via the add_transition_probabilit

        self.P = np.zeros((self.num_states,self.num_states,self.num_actions))
        for action in range(self.num_actions):
            for state in range(self.num_states):


                # check if the state is the goal state or an obstructed state - transition to end
                row_col = seq_to_col_row(state, self.num_cols)
                if self.obs_states is not None:
                    end_states = np.vstack((self.obs_states, self.goal_states))
                else:
                    end_states = self.goal_states

                if any(np.sum(np.abs(end_states-row_col), 1) == 0):
                    self.P[state, state, action] = 1

                # else consider stochastic effects of action
                else:
                    for dir in range(-1,2,1):

                        direction = self._get_direction(action, dir)
                        next_state = self._get_state(state, direction)
                        if dir == 0:
                            prob = self.p_good_trans
                        elif dir == -1:
                            prob = (1 - self.p_good_trans)*(self.bias)
                        elif dir == 1:
                            prob = (1 - self.p_good_trans)*(1-self.bias)

                        self.P[state, next_state, action] += prob

                # make restart states transition back to the start state with
                # probability 1
                if self.restart_states is not None:
                    if any(np.sum(np.abs(self.restart_states-row_col),1)==0):
                        next_state = row_col_to_seq(self.start_state, self.num_cols)
                        self.P[state,:,:] = 0
                        self.P[state,next_state,:] = 1
        return self

    def _get_direction(self, action, direction):

        left = [2,3,1,0]
        right = [3,2,0,1]
        if direction == 0:
            new_direction = action
        elif direction == -1:
            new_direction = left[action]
        elif direction == 1:
            new_direction = right[action]
        else:
            raise Exception("getDir received an unspecified case")
        return new_direction

    def _get_state(self, state, direction):
```

```
        row_change = [-1,1,0,0]
        col_change = [0,0,-1,1]
        row_col = seq_to_col_row(state, self.num_cols)
        row_col[0,0] += row_change[direction]
        row_col[0,1] += col_change[direction]

        # check for invalid states
        if self.obs_states is not None:
            if (np.any(row_col < 0) or
                np.any(row_col[:,0] > self.num_rows-1) or
                np.any(row_col[:,1] > self.num_cols-1) or
                np.any(np.sum(abs(self.obs_states - row_col), 1)==0)):
                next_state = state
            else:
                next_state = row_col_to_seq(row_col, self.num_cols)[0]
        else:
            if (np.any(row_col < 0) or
                np.any(row_col[:,0] > self.num_rows-1) or
                np.any(row_col[:,1] > self.num_cols-1)):
                next_state = state
            else:
                next_state = row_col_to_seq(row_col, self.num_cols)[0]

        return next_state

    def reset(self):
      return int(self.start_state_seq)

    def step(self, state, action):
        p, r = 0, np.random.random()
        for next_state in range(self.num_states):

            p += self.P[state, next_state, action]

            if r <= p:
                break

        if(self.wind and np.random.random() < 0.4):

          arr = self.P[next_state, :, 3]
          next_next = np.where(arr == np.amax(arr))
          next_next = next_next[0][0]
          return next_next, self.R[next_next]
        else:
          return next_state, self.R[next_state]
```

In [2]:

```
# specify world parameters
num_cols = 10
num_rows = 10
obstructions = np.array([[0,7],[1,1],[1,2],[1,3],[1,7],[2,1],[2,3],
                         [2,7],[3,1],[3,3],[3,5],[4,3],[4,5],[4,7],
                         [5,3],[5,7],[5,9],[6,3],[6,9],[7,1],[7,6],
                         [7,7],[7,8],[7,9],[8,1],[8,5],[8,6],[9,1]])
bad_states = np.array([[1,9],[4,2],[4,4],[7,5],[9,9]])
restart_states = np.array([[3,7],[8,2]])
start_state = np.array([[3,6]])
goal_states = np.array([[0,9],[2,2],[8,7]])

# create model
gw = GridWorld(num_rows=num_rows,
               num_cols=num_cols,
               start_state=start_state,
               goal_states=goal_states, wind = False)
gw.add_obstructions(obstructed_states=obstructions,
                    bad_states=bad_states,
                    restart_states=restart_states)
gw.add_rewards(step_reward=-1,
               goal_reward=10,
               bad_state_reward=-6,
               restart_state_reward=-100)
gw.add_transition_probability(p_good_transition=1,
                              bias=0.5)
env = gw.create_gridworld()
```

In [3]:
```
print("Number of actions", env.num_actions) #0 -> UP, 1-> DOWN, 2 -> LEFT, 3-> RIGHT
print("Number of states", env.num_states)
print("start state", env.start_state_seq)
print("goal state(s)", env.goal_states_seq)
```

```
Number of actions 4
Number of states 100
start state [36]
goal state(s) [ 9 22 87]
```

In [4]:
```
# Install relevant libraries
!pip install numpy matplotlib tqdm scipy
```

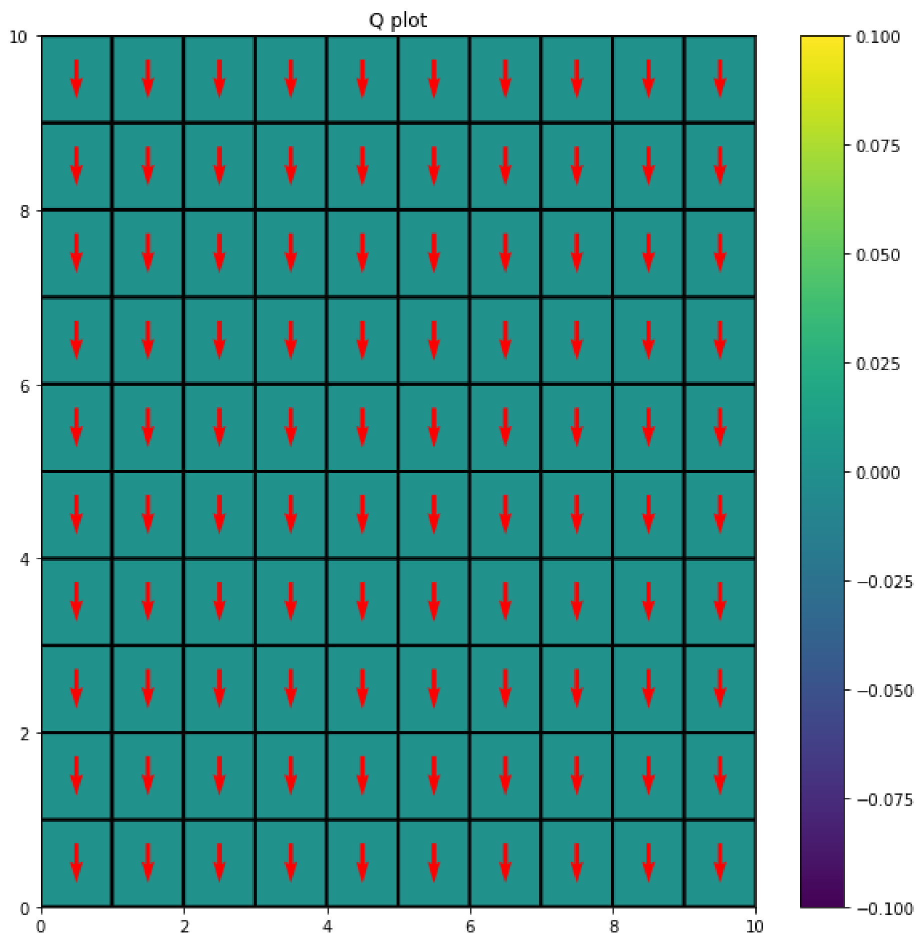
```
Requirement already satisfied: numpy in c:\programdata\anaconda3\lib\site-packages (1.20.1)
Requirement already satisfied: matplotlib in c:\programdata\anaconda3\lib\site-packages (3.3.4)
Requirement already satisfied: tqdm in c:\programdata\anaconda3\lib\site-packages (4.59.0)
Requirement already satisfied: scipy in c:\programdata\anaconda3\lib\site-packages (1.6.2)
Requirement already satisfied: cycler>=0.10 in c:\programdata\anaconda3\lib\site-packages (from matp
lotlib) (0.10.0)
Requirement already satisfied: pillow>=6.2.0 in c:\programdata\anaconda3\lib\site-packages (from mat
plotlib) (8.2.0)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.3 in c:\programdata\anaconda3
\lib\site-packages (from matplotlib) (2.4.7)
Requirement already satisfied: kiwisolver>=1.0.1 in c:\programdata\anaconda3\lib\site-packages (from
matplotlib) (1.3.1)
Requirement already satisfied: python-dateutil>=2.1 in c:\programdata\anaconda3\lib\site-packages (f
rom matplotlib) (2.8.1)
Requirement already satisfied: six in c:\programdata\anaconda3\lib\site-packages (from cycler>=0.10-
>matplotlib) (1.15.0)
```

In [5]:
```
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from IPython.display import clear_output
%matplotlib inline
```

In [6]:
```
#from numpy.random.mtrand import beta
#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].any() or rg.rand() < epsilon:
        return rg.choice(Q.shape[-1])
    else:
        return np.argmax(Q[state])

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

    return rg.choice(Q.shape[-1], p = np.exp((Q[state])/beta)/sum(np.exp((Q[state])/beta)))
```

In [7]:
```
def plot_Q(Q,message ="Q plot"):
    plt.figure(figsize=(10,10))
    plt.title(message)
    plt.pcolor(Q.reshape(10,10,-1).max(-1), edgecolors='k', linewidths=2)
    plt.colorbar()
    def x_direct(a):
        if a in [0, 1]:
            return 0
        return 1 if a == 3 else -1
    def y_direct(a):
        if a in [3, 2]:
            return 0
        return -1 if a == 0 else 1
    policy = Q.reshape(10,10,-1).argmax(-1)
    policyx = np.vectorize(x_direct)(policy)
    policyy = np.vectorize(y_direct)(policy)
    idx = np.indices(policy.shape)
    plt.quiver(idx[1].ravel()+0.5, idx[0].ravel()+0.5, policyx.ravel(), policyy.ravel(), pivot="middl
    plt.show()
```

In [8]:
```
Q = np.zeros((num_cols* num_rows,env.num_actions))

plot_Q(Q)

Q.shape
```

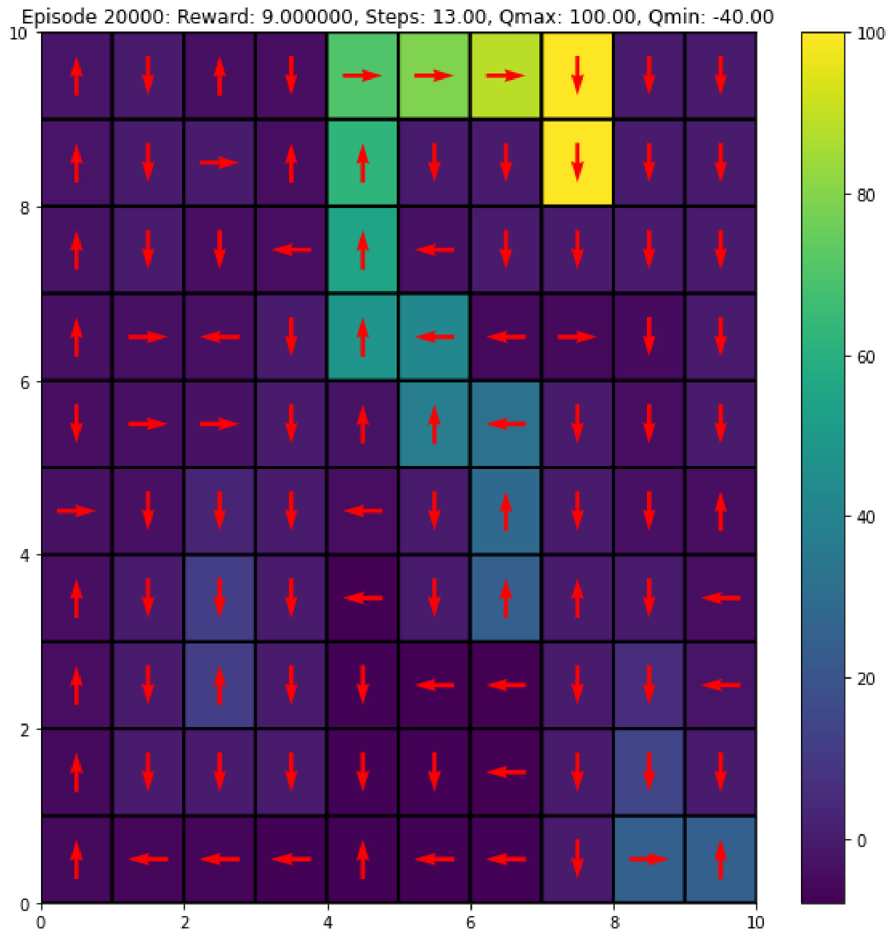


Out[8]:
```
(100, 4)
```

In [9]:
```
Q = np.zeros((num_cols * num_rows,env.num_actions))

alpha0 = 0.4
gamma = 0.9
episodes = 20000
beta = 0.9
```

In [12]:
```
print_freq = 1000

def sarsa(env, Q, gamma = 0.8, 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,beta, state)
        done = False

        while not done:
            state_next, reward = env.step(state,action)
            action_next = choose_action(Q,beta, state_next)
            #print(reward)
            # update equation
            Q[state, action] += alpha*(reward + gamma*Q[state_next, action_next] - Q[state, action])
            tot_reward += reward
            steps += 1
            if state in env.goal_states_seq or steps == 99:
              #reward = gw.goal_reward
              done = True



            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(steps_to_completio
                                                                          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
```

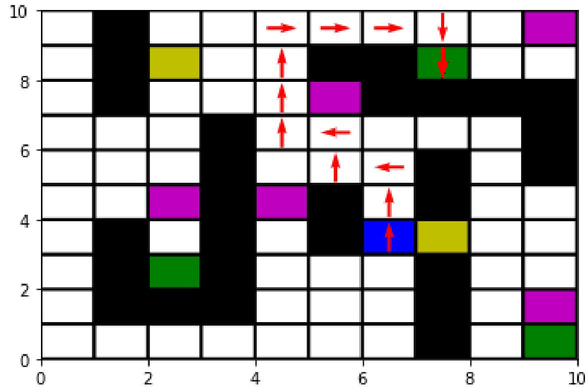



```
100%|██████████| 20000/20000 [01:08<00:00, 290.60it/s]
```

In [14]:
```
def get_best_route(env,Q,):
  state = env.reset()
  done = False
  states = []
  actions = []
  rewards = 0
  steps = 0
  reward = 0
  while not done:
    action = Q[state].argmax()
    states.append(state)
    actions.append(action)

    state , reward = env.step(state,action)
    rewards += rewards
    steps +=1
    if state in env.goal_states or steps == 99:
      return states, actions, rewards
```

In [15]:
```
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap as lcm
def plot_bestpolicy(env,state,action,reward):
        colors = ['w','b','k','m','y','g']
        cmap = lcm(colors)
        a_1 = np.zeros((env.num_rows,env.num_cols))
        a_1[env.start_state[0][0],env.start_state[0][1]]=1
        for j in env.obs_states:
            a_1[j[0],j[1]]=2
        for j in env.bad_states:
            a_1[j[0],j[1]]=3
        for j in env.restart_states:
            a_1[j[0],j[1]]=4
        for j in env.goal_states:
            a_1[j[0],j[1]]=5
        plt.pcolor(a_1, linewidths=2,cmap=cmap, edgecolors='k')

        m,n = [],[]
        for j in state:
            m.append(j%10+0.5)
            n.append(j//10+0.5)

        c_1,c_2 = [],[]
        for j in action:
            if j==0:
                c_1.append(0)
                c_2.append(-1)
            elif j==1:
                c_1.append(0)
                c_2.append(1)
            elif j==2:
                c_1.append(-1)
                c_2.append(0)
            else:
                c_1.append(1)
                c_2.append(0)
        plt.quiver(m,n,c_1,c_2, pivot="middle", color='red')
        plt.show()
```

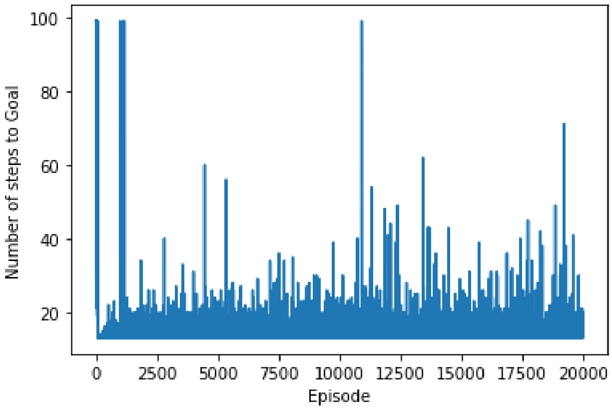
In [16]:
```
a_1,a,r =get_best_route(env,Q)
plot_bestpolicy(env,a_1,a)
```

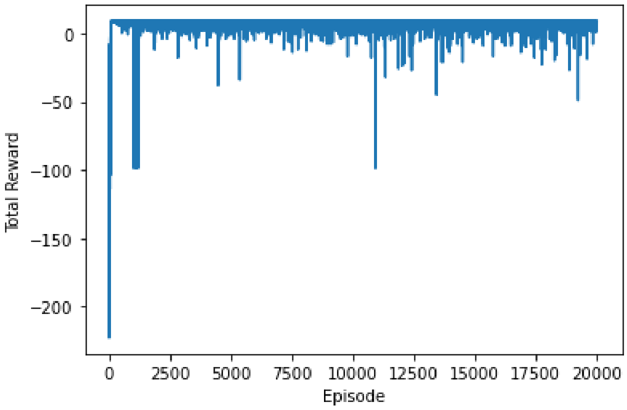


In [17]:
```
Q_avgs, reward_avgs, steps_avgs = [], [], []
num_expts = 3

for i in range(num_expts):
    print("Experiment: %d"%(i+1))
    Q = np.zeros((num_cols * num_rows,env.num_actions))
    rg = np.random.RandomState(i)
    Q, rewards, steps = sarsa(env, Q)
    Q_avgs.append(Q.copy())
    reward_avgs.append(rewards)
    steps_avgs.append(steps)
```

```
  0%|          | 5/20000 [00:00<07:11, 46.34it/s]

Experiment: 1

100%|██████████| 20000/20000 [01:01<00:00, 323.57it/s]
  0%|          | 7/20000 [00:00<05:21, 62.20it/s]

Experiment: 2

100%|██████████| 20000/20000 [01:00<00:00, 328.21it/s]
  0%|          | 6/20000 [00:00<06:15, 53.27it/s]

Experiment: 3

100%|██████████| 20000/20000 [01:01<00:00, 324.51it/s]
```

In [18]:
```
plt.xlabel('Episode')
plt.ylabel('Number of steps to Goal')
plt.plot(np.arange(episodes),steps)
plt.show()
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.plot(np.arange(episodes),rewards)
plt.show()
```

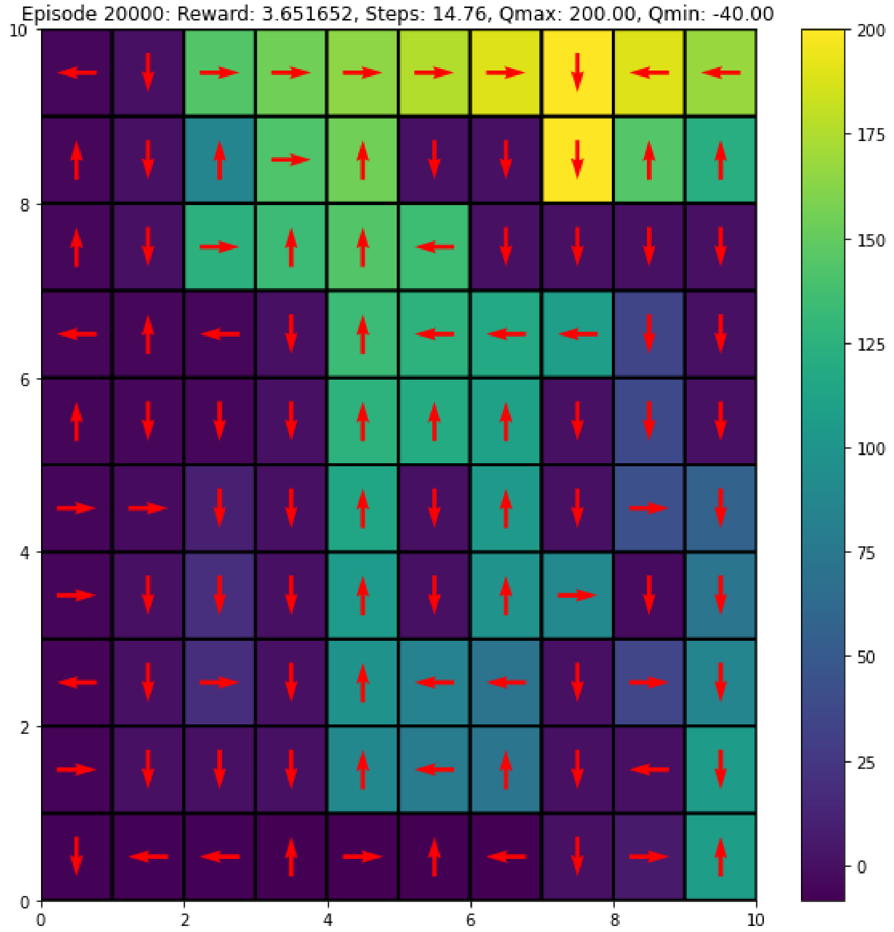




In [19]:
```
Q = np.zeros((num_cols * num_rows,env.num_actions))

alpha0 = 0.4
gamma = 0.95
episodes = 20000
epsilon0 = 0.1
```

In [20]:
```
print_freq = 1000

def sarsa(env, Q, gamma = 0.95, plot_heat = False, choose_action = choose_action_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,epsilon0)
        done = False

        while not done:
            state_next, reward = env.step(state,action)
            action_next = choose_action(Q, state_next,epsilon0)
            #print(reward)
            # update equation
            Q[state, action] += alpha*(reward + gamma*Q[state_next, action_next] - Q[state, action])
            tot_reward += reward
            steps += 1
            if state in env.goal_states_seq or steps == 99:
              #reward = gw.goal_reward
              done = True



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

    return Q, episode_rewards, steps_to_completion
```

In [21]:
```
Q, rewards, steps = sarsa(env, Q, gamma = gamma, plot_heat=True, choose_action= choose_action_epsilon
```

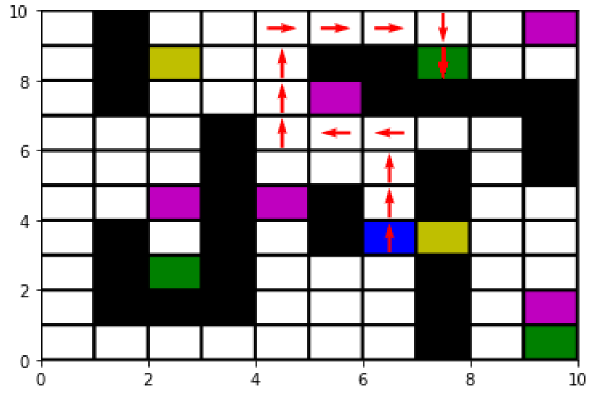



```
100%|██████████| 20000/20000 [00:57<00:00, 348.62it/s]
```

In [22]:
```
def get_best_route(env,Q,):
  state = env.reset()
  done = False
  states = []
  actions = []
  rewards = 0
  steps = 0
  reward = 0
  while not done:
    action = Q[state].argmax()
    states.append(state)
    actions.append(action)

    state , reward = env.step(state,action)
    rewards += rewards
    steps +=1
    if state in env.goal_states or steps == 99:
      return states, actions, rewards
```

In [23]:
```
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap as lcm
def plot_bestpolicy(env,state,action,reward):
        colors = ['w','b','k','m','y','g']
        cmap = lcm(colors)
        a_1 = np.zeros((env.num_rows,env.num_cols))
        a_1[env.start_state[0][0],env.start_state[0][1]]=1
        for j in env.obs_states:
            a_1[j[0],j[1]]=2
        for j in env.bad_states:
            a_1[j[0],j[1]]=3
        for j in env.restart_states:
            a_1[j[0],j[1]]=4
        for j in env.goal_states:
            a_1[j[0],j[1]]=5
        plt.pcolor(a_1, linewidths=2,cmap=cmap, edgecolors='k')

        m,n = [],[]
        for j in state:
            m.append(j%10+0.5)
            n.append(j//10+0.5)

        c_1,c_2 = [],[]
        for j in action:
            if j==0:
                c_1.append(0)
                c_2.append(-1)
            elif j==1:
                c_1.append(0)
                c_2.append(1)
            elif j==2:
                c_1.append(-1)
                c_2.append(0)
            else:
                c_1.append(1)
                c_2.append(0)
        plt.quiver(m,n,c_1,c_2, pivot="middle", color='red')
        plt.show()
```

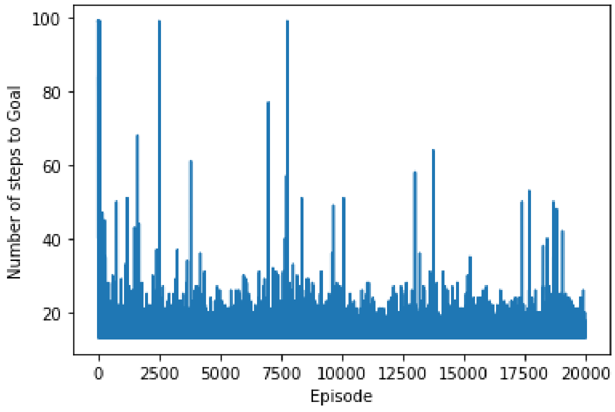
In [24]:
```
a_1,a,r =get_best_route(env,Q)
plot_bestpolicy(env,a_1,a,r)
```

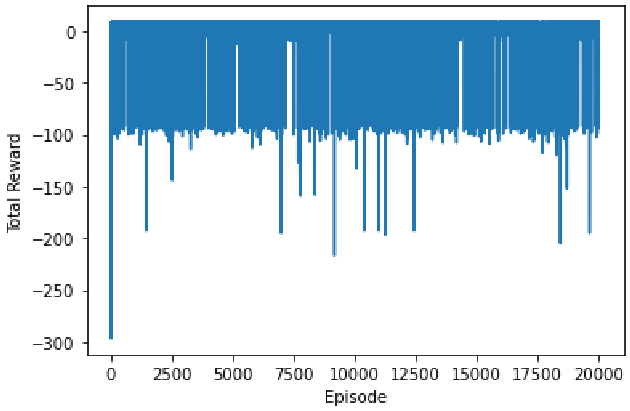


In [25]:
```
Q_avgs, reward_avgs, steps_avgs = [], [], []
num_expts = 3

for i in range(num_expts):
    print("Experiment: %d"%(i+1))
    Q = np.zeros((num_cols * num_rows,env.num_actions))
    rg = np.random.RandomState(i)
    Q, rewards, steps = sarsa(env, Q)
    Q_avgs.append(Q.copy())
    reward_avgs.append(rewards)
    steps_avgs.append(steps)
```

```
  0%|          | 20/20000 [00:00<03:26, 96.97it/s]

Experiment: 1

100%|██████████| 20000/20000 [00:45<00:00, 436.46it/s]
  0%|          | 11/20000 [00:00<03:27, 96.54it/s]

Experiment: 2

100%|██████████| 20000/20000 [00:45<00:00, 436.43it/s]
  0%|          | 10/20000 [00:00<03:30, 94.87it/s]

Experiment: 3

100%|██████████| 20000/20000 [00:45<00:00, 440.85it/s]
```

In [26]:
```
plt.xlabel('Episode')
plt.ylabel('Number of steps to Goal')
plt.plot(np.arange(episodes),steps)
plt.show()
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.plot(np.arange(episodes),rewards)
plt.show()
```

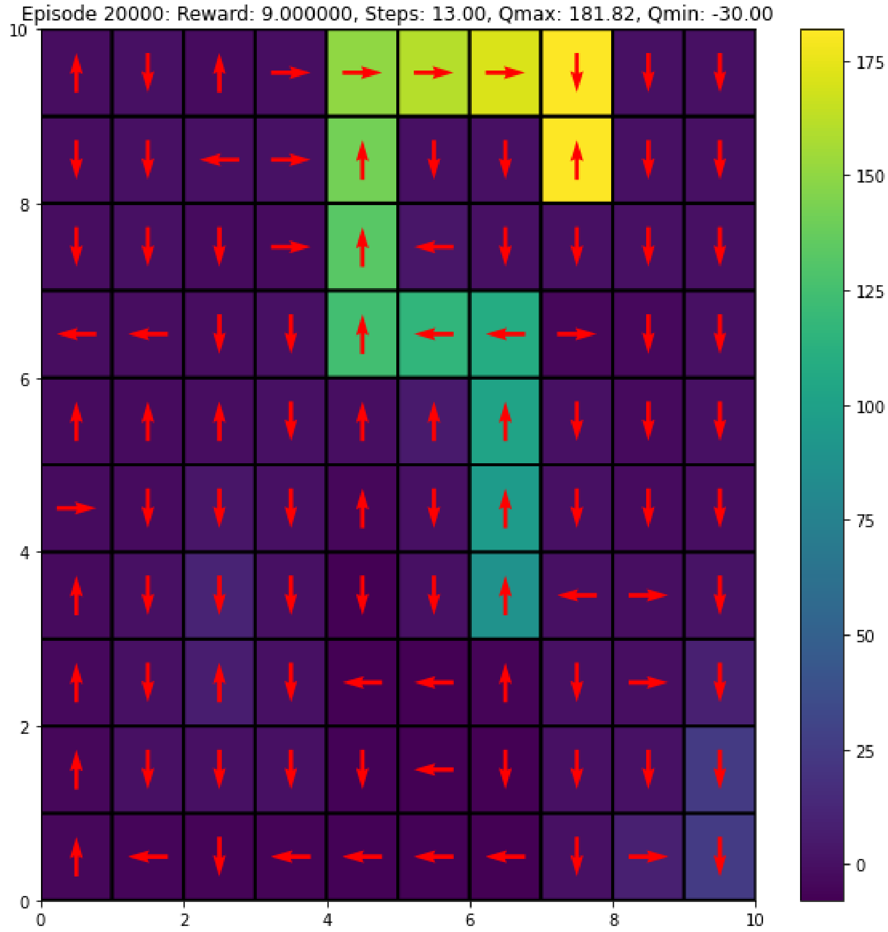




In [27]:
```
Q = np.zeros((num_cols * num_rows,env.num_actions))

alpha0 = 0.3
gamma = 0.945
episodes = 20000
beta = 2.5
```

In [28]:
```
print_freq = 1000

def q_learn(env, Q, gamma = 0.945, 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)
        alpha = alpha0
    for ep in tqdm(range(episodes)):
        tot_reward, steps = 0, 0
        #alpha=(1/(ep+1))
        # Reset environment
        state = env.reset()
       # beta = np.maximum(5 - (5*ep /40000), 0)
        done = False
        while not done:
            action = choose_action(Q,beta, state)

            state_next, reward = env.step(state,action)
            action_next = np.argmax(Q[state_next])
            #action_next = choose_action(Q, state_next, epsilon0)
            # update equation
            maxQ = (Q[state_next,action_next])

            Q[state, action] += alpha0*(reward + gamma * maxQ - Q[state, action])
            tot_reward += reward
            steps += 1
            if state in env.goal_states_seq or steps == 99:
              #reward = gw.goal_reward
              done = True


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

    return Q, episode_rewards, steps_to_completion
```

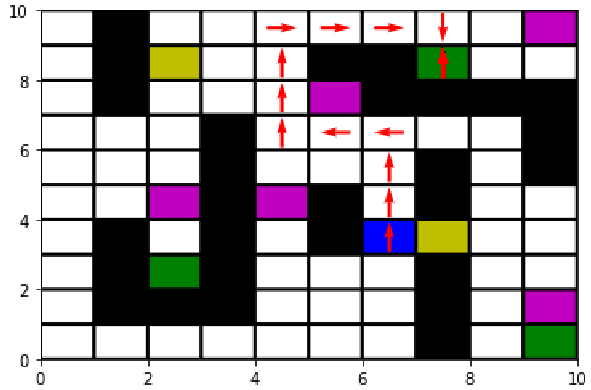
In [29]:
```
Q, rewards, steps = q_learn(env, Q, gamma = gamma, plot_heat=True, choose_action= choose_action_softm
```




```
100%|██████████| 20000/20000 [01:10<00:00, 283.92it/s]
```

In [30]:
```
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap as lcm
def plot_bestpolicy(env,state,action,reward):
        colors = ['w','b','k','m','y','g']
        cmap = lcm(colors)
        a_1 = np.zeros((env.num_rows,env.num_cols))
        a_1[env.start_state[0][0],env.start_state[0][1]]=1
        for j in env.obs_states:
            a_1[j[0],j[1]]=2
        for j in env.bad_states:
            a_1[j[0],j[1]]=3
        for j in env.restart_states:
            a_1[j[0],j[1]]=4
        for j in env.goal_states:
            a_1[j[0],j[1]]=5
        plt.pcolor(a_1, linewidths=2,cmap=cmap, edgecolors='k')

        m,n = [],[]
        for j in state:
            m.append(j%10+0.5)
            n.append(j//10+0.5)

        c_1,c_2 = [],[]
        for j in action:
            if j==0:
                c_1.append(0)
                c_2.append(-1)
            elif j==1:
                c_1.append(0)
                c_2.append(1)
            elif j==2:
                c_1.append(-1)
                c_2.append(0)
            else:
                c_1.append(1)
                c_2.append(0)
        plt.quiver(m,n,c_1,c_2, pivot="middle", color='red')
        plt.show()
```

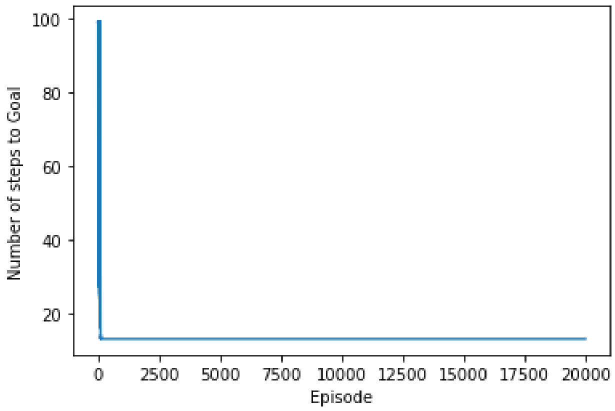
In [31]:
```
a_1,a,r =get_best_route(env,Q)
plot_bestpolicy(env,a_1,a,r)
```

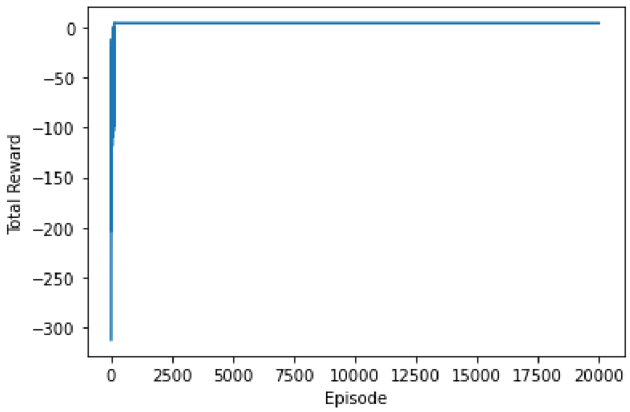


In [32]:
```
Q_avgs, reward_avgs, steps_avgs = [], [], []
num_expts = 3

for i in range(num_expts):
    print("Experiment: %d"%(i+1))
    Q = np.zeros((num_cols * num_rows,env.num_actions))
    rg = np.random.RandomState(i)
    Q, rewards, steps = q_learn(env, Q)
    Q_avgs.append(Q.copy())
    reward_avgs.append(rewards)
    steps_avgs.append(steps)
```

```
  0%|          | 6/20000 [00:00<05:48, 57.38it/s]

Experiment: 1

100%|██████████| 20000/20000 [01:00<00:00, 331.66it/s]
  0%|          | 5/20000 [00:00<07:46, 42.86it/s]

Experiment: 2

100%|██████████| 20000/20000 [00:59<00:00, 333.61it/s]
  0%|          | 5/20000 [00:00<06:40, 49.93it/s]

Experiment: 3

100%|██████████| 20000/20000 [00:50<00:00, 392.68it/s]
```

In [33]:
```
plt.xlabel('Episode')
plt.ylabel('Number of steps to Goal')
plt.plot(np.arange(episodes),steps)
plt.show()
plt.xlabel('Episode')
plt.ylabel('Total Reward')
plt.plot(np.arange(episodes),rewards)
plt.show()
```

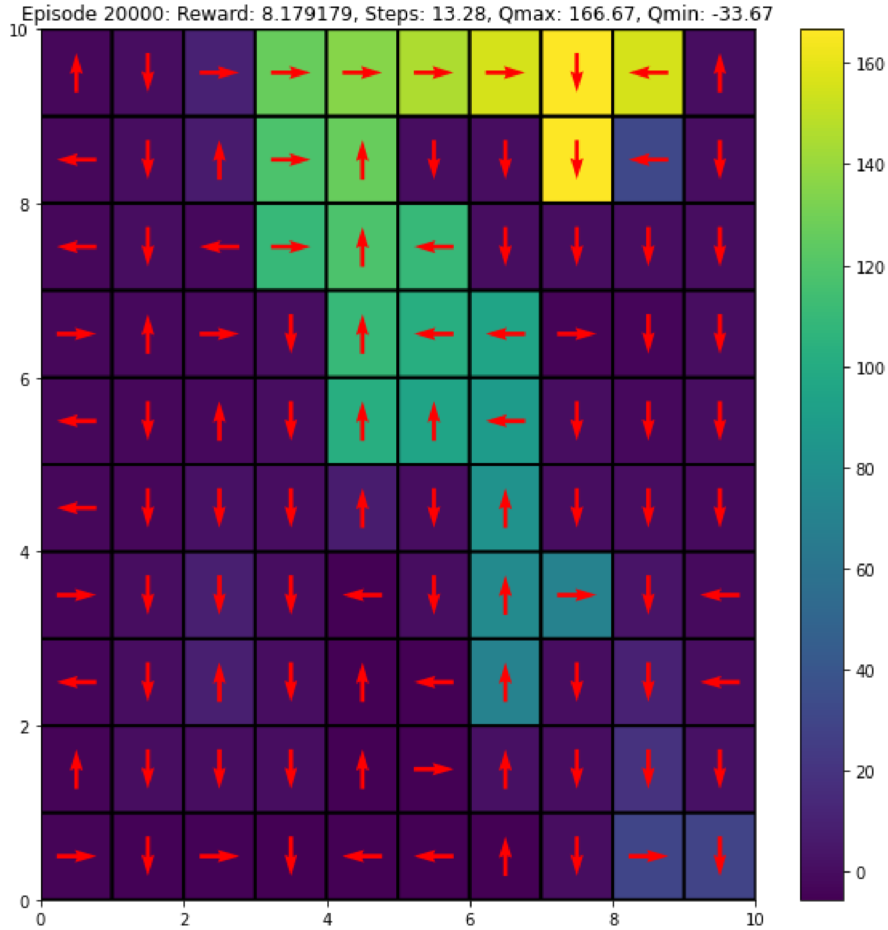




In [34]:
```
Q = np.zeros((num_cols * num_rows,env.num_actions))

alpha0 = 0.1
gamma = 0.94
episodes = 20000
epsilon0 = 0.02
```

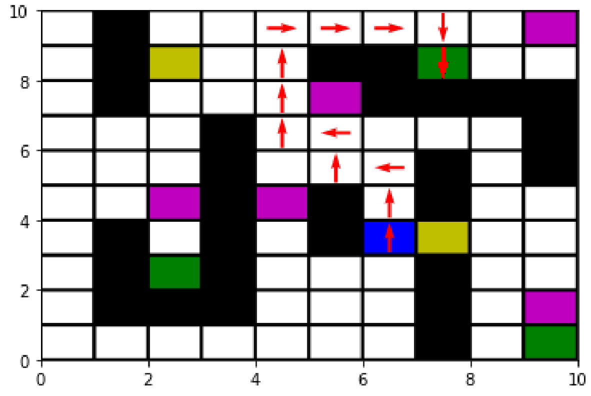
In [35]:
```
print_freq = 1000

def q_learn(env, Q, gamma = 0.94, plot_heat = False, choose_action = choose_action_epsilon):

    episode_rewards = np.zeros(episodes)
    steps_to_completion = np.zeros(episodes)
    if plot_heat:
        clear_output(wait=True)
        plot_Q(Q)
        epsilon = epsilon0

    for ep in tqdm(range(episodes)):
        #alpha = (1/(ep+1))
        alpha = alpha0
        tot_reward, steps = 0, 0
        #epsilon = np.maximum(1-(ep /5000), 0)
        # Reset environment
        state = env.reset()

        done = False
        while not done:
            action = choose_action(Q, state,epsilon0)
            #state_next, reward = env.step(action)
            state_next, reward = env.step(state,action)
            action_next = np.argmax(Q[state_next])

            # update equation
            maxQ = (Q[state_next,action_next])

            Q[state, action] += alpha0*(reward + gamma * maxQ - Q[state, action])
            if state in env.goal_states_seq or steps == 99:
              #reward = gw.goal_reward
              done = True
            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(steps_to_completio
                                                                               Q.max(), Q.min()))

    return Q, episode_rewards, steps_to_completion
```

In [36]:
```
Q, rewards, steps = q_learn(env, Q, gamma = gamma, plot_heat=True, choose_action= choose_action_epsil
```




```
100%|██████████| 20000/20000 [00:52<00:00, 383.97it/s]
```

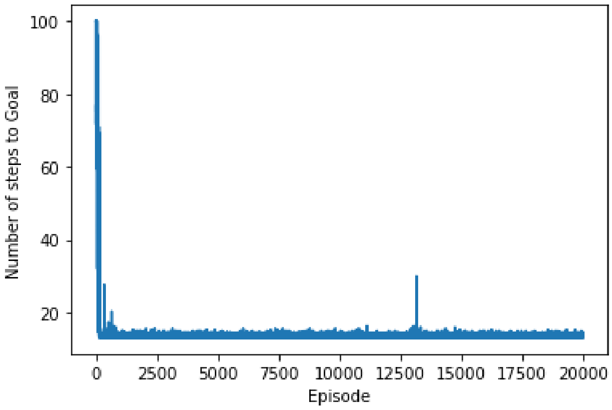
In [37]:
```
a_1,a,r =get_best_route(env,Q)
plot_bestpolicy(env,a_1,a,r)
```

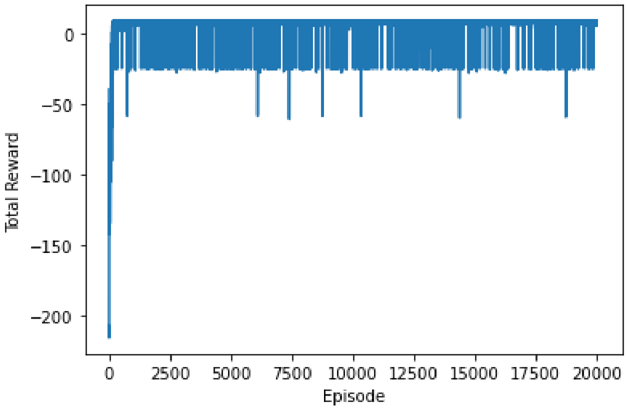


In [38]:
```
Q_avgs, reward_avgs, steps_avgs = [], [], []
num_expts = 3

for i in range(num_expts):
    print("Experiment: %d"%(i+1))
    Q = np.zeros((num_cols * num_rows,env.num_actions))
    rg = np.random.RandomState(i)
    Q, rewards, steps = q_learn(env, Q)
    Q_avgs.append(Q.copy())
    reward_avgs.append(rewards)
    steps_avgs.append(steps)
```

```
  0%|          | 10/20000 [00:00<03:28, 95.86it/s]

Experiment: 1

100%|██████████| 20000/20000 [00:42<00:00, 471.33it/s]
  0%|          | 9/20000 [00:00<03:55, 84.94it/s]

Experiment: 2

100%|██████████| 20000/20000 [00:42<00:00, 474.50it/s]
  0%|          | 10/20000 [00:00<03:30, 95.15it/s]

Experiment: 3

100%|██████████| 20000/20000 [00:41<00:00, 479.97it/s]
```

In [39]:
```
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()
```





In [ ]: