Report Assignment 3

Thi Tuyet Nhi Nguyen

Faculty of Applied Science, Memorial University of Newfoundland

The code is available at: Reinforcement_Learning/Assignment 3 at main · snoween/Reinforcement_Learning

I. Programming part:

To do this assignment, first need to import necessary libraries as below:

```
import numpy as np
import matplotlib.pyplot as plt
```

Also, we need to set up GridWorld for this Part as shown below:

```
class GridWorld:

    def __init__(self):
        # set up parameters
        self.n_row = 5
        self.n_col = 5
        self.n_state = self.n_row * self.n_col
        self.n_action = 4

        # Available actions
        #                      left      down      right     up
        self.action =       [ [0, -1],   [1, 0],   [0, 1],   [-1, 0]]
        self.action_text =  ['\u2190', '\u2193', '\u2192', '\u2191']

        # Gridworld map
        #                  0   1   2   3   4
        self.map = [['T','W','W','W','T'], # 0          map[0][4] = 'T'
                    ['W','W','W','W','W'], # 1
                    ['R','R','W','R','R'], # 2          map[2][3] = 'R'
                    ['W','W','W','W','W'], # 3
                    ['B','W','W','W','W']] # 4          map[4][0] = 'B'

        # Define the initial state "B"
        row_0, col_0 = find_element(self.map, 'B')
        self.state_0 = row_0 * self.n_row + col_0

        self.model = [[[] for _ in range(self.n_action)] for _ in range(self.n_state)]
        for s in range(self.n_state):
            for a in range(self.n_action):
                row, col = np.divmod(s,self.n_row)
                act = self.action[a]  # 0 left, 1 down, 2 right, 3 up
                row_, col_ = row + act[0], col + act[1]
                state_ = row_ * self.n_row + col_
                outsidecheck = (row_ < 0) or (col_ < 0) or (row_ >
                self.n_row - 1) or (col_ > self.n_col - 1)

                # Blue
                if self.map[row][col] == 'B':
```

```
                if outsidecheck:
                    self.model[s][a].append([1.0, s, -1.0, False])
                else:
                    self.model[s][a].append([1.0, state_, -1.0, False])

            # White
            elif self.map[row][col] == 'W':
                if outsidecheck:
                    self.model[s][a].append([1.0, s, -1.0, False])
                elif self.map[row_][col_] == 'R':
                    self.model[s][a].append([1.0, self.state_0,
                    -20.0, False])
                elif self.map[row_][col_] == 'T':
                    self.model[s][a].append([1.0, state_, -1.0, True])
                else:
                    self.model[s][a].append([1.0, state_, -1.0, False])

            # Red
            elif self.map[row][col] == 'R':
                self.model[s][a].append([1.0, s, 0.0, False])

            # Black (Terminal)
            elif self.map[row][col] == 'T':
                self.model[s][a].append([1.0, s, 0.0, True])

            else:
                raise ValueError("Unknown value!")
```

Functions to select and find policy:

```
def argmax_random(input_array):
    return np.random.choice(np.flatnonzero(input_array == np.max(input_array)))

def select_action(q, eps, env):
    if np.random.rand() > eps:  # -greedy
        return argmax_random(q)  # randomely select the largest action
    else:
        return np.random.choice([i for i in range(env.n_action)])

def find_policy(gamma, V, env):
    pol = [[0] for _ in range(env.n_state)]  # initialize a deterministic policy

    for s in range(env.n_state):  # sweep all the states in the state space
        temp1 = []
        for a in range(env.n_action):
            temp2 = 0
            for p, s_, r, _ in env.model[s][a]:
                temp2 += p * (r + gamma * V[s_])
            temp1.append(temp2)

        pol[s] = (np.unique(np.argwhere(temp1 == np.max(temp1)))).tolist()
    return pol
```

A function to perform movement transitions:

```
def print_policy(policy, env):
    policy_visual = ['' for _ in range(env.n_state)]

    for s in range(env.n_state):
        lenth = len(policy[s])
        if lenth == 4:
            policy_visual[s] += 'o'  # 'o' means 4 directions are all available
        else:
            for a in range(lenth):
                policy_visual[s] += env.action_text[policy[s][a]]
    return policy_visual

def plot_trajectory(policy, env):
    trajectory_visual = [' ' for _ in range(env.n_state)]
    s = env.state_0
    for _ in range(100):
        trajectory_visual[s] = env.action_text[policy[s][0]]
        _, s_, _, t = env.model[s][policy[s][0]][0]
        s = s_

        if t:  # if next state is the terminal state
            trajectory_visual[s] = 'x'
            break
    return trajectory_visual
```

The code for Sarsa algorithm:

```
def Sarsa(max_ep, gamma, epsilon, alpha, env):
    # Initialize
    policy_opt = [[0] for _ in range(env.n_state)]
    Q_all = [[0 for _ in range(env.n_action)] for _ in range(env.n_state)]
    Q_opt = [0 for _ in range(env.n_state)]
    reward_trace = [0 for _ in range(max_ep)]

    # for each episode
    for ep in range(max_ep):
        # initialize
        s = env.state_0
        a = select_action(Q_all[s], epsilon, env)
        r_sum = 0

        while 1:
            _, s_, r, t = env.model[s][a][0]  # get new state and reward
            a_ = select_action(Q_all[s_], epsilon, env)
            Q_all[s][a] += alpha * (r + gamma * Q_all[s_][a_] - Q_all[s][a])
            s, a = s_, a_  # update state and action
            r_sum += r  # update reward sum
            if t:  # if next state is the terminal state
                break

        reward_trace[ep] = r_sum  # record reward sum

    for s in range(env.n_state):
        policy_opt[s] = (np.unique(np.argwhere(Q_all[s] == np.max(Q_all[s])))).tolist()

    Q_opt = np.max(Q_all, axis=1)  # find the optimal Q function

    return Q_all, Q_opt, policy_opt, reward_trace
```

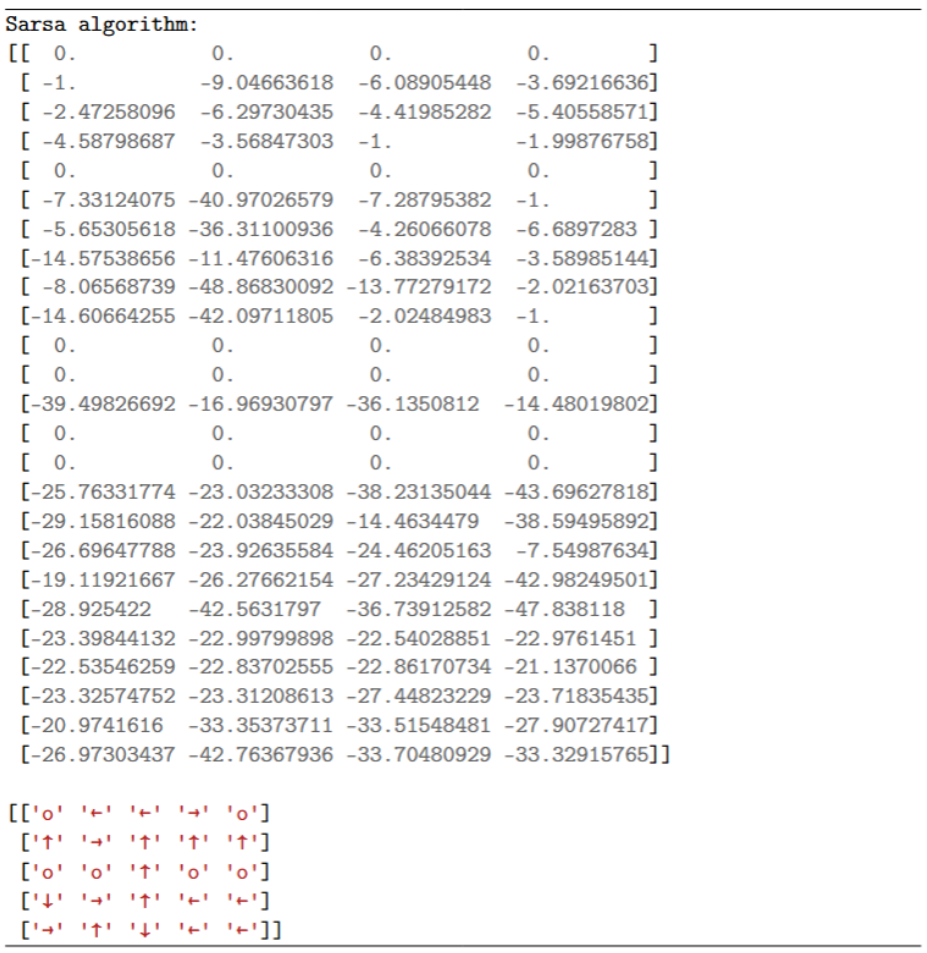
The code for Q-learning algorithm:

```
def Q_learning(max_ep, gamma, epsilon, alpha, env):
    # Initialize
    policy_opt = [[0] for _ in range(env.n_state)]
    Q_all = [[0 for _ in range(env.n_action)] for _ in range(env.n_state)]
    Q_opt = [0 for _ in range(env.n_state)]
    reward_trace = [0 for _ in range(max_ep)]

    # for each episode
    for ep in range(max_ep):
        # initialize
        s = env.state_0
        r_sum = 0

        while 1:
            a = select_action(Q_all[s], epsilon, env)
            _, s_, r, t = env.model[s][a][0]  # get new state and reward
            Q_max = np.max(Q_all[s_])  # find the maximum q value
            Q_all[s][a] += alpha * (r + gamma * Q_max - Q_all[s][a])
            s = s_
            r_sum += r
            if t:
                break

        reward_trace[ep] = r_sum  # record reward sum

    for s in range(env.n_state):
        policy_opt[s] = (np.unique(np.argwhere(Q_all[s] == np.max(Q_all[s])))).tolist()

    Q_opt = np.max(Q_all, axis=1)

    return Q_all, Q_opt, policy_opt, reward_trace
```

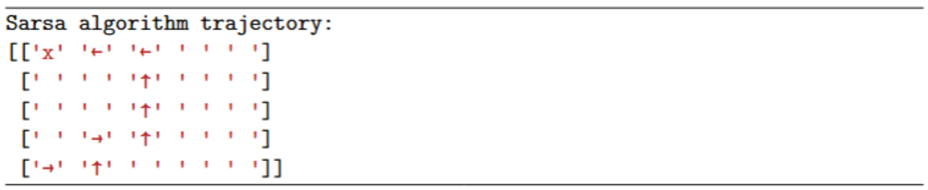
Set up initial parameters:

```
# Initialize the environment, set up parameters
max_ep = 10000
gam = 0.99  # discount factor
eps = 0.2  # epsilon, Algorithm parameter: small  > 0
alp = 0.5  # learning rate
Env = GridWorld()

# Find the optimal policy
Q_1, Q_opt_1, pol_opt_1, reward_plot_1 = Sarsa(max_ep, gam, eps, alp, Env)
Q_2, Q_opt_2, pol_opt_2, reward_plot_2 = Q_learning(max_ep, gam, eps, alp, Env)
```

The results of Sarsa algorithm:

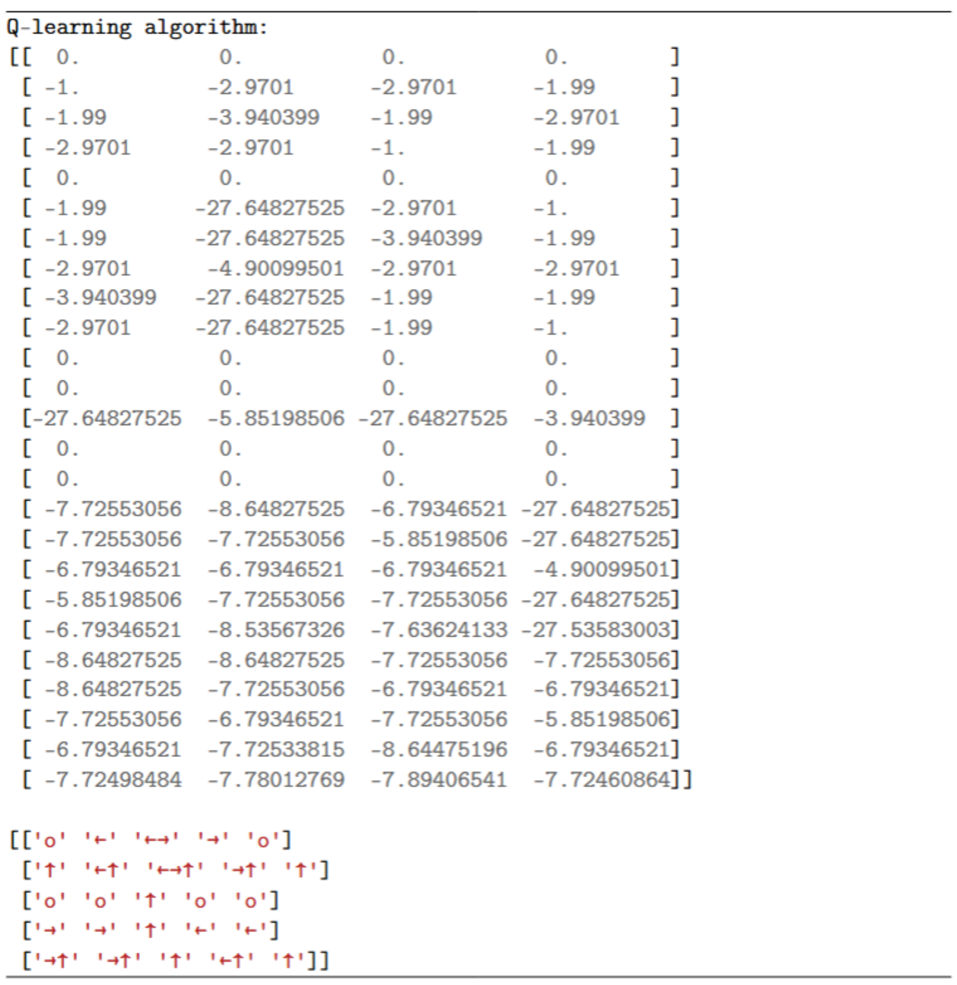


```
Sarsa algorithm:
[[  0.           0.           0.           0.        ]
 [ -1.          -9.04663618  -6.08905448  -3.69216636]
 [ -2.47258096  -6.29730435  -4.41985282  -5.40558571]
 [ -4.58798687  -3.56847303  -1.          -1.99876758]
 [  0.           0.           0.           0.        ]
 [ -7.33124075 -40.97026579  -7.28795382  -1.        ]
 [ -5.65305618 -36.31100936  -4.26066078  -6.6897283 ]
 [-14.57538656 -11.47606316  -6.38392534  -3.58985144]
 [ -8.06568739 -48.86830092 -13.77279172  -2.02163703]
 [-14.60664255 -42.09711805  -2.02484983  -1.        ]
 [  0.           0.           0.           0.        ]
 [  0.           0.           0.           0.        ]
 [-39.49826692 -16.96930797 -36.1350812  -14.48019802]
 [  0.           0.           0.           0.        ]
 [  0.           0.           0.           0.        ]
 [-25.76331774 -23.03233308 -38.23135044 -43.69627818]
 [-29.15816088 -22.03845029 -14.4634479  -38.59495892]
 [-26.69647788 -23.92635584 -24.46205163  -7.54987634]
 [-19.11921667 -26.27662154 -27.23429124 -42.98249501]
 [-28.925422   -42.5631797  -36.73912582 -47.838118  ]
 [-23.39844132 -22.99799898 -22.54028851 -22.9761451 ]
 [-22.53546259 -22.83702555 -22.86170734 -21.1370066 ]
 [-23.32574752 -23.31208613 -27.44823229 -23.71835435]
 [-20.9741616  -33.35373711 -33.51548481 -27.90727417]
 [-26.97303437 -42.76367936 -33.70480929 -33.32915765]]

[['o' '←' '←' '→' 'o']
 ['↑' '→' '↑' '↑' '↑']
 ['o' 'o' '↑' 'o' 'o']
 ['↓' '→' '↑' '←' '←']
 ['→' '↑' '↓' '←' '←']]
```

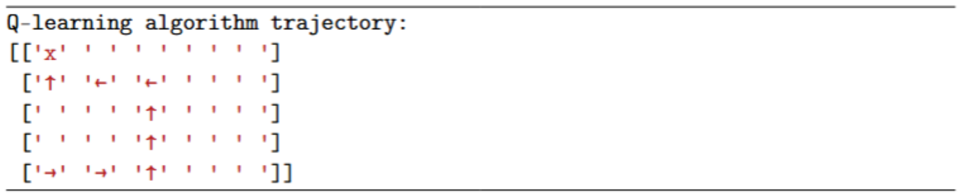


```
Sarsa algorithm trajectory:
[['x' '←' '←' ' ' ' ']
 [' ' ' ' '↑' ' ' ' ']
 [' ' ' ' '↑' ' ' ' ']
 [' ' '→' '↑' ' ' ' ']
 ['→' '↑' ' ' ' ' ' ']]
```

Q-learning algorithm results:

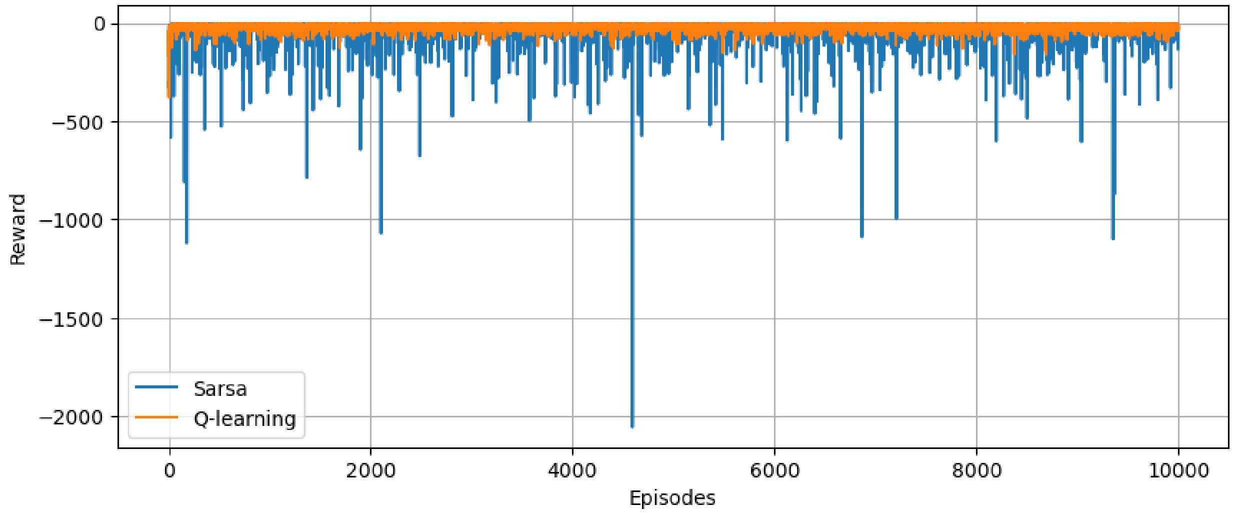


```
Q-learning algorithm:
[[  0.           0.           0.           0.        ]
 [ -1.          -2.9701      -2.9701      -1.99       ]
 [ -1.99        -3.940399    -1.99        -2.9701     ]
 [ -2.9701      -2.9701      -1.          -1.99       ]
 [  0.           0.           0.           0.        ]
 [ -1.99       -27.64827525  -2.9701      -1.        ]
 [ -1.99       -27.64827525  -3.940399    -1.99      ]
 [ -2.9701      -4.90099501  -2.9701      -2.9701     ]
 [ -3.940399   -27.64827525  -1.99        -1.99      ]
 [ -2.9701     -27.64827525  -1.99        -1.        ]
 [  0.           0.           0.           0.        ]
 [  0.           0.           0.           0.        ]
 [-27.64827525  -5.85198506 -27.64827525  -3.940399  ]
 [  0.           0.           0.           0.        ]
 [  0.           0.           0.           0.        ]
 [ -7.72553056  -8.64827525  -6.79346521 -27.64827525]
 [ -7.72553056  -7.72553056  -5.85198506 -27.64827525]
 [ -6.79346521  -6.79346521  -6.79346521  -4.90099501]
 [ -5.85198506  -7.72553056  -7.72553056 -27.64827525]
 [ -6.79346521  -8.53567326  -7.63624133 -27.53583003]
 [ -8.64827525  -8.64827525  -7.72553056  -7.72553056]
 [ -8.64827525  -7.72553056  -6.79346521  -6.79346521]
 [ -7.72553056  -6.79346521  -7.72553056  -5.85198506]
 [ -6.79346521  -7.72533815  -8.64475196  -6.79346521]
 [ -7.72498484  -7.78012769  -7.89406541  -7.72460864]]

[['o' '←' '←→' '→' 'o']
 ['↑' '←↑' '←→↑' '→↑' '↑']
 ['o' 'o' '↑' 'o' 'o']
 ['→' '→' '↑' '←' '←']
 ['→↑' '→↑' '↑' '←↑' '↑']]
```



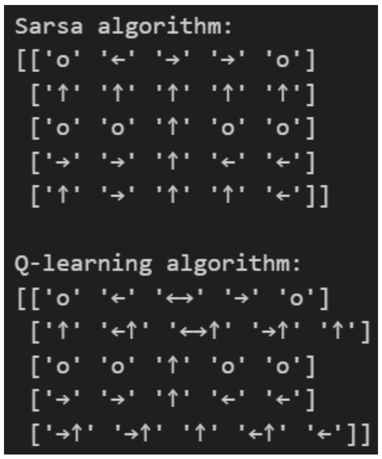
```
Q-learning algorithm trajectory:
[['x' ' ' ' ' ' ' ' ']
 ['↑' '←' '←' ' ' ' ']
 [' ' ' ' '↑' ' ' ' ']
 [' ' ' ' '↑' ' ' ' ']
 ['→' '→' '↑' ' ' ' ']]
```

Plot a trajectory of an agent utilizing the policy learned by each of the methods.

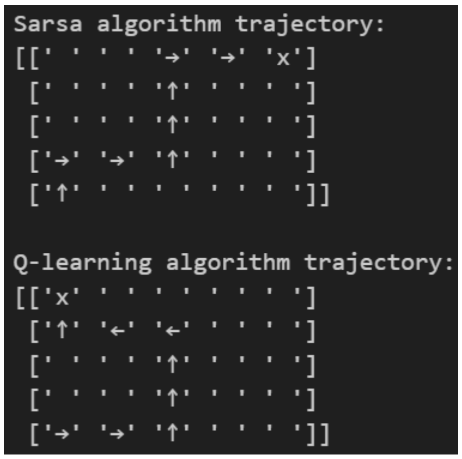


<u>Explain part: Are they different or similar? Why or why not? You may assume to use ϵ-greedy action selection for this task. How does the sum of rewards over an episode behaves for each of these two methods.</u>

With initial parameters are: discount factor $\gamma = 0.99$, epsilon greedy action selection with epsilon = 0.2, learning rate alpha = 0.5, and a policy with equiprobable moves, the results of the two methods are shown in the following figures.



```
Sarsa algorithm:
[['o' '←' '→' '→' 'o']
 ['↑' '↑' '↑' '↑' '↑']
 ['o' 'o' '↑' 'o' 'o']
 ['→' '→' '↑' '←' '←']
 ['↑' '→' '↑' '↑' '←']]

Q-learning algorithm:
[['o' '←' '↔' '→' 'o']
 ['↑' '←↑' '↔↑' '→↑' '↑']
 ['o' 'o' '↑' 'o' 'o']
 ['→' '→' '↑' '←' '←']
 ['→↑' '→↑' '↑' '←↑' '←']]
```



```
Sarsa algorithm trajectory:
[[' ' ' ' '→' '→' 'x']
 [' ' ' ' '↑' ' ' ' ']
 [' ' ' ' '↑' ' ' ' ']
 ['→' '→' '↑' ' ' ' ']
 ['↑' ' ' ' ' ' ' ' ']]

Q-learning algorithm trajectory:
[['x' ' ' ' ' ' ' ' ']
 ['↑' '←' '←' ' ' ' ']
 [' ' ' ' '↑' ' ' ' ']
 [' ' ' ' '↑' ' ' ' ']
 ['→' '→' '↑' ' ' ' ']]
```

Fig. 1: (Left) Optimal policy; (Right) Trajectory under the optimal policy.

According to the optimal derived policy, Q-learning algorithm performs better than the Sarsa algorithm. Q-learning algorithm can almost give all the correct choices at each state (except for the bottom right corner), while Sarsa can only give some of the correct choices. For example, at the starting grid, Q-learning algorithm shows 2 choices: go left and go up, while the Sarsa algorithm only shows 1 choice: go up.

The trajectory of the two algorithms is nearly the same, both avoid the red grids and move to the black grid in the smallest steps. Hence, the similar results are caused by the nature of this 5 by 5 grid world problem. This problem is symmetrical, if the agent does not try to move out, move back towards the starting point, or move to the red grid, any trajectories can be one of the best choices.

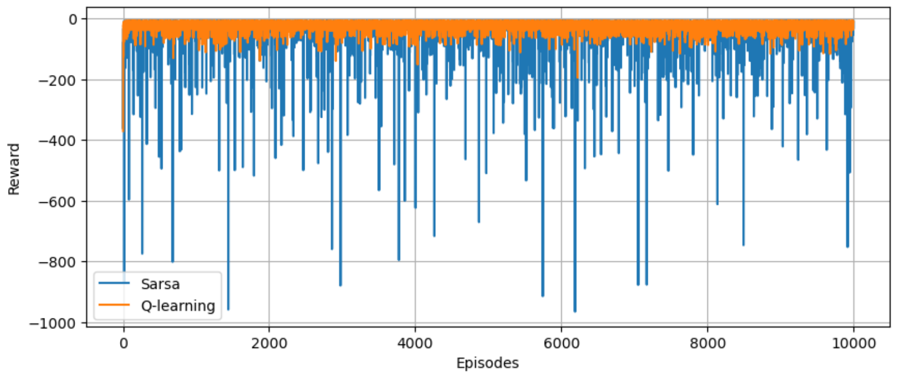


Fig. 2: Sum of rewards.

Fig. 2 shows the sum of rewards over each episode for these two algorithms. It is obvious that the sum of rewards for Q-learning is much higher than that of Sarsa's, which means that Q-learning is less likely to move to the red grid and move to the black grid faster than Sarsa algorithm.