Assignment 3

余北辰 519030910245

1 Experimental Purpose

- Implement Sarsa and Q-learning method separately
- Try different ϵ to investigate their impacts on performances

2 Experimental Background

In this section, we present the methods of Sarsa and Q-learning separately. Some of the content in this section is referenced from Wikipedia.

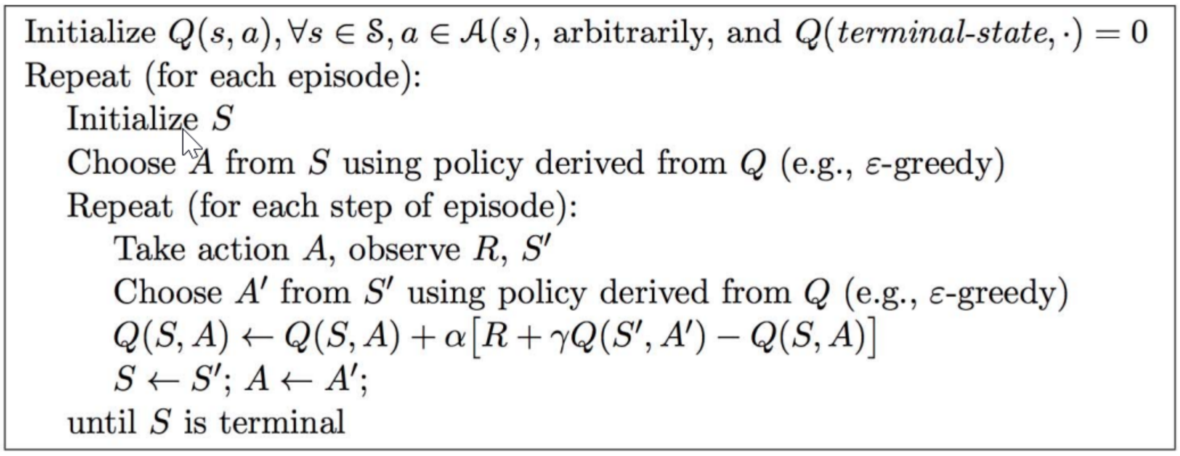
2.1 Sarsa

Sarsa is the abbreviation of "State–action–reward–state–action", which is an on-policy learning algorithm. A Sarsa agent interacts with the environment and updates the policy based on actions taken. Q values represent the possible reward received in the next time step for taking action a in state s, plus the discounted future reward received from the next state-action observation. The update function of Q values is shown as below:

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

The Q value for a state-action is updated by an error, adjusted by the learning rate α.

The pseudo-code for Sarsa is shown below:



```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Repeat (for each step of episode):
        Take action $A$, observe $R$, $S'$
        Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S',A') - Q(S,A)\big]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until $S$ is terminal
```

2.2 Q-learning

Q-learning is another model-free reinforcement learning algorithm to learn the value of an action in a particular state. Different with Sarsa, it is an off-policy algorithm. The only different of Sarsa and Q-learning is how the method updates the Q values. On-policy Sarsa learns action values relative to the policy it follows, while off-policy Q-learning does it relative to the greedy policy.

The update function of Q values is shown as below:

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

The pseudo-code for Q-learning is shown below:

```
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
    Initialize $S$
    Repeat (for each step of episode):
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
        $S \leftarrow S'$;
    until $S$ is terminal
```

3 Experimental Procedure

The source code for this experiment includes `cliff_walking.py`, `sarsa.py`, `q_learning.py` and `main.py`.

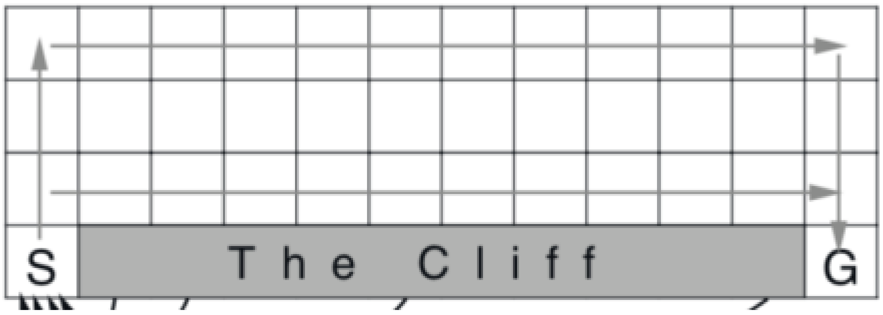
3.1 cliff_walking.py

`cliff_walking.py` builds the Gridworld environment. This module references [reinforcement-learning/cliff_walking.py at master · dennybritz/reinforcement-learning (github.com)](https://github.com).

The gridworld should be instantiated using the instruction as below:

```
gridworld = CliffWalkingEnv()
```

The gridworld will be instantiated as below, in which state 36 is set as the start state and state 47 is set as the terminal state.



The number of the states, the number of the actions and the state transfer matrix in gridworld can be obtained using the instruction as below:

```
state_number = gridworld.nS # 48
action_number = gridworld.nA # 4
state_transfer_matrix = gridworld.P # P[state][action] = (prob, next_state,
reward, is_done)
```

3.2 sarsa.py

`sarsa.py` is implemented based on the above pseudo-code.

First we need to define the ϵ-greedy policy:

$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon & \text{if } a^* = \arg\max_{a \in \mathcal{A}} Q(s,a) \\ \epsilon/m & \text{otherwise} \end{cases}$$

```python
def epsilon_greedy_action(gridworld, epsilon, Q, state):
    action_list = epsilon * np.ones(gridworld.nA) / gridworld.nA
    best_action = np.argmax(Q[state])
    action_list[best_action] += 1 - epsilon
    return action_list
```

Then we can use the above ϵ-greedy policy to both choose the next action and update the Q values. We implement the sarsa algorithm as below:

```python
def sarsa(gridworld, episode_num, epsilon, alpha=0.5, gamma=0.99):
    Q = np.zeros([gridworld.nS, gridworld.nA])
    for i in range(episode_num):
        is_done = False
        state = gridworld.reset()
        action_list = epsilon_greedy_action(gridworld, epsilon, Q, state)
        action = np.random.choice(gridworld.nA, 1, p=action_list)[0]
        while not is_done:
            prob, next_state, reward, is_done = gridworld.P[state][action]
[0]
            action_list = epsilon_greedy_action(
                gridworld, epsilon, Q, next_state)
            next_action = np.random.choice(gridworld.nA, 1, p=action_list)
[0]
            Q[state][action] = Q[state][action] + alpha * \
                (reward+gamma*Q[next_state][next_action] - Q[state][action])
            state = next_state
            action = next_action
    policy = np.zeros([gridworld.nS, gridworld.nA])
    is_done = False
    state = gridworld.reset()
    while not is_done:
        best_action = np.argmax(Q[state])
        policy[state] = np.eye(gridworld.nA)[best_action]
        prob, state, reward, is_done = gridworld.P[state][best_action][0]
    return policy, Q
```

3.3 q_learning.py

`q_learning.py` is implemented based on the above pseudo-code.

The defintion of ϵ-greedy policy is the same as that in `sarsa.py`.

We implement the Q-learning algorithm as below, while using the ϵ-greedy policy to choose the next action and greedy policy to update the Q values :

```python
def q_learning(gridworld, episode_num, epsilon, alpha=0.5, gamma=0.99):
    Q = np.zeros([gridworld.nS, gridworld.nA])
    for i in range(episode_num):
        is_done = False
        state = gridworld.reset()
        while not is_done:
            action_list = epsilon_greedy_action(gridworld, epsilon, Q, state)
            action = np.random.choice(gridworld.nA, 1, p=action_list)[0]
            prob, next_state, reward, is_done = gridworld.P[state][action][0]
            best_action = np.argmax(Q[state])
            Q[state][action] = Q[state][action] + alpha * \
                (reward+gamma*Q[next_state][best_action] - Q[state][action])
            state = next_state
    policy = np.zeros([gridworld.nS, gridworld.nA])
    is_done = False
    state = gridworld.reset()
    while not is_done:
        best_action = np.argmax(Q[state])
        policy[state] = np.eye(gridworld.nA)[best_action]
        prob, state, reward, is_done = gridworld.P[state][best_action][0]
    return policy, Q
```

3.4 main.py

`main.py` instantiates the environment and test Sarsa and Q-learning separately.

```
def test_sarsa(gridworld, episode_num, epsilon):
    policy, Q = sarsa.sarsa(gridworld, episode_num, epsilon)
    print('----------------sarsa----------------')
    print('Q Function:')
    print(np.around(np.reshape(Q,(4,12,4)), decimals=1))
    print('Policy:(UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3)')
    policy_show = ['x' for x in range(48)]
    for s in range(48):
        for a in range(4):
            if policy[s][a] == 1:
                policy_show[s] = a
    policy_show[47] = 'T'
    print(np.reshape(policy_show, gridworld.shape))
    print('episode Num:')
    print(episode_num)
    print('-------------------------------------')

def test_q_learning(gridworld, episode_num, epsilon):
    policy, Q = q_learning.q_learning(gridworld, episode_num, epsilon)
    print('----------------sarsa----------------')
    print('Q Function:')
    print(np.around(np.reshape(Q,(4,12,4)), decimals=1))
    print('Policy:(UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3)')
    policy_show = ['x' for x in range(48)]
    for s in range(48):
        for a in range(4):
            if policy[s][a] == 1:
                policy_show[s] = a
    policy_show[47] = 'T'
    print(np.reshape(policy_show, gridworld.shape))
    print('episode Num:')
    print(episode_num)
    print('-------------------------------------')
```

`test_sarsa()` and `test_q_learning()` separately use the method of Sarsa and Q-learning to finish the Cliff Walking. The Q values, the final policy and the number of episodes used are shown above.

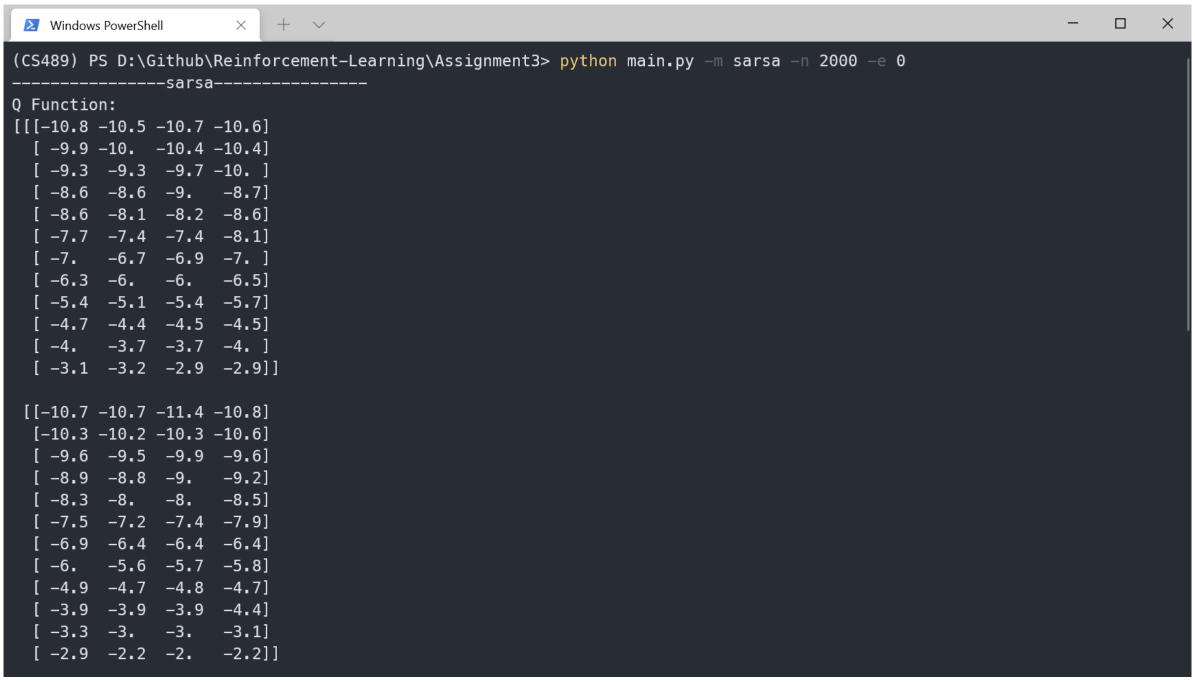
```python
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('-m', '--method', type=str, default='q_learning')
    parser.add_argument('-n', '--episode_num', type=int, default=1000)
    parser.add_argument('-e', '--epsilon', type=float, default=0.1)
    args = parser.parse_args()
    method = args.method
    episode_num = args.episode_num
    epsilon = args.epsilon

    gridworld = CliffWalkingEnv()
    if method == 'sarsa':
        test_sarsa(gridworld, episode_num, epsilon)
    if method == 'q_learning':
        test_q_learning(gridworld, episode_num, epsilon)
```

`main()` use parser to get the parameters entered on the command line, and depending on these parameters, the method of evaluation, the number of episodes used and the value of ϵ are chosen.

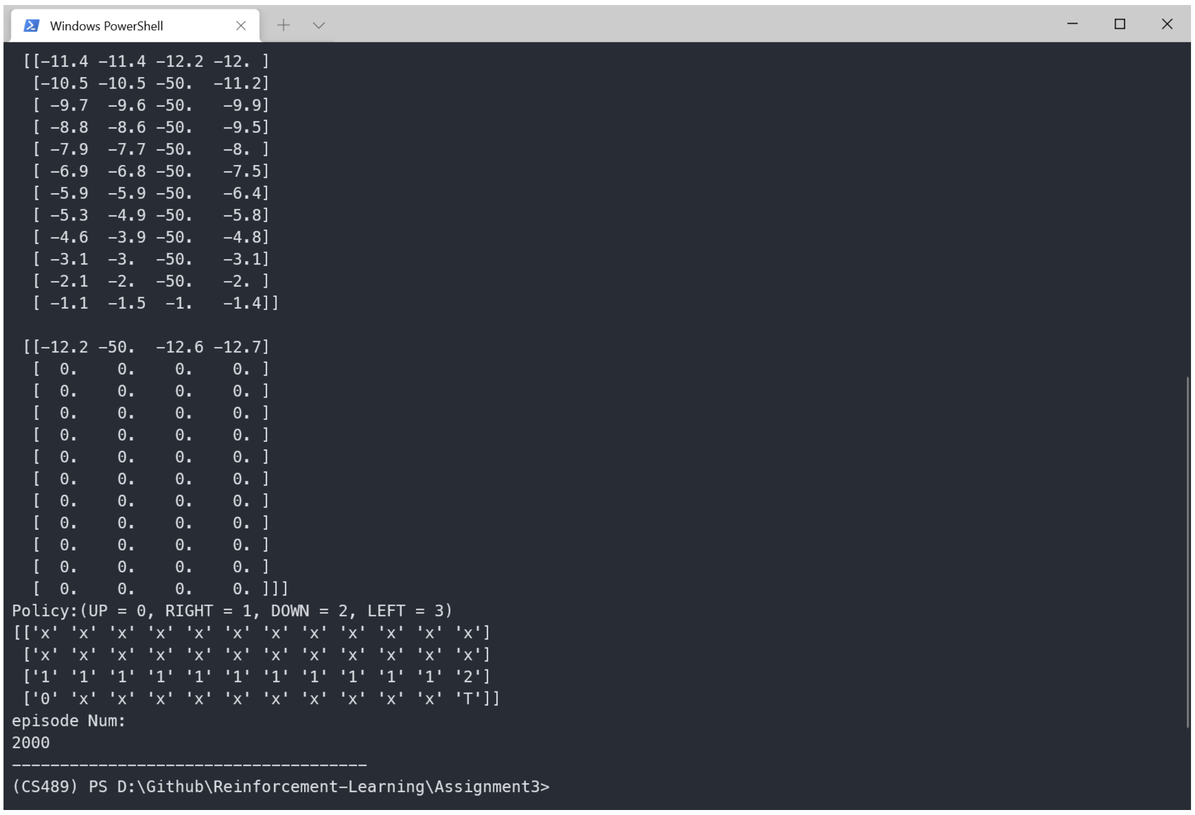
4 Experimental Result

When $\epsilon = 0$:

The result of sarsa:

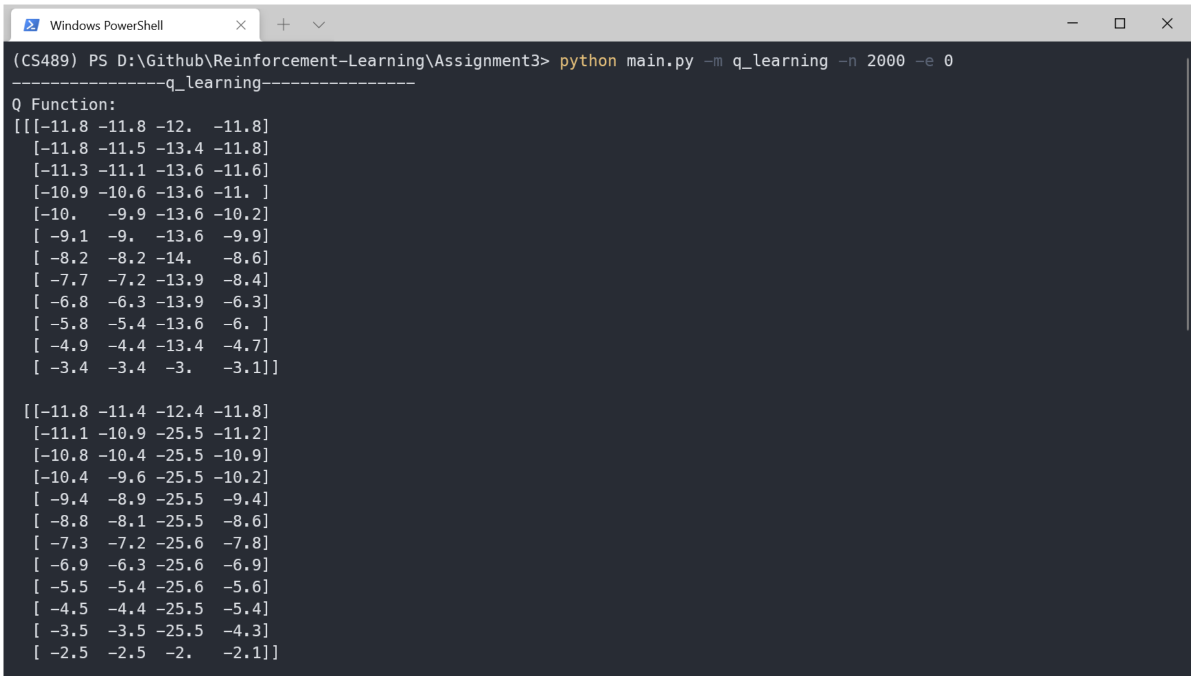


```
(CS489) PS D:\Github\Reinforcement-Learning\Assignment3> python main.py -m sarsa -n 2000 -e 0
----------------sarsa----------------
Q Function:
[[[-10.8 -10.5 -10.7 -10.6]
  [ -9.9 -10.  -10.4 -10.4]
  [ -9.3  -9.3  -9.7 -10. ]
  [ -8.6  -8.6  -9.   -8.7]
  [ -8.6  -8.1  -8.2  -8.6]
  [ -7.7  -7.4  -7.4  -8.1]
  [ -7.   -6.7  -6.9  -7. ]
  [ -6.3  -6.   -6.   -6.5]
  [ -5.4  -5.1  -5.4  -5.7]
  [ -4.7  -4.4  -4.5  -4.5]
  [ -4.   -3.7  -3.7  -4. ]
  [ -3.1  -3.2  -2.9  -2.9]]

 [[-10.7 -10.7 -11.4 -10.8]
  [-10.3 -10.2 -10.3 -10.6]
  [ -9.6  -9.5  -9.9  -9.6]
  [ -8.9  -8.8  -9.   -9.2]
  [ -8.3  -8.   -8.   -8.5]
  [ -7.5  -7.2  -7.4  -7.9]
  [ -6.9  -6.4  -6.4  -6.4]
  [ -6.   -5.6  -5.7  -5.8]
  [ -4.9  -4.7  -4.8  -4.7]
  [ -3.9  -3.9  -3.9  -4.4]
  [ -3.3  -3.   -3.   -3.1]
  [ -2.9  -2.2  -2.   -2.2]]
```

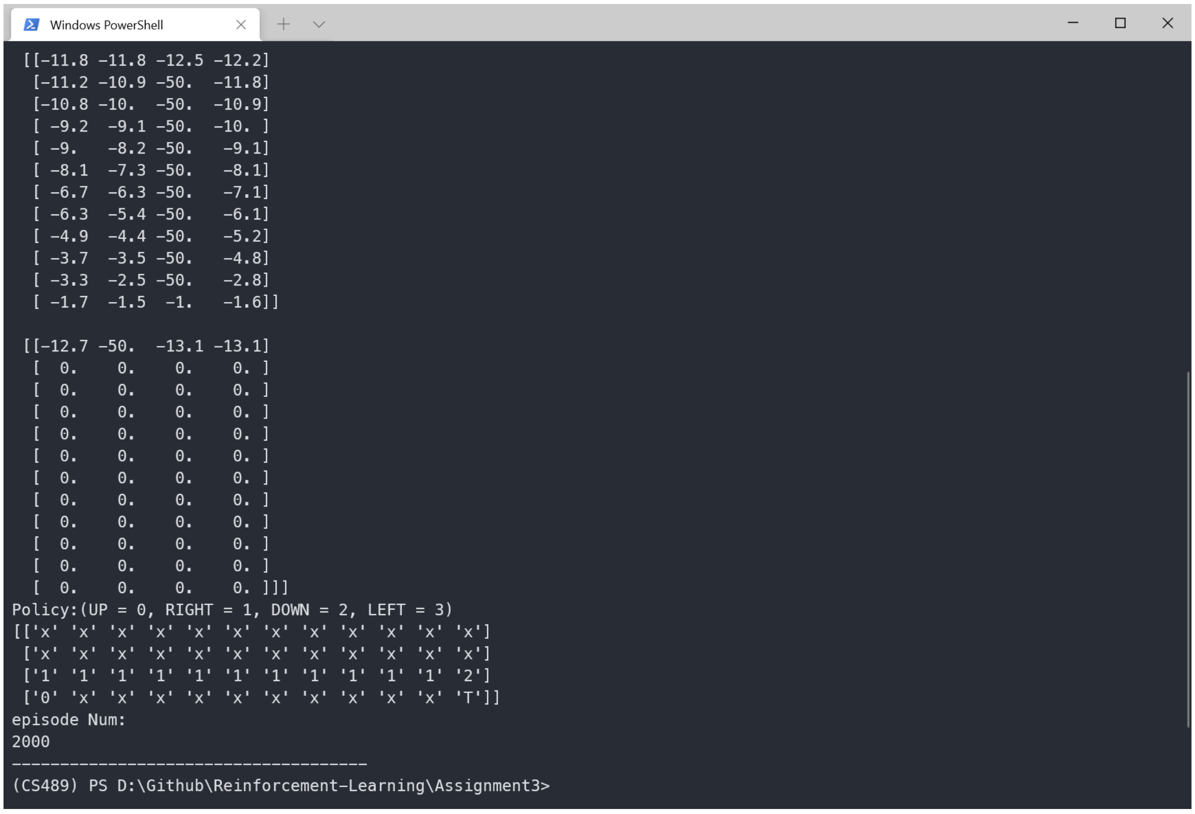


```
 [[-11.4 -11.4 -12.2 -12. ]
  [-10.5 -10.5 -50.  -11.2]
  [ -9.7  -9.6 -50.   -9.9]
  [ -8.8  -8.6 -50.   -9.5]
  [ -7.9  -7.7 -50.   -8. ]
  [ -6.9  -6.8 -50.   -7.5]
  [ -5.9  -5.9 -50.   -6.4]
  [ -5.3  -4.9 -50.   -5.8]
  [ -4.6  -3.9 -50.   -4.8]
  [ -3.1  -3.  -50.   -3.1]
  [ -2.1  -2.  -50.   -2. ]
  [ -1.1  -1.5  -1.   -1.4]]

 [[-12.2 -50.  -12.6 -12.7]
  [  0.    0.    0.    0. ]
  [  0.    0.    0.    0. ]
  [  0.    0.    0.    0. ]
  [  0.    0.    0.    0. ]
  [  0.    0.    0.    0. ]
  [  0.    0.    0.    0. ]
  [  0.    0.    0.    0. ]
  [  0.    0.    0.    0. ]
  [  0.    0.    0.    0. ]
  [  0.    0.    0.    0. ]
  [  0.    0.    0.    0. ]]]
Policy:(UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3)
[['x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x']
 ['x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x']
 ['1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '2']
 ['0' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'T']]
episode Num:
2000
-------------------------------------
(CS489) PS D:\Github\Reinforcement-Learning\Assignment3>
```

The result of Q-learning:



```
(CS489) PS D:\Github\Reinforcement-Learning\Assignment3> python main.py -m q_learning -n 2000 -e 0
----------------q_learning----------------
Q Function:
[[[-11.8 -11.8 -12.  -11.8]
  [-11.8 -11.5 -13.4 -11.8]
  [-11.3 -11.1 -13.6 -11.6]
  [-10.9 -10.6 -13.6 -11. ]
  [-10.   -9.9 -13.6 -10.2]
  [ -9.1  -9.  -13.6  -9.9]
  [ -8.2  -8.2 -14.   -8.6]
  [ -7.7  -7.2 -13.9  -8.4]
  [ -6.8  -6.3 -13.9  -6.3]
  [ -5.8  -5.4 -13.6  -6. ]
  [ -4.9  -4.4 -13.4  -4.7]
  [ -3.4  -3.4  -3.   -3.1]]

 [[-11.8 -11.4 -12.4 -11.8]
  [-11.1 -10.9 -25.5 -11.2]
  [-10.8 -10.4 -25.5 -10.9]
  [-10.4  -9.6 -25.5 -10.2]
  [ -9.4  -8.9 -25.5  -9.4]
  [ -8.8  -8.1 -25.5  -8.6]
  [ -7.3  -7.2 -25.6  -7.8]
  [ -6.9  -6.3 -25.6  -6.9]
  [ -5.5  -5.4 -25.6  -5.6]
  [ -4.5  -4.4 -25.5  -5.4]
  [ -3.5  -3.5 -25.5  -4.3]
  [ -2.5  -2.5  -2.   -2.1]]
```



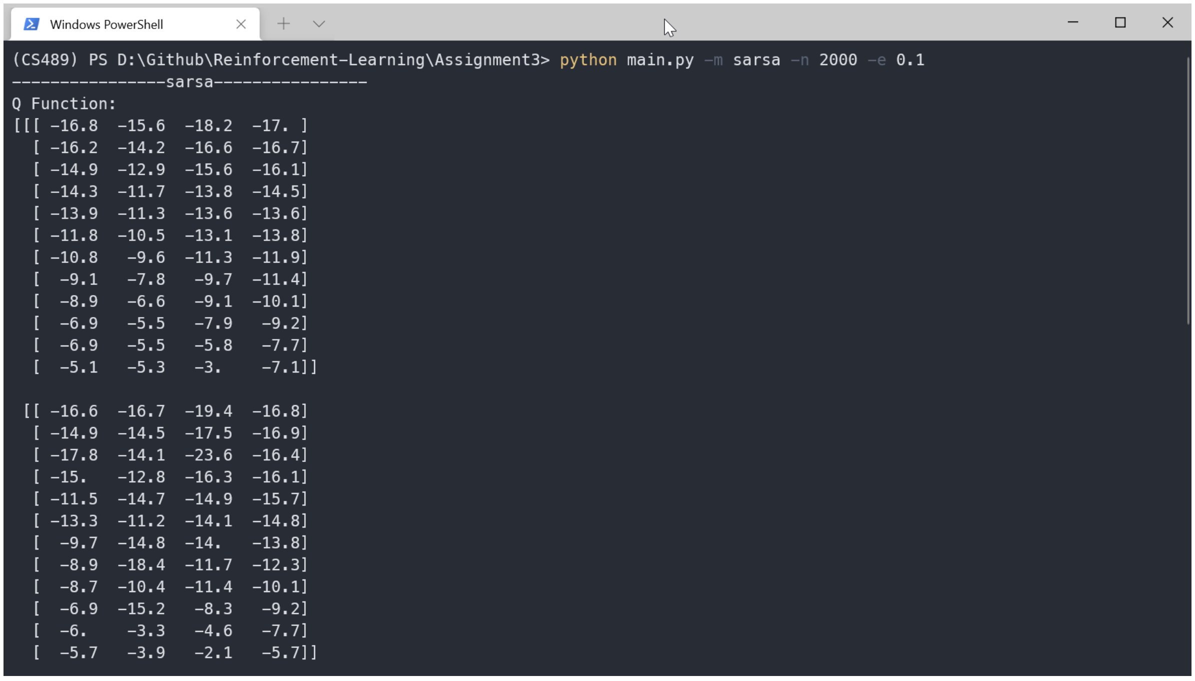
```
 [[-11.8 -11.8 -12.5 -12.2]
  [-11.2 -10.9 -50.  -11.8]
  [-10.8 -10.  -50.  -10.9]
  [ -9.2  -9.1 -50.  -10. ]
  [ -9.   -8.2 -50.   -9.1]
  [ -8.1  -7.3 -50.   -8.1]
  [ -6.7  -6.3 -50.   -7.1]
  [ -6.3  -5.4 -50.   -6.1]
  [ -4.9  -4.4 -50.   -5.2]
  [ -3.7  -3.5 -50.   -4.8]
  [ -3.3  -2.5 -50.   -2.8]
  [ -1.7  -1.5  -1.   -1.6]]

 [[-12.7 -50.  -13.1 -13.1]
  [  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. ]]]
Policy:(UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3)
[['x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x']
 ['x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x']
 ['1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '2']
 ['0' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'T']]
episode Num:
2000
-------------------------------------
(CS489) PS D:\Github\Reinforcement-Learning\Assignment3>
```

Both of the two methods choose the optimal path.

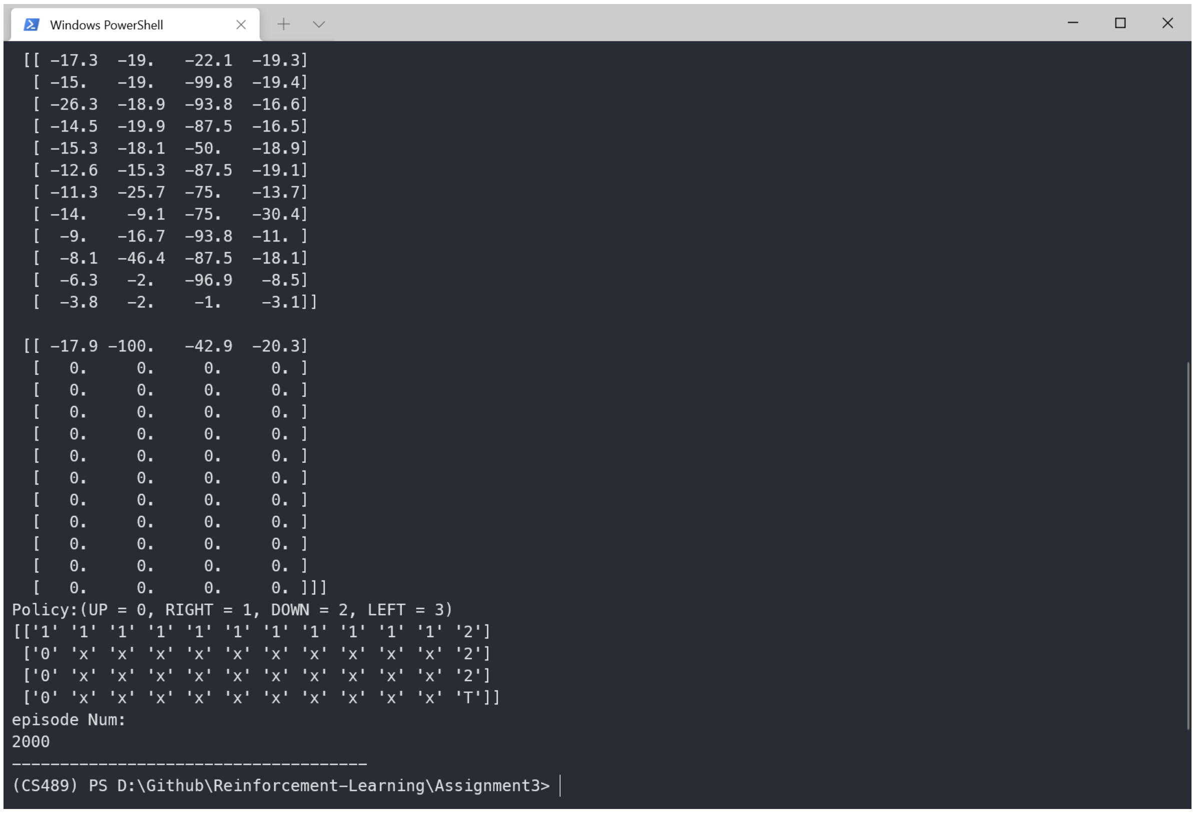
When $\epsilon = 0$, both the two methods are equal to TD(0), and the exploration is removed. That's the reason why the dangerous of cliff are not found by both the two methods.

When $\epsilon = 0.1$:

The result of sarsa:

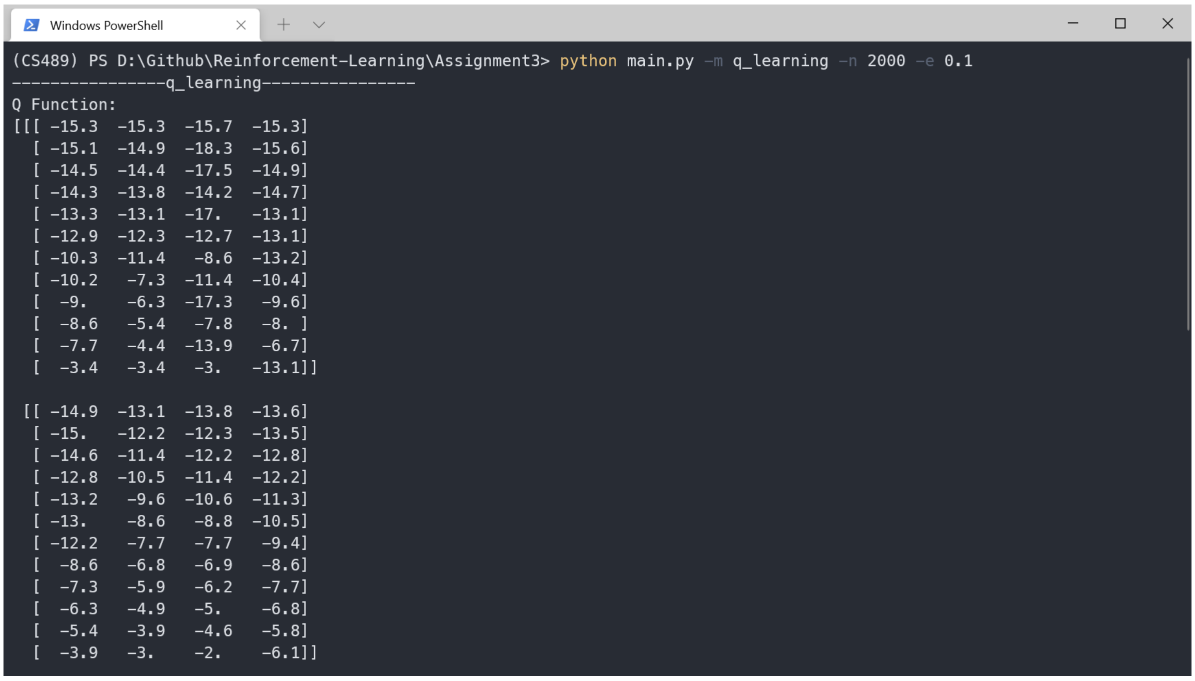


```
(CS489) PS D:\Github\Reinforcement-Learning\Assignment3> python main.py -m sarsa -n 2000 -e 0.1
----------------sarsa----------------
Q Function:
[[[ -16.8  -15.6  -18.2  -17. ]
  [ -16.2  -14.2  -16.6  -16.7]
  [ -14.9  -12.9  -15.6  -16.1]
  [ -14.3  -11.7  -13.8  -14.5]
  [ -13.9  -11.3  -13.6  -13.6]
  [ -11.8  -10.5  -13.1  -13.8]
  [ -10.8   -9.6  -11.3  -11.9]
  [  -9.1   -7.8   -9.7  -11.4]
  [  -8.9   -6.6   -9.1  -10.1]
  [  -6.9   -5.5   -7.9   -9.2]
  [  -6.9   -5.5   -5.8   -7.7]
  [  -5.1   -5.3   -3.    -7.1]]

 [[ -16.6  -16.7  -19.4  -16.8]
  [ -14.9  -14.5  -17.5  -16.9]
  [ -17.8  -14.1  -23.6  -16.4]
  [ -15.   -12.8  -16.3  -16.1]
  [ -11.5  -14.7  -14.9  -15.7]
  [ -13.3  -11.2  -14.1  -14.8]
  [  -9.7  -14.8  -14.   -13.8]
  [  -8.9  -18.4  -11.7  -12.3]
  [  -8.7  -10.4  -11.4  -10.1]
  [  -6.9  -15.2   -8.3   -9.2]
  [  -6.    -3.3   -4.6   -7.7]
  [  -5.7   -3.9   -2.1   -5.7]]
```

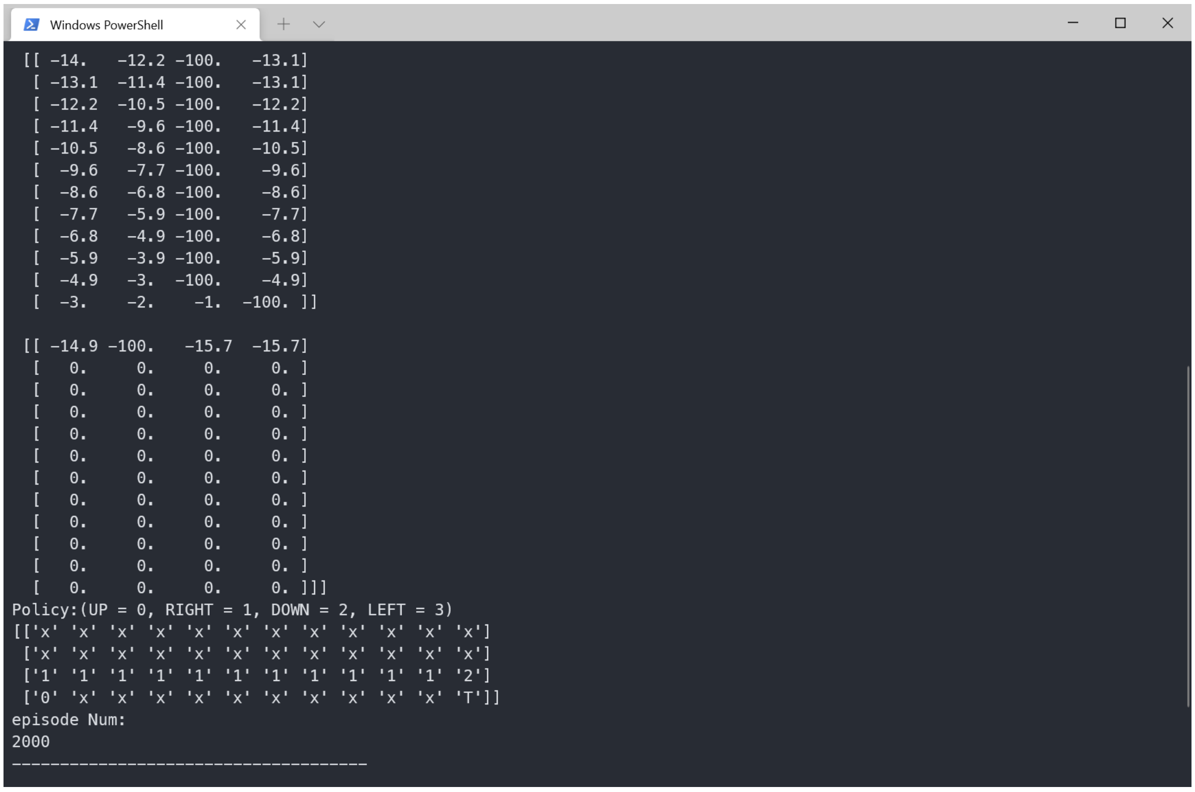


```
 [[ -17.3  -19.   -22.1  -19.3]
  [ -15.   -19.   -99.8  -19.4]
  [ -26.3  -18.9  -93.8  -16.6]
  [ -14.5  -19.9  -87.5  -16.5]
  [ -15.3  -18.1  -50.   -18.9]
  [ -12.6  -15.3  -87.5  -19.1]
  [ -11.3  -25.7  -75.   -13.7]
  [ -14.    -9.1  -75.   -30.4]
  [  -9.   -16.7  -93.8  -11. ]
  [  -8.1  -46.4  -87.5  -18.1]
  [  -6.3   -2.   -96.9   -8.5]
  [  -3.8   -2.    -1.    -3.1]]

 [[ -17.9 -100.   -42.9  -20.3]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]]]
Policy:(UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3)
[['1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '2']
 ['0' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' '2']
 ['0' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' '2']
 ['0' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'T']]
episode Num:
2000
-------------------------------------
(CS489) PS D:\Github\Reinforcement-Learning\Assignment3>
```

The result of Q-learning:



```
(CS489) PS D:\Github\Reinforcement-Learning\Assignment3> python main.py -m q_learning -n 2000 -e 0.1
----------------q_learning----------------
Q Function:
[[[ -15.3  -15.3  -15.7  -15.3]
  [ -15.1  -14.9  -18.3  -15.6]
  [ -14.5  -14.4  -17.5  -14.9]
  [ -14.3  -13.8  -14.2  -14.7]
  [ -13.3  -13.1  -17.   -13.1]
  [ -12.9  -12.3  -12.7  -13.1]
  [ -10.3  -11.4   -8.6  -13.2]
  [ -10.2   -7.3  -11.4  -10.4]
  [  -9.    -6.3  -17.3   -9.6]
  [  -8.6   -5.4   -7.8   -8. ]
  [  -7.7   -4.4  -13.9   -6.7]
  [  -3.4   -3.4   -3.   -13.1]]

 [[ -14.9  -13.1  -13.8  -13.6]
  [ -15.   -12.2  -12.3  -13.5]
  [ -14.6  -11.4  -12.2  -12.8]
  [ -12.8  -10.5  -11.4  -12.2]
  [ -13.2   -9.6  -10.6  -11.3]
  [ -13.    -8.6   -8.8  -10.5]
  [ -12.2   -7.7   -7.7   -9.4]
  [  -8.6   -6.8   -6.9   -8.6]
  [  -7.3   -5.9   -6.2   -7.7]
  [  -6.3   -4.9   -5.    -6.8]
  [  -5.4   -3.9   -4.6   -5.8]
  [  -3.9   -3.    -2.    -6.1]]
```



```
 [[ -14.   -12.2 -100.   -13.1]
  [ -13.1  -11.4 -100.   -13.1]
  [ -12.2  -10.5 -100.   -12.2]
  [ -11.4   -9.6 -100.   -11.4]
  [ -10.5   -8.6 -100.   -10.5]
  [  -9.6   -7.7 -100.    -9.6]
  [  -8.6   -6.8 -100.    -8.6]
  [  -7.7   -5.9 -100.    -7.7]
  [  -6.8   -4.9 -100.    -6.8]
  [  -5.9   -3.9 -100.    -5.9]
  [  -4.9   -3.  -100.    -4.9]
  [  -3.    -2.    -1.  -100. ]]

 [[ -14.9 -100.   -15.7  -15.7]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]
  [   0.     0.     0.     0. ]]]
Policy:(UP = 0, RIGHT = 1, DOWN = 2, LEFT = 3)
[['x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x']
 ['x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x']
 ['1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '1' '2']
 ['0' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'x' 'T']]
episode Num:
2000
-------------------------------------
```

Q-learning chooses the optimal path while Sarsa chooses the safer path.

The reason of the difference is that when updating the Q values, on-policy Sarsa learns action values relative to the policy it follows, while off-policy Q-learning does it relative to the greedy policy. So after the Q function is converged, the agent in Q-learning will never get into the Q value of the cliff while the agent in Sarsa has a certain probability to get into it. That is the reason why Sarsa choose to keep away from the cliff while Q-learning does not find the potential hazards.

5 Experimental Summary

- Systematically learn and understand Sarsa and Q-learning
- Implement and successfully run the algorithm of Sarsa and Q-learning based on pseudo-code
- Verify and Analyze the differences between the two methods based on the results