上海交通大學

SHANGHAI JIAOTONG UNIVERSITY

强化学习课程设计 - Project3

Model-free Control

姓名：薛春宇

学号：518021910698

完成时间：2021/4/9

1 实验目的

分别实现Sarsa算法 (on-policy learning) 和Q-Learning算法 (off-policy learning) 来解决Model-free Control中无折扣因子的Cliff Walking问题，并通过调整不同的ϵ值来直观地感受Sarsa与Q-Learning在路径决策上的不同。

2 实验准备

在本节中，我们将分别介绍Sarsa算法和Q-Learning算法的相关内容，以为接下来的代码实现做准备。

2.1 Sarsa算法

Sarsa算法是一种On-policy的Model-free控制算法，其主要思想是将探索和决策两个过程合并在一起，使用一个更新策略 $\epsilon - greedy$ 来不断进行迭代。

具体来说，Sarsa算法首先随机初始化一个state（在本例中的Cliff Walking问题中，为除起、终点以及Cliff以外的其他state），在当前state下利用$\epsilon - greedy$选择一个即将采用的action，然后开始迭代：

(1) **说到做到地**采取当前 action，观察下一个 state'，再利用$\epsilon - greedy$寻找下一个action'，且**说到做到地**在下一次迭代中采取该action'

(2) 更新Q(state, action)，基于当前的Q(state, action)和即将要采用的Q(state'，action')

(3) 移动：state = state'，action = action'

(4) 重复上述迭代过程直到到达terminal state

整个policy iteration的伪代码如下所示：

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., ε-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-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

我们的python代码也是基本按照上述伪代码的思想进行实现的。

2.2 Q-Learning 算法

Q-Learning 算法是一种 Off-policy 的 Model-free 控制算法，其主要思想是将探索和决策两个过程分开实现，使用更新策略 $\epsilon - greedy$ 来不断进行探索，使用贪心策略来进行策略改进，即决策，以各执行上述两个过程一次为一次迭代。

具体来说，Sarsa 算法首先随机初始化一个 state（在本例中的 Cliff Walking 问题中，为除起、终点以及 Cliff 以外的其他 state），然后开始迭代：

(1) 当前 state 下利用 $\epsilon - greedy$ 选择一个即将采用的 action

(2) **说到做到地**采取这个 action，观察下一个 state'，再利用**普通的基于 Q 的贪心策略**寻找下一个 action'，且**仅假设**采取该 action'（实际采取哪个 action'需要在每次迭代的第一步决定）

(3) 更新 Q(state, action)，基于当前的 Q(state, action)和即将要采用的 Q(state'，action')

(4) 移动：state = state'

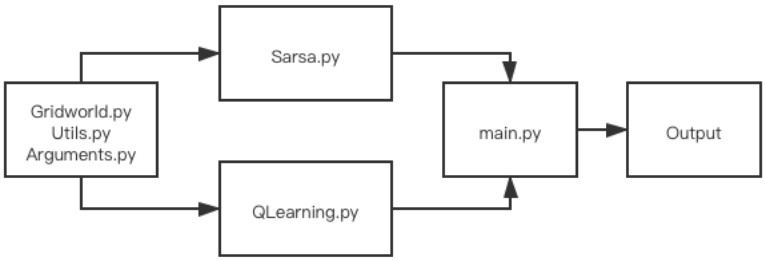
(5) 重复上述迭代过程直到到达 terminal state

Q-Learning 算法的伪代码如下所示：

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
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-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 实验内容

本次实验的代码及逻辑结构如下所示：



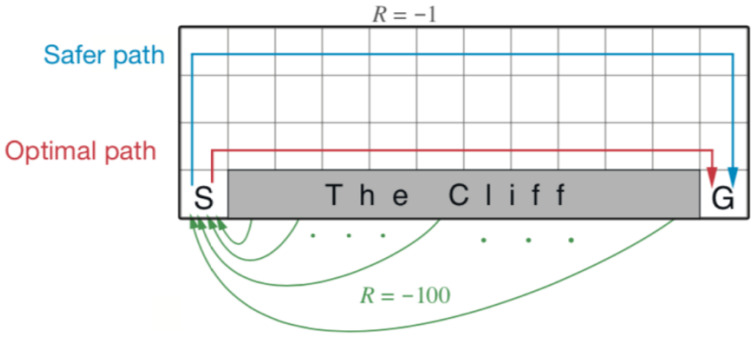
接下来，我们将对几个主要文件进行逐一分析，并分析两类控制算法的差异。相关代码位于./code 目录下。

3.1 Gridworld.py

该模块的实现参考了 Sutton 版 RL 提供的 Gridworld 类，实现了一个可实例化的格子世界，可以通过如下指令进行实例化：

```
# Grid Map Definition
# Order: from top left is 0, to right is 1, 2, ..., to down is 12, 24, ...
env_for_Sarsa = GridworldEnv([4, 12])
env_for_QLearning = GridworldEnv([4, 12])
```

需要注意的是，本项目中的 Terminal State、Action Tpye、Reward 和概率转移矩阵 P 均需要在 Gridworld.py 中声明。本项目中的 Gridworld 需要被实例化为如下结构（其中 36 号 grid 为起点 S，47 号 grid 为终点 G，37～46 号 grid 为 Cliff state）：



3.2 Utils.py & Arguments.py

Utils.py 主要实现了 $\epsilon - greedy$ 概率选择策略，通过调用 random_seed，将所有的 action 作为 key 存在一个字典中，value 对应它们的概率大小，注意这些概率是对 0~1 区间进行划分的子区间宽度：

```
# Implementation of epsilon-greedy policy
# _epsilon: 0.1 by default
# Return: a action decided by epsilon-greedy policy
def epsilon_greedy_policy(state, env, Q, _epsilon=0.1):
    # m in epsilon-greedy policy
    m_size = env.nA
    # Best action
    best_action = calculate_action_value_based_on_Q(state, Q)
    # If epsilon is 0
    if _epsilon == 0:
        return best_action

    # Construct probability dict = {action1: prob1, action2: prob2, ...}
    prob_dict = {}
    cur_prob_sum = 0
    for action in range(env.nA):
        # In case of accuracy loss, which means that cur_prob_sum is not 1 in the end
        if action == env.nA - 1:...

        if action == best_action:
            prob_dict[action] = cur_prob_sum + _epsilon / m_size + 1 - _epsilon
            cur_prob_sum += _epsilon / m_size + 1 - _epsilon
        else:
            prob_dict[action] = cur_prob_sum + _epsilon / m_size
            cur_prob_sum += _epsilon / m_size

    # Determine which action to do by epsilon-greedy policy
    # Random seed
    random_seed = secret_generator.randint(0, 1000000) * 0.000001
    # print("Random seed: ", random_seed, " prob dict: ", prob_dict)
    # Check
    for action in prob_dict:
        if random_seed <= prob_dict[action]:
            return action
```

其中，calculate_action_value_based_on_Q()为自行实现的获取当前 state 基于贪心的最佳

action 的函数。

此外，Arguments.py 主要定义了一些模型相关的参数，包括ϵ，γ，α和迭代轮数：

```python
# Arguments.py

# Arguments
_discount_factor = 1.0
_alpha = 0.01
# _epsilon should not be very small, or Sarsa will choose the first safe path, which is more like a deterministic policy
# _epsilon should also not be very large, or the result will be hard to converge
_epsilon = 0.4
# At least iterate 300000 rounds, in order to guarantee the stability of the algorithm
basic_rounds = 300000
```

3.3 Sarsa.py

该模块的实现基本遵循了 2.1 节中给出的伪代码的思想，并将整体内容设置成一个可调用的函数 Sarsa_learning()，供用户在 main.py 中调用。

在函数的开始部分，我们根据已知的参数大小（env.nS 为 Gridworld 的全部状态数，env.nA 为 Gridworld 的全部行为数）来创建行为值函数 Q 的容器：

```python
    print("Begin Sarsa Learning...")
    # Action value function
    Q = np.zeros([env.nS, env.nA])
```

按照 2.1 节中伪代码的思想，我们用一个 while 循环来表示不断在 Gridworld 中探索,并首先随机初始化一个合适的 state，根据 $\epsilon - greedy$ 策略选择一个初始 action。

```python
    count = 0
    # Repeat loop to update Q function
    while count < basic_rounds:
        # Update counter
        count += 1
        # Display Info
        if count % 10000 == 0:
            print("Iteration: ", count, "/", basic_rounds)
        # Initialize start state
        cur_state = secret_generator.randint(0, 35)
        # Choose start action based on epsilon-greedy policy
        cur_action = epsilon_greedy_policy(cur_state, env, Q, _epsilon)
```

接下来，我们开始探索以及策略改进，通过不停调用 $\epsilon - greedy$ 策略来更新行为值函数 Q 以及探索下一个状态和行为，并移动 agent（该 while 循环包含在上面的 while 内部）：

```python
        # Explore the episode (not traverse, since the episode is not deterministic)
        while True:
            # Arrive at the terminal state, break
            if is_terminal_state(cur_state):
                break

            # Take current action
            for prob, next_state, reward, is_done in env.P[cur_state][cur_action]:
                # Choose next action based on epsilon-greedy policy
                next_action = epsilon_greedy_policy(next_state, env, Q, _epsilon)
                # Update Q function
                Q[cur_state][cur_action] = Q[cur_state][cur_action] + _alpha * (reward
                                                                               + _discount_factor * Q[next_state][
                                                                                   next_action] - Q[cur_state][
                                                                                   cur_action])
                # Move
                cur_state = next_state
                cur_action = next_action
```

在达到最大迭代次数后，跳出最外层的 while 循环，开始构造确定性策略。注意，在构造确定性策略时，我们直接使用贪心算法进行最优路径的选择：

```
    # Policy vector
    my_policy = np.zeros(env.nS)
    # Initialize the policy
    for state in range(env.nS):
        my_policy[state] = -1
    # Generate a deterministic policy
    my_state = start_state
    while not is_terminal_state(my_state):
        # Choose the best action based on greedy
        my_action = np.argmax(Q[my_state])
        # Write into policy
        my_policy[my_state] = my_action
        # Move
        for prob, next_state, reward, is_done in env.P[my_state][my_action]:
            my_state = next_state

    # Return
    return my_policy, Q
```

3.4 QLearning.py

该模块的实现基本遵循了 2.2 节中给出的伪代码的思想，并将整体内容设置成一个可调用的函数 Q_learning()，供用户在 main.py 中调用。在函数的开始部分，我们根据已知的参数大小（env.nS 为 Gridworld 的全部状态数，env.nA 为 Gridworld 的全部行为数）来创建行为值函数 Q 的容器：

```
    print("Begin Q-Learning...")
    # Action value function
    Q = np.zeros([env.nS, env.nA])
```

按照 2.1 节中伪代码的思想，我们用一个 while 循环来表示不断在 Gridworld 中探索,并首先随机初始化一个合适的 state。

```
    count = 0
    # Repeat loop to update Q function
    while count < basic_rounds:
        # Update counter
        count += 1
        # Display Info
        if count % 10000 == 0:
            print("Iteration: ", count, "/", basic_rounds)
        # Initialize start state
        cur_state = secret_generator.randint(0, 35)
```

接下来，我们开始探索以及策略改进，首先调用 $\epsilon - greedy$ 策略选择当前 state 下应该采取的 action，再用贪心算法**假设**下一个 state'应该选择的 action'，并更新 Q 及状态移动：

```
    # Explore the episode (not traverse, since the episode is not deterministic)
    while True:
        # Arrive at the terminal state, break
        if is_terminal_state(cur_state):
            break

        # Choose action based on epsilon-greedy policy
        cur_action = epsilon_greedy_policy(cur_state, env, Q, _epsilon)

        # Take current action
        for prob, next_state, reward, is_done in env.P[cur_state][cur_action]:
            # Choose ASSUMED next action based on NORMAL greedy policy
            assumed_next_action = np.argmax(Q[next_state])
            # Update Q function
            Q[cur_state][cur_action] = Q[cur_state][cur_action] + _alpha * (
                    reward + _discount_factor * Q[next_state][assumed_next_action] - Q[cur_state][cur_action])
            # Move
            cur_state = next_state
```

注意，上述的 while 循环包含在前面的 while 循环内部。

在达到最大迭代次数后，跳出最外层的 while 循环，开始构造确定性策略。注意，在构造确定性策略时，我们直接使用贪心算法进行最优路径的选择：

```
    # Policy vector
    my_policy = np.zeros(env.nS)
    # Initialize the policy
    for state in range(env.nS):
        my_policy[state] = -1
    # Generate a deterministic policy
    my_state = start_state
    while not is_terminal_state(my_state):
        # Choose the best action based on greedy
        my_action = np.argmax(Q[my_state])
        # Write into policy
        my_policy[my_state] = my_action
        # Move
        for prob, next_state, reward, is_done in env.P[my_state][my_action]:
            my_state = next_state

    # Return
    return my_policy, Q
```

3.5 main.py

主函数中，我们首先进行了 Cliff gridworld 的初始化定义：

```
# Grid Map Definition
# Order: from top left is 0, to right is 1, 2, ..., to down is 12, 24, ...
env_for_Sarsa = GridworldEnv([4, 12])
env_for_QLearning = GridworldEnv([4, 12])
```

然后，分别设置两个封装函数，对上述两种算法的调用进行封装，以及输出的格式化：

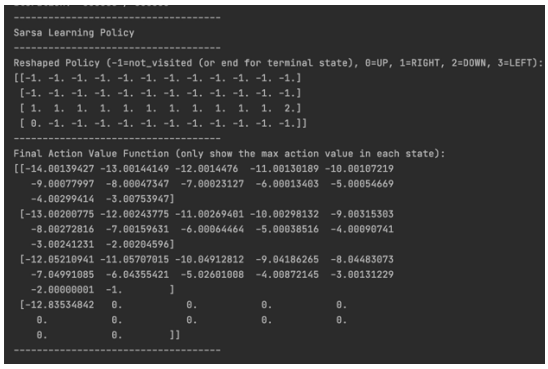
```
def Sarsa_learning_policy(env):
    # Sarsa learning
    policy, Q = Sarsa_learning(env, _discount_factor, _alpha, _epsilon, basic_rounds)
    # Output
    print("-----------------------------------")
    print("Sarsa Learning Policy")
    print("-----------------------------------")
    print("Reshaped Policy (-1=not_visited (or end for terminal state), 0=UP, 1=RIGHT, 2=DOWN, 3=LEFT):")
    print(np.reshape(policy, env.shape))
    print("-----------------------------------")
    print("Final Action Value Function (only show the max action value in each state):")
    print(np.reshape(np.max(Q, axis=1), env.shape))
    print("-----------------------------------")
    print("")


def Q_learning_policy(env):
    # Q-learning
    policy, Q = Q_learning(env, _discount_factor, _alpha, _epsilon, basic_rounds)
    # Output
    print("-----------------------------------")
    print("Q-Learning Policy")
    print("-----------------------------------")
    print("Reshaped Policy (-1=not_visited (or end for terminal state), 0=UP, 1=RIGHT, 2=DOWN, 3=LEFT):")
    print(np.reshape(policy, env.shape))
    print("-----------------------------------")
    print("Final Action Value Function (only show the max action value in each state):")
    print(np.reshape(np.max(Q, axis=1), env.shape))
    print("-----------------------------------")
    print("")
```

最后，我们在 main()函数中调用上述两个封装函数，即可完成算法的调用及结果输出。

4 实验结果

本节中，我们将基于不同 ϵ 值，分别使用 Sarsa 算法和 Q-Learning 算法进行对比实验，讨论两种算法在路径决策上的差异性。

首先，我们将 ϵ 值设置为 0.001，运行算法：

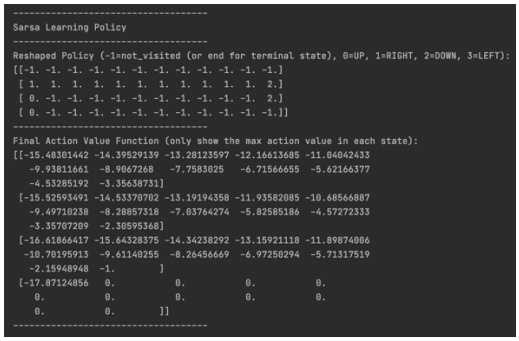


```
------------------------------------
Sarsa Learning Policy
------------------------------------
Reshaped Policy (-1=not_visited (or end for terminal state), 0=UP, 1=RIGHT, 2=DOWN, 3=LEFT):
[[-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  2.]
 [ 0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]]
------------------------------------
Final Action Value Function (only show the max action value in each state):
[[-14.00139427 -13.00144149 -12.0014476  -11.00130189 -10.00107219
   -9.00077997  -8.00047347  -7.00023127  -6.00013403  -5.00054669
   -4.00299414  -3.00753947]
 [-13.00200775 -12.00243775 -11.00269401 -10.00298132  -9.00315303
   -8.00272816  -7.00159631  -6.00064464  -5.00038516  -4.00090741
   -3.00241231  -2.00204596]
 [-12.05210941 -11.05707015 -10.04912812  -9.04186265  -8.04483073
   -7.04991085  -6.04355421  -5.02601008  -4.00872145  -3.00131229
   -2.00000001  -1.        ]
 [-12.83534842   0.           0.           0.           0.
    0.           0.           0.           0.           0.
    0.           0.        ]]
------------------------------------
```

```
------------------------------------
Q-Learning Policy
------------------------------------
Reshaped Policy (-1=not_visited (or end for terminal state), 0=UP, 1=RIGHT, 2=DOWN, 3=LEFT):
[[-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  2.]
 [ 0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]]
------------------------------------
Final Action Value Function (only show the max action value in each state):
[[-14.         -13.         -12.         -11.         -10.
   -9.          -8.          -7.          -6.          -5.
   -4.          -3.        ]
 [-13.         -12.         -11.         -10.          -9.
   -8.          -7.          -6.          -5.          -4.
   -3.          -2.        ]
 [-12.         -11.         -10.          -9.          -8.
   -7.          -6.          -5.          -4.          -3.
   -2.          -1.        ]
 [-12.85637827   0.           0.           0.           0.
    0.           0.           0.           0.           0.
    0.           0.        ]]
------------------------------------
```

Sarsa 算法（$\epsilon = 0.001$）　　Q-Learning 算法（$\epsilon = 0.001$）

可以看到，两种算法均选择了 cost 相对最小的最优路径。接着，我们将 ϵ 值设置为 0.1，运行算法：

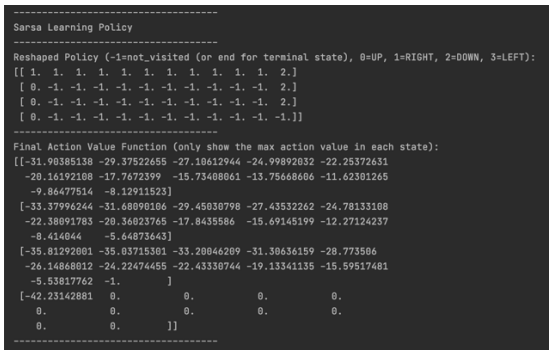


```
------------------------------------
Sarsa Learning Policy
------------------------------------
Reshaped Policy (-1=not_visited (or end for terminal state), 0=UP, 1=RIGHT, 2=DOWN, 3=LEFT):
[[-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  2.]
 [ 0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.  2.]
 [ 0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]]
------------------------------------
Final Action Value Function (only show the max action value in each state):
[[-15.48301442 -14.39529139 -13.28123597 -12.16613685 -11.04042433
   -9.93811661  -8.9067268   -7.7583025   -6.71566655  -5.62166377
   -4.53285192  -3.35638731]
 [-15.52593491 -14.53370702 -13.19194358 -11.93582085 -10.68566887
   -9.49710238  -8.28857318  -7.03764274  -5.82585186  -4.57272333
   -3.35707209  -2.30595368]
 [-16.61866417 -15.64328375 -14.34238292 -13.15921118 -11.89874086
  -10.70195913  -9.61140255  -8.26456669  -6.97250294  -5.71317519
   -2.15948948  -1.        ]
 [-17.87124856   0.           0.           0.           0.
    0.           0.           0.           0.           0.
    0.           0.        ]]
------------------------------------
```

```
------------------------------------
Q-Learning Policy
------------------------------------
Reshaped Policy (-1=not_visited (or end for terminal state), 0=UP, 1=RIGHT, 2=DOWN, 3=LEFT):
[[-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  2.]
 [ 0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]]
------------------------------------
Final Action Value Function (only show the max action value in each state):
[[-14. -13. -12. -11. -10.  -9.  -8.  -7.  -6.  -5.  -4.  -3.]
 [-13. -12. -11. -10.  -9.  -8.  -7.  -6.  -5.  -4.  -3.  -2.]
 [-12. -11. -10.  -9.  -8.  -7.  -6.  -5.  -4.  -3.  -2.  -1.]
 [-13.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]]
------------------------------------
```

Sarsa 算法（$\epsilon = 0.1$）　　Q-Learning 算法（$\epsilon = 0.1$）

当 ϵ 值取到 0.1 时，Sarsa 算法选择了一个相对更安全，但并非 cost 最优的路径；而 Q-Learning 算法仍选择 cost 最优的路径。最后，我们将 ϵ 值设置为 0.4，运行算法：



```
------------------------------------
Sarsa Learning Policy
------------------------------------
Reshaped Policy (-1=not_visited (or end for terminal state), 0=UP, 1=RIGHT, 2=DOWN, 3=LEFT):
[[ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  2.]
 [ 0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.  2.]
 [ 0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.  2.]
 [ 0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]]
------------------------------------
Final Action Value Function (only show the max action value in each state):
[[-31.90385138 -29.37522655 -27.10612944 -24.99892032 -22.25372631
  -20.16192108 -17.7672399  -15.73408861 -13.75668606 -11.62301265
   -9.86477514  -8.12911523]
 [-33.37996244 -31.68090186 -29.45030798 -27.43532262 -24.78133108
  -22.38091783 -20.36023765 -17.8435586  -15.69145199 -12.27124237
   -8.414044    -5.64873643]
 [-35.81292001 -35.03715301 -33.20046209 -31.30636159 -28.773586
  -26.14868012 -24.22474455 -22.43330744 -19.13341135 -15.59517481
   -5.53817762  -1.        ]
 [-42.23142881   0.           0.           0.           0.
    0.           0.           0.           0.           0.
    0.           0.        ]]
------------------------------------
```

```
------------------------------------
Q-Learning Policy
------------------------------------
Reshaped Policy (-1=not_visited (or end for terminal state), 0=UP, 1=RIGHT, 2=DOWN, 3=LEFT):
[[-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [-1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]
 [ 1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  1.  2.]
 [ 0. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1. -1.]]
------------------------------------
Final Action Value Function (only show the max action value in each state):
[[-14. -13. -12. -11. -10.  -9.  -8.  -7.  -6.  -5.  -4.  -3.]
 [-13. -12. -11. -10.  -9.  -8.  -7.  -6.  -5.  -4.  -3.  -2.]
 [-12. -11. -10.  -9.  -8.  -7.  -6.  -5.  -4.  -3.  -2.  -1.]
 [-13.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.   0.]]
------------------------------------
```

Sarsa 算法（$\epsilon = 0.4$）　　Q-Learning 算法（$\epsilon = 0.4$）

当 ϵ 值取到 0.4 时，Sarsa 算法选择了一个所有路径选择中最安全的路径，但并非 cost 最优的路径；而 Q-Learning 算法仍选择 cost 最优的路径。

结合上述实验结果，分别分析 Sarsa 和 Q-Learning 算法的特性可以发现：

(1) Sarsa 是说到做到型, 所以我们也叫他 On-policy, 在线学习, 学着自己在做的事情

(2) Q-Learning 是说到但并不一定做到, 所以它也叫作 Off-policy, 离线学习

而因为有了贪心选择 max_Q，Q-Learning 也是一个特别勇敢的算法。因为 Q-Learning 机器人永远都会选择最近的一条通往成功的道路, 不管这条路会有多危险。而 Sarsa 则是相当保守, 他会选择离危险远远的, 拿到宝藏是次要的, 保住自己的小命才是王道。这就是使用 Sarsa 方法和使用 Q-Learning 方法的不同之处。

5 实验心得

在本次实验的过程中，我首先在课堂授课内容的基础上，系统性地学习了 Sarsa 算法和 Q-Learning 算法的相关知识，并掌握了其在代码层面上的实现方法。在实现的过程中，由于从伪代码到 python 代码的差别还是比较大的，我遇到了一些困难，其中就包括了在实现 $\epsilon - greedy$ 策略选择的时候，由于当 ϵ 值取的足够小时，python 会出现一些精度丢失，导致算法出现 bug。在发现这个 bug 之后，我进行了一定的代码优化，解决了这个 bug。

整个过程约花费半天时间，在本次实验中，我不仅了解了两种 Model-free 控制方案的基本知识，掌握了从伪代码到可运行代码的复现方法，更是提高了自身发现 bug，解决 bug 的能力。希望在接下来的实验中也能收获满满！