Assignment 3

BY YUSEN ZHENG

March 25, 2023

Email: zys0794@sjtu.edu.cn

Student ID: 520021911173

1 Introduction

In this assignment, we built the **Cliff Walking environment** and used **Sara and Q-learning algorithm** to search the optimal travel path. Also, we studied **the impacts of the ϵ value** on performances. The Cliff Walking task we studied is shown below:

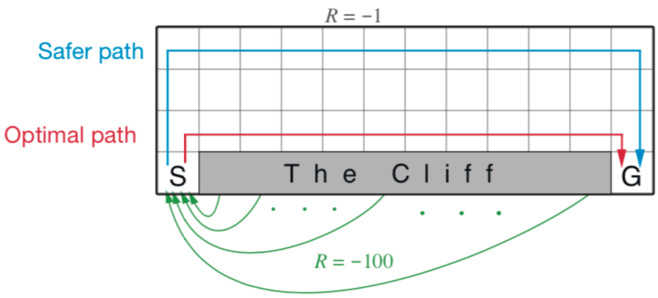

Figure 1. Cliff Walking

2 Cliff Walking Environment

We built the cliff walking environment in this task, and assigned a continuous coordinate ID to each grid point (like in GridWorld). As shown below, position 36 is start, position 47 is goal, and position 37–46 is cliff.

0	1	2	3	4	5	6	7	8	9	10	11
12	13	14	15	16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44	45	46	47

Figure 2. Cliff Walking Environment

We defined the **Grid** and **Cliff** classes, which record information about each grid point and the entire environment, respectively.

```
class Grid:
    def __init__(self, position, value=.0, is_start=False, is_goal=False,
is_cliff=False):
        self.val = value
        self.pos = position
        self.act = None
```

```
        self.is_start = is_start
        self.is_goal = is_goal
        self.is_cliff = is_cliff


class Cliff:
    def __init__(self, width, height, start, goal, cliff_list, gamma=1, r=-1,
r_cliff=-100):
        self.w = width
        self.h = height
        self.start = start
        self.goal = goal
        self.cliff_list = cliff_list
        self.gamma = gamma
        self.r = r
        self.r_cliff = r_cliff
        self.grid_list = []
        for i in range(width*height):
            self.grid_list.append(
                Grid(i, is_start=i == start, is_goal=i == goal, is_cliff=i in
cliff_list))
```

We defined the `__str__` function (see the Appendix for details), which can print information about the Cliff Walking environment.

```
Cliff Walking Env:
| 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 |
| S | x | x | x | x | x | x | x | x | x | x | G |
```

Figure 3. Print Cliff Walking Env

We implemented the `epsilon-greedy` function and the `step` function. The former uses the ϵ-greedy algorithm to return an action in state `s`, and the latter returns the reward `r` obtained after taking action `a` in state `s` and the new state `s_next` entered. The numbers `[0,1,2,3]` represent the actions `[^, >, v, <]` respectively.

```
def epsilon_greedy(self, Q, epsilon):
    if random.random() < epsilon:
        return random.randint(0, 3)
    else:
        return Q.index(max(Q))

def step(self, s, a):
    if a == 0:
        s_next = s - self.w if s >= self.w else s
    elif a == 1:
        s_next = s + 1 if (s+1) % self.w != 0 else s
    elif a == 2:
        s_next = s + self.w if s < self.w*(self.h-1) else s
    elif a == 3:
        s_next = s - 1 if s % self.w != 0 else s
    if s_next in self.cliff_list:
```

```
            return self.start, self.r_cliff
        return s_next, self.r
```

3 Sarsa

3.1 Algorithm

Saras is an on-policy TD Alg., since it takes the same behavior policy and target policy. The Alg.'s details is shown below:

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop 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

3.2 Implement

```
def sarsa(self, epsilon=.1, alpha=.2, num_episodes=10000):
    Q = [[0 for _ in range(4)] for _ in range(self.w*self.h)]
    for _ in range(num_episodes):
        s = self.start
        a = self.epsilon_greedy(Q[s], epsilon)
        while not self.grid_list[s].is_goal:
            s_next, r = self.step(s, a)
            a_next = self.epsilon_greedy(Q[s_next], epsilon)
            Q[s][a] += alpha * \
                (r + self.gamma * Q[s_next][a_next] - Q[s][a])
            s = s_next
            a = a_next
    s = self.start
    while s != self.goal:
        self.grid_list[s].act = ['^', '>', 'v', '<'][Q[s].index(max(Q[s]))]
        s, _ = self.step(s, Q[s].index(max(Q[s])))
```

3.3 Result

We set step size `alpha=0.2` and sampled `num_episodes=10000` episodes. Then we tried several values about `epsilon=1, 0.1, 0.001, 0`. The optimal travel path found by Sarsa Alg. is shown below, respectively.

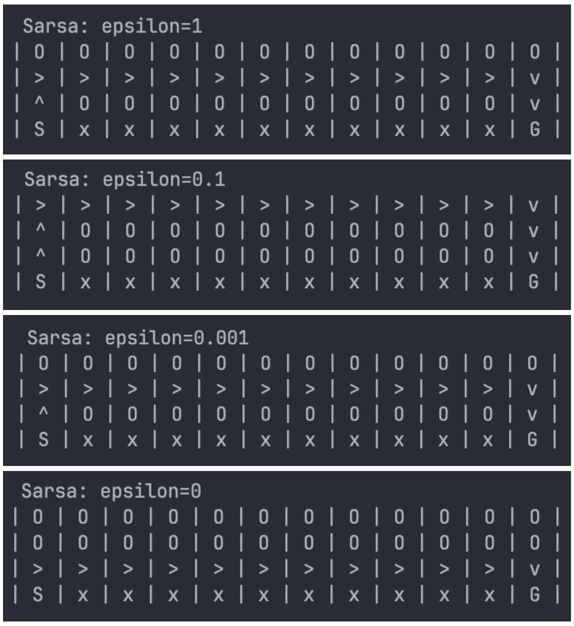

Figure 4. Sarsa with different epsilon values

4 Q-learning

4.1 Algorithm

Q-Learning is an off-policy TD Alg., since it takes the different behavior policy and target policy. The Alg.'s details is shown below:

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop 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

4.2 Implement

```
def q_learning(self, epsilon=.1, alpha=.2, num_episodes=10000):
    Q = [[0 for _ in range(4)] for _ in range(self.w*self.h)]
    for _ in range(num_episodes):
        s = self.start
        while not self.grid_list[s].is_goal:
            a = self.epsilon_greedy(Q[s], epsilon)
            s_next, r = self.step(s, a)
```

```
                Q[s][a] += alpha * \
                    (r + self.gamma * max(Q[s_next]) - Q[s][a])
                s = s_next
        s = self.start
        while s != self.goal:
            self.grid_list[s].act = ['^', '>', 'v', '<'][Q[s].index(max(Q[s]))]
            s, _ = self.step(s, Q[s].index(max(Q[s])))
```

4.3 Result

We set step size `alpha=0.2` and sampled `num_episodes=10000` episodes. Then we tried several values about `epsilon=1, 0.1, 0.001, 0`. The optimal travel path found by Q-Learning Alg. is shown below, respectively.

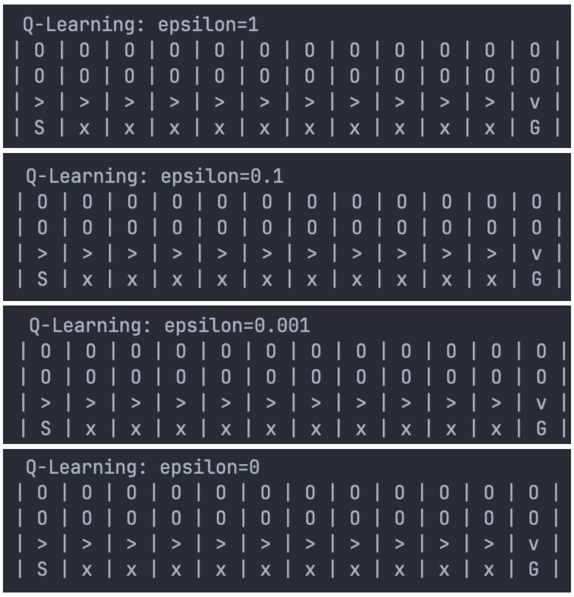

Figure 5. Q-Learning with different epsilon values

5 Conclusion

Comparing the results of the two TD algorithms, we found that:

- When ϵ is not zero, Sarsa Alg. will find a relatively safer travel path then Q-Learning Alg.. On the contrary, the Q-Learning Alg. will find a optimal travel path, but at the same time the agent has the risk of falling off the cliff.

- The change of ϵ will not affect the convergence result of Q-Learning Alg., because ϵ-greedy is not used in the TD-target part of Q-Learning.

- When ϵ is close to zero, the result of the Sarsa Alg. tends to close to the result of the Q-Learning Alg.. Since $\epsilon = 0$ means no exploration, both Alg. will choose the optimal travel path close to the cliff.

- When $\epsilon = 1$, i.e. a completely uniform and random strategy is used when selecting the action of each state. In this case, Sarsa will select the the middle travel path, since the lower path may fall into the cliff (reward=-100), and the upper path may go out of the boundary (reward=-1 for every time wasted), and both will get a negative reward.

A Source Code

```
import random


class Grid:
    def __init__(self, position, value=.0, is_start=False, is_goal=False,
is_cliff=False):
        self.val = value
        self.pos = position
        self.act = None
        self.is_start = is_start
        self.is_goal = is_goal
        self.is_cliff = is_cliff


class Cliff:
    def __init__(self, width, height, start, goal, cliff_list, gamma=1, r=-1,
r_cliff=-100):
        self.w = width
        self.h = height
        self.start = start
        self.goal = goal
        self.cliff_list = cliff_list
        self.gamma = gamma
        self.r = r
        self.r_cliff = r_cliff
        self.grid_list = []
        for i in range(width*height):
            self.grid_list.append(
                Grid(i, is_start=i == start, is_goal=i == goal, is_cliff=i in
cliff_list))

    def __str__(self) -> str:
        grid_str = ''
        for i in range(self.w*self.h):
            if i % self.w == 0:
                grid_str += '| '
            if self.grid_list[i].is_start:
                grid_str += 'S'
            elif self.grid_list[i].is_goal:
                grid_str += 'G'
            elif self.grid_list[i].is_cliff:
                grid_str += 'x'
            elif self.grid_list[i].act == None:
                grid_str += '0'
            else:
                grid_str += self.grid_list[i].act
            if (i+1) % self.w == 0:
                grid_str += ' |\n'
            else:
                grid_str += ' | '
        return grid_str

    def epsilon_greedy(self, Q, epsilon):
        if random.random() < epsilon:
```

```
            return random.randint(0, 3)
        else:
            return Q.index(max(Q))

    def step(self, s, a):
        if a == 0:
            s_next = s - self.w if s >= self.w else s
        elif a == 1:
            s_next = s + 1 if (s+1) % self.w != 0 else s
        elif a == 2:
            s_next = s + self.w if s < self.w*(self.h-1) else s
        elif a == 3:
            s_next = s - 1 if s % self.w != 0 else s
        if s_next in self.cliff_list:
            return self.start, self.r_cliff
        return s_next, self.r

    def sarsa(self, epsilon=.1, alpha=.2, num_episodes=10000):
        Q = [[0 for _ in range(4)] for _ in range(self.w*self.h)]
        for _ in range(num_episodes):
            s = self.start
            a = self.epsilon_greedy(Q[s], epsilon)
            while not self.grid_list[s].is_goal:
                s_next, r = self.step(s, a)
                a_next = self.epsilon_greedy(Q[s_next], epsilon)
                Q[s][a] += alpha * \
                    (r + self.gamma * Q[s_next][a_next] - Q[s][a])
                s = s_next
                a = a_next
        s = self.start
        while s != self.goal:
            self.grid_list[s].act = ['^', '>', 'v', '<'][Q[s].index(max(Q[s]))]
            s, _ = self.step(s, Q[s].index(max(Q[s])))

    def q_learning(self, epsilon=.1, alpha=.2, num_episodes=10000):
        Q = [[0 for _ in range(4)] for _ in range(self.w*self.h)]
        for _ in range(num_episodes):
            s = self.start
            while not self.grid_list[s].is_goal:
                a = self.epsilon_greedy(Q[s], epsilon)
                s_next, r = self.step(s, a)
                Q[s][a] += alpha * \
                    (r + self.gamma * max(Q[s_next]) - Q[s][a])
                s = s_next
        s = self.start
        while s != self.goal:
            self.grid_list[s].act = ['^', '>', 'v', '<'][Q[s].index(max(Q[s]))]
            s, _ = self.step(s, Q[s].index(max(Q[s])))


if __name__ == '__main__':
    eps = .001
    Cliff1 = Cliff(12, 4, start=36, goal=47, cliff_list=range(37, 47))
    print('\n Cliff Walking Env:')
    print(Cliff1)
    print(f'\n Sarsa: epsilon={eps}')
```

```
Cliff1.sarsa(epsilon=eps)
print(Cliff1)
Cliff2 = Cliff(12, 4, start=36, goal=47, cliff_list=range(37, 47))
print(f'\n Q-Learning: epsilon={eps}')
Cliff2.q_learning(epsilon=eps)
print(Cliff2)
```