Reinforcement Learning Assignment 3

Chunlin Chen, 519021910463

March 30, 2023

1 Problem Description

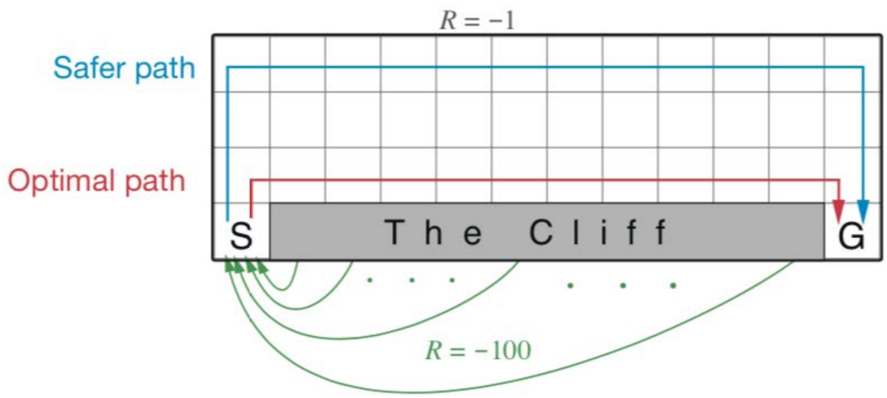


Figure 1: Cliff Walking

Consider the gridworld shown in the Figure 1. This is a standard undiscounted, episodic task, with start state (S), goal state (G), and the usual actions causing movement up, down, right, and left. Reward is -1 on all transitions except those into the region marked “The Cliff”. Stepping into this region incurs a reward of -100 and sends the agent instantly back to the start.

Search the optimal path by implementing the following 2 algorithms:

- Sarsa
- Q-Learning

2 Implementation of Model-free Control Algorithms

2.1 The Cliff Walking Environment

The implementation of the cliff walking environment is similar to that of the normal gridworld environment, and we can implement it in a class like this:

```
class CliffWalking(DiscreteEnv):
    metadata = {'render.modes': ['human', 'ansi']}

    def limit_coordinates(self, coord):
        coord[0] = min(coord[0], self.shape[0] - 1)
        coord[0] = max(coord[0], 0)
        coord[1] = min(coord[1], self.shape[1] - 1)
        coord[1] = max(coord[1], 0)
        return coord

    def calculate_transition_prob(self, current, delta):
        new_position = np.array(current) + np.array(delta)
        new_position = self.limit_coordinates(new_position).astype(int)
        new_state = np.ravel_multi_index(tuple(new_position), self.shape)
        reward = -100.0 if self._cliff[tuple(new_position)] else -1.0
        is_done = self._cliff[tuple(new_position)] or
            (tuple(new_position) == (3,11))
        return [(1.0, new_state, reward, is_done)]

    def __init__(self, shape=(4,12)):
        self.shape = shape

        nS = np.prod(shape)
        nA = 4

        N = 0
        E = 1
        S = 2
        W = 3

        # Cliff Location
        self._cliff = np.zeros(self.shape, dtype=np.bool)
        self._cliff[3, 1:-1] = True

        P = {}
        for s in range(nS):
            position = np.unravel_index(s, self.shape)
            P[s] = { a : [] for a in range(nA) }
            P[s][N] = self.calculate_transition_prob(position, [-1, 0])
            P[s][E] = self.calculate_transition_prob(position, [0, 1])
            P[s][S] = self.calculate_transition_prob(position, [1, 0])
```

```
        P[s][W] = self.calculate_transition_prob(position, [0, -1])

    # Start in (3, 0)
    isd = np.zeros(nS)
    isd[np.ravel_multi_index((3,0), self.shape)] = 1.0

    self.P = P

    super(CliffWalking, self).__init__(nS, nA, P, isd)
```

2.2 Sarsa

We all know that the model-free control uses ϵ-Greedy Exploration for policy improvement, which can be expressed like this:

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

Therefore we first implement the ϵ-greedy policy as shown below:

```
def epsilon_greedy_policy(Q, epsilon, nA):
    def policy_fn(state):
        action_space = np.ones(nA, dtype=float) * epsilon / nA
        optimal_action = np.argmax(Q[state])
        action_space[optimal_action] += (1.0 - epsilon)
        return action_space
    return policy_fn
```

Let us take a look at the pseudocode of the Sarsa algorithm illustrated in Figure 2:

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

Figure 2: The Pseudocode of Sarsa

According to the pseudocode, we can implement the Sarsa algorithm as shown below:

```
def Sarsa(env, num_episodes, learning_rate, epsilon,
    discount_factor=1.0):
   print("Learning rate: {}, Epsilon: {}".format(learning_rate,
       epsilon))
   # Initialize Q
   Q = defaultdict(lambda: np.zeros(env.nA))

   policy = epsilon_greedy_policy(Q, epsilon, env.nA)

   for episode_index in range(1, num_episodes + 1):
      if episode_index % 1000 == 0:
         print("\rEpisode {}/{}.".format(episode_index, num_episodes),
             end="")
         sys.stdout.flush()

      state = env.reset()
      action_prob = policy(state)
      action = np.random.choice(np.arange(len(action_prob)),
          p=action_prob)

      while True:
         next_state, reward, done = env.step(action)

         next_action_prob = policy(next_state)
         next_action =
             np.random.choice(np.arange(len(next_action_prob)),
             p=next_action_prob)

         TD_target = reward + discount_factor *
             Q[next_state][next_action]
         TD_error = TD_target - Q[state][action]
         Q[state][action] += learning_rate * TD_error

         if done:
            break

         state = next_state
         action = next_action

   return Q
```

2.3 Q-Learning

The pseudocode of the Q-Learning method is shown in Figure 3, which is slightly different from Sarsa in the way of updating the action-value function. We can implement the algorithm like this:

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

Figure 3: The Pseudocode of Q-Learning

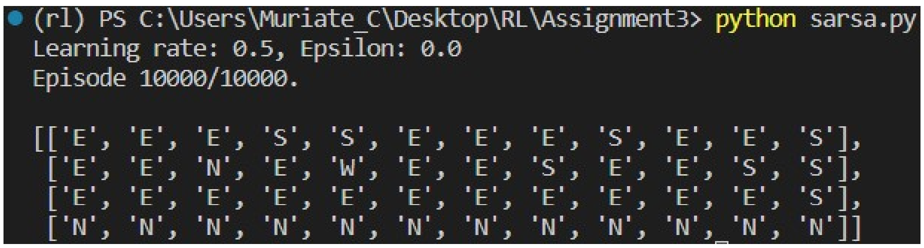
```
def Q_learning(env, num_episodes, learning_rate, epsilon,
     discount_factor=1.0):
    print("Learning rate: {}, Epsilon: {}".format(learning_rate,
         epsilon))
    # Initialize Q
    Q = defaultdict(lambda: np.zeros(env.nA))

    policy = epsilon_greedy_policy(Q, epsilon, env.nA)

    for episode_index in range(1, num_episodes + 1):
        if episode_index % 1000 == 0:
            print("\rEpisode {}/{}.".format(episode_index, num_episodes),
                 end="")
            sys.stdout.flush()

        state = env.reset()

        while True:
            action_prob = policy(state)
            action = np.random.choice(np.arange(len(action_prob)),
                 p=action_prob)
            next_state, reward, done = env.step(action)

            optimal_next_action = np.argmax(Q[next_state])
            TD_target = reward + discount_factor *
                 Q[next_state][optimal_next_action]
            TD_error = TD_target - Q[state][action]
            Q[state][action] += learning_rate * TD_error

            if done:
                break

            state = next_state
```

```python
    return Q
```

3 Results Analysis

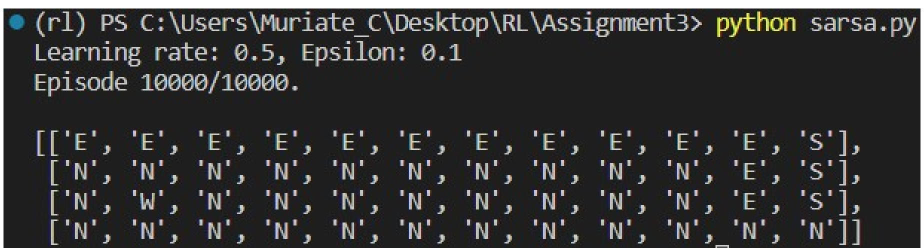
3.1 Sarsa

We set the number of episodes to be 10000 and the learning rate α to be 0.5, and we tested the result under different ϵ, and the optimal path derived from the final Q is shown in Figure 4 and Figure 5.



```
(rl) PS C:\Users\Muriate_C\Desktop\RL\Assignment3> python sarsa.py
Learning rate: 0.5, Epsilon: 0.0
Episode 10000/10000.

[['E', 'E', 'E', 'S', 'S', 'E', 'E', 'E', 'S', 'E', 'E', 'S'],
 ['E', 'E', 'N', 'E', 'W', 'E', 'E', 'S', 'E', 'E', 'S', 'S'],
 ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'S'],
 ['N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N']]
```

Figure 4: The optimal path when $\epsilon = 0$



```
(rl) PS C:\Users\Muriate_C\Desktop\RL\Assignment3> python sarsa.py
Learning rate: 0.5, Epsilon: 0.1
Episode 10000/10000.

[['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'S'],
 ['N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'E', 'S'],
 ['N', 'W', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'E', 'S'],
 ['N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N']]
```

Figure 5: The optimal path when $\epsilon = 0.1$

We can find that when $\epsilon = 0$, the Sarsa algorithm produces the optimal path, which is the shortest path towards the target without stepping into the cliff; while the algorithm tends to produce a safer path when $\epsilon = 0.1$.

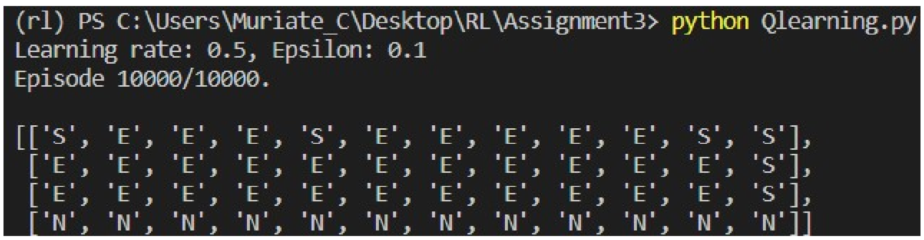
3.2 Q-Learning

Same as above, we set the number of episodes to be 10000 and the learning rate α to be 0.5, and we tested the result under different ϵ, and the optimal path derived from the final Q is shown in Figure 6 and Figure 7.

We can find that the Q-Learning algorithm both produces the shortest path when $\epsilon = 0.1$ and $\epsilon = 0$.

```
(rl) PS C:\Users\Muriate_C\Desktop\RL\Assignment3> python Qlearning.py
Learning rate: 0.5, Epsilon: 0.0
Episode 10000/10000.

[['N', 'N', 'E', 'E', 'N', 'E', 'S', 'W', 'S', 'S', 'S', 'S'],
 ['E', 'N', 'W', 'E', 'E', 'E', 'E', 'E', 'S', 'E', 'E', 'S'],
 ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'S'],
 ['N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N']]
```

Figure 6: The optimal path when $\epsilon = 0$



```
(rl) PS C:\Users\Muriate_C\Desktop\RL\Assignment3> python Qlearning.py
Learning rate: 0.5, Epsilon: 0.1
Episode 10000/10000.

[['S', 'E', 'E', 'E', 'S', 'E', 'E', 'E', 'E', 'E', 'S', 'S'],
 ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'S'],
 ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'S'],
 ['N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N']]
```

Figure 7: The optimal path when $\epsilon = 0.1$

3.3 Conclusion

In the experiment, we find that Sarsa tends to choose the safer path while Q-Learning tends to choose the optimal path.