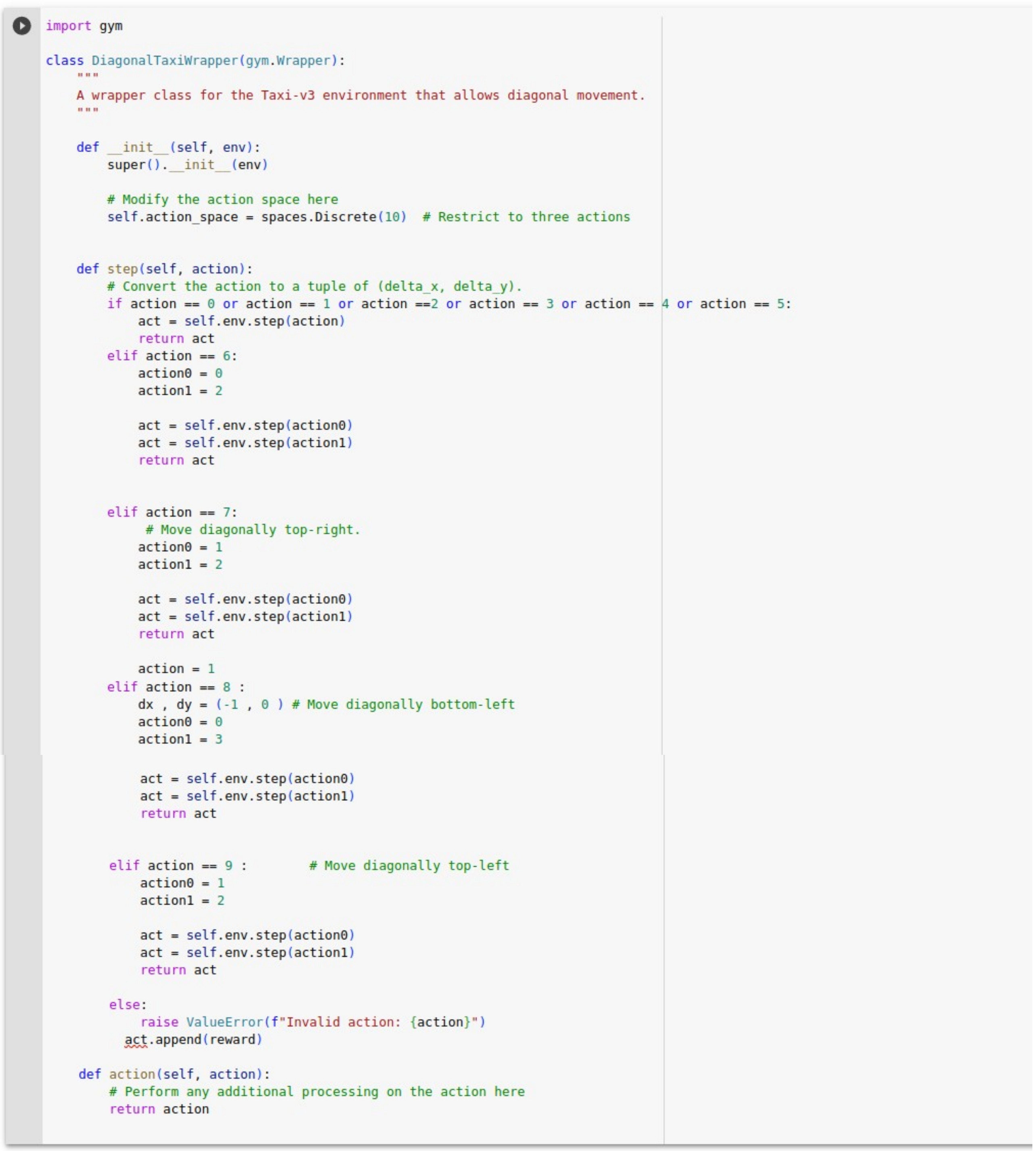
In The Name Of God

Reinforcement Learning   Second Project   Soroush Naseri

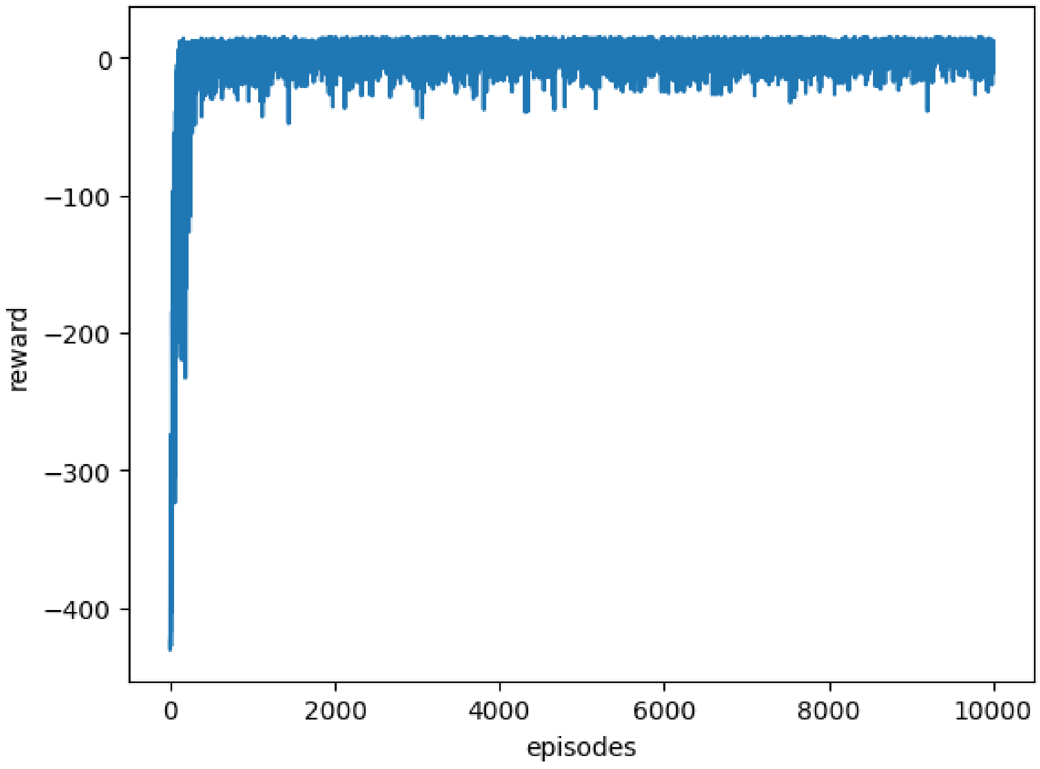
There are several ways to write a wrapper that has diagonal paths. The method we use in this code is to use two step functions, one to go up or down and the other to go straight. In general, one of the ideas that can be used to go from one side of the table to another is to move the state to the state next to the target state and then call the step function in such a way as to reach the final state. Let's write the wrapper code below according to the purpose of the change.



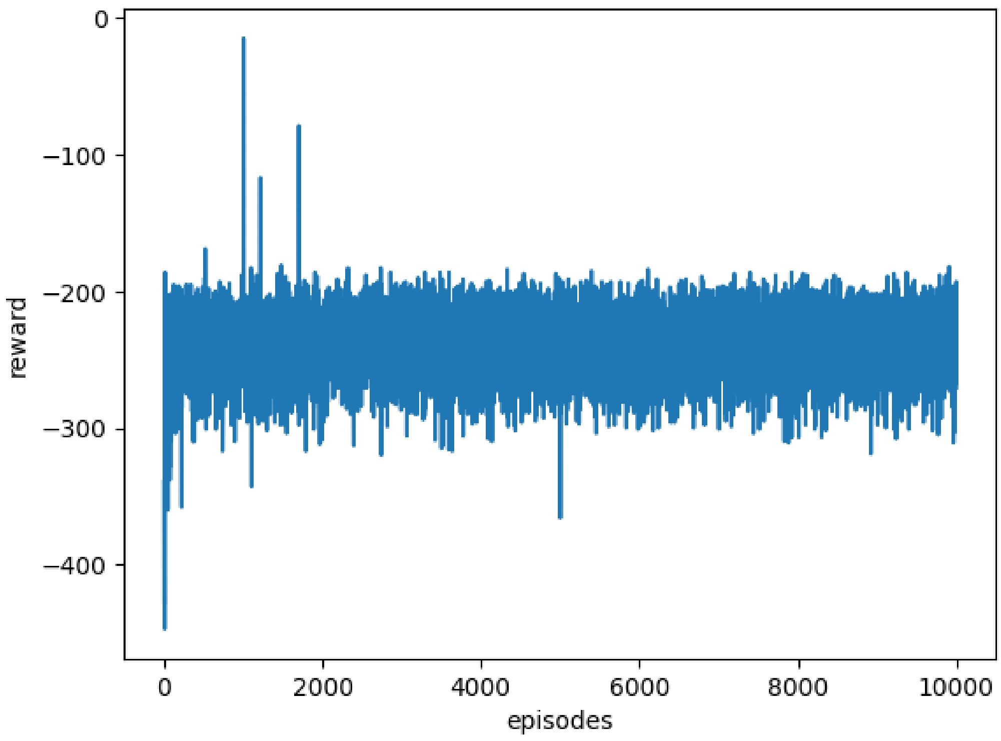
```python
import gym

class DiagonalTaxiWrapper(gym.Wrapper):
    """
    A wrapper class for the Taxi-v3 environment that allows diagonal movement.
    """

    def __init__(self, env):
        super().__init__(env)

        # Modify the action space here
        self.action_space = spaces.Discrete(10)  # Restrict to three actions


    def step(self, action):
        # Convert the action to a tuple of (delta_x, delta_y).
        if action == 0 or action == 1 or action ==2 or action == 3 or action == 4 or action == 5:
            act = self.env.step(action)
            return act
        elif action == 6:
            action0 = 0
            action1 = 2

            act = self.env.step(action0)
            act = self.env.step(action1)
            return act


        elif action == 7:
              # Move diagonally top-right.
            action0 = 1
            action1 = 2

            act = self.env.step(action0)
            act = self.env.step(action1)
            return act

            action = 1
        elif action == 8 :
            dx , dy = (-1 , 0 ) # Move diagonally bottom-left
            action0 = 0
            action1 = 3

            act = self.env.step(action0)
            act = self.env.step(action1)
            return act


        elif action == 9 :        # Move diagonally top-left
            action0 = 1
            action1 = 2

            act = self.env.step(action0)
            act = self.env.step(action1)
            return act

        else:
            raise ValueError(f"Invalid action: {action}")
          act.append(reward)

    def action(self, action):
        # Perform any additional processing on the action here
        return action
```

First, we examine the Q-learning algorithm.

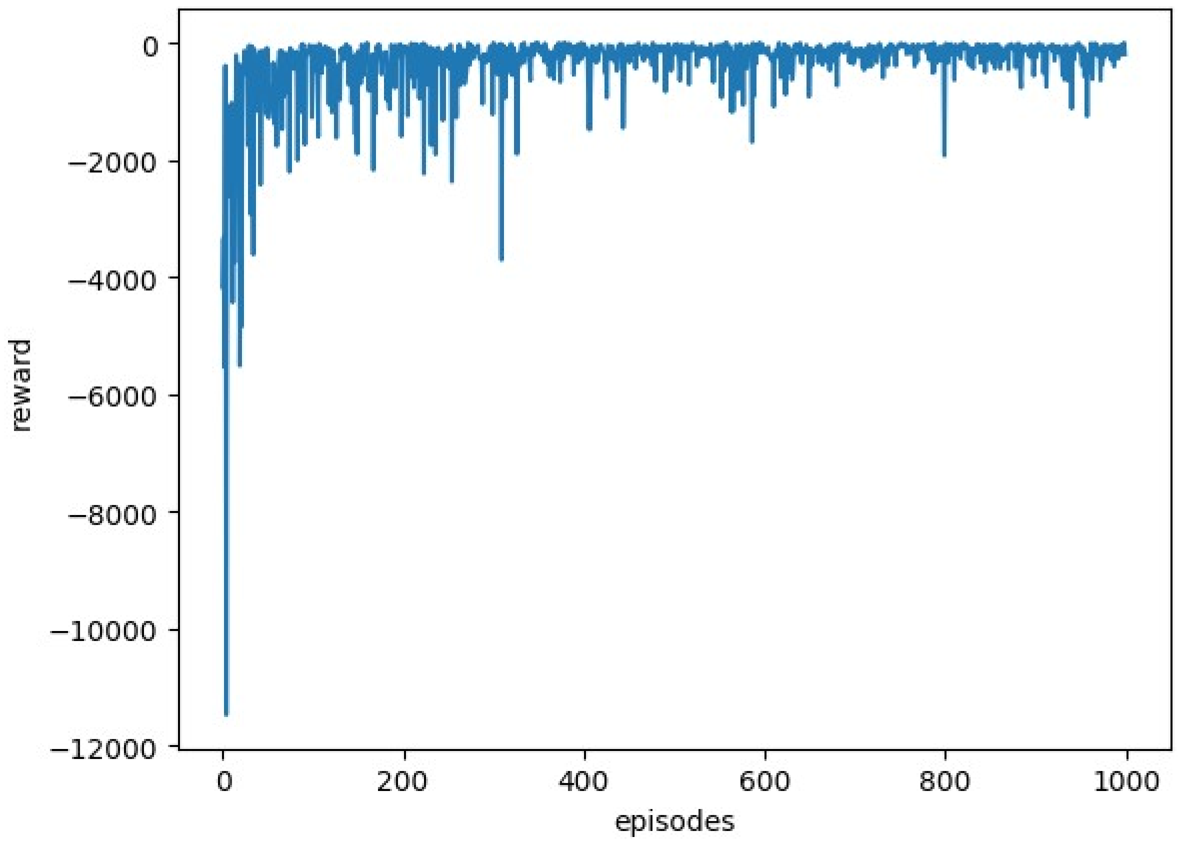
Consider that the gamma is equal to 0.9 :



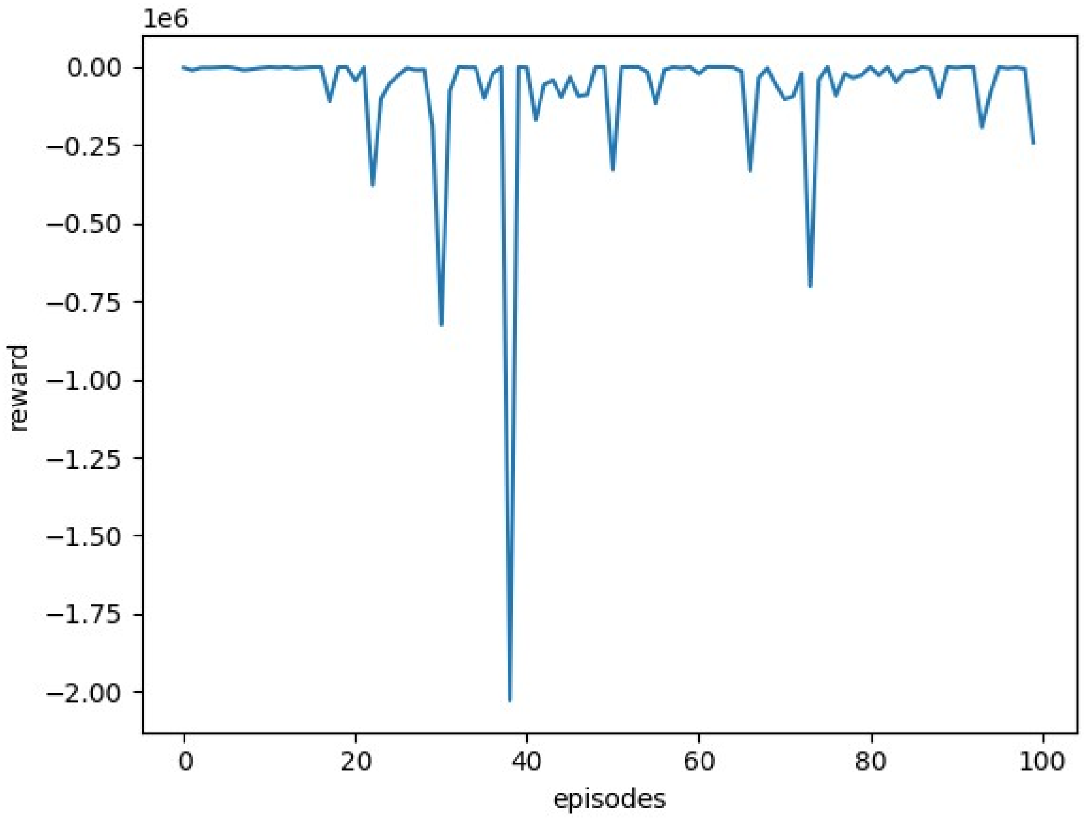
and for gamma = 0 :



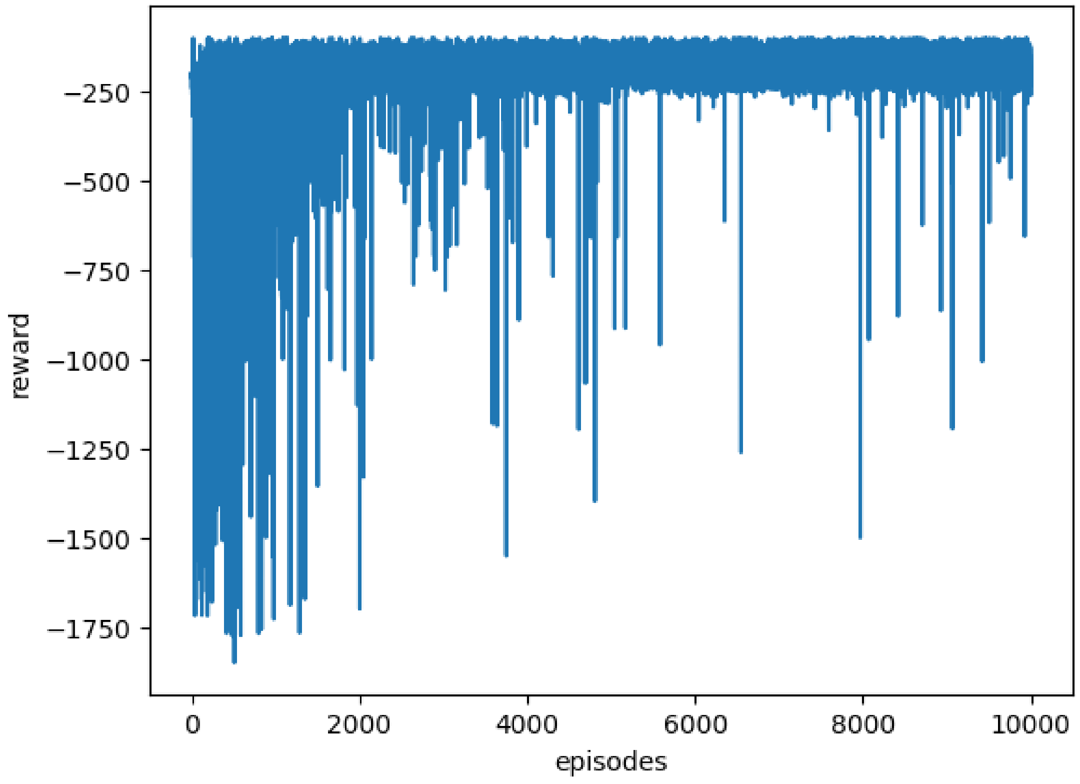
Second, we examine SARSA algorithm :
First consider that gamma = 0.9 :


0
−2000
−4000
−6000
−8000
−10000
−12000
reward
0
200
400
600
800
1000
episodes

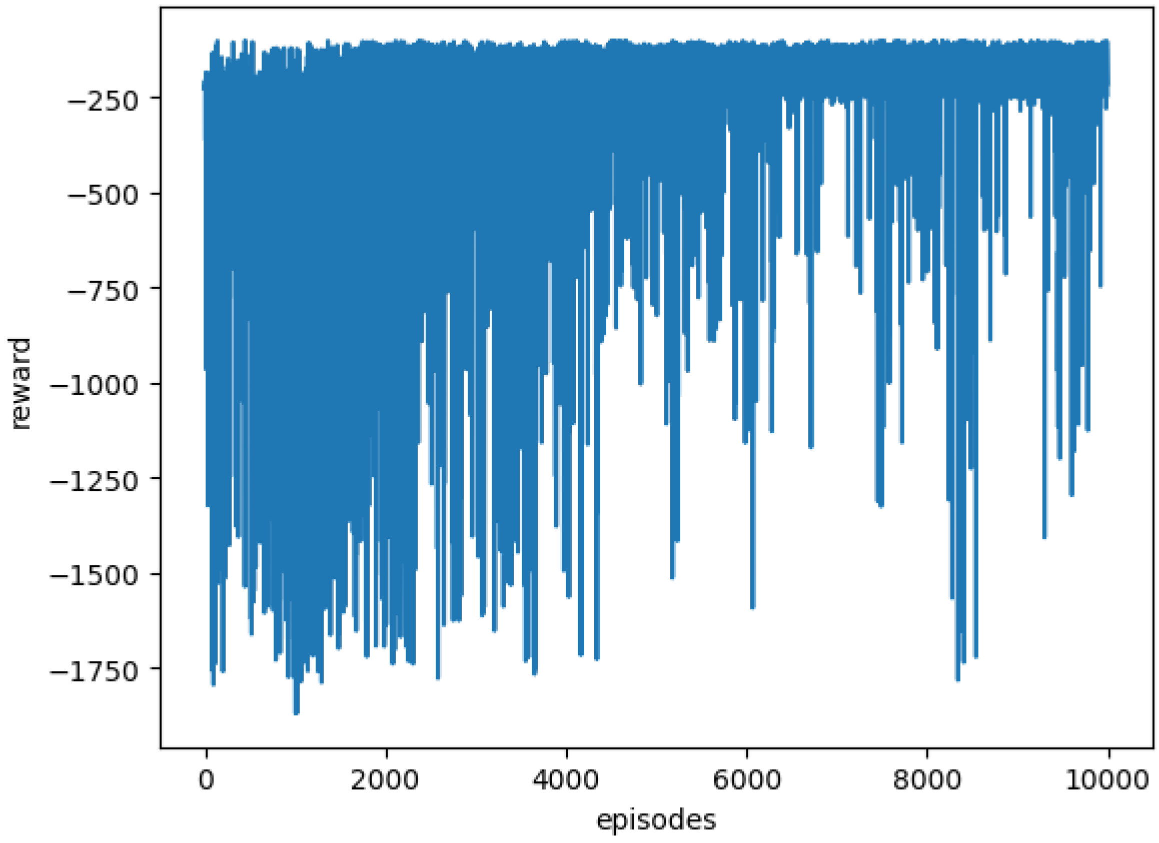
for gamma = 0 :


1e6
0.00
−0.25
−0.50
−0.75
−1.00
−1.25
−1.50
−1.75
−2.00
reward
0
20
40
60
80
100
episodes

And finally for Monte Carlo algorithm, with gamma = 0.9 and 10000 iterations :



and for gamma = 0 :



If we want to point out the differences between these two algorithms, in the Q-Learning algorithm, which is an offline algorithm, in fact, we use another publicity for the target, and it is in this way that the highest q value of the next house is considered as the target. We take. But in the sarsa algorithm, which is an online algorithm, both policies are the same. You can see its below :
Q-Learning :

$$Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$$

and Sarsa :

$$Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma Q(S', A') - Q(S, A)\right]$$

If we pay attention to the code, at first sight it seems that the SARSA algorithm is more cautious because it works realistically while updating, but Q-Learning actually works optimistically in a way, and this may cause it to take more risks.