CS489 Assignment 3 Report

517030910214 Hongzhou Liu

0. Introduction

In this assignment, we are required to implement SARSA and Q-Learning in CliffWalking environment. CliffWalking is a variant of GridWorld. In CliffWalking, the start state is specified and there's a cliff lying between start state and goal state. Once our agent moves onto the cliff, it slips directly to the start state and receives a reward of -100. In other case, a move will incur a reward of -1.

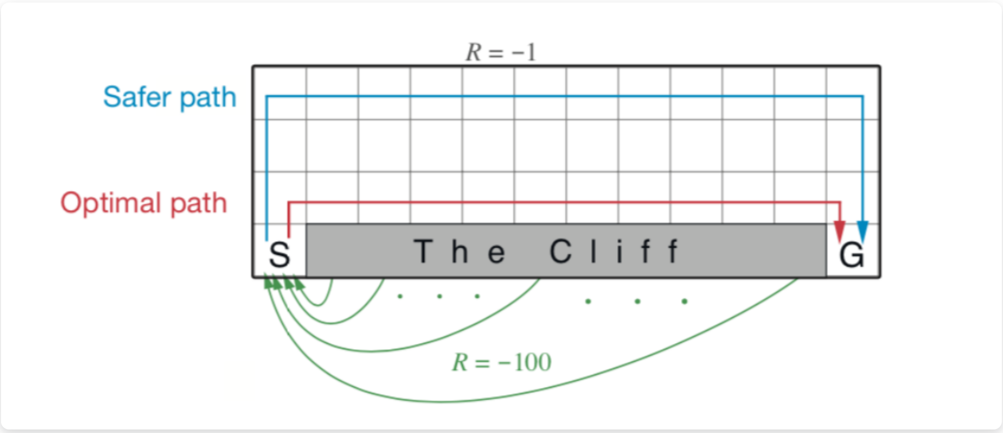


Fig.1 CliffWalking

Environment:

- Ubuntu 18.04 LTS
- Python 3.7.7

It will take a few changes on our original GridWorld to convert it into CliffWalking. As the following code shows, once the agent step onto the cliff, it will be instantly sent to the start state and receives a reward of -100.

```
or cliffState in range(self.startState + 1, self.goalState):
        edgeState = self.trans[cliffState][self.action['n']]
        self.trans[edgeState][self.action['s']] = self.startState
        self.reward[edgeState][self.action['s']] = -100.0
    self.trans[self.startState][self.action['e']] = self.startState
    self.reward[self.startState][self.action['e']] = -100.0
```

1. SARSA

1.1 Implementation

SARSA is a kind of on-policy TD learning algorithm. When doing policy improvement, we cannot consider all the states because TD is model free. Thus we have to use action value to evaluate policy. In SARSA, we need to modify TD learning. We should generate action from ϵ-soft policy and update action value by

$$Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma Q(S',A') - Q(S,A))$$

where A is the chosen action, S' and R are the next state and reward when taking A under S, A' is another action chosen under S'.

Firstly, we need to implement a method that calculates ϵ-soft policy and a method that generates an action from the policy.

The ϵ-soft policy is defined as

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

Thus, we can implement it easily as:

```
def __updatePolicy(self, s, epsilon):
  optimalAction = np.argmax(self.QValue[s])
  for a in self.action.values():
    if a != optimalAction:
      self.policy[s][a] = epsilon / len(self.action)
    else:
      self.policy[s][a] = 1 - epsilon + epsilon / len(self.action)
```

In SARSA, we need to generate an action A from S using ϵ-soft policy derived from Q, which is totally different from generate an action from uniform random policy. In `__actionGenerator`, we divide $[0, 1)$ into four intervals. The length of each interval corresponds to the policy value of state s and action a. `accProb` is an array that stores the edge values of intervals. Then, we just need to throw a stone (`prob`) into the line $[0, 1)$ and output the interval it falls into.

```
def __actionGenerator(self, s, epsilon):
  prob = random.random()
  accProb = [sum(self.policy[s][:i]) for i in range(len(self.action) + 1)]
  for i in range(len(self.action)):
    if prob >= accProb[i] and prob < accProb[i + 1]:
      a = i
  return a
```

At last, we can implement SARSA, the code is shown as

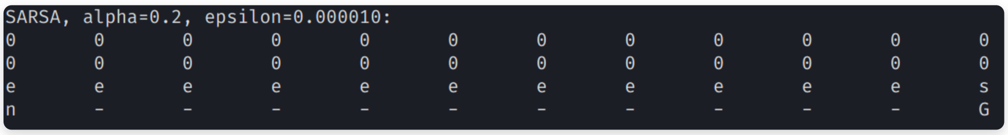
```
  def SARSA(self, alpha, epsilon, iterTimes):
    #Initialize
    self.QValue = [[random.random() for j in range(len(self.action))] for i in range(self.state)]
    self.policy = [[0.0 for j in range(len(self.action))] for i in range(self.state)]
    for a in self.action.values():
      self.QValue[self.goalState][a] = 0.0
    for s in range(self.state):
      self.__updatePolicy(s, epsilon)

    #Iteration
    for _ in range(iterTimes):
      curState = self.startState
      curAction = self.__actionGenerator(curState, epsilon)
      while curState != self.goalState:
        nextState = self.trans[curState][curAction]
        curReward = self.reward[curState][curAction]
        nextAction = self.__actionGenerator(nextState, epsilon)
        self.QValue[curState][curAction] = self.QValue[curState][curAction] + alpha * (curReward + self.gamma * self.QValue[nextState][nextAction] - self.QValue[curState][curAction])
        self.__updatePolicy(curState, epsilon)
        curState = nextState
        curAction = nextAction
```

In the initialize part, we should initialize Q-value and generate a policy from it. Then, the code follows the SARSA algorithm to update Q-value. Also, we should always update our policy after the update of Q-value.

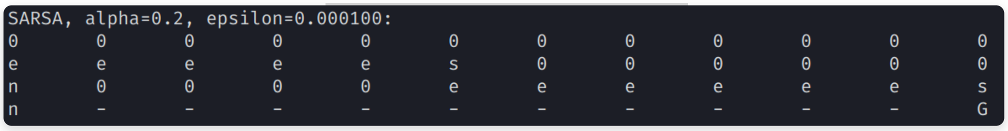
1.2 Result

We run SARSA with different ϵ. The result differs when we feed it in with different ϵ.



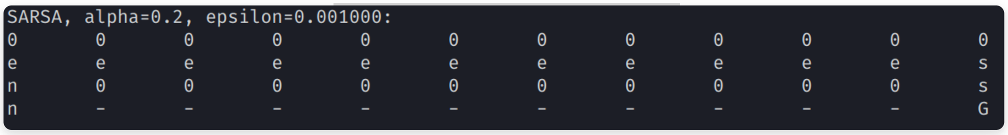
```
SARSA, alpha=0.2, epsilon=0.000010:
0       0       0       0       0       0       0       0       0       0       0       0
0       0       0       0       0       0       0       0       0       0       0       0
e       e       e       e       e       e       e       e       e       e       e       s
n       -       -       -       -       -       -       -       -       -       -       G
```

Fig.2 SARSA, epsilon=0.00001



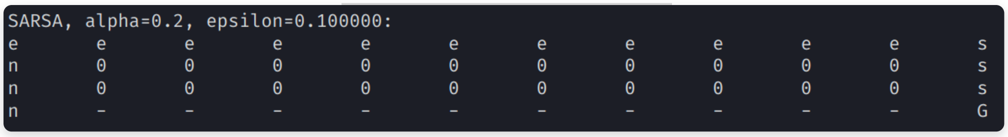
```
SARSA, alpha=0.2, epsilon=0.000100:
0       0       0       0       0       0       0       0       0       0       0       0
e       e       e       e       e       s       0       0       0       0       0       0
n       0       0       0       0       e       e       e       e       e       e       s
n       -       -       -       -       -       -       -       -       -       -       G
```

Fig.3 SARSA, epsilon=0.0001



```
SARSA, alpha=0.2, epsilon=0.001000:
0       0       0       0       0       0       0       0       0       0       0       0
e       e       e       e       e       e       e       e       e       e       e       s
n       0       0       0       0       0       0       0       0       0       0       s
n       -       -       -       -       -       -       -       -       -       -       G
```

Fig.4 SARSA, epsilon=0.001



```
SARSA, alpha=0.2, epsilon=0.100000:
e       e       e       e       e       e       e       e       e       e       e       s
n       0       0       0       0       0       0       0       0       0       0       s
n       0       0       0       0       0       0       0       0       0       0       s
n       -       -       -       -       -       -       -       -       -       -       G
```

Fig.5 SARSA, epsilon=0.1

As we can see, when ϵ is large, the agent will choose a safe path, i.e. a farther path from the cliff, to approach the goal in spite of getting less reward. However, as ϵ decreases, the agent will choose a path which is more dangerous. It is because, when ϵ is small, the ϵ-greedy policy improvement will limit the probability of observation of other states and force the agent to choose the optimal path. And when ϵ is large, the agent may choose those non-optimal paths.

2. Q-Learning

2.1 Implementation

Q-Learning is a kind of off-policy learning algorithm. There are a target policy we want to learn about and a behavior policy the agent follows in order to explore the environment. The term off-policy means the agent learn from data off the target policy.

In Q-Learning, we will improve a target policy π greedily and follow a behavior policy μ, which is ϵ-greedy. It means the agent will choose an action A following the behavior policy. When it comes to update $Q(S, A)$ value, we do it greedily by

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

where S' is the state after S taking action A. Actually, we choose A^* greedily by $A^* = \arg\max_a Q(S', a)$, however, $Q(S, A^*)$ equals to $\max_a Q(S, a)$. Thus we directly use $\max$ when updating $Q(S, A)$.

Base on `__updatePolicy` and `actionGenerator`, we can implement Q-Learning as

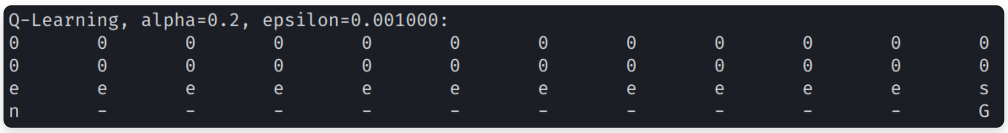
```python
def Q_Learning(self, alpha, epsilon, iterTimes):
    #Initialize
    self.QValue = [[random.random() for j in range(len(self.action))] for i in range(self.state)]
    self.policy = [[0.0 for j in range(len(self.action))] for i in range(self.state)]
    for a in self.action.values():
        self.QValue[self.goalState][a] = 0.0
    for s in range(self.state):
        self.__updatePolicy(s, epsilon)

    #Iteration
    for _ in range(iterTimes):
        curState = self.startState
        while curState != self.goalState:
            curAction = self.__actionGenerator(curState, epsilon)
            nextState = self.trans[curState][curAction]
            curReward = self.reward[curState][curAction]
            self.QValue[curState][curAction] = self.QValue[curState][curAction] + alpha * (curReward + self.gamma * max(self.QValue[nextState]) - self.QValue[curState][curAction])
            self.__updatePolicy(curState, epsilon)
            curState = nextState
```

The initialize part is the same as SARSA. In the iteration part, we choose action ϵ-greedily using `__actionGenerator` and update Q value greedily just using `max`.

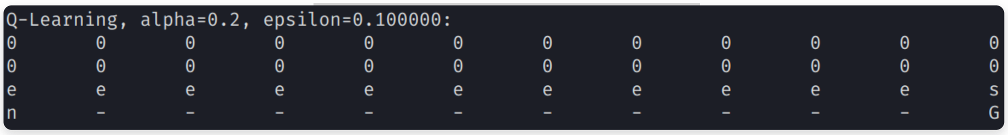
2.2 Result

As we can see, no matter what value ϵ is, the Q-Learning will find the optimal way to approach the goal. The power of off-policy learning reveals.



```
Q-Learning, alpha=0.2, epsilon=0.001000:
0        0        0        0        0        0        0        0        0        0        0        0
0        0        0        0        0        0        0        0        0        0        0        0
e        e        e        e        e        e        e        e        e        e        e        s
n        -        -        -        -        -        -        -        -        -        -        G
```

Fig.6 Q-Learning, epsilon=0.001



```
Q-Learning, alpha=0.2, epsilon=0.100000:
0        0        0        0        0        0        0        0        0        0        0        0
0        0        0        0        0        0        0        0        0        0        0        0
e        e        e        e        e        e        e        e        e        e        e        s
n        -        -        -        -        -        -        -        -        -        -        G
```

Fig.7 Q-Learning, epsilon=0.1