Report of Assignment 2

717030210005 Zhexin Jin

1. Environment

```
class Grid:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.value = 0.0
        self.next_value = 0.0
        self.cnt = 0
        self.next_cnt = 0
        self.sum = 0.0
        self.next_sum = 0.0
        if [self.x, self.y] == [0, 0] or [self.x, self.y] == [3, 3]:
            self.action = []
        else:
            self.action = [(1, 0), (0, 1), (-1, 0), (0, -1)]
        self.reward = -1.0
        self.policy = self.action
        self.gamma = 0.5
```

I use a class named Grid to describe every grid in gridworld which is a 4*4 array.

This class has some parameters. Some of them have been explained in last report, and some are new. In MC methods, the equation value = sum / cnt is always satisfied.

This class have some methods:

```
    def updateValue(self):
        self.value = self.next_value
        self.sum = self.next_sum
        self.cnt = self.next_cnt
```

The function above will set the value of the next state to the current state.

```
    def updatePolicy(self, gridworld):
        max_value = -float("inf")
        self.policy = []
        for i in range(len(self.action)):
            next_x = self.x + self.action[i][0]
            next_y = self.y + self.action[i][1]
            if next_x > 3 or next_x < 0 or next_y > 3 or next_y < 0:
                next_x, next_y = self.x, self.y
            if round(gridworld[next_x][next_y].value, 1) == round(max_value, 1):
                self.policy.append(self.action[i])
            elif round(gridworld[next_x][next_y].value, 1) > round(max_value, 1):
                self.policy = [self.action[i]]
                max_value = gridworld[next_x][next_y].value
```

The function above is used to update the policy to find the movement to the grid nearby which has the minimal value.

```
    def nextGrid(self):
        i = randint(0,len(self.action)-1)
        next_x = self.x + self.action[i][0]
        next_y = self.y + self.action[i][1]
        if next_x > 3 or next_x < 0 or next_y > 3 or next_y < 0:
            next_x, next_y = self.x, self.y
        return next_x,next_y
```

The function above can return the coordinates of a nearby grid randomly according to the policy.

2. First-Visit MC

```
def firstVisitMC():
    # init gridworld
    gridworld = [[Grid(i, j) for j in range(4)] for i in range(4)]

    for k in range(100000):
        # generate episode
        episode = [gridworld[randint(0, 3)][randint(0, 3)]]
        generateEpisode(gridworld,episode)

        # next value
        for i in range(4):
            for j in range(4):
                if gridworld[i][j] in episode:
                    nextValueMC(gridworld,episode,episode.index(gridworld[i][j]))

        # update value
        for i in range(4):
            for j in range(4):
                gridworld[i][j].updateValue()

    # update policy

    for i in range(4):
```

```
        for j in range(4):
            gridworld[i][j].updatePolicy(gridworld)

    # show gridworld
    for i in range(4):
        print([round(gridworld[i][j].value,1) for j in range(4)])

    # show policy
    for i in range(4):
        print([gridworld[i][j].policy for j in range(4)])
```

Process:

1. Initialize gridworld.
2. Generate an episode.
3. Calculate next value using first-visit MC methods. (The function nextValueMC() will be explained later.)
4. Update value.
5. Go to 2. Repeat 100000 times.
6. Update policy.
7. Show value of every grid.
8. Show final policy.

```
def nextValueMC(gridworld,episode,idx):
    grid = episode[idx]
    temp_episode = episode[idx+1:]
    for i in range(0,len(temp_episode)):
        grid.next_sum += temp_episode[i].reward * temp_episode[i].gamma**i
    grid.next_cnt += 1
    grid.next_value = grid.next_sum / grid.next_cnt
```

The function above will update the next_sum, next_cnt and next_value of No.idx grid in the episode.

3. Every-Visit MC

```
def everyVisitMC():
    # init gridworld
    gridworld = [[Grid(i, j) for j in range(4)] for i in range(4)]

    for k in range(100000):
        # generate episode
        episode = [gridworld[randint(0, 3)][randint(0, 3)]]
        generateEpisode(gridworld,episode)

        # next value
        for idx in range(len(episode)):
            nextValueMC(gridworld,episode,idx)

        # update value
        for i in range(4):
            for j in range(4):
                gridworld[i][j].updateValue()
```

```
    # update policy
    for i in range(4):
        for j in range(4):
            gridworld[i][j].updatePolicy(gridworld)

    # show gridworld
    for i in range(4):
        print([round(gridworld[i][j].value,1) for j in range(4)])

    # show policy
    for i in range(4):
        print([gridworld[i][j].policy for j in range(4)])
```

The process is similar to firstVisitMC().

The only change is the part of calculating next value.

It will calculate next value for every grid in episode, and some of them will be calculate more than once.

4. TD(0)

```
def TD0():
    # init gridworld
    gridworld = [[Grid(i, j) for j in range(4)] for i in range(4)]

    for k in range(100000):
        # generate episode
        episode = [gridworld[randint(0, 3)][randint(0, 3)]]
        generateEpisode(gridworld,episode)

        # next value
        for idx in range(len(episode)-1):
            nextValueTD(gridworld,episode,idx)

        # update value
        for i in range(4):
            for j in range(4):
                gridworld[i][j].updateValue()

    # update policy
    for i in range(4):
        for j in range(4):
            gridworld[i][j].updatePolicy(gridworld)

    # show gridworld
    for i in range(4):
        print([round(gridworld[i][j].value,1) for j in range(4)])

    # show policy
    for i in range(4):
        print([gridworld[i][j].policy for j in range(4)])
```

The process is similar to everyVisitMC(), but in the part of calculating next value, I replace nextValueCM() with nextValueTD().

```
def nextValueTD(gridworld,episode,idx):
    grid = episode[idx]
    next_grid = episode[idx+1]
    grid.next_value += 0.0001 * (next_grid.reward + next_grid.gamma * next_grid.value -
grid.value)
```

5. Conclusion

```
def main():
    print("first-visit MC:")
    firstVisitMC()
    print()
    print("every-visit MC:")
    everyVisitMC()
    print()
    print("TD(0):")
    TD0()
    print()
```

I have tested every method, and each method took about 10 seconds.

Finally, they all converged to the same situation. (I don't know if it's a coincidence or not.)

0.0 -1.7 -1.9 -2.0

-1.7 -1.9 -2.0 -1.9

-1.9 -2.0 -1.9 -1.7

-2.0 -1.9 -1.7 0.0

We can easily see the final policy.