Assignment 1 Report

717030210005 Zhexin Jin

1. Environment

First, initialize gridworld which is like a 4*4 array.

```
gridworld = [[Grid(i,j) for j in range(4)] for i in range(4)]
```

The element in gridworld is class "Grid".

Here is the definition of class Grid.

```
class Grid:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.state = 0.0
        self.next_state = 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.action_p = [0.25, 0.25, 0.25, 0.25]
        self.policy = self.action
        self.reward = -1
```

Self.x and self.y are the coordinates of this grid in gridworld.

Self.state denotes the state at this grid.

Self.next_state denotes the next state at this grid which will be used to judge the stability of the state.

Self.action contains the four movements: (1,0) : east; (0,1) : south; (-1,0) : west; (0,-1):north.

Self.action_p denotes the probabilities of actions.

Self.policy contains the possible movements according to the policy.

Self.reward is always -1 which is the reward of each movements.

This class have some methods:

```
def nextState(self, gridworld):
    self.next_state = 0
    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
        self.next_state += self.action_p[i] * (self.reward + gridworld[next_x]
[next_y].state)
```

The function above is used to calculate the next state of this grid.

```
def updateState(self):
    self.state = self.next_state
```

The function above is used to update the state.

2. Policy

```
def updatePolicy(self, gridworld):
    max_state = -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].state, 3) == round(max_state, 3):
            self.policy.append(self.action[i])
        elif round(gridworld[next_x][next_y].state, 3) > round(max_state, 3):
            self.policy = [self.action[i]]
            max_state = gridworld[next_x][next_y].state
```

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

3. Process

```
# init gridworld
gridworld = [[Grid(i,j) for j in range(4)] for i in range(4)]
stable = False
k = 1

while not stable:
    stable = True
    # calculate nextstate
    for i in range(4):
        for j in range(4):
            gridworld[i][j].nextState(gridworld)

    # judge stable
```

```
    for i in range(4):
        for j in range(4):
            if abs(gridworld[i][j].next_state - gridworld[i][j].state) > 0.01:
                stable = False

    # update state
    for i in range(4):
        for j in range(4):
            gridworld[i][j].updateState()

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

    print("k =",k)
    k += 1

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

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

    # enter to continue
    input()
```

The comments should be clear enough to describe the process.