CSE 584 HW2 (Yucheng Zhou)

The code I will discuss here is from this site: https://github.com/SS-YS/MDP-with-Value-Iteration-and-Policy-Iteration/blob/main/valueIteration.py

1. This code implements the value iteration algorithm in an MDP as discussed in our class. And it aims to solve a 3x4 grid world example by finding the optimal policy to maximize the expected cumulative reward. The problem here is defined with states, actions, rewards, and transition probabilities as follows: The actions can be moving left, right, up, or down from a state. The intended action occurs with probability 0.8, but with probability 0.2 the agent moves at right angles to the intended direction. If colliding with the wall, it will stay at the same state. There are two terminal states with reward +1 and -1 respectively, and all other states have a constant reward (e.g. -0.01). In the code, it first defines hyperparameters like reward, discount, actions, initial values of the states. And the initial environment is visualized. Then it starts to update the values according to the value iteration algorithm, and finally obtains and prints the optimal policy as desired. The technical details will be discussed in the following point.

2. The core technical reinforcement learning section in the code is the following 4 functions. And I have added comments for each line of them.

```
#Get the value of the state reached by performing the given action from the given state
def getU(U, r, c, action):
    dr, dc = ACTIONS[action]   # get the action from the action list following the input action index
    newR, newC = r+dr, c+dc # get the new state position after performing the action
    if newR < 0 or newC < 0 or newR >= NUM_ROW or newC >= NUM_COL or (newR == newC == 1): # if collide with the boundary or the wall
        return U[r][c] # it will not move, the value is just at (r, c)
    else:
        return U[newR][newC] # return the value at the new state position

# Calculate the value of a state given an action
def calculateU(U, r, c, action):
    u = REWARD # initialize value u to add reward at first
    u += 0.1 * DISCOUNT * getU(U, r, c, (action-1)%4)
    # according the the Bellman formula, add the transition probability*discount*value of next state, here 0.1 follows the action at right angles to the intended direction.
    u += 0.8 * DISCOUNT * getU(U, r, c, action)
```

```
        # here 0.8 follows the intended direction.
        u += 0.1 * DISCOUNT * getU(U, r, c, (action+1)%4)
        # here 0.1 follows the action at right angles to the intended direction.
        return u # return the summed values for later use.

def valueIteration(U): # the function of value iteration
        # print("During the value iteration:\n")
        while True: # Enter the loop to do the value iteration
                nextU = [[0, 0, 0, 1], [0, 0, 0, -1], [0, 0, 0, 0], [0, 0, 0, 0]] # initialize the values
                error = 0    # initialize the error to determine stop condition
                for r in range(NUM_ROW):    # for each row
                        for c in range(NUM_COL): # for each column, we identify each state
                                if (r <= 1 and c == 3) or (r == c == 1):
                                        # we skip the terminal states and the state that is not achievable
                                        continue    # we skip and loop to the next item
                                nextU[r][c] = max([calculateU(U, r, c, action) for action in
range(NUM_ACTIONS)])
                                # update values according to the Bellman formula, taking the max
of it
                                error = max(error, abs(nextU[r][c]-U[r][c]))
                                # keep the largest error by comparing the updated value of this
iteration with the previous value
                U = nextU # we assign the new value table as the U (original table)
                # Then we check the iteration stop condition. If the error between each step
is small enough, we will stop the loop.
                if error < MAX_ERROR * (1-DISCOUNT) / DISCOUNT:
                        break
        return U #Output the optimal converged values.

# Get the optimal policy from U
def getOptimalPolicy(U):
        policy = [[-1, -1, -1, -1] for i in range(NUM_ROW)] # Initialize the policy table
        for r in range(NUM_ROW): # for each row
                for c in range(NUM_COL): # for each column, we identify each state
                        if (r <= 1 and c == 3) or (r == c == 1):
                                # we skip the terminal states and the state that is not achievable
                                continue # skip and loop to the next item
                        maxAction, maxU = None, -float("inf")
                        # initialize the optimal action related variables
                        for action in range(NUM_ACTIONS): # for each action
                                u = calculateU(U, r, c, action) # calculate the summed value if
taking that action
                                if u > maxU: # compare with the largest one among the 4 actions
                                        maxAction, maxU = action, u #record the current optimal
```

```
action index
                    policy[r][c] = maxAction    # update the optimal policy at the current
state as the action that gives max u
        return policy # return the optimal policy table
```