```python
'''
Code Introduction:
The defined class MDP takes 2 positional arguments: Discount factor, Noise
This class has the following inbuilt functions built in them:
1. Transition Model: Takes current state and action as inputs and returns resulting state.
2. q_value function: Takes state and action as inputs and returns q value
3. value_iterations function:  prints Q*(s,a) value for all states along with the best recommended action
4. policy function: Takes current state as input and prints out the optimal path from current state to terminal state
'''

# Defining a class MDP which takes discount factor and noise as inputs
# i.e. inputs: 0 < Discount Factor, Noise < 1
class Mdp:
    def __init__(self, discount_factor, noise):
        self.discount_factor = discount_factor
        self.noise = noise
        self.actions = ["A1", "A2", "A3", "A4"]
        self.action_cost = {
            "A1": -1.5,
            "A2": -2,
            "A3": -0.5,
            "A4": -0.5
        }
        self.state_value = {}
        self.new_state_value_actions = {}
        self.states_best_actions = {}
        self.new_state_value = {}

    # Transition model which takes current state and action as inputs and returns resulting state
    # Input state in format (column, row, direction)
    # Input action as "Ai" where i = 1,2,3,4
    def transition_model(self, state, action):
```

```python
        column_initial, row_initial, direction_initial = state

        # Defining inaccessible states for agent
        blocked_states = [(2, 2), (2, 3), (3, 2)]
        blocked_moves = []
        for z, y in blocked_states:
            for i in [1, 2, 3, 4]:
                blocked_moves.append((z,y,i))

        # returning initial state if agent tries to run in barriers
        if (column_initial, row_initial, direction_initial) in [(2, 5, 4), (3, 5, 3), (5, 3, 1), (5, 4, 2)]:
            if action in ["A1", "A2"]:
                return column_initial, row_initial, direction_initial

        # Assigning new direction to the agent depending on action A3 or A4
        rotating_actions = [1, 3, 2, 4, 1, 3, 2, 4]
        if action == "A3":
            direction = rotating_actions[rotating_actions.index(direction_initial)+1]
        elif action == "A4":
            direction = rotating_actions[rotating_actions.index(direction_initial)-1]
        else:
            direction = direction_initial

        # Defining the effect of actions A1 and A2 in initial direction 1/2/3/4
        steps = 1
        action_definition = {
            1: (column_initial, (row_initial+steps), direction),
            2: (column_initial, row_initial-steps, direction),
            3: ((column_initial-steps), row_initial, direction),
```

```
            4: (column_initial+steps, row_initial, direction)
        }

        '''
        Defining the number of steps agent should take in a particular action i.e.
        for A1, no. of steps = 1 in facing direction
        for A2, no. of steps = 2 in facing direction
        for A3 and A4, no steps
        '''
        if action == 'A1':
            steps = 1
        elif action == 'A2':
            (c, r, d) = action_definition[direction_initial]
            if (c, r) in blocked_states or (c, r) in [(4, 4), (5, 5), (5, 3), (2, 5), (3, 5)]:
                return column_initial, row_initial, direction_initial
            else:
                steps = 2
        else:
            steps = 0

        # Updating the dictionary to according to the number of steps the agent should take
        action_definition = {
            1: (column_initial, (row_initial+steps), direction),
            2: (column_initial, row_initial-steps, direction),
            3: ((column_initial-steps), row_initial, direction),
            4: (column_initial+steps, row_initial, direction)
        }
        # Assigning the appropriate resulting stage after factoring in the effect of action on initial state
        resulting_state = action_definition[
```

```
direction_initial]
        (column, row, direction) = resulting_state

        # Filtering out the result in case action is making the agent fall out of the grid
        if (resulting_state in blocked_moves) or row > 5 or column > 5 or row < 1 or column < 1:
            return column_initial, row_initial, direction_initial
        else:
            return resulting_state

    '''
    Function to calculate Q value, which returns the expected value of utility for a particular action a in
    a state s.
    '''
    def q_value(self, state, action):
        actions = ["A1", "A2", "A3", "A4"]
        actions.remove(action)
        qval = (1-self.noise)*(self.action_cost[action] +
                                    self.discount_factor*self.state_value[self.transition_model(state, action)])+\
        (self.noise/3)*(self.action_cost[actions[0]] +
                               self.discount_factor*self.state_value[self.transition_model(state, actions[0])])+\
        (self.noise/3)*(self.action_cost[actions[1]] +
                               self.discount_factor*self.state_value[self.transition_model(state,actions[1])])+\
        (self.noise/3)*(self.action_cost[actions[2]] +
                               self.discount_factor*self.state_value[self.transition_model(state,actions[2])])
```

```python
        return qval

    # Defining a Value Iteration function which prints first 10 iterations and final values after 100 iterations
    def value_iterations(self):
        states = []
        # Initializing the states list containing all states on the grid
        for i in [1, 2, 3, 4, 5]:
            for t in [1, 2, 3, 4, 5]:
                for robot_direction in [1, 2, 3, 4]:
                    states.append((i, t, robot_direction))

        # Assign an initial value of 0 to all states
        for x in states:
            self.state_value[x] = 0

        # For terminal and blocked states, assign suitable values
        for col, ro, cost in [(4, 4, -1000), (5, 5, 100), (2, 3, -100000), (2, 2, -100000), (3, 2, -100000)]:
            for d in [1, 2, 3, 4]:
                self.state_value[col, ro, d] = cost

        # Value Iteration containing 100 iterations
        for i in range(100):
            if i < 10:
                print(f'Iteration {i+1}:')
            for a, b, c in states:
                # If state is blocked/terminal, value is fixed
                if (a, b) in [(4, 4), (5, 5), (3, 2), (2, 2), (2, 3)]:
                    self.new_state_value_actions[a, b, c] = (self.state_value[a, b, c], "No Action")
                # for accessible states, value should be updated in subsequent iterations
                else:
```

```
                    self.new_state_value[a, b, c] = [round(self.q_value((a, b, c), act), 2) for act in
       ["A1", "A2", "A3", "A4"]]
                    self.new_state_value_actions[a, b, c] = max(self.new_state_value[a, b, c]),\
                            self.actions[(self.new_state_value[a, b, c]).index(max(self.new_state_value[a, b, c]))]
                # Forming a dictionary states_best_actions to keep track of best action in a particular state
                val, act = self.new_state_value_actions[a, b, c]
                self.states_best_actions[(a,b,c)] = f'State {a,b,c} V = {val}      Best Action: {act}'
                # Updating the state value dictionary with new values
                self.state_value[a, b, c] = val
            # Printing the first 10 value iterations
            if i < 10:
                for key, value in self.states_best_actions.items():
                    print(value)
            i += 1
        print(f'\n(Values, Best Action) after 100 iterations: {self.new_state_value_actions}')

    '''
    Defining a policy function with input: current state
    It returns the optimal path to reach the terminal state from current state
    '''
    def policy(self, state):
        print(f"\nPolicy Extraction with initial state : {state}")
        (c, r, d) = state
        while (c, r) not in [(4, 4), (5, 5)]:
            (v, ac) = self.new_state_value_actions[(
```

```
c, r, d)]
            print(f'Current State: {state}, Best Action: '
                  f'{self.actions[(self.new_state_value[c,r,d]).index(max(self.new_state_value[c,r,d]))]}')
            state = self.transition_model((c, r, d), ac)
            (c, r, d) = state
        print(f'Final State: {state}')

# initialize a class instance: puzzle = Mdp(1, 0)
# print value iterations using puzzle.value_iterations()
# Print policy: puzzle.policy((1, 1, 4))
```