ID: 190041220
Name: Tasfia Tasneem Annesha
Lab Group: 2B

Lab 5: Markov Decision Process

Question 1 (4 points): Value Iteration

The primary task in the runValueIteration() function is to iterate through each state and, for each state, gather all possible actions. Subsequently, for each action, the corresponding Q-values are collected. The state's value is then updated to the maximum among these Q-values. This iterative process is repeated for a fixed number of iterations, allowing the agent to refine its value estimates over time.

```
    def runValueIteration(self):
        # Write value iteration code here
        "*** YOUR CODE HERE ***"

        #we know, V*(s)=max Q*(s,a)
        #v---different a(action)---->different chance nodes
(q values)---s,a,s'----->R(s,a,s')

        for _ in range(self.iterations):
            values_k1 = self.values.copy()
            for state in self.mdp.getStates():
                qValues = []
                for action in self.mdp.getPossibleActions(state):
                    qValue = self.getQValue(state, action) # Q*(s,a)
                    qValues.append(qValue)
                if len(qValues) > 0:
                    #print("The qValues of state-->",state,qValues)
                    values_k1[state] = max(qValues)# max Q*(s,a)
            self.values = values_k1
```

computeQValueFromValues() function :

The process of computing Q-values in the computeQValueFromValues() function entails taking into account the probabilities of the possible next states.
The Q-value for each subsequent state is determined by multiplying the immediate reward for performing the particular action by the probability of transitioning to that state. The value of the following state is then multiplied by the discount factor and this result. The final Q-value for the specified action is represented by the sum of these computed values.

```
    def computeQValueFromValues(self, state, action):# bellman
equations
        """
          Compute the Q-value of action in state from the
          value function stored in self.values.
        """
        "*** YOUR CODE HERE ***"
        # util.raiseNotDefined()
        qValue = 0
        for nextState, probability in
self.mdp.getTransitionStatesAndProbs(state, action):
            qValue += probability * (self.mdp.getReward(state, action,
nextState) + self.discount * self.getValue(nextState))
            #Q*(s,a)= summation over all the states(T(s,a,s') or
P(s'|s,a)) (R(s,a,s')+ discount * V*(s'))
        return qValue
```

computeActionFromValues() function:

In the computeActionFromValues() function which is basically policy extraction, the task is to iterate through all possible actions and gather their corresponding Q-values using a separate function. The argMax function is then employed to identify and return the action that yields the highest Q-value, indicating the optimal action to take in a given state.

```
    def computeActionFromValues(self, state):#Policy extraction
        """
          The policy is the best action in the given state
          according to the values currently stored in self.values.
```

```
          You may break ties any way you see fit.  Note that if
          there are no legal actions, which is the case at the
          terminal state, you should return None.
        """
        "*** YOUR CODE HERE ***"
        # util.raiseNotDefined()
        qValues = util.Counter()
        for action in self.mdp.getPossibleActions(state):
            qValues[action] = self.getQValue(state, action)
        return qValues.argMax() #pi*(s)= argmax over action (Q*(s,a))
```

```
    def getPolicy(self, state):
        return self.computeActionFromValues(state)

    def getAction(self, state):
        "Returns the policy at the state (no exploration)."
        return self.computeActionFromValues(state)

    def getQValue(self, state, action):
        return self.computeQValueFromValues(state, action)
```

Question 2 (1 point): Bridge Crossing Analysis:

Here the ques tells us to modify either the discount parameter or the noise parameter in the BridgeGrid environment in the `question2()` function of analysis.py so that the optimal policy encourages the agent to attempt to cross the bridge. It is told to use the default values of 0.9 for discount and 0.2 for noise as a reference.
Here, noise refers to how often an agent ends up in an unintended successor state when they perform an action. So here I make the noise 0.0 so that it perform the right task.Thus the agent successfully crosses the bridge

```
def question2():
    #With the default discount of 0.9 and the default noise of 0.2,
    #The optimal policy does not cross the bridge.
    #Here, noise refers to how often an agent ends up in an unintended
successor state when they perform an action.
    answerDiscount = 0.9
    answerNoise = 0.0
    return answerDiscount, answerNoise
```

<u>Question 3 (5 points): Policies</u>

In order to generate optimal policies of various kinds, we had to select values for the discount, noise, and living reward parameters for DiscountGrid MDP. These values are listed below:

3(a) Prefer the close exit (+1), risking the cliff (-10):

The agent must prioritize the shortest distance in this case. As a result, I gave the living reward a very low value of -1.The discount is set to one, indicating that the agent values rewards in the distant future more than those in the near future. As a result, take the shorter path risking the cliff in order to gain more points in the future. The noise level is set to 0.5, so the agent enters an unintended state only half of the time.

```
#Prefer the close exit (+1), risking the cliff (-10)
def question3a():
    # agent cares a lot about rewards in the distant future. Thus
making sure to take the shorter path risking the cliff
    answerDiscount = 1
    # the agent goes to unintended state only half of the time
    answerNoise = 0.5
    # lower value for shortest distance
    answerLivingReward = -1
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

b. Prefer the close exit (+1), avoiding the cliff (-1):

In this scenario, the agent needs to emphasize the shorter distance while steering clear of the cliff. To achieve this, a moderately low living reward, specifically -0.9, is assigned, incentivizing the agent to prioritize reaching the goal quickly. The discount parameter is set to 0.3, signaling that the agent places less importance on distant rewards compared to immediate ones. This ensures a cautious approach near the cliff to maximize immediate gains. Additionally, a noise value of 0.2 is chosen, minimizing the occurrence of unintended state transitions, contributing to the agent's overall effectiveness in navigating the environment.

```
#Prefer the close exit (+1), avoiding the cliff (-1)
def question3b():
    #the agent cares less about rewards in the distant future, avoid
cliff and more score in near future
    answerDiscount = 0.3
```

```
    #the agent goes to unintended state very less number of times
    answerNoise = 0.2
    # must prioritize the shorter distance, at the same time avoid the
cliff
    answerLivingReward = -0.9
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

c. Prefer the distant exit (+10), risking the cliff (-10)

Here, the agent is required to prioritize reaching the distant exit while balancing the risk involved. To achieve this balance, a moderately negative living reward of -0.5 is assigned, encouraging the agent to consider both risk and reward in its decisions. The discount parameter is set to 0.9, indicating a high value placed on rewards in the distant future, motivating the agent to take risks near the cliff for potentially higher long-term gains. Additionally, a low noise value of 0.1 is chosen, minimizing unintended state transitions and promoting more precise navigation by the agent.

```
#Prefer the distant exit (+10), risking the cliff (-10)
def question3c():
    answerDiscount = 0.9 # cares much about distance future, risking
cliff
    answerNoise = 0.1 # the agent goes to unintended state very less
number of times
    answerLivingReward = -0.5 #taking risk, prioritizing distance
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

d. Prefer the distant exit (+10), avoiding the cliff (-10)

In this scenario, the agent is tasked with prioritizing the distant exit while steering clear of the cliff. To achieve this, a very slight negative living reward, specifically -0.01, is assigned, nudging the agent towards choosing the longer path. The discount parameter is set to 0.5, indicating that the agent places moderate emphasis on immediate rewards relative to those in the distant future. This encourages the agent to be cautious near the cliff, prioritizing immediate gains over long-term rewards. Additionally, a low noise value of 0.2 is chosen, ensuring the agent encounters unintended state transitions infrequently, promoting a more reliable and deliberate navigation strategy.

```
#Prefer the distant exit (+10), avoiding the cliff (-10)
def question3d():
    answerDiscount = 0.5 #prioritize the distant exit at the same time
avoid the cliff
    answerNoise = 0.2- #the agent goes to unintended state very less
number of times
    answerLivingReward = -0.01 # prioritize the distant exit at the
same time avoid the cliff
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

e. Avoid both exits and the cliff (so an episode should never terminate)

In this case, the agent must take indefinite action. To ensure that the living reward was given to +1.As a result, the agent will take steps to avoid exits and cliffs in order to increase his score. To achieve the same result, the discount and noise are both set to zero.

```
# Avoid both exits and the cliff (so an episode should never terminate)
def question3e():
    answerDiscount = 0
    answerNoise = 0
    answerLivingReward = 1
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

Question 4 (1 point): Asynchronous Value Iteration

Task 4 closely resembles Task 1, with the key distinction being that in each iteration, only one state is examined. The sequential checking of states is achieved through the implementation of a straightforward queuing mechanism. After being checked in a particular iteration, each state is then moved to the end of the queue, ensuring that the next iteration involves the examination of the subsequent state in the queue.
In value iteration each iteration, all states are considered simultaneously for update. But in asynchronous value iteration only one state is considered in each iteration.The algorithm sequentially checks states using a queuing mechanism, updating one state at a time and cycling through the queue.

```
class AsynchronousValueIterationAgent(ValueIterationAgent):
    """
        * Please read learningAgents.py before reading this.*
```

```
        An AsynchronousValueIterationAgent takes a Markov decision
process
        (see mdp.py) on initialization and runs cyclic value iteration
        for a given number of iterations using the supplied
        discount factor.
    """
    def __init__(self, mdp, discount = 0.9, iterations = 1000):
        """
          Your cyclic value iteration agent should take an mdp on
          construction, run the indicated number of iterations,
          and then act according to the resulting policy. Each
iteration
          updates the value of only one state, which cycles through
          the states list. If the chosen state is terminal, nothing
          happens in that iteration.

          Some useful mdp methods you will use:
              mdp.getStates()
              mdp.getPossibleActions(state)
              mdp.getTransitionStatesAndProbs(state, action)
              mdp.getReward(state)
              mdp.isTerminal(state)
        """
        ValueIterationAgent.__init__(self, mdp, discount, iterations)
        self.mdp = mdp
        self.discount = discount
        self.iterations = iterations
        self.values = util.Counter() # A Counter is a dict with default
0

        self.runValueIteration()

    def runValueIteration(self):
        "*** YOUR CODE HERE ***"
        states = self.mdp.getStates()
    #to iterate through the states in the order provided by the
getStates() functions
    #and update one state per iteration
        for _ in range(self.iterations):
            # to update one state per iteration, cycling through the
list.
            state = states[0] #first state
            values_k1 = self.values.copy()
            qValues = []
```

```
            for action in self.mdp.getPossibleActions(state):
                qValue = self.getQValue(state, action)# Q(s,a)
                qValues.append(qValue)
            if len(qValues) > 0:
                values_k1[state] = max(qValues) #bellman equation
            self.values = values_k1
            states.remove(state)#The first state in the list is
removed.
            states.append(state)#same state appended at the end
```

Question 5 (3 points): Prioritized Sweeping Value Iteration:

Here in this task we have to implement the `runValueIteration` method in the `PrioritizedSweepingValueIterationAgent` class in `valueIterationAgents.py`. Following the prioritized sweeping algorithm, computing predecessors for all states, initializing a priority queue, and iteratively updating states based on the highest Q-value differences.

```
class
PrioritizedSweepingValueIterationAgent(AsynchronousValueIterationAgent)
:
    """
        * Please read learningAgents.py before reading this.*

        A PrioritizedSweepingValueIterationAgent takes a Markov
decision process
        (see mdp.py) on initialization and runs prioritized sweeping
value iteration
        for a given number of iterations using the supplied parameters.
    """
    def __init__(self, mdp, discount = 0.9, iterations = 100, theta =
1e-5):
        """
          Your prioritized sweeping value iteration agent should take
an mdp on
          construction, run the indicated number of iterations,
          and then act according to the resulting policy.
        """
        self.theta = theta
        AsynchronousValueIterationAgent.__init__(self, mdp, discount,
iterations)
```

```
    def runValueIteration(self):
        "*** YOUR CODE HERE ***"
        # compute predecessors for each state
        predecessors = defaultdict(set)

        # iterated over them to check if it is a terminal state or not.
If it is not a terminal state
        for s in self.mdp.getStates():
            if not self.mdp.isTerminal(s):
                for a in self.mdp.getPossibleActions(s):# all successor
in which can end up from this state using all the available actions.
                    for nextState, prob in
self.mdp.getTransitionStatesAndProbs(s, a):
                        predecessors[nextState].add(s)

        # Initialize a priority queue
        priority_queue = util.PriorityQueue()

        for s in self.mdp.getStates():
            if not self.mdp.isTerminal(s):
                # Find the absolute difference between current value
and highest Q-value
                diff = abs(self.values[s] -
self.computeQValueFromValues(s, self.getAction(s)))

                # ◊ Push s into the priority queue with priority -diff
(note that this is negative)
                priority_queue.update(s, -diff)

        for iteration in range(self.iterations):

            #If the priority queue is empty, then terminate.
            if priority_queue.isEmpty():
                break

            s = priority_queue.pop()  # Pop a state from the priority
queue

            # Update s’s value (if it is not a terminal state) in
self.values.
            if not self.mdp.isTerminal(s):
```

```
                self.values[s] = self.computeQValueFromValues(s,
self.getAction(s))

            #For each predecessor p of s
            for p in predecessors[s]:
                # Find the absolute difference between current value of
predecessor and highest Q-value
                diff = abs(self.values[p] -
self.computeQValueFromValues(p, self.getAction(p)))

                if diff > self.theta:
                    # Push the predecessor into the priority queue with
negative priority if not already there
                    if not priority_queue.update(p, -diff):
                        priority_queue.update(p, -diff)
```