Lab 5

Tasnimul Hasnat

190041113

Sunday, Nov 12 2023

Task 1

The first question required us to implement a value iteration agent, which is already solved for us. For this in the ValueIterationAgent class we are given some functions which are already implemented and we have to implement the rest.

In the runValueIteration() function what is basically done is that for each of the states we collect all the possible actions and then for each of the actions we collect the Q-values. Then the value of the state is updated to the maximum of the Q-values. This entire process is repeated for a fixed amount of iterations.

In the computeQValueFromValues(), for each of the next states and the probability associated with that specific next state, the Q-values are calculated and all are added together. To calculate the Q-value, the probability is multiplied with the reward to reach the next state taking the action and added with the multiplication of the discount factor and the value of the next state. Thus the summation of them all is returned.

In the computeActionFromValues() function, for each of the possible actions, the Q-values are collected using another function and argMax is used to return the action that gives the maximum Q-value.

P.T.O

```python
def runValueIteration(self):
    # Write value iteration code here
    "*** YOUR CODE HERE ***"
    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)
                qValues.append(qValue)
            if len(qValues) > 0:
                values_k1[state] = max(qValues)
        self.values = values_k1
```

```python
def computeQValueFromValues(self, state, action):
    """
      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))
    return qValue


def computeActionFromValues(self, state):
    """
      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()
```

Task 2

This question required us to change only ONE of the discount and noise parameters so that the optimal policy causes the agent to attempt to cross the bridge in BridgeGrid.
For this I reduced the default value of noise from 0.2 to 0.01 so that the agent doesn't end up in an unintended successor state so frequently when they perform an action. Thus the agent successfully crosses the bridge.

```
def question2():
    answerDiscount = 0.9
    answerNoise = 0.01
    return answerDiscount, answerNoise
```

P.T.O

Task 3

In this question, we were required to choose values of the discount, noise, and living reward parameters for DiscountGrid MDP to produce optimal policies of several different types that are given below:

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

In this case the agent must prioritize the shortest distance. That's why I gave the living reward a very low value that is -1.

The discount is set to 1 so that the agent cares a lot about rewards in the distant future relative to those in the immediate future. Thus making sure to take the shorter path risking the cliff to get more score in the distant future.

The noise is set to 0.5 so that the agent goes to unintended state only half of the time.

```
def question3a():
    answerDiscount = 1
    answerNoise = 0.5
    answerLivingReward = -1
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

P.T.O

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

In this case the agent must prioritize the shorter distance, at the same time avoid the cliff. That's why I gave the living reward a lower value but not the lowest that is -0.9.

The discount is set to 0.3 so that the agent cares less about rewards in the distant future relative to those in the immediate future. Thus making sure to avoid the cliff to get a higher score in the immediate future.

The noise is set to 0.2 so that the agent goes to unintended state very less number of times.

```
def question3b():
    answerDiscount = 0.3
    answerNoise = 0.2
    answerLivingReward = -0.9
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

P.T.O

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

In this case the agent must prioritize the distant exit. That's why I gave the living reward a negative value in between that is -0.5. Thus a balance between taking the risk and prioritizing the distant exit.

The discount is set to 0.9 so that the agent cares much about rewards in the distant future relative to those in the immediate future. Thus making sure to risk the cliff to get a higher score in the distant future.

The noise is set to 0.1 so that the agent goes to unintended state very less number of times.

```
def question3c():
    answerDiscount = 0.9
    answerNoise = 0.1
    answerLivingReward = -0.5
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

P.T.O

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

In this case the agent must prioritize the distant exit and at the same time avoid the cliff. That's why I gave the living reward a very small negative value that is -0.01. Thus the agent chooses the longer path.

The discount is set to 0.5 so that the agent cares not so much about rewards in the distant future relative to those in the immediate future. Thus making sure to avoid the cliff.

The noise is set to 0.2 so that the agent goes to unintended state very less number of times.

```
def question3d():
    answerDiscount = 0.5
    answerNoise = 0.2
    answerLivingReward = -0.01
    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 steps indefinitely. To ensure that I gave the living reward to +1.

Thus the agent will take steps and avoid the exits, cliff to get more score.

The discount and noise are set to 0 to achieve the same goal.

```
def question3e():
    answerDiscount = 0
    answerNoise = 0
    answerLivingReward = 1
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

Task 4

The question required us to write an asynchronous value iteration agent. For this I worked on the class named AynchronousValueIterationAgent, where I have to write a value iteration function that updates only one state in each iteration.

In the runValueIteration() function, for each of the iterations I considered only one state. I used modulus to make it cyclic. Then for the state I checked if it is terminal or not. If It is not, then for each of the actions from that state I collected the Q-values using another function and took the maximum among them as the new value for that specific state. The process is continued until the given number of iterations is reached.

```
class AsynchronousValueIterationAgent(ValueIterationAgent):
+--  8 lines: """.....
+-- 17 lines: def __init__(self, mdp, discount = 0.9, iterations = 1000):

    def runValueIteration(self):
        "*** YOUR CODE HERE ***"
        states = self.mdp.getStates()
        num_s = len(states)
        for i in range(self.iterations):
            state = states[i%num_s]
            if not self.mdp.isTerminal(state):
                values_k1=[]
                for action in self.mdp.getPossibleActions(state):
                    q_value = self.computeQValueFromValues(state,action)
                    values_k1.append(q_value)
                self.values[state] = max(values_k1)
```

Task 5

For the last task we needed to implement the PrioritizedSweepingValueIterationAgent class that implements prioritized sweeping value iteration for a given Markov Decision Process. The agent utilizes a priority queue to efficiently update state values, prioritizing states with larger discrepancies between their current values and the maximum Q-values. The algorithm initializes predecessors for each state, computes initial priority values, and then iteratively updates state values based on the maximum Q-values. The process continues until a specified number of iterations or until the priority queue is empty. The agent's behavior is determined by the resulting policy after the specified number of iterations. The code employs modular functions such as getPossibleActions and getTransitionStatesAndProbs from the MDP, and it includes a threshold to control convergence.

P.T.O

```python
class PrioritizedSweepingValueIterationAgent(AsynchronousValueIterationAgent):
+-- 16 lines: """.....
    def runValueIteration(self):
        "*** YOUR CODE HERE ***"

        mdp = self.mdp
        values = self.values
        discount = self.discount
        iterations = self.iterations
        theta = self.theta
        states = mdp.getStates()


        predecessors = {} # dict
        for state in states:
          predecessors[state] = set()

        pq = util.PriorityQueue()

        # computes predecessors and puts initial stuff into pq
        for state in states:
          Q_s = util.Counter()

          for action in mdp.getPossibleActions(state):
            # assigning predecessors
            T = mdp.getTransitionStatesAndProbs(state, action)
            for (nextState, prob) in T:
              if prob != 0:
                predecessors[nextState].add(state)

            # computing Q values for determining diff's for the pq
            Q_s[action] = self.computeQValueFromValues(state, action)
```

```
    if not mdp.isTerminal(state): # means: if non terminal state
      maxQ_s = Q_s[Q_s.argMax()]
      diff = abs(values[state] - maxQ_s)
      pq.update(state, -diff)


  # now for the actual iterations
  for i in range(iterations):
    if pq.isEmpty():
      return

    state = pq.pop()

    if not mdp.isTerminal(state):
      Q_s = util.Counter()
      for action in mdp.getPossibleActions(state):
        Q_s[action] = self.computeQValueFromValues(state, action)

      values[state] = Q_s[Q_s.argMax()]

    for p in predecessors[state]:
      Q_p = util.Counter()
      for action in mdp.getPossibleActions(p):
        # computing Q values for determining diff's for the pq
        Q_p[action] = self.computeQValueFromValues(p, action)

      #if not mdp.isTerminal(state): # means: if non terminal state
      maxQ_p = Q_p[Q_p.argMax()]
      diff = abs(values[p] - maxQ_p)

      if diff > theta:
        pq.update(p, -diff)
```

— XXX —