ISLAMIC UNIVERSITY OF TECHNOLOGY

Organization of Islamic Cooperation

Board Bazar, Gazipur

Lab Report 5

CSE 4712

Md. Bakhtiar Hasan

20th November, 2022

## Question 01

The first task was the basic implementation of value iteration. The solution for this was provided, but the code is still being examined.

The main concept of value iteration is that we iterate times over all the states, and for each state we calculate the value based on the maximum q-value achievable from that state. By iterating times, we ensure that the values converge. This follows from the formula:

This equation is executed in the method runValueIteration.

def runValueIteration(*self*):  
 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

To calculate the q-value for each (state, action) pair, we have to take into account the instant reward the agent gets by taking the action, the value of the state the agent ends up in, the discount factor, and the probability of the action leading to that state. This is done using the method getValue, which calls the method computeQValueFromValues.

def computeQValueFromValues(*self*, state, action):  
 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

PYTHON

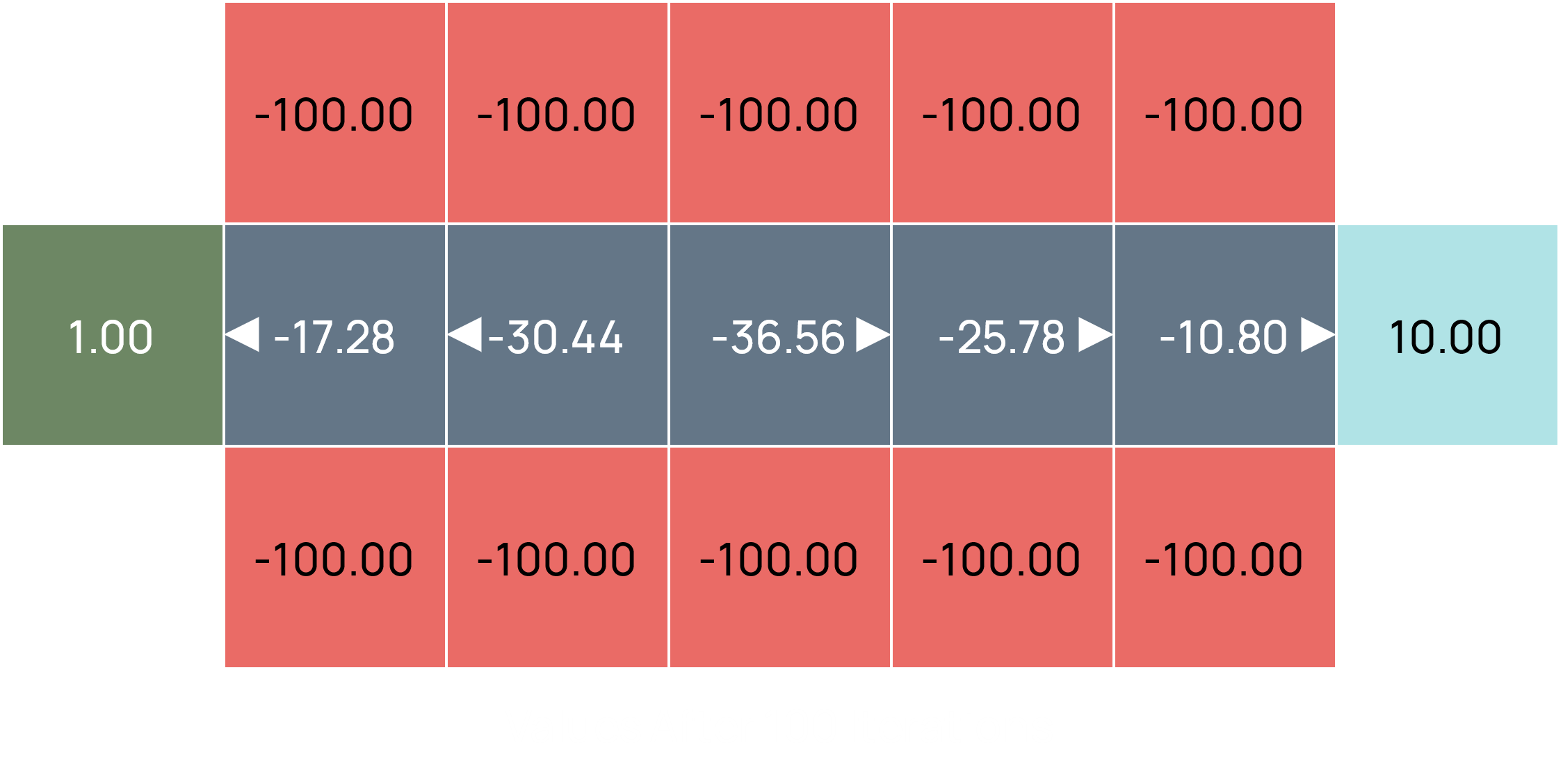
The two blocks of code above deal with the calculation of the values of the different states. However, the actual solution to an MDP is a policy, which is calculated by taking the optimal action at each state based on the values of its neighbouring states. This is handled in the method getPolocy, which calls the method computeActionFromValues.

def computeActionFromValues(*self*, state):  
 qValues = util.Counter()  
 for action in *self*.mdp.getPossibleActions(state):  
 qValues[action] = *self*.getQValue(state, action)  
 return qValues.argMax()

PYTHON

## Question 02

The second task setup a scenario where the agent was on a bridge going over a chasm, with positive terminal states at either end.



The agent was initially close to the low-reward terminal states, which resulted in the agent trying to head towards that state. Our task was to fine-tune the discount and noise parameters so that the agent went towards the high-reward terminal state instead. The tricky part was that we were only allowed to fine-tune one of the two parameters.

The default values of the parameters were for the discount and for the noise. Changing the noise parameter to lead to successful results. This happened because the agent no longer had any possibility of falling into the chasm, which made the high-reward terminal state more attractive than the low-reward one.

def question2():  
 answerDiscount = 0.9  
 answerNoise = 0.0 # no possibility of failing  
 return answerDiscount, answerNoise

PYTHON

## Question 03

The third task was similar to the second one, in that another scenario was provided to us.



There were 5 parts to this task, and we had three parameters to tune, the two previous parameters and a new parameter, the living reward. The agent can go either to the low-reward terminal state or the high-reward one, and they can take either the shorter, riskier path, or the longer, safer one.

To make the agent go down the riskier path, we can set the noise to , which prevents the agent from ever falling into the chasm. To make the agent go to the closer reward, we can set the discount to a low value, which makes the high-reward terminal state less valuable since it takes more moves to reach it. To get the opposite effect, we can set the discount to a high value.

Using the logic above, the first four parts of the tasks are as follows:

1. Go to the low-reward terminal state using the shorter path.

def question3a():  
 answerDiscount = 0.1 # closer reward more valuable  
 answerNoise = 0 # no possibility of failing  
 answerLivingReward = 0 # no benefit to staying alive longer  
 return answerDiscount, answerNoise, answerLivingReward

PYTHON

1. Go to the low-reward terminal state using the longer path.

def question3b():  
 answerDiscount = 0.1 # closer reward more valuable  
 answerNoise = 0.1 # risky to go near edge  
 answerLivingReward = 0 # no benefit to staying alive longer   
 return answerDiscount, answerNoise, answerLivingReward

PYTHON

1. Go to the high-reward terminal state using the shorter path.

def question3c():  
 answerDiscount = 0.9 # distant reward more valuable  
 answerNoise = 0 # no possibility of failing  
 answerLivingReward = 0 # no benefit to staying alive longer   
 return answerDiscount, answerNoise, answerLivingReward

PYTHON

1. Go to the high-reward terminal state using the longer path.

def question3d():  
 answerDiscount = 0.9 # distant reward more valuable  
 answerNoise = 0.1 # risky to go near edge  
 answerLivingReward = 0 # no benefit to staying alive longer  
 return answerDiscount, answerNoise, answerLivingReward

PYTHON

Notice that for all four cases above, the living reward was set to . This was done to prevent the agent from taking the amount of time it has been alive into account, which would make the thought process for the four cases above unnecessarily complicated.

The fifth task, however, essentially asks that the agent stay alive forever. To accomplish this, we can set a high living reward and set the discount to so that there is no benefit to going to the terminal states. Additionally, the noise can also be set to , since we do not need to take into account the path which the agent follows.

def question3e():  
 answerDiscount = 0 # never go to terminal state  
 answerNoise = 0 # no risk of failure  
 answerLivingReward = 1 # stay alive forever  
 return answerDiscount, answerNoise, answerLivingReward

PYTHON

## Question 04

The final task was an implementation of temporal difference learning. The only difference between this and value iteration is that instead of updating the value of every state in each iteration, we only update the value of one. Since this is the only difference, the implementation of this inherits the code of value iteration and just modifies the runValueIteration method.

def runValueIteration(*self*):  
 states = *self*.mdp.getStates()  
 for i in range(*self*.iterations):  
 index = i % len(states) # i could be larger than len(states)  
 qValues = []  
 for action in *self*.mdp.getPossibleActions(states[index]):  
 qValue = *self*.getQValue(states[index], action)  
 qValues.append(qValue)  
 if len(qValues) > 0:  
 *self*.values[states[index]] = max(qValues)

PYTHON