CPSC 535 Project 1: Savvy Traveler

Group Members:

1. Priyansha Singh
2. Akshaya Kizhkkencherry
3. Archana Tella


```
/**
   CPSC535-Project1(Savvy Teaveler)

   Group Members:
    Akshaya Kizhkkencherry     AkshayaR@csu.fullerton.edu
    Priyansha Singh            priyansha@csu.fullerton.edu
    Archana Tella              archanatella@csu.fullerton.edu


 */
```

Summary

The aim of this project is to solve the Savvy Traveler problem. If the traveler journeys across the US by air, there are a lot of flights to various destinations across the country with one or more hops. We calculate (i) what route will maximize the probability to arrive on time between two given cities, and (ii) what city is the most reliable travel destination. The flight routes are illustrated using an undirected graph. The cities are represented by the vertices and the flight path represents the edges from the vertices. The probability values are listed across the edges.

Given a graph and two cities, we need to compute the path with the highest probability of on-time arrival. If the path has a single edge, then it will be the probability of that edge. If the path has multiple edges, then multiply all the probabilities. When alternating paths are available, select the one with the highest probability.

Given a graph, for any city we measure the reliability of that city as the sum of all probabilities of on-time arrivals from each other city. The most reliable city is the one with the highest reliability.

Constraints:

1. Source and destination nodes cannot be the same. (Start and end vertex should be different)
2. 0 <= Max Probability <= 1
3. There is at most one edge between every two nodes (i.e no of hops in between the source and destination should at least be 1)

Pseudocodes

PseudoCode for Calculating Maximum probability and Path(Part A) (CalMaxPathProb.java):

function maxProbability(int n, String nodes[], String[][] edges, double[] succProb, String start, String end) {

Result output = new Result(); //create default output with probability 0.0

//create adjMap from the given edges to keep track of nodes reachable from each other with their probabilities

Map<String, List<LinkedNodes>> adjMap = new HashMap<String, List<LinkedNodes>>();

for each node in nodes {

add edges to adjMap with probabililties

}

//Create a priority queue with a comparator to compare the elements of the queue and assign the element with the highest probability a higher priority

PriorityQueue<Path> pQueue = new PriorityQueue<Path>((a, b) -> Double.compare(b.getCurrentProb(), a.getCurrentProb()));

//adding start node to priority queue

pQueue.offer(new Path(1.0, start));

//create visited set to keep track of an already visited node

Set<String> visited = new LinkedHashSet<String>();

while pQueue is not empty {

Path currentPath = pQueue.poll(); //retrieving head of the queue

String node = currentPath.getCurrentNode();

if (visited.contains(node)) {

//break the iteration

```
}

//if not visited add to the set

visited.add(node);

//get the list of adjacent nodes for currentNode(node)

for each linkedNode in adjMap {

//create prevNodes list to keep track of the predecessor nodes in order to return the most probable path

ArrayList<String> prevNodes = new ArrayList<String>();

//add currentnode source to the prevNode list

 prevNodes.add(LinkedNodes.getSource());

 //compute the product of the probability of the current node being processed with the linked nodes probabilities and adding to the priority queue

pQueue.offer(new Path(currentPath.getCurrentProb() * LinkedNodes.getProb(),
LinkedNodes.getDest(),

                                                    LinkedNodes.getSource(), prevNodes));

}

//if destination node is found, return the path and probability calculated

if (node.equals(end)) {

//adding destination node to the captured path

currentPath.getOutputPath().add(end);

return output //output should contain maxProbability and mostprobable path

                                    }

                    }

//otherwise return default output

return output;
```

```
        }


}
```

PseudoCode for Calculating Maximum Reliability (Part B)

```
Initialize node array with all the nodes

initialize a totalProb variable //to store summed probability for each node

for each node in Node array {

//compute maxProbability for each node with every other node in the array by calling the same function maxProbability created for part A

double totalProb = 0.0;

for (String k : nodes) {

Result reliableCityResult = sol.maxProbability(8, nodes, graphEdges, graphProb, e, k, false);

//sum all the probabilties

        totalProb = totalProb + reliableCityResult.getMaxProbValue();

        }

  //create a hashmap with key as node and value as totalprobability for each node

        probMap.put(e, totalProb);

//check the max probability value in the map and Iterate the hashmap to get the key(node)

        maxValueInMap = (Collections.max(probMap.values()));

          Iterate maxValueInMap

        return key //most reliable Node

                                }
```

PseudoCode for Main Caller Function to call both the functions above and print result (MainProgCaller.java)

Input: nodes, Edges{}, Probability{}

Output: MaxProbValue, MaxProbPath

nodes= string Arr { "A", "B", "C", "D", "E", "F", "G", "H" };

graphProb=NULL

graphEdges=NULL

Source = NULL

Destination: NULL //initialize Probability, Graph Edges, Source and Destination as Null values.

Declare Edges= Array {"A","B"},{"A","C"}, { "D","A" },{ "D","G" } ,{"D","F"},{"C","F"}, {"B","C"},{"B","F"},{"B","E"}, {"E","F"},{"F","G"},{"G","H"} ,{"F","H"},{"E","H"} //Example1

Declare Probability = Array {0.8,0.7,0.9,0.8,0.6,0.9,0.8,0.6,0.6,0.8,0.7,0.9,0.7,0.6}

src="A"

dest="F"

probMap = HashMap(edges, Probability)

maxValueInMap = 0.0

Maximum Probability = CalMaxProbPath() //calculating maximum probability for the given source and destination. The function is called.

result = maxProbability(vertices, nodes, graphEdges, graphProb,src, dest, true)

```
//calculating reliability of each node with every other node
for (String e : nodes) {
    double totalProb = 0.0;
for (String k : nodes) {
      reliableCityResult = maxProbability(8, nodes, graphEdges, graphProb, e, k, false);
      totalProb = totalProb + maxProbValue  // Calculate the total Probabilty to find out the most reliable city
}
```

```
for (String key : getKeys(probMap, maxValueInMap)) {
            Print key
          }

print  MaxProbPath
print MaxProbValue
       }

//Exception in case of invalid value
Catch ValueError
{
print ErrorMessage

}
```

Steps to run the program

1.Download GraphPathProb.jar from github

2.Run command java -jar GraphPathProb.jar

3.As shown in the below screenshots, user input will be requested to enter graph no as 1,2 or 3

4.If you Enter 1 -displays result for graph1 and so on.

5. or alternatively you can import the source code in eclipse and run MainProgCaller.java class

Note:-Make sure Java runtime environment-java version "1.8.0_321" or above is installed and set in system in order to run the above program.To install follow the steps in the page: https://docs.oracle.com/goldengate/1212/gg-winux/GDRAD/java.htm#BGBFHBEA

Output Screenshots:

Graph-1

```
C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19043.1526]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Priyansha\Documents>java -jar GraphPathProb.jar
**Enter graph no as 1,2 or 3**
1
Most Reliable City for Part B is:: F
Maximum Probable Path for Part A:: [A, C, F]
Maximum Probability Value for Part A:: 0.63
```

Graph-2

```
C:\Users\Priyansha\Documents>java -jar GraphPathProb.jar
**Enter graph no as 1,2 or 3**
2
Most Reliable City for Part B is:: D
Maximum Probable Path for Part A:: [C, F, E, D, A]
Maximum Probability Value for Part A:: 0.5184000000000001
```

Graph-3

```
C:\Users\Priyansha\Documents>java -jar GraphPathProb.jar
**Enter graph no as 1,2 or 3**
3
Most Reliable City for Part B is:: A
Maximum Probable Path for Part A:: [E, A, D, C]
Maximum Probability Value for Part A:: 0.6480000000000001
```

Time complexity : O(|E|\log |E|) //E- number of edges, V - number of vertices

Space complexity : O(|V| + |E|)