**ECE361 Lab 5 Instructions**

**Shortest Path Routing Algorithm**

**Goal**: To implement a simplified version of Dijkstra’s algorithm in small networks.

**Description:**

One of the problems with computer networks is to find the shortest path between a certain node and all other nodes in the network. To solve such problem, the network is represented by a graph, with nodes and edges connecting the nodes. An edge between two nodes indicates there is a physical connection between them. Normally, these edges are assigned with weights. In this lab, weights of edges are the delays of sending data between nodes.

Dijkstra’s algorithm is an effective method used to solve a single source routing problem, i.e. it finds the shortest path from a certain source node to all other nodes in the network. In this lab, you will represent the network by a graph with nodes and edges, and run Dijkstra’s algorithm to find the shortest paths from each node to all other nodes.

**Tasks:**

**1- Graph representation of the network**

The first step is to create Node and Edge classes that will be used to create the graph. The definition of these classes is provided here. Just copy and paste these classes, you don’t need to add anything to them.

**class** Node **implements** Comparable<Node>

{

**public** **final** **int** name; // Node’s name

**public** Edge[] neighbors; // set of neighbors to this node

**public** **double** minDistance = Double.*POSITIVE\_INFINITY*; //Minimum weight, //initially inf

**public** Node previous; // to keep the path

**public** Node(**int** argName) // constructor to create an instance of this class

{

name = argName;

}

**public** **int** compareTo(Node other)

{

**return** Double.*compare*(minDistance, other.minDistance);

}

}

**class** Edge

{

**public** **final** Node target; // destination node

**public** **final** **double** weight; // the delay, in ms

**public** Edge(Node argTarget, **double** argWeight) //constructor to create an instance

{

target = argTarget;

weight = argWeight;

}

}

**2- Reading weights from server**

Next, you have to create a class RoutingClient that will include the implementation of Dijkstra’s algorithm.

a- In the main method of this class, create a socket to communicate with the server. In addition, use BufferedReader and DataOutputStream to create reader and writer to communicate with the server.

b- Once a connection is established, you need to send the number of nodes in the network to the server. Use Scanner to read the number of nodes from the user. This can be done as

Scanner scr = **new** Scanner(System.in);

Use nextInt() method of scanner to read the number of nodes from the user, and store it in a variable, noNodes. Send that number to the server.

c- Next, the server will send a string that contains (noNodes X noNodes) double values. These values represent the weights of the edges. The first noNodes values are the delays between first node and all other nodes. The next noNodes values are the delays between second node and all other nodes, etc. These values are randomly generated in the range 100 – 1000 ms. Note that the network is not totally connected, meaning that there is not always a physical connection between each pair of nodes. If no physical edge exists between a pair of nodes, a value of Infinity would be assigned to that node.

To read these values from the server, use readLine() of BufferedReader to read the string.

d- Create an adjacency matrix to store the values read from the server.

**double**[][] matrix = **new double**[noNodes][noNodes];

e- Use StringTokenizer to store the values read from the server in matrix. The first noNodes values should be stored in the first row of matrix. Next noNodes values should be in the second row of matrix and so on. Note that the values read would be of type String, and matrix is of type **double**. Hence, you need to convert the strings to **double** values.

f- Now, create a list to store the nodes. All nodes are now in nodeList, namely node 0 to node noNodes -1

List<Node> nodeList = **new** ArrayList<Node>();

**for**(**int** i = 0; i < noNodes; i++){

nodeList.add(**new** Node(i));

}

**3- Implementation of Dijkstra’s methods**

In the class RoutingClient, create the following three methods. The implementation of the first method adjacenyToEdges is provided. Your task is to write the implementation of the remaining two methods as explained.

a- adjacenyToEdges: This method is used to create the edges, given the adjacency matrix, and store them in the nodes.

**public static void** adjacenyToEdges(double[][] matrix, List<Node> v)

{

**for**(**int** i = 0; i < noNodes; i++)

{

v.get(i).neighbors = **new** Edge[noNodes];

**for**(**int** j = 0; j < noNodes; j++)

{

v.get(i).neighbors[j] = **new** Edge(v.get(j), matrix[i][j]);

}

}

}

b- **public static void** computePaths(Node source)

This method is the core of Dijkstra’s algorithm. It takes as an input a source node, and computes the paths to all other nodes in the network. The Priority Queue data structure is used for the implementation.

- set minDistance of the source to be 0.

- Define a priority queue as follows:

PriorityQueue<Node> NodeQueue = **new** PriorityQueue<Node>();

- Add the source node to the priority queue

- while the priority queue has more elements

{ - poll the node at the top of the queue, call it sourceNode

- for each edge in sourceNode (edges going out from sourceNode):

{

- Define targetNode as the target of current edge

- Define distanceThroughSource = minDistance of sourceNode + weight of the edge

- if (distanceThroughSource < minDistance of targetNode)

{

- Remove targetNode from the queue

- Set minDistance of targetNode to distanceThroughSource

- Set previous of targetNode to sourceNode;

- Add targetNode to the queue;

}

}

}

c- **public static** List<Integer> getShortestPathTo(Node targetNode)

The method computePaths implemented in step b finds the shortest paths from source node to all other nodes in the network. previous field of the node is used to keep track of the path. In this method, getShortestPathTo, we use previous to find the total path from sourceNode to targetNode. The implementation of this method is as follows:

- Define a list path to store the path from source to target:

List<Integer> path = **new** ArrayList<Integer>();

- Add targetNode to the list

- Add previous of targetNode to list

- Add previous of previous to list and so on, until previous is null.

- The path now starts from targetNode to sourceNode, you need to reverse the order so that the path starts from sourceNode to targetNode.

**4- Finding Shortest Paths**

Now, you have all the tools ready and you can find the shortest paths from any node to all other nodes. Going back to the main method of class RoutingClient, you should do the following:

- Call adjacenyToEdges with passing the proper arguments.

- for each node in the network, do the following:

{ - Call computePaths with passing the current node.

- Call getShortestPathTo with passing all nodes, one at a time, to find the actual path to each node.

- Print the total time from the current node to each node, along with the shortest path

- For all nodes in nodeList, reset their minDistance and previous, to be used with next node.

}

**Sample output:**

**Client:**

Connected to : localhost:9876

Enter number of nodes in the network, 0 to Quit:

6

Adjacency Matrix:

911.0 973.0 658.0 479.0 798.0 Infinity

231.0 188.0 256.0 197.0 399.0 288.0

715.0 144.0 800.0 Infinity 265.0 686.0

843.0 Infinity 159.0 562.0 309.0 626.0

935.0 Infinity 268.0 254.0 631.0 Infinity

482.0 Infinity 699.0 Infinity 653.0 333.0

Node 0

Total time to reach node 0: 0.0 ms, Path: [0]

Total time to reach node 1: 782.0 ms, Path: [0, 3, 2, 1]

Total time to reach node 2: 638.0 ms, Path: [0, 3, 2]

Total time to reach node 3: 479.0 ms, Path: [0, 3]

Total time to reach node 4: 788.0 ms, Path: [0, 3, 4]

Total time to reach node 5: 1070.0 ms, Path: [0, 3, 2, 1, 5]

Node 1

Total time to reach node 0: 231.0 ms, Path: [1, 0]

Total time to reach node 1: 0.0 ms, Path: [1]

Total time to reach node 2: 256.0 ms, Path: [1, 2]

Total time to reach node 3: 197.0 ms, Path: [1, 3]

Total time to reach node 4: 399.0 ms, Path: [1, 4]

Total time to reach node 5: 288.0 ms, Path: [1, 5]

Node 2

Total time to reach node 0: 375.0 ms, Path: [2, 1, 0]

Total time to reach node 1: 144.0 ms, Path: [2, 1]

Total time to reach node 2: 0.0 ms, Path: [2]

Total time to reach node 3: 341.0 ms, Path: [2, 1, 3]

Total time to reach node 4: 265.0 ms, Path: [2, 4]

Total time to reach node 5: 432.0 ms, Path: [2, 1, 5]

Node 3

Total time to reach node 0: 534.0 ms, Path: [3, 2, 1, 0]

Total time to reach node 1: 303.0 ms, Path: [3, 2, 1]

Total time to reach node 2: 159.0 ms, Path: [3, 2]

Total time to reach node 3: 0.0 ms, Path: [3]

Total time to reach node 4: 309.0 ms, Path: [3, 4]

Total time to reach node 5: 591.0 ms, Path: [3, 2, 1, 5]

Node 4

Total time to reach node 0: 643.0 ms, Path: [4, 2, 1, 0]

Total time to reach node 1: 412.0 ms, Path: [4, 2, 1]

Total time to reach node 2: 268.0 ms, Path: [4, 2]

Total time to reach node 3: 254.0 ms, Path: [4, 3]

Total time to reach node 4: 0.0 ms, Path: [4]

Total time to reach node 5: 700.0 ms, Path: [4, 2, 1, 5]

Node 5

Total time to reach node 0: 482.0 ms, Path: [5, 0]

Total time to reach node 1: 843.0 ms, Path: [5, 2, 1]

Total time to reach node 2: 699.0 ms, Path: [5, 2]

Total time to reach node 3: 907.0 ms, Path: [5, 4, 3]

Total time to reach node 4: 653.0 ms, Path: [5, 4]

Total time to reach node 5: 0.0 ms, Path: [5]

Quit

**Server:**

Server online.  
Host name: localhost  
Host Address: [xxx.xxx.x.xx:9876](http://192.168.0.12:9876)  
waiting for requests.  
request received. request number: 1 client: /[127.0.0.1:58184](http://127.0.0.1:58184)  
connection established:  
service type:ROUTING\_SERVER

mode:VERBOSE

client id:1

socket: Socket[addr=/127.0.0.1,port=53466,localport=9876]

[19:00:31] Number of nodes in the network is 6

[19:00:31] Adjacency Matrix

911.0 973.0 658.0 479.0 798.0 Infinity

231.0 188.0 256.0 197.0 399.0 288.0

715.0 144.0 800.0 Infinity 265.0 686.0

843.0 Infinity 159.0 562.0 309.0 626.0

935.0 Infinity 268.0 254.0 631.0 Infinity

482.0 Infinity 699.0 Infinity 653.0 333.0

[19:00:31] Node 0

Total time to reach node 0: 0.0 ms, Path: [0]

Total time to reach node 1: 782.0 ms, Path: [0, 3, 2, 1]

Total time to reach node 2: 638.0 ms, Path: [0, 3, 2]

Total time to reach node 3: 479.0 ms, Path: [0, 3]

Total time to reach node 4: 788.0 ms, Path: [0, 3, 4]

Total time to reach node 5: 1070.0 ms, Path: [0, 3, 2, 1, 5]

[19:00:31] Node 1

Total time to reach node 0: 231.0 ms, Path: [1, 0]

Total time to reach node 1: 0.0 ms, Path: [1]

Total time to reach node 2: 256.0 ms, Path: [1, 2]

Total time to reach node 3: 197.0 ms, Path: [1, 3]

Total time to reach node 4: 399.0 ms, Path: [1, 4]

Total time to reach node 5: 288.0 ms, Path: [1, 5]

[19:00:31] Node 2

Total time to reach node 0: 375.0 ms, Path: [2, 1, 0]

Total time to reach node 1: 144.0 ms, Path: [2, 1]

Total time to reach node 2: 0.0 ms, Path: [2]

Total time to reach node 3: 341.0 ms, Path: [2, 1, 3]

Total time to reach node 4: 265.0 ms, Path: [2, 4]

Total time to reach node 5: 432.0 ms, Path: [2, 1, 5]

[19:00:31] Node 3

Total time to reach node 0: 534.0 ms, Path: [3, 2, 1, 0]

Total time to reach node 1: 303.0 ms, Path: [3, 2, 1]

Total time to reach node 2: 159.0 ms, Path: [3, 2]

Total time to reach node 3: 0.0 ms, Path: [3]

Total time to reach node 4: 309.0 ms, Path: [3, 4]

Total time to reach node 5: 591.0 ms, Path: [3, 2, 1, 5]

[19:00:31] Node 4

Total time to reach node 0: 643.0 ms, Path: [4, 2, 1, 0]

Total time to reach node 1: 412.0 ms, Path: [4, 2, 1]

Total time to reach node 2: 268.0 ms, Path: [4, 2]

Total time to reach node 3: 254.0 ms, Path: [4, 3]

Total time to reach node 4: 0.0 ms, Path: [4]

Total time to reach node 5: 700.0 ms, Path: [4, 2, 1, 5]

[19:00:31] Node 5

Total time to reach node 0: 482.0 ms, Path: [5, 0]

Total time to reach node 1: 843.0 ms, Path: [5, 2, 1]

Total time to reach node 2: 699.0 ms, Path: [5, 2]

Total time to reach node 3: 907.0 ms, Path: [5, 4, 3]

Total time to reach node 4: 653.0 ms, Path: [5, 4]

Total time to reach node 5: 0.0 ms, Path: [5]

connection to 1 closed.