CS 6385.0W1

Algorithmic Aspects of Telecommunication Network

Summer 2017

Project 1

A Basic Network Design Model

by

Bhakti Khatri [brk160030]

Introduction

In this project, a basic network design model is implemented. Given the unit cost of creating a link between pair of nodes and demand values between each pair, this network model calculates the optimum cost of the network satisfying the traffic demands of pair of nodes in the graph. The output of this design model is a minimum cost network topology with assigned capacities.

Problem Description

Consider the following network design problem. Given *N* nodes and a demand of transporting data from node *i* to node *j* (*i,j* = 1*,...,N, i* 6= *j*) at a speed of *bij* Mbit/s.

We can build links between any pair of nodes. The cost for unit capacity (=1 Mbit/s) on a link from node *i* to *j* is *aij.* Higher capacity costs proportionally more, lower capacity costs proportionally less.

Set *aii* = 0*, bii* = 0 for all *i* so we do not have to take care of the case when *i* = *j* in the formulas.

The goal is to design which links will be built and with how much capacity, so that the given demand can be satisfied and the overall cost is minimum.

Hence, following are the inputs:

(1) <u>A complete, bidirectional graph G</u>
(Assumption: None of the edges should be excluded in advance)

(2) <u>Unit cost matrix</u>

- a_{ij} is the unit cost of creating a link with unit bandwidth between node i and node j
- Cost of a link is linearly proportional to the bandwidth of the link. Hence, if we create a link with 5 Mbps, it will cost 5 times the unit cost of that link
- a_{ij} is not equal to a_{ji}
- $a_{ii} = 0$

(3) <u>Traffic demand Matrix</u>

- b_{ij} is the demand of transporting data from node i to node j
- $b_{ij} = 0$ means there is no demand of transporting data from node I to node j
- $b_{ii} = 0$

Output is a network topology represented by directed graph with capacities assigned to every link.
Output will be a minimum cost network with capacities assigned to the edges thereby satisfying a given demand.
Note that the links carrying no traffic will not be built in the final optimum network graph.

Solution

Observe that the cheapest way of sending b_{ij} amount of flow from node k to l is to send it all along a path for which the sum of the link costs is minimum. If this path consists of nodes $k = i_1, i_2, \ldots, i_{r-1}, i_r = l$, then the resulting piece of cost is

$$b_{kl}(a_{i_1,i_2} + \ldots + a_{i_{r-1},i_r})$$

Due to the linear nature of the model, these costs simply sum up, independently of each other. This suggests the following simple algorithm:

Algorithm

- Find a minimum cost path between each pair k,l of nodes, with edge weights a_{ij}. This can be done by any standard shortest path algorithm that you met in earlier courses. We use Dijkstra's algorithm in the implementation.

- Set the capacity of link (i,j) to the sum of those b_{kl} values for which (i,j) is on the min cost path found for k,l.

- The optimum cost can be expressed explicitly. Let E_{kl} be the set of edges that are on the min cost $k \to l$ path. Then, according to the above, the optimal cost is:

$$Z_{opt} = \sum_{k,l} \left(b_{kl} \sum_{(i,j) \in E_{kl}} a_{ij} \right)$$

- ➢ The package com.bhakti.atn.dijkstra contains the implementation of Dijkstra's algorithm to calculate the shortest path.
- ➢ The package com.bhakti.atn.calculatecost contains the implementation of a simple network design.
  - ✓ NetworkDesignParameters.java is the class that calculates the cost matrix, traffic demand matrix based on the instructions given in the project description document.
  - ✓ Let the number of nodes be $N = 30$ in each example.
  - ✓ For each example, generate the a_{ij}, b_{ij} values according to the rules described below. In these rules k is a parameter that will change in the experiments.

For generating the b_{ij} values, take your 10-digit student ID, and repeat it 3times, to obtain a 30-digit number. For example, if the ID is 0123456789, then after repetition it becomes 012345678901234567890123456789. Let $d_1, d_2, \ldots, d_{30}$ denote the digits in this 30-digit number. Then the value of b_{ij} is computed by the formula $b_{ij} = |d_i - d_j|$.

- ✓ For example, using the above sample ID, the value of $b_{3,7}$ will be $b_{3,7} = |d_3 - d_7| = |2 - 6| = 4$.
- ✓ For generating the a_{ij} values, do the following. For any given i, pick k random indices $j_1, j_2, \ldots, j_k$, all

different from each other and from *i*. Then set

$aij1 = aij2 = ... = aijk = 1,$

and set $a_{ij} = 300$, whenever j /= $j_1,...,j_k$.

✓ Carry out this independently for every *i*.

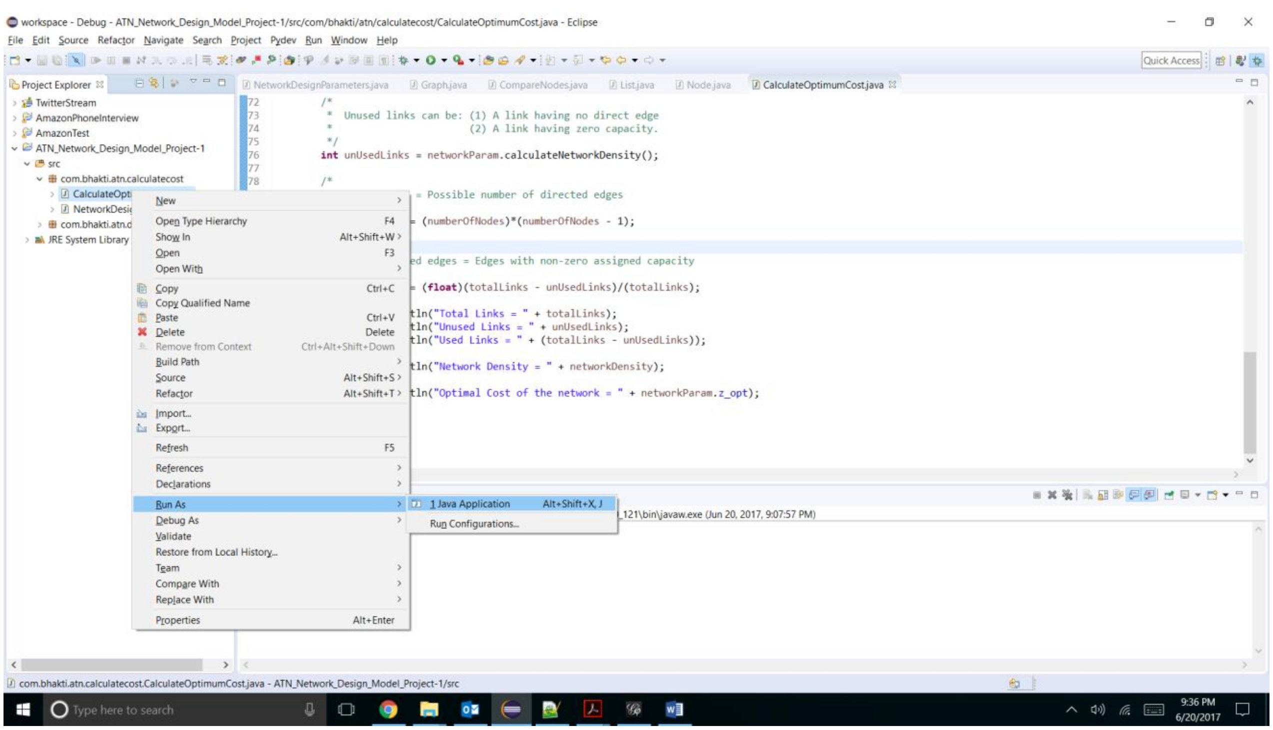
Remark: The effect of this is that for every node *i* there will be *k* low cost links going out of the node, the others will have large cost. The shortest path algorithm will try to avoid the high cost links, so it effectively means that we limit the number of links that go out of the node, thus limiting the network density.

✓ Run your program with k = 3,4,5...,15. For each run generate new random a_{ij}, b_{ij} parameters independently.

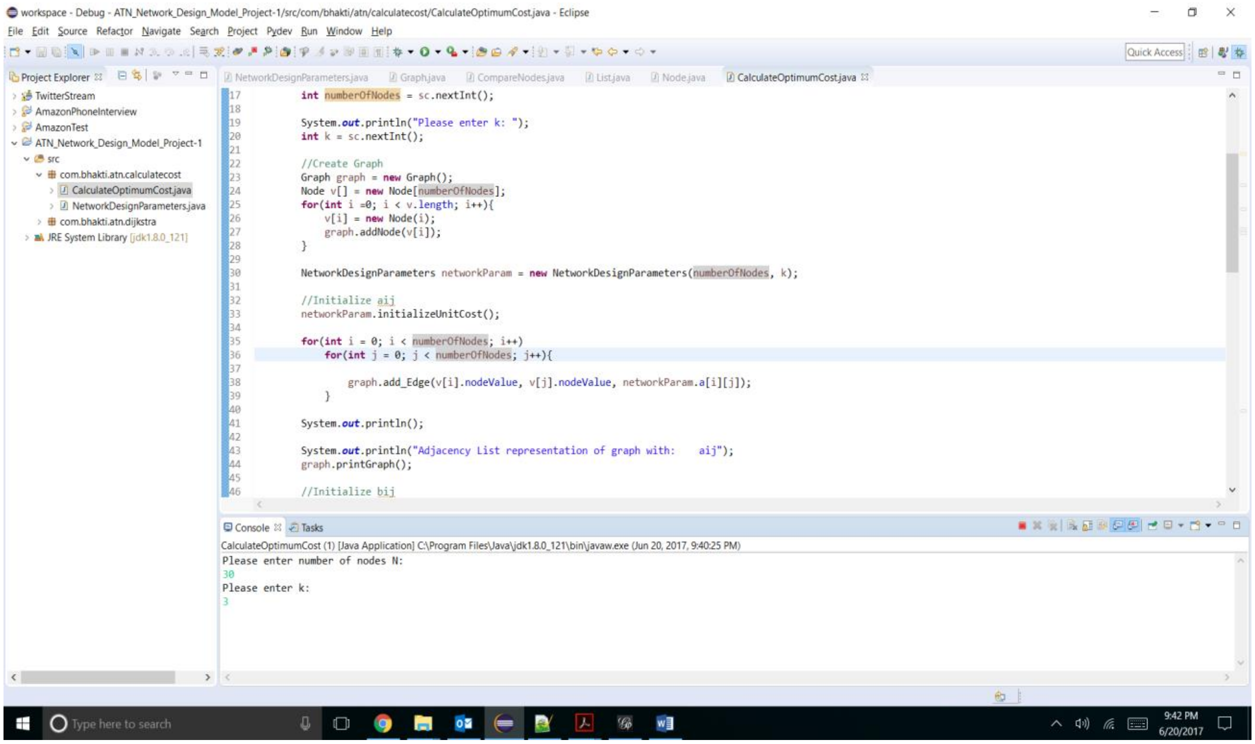
➢ CalculateOptimumCost.java is the driver class which accepts the number of nodes and value of k from the user and as an output it gives a network topology with assigned capacities. Since the parameter k will be chosen randomly, that is, k random outgoing edges for a particular source will be chosen. The program also calculates the optimum cost of the network and the network density.

Instructions on how to run a program

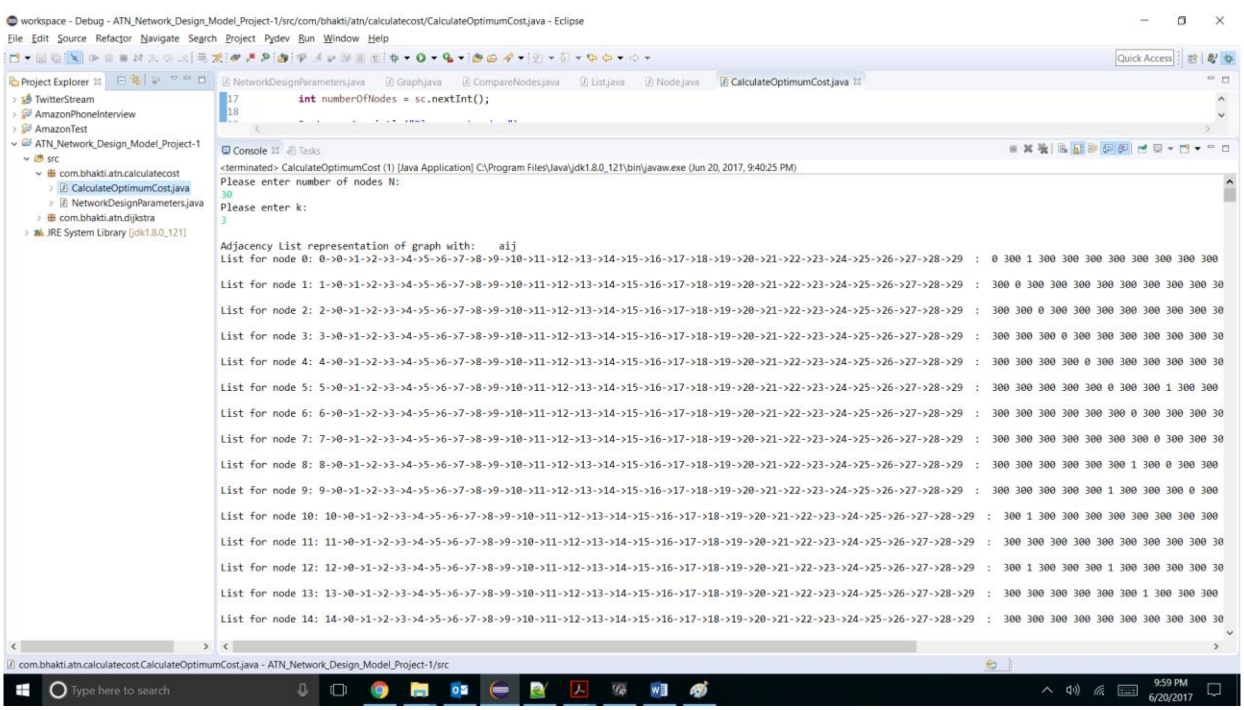
(1) Run CalculateOptimumCost.java of the package com.bhakti.atn.calculatecost

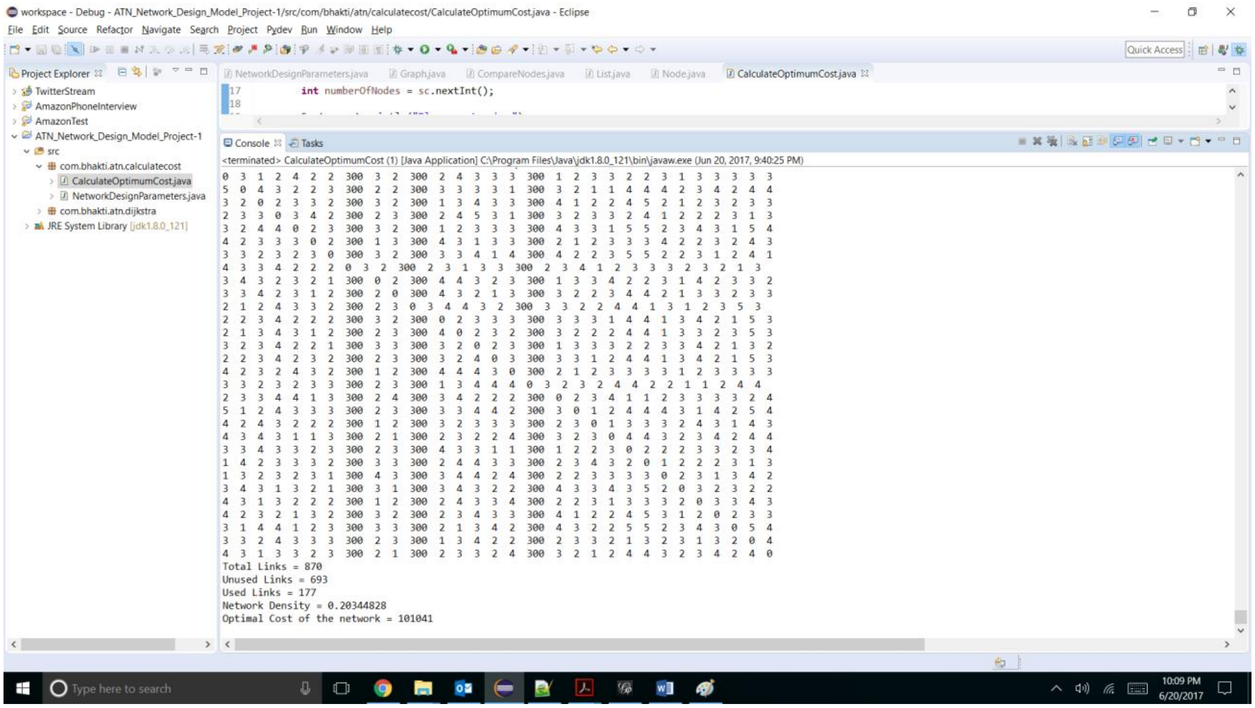


(2) Enter the number of nodes and value of k

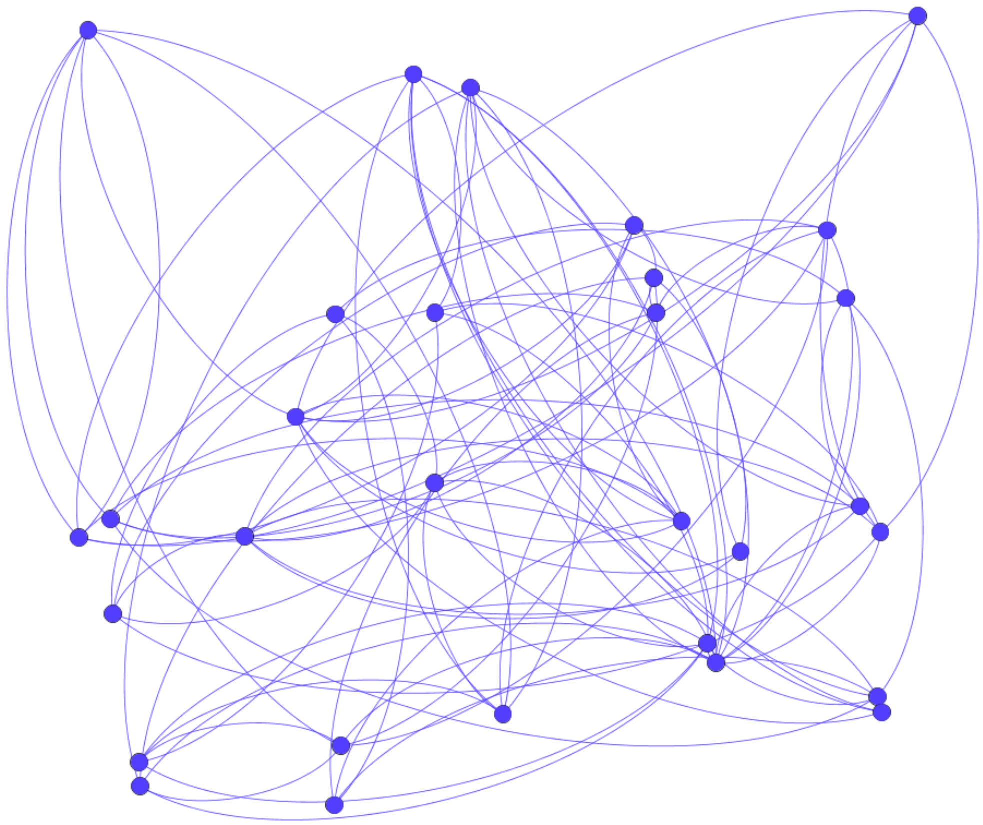


(3) Sample Output

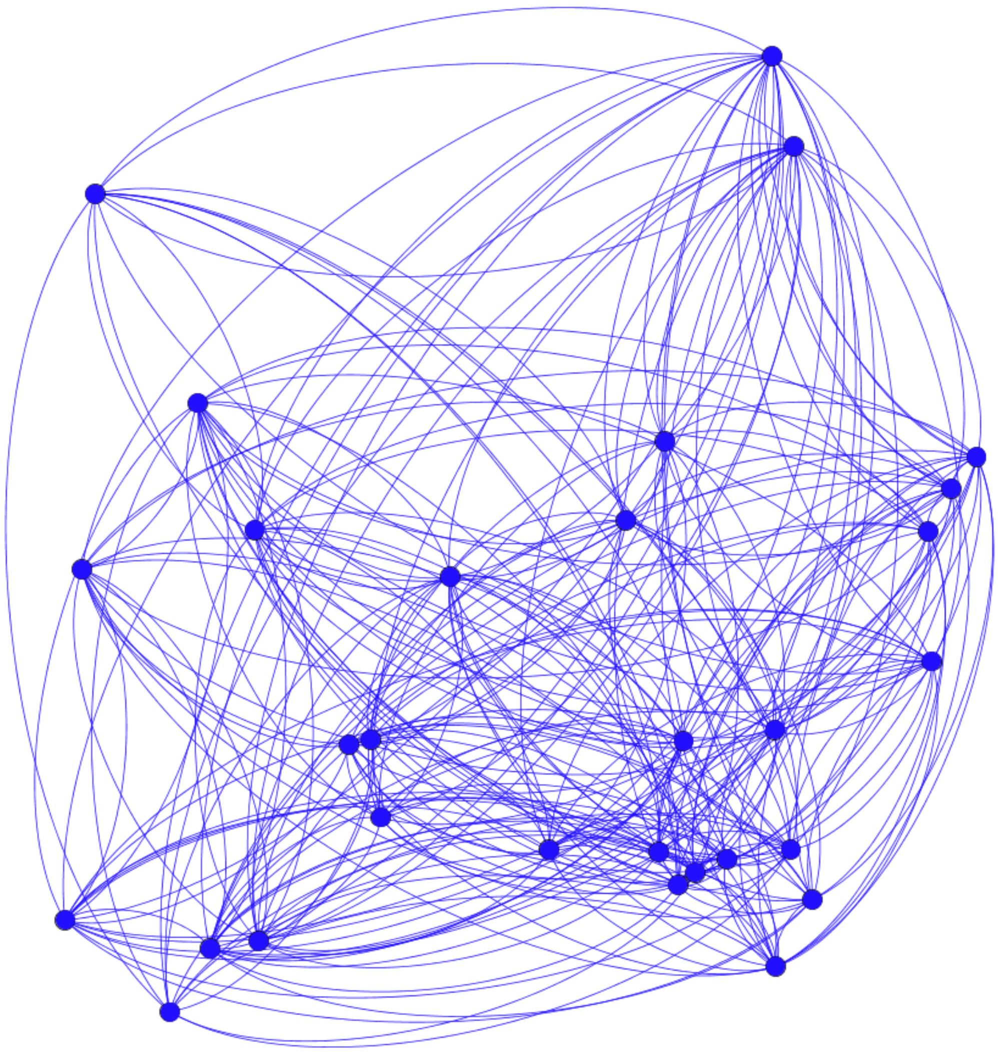



workspace - Debug - ATN_Network_Design_Model_Project-1/src/com/bhakti/atn/calculatecost/CalculateOptimumCost.java - Eclipse
File Edit Source Refactor Navigate Search Project Pydev Run Window Help
Quick Access
Project Explorer
TwitterStream
AmazonPhoneInterview
AmazonTest
ATN_Network_Design_Model_Project-1
src
com.bhakti.atn.calculatecost
CalculateOptimumCost.java
NetworkDesignParameters.java
com.bhakti.atn.dijkstra
JRE System Library [jdk1.8.0_121]
NetworkDesignParameters.java
Graph.java
CompareNodes.java
List.java
Node.java
CalculateOptimumCost.java
17 int numberOfNodes = sc.nextInt();
18
Console
Tasks
<terminated> CalculateOptimumCost (1) [Java Application] C:\Program Files\Java\jdk1.8.0_121\bin\javaw.exe (Jun 20, 2017, 9:40:25 PM)
0 3 1 2 4 2 2 300 3 2 300 2 4 3 3 3 300 1 2 3 3 2 2 3 1 3 3 3 3 3
5 0 4 3 2 2 3 300 2 2 300 3 3 3 3 1 300 3 2 1 1 4 4 4 2 3 4 2 4 4
3 2 0 2 3 3 2 300 3 2 300 1 3 4 3 3 300 4 1 2 2 4 5 2 1 2 3 2 3 3
2 3 3 0 3 4 2 300 2 3 300 2 4 5 3 1 300 3 2 3 3 2 4 1 2 2 2 3 1 3
3 2 4 4 0 2 3 300 3 2 300 1 2 3 3 3 300 4 3 3 1 5 5 2 3 4 3 1 5 4
4 2 3 3 3 0 2 300 1 3 300 4 3 1 3 3 300 2 1 2 3 3 3 4 2 2 3 2 4 3
3 3 2 3 2 3 0 300 3 2 300 3 3 4 1 4 300 4 2 2 3 5 5 2 2 3 1 2 4 1
4 3 3 4 2 2 2 0 3 2 300 2 3 1 3 3 300 2 3 4 1 2 3 3 3 2 3 2 1 3
3 4 3 2 3 2 1 300 0 2 300 4 4 3 2 3 300 1 3 3 4 2 2 3 1 4 2 3 3 2
3 3 4 2 3 1 2 300 2 0 300 4 3 2 1 3 300 3 2 2 3 4 4 2 1 3 3 2 3 3
2 1 2 4 3 3 2 300 2 3 0 3 4 4 3 2 300 3 3 2 2 4 4 1 3 1 2 3 5 3
2 2 3 4 2 2 2 300 3 2 300 0 2 3 3 3 300 3 3 3 1 4 4 1 3 4 2 1 5 3
2 1 3 4 3 1 2 300 2 3 300 4 0 2 3 2 300 3 2 2 2 4 4 1 3 3 2 3 5 3
3 2 3 4 2 2 1 300 3 3 300 3 2 0 2 3 300 1 3 3 3 2 2 3 3 4 2 1 3 2
2 2 3 4 2 3 2 300 2 3 300 3 2 4 0 3 300 3 3 1 2 4 4 1 3 4 2 1 5 3
4 2 3 2 4 3 2 300 1 2 300 4 4 4 3 0 300 2 1 2 3 3 3 3 1 2 3 3 3 3
3 3 2 3 2 3 3 300 2 3 300 1 3 4 4 4 0 3 2 3 2 4 4 2 2 1 1 2 4 4
2 3 3 4 4 1 3 300 2 4 300 3 4 2 2 2 300 0 2 3 4 1 1 2 3 3 3 3 2 4
5 1 2 4 3 3 3 300 2 3 300 3 3 4 4 2 300 3 0 1 2 4 4 4 3 1 4 2 5 4
4 2 4 3 2 2 2 300 1 2 300 3 2 3 3 3 300 2 3 0 1 3 3 3 2 4 3 1 4 3
4 3 4 3 1 1 3 300 2 1 300 2 3 2 2 4 300 3 2 3 0 4 4 3 2 3 4 2 4 4
3 3 4 3 3 2 3 300 2 3 300 4 3 3 1 1 300 1 2 2 3 0 2 2 2 3 3 2 3 4
1 4 2 3 3 3 2 300 3 3 300 2 4 4 3 3 300 2 3 4 3 2 0 1 2 2 2 3 1 3
1 3 2 3 2 3 1 300 4 3 300 3 4 4 2 4 300 2 2 3 3 3 3 0 2 3 1 3 4 2
3 4 3 1 3 2 1 300 3 1 300 3 4 3 2 2 300 4 3 3 4 3 5 2 0 3 2 3 2 2
4 3 1 3 2 2 2 300 1 2 300 2 4 3 3 4 300 2 2 3 1 3 3 3 2 0 3 3 4 3
4 2 3 2 1 3 2 300 3 2 300 2 3 4 3 3 300 4 1 2 2 4 5 3 1 2 0 2 3 3
3 1 4 4 1 2 3 300 3 3 300 2 1 3 4 2 300 4 3 2 2 5 5 2 3 4 3 0 5 4
3 3 2 4 3 3 3 300 2 3 300 1 3 4 2 2 300 2 3 3 2 1 3 2 3 1 3 2 0 4
4 3 1 3 3 2 3 300 2 1 300 2 3 3 2 4 300 3 2 1 2 4 4 3 2 3 4 2 4 0
Total Links = 870
Unused Links = 693
Used Links = 177
Network Density = 0.20344828
Optimal Cost of the network = 101041
Type here to search
10:09 PM
6/20/2017

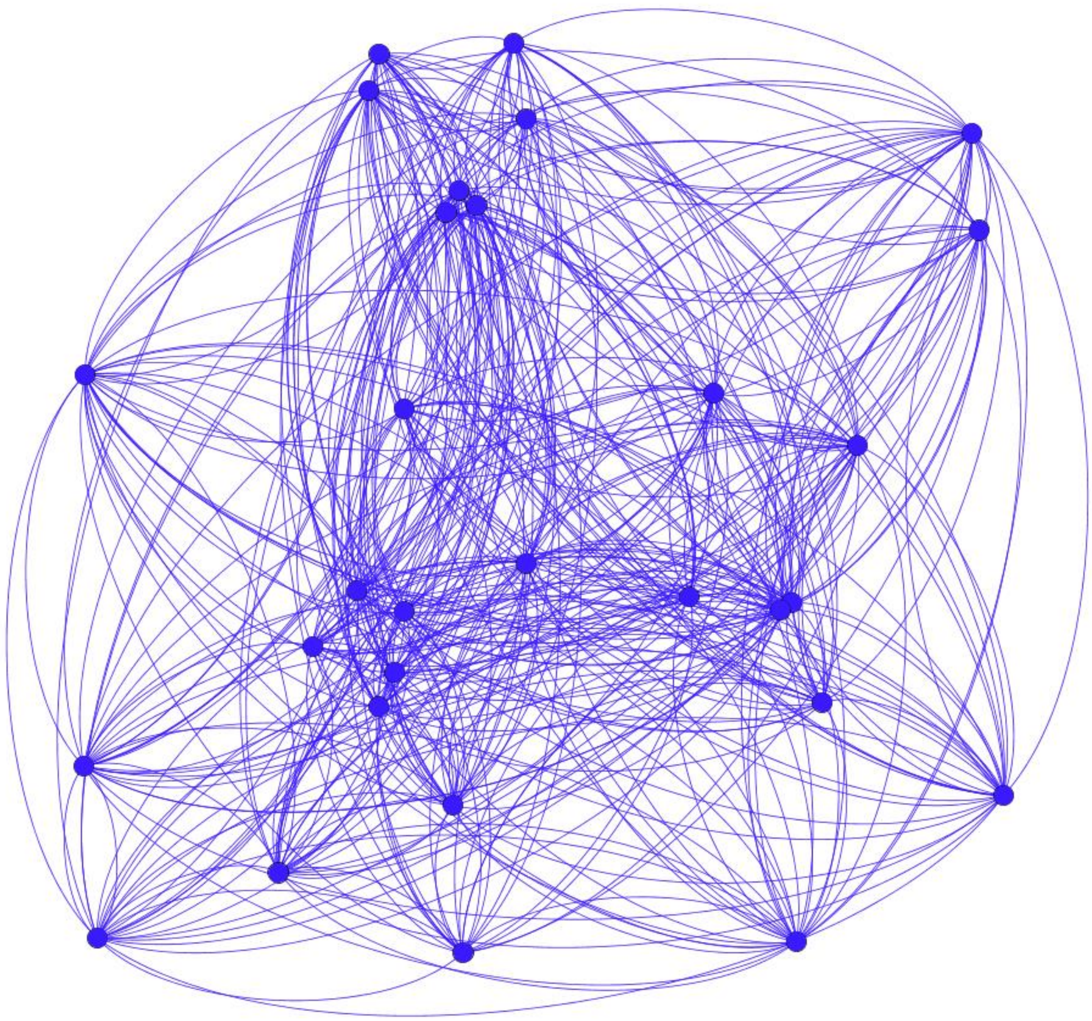
Graphical Representation of a network topology



k	3
Total links	870
Unused links	780
Used Links	90
Network Density	0.10344828
Optimum cost	8266



k	9
Total links	870
Unused links	600
Used Links	270
Network Density	0.31034482
Optimum cost	4702



k	15
Total links	870
Unused links	420
Used Links	450
Network Density	0.51724136
Optimum cost	4038

Observations and Analysis

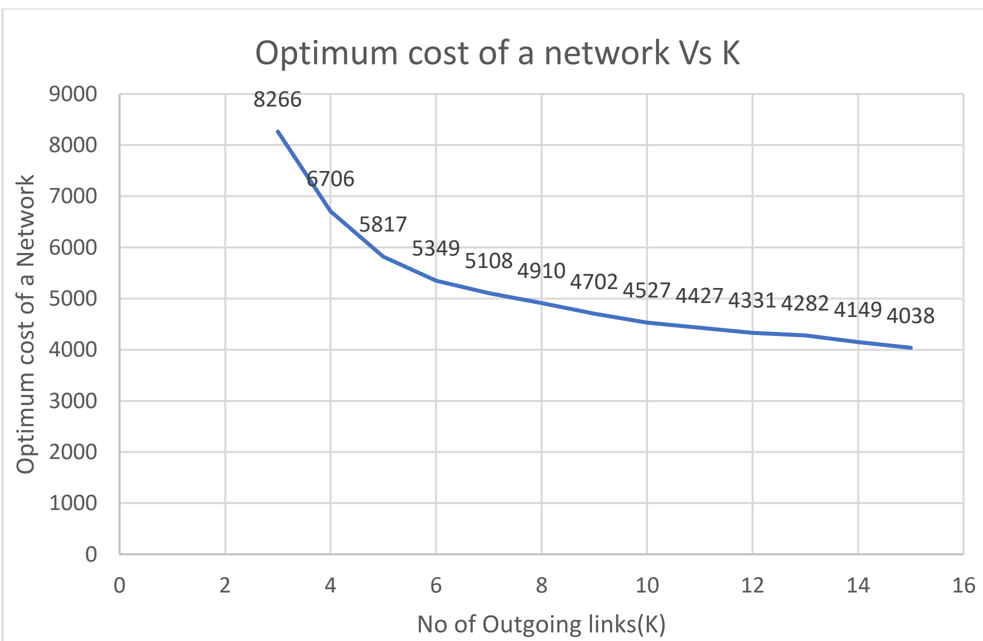
Number of nodes (N)	No of outgoing links (K)	Total Links [N * (N - 1)]	Unused Links	Used Links (Directed Edges)	Network Desnsity	Optimum cost of a network
30	3	870	780	90	0.10344828	8266
30	4	870	750	120	0.13793103	6706
30	5	870	720	150	0.1724138	5817
30	6	870	690	180	0.20689656	5349
30	7	870	660	210	0.2413793	5108
30	8	870	630	240	0.27586207	4910
30	9	870	600	270	0.31034482	4702
30	10	870	570	300	0.3448276	4527
30	11	870	540	330	0.37931034	4427
30	12	870	510	360	0.41379312	4331
30	13	870	480	390	0.44827586	4282
30	14	870	450	420	0.4827586	4149
30	15	870	420	450	0.51724136	4038

- By using the fact that link costs are directly proportional to the bandwidth requirement of the link (let's say capacity of a link), we have used shortest path algorithm in which link weights are simply the unit link costs.
- Hence, for every source and destination pair, we pick up a path with minimum cost and pass the flow through it.
- Let k be the number of outgoing links for a particular source having the unit link costs.

  The program is designed in such a way that depending on the value of k, the number of links equal to k would be the outgoing links for a source with minimum link cost (weight of such edges in the graph is taken 1). All other outgoing links are assumed to have higher link cost (weight of such edges in the graph is taken 300).
- The parameter k significantly controls following:
  - ✓ The cost of the network
  - ✓ Network density
- Based on the results from the implementation of basic network design using shortest path based fast solution, we can draw some conclusions which are described with the help of plot graph.

(1) Effect of k on the optimum cost of a network

No of outgoing links (K)	Optimum cost of a network
3	8266
4	6706
5	5817
6	5349
7	5108
8	4910
9	4702
10	4527
11	4427
12	4331
13	4282
14	4149
15	4038

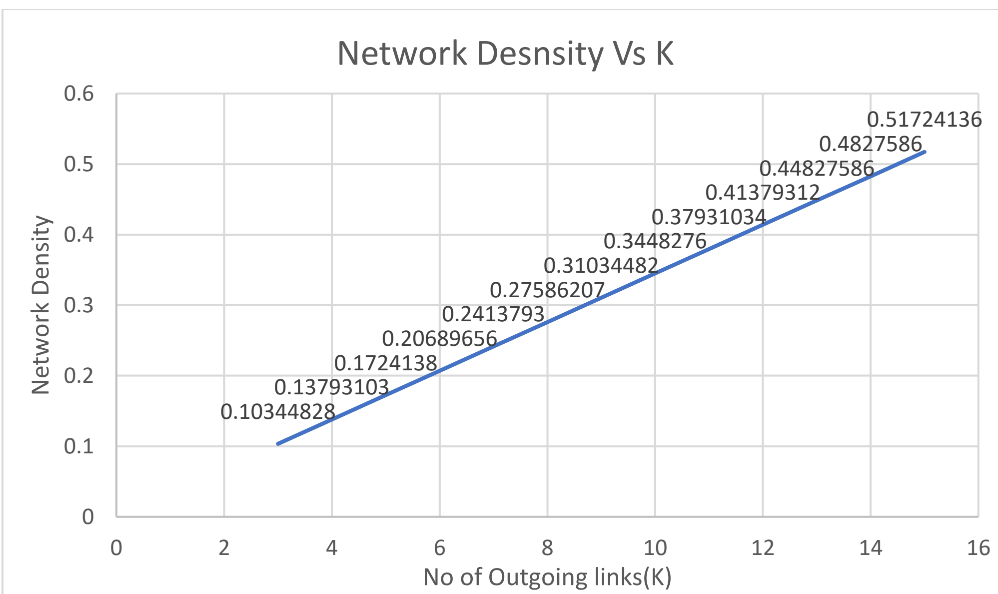


From the graph, we observe that cost of a network depends largely on the value of k. As the number of outgoing links from the source increases, maximum number of the edges coming in the shortest path are used. This leads to optimum network topology. Therefore, the cost of the total network gets reduced.

Therefore, we observe that the parameter k controls the total cost of a network. If k is small then less number of cheap links or low cost links leave the source therefore cost of the network is comparatively more, whereas if k is large then the number of links leaving the source are more thereby obtaining the network with low cost.

(2) Effect of k on network density

No of outgoing links (K)	Network Desnsity
3	0.10344828
4	0.13793103
5	0.1724138
6	0.20689656
7	0.2413793
8	0.27586207
9	0.31034482
10	0.3448276
11	0.37931034
12	0.41379312
13	0.44827586
14	0.4827586
15	0.51724136



Network density is defined the ratio of actual connections to potential connection. In other words, we can define network density as the ratio of number of used links to the total number of links in the network. The parameter k controls the network density. A value of the network density lies between 0 and 1. From the graph we observe that there is linear relationship between the value of k and the network density. As the number of outgoing links from the source increases, more number of links that fall on the shortest paths are used. Hence, the density of the network increases.

For N = 30, if we choose k =29 we would get maximum network density, that is, we would get a network topology with maximum number of links built that satisfy the traffic demands with minimum cost. In this case the network density becomes 1.

APPENDIX

Source Code:

CalculateOptimumCost.java

```
package com.bhakti.atn.calculatecost;

import java.util.Scanner;

import com.bhakti.atn.dijkstra.Graph;
import com.bhakti.atn.dijkstra.Node;

public class CalculateOptimumCost {


    public static void main(String[] args) {

            @SuppressWarnings("resource")
            Scanner sc = new Scanner(System.in);

            System.out.println("Please enter number of nodes N: ");
            int numberOfNodes = sc.nextInt();

            System.out.println("Please enter k: ");
            int k = sc.nextInt();

            //Create Graph
            Graph graph = new Graph();
            Node v[] = new Node[numberOfNodes];
            for(int i =0; i < v.length; i++){
                    v[i] = new Node(i);
                    graph.addNode(v[i]);
            }

            NetworkDesignParameters networkParam = new
```

```
NetworkDesignParameters(numberOfNodes, k);

		//Initialize aij
		networkParam.initializeUnitCost();

		for(int i = 0; i < numberOfNodes; i++)
			for(int j = 0; j < numberOfNodes; j++){

				graph.add_Edge(v[i].nodeValue, v[j].nodeValue, networkParam.a[i][j]);
			}

		System.out.println();

		System.out.println("Adjacency List representation of graph with:		aij");
		graph.printGraph();

		//Initialize bij
		networkParam.initializeTrafficDemand();

		System.out.println();
		System.out.println("Dijkstra's Algorithm Results:");

		//Calculate cost of each link
		for(int i = 0; i < v.length; i++){
			networkParam.calculateLinkCost(graph, v[i].nodeValue);
		}

		/*
		 * Following function prints the matrix which stores the minimum cost from the source.
		 */
		System.out.println("Minimum Unit Path Cost Matrix - based on shortest paths");
		networkParam.printUnitMinPathCostMatrix();



		//Calculate the network density.

		/*
		 *  Network Density = (No of directed Edges) / (Possible number of directed edges)
		 */
		float networkDensity = 0;

		/*
		 *  Unused links can be: (1) A link having no direct edge
		 *			(2) A link having zero capacity.
		 */
		int unUsedLinks = networkParam.calculateNetworkDensity();
```

```
		/*
		 * Total links  = Possible number of directed edges
		 */
		int totalLinks = (numberOfNodes)*(numberOfNodes - 1);

		/*
		 * No of directed edges = Edges with non-zero assigned capacity
		 */
		networkDensity = (float)(totalLinks - unUsedLinks)/(totalLinks);

		System.out.println("Total Links = " + totalLinks);
		System.out.println("Unused Links = " + unUsedLinks);
		System.out.println("Used Links = " + (totalLinks - unUsedLinks));

		System.out.println("Network Density = " + networkDensity);

		System.out.println("Optimal Cost of the network = " + networkParam.z_opt);

	}

}
```

NetworkDesignParameters.java:

```
package com.bhakti.atn.calculatecost;

import java.util.ArrayList;
import java.util.Collections;

import com.bhakti.atn.dijkstra.Graph;

public class NetworkDesignParameters {

	/*
	 * Unit Cost Matrix
	 */
	int a[][];

	/*
	 * Traffic Demand Matrix
	 */
	int b[][];

	/*
	 * Unit cost of each link via shortest path
	 */
	int unitMinPathCost[][];
```

```
/*
 * Cost of each link via shortest path
 */
int minPathCost[][];

/*
 * Optimum cost of network
 */
int z_opt = 0;

static int k;

/*
 * Number of nodes
 */
static int numberOfNodes;

public NetworkDesignParameters(int numberOfNodes, int k) {

        NetworkDesignParameters.numberOfNodes = numberOfNodes;
        NetworkDesignParameters.k = k;

        unitMinPathCost = new int[numberOfNodes][numberOfNodes];
        minPathCost = new int[numberOfNodes][numberOfNodes];

}

/*
 * Initializes unit cost for each link
 */
public void initializeUnitCost() {

        a = new int[numberOfNodes][numberOfNodes];
        for(int i = 0; i < numberOfNodes; i++){

                for (int j = 0; j < numberOfNodes; j++) {
                        if(i == j)
                                a[i][j] = 0;
                        else
                                a[i][j] = 300;
                }
        }

        for(int i = 0; i < numberOfNodes; i++){

                ArrayList<Integer> list = new ArrayList<Integer>();
    for (int m=0; m < numberOfNodes; m++) {
        if(i != m)
```

```
                    list.add(new Integer(m));
            }

            Collections.shuffle(list);

            for (int p=0; p < k; p++) {
               //System.out.println(""+i+"---"+list.get(p));
                 //System.out.println(i);
                 //System.out.println(list.get(p));
               a[i][list.get(p)] = 1;
            }
                 }
        }

        /*
         * Initializes Traffic Demand for each link
         */
        public void initializeTrafficDemand() {
                //int studentID1[] = {2, 0, 2, 1, 3, 0};

                int studentID1[] = {2, 0, 2, 1, 3, 0, 9, 3, 1, 8,2, 0, 2, 1, 3, 0, 9, 3, 1, 8,2, 0, 2, 1, 3, 0, 9, 3, 1,
8};

                b = new int[numberOfNodes][numberOfNodes];

                System.out.println("Input Traffic demand matrix bij");
                for(int i = 0; i < b.length; i++) {
                        for(int j = 0; j < b.length; j++){

                                b[i][j] = Math.abs(studentID1[i] - studentID1[j]);
                                //System.out.println("Traffic demand of " + " ("+(i) + " , " + (j) + ") :" +
b[i][j]);
                                System.out.print(b[i][j] + " ");
                                }
                        System.out.println();
                }
        }

        /*
         * Calculates total cost for each link based on shortest path
         */
        public void calculateLinkCost(Graph graph, int source) {

                graph.createMinHeap(numberOfNodes, source);
                graph.calculateShortestPath();
```

```
		System.out.println();
		System.out.println("Source " + source);
		for(int i = 0; i < numberOfNodes; i++){
			if(source != i) {
				unitMinPathCost[source][i] = graph.printDistanceParent(i);
				minPathCost[source][i] = unitMinPathCost[source][i] * b[source][i];

				//System.out.print(unitMinPathCost[source][i]);
				//System.out.println();
				z_opt += minPathCost[source][i];


				System.out.println("Cost of the link: "+ "("+(+source+ ", "+ i) +"):" +
minPathCost[source][i]);
			}
		}
		System.out.println();
	}

	public void printUnitMinPathCostMatrix(){
		for(int i =0; i<numberOfNodes; i++){
			for(int j=0; j<numberOfNodes; j++){
				System.out.print(unitMinPathCost[i][j]+"  ");
			}
			System.out.println();
		}
	}

	/*
	 *  Calculates network density. Network density = (No of directed edges)/(no of nodes)(no of
nodes - 1)
	 */
	public int calculateNetworkDensity(){
		int count = 0;
		for(int i =0; i< numberOfNodes; i++){
			for(int j =0; j< numberOfNodes; j++){
				//condition to check unused link
				if (i != j & (unitMinPathCost[i][j] == 0 || (unitMinPathCost[i][j] != 1 &
unitMinPathCost[i][j] != 300))){
					count ++;

				}
			}
		}
		return count;
	}
```

```
}
```

Graph.java:

```java
package com.bhakti.atn.dijkstra;

import java.util.Comparator;
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.Map;
import java.util.PriorityQueue;
import java.util.Set;

public class Graph {

	/*
	 * This map stores the Linked list for each node in the graph
	 */
	Map<Integer, List> map = new HashMap<Integer, List>();

	/*
	 * This map stores the 'visited' status of each node in the graph
	 */
	Map<Integer, Boolean> visited = new HashMap<Integer, Boolean>();

	/*
	 * This is a Min-Heap
	 */
	PriorityQueue<Node> queue;

	/*
	 * This set holds the nodes already traced i.e. whose shortest path is found
	 */
	//Set<Node> tracedNode = new LinkedHashSet<Node>();
	Set<Node> tracedNode;

	/*
	 * This array is used to store parent of each node
	 */
	int parent[] = new int[30];

	/*
	 * This method returns the parent array
	 */
	public int[] getParent() {
		return parent;
```

```java
	}


	public PriorityQueue<Node> getQueue() {
		return queue;
	}


	/*
	 * This method adds node to the graph, creates a Linked List for each node and sets the 'visited' status to false
	 */
	public void addNode(Node n){
		List list = new List(n);
		map.put(n.nodeValue, list);
		visited.put(n.nodeValue, false);
	}

	/*
	 * This method adds the weighted edge between 2 nodes in the undirected graph
	 */
	public void add_Edge(int v1, int v2, int weight) {
		List listOne = map.get(v1);
		Node headOne = listOne.head;
		while(headOne.next != null) {
			headOne = headOne.next;
		}
		Node nodeOne = new Node();
		nodeOne.nodeValue = v2;
		nodeOne.distance = weight;
		headOne.next = nodeOne;

		/*List listTwo = map.get(v2);
		Node headTwo = listTwo.head;
		while(headTwo.next != null) {
			headTwo = headTwo.next;
		}
		Node nodeTwo = new Node();
		nodeTwo.nodeValue = v1;
		nodeTwo.distance = weight;
		headTwo.next = nodeTwo;*/
	}

	/*
	 * This method creates and initializes a Min-Heap
	 */
	public void createMinHeap(int size, int start) {
		Comparator<Node> comparator = new CompareNodes();
```

```
		queue = new PriorityQueue<Node>(size, comparator);
		for(Integer i : map.keySet()) {
			if(i == start)
				queue.add(new Node(i, 0));
			else
				queue.add(new Node(i, Integer.MAX_VALUE));
		}
	}

	/*
	 * This method calculates the shortest path from source node
	 */
	public void calculateShortestPath() {
		tracedNode = new LinkedHashSet<>();
		while(!queue.isEmpty()) {
			Node start = queue.remove();
			tracedNode.add(start);

			//System.out.println("Start value: " + start.nodeValue + " Start distance: " +
start.distance);

			List list = map.get(start.nodeValue);
			Node listNode = list.head.next;

			while(listNode != null) {
				Node heapNode = locateHeapNode(listNode);
				if((heapNode != null) && ((start.distance + listNode.distance) <
heapNode.distance)) {
					queue.remove(heapNode);
					heapNode.distance = start.distance + listNode.distance;
					heapNode.parent = start;
					queue.add(heapNode);
					parent[heapNode.nodeValue] = heapNode.parent.nodeValue;
				}
				listNode = listNode.next;
			}

			//System.out.println("Heap after extracting source with value " +
start.nodeValue);
			//printHeap();
		}
	}

	/*
	 * This method locates the node in the heap
	 */
	public Node locateHeapNode(Node nodeToFind) {
		Iterator<Node> itr = queue.iterator();
```

```
        while(itr.hasNext()) {
            Node n = itr.next();
            if(n.nodeValue == nodeToFind.nodeValue) {
                return n;
            }
        }
        return null;
    }

    /*
     * This method prints the weighted graph
     */
    public void printGraph() {
        for (List list : map.values()) {
            Node itrOne = list.head;
            System.out.print("List for node " + itrOne.nodeValue + ": ");
            while(itrOne.next != null) {
                System.out.print(itrOne.nodeValue + "->");
                itrOne = itrOne.next;
            }
            System.out.print(itrOne.nodeValue);
            System.out.print("  :  ");

            Node itrTwo = list.head.next;
            while(itrTwo != null) {
                System.out.print(itrTwo.distance + " ");
                itrTwo = itrTwo.next;
            }
            System.out.println();
            System.out.println();
        }
    }

    /*
     * This method prints the priority queue
     */
    public void printHeap() {
        if(queue.size() != 0) {
            System.out.println("Priority Queue:");
            for (Node node : queue) {
                System.out.println(node.nodeValue + " " + node.distance);
            }
        } else{
            System.out.println("Heap is empty");
        }
    }

    /*
```

```
     * This method prints each node, its distance from the source and and its parent
     */
    public void printDistanceParent() {
            Iterator<Node> itr = tracedNode.iterator();
            while(itr.hasNext()) {
                    Node n = itr.next();
                    if(n.parent != null) {
                            System.out.println("Distance of node " + n.nodeValue + ": "+
n.distance);
                    } else {
                            System.out.println("Distance of node " + n.nodeValue + ": " +
n.distance);
                    }
            }
    }


    public int printDistanceParent(int destination) {
            Node n = null;
            Iterator<Node> itr = tracedNode.iterator();
            while(itr.hasNext()) {
                    n = itr.next();
                    if(n.parent != null && n.nodeValue == destination) {
                            System.out.println("Distance of node " + n.nodeValue + ": "+
n.distance);
                            break;

                    }
            }
            return n.distance;

    }
    /*
     * This method prints the shortest path for each node
     */
    public void printPath(int parent[], int j) {
            if(parent[j] == 0) {
                    System.out.print(j + " ");
                    return;
            }
            printPath(parent, parent[j]);
            System.out.print(j + " ");
    }
}
```

CompareNodes.java:

```java
package com.bhakti.atn.dijkstra;

import java.util.Comparator;

public class CompareNodes implements Comparator<Node>{

	@Override
	public int compare(Node o1, Node o2) {

		return (int) (o1.distance - o2.distance);
	}

}
```

List.java:

```java
package com.bhakti.atn.dijkstra;

public class List {

	/*
	 * This is head node for the Linked List
	 */
	Node head;

	public List() {
	}

	public List(Node v) {
		head = v;
	}

	/*
	 * This method adds a node to the queue
	 */
	public void push(int num) {
		Node temp = head;
		Node n = new Node(num);
		if(head == null) {				//If List is empty
			head = n;
			head.next = null;
			return;
		}
		while(temp.next != null) {
			temp = temp.next;				//If List is
not empty
		}
		temp.next = n;
		n.next = null;
	}

	/*
	 * This method pops the head of the queue
```

```java
         */
        public Node poll() {
                Node temp = head;
                if(head == null)
                        return head;
                head = head.next;
                return temp;
        }

        /*
         * This method calculates the size of the list
         */
        public int getSize() {
                Node temp = head;
                int count = 0;
                while(temp != null){
                        count++;
                        temp = temp.next;
                }
                return count;
        }
}
```

Node.java:

```java
package com.bhakti.atn.dijkstra;

public class Node {

    public int nodeValue;
    public int distance;
        public Node next;
        public Node parent;

        public Node() {

        }

        public Node(int value) {
                this.nodeValue = value;
        }

        public Node(int nodeValue, int distance) {
                this.nodeValue = nodeValue;
                this.distance = distance;
        }
}
```