AN APPLICATION TO NETWORK DESIGN

Algorithmic Aspect of Telecommunication Networks

OCTOBER 7, 2014

RAHUL SINGHAL

Table of Contents

Software Specifications

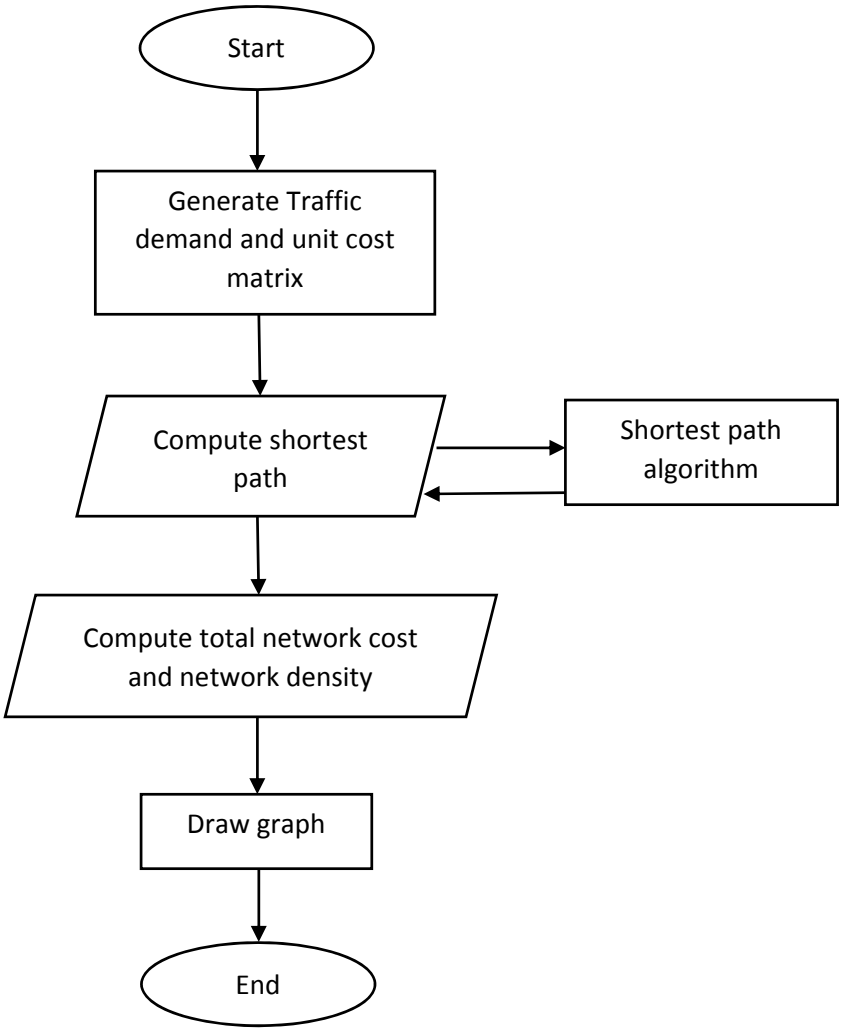
Input

- Number of nodes, N
- Traffic demand values, b_{ij}
- Unit cost values for potential links, a_{ij}

Output

- Computes total cost of the designed network
- Generate network topology

Flow chart



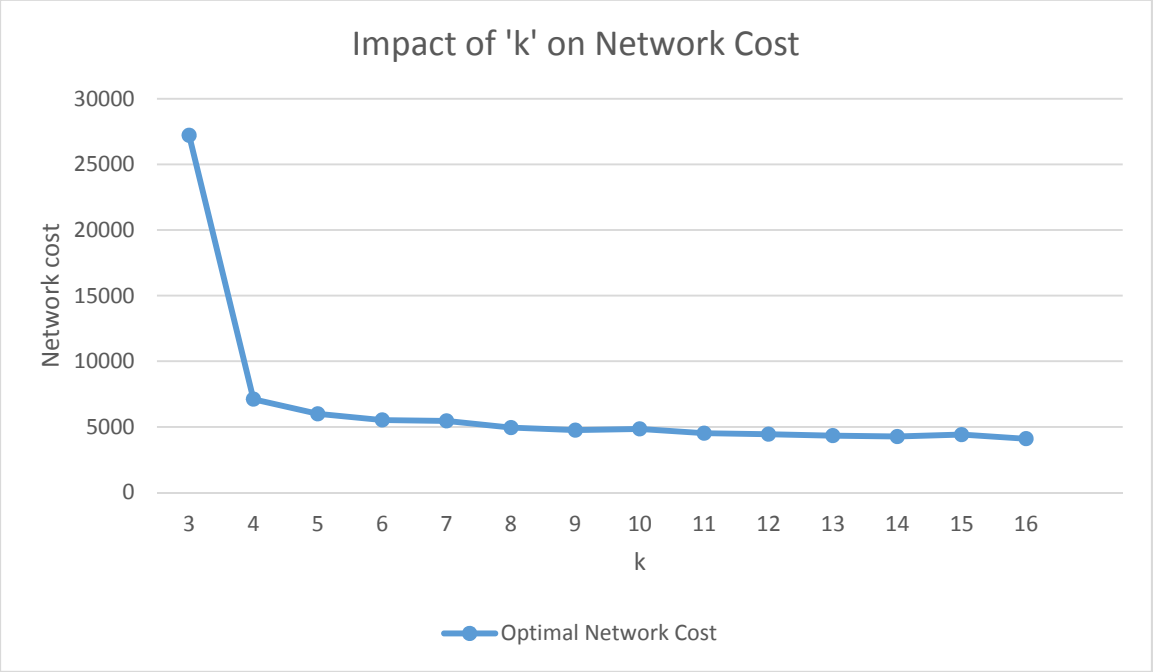
Results

Console Output

The application runs for N=37 nodes in the system and varies the number of random indices k from 3 to 16, for which the random indices are generated and hence the unit cost matrix with k links of cost 1 unit.

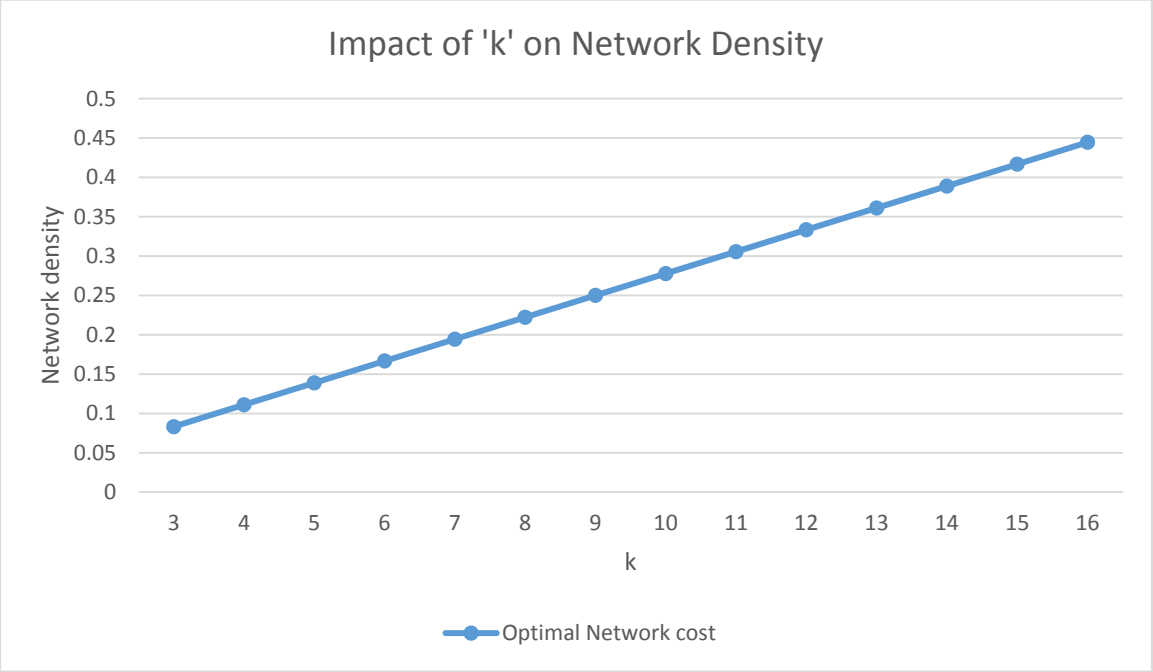
K	Network Cost	Network Density
3	27216	0.08333333333333333
4	7109	0.1111111111111111
5	5992	0.1388888888888889
6	5524	0.16666666666666666
7	5450	0.19444444444444445
8	4943	0.2222222222222222
9	4746	0.25
10	4838	0.2777777777777778
11	4514	0.3055555555555556
12	4438	0.3333333333333333
13	4326	0.3611111111111111
14	4258	0.3888888888888889
15	4407	0.4166666666666667
16	4093	0.4444444444444444

Impact of ‘k’ on total cost



From the above graph we can conclude that the optimum network cost is inversely proportional to k. As the value of k increases, k outgoing edges from the node have least cost path to other nodes. Hence, decreases the total network cost by taking the least cost path after the shortest path computation.

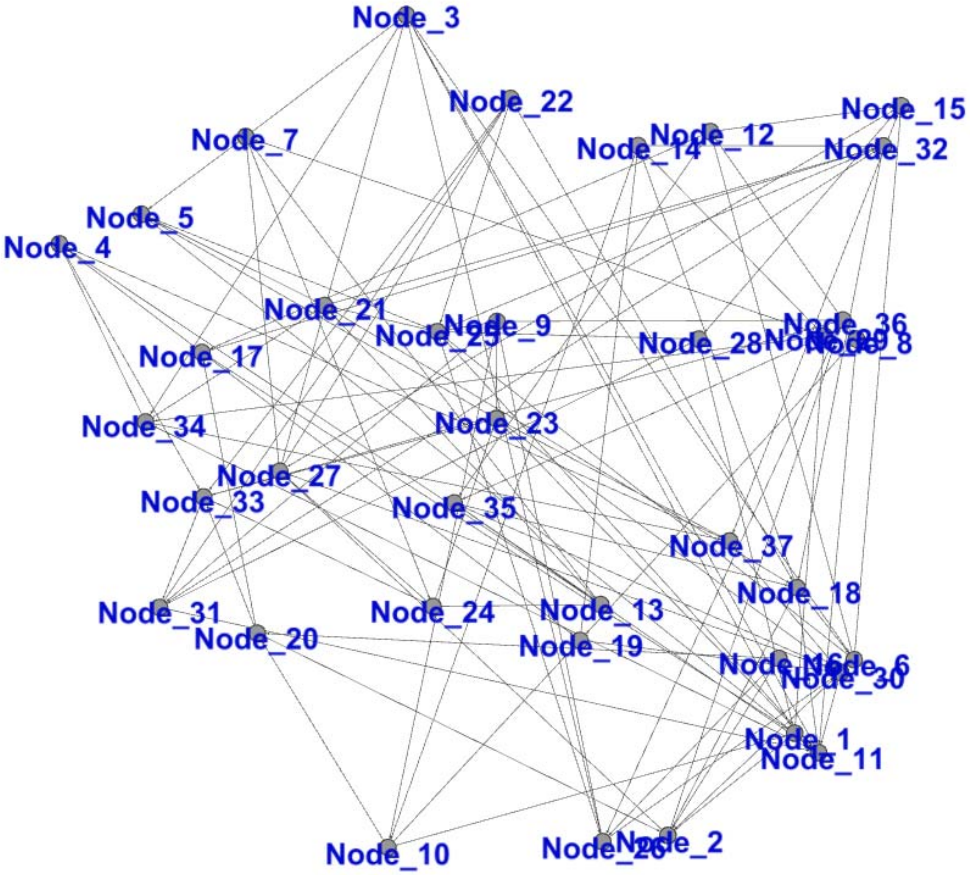
Impact of ‘k’ on network density



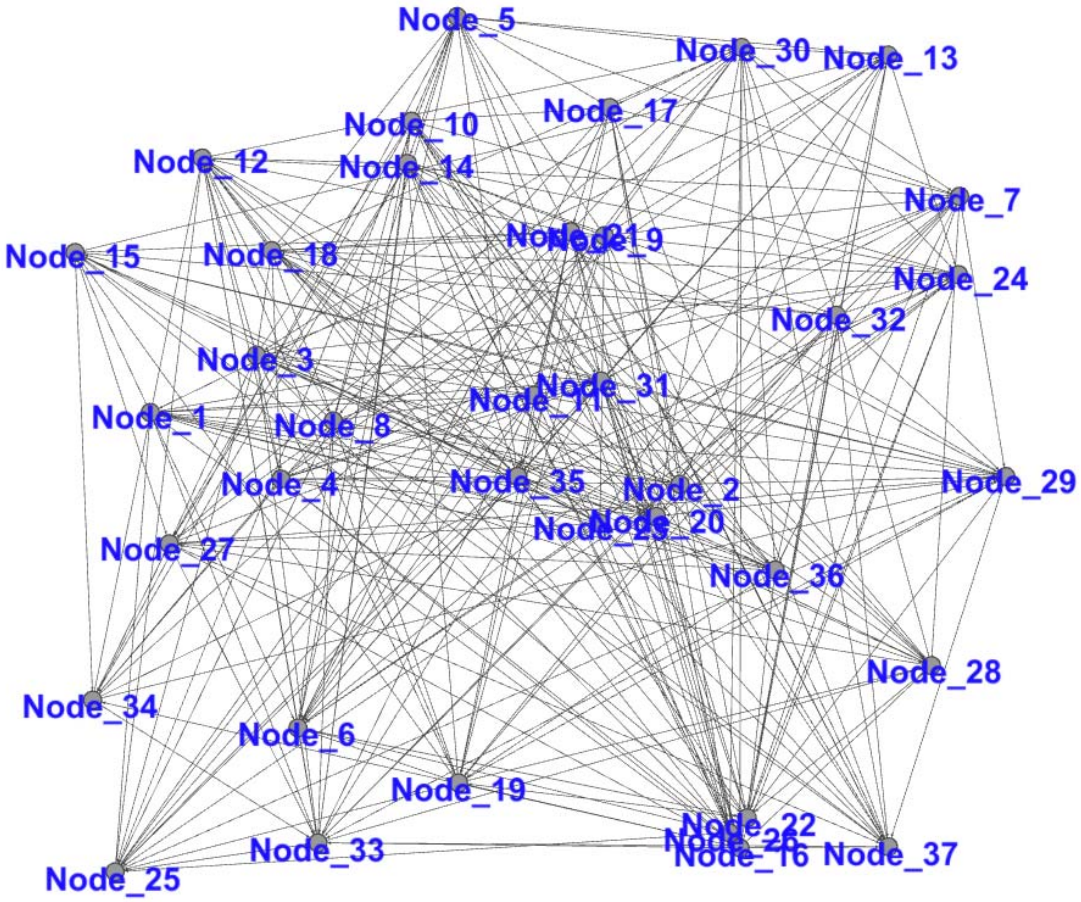
From the above graph we conclude that the network density is directly proportional to k. As the value of k increases, the network density becomes denser, as the value of k increases the number of least cost path also increases and from the shortest path computation the algorithm will have many nodes traversing through the least cost path thereby increasing the network density at each node.

Network Graph

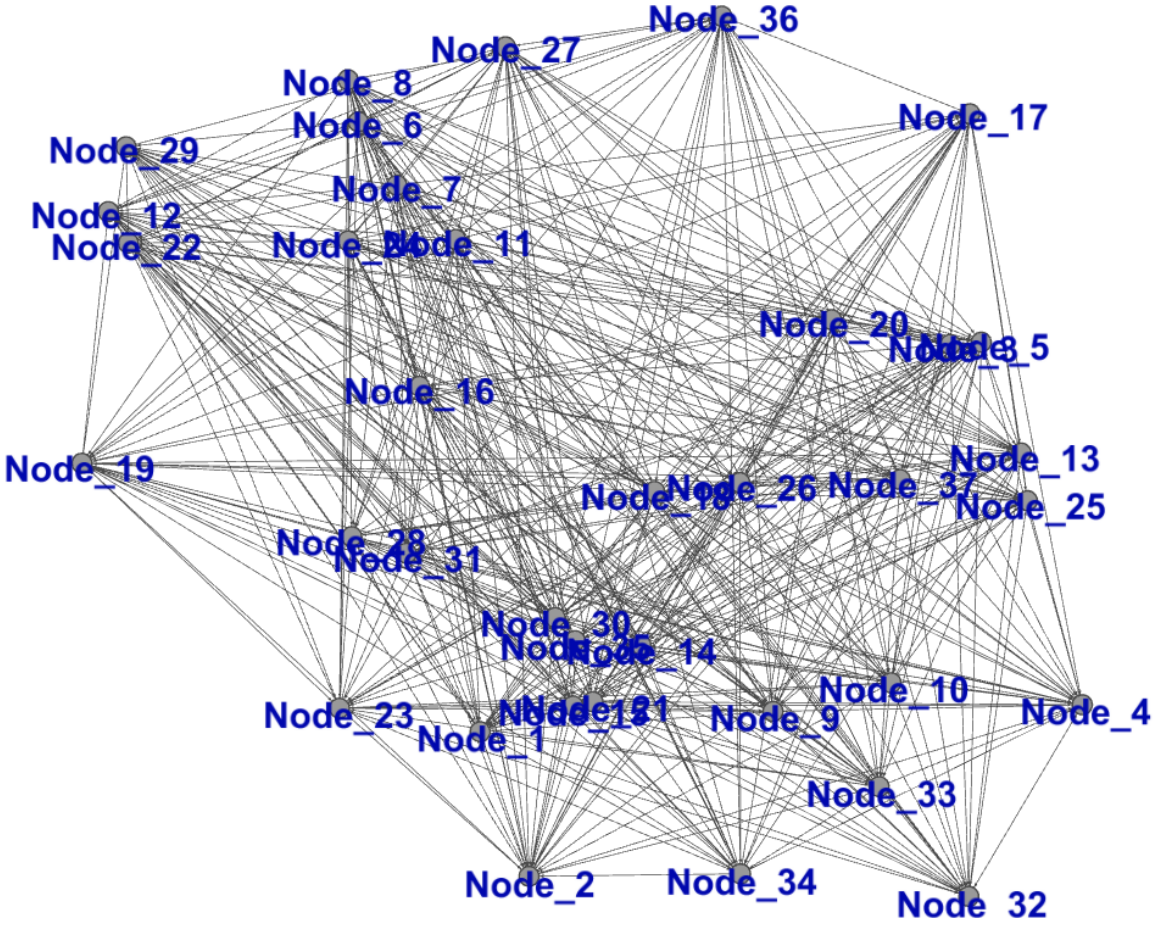
K=3



K=9


Node_5
Node_30
Node_13
Node_17
Node_10
Node_12
Node_14
Node_7
Node_15
Node_18
Node_24
Node_32
Node_3
Node_31
Node_1
Node_8
Node_4
Node_35
Node_2
Node_29
Node_20
Node_27
Node_36
Node_28
Node_34
Node_6
Node_19
Node_22
Node_33
Node_37
Node_25

K=16


Node_36
Node_27
Node_8
Node_6
Node_17
Node_29
Node_7
Node_12
Node_22
Node_11
Node_20
Node_16
Node_13
Node_19
Node_26
Node_37
Node_25
Node_31
Node_30
Node_14
Node_10
Node_4
Node_23
Node_9
Node_1
Node_33
Node_2
Node_34
Node_32

Conclusion

We conclude from the experiments that:

- As the value of the node connectivity parameter ‘k’ increases the density of the network increases.

The density does depends on the value of ‘k’ which gives the total number of outgoing edges from a node, as the parameter ‘k’ increases the density increases proportionally. As the value of ‘k’ increases, the number of least cost edges also increases thereby from the shortest path algorithm, the network becomes denser due to the increase in shortest path via the least cost edge. Hence the density of the network increases.

- The total cost of the network decreases with the increase in ‘k’ value

As long as there is no disconnected node in the total cost the network decreases with increase in ‘k’ value. As the value of ‘k’ increases, which is the number of outgoing edges for each node this increases the number of connected edges. Hence there is a greater chance of having shorter paths from a node which decreases the cost of the network.

Hence, the goal of designing an optimal network is achieved.

Source Code

Network Analysis module

```
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Random;


public class NetworkAnalysis {

	private final int N = 37;
	private int[][] trafficDemand;
	private int[][] unitCost;
	private ShortestPath shortPath;
	int[][] shortestPath;
	private int count;

	public NetworkAnalysis() {
		trafficDemand = new int[N][N];
		unitCost = new int[N][N];

		for(int i=0; i<N; i++) {
			for(int j=0; j<N; j++) {
				trafficDemand[i][j] = 0;
				unitCost[i][j] = 0;
			}
		}
	}

	private void generateTrafficDemand() {
		for(int i=0; i<N; i++) {
			for(int j=0; j<N; j++) {
				trafficDemand[i][j] = 0;
			}
		}

		Random rand = new Random();
		for(int i=0; i<N; i++) {
			for(int j=0; j<N; j++) {
				int n = rand.nextInt(5);
				if(i == j) continue;
				trafficDemand[i][j] = n;
			}
		}
	}

	private ArrayList<Integer> getRandomIndices(int currentI, int kIndices) {
		Random rand = new Random();
		int randomIndex;
		ArrayList<Integer> randomIndices = new ArrayList<>();
		for(int i=0; i<kIndices; i++) {
			do {
				randomIndex = rand.nextInt(N);
```

```
                } while(randomIndex == currentI ||
randomIndices.contains(randomIndex));
                randomIndices.add(randomIndex);
            }
            return randomIndices;
        }

        private void generateUnitCost(int k) {
            for(int i=0; i<N; i++) {
                for(int j=0; j<N; j++) {
                    unitCost[i][j] = 0;
                }
            }

            for(int i=0; i<N; i++) {
                ArrayList<Integer> randomIndices = getRandomIndices(i, k);

                for(int j=0; j<N; j++) {
                    unitCost[i][j] = 250;
                }
                for(int randomIndex : randomIndices) {
                    unitCost[i][randomIndex] = 1;
                }
                unitCost[i][i] = 0;
            }
        }

        private void computeCapacity() {
            count = 0;
            for(int i=0; i<N; i++) {
                for(int j=0; j<N; j++) {
                    if(unitCost[i][j] == 1) {
                        count++;
                    }
                }
            }
        }

        private void computeShortestPath() {
            shortPath = new ShortestPath(unitCost);
            shortPath.computeShortestPath();
        }

        private int computeCost() {
            shortestPath = shortPath.getShortestPath();
            int totalCost = 0;
            for(int i=0; i<N; i++) {
                for(int j=0; j<N; j++) {
                    totalCost += shortestPath[i][j] * trafficDemand[i][j];
                }
            }
            return totalCost;
        }

        private double computeDensity() {
```

```
		double directedEdge = 0;
		directedEdge = (double) count;
		double deno = (double) (N*(N-1));
		return (directedEdge / deno);
	}

	private void createGraph(int z) {
		FileWriter str = null;
		try {
			str = new
FileWriter("E:\\Programming\\eworkspace\\ATN\\NetworkAnalysis\\"+z+".csv");
			for(int k=0; k<N; k++) {
				str.append(";Node_"+(k+1));
			}
			str.append("\n");
			for(int i=0; i<N; i++) {
				str.append("Node_"+(i+1));
				for(int j=0; j<N; j++) {
					if(shortestPath[i][j] == 1) {
						str.append(";1");
					} else {
						str.append(";0");
					}
				}
				str.append("\n");
			}
		} catch (IOException ex) {
			ex.printStackTrace();
		} finally {
			try {
				str.flush();
				str.close();
			} catch (IOException e) {
				e.printStackTrace();
			}

		}


	}

	public static void main(String[] args) {
		int start = 3;
		int end = 16;
		NetworkAnalysis n = new NetworkAnalysis();
		for(int k=start; k<=end; k++) {
			n.generateTrafficDemand();
			n.generateUnitCost(k);
			n.computeShortestPath();
			n.computeCapacity();
			System.out.println("K = " + k);
			System.out.println("Network cost = " + n.computeCost());
			System.out.println("Network density = " + n.computeDensity());
			System.out.println("\n");
			n.createGraph(k);
```

```
			}
		}
}
```

Shortest Path module

```
public class ShortestPath {

	private int[][] unitCost;
	private int[][] shortestPath;
	private int noOfNodes;

	public int[][] getShortestPath() {
		return shortestPath;
	}

	public ShortestPath(int[][] unitCost) {
		this.unitCost = unitCost;
		noOfNodes = this.unitCost.length;
		shortestPath = new int[noOfNodes][noOfNodes];

		for(int i=0; i<noOfNodes; i++) {
			for(int j=0; j<noOfNodes; j++) {
				shortestPath[i][j] = unitCost[i][j];
			}
		}
	}


	private void setShortestPath(int[][] newShortestPath) {
		for(int i=0; i<noOfNodes; i++) {
			for(int j=0; j<noOfNodes; j++) {
				shortestPath[i][j] = newShortestPath[i][j];
			}
		}

	}

	//Floyd-warshall's all pair shortest path algorithm
	public int[][] computeShortestPath() {

		for(int k=0; k<noOfNodes; k++) {
	       int[][] newShortestPath = new int[noOfNodes][noOfNodes];
	       for (int i = 0; i < noOfNodes; i++) {
	           for (int j = 0; j < noOfNodes; j++) {
	               if(i == j)  continue;
	               if(shortestPath[i][j] > shortestPath[i][k] + shortestPath[k][j])
{
	                   newShortestPath[i][j] = shortestPath[i][k] +
shortestPath[k][j];
	               } else{
	                   newShortestPath[i][j] = shortestPath[i][j];
	               }
	           }
```

```
                }
                setShortestPath(newShortestPath);
            }
            return shortestPath;
        }
}
```

References

- http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
- http://gephi.github.io/