ALGORITHMIC ASPECTS OF TELECOMMUNICATION NETWORKS

PROJECT – 1

"AN APPLICATION TO NETWORK DESIGN"

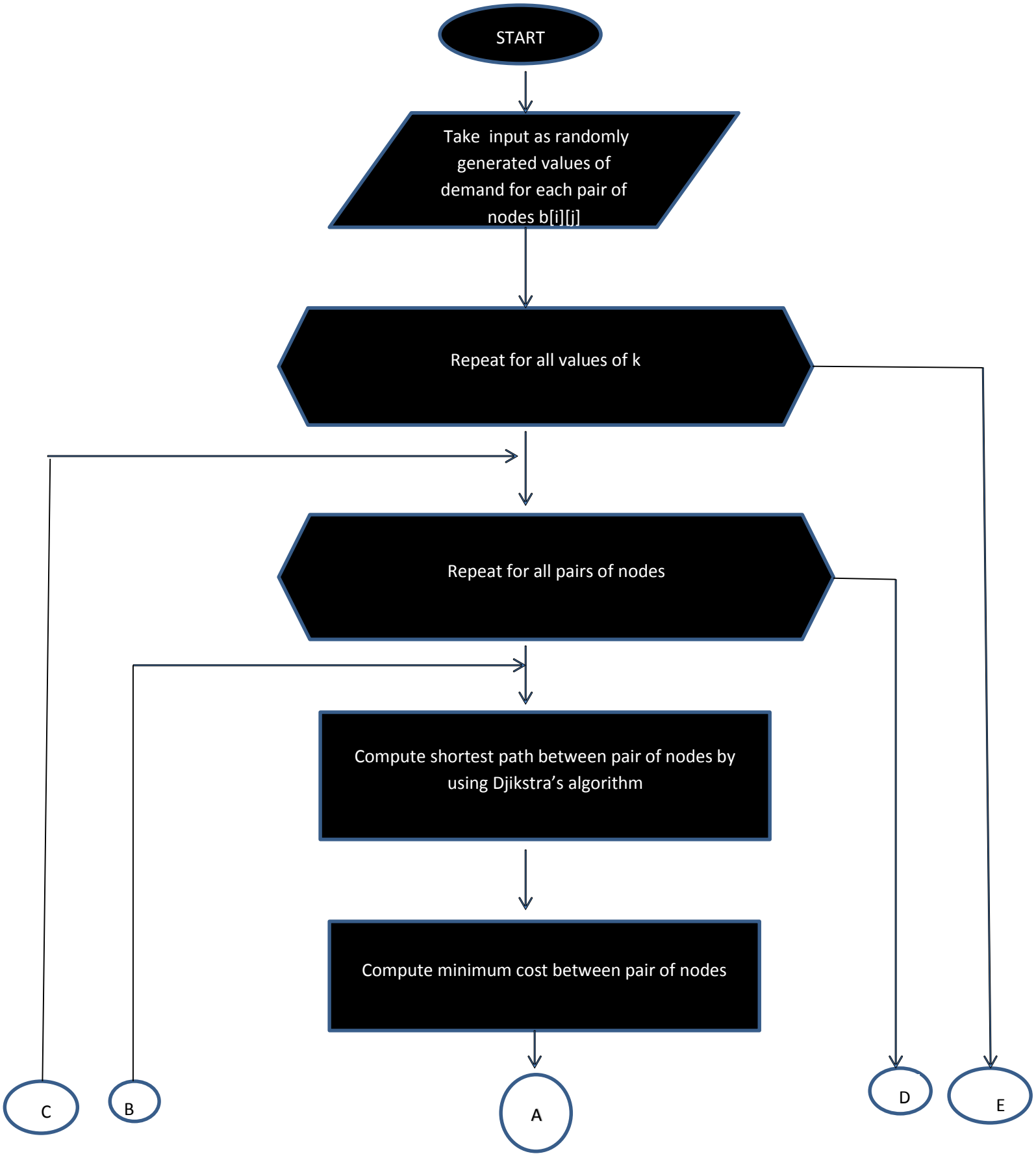
Submitted By –
Siddharth Sundewar
2021177557
Sxs137330@utdallas.edu

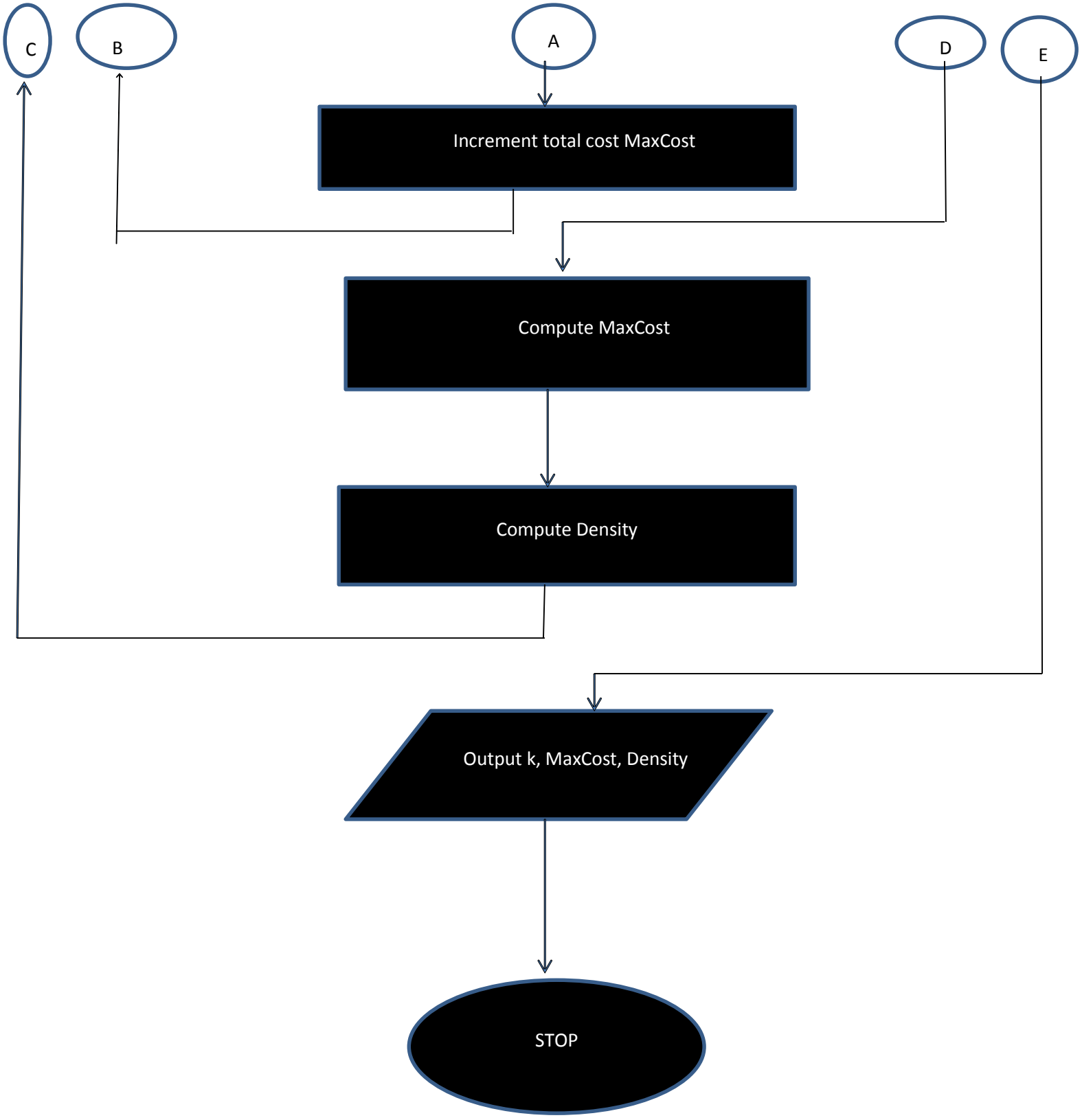
CONTENTS

ABSTRACT

- The essence of this project is to design various network topologies with varying maximum outgoing degrees for each topology (k) and analyze the effect of varying k on the density and optimum cost of the links of the topology.
- Takes as input for each pair of nodes in each topology – cost and demand of traffic between nodes as random generated samples, as indicated in the project description.
- Makes use of the Djikstra's algorithm that computes shortest paths between each pair of nodes for each value of k.
- For each value of k, MaxCost ie. sum total cost of all edges that form shortest paths in the topology is computed along with the density.
- Total cost and Density are analyzed and plotted for varying values of k

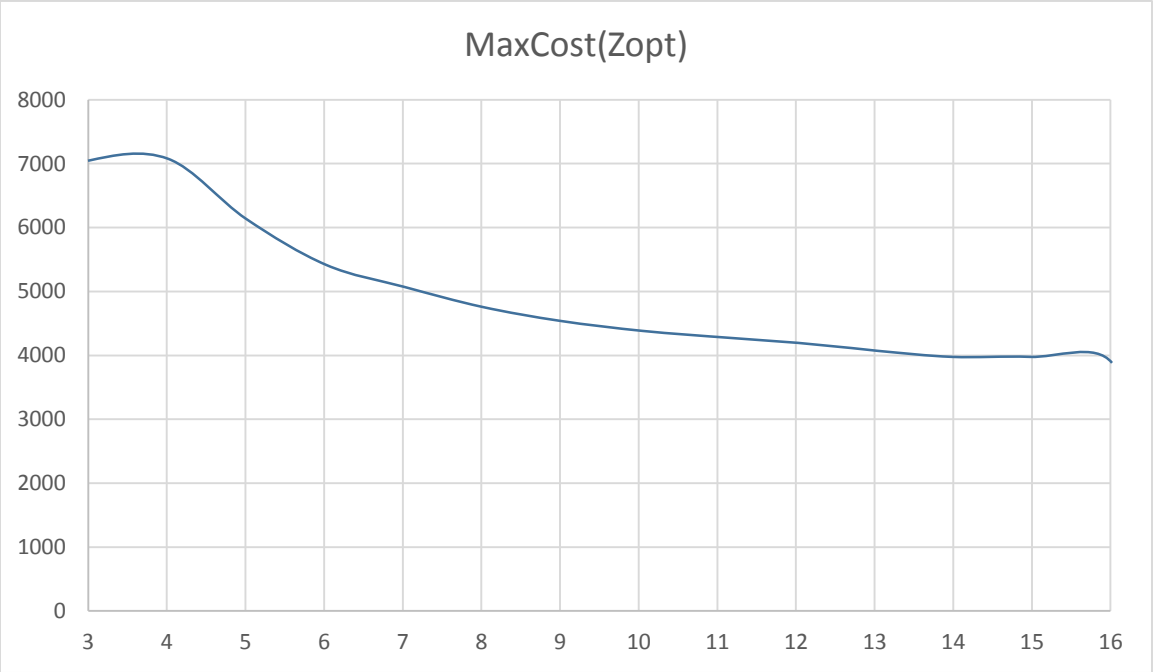
FLOWCHART


START
Take input as randomly generated values of demand for each pair of nodes b[i][j]
Repeat for all values of k
Repeat for all pairs of nodes
Compute shortest path between pair of nodes by using Djikstra's algorithm
Compute minimum cost between pair of nodes
A
B
C
D
E


C
B
A
D
E
Increment total cost MaxCost
Compute MaxCost
Compute Density
Output k, MaxCost, Density
STOP

GRAPHICAL OBSERVATIONS

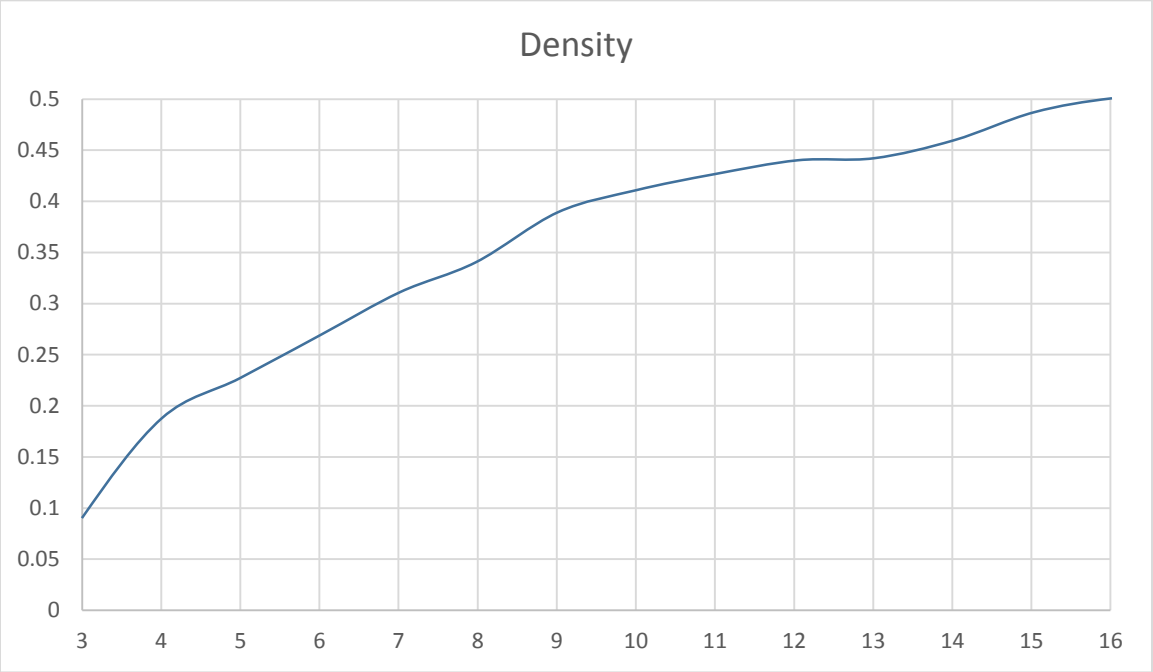
- Maximum Cost(MaxCost) versus Maximum Outgoing Degree :



K

- **Explanation** : As the maximum outgoing degree for each node is k increases, the optimum cost of the network attributed by all the links forming the shortest paths(MaxCost) should decrease (as exhibited by the graph) because with increasing k, more routes are discovered to other nodes and hence more shorter paths are found out. Hence their sum MaxCost should obviously decrease.

- Density versus Maximum Outgoing Degree(k) :



K

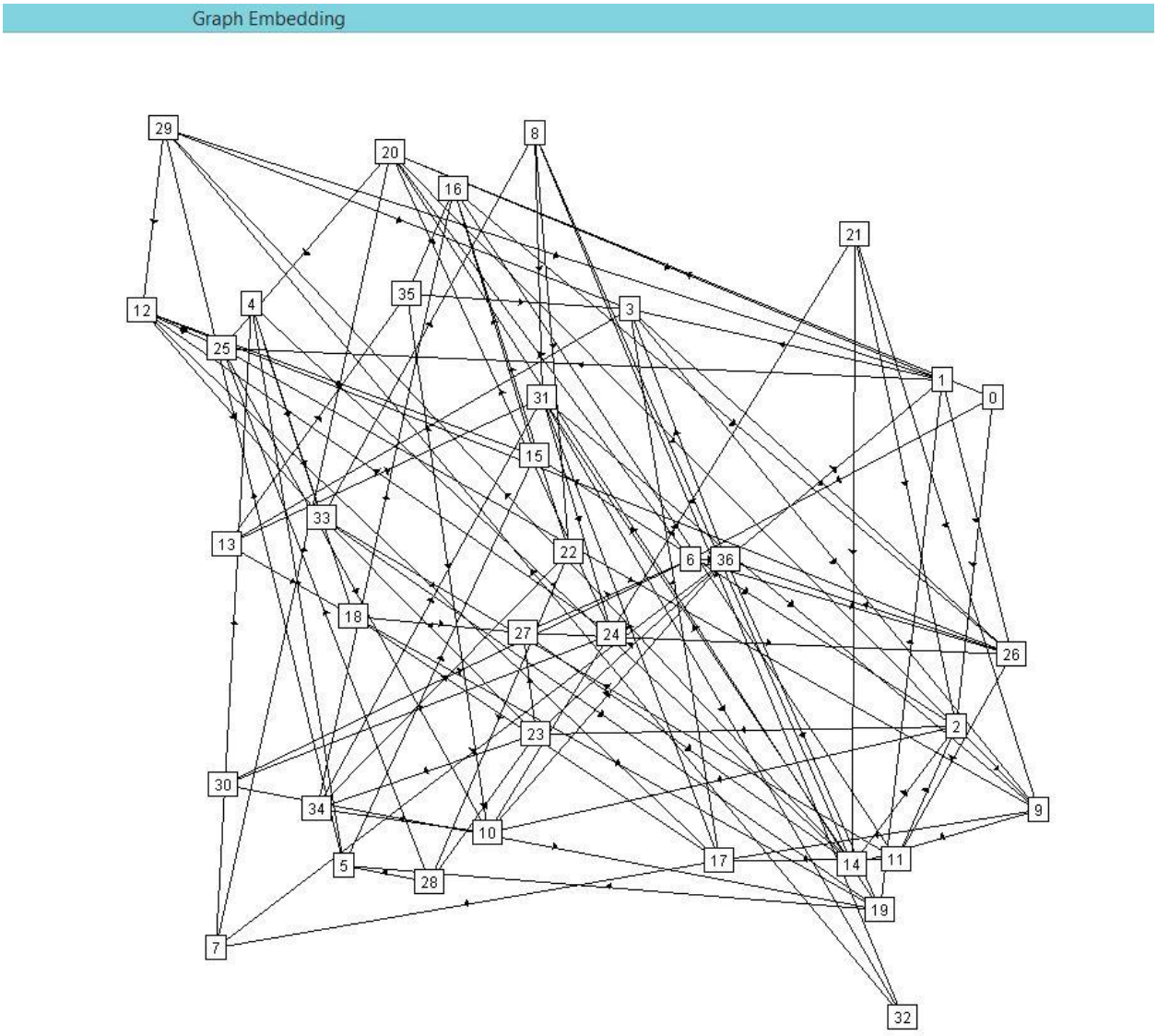
- **Explanation**: As the maximum outgoing degree for each node (k) increases, the density of the network ie. the number of links forming the shortest paths to the total number of links possible should also increase (as exhibited by the graph) because with increasing k, more routes are discovered to other nodes , thus cluttering the network more. Hence the density should obviously increase.

OUTPUTS OF THE ALGORITHM

K	MaxCost	Density
3	7051	0.08033033
4	7166	0.18393393
5	6132	0.22372372
6	5588	0.27477476
7	5126	0.31756756
8	4853	0.3490991
9	4571	0.3858859
10	4513	0.4174174
11	4374	0.4361862
12	4240	0.44894895
13	4144	0.4451952
14	4003	0.46696696
15	4024	0.481982
16	3939	0.5097598

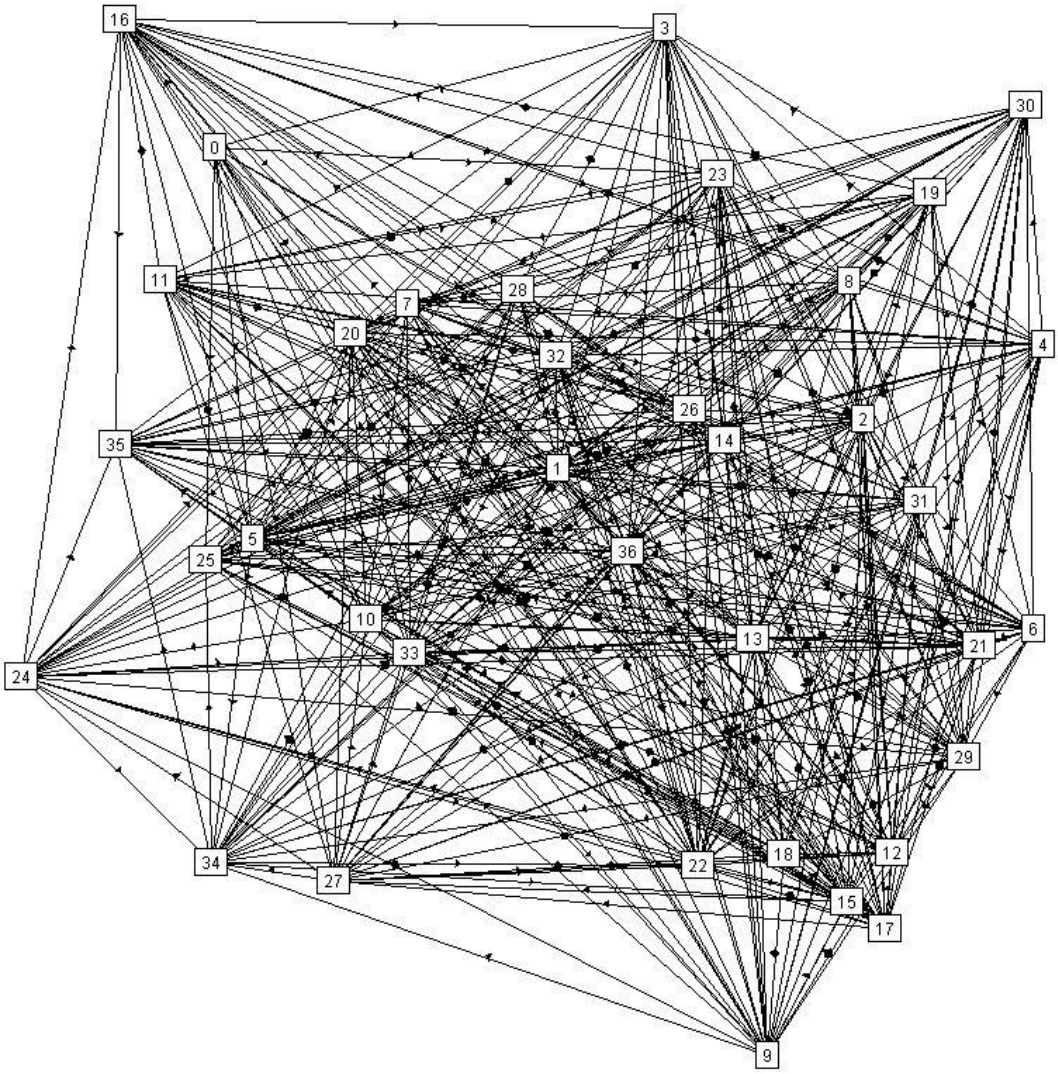
Network Topology Generated

K = 3



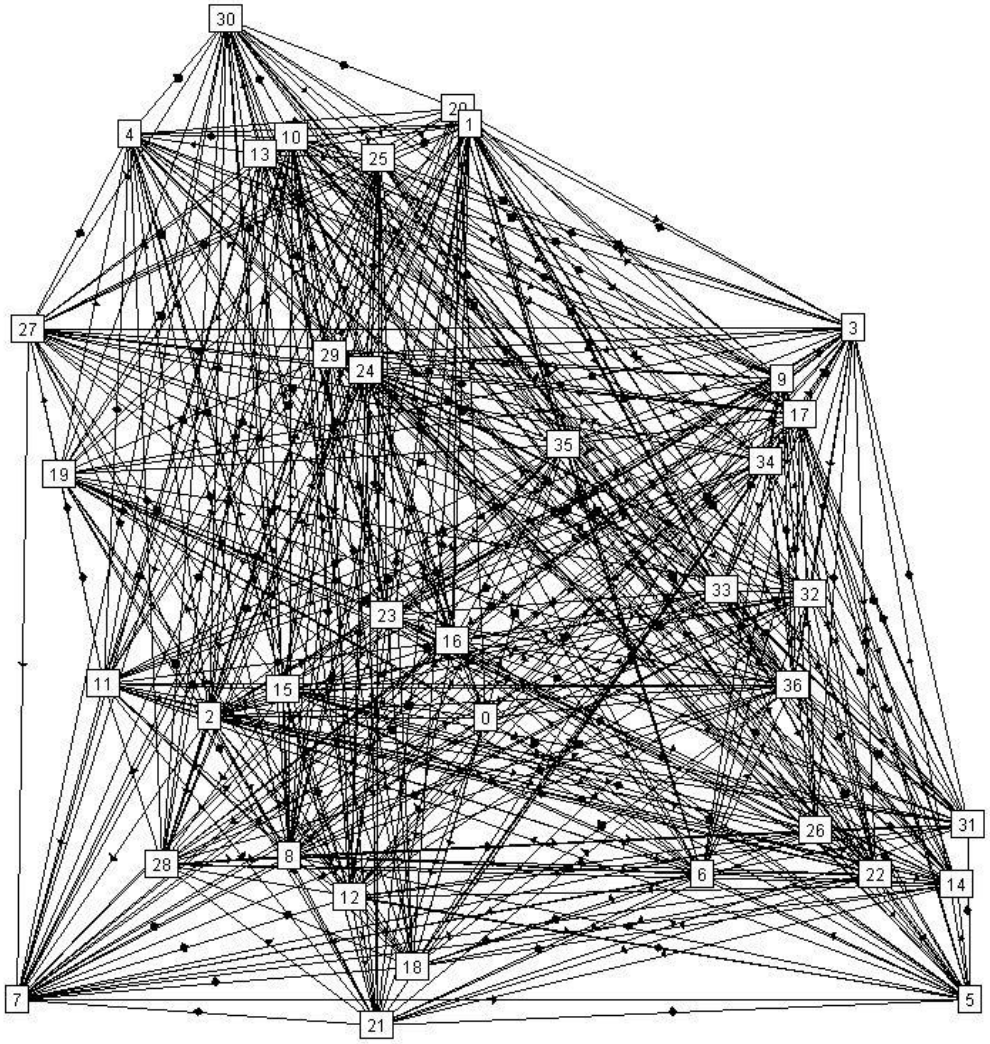
K = 10

Graph Embedding



K = 16

Graph Embedding



Dijkstra’s algorithm

Dijkstra's algorithm is a graph search algorithm that solves the single-source shortest paths for a graph with nonnegative path costs, producing a shortest path tree. This algorithm is often used in routing and as a subroutine in other graph algorithms.

I have made use of Dijkstra’s algorithm to compute the shortest paths between each pair of nodes under consideration.

Pseudo Code :

- Initialize the cost of each node to ∞
- Initialize the cost of the source to 0
- While there are unknown nodes left in the graph

Select the unknown node b with the lowest cost
Mark b as known
For each node a adjacent to b

✓ a’s cost = min(a’s old cost, b’s cost +
cost of (b, a))

FAST SOLUTION METHOD

- 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. Here I have employed Dijkstra's algorithm.

- 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:

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

SOURCE CODE

NetworkGenerator:

```
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
public class NetworkGenerator {
/**
* @param args
*/
public static void main(String[] args) {
	int NODE=37; //number of nodes given
	int b[][]=new int[NODE][NODE]; //traffic demand between pairs of nodes
	int a[][]=new int[NODE][NODE]; //unit cost of traffic between pairs of
nodes
	//possible values to choose from for demand
	int GivenArray[]={0,1,2,3,4}; //array to store possible values for bij
	Random generator = new Random(); //creating object of Random library
class
	try
	{
		for(int i=0;i<NODE;i++)
		{
			for(int j=0;j<NODE;j++)
			{
				b[i][j] = generator.nextInt(GivenArray.length);
				//generating a random value from GivenArray
			}
		}
	}
	catch(ArrayIndexOutOfBoundsException e)
	{
		System.out.println("Array out of bounds");
	}//iterating through each and every value of k
	for(int k=3; k<17; k++)
	{
		int MaxCost=0;
		float density;
		int count=0;
		//Creating an object of ShortestPath class
		DjikstraShortestPath sp=new DjikstraShortestPath();
		//iterating through every value from 0 to n-1
		for(int i=0;i<NODE;i++)
		{
			//creating an array to store 0 to n-1
			int indices[]=new int[NODE];
			for(int l=0;l<NODE;l++)
			{
				indices[l]=l;
			}
			//creating an array list called randomList
			List<Integer> randomList = new ArrayList<Integer>();
			//creating an object of RandomGeneration
			RandomGenerator r=new RandomGenerator();
			//retrieve k random indices from RandomGeneration class
			randomList=r.createRandomIndices(indices,k,i,NODE);
			//System.out.println("Back to Main");
			//iterating through 0 to NODE-1
```

```
                for(int j=0;j<NODE;j++)
                {
                        // System.out.println("In printing method\n");
                        //setting diagonal elements to zero
                        if(i==j)
                        {
                                a[i][j]=0;
                        }
                        //set cost to 1 for indices that are picked in
randomlist
                        if(randomList.contains(j)&& i!=j)
                        {
                                a[i][j]=1;
                        }
                        else
                        //setting a[i][j] to 250 if j is not contained in
randomlist
                        {
                                if(i!=j){
                                        a[i][j]=250;
                                }
                        }
                        //System.out.print(a[i][j]);
                        //System.out.print("\t\t\t");
                        // System.out.println("Done printing");
                        //Invoking findpath method for every pair i,j
                        sp.findpath(i,j,a);
                        int hopCount=sp.getNumberOfHops();
                        int path[]=sp.getPath();
                        //retrieve distance to get from i to j
                        int distance=sp.getDistance();

                        //computing demand times unit cost for each link
                        MaxCost+=b[i][j]*distance;
                }
        }
        //counter variable to store number of edges making up shortest
path
        count=0;
        //retrieving path matrix
        int c[][]=sp.getCount();
        //incrementing that link in matrix if it exists
        for(int x=0;x<c.length;x++)
        {
                for(int y=0;y<c.length;y++)
                {
                        if(c[x][y]>0)
                                count++;
                }
        }
        for (int i = 0; i < c.length; i++)
        {
                System.out.print(" ");
                for (int j = 0; j < c.length; j++)
                {
                        if(c[i][j]!=0)
                        {
                                System.out.print("1");
                                if(j!=c.length)
                                        System.out.print(" ");

                        }
                        else
                        {
                                System.out.print("0");
                                if(j!=c.length)
                            System.out.print(" ");
                        }
```

```
                }
                System.out.println(" ");
                if(i!=c.length)
                System.out.print(" ");
            }
            System.out.println("Count is "+ count+" NODE ="+NODE);
            density=(float)(count)/(NODE*(NODE-1));
            System.out.println("k = "+k+"\n"+"MaxCost(Zopt) =
"+MaxCost+"\n"+"Density = "+density);
            System.out.println("\n");
      }
      }
}
```

DjikstraShortestPath:

```
public class DjikstraShortestPath {

      final int Temp=0;
      final int PERM= 1;
      final int NODE= 37;
      final int Value=250;
      int NumberOfHops;
      int path[];
      int distance;
      int Count[][]= new int[37][37];
      public int[][] getCount() {
            return Count;
      }
      public void setC(int[][] NewCount) {
            this.Count = NewCount;
      }
      public int getDistance() {
            return distance;
      }
      public void setDistance(int distance) {
            this.distance = distance;
      }
      public int getNumberOfHops() {
            return NumberOfHops;
      }
      public void setNumberOfHops(int numberOfHops) {
            this.NumberOfHops = numberOfHops;
      }
      public int[] getPath() {
            return path;
      }
      public void setPath(int[] path) {
            this.path = path;
      }
      public DjikstraShortestPath()
      {
            for(int x=0; x<Count.length; x++)
            {
                  for(int y=0; y<Count.length; y++)
                  {Count[x][y]=0;
                  }
            }
      }
      void findpath(int s,int d,int a[][])
      {
            int i,min,count=0;
            int current,newdist,u,v,n=NODE;
            int setdist=0;
```

```
		int path[]=new int[n];

		NodeGenerator[] node=new NodeGenerator[n];
		/* Make all nodes temporary */
		for(i=0; i<n; i++)
		{
			node[i] = new NodeGenerator();
			node[i].setPredecessor(-1);
			node[i].setDistance(Value);
			node[i].setStatus(Temp);
		}
		/*Source node should be permanent*/
		node[s].setPredecessor(-1);
		node[s].setDistance(0);
		node[s].setStatus(PERM);
		/*Starting from source node until destination is found*/
		current=s;
		while(current!=d)
		{
			for(i=0; i<n; i++)
			{
				/*Checks for adjacent temporary nodes */
				if ( a[current][i] > 0 && node[i].getStatus() == Temp
)
				{
					newdist=node[current].getDistance() +
a[current][i];
					/*Checks for Relabeling*/
					if( newdist < node[i].getDistance() )
					{
						node[i].setPredecessor(current);
						node[i].setDistance(newdist);
					}
				}
			}/*End of for*/
			/*Search for temporary node with minimum distance make it
current
			node*/
			min=Value;
			current=-1;
			for(i=1; i<n; i++)
			{
				if(node[i].getStatus() == Temp &&
node[i].getDistance() < min)
				{
					min = node[i].getDistance();
					current=i;
				}
			}/*End of for*/
			if(current==-1) /*If Source or Sink node is isolated*/
				return ;
			node[current].setStatus(PERM);
		}/*End of while*/
		/* Getting full path in array from destination to source */
		while( current!=-1 )
		{
			//	count++;
			path[count++]=current;
			current=node[current].getPredecessor();
		}
		/*Getting distance from source to destination*/
		for( i=count-1; i>0; i--)
		{
			u=path[i];
			v=path[i-1];
			setdist= setdist + a[u][v];
			Count[u][v]=Count[u][v]+1;
		}
```

```
            this.setNumberOfHops(count);
            this.setPath(path);
            this.setDistance(setdist);
            this.setC(Count);

      }/*End of findpath()*/
}
```

RandomGenerator:

```
import java.util.ArrayList;

import java.util.Collections;

import java.util.List;

public class RandomGenerator {

//

      public List<Integer> createRandomIndices(int[] indices,int k,int i,int n)

      {

            //creating a list to store 0 to n-1

            List<Integer> list = new ArrayList<Integer>();

            Integer arr[] = new Integer[37];

            //creating list to store random k values

            List<Integer> randomList = new ArrayList<Integer>();

            for(int j=0;j<n;j++)

            {

                  // System.out.println("In for loop");

                  arr[j] = new Integer(indices[j]);

                  list.add(arr[j]);

            }

            //System.out.println("Done adding to list\n");

            //removing the possibility of selecting j=i

            list.remove(i);

            //shuffling the list to facilitate randomization

            Collections.shuffle(list);

            for(int j=0; j<k; j++)

            {

                  {

                        //Retrieving the first k elements one by one
```

```
                        int z= (int) list.get(j);

                        //Adding reteieved element to randomList

                        randomList.add(z);

                  }

            }

            //return randomlist to main method

            return randomList;

      }

}
```

Node Generator :

```
public class NodeGenerator {
      //integer to identify predecessor node
      int Pred;
      int dist; /*minimum distance of node from source*/
      //stores status - permanent or temporary
      int status;
      public NodeGenerator() {
            super();
      }
      public int getDistance() {
            return dist;
      }
      public void setDistance(int dist) {
            this.dist = dist;
      }
      public int getPredecessor() {
            return Pred;
      }
      public void setPredecessor(int pred) {
            this.Pred = pred;
      }
      public int getStatus() {
            return status;
      }
      public void setStatus(int status) {
            this.status = status;
      }
}
```

Explanation of Modules

- NetworkGenerator: Contains the main method, which calls the RandomGenerator and DjikstraShortestPath computation functions.

- RandomGenerator: Contains the createRandomIndices method which generates random k values from the i values passed without repetition.

- DjikstraShortestPath: Contains the method findPath which computes the shortest path between each pair of nodes using Djikstra's algorithm. It Returns the shortest path and number of hops between the nodes.

- NodeGenerator: Has member variables- predecessor, status and distance. Also contains getters and setters for the same for usage within the DjikstraShortestPath class. These three properties apply to each node of the topology.

REFERENCES

- Dijkstra's algorithm was taken off the shelf source available on algolist.com. It was a JAVA language implementation.
- The description of Dijkstra's algorithm was taken as a basis from Wikipedia.org and then translated into my own words.
- The network topology is generated from the website http://www.cs.rpi.edu/research/groups/pb/graphdraw/headpage.html.
- The formulae for computation of the Zopt(MaxCost) and Density values was borrowed from material posted by Dr Andras Farago in his lecture notes.