Algorithmic Aspects of Telecommunication Networks
CS 6385.001: Project #2

Due on Tuesday November 5, 2019 at 11:59am

Instructor: Prof. Andras Farago

Shyam Patharla (sxp178231)

Contents

1 Introduction

- In this project, we try to implement the basic network design model
- Give an input number of nodes, traffic demands between different nodes and a parameter k, our program outputs a network topology i.e a directed graph with links and capacities assigned to these links
- We do this for range of values of k
- We also analyze how the cost and density of the output network vary with the value of k
- We discuss the possible reasons for the above characteristics.

2 Design Decisions

- We implement the solution in the **Java** programming language
- The program modules were run on a **Mac** operating system

3 Solution Approach

3.1 Generating Input Examples

Module 1 consists of generating input graphs for simulating the algorithm. These parameters are then passed on to second module which runs the Nagamochi Ibaraki algorithm on them. The graphs are generated as follows.

- For all examples, we set the number of nodes in the network to 20 i.e $n = 20$.
- The number of edges is taken from the set [19,190] in steps of 3 i.e. m=19,22,25,....190
- For each pair of values of m and n we generate 5 graphs
- The m edges are selected randomly
- Self loops and parallel edges are avoided.

3.2 Nagamochi Ibaraki Algorithm

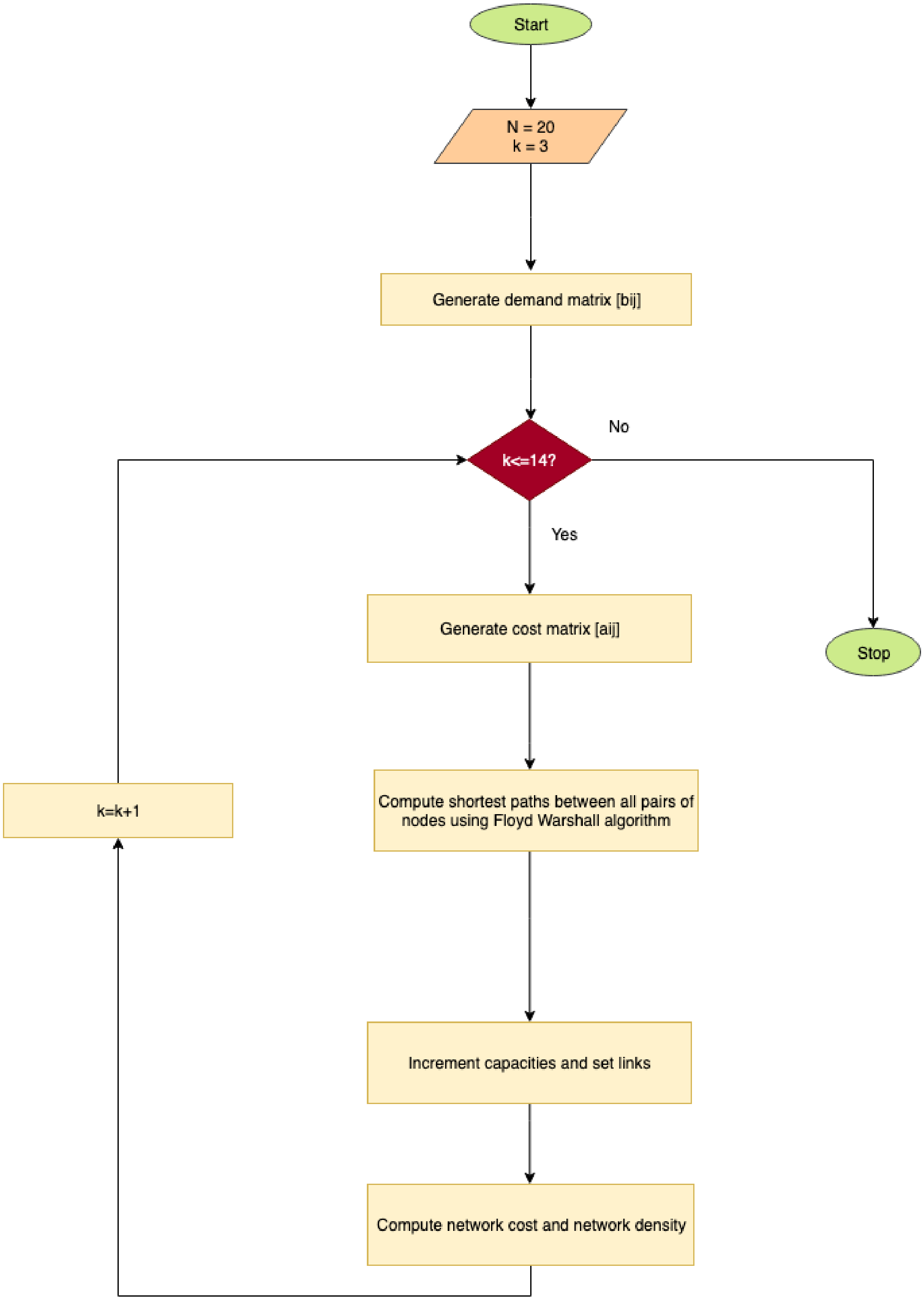
Module 2 receives input parameters from the first module (graph).

- All pairs shortest paths
  - We compute the shortest distances between all pairs of nodes using the **Floyd-Warshall algorithm** i.e. and store them in a matrix *dist*
  - While computing the above *dist* matrix, we also store the shortest paths between each pair of nodes (i,j) in the pred matrix. Each entry $\mathbf{pred_{ij}}$ gives the node with **maximum** number in the shortest path between nodes i and j.
  - **Special case:** If there is no node in the shortest path between i and j, $\mathrm{pred_{ij}}$=-1
- Design Links and Capacities
  - The *capacities* matrix and *links* matrix are initiated with all values equal to **zero**
  - Let S be the shortest path between nodes k and l, where k≠l. For each edge (i,j) on S
    - ∗ If $\mathbf{link_{ij}}$ has not been set to 1, set it to 1
    - ∗ The capacity of $\mathrm{link_{ij}}$ is incremented by $\mathbf{demand_{kl}}$
  - The above process is repeated for each pair of nodes (k,l) where k≠l.
- The *links* and *capacities* matrices are passed on to Module 3, which analyzes how the total cost of the network and densities change with respect to our k parameter.

3.3 Presentation of Results

- Module 3 takes the output parameters of *Module 2* i.e. the capacities and link matrices and computes
  - ∗ the **total cost** of the network
  - ∗ the **density** of the network i.e. the number of links with **non-zero** capacities divided by the maximum possible number of links which is N*(N-1)
- The above computation is repeated for all k values
- We then try to answer questions such as:
  - ∗ How does the total cost of the network depend on k?
  - ∗ How does the density of the obtained network depend on k?
- We show some of the obtained network topologies graphically

We present the execution of our project using the flow chart below.


Start
N = 20
k = 3
Generate demand matrix [bij]
k<=14?
No
Yes
Generate cost matrix [aij]
Stop
k=k+1
Compute shortest paths between all pairs of nodes using Floyd Warshall algorithm
Increment capacities and set links
Compute network cost and network density

4 Nagamochi Ibaraki Algorithm - Explanation

- Suppose we have N vertices and the vertices are numbered from 1,2,...,N
- For every pair of vertices i and j we define a path $\pi(i, j, r)$ which is the shortest path from i to j that passes through vertices numbered atmost r.
- This problem has a simple recursive structure
- **Base Case**: The path $\pi(i, j, 0)$ cannot pass through any vertices, so it is the edge(if any) between i an j
- **Recursive Case**: The path $\pi(i, j, r)$ will have two cases
  - The path does not pass through the vertex labeled r, so it must pass through the vertices numbered atmost r-1 $\implies \pi(i, j, r) = \pi(i, j, r-1)$
  - The path passes through the vertex labeled r. It will consist of the path from i to r and the path from r to j, each of which passes through vertices numbered atmost r-1
- We first try our hands at a straightforward dynamic programming approach, which is shown below
- We can simplify the algorithm by removing the third dimension. We can also modify the algorithm to support numbering the vertices arbitrarily

Algorithm 1 NagamochiIbarakiAlgorithm

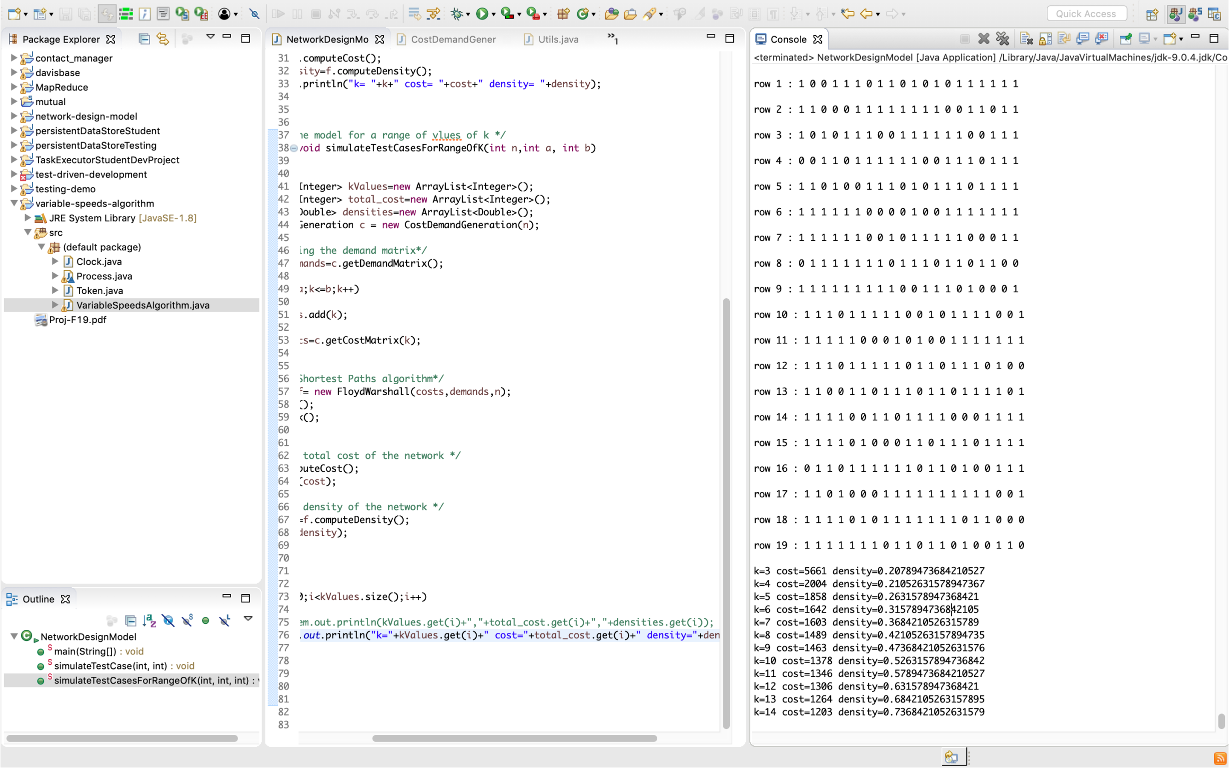
```
procedure NagamochIbaraki($V, E$)
    n=nodes.length
    if $n = 2$ then
        return degree (y)
    end if
    v[] = maximumAdjacencyOrdering(V)
    $x = v_{n-1}$
    $y = v_n$
    a=$degree(y)$
    $G_{xy}$=graph created by merging nodes x and y
    b = NagamochiIbaraki($V_{xy}, E_{xy}$)
    return min(a,b)
end procedure
```

- The **dist** matrix gives the shortest path distances between all pairs of nodes.

- The **pred** matrix gives the node with maximum number in the shortest path between any two nodes.

5 Observations and Analysis

- The program produces an output as shown in the figure below.

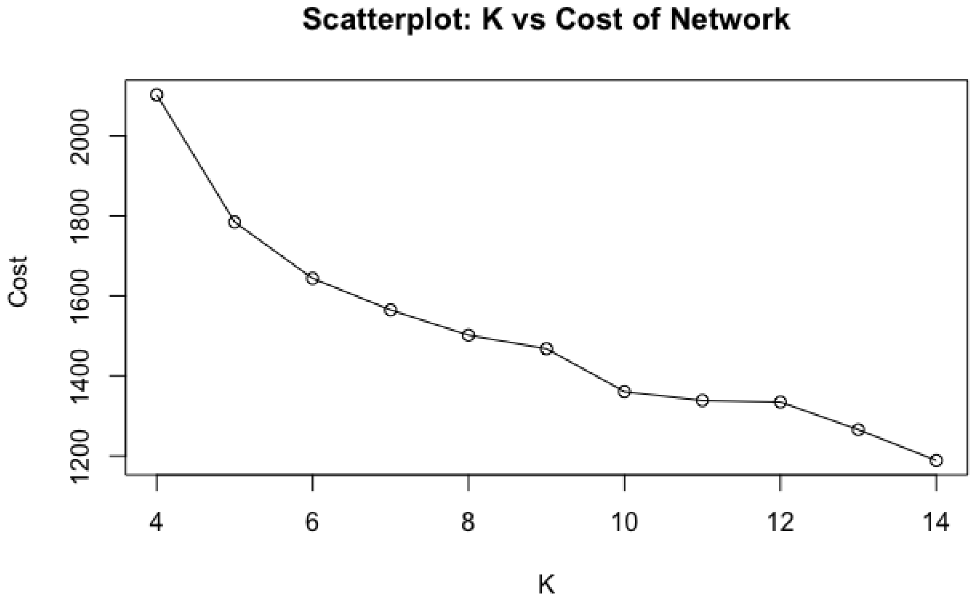


```
.computeCost();
sity=f.computeDensity();
.println("k= "+k+" cost= "+cost+" density= "+density);



ie model for a range of vlues of k */
void simulateTestCasesForRangeOfK(int n,int a, int b)


Integer> kValues=new ArrayList<Integer>();
Integer> total_cost=new ArrayList<Integer>();
Double> densities=new ArrayList<Double>();
Generation c = new CostDemandGeneration(n);

ing the demand matrix*/
mands=c.getDemandMatrix();

a;k<=b;k++)

s.add(k);

ts=c.getCostMatrix(k);


Shortest Paths algorithm*/
F= new FloydWarshall(costs,demands,n);
();
x();


 total cost of the network */
puteCost();
(cost);

 density of the network */
=f.computeDensity();
density);




0;i<kValues.size();i++)

em.out.println(kValues.get(i)+","+total_cost.get(i)+","+densities.get(i));
.out.println("k="+kValues.get(i)+" cost="+total_cost.get(i)+" density="+den
```

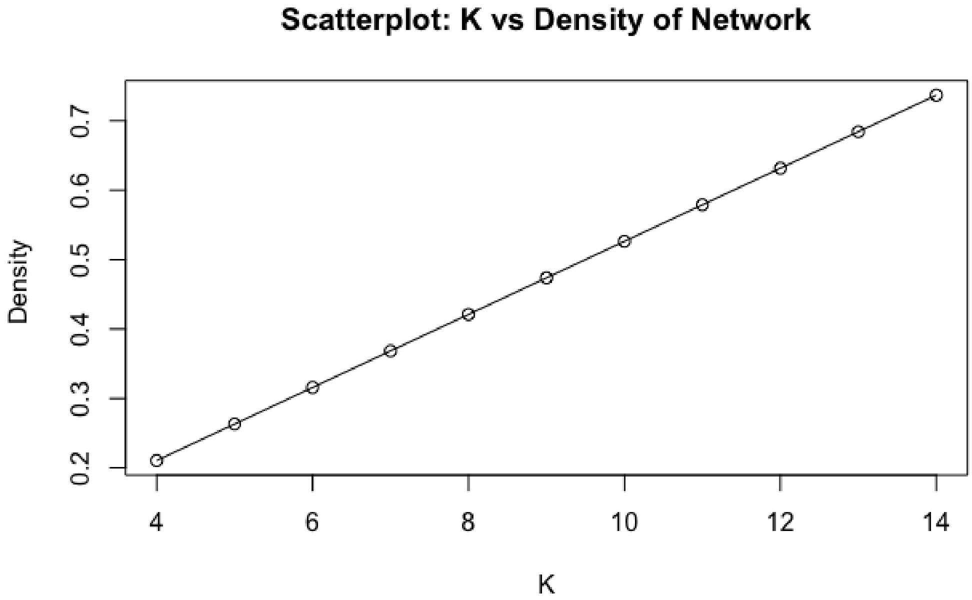
```
<terminated> NetworkDesignModel [Java Application] /Library/Java/JavaVirtualMachines/jdk-9.0.4.jdk/Co

row 1 : 1 0 0 1 1 1 0 1 1 0 1 0 1 0 1 1 1 1 1 1

row 2 : 1 1 0 0 0 1 1 1 1 1 1 1 1 0 0 1 1 0 1 1

row 3 : 1 0 1 0 1 1 1 0 0 1 1 1 1 1 1 0 0 1 1 1

row 4 : 0 0 1 1 0 1 1 1 1 1 1 0 0 1 1 1 0 1 1 1

row 5 : 1 1 0 1 0 0 1 1 1 0 1 0 1 1 1 0 1 1 1 1

row 6 : 1 1 1 1 1 1 0 0 0 0 1 0 0 1 1 1 1 1 1 1

row 7 : 1 1 1 1 1 1 0 0 1 0 1 1 1 1 1 0 0 0 1 1

row 8 : 0 1 1 1 1 1 1 1 0 1 1 1 0 1 1 0 1 1 0 0

row 9 : 1 1 1 1 1 1 1 1 1 0 0 1 1 1 0 1 0 0 0 1

row 10 : 1 1 1 0 1 1 1 1 1 0 0 1 0 1 1 1 1 0 0 1

row 11 : 1 1 1 1 1 0 0 0 1 0 1 0 0 1 1 1 1 1 1 1

row 12 : 1 1 1 0 1 1 1 1 1 0 1 1 0 1 1 1 0 1 0 0

row 13 : 1 1 0 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1 0 1

row 14 : 1 1 1 1 0 0 1 1 0 1 1 1 1 0 0 0 1 1 1 1

row 15 : 1 1 1 1 0 1 0 0 0 1 1 0 1 1 1 0 1 1 1 1

row 16 : 0 1 1 0 1 1 1 1 1 1 0 1 1 0 1 0 0 1 1 1

row 17 : 1 1 0 1 0 0 0 1 1 1 1 1 1 1 1 1 1 0 0 1

row 18 : 1 1 1 1 0 1 0 1 1 1 1 1 1 1 0 1 1 0 0 0

row 19 : 1 1 1 1 1 1 1 0 1 1 0 1 1 0 1 0 0 1 1 0

k=3 cost=5661 density=0.20789473684210527
k=4 cost=2004 density=0.21052631578947367
k=5 cost=1858 density=0.2631578947368421
k=6 cost=1642 density=0.3157894736842105
k=7 cost=1603 density=0.3684210526315789
k=8 cost=1489 density=0.42105263157894735
k=9 cost=1463 density=0.47368421052631576
k=10 cost=1378 density=0.5263157894736842
k=11 cost=1346 density=0.5789473684210527
k=12 cost=1306 density=0.631578947368421
k=13 cost=1264 density=0.6842105263157895
k=14 cost=1203 density=0.7368421052631579
```

- The output results are stored in a *csv* file. The graphs are generated in **R**.

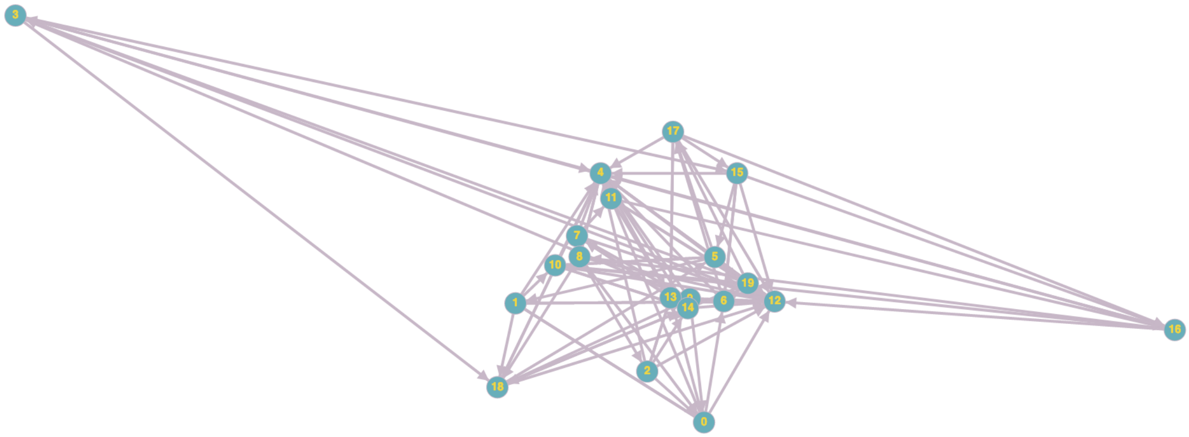
- We plot the graph of the parameter **k** vs. **cost of the network**. We can clearly see that the cost of the network **decreases** with increase in the value of k.



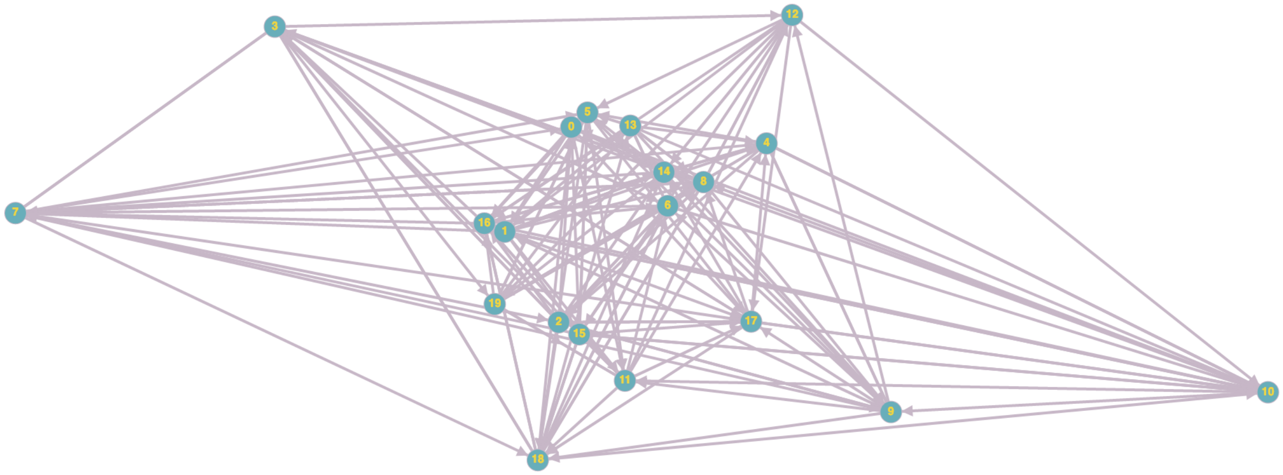
- We plot the graph of the parameter **k** against the **density of the network**. We can clearly see that the density of the network **increases** with increase in the value of k.



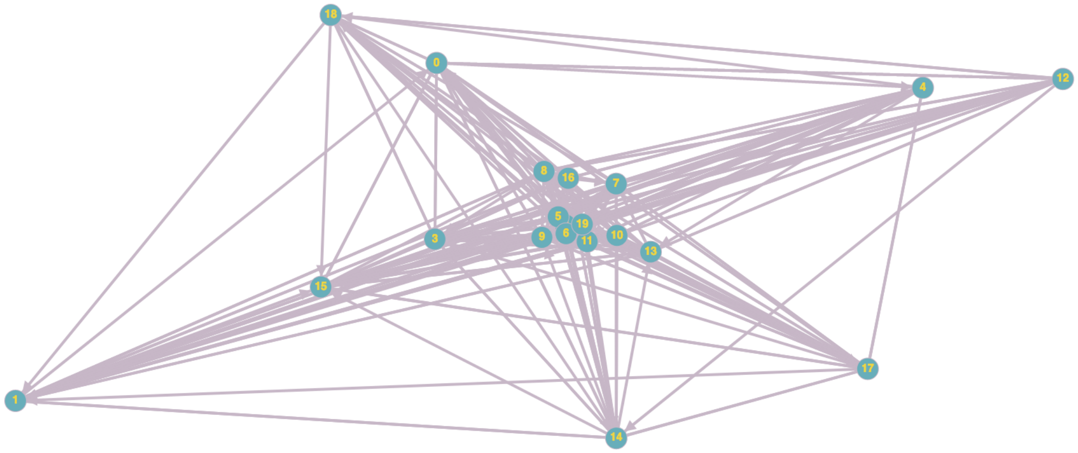
- Network topology for k=3



- Network topology for k=8



- Network topology for k=14



- We can clearly see that the **density** of the resulting network **increases** with increasing value of k.

6 Discussion

- Consider a single run of our program with any value of k (say k=5). For each node i, there will be 5 low cost links going out of i (cost=1). When the shortest path algorithm executes, it will try to **avoid** the high cost links (cost=100) going out of node i.
- Thus by **limiting k**, we limit the number of links that go out of any node i, and hence limit the **density** of the network
- This is the reason we observe that as k **increases**, the density of our resulting network also **increases**
- Consider any pair of nodes (s,t). Let i be any node i on the shortest path from s to t (t cannot be included in this set). As k increases we have more options at all such nodes i to reach t and we could take different paths.
- The number of edges which repeatedly fall in the shortest path between any 2 pair of nodes **decreases** and this is why the cost of the network **decreases**.
- The network has more of different edges of various edges of different costs rather than few edges some of which may have high costs

7 ReadMe File

This section shows how to run the project files.

- Downloads the project files and store them in a folder
- Open the project folder in Eclipse
- Open the file **NetworkDesignModel.java**
- Right Click − > Run as − > Java Application
- Alternatively,navigate to the folder in **terminal** and run the following commands
  - javac NetworkDesignModel.java
  - java NetworkDesignModel

8 Code

Module 1: InputGeneration.java

```java
import java.util.ArrayList;


public class InputGeneration {



  public int[][] generateGraph(int n,int m)
  {

    ArrayList<Edge> pairs=selectMEdges(m,n);

    int[][] arr=new int[n][n];
    for(int i=0;i<n;i++) {
      for(int j=0;j<n;j++)
      {
        arr[i][j]=0;
      }
    }

    for(int i=0;i<pairs.size();i++)
    {
      Edge edge=pairs.get(i);
      arr[edge.first()][edge.second()]=1;
      arr[edge.second()][edge.first()]=1;
    }

    //Utils.printMatrix(arr, n,”Graph topology”);
    return arr;
  }

  public ArrayList<Integer> getNodes(int n)
  {
    ArrayList<Integer>nodes=new ArrayList<Integer>();

  for(int j=0;j<n;j++)
  {
    nodes.add(j);
  }
  return nodes;
  }



  public  ArrayList<Edge> selectMEdges(int m, int n)
  {

    int low=0;
    int high=n-1;
    int range=high-low+1;

    ArrayList<Edge> edges=new ArrayList<Edge>();
    int i,j;
    while(edges.size()!=m)
    {
      i=(int)(Math.random()*range);
      j=(int)(Math.random()*range);
      if(i==j)
      {
        continue;
      }
      Edge e=new Edge(i,j);
      if(!checkEdgeExists(e,edges))
      {

        //System.out.println("Edge "+edges.size()+" : ("+ i+" "+j+") selected ");
        edges.add(new Edge(i,j));
      }
      else
      {
        //System.out.println("edge "+i+" "+j+") already there");
        }

    }
    return edges;

  }

  static boolean checkEdgeExists(Edge e, ArrayList<Edge> edges)
  {
    for(int i=0;i<edges.size();i++)
    {
      Edge f=edges.get(i);
      if(f.first()==e.first()&&f.second()==e.second())
      {
        return true;
      }
    }
    return false;
  }
}
```

Module 2: FloydWarshall.java

```java
import java.util.ArrayList;


public class InputGeneration {



  public int[][] generateGraph(int n,int m)
  {

    ArrayList<Edge> pairs=selectMEdges(m,n);

    int[][] arr=new int[n][n];
    for(int i=0;i<n;i++) {
      for(int j=0;j<n;j++)
      {
        arr[i][j]=0;
      }
    }

    for(int i=0;i<pairs.size();i++)
    {
      Edge edge=pairs.get(i);
      arr[edge.first()][edge.second()]=1;
      arr[edge.second()][edge.first()]=1;
    }

    //Utils.printMatrix(arr, n,”Graph topology”);
    return arr;
  }

  public ArrayList<Integer> getNodes(int n)
  {
    ArrayList<Integer>nodes=new ArrayList<Integer>();

  for(int j=0;j<n;j++)
  {
    nodes.add(j);
  }
  return nodes;
  }



  public  ArrayList<Edge> selectMEdges(int m, int n)
  {

    int low=0;
    int high=n-1;
    int range=high-low+1;

    ArrayList<Edge> edges=new ArrayList<Edge>();
    int i,j;
    while(edges.size()!=m)
    {
      i=(int)(Math.random()*range);
      j=(int)(Math.random()*range);
      if(i==j)
      {
        continue;
      }
      Edge e=new Edge(i,j);
      if(!checkEdgeExists(e,edges))
      {

        //System.out.println("Edge "+edges.size()+" : ("+ i+" "+j+") selected ");
        edges.add(new Edge(i,j));
      }
      else
      {
        //System.out.println("edge "+i+" "+j+") already there");
        }

    }
    return edges;

  }

  static boolean checkEdgeExists(Edge e, ArrayList<Edge> edges)
  {
    for(int i=0;i<edges.size();i++)
    {
      Edge f=edges.get(i);
      if(f.first()==e.first()&&f.second()==e.second())
      {
        return true;
      }
    }
    return false;
  }
}
```

Module 3: NetworkDesignModel.java

```
import java.util.ArrayList;


public class InputGeneration {



  public int[][] generateGraph(int n,int m)
  {

    ArrayList<Edge> pairs=selectMEdges(m,n);

    int[][] arr=new int[n][n];
    for(int i=0;i<n;i++) {
      for(int j=0;j<n;j++)
      {
        arr[i][j]=0;
      }
    }

    for(int i=0;i<pairs.size();i++)
    {
      Edge edge=pairs.get(i);
      arr[edge.first()][edge.second()]=1;
      arr[edge.second()][edge.first()]=1;
    }

    //Utils.printMatrix(arr, n,”Graph topology”);
    return arr;
  }

  public ArrayList<Integer> getNodes(int n)
  {
    ArrayList<Integer>nodes=new ArrayList<Integer>();

  for(int j=0;j<n;j++)
  {
    nodes.add(j);
  }
  return nodes;
  }



  public  ArrayList<Edge> selectMEdges(int m, int n)
  {

    int low=0;
    int high=n-1;
    int range=high-low+1;

    ArrayList<Edge> edges=new ArrayList<Edge>();
    int i,j;
    while(edges.size()!=m)
    {
      i=(int)(Math.random()*range);
      j=(int)(Math.random()*range);
      if(i==j)
      {
        continue;
      }
      Edge e=new Edge(i,j);
      if(!checkEdgeExists(e,edges))
      {

        //System.out.println("Edge "+edges.size()+" : ("+ i+" "+j+") selected ");
        edges.add(new Edge(i,j));
      }
      else
      {
        //System.out.println("edge "+i+" "+j+") already there");
        }

    }
    return edges;

  }

  static boolean checkEdgeExists(Edge e, ArrayList<Edge> edges)
  {
    for(int i=0;i<edges.size();i++)
    {
      Edge f=edges.get(i);
      if(f.first()==e.first()&&f.second()==e.second())
      {
        return true;
      }
    }
    return false;
  }
}
```

Utils.java

```java
import java.util.ArrayList;

public class Utils {
  public static void printMatrix(int[][] arr, int n, String s)
  {
    System.out.println(s);

    for(int i=0;i<n;i++)
    {
      System.out.print(”row ”+i+” : ”);
      for(int j=0;j<n;j++)
      {
        System.out.print(arr[i][j]+” ”);
      }
      System.out.println(”\n”);
    }
  }

  public static void printList(ArrayList<Integer> arr,String s)
  {
    int n=arr.size();
    System.out.println(s);
    for(int i=0;i<n;i++)
    {
      System.out.print(arr.get(i)+” ”);
    }
      System.out.println(”\n”);

  }



}
```

Edge.java

```java

public class Edge {

  private int i;
  private int j;
  public Edge(int x,int y)
  {
    this.i=x;
    this.j=y;
  }

  public int first()
  {
    return i;
  }

  public int second()
  {
    return j;
  }

}
```

9 References

- Lecture Notes - An Application to Network Design