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

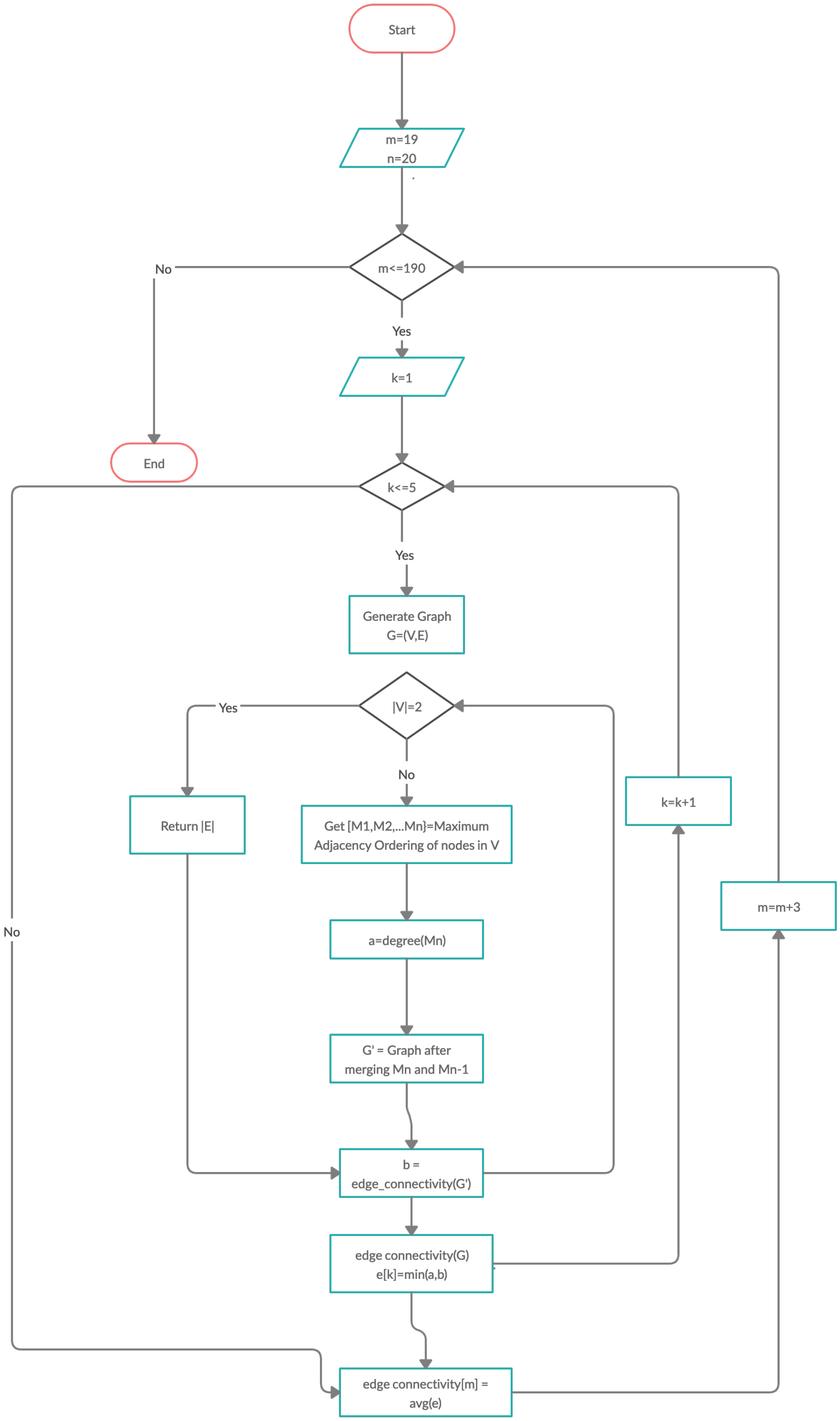
Module 2 receives input parameters from the first module (graph).

- 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, $pred_{ij}$=-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 $link_{ij}$ 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 **edge connectivity** for a graph with n nodes and m edges
- For each value of m we had generated 5 graphs. The edge connectivity for that value of m is the average of the edge connectivities of these 5 graphs
- The above computation is repeated for all **m** values
- We compute the **spread** of the each edge connectivity value.
- The *spread* of an edge connectivity value is the difference between the smallest and largest value of m for which the value occurs.
- We then try to answer questions such as:
  - How does the edge connectivity of the network vary with the value of m?
  - How does spread of an edge connectivity value vary with the value of m?

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



4 Nagamochi Ibaraki Algorithm - Explanation

- The goal is to find the edge connectivity of a graph G=(V,E) i.e. the minimum number of edges which must be deleted to disconnect G
- **Base Case**: There are only two nodes i and j in the graph, so the edge connectivity of the graph is the number of edges (if any) between i and j
- **Recursive Case**:
  - There are n nodes in the current graph, $n >= 3$
  - We find the maximum adjacency ordering of the nodes in the current graph. Let it be $v_1, v_2, ..v_{n-1}, v_n$
  - We take the last two nodes in the above ordering, x=v_{n-1} and y=v_n. Then, $\lambda(x, y)$= degree(y)
  - Merge nodes x and y into one node. Let G_{xy} be the resulting graph.
  - Find the edge connectivity of G_{xy}. Let it be b.
  - Return min(a,b)

Algorithm 1 NagamochiIbarakiAlgorithm

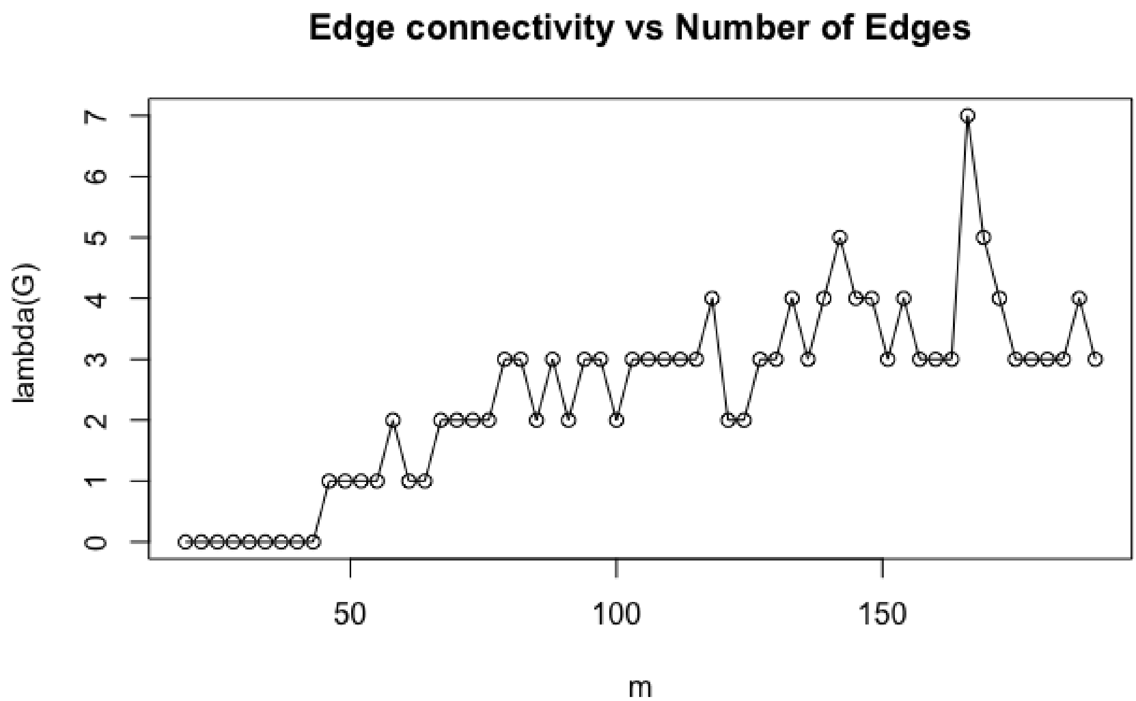
procedure NagamochIbaraki(V, E)
    $n =\mid V \mid$
    if $n = 2$ **then**
        return $\mid E \mid$
    end if
    $[v_1, v_2, ...v_n]$ = maximumAdjacencyOrdering(V)
    $x = v_{n-1}$
    $y = v_n$
    a=$degree(y)$
    G_{xy}=graph created by merging nodes x and y
    b = NagamochiIbaraki(V_{xy}, E_{xy})
    return min(a,b)
end procedure

- The result is the edge connectivity of the graph.

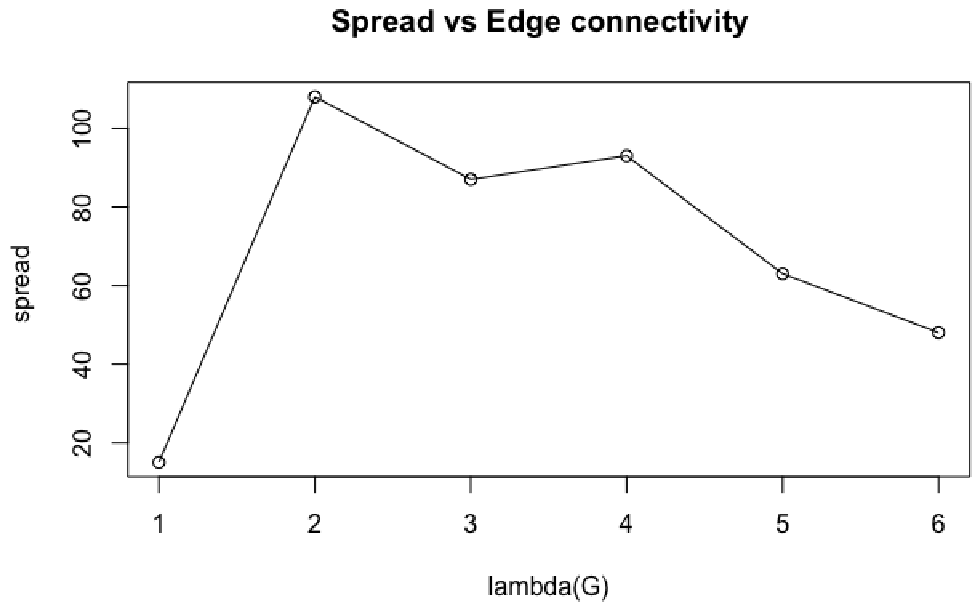
5 Observations and Analysis

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

- The output results are stored in a *csv* file. The graphs are generated in **R**
- We plot the graph of the number of edges **m** vs. the edge connectivity $\lambda(G)$. We can clearly see that the cost of the network **decreases** with increase in the value of k.



- We plot the graph of edge connectivity values **lambda(G)** against the **spread**. We can clearly see that the density of the network **increases** with increase in the value of k.



- 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 **Presentation.java**
- Right Click − > Run as − > Java Application
- Alternatively,navigate to the folder in **terminal** and run the following commands
  - javac Presentation.java
  - java Presentation

8 Code

Module 1: InputGeneration.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.getFirst()][edge.getSecond()]=1;
      arr[edge.getSecond()][edge.getFirst()]=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.getFirst()==e.getFirst()&&f.getSecond()==e.getSecond())
      {
        return true;
      }
    }
    return false;
  }
}
```

Module 2: NagamochiIbaraki.java

```
import java.util.ArrayList;

public class NagamochiIbaraki {

  private int n;
  private ArrayList<Integer> nodes;
  private int[][] graph;
  NagamochiIbaraki(int n)
  {
    this.n=n;
  }


  public static void main(String[] args)
  {
    int n=20;
    InputGeneration inputGeneration=new InputGeneration();
    int [][]graph=inputGeneration.generateGraph(20, 19);
    NagamochiIbaraki nagamochiIbaraki=new NagamochiIbaraki(n);
    System.out.println("Graph connectivity: "+nagamochiIbaraki.isConnected(
   graph));
  }

  public int runAlgorithm(int graph[][],ArrayList<Integer> nodes )
  {
    int p=nodes.size();
    //System.out.println("Number of nodes: "+p);
    if(nodes.size()==2)
    {
      //System.out.println("Reached base case");
      //Utils.printMatrix(graph, n, "Contracted graph");
      return getDegree(graph,nodes);
    }

    //int p=nodes.size();

      nodes=maximumAdjacency(graph,nodes);
      //Utils.printList(nodes,"Maximum Adjacency Ordering");
      int a =getDegree(graph,nodes);
      graph=contractGraph(graph,nodes);
      nodes.remove(nodes.size()-1);
      int b=runAlgorithm(graph,nodes);
      return Math.min(a,b);

  }


  public int getDegree(int graph[][],ArrayList<Integer>nodes)
```

```
{
  int p=nodes.size();
  int node=nodes.get(p-1);
  int degree=0;
  for(int i=0;i<p-1;i++)
  {

      degree=degree+graph[node][nodes.get(i)];

  }

  return degree;
}

public int[][] contractGraph(int graph[][],ArrayList<Integer>nodes)
{
  int p=nodes.size();
  int x=nodes.get(p-2);
  int y=nodes.get(p-1);
  graph[x][y]=0;
  graph[y][x]=0;
  for(int i=0;i<p-2;i++)
  {

    int k=nodes.get(i);

      graph[x][k]+=graph[y][k];

  }

  //System.out.println("Node "+y+" contracted into "+x);
  return graph;
}

public ArrayList<Integer> maximumAdjacency(int [][] graph, ArrayList<Integer> nodes)
{


  ArrayList<Integer> temp=new ArrayList<Integer>();
  int p=nodes.size();
  int v=(int)(Math.random()*p);
  int v1=nodes.get(v);

  temp.add(v1);

  for(int i=2;i<=p;i++)
  {
    v=chooseNode(graph,nodes,temp);
```

```
      temp.add(v);

    }
    return temp;
  }


  public int chooseNode(int graph[][], ArrayList<Integer> nodes,ArrayList<
   Integer> cur_set)
  {
    int node=0,res=-1,count;
    int p=nodes.size();
    int q=cur_set.size();
    for(int i=0;i<p;i++)
    {
      int k=nodes.get(i);
      if(cur_set.contains(k))
      {
        continue;
      }

      count=0;
      for(int j=0;j<q;j++)
      {
        if(graph[k][cur_set.get(j)]==1)
            {count++;
            }

      }
      if(count>res)
      {
        res=count;
        node=k;
      }
    }
    return node;
  }

  public boolean isConnected(int graph[][])
  {
    nodes=new ArrayList<Integer>();
    this.graph=graph;
    DFS(0);
    if(nodes.size()!=n)
    {
      return false;
    }
    return true;

```

```
  }


  public void DFS(int v)
  {
    nodes.add(v);
    for(int i=0;i<n;i++)
    {
      if(graph[v][i]!=0&&!nodes.contains(i))
      {
        DFS(i);
      }
    }

  }
}
```

Module 3: Presentation.java

```java
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map;
public class Presentation {

  public static void main(String[] args)
  {
    int n=20;
    int graph[][];
    InputGeneration inputGeneration=new InputGeneration();
    ArrayList<Integer>nodes =inputGeneration.getNodes(n);
    NagamochiIbaraki nagamochiIbaraki=new NagamochiIbaraki(n);

    /* Running our analysis */
    Map<Integer,Edge> mp=new HashMap<Integer,Edge>();
    for(int m=19;m<=190;m=m+3)
    {

      int sum=0;
      /* taking average of 5 trials */
      for(int i=0;i<5;i++)
      {
        graph=inputGeneration.generateGraph(n, m);
        if(nagamochiIbaraki.isConnected(graph))
        {
          sum+=nagamochiIbaraki.runAlgorithm(graph, nodes);
        }


      }
      sum=sum/5;
      System.out.println("m="+m+"  edge connectivity: "+sum);
      //System.out.println(m+"," + sum);

      /* update lowest and highest values of m for which edge connectivity=sum */
      if(!mp.containsKey(sum))
      {
        mp.put(sum, new Edge(m,m));
      }
      else
      {
        Edge e=mp.get(sum);
        mp.put(sum, new Edge(e.getFirst(),m));
      }

    }

```

```
    System.out.println("Spread of edge connectivities...");

      mp.forEach((k,v)->System.out.println("Minimum cut="+k+" lowest="+v.
    getFirst()+" highest: "
      +v.getSecond()+" spread: "+Math.abs(v.getSecond()-v.getFirst())));
      //mp.forEach((k,v)->System.out.println(k+","+Math.abs(v.getSecond()-v.
    getFirst())));

  }

}
```

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 first;
  private int second;
  public Edge(int x,int y)
  {
    this.first=x;
    this.second=y;
  }

```

```
  public int getFirst()
  {
    return first;
  }

  public int getSecond()
  {
    return second;
  }

  public void setFirst(int s)
  {
    this.first=s;
  }

  public void setSecond(int s)
  {
    this.second=s;
  }

}
```

Visualization.R.java

```
data <- read.csv(file="/users/psprao/eclipse-workspace/Nagamochi-Ibaraki/
    output1.csv")

# Getting K, cost and density  data
m<-data[,1]
lambda<-data[,2]


# Scatterplot of K vs Cost of Network
plot(m,lambda,xlab="m",ylab="lambda(G) ",main="Edge connectivity vs Number of
    Edges")
lines(lambda~m)


data <- read.csv(file="/users/psprao/eclipse-workspace/Nagamochi-Ibaraki/
    output.csv")

# Getting K, cost and density  data
lambda<-data[,1]
spread<-data[,2]


# Scatterplot of K vs Cost of Network
plot(lambda,spread,xlab="lambda(G)",ylab="spread ",main="Spread vs Edge
    connectivity  ")
lines(spread~lambda)
```

9 References

- Lecture Notes - An Application to Network Design