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 Nagamochi Ibaraki algorithm for finding the edge connectivity of a connected graph
- Give an input graph with n nodes and m edges, our program outputs the edge connectivity of the graph
- We do this for range of values of m
- We also analyze how the edge connectivity varies with the value of m and assess the spread of the edge connectivity values
- 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

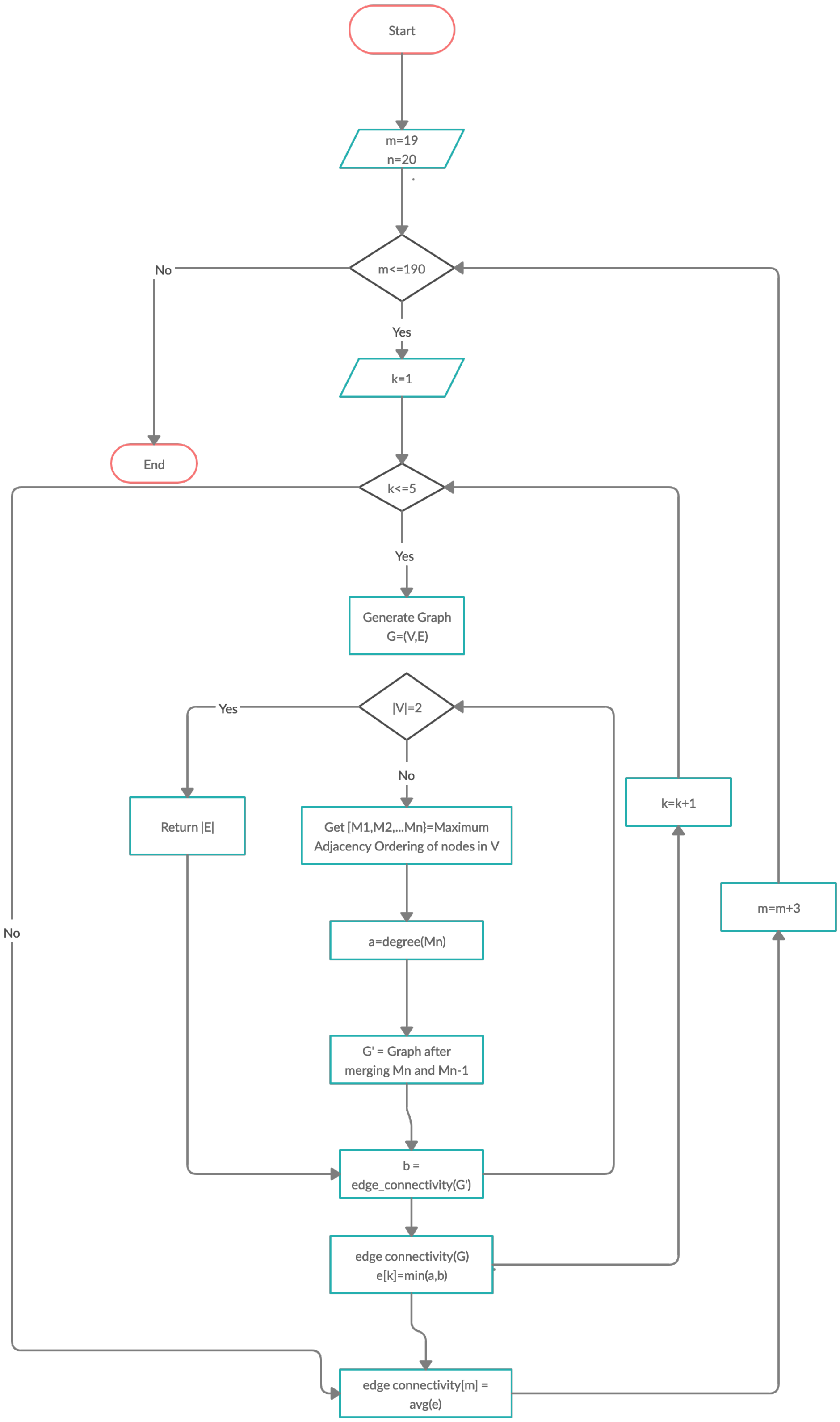
Module 2 receives input parameters from the first module (graph). It has the following functions.

- **isConnected()**
  - checks if the input graph is connected using depth-first search
  - If the input graph is disconnected, its edge connectivity is zero
- **runAlgorithm()**
  - Runs the Nagamochi Ibaraki algorithm on an input graph
  - If there are only two nodes in the graph, the number of edges between the two nodes is the edge connectivity
  - If the number of nodes in the current graph is greater than 2, we run the *maximum adjacency ordering* algorithm
  - Compute the **degree** between the last node in the ordering
  - The last two nodes in the ordering are merged
  - The **edge connectivity** of the *merged graph* is computed recursively
  - The edge connectivity of the graph is minimum of the values of the two values
- **maximumAdjacency()**
  - Returns the maximum adjacency ordering of the nodes in the current graph.
  - The first node in the ordering is chosen *randomly*
  - Given the first k nodes $(v_1, v_2, ...v_k)$ are chosen, the node which has the *maximum* edges to the set $(v_1, ...v_k)$ is chosen to be v_{k+1}
- **contractGraph()**
  - Contracts two nodes in a graph
  - All resulting self loops are removed
  - Parallel edges are kept
- **dfs()**
  - Runs a depth first search on a graph

Modul2 2 computes the edge connectivity of an input graph and passes it on to Module 3.

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 in the range [19,190] in steps of 3
- 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 the **edge connectivity**?
- We present the execution of our project using the flow chart below.


Start
m=19
n=20
m<=190
No
Yes
k=1
End
k<=5
Yes
Generate Graph
G=(V,E)
|V|=2
Yes
No
Return |E|
Get [M1,M2,...Mn]=Maximum
Adjacency Ordering of nodes in V
k=k+1
m=m+3
No
a=degree(Mn)
G' = Graph after
merging Mn and Mn-1
b =
edge_connectivity(G')
edge connectivity(G)
e[k]=min(a,b)
edge connectivity[m] =
avg(e)

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
- If the graph is not connected, its edge connectivity is **zero**
- **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, $a = \lambda(x, y)$= 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)

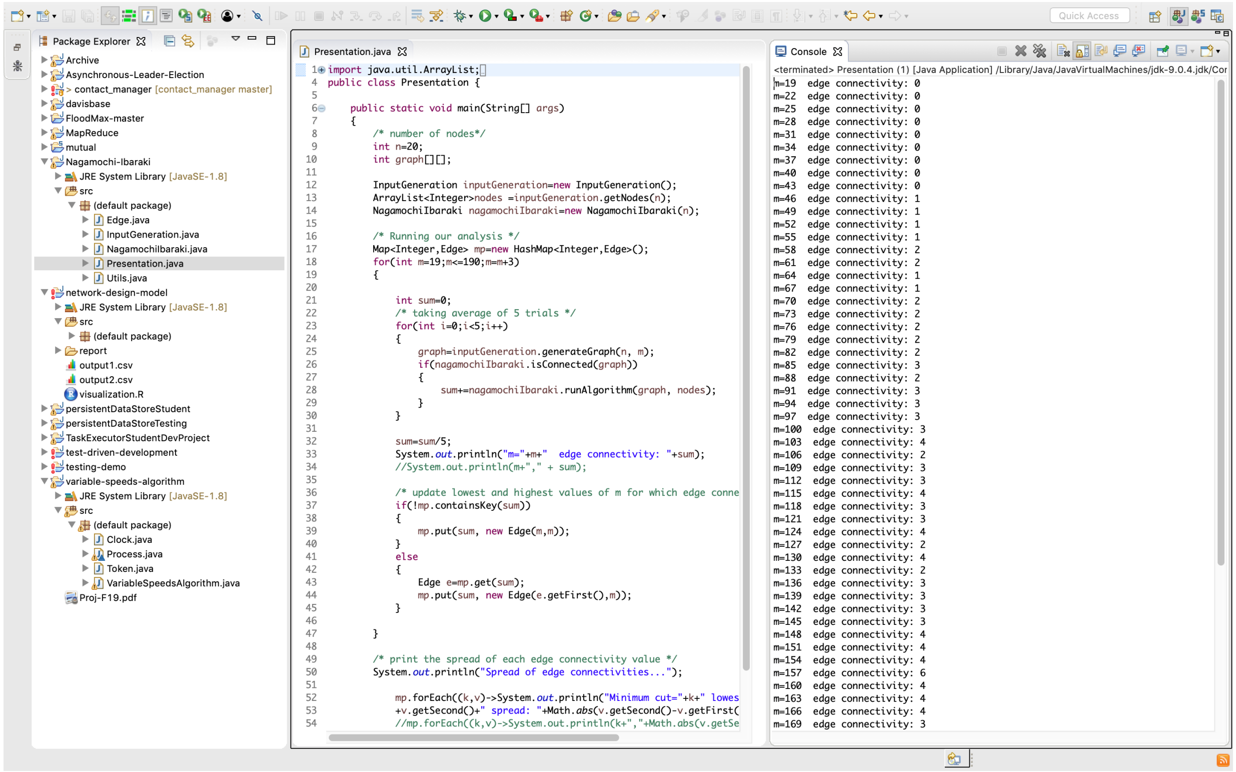
Algorithm 1 NagamochiIbarakiAlgorithm

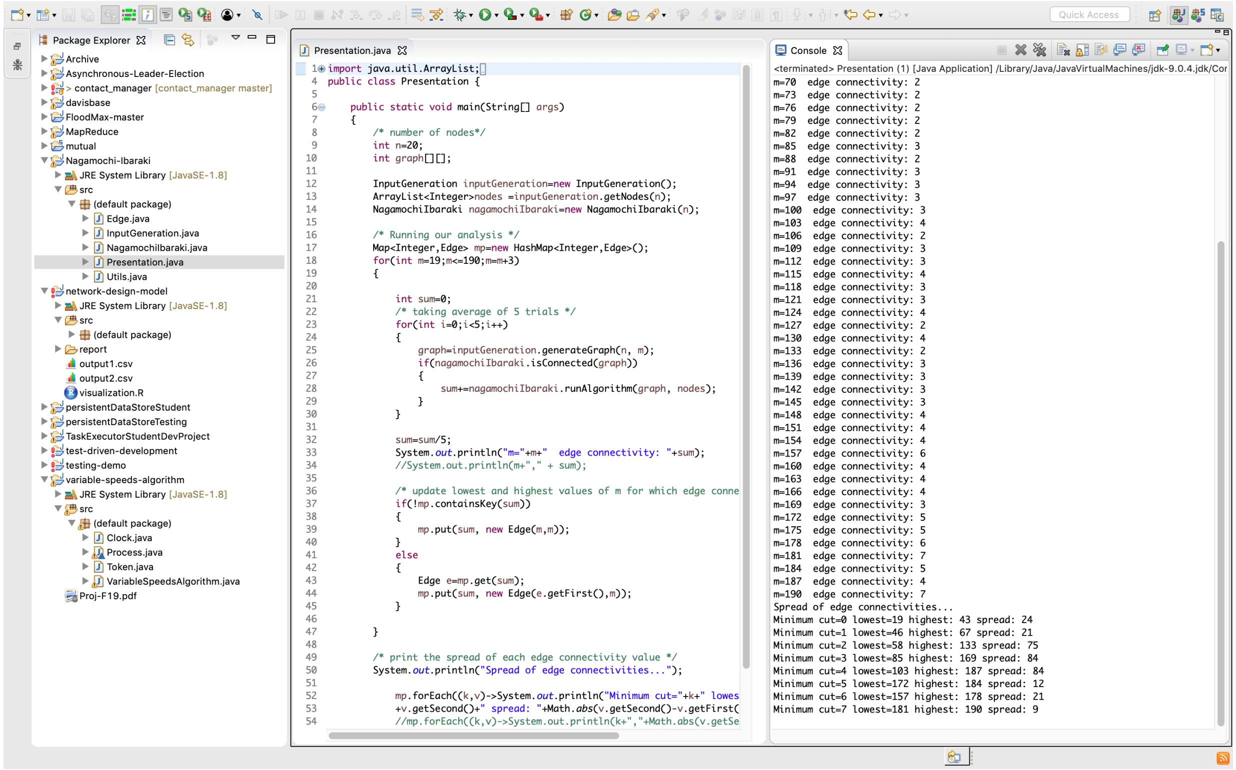
procedure NagamochIbaraki(V, E)
    $n =\mid V \mid$
    if $n = 2$ **then**
        return $\mid E \mid$
    end if
    $[v_1, v_2, ...v_n]$ = maximumAdjacencyOrdering(V)
    $x = v_{n-1}$
    $y = v_n$
    a=$degree(y)$
    G_{xy}=graph created by merging nodes x and y
    b = NagamochiIbaraki(V_{xy},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

- The result is the edge connectivity of the graph.

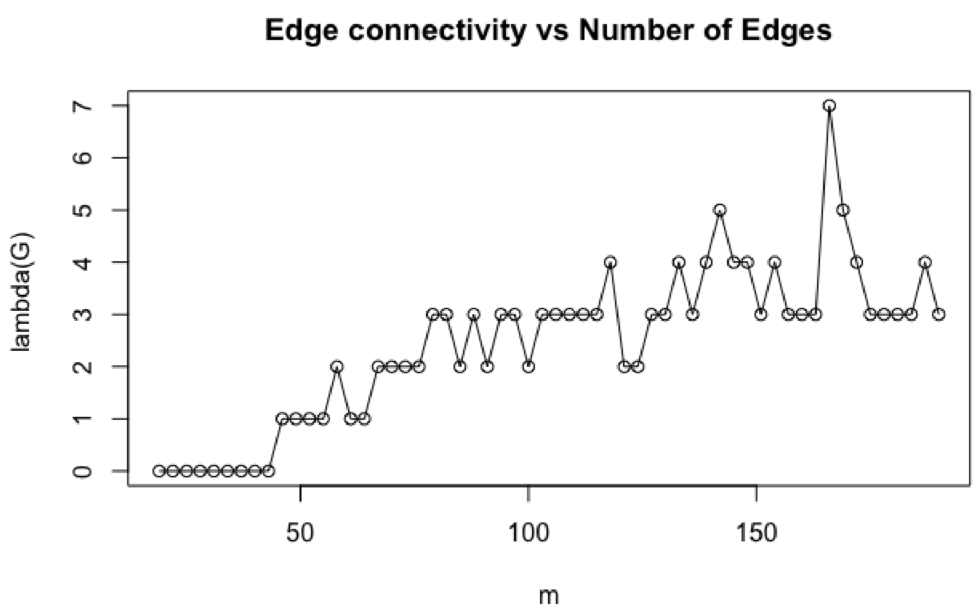
5 Observations and Analysis

- The program produces outputs as shown in the figures below.

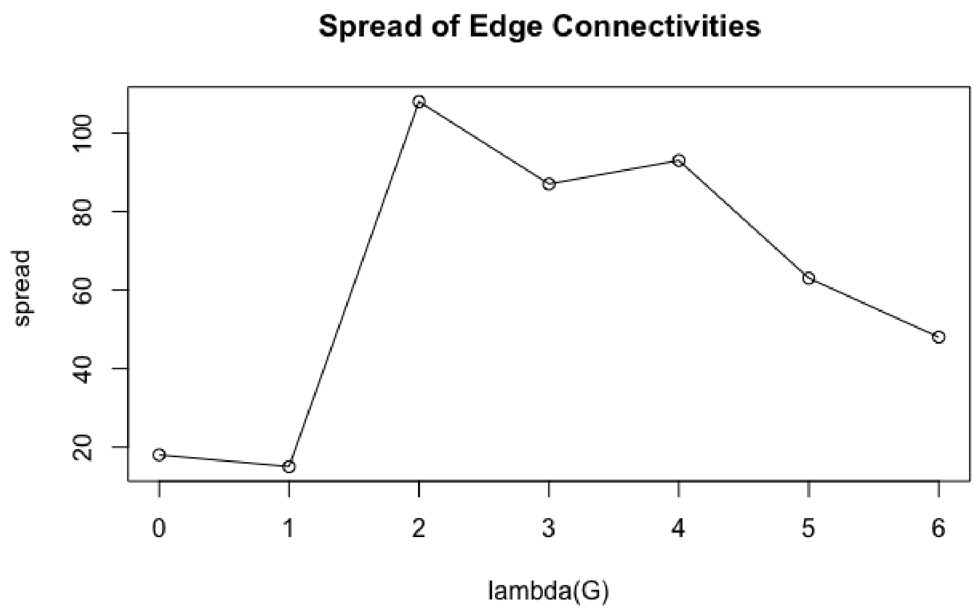




- The output results are stored in a *csv* file. The graphs are generated in **R**.
- We plot the graph of the *number of edges* m vs. the *edge connectivity* $\lambda(G)$



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



- We can clearly see that the **spread** of edge connectivity values increases, saturates in the middle and then decreases

6 Discussion

- As the number of edges in the graph increases, the number of edge disjoint paths between any two nodes tends to increase, hence the edge connectivity of the graph also **increases**
- In our observations, the highest values of spread occur for $\lambda(G) = 2, 3, 4$ with decreasing values on either side
- The reason is that a given value of edges connectivity (especially the ones in the middle) tends to persist for a range of m values, especially since for each graph we choose edges randomly, which sort of randomises our progression as m increases

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

```java
import java.util.ArrayList;

public class NagamochiIbaraki {

    private int n;
    private ArrayList < Integer > nodes;
    private int[][] graph;

    public 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 (p == 2) {
            //System.out.println("Reached base case");
            //Utils.printMatrix(graph, n, "Contracted graph");
            return getDegree(graph, nodes);
        }



        nodes = maximumAdjacency(graph, nodes);
        //Utils.printList(nodes,"Maximum Adjacency Ordering");
        int a = getDegree(graph, nodes);
        graph = contractGraph(graph, nodes);
        nodes.remove(p - 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

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

  public static void main(String[] args)
  {
    /* number of nodes*/
    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));
      }

    }
```

```

    /* print the spread of each edge connectivity value */
    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)
```