11/3/2014

Implementation of Nagamochi-Ibaraki Algorithm

Algorithmic Aspect of Telecommunication Networks

Rahul Singhal
RXS132730

Table of Contents

1 Introduction 2

2 Nagamochi-Ibaraki Algorithm 2

3 Data structure and Algorithmic implementation 2

4 Experiment Results 4

4.1 λ(G) vs Density 4

4.2 Critical Edge vs Density 5

4.3 Graph Topology 6

5 Observations from the Results 10

5.1 λ dependency on density 10

5.2 Critical Edge vs Density 10

6 Conclusion 10

7 Source Code 10

8 Readme 18

1 Introduction

This project attempts to measure the network reliability by finding the minimum cut of a network graph G. The minimum-cut is the number of edges that can be removed from a graph to reduce the connectivity of the graph.

$\lambda(G)$ be the edge-connectivity of the graph G i.e. the number of edges that are to be removed in order to render the graph G as disconnected.

$\lambda(i, j)$ be the connectivity between nodes i and j i.e. the number of edges to be removed so that there is no path from i to j.

$G_{i,j}$ be the graph obtained by the merging of vertices i and j into one single node ij such that all the edges incident on i and all the edges incident on j are not incident on ij.

The goal of this project is to compute $\lambda(G)$ for a variety of test cases and study its behavior against the density of the graph which is defines as the average number of edges incident on any node of the graph.

2 Nagamochi-Ibaraki Algorithm

For any i and j $\lambda(G) = \min\{ \lambda(i, j), \lambda(G_{ij}) \}$. The problem lies in picking the right i and j. For V vertices we can get V(V-1)/2 i and j pairs. The Nagamochi-Ibaraki algorithm simplifies this by using (Maximum Adjacency) MA ordering with which we can arrive at the result in V-1 iterations.

The Maximum Adjacency (MA) ordering of a set of nodes v1, v2,…vn can be found by the following steps

i. Pick a node v1 at random
ii. Pick vi+1 such that the maximum number of edges are connecting vi+1 to the set {v1,…vi}

Given a MA ordering v1,..vn; λ(vn-1, vn) = d(vn), where d(vn) is the degree of node vn. For all the other cases, from a list of k remaining nodes, find the Maximum Adjacency ordering.

3 Data structure and Algorithmic implementation

Graph: Used to store network graph in the form of an adjacency matrix, and methods for initializing the graph.

a. AdjacencyMatrix: Stores the number of parallel edges between any two nodes i and j given by a_{ij}.
b. isConnected() : This method returns a Boolean value indicating if the graph is connected or not.

Input:

a. Number of nodes (n)
b. Number of edges (m)

Program execution:

```
V: initialize to 20
Initialize a set of array lists to store the statistics from each iteration
densities: store the density of the graph at each iteration
lamdaG: store λ(G) values for each graph
criticalEdges: store the number of critical edges in the network
for m=40,45,50,…400
        Initialize graph G with V=20
        G.adjacencyMatrix[i,j] = 0
        edges =m
                while edges > 0
                        randomly pick i and j from 0 to V-1
                        adjacencyMatrix[i,j] += 1
                        adjacencyMatrix[j,i] += 1
                        edges = edges – 1
                S = set of all vertices of the graph
                criticalEdges = 0
                lamdaG = lamdaG U {computeLamda(G, S)}
                densities = densities U {2*m / V}
```

Graph connectivity λ(G)

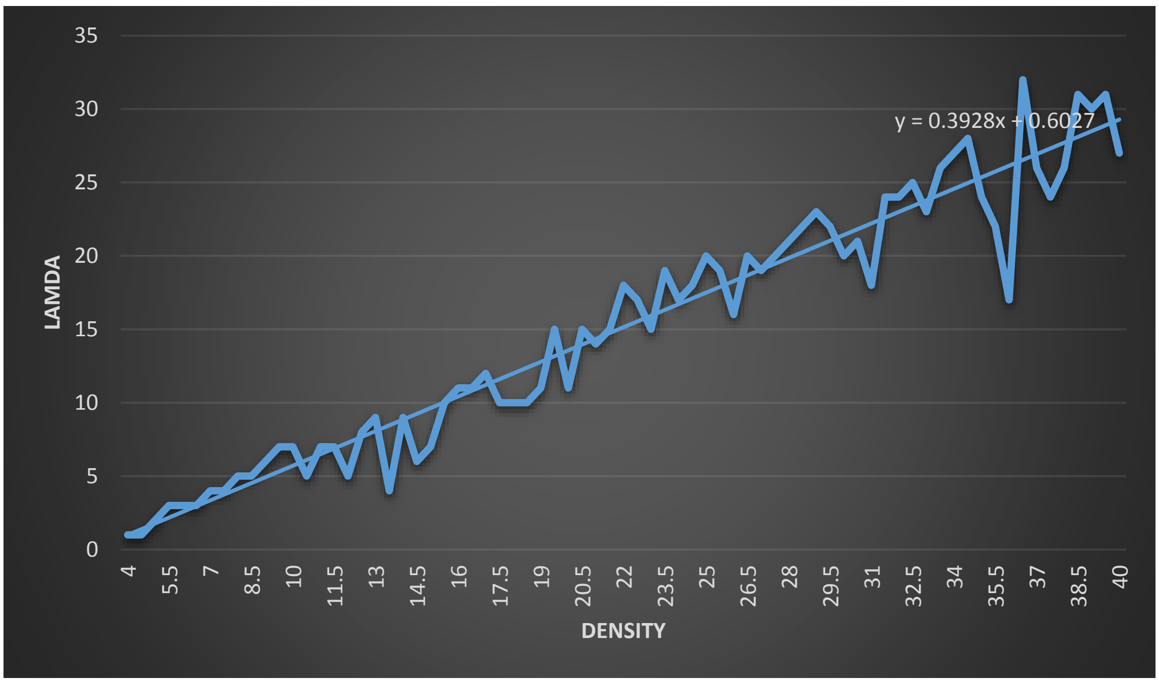
```
computeLamda(G,S)
        if S.size() == 2 //only two nodes are remaining
                return adjacencyMatrix[S.last, S.last-1]
        MA = initialize an empty list MA for Maximum Adjacency ordering
        createMAOrdering(MA, G, S) //adds MA list with elements in MA order
        x = last element of MA
        y = penultimate element of MA
        λxy = degreeOf(y)
        edgeCriticality = AdjacencyMatrix[x,y]
        Gxy = merge(x,y,G) //merge the nodes x & y in the graph G
        λ(Gxy) = computeLamda(Gxy)
        if(λxy < λ(Gxy))
                criticalEdges = edgeCriticality + criticalEdges
                return λxy
        else
                return λ(Gxy)

createMAOrdering(MA, G)
        v1 = randomly choose a vertex from S
        MA = MA U {v1}
        for i in 1..S.size – 1 // remaining nodes of set S
                max = maximum edges to MA
                maIndex = vertex with max degree to MA set
```

```
for s in S
        if MA contains s
                continue
for m in MA
        if(max < AdjacecnyMatrix[s,m])
                max = AdjacencyMatrix[s,m]
                maIndex = s
        MA = MA U {maIndex}
```

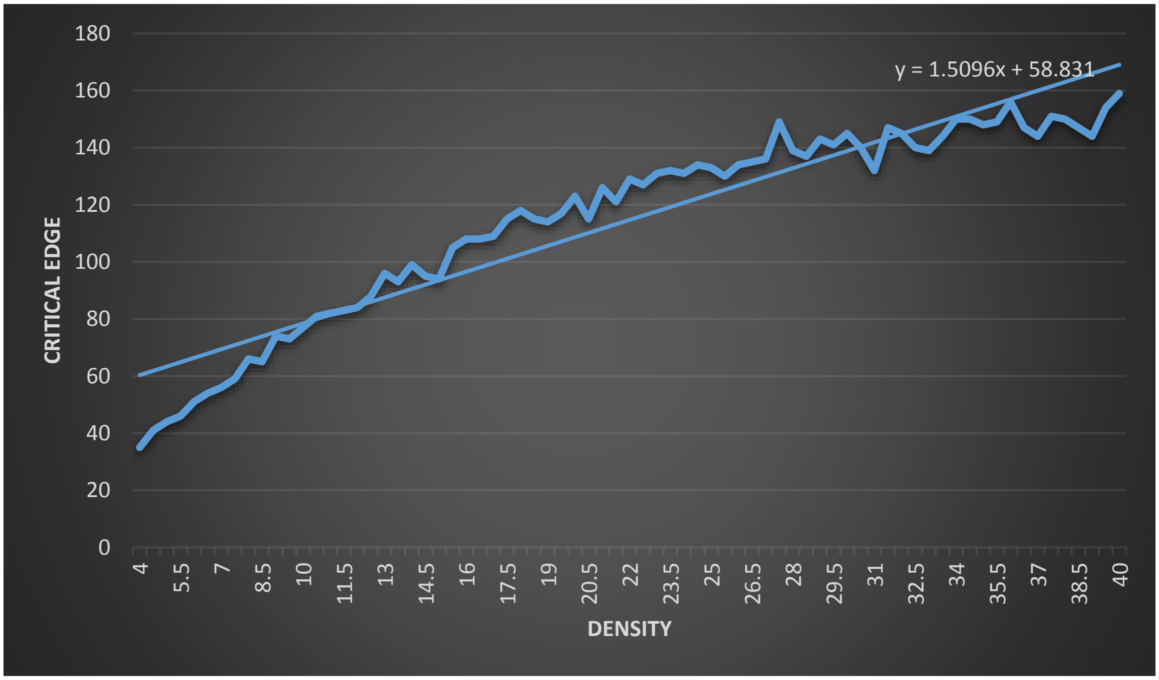
4 Experiment Results

4.1 λ(G) vs Density



The above graph depicts the relationship between Lamda vs Density of the graph. Also, the linear line shows the relationship of the graph with the function.

4.2 Critical Edge vs Density



The above graph depicts the relationship between Critical Edge vs Density of the graph. Also, the linear line shows the relationship of the graph with the function.

4.3 Graph Topology

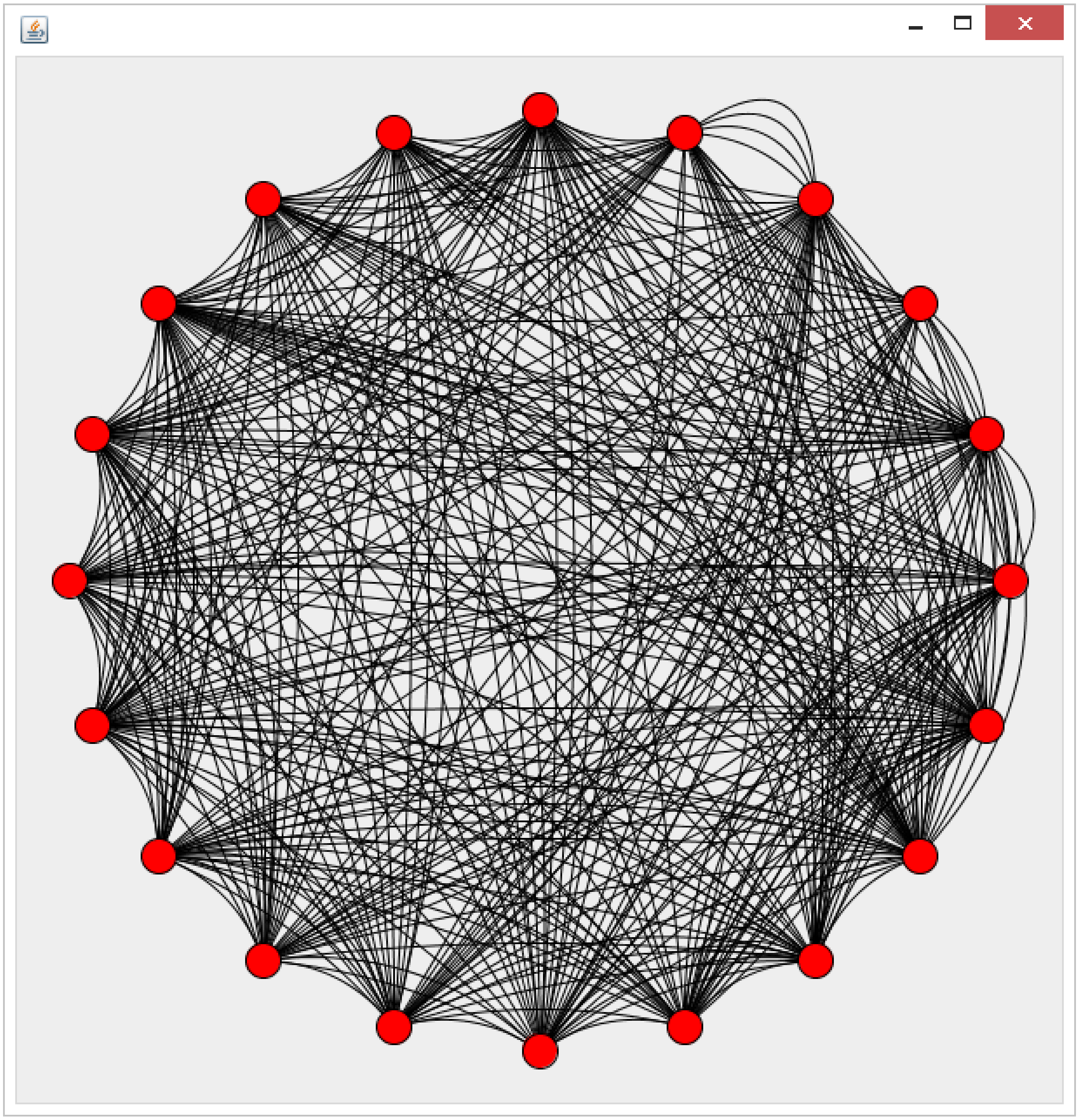


Figure 1Main topology for n = 20 and m = 400

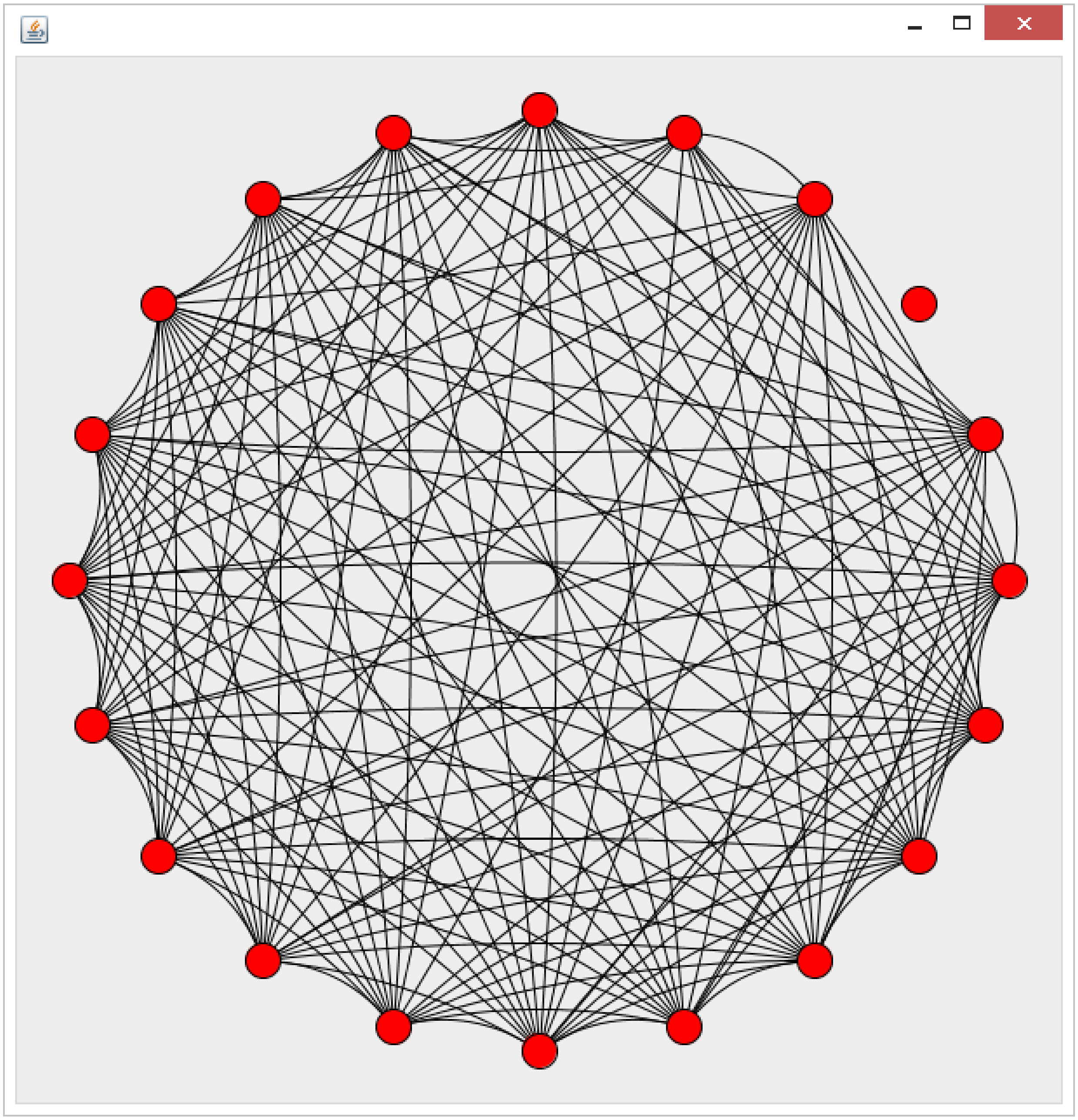


Figure 2Critical Edges for n = 20 and m = 400

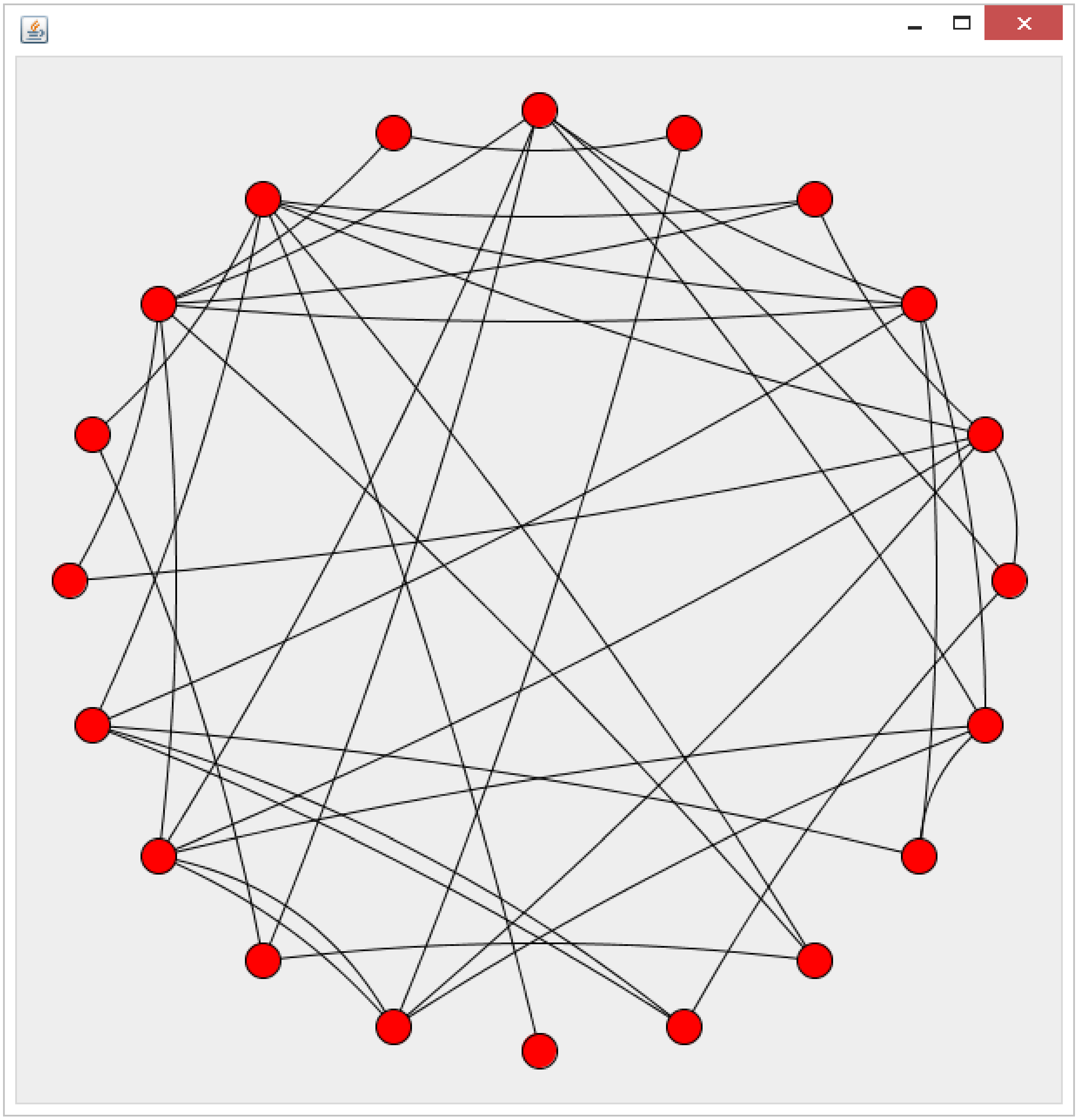


Figure 3Main topology for n = 20 and m = 40

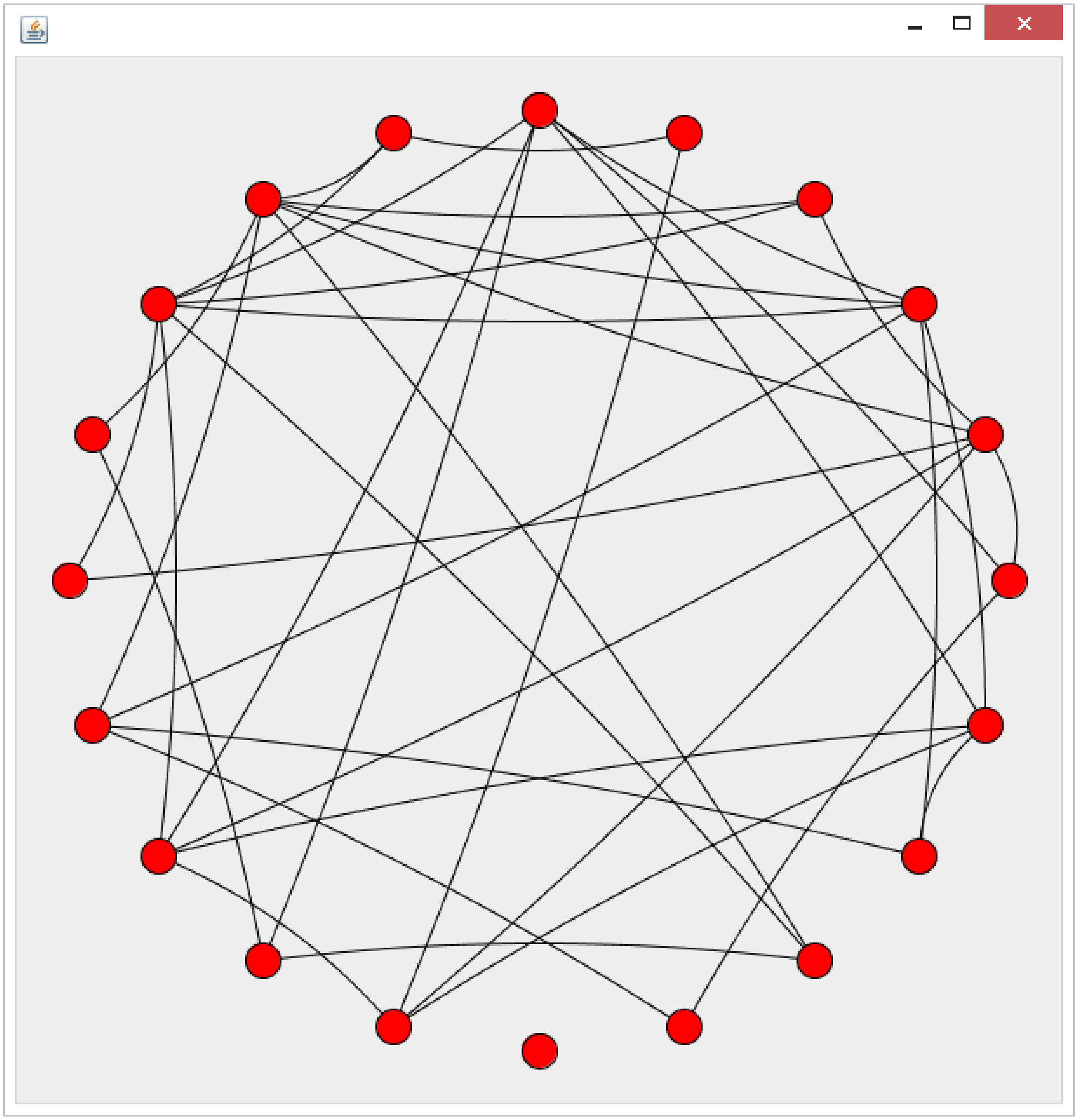


Figure 4Critical Edges for n = 20 and m = 40

5 Observations from the Results

5.1 λ dependency on density

λ tends to be non-decreasing on density. This was expected as m increases, the density of the graph increases too; more edges are being added to the graph, hence λ should be at least as big as the previous value. If any new parallel edges are being added to the critical links of the graph, it needs to be removed so λ would either increase or remains constant.

5.2 Critical Edge vs Density

The critical edges vs density has an increasing linear graph for higher density as the number of critical edges increases for two reasons:

i. The connectivity of the graph increases with more denser graph
ii. Higher the connectivity of the graph, more choice for which of the edges will become critical edges

6 Conclusion

The above approach gives us a very fast program for finding the minimum cut of a graph. As we can observe from the plots, minimum cut directly correlates with the graph density in the sense that edge connectivity trends in a non-decreasing fashion against the graph density.

7 Source Code

```
import java.util.ArrayList;
import java.util.Random;

public class NIAlgorithm {

	int number_of_nodes = -1;
	int number_of_edges = -1;
	ArrayList<Integer> lowestDegree = new ArrayList<>();
	ArrayList<Double> densities = new ArrayList<>();
	ArrayList<Integer> lamdaG = new ArrayList<>();
	ArrayList<Integer> criticalEdges = new ArrayList<>();
	ArrayList<Integer> critEdge = new ArrayList<>();
	ArrayList<Integer> allNodes = null;
	private int criticalEdgeCount;
	static int[][] adjacencyMatrix = null;
	static int[][] adjMat = null;
	static int[] degreeVector = null;
	static Random rand = null;
	static DrawGraph g = null;
	static int count = 0;

	public NIAlgorithm(int number_of_nodes, int number_of_edges) {
		this.number_of_edges = number_of_edges;
		this.number_of_nodes = number_of_nodes;
```

```
    }

    public void GenerateGraph() {
        for (int i = 0; i < number_of_nodes; i++) {
            degreeVector[i] = 0;
            for (int j = 0; j < number_of_nodes; j++) {
                adjacencyMatrix[i][j] = 0;
            }
        }
        // allNodes is an arrayList which stores the nodes in the network
        allNodes = new ArrayList<>();
        for (int i = 0; i < number_of_nodes; i++) {
            allNodes.add(i);
        }
        // generate the adjacency matrix for tracking the number of edges &
        // degree of a node
        int edges = number_of_edges;
        while (edges > 0) {
            int i = rand.nextInt(number_of_nodes);
            int j = rand.nextInt(number_of_nodes);
            if (i != j) {
                edges--;
                adjacencyMatrix[i][j] += 1;
                adjacencyMatrix[j][i] += 1;
                degreeVector[i] += 1; // degree of a vertices
                degreeVector[j] += 1; // degree of a vertices
            }
        }

    }

    private void ComputeMinDegreeAndDensity() {
        // compute the minimum degree of the network
        int minDegree = degreeVector[0];
        for (int z = 0; z < number_of_nodes; z++) {
            if (minDegree > degreeVector[z]) {
                minDegree = degreeVector[z];
            }
        }
        // lowest degree and density of the graph
        lowestDegree.add(minDegree);
        // compute and record the densities, 2*m/n
        densities.add((2 * number_of_edges) / (double) number_of_nodes);
    }

    private void ComputeLamdaG() {
        // check if the graph is connected
        if (!networkIsConnected()) {
            lamdaG.add(0);
            criticalEdges.add(0);
            return;
        }
        // new adjacency matrix for computation purposes
        int[][] localAdjMatrix = new int[number_of_nodes][number_of_nodes];
        for (int i = 0; i < number_of_nodes; i++) {
```

```
                for (int j = 0; j < number_of_nodes; j++) {
                        localAdjMatrix[i][j] = adjacencyMatrix[i][j];
                }
        }
        // initialization of criticalEdgeCount to 0, modifies in the Lamda
        // function call
        criticalEdgeCount = 0;
        // compute the connectivity of the graph
        int lG = Lamda(localAdjMatrix, allNodes);
        // recording the lamda / connectivity of a graph
        lamdaG.add(lG);
        criticalEdges.add(criticalEdgeCount);
}

private int Lamda(int[][] localAdjMatrix, ArrayList<Integer> allNodes2) {
        // exit condition of the recursion, returns the last 2 values of the
        // adjacencyMatrix
        if (allNodes2.size() == 2) {
                return localAdjMatrix[allNodes2.get(0)][allNodes2.get(1)];
        }

        ArrayList<Integer> MAList = new ArrayList<>();
        // compute the MA (Maximim ordering) for the localAdjacency Matrix
        createMAOrder(MAList, localAdjMatrix, allNodes2);
        int x = MAList.get(MAList.size() - 2); // Vn-1
        int y = MAList.get(MAList.size() - 1); // Vn
        // compute the lamda XY value from the adjacency matrix for the yth
        // vertex
        int lamdaXY = degreeOf(y, localAdjMatrix);
        int criticalEdge = localAdjMatrix[x][y];
        // merge the x & y vertex for the given adjacency matrix
        merge(x, y, localAdjMatrix);
        // remove the yth vertex
        for (int i = 0; i < allNodes2.size(); i++) {
                if (allNodes2.get(i) == y) {
                        allNodes2.remove(i);
                        break;
                }
        }

        int lamdaGXY = Lamda(localAdjMatrix, allNodes2);
        if (lamdaXY < lamdaGXY) {
                // removing x,y decreases the connectivity of the graph
                criticalEdgeCount += criticalEdge;
                return lamdaXY;
        } else {
                return lamdaGXY;
        }
}

private void merge(int x, int y, int[][] localAdjMatrix) {
        localAdjMatrix[x][y] = 0;
        localAdjMatrix[y][x] = 0;
        for (int i = 0; i < number_of_nodes; i++) {
                if (i != x) {
```

```
                    localAdjMatrix[x][i] += localAdjMatrix[y][i];
                    localAdjMatrix[i][x] = localAdjMatrix[x][i];
                    localAdjMatrix[y][i] = 0;
                    localAdjMatrix[i][y] = 0;
                }
            }
            count++;
            if (count == 1) {
                // g.jung(localAdjMatrix, number_of_nodes);
            }
    }

    private int degreeOf(int y, int[][] localAdjMatrix) {
            // compute the degree of the graph for the yth vertex
            int degree = 0;
            for (int i = 0; i < number_of_nodes; i++) {
                degree += localAdjMatrix[y][i];
            }
            return degree;
    }

    private void createMAOrder(ArrayList<Integer> mAList,
                int[][] localAdjMatrix, ArrayList<Integer> allNodes2) {
            // randomly select a vertex from the graph
            int v = allNodes2.get(Math.abs(rand.nextInt() % allNodes2.size()));
            // add the random vertex v to a list
            mAList.add(v);
            // loop until (allNodes2's size - 1) except v
            for (int i = 0; i < allNodes2.size() - 1; i++) {
                int[] degreeVectorToMA = new int[allNodes2.size()];
                int max = 0;
                int maxIndex = 0;

                for (int j = 0; j < allNodes2.size(); j++) {
                    // get the jth vertex
                    int jIndex = allNodes2.get(j);
                    // check if the maximum adjacency list contains the jth
                    vertex,
                    // if so continue
                    if (mAList.contains(jIndex))
                        continue;
                    // else, for each of the maximum adjacency list vertex
                    for (Integer k : mAList) {
                        // compute the number of edges of the adjacency
                        matrix
                        degreeVectorToMA[j] = localAdjMatrix[jIndex][k];
                        // get the vertex with the maximum number of edges
                        if (degreeVectorToMA[j] > max) {
                            maxIndex = jIndex;
                            max = degreeVectorToMA[j];
                        }
                    }
                }
                // add the vertex which has the maximum edges to a list
                mAList.add(maxIndex);
```

```
        }
    }

    private boolean networkIsConnected() {
        boolean[] toVisit = new boolean[number_of_nodes];
        int vertices = number_of_nodes;
        ArrayList<Integer> nextNode = new ArrayList<>();
        nextNode.add(0);

        while (nextNode.size() != 0) {
            int i = nextNode.get(0);
            toVisit[i] = true;
            vertices--;
            nextNode.remove(0);
            for (int k = 0; k < number_of_nodes; k++) {
                if (adjacencyMatrix[i][k] > 0 && !toVisit[k]
                        && !nextNode.contains(k)) {
                    nextNode.add(k);
                }
            }
        }

        if (vertices == 0) {
            return true;
        } else {
            System.out.println("Not connected nodes are..");
            for (int i = 0; i < number_of_nodes; i++) {
                if (!toVisit[i]) {
                    System.out.println(i);
                }
            }
            return false;
        }
    }

    private int CriticalEdge(int[][] adjacencyMatrix, int min) {
        int nodes = number_of_nodes;
        // initialize criticalMatrix
        int[][] criticalMatrix = new int[nodes][nodes];
        for (int i = 0; i < criticalMatrix.length; i++) {
            for (int j = 0; j < criticalMatrix.length; j++) {
                criticalMatrix[i][j] = 0;
            }
        }
        // determine the new cut of the network
        int new_cut = 0;
        int critical_Edge = 0;
        int[][] temp = new int[nodes][nodes];
        for (int i = 0; i < nodes; i++) {
            for (int j = i; j < nodes; j++) {
                for (int m = 0; m < nodes; m++) {
                    for (int k = 0; k < nodes; k++) {
                        temp[m][k] = adjacencyMatrix[m][k];
                    }
                }
```

```java
                    if (temp[i][j] != 0) {
                        temp[i][j] = temp[j][i] = (temp[i][j] - 1);
                        // compute critical edges for the adjacency matrix /
                        temp
                        new_cut = ComputeCriticalEdges(temp);
                        // incrment critical Edge count & set criticalMatrix
                        values
                        if (new_cut < min) {
                            critical_Edge++;
                            criticalMatrix[i][j] = 1;
                        }
                        temp[i][j] = temp[j][i] = (temp[i][j] + 1);
                    }
                }
            }
            // uncomment the below line to draw the graph for critical edges of the
            // network
            // g.jung(criticalMatrix, nodes);

            return critical_Edge;
        }

        private int ComputeCriticalEdges(int[][] adjacencyMatrix) {
            int nodes = number_of_nodes;
            int edges = number_of_edges;
            int lamda_XY = 0;
            int lamda_G = 500;
            int Min_cut = 0;
            // loop until there are 2 nodes left
            while (nodes >= 2) {

                int maxEdges = 0;
                int nextNode = 0;
                int count = 0;
                int MAcount = 0;
                int[] MA = new int[nodes];
                // randomly choose first vertex
                int firstNode = rand.nextInt(nodes);
                // add the random first vertex to MA array
                MA[MAcount++] = firstNode;

                int m = 0, n = 0;
                // create a local copy of the adjacency matrix
                int[][] adjMatrix = new int[nodes][nodes];
                for (int i = 0; i < nodes; i++) {
                    for (int j = 0; j < nodes; j++) {
                        adjMatrix[i][j] = adjacencyMatrix[i][j];
                    }
                }

                while (count < (nodes - 1)) {
                    // compute the vertex with maximum degree
                    maxEdges = 0;
                    for (int j = 0; j < nodes; j++) {
                        if (adjMatrix[firstNode][j] >= maxEdges) {
```

```
                    maxEdges = adjMatrix[firstNode][j];
                    nextNode = j;
                }
            }
            // add the maximum degree vertex to the MA array
            MA[MAcount++] = nextNode;
            // if there are only 2 nodes left, compute the lamdaXY
from Vn-1
            // & Vn
            if (count == (nodes - 2)) {
                lamda_XY = adjMatrix[firstNode][nextNode];
            }
            // merge the vertices of randomly selected first vertex
            for (int j = 0; j < nodes; j++) {
                adjMatrix[firstNode][j] += adjMatrix[nextNode][j];
                adjMatrix[j][firstNode] += adjMatrix[nextNode][j];
                adjMatrix[firstNode][firstNode] = 0;
                adjMatrix[nextNode][j] = adjMatrix[j][nextNode] = 0;
            }
            count++;
        }

        int lastnode, lastsecondnode;
        lastnode = MA[MAcount - 1]; // Vn
        lastsecondnode = MA[MAcount - 2]; // Vn-1
        // merge the edges for lastNode and secondLastNode vertices
        for (int j = 0; j < nodes; j++) {
            adjacencyMatrix[lastsecondnode][j] +=
adjacencyMatrix[lastnode][j];
            adjacencyMatrix[j][lastsecondnode] +=
adjacencyMatrix[j][lastnode];
            adjacencyMatrix[lastsecondnode][lastsecondnode] = 0;
            adjacencyMatrix[lastnode][j] =
adjacencyMatrix[j][lastnode] = 0;
        }
        // compare the lamdaG with lamdaXY
        if (lamda_G >= lamda_XY) {
            lamda_G = lamda_XY;
        }
        nodes--;
        // recompute the adjacency matrix
        int[][] temp = new int[nodes][nodes];
        for (int i = 0; i < adjacencyMatrix.length; i++) {
            if (i != lastnode) {
                n = 0;
                for (int j = 0; j < adjacencyMatrix.length; j++) {
                    if (j != lastnode) {
                        temp[m][n] = adjacencyMatrix[i][j];
                        n++;
                    }
                }
                m++;
            } else
                continue;
        }
```

```
                adjacencyMatrix = temp;
                Min_cut = lamda_G;
            }
            return Min_cut;
        }

        public static void main(String[] args) {
            int number_of_nodes = 20; // number of nodes in the network
            int number_of_edges = -1;
            NIAlgorithm niAlgo = new NIAlgorithm(number_of_nodes, number_of_edges);
            rand = new Random(); // Random number generation
            g = new DrawGraph(); // Draw graph for the given input
            // perform the experiment for number_of_edges values 40..400
            for (int i = 40; i <= 400; i += 5) {
                niAlgo.number_of_edges = i;

                adjacencyMatrix = new int[number_of_nodes][number_of_nodes];
                degreeVector = new int[number_of_nodes];

                niAlgo.GenerateGraph();
                // uncomment for drawing graph, after generating the adjacency
                // matrix
                // g.jung(adjacencyMatrix, number_of_nodes);
                niAlgo.ComputeMinDegreeAndDensity();
                niAlgo.ComputeLamdaG();
                int min = niAlgo.ComputeCriticalEdges(adjacencyMatrix);
                int critEdge = niAlgo.CriticalEdge(adjacencyMatrix, min);
                niAlgo.critEdge.add(critEdge);
            }
            System.out.println("Lambda Values\t:" + niAlgo.lamdaG);
            System.out.println("Densities\t:" + niAlgo.densities);
            System.out.println("Lowest Degrees\t:" + niAlgo.lowestDegree);
            System.out.println(niAlgo.criticalEdges);
            System.out.println("Critical Edges\t:" + niAlgo.critEdge);
        }
}
```

//Code for drawing graph (**Referenced** http://jung.sourceforge.net/presentations/JUNG_M2K.pdf and http://www.cs.columbia.edu/~sh553/teaching/w3134-s07/homework/jungBasicExample.java)

```
import java.awt.Dimension;
import javax.swing.JFrame;
import edu.uci.ics.jung.algorithms.layout.CircleLayout;
import edu.uci.ics.jung.graph.Graph;
import edu.uci.ics.jung.graph.SparseMultigraph;
import edu.uci.ics.jung.visualization.VisualizationImageServer;

/*
 * BasicGraphCreation.java
 * Copyright March 1, 2007 Grotto Networking
 *
 */
```

```java
/**
 * The simplest JUNG2 graph creation example possible? Shows how easy the new
 * JUNG2 graph interface and implementation class is to use.
 *
 * @author Dr. Greg M. Bernstein
 */

public class DrawGraph {
	//Draws graph for the given adjacency matrix
	public void jung(int[][] adjacencyMatrix, int nodeCount) {

		Graph<Integer, String> g2 = new SparseMultigraph<Integer, String>();
		for (int i = 0; i < nodeCount; i++) {
			g2.addVertex(i);
		}
		for (int i = 0; i < nodeCount; i++) {
			for (int j = i; j < nodeCount; j++) {
				int count = 0;
				count = adjacencyMatrix[i][j];
				while (count > 0) {
					String name = "" + i + "" + j + "-" + count;
					g2.addEdge(name, i, j);
					count--;
				}

			}
		}

		VisualizationImageServer<Integer, String> vs = new
VisualizationImageServer<Integer, String>(
				new CircleLayout<Integer, String>(g2), new Dimension(600,
600));
		JFrame frame = new JFrame();
		frame.getContentPane().add(vs);
		frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
		frame.pack();
		frame.setVisible(true);
	}

}
```

8 Readme

Create java file with class name as 'NIAlgorithm'.

(1). Language used: Java

(2). Compilation Command or the IDE you are using.
IDE used- Eclipse
For compilation using command prompt refer below section.

(3). Command for running.

For compilation using command prompt refer below section.
With Eclipse IDE, Press CTRL+F11 to run the program.
Please, follow the sequence of instructions given.
******* INSTRUCTIONS *******

For compiling and executing using command prompt:
1. Open a command prompt window and browse up to this extracted folder.
Navigate to the 'src' folder.

2. Compilation of all the file/s
Command to be entered: javac <filename>.java
e.g. javac Program.java

3. Execution of compiled files
Command to be entered: java <filename>
e.g. java Program