CS 6385.001 - Algorithmic Aspects of Telecommunication Networks - F15
Project 2 Report

Peng Li - pxl141030@utdallas.edu

November 3, 2015

1 Introduction

This project is to design a software program that implements the Nagamochi-Ibaraki algorithm to find the minimum cut of an undirected graph. Based on that implementation, we will also experiment with it to find out how the graph connectivity ($\lambda(G)$) and number of critical edges ($C(G)$) change as the average degree ($d = 2m/n$) of the graph changes.

The input graph of this project is generated as follows:

- Number of nodes of graph n is fixed $n = 22$;
- Number of edges of graph m vary between 40 and 400, increasing in steps of 5;
- Once m is fixed for an experiment, the m edges are generated randomly among all possibilities; with parallel edges allowed, but self-loop prohibited.

Note that the randomly generated input graph may not necessarily be a connected graph, also the generated graph may not necessarily be a simple graph. Disconnected graph and multigraph are both allowed in this experiment.

During the implementation of this project task, the following assumptions have been made:

- For each m value (from 40 to 400), the experiment calculates the graph connectivity ($\lambda(G)$) and number of critical edges ($C(G)$) of 50 independently randomly generated graphs. The calculated connectivity and number of critical edges of the 50 graphs are averaged as the result for a given m.
- In case an input graph is disconnected, the algorithm output value zero for connectivity ($\lambda(G) = 0$) and value zero for number of critical edges ($C(G) = 0$).

We will be able to see that as the average degree (d) of a graph increases, the graph connectivity (λ) and number of critical edges (C) also increases.

2 Program Description

2.1 Algorithm

Before we describe the algorithm, some notations are worth mentioning:

$\lambda(G)$ - the edge connectivity of graph G;
$\lambda(x, y)$ - the connectivity between two different node x and y;

G_{xy} - the graph obtained from G by contracting (merging) nodes x, y. In this operation we omit the possibly arising loop (if x, y are connected in G), but keep the parallel edges;
$d(x)$ - the degree of node x.

Nagamochi-Ibaraki algorithm is a deterministic algorithm for calculating graph connectivity. It is based on the following theorems.

Theorem 1. *Let G_{xy} be the graph obtained from G by contracting nodes x and y. Then for any two nodes x and y, the following holds:*

$$\lambda(G) = min\,\{\lambda(x,y), \lambda(G_{xy})\}$$

Theorem 2. *In any Maximum Adjacency (MA) ordering $v_1, ..., v_n$ of nodes, the following holds:*

$$\lambda(v_{n-1}, v_n) = d(v_n)$$

Definition The MA ordering $v_1, ..., v_i$ of the nodes is generated recursively by the following algorithm:

- Take any of the nodes for v_1;
- Once $v_1, ..., v_i$ is already chosen, take a node for v_{i+1} that has the maximum number of edges connecting it with the set $\{v_1, ..., v_i\}$.

The algorithm uses the *Maximum Adjacency (MA) ordering* to choose nodes used for contraction (the last 2 nodes in the MA ordering), and then apply *Theorem 1* on the contracted graph. This process is carried out recursively until there are only 2 nodes left in the resulting contracted graph.

In case that the graph is disconnected, the number of nodes contained in the MA ordering will be less than the actual number of nodes in the graph. In that case, the algorithm will output *zero* connectivity without further recursing.

Algorithm 1 Nagamochi-Ibaraki Algorithm

```
Nagamochi-Ibaraki(G(V, E))
if |V| = 2 then
    Compute the number of edges between the 2 nodes x, y as λ(x, y)
    return λ(x, y)
else
    Compute MA ordering of G
    if Number of nodes in MA ordering < |V| then
        return 0
    end if
    Select the last 2 nodes x, y in MA ordering of G
    Compute λ(x, y)
    Contract nodes x, y, resulting in a new graph G_xy
    Compute λ(G_xy) = Nagamochi-Ibaraki(G_xy)
    return λ(G) = min {λ(x, y), λ(G_xy)}
end if
```

2.2 Implementation

The project is implemented using Java JDK 1.7.

The implementation of the Nagamochi-Ibaraki algorithm is formulated in NagamochiIbaraki class. This class includes static methods for creating maximum adjacency ordering, graph contraction and recursive method for Nagamochi-Ibaraki minimum cut algorithm; it also contains some utility methods for generating input and graph visualization. For source code, please refer to Appendix A.1

The Project2 class is the driving class that makes calls to functions in NagamochiIbaraki class. It also implements the calculation of average values for 50 independent experiments for each given m. For source code, please refer to Appendix A.3

The UndirectedEdge class is only a dummy class used to represent the graph edges in the software implementation. In this project, all edges of the graph are undirected and equal weighted edges. Therefore using this wrapping class is sufficient to represent them with minimum programming overhead. For source code, please refer to Appendix A.2

Note that the Project2 class and NagamochiIbaraki class requires dependencies on the following third party libraries:

- JUNG 2 Java Universal Network / Graph Framework, this library is used for graph representation and visualization.
  download at http://sourceforge.net/projects/jung/files/
- Opencsv, this library is used to write the output of the experiment into a .csv file for easier manipulation.
  download at http://sourceforge.net/projects/opencsv/

2.3 How To Run

Before running the program, make sure to have the third party Java libraries as described in section 2.2 included in the class path.

- Create package named "project2" and copy all source code files into that package;
- Create a folder named "data" in your project root folder;
- In terminal, run "javac Project2.java" and then "java Project2";

The program should print out the experiment result to terminal as well as output it to ".data/result.csv" file.

3 Experiment Result

The experiment result is shown in Table 1. A graphical representation of how the average edge connectivity ($\lambda(G)$) and average number of critical edges ($C(G)$) depends on average degree (d) of the graph is shown in Figure 1.

Table 1: Experiment Result

# of nodes	average degree	average connectivity	average # of critical edges
40	3.64	0.66	1.94
45	4.09	0.78	1.88
50	4.55	1.02	2.18
55	5.00	1.28	2.74
60	5.45	1.60	2.96
65	5.91	1.86	3.30

Continued on next page

Table 1 – *Continued from previous page*

# of nodes	average degree	average connectivity	average # of critical edges
70	6.36	2.34	4.24
75	6.82	2.58	4.50
80	7.27	2.84	5.58
85	7.73	3.04	5.26
90	8.18	3.34	5.58
95	8.64	3.56	5.64
100	9.09	4.00	6.68
105	9.55	4.40	6.88
110	10.00	4.54	6.70
115	10.45	5.10	7.82
120	10.91	4.96	7.06
125	11.36	5.42	7.60
130	11.82	5.96	8.54
135	12.27	6.44	9.76
140	12.73	6.84	10.04
145	13.18	7.24	10.40
150	13.64	7.26	9.62
155	14.09	7.58	12.00
160	14.55	8.10	11.18
165	15.00	8.56	12.08
170	15.45	8.92	12.78
175	15.91	8.92	13.44
180	16.36	8.90	12.42
185	16.82	9.18	10.98
190	17.27	9.84	12.60
195	17.73	10.28	11.96
200	18.18	10.64	13.08
205	18.64	10.90	14.74
210	19.09	11.86	18.88
215	19.55	11.76	13.32
220	20.00	12.04	14.24
225	20.45	12.48	15.08
230	20.91	12.70	16.18
235	21.36	13.02	16.30
240	21.82	13.42	15.44
245	22.27	14.36	19.48
250	22.73	14.26	17.26
255	23.18	14.48	18.80
260	23.64	14.76	19.26
265	24.09	15.28	19.36
270	24.55	15.94	18.98
275	25.00	16.22	18.56
280	25.45	16.16	19.82
285	25.91	16.92	21.42
290	26.36	17.54	23.52
295	26.82	17.38	22.10
300	27.27	18.16	21.16
305	27.73	18.50	21.86
310	28.18	18.08	20.22
315	28.64	19.16	21.94
320	29.09	19.40	23.58

Continued on next page

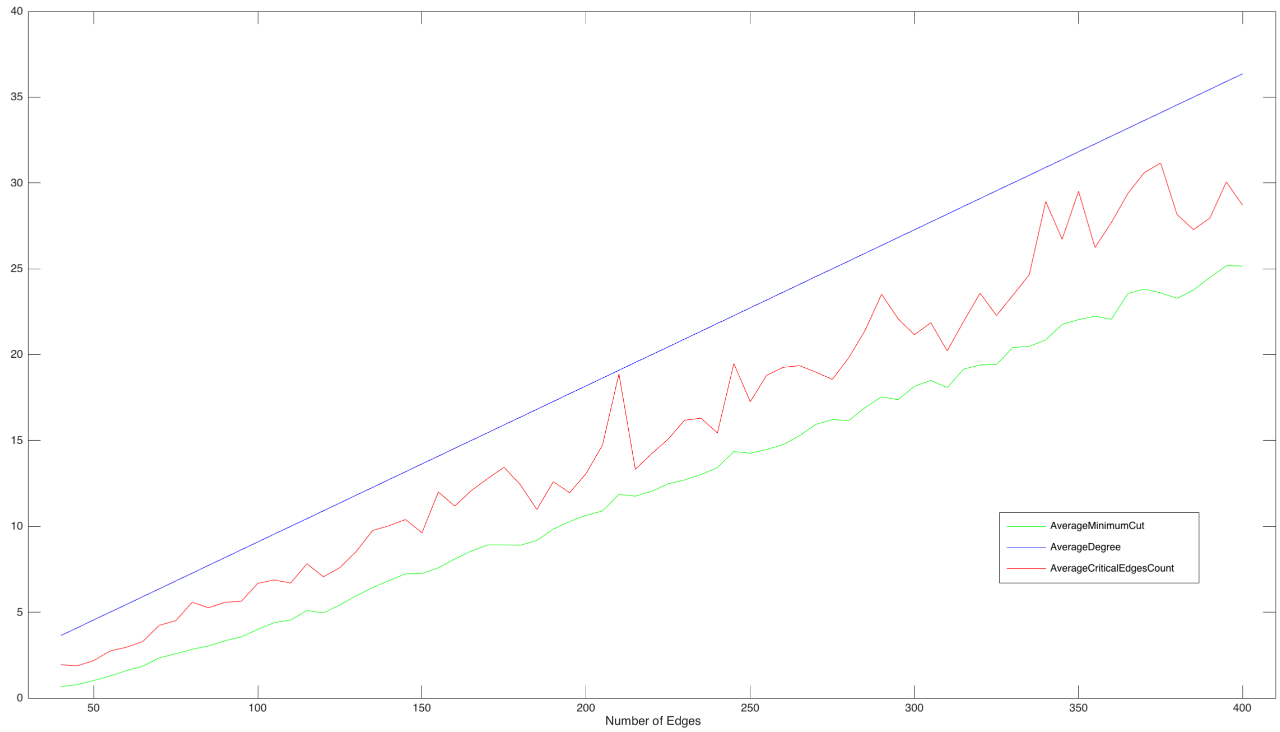

Figure 1: Relationship between $\lambda(G)$, $C(G)$ and d

Table 1 – *Continued from previous page*

# of nodes	average degree	average connectivity	average # of critical edges
325	29.55	19.42	22.28
330	30.00	20.42	23.46
335	30.45	20.48	24.66
340	30.91	20.86	28.92
345	31.36	21.76	26.72
350	31.82	22.04	29.52
355	32.27	22.24	26.24
360	32.73	22.06	27.70
365	33.18	23.56	29.38
370	33.64	23.82	30.60
375	34.09	23.60	31.16
380	34.55	23.28	28.16
385	35.00	23.76	27.28
390	35.45	24.50	27.96
395	35.91	25.18	30.06
400	36.36	25.16	28.70

Table 1 – *End of table*

As we can see in the result and Figure 1. The average edge connectivity and number of critical edges increase as the average degree of the graph increases. Average edge connectivity $\lambda(G)$ has a linear relationship with average degree d. This is because that the edges of the graph are generated randomly, so that increasing average degree of the graph yields in increase of expected degree of **each** node. If the process for generating edges follows a specific rule instead of randomization, this linear relationship might not hold.

Average of number of critical edges is also linear with average degree of the graph. However, if we look further, there are 2 interesting observations:

- This linear relationship is more noisy than the one between average connectivity and average degree of graph. This is because that the number of critical edges depends not only on the average connectivity of the graph, but also very much depends on how the graph is connected.
- The number of critical edges falls between average connectivity and average degree of the graph. This is because that each graph may have more than one minimum cut configuration with the same minimum cut value. These minimum cuts normally share some edges, all edges in these minimum cuts are critical edges. Therefore, the number of critical edges are larger than the edge connectivity of the graph.

A Source Code

A.1 NagamochiIbaraki.java

```java
package project2;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Paint;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

import javax.swing.JFrame;

import org.apache.commons.collections15.Transformer;

import edu.uci.ics.jung.algorithms.layout.CircleLayout;
import edu.uci.ics.jung.algorithms.layout.Layout;
import edu.uci.ics.jung.graph.UndirectedOrderedSparseMultigraph;
import edu.uci.ics.jung.graph.UndirectedSparseMultigraph;
import edu.uci.ics.jung.visualization.BasicVisualizationServer;
import edu.uci.ics.jung.visualization.decorators.ToStringLabeller;
import edu.uci.ics.jung.visualization.renderers.Renderer.VertexLabel.Position;

public class NagamochiIbaraki {

  private static final int MAX_VERTICES_COUNT = 22;

  /**
   * find minimum cut of a graph using Nagamochi-Ibaraki algorithm
   *
   * @param graph
   *            input graph
   * @return minimum cut
   */
  public static int minCutNIAlgorithm(
      UndirectedSparseMultigraph<Integer, UndirectedEdge> graph) {
    // input check
    if (graph == null || graph.getVertexCount() == 1) {
      System.out
          .println("Error: Graph does not contrain enough vertices.");
      return 0;
    }

    /*
     * base case: if there are only 2 vertices in the graph, return the
     * number of edges between the 2 vertices
     */
    if (graph.getVertexCount() == 2) {
      Iterator<Integer> itVertex = graph.getVertices().iterator();
      Integer vertex = itVertex.next();
      return graph.getIncidentEdges(vertex).size();
    } else {
      // find MA ordering
      ArrayList<Integer> maOrdering = findMAOrdering(graph);
      /* if graph is disconnected, return minimum cut - zero */
      if (maOrdering == null) {
        return 0;
      }

      /*
       * otherwise, find lambda(from, to) and compare it with the minimum
       * cut of the contracted graph
       */
      Integer from = maOrdering.get(maOrdering.size() - 1);
```

```
      Integer to = maOrdering.get(maOrdering.size() - 2);
      int lambdaFromTo = graph.getIncidentEdges(from).size();

      contractGraph(graph, from, to);
      // recursive call minCutNIAlgorithm()
      return Math.min(lambdaFromTo, minCutNIAlgorithm(graph));
    }
  }

  /**
   * create a Maximum Adjacency ordering of a graph, starting point is chosen
   * at random.
   *
   * @param graph
   *            input undirected graph
   * @return ordered vertices if such ordering exists, or null if graph is
   *         empty or disconnected
   */
  public static ArrayList<Integer> findMAOrdering(
      UndirectedSparseMultigraph<Integer, UndirectedEdge> graph) {

    int nVertices = graph.getVertexCount();
    ArrayList<Integer> vertices = new ArrayList<Integer>();

    // check input if input graph has no vertices
    vertices.addAll(graph.getVertices());
    if (vertices.size() == 0) {
      return null;
    }

    ArrayList<Integer> result = new ArrayList<Integer>();
    /*
     * connectingEdgeCount tracks the number of connecting edges with
     * existing MA ordered vertex set
     */
    int[] connectingEdgeCount = new int[MAX_VERTICES_COUNT];

    // start with the first nodes in the vertices list
    Integer nextVertex = vertices.get(0);

    while (result.size() < nVertices && nextVertex != null) {
      result.add(nextVertex);

      Set<UndirectedEdge> incidentEdges = new HashSet<UndirectedEdge>();
      incidentEdges.addAll(graph.getIncidentEdges(nextVertex));

      for (UndirectedEdge e : incidentEdges) {
        Integer firstEndpoint = graph.getEndpoints(e).getFirst();
        Integer secondEndpoint = graph.getEndpoints(e).getSecond();

        // update connectingEdgeCount Array
        if (result.contains(firstEndpoint)
            && result.contains(secondEndpoint)) {
          connectingEdgeCount[nextVertex]--;
        } else {
          Integer endPoint = (!result.contains(firstEndpoint)) ? firstEndpoint
              : secondEndpoint;
          connectingEdgeCount[endPoint]++;
        }
      }

      int maxCount = 0;
      Integer maxVertex = null;
      for (int i = 0; i < connectingEdgeCount.length; i++) {
        if (connectingEdgeCount[i] > maxCount) {
          maxCount = connectingEdgeCount[i];
          maxVertex = i;
        }
```

```
      }
      nextVertex = maxVertex;
    }
    if (result.size() < nVertices) {
      return null;
    } else {
      return result;
    }
  }

  /**
   * graph contraction given 2 vertices, edges are reserved expect self loops;
   * output error message if one or more vertices does not exist in the given
   * graph
   *
   * @param graph
   *            contraction to be performed on
   * @param from
   *            vertex to be contracted
   * @param to
   *            vertex to be merged to
   */
  public static void contractGraph(
      UndirectedSparseMultigraph<Integer, UndirectedEdge> graph,
      Integer from, Integer to) {
    if (!graph.containsVertex(from) || !graph.containsVertex(to)) {
      System.out
          .println("Error: vertices to be contracted does not exist in graph.");
    } else {
      Set<UndirectedEdge> edges = new HashSet<UndirectedEdge>();
      edges.addAll(graph.getIncidentEdges(from));

      for (UndirectedEdge e : edges) {
        Integer firstEndpoint = graph.getEndpoints(e).getFirst();
        Integer secondEndpoint = graph.getEndpoints(e).getSecond();
        Integer endPoint = (firstEndpoint == from) ? secondEndpoint
            : firstEndpoint;

        if (endPoint != to) {
          graph.addEdge(new UndirectedEdge(), endPoint, to);
        }
      }
      graph.removeVertex(from);
    }
  }

  /**
   * find number of critical edges of a given graph, remove edges one by one
   * and calculate minimum cut of the remaining graph. Parallel edges are
   * tested only once.
   *
   * @param edgeMatrix
   *            edge matrix of the input graph
   * @param minCut
   *            minimum cut of the input graph
   * @return number of critical edges
   */
  public static int findCriticalEdges(int[][] edgeMatrix, int minCut) {

    int result = 0;
    for (int i = 0; i < edgeMatrix.length; i++) {
      for (int j = 0; j < edgeMatrix[i].length; j++) {
        if (edgeMatrix[i][j] > 0) {
          edgeMatrix[i][j]--;
          int minCutTest = minCutNIAlgorithm(createGraph(edgeMatrix));

          if (minCutTest < minCut) {
            result = result + edgeMatrix[i][j] + 1;
```

```java
        }
        edgeMatrix[i][j]++;
      }
    }
  }
  return result;
}

/*
 * Utility method - create a UndirectedSparseMultigraph given n and m
 */
public static UndirectedSparseMultigraph<Integer, UndirectedEdge> createGraph(
    int[][] edgeMatrix) {

  UndirectedOrderedSparseMultigraph<Integer, UndirectedEdge> result = new
      UndirectedOrderedSparseMultigraph<Integer, UndirectedEdge>();

  // add nodes
  for (int i = 0; i < edgeMatrix.length; i++) {
    result.addVertex(i);
  }

  // add edges
  for (int i = 0; i < edgeMatrix.length; i++) {
    for (int j = 0; j < edgeMatrix[i].length; j++) {
      for (int k = 0; k < edgeMatrix[i][j]; k++) {
        result.addEdge(new UndirectedEdge(), i, j);
      }
    }
  }
  return result;
}

/*
 * Utility method - given number of nodes and number of edges m, generate
 * randomly the edges of the graph
 */
public static int[][] genEdgeMatrix(int n, int m) {
  int[][] result = new int[n][n];

  // initialize edge matrix with zero edges
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      result[i][j] = 0;
    }
  }
  int count = 0;
  while (count < m) {
    int startNode = randomGen(n - 1);
    int endNode = randomGen(n - 1);
    // no self-loops
    if (startNode != endNode) {
      result[startNode][endNode] += 1;
      count++;
    }
  }
  return result;
}

public static int findEdgeCount(int[][] m) {
  int result = 0;
  for (int i = 0; i < m.length; i++) {
    for (int j = 0; j < m[i].length; j++) {
      result += m[i][j];
    }
  }
  return result;
}
```

```java

  /*
   * Utility method - generate a random integer from [0, 1, ..., range]
   */
  private static int randomGen(int range) {
    return (int) ((range + 1) * Math.random());
  }

  /*
   * Utility method - print matrix
   */
  public static void printMatrix(int m[][]) {
    for (int i = 0; i < m.length; i++) {
      for (int j = 0; j < m[i].length; j++) {
        System.out.format("%3d" + " ", m[i][j]);
      }
      System.out.println();
    }
  }

  /*
   * Utility method - for graph visualization
   */
  public static void visualizeGraph(
      UndirectedSparseMultigraph<Integer, UndirectedEdge> graph,
      String text) {

    Layout<Integer, UndirectedEdge> layout = new CircleLayout<Integer, UndirectedEdge>(
        graph);
    layout.setSize(new Dimension(800, 800));

    BasicVisualizationServer<Integer, UndirectedEdge> vv = new BasicVisualizationServer<Integer,
        UndirectedEdge>(
        layout);
    Transformer<Integer, Paint> vertexPaint = new Transformer<Integer, Paint>() {
      public Paint transform(Integer i) {
        return Color.YELLOW;
      }
    };

    vv.setPreferredSize(new Dimension(800, 800));
    vv.getRenderContext().setVertexFillPaintTransformer(vertexPaint);
    vv.getRenderContext().setVertexLabelTransformer(
        new ToStringLabeller<Integer>());
    vv.getRenderer().getVertexLabelRenderer().setPosition(Position.CNTR);

    JFrame frame = new JFrame(text);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(vv);
    frame.pack();
    frame.setVisible(true);
  }

}
```

A.2 Project2.java

```
package project2;

import java.io.FileWriter;
import java.io.IOException;

import com.opencsv.CSVWriter;

import edu.uci.ics.jung.graph.UndirectedSparseMultigraph;

public class Project2 {

  public static final int NUMBER_OF_VERTICES = 22;
  public static final int NUMBER_OF_EXPERIMENT = 50;
  public static final int LOWER_BOUND_EDGES = 40;
  public static final int UPPER_BOUND_EDGES = 400;
  public static final int INCREMENT_EDGES = 5;

  public static void main(String[] args) throws IOException {

    // output to csv file
    String csv = "data/result.csv";
    CSVWriter writer = new CSVWriter(new FileWriter(csv));
    String[] header = "Number of Vertices, Number of Edges, Average Degree, Average
        Connectivity, Average Critical Edges Count"
        .split(",");
    writer.writeNext(header);

    for (int edgeCount = LOWER_BOUND_EDGES; edgeCount <= UPPER_BOUND_EDGES; edgeCount +=
        INCREMENT_EDGES) {
      // repeat each experiment and calculate average
      int minCutTotal = 0;
      float minCutAvg = 0;
      int criticalEdgeTotal = 0;
      float criticalEdgeAvg = 0;

      for (int experimentCount = 0; experimentCount < NUMBER_OF_EXPERIMENT; experimentCount++) {
        int[][] edgeMatrix = NagamochiIbaraki.genEdgeMatrix(
            NUMBER_OF_VERTICES, edgeCount);
        UndirectedSparseMultigraph<Integer, UndirectedEdge> graph = NagamochiIbaraki
            .createGraph(edgeMatrix);
        int minCut = NagamochiIbaraki.minCutNIAlgorithm(graph);
        minCutTotal += minCut;

        int criticalEdge = 0;

        if (minCut != 0) {
          criticalEdge = NagamochiIbaraki.findCriticalEdges(
              edgeMatrix, minCut);
          criticalEdgeTotal += criticalEdge;
        }
      }

      minCutAvg = (float) minCutTotal / NUMBER_OF_EXPERIMENT;
      criticalEdgeAvg = (float) criticalEdgeTotal / NUMBER_OF_EXPERIMENT;

      // output to csv file
      String record = String.valueOf(NUMBER_OF_VERTICES)
          + ","
          + String.valueOf(edgeCount)
          + ","
          + String.valueOf((float) 2 * edgeCount / NUMBER_OF_VERTICES)
          + "," + String.valueOf(minCutAvg) + ","
          + String.valueOf(criticalEdgeAvg);
      String[] recordArray = record.split(",");
      writer.writeNext(recordArray);

```

```java
      // output to console
      System.out.println("Number of Vertices: " + NUMBER_OF_VERTICES
          + "; Number of Edges: " + edgeCount + "; Average degree: "
          + (float) 2 * edgeCount / NUMBER_OF_VERTICES
          + "; Average connectivity: " + minCutAvg
          + "; Average number of critical edges: " + criticalEdgeAvg);
    }
    writer.close();
  }
}
```

A.3 UndirectedEdge.java

```java
package project2;

/*
 * this is a dummy class used only to represent edges
 */
public class UndirectedEdge {

}
```