2. **Explain how your implementation of the algorithm works. Provide pseudo code for the description, with sufficient comments to make it readable and understandable by a human.**

The basic structure of the algorithm as shown in the project description has been implemented into a Java program as shown below.

```
lamb(double[][] g) {
    if (g.length == 2) {
        return g[0][1];
    } else {
        ArrayList<Integer> mao = maximumAdjacencyOrder(g);
        Integer x = mao.get(mao.size() - 2);
        Integer y = mao.get(mao.size() - 1);
        double degree = degree(g, y);
        double[][] gXYmerged = merge(g, x, y);
        double lambdaOfgXY = lamb(gXYmerged);
        return Math.min(degree, lambdaOfgXY);
    }
}
```

PSEUDOCODE

- The **Base Case** is checked first as g.length == 2 meaning that if there are only 2 nodes (let's say node x and node y) in the graph then there is no need to recursively merge the 2 nodes because all edges between the 2 nodes would result in self-loops and thus deleted within the merge(g, x, y) algorithm. Therefore, we should return the number of edges (including parallel edges) between nodes the 2 nodes and return that as the edge connectivity of the graph of node-size == 2
- **Not Base Case:**
    - Compute the maximum adjacency order of the graph of node-size > 2

        - This maximum adjacency order is an implementation of the lecture note's algorithm, where we always choose node indexed at 0 as the first node of the Maximum Adjacency Order.
        - Then we keep on choosing vertices to add into this order based on the highest edge connectivity between the current set of vertices within the maximum adjacency order and nodes not yet added into that set
    - Then we choose the last 2 nodes that maximum adjacency order and do 2 things:
        - Compute the degree of the last node, the degree is the edge-connectivity of the whole graph
        - Generate new graph by merging those 2 nodes. Then compute recursively the lambda(new graph) to return the edge-connectivity of that new graph
    - Return the minimum of the 2 values as the edge-connectivity of the graph of node-size > 2

3. **Write a computer program that implements the algorithm. You may use any programming language under any operating system, this is entirely of your choice.**

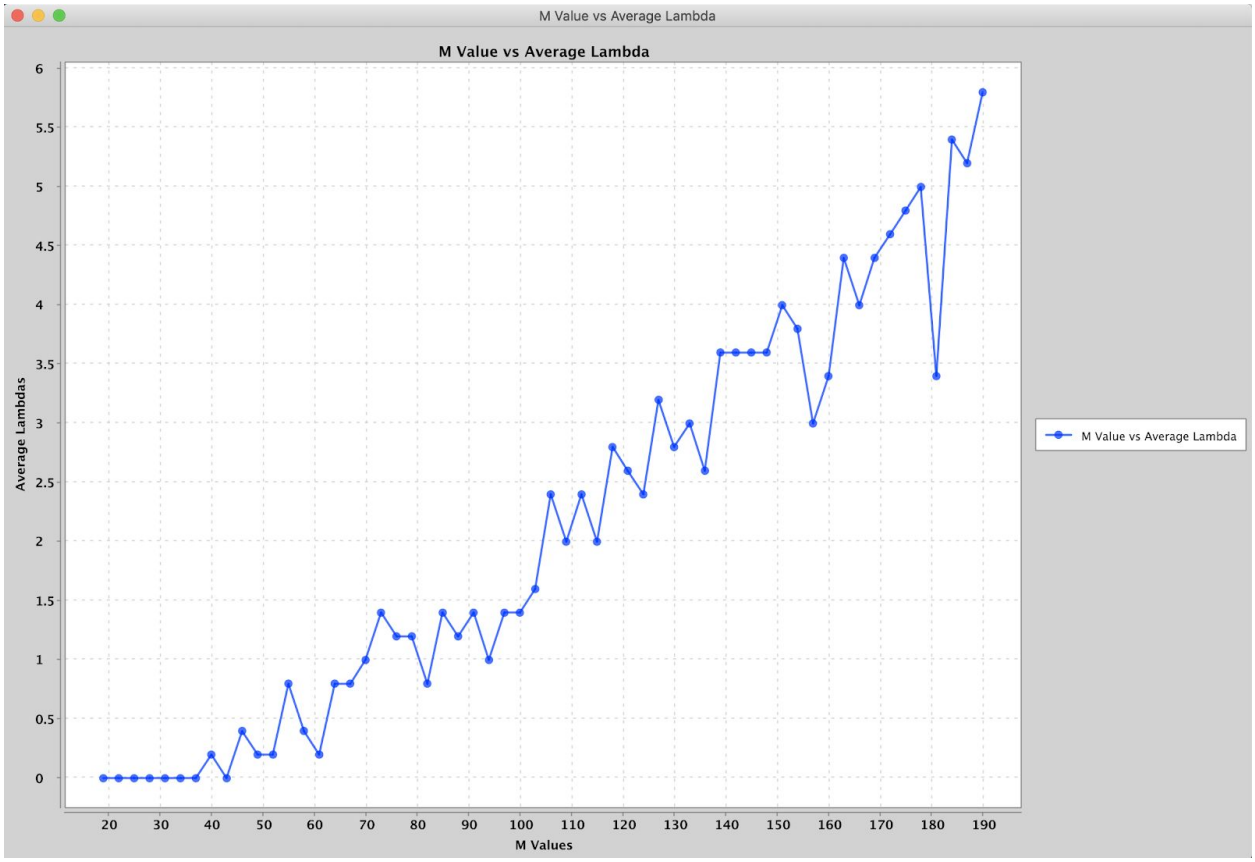
   Done, implemented in java version 1.8 shown in the software package

4. **Run the program on randomly generated examples. Let the number of nodes be fixed at n = 20, while the number m of edges will vary between 19 and 190, in steps of 3. For any such value of m, the program creates 5 graphs with n = 20 nodes and m edges. The actual edges are selected randomly. Parallel edges and self-loops are not allowed in the graph generation. Note, however, that the Nagamochi-Ibaraki algorithm allows parallel edges in its internal working, as they may arise due to the merging of nodes.**

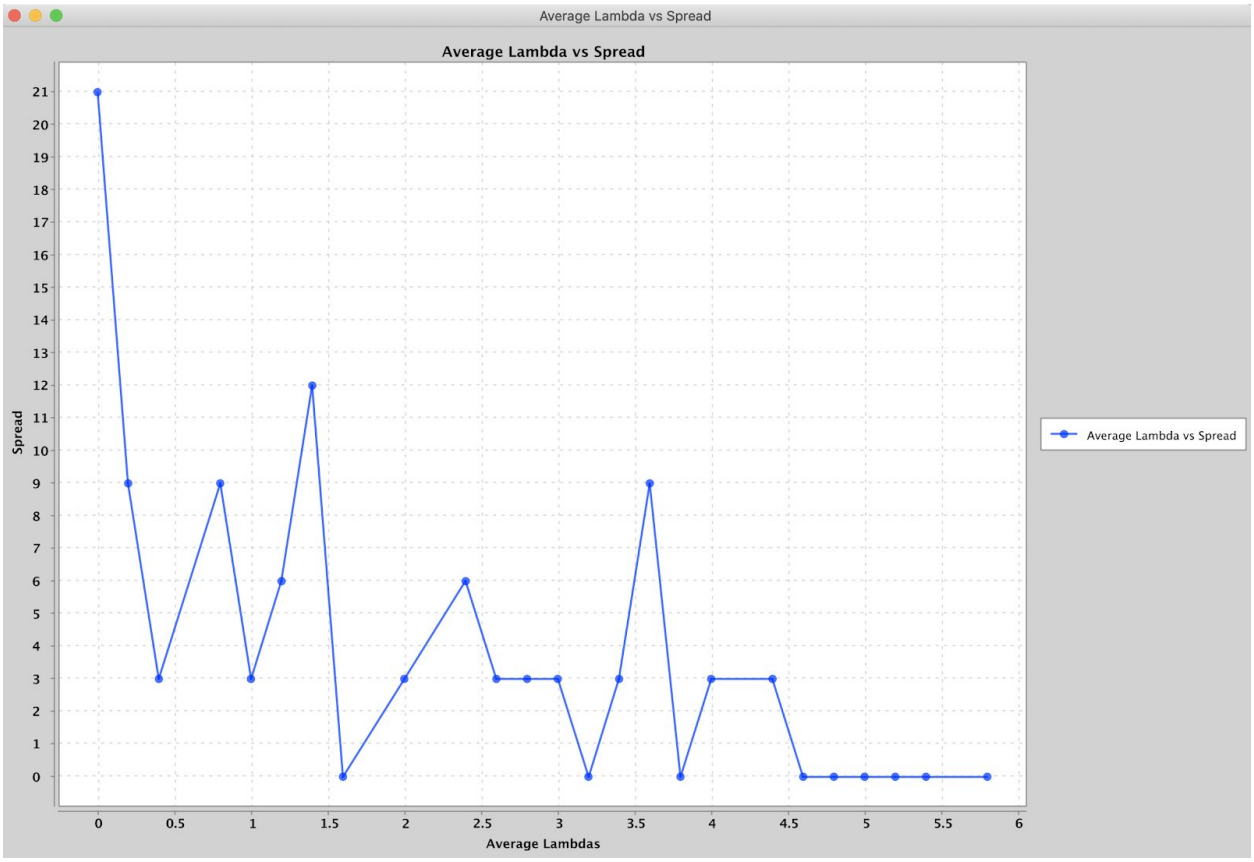
   Done, implemented in java version 1.8 shown in the software package

5. **Experiment with your random graph examples to find an experimental connection between the number of edges in the graph and its con- nectivity λ(G). (If the graph happens to be disconnected, then take λ(G) = 0.) Show the found relationship graphically in a diagram, ex- hibiting λ(G) as a function of m, while keeping n = 20 fixed. Since the edges are selected randomly, therefore, we reduce the effect of ran- domness in the following**

way: run 5 independent experiments for every value of m, one with each generated example for this m, and average out the results. (This is why 5 graphs were created for every m.)



6. **For every connectivity value λ = λ(G) that occurred in the experiments, record the largest and smallest number of edges with which this λ value occurred. Let us call their difference the spread of λ, and let us denote it by s(λ). Show in a diagram how s(λ) depends on λ, based on your experiments.**



7. **Give a short explanation why the functions found in items 4 and 5 look the way they look. In other words, try to argue that they indeed show the behavior that can be intuitively expected, so your program likely works correctly. Note that it is part of your task to find out what behavior can be expected.**

   M Value vs Lambda Value

   - M value is the number of edges in the graph, as the number of edges in the graph increases the graph indicates the proportional increase of the lambda value. This is as expected, because due to the definition of lambda (aka edge connectivity of the graph) which is the minimum number of edges needed to be deleted for graph to be disconnected. So the larger the value of m the more edges would be present in the graph and thus the more edges needed to be removed for the graph to be considered disconnected

   Lambda Value vs Spread (of M values)

- As the lambda value increases the spread decreases. This is an expected outcome of the ontological case of this domain.
- This is best explained with lambdas resulting in value 0 and by looking at the corresponding M values of the previous graph. As seen in the first graph the number of lambdas = 0 in the first graph is plenty and is near the left side of the graph. This is because as explained in the previous point, the smaller the value M (number of edges in the graph) the smaller the Lambda value (edge-connectivity) thus the range form M = 21 to 40 something all but 1 equals = 0. This is the case for other runs of the application that most M values on the left side of the graph results in 0 for lambda value. This is because adding edges from that range doesn't usually result in all nodes being connected. Thus most of that range is lambda = 0. However, as the number of edges increases, the lambda value proportionally increases. The spread on the right side of the graph number 2 is mostly spread of 0 this is because we have calculated the average of 5 runs of lambda per M values and this results in non-integer values for average lambda which greatly increases the probability that average lambda is some unique number among the list of average lambdas thus resulting in a spread of 0. The case in between shows the decreases of spread as average lambda increases until reaching a plateau at 0 as shown on the right side of the graph

8. **Also include a section in the project document that is often referred to in a software package as "ReadMe file." The ReadMe file (or section) provides instructions on how to run the program. Even though in the default case, we do not plan to actually run the program, we should be able to do it, if needed. For example, if something does not seem right, and we want to double check whether the program indeed works correctly.**

   README.md Included within the software package

APPENDIX

Pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.marcuschiu</groupId>
  <artifactId>telecommunications-hw-two</artifactId>
  <version>1.0-SNAPSHOT</version>

  <dependencies>
      <dependency>
          <groupId>org.knowm.xchart</groupId>
          <artifactId>xchart</artifactId>
          <version>3.5.4</version>
      </dependency>
      <dependency>
          <groupId>org.graphstream</groupId>
          <artifactId>gs-core</artifactId>
          <version>1.1.1</version>
      </dependency>
  </dependencies>

  <build>
      <plugins>
          <plugin>
              <groupId>org.apache.maven.plugins</groupId>
              <artifactId>maven-compiler-plugin</artifactId>
              <configuration>
                  <source>1.8</source>
                  <target>1.8</target>
              </configuration>
          </plugin>
      </plugins>
  </build>
</project>
```

Main.java

```
import org.knowm.xchart.XChartPanel;
import org.knowm.xchart.XYChart;
import org.knowm.xchart.XYChartBuilder;

import javax.swing.*;
import java.awt.*;
import java.util.*;

public class Main {

  private static double[][] copy2Darray(double[][] old) {
      double[][] current = new double[old.length][old.length];
      for (int i = 0; i < old.length; i++)
          System.arraycopy(old[i], 0, current[i], 0, old[i].length);
      return current;
```

```
    }

    private static double[][] removeRow(double[][] g, int rowRemove) {
        int row = g.length;
        int col = g[0].length;

        double[][] gCopy = new double[row - 1][col];

        for (int i = 0; i < rowRemove; i++) {
            System.arraycopy(g[i], 0, gCopy[i], 0, col);
        }
        for (int i = rowRemove + 1; i < g.length; i++) {
            System.arraycopy(g[i], 0, gCopy[i - 1], 0, col);
        }

        return gCopy;
    }

    private static double[][] removeCol(double[][] array, int colRemove) {
        int row = array.length;
        int col = array[0].length;

        double[][] newArray = new double[row][col - 1];

        for (int i = 0; i < row; i++) {
            for (int j = 0, currColumn = 0; j < col; j++) {
                if (j != colRemove) {
                    newArray[i][currColumn++] = array[i][j];
                }
            }
        }

        return newArray;
    }

    private static double[][] merge(double[][] g, int x, int y) {
        double[][] gCopy = copy2Darray(g);

        for (int i = 0; i < g.length; i++) {
            gCopy[i][x] = gCopy[i][x] + gCopy[i][y];
        }
        gCopy = removeCol(gCopy, y);

        for (int i = 0; i < gCopy[0].length; i++) {
            gCopy[x][i] = gCopy[x][i] + gCopy[y][i];
        }
        gCopy = removeRow(gCopy, y);

        for (int i = 0; i < gCopy.length; i++) {
            gCopy[i][i] = 0;
```

```
        }

        return gCopy;
    }

    private static double numEdgesFromNextVertex2ChosenVertices(double[][] g, ArrayList<Integer> chosenVertices,
int nextVertex) {
        double pathSum = 0;
        for (Integer integer : chosenVertices) {
            pathSum += g[nextVertex][integer];
        }
        return pathSum;
    }

    private static ArrayList<Integer> possible(double[][] g, ArrayList<Integer> chosenVertices) {
        ArrayList<Integer> possibleVertices = new ArrayList<>();
        for (int i = 0; i < g.length; i++) {
            if (!chosenVertices.contains(i)) {
                possibleVertices.add(i);
            }
        }
        return possibleVertices;
    }

    private static ArrayList<Integer> maximumAdjacencyOrder(double[][] g) {
        ArrayList<Integer> chosenVertices = new ArrayList<>();
        while (chosenVertices.size() != g.length) {
            ArrayList<Integer> possibleVertices = possible(g, chosenVertices);
            int nextVertex = possibleVertices.get(0);
            double numEdgesFromNextVertex2ChosenVertices = numEdgesFromNextVertex2ChosenVertices(g,
chosenVertices, nextVertex);
            for (int possibleVertex : possibleVertices) {
                double val = numEdgesFromNextVertex2ChosenVertices(g, chosenVertices, possibleVertex);
                if (val > numEdgesFromNextVertex2ChosenVertices) {
                    nextVertex = possibleVertex;
                    numEdgesFromNextVertex2ChosenVertices = val;
                }
            }
            chosenVertices.add(nextVertex);
        }
        return chosenVertices;
    }

    private static int degree(double[][] g, int vertex) {
        int degree = 0;
        for (int i = 0; i < g.length; i++) {
            degree += g[vertex][i];
        }
        return degree;
    }
```

```
    private static double lamb(double[][] g) {
        if (g.length == 2) {
            return g[0][1];
        } else {
            ArrayList<Integer> mao = maximumAdjacencyOrder(g);
            Integer x = mao.get(mao.size() - 2);
            Integer y = mao.get(mao.size() - 1);
            double degree = degree(g, y);
            double[][] gXYmerged = merge(g, x, y);
            double lambdaOfgXY = lamb(gXYmerged);
            return Math.min(degree, lambdaOfgXY);
        }
    }

    private static double[][] generateRandomGraph(Integer numNodes, Integer numEdges) {
        double[][] g = new double[numNodes][numNodes];

        Random rand = new Random();
        for (int e = 0; e < numEdges; e++) {
            int i = rand.nextInt(numNodes);
            int j = rand.nextInt(numNodes);
            if (i == j || g[i][j] == 1) {
                e--;
            } else {
                g[i][j] = 1;
            }
        }

        return g;
    }

    private static double averageOfArrayList(ArrayList<Double> marks) {
        Double sum = 0d;
        if (!marks.isEmpty()) {
            for (Double mark : marks) {
                sum += mark;
            }
            return sum / marks.size();
        }
        return sum;
    }

    private static void showLineGraph(double[] k, double[] y, String title, String xTitle, String yTitle) {
        final XYChart chart = new XYChartBuilder().width(1200).height(800)
                .title(title)
                .xAxisTitle(xTitle)
                .yAxisTitle(yTitle)
                .build();
        chart.addSeries(title, k, y);
```

```
        javax.swing.SwingUtilities.invokeLater(() -> {
            JFrame frame = new JFrame(title);
            frame.setLayout(new BorderLayout());
            JPanel chartPanel = new XChartPanel<>(chart);
            frame.add(chartPanel, BorderLayout.CENTER);
            frame.pack();
            frame.setVisible(true);
        });
    }

    private static ArrayList<Double> getSortedSet(ArrayList<Double> averages) {
        Set<Double> averagesSet = new HashSet<>(averages);
        ArrayList<Double> averagesSetSorted = new ArrayList<>(averagesSet);
        Collections.sort(averagesSetSorted);
        return averagesSetSorted;
    }

    private static ArrayList<Double> something(ArrayList<Double> averages, ArrayList<Double> averagesSetSorted) {
        ArrayList<Double> spread = new ArrayList<>();

        Collections.sort(averages);
        for (Double average : averagesSetSorted) {
            double smallest = findSmall(averages, average);
            double largest = findLarge(averages, average);
            spread.add(largest - smallest);
        }

        return spread;
    }

    private static int findSmall(ArrayList<Double> sortedAverages, double average) {
        for (int i = 0; i < sortedAverages.size(); i++) {
            if (sortedAverages.get(i) == average) {
                return (i * 3) + 19;
            }
        }
        return 0;
    }

    private static int findLarge(ArrayList<Double> sortedAverages, double average) {
        for (int i = sortedAverages.size() - 1; i >= 0; i--) {
            if (sortedAverages.get(i) == average) {
                return (i * 3) + 19;
            }
        }
        return 0;
    }

    public static void main(String[] args) {
```

```
        // Calculate Lambda Values of Random Graphs of M Edges
        ArrayList<Double> averages = new ArrayList<>();
        ArrayList<Double> xValues = new ArrayList<>();
        for (int m = 19; m <= 190; m += 3) {
            xValues.add((double) m);
            ArrayList<Double> edgeConnectivities = new ArrayList<>();
            for (int i = 0; i < 5; i++) {
                edgeConnectivities.add(lamb(generateRandomGraph(20, m)));
            }
            averages.add(averageOfArrayList(edgeConnectivities));
        }

        // Display (M vs Lambda)
        double[] averagesArray = averages.stream().mapToDouble(i -> i).toArray();
        double[] xValuesArray = xValues.stream().mapToDouble(i -> i).toArray();
        showLineGraph(xValuesArray, averagesArray, "M Value vs Average Lambda", "M Values", "Average Lambdas");

        // Display (Lambda vs Spread)
        ArrayList<Double> averagesSetSorted = getSortedSet(averages);
        ArrayList<Double> spread = something(averages, averagesSetSorted);
        double[] averagesSetSortedArray = averagesSetSorted.stream().mapToDouble(i -> i).toArray();
        double[] spreadArray = spread.stream().mapToDouble(i -> i).toArray();
        showLineGraph(averagesSetSortedArray, spreadArray, "Average Lambda vs Spread", "Average Lambdas",
"Spread");
    }
}
```