CS 6385.001 - Algorithmic Aspects of Telecommunication Networks - F15
Project 1 Report

Peng Li - pxl141030@utdallas.edu

October 5, 2015

1 Introduction

This project is to design a software program that implements the network design task as taught in class. The input of the problem is the number of nodes N in the network, traffic demand value b_{ij} from node i to node j for each pair of nodes i and j, and traffic unit cost a_{ij} from node i to node j for each pair of nodes i and j.

Although N, b_{ij} and a_{ij} are input to this problem, in this project they need to be generated according to specific requirements:

- 30 nodes for all network design experiments (N = 30);
- b_{ij} is independently generated from set [0, 1, 2, 3, 4], we made the assumption in this implementation that all $b_{ii} = 0$;
- for any given node i, there will be k different low cost links with weight 1 from node i to other node j, they are randomly generated ($a_{ij} = 1$), and the rest (N - k - 1) links from node i will have high cost of 300 ($a_{ij} = 300$). Although the algorithm for computing shortest path can handle 0 weight self loops, we made the assumption that all $a_{ii} = 300$.

The goal is to design which links will be built and with how much capacity, so that the given traffic demand can be satisfied and the overall cost is minimum.

Instead of formulating linear programming problem, we utilize the shortest path algorithm - in particular, Dijkstra single source shortest path algorithm. This algorithm finds the shortest path from a source node to any other nodes in the graph. We will NOT implement the shortest path algorithm in the project, instead we will utilize the implementation from Princeton University Java algorithms and clients repo (http://algs4.cs.princeton.edu/code/).

As output, the program generates a network topology, with capacities assigned to the links. The program also calculates the total cost and density of the designed network.

We will experiment the network design algorithm with different k values (k = 3, 4, 5..., 15). For each k value, we experiment the design algorithm with 50 independently generated network traffic demands and link unit costs, we computes the average value of the network cost and density from these 50 experiments for each k.

We will be able to see that as k is getting bigger, the designed network cost is getting lower and network density is getting higher. In the end of the project, we also plot some of the designed network topologies graphically (when k = 3, 7, 11 and 15).

2 Program Description

2.1 General

The implementation uses Java JDK 1.7.

The network design problem is formulated into NetworkDesign class. When a new experiment needs to be carried out, we create a new instance of NetworkDesign class, or updates some of the instance variables (generate new random a_{ij} and b_{ij}) of an existing NetworkDesign object. The class implements several instance methods to generate random input variables, compute shortest path for all nodes, calculate network cost and density; it also implements a static method used to visualized the generated network topology.

Note that the NetworkDesign class requires dependencies on the following third party libraries:

- Princeton University Java algorithms and clients code , we will use the Dijkstra single source shortest path algorithm implemented in this library;
  download at http://algs4.cs.princeton.edu/code/algs4.jar
- JUNG 2 Java Universal Network / Graph Framework , we will use this library purely for graph visualization reason.
  download at http://sourceforge.net/projects/jung/files/

There is also a Project1 class. This is the driving class that makes calls to functions in NetworkDesign class. It also implements the calculation of average values for 50 independent experiments for each given k.

2.2 NetworkDesign class

The UML of Network Design class is shown in Figure 1.

The following methods implement the key functions for the network design task:

- setTrafficDemand(): void - this method generates independent random traffic demand value from [0, 1, 2, 3, 4] for every b_{ij}, and it sets all b_{ii} to 0;
- setUnitCost(): void - this method initialize all values in the unitCost matrix to 300, and then randomly generates numberOfLowCostEdge different indices j (that is not equal to i) for each node i and set a_{ij} to 1;
- setFlow(): void - this method calculates shortest path for each node i to any other nodes in the network, it updates the flow matrix based on the calculation result;
- setTotalCost(): void - this method calculates the total cost of the calculated network topology, it updates the totalCost variable;
- setDensity(): void - this method calculates the density of the calculated network topology, it updates the density variable;
- printInput(): void - prints to terminal the trafficDemand and unitCost matrices;
- printOutput(): void - prints to terminal the flow matrix;
- static visualizeGraph(int[][], int, int) - visualize the designed network topology to user screen.

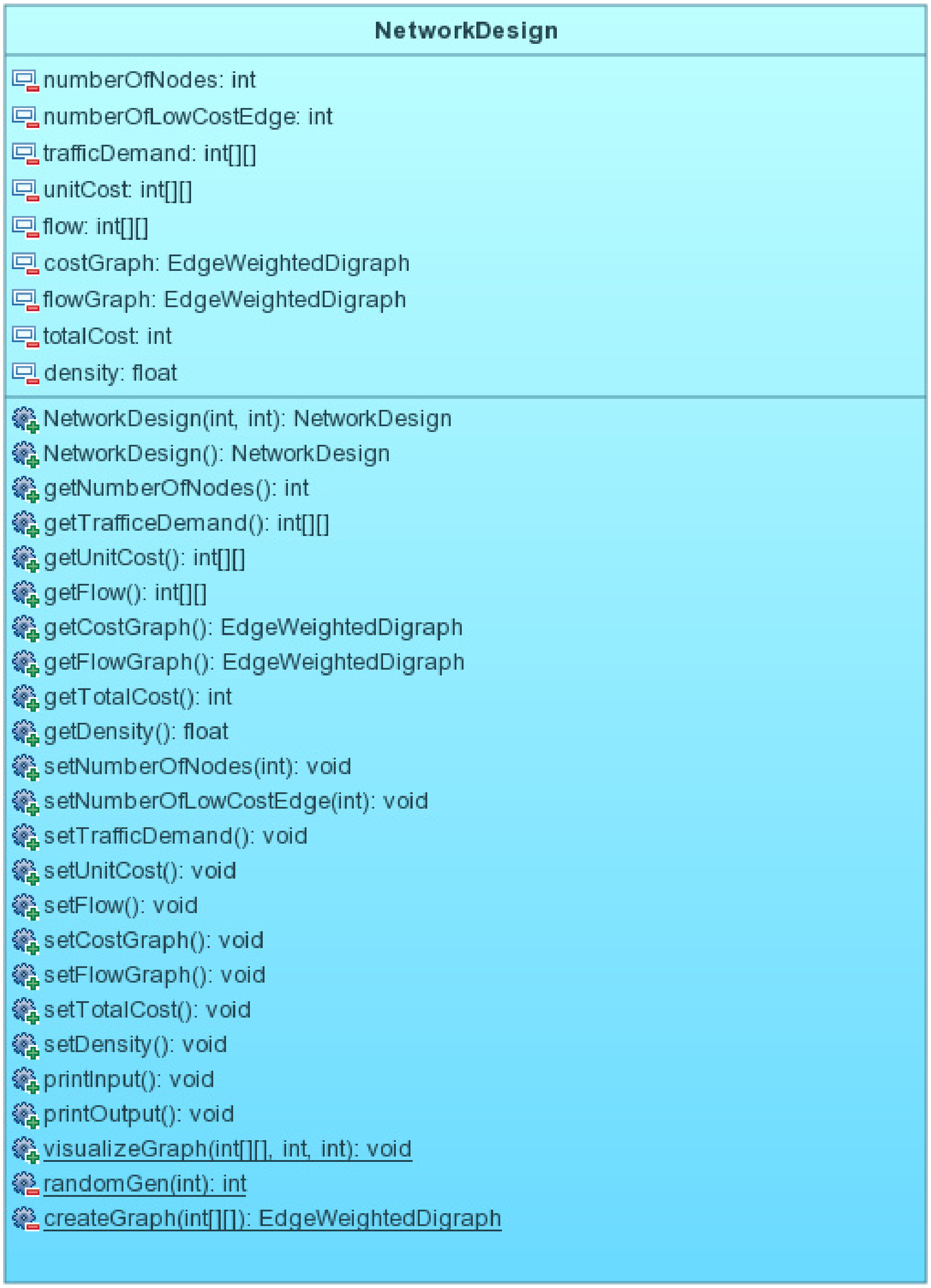

Figure 1: NetworkDesign class UML

2.3 Project1 class

Project1 class contains the main() method.

The main() method starts by creating a NetworkDesign object with 30 nodes. It then iterate through k from 3 to 15, each iteration contains a inner loop that has 50 iterations. In each of the 50 inner iterations, the program generates a new experiment by updating new network demands and unit cost value, it then calculates the new designed network together with its cost and density. After all 50 iterations, the program calculates the average cost and density for a given k.

After iterating through all the k from 3 to 15, the program prints out the average designed network cost and density to terminal.

It also generate a visualization of the designed network for k = 3, 7, 11 and 15.

2.4 How To Run

Before running the program, make sure to have the Java library as described in section 2.1 included in the class path.

- Create package named project1 and copy NetworkDesign.java and Project1.java into that package;
- In terminal, run "javac Project1.java" and then "java Project1";
- The program should print the average network cost and density for k from 3 to 15, and print out visualization of designed network topology for k = [4, 7, 11 and 15].

3 Experiment Result

The experiment result is shown in Table 1. A graphical representation of how the cost and density of the designed network depends on k is shown in Figure 2 and Figure 3.

Table 1: Experiment Result

k value	average network cost	average network density
3	34719.60	14.92%
4	10441.84	14.76%
5	5426.46	17.50%
6	4025.42	20.76%
7	3556.50	24.06%
8	3072.52	27.24%
9	3004.18	30.35%
10	2904.46	33.25%
11	2826.86	36.03%
12	2758.90	38.64%
13	2706.56	41.20%
14	2641.68	43.56%
15	2572.20	46.19%

For each node in the network, k represents the number of low cost (1) links from each nodes to some other nodes in the graph, there are other (N - k - 1) links that has high cost (300) to the other (N - k - 1) nodes. Therefore, the input graph to Dijkstra single source shortest path algorithm is a complete digraph,

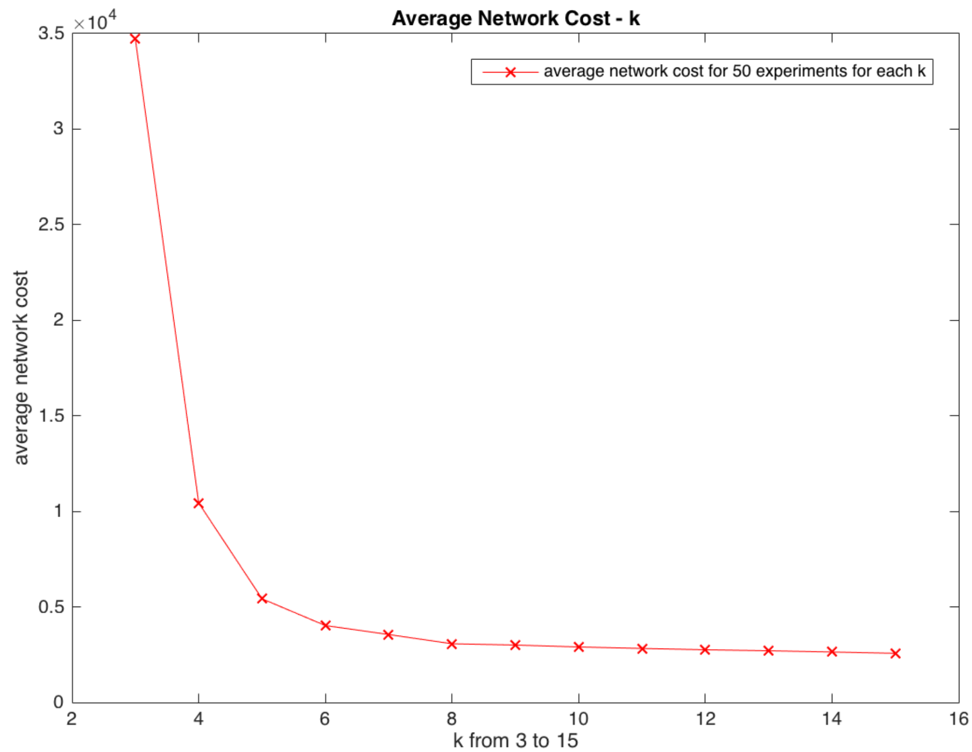

Figure 2: relation between total network cost and k

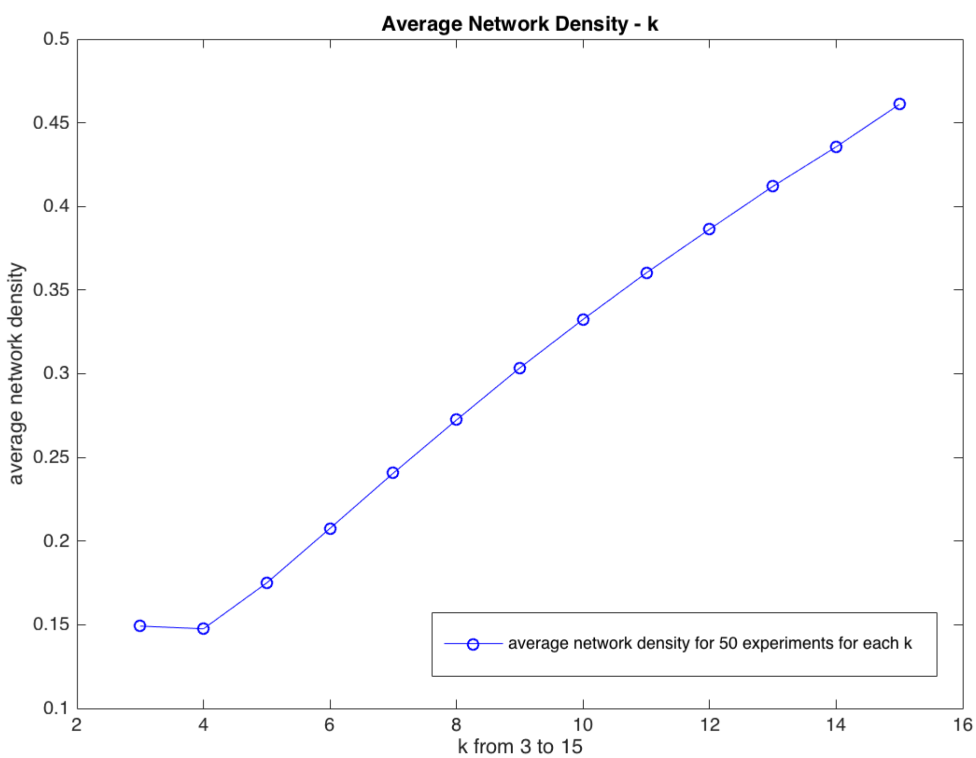

Figure 3: relation between network density and k

that means every node has links to all the other nodes in the network. This guarantees that a shortest path from any node to all other nodes always exists.

Because of the significance of the weight difference between low cost links and high cost links (1 v.s. 300), Dijkstra algorithms will always try to include the low cost links into the shortest path, even if there is a direct high cost link from the source to the destination. For example, even if the shortest path has to include all 29 low cost links (which is the maximum number of links in the case), which give a total weight of 29, it is still better than a direct link of weight 300.

On the other hand, if Dijkstra algorithm could not find a path from source to destination all using low cost links, it will use the high cost direct link from source to destination as shortest path, which will increase the total network cost significantly.

When k is small, e.g. k = 3, the network is not very much connected by low cost links. Although Dijkstra algorithms tries to utilize the low cost edge as much as possible, there are always cases that Dijkstra could not find a path from a source to a destination using low cost links only. Therefore, some high cost links are included in the designed network which results in a high network cost. Another reason the network cost is high is that even if some traffic demand is satisfied by using low cost links only, they are carried out in a very long path which includes several low cost links, this result in a high bandwidth requirement on those links and therefore contributes to the total cost as well. Because of this, the low cost links are used intensively when k is small in order to avoid using high cost links, the designed network will have fewer links but a higher average bandwidth per link. Therefore the network density is low when k is low.

As k gets larger, the network is getting more connected with low cost links. After a certain threshold, the network is very much connected by low cost links only. That's why we see a significant drop in network cost when k changes from 3 to 6, during this phase, the contribution of the cost drop is mainly from the replacement of high cost links by low cost links. As k gets even larger, the network cost continues to drop, but in a much lower speed, the contribution of cost drop during this phase is mainly from the replacement of longer low cost path by shorter low cost path, as number of low cost path grows.

As argued above, the average bandwidth per link for low cost links is getting lower as it is getting more available (as k gets larger). Therefore, more low cost links will be included in the designed network, therefore, the network density gets higher as k grows.

Something is missing in this model is that we did not consider the construction cost for links between nodes in the network. As k grows and network density gets larger, the construction cost for the designed network will also increase. In reality, this also need to be taken into consideration when designing the network. But this is not in the scope of this project.

Below is the network topology visualization when k = 3, 7, 11 and 15, note that

- The label of the 30 nodes are from 0 to 29;
- Due to the complexity of the graph, bandwidth of the links are not shown in the visualization.

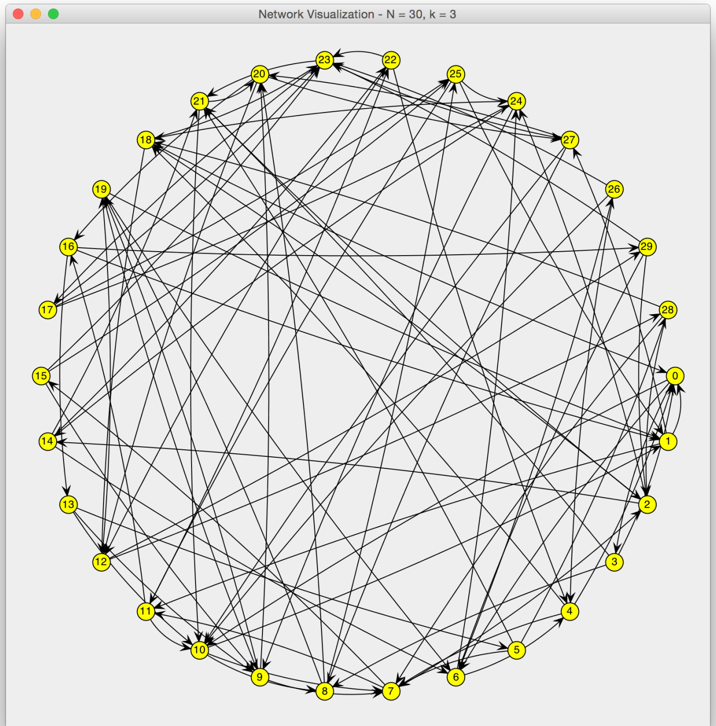

Figure 4: network topology k = 3

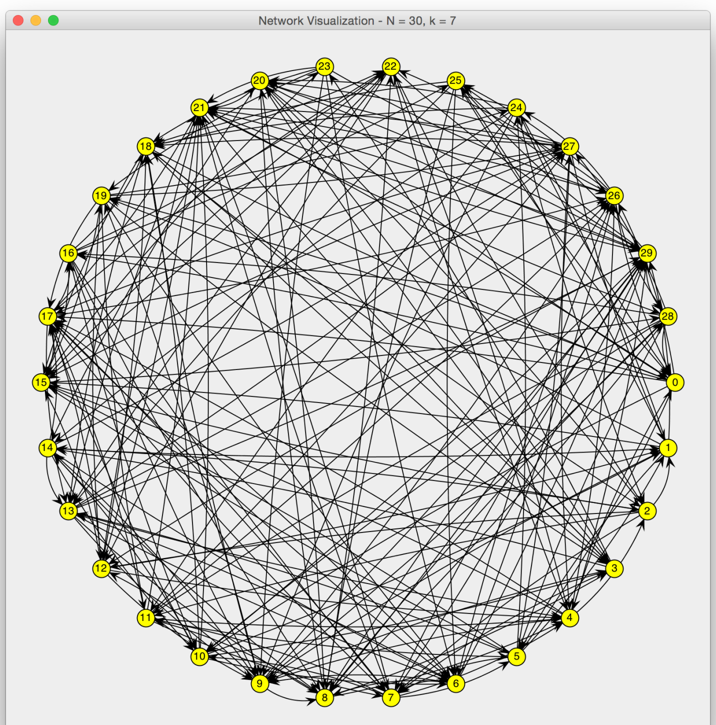

Figure 5: network topology k = 7

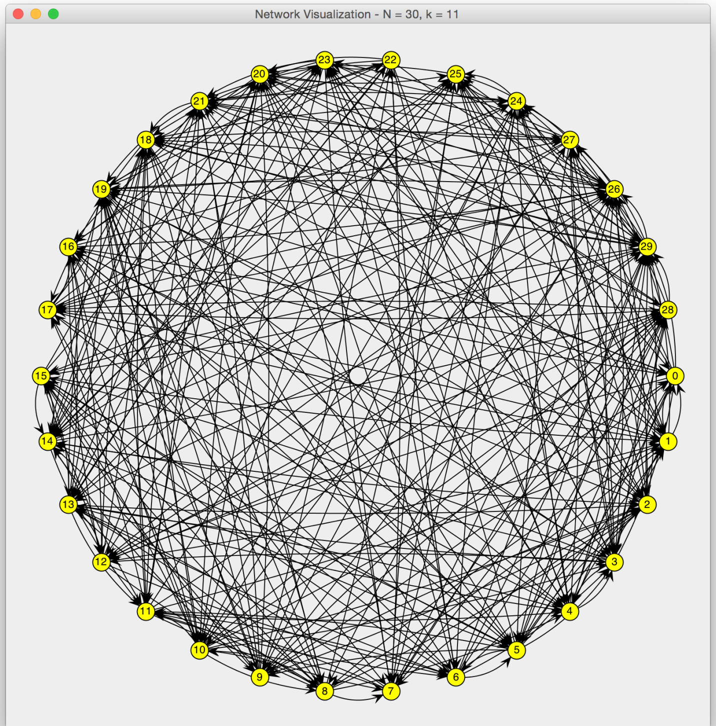

Figure 6: network topology k = 11

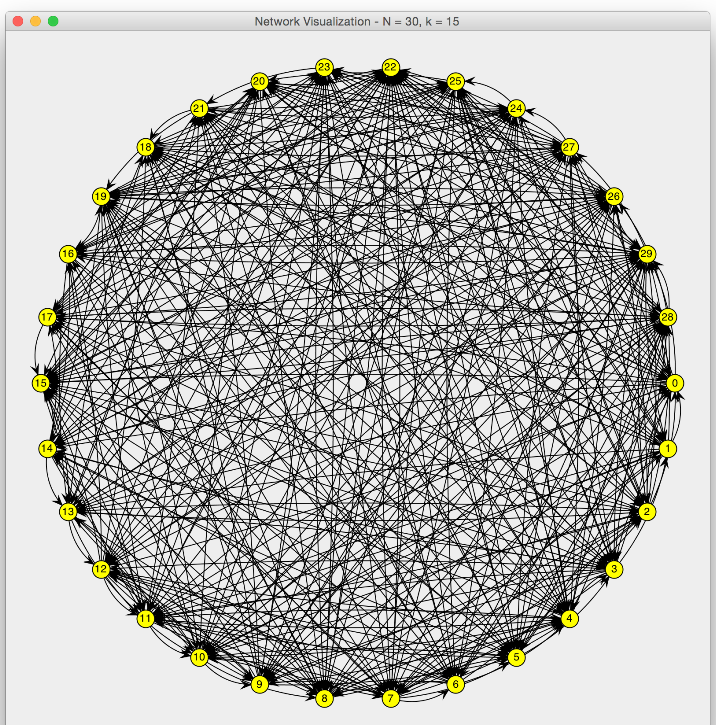

Figure 7: network topology k = 15

A Source Code

A.1 NetworkDesign.java

```java
package project1;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.Paint;
import java.util.HashSet;

import javax.swing.JFrame;

import org.apache.commons.collections15.Transformer;

import edu.princeton.cs.algs4.DijkstraSP;
import edu.princeton.cs.algs4.DirectedEdge;
import edu.princeton.cs.algs4.EdgeWeightedDigraph;
import edu.uci.ics.jung.algorithms.layout.CircleLayout;
import edu.uci.ics.jung.algorithms.layout.Layout;
import edu.uci.ics.jung.graph.DirectedSparseGraph;
import edu.uci.ics.jung.graph.Graph;
import edu.uci.ics.jung.graph.util.EdgeType;
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 NetworkDesign {

  private static final int DEFAULT_NUMBER_OF_NODES = 30;
  private static final int DEFAULT_NUMBER_OF_LOW_COST_EDGE = 5;
  private static final int MAX_DEMAND = 4;
  private static final int LOW_COST = 1;
  private static final int HIGH_COST = 300;

  private int numberOfNodes;
  private int numberOfLowCostEdge;
  private int[][] trafficDemand;
  private int[][] unitCost;
  private int[][] flow;
  private EdgeWeightedDigraph costGraph;
  private EdgeWeightedDigraph flowGraph;
  private int totalCost;
  private float density;

  public NetworkDesign(int numberOfNodes, int numberOfLowCostEdge) {
    this.numberOfNodes = numberOfNodes;
    this.numberOfLowCostEdge = numberOfLowCostEdge;
  }

  public NetworkDesign() {
    this.numberOfNodes = DEFAULT_NUMBER_OF_NODES;
    this.numberOfLowCostEdge = DEFAULT_NUMBER_OF_LOW_COST_EDGE;
  }

  public int getNumberOfNodes() {
    return numberOfNodes;
  }

  public int getNumberOfLowCostEdge() {
    return numberOfLowCostEdge;
  }

  public int[][] getTrafficeDemand() {
    return trafficDemand;
  }

```

```java
  public int[][] getUnitCost() {
    return unitCost;
  }

  public int[][] getFlow() {
    return flow;
  }

  public EdgeWeightedDigraph getCostGraph() {
    return costGraph;
  }

  public EdgeWeightedDigraph getFlowGraph() {
    return flowGraph;
  }

  public int getTotalCost() {
    return totalCost;
  }

  public float getDensity() {
    return density;
  }

  public void setNumberOfNodes(int n) {
    if (n <= 0) {
      throw new IllegalArgumentException(
          "number of nodes must be a positive number");
    } else {
      numberOfNodes = n;
    }
  }

  public void setNumberOfLowCostEdge(int n) {
    if ((n <= 0) || (n >= numberOfNodes)) {
      throw new IllegalArgumentException(
          "number of low cost edege must be a positive number, and must be less than number of nodes");
    } else {
      numberOfLowCostEdge = n;
    }
  }

  public void setTrafficDemand() {
    trafficDemand = new int[numberOfNodes][numberOfNodes];

    for (int i = 0; i < trafficDemand.length; i++) {
      for (int j = 0; j < trafficDemand[i].length; j++) {
        if (i == j) {
          trafficDemand[i][j] = 0;
        } else {
          trafficDemand[i][j] = randomGen(MAX_DEMAND);
        }
      }
    }
  }

  public void setUnitCost() {
    unitCost = new int[numberOfNodes][numberOfNodes];

    // initialize the all values of the cost matrix to HIGH_COST
    for (int i = 0; i < unitCost.length; i++) {
      for (int j = 0; j < unitCost[i].length; j++) {
        unitCost[i][j] = HIGH_COST;
      }
    }

    // randomly select k different edges that has cost LOW_COST
```

```
    for (int i = 0; i < unitCost.length; i++) {
      HashSet<Integer> indices = new HashSet<Integer>();
      while (indices.size() < numberOfLowCostEdge) {
        int index = randomGen(numberOfNodes - 1);
        if (!(index == i) && !indices.contains(index)) {
          indices.add(index);
        }
      }
      for (Integer k : indices) {
        unitCost[i][k] = LOW_COST;
      }
    }
  }

  public void setFlow() {
    flow = new int[numberOfNodes][numberOfNodes];
    for (int i = 0; i < flow.length; i++) {
      for (int j = 0; j < flow[i].length; j++) {
        flow[i][j] = 0;
      }
    }

    // use Dijkstra single source shortest path algorithm
    // calculate shortest path to all other nodes from source node
    for (int s = 0; s < costGraph.V(); s++) {
      DijkstraSP sp = new DijkstraSP(costGraph, s);

      for (int t = 0; t < costGraph.V(); t++) {
        if (sp.hasPathTo(t)) {
          for (DirectedEdge e : sp.pathTo(t)) {
            flow[e.from()][e.to()] += trafficDemand[s][t];
          }
        }
      }
    }
  }

  public void setCostGraph() {
    costGraph = createGraph(unitCost);
  }

  public void setFlowGraph() {
    flowGraph = createGraph(flow);
  }

  public void setTotalCost() {
    totalCost = 0;
    for (int i = 0; i < flow.length; i++) {
      for (int j = 0; j < flow[i].length; j++) {
        totalCost += flow[i][j] * unitCost[i][j];
      }
    }
  }

  public void setDensity() {
    int countEdge = 0;
    for (int i = 0; i < flow.length; i++) {
      for (int j = 0; j < flow[i].length; j++) {
        if (flow[i][j] > 0) {
          countEdge++;
        }
      }
    }

    density = (float) countEdge / (numberOfNodes * (numberOfNodes - 1));

  }

```

```
  public void printInput() {
    for (int i = 0; i < trafficDemand.length; i++) {
      for (int j = 0; j < trafficDemand[i].length; j++) {
        System.out.format("%3d" + " ", trafficDemand[i][j]);
      }
      System.out.println();
    }

    for (int i = 0; i < unitCost.length; i++) {
      for (int j = 0; j < unitCost[i].length; j++) {
        System.out.format("%3d" + " ", unitCost[i][j]);
      }
      System.out.println();
    }
    System.out.println(costGraph);
  }

  public void printOutput() {
    for (int i = 0; i < flow.length; i++) {
      for (int j = 0; j < flow[i].length; j++) {
        System.out.format("%3d" + " ", flow[i][j]);
      }
      System.out.println();
    }
    System.out.println(flowGraph);
    System.out.println("Total Cost is: " + totalCost);
    System.out.println("Network density is: " + density);
  }

  // method for flow graph visualization
  public static void visualizeGraph(int[][] matrix, int N, int k) {
    Graph<Integer, String> graph = new DirectedSparseGraph<Integer, String>();

    for (int i = 0; i < matrix.length; i++) {
      graph.addVertex((Integer) i);
    }

    for (int i = 0; i < matrix.length; i++) {
      for (int j = 0; j < matrix[i].length; j++) {
        // only select edge that has flow bigger than 0
        if (matrix[i][j] > 0) {
          graph.addEdge(i + "->" + j, i, j, EdgeType.DIRECTED);
        }
      }
    }

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

    BasicVisualizationServer<Integer, String> vv = new BasicVisualizationServer<Integer, String>(
        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("Network Visualization - N = " + N + ", k = "
        + k);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.getContentPane().add(vv);
```

```java
    frame.pack();
    frame.setVisible(true);
  }

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

  private static EdgeWeightedDigraph createGraph(int[][] edgeMatrix) {
    EdgeWeightedDigraph output = new EdgeWeightedDigraph(edgeMatrix.length);
    for (int i = 0; i < edgeMatrix.length; i++) {
      for (int j = 0; j < edgeMatrix[i].length; j++) {
        DirectedEdge edge = new DirectedEdge(i, j, edgeMatrix[i][j]);
        output.addEdge(edge);
      }
    }
    return output;
  }
}
```

A.2 Project1.java

```java
package project1;

public class Project1 {

  private static final int NUMBER_OF_ITERATIONS = 50;

  public static void main(String[] args) {
    NetworkDesign network = new NetworkDesign();

    float[] avgTotalCost = new float[16];
    float[] avgDensity = new float[16];

    for (int i = 0; i < 16; i++) {
      avgTotalCost[i] = 0;
      avgDensity[i] = 0;
    }

    for (int k = 3; k <= 15; k++) {
      // run the test NUMBER_OF_ITERATIONS times for every k from 3 to 15
      // calculate the average value of total cost and network density
      network.setNumberOfLowCostEdge(k);
      float costSum = 0;
      float densitySum = 0;
      for (int i = 0; i < NUMBER_OF_ITERATIONS; i++) {

        network.setTrafficDemand();
        network.setUnitCost();
        network.setCostGraph();
        network.setFlow();
        network.setFlowGraph();
        network.setTotalCost();
        network.setDensity();
        costSum += network.getTotalCost();
        densitySum += network.getDensity();

        if ((k + 1) % 4 == 0 && i == 8) {
          NetworkDesign.visualizeGraph(network.getFlow(),
              network.getNumberOfNodes(),
              network.getNumberOfLowCostEdge());
        }
      }
      avgTotalCost[k] = costSum / NUMBER_OF_ITERATIONS;
      avgDensity[k] = densitySum / NUMBER_OF_ITERATIONS;
    }

    for (int i = 3; i < avgTotalCost.length; i++) {
      System.out.format("%6.2f  ", avgTotalCost[i]);
    }

    System.out.println();

    for (int i = 3; i < avgDensity.length; i++) {
      System.out.format("%6.4f  ", avgDensity[i]);
    }

  }
}
```