CE 6385.0W1

Algorithmic Aspects of Telecommunication Networks (Su22)

Instructor: Dr. Andras Farago

Project 1 - Analyzing Algorithmic Aspects of Shortest-Path based Network Design Algorithm

Submitted By

Sunil Kumar, Arjun (AXS210011)

Date: 05 July 2022

0. Objective

For this project, our primary objectives are as follows.

1. Create a program to **compute the optimal total cost** and **print the optimal network topology (with network density)** solving the given network demand problem defined with
    - N: Number of Nodes
    - K: A dynamic argument
    - a_{ij}: Unit Cost for the link between node i to j (Generated using K)
    - $b_{ij}\mathrm{Mbit/s}$: Traffic Demand between node i to j (Generated using NETID)
2. **Explain** how the “Shortest path based fast algorithm” is implemented in the program using FlowChart etc.
3. **Plot** the graph for and justify the results
    - Total Cost vs K (for K = 3 to 14)
    - Density vs K
4. **Visualize** network topology for K = 3, 8 & 14
5. **Detail** about Program Structure, Code and execution instructions, etc

1. Introduction

Network design is a process of finding an optimal topological design with minimal cost that satisfies the required constraints (a_{ij}, $b_{ij}\mathrm{Mbit/s}$ etc) for the nodes and edges.

There are several ways to solve this. So far, in this course, we have seen two of them:

- Linear Programming
- Shortest Path Based Fast Algorithm

1.1 Linear Programming

This approach involves defining

- A linear **objective function**, in our case minimum total cost,
- that is subjected to **constraints** specific to the output network requirements.

Once you have the LP formulation, we can solve it using brute force general LP

algorithms. But it could get slow as the variables in the constraints can get large.

Therefore in this project, we will focus on the “Shortest Path Fast Algorithm” for solving this.

1.2 Shortest Path Based Fast Algorithm

We observe that the cheapest way of sending b_{kl} amount of flow from node k to l is to send it all along the cheapest cost path.

If this path consists of nodes $k = i_1, i_2, \ldots, i_{r-1}, i_r = l,$ then a portion of the total optimal cost is

$$b_{kl}\left(a_{i_1,i_2} + \ldots + a_{i_{r-1},i_r}\right)$$

Due to the linear nature of the model, to find the total cost, we can simply sum them up independently.

ie, if E_{kl} is the set of edges that are on the min cost $k \to l$ path. Then, according to the above argument, the optimal cost is

$$Z_{\text{opt}} = \sum_{k,l} \left(b_{kl} \sum_{(i,j)\in E_{kl}} a_{ij} \right).$$

2. Implementation Details

The Project is developed in Java 8 with Maven Dependency Manager. I am also using some external libraries for

- Dijkstra Shortest Path [1]
- Visualizing graph [2]

The code is well organized into 3 main modules (*Fig 1*) segregating the responsibilities.

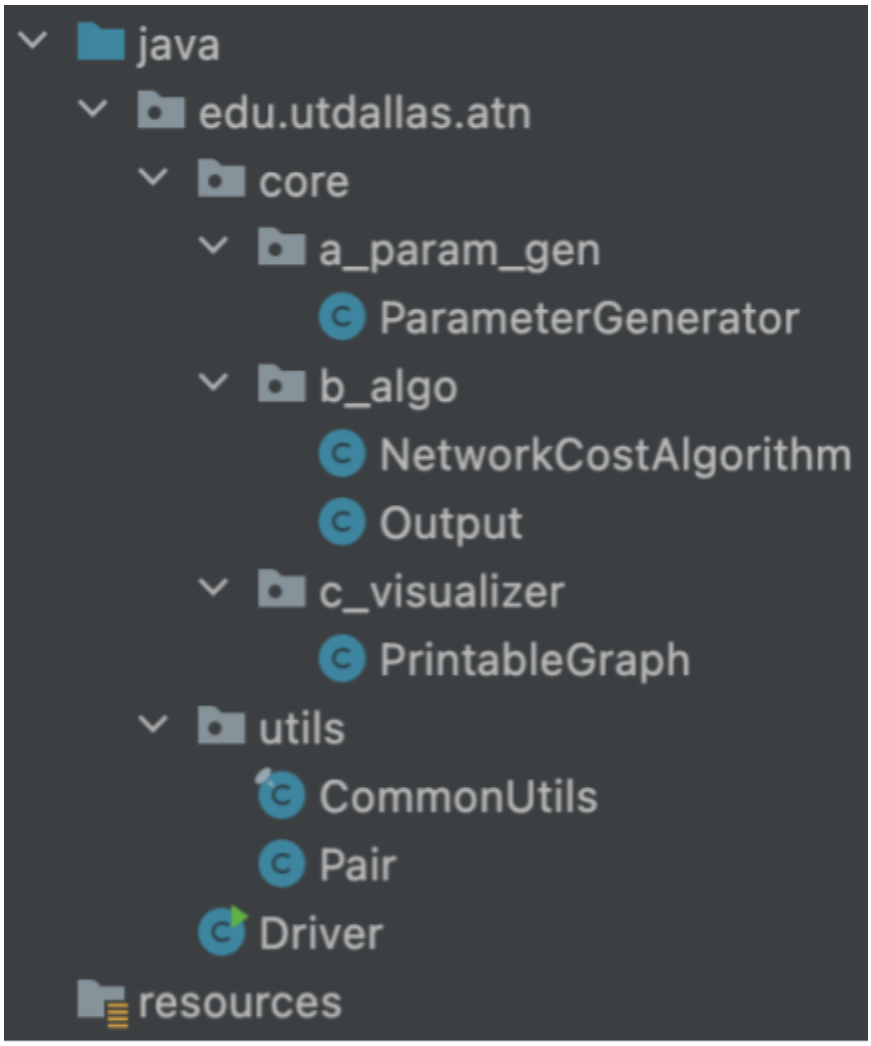

Fig 1: Project Structure

2.1 Driver

Driver class is the entry point for this program, that accepts user argument k. It then pipelines the overall flow ie,

- generating inputs
  - a_{ij} using K
  - b_{ij} using NetID
- running Dijkstra's algorithm for finding the shortest distance between each node and using that to compute the optimal total cost, density, and topology

- finally printing the topology graph as PNG.

We will now, go through each module one by one.

2.1 ParameterGenerator

This is the class that mainly deals with input generation. It accepts node count, (in our case, we are using it as 21), NETID, and the dynamic argument *k*.

We have 2 functions

1. Generate Traffic Demand: We generate this using the formula

   $$b_{ij} = |d_i - d_j|$$

   Where d_i is the i^{th} index value of a 21-digit number (created by appending NETID repeatedly, until it reached a size of 21).

2. Generate Unit Costs a_{ij}: We generate this by
   - picking *k* random indices in each row, and set it as 1,
   - Set self-edges as 0, and
   - the rest set as 100.

2.2 NetworkCostAlgorithm

This is the class responsible for running Dijkstra's algorithm and computing the optimal cost. It accepts a_{ij}, b_{ij}, *k* and *n* as inputs.

We are using **SimpleDirectedWeightedGraph** from the **jgrapht**[1] library. First, we populate all the *N* vertices in this graph object. Then we populate all the edges, with the endpoints and their corresponding weights, using a_{ij}.

We have a total of 3 functions, ie

1. findShortestPathCost(): This function is responsible for calling the library function and returning the **magnitude** of the shortest path using Dijkstra's algorithm.

2. findShortestPathEdges(): This function is responsible for calling the library function and returning the **Shortest Path** (ie List of Points) using Dijkstra's algorithm.

3. Calculate(): We use the above two functions here to calculate the optimal cost, density, and optimal topology and return it as an **Output** Object to the *Driver* class.

   As a part of the algorithm, we run 2 for-loops to **iterate over each pair** of vertices,

   - we find the **optimal path** cost between that vertices using the Dijkstra algorithm
   - Multiply that optimal cost with the **traffic demand** for that edge.

   and finally, sum up all these costs. This will give the total optimal cost.

   In this same for loop, we include the logic for noting down the **optimal path** (topology), so that it can be returned within the Output Object.

   The density is calculated in the end, using the calculated network topology.

2.3 PrintableGraph

This is the class that is responsible for creating the PNG image of the Network Topology. From the NetworkCostAlgorithm we will get the Output object, which is also containing the Optimal Topology. We use this information, and pass it **Graphviz [2]** library for printing the topology. The image is saved in the output folder, with a file name, graph_*k*.png, where *k* is the dynamic argument.

2.4 Misc Classes

There are a few helper classes that are used in the overall flow.

- **Output**: A wrapper object containing cost, density, and topology.
- **CommonUtils**: Contains a function for returning NETID
- **Pair**: A Key-Value pair wrapper

The overall process flow is shown in *Fig 2*.

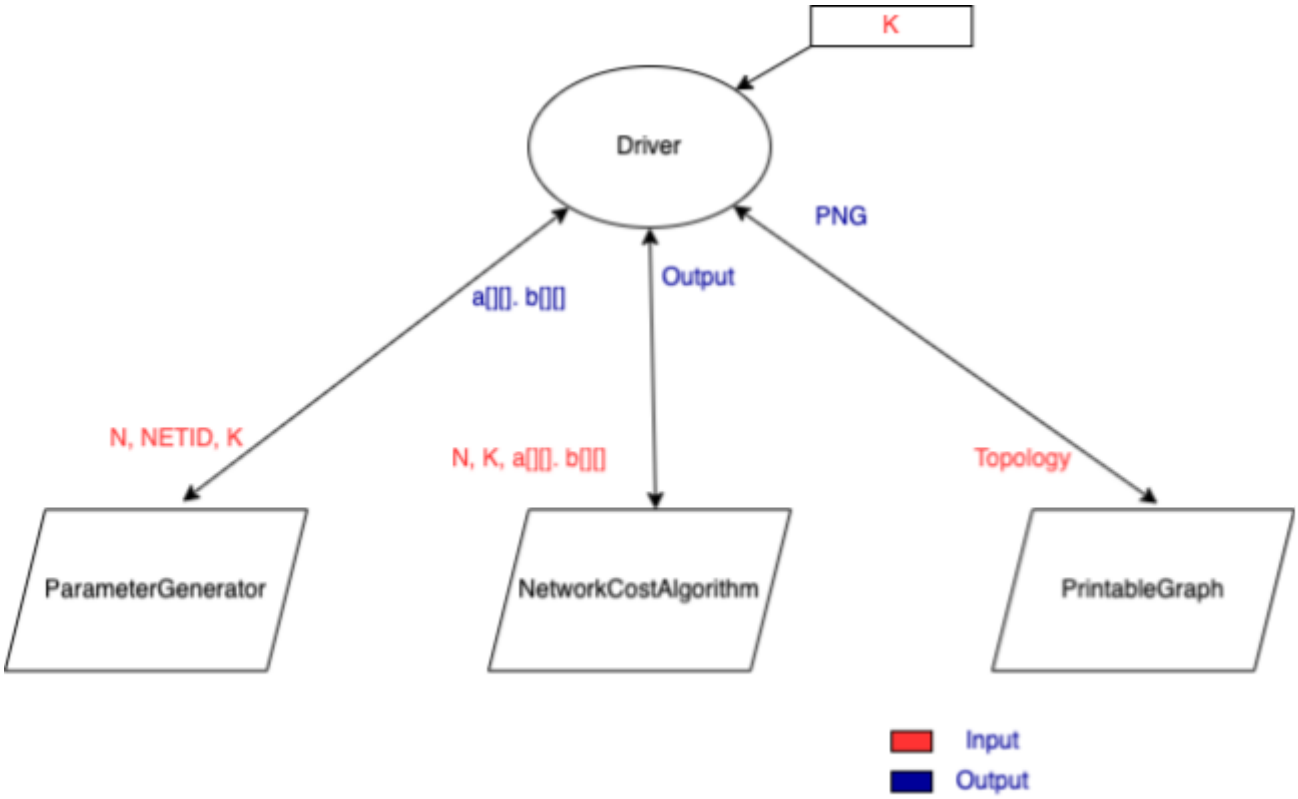

Fig 2: Flow Chart describing how Driver interacts with the 3 modules.

3. Analysis and Interpretation

3.1 Graphs Plotting

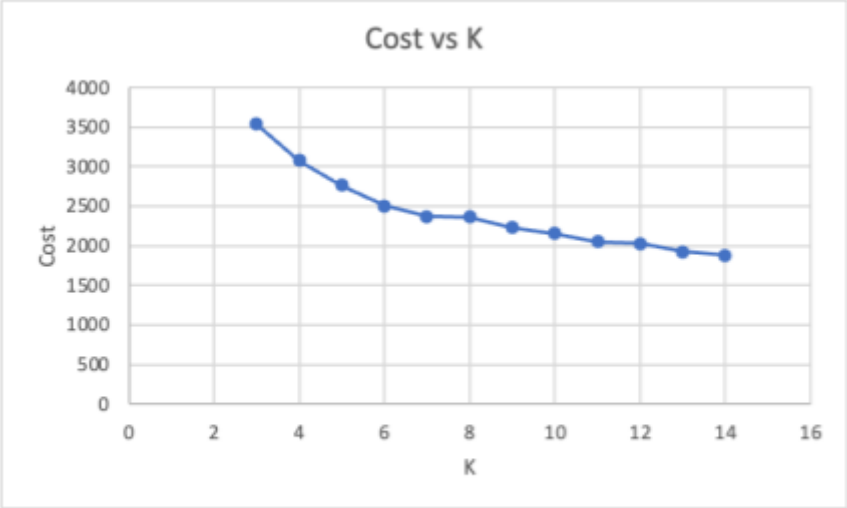

Fig 3: Cost vs K graph

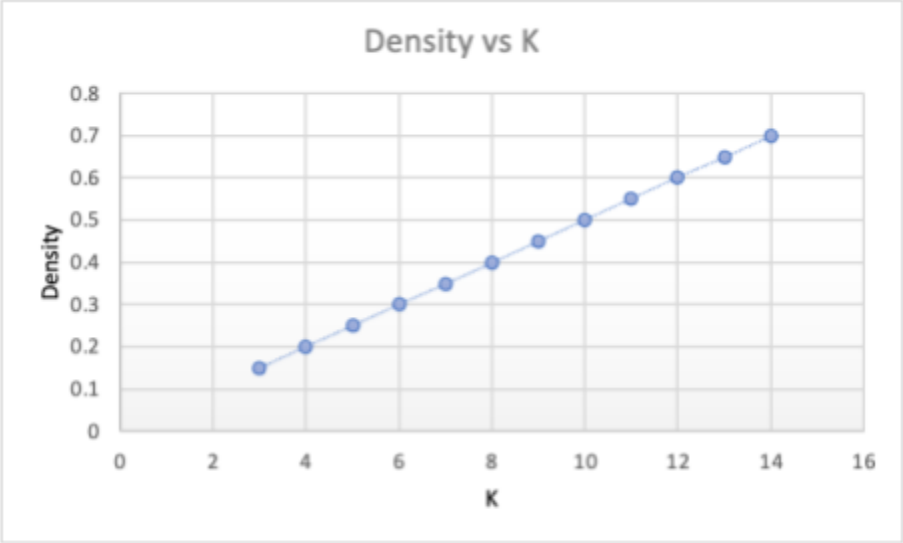

Fig 4: Density vs K

3.2 Visualization

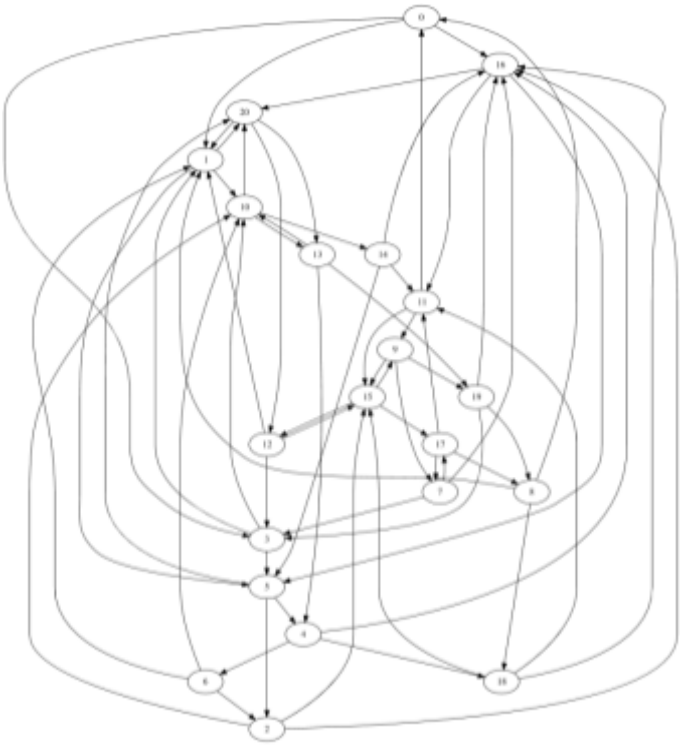

Fig 5: Network Topology when K = 3

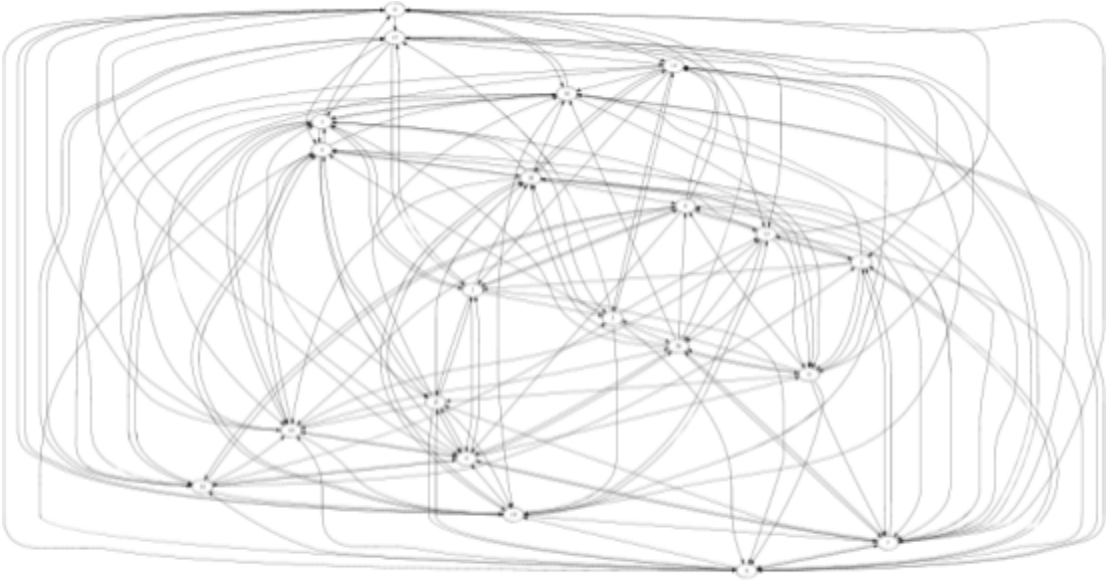

Fig 6: Network Topology when K = 8

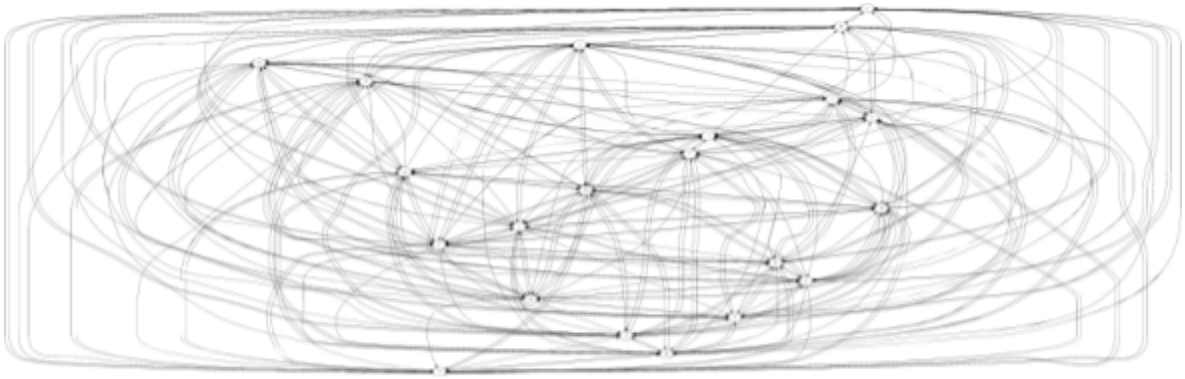

Fig 7: Network Topology when K = 14

3.3 Justification

1. Optimal Cost: As per *Figure 3*, you can see that the Optimal Cost decreases logarithmically, with the increase in *k*. This behavior is due to:

- The algorithm for generating the unit cost matrix.
- Selection of the Shortest Path-based Network Design Algorithm

As the k increases, more and more low-cost outbound links are added to the graph due to the unit cost generation algorithm.

This gives Dijkstra's algorithm to explore new cheaper routes and thereby reducing the net cost between two nodes.

As the Optimal cost is the summation of these individual costs, there is a significant reduction (logarithmic) in the Total Optimal Cost.

2. Density: As per *Figure 4*, you can see that density increases with *k*. This behavior is due to the algorithm used for generating the unit cost matrix using the argument *k*. As k increases, the number of low-cost links (ie with weight 1) going out of the nodes increases. Due to this, several connected shorter edges that are cheaper than the expensive direct path will be added to the optimal topology, thereby increasing the edge count, and density.

3. Network Topology: As you can see, from *Figures 5,6 & 7*, the graph is becoming denser and denser as the value of K increases. This can be explained with the same reasoning given above for Density.

4. ReadMe

I am assuming that the machine has Java 8, and Maven already installed.

To run the project, follow the below instructions:

1. Download the complete Maven project and unzip it.
2. Inside the project folder run the below commands to create a uber(fat) jar.

   > mvn clean install

3. After this, the fat jar is generated in the /target folder.
4. To run the application, use the below commands

   > java -jar target/uber-atn-project1-1.0-SNAPSHOT.jar

5. Appendix:

5.1 Project structure

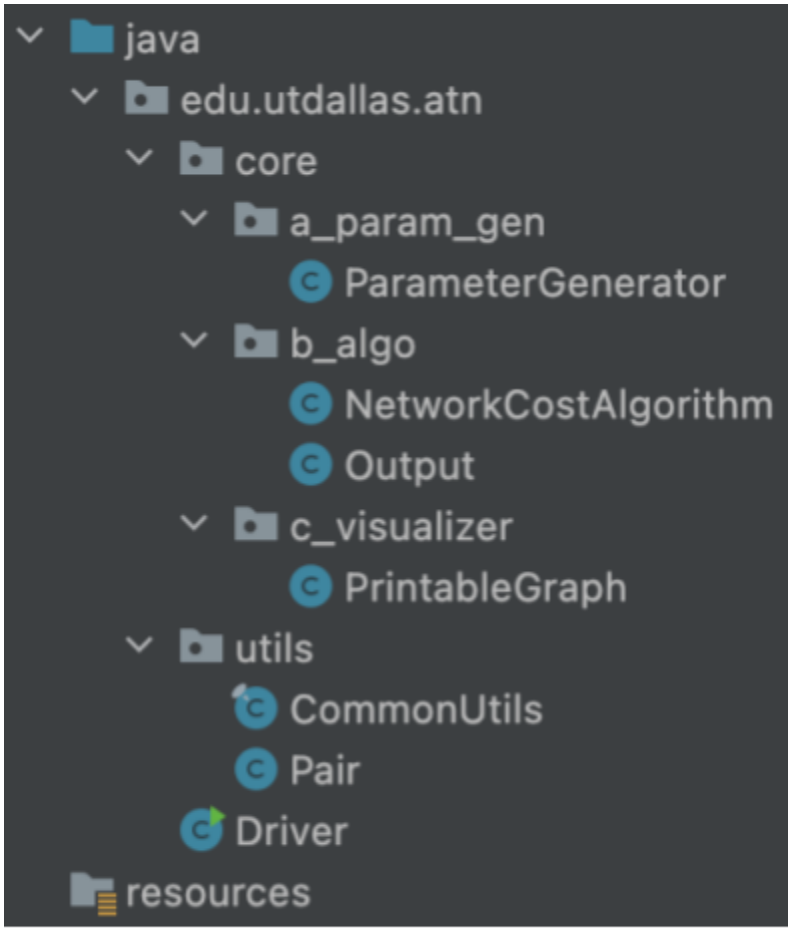

Fig 8: Project Structure

5.2 Source Code

1. Driver.java

```java
package edu.utdallas.atn;

import edu.utdallas.atn.core.a_param_gen.ParameterGenerator;
import edu.utdallas.atn.core.b_algo.NetworkCostAlgorithm;
import edu.utdallas.atn.core.b_algo.Output;

import java.util.Scanner;

public class Driver {
  public static void main(String[] args) {

    // 1. Input Generation
    Scanner sc = new Scanner(System.in);

    int n = 21; // may be based on 2021!
    System.out.println("Enter K");
    int k = sc.nextInt();

    ParameterGenerator pg = new ParameterGenerator(n);
    pg.setDynamicArgument(k);

    int[][] a = pg.generateUnitCosts();
    int[][] b = pg.generateTrafficDemands();

    // 2. Init the Algorithm
    NetworkCostAlgorithm algo = new NetworkCostAlgorithm(n, k, a, b);
    Output ans = algo.calculate();
    System.out.println("Total cost for the network: " + ans.getCost());
    System.out.println("Density for the network: " + ans.getDensity());

    // 3. Visualize
    ans.getGraph().render();
  }
}
```

2. ParameterGenerator.java

```java
package edu.utdallas.atn.core.a_param_gen;

import edu.utdallas.atn.utils.CommonUtils;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class ParameterGenerator {

  private final int[] userId;
  private final int nodeCount;
  int dynamicArgument;

  public ParameterGenerator(int nodeCount) {
    this.nodeCount = nodeCount;
    this.userId = CommonUtils.getUserId();
  }

  // b[i][j]
  public int[][] generateTrafficDemands() {
    int[][] b = new int[nodeCount][nodeCount];

    for (int i = 0; i < nodeCount; i++) {
      for (int j = 0; j < nodeCount; j++) {
        b[i][j] = Math.abs(userId[i] - userId[j]);
      }
    }

    return b;
  }

  // a[i][j]
  public int[][] generateUnitCosts() {

    int[][] a = new int[nodeCount][nodeCount];

    for (int i = 0; i < nodeCount; i++) {

      // pick random k indices
```

```
      List<Integer> list = new ArrayList<>();
      for (int currIdx = 0; currIdx < nodeCount; currIdx++)
            if (currIdx != i) list.add(currIdx);
      Collections.shuffle(list);
      list = list.subList(0, dynamicArgument);

      for (int j = 0; j < nodeCount; j++) {
        if (i == j) a[i][j] = 0; // self edge costs 0
        else if (list.contains(j)) a[i][j] = 1; // set k indices as 1
        else a[i][j] = 100; // rest as 100
      }
    }

    return a;
  }

  public void setDynamicArgument(int dynamicArgument) {
    this.dynamicArgument = dynamicArgument;
  }
}
```

3. Output.java

```java
package edu.utdallas.atn.core.b_algo;

import edu.utdallas.atn.core.c_visualizer.PrintableGraph;

public class Output {
  private final PrintableGraph graph;
  private final int cost;
  private final double density;

  public Output(PrintableGraph graph, int cost, double density) {
    this.graph = graph;
    this.cost = cost;
    this.density = density;
  }

  public PrintableGraph getGraph() {
    return graph;
  }

  public int getCost() {
    return cost;
  }

  public double getDensity() {
    return density;
  }
}
```

4. NetworkCostAlgorithm.java

```
package edu.utdallas.atn.core.b_algo;

import edu.utdallas.atn.core.c_visualizer.PrintableGraph;
import edu.utdallas.atn.utils.Pair;
import org.jgrapht.GraphPath;
import org.jgrapht.alg.shortestpath.DijkstraShortestPath;
import org.jgrapht.graph.DefaultWeightedEdge;
import org.jgrapht.graph.SimpleDirectedWeightedGraph;

import java.util.ArrayList;
import java.util.List;

public class NetworkCostAlgorithm {

  private final int n;
  private final int k;

  private final int[][] a;
  private final int[][] b;

  private final SimpleDirectedWeightedGraph<
                Integer,
                DefaultWeightedEdge> graph;

  public NetworkCostAlgorithm(int n, int k, int[][] a, int[][] b) {
    this.graph = new SimpleDirectedWeightedGraph<>
                            (DefaultWeightedEdge.class);
    this.n = n;
    this.k = k;

    this.a = a;
    this.b = b;

    // add vertices
    for (int i = 0; i < n; i++) graph.addVertex(i);

    // add edges
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
```

```
        if (i == j) continue;

        DefaultWeightedEdge edge = graph.addEdge(i, j);
        double weight = a[i][j];
        graph.setEdgeWeight(edge, weight);
      }
    }
  }

  private int findShortestPathCost(int start, int end) {
    GraphPath<Integer, DefaultWeightedEdge> path =
        DijkstraShortestPath.findPathBetween(graph, start, end);

    int cost = 0;
    for (DefaultWeightedEdge edge : path.getEdgeList())
        cost += graph.getEdgeWeight(edge);

    return cost;
  }

  private List<Pair<Integer, Integer>>
          findShortestPathEdges(int start, int end) {

    GraphPath<Integer, DefaultWeightedEdge> path =
        DijkstraShortestPath.findPathBetween(graph, start, end);

    List<Pair<Integer, Integer>> result = new ArrayList<>();
    for (int i = 0; i < path.getVertexList().size() - 1; i++) {
      result.add(new Pair<>(path.getVertexList().get(i),
                            path.getVertexList().get(i + 1)));
    }

    return result;
  }

  public Output calculate() {

    PrintableGraph printableGraph = new PrintableGraph(k);
    int totalCost = 0;

    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
```

```
        // calc cost
        int currCost = b[i][j] * findShortestPathCost(i, j);
        totalCost += currCost;

        // calc graph
        for (Pair<Integer, Integer> edge : findShortestPathEdges(i, j)) {
          printableGraph.addEdge(edge.getKey(), edge.getValue());
        }
      }
    }

    // calc density
    double totalPossibleNumberOfDirectedEdges = n * (n - 1);
    double numberOfDirectedEdgesWithNonZeroWeights =
                    printableGraph.getEdgeCount();
    double density =
        numberOfDirectedEdgesWithNonZeroWeights /
        totalPossibleNumberOfDirectedEdges;

    return new Output(printableGraph, totalCost, density);
  }
}
```

5. PrintableGraph.java

```
package edu.utdallas.atn.core.c_visualizer;

import guru.nidi.graphviz.engine.Format;
import guru.nidi.graphviz.engine.Graphviz;
import guru.nidi.graphviz.model.MutableGraph;
import java.io.File;
import static guru.nidi.graphviz.model.Factory.mutGraph;
import static guru.nidi.graphviz.model.Factory.mutNode;

public class PrintableGraph {

  MutableGraph printableGraph;
  String imageName;

  public PrintableGraph(int k) {
    printableGraph = mutGraph("Optimal Network").setDirected(true);
    imageName = "graph_" + k + ".png";
  }

  public void addEdge(int start, int end) {
    printableGraph.add(mutNode("" + start).addLink(mutNode("" + end)));
  }

  public int getEdgeCount() {
    return printableGraph.edges().size();
  }

  public void render() {

    try {
      Graphviz.fromGraph(printableGraph)
          .width(800)
          .render(Format.PNG)
          .toFile(new File("output/" + imageName));
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
}
```

6. CommonUtils.java

```
package edu.utdallas.atn.utils;

public final class CommonUtils {

  public static int[] getUserId() {
    return new int[]
    {2, 0, 2, 1, 5, 9, 6, 0, 7, 4, 2, 0, 2, 1, 5, 9, 6, 0, 7, 4, 2};
  }
}
```

7. Pair.java

```
package edu.utdallas.atn.utils;

public class Pair<K, V> {
  K key;
  V value;

  public Pair(K key, V value) {
    this.key = key;
    this.value = value;
  }

  public K getKey() {
    return key;
  }

  public V getValue() {
    return value;
  }
}
```

7. 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>edu.utdallas</groupId>
    <artifactId>atn-project1</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
        <maven-shade-plugin.version>3.2.2</maven-shade-plugin.version>
       <maven-compiler-plugin.version>3.8.1</maven-compiler-plugin.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.jgrapht</groupId>
            <artifactId>jgrapht-core</artifactId>
            <version>1.4.0</version>
        </dependency>


        <!-- Graph Visualize-->
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-core</artifactId>
            <version>2.13.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-slf4j-impl</artifactId>
            <version>2.13.0</version>
        </dependency>

        <dependency>
            <groupId>guru.nidi</groupId>
```

```
            <artifactId>graphviz-java</artifactId>
            <version>0.16.2</version>
        </dependency>

    </dependencies>

    <build>
        <plugins>

            <!-- Maven Compile -->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>${maven-compiler-plugin.version}</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>

            <!-- Uber Jar-->
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-shade-plugin</artifactId>
                <version>${maven-shade-plugin.version}</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <filters>
                        <filter>
                            <artifact>*:*</artifact>
                            <excludes>
                                <exclude>META-INF/*.SF</exclude>
                                <exclude>META-INF/*.DSA</exclude>
                                <exclude>META-INF/*.RSA</exclude>
                            </excludes>
```

```
                        </filter>
                    </filters>

      <finalName>uber-${project.artifactId}-${project.version}</finalName>
                    <transformers>
                        <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTra
nsformer">
                            <mainClass>edu.utdallas.atn.Driver</mainClass>
                        </transformer>
                    </transformers>
                </configuration>
            </plugin>
        </plugins>
    </build>


</project>
```

6. Bibliography

1. **Jgrapht:** *https://jgrapht.org/*
2. **GraphViz:** *https://github.com/nidi3/graphviz-java*