CE 6385.0W1

Algorithmic Aspects of Telecommunication Networks (Su22)

Instructor: Dr. Andras Farago

Project 2 - Design and Analyze Heuristic Algorithms for Network Topology Design

Submitted By

Sunil Kumar, Arjun (AXS210011)

Date: 21 July 2022

0. Objective

For this project, our primary objectives are as follows.

1. **Designing** 2 Heuristic Algorithms satisfying the given network constraints:
    a. All nodes are connected
    b. The degree of each vertex is >= 3
    c. The diameter of the graph is <=4
    d. Total cost(geometric total length) is as low as possible
2. **Prototyping** the solution in Java, supporting
    a. Generation of N random node coordinates
    b. Calculation of optimal cost for the output topology using both heuristic algorithms
    c. Visualization of the network topology
3. **Compare** the cost and visual output of 2 Heuristic Algorithms for N= 15, 20, 25,30,35.
4. Draw an **analysis** of how 2 algorithms perform.

1. Introduction

Heuristic algorithms are algorithms that are designed to provide near-optimal solutions to NP problems. They are faster than Greedy solutions. Sometimes Greedy Algorithms could take exponential time for solving an NP problem. For most of the real-world use cases, we are good with near-optimal solutions rather than the supreme best, if the result can be computed faster.

A popular use case of Heuristic algorithms is in Database Planners, where we want to determine the order of executing relational operators for the given SQL. Many popular databases like Ingres, Postgres, etc, use heuristics to create a query execution plan.

2. Algorithm Design

I will be designing 2 heuristic algorithms to solve the given network design problem.

2.1 Relevant Courseworks

In the process of coming up with the 2 algorithms, I will be mostly using techniques learned in the following courses:

- Computational Geometry [1]
- Design and analysis of algorithms [2]
- Distributed Computing [3]
- Algorithmic aspects of telecommunication networks. (of course)

Computation Geometry [1]: I have seen a few algorithms for k-means clustering, KD trees, and Incremental construction which I believe can be used in this project. Due to the spatial nature of this project, I also plan to use the concepts from my spatial partitioning project[4], that I developed in this course work.

Design & Analysis of Algorithms [2]: This serves as a base for algorithm design, data structure choice, and complexity analysis.

Distributed Computing [3]: If I can reduce the problem to non-overlapping sub-problems, then I can run distributed algorithm on each sub-problems to optimize the solution further. We might not implement this, but it is worth thinking in that direction to further improve the speed and performance.

2.2 Algorithm 1: Incremental Reduction

The simplest solution that satisfies the given network constraints would be to convert the graph to a complete graph, ie inter-connect all the vertices. This would result in higher cost, as we would be having unnecessary edges, which could be dropped and yet satisfy the graph constraints. The algorithm that we develop now evolves from this idea.

In computation geometry, I learned a technique called Incremental Construction for solving Linear Programing problems. Simply said, we incrementally add LP constraints one at a time, updating the current optimal, until all the constraints are added.

We will do something similar, but instead of incrementally adding, we will delete edges one by one from the complete graph, and validate constraints in each iteration. This would be the exact opposite of incremental construction, hence I named it “Incremental reduction.”

As I mentioned, in every iteration, we delete the largest edge and check the graph constraints.

- If satisfied, we proceed to the next iteration, with the new graph.
- If not, we stop and return the old graph. (ie undoing the last edge deletion)

Pseudocode

Below is the pseudocode for the logic.

1. Initialize *current_graph* with all the vertices
2. Make the *current_graph* a complete graph (ie connect all vertices)

3. Declare a *max_heap* for holding edges in the sorted order of edge distance.
    a. Push all the edges in the *current_graph* to the *max_heap*

4. While the *max_heap* is not empty, perform the following
    a. Pick the *largest_edge* from the *max_heap*

    b. create a new clone of the *current_graph*, say *temp_graph*
    c. Remove the *largest_edge* from the *temp_graph*

    d. check if the *temp_graph* satisfies our graph conditions
        - All nodes reachable
        - Diameter <= 4
        - Degree >= 3
    e. if the above conditions are satisfied, we move to the next iteration setting the *current_graph* as *temp_graph*.
    f. if the above conditions fail, we break this loop and return the *current_graph* as output. (ie undoing the *largest_edge* deletion)

2.2 Algorithm 2: Spatial Partitioned Algorithm

By looking at the cartesian coordinate system problem, and its constraints, I was trying to map it to the geospatial domain and borrow techniques from that domain to simplify the solution.

Thoughts

These were the thoughts that helped me formulate the solution:

- 2nd constraint, ie Diameter should be <=4.
- If we can reduce the problem to smaller non-overlapping partitions, we can reduce the number of long-distance edges, which can reduce the cost significantly for the sparse datasets.
- If we can generalize this to a more real-world use case, say a geospatial problem, we can use spatial techniques similar to geo-hashing [5], etc to simplify the algorithm.

How to visualize?

I could not find a good library to draw cartesian points in Java. As the points are coordinate-based, I thought of converting this problem to a geospatial landscape so that I can convert the output topology to GeoJSON [12] and visualize it easily on any online GeoJSON editors [6]. This was simple for me, as I already had worked with GeoJSON in the past for my computational geometry project [4].

Terminology:

- Complete graph: A simple undirected graph in which every pair of distinct vertices is connected by a unique edge
- Segments, Rectangles, Cluster, & Partitions are used interchangeably.
- Node, vertex, points & coordinate are used interchangeably.

How should I partition the points?

I create a bounding box, which covers all the points in our dataset. Then, I partitioned the bounding rectangle into 9 sub-rectangles as shown in *Fig 1*.

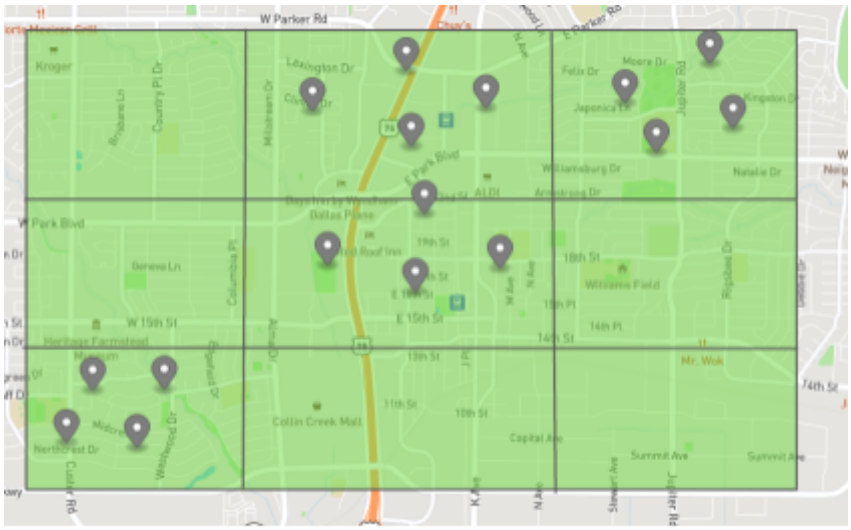

Fig 1: Partition point sets into sub-rectangles

Each sub-rectangle would act as a cluster. We create a complete graph for nodes within each cluster as shown in *Fig 2*.

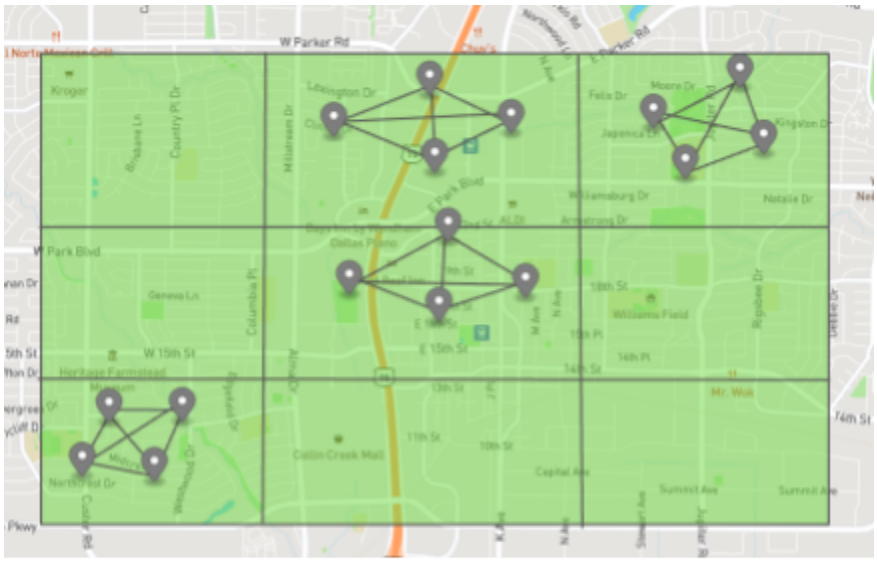

Fig 2: Create Complete Graph in each sub-rectangle

We pick the first node in each cluster to be the cluster master as shown in *Fig 3*.

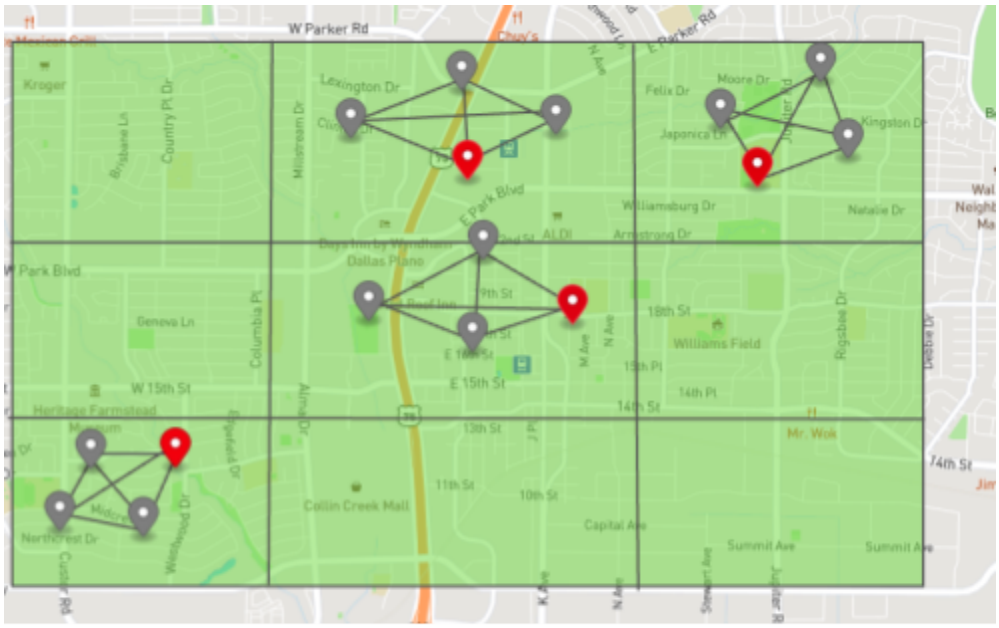

Fig 3: Pick Cluster_Master from each sub-rectangle

Now we can think of 2 topologies for connecting cluster masters

- Hub Spoke Model (if the center rectangle is not empty) [7]
- Complete graph of cluster masters (failover)

Hub Spoke Model of Cluster Masters:

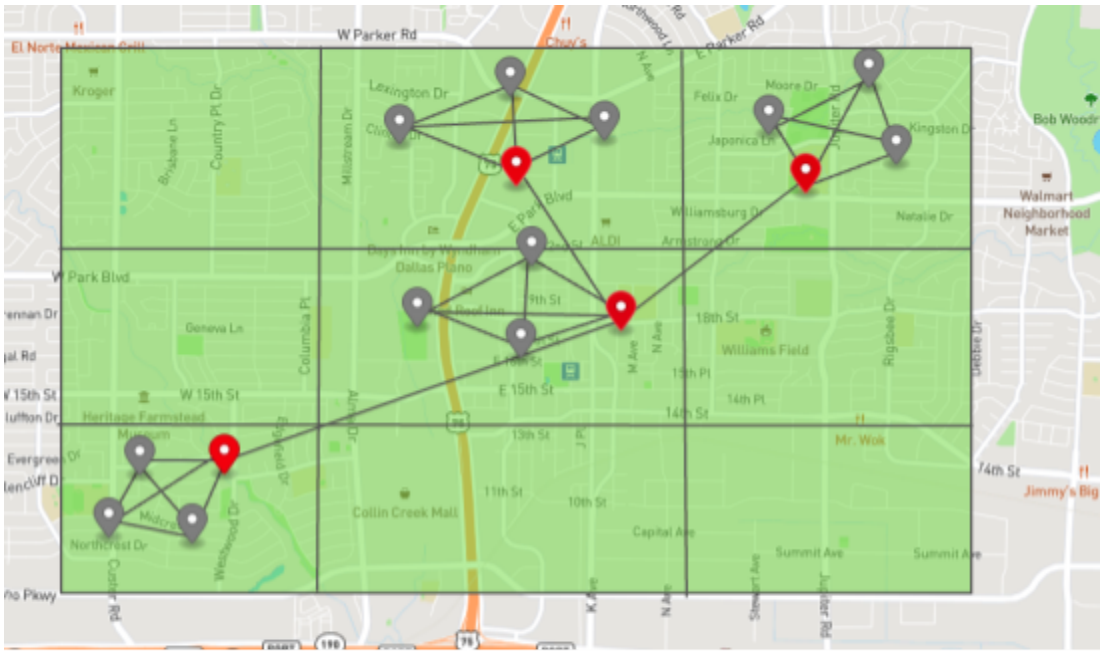

Fig 4: Hub-Spoke of cluster_masters

Here we connect all the boundary sub-rectangles *cluster_masters* to the center sub-rectangle *cluster_master* as shown in *Fig 4*. This is the most efficient graph, that costs less and still satisfies, the Diameter <=4 constraints.

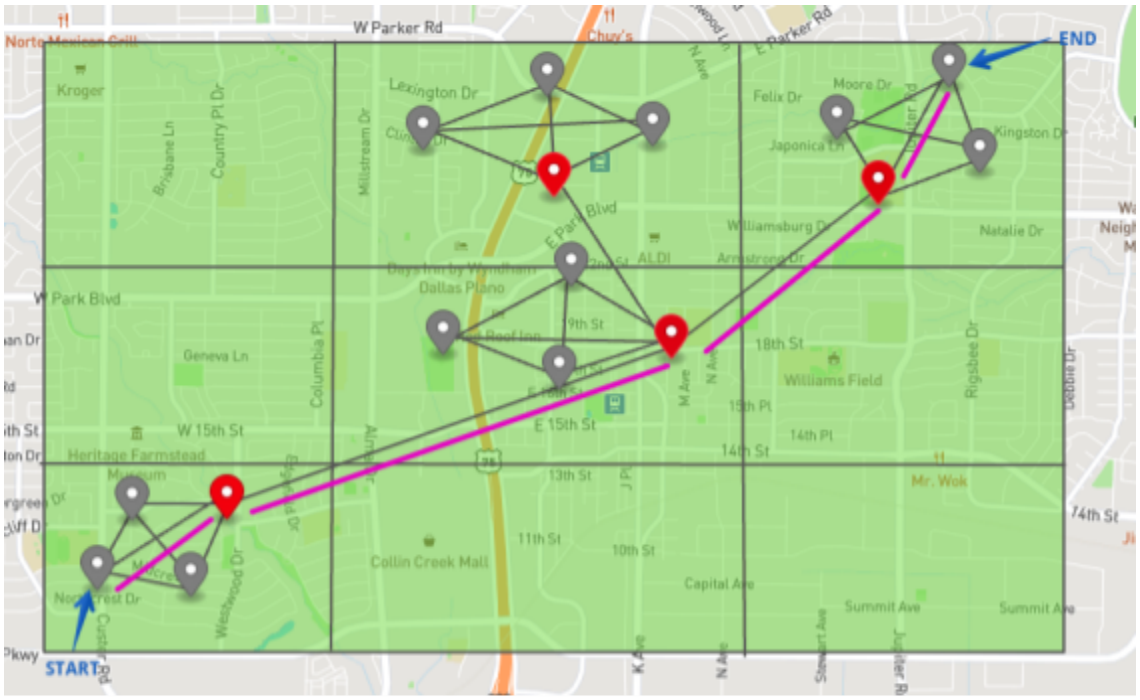

Fig 5: 4 hop path from START to END

Let's take a simple example as above (*Fig 5*). We can travel from any point to any other point in 4 hops, ie, the diameter is <=4.

Notes:

1. There are chances that a sub-rectangle might not have >=4 vertices and the slave vertices might not be satisfying the degree>=3 constraints. In this case, we do a correction procedure, that will be discussed later.

The gist of the correction procedure is that after completion of the Heuristic 2 algorithm, you pick each of the vertexes that have degrees <3. We find the closest neighboring vertices of each of these vertices and add edges between them to satisfy the degree condition. This is a simple approach and serves our purpose.

2. The hub-spoke model only works if the center sub-rectangle is not empty. If it is empty we create a complete graph of cluster_masters. It will be discussed next.

Complete Graph of cluster_masters

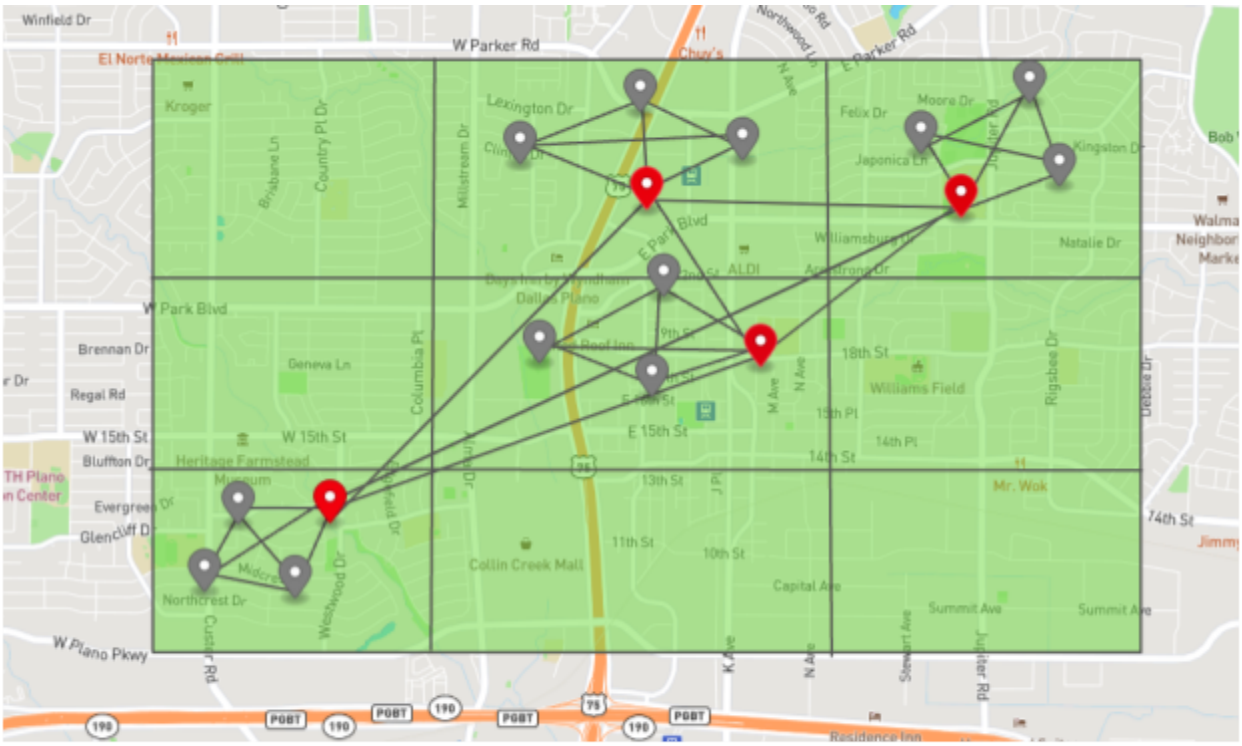

Fig 6: Complete graph of cluster_masters

Here, you can see that we are creating a complete graph of *cluster_masters* as shown in *Fig 6*. This ensures that we can travel between every vertex in less than 3 hops (ie Diameter <=3).

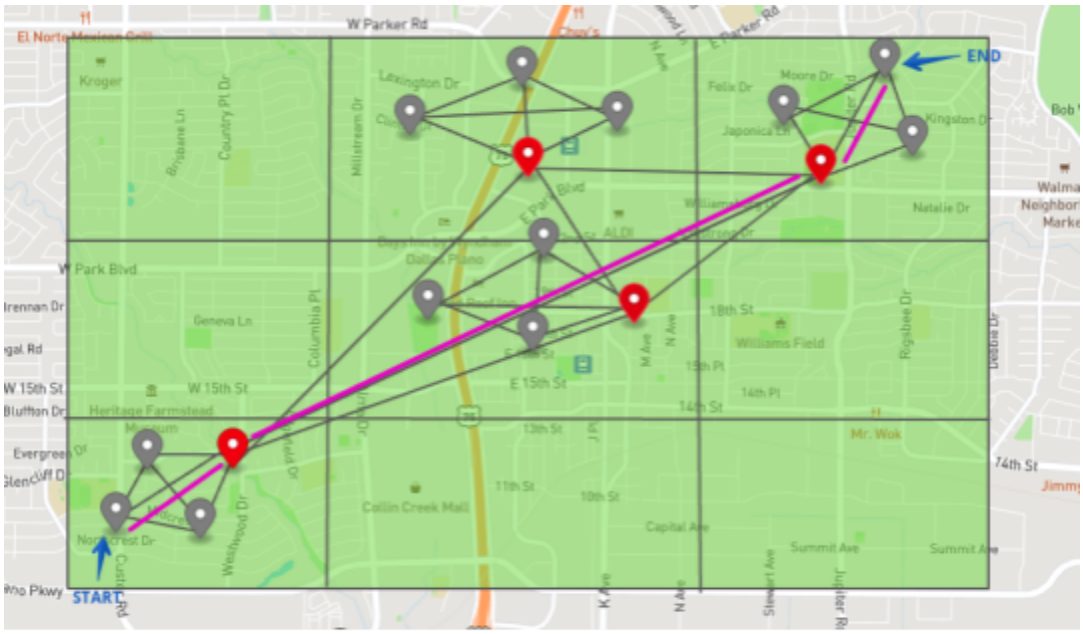

Fig 25: 3 hop path between START & END

Let's take a simple example as shown in *Fig 25*. In the worst case, we will have 3 hops

1. Edge from START node to *cluster_master* of that sub-rectangle
2. Edge from *cluster_master* of START-sub-rectangle to *cluster_master* of END-sub_rectangle.
3. END-sub_rectangle *cluster_master* to END node.

Rationale:

In this algorithm, I am trying to ensure

- Strong connectivity at the individual cluster level, so that every cluster node forms a complete graph, thereby ensuring degree >=3.
- Strong connectivity at cluster_master level, so that we can ensure that edges between every pair of vertices are <=4.

The advantage of this **hierarchical** approach is that

- You **reduce the cost**, as we reduce the number of long distant edges.
- You increase the strong connection locally, thereby trying to **maintain Degree>=3**
- You reduce the hop distance and maintain **diameter <=4**.

Pseudocode:

1. Create a 3x3 partition of the whole point space
    a. This involved finding the top_right and bottom_left points among the points and drawing a bounded rectangle using that.
    b. Then we calculate the width & height and calculate the delta to be added to get the coordinates of sub-rectangle corners.
2. Map points to these segments
    a. We check if the point lies in the sub-rectangle or not using coordinate comparisons.
3. Create a complete graph with all the coordinates within the individual sub-rectangle.
4. Pick one point (ie cluster_master) from each sub-rectangle. Connect all the cluster_masters in hub-spoke or complete graph topology.
5. Check if all the vertices have a degree >=3, if not perform the below
    a. Find the vertex which has a degree <3, and calculate the difference in degree(let's call it k).
    b. Find the k nearest neighbors and connect the vertex to that nodes.
    c. Repeat a. until all the vertices have degree >=3.

3. Implementation Details

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

- **geojson-jackson** [11]: for generating output geo-json
- **lucene-spatial** [9]: for calculating the distance between two coordinates.
- **JUnit** [10]: for testing individual classes.

The code is well organized into 3 main modules (*Fig 7*) segregating the responsibilities.

Fig 7: Project Structure

3.1 Driver.java

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

- generating random coordinates
- running the Heuristic algorithm1 & Heuristic Algorithm 2.
- finally printing the GeoJSON that can be passed on to online editors.

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

3.2 ParameterGenerator.java

This is the class that mainly deals with input generation. It accepts node count and generates N coordinates (Latitude & Longitude) points.

Here we are using the inbuild random generation function to generate decimal values for latitude, and longitude within a particular range.

```
ThreadLocalRandom.current().nextDouble(min, max);
```

I love Texas! And for that reason, I have used latitude and longitude range to be around the Plano region.

3.3 HeuristicAlgorithm1.java

This class implements the Heuristic 1 algorithm. It accepts a list of points and returns the output graph. Note that we are using a custom data structure for Point and Graph to abstract out common functions. It will be discussed in detail later.

3.4 HeuristicAlgorithm2.java

This class pretty much does the same thing as the class above but implements the Heuristic 2 algorithm. As described in the pseudo-code, we will

1. First, create a bounding box, and create sub-rectangles
2. Map all the points to one of the sub-rectangles
3. Interconnect all points within each sub-rectangle.
4. Cluster Master, ie the first node from every sub-rectangle is picked.
5. We connect all the cluster masters to form a complete graph for cluster masters or hub-spoke model.
6. If any node doesn't satisfy the degree constrain ie degree <3, then we do the correction process on all these unsatisfied nodes.
    a. We find the nearest neighbors and connect them with this node to ensure that the degree constraints are met.

3.5 GoeJsonSerializer.java

This class is mainly responsible for converting the Graph object to a GeoJSON string. We also add additional properties like marker color etc to format the final map. We use an external library[11] for GeoJSON conversion.

3.6 Edge.java

This class represents an edge between 2 coordinates.

3.7 Graph.java

This class represents the graph. It is a composite data structure that contains

- Adjacency matrix
- Index: Point to Index
- Reverse Index: Index to Point

This class has functions:

- addEdge()
- removeEdge()
- makeItCompleteGraph()
- getSmallestDegree()
- getDiameter()
- getEdges()
- getCost()

To check the diameter, we use Floyd-Warshall [8] algorithm to find the all-pair shortest path distance.

3.8 Point.java

This class is representing latitude, and longitude. It also has the **distanceTo()** function which is implemented using an external library [9].

3.9 Rectangle.java

This class represents a rectangle. We use the bottom-left and top-right points of the rectangle. It also has a **contains()** [13] which determines if a point lies inside the rectangle or not.

4. Analysis and Interpretation

4.1 Visualization

When N=15

Fig 8: Complete Graph, Cost: 226,019

Fig 9: Heuristic 1, Cost: 126,779

Fig 10: Heuristic 2, Cost: 33,232

When N=20

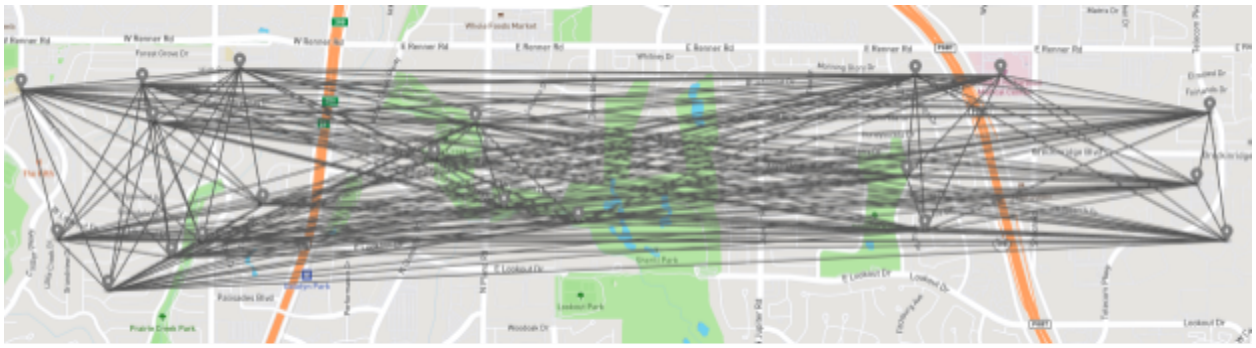

Fig 11: Complete Graph, Cost: 567,650

Fig 12: Heuristic 1, Cost: 86,288

Fig 13: Heuristic 2, Cost: 38,076

When N=25

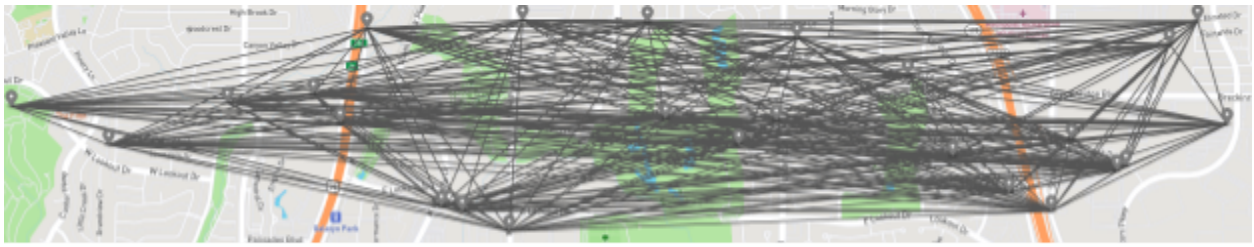

Fig 14:Complete Graph, Cost: 794,817

Fig 15: Heuristic 1, Cost: 142,957

Fig 16: Heuristic 2, Cost: 46,695

When N=30

Fig 17: Complete Graph, Cost: 1,148,755

Fig 18: Heuristic 1, Cost: 217,231

Fig 19: Heuristic 2, Cost: 59,988

When N=35

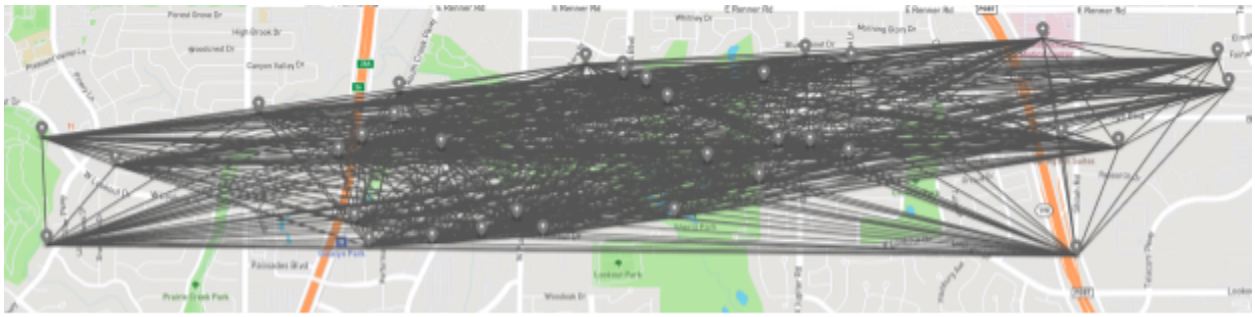

Fig 20: Complete Graph, Cost: 1,309,024

Fig 21: Heuristic 1, Cost: 355,699

Fig 22: Heuristic 2, Cost: 78,422

4.2 Cost Plotting

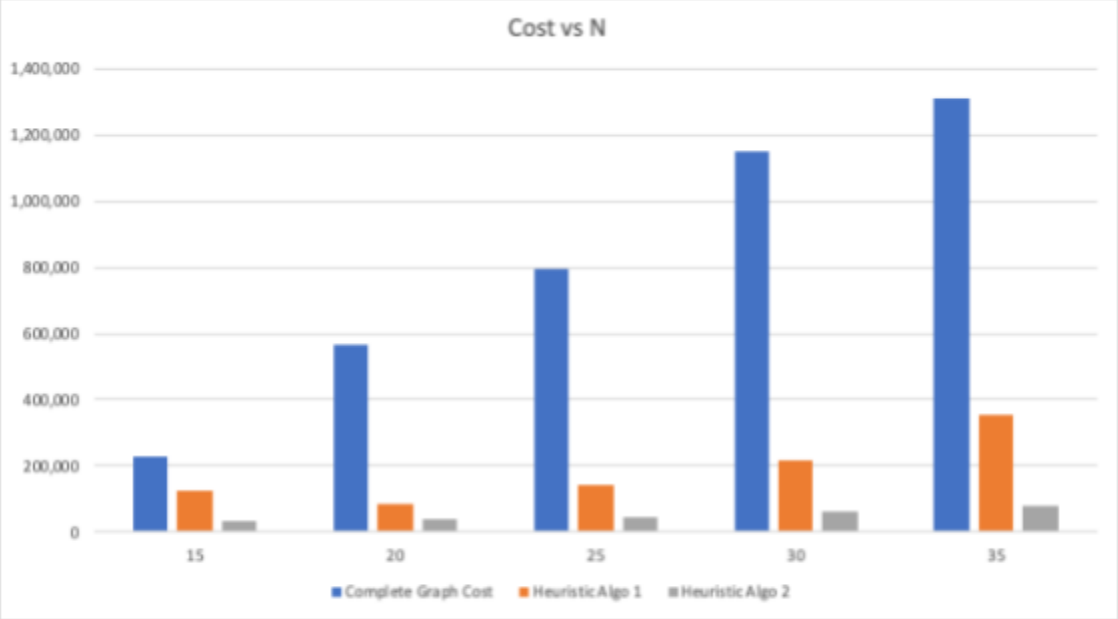

Fig 23: Cost comparison for Complete Graph, Heuristic Algo1, Heuristic Algo2

4.3 Data Analysis

Complete Graph:

Complete graph is by far the simplest and most expensive solution to this network design problem. Taking the complete graph cost as a reference value, we can understand the cost benefits of both the heuristic algorithms.

Heuristic Algorithm 1:

You can see in *Fig 23*, that, as N increases, Heuristic 1, significantly reduces the cost. It mainly removes expensive edges, which when removed don't impact the graph constraints. Also, if you see Figs: 9,12,15,18,21; the graph is less dense than the complete graph shown in Figs: 8,11,14,17,20.

Heuristic Algorithm 2:

You can see from *Fig 23*, that Heuristic Algorithm 2, significantly outperformed Heuristic Algorithm 1. The main reason would be that it is able to intelligently skip long edges, and maintain only long edges between cluster_masters. This would drastically reduce the cost, added by long edges.

Fig 10 & *Fig 22* clearly portrays the sparse graph, that we are able to create using this algorithm. This algorithm generates localized clusters and the clusters_masters are either hub-spoke connected or complete graphs. Using spatial partitioning has significantly helped in optimizing the network cost.

5. 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-project2-1.0-SNAPSHOT.jar

6. Appendix:

6.1 Project structure

Fig 24: Project Structure

6.2 Source Code

1. Driver.java

```java
package edu.utdallas.atn.p2;

import com.fasterxml.jackson.core.JsonProcessingException;
import edu.utdallas.atn.p2.core.a_param_gen.ParameterGenerator;
import edu.utdallas.atn.p2.core.b_algos.HeuristicsAlgo1;
import edu.utdallas.atn.p2.core.b_algos.HeuristicsAlgo2;
import edu.utdallas.atn.p2.core.c_visualize.GeoJsonSerializer;
import edu.utdallas.atn.p2.domain.Graph;
import edu.utdallas.atn.p2.domain.Point;

import java.util.List;
import java.util.Scanner;

public class Driver {
 public static void main(String[] args) throws JsonProcessingException {
   // 1. Input Generation
   Scanner sc = new Scanner(System.in);
   System.out.println("Enter N");
   int n = sc.nextInt();

   ParameterGenerator pg = new ParameterGenerator(n);
   List<Point> coordinates = pg.generateCoordinates();

   // 2. Execution

   // 2.1 Calculating the cost of complete graph
   Graph result0 = new Graph(coordinates);
   result0.makeItCompleteGraph();

   // 2.2 Calculating the cost of Heuristic 1 & Heuristic 2 algorithms
   Graph result1 = new HeuristicsAlgo1(coordinates).solve();
   Graph result2 = new HeuristicsAlgo2(coordinates).solve();

   // 3. Visualization
   String geoJson0 = new GeoJsonSerializer(result0).toJson();
   System.out.println(geoJson0);
   System.out.println("-----------");

   String geoJson1 = new GeoJsonSerializer(result1).toJson();
   System.out.println(geoJson1);
   System.out.println("-----------");
```

```
   String geoJson2 = new GeoJsonSerializer(result2).toJson();
   System.out.println(geoJson2);

   System.out.println("Complete Graph Cost: " + Math.round(result0.getCost()));
   System.out.println("Heuristic Cost 1: " + Math.round(result1.getCost()));
   System.out.println("Heuristic Cost 2: " + Math.round(result2.getCost()));
 }
}
```

2. ParameterGenerator.java

```java
package edu.utdallas.atn.p2.core.a_param_gen;

import edu.utdallas.atn.p2.domain.Point;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ThreadLocalRandom;

public class ParameterGenerator {

 private static final double MIN_LAT = 32.984100;
 private static final double MAX_LAT = 32.995995;

 private static final double MIN_LNG = -456.732388;
 private static final double MAX_LNG = -456.650677;

 private final int nodeCount;

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

 public List<Point> generateCoordinates() {

   List<Point> coordinates = new ArrayList<>();

   for (int i = 0; i < nodeCount; i++) {
     double lat = getRandomCoordinate(MIN_LAT, MAX_LAT);
     double lng = getRandomCoordinate(MIN_LNG, MAX_LNG);
     coordinates.add(new Point(lat, lng));
   }

   return coordinates;
 }

 private double getRandomCoordinate(double min, double max) {
   return ThreadLocalRandom.current().nextDouble(min, max);
 }
}
```

3. HeuristicsAlgo1.java

```
package edu.utdallas.atn.p2.core.b_algos;

import edu.utdallas.atn.p2.domain.Edge;
import edu.utdallas.atn.p2.domain.Graph;
import edu.utdallas.atn.p2.domain.Point;
import java.util.List;
import java.util.PriorityQueue;

public class HeuristicsAlgo1 {

 private Graph graph;
 private final PriorityQueue<Edge> edgesPq;

 public HeuristicsAlgo1(List<Point> coordinates) {
   this.graph = new Graph(coordinates);
   this.edgesPq = new PriorityQueue<>((a, b) -> Double.compare(b.getDistance(),
a.getDistance()));
 }

 public Graph solve() {
   // 1. make it a complete graph
   this.graph.makeItCompleteGraph();

   // 2. Push all the edges to the PQ
   this.edgesPq.addAll(this.graph.getEdges());

   int numberOfEdgeRemovals = 0;
   while (!edgesPq.isEmpty()) {

     Edge largestDistanceEdge = edgesPq.poll();
     Graph newGraph = new Graph(this.graph);
     newGraph.removeEdge(largestDistanceEdge);

     if (!newGraph.getDiameter().isPresent()
         || newGraph.getDiameter().get() > 4
         || newGraph.getSmallestDegree() < 3) break;

     numberOfEdgeRemovals++;
     this.graph = newGraph;
   }

   //    System.out.println(numberOfEdgeRemovals);
   return this.graph;
 }
}
```

4. HeuristicsAlgo2.java

```java
package edu.utdallas.atn.p2.core.b_algos;

import edu.utdallas.atn.p2.domain.Edge;
import edu.utdallas.atn.p2.domain.Graph;
import edu.utdallas.atn.p2.domain.Point;
import edu.utdallas.atn.p2.domain.Rectangle;

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

public class HeuristicsAlgo2 {
 private final List<Point> coordinates;

 public HeuristicsAlgo2(List<Point> coordinates) {
   this.coordinates = coordinates;
 }

 public Graph solve() {
   // 1. Create segments
   Rectangle[][] rectangles = generateSegments();

   // 2. Add Points to segments
   bucketPointsToSegments(rectangles);

   // 3. Convert Rectangle points to individual graphs
   Graph[][] segmentGraphs = generateSegmentGraphs(rectangles);

   // 4. Strongly connect individual segmentGraphs
   makeIndividualSegmentGraphsCompleteGraph(segmentGraphs);

   // 5. Add edge from each segment to the center segment
   Graph output = new Graph(coordinates);
   stronglyConnectAllSegments(segmentGraphs, output);

   // 6. Merge rest of the segments to the main graph
   mergeSegmentsToMainGraph(segmentGraphs, output);

   output.checkAndCorrectDegreeConstraint(3);

   return output;
 }
```

```
  private void mergeSegmentsToMainGraph(Graph[][] segmentGraphs, Graph output) {
    for (int r = 0; r < 3; r++) {
      for (int c = 0; c < 3; c++) {
        for (Edge edge : segmentGraphs[r][c].getEdges()) {
          output.addEdge(edge);
        }
      }
    }
  }

  private void stronglyConnectAllSegments(Graph[][] segmentGraphs, Graph output) {

    Optional<Point> middleSegmentCenter = segmentGraphs[1][1].getCenter();
    if (middleSegmentCenter.isPresent()) {
      for (int r = 0; r < 3; r++) {
        for (int c = 0; c < 3; c++) {
          if (r == 1 && c == 1) continue;
          Optional<Point> segmentCenter = segmentGraphs[r][c].getCenter();
          if (segmentCenter.isPresent())
            output.addEdge(new Edge(segmentCenter.get(),
middleSegmentCenter.get()));
        }
      }
    } else {
      List<Point> segmentCenters = new ArrayList<>();
      for (int r = 0; r < 3; r++) {
        for (int c = 0; c < 3; c++) {
          if (segmentGraphs[r][c].getCenter().isPresent())
            segmentCenters.add(segmentGraphs[r][c].getCenter().get());
        }
      }

      for (int i = 0; i < segmentCenters.size(); i++) {
        for (int j = 0; j < segmentCenters.size(); j++) {
          if (i == j) continue;
          output.addEdge(new Edge(segmentCenters.get(i), segmentCenters.get(j)));
        }
      }
    }
  }

  private void makeIndividualSegmentGraphsCompleteGraph(Graph[][] segmentGraphs) {
    for (int r = 0; r < 3; r++) {
      for (int c = 0; c < 3; c++) {
        segmentGraphs[r][c].makeItCompleteGraph();
      }
    }
```

```
}

private void bucketPointsToSegments(Rectangle[][] rectangles) {
  for (Point point : coordinates) {

    boolean found = false;
    for (int r = 0; r < 3; r++) {
      if (found) break;

      for (int c = 0; c < 3; c++) {
        if (found) break;
        if (rectangles[r][c].contains(point)) {
          rectangles[r][c].add(point);
          found = true;
        }
      }
    }
  }
}

private Graph[][] generateSegmentGraphs(Rectangle[][] segments) {
  Graph[][] segmentGraphs = new Graph[3][3];
  for (int r = 0; r < 3; r++) {
    for (int c = 0; c < 3; c++) {
      segmentGraphs[r][c] = segments[r][c].generateGraph();
    }
  }
  return segmentGraphs;
}

private Rectangle[][] generateSegments() {
  double maxLat = Integer.MIN_VALUE;
  double minLat = Integer.MAX_VALUE;

  double maxLng = Integer.MIN_VALUE;
  double minLng = Integer.MAX_VALUE;

  for (Point point : coordinates) {
    maxLat = Math.max(maxLat, point.getLat());
    minLat = Math.min(minLat, point.getLat());

    maxLng = Math.max(maxLng, point.getLng());
    minLng = Math.min(minLng, point.getLng());
  }

  // Adding an extra padding
  maxLat += 0.0005;
```

```
    minLat -= 0.0005;

    maxLng += 0.0005;
    minLng -= 0.0005;

    // 1. Generate Boundary Rectangle
    Point outerRectangleTopLeft = new Point(minLat, maxLng);
    Point outerRectangleTopRight = new Point(maxLat, maxLng);

    Point outerRectangleBottomLeft = new Point(minLat, minLng);
    Point outerRectangleBottomRight = new Point(maxLat, minLng);

    // 2. Find deltaX & deltaY
    double width = outerRectangleBottomLeft.getLat() -
outerRectangleBottomRight.getLat();
    double height = outerRectangleBottomLeft.getLng() -
outerRectangleTopLeft.getLng();

    double deltaLat = width / 3.0;
    double deltaLng = height / 3.0;

    // 3. Create segment corners
    Point[][] points = new Point[4][4];
    for (int r = 0; r < 4; r++) {
      for (int c = 0; c < 4; c++) {

        // TODO: This portion somehow works
        double lat = outerRectangleTopRight.getLat() + r * deltaLat;
        double lng = outerRectangleTopRight.getLng() + c * deltaLng;
        points[r][c] = new Point(lat, lng);
      }
    }

    // 4. Create segments
    Rectangle[][] segments = new Rectangle[3][3];
    for (int r = 0; r < 3; r++) {
      for (int c = 0; c < 3; c++) {
        Point bottomLeft = points[r + 1][c + 1];
        Point topRight = points[r][c];
        segments[r][c] = new Rectangle(bottomLeft, topRight);
      }
    }

    return segments;
  }
}
```

5. GeoJsonSerializer.java

```
package edu.utdallas.atn.p2.core.c_visualize;

import com.fasterxml.jackson.databind.ObjectMapper;
import edu.utdallas.atn.p2.domain.Edge;
import edu.utdallas.atn.p2.domain.Graph;
import edu.utdallas.atn.p2.domain.Point;
import org.geojson.Feature;
import org.geojson.FeatureCollection;
import org.geojson.LineString;
import org.geojson.LngLatAlt;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class GeoJsonSerializer {

  private final List<Point> points;
  private final List<Edge> edges;
  private final Map<String, Object> properties;

  public GeoJsonSerializer(List<Point> points, Map<String, Object>
properties) {
    this(points, new ArrayList<>(), properties);
  }

  public GeoJsonSerializer(Graph graph) {
    this(graph.getCoordinates(), graph.getEdges(), new HashMap<>());
  }

  public GeoJsonSerializer(Graph graph, Map<String, Object> properties) {
    this(graph.getCoordinates(), graph.getEdges(), properties);
  }

  public GeoJsonSerializer(
      List<Point> coordinates, List<Edge> edges, Map<String, Object>
properties) {
    this.points = coordinates;
    this.edges = edges;
```

```
    this.properties = properties;
  }

  public String toJson() {
    FeatureCollection featureCollection = new FeatureCollection();

    for (Point point : this.points) {
      Feature _point = new Feature();
      _point.setGeometry(new org.geojson.Point(point.getLng(),
point.getLat()));
      _point.setProperties(properties);
      _point.setProperty("marker-size", "small");
      featureCollection.add(_point);
    }

    for (Edge edge : edges) {
      Feature _line = new Feature();
      Point start = edge.getStart();
      Point end = edge.getEnd();

      _line.setGeometry(
          new LineString(
              new LngLatAlt(start.getLng(), start.getLat()),
              new LngLatAlt(end.getLng(), end.getLat())));

      featureCollection.add(_line);
    }

    String json = "{}";
    try {
      json = new ObjectMapper().writeValueAsString(featureCollection);
    } catch (Exception ex) {
      ex.printStackTrace();
    }

    return json;
  }
}
```

6. Edge.java

```
package edu.utdallas.atn.p2.domain;

public class Edge {
 Point start;
 Point end;

 public Edge(Point start, Point end) {
   this.start = start;
   this.end = end;
 }

 public Point getStart() {
   return start;
 }

 public Point getEnd() {
   return end;
 }

 public Double getDistance() {
   return start.distanceTo(end);
 }
}
```

7. Graph.java

```
package edu.utdallas.atn.p2.domain;

import java.util.*;

public class Graph {

 private final Map<Point, Integer> index;
 private final Map<Integer, Point> reverseIndex;

 private final int n;
 private final boolean[][] connectivityAdjMatrix;

 public Graph(Graph graph) {
   this.n = graph.n;

   // Deep Copy, Map
   this.index = new HashMap<>(graph.index);
   this.reverseIndex = new HashMap<>(graph.reverseIndex);

   // Deep Copy, 2D Array
   this.connectivityAdjMatrix = new boolean[n][n];
   for (int r = 0; r < n; r++)
     this.connectivityAdjMatrix[r] = Arrays.copyOf(graph.connectivityAdjMatrix[r],
n);
 }

 public Graph(List<Point> points) {

   this.index = new HashMap<>();
   int counter = 0;
   for (Point point : points) this.index.put(point, counter++);

   this.reverseIndex = new HashMap<>();
   this.index.forEach((k, v) -> reverseIndex.put(v, k));

   this.n = index.size();
   this.connectivityAdjMatrix = new boolean[n][n];
 }

 public void addEdge(Edge edge) {
   Point start = edge.getStart();
   Point end = edge.getEnd();

   int startIndex = index.get(start);
   int endIndex = index.get(end);
```

```
  connectivityAdjMatrix[startIndex][endIndex] = true;
  connectivityAdjMatrix[endIndex][startIndex] = true;
}

public void removeEdge(Edge edge) {
  Point start = edge.getStart();
  Point end = edge.getEnd();

  int startIndex = index.get(start);
  int endIndex = index.get(end);

  connectivityAdjMatrix[startIndex][endIndex] = false;
  connectivityAdjMatrix[endIndex][startIndex] = false;
}

public void makeItCompleteGraph() {
  for (int r = 0; r < n; r++) {
    for (int c = 0; c < n; c++) {
      connectivityAdjMatrix[r][c] = true;
    }
  }
}

public int getSmallestDegree() {
  int minDegree = Integer.MAX_VALUE;
  for (int r = 0; r < n; r++) {
    int degree = 0;
    for (int c = 0; c < n; c++) {
      if (r == c) continue;
      if (connectivityAdjMatrix[r][c]) degree++;
    }
    minDegree = Math.min(degree, minDegree);
  }
  return minDegree;
}

public Optional<Integer> getDiameter() {

  // 1. create adjMatrix
  int[][] adjMatrix = new int[n][n];
  for (int r = 0; r < n; r++)
    for (int c = 0; c < n; c++) {
      if (r == c) adjMatrix[r][c] = 0;
      else if (connectivityAdjMatrix[r][c]) adjMatrix[r][c] = 1;
      else adjMatrix[r][c] = 1000;
    }
```

```
    // 2. run floyd-warshall to get the max hop-count from a vertex to every other
vertex.
    for (int k = 0; k < n; k++) {
      for (int r = 0; r < n; r++) {
        for (int c = 0; c < n; c++) {
          adjMatrix[r][c] = Math.min(adjMatrix[r][k] + adjMatrix[k][c],
adjMatrix[r][c]);
        }
      }
    }

    // 3. find the largest hop distance.
    int largestHopDistance = 0;
    for (int r = 0; r < n; r++) {
      for (int c = 0; c < n; c++) {

        // Checks if all the nodes are connected in the graph.
        // If not, there is no point in getting the largest Hop Distance.
        if (adjMatrix[r][c] >= 1000) return Optional.empty();

        // finding the maximum
        largestHopDistance = Math.max(largestHopDistance, adjMatrix[r][c]);
      }
    }

    return Optional.of(largestHopDistance);
  }

  public List<Point> getCoordinates() {
    return new ArrayList<>(index.keySet());
  }

  public Optional<Point> getCenter() {
    return getCoordinates().size() == 0 ? Optional.empty() :
Optional.of(getCoordinates().get(0));
  }

  public List<Edge> getEdges() {
    List<Edge> edges = new ArrayList<>();
    for (int r = 0; r < n; r++) {
      for (int c = r + 1; c < n; c++) {
        if (!connectivityAdjMatrix[r][c]) continue;

        Point start = reverseIndex.get(r);
        Point end = reverseIndex.get(c);
```

```
        edges.add(new Edge(start, end));
      }
    }

    return edges;
  }

  public void checkAndCorrectDegreeConstraint(int expectedDegree) {
    for (int r = 0; r < n; r++) {
      int actualDegree = 0;
      for (int c = 0; c < n; c++) {
        if (r == c) continue;
        if (connectivityAdjMatrix[r][c]) actualDegree++;
      }
      if (actualDegree < expectedDegree) improviseDegreeTo(r, expectedDegree -
actualDegree);
    }
  }

  private void improviseDegreeTo(int nodeIndex, int diff) {
    // 1. Get current Neighbours (to ensure that you don't re-add them.
    List<Point> currentNeighbours = new ArrayList<>();
    for (int c = 0; c < n; c++)
      if (connectivityAdjMatrix[nodeIndex][c])
currentNeighbours.add(reverseIndex.get(c));

    // 2. Find the closest neighbour
    Point root = reverseIndex.get(nodeIndex);
    Queue<Point> neighbours = new ArrayDeque<>(closestNeighbours(root));
    neighbours.removeAll(currentNeighbours);

    // 3. attach to the nearest feasible neighbour
    while (diff > 0) {
      addEdge(new Edge(root, neighbours.poll()));
      diff--;
    }
  }

  public List<Point> closestNeighbours(Point point) {
    List<Point> result = new ArrayList<>(index.keySet());
    result.remove(point);

    Collections.sort(result, (a, b) -> Double.compare(a.distanceTo(point),
b.distanceTo(point)));
    return result;
  }
```

```
 public Double getCost() {
   double total = 0;
   for (Edge edge : getEdges()) {
     total += edge.getDistance();
   }

   return total;
 }
}
```

8. Point.java

```
package edu.utdallas.atn.p2.domain;

import org.apache.lucene.util.SloppyMath;

import java.util.Objects;

public class Point {
 private final double lat;
 private final double lng;

 public Point(double lat, double lng) {
   this.lat = lat;
   this.lng = lng;
 }

 public double getLat() {
   return lat;
 }

 public double getLng() {
   return lng;
 }

 public double distanceTo(Point end) {
   return SloppyMath.haversinMeters(this.getLat(), this.getLng(), end.getLat(),
end.getLng());
 }

 @Override
 public boolean equals(Object o) {
   if (this == o) return true;
   if (!(o instanceof Point)) return false;
   Point point = (Point) o;
   return Double.compare(point.getLat(), getLat()) == 0
       && Double.compare(point.getLng(), getLng()) == 0;
 }

 @Override
 public int hashCode() {
   return Objects.hash(getLat(), getLng());
 }
}
```

9. Rectangle.java

```
package edu.utdallas.atn.p2.domain;

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

public class Rectangle {

 private final Point southWest;
 private final Point northEast;

 private final List<Point> points;

 public Rectangle(Point southWest, Point northEast) {
   this.southWest = southWest;
   this.northEast = northEast;
   this.points = new ArrayList<>();
 }

 public boolean contains(Point point) {

   double swLatitude = southWest.getLat();
   double swLongitude = southWest.getLng();
   double neLatitude = northEast.getLat();
   double neLongitude = northEast.getLng();
   double latitude = point.getLat();
   double longitude = point.getLng();

   boolean longitudeContained = false;
   boolean latitudeContained = false;

   // Check if the bbox contains the prime meridian (longitude 0.0).
   if (swLongitude < neLongitude) {
     if (swLongitude < longitude && longitude < neLongitude) {
       longitudeContained = true;
     }
   } else {
     // Contains prime meridian.
     if ((0 < longitude && longitude < neLongitude)
         || (swLongitude < longitude && longitude < 0)) {
       longitudeContained = true;
     }
   }

   if (swLatitude < neLatitude) {
     if (swLatitude < latitude && latitude < neLatitude) {
```

```
        latitudeContained = true;
      }
    } else {
      // The poles. Don't care.
    }

    return (longitudeContained && latitudeContained);
  }

  public void add(Point point) {
    this.points.add(point);
  }

  public Graph generateGraph() {
    return new Graph(this.points);
  }
}
```

10. 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">
    <parent>
        <artifactId>atn-projects</artifactId>
        <groupId>edu.utdallas</groupId>
        <version>1.0-SNAPSHOT</version>
    </parent>
    <modelVersion>4.0.0</modelVersion>

    <artifactId>atn-project2</artifactId>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
    </properties>

    <dependencies>
        <dependency>
            <groupId>de.grundid.opendatalab</groupId>
            <artifactId>geojson-jackson</artifactId>
            <version>1.14</version>
        </dependency>

        <dependency>
            <groupId>org.apache.lucene</groupId>
            <artifactId>lucene-spatial</artifactId>
            <version>8.2.0</version>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.13.2</version>
            <scope>test</scope>
        </dependency>


    </dependencies>

</project>
```

7. Bibliography

1. Computational Geometry: https://catalog.utdallas.edu/2021/graduate/courses/cs6319

2. Design & Analysis of Algo: https://catalog.utdallas.edu/2017/graduate/courses/cs6363

3. Distributed Computing: https://catalog.utdallas.edu/2020/graduate/courses/cs6380

4. Spatial RDD: https://github.com/utd-db/spatial-rdd

5. Geo-Hash: https://www.movable-type.co.uk/scripts/geohash.html

6. GeoJSON editor: https://geojson.io/

7. Hub-Spoke Model: https://en.wikipedia.org/wiki/Spoke%E2%80%93hub_distribution_paradigm

8. Floyd Warshall: https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm

9. Lucene Spatial: https://mvnrepository.com/artifact/org.apache.lucene/lucene-spatial/8.2.0

10. JUnit: https://mvnrepository.com/artifact/junit/junit/4.10

11. GeoJSON Jackson: https://github.com/opendatalab-de/geojson-jackson

12. GeoJSON: https://geojson.org/

13. Stackoverflow Solution for checking if point lies within a rectangle: https://stackoverflow.com/questions/48077546/find-if-given-lat-lng-co-ordinates-lies-inside-a-square-rectangle