Algorithmic Aspects of Telecommunication Networks

Project 3

Kevin Chen
E-mail: nkc160130@utdallas.edu

Table of Contents

Overview

The goal of this project is to both create and implement two different heuristic algorithms for a network topology design problem, and experiment with it. The set of nodes of the graph are on a plane, of which there are n of them. The network topology generated has the following properties:

1. The graph is connected, i.e. it contains all the given nodes.
2. The degree of each vertex in the graph is at least 3, i.e. each node is connected to 3 other nodes
3. The diameter of the graph is at most 4, i.e. any node can be reached from any other node in at most 4 hops. Note that this does not depend on the geometric distance.
4. The total cost of the network topology is as low as possible. This total cost is calculated by the total geometric length of all the links.

The heuristic algorithm is only used to satisfy the fourth property. All the other properties are considered constraints on the graph; that is, the resulting network topology must have those properties. The algorithms used to satisfy the fourth property are of choice, from either general-purpose heuristic optimization algorithms such as Simulated Annealing, Tabu Search or Genetic Algorithm.

Algorithms

Worst to Best Algorithm

This is an algorithm that I came up with and is basically a greedy algorithm. The premise is simple; for every vertex, take the edge with the highest cost, create a list of all possible edges (leave out edges that are already on the graph) and iterate through them, least to greatest. Then, either pick an edge that has a lower cost and still satisfies the constraints or leave the edge as is.

Genetic Algorithm

This is an algorithm based upon the principle of reproduction and natural selection in biology. There is an initial population, which, in this case, is several individual graphs which satisfy constraints 1-3 from above. I start the population with 32 individuals. Then, iterating through a user-specified number of generations, it will produce candidates for the next generation. First, from the current generation (which is initially the initial population), the top 5% least cost graphs automatically get to go to the next generation. With a user-specified probability, an individual will be chosen to have a mutated child; in this case, removing a random number of edges and replacing them with the same number of random edges. The rest of the individuals will be used to create cross-over children: that is, a mother and father graph's edges will be split in a random place, then recombined to form a child graph. If any of these generated children no longer satisfied the constraints, they are automatically not considered to be candidates for the next generation. The next generation is then determined by taking the top candidates, sorted in order by total cost; the size of the next generation should be no more than twice the size of the initial population. Then, the best child of the last generation will be returned as the solution.

Implementation

This software was coded with Kotlin, which is a general-purpose programming language designed to maintain full compatibility and interoperability with Java code, while simplifying Java syntax, especially with regards to typing. A tool used in this project is Maven, for build automation in the project and integrating external libraries. JGraphT [1] is used to render the network topology. To render the graphs, I outputted the graphs into a text file format, read them in with Python, and used Matplotlib [2] to render the network topology graphically.

The pseudocode for the Worst to Best Algorithm is as follows:

```
current_graph := min_constraint_graph
candidate_graph := min_constraint_graph

for vertex in min_constraint_graph:
    current_vertex_edges := vertex.get_all_edges()
    current_vertex_edges.sort()
    worst_edge := current_vertex_edges.last()

    all_possible_vertex_edges := vertex.get_all_possible_edges()
    all_possible_vertex_edges.sort()

    for edge in all_possible_vertex_edges:
        if edge.cost > worst_edge.cost:
            candidate_graph := current_graph
            break

        candidate_graph.remove_from_graph(worst_edge)
        candidate_graph.add_to_graph(edge)

        if candidate_graph.satisfies_all_constraints():
            current_graph := candidate_graph
            break

return current_graph
```

Figure 1. Pseudocode for Worst to Best Algorithm

The pseudocode for the Genetic Algorithm is as follows:

```
initial_population := generate_random_graphs(n, population_size)
current_generation := initial_population

mutation_chance := 0.8
random_graph_vertices := initial_population[0].vertices

for i in number_of_generations:
    next_generation := new_list()
    current_generation.sort()

    # Getting the elite and adding to next generation
    elite_children := current_generation.sublist(0, current_generation.size * 0.0
5)
    next_generation.add(elite_children)
    current_generation.remove(elite_children)

    # Getting the mutation child, if applicable
    mutation_child := random_graph_vertices
    mutation_child_index := -1
    if get_random_double(0, 1) < mutation_chance:
        mutation_child_index := get_random_int(0, current_generation.size)
        mutation_child := current_generation[mutation_child_index]

        remove_random_edges := get_random_edges(current_generation[mutation_child
_index])
        mutation_child.remove_edges(remove_random_edges)

        add_random_edges := gen_random_edges(current_generation[mutation_child_in
dex])
        mutation_child.add_edges(add_random_edges)

        current_generation.remove(current_generation[mutation_child_index])

    # Getting the crossover children
    crossover_children := new_list()
    for mother in current_generation:
        for father in current_generation:
            if mother != father:
                crossover_child := mother.vertices
                splitPoint := get_random_double(0, 1)
                child_mother_edges := mother.edgeList.sublist(0, mother.edgeList.
size * splitPoint)
```

```
                child_father_edges := father.edgeList.sublist(father.edgeList.siz
e * splitPoint, father.edgeList.size)
                child_edges := append(child_mother_edges, child_father_edges)
                for edge in child_edges:
                    crossover_child.add_edges(edge)
                crossover_children.add(crossover_child)

    # Adding to next generation list, if the child satisfies all constraints
    if mutation_child.satisfies_all_constraints():
        next_generation.add(mutation_child)
    for child in crossover_children:
        if child.satisfies_all_constraints():
            next_generation.add(child)

    next_generation.sort()
    current_generation.clear()

    # Choosing what next generations to include in the next generation (from lowe
st to highest total cost)
    # Max size of the current generation is 2 * initial population size
    for individual in next_generation:
        current_generation.add(individual)

        if current_generation.size > initial_population.size:
            break

current_generation.sort()
return current_generation[0]
```

Figure 2. Pseudocode for Genetic Algorithm

Observations

Example 1: n = 15

- Worst to Best Algorithm

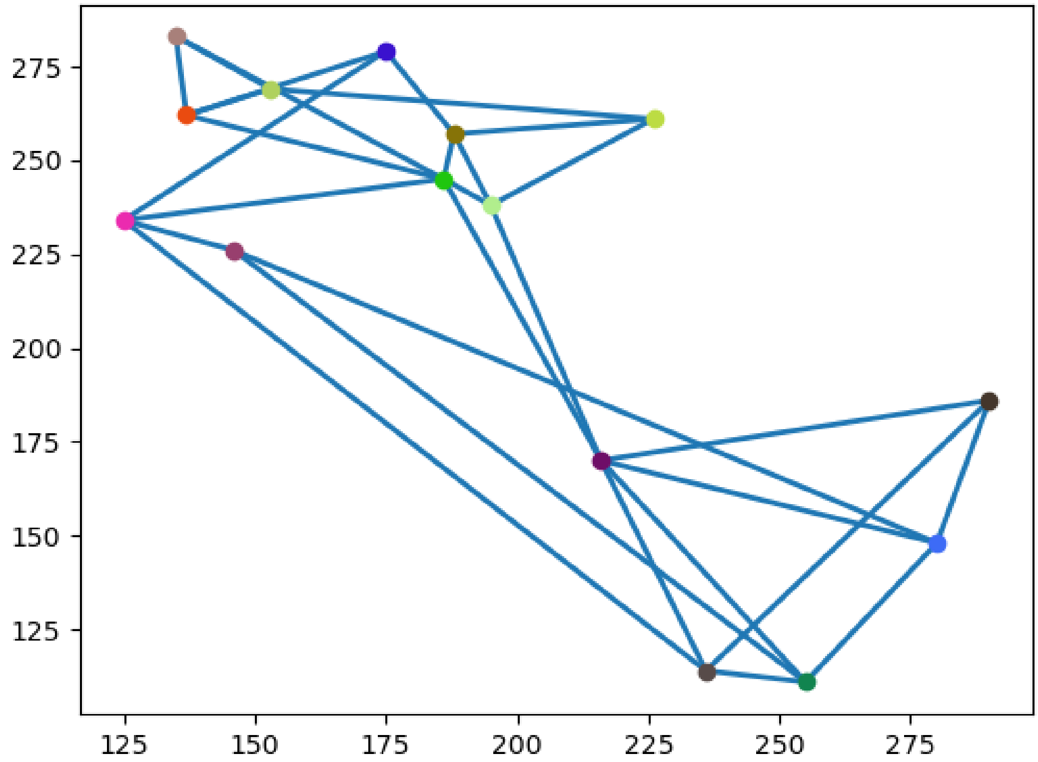

Figure 3. Worst-to-best algorithm network topology on Example 1.

  - Total cost: 1685.47
  - Run time: 0.01 seconds

- Genetic Algorithm

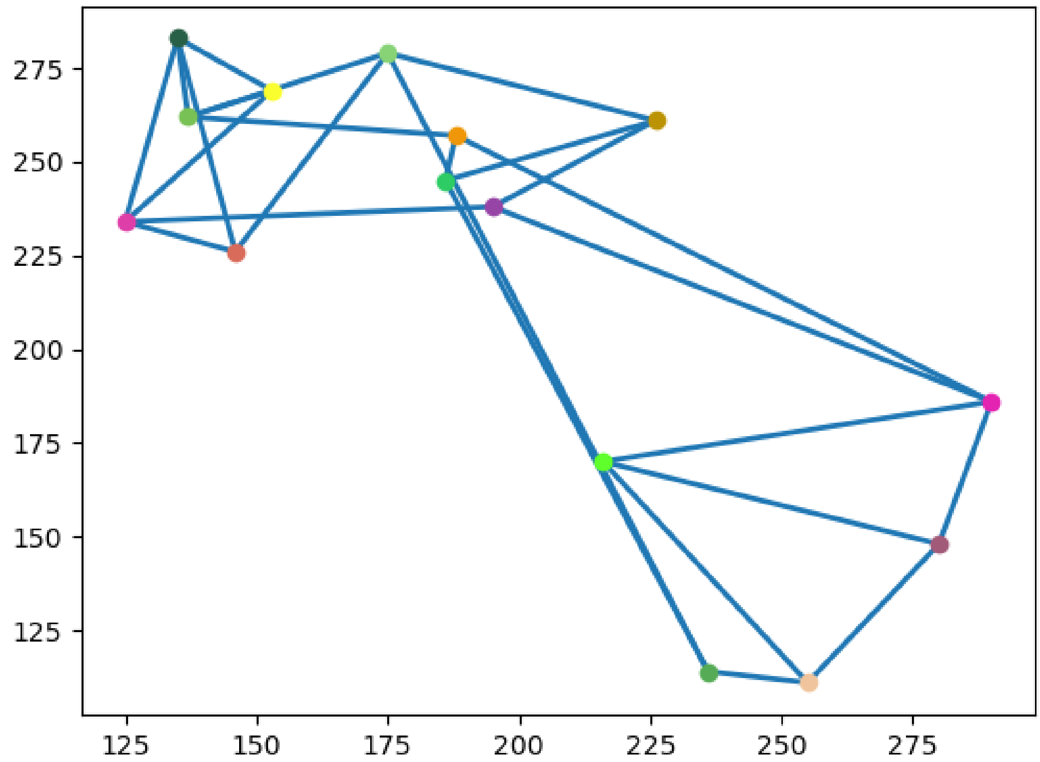

Figure 4. Genetic Algorithm network topology on Example 1.

- ○ Total cost: 1474.06
- ○ Run time: 59.44 seconds

Example 2: n = 20

- Worst to Best Algorithm

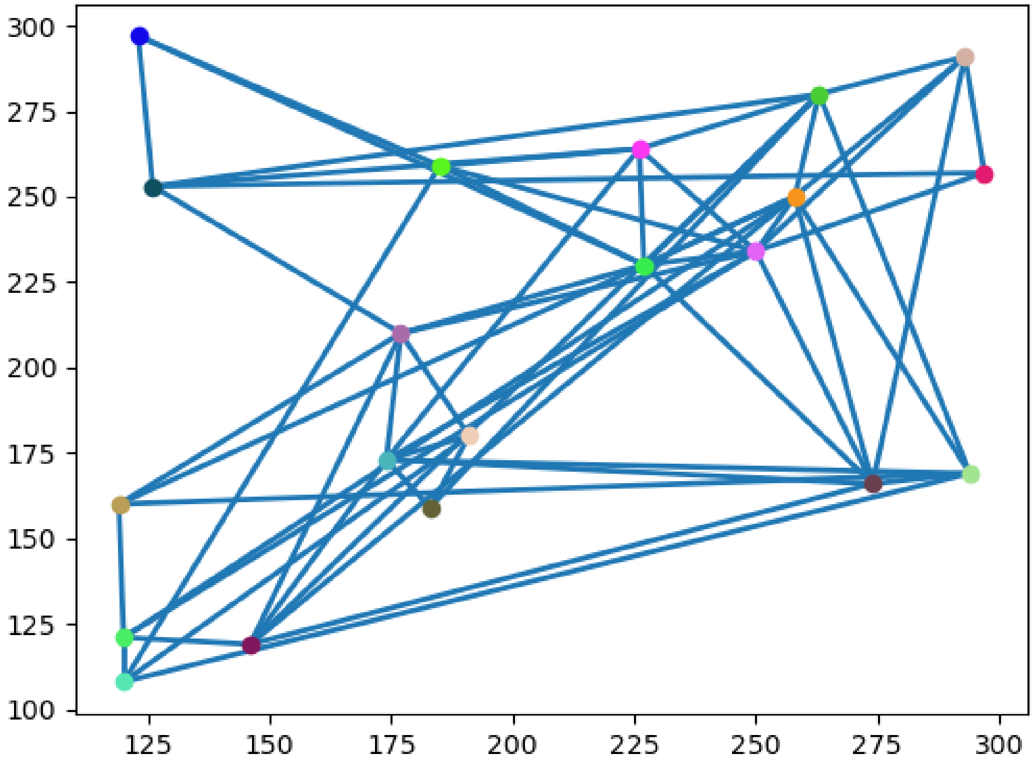

Figure 5. Worst to Best Algorithm network topology on Example 2.

- Total cost: 4792.09
- Run time: 0.022 seconds

- Genetic Algorithm

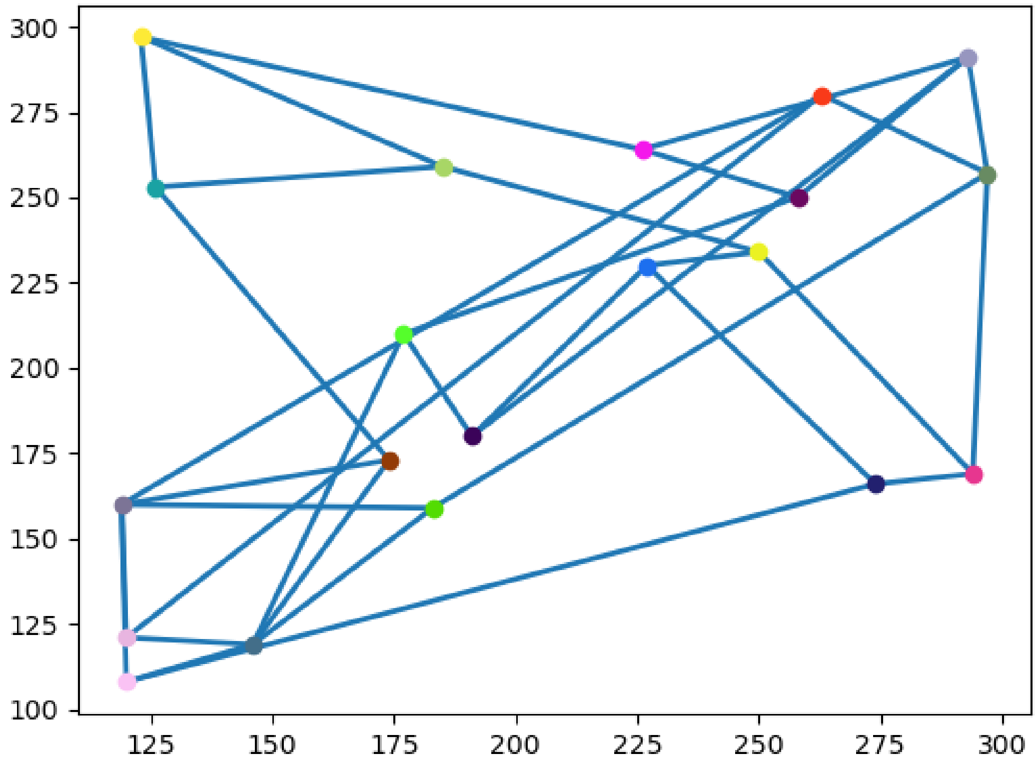

Figure 6. Genetic Algorithm network topology on Example 2.

- Total cost: 2502.38
- Run time: 124.78 seconds

Example 3: n = 25

- Worst to Best Algorithm

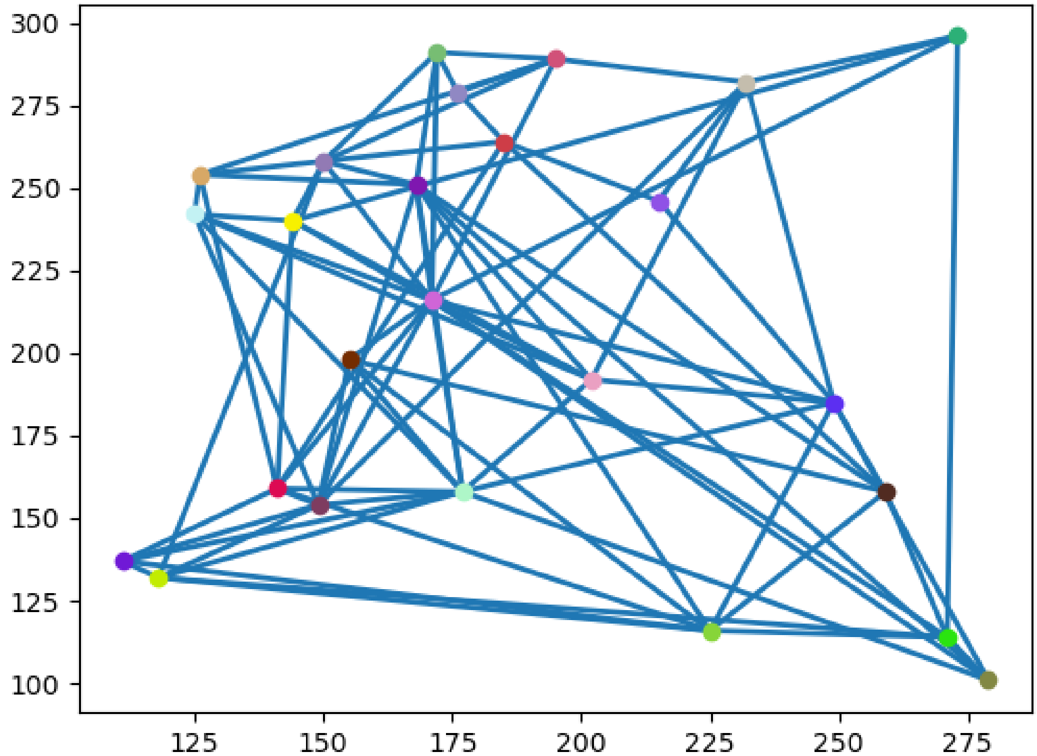

Figure 7. Worst to Best Algorithm network topology on Example 3.

- Total cost: 5822.27
- Run time: 0.033 seconds

- Genetic Algorithm

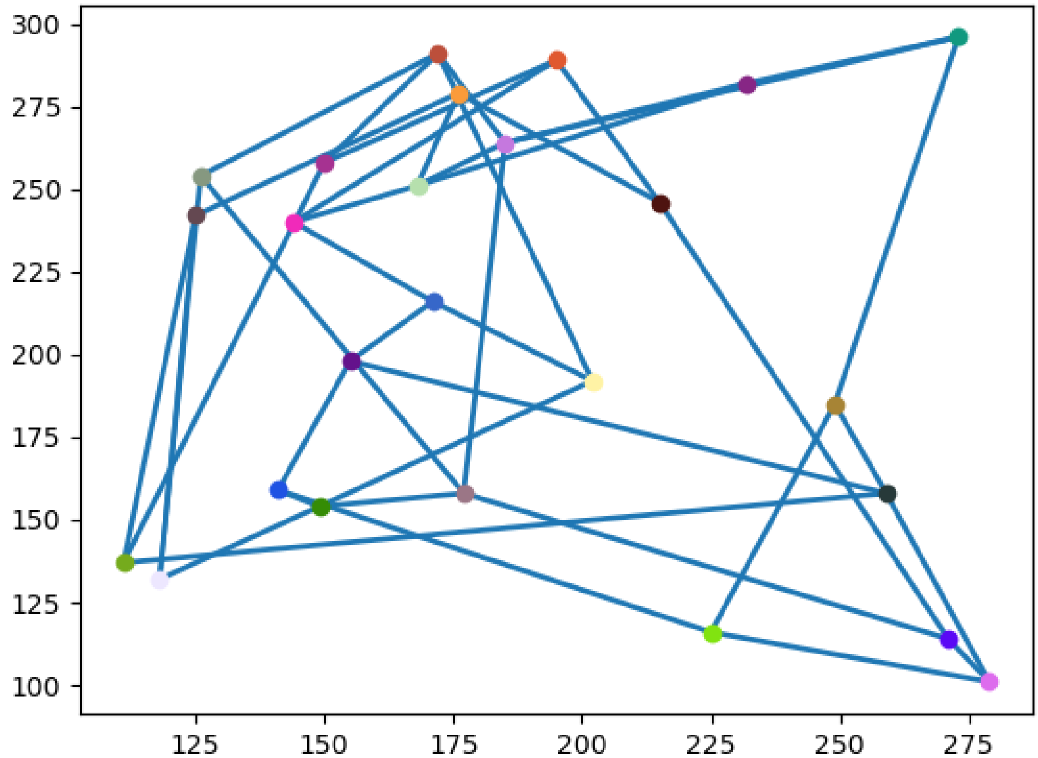

Figure 8. Genetic Algorithm network topology on Example 3.

- Total cost: 2766.25
- Run time: 208.47 seconds

Example 4: n = 30

- Worst to Best Algorithm

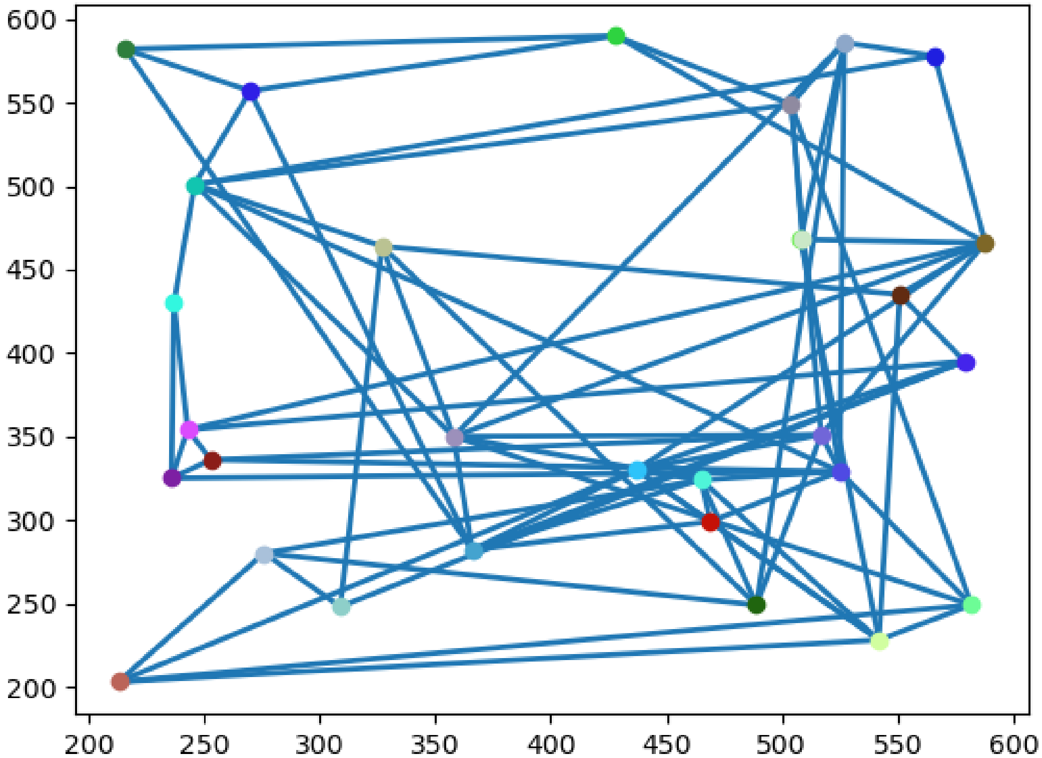

Figure 9. Worst to Best Algorithm network topology for Example 4.

- Total cost: 12568.92
- Run time: 0.069 seconds

- Genetic Algorithm

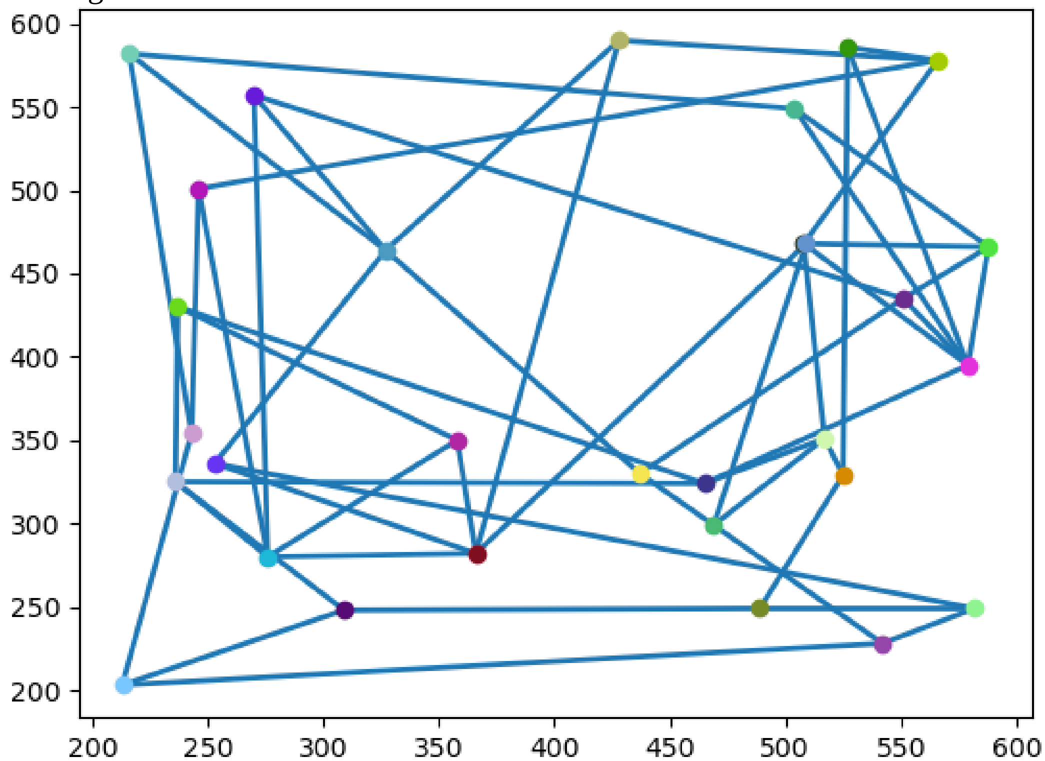

Figure 10. Genetic Algorithm network topology for Example 4.

- Total cost: 8005.16
- Run time: 304.38 seconds

Example 5: n = 35

- Worst to Best Algorithm

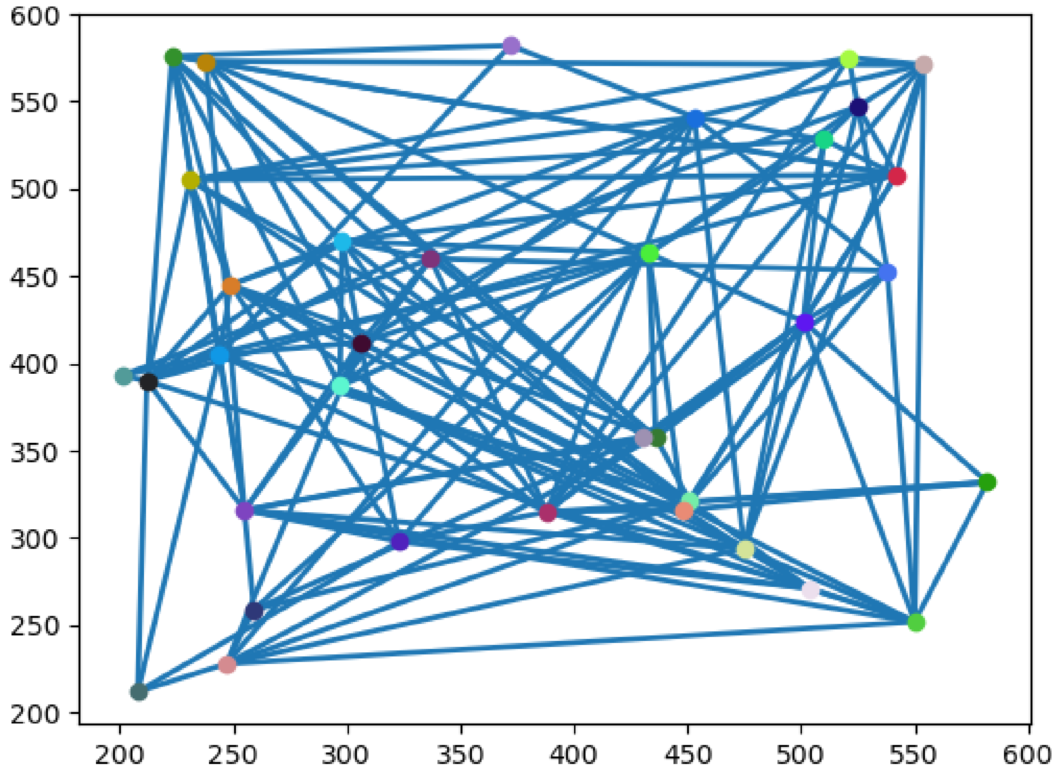

Figure 11. Worst to Best Algorithm network topology on Example 5.

- Total cost: 24508.96
- Run time: 0.085 seconds

- Genetic Algorithm

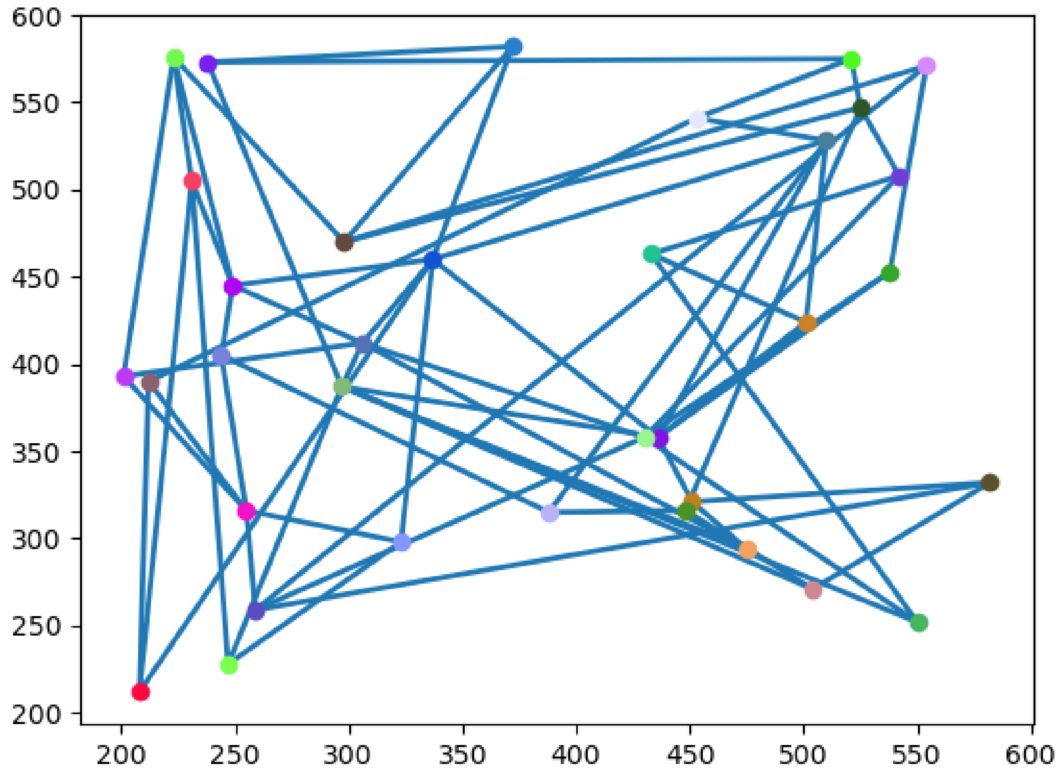

Figure 12. Genetic Algorithm network topology on Example 5.

- Total cost: 9865.42
- Run time: 489.65 seconds

Generation of Random Input

As shown in the diagrams, the vertices used for the network topology for both algorithms are the same. To generate the vertices randomly, a class called `HeuristicNetworkTop` takes in a n value, and another class called `Coordinate2D`, the vertex type of the graph, has a method which will generate random points in a certain specified range, and these random points are then added to the network.

```
nodes := new_graph()

for i in 0 until n:
    new_coordinate := Coordinate2D.generate_random_coordinate(min, max)
    while new_coordinate in nodes.vertices:
        new_coordinate := Coordinate2D.generate_random_coordinate(min, max)

    nodes.add(new_coordinate)

return nodes
```

Figure 13. Pseudocode of random points generation

Also, to create starter graphs that will be optimized by the algorithm, the `HeuristicNetworkTop` class has a method to generate a new minimum constraint graph. This method will essentially generate new random edges until the graph satisfies all constraints, minus the total cost constraint.

```
min_constraint_graph := new_graph()
min_constraint_graph.add_vertices(nodes)

while not min_constraint_graph.is_connected or min_constraint_graph.min_degree <
3 or min_constraint_graph.diameter > 4:
    source_vertex := min_constraint_graph.get_random_vertex()
    destination_vertex := min_constraint_graph.get_random_vertex()
    while source_vertex == destination_vertex:
        destination_vertex := min_constraint_graph.get_random_node()

    random_edge := min_constraint_graph.add_new_edge()

return min_constraint_graph
```

Figure 14. Pseudocode of minimum constraint graph generation

Conclusion

From these five examples, and ten graphs, the Genetic Algorithm consistently outperforms the Worst to Best Algorithm in terms of achieving an optimal result. The only time that the Worst to Best Algorithm will outperform the Genetic Algorithm is if the former algorithm gets lucky and gets a graph with a lower cost to start than the Genetic Algorithm's entire initial population of minimum constraint graphs. The difference between the Worst to Best Algorithm and the Genetic Algorithm in terms of total cost of the optimal network topology ranges from 211.41 for $n = 15$ to 14643.54 for $n = 35$. The total cost for the optimal graphs generally increases with the value of n for both algorithms. This is shown in Figures 16 and 17. In terms of run time, however, the Worst to Best Algorithm performs faster than the Genetic Algorithm in all cases. As with the total cost, the difference in run time increases with the value of n for both but changes much more drastically with the Genetic Algorithm. This can be seen in Figures 18 and 19. The Genetic Algorithm takes more time to run because it does much more data processing; it has 32 minimum constraint graphs to start with, and in terms of data processing, it needs to generate mutations and crossovers, which are memory intensive tasks. The Genetic Algorithm must also be repeated for multiple generations, while the Worst to Best Algorithm is only going through all the vertices in the graph. Since the Genetic Algorithm has more data to work with, however, it outperforms the Worst to Best Algorithm in terms of lowest total cost on average. As the Genetic Algorithm is iterative, even with mutations, it will eventually reach a point where the network topology cannot be optimized further. Figure 15 shows how the best individual will eventually stabilize after several generations. In this case, for $n = 15$, it seems that the algorithm will stabilize after approximately 14 generations.

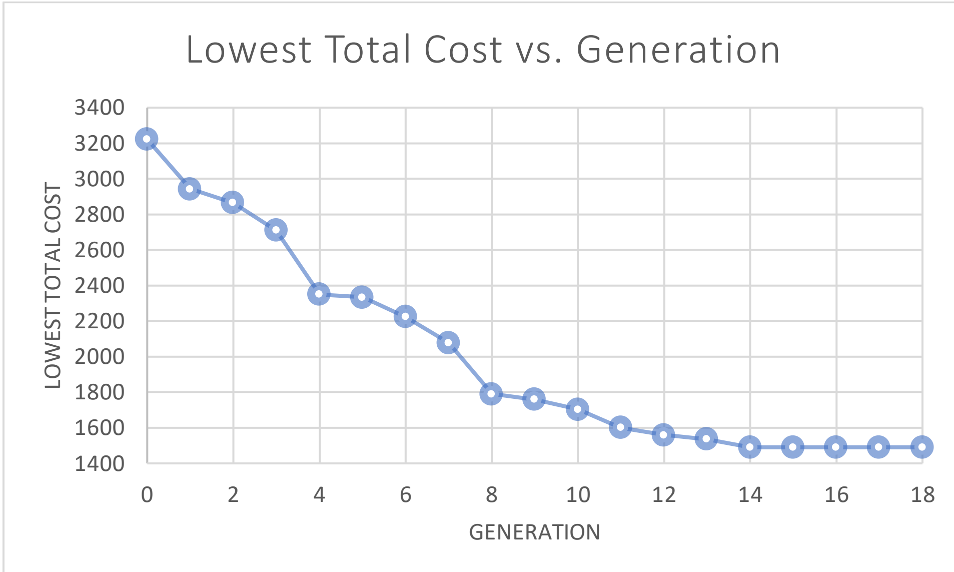

Figure 15. A graph that shows the lowest total cost for the fittest network topology for each generation. This graph was generated with a graph that has n = 15.

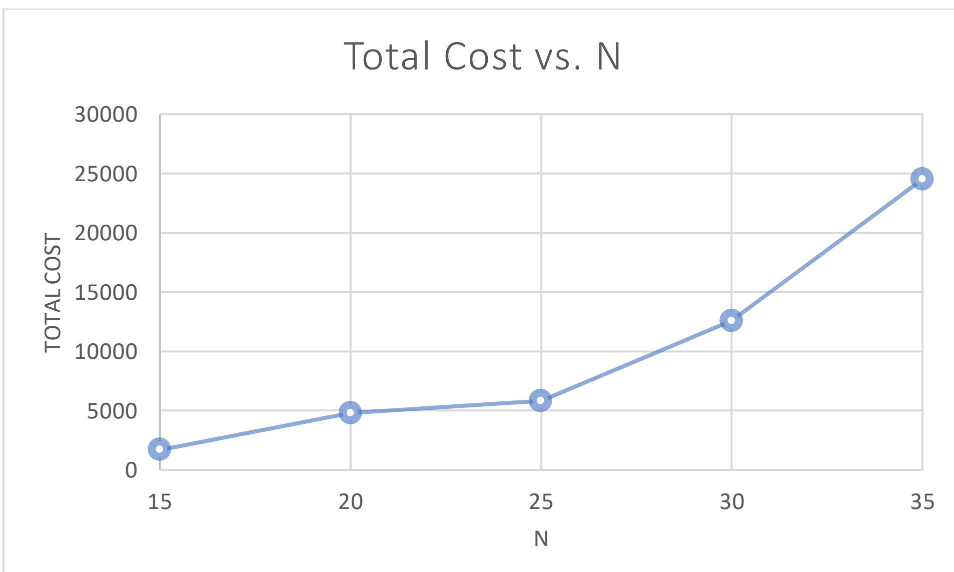

Figure 16. This graph shows the Worst to Best Algorithm's best graph's total cost vs. n.

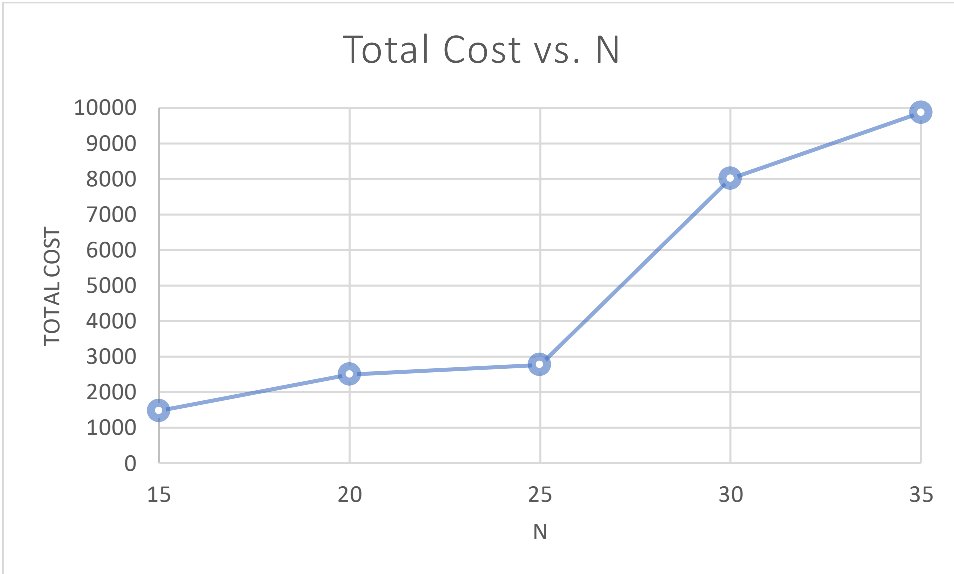

Figure 17. This graph shows the Genetic Algorithm's best graph's total cost vs. n.

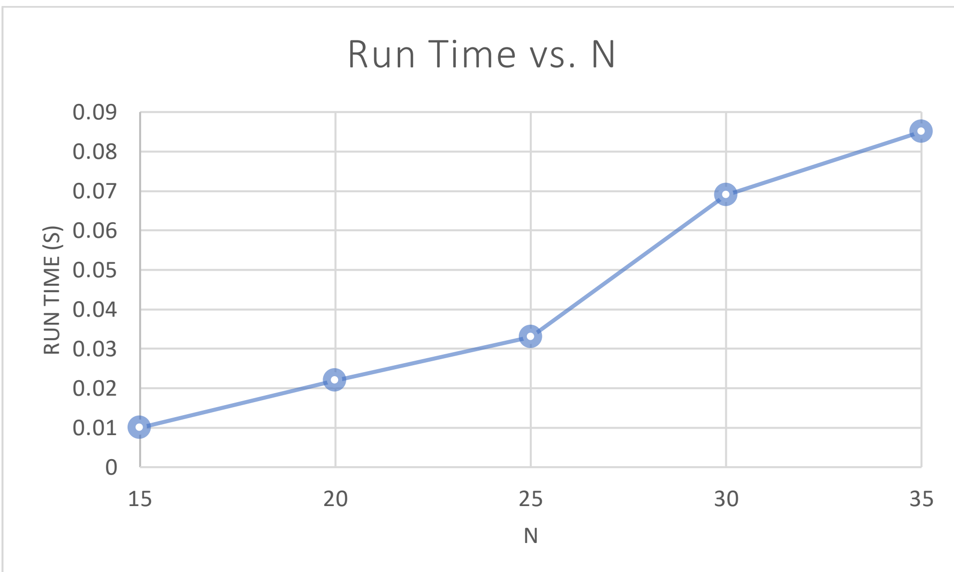

Figure 18. This graph shows the change in run time for the Worst to Best Algorithm with a change in n.

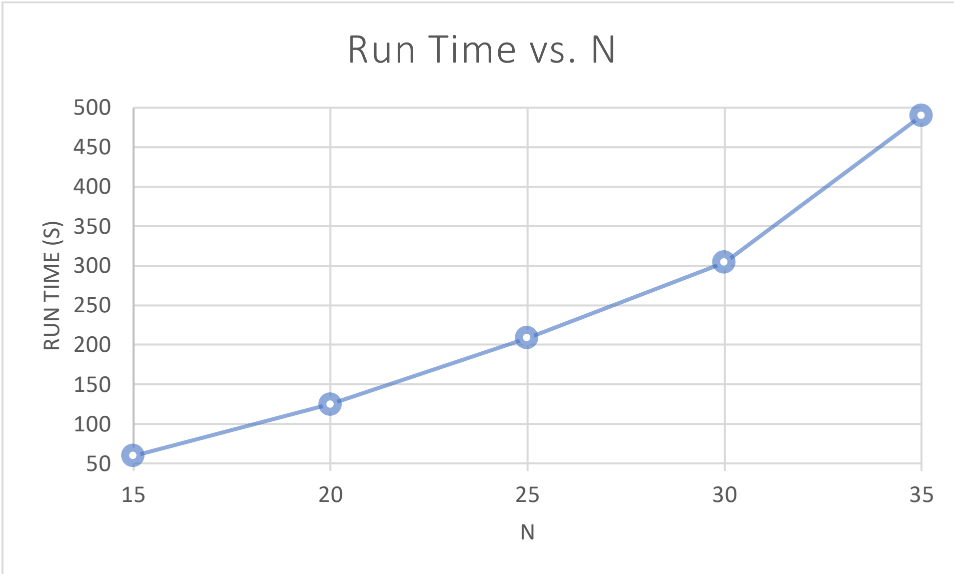

Figure 19. This graph shows the change in run time for the Genetic Algorithm with a change in n.

Readme

Instructions on how to run Project 3:

Note: These instructions assume that you already have Maven and Java 11 installed. If not, please install these first before following these steps.

1. Create a new folder, with a descriptive name (e.g. project-3) and go to this directory.
2. Copy the `pom.xml` to the root of this folder.
3. Create new folders within this folder, with this structure:
   `src/main/kotlin/com/cs6385`
4. Copy the `HeuristicNetworkTop.kt`, `Runner.kt`, `WorstToBestAlg.kt`, `GeneticAlg.kt` and `Coordinate2D.kt` files to this folder.
5. Go back to the root folder `project-3` folder and run
   ```
   mvn clean package
   ```
   in the terminal.
6. A new `target` folder should have been created after the build is successful. Go into this target folder and run
   ```
   java -jar project3-1.0-SNAPSHOT.jar <nValue>
   ```

Project Files

Runner.kt

```
package com.cs6385

import org.jgrapht.graph.DefaultUndirectedWeightedGraph
import org.jgrapht.graph.DefaultWeightedEdge
import java.io.File
import java.lang.RuntimeException

fun main(args: Array<String>) {
    if (args.size != 1) throw RuntimeException("ERROR: Not enough arguments,
expected 1: n")

    if (args[0].toIntOrNull() == null) throw RuntimeException("ERROR:
Parameter n needs to be an integer")

    if (args[0].toInt() < 15 || args[0].toInt() > 100) throw RuntimeException(
        "ERROR: Parameter n is either too small or too large! Try a value
between 15 and 100."
    )

    val networkTop = HeuristicNetworkTop(args[0].toInt())

    // Getting initial population
    var initialPopulation =
hashMapOf<DefaultUndirectedWeightedGraph<Coordinate2D, DefaultWeightedEdge>,
Double>()
    for (i in 0 until 32) {
        val curGraph = networkTop.getMinConstraintGraph()
        initialPopulation[curGraph] =
HeuristicNetworkTop.getTotalCost(curGraph)
    }

    initialPopulation = initialPopulation.toList().sortedBy { (_, value) ->
value }
        .toMap() as HashMap<DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge>, Double>

    var startTime = System.currentTimeMillis()
    val worstToBestInitialGraph = networkTop.getMinConstraintGraph()
    print("Worst to best algorithm:\nInitial graph: ")
    println("${HeuristicNetworkTop.getTotalCost(worstToBestInitialGraph)}")
    // Running worst to best algorithm
    val worstToBestAlg = WorstToBestAlg(worstToBestInitialGraph).run()
    println("${HeuristicNetworkTop.getTotalCost(worstToBestAlg)}\n")

    var output = "Vertices:\n"
    for (vertex in worstToBestAlg.vertexSet()) {
        output += vertex.toString() + "\n"
```

```
    }
    output += "Edges:\n"
    for (edge in worstToBestAlg.edgeSet()) {
        output += edge.toString() + "\n"
    }
    println("Run time: ${(System.currentTimeMillis() - startTime) / 1000.0}
seconds\n\n")

    File("worstToBestAlgorithmGraph.txt").writeText(output)


    startTime = System.currentTimeMillis()
    print("Genetic algorithm:\nInitial population: ")
    for (individual in initialPopulation) {
        print("${initialPopulation[individual.key]} ")
    }
    println()
    // Running genetic algorithm
    val genAlg = GeneticAlg(
        initialPopulation.keys.toList() as
MutableList<DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge>>,
        0.8
    ).run(200)
    println(genAlg?.let(HeuristicNetworkTop.Companion::getTotalCost))

    output = "Vertices:\n"
    if (genAlg != null) {
        for (vertex in genAlg.vertexSet())
            output += vertex.toString() + "\n"
    }
    output += "Edges:\n"
    if (genAlg != null) {
        for (edge in genAlg.edgeSet())
            output += edge.toString() + "\n"
    }
    println("Run time: ${(System.currentTimeMillis() - startTime) / 1000.0}
seconds")

    File("geneticAlgorithmGraph.txt").writeText(output)
}
```

HeuristicNetworkTop.kt

```
package com.cs6385

import org.jgrapht.Graph
import org.jgrapht.GraphMetrics
import org.jgrapht.GraphTests.isConnected
import org.jgrapht.graph.DefaultUndirectedGraph
import org.jgrapht.graph.DefaultUndirectedWeightedGraph
import org.jgrapht.graph.DefaultWeightedEdge
import java.security.SecureRandom

const val MIN = -100
const val MAX = 100

class HeuristicNetworkTop {

    var network: DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge> =
        DefaultUndirectedWeightedGraph(DefaultWeightedEdge::class.java)
    private val randGenerator: SecureRandom = SecureRandom()


    constructor(n: Int) {

        var coords = mutableListOf<Coordinate2D>()
        for (i in 0 until n) {
            var randCoord = Coordinate2D.genRandomCoordinate(MIN * (n / 15),
(MAX * (n / 15)))

            while (randCoord in coords) {
                randCoord = Coordinate2D.genRandomCoordinate(MIN * (n / 15),
(MAX * (n / 15)))
            }

            coords.add(randCoord)
            network.addVertex(randCoord)
        }

    }

    fun getMinConstraintGraph(): DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge> {
        val minConstraintGraph = network.clone() as
DefaultUndirectedGraph<Coordinate2D, DefaultWeightedEdge>
        val chosenEdges = mutableListOf<Pair<Coordinate2D, Coordinate2D>>()
        val chosenEdgesWeights = mutableListOf<Double>()

        val vertexList = mutableListOf<Coordinate2D>()
        vertexList.addAll(network.vertexSet())
```

```
        // Violates degree and diameter constraints
        while (!isConnected(minConstraintGraph)
|| !graphIsAtLeastDegree(minConstraintGraph, 3)
            || GraphMetrics.getDiameter(minConstraintGraph) > 4
        ) {

            if (!graphIsAtLeastDegree(minConstraintGraph, 3) ||
                (isConnected(minConstraintGraph) &&
GraphMetrics.getDiameter(minConstraintGraph) > 4)
            ) {
                var chosenVertex1 =
vertexList[randGenerator.nextInt(network.vertexSet().size)]
                var chosenVertex2 =
vertexList[randGenerator.nextInt(network.vertexSet().size)]

                while (chosenVertex1 == chosenVertex2
                    || Pair(chosenVertex1, chosenVertex2) in chosenEdges
                    || Pair(chosenVertex2, chosenVertex1) in chosenEdges
                ) {
                    chosenVertex1 =
vertexList[randGenerator.nextInt(network.vertexSet().size)]
                    chosenVertex2 =
vertexList[randGenerator.nextInt(network.vertexSet().size)]
                }

                minConstraintGraph.addEdge(chosenVertex1, chosenVertex2)
                chosenEdges.add(Pair(chosenVertex1, chosenVertex2))

chosenEdgesWeights.add(chosenVertex1.getEuclideanDistance(chosenVertex2))
            }
        }

        val minConstraintWeightedGraph =
            DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge>(DefaultWeightedEdge::class.java)
        for (vertex in minConstraintGraph.vertexSet()) {
            minConstraintWeightedGraph.addVertex(vertex)
        }

        for ((i, edge) in minConstraintGraph.edgeSet().withIndex()) {
            minConstraintWeightedGraph.addEdge(
                minConstraintGraph.getEdgeSource(edge),
                minConstraintGraph.getEdgeTarget(edge)
            )
            minConstraintWeightedGraph.setEdgeWeight(
                minConstraintGraph.getEdgeSource(edge),
                minConstraintGraph.getEdgeTarget(edge),
                chosenEdgesWeights[i]
            )
        }
```

```
        return minConstraintWeightedGraph
    }


    companion object {
        fun graphIsAtLeastDegree(graph: Graph<Coordinate2D,
DefaultWeightedEdge>, degree: Int): Boolean {
            for (vertex in graph.vertexSet()) {
                if (graph.degreeOf(vertex) < degree) return false
            }

            return true
        }

        fun getTotalCost(graph: Graph<Coordinate2D, DefaultWeightedEdge>):
Double {
            var totalCost = 0.0

            for (edge in graph.edgeSet())
                totalCost += graph.getEdgeWeight(edge)

            return totalCost
        }
    }

}
```

WorstToBestAlg.kt

```
package com.cs6385

import org.jgrapht.GraphMetrics
import org.jgrapht.GraphTests
import org.jgrapht.graph.DefaultUndirectedGraph
import org.jgrapht.graph.DefaultUndirectedWeightedGraph
import org.jgrapht.graph.DefaultWeightedEdge

class WorstToBestAlg {
    private val initialGraph: DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge>

    constructor(initGraph: DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge>) {
        initialGraph = initGraph.clone() as
DefaultUndirectedWeightedGraph<Coordinate2D, DefaultWeightedEdge>
    }

    fun run(): DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge> {
        var currentGraph = initialGraph.clone() as
DefaultUndirectedWeightedGraph<Coordinate2D, DefaultWeightedEdge>

        // Iterate through all vertices in the current graph
        var candidateGraph = currentGraph.clone() as
DefaultUndirectedWeightedGraph<Coordinate2D, DefaultWeightedEdge>
        for (thisVertex in currentGraph.vertexSet()) {
            val edgesOfVertex = mutableListOf<Pair<DefaultWeightedEdge,
Double>>()
            for (edge in currentGraph.edgeSet()) {
                if (currentGraph.getEdgeSource(edge) == thisVertex)
                    edgesOfVertex.add(Pair(edge,
currentGraph.getEdgeWeight(edge)))
            }

            // Sort edges by weight
            fun edgeSorter(edge: Pair<DefaultWeightedEdge, Double>) =
edge.second
            edgesOfVertex.sortBy { edgeSorter(it) }

            // Take worst edge
            if (edgesOfVertex.isEmpty())
                continue

            val worstEdge = edgesOfVertex.last().first

            // Get a list of all possible edges for this vertex
            var allPossibleEdges = hashMapOf<Pair<Coordinate2D, Coordinate2D>,
Double>()
```

```
            for (anotherVertex in currentGraph.vertexSet()) {
                if (anotherVertex != thisVertex)
                    allPossibleEdges[Pair(thisVertex, anotherVertex)] =
thisVertex.getEuclideanDistance(anotherVertex)
            }

            // Remove edges that already exist
            val edgesOfVertexNoWeight = mutableListOf<Pair<Coordinate2D,
Coordinate2D>>()
            for (edge in edgesOfVertex.map { it.first }) {

edgesOfVertexNoWeight.add(Pair(currentGraph.getEdgeSource(edge),
currentGraph.getEdgeTarget(edge)))
            }
            val removeEdges = mutableListOf<Pair<Coordinate2D,
Coordinate2D>>()
            for (edge in allPossibleEdges) {
                if (edge.key in edgesOfVertexNoWeight)
                    removeEdges.add(edge.key)
            }
            for (edge in removeEdges) {
                allPossibleEdges.remove(edge)
            }

            // Sort all possible edges by weight
            allPossibleEdges = allPossibleEdges.toList().sortedBy { (_, value)
-> value }
                .toMap() as HashMap<Pair<Coordinate2D, Coordinate2D>, Double>

            // Remove worst edge and swap out with best edge that still
satisfies constraints
            for (edge in allPossibleEdges.keys) {

                // This candidate edge is worse than the current worst edge
                if (edge.first.getEuclideanDistance(edge.second) >
currentGraph.getEdgeWeight(worstEdge))
                    continue

                candidateGraph.removeEdge(
                    candidateGraph.getEdgeSource(worstEdge),
                    candidateGraph.getEdgeTarget(worstEdge)
                )
                candidateGraph.addEdge(edge.first, edge.second)
                candidateGraph.setEdgeWeight(edge.first, edge.second,
edge.first.getEuclideanDistance(edge.second))

                // The new edge does not fit the constraint
                if (this.getFitness(candidateGraph) ==
Double.POSITIVE_INFINITY) {
                    candidateGraph =
                        currentGraph.clone() as
```

```
DefaultUndirectedWeightedGraph<Coordinate2D, DefaultWeightedEdge>
                }

                // The new edge satisfies the constraint
                else
                    break
            }

            currentGraph = candidateGraph.clone() as
DefaultUndirectedWeightedGraph<Coordinate2D, DefaultWeightedEdge>
        }

        return currentGraph
    }

    private fun getFitness(individual:
DefaultUndirectedWeightedGraph<Coordinate2D, DefaultWeightedEdge>): Double {
        val individualNoWeight =
            DefaultUndirectedGraph<Coordinate2D,
DefaultWeightedEdge>(DefaultWeightedEdge::class.java)
        for (vertex in individual.vertexSet())
            individualNoWeight.addVertex(vertex)
        for (edge in individual.edgeSet())
            individualNoWeight.addEdge(individual.getEdgeSource(edge),
individual.getEdgeTarget(edge))

        // Huge penalty function
        if (!GraphTests.isConnected(individualNoWeight)
            || !HeuristicNetworkTop.graphIsAtLeastDegree(individualNoWeight,
3)
            || GraphMetrics.getDiameter(individualNoWeight) > 4
        ) {
            return Double.POSITIVE_INFINITY
        }

        // Distance from cost to 0
        return HeuristicNetworkTop.getTotalCost(individual)
    }
}
```

GeneticAlg.kt

```
package com.cs6385

import org.jgrapht.GraphMetrics
import org.jgrapht.GraphTests.isConnected
import org.jgrapht.graph.DefaultUndirectedGraph
import org.jgrapht.graph.DefaultUndirectedWeightedGraph
import org.jgrapht.graph.DefaultWeightedEdge
import java.lang.RuntimeException
import java.security.SecureRandom

class GeneticAlg {
    private var initialPopulation:
MutableList<DefaultUndirectedWeightedGraph<Coordinate2D, DefaultWeightedEdge>>
=
        mutableListOf()
    private var mutationProbability: Double = 0.2
    private val randGenerator: SecureRandom = SecureRandom()


    constructor(init_population:
MutableList<DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge>>) {
        if (init_population.size < 2) {
            throw RuntimeException(
                "ERROR: Initial population size for genetic algorithm is
invalid!\n" +
                        "Initial population size must be at least 2."
            )
        }
        initialPopulation = init_population
    }

    constructor(
        init_population:
MutableList<DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge>>,
        mutation_probability: Double
    ) : this(init_population) {
        mutationProbability = mutation_probability
    }

    fun run(numRuns: Int): DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge>? {
        // Set the initial current generation to the initial population
        val currentGen =
mutableListOf<DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge>>()
        currentGen.addAll(initialPopulation)
```

```
        // Run for a specified number of runs
        for (r in 0 until numRuns) {

            // Not enough individuals in the generation
            if (currentGen.size < 2)
                throw RuntimeException("ERROR: The current generation needs to
have at least 2 individuals, this generation has ${currentGen.size}")

            var candidates =
mutableListOf<Pair<DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge>, Double>>()

            // Get elite from the current generation, passed on to the next
generation
            var currentGenWithWeights =
                hashMapOf<DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge>, Double>()
            for (individual in currentGen) {
                currentGenWithWeights[individual] =
HeuristicNetworkTop.getTotalCost(individual)
            }
            currentGenWithWeights = currentGenWithWeights.toList().sortedBy
{ (_, value) -> value }
                .toMap() as
HashMap<DefaultUndirectedWeightedGraph<Coordinate2D, DefaultWeightedEdge>,
Double>

            val eliteCurrent = currentGenWithWeights.keys.toList().subList(
                0, kotlin.math.ceil(currentGenWithWeights.keys.toList().size *
0.05)
                    .toInt()
            )
            for (eliteIndividual in eliteCurrent) {
                if (this.getFitness(eliteIndividual) !=
Double.POSITIVE_INFINITY)
                    candidates.add(Pair(eliteIndividual,
this.getFitness(eliteIndividual)))
                currentGen.remove(eliteIndividual)
            }

            // Generate mutation child (if applicable)
            var mutationChild: DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge>
            var mutated = -1
            if (randGenerator.nextDouble() < mutationProbability) {

                mutated = randGenerator.nextInt(currentGen.size)

                // Copy old to mutation
                mutationChild = this.doMutate(currentGen[mutated])
```

```
                if (this.getFitness(mutationChild) !=
Double.POSITIVE_INFINITY)
                    candidates.add(Pair(mutationChild,
this.getFitness(mutationChild)))
            }

            // Generate crossover children
            val crossOverChildren =
mutableListOf<DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge>>()
            for (i in 0 until currentGen.size) {
                for (j in 0 until currentGen.size) {
                    if (i != j && i != mutated && j != mutated)
                        crossOverChildren.add(this.doCrossover(currentGen[i],
currentGen[j]))
                }
            }

            for (child in crossOverChildren) {
                if (this.getFitness(child) != Double.POSITIVE_INFINITY) {
                    candidates.add(Pair(child, this.getFitness(child)))
                }
            }

            fun candidateSelector(individual:
Pair<DefaultUndirectedWeightedGraph<Coordinate2D, DefaultWeightedEdge>,
Double>) =
                individual.second

            candidates.sortBy { candidateSelector(it) }

            // Get next generation
            val nextGen =
mutableListOf<DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge>>()

            var curCandidate = 0
            while (nextGen.size < initialPopulation.size * 2) {
                nextGen.add(candidates[curCandidate].first)
                curCandidate++
            }

            // Make the next generation the current generation
            currentGen.clear()
            currentGen.addAll(nextGen)
            nextGen.clear()
        }

        val currentGenFitness = mutableListOf<Double>()
        for (individual in currentGen)
```

```
            currentGenFitness.add(this.getFitness(individual))
        currentGenFitness.sort()

        val optimalFitness = currentGenFitness[0]

        for (individual in currentGen) {
            if (this.getFitness(individual) == optimalFitness)
                return individual
        }

        return null
    }

    private fun doCrossover(
        mother: DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge>,
        father: DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge>
    ): DefaultUndirectedWeightedGraph<Coordinate2D, DefaultWeightedEdge> {
        val child = DefaultUndirectedWeightedGraph<Coordinate2D,
DefaultWeightedEdge>(DefaultWeightedEdge::class.java)
        for (vertex in mother.vertexSet()) {
            child.addVertex(vertex)
        }

        // Select random split point
        val splitPoint = randGenerator.nextDouble()

        // Get part of the mother's genes and part of the father's genes and
recombine into the child's genes
        val childList = mutableListOf<DefaultWeightedEdge>()
        childList.addAll(mother.edgeSet().toList().subList(0,
(mother.edgeSet().size * splitPoint).toInt()))
        childList.addAll(
            father.edgeSet().toList().subList((father.edgeSet().size *
splitPoint).toInt(), father.edgeSet().size)
        )

        // Add these genes to the child graph
        while (childList.isNotEmpty()) {
            val edge = childList.first()
            child.addEdge(child.getEdgeSource(edge),
child.getEdgeTarget(edge))
            child.setEdgeWeight(child.getEdgeSource(edge),
child.getEdgeTarget(edge), child.getEdgeWeight(edge))
            childList.remove(edge)
        }

        return child
    }
```

```
    private fun doMutate(individual:
DefaultUndirectedWeightedGraph<Coordinate2D, DefaultWeightedEdge>):
DefaultUndirectedWeightedGraph<Coordinate2D, DefaultWeightedEdge> {
        val mutatedIndividual = individual.clone() as
DefaultUndirectedWeightedGraph<Coordinate2D, DefaultWeightedEdge>

        // Determine how many edges to change
        val numEdgesChange = randGenerator.nextInt(individual.edgeSet().size -
1) + 1

        // Remove certain edges from the graph
        for (i in 0 until numEdgesChange) {
            // Get random edge to remove
            val randomEdge =

mutatedIndividual.edgeSet().toList()[randGenerator.nextInt(mutatedIndividual.e
dgeSet().size)]
            mutatedIndividual.removeEdge(randomEdge)
        }

        // Add random edges to the graph
        for (i in 0 until numEdgesChange) {
            val sourceVertex =

mutatedIndividual.vertexSet().toList()[randGenerator.nextInt(mutatedIndividual
.vertexSet().size)]
            var destinationVertex =

mutatedIndividual.vertexSet().toList()[randGenerator.nextInt(mutatedIndividual
.vertexSet().size)]

            while (sourceVertex == destinationVertex) {
                destinationVertex =

mutatedIndividual.vertexSet().toList()[randGenerator.nextInt(mutatedIndividual
.vertexSet().size)]
            }

            mutatedIndividual.addEdge(sourceVertex, destinationVertex)
            mutatedIndividual.setEdgeWeight(
                sourceVertex,
                destinationVertex,
                sourceVertex.getEuclideanDistance(destinationVertex)
            )
        }

        return mutatedIndividual
    }

    private fun getFitness(individual:
DefaultUndirectedWeightedGraph<Coordinate2D, DefaultWeightedEdge>): Double {
```

```
        val individualNoWeight =
            DefaultUndirectedGraph<Coordinate2D,
DefaultWeightedEdge>(DefaultWeightedEdge::class.java)
        for (vertex in individual.vertexSet())
            individualNoWeight.addVertex(vertex)
        for (edge in individual.edgeSet())
            individualNoWeight.addEdge(individual.getEdgeSource(edge),
individual.getEdgeTarget(edge))

        // Huge penalty function
        if (!isConnected(individualNoWeight)
            || !HeuristicNetworkTop.graphIsAtLeastDegree(individualNoWeight,
3)
            || GraphMetrics.getDiameter(individualNoWeight) > 4
        ) {
            return Double.POSITIVE_INFINITY
        }

        // Distance from cost to 0
        return HeuristicNetworkTop.getTotalCost(individual)
    }

}
```

Coordinate2D.kt

```
package com.cs6385

import kotlin.math.sqrt
import kotlin.math.pow

import java.security.SecureRandom

class Coordinate2D {

    constructor(x_val: Int, y_val: Int) {
        x = x_val
        y = y_val
    }

    var x: Int

    var y: Int

    override fun equals(rhs: Any?): Boolean {
        return if (rhs is Coordinate2D) {
            this.x == rhs.x && this.y == rhs.y
        } else
            false
    }

    override fun toString(): String {
        return "(${this.x}, ${this.y})"
    }

    fun getEuclideanDistance(rhs: Coordinate2D): Double {
        return sqrt((this.x - rhs.x / 1.0).pow(2.0) + (this.y - rhs.y /
1.0).pow(2.0))
    }

    companion object {
        fun genRandomCoordinate(lower: Int, upper: Int): Coordinate2D {
            var x = SecureRandom().nextInt(upper - lower) + upper
            var y = SecureRandom().nextInt(upper - lower) + upper

            return Coordinate2D(x, y)
        }
    }
}
```

pom.xml

```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/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>com.cs6385</groupId>
    <artifactId>project3</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>com.cs6385 Project 3</name>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <kotlin.version>1.3.72</kotlin.version>
        <kotlin.code.style>official</kotlin.code.style>
        <junit.version>4.12</junit.version>
        <maven.compiler.release>11</maven.compiler.release>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.jgrapht</groupId>
            <artifactId>jgrapht-core</artifactId>
            <version>1.4.0</version>
        </dependency>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-stdlib</artifactId>
            <version>${kotlin.version}</version>
        </dependency>
        <dependency>
            <groupId>org.jetbrains.kotlin</groupId>
            <artifactId>kotlin-test-junit</artifactId>
            <version>${kotlin.version}</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>${junit.version}</version>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
```

```
        <sourceDirectory>src/main/kotlin</sourceDirectory>
        <testSourceDirectory>src/test/kotlin</testSourceDirectory>

        <plugins>
            <plugin>
                <groupId>org.jetbrains.kotlin</groupId>
                <artifactId>kotlin-maven-plugin</artifactId>
                <version>${kotlin.version}</version>
                <executions>
                    <execution>
                        <id>compile</id>
                        <phase>compile</phase>
                        <goals>
                            <goal>compile</goal>
                        </goals>
                    </execution>
                    <execution>
                        <id>test-compile</id>
                        <phase>test-compile</phase>
                        <goals>
                            <goal>test-compile</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.8.1</version>
                <configuration>
                    <release>${maven.compiler.release}</release>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <version>3.2.0</version>
                <configuration>
                    <archive>
                        <manifest>
                            <addClasspath>true</addClasspath>
                            <classpathPrefix>libs/</classpathPrefix>
                            <mainClass>
                                com.cs6385.RunnerKt
                            </mainClass>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
```

```
                <artifactId>maven-shade-plugin</artifactId>
                <version>3.2.3</version>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>shade</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>

</project>
```

References

[1] https://jgrapht.org/

[2] https://matplotlib.org/