INFO 6205 Spring 2023 Project

Traveling Salesman Problem Solution Using Christofides Algorithm and Optimization Techniques

Built by -

Abhinav Choudhary - 002780326

Krishnna Sarrdah - 002771329

Hariharan Sundaram - 002915360

Introduction

Aim

The objective of this report is to conduct a thorough investigation into the efficacy of the Christofides algorithm, along with its optimization techniques, including tactical approaches such as 2-opt and 3-opt improvements, as well as strategic methods such as simulated annealing and genetic algorithm, in solving the traveling salesman problem. Through the utilization of these methods, we aim to achieve accurate and high-quality approximate solutions. Furthermore, this report will also address the limitations and challenges associated with these methods, and provide valuable recommendations for further research in this field.

Approach

Our project entailed the development of a Java-based solution to implement the Christofides algorithm and optimize it using tactical and strategic methods, with a strong focus on adhering to Object-Oriented Programming (OOP) principles for achieving modularity, reusability, and readability. The initial phase involved designing robust data structures, including graph, node, edge classes, to effectively model the problem domain and facilitate efficient computation.

Subsequently, we implemented the Christofides algorithm, which entailed a series of intricate steps. Firstly, we computed the distances between nodes by applying the Haversine formula to their latitude and longitude values obtained from the given crime dataset. Next, we utilized

Prim's algorithm to construct the Minimum Spanning Tree (MST) of the graph and identify the odd vertices. Leveraging Blossom’s perfect matching algorithm, we matched the edges of the MST to create a subgraph with an Eulerian circuit. We then utilized the Eulerian circuit to construct a Hamiltonian cycle by eliminating redundant vertices.

To optimize the resulting tour, we employed a range of advanced techniques, including 2-opt and 3-opt improvements, simulated annealing, and ant colony optimization. Additionally, we leveraged the JavaFX library to develop graphical representations that allowed us to visually analyze the accuracy and performance of our solution.

Upon completing the implementation of algorithms and optimization methods, we conducted thorough testing and comprehensive analysis of the results. The use of graphical representations enabled us to illustrate the accuracy and performance of our program in a visually impactful manner. Overall, our approach combined theoretical analysis, sophisticated programming techniques, and meticulous experimentation to thoroughly investigate the effectiveness of the Christofides algorithm and its optimization methods in obtaining high-quality approximate solutions to the challenging traveling salesman problem. The incorporation of OOP concepts, robust data structures, and external libraries such as JavaFX contributed to the modularity, reusability, and readability of our project, underscoring our commitment to delivering a technically proficient and professional solution.

Program

Data Structures Used:

- Lists(Arraylists)
- Trees
- Hashmaps
- Hashset
- Graphs
- PriorityQueue
- Customized Classes(Graph, Edge, Node)

Classes Used

Package Christofides

- ***Binary Heap:*** The BinaryHeap class implements a binary min-heap data structure that maintains a collection of key-value pairs, where keys are doubles and values are integers. It provides methods for inserting key-value pairs, deleting the minimum key-value pair, and maintaining the heap property, as well as keeping track of the positions of the values in the heap for efficient operations.

- ***Blossom graph:*** The BlossomGraph class represents a graph data structure used in the Blossoms Algorithm. It stores information about nodes, edges, weights, and adjacency relationships. It provides methods for retrieving node and edge information, checking adjacency between nodes, and calculating weights and degrees. It also maintains mappings between node ids and indices for efficient graph operations.

- ***Edge*:** The Edge class represents an edge between two nodes in a graph, with methods for calculating the weight of the edge based on latitude and longitude, and comparing edge weights.

- ***EulerTour***: The EulerTour class builds an Eulerian tour (a cycle that visits all edges of a graph exactly once) from a given graph using a Depth-First Search (DFS) algorithm. It also provides methods to build the resulting tour as a graph or as a list of nodes, and keeps track of visited edges to ensure each edge is visited only once.

- ***Graph***: The Graph class represents an undirected graph and provides methods for adding edges, retrieving nodes and edges, calculating weights, and other graph-related operations. It uses an adjacency list to store edges and provides methods to access and manipulate graph properties such as edge count, vertex count, degrees of nodes, and total weight of edges.

- ***MST*:** The MST class calculates the minimum spanning tree (MST) of a given graph using Prim's algorithm. It builds the MST by iteratively selecting the edges with the minimum weight, ensuring that all nodes are connected with the shortest distance in the process. It also provides methods to compute the total weight of the MST and to build a new graph representing the MST.

- ***Node***: The Node class represents a node in a graph. It stores the node's ID, as well as its x and y coordinates. It provides methods to get and set the ID, x, and y coordinates of the node, as well as methods for equality comparison and hash code calculation.

- ***TSPRoute***: This class is for solving the Traveling Salesman Problem (TSP) using various optimization techniques, including Genetic Algorithm, Simulated Annealing, 2-Opt, and 3-Opt. It reads input data, builds a Minimum Spanning Tree (MST) and computes an Euler Tour. It also includes functions to run the optimization algorithms and print the total weight of the optimized TSP tour.

- ***BlossomMatching:*** This class computes the minimum weight perfect matching(MWPM) between a set of vertices passed to it as input parameters. MWPM is a process where vertices are paired so as to minimize the total weight of matching.

Package Christofides.optimizations

- ***Genetic AlgoSolver***: The provided code is an implementation of a Genetic Algorithm for solving the Traveling Salesman Problem (TSP). It includes methods for initializing a population, evaluating the fitness of individuals, selecting parents through tournament selection, performing crossover, mutating individuals, and building a tour from the best solution. The code uses a representation of tours as arrays of strings representing node IDs, and employs basic genetic operators such as crossover and mutation to evolve the population towards finding an optimal TSP solution.

- ***Helper***: The "Helper" class provides various utility methods for optimizing the Christofides algorithm for the Traveling Salesman Problem (TSP). It includes methods for calculating distances between nodes, mapping nodes to their IDs, converting between different representations of tours, and generating an ArrayList of edges from a tour.

- ***SimulatedAnnealingSolver***: The SimulatedAnnealingSolver class is an implementation of the Simulated Annealing algorithm for solving the Traveling Salesman Problem. It takes a graph representation of a TSP tour as input and uses simulated annealing to optimize the tour by iteratively swapping cities and accepting or rejecting the swaps based on a probability determined by the current temperature and the cost of the new tour. The class calculates the cost of a given tour, generates neighbors by swapping cities, and calculates the acceptance probability for the annealing process.

- ***ThreeOpt***: This class is a Java implementation of the 3-opt optimization algorithm for solving the Traveling Salesman Problem. It includes two approaches for performing the 3-opt exchange on a tour and uses HashMaps and ArrayLists to represent the graph and tour. The goal is to improve the total weight (distance) of the tour by iteratively exchanging edges to find a shorter tour.

- ***TwoOpt***: The TwoOpt class is a Java implementation of the 2-opt optimization algorithm used for solving the Traveling Salesman Problem. It takes a Graph representing a tour as input and iteratively swaps pairs of edges to find a shorter tour. The class includes methods for calculating distances between nodes, performing the 2-opt swaps, and building a new tour. The goal is to improve the total weight (distance) of the tour by optimizing the order of nodes.

Package uiHelper

- ***DataNormalizer***: The "DataNormalizer" class is a Java class that normalizes and denormalizes the latitude and longitude values of nodes in a graph. It calculates the min and max latitude and longitude values and then scales the values to a normalized range using a normalization factor. The class also provides methods to denormalize the data back to the original range.

- ***DataProvider***: The "DataProvider" class is a Java class that reads data from a text file, parses it, and creates a list of nodes with crime-related information such as crimeID, latitude, and longitude. It provides methods to access and modify the crime-related data for each node.

- ***RandomGraphGenerater***: The "RandomGraphGenerator" class is a Java class that generates random graphs with nodes and edges for the Traveling Salesman Problem (TSP). It creates nodes with random coordinates and edges with random weights within specified limits. It also has a method to generate graphs with loops by connecting nodes to their adjacent neighbors.

- ***UICreator***: The UICreator class is a JavaFX application that displays a graph with nodes and edges representing a TSP (Traveling Salesman Problem) route. It reads data from a file, normalizes the data, and calculates the center point and scaling factors. It then creates circles for the nodes and lines for the edges, scaling their coordinates based on the calculated values. Buttons for different optimization algorithms are added to the UI.

Algorithms Used

Christofides Algorithm

The Christofides algorithm is a heuristic approach used to find approximate solutions for the Traveling Salesman Problem (TSP), a well-known problem in computer science. The algorithm can be summarized in the following steps:

1. Construct the minimum spanning tree (MST) of the input graph using a standard Prim's algorithm.
2. Identify the set of vertices in the MST that have odd degrees and find the minimum-weight perfect matching (MWPM) among them using the Blossom algorithm.
3. Combine the MST and the MWPM to create a multigraph where every vertex has an even degree.
4. Find an Eulerian path in the multigraph, which is a closed path that traverses each edge exactly once.
5. Convert the Eulerian path into a Hamiltonian circuit by eliminating repeated vertices while preserving the visitation order.
6. Output the Hamiltonian circuit as the approximate solution for the TSP.

By following these steps, the Christofides algorithm provides a solution that is guaranteed to be at most 1.5 times the optimal solution, making it a popular choice for solving TSP instances.

Blossom Matching Algorithm

The Hungarian algorithm is a very efficient graph matching algorithm, but it only works on weighted bipartite graphs as it cannot deal with odd length cycles. The issue lies in finding augmenting paths, since if there is an odd cycle, the algorithm will calculate the wrong augmenting path because it will miss some of the edges. Additionally, a cycle may make the algorithm loop forever, reducing, in the case of minimum-weight optimization, the weight infinitely many times, inhibiting proper termination. The Blossom algorithm aims to fill this gap by solving more generalized graph matching problems, and can be used when the graph is not guaranteed to be bipartite.

The blossom algorithm takes a general graph and finds a maximum matching *M*. The algorithm starts with an empty matching and then iteratively improves it by adding edges, one at a time, to build augmenting paths in the matching *M*. Adding to an augmenting path can grow a matching since every other edge in an augmenting path is an edge in the matching; as more edges are added to the augmenting path, more matching edges are discovered. The blossom algorithm has three possible results after each iteration. Either the algorithm finds no augmenting paths, in which case it has found a maximum matching; an augmenting path can be found in order to improve the outcome; or a blossom can be found in order to manipulate it and discover a new augmenting path.

2-opt Algorithm

The 2-opt optimization algorithm is a widely used and simple heuristic for enhancing the quality of solutions in the Traveling Salesman Problem (TSP). It follows an iterative local search approach that starts with an initial solution, typically obtained from a construction heuristic. The algorithm then repeatedly swaps pairs of edges in the tour to optimize its length. The name "2-opt" comes from the fact that it involves swapping two edges at a time.
The fundamental concept of the 2-opt algorithm is to continuously swap pairs of edges in the tour to find shorter paths. The algorithm compares the length of the original tour with the length of the tour obtained after swapping two edges. If the swapped tour is shorter, the swap is performed. This process is repeated until no further improvements can be made, or a predefined termination condition is met, such as a maximum number of iterations or a time limit. The time complexity of the 2-opt algorithm is O(n^2), where n is the number of cities in the TSP. Despite its simplicity, the 2-opt algorithm can often yield substantial improvements to the initial solution and converge to a locally optimal solution. However, it does not guarantee to find the global optimal solution, as it may get stuck in local optima. Hence, it is commonly used as a local search component in combination with other algorithms or heuristics for solving TSP instances.

3-opt algo

The 3-opt optimization algorithm is a popular local search-based heuristic used for solving the traveling salesman problem (TSP), which is a classic combinatorial optimization problem. The algorithm operates on a cycle that represents a tour that visits all cities in a TSP instance. The basic idea is to improve the tour by swapping three edges at a time to create a new cycle, and then evaluating the length of the new cycle to determine if it is shorter than the current one. The 3-opt algorithm uses a systematic approach to select three edges from the current cycle and rearrange their connections to form a new cycle. This involves exploring all possible combinations of three edges, known as 3-opt moves, and selecting the one that results in the greatest reduction in tour length. The algorithm then repeats this process until no further improvements can be made, or a stopping criterion is met.

One of the strengths of the 3-opt algorithm is that it can quickly escape local optima by exploring different neighborhood structures of the current solution. This allows it to effectively search for potentially better solutions in a large solution space. However, the algorithm's performance depends heavily on the quality of the initial solution, as well as the selection strategy for 3-opt moves. Therefore, various enhancements and modifications, such as using different types of neighborhood structures or incorporating randomization, have been proposed to improve the algorithm's performance.

Despite being a relatively simple heuristic, the 3-opt algorithm has been proven to be effective in finding near-optimal solutions for small to moderate-sized TSP instances. It has been widely used in various applications, such as routing, logistics, and transportation planning. Furthermore, the 3-opt algorithm can be easily extended to handle variations of the TSP, such as the multiple traveling salesman problem (mTSP) or the vehicle routing problem (VRP), making it a versatile and widely used optimization technique in the field of operations research and combinatorial optimization.

Simulated Annealing

The Simulated Annealing algorithm is a stochastic optimization technique used for solving complex combinatorial optimization problems, such as the well-known traveling salesman problem (TSP). The algorithm is grounded on the analogy of the annealing process observed in metallurgy, where controlled heating and gradual cooling are employed to reduce structural defects and enhance material purity. In a similar vein, the Simulated Annealing algorithm employs a probabilistic approach to explore the vast solution space of combinatorial problems and identify near-optimal solutions.

At the outset, the algorithm initializes with an initial solution and then iteratively generates neighboring solutions through local search operations, such as edge swapping in the case of TSP. Each neighboring solution is assessed using an objective function, such as the total tour length in TSP, to determine its quality. Importantly, the algorithm accepts suboptimal solutions with a certain probability, which allows it to escape from local optima and explore different areas of the solution space. The acceptance probability for suboptimal solutions decreases over time as the algorithm "cools down" based on a temperature schedule. This temperature schedule governs the trade-off between exploration and exploitation, facilitating global and local improvements to the solution. The algorithm continues until a predetermined stopping criterion is met, such as reaching a maximum number of iterations or a specified temperature threshold.

Genetic Algorithm

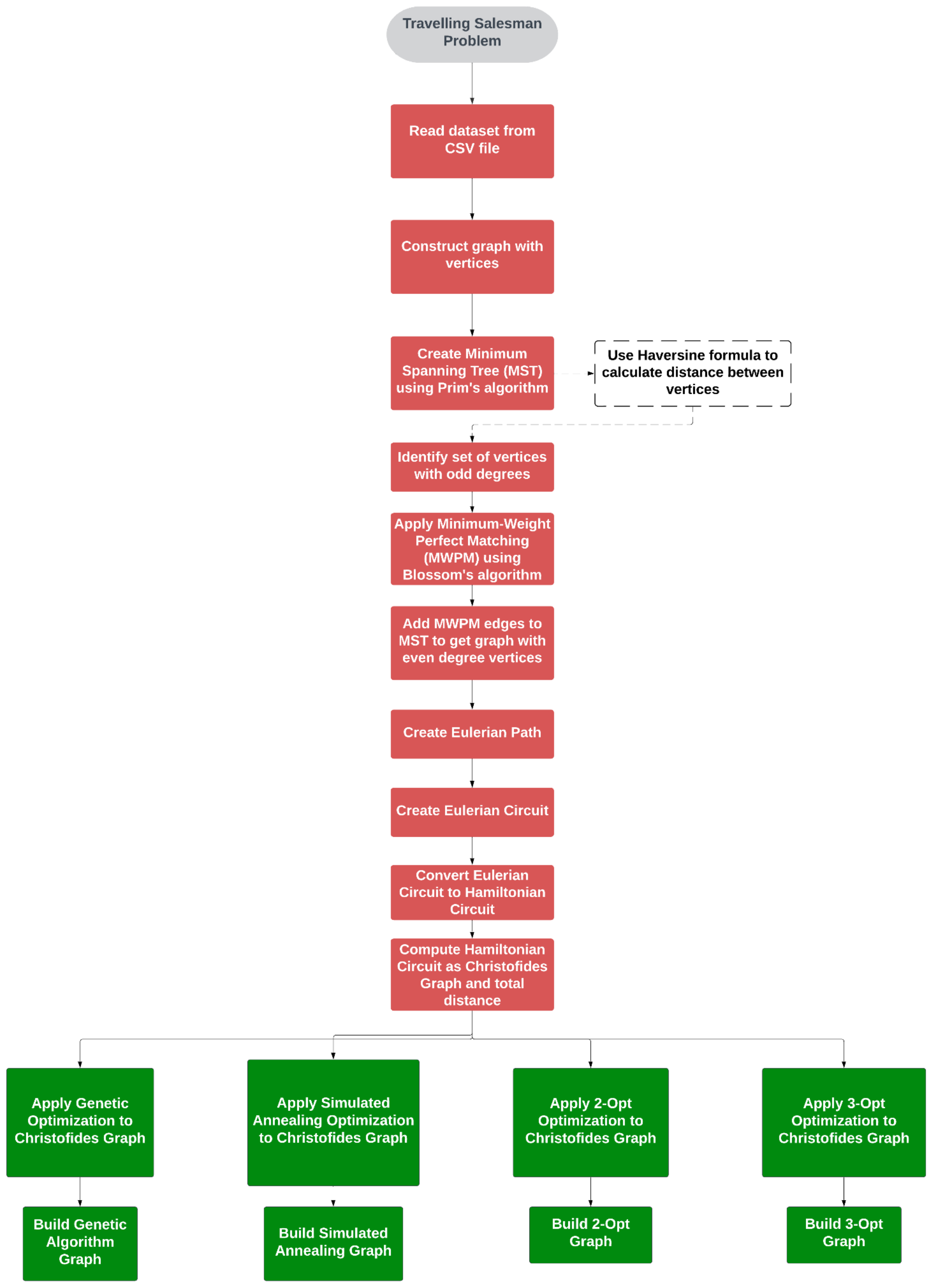
The Genetic Algorithm (GA) is a potent optimization technique utilized for solving complex optimization problems, specifically the Traveling Salesman Problem (TSP) with the Christofides heuristic. TSP is a classic combinatorial optimization problem that aims to determine the shortest route that visits a given set of cities and returns to the starting city. The Christofides heuristic is a well-known algorithm that provides an initial solution for TSP by constructing a minimum spanning tree and augmenting it with a set of edges to form a tour that is guaranteed to be no more than 50% longer than the optimal solution.

In the GA-based optimization approach for TSP with the Christofides heuristic, a population of candidate solutions, represented as permutations of cities, is evolved over generations using genetic operators. The algorithm initiates with an initial population of randomly generated solutions and iteratively applies genetic operators, such as crossover (recombination) and mutation, to generate new offspring solutions. Crossover combines genetic information from two parent solutions to create one or more offspring solutions, while mutation introduces small random changes in a solution to explore different regions of the solution space. The offspring solutions are then evaluated using an objective function, such as the total tour length, and the fittest solutions are selected to form the next generation. This iterative process is repeated for a predetermined number of generations or until a stopping criterion is met, such as reaching a certain fitness threshold or exhausting computational resources. Through the iterative evolution of the population and the application of genetic operators, the GA-based optimization approach systematically explores the solution space, exploits promising solutions, and progressively converges to a near-optimal solution for TSP with the Christofides heuristic.

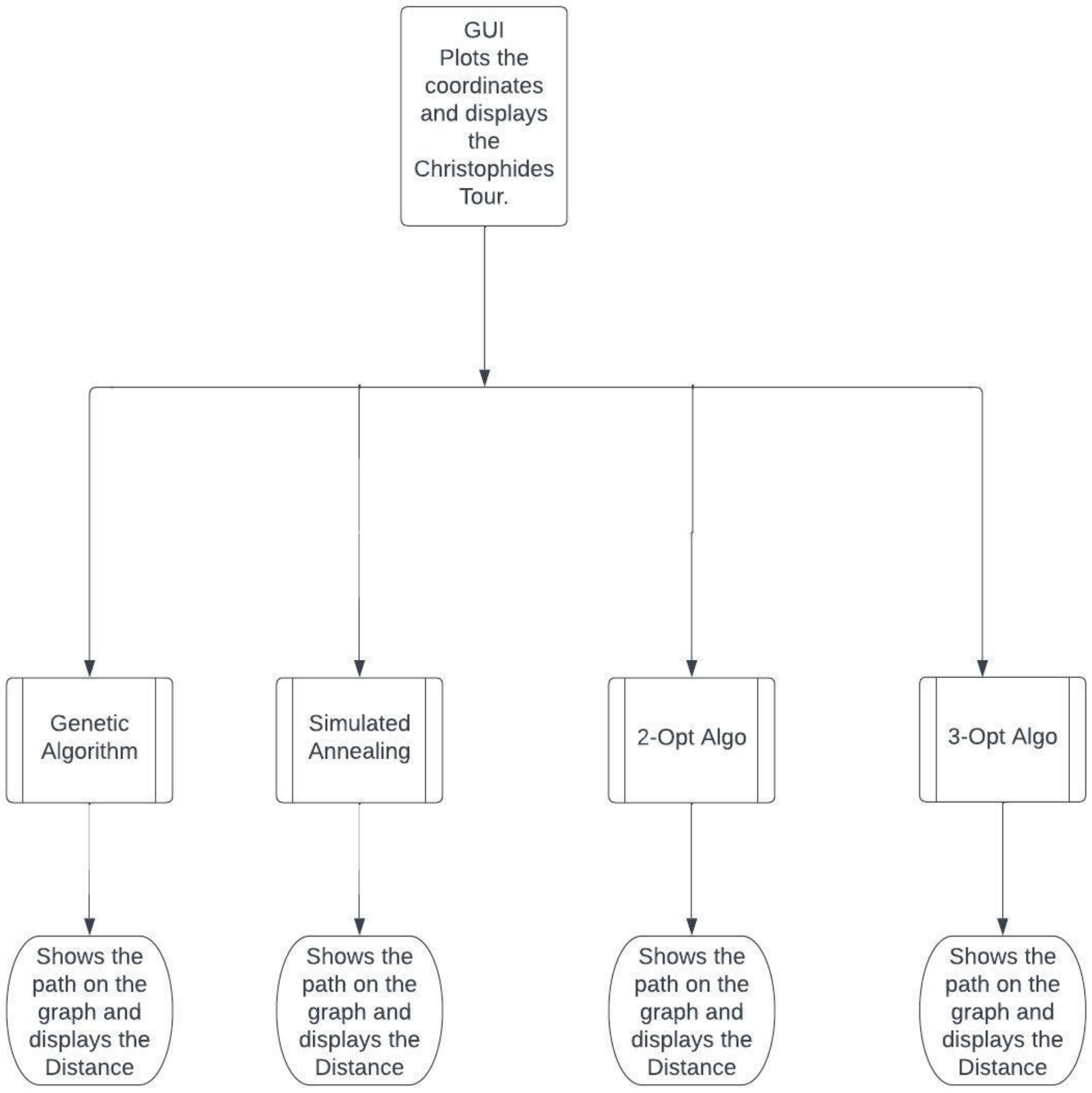
Invariants

1. Input format should be csv containing a list of vertices where each row contains id, latitude and longitude value
2. The graph created should have weighted undirected edges
3. Number of vertices in the MST with odd degree should be an even number
4. Blossom algorithm should compute matching only for the vertices with odd degree
5. Each vertex in the graph should have an even degree before it is given to Euler circuit as input
6. Every edge in the graph should be visited only once
7. An eulerian circuit built should visit every vertex only once and end at the same vertex where the cycle started

Flow Chart (Algorithms)


Travelling Salesman Problem
Read dataset from CSV file
Construct graph with vertices
Create Minimum Spanning Tree (MST) using Prim's algorithm
Use Haversine formula to calculate distance between vertices
Identify set of vertices with odd degrees
Apply Minimum-Weight Perfect Matching (MWPM) using Blossom's algorithm
Add MWPM edges to MST to get graph with even degree vertices
Create Eulerian Path
Create Eulerian Circuit
Convert Eulerian Circuit to Hamiltonian Circuit
Compute Hamiltonian Circuit as Christofides Graph and total distance
Apply Genetic Optimization to Christofides Graph
Apply Simulated Annealing Optimization to Christofides Graph
Apply 2-Opt Optimization to Christofides Graph
Apply 3-Opt Optimization to Christofides Graph
Build Genetic Algorithm Graph
Build Simulated Annealing Graph
Build 2-Opt Graph
Build 3-Opt Graph

Flow Chart(GUI)


GUI
Plots the coordinates and displays the Christophides Tour.
Genetic Algorithm
Simulated Annealing
2-Opt Algo
3-Opt Algo
Shows the path on the graph and displays the Distance
Shows the path on the graph and displays the Distance
Shows the path on the graph and displays the Distance
Shows the path on the graph and displays the Distance

Observations and Graphical Analysis

Minimum Cost Maximum Matching

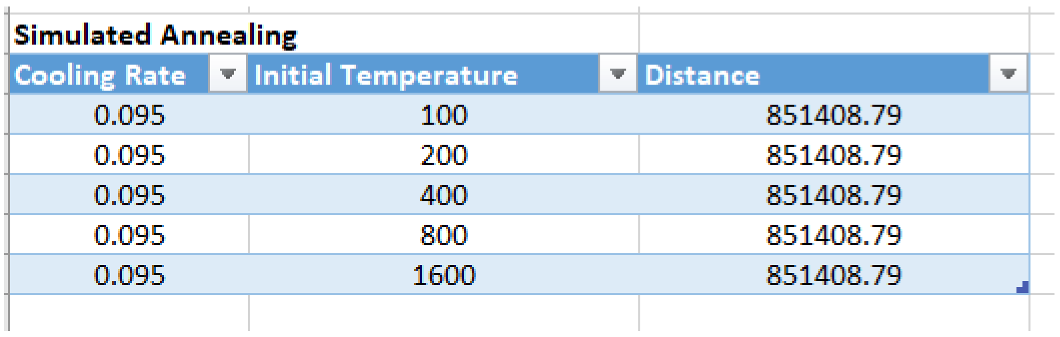
- After creating Minimum Spanning Tree (MST), we passed the resultant graph to Blossom's algorithm which comes under Minimum Cost Maximum Matching approach. This is our Christofides graph.
- Performing the operation provided a result which was around 30% longer than the distance of MST.
- In our case, MST distance was 650,787.41m and result of Christofides graph distance was 851,408.79m.
- The time complexity of the algorithm is O(n3) owing to blossom matching algorithm

Genetic Optimization

- We passed the result of Christofides graph to Genetic optimization. We did our benchmarking by varying population size and generation size.
- After varying the parameters, we found out that the best distance is observed when population is 1000 and number of generations is 1000.
- The overall best result of genetic optimization still provided significantly worse results. The best result of all the variations was still over 400% more than MST.
- Varying other parameters higher was also not the most optimal option as execution time was also increasing exponentially.
- Overall, Genetic optimization was the worse amongst all the optimization techniques we applied.

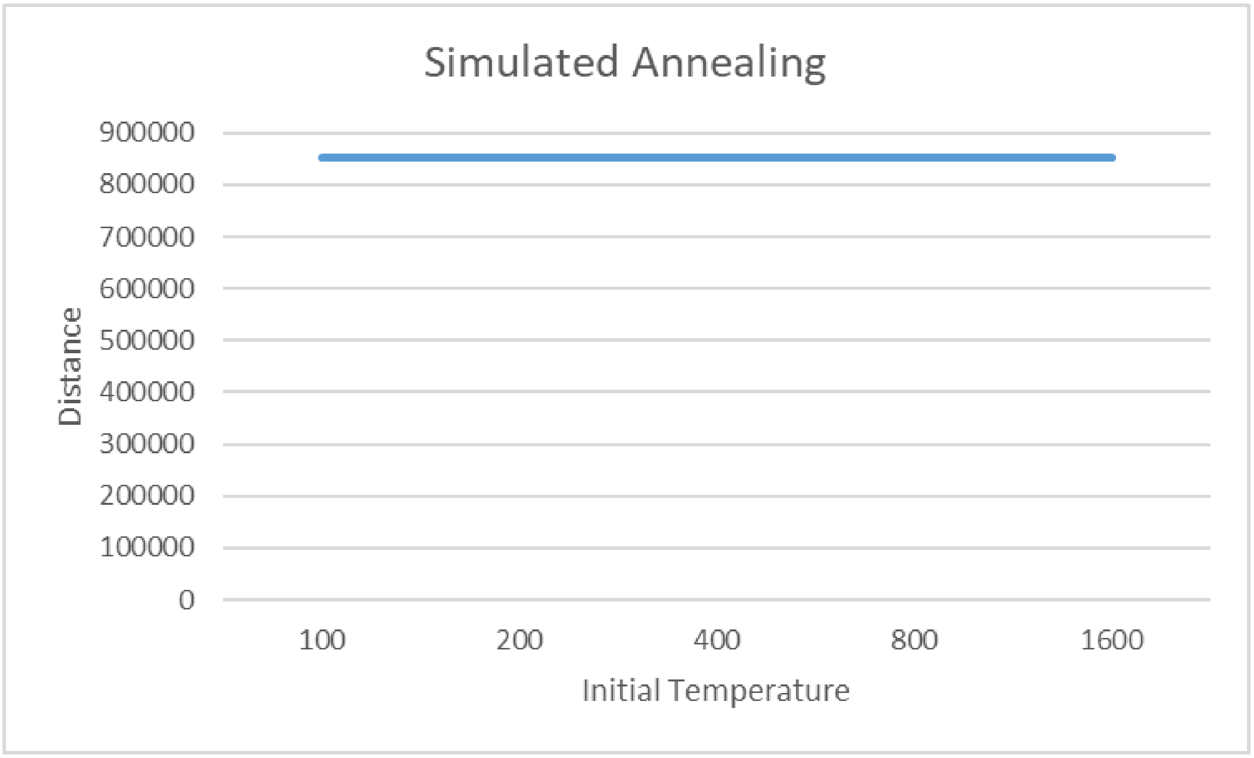
Simulated Annealing

- We passed the result of Christofides graph to Simulated Annealing optimization. We varied the initial temperature and cooling rate.
- After varying the parameters, we found out that the most ideal initial temperature was 1000 and cooling rate was 0.095.
- The optimization ran relatively quickly but the improvement in distance was very minute, in decimal points.
- In the below graph, you can see that the improvement is not significant, if we round the result to 2 decimal places then we can't even see the improvement. Thus, even with varying the parameters, Simulated Annealing was not the most optimal algorithm.



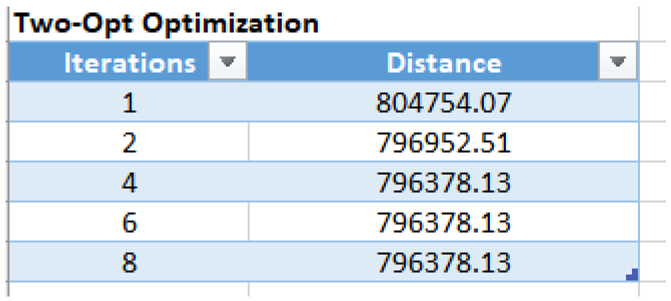
Simulated Annealing

Cooling Rate	Initial Temperature	Distance
0.095	100	851408.79
0.095	200	851408.79
0.095	400	851408.79
0.095	800	851408.79
0.095	1600	851408.79



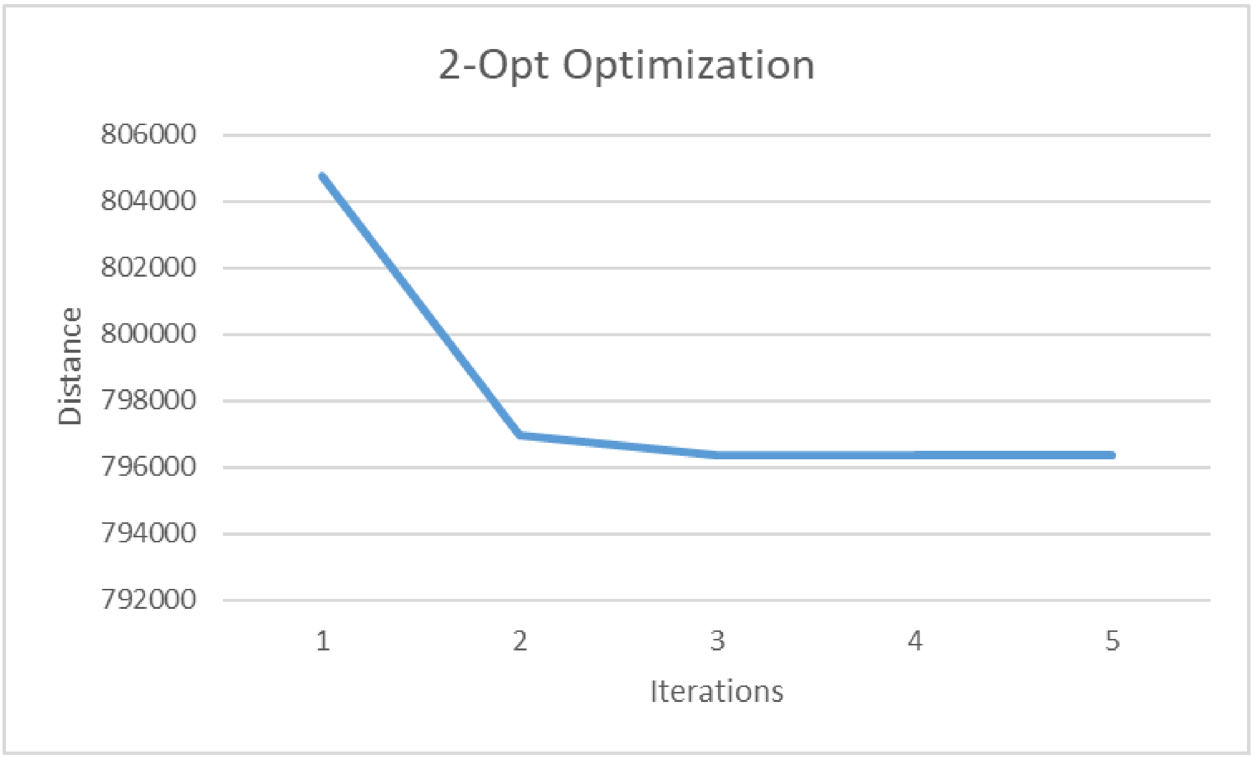
Two-Opt Optimization

- We passed the result of Christofides graph to 2-opt optimization. We ran the code until no further swaps were possible.
- Two-opt optimization gave the best result by far, bringing us closer to 22% longer than MST.
- The distance of MST was 650,787.41m whereas distance after two-opt came out to 796,378.13m.
- This optimization took approximately 3-4 minutes to run with the complete dataset.
- If we look at the below graph, we can clearly see that as the number of iterations increase the distance is improved or lowered, bringing us closer to the distance of MST. After a certain point, increasing the number of iterations didn't bring any significant improvement.



Two-Opt Optimization

Iterations	Distance
1	804754.07
2	796952.51
4	796378.13
6	796378.13
8	796378.13



Three-Opt Optimization

- We passed the result of Christofides graph to 3-opt optimization. We tried two different approaches.
- The first approaches tries to reduce time complexity by creating a gain formula which tries to find distance between various vertices. If this gain value is less than 0, then we apply exchange to exchange nodes and create new edges and provided the updated graph as the optimized version.
- The improvement using the first approach was very less (in decimal values) but it was able to provide results relatively quickly.
- In the second approach, we ran three nested loops bringing the time complexity to n^3. We swapped nodes and created a new graph, then we calculated distance of the new graph and compared it with the old one. If the new distance is better than old distance, we update the distance and store the new graph as optimized version.

- In the second approach, the code took a long time to produce results. We terminated the execution after 20 minutes and thus were not able to get the most optimized graph.
- At the end, we went with the first approach since it was able to provide results but we believe that the second approach can provide the best result barring time constraints.

Results and Mathematical Analysis

Haversine Formula

The haversine formula determines the great-circle distance between two points on a sphere given their longitudes and latitudes. Important in navigation, it is a special case of a more general formula in spherical trigonometry, the law of haversines, that relates the sides and angles of spherical triangles. It is computed as following -

d = 2r * arcsin(sqrt(sin^2((lat2-lat1)/2) + cos(lat1) * cos(lat2) * sin^2((lon2-lon1)/2)))

- d is the distance between the two points in the same units as r (usually kilometers)
- r is the radius of the sphere (usually the Earth's radius, which is approximately 6371 kilometers)
- lat1 and lat2 are the latitude coordinates of the two points, in radians
- lon1 and lon2 are the longitude coordinates of the two points, in radians

Simulated Annealing

The analysis of Simulated Annealing involved proving convergence properties, such as convergence to a global optimum under certain conditions. One important result is the convergence theorem, which states that under certain conditions the algorithm converges to global optimal with probability approaching 1.

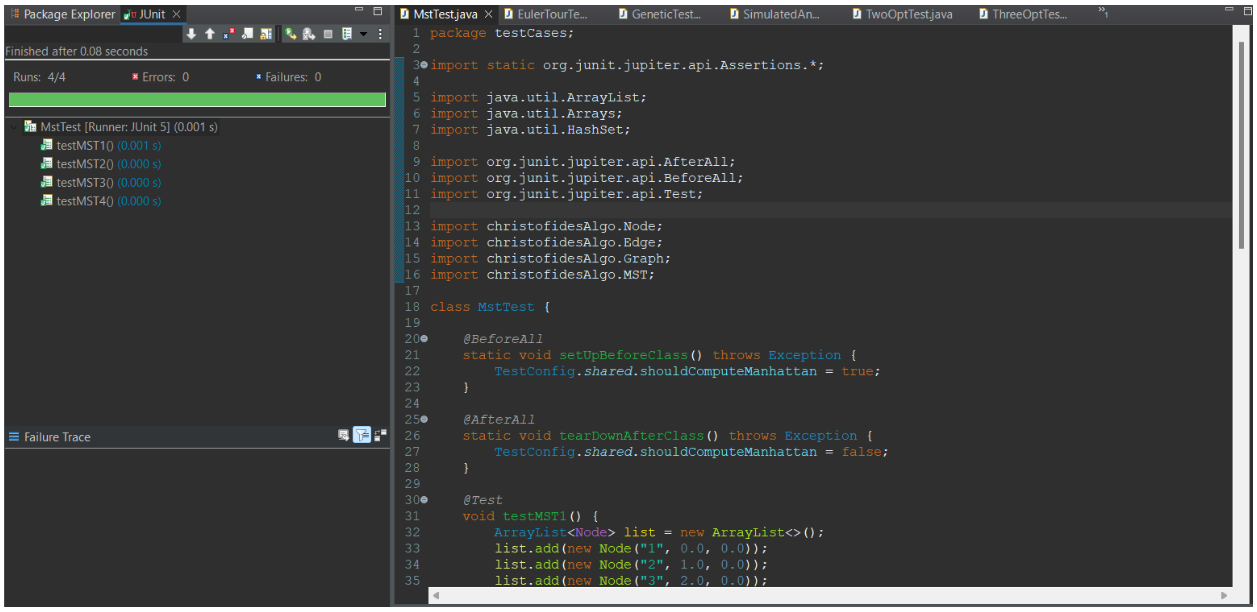
Following are some of the formulas related to simulated annealing:

1. The acceptance probability formula:
   $P = \exp(-\Delta E/T)$
2. The cooling schedule formula:
   $T(t+1) = \alpha T(t)$
3. The convergence rate formula:
   $\Pr(X \geq \varepsilon) \leq \exp(-c\varepsilon^p)$

The analysis of simulated annealing can be quite complex and involves a range of mathematical techniques, including probability theory, statistical mechanics, and optimization theory.

Unit tests

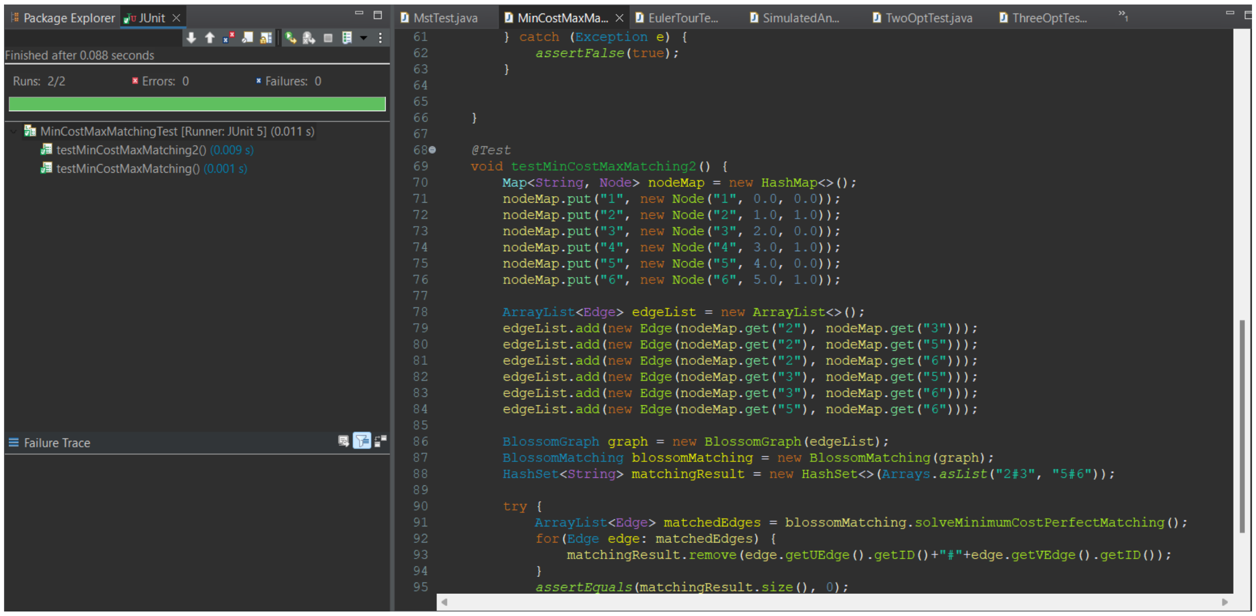
MST Test



```
Finished after 0.08 seconds
Runs: 4/4    Errors: 0    Failures: 0

MstTest [Runner: JUnit 5] (0.001 s)
    testMST1() (0.001 s)
    testMST2() (0.000 s)
    testMST3() (0.000 s)
    testMST4() (0.000 s)
```

```java
package testCases;

import static org.junit.jupiter.api.Assertions.*;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashSet;

import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;

import christofidesAlgo.Node;
import christofidesAlgo.Edge;
import christofidesAlgo.Graph;
import christofidesAlgo.MST;

class MstTest {

    @BeforeAll
    static void setUpBeforeClass() throws Exception {
        TestConfig.shared.shouldComputeManhattan = true;
    }

    @AfterAll
    static void tearDownAfterClass() throws Exception {
        TestConfig.shared.shouldComputeManhattan = false;
    }

    @Test
    void testMST1() {
        ArrayList<Node> list = new ArrayList<>();
        list.add(new Node("1", 0.0, 0.0));
        list.add(new Node("2", 1.0, 0.0));
        list.add(new Node("3", 2.0, 0.0));
```

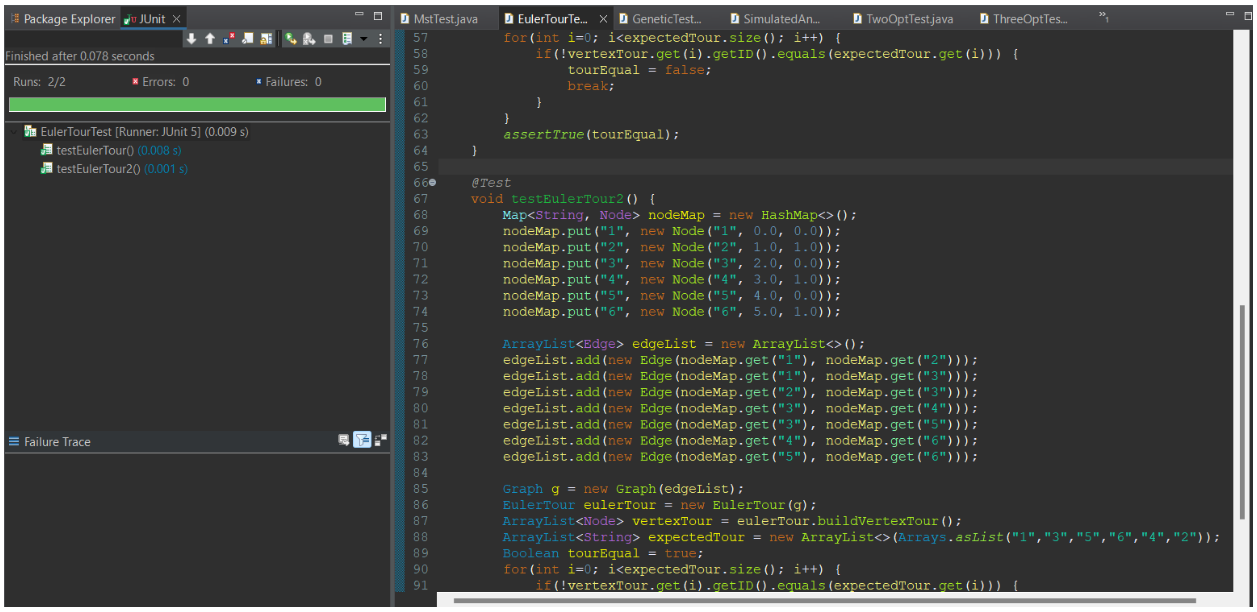
MinCostMaxMatching



```
Finished after 0.088 seconds
Runs: 2/2    Errors: 0    Failures: 0

MinCostMaxMatchingTest [Runner: JUnit 5] (0.011 s)
    testMinCostMaxMatching2() (0.009 s)
    testMinCostMaxMatching() (0.001 s)
```

```java
        } catch (Exception e) {
            assertFalse(true);
        }


    }

    @Test
    void testMinCostMaxMatching2() {
        Map<String, Node> nodeMap = new HashMap<>();
        nodeMap.put("1", new Node("1", 0.0, 0.0));
        nodeMap.put("2", new Node("2", 1.0, 1.0));
        nodeMap.put("3", new Node("3", 2.0, 0.0));
        nodeMap.put("4", new Node("4", 3.0, 1.0));
        nodeMap.put("5", new Node("5", 4.0, 0.0));
        nodeMap.put("6", new Node("6", 5.0, 1.0));

        ArrayList<Edge> edgeList = new ArrayList<>();
        edgeList.add(new Edge(nodeMap.get("2"), nodeMap.get("3")));
        edgeList.add(new Edge(nodeMap.get("2"), nodeMap.get("5")));
        edgeList.add(new Edge(nodeMap.get("2"), nodeMap.get("6")));
        edgeList.add(new Edge(nodeMap.get("3"), nodeMap.get("5")));
        edgeList.add(new Edge(nodeMap.get("3"), nodeMap.get("6")));
        edgeList.add(new Edge(nodeMap.get("5"), nodeMap.get("6")));

        BlossomGraph graph = new BlossomGraph(edgeList);
        BlossomMatching blossomMatching = new BlossomMatching(graph);
        HashSet<String> matchingResult = new HashSet<>(Arrays.asList("2#3", "5#6"));

        try {
            ArrayList<Edge> matchedEdges = blossomMatching.solveMinimumCostPerfectMatching();
            for(Edge edge: matchedEdges) {
                matchingResult.remove(edge.getUEdge().getID()+"#"+edge.getVEdge().getID());
            }
            assertEquals(matchingResult.size(), 0);
```

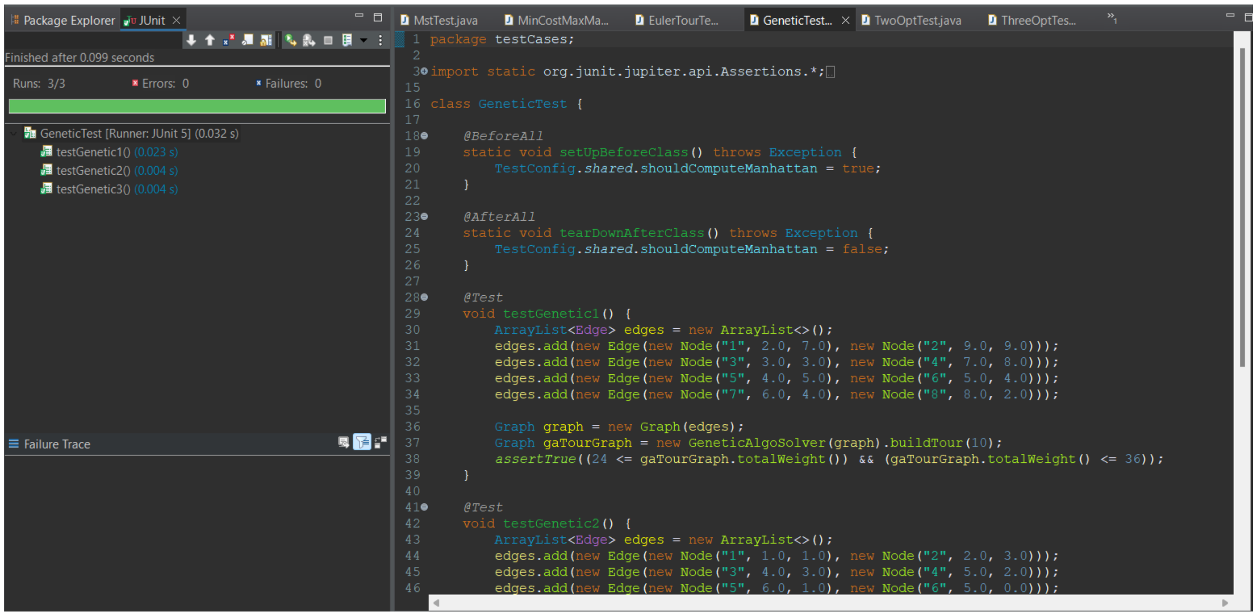
Euler Tour



```
Finished after 0.078 seconds
Runs: 2/2    Errors: 0    Failures: 0

EulerTourTest [Runner: JUnit 5] (0.009 s)
    testEulerTour() (0.008 s)
    testEulerTour2() (0.001 s)
```

```java
        for(int i=0; i<expectedTour.size(); i++) {
            if(!vertexTour.get(i).getID().equals(expectedTour.get(i))) {
                tourEqual = false;
                break;
            }
        }
        assertTrue(tourEqual);
    }

    @Test
    void testEulerTour2() {
        Map<String, Node> nodeMap = new HashMap<>();
        nodeMap.put("1", new Node("1", 0.0, 0.0));
        nodeMap.put("2", new Node("2", 1.0, 1.0));
        nodeMap.put("3", new Node("3", 2.0, 0.0));
        nodeMap.put("4", new Node("4", 3.0, 1.0));
        nodeMap.put("5", new Node("5", 4.0, 0.0));
        nodeMap.put("6", new Node("6", 5.0, 1.0));

        ArrayList<Edge> edgeList = new ArrayList<>();
        edgeList.add(new Edge(nodeMap.get("1"), nodeMap.get("2")));
        edgeList.add(new Edge(nodeMap.get("1"), nodeMap.get("3")));
        edgeList.add(new Edge(nodeMap.get("2"), nodeMap.get("3")));
        edgeList.add(new Edge(nodeMap.get("3"), nodeMap.get("4")));
        edgeList.add(new Edge(nodeMap.get("3"), nodeMap.get("5")));
        edgeList.add(new Edge(nodeMap.get("4"), nodeMap.get("6")));
        edgeList.add(new Edge(nodeMap.get("5"), nodeMap.get("6")));

        Graph g = new Graph(edgeList);
        EulerTour eulerTour = new EulerTour(g);
        ArrayList<Node> vertexTour = eulerTour.buildVertexTour();
        ArrayList<String> expectedTour = new ArrayList<>(Arrays.asList("1","3","5","6","4","2"));
        Boolean tourEqual = true;
        for(int i=0; i<expectedTour.size(); i++) {
            if(!vertexTour.get(i).getID().equals(expectedTour.get(i))) {
```

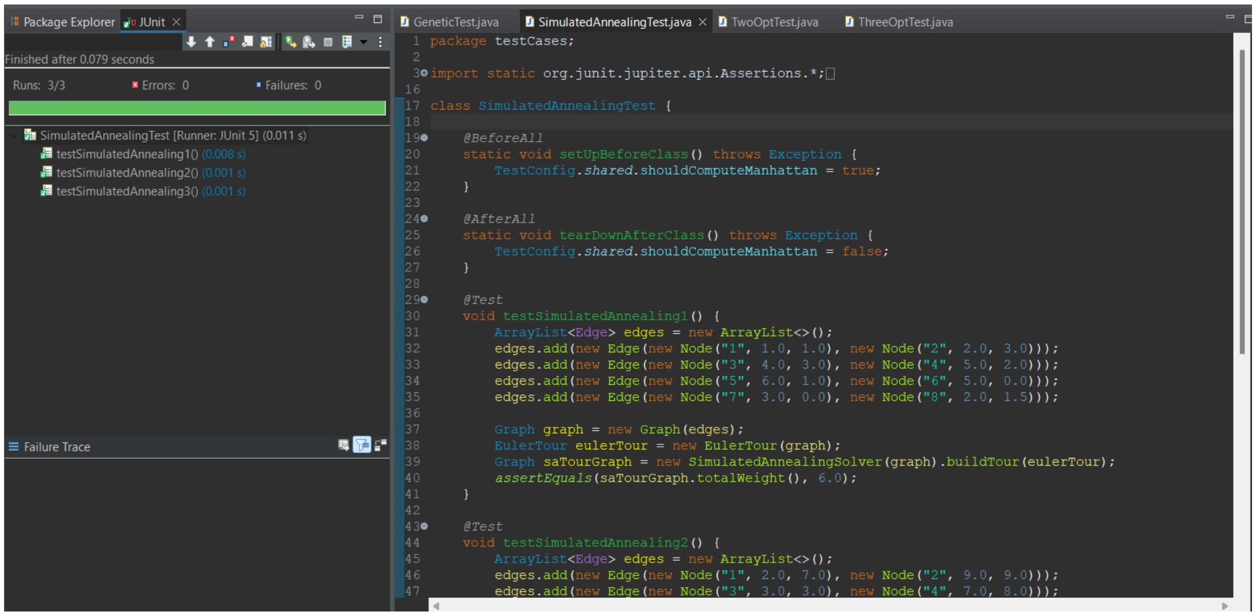
Genetic Optimization



```
Finished after 0.099 seconds
Runs: 3/3    Errors: 0    Failures: 0

GeneticTest [Runner: JUnit 5] (0.032 s)
    testGenetic1() (0.023 s)
    testGenetic2() (0.004 s)
    testGenetic3() (0.004 s)
```

```java
package testCases;

import static org.junit.jupiter.api.Assertions.*;

class GeneticTest {

    @BeforeAll
    static void setUpBeforeClass() throws Exception {
        TestConfig.shared.shouldComputeManhattan = true;
    }

    @AfterAll
    static void tearDownAfterClass() throws Exception {
        TestConfig.shared.shouldComputeManhattan = false;
    }

    @Test
    void testGenetic1() {
        ArrayList<Edge> edges = new ArrayList<>();
        edges.add(new Edge(new Node("1", 2.0, 7.0), new Node("2", 9.0, 9.0)));
        edges.add(new Edge(new Node("3", 3.0, 3.0), new Node("4", 7.0, 8.0)));
        edges.add(new Edge(new Node("5", 4.0, 5.0), new Node("6", 5.0, 4.0)));
        edges.add(new Edge(new Node("7", 6.0, 4.0), new Node("8", 8.0, 2.0)));

        Graph graph = new Graph(edges);
        Graph gaTourGraph = new GeneticAlgoSolver(graph).buildTour(10);
        assertTrue((24 <= gaTourGraph.totalWeight()) && (gaTourGraph.totalWeight() <= 36));
    }

    @Test
    void testGenetic2() {
        ArrayList<Edge> edges = new ArrayList<>();
        edges.add(new Edge(new Node("1", 1.0, 1.0), new Node("2", 2.0, 3.0)));
        edges.add(new Edge(new Node("3", 4.0, 3.0), new Node("4", 5.0, 2.0)));
        edges.add(new Edge(new Node("5", 6.0, 1.0), new Node("6", 5.0, 0.0)));
```

Simulated Annealing Optimization

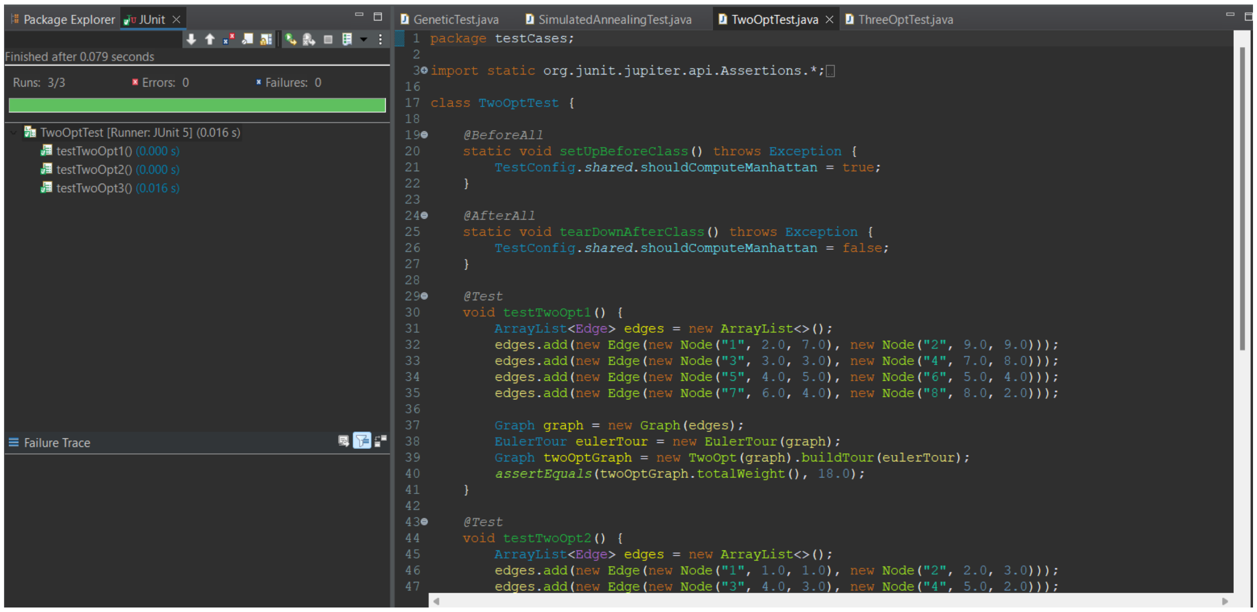


Finished after 0.079 seconds
Runs: 3/3 Errors: 0 Failures: 0

```
SimulatedAnnealingTest [Runner: JUnit 5] (0.011 s)
    testSimulatedAnnealing1() (0.008 s)
    testSimulatedAnnealing2() (0.001 s)
    testSimulatedAnnealing3() (0.001 s)
```

```java
package testCases;

import static org.junit.jupiter.api.Assertions.*;

class SimulatedAnnealingTest {

    @BeforeAll
    static void setUpBeforeClass() throws Exception {
        TestConfig.shared.shouldComputeManhattan = true;
    }

    @AfterAll
    static void tearDownAfterClass() throws Exception {
        TestConfig.shared.shouldComputeManhattan = false;
    }

    @Test
    void testSimulatedAnnealing1() {
        ArrayList<Edge> edges = new ArrayList<>();
        edges.add(new Edge(new Node("1", 1.0, 1.0), new Node("2", 2.0, 3.0)));
        edges.add(new Edge(new Node("3", 4.0, 3.0), new Node("4", 5.0, 2.0)));
        edges.add(new Edge(new Node("5", 6.0, 1.0), new Node("6", 5.0, 0.0)));
        edges.add(new Edge(new Node("7", 3.0, 0.0), new Node("8", 2.0, 1.5)));

        Graph graph = new Graph(edges);
        EulerTour eulerTour = new EulerTour(graph);
        Graph saTourGraph = new SimulatedAnnealingSolver(graph).buildTour(eulerTour);
        assertEquals(saTourGraph.totalWeight(), 6.0);
    }

    @Test
    void testSimulatedAnnealing2() {
        ArrayList<Edge> edges = new ArrayList<>();
        edges.add(new Edge(new Node("1", 2.0, 7.0), new Node("2", 9.0, 9.0)));
        edges.add(new Edge(new Node("3", 3.0, 3.0), new Node("4", 7.0, 8.0)));
```

Two Opt Optimization

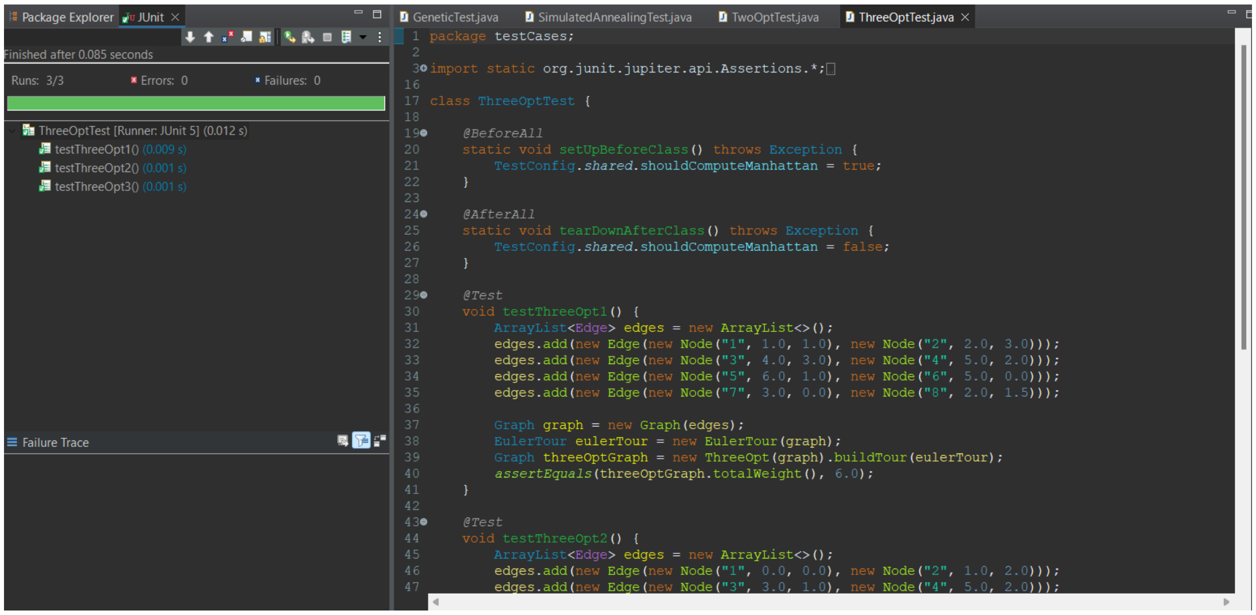


Finished after 0.079 seconds
Runs: 3/3 Errors: 0 Failures: 0

```
TwoOptTest [Runner: JUnit 5] (0.016 s)
    testTwoOpt1() (0.000 s)
    testTwoOpt2() (0.000 s)
    testTwoOpt3() (0.016 s)
```

```java
package testCases;

import static org.junit.jupiter.api.Assertions.*;

class TwoOptTest {

    @BeforeAll
    static void setUpBeforeClass() throws Exception {
        TestConfig.shared.shouldComputeManhattan = true;
    }

    @AfterAll
    static void tearDownAfterClass() throws Exception {
        TestConfig.shared.shouldComputeManhattan = false;
    }

    @Test
    void testTwoOpt1() {
        ArrayList<Edge> edges = new ArrayList<>();
        edges.add(new Edge(new Node("1", 2.0, 7.0), new Node("2", 9.0, 9.0)));
        edges.add(new Edge(new Node("3", 3.0, 3.0), new Node("4", 7.0, 8.0)));
        edges.add(new Edge(new Node("5", 4.0, 5.0), new Node("6", 5.0, 4.0)));
        edges.add(new Edge(new Node("7", 6.0, 4.0), new Node("8", 8.0, 2.0)));

        Graph graph = new Graph(edges);
        EulerTour eulerTour = new EulerTour(graph);
        Graph twoOptGraph = new TwoOpt(graph).buildTour(eulerTour);
        assertEquals(twoOptGraph.totalWeight(), 18.0);
    }

    @Test
    void testTwoOpt2() {
        ArrayList<Edge> edges = new ArrayList<>();
        edges.add(new Edge(new Node("1", 1.0, 1.0), new Node("2", 2.0, 3.0)));
        edges.add(new Edge(new Node("3", 4.0, 3.0), new Node("4", 5.0, 2.0)));
```

Three Opt Optimization

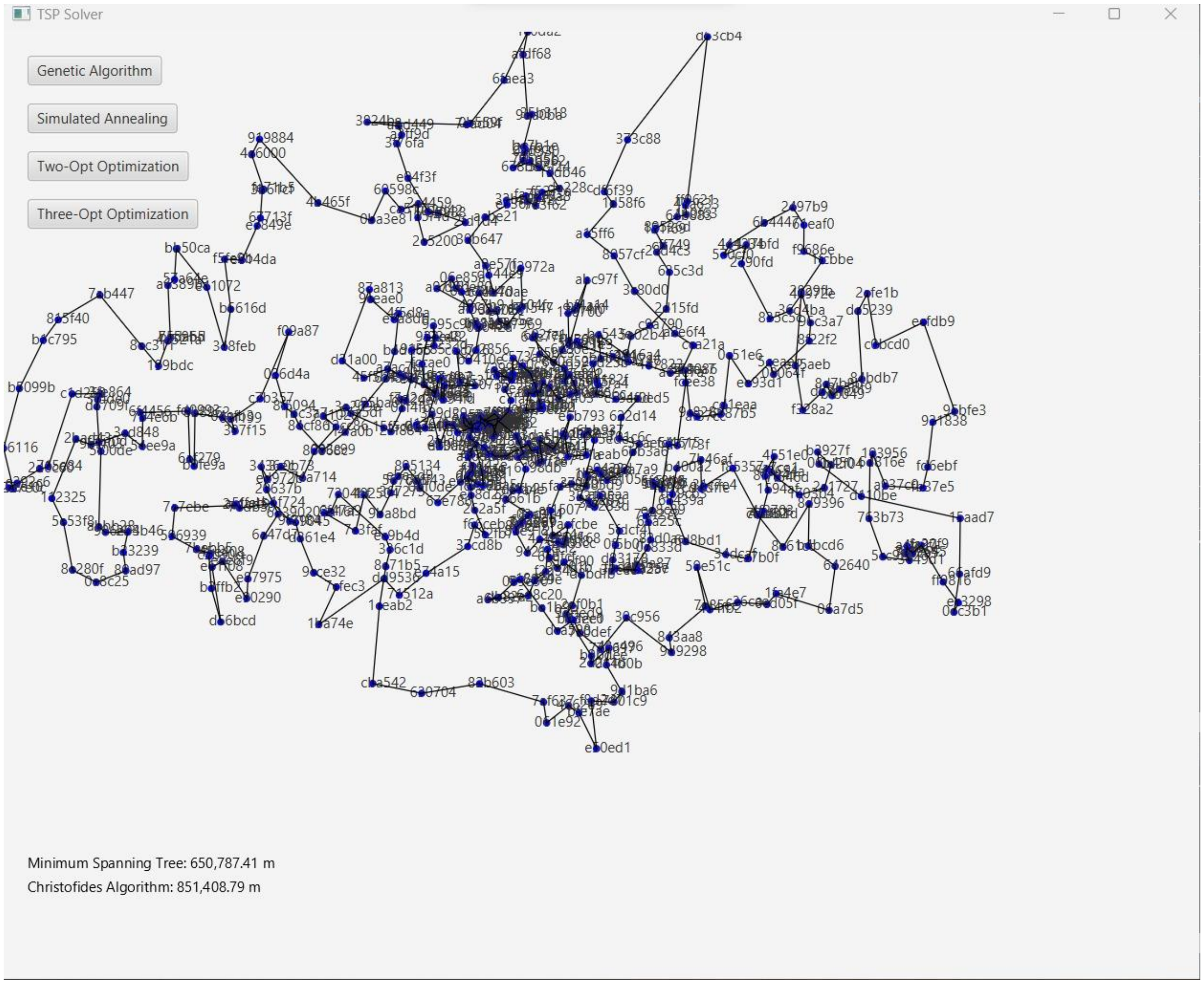


```
Finished after 0.085 seconds
Runs: 3/3        Errors: 0        Failures: 0

ThreeOptTest [Runner: JUnit 5] (0.012 s)
    testThreeOpt1() (0.009 s)
    testThreeOpt2() (0.001 s)
    testThreeOpt3() (0.001 s)
```

```java
package testCases;

import static org.junit.jupiter.api.Assertions.*;

class ThreeOptTest {

    @BeforeAll
    static void setUpBeforeClass() throws Exception {
        TestConfig.shared.shouldComputeManhattan = true;
    }

    @AfterAll
    static void tearDownAfterClass() throws Exception {
        TestConfig.shared.shouldComputeManhattan = false;
    }

    @Test
    void testThreeOpt1() {
        ArrayList<Edge> edges = new ArrayList<>();
        edges.add(new Edge(new Node("1", 1.0, 1.0), new Node("2", 2.0, 3.0)));
        edges.add(new Edge(new Node("3", 4.0, 3.0), new Node("4", 5.0, 2.0)));
        edges.add(new Edge(new Node("5", 6.0, 1.0), new Node("6", 5.0, 0.0)));
        edges.add(new Edge(new Node("7", 3.0, 0.0), new Node("8", 2.0, 1.5)));

        Graph graph = new Graph(edges);
        EulerTour eulerTour = new EulerTour(graph);
        Graph threeOptGraph = new ThreeOpt(graph).buildTour(eulerTour);
        assertEquals(threeOptGraph.totalWeight(), 6.0);
    }

    @Test
    void testThreeOpt2() {
        ArrayList<Edge> edges = new ArrayList<>();
        edges.add(new Edge(new Node("1", 0.0, 0.0), new Node("2", 1.0, 2.0)));
        edges.add(new Edge(new Node("3", 3.0, 1.0), new Node("4", 5.0, 2.0)));
```

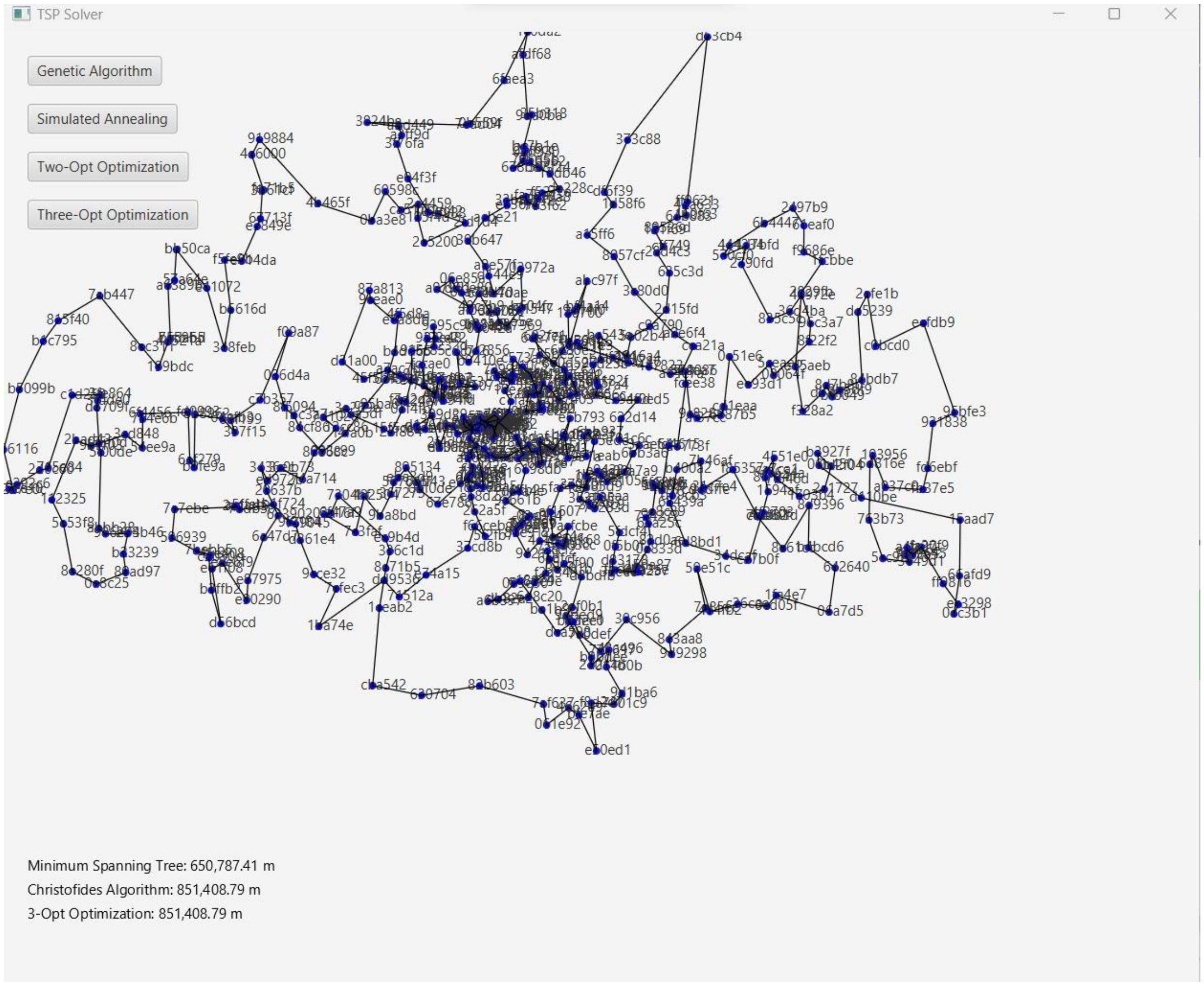
Output:

1. MST and Christofides



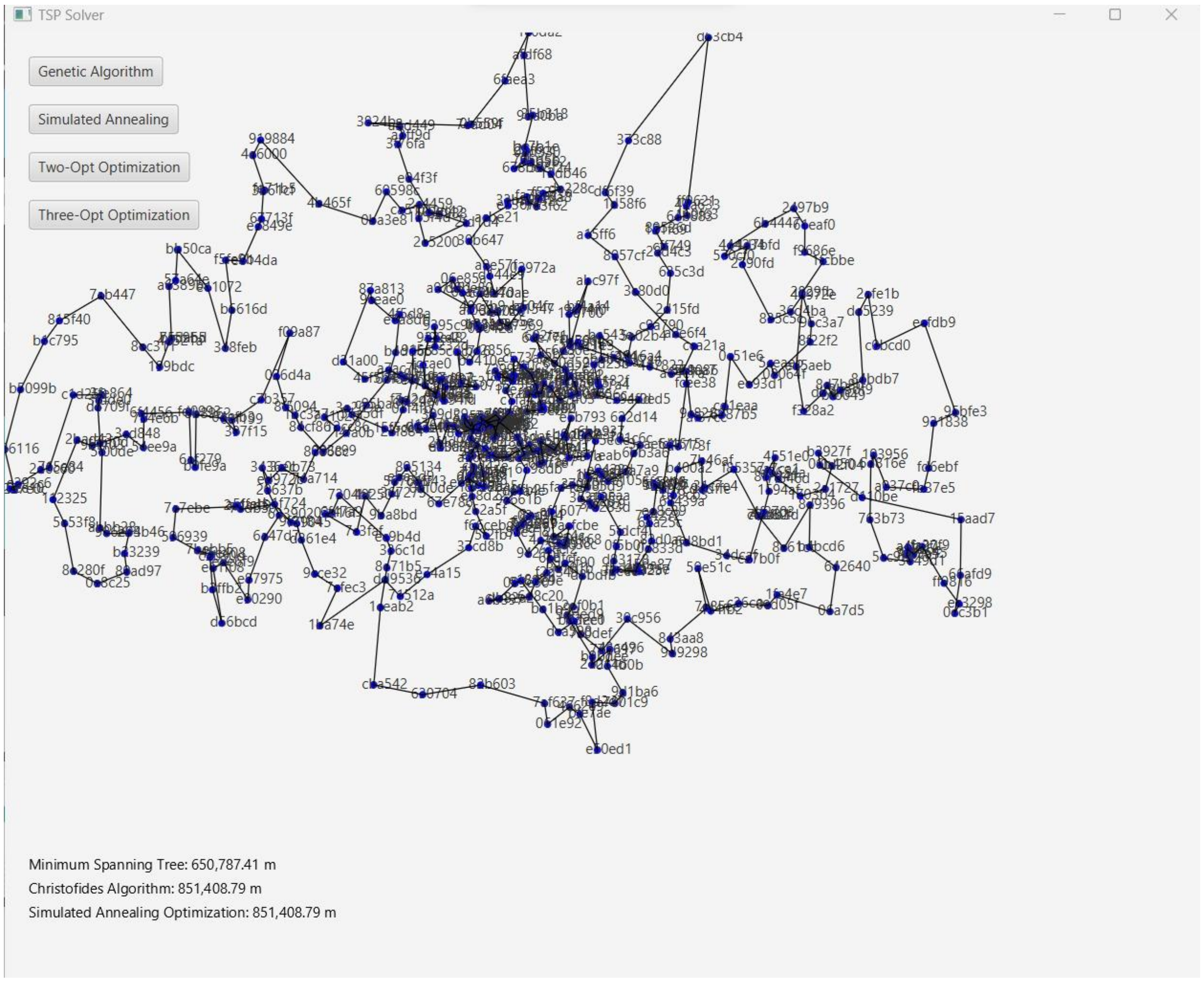
Minimum Spanning Tree: 650,787.41 m
Christofides Algorithm: 851,408.79 m

2. 3 Opt



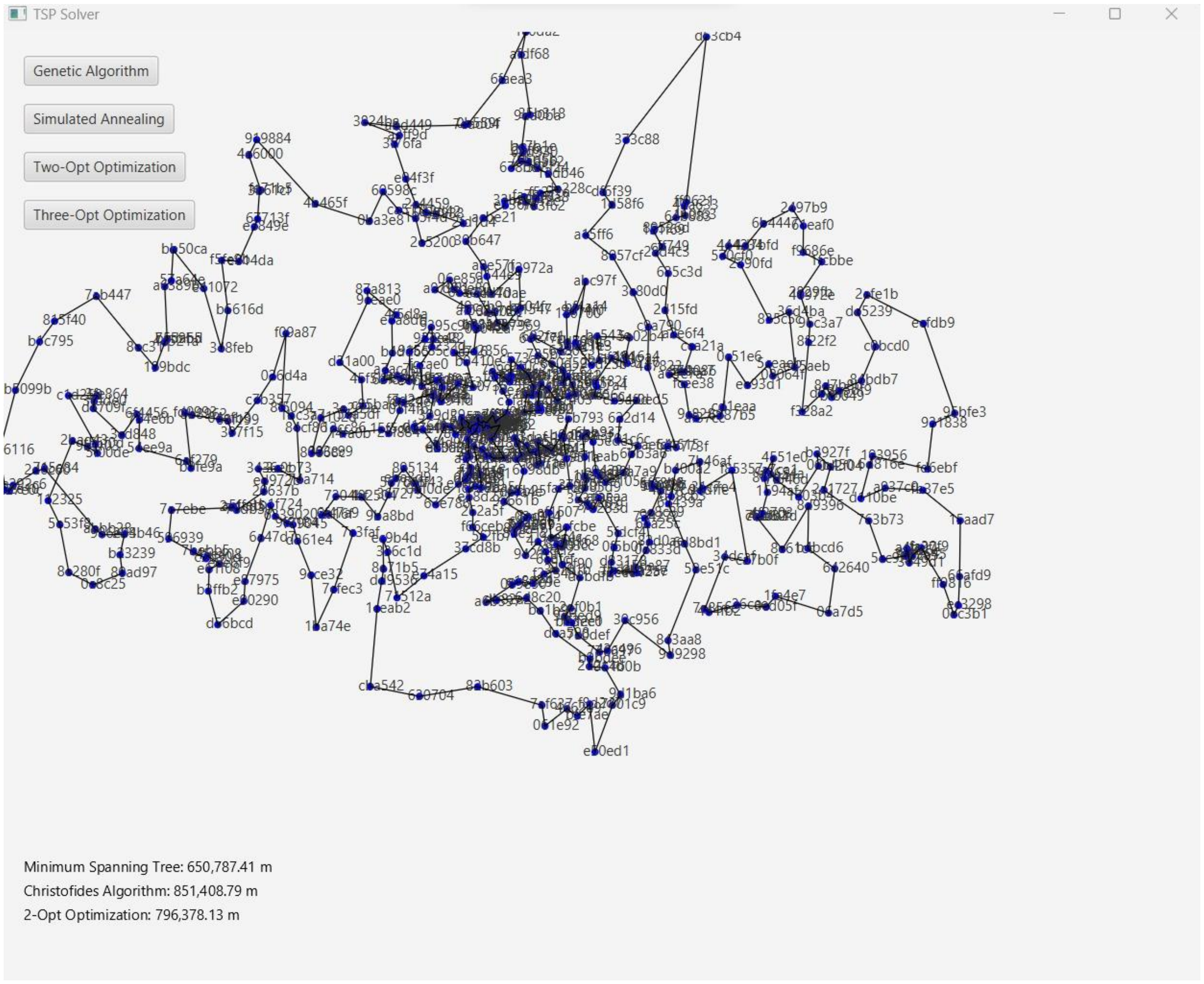
Minimum Spanning Tree: 650,787.41 m
Christofides Algorithm: 851,408.79 m
3-Opt Optimization: 851,408.79 m

3. Simulated Annealing



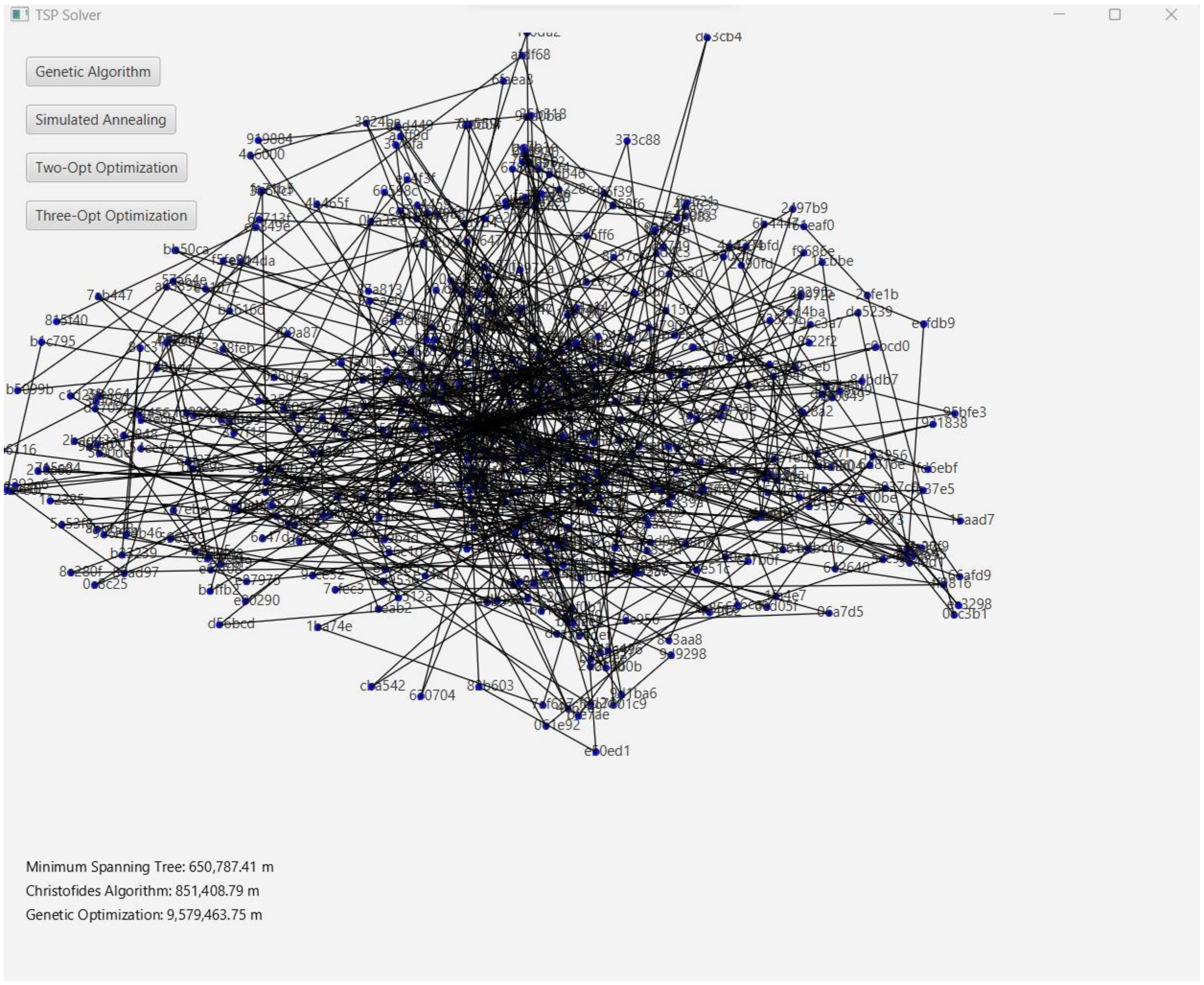
Minimum Spanning Tree: 650,787.41 m
Christofides Algorithm: 851,408.79 m
Simulated Annealing Optimization: 851,408.79 m

4. 2 Opt



Minimum Spanning Tree: 650,787.41 m
Christofides Algorithm: 851,408.79 m
2-Opt Optimization: 796,378.13 m

5. Genetic Algorithm



Minimum Spanning Tree: 650,787.41 m
Christofides Algorithm: 851,408.79 m
Genetic Optimization: 9,579,463.75 m

Conclusion

The present report delves into an exhaustive investigation of the efficacy of the Christofides algorithm, alongside its optimization methods, encompassing tactical techniques such as 3-opt and 2-opt improvement, as well as strategic approaches like simulated annealing and Genetic Algorithm, in obtaining superior-quality approximate solutions to the widely studied traveling salesman problem. The empirical findings reveal that the Christofides method yields high-quality approximate solutions, boasting a worst-case performance guarantee of 1.5 times the optimal solution. Notably, the incorporation of optimization methods, such as simulated-annealing and 3-opt are slightly improving the results while 2-opt gives significant tour improvements. Additionally, usage of the Blossom Algorithm for minimum weight maximum matching in Christofides by itself provides a very optimized tour.

Additionally, the utilization of simulated annealing and ant colony optimization showcases promising results, although requiring meticulous parameter tuning. In our current implementation, we did not see good results with Genetic algorithm optimization which at times worsened the optimized path.

To ensure the program's accuracy and reliability, we adopted object-oriented programming principles and conducted rigorous unit testing in Java using JUnit.

With our investigation, we have observed that a combination of Christofides algorithm along with 2-OPT optimization technique gives very good results when tested with the Crime Dataset given. With input of 585 points, we have obtained **Minimum Spanning Tree(MST)** distance as **650,787m** and the optimized tour with Christofides and 2-OPT optimization giving a total tour distance of **796,378m** which is approximately just **1.223x or 22.3%** higher than the MST distance.

In conclusion, our meticulously designed and implemented Java-based solution presents a practical and robust tool for tackling the traveling salesman problem.

References

1. Reducible. (2022, July 26). The Traveling Salesman Problem: When Good Enough Beats Perfect [Video]. Retrieved from https://www.youtube.com/watch?v=GiDsjIBOVoA&ab_channel=Reducible
2. Efficient Algorithms for Finding Maximum Matching in Graphs https://www.cs.kent.edu/~dragan/GraphAn/p23-galil.pdf
3. Blossom Algorithm https://brilliant.org/wiki/blossom-algorithm/
4. Algorithms for Maximum Cardinality Matching and Minimum Cost Perfect Matching Problems in General Graphs https://github.com/dilsonpereira/Minimum-Cost-Perfect-Matching
5. Github Repository (TSPVisualizationFX) mhrimaz/TSPVisualizationFX: Traveling Salesman Problem Using Genetic Algorithm and JavaFX Visualization (github.com)
6. Github Repository (jackspyder / 2-opt) 2-opt/TwoOpt.java at master · jackspyder/2-opt (github.com)