INFO 6205 Spring 2023 Project

Traveling Salesman

1. Introduction

The Traveling Salesman Problem (TSP) is a classic problem in computer science, which involves finding the shortest possible route that visits a set of cities exactly once and returns to the starting city. The problem is NP-hard, which means that finding the exact solution for large problems is infeasible. However, there are various approximation algorithms that can produce near-optimal solutions. One such algorithm is the Christofides algorithm, which is the subject of this report.

1. Aim

The aim of this report is to describe how to find a solution to the Traveling Salesman Problem using the Christofides algorithm.

2. Approach

Our approach to solving the traveling salesman problem using Christofides algorithm involves several key steps including Minimum Spanning Tree, Odd Degree Vertices, Perfect Matching, Eulerian Circuit, and Hamiltonian Circuit. These steps are used to create an efficient path that visits each vertex exactly once. By following this approach, we can obtain near-optimal solutions to TSP problems within a reasonable amount of time.

2. Program

1. Data Structures & classes

Data Structures

- ArrayList
- HashMap
- Stack
- HashSet
- Array

Classes

- ChristofidesAlgorithm
- CircleData
- Edge
- Graph
- GraphFromCSV
- Hamiltonian

- HierholzerAlgorithm
- MinimumWeightPerfectMatching
- MSThandlerclass
- OddDegreehandlerclass
- OddDegreeVertices
- PrimsMST
- SimulatedAnnealing
- ThreeOpt
- TSPUI
- TwoOpt
- Ant
- AntColonyOptimization

2. Algorithm

- Prims Algorithm
- Hierholzer Algorithm
- Hamiltonian Algorithm
- 2-Opt Algorithm
- 3-Opt Algorithm
- Simulated Annealing
- Ant Colony Optimization

3. Invariants

There are several invariants that are maintained throughout the algorithm's execution. These invariants are important properties or conditions that remain unchanged during the algorithm's steps, and they help ensure the correctness and effectiveness of the algorithm. The main invariants in the Christofides algorithm are:

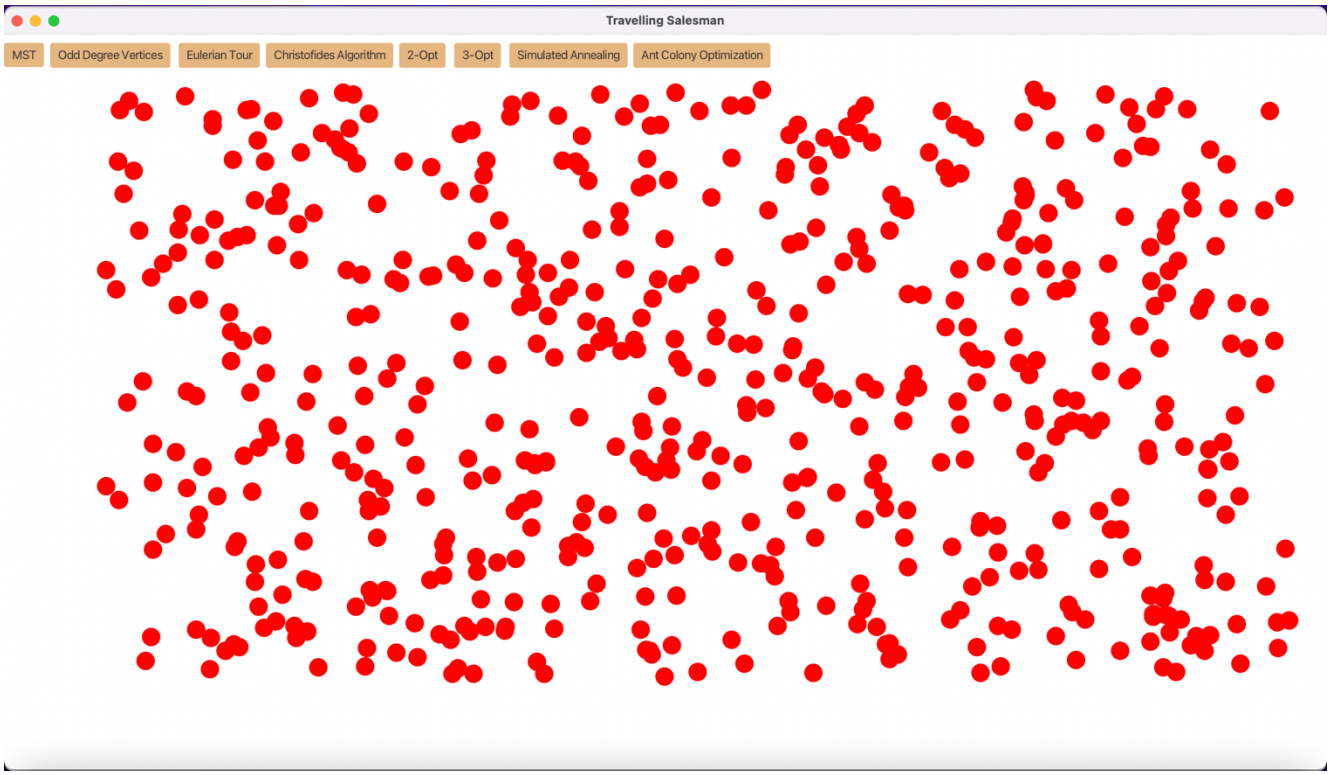
1. **Graph Invariant**: The input graph remains unchanged throughout the algorithm. This means that the original graph, which represents the cities and their distances or costs, is not modified during the algorithm's execution. The algorithm only constructs additional data structures, such as subgraphs or minimum spanning trees, to assist in finding a solution, but the original graph is not altered.
2. **Eulerian Graph Invariant:** The algorithm requires the input graph to be an undirected graph that has an Eulerian cycle, which is a cycle that traverses all edges exactly once and returns to the starting vertex. This invariant is maintained throughout the algorithm, as the algorithm constructs an Eulerian cycle as part of its solution
3. **Minimum Spanning Tree (MST) Invariant:** The algorithm constructs a minimum spanning tree (MST) of the input graph, which is a tree that spans all vertices with the minimum possible total edge weight. This MST is used to find an

approximate solution for the TSP. The invariant is maintained throughout the algorithm, and the constructed MST is not modified once it is generated.

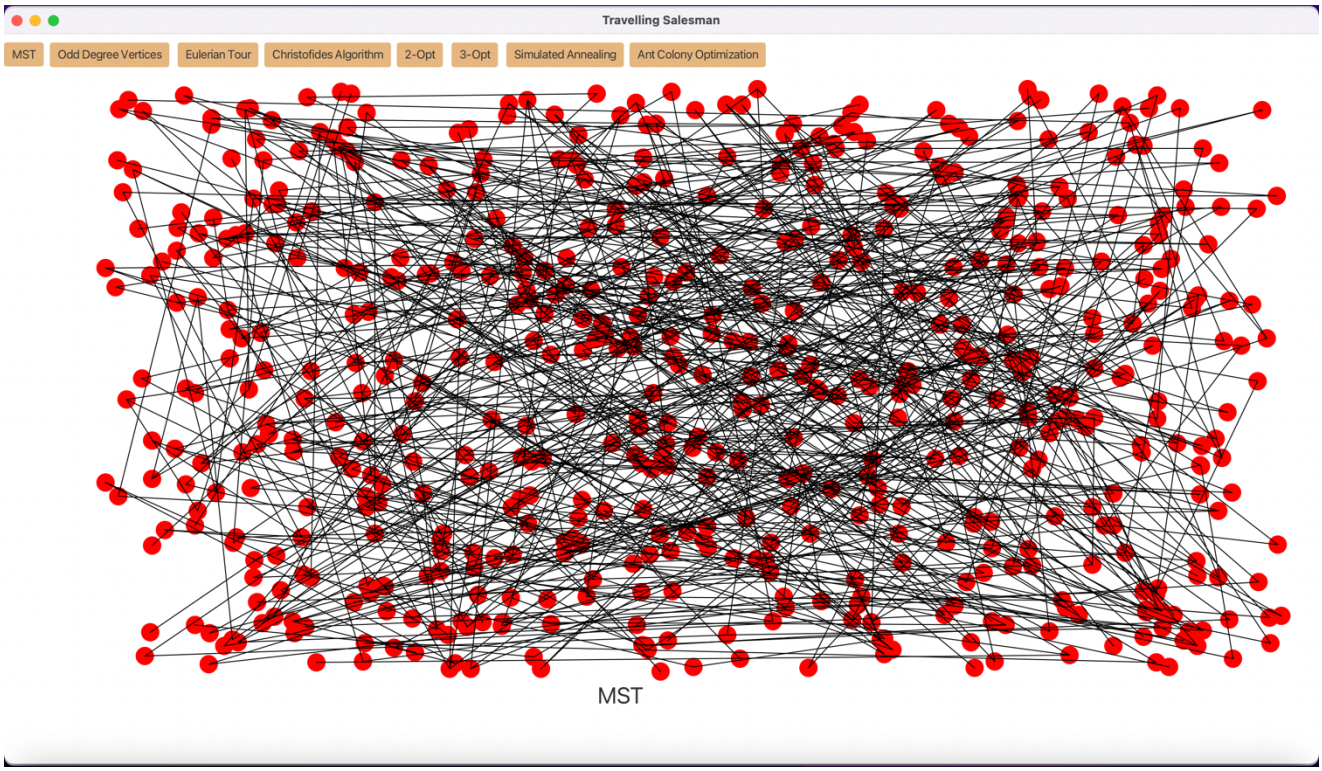
4. **Matching Invariant:** The algorithm constructs a perfect matching (a set of edges that do not share any vertices) on the subgraph formed by the odd-degree vertices of the MST. This perfect matching is used to create an Eulerian cycle in the final solution. The invariant is maintained throughout the algorithm, and the constructed matching is not altered once it is generated.
5. **Solution Invariant:** The algorithm maintains a solution to the TSP problem that is a combination of the Eulerian cycle obtained from the minimum spanning tree and the additional edges added from the perfect matching. This solution is iteratively improved by removing shortcut edges and finding a minimum-weight Hamiltonian cycle, which is a cycle that visits all vertices exactly once. The solution invariant is maintained throughout the algorithm, and the solution is refined until a near-optimal solution is obtained.

3. Flow Charts (UI Flow)

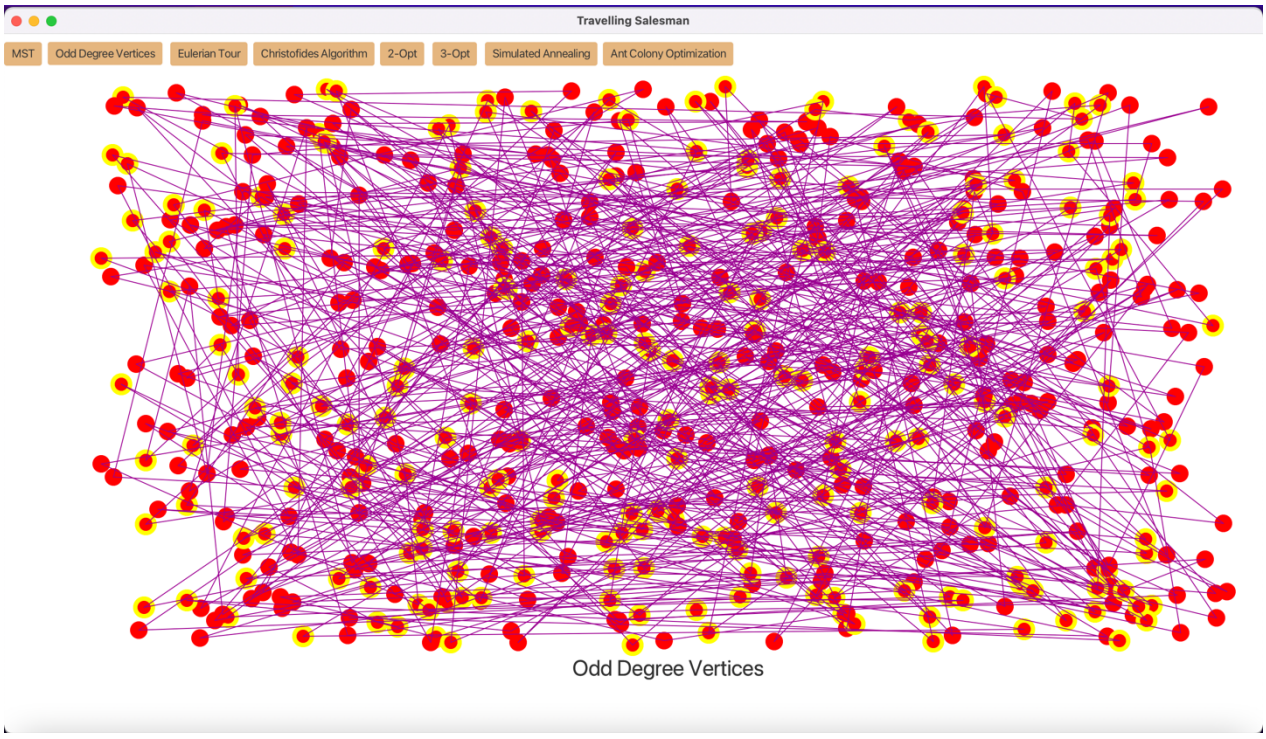
- On running the TSPUI.java file, Travelling Salesman graphical user interface will be opened which will have button to run all algorithms. On click of the buttons, each algorithm will be executed and the route connecting the vertices will be displayed.



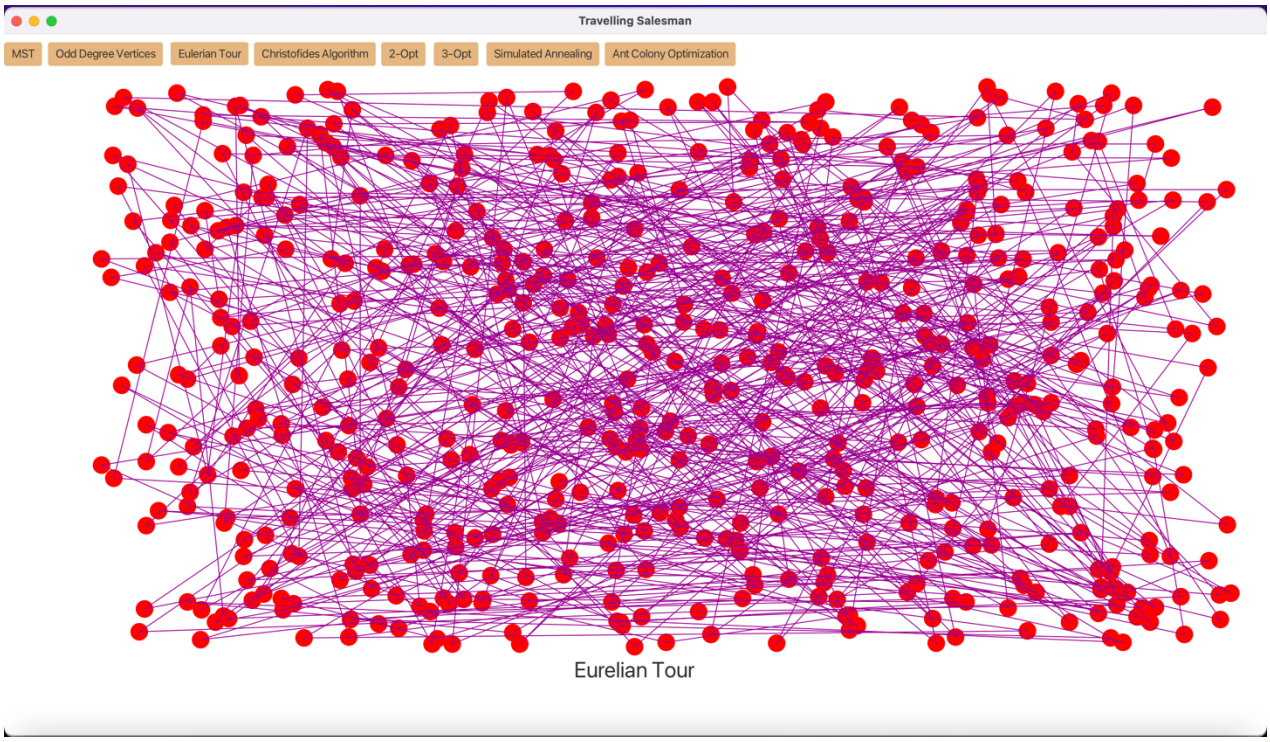
- Below is the minimum spanning route displayed when user clicks on MST Button.



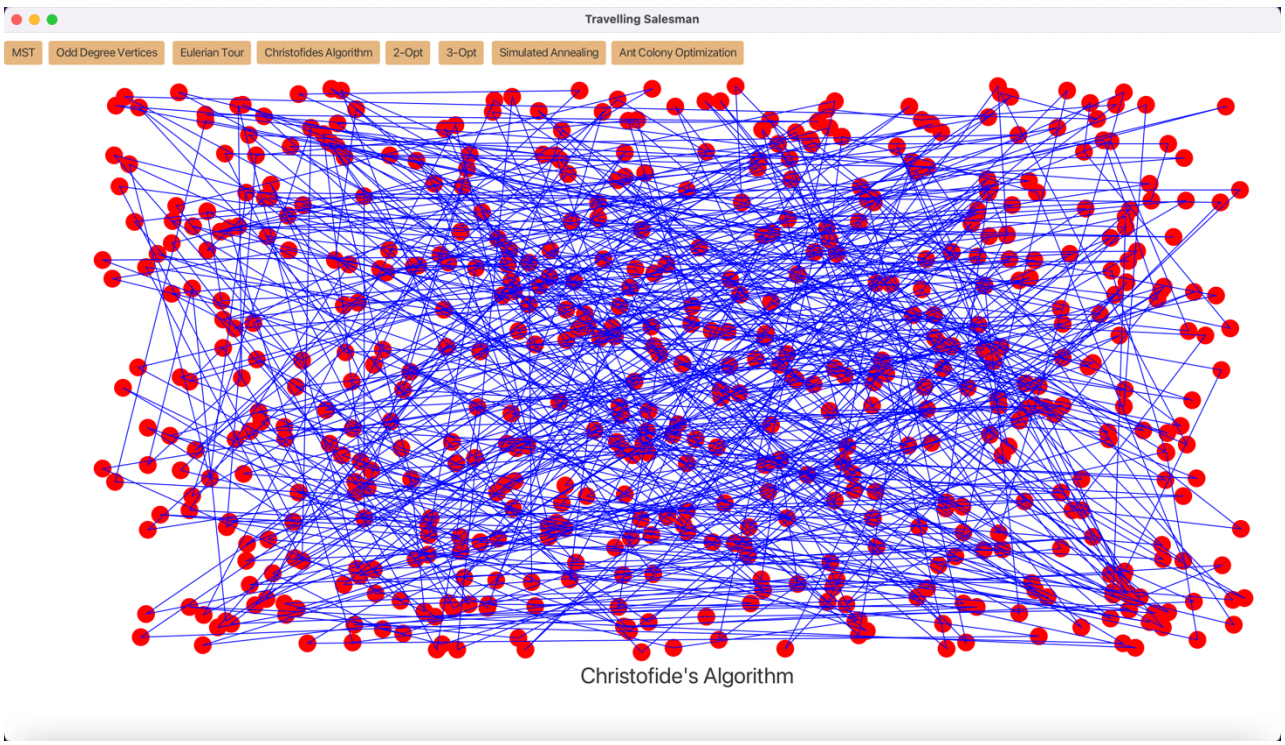
- Below are the vertices highlighted in yellow when user clicks on Odd Degree Vertices Button.



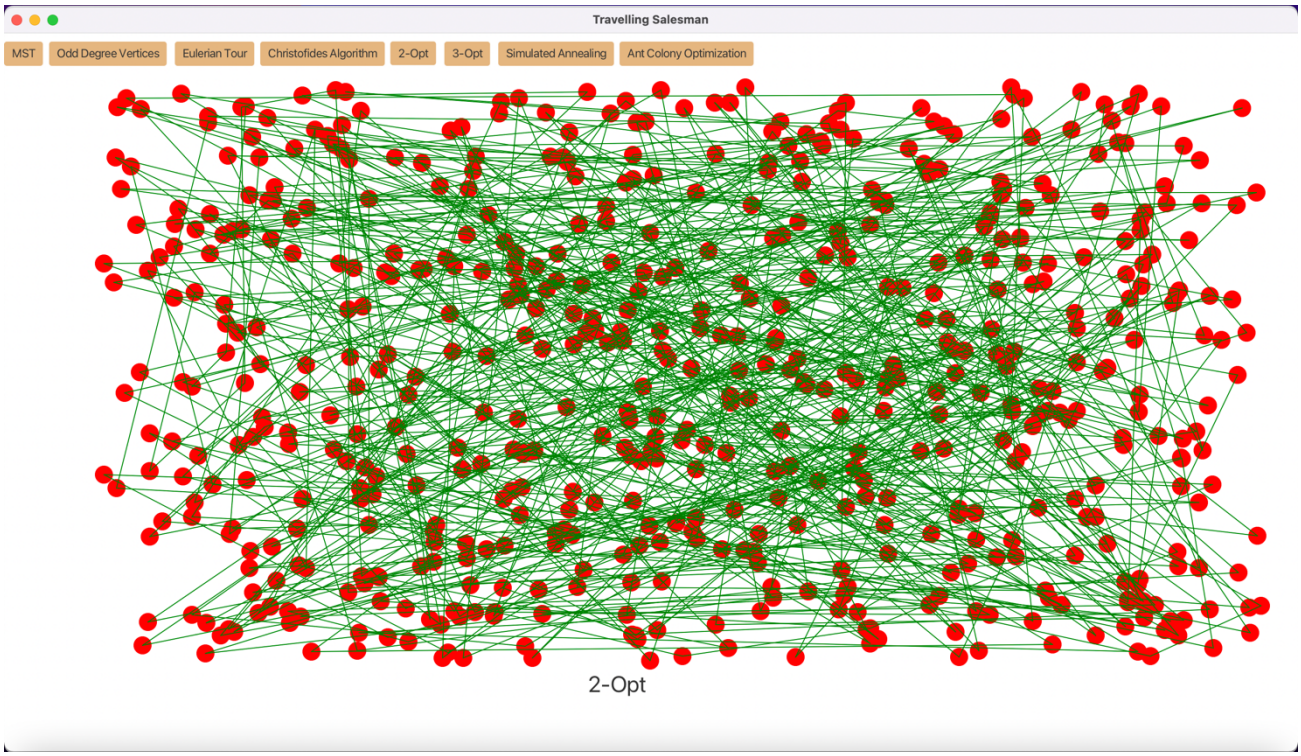
- Below is the Eulerian Tour route displayed when user clicks on Eulerian Tour Button.



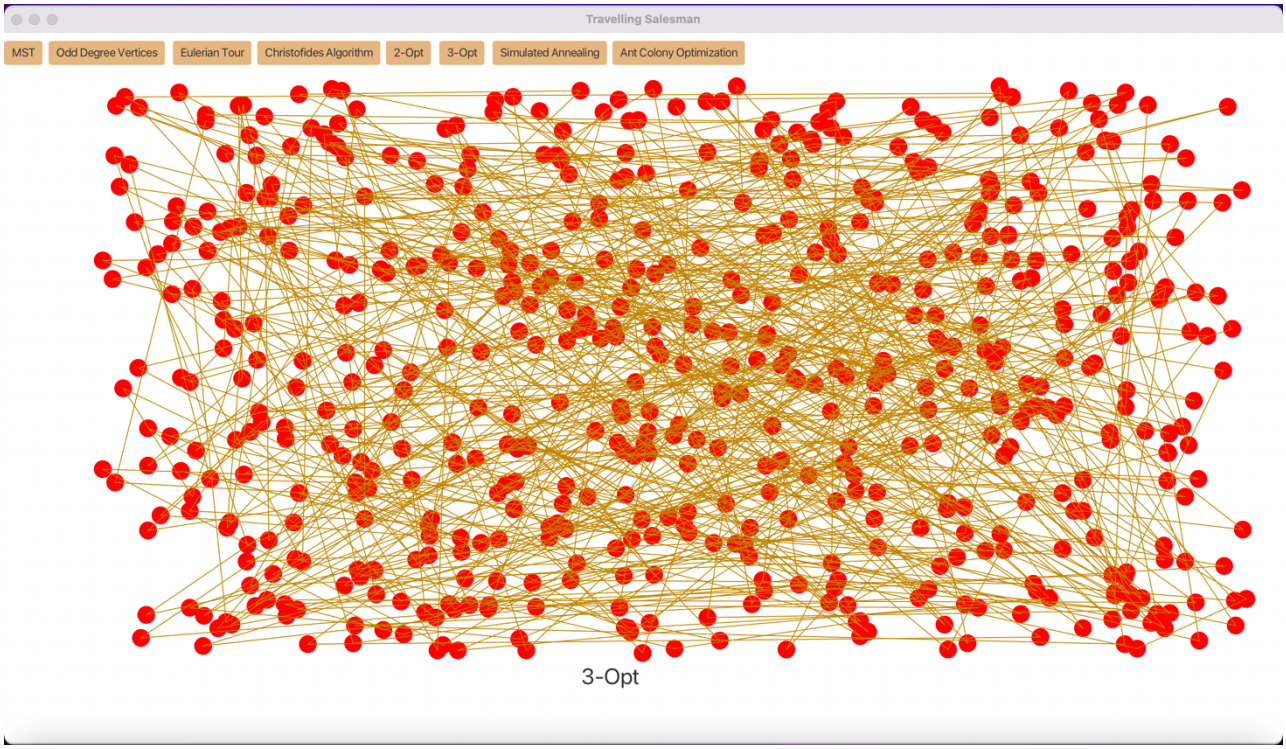
- Below is the Christofides's Algorithm route displayed when user clicks on Christofides's Algorithm Button.



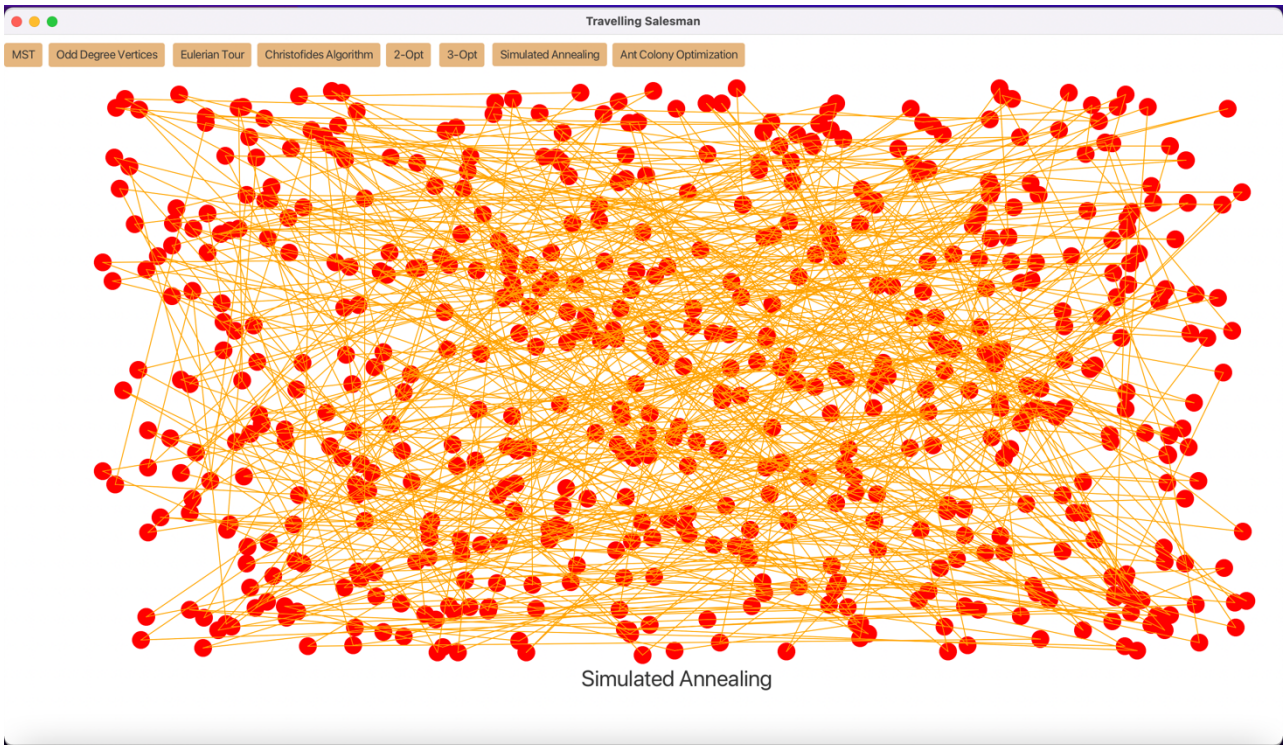
- Below is the 2-Opt route displayed when user clicks on 2-Opt Button.



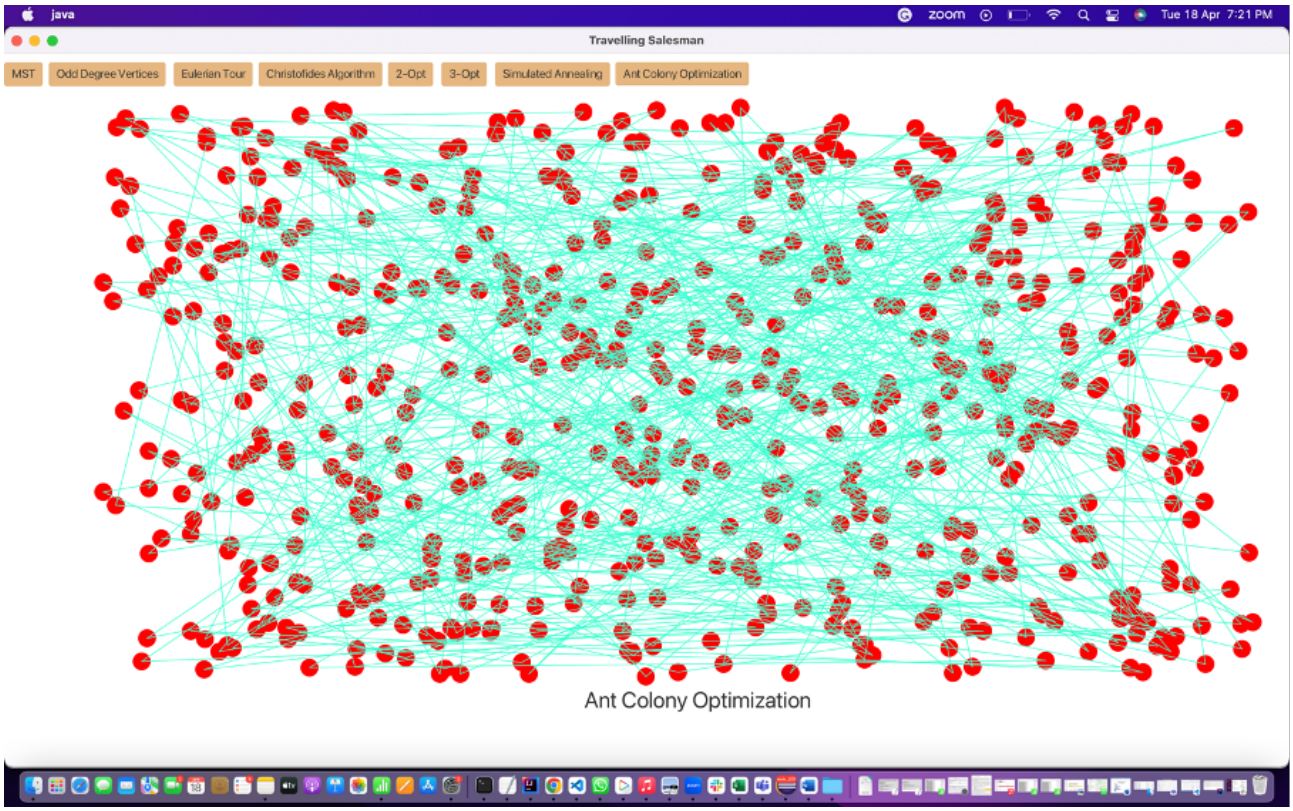
- Below is the 3-Opt route displayed when user clicks on 3-Opt Button.



- Below is the Simulated Annealing route displayed when user clicks on Simulated Annealing Button.



- Below is the Ant Colony optimization displayed when user clicks on Ant Colony optimization Button.

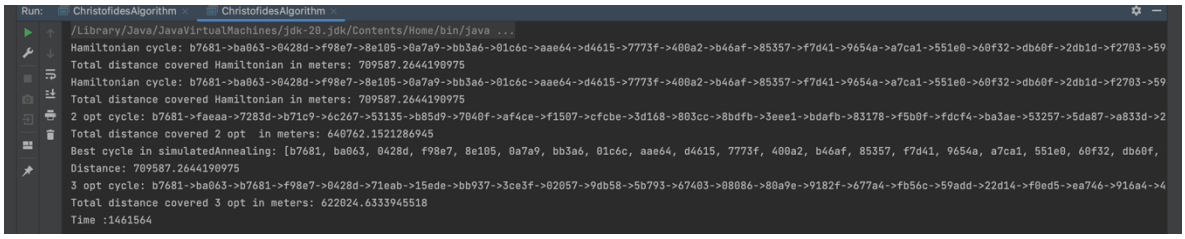


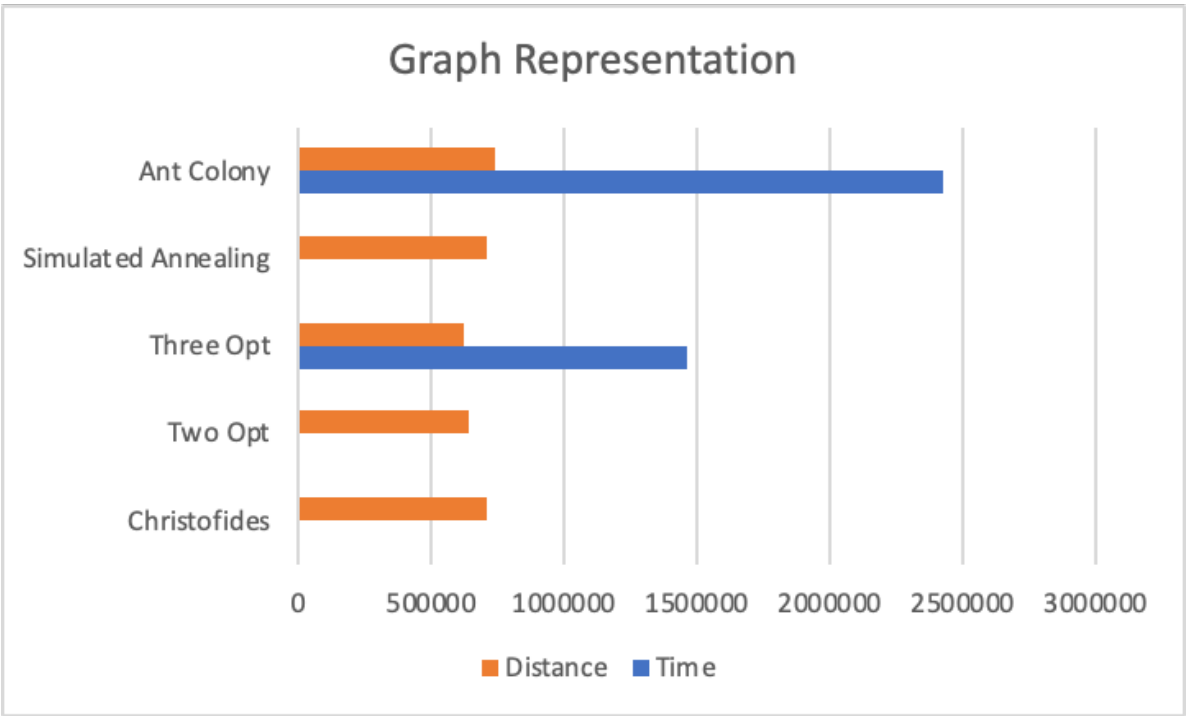
4. Observations & Graphical Analysis

Observations

1. Based on the given data set as well as latitudes and longitudes, the total distance covered in Hamiltonian is 709587.2644190975 meters.
2. The total time needed to cover the distance in Hamiltonian algorithm for the given data set is 1292 milli secs.
3. The total distance covered for the same data points in 2-Opt is 640762.1521286945 meters.
4. The total time needed to cover same set in 2-Opt is 6525 milli secs.
5. The total distance covered for the same data points in Simulated Annealing is 709587.2644190975 meters.
6. Simulated Annealing didn't result in distance better than Hamiltonian distance.
7. The total time needed to cover same set in Simulated Annealing is 170 milli secs.
8. The total distance covered for the same data points in 3-Opt is 622024.6333945518 meters.
9. The total time needed to cover same set in 3-Opt is 1461564 milli secs.
10. As per our time analysis, 3-Opt takes longer than Hamiltonian, 2-Opt and Simulated Annealing algorithm and is the slowest algorithm whereas Simulated Annealing is the fastest algorithm than Hamiltonian, 2-Opt and 3-Opt.
11. As per our distance analysis, distance covered by 3-Opt is the shortest among other algorithms whereas distance covered by Hamiltonian and Simulated Annealing is same as well as the longest distance.

Graphical Analysis





5. Results & Mathematical Analysis

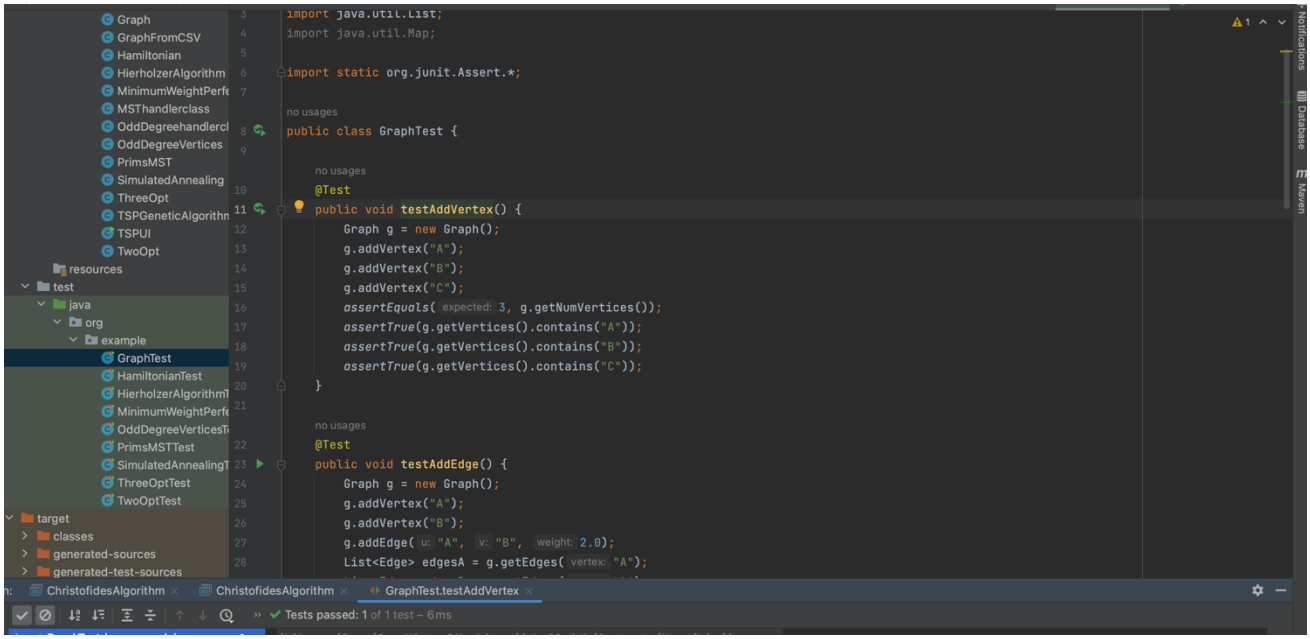
The Three opt algorithm has a worst-case time complexity of O(n^3) but it results in the best distance since it checks every route.
The distance is 622024.6333945518 meters.

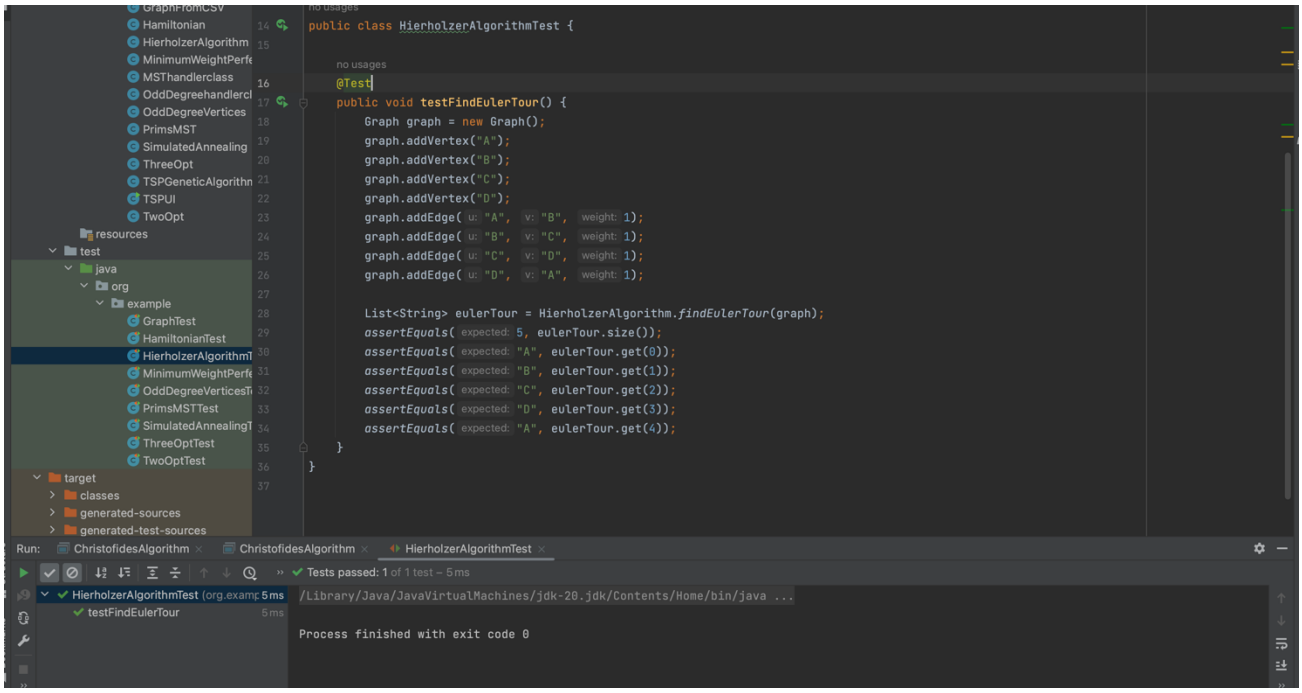
Time taken is 1461564 milli secs.

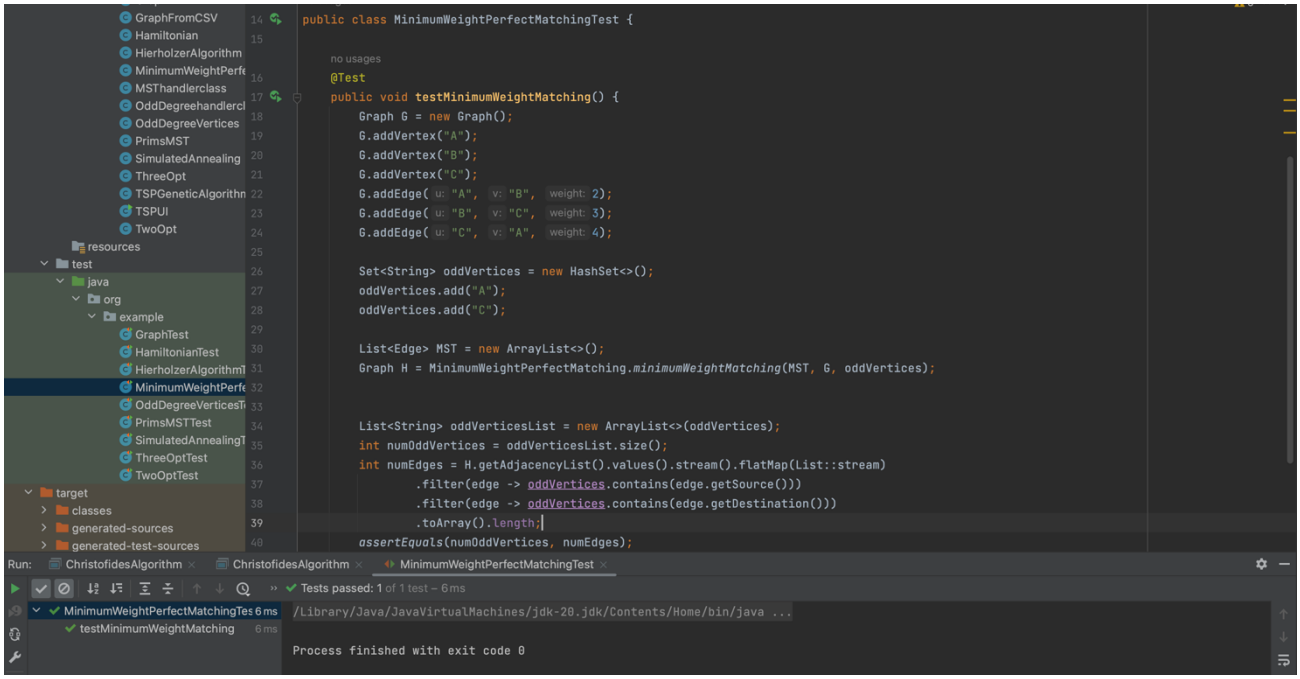
Time Complexity while comparing all optimizations:

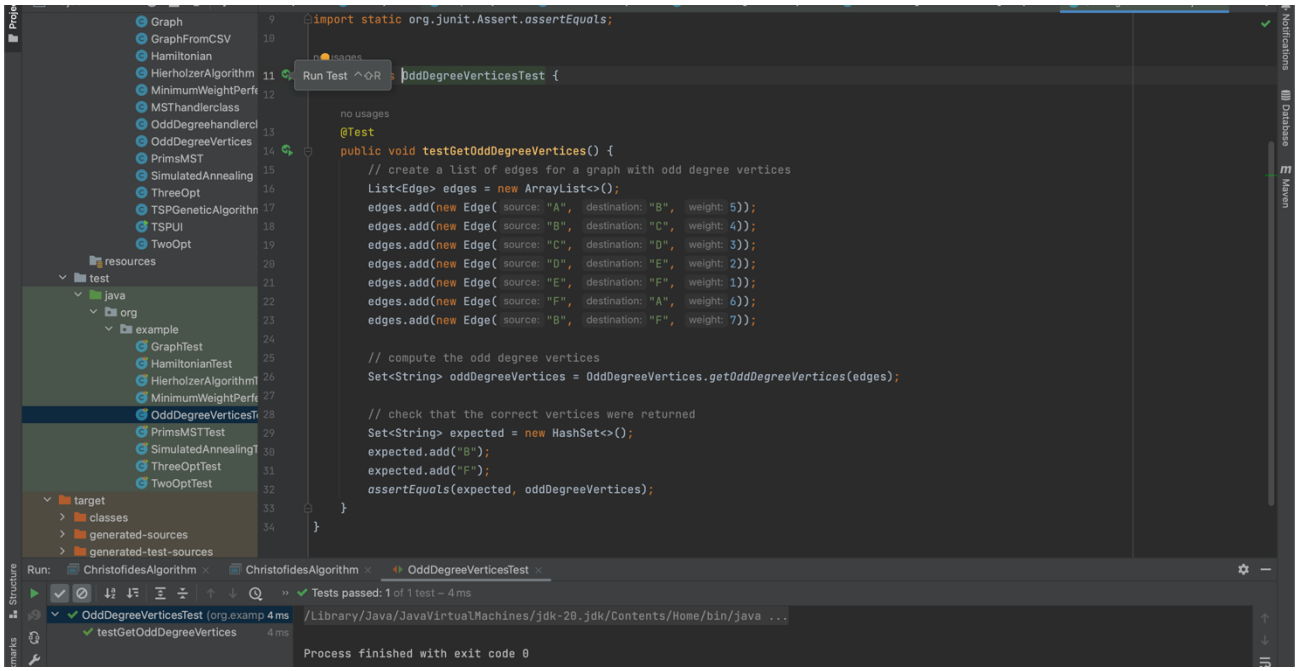
Ant Colony > 3-Opt > 2-Opt > Hamiltonian > Simulated Annealing

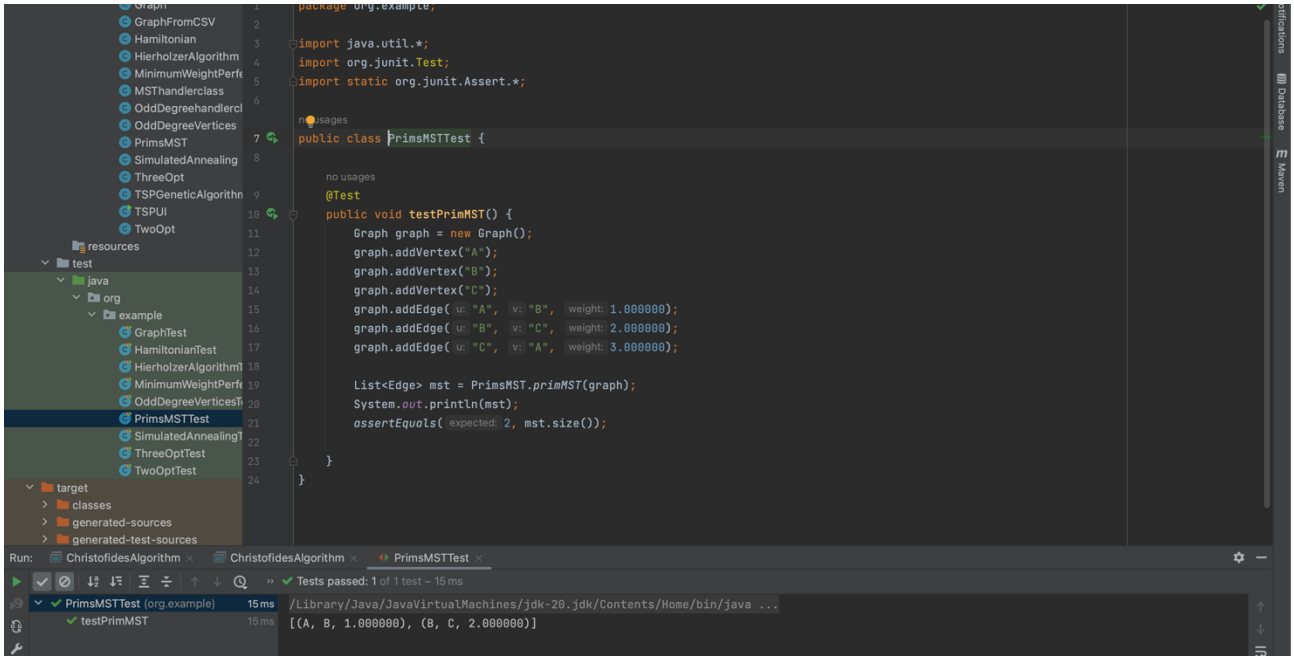
6. Unit Tests

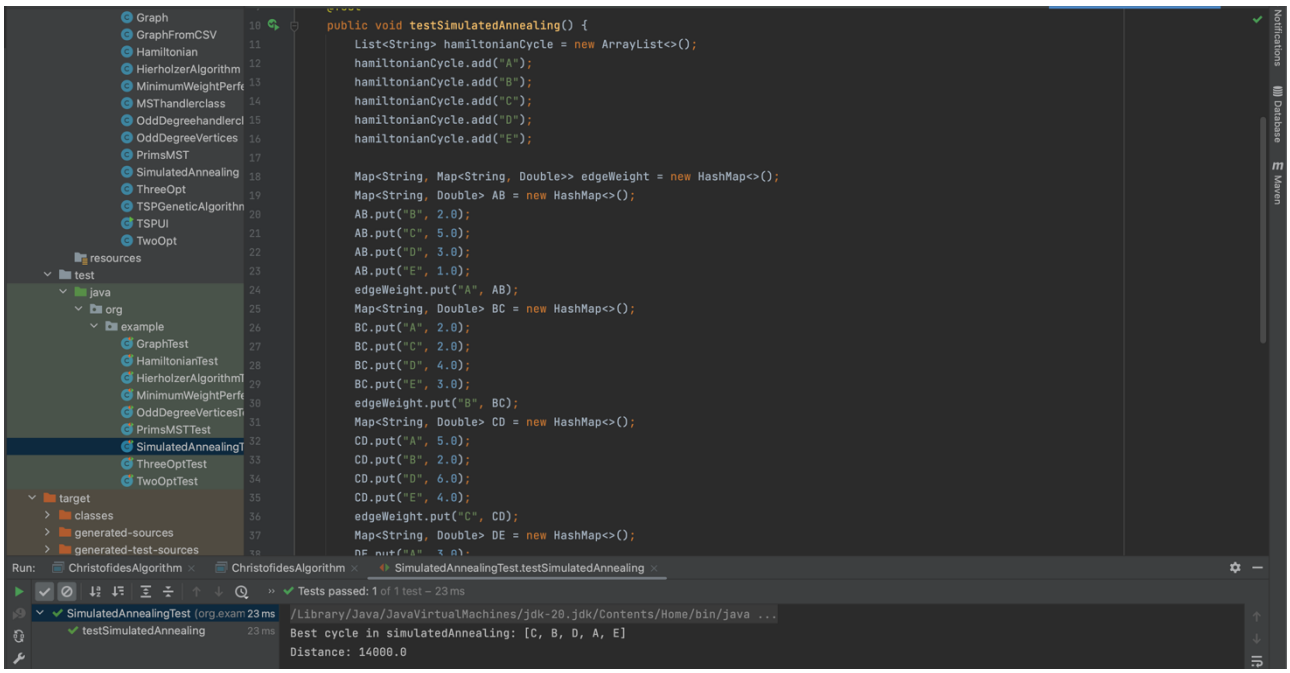

import java.util.List;
import java.util.Map;
import static org.junit.Assert.*;
public class GraphTest {
@Test
public void testAddVertex() {
Graph g = new Graph();
g.addVertex("A");
g.addVertex("B");
g.addVertex("C");
assertEquals(expected: 3, g.getNumVertices());
assertTrue(g.getVertices().contains("A"));
assertTrue(g.getVertices().contains("B"));
assertTrue(g.getVertices().contains("C"));
}
@Test
public void testAddEdge() {
Graph g = new Graph();
g.addVertex("A");
g.addVertex("B");
g.addEdge(u: "A", v: "B", weight: 2.0);
List<Edge> edgesA = g.getEdges(vertex: "A");
Tests passed: 1 of 1 test – 6ms

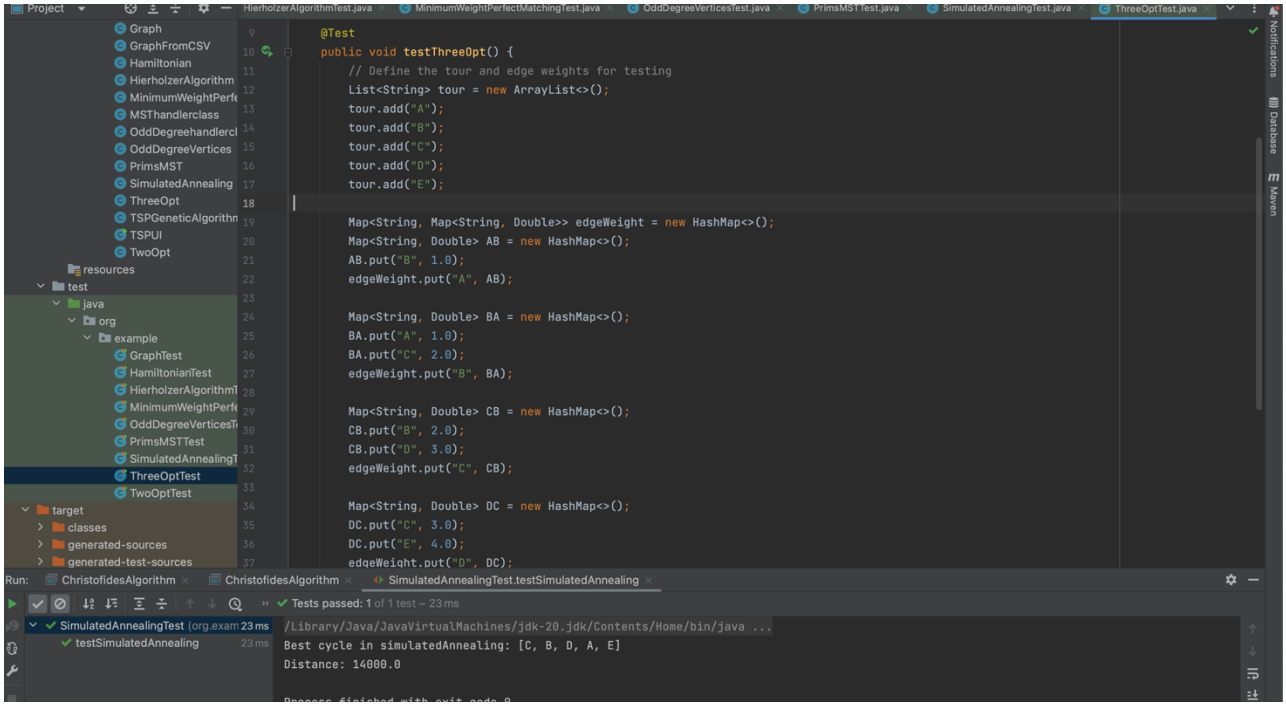

public class HierholzerAlgorithmTest {
@Test
public void testFindEulerTour() {
Graph graph = new Graph();
graph.addVertex("A");
graph.addVertex("B");
graph.addVertex("C");
graph.addVertex("D");
graph.addEdge(u: "A", v: "B", weight: 1);
graph.addEdge(u: "B", v: "C", weight: 1);
graph.addEdge(u: "C", v: "D", weight: 1);
graph.addEdge(u: "D", v: "A", weight: 1);
List<String> eulerTour = HierholzerAlgorithm.findEulerTour(graph);
assertEquals(expected: 5, eulerTour.size());
assertEquals(expected: "A", eulerTour.get(0));
assertEquals(expected: "B", eulerTour.get(1));
assertEquals(expected: "C", eulerTour.get(2));
assertEquals(expected: "D", eulerTour.get(3));
assertEquals(expected: "A", eulerTour.get(4));
}
}
Tests passed: 1 of 1 test – 5 ms
HierholzerAlgorithmTest (org.examp 5 ms
testFindEulerTour 5 ms
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java ...
Process finished with exit code 0

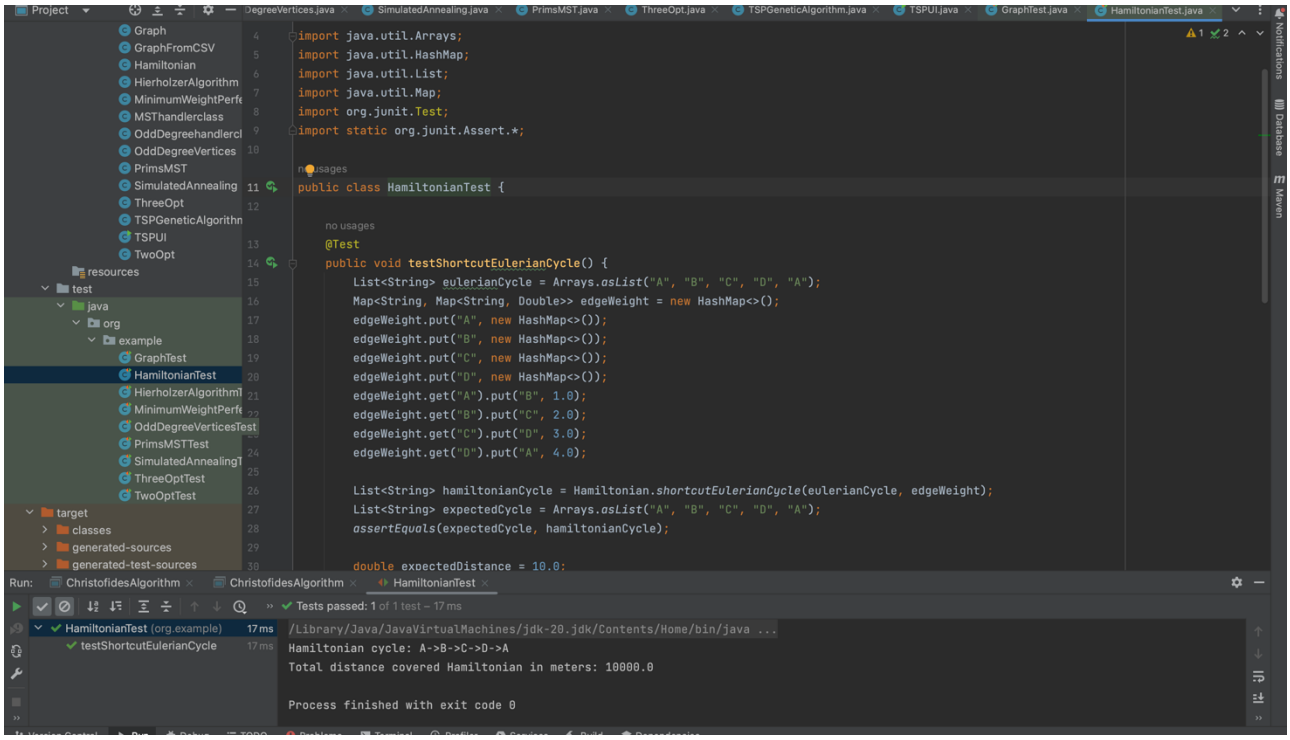

public class MinimumWeightPerfectMatchingTest {
no usages
@Test
public void testMinimumWeightMatching() {
Graph G = new Graph();
G.addVertex("A");
G.addVertex("B");
G.addVertex("C");
G.addEdge(u: "A", v: "B", weight: 2);
G.addEdge(u: "B", v: "C", weight: 3);
G.addEdge(u: "C", v: "A", weight: 4);
Set<String> oddVertices = new HashSet<>();
oddVertices.add("A");
oddVertices.add("C");
List<Edge> MST = new ArrayList<>();
Graph H = MinimumWeightPerfectMatching.minimumWeightMatching(MST, G, oddVertices);
List<String> oddVerticesList = new ArrayList<>(oddVertices);
int numOddVertices = oddVerticesList.size();
int numEdges = H.getAdjacencyList().values().stream().flatMap(List::stream)
.filter(edge -> oddVertices.contains(edge.getSource()))
.filter(edge -> oddVertices.contains(edge.getDestination()))
.toArray().length;
assertEquals(numOddVertices, numEdges);
Run: ChristofidesAlgorithm ChristofidesAlgorithm MinimumWeightPerfectMatchingTest
Tests passed: 1 of 1 test – 6 ms
MinimumWeightPerfectMatchingTes 6 ms
testMinimumWeightMatching 6 ms
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java ...
Process finished with exit code 0


import static org.junit.Assert.assertEquals;
Run Test ^⇧R
OddDegreeVerticesTest {
no usages
@Test
public void testGetOddDegreeVertices() {
// create a list of edges for a graph with odd degree vertices
List<Edge> edges = new ArrayList<>();
edges.add(new Edge(source: "A", destination: "B", weight: 5));
edges.add(new Edge(source: "B", destination: "C", weight: 4));
edges.add(new Edge(source: "C", destination: "D", weight: 3));
edges.add(new Edge(source: "D", destination: "E", weight: 2));
edges.add(new Edge(source: "E", destination: "F", weight: 1));
edges.add(new Edge(source: "F", destination: "A", weight: 6));
edges.add(new Edge(source: "B", destination: "F", weight: 7));
// compute the odd degree vertices
Set<String> oddDegreeVertices = OddDegreeVertices.getOddDegreeVertices(edges);
// check that the correct vertices were returned
Set<String> expected = new HashSet<>();
expected.add("B");
expected.add("F");
assertEquals(expected, oddDegreeVertices);
}
}
Run: ChristofidesAlgorithm ChristofidesAlgorithm OddDegreeVerticesTest
Tests passed: 1 of 1 test – 4 ms
OddDegreeVerticesTest (org.examp 4 ms
testGetOddDegreeVertices 4 ms
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java ...
Process finished with exit code 0


GraphFromCSV
Hamiltonian
HierholzerAlgorithm
MinimumWeightPerfe
MSThandlerclass
OddDegreehandlercl
OddDegreeVertices
PrimsMST
SimulatedAnnealing
ThreeOpt
TSPGeneticAlgorithn
TSPUI
TwoOpt
resources
test
java
org
example
GraphTest
HamiltonianTest
HierholzerAlgorithmT
MinimumWeightPerfe
OddDegreeVerticesTe
PrimsMSTTest
SimulatedAnnealingT
ThreeOptTest
TwoOptTest
target
classes
generated-sources
generated-test-sources
package org.example;
import java.util.*;
import org.junit.Test;
import static org.junit.Assert.*;
no usages
public class PrimsMSTTest {
no usages
@Test
public void testPrimMST() {
Graph graph = new Graph();
graph.addVertex("A");
graph.addVertex("B");
graph.addVertex("C");
graph.addEdge(u: "A", v: "B", weight: 1.000000);
graph.addEdge(u: "B", v: "C", weight: 2.000000);
graph.addEdge(u: "C", v: "A", weight: 3.000000);
List<Edge> mst = PrimsMST.primMST(graph);
System.out.println(mst);
assertEquals(expected: 2, mst.size());
}
}
Run:
ChristofidesAlgorithm
ChristofidesAlgorithm
PrimsMSTTest
Tests passed: 1 of 1 test – 15 ms
PrimsMSTTest (org.example) 15 ms
testPrimMST 15 ms
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java ...
[(A, B, 1.000000), (B, C, 2.000000)]


Graph
GraphFromCSV
Hamiltonian
HierholzerAlgorithm
MinimumWeightPerfe
MSThandlerclass
OddDegreehandlercl
OddDegreeVertices
PrimsMST
SimulatedAnnealing
ThreeOpt
TSPGeneticAlgorithn
TSPUI
TwoOpt
resources
test
java
org
example
GraphTest
HamiltonianTest
HierholzerAlgorithmT
MinimumWeightPerfe
OddDegreeVerticesTe
PrimsMSTTest
SimulatedAnnealingT
ThreeOptTest
TwoOptTest
target
classes
generated-sources
generated-test-sources
public void testSimulatedAnnealing() {
List<String> hamiltonianCycle = new ArrayList<>();
hamiltonianCycle.add("A");
hamiltonianCycle.add("B");
hamiltonianCycle.add("C");
hamiltonianCycle.add("D");
hamiltonianCycle.add("E");
Map<String, Map<String, Double>> edgeWeight = new HashMap<>();
Map<String, Double> AB = new HashMap<>();
AB.put("B", 2.0);
AB.put("C", 5.0);
AB.put("D", 3.0);
AB.put("E", 1.0);
edgeWeight.put("A", AB);
Map<String, Double> BC = new HashMap<>();
BC.put("A", 2.0);
BC.put("C", 2.0);
BC.put("D", 4.0);
BC.put("E", 3.0);
edgeWeight.put("B", BC);
Map<String, Double> CD = new HashMap<>();
CD.put("A", 5.0);
CD.put("B", 2.0);
CD.put("D", 6.0);
CD.put("E", 4.0);
edgeWeight.put("C", CD);
Map<String, Double> DE = new HashMap<>();
Run:
ChristofidesAlgorithm
ChristofidesAlgorithm
SimulatedAnnealingTest.testSimulatedAnnealing
Tests passed: 1 of 1 test – 23 ms
SimulatedAnnealingTest (org.exam 23 ms
testSimulatedAnnealing 23 ms
/Library/Java/JavaVirtualMachines/jdk-20.jdk/Contents/Home/bin/java ...
Best cycle in simulatedAnnealing: [C, B, D, A, E]
Distance: 14000.0





7. Conclusion

Christofides algorithm is an efficient and effective heuristic algorithm for approximating the TSP solution with a performance guarantee. It is an approximation algorithm that guarantees that its solutions will be within a factor of 3/2 of the optimal solution length.

In conclusion, each of the algorithms mentioned above has its strengths and weaknesses in solving the TSP. The Christofides algorithm is simple and efficient but may not always produce optimal solutions.

2-opt and 3-opt are effective for small to moderate-sized TSP. Ant Colony and Simulated Annealing is good for large-sized data sets. The choice of algorithm depends on the size and complexity of the TSP instance, as well as the computational resources available.

8. References

1) https://www.geeksforgeeks.org/haversine-formula-to-find-distance-between-two-points-on-a-sphere/
2) https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/
3) https://www.geeksforgeeks.org/approximate-solution-for-travelling-salesman-problem-using-mst/
4) https://www.geeksforgeeks.org/simulated-annealing/
5) https://www.geeksforgeeks.org/introduction-to-ant-colony-optimization/