INFO 6205 – PROGRAM STRUCTURES AND ALGORITHMS
SPRING 2023 PROJECT
TRAVELING SALESMAN PROBLEM SOLUTION USING CHRISTOFIDES ALGORITHM AND OTHER OPTIMIZATION TECHNIQUES

Team Members

1. Rohit Panicker – 002791446
2. Sharun Kumar Kakkad Sasikumar – 002774079
3. Sai Tej Sunkara - 002728188

Introduction

Aim: The aim of this paper is to analyze the effectiveness of Christofides algorithm along with strategical and tactical optimizations such as Random Optimization, 2-opt and Ant Colony optimizations for Traveling Salesman Problem (TSP). We aim to provide insights into the strengths and weaknesses of each approach, as well as recommendations for practitioners seeking to solve similar optimization problems.

Approach: In this paper, we apply the Christofides algorithm to solve the Traveling Salesman Problem (TSP), which is a classic combinatorial optimization problem. To further improve the solution quality, we also investigate several meta-heuristic optimization techniques, including Greedy Match, Simulated Annealing, Ant Colony Optimization, Random Optimization, and Two Optimization Technique.

We have created a Java code which implements Christofides algorithm along with the above-mentioned optimization techniques. The algorithm is implemented using modularity and Object-oriented concepts for enabling reusability and unit testing.

We have implemented several skeleton POJO classes along with various interfaces which supports the implementation of this algorithm. To generate the data-infrastructure for this algorithm, we have utilized a csv dataset which represents each vertex/node as a pair of latitude and longitude and we use this geo-coordinates to populate a logical data-infrastructure which is utilized later as an input for Christofides algorithm and for performing various tactical and strategical optimizations.

To create a graph with weighted edges , we use Haversine formula to calculate the distance between two nodes. Once this step has been completed, we pass this input to the Prims

algorithms for creating a MST. Then we find out the edges with odd degrees within the MST and apply minimum weight perfect matching algorithm to connect all the vertices in such a way such that the cost is minimum. Then we combine the edges generated using minimum weight perfect matching algorithm and MST to generate a Multigraph, which ensures that all the vertices have even degrees. Later on we perform a Eulerian tour on this multigraph, which is later used as an input for generating a TSP tour where we skip previously visited vertices.

This TSP tour is later forwarded to Optimization techniques such as Random Optimization, 2-opt optimization, Ant Colony optimization and Simulated annealing for generating a tour with optimal value.

Program

All Classes used:

ChristofidesAlgorithm: This is the main class that implements the Christofides algorithm to solve the Traveling Salesman Problem.

PrimsAlgorithm: This is a helper class that implements Prim's algorithm to generate a minimum spanning tree of the input graph.

GreedyMatch: This is another helper class that implements the greedy matching algorithm to find a perfect matching of the odd-degree vertices in the minimum spanning tree.

MultiGraph: This is a data structure that represents a weighted multigraph, which is used to store the intermediate results of the algorithm.

GraphNode: This is a data structure that represents a node in the multigraph, which contains information about its index, degree, and adjacency list.

EulerCircuitGenerator: This is the main class that generates an Euler circuit in the multigraph, which represents a feasible TSP tour.

Edge: This class represents the edge of a graph and has instance properties such as to, from and edge weight to represent vertices of that edge and the cost of that edge.

Node: This class is a facilitator class for the greedy match process which helps in finding and isolating the nodes in a MST with odd degrees.

OptimizationHelperFunctions: Contains helper functions which are common and are needed while performing various optimization techniques on top of Christofides algorithm.

RandomOptimization: Implements the Random Optimization algorithm for obtaining the optimized value for TSP route.

RandomOptimizerBruteForce: Performs the Random Optimization technique but performs all possible combination of nodes. TC: O(N^2)

TwoOptOptimization: Implements the 2-Opt Optimization algorithm for obtaining the optimized value for TSP route.

Data Modals:
Location: - class that contains the location object (latitude, longitude) and some relevant helper functions for the same

TravelPath: - Object that encapsulates a route, as well as the collection of locations[] (which are normalized) that is also used as a param to draw the points on to the screen

Data Classes:
DataSet - Class that reads the data from the CSV and supplies with locations[] and normalized locations for plotting purposes

Classes for UI:
DRAW_MODE - enum used for draw mode in the window => to draw MST or an actual travel route

HeadlessPanel, HeadlessTspWindow: this is used in unit test cases where algorithm needs to be tested without showing a UI, so a variant of the TravellingSalesmanWindow that does nothing is used

Panel, TravellingSalesmanWindow - panel and window components in java awt that helps to plot the points and routes

Algorithm Classes:
AntColonyOptimization - Algorithm for ant colony optimization

SimulatedAnnealing - Algorithm for ant simulated annealing

Algorithm Steps Running Classes:
IAlgorithmStep - Interface that defines the abstraction as to how a step should be shaped in order to be consumed by the driving class

PrintLocations, PrintPath, PrintRoute, WinTitle - Helper classes that can be inserted between steps to do certain granular functions

AntColony, GenerateEulerCircuit, GenerateMst, RandomOptimization, SimulatedAnnealingStep, TwoOpt - adapter classes to run the algorithms as steps

Utilities:
TSPUtilities - utility classes with various helper functions

Main Driver Classes:

App - Point of entry of the program

TspSolverUI - Main driver of the program that defines the various steps and calls them

All Interfaces used:

EdgeInterface: This interface contains methods to be implemented by the Edge class. Contains method signature for getting the two vertices of a particular edge.

GraphNodeInterface: This interface contains methods to be implemented by the GraphNode class. Contains method signature for creating and supporting the multigraph generation process.

NodeInterface: This interface contains methods to be implemented by the Node class. Contains method signature supporting Greedy Match process.

All Data Structures used:

double[][] weightMatrix: This is a 2D array that represents the weight matrix of the input graph, which contains the distances between all pairs of vertices. It is passed as a parameter to the run method of the ChristofidesAlgorithm class.

int[] minimumSpanningTree: This is an array that represents the minimum spanning tree of the input graph, which is generated by the PrimsAlgorithm class.

int[][] matchGraph: This is a 2D array that represents the weighted bipartite graph obtained by applying the greedy matching algorithm on the minimum spanning tree. It is used to build the multigraph in the MultiGraph class.

int[] route: This is an array that represents the final TSP tour generated by the algorithm, which is returned by the run method of the ChristofidesAlgorithm class.

GraphNode[] nodes: This is an array of GraphNode objects that represent the nodes in the multigraph, which are generated by the MultiGraph class in the ChristofidesAlgorithm.

LinkedList<Integer> path: This is a linked list that represents the path in the multigraph, which is generated by the algorithm.

Vector<Integer> tmpPath: This is a vector that stores temporary paths in the multigraph, which are used to generate the final path.

boolean[] inPath: This is a boolean array that keeps track of whether a node has been visited in the path or not.

int[] route: This is an array that represents the final TSP tour generated by the algorithm, which is returned by the generateEulerCircuit method.

All Algorithms:

Algorithm: Christofides Algorithm
The Christofides algorithm is an algorithm for finding approximate solutions to the travelling salesman problem. It is an approximation algorithm that guarantees that its solutions will be within a factor of 3/2 of the optimal solution length.

Algorithm Steps:

1. Compute the minimum spanning tree (MST) of the given graph G using Prims or Kruskals.
2. Let O be the set of vertices in MST that have odd degree.
3. Create the subgraph based on the odd vertices
4. Compute a minimum weight perfect matching M among the subgraph created in step 3.
5. Combine the edges in the MST and the edges in M to form a multigraph where each vertex has even degree (IN/OUT).
6. Let E be the set of edges in H that appear exactly once (i.e., without multiplicity).
7. Find an Eulerian circuit by traversing the each edge of the graph once.
8. Convert the Eulerian circuit obtained in step 7, into a Hamiltonian circuit by visiting each vertex exactly once and avoid all previously visited vertices.

As mentioned above this algorithm guarantees that its solutions will be within a factor of 3/2 of the optimal solution length. The algorithm has a time complexity of O(N^2 log N) for a graph (G) with N vertices.

Algorithm: Ant Colony Optimization on Christofides Algorithm

Step 1. Initialize the pheromone level τ_{ij} for all edges (i, j) to a small positive value.
Step 2. Initialize a set of m ants, each placed on a random vertex.
Step 3. For each ant k (k = 1, 2, ..., m):
a. Create a feasible tour T_k of the graph G by following the construction heuristic of the Christofides algorithm.
b. Compute the tour length L_k of the tour T_k.
c. For each edge (i, j) in the tour T_k:
i. Update the pheromone level τ_{ij} as follows:
$\tau_{ij} = (1 - \rho)\tau_{ij} + \rho / L_k$, where ρ is the pheromone evaporation rate.
Step 4. Compute the tour length L_{min} of the best tour found so far and the corresponding tour T_{min}.
Step 5. Repeat steps 2-4 until a stopping criterion is met (e.g., a maximum number of iterations is reached or a time limit is exceeded).

Algorithm: Simulated Annealing on Christofides Algorithm

Step 1. Initialize the current solution as a feasible tour T and set the temperature T0 to a high value.
Step 2. Repeat until the stopping criterion is met:

- a. Choose a neighbor solution T' by applying a local search operator to T (e.g., 2-opt, 3-opt, or any other suitable operator).
- b. Compute the cost difference $\Delta C = C(T') - C(T)$, where C(T) and C(T') are the costs of the current and neighbor solutions, respectively.
- c. If $\Delta C < 0$, accept T' as the new current solution.
- d. If $\Delta C > 0$, accept T' as the new current solution with probability $\exp(-\Delta C/T)$, where T is the current temperature.
- e. Reduce the temperature T according to a cooling schedule (e.g., geometric cooling, linear cooling, or any other suitable schedule).

Step 3. Return the best solution found during the search.

Algorithm: Random Optimization on Christofides Algorithm

Step 1. Initialize a set of N random feasible tours $T_1, T_2, ..., T_N$.
Step 2. For each tour T_i (i = 1, 2, ..., N):

- a. Apply a random perturbation to T_i to generate a new tour T'_i.
- b. Compute the cost difference $\Delta C = C(T'_i) - C(T_i)$, where $C(T_i)$ and $C(T'_i)$ are the costs of the current and perturbed tours, respectively.
- c. If $\Delta C < 0$, accept T'_i as the new tour T_i.
- d. If $\Delta C > 0$, accept T'_i as the new tour T_i with probability $\exp(-\Delta C/T)$, where T is a temperature parameter that controls the acceptance probability.

Step 3. Return the best tour found during the search.

Algorithm: 2 Opt Optimization on Christofides Algorithm

Step 1. Initialize a feasible tour T using the Christofides algorithm.
Step 2. Repeat until no improvement is found:

- a. For each pair of edges (i, j) and (k, l) in T, where $i < k$ and $j < l$:
    - i. Compute the cost difference $\Delta C = C(i, j) + C(k, l) - C(i, k) - C(j, l)$, where C(i, j) and C(k, l) are the costs of the edges (i, j) and (k, l), respectively, and C(i, k) and C(j, l) are the costs of the edges (i, k) and (j, l), respectively.
    - ii. If $\Delta C < 0$, reverse the sub-tour from node i+1 to k.

Step 3. Return the best tour found during the search.

Explanation:

Step 1: To start, we compute a minimum spanning tree T for the given graph G using a standard algorithm like Prim's or Kruskal's algorithm. We are using Prim's algorithm currently.

Step 2: Next, we find the set of vertices O in T with odd degree. This is because any Hamiltonian cycle in G must have an even number of edges incident on each vertex, and so any vertex with odd degree must be part of the cycle.

Step 3: We compute a minimum-weight perfect matching M on the vertices in O.

Step 4: We form a new graph H by taking the union of T and M.

Step 5: We find an Eulerian cycle in H starting and ending at any vertex.

Step 6: Finally, we traverse the Eulerian cycle to obtain a Hamiltonian cycle in G.

The algorithm guarantees that the output is a Hamiltonian cycle in G that has weight no more than 1.5 times the weight of a minimum-weight Hamiltonian cycle in G.

Invariants

The invariants in Christofides algorithm

1. The input graph is undirected, connected, and has non-negative edge weights.

The algorithm constructs a minimum spanning tree (MST) of the input graph using Kruskal's algorithm or Prim's algorithm.

2. The MST is transformed into an Eulerian graph by adding edges to the MST to create an even degree for each vertex in the graph.

3. The algorithm finds an Eulerian circuit in the Eulerian graph.

4. The Eulerian circuit is transformed into a Hamiltonian circuit by shortcutting repeated vertices.

These invariants ensure that the algorithm produces a valid solution to the TSP that is at most twice the length of the optimal solution.

The invariants in Simulated Annealing on Christofides Algorithm

1. The input graph is undirected, connected, and has non-negative edge weights.

2. The Christofides algorithm constructs a minimum spanning tree (MST) of the input graph using Kruskal's algorithm or Prim's algorithm.

3. The MST is transformed into an Eulerian graph by adding edges to the MST to create an even degree for each vertex in the graph.

4. The SA algorithm generates an initial solution by using the Eulerian circuit obtained from the Christofides algorithm.

5. The SA algorithm iteratively improves the initial solution by randomly selecting neighboring solutions and accepting them with a probability that depends on the quality of the new solution and a temperature parameter that decreases over time.

6. The SA algorithm terminates when a stopping criterion is met, such as a maximum number of iterations or a minimum temperature.

These invariants ensure that the SA algorithm applied to the Christofides algorithm produces a valid solution to the TSP that can be further improved by exploring the solution space using a probabilistic approach. The quality of the final solution obtained depends on the temperature schedule used and the stopping criterion chosen.

The invariants in Ant Colony Optimization on Christofides Algorithm

1. The input graph is undirected, connected, and has non-negative edge weights.

2. The MST is transformed into an Eulerian graph by adding edges to the MST to create an even degree for each vertex in the graph.

3. The ACO algorithm initializes a colony of artificial ants that traverse the graph to construct a solution to the TSP.

4. At each step of the algorithm, each ant chooses its next vertex to visit based on a probabilistic decision rule that depends on the pheromone level on the edges and the distance between the vertices.

5. After all ants complete a tour of the graph, the pheromone level on the edges is updated based on the quality of the solutions found by the ants.

6. The ACO algorithm iteratively improves the solutions by repeating the ant tour and pheromone update steps.

7. The ACO algorithm terminates when a stopping criterion is met, such as a maximum number of iterations or a minimum change in the pheromone level.

These invariants ensure that the ACO algorithm applied to the Christofides algorithm produces a valid solution to the TSP that can be further improved by exploring the solution space using a probabilistic approach. The quality of the final solution obtained depends on the pheromone update rule, the probabilistic decision rule, and the stopping criterion chosen.

The invariants in Random Optimization on Christofides Algorithm

Invariants refer to properties that are preserved throughout the optimization process. In the case of Random Optimization on Christofides Algorithm, there are a few invariants that are maintained:

1. **Feasibility:** The solution generated at any iteration of the algorithm must be a feasible solution to the TSP problem. That is, the solution must satisfy the constraints of the problem, which require that each city is visited exactly once, and the tour is closed.

2. **Tour length:** The length of the tour must be maintained as an invariant. This means that the total distance traveled by the salesman between all the cities must be calculated and stored, and this value must be updated at every iteration of the algorithm.

3. **Local optimality:** At each iteration, the algorithm must generate a locally optimal solution. That is, the solution generated must be the best solution possible given the current state of the algorithm.

4. **Randomness:** The algorithm must make use of randomization in order to explore the search space and avoid getting stuck in local optima.
Overall, the goal of Random Optimization on Christofides Algorithm is to generate a globally optimal solution to the TSP problem by iteratively improving the current solution through the maintenance of these invariants.

The invariants in Two Opt Technique on Christofides Algorithm

When Two Opt Optimization is applied to Christofides Algorithm, there are a few invariants that are maintained:

1. **Feasibility:** The solution generated at any iteration of the algorithm must be a feasible solution to the TSP problem. That is, the solution must satisfy the constraints of the problem, which require that each city is visited exactly once, and the tour is closed.

2. **Tour length:** The length of the tour must be maintained as an invariant. This means that the total distance traveled by the salesman between all the cities must be calculated and stored, and this value must be updated at every iteration of the algorithm.

3. **Local optimality:** At each iteration, the algorithm must generate a locally optimal solution. That is, the solution generated must be the best solution possible given the current state of the algorithm.

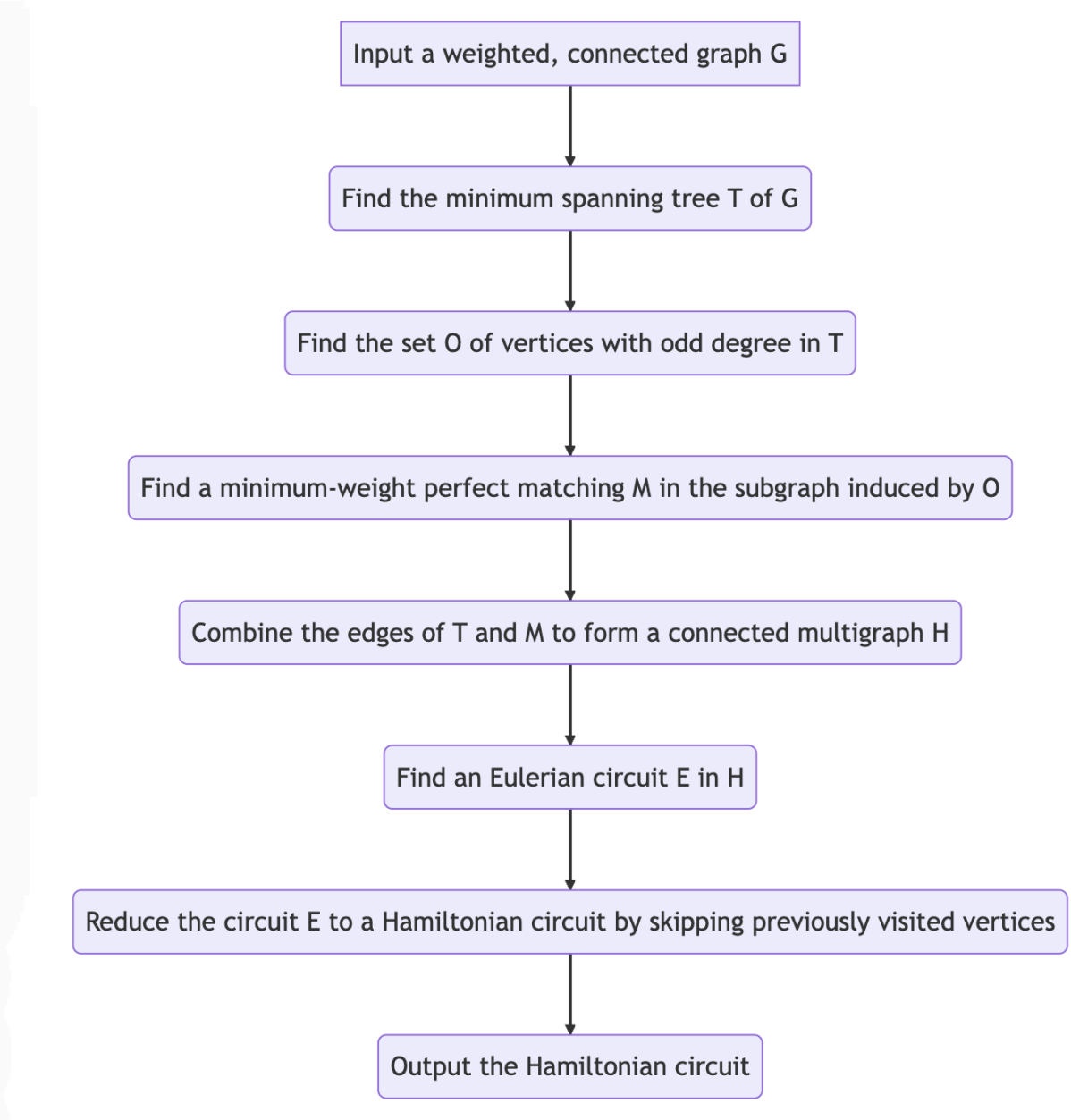
4. **Two-Opt optimality:** The algorithm must maintain the Two-Opt optimality invariant, which means that the tour generated by the algorithm must not contain any edges that could be swapped to create a shorter tour. In other words, the algorithm must ensure that it has explored all possible two-edge swaps and that the current solution is locally optimal with respect to these swaps.

5. **Convergence:** The algorithm must converge to a globally optimal solution after a finite number of iterations. This is ensured by the Two-Opt optimality invariant, which guarantees that the algorithm will eventually reach a globally optimal solution.

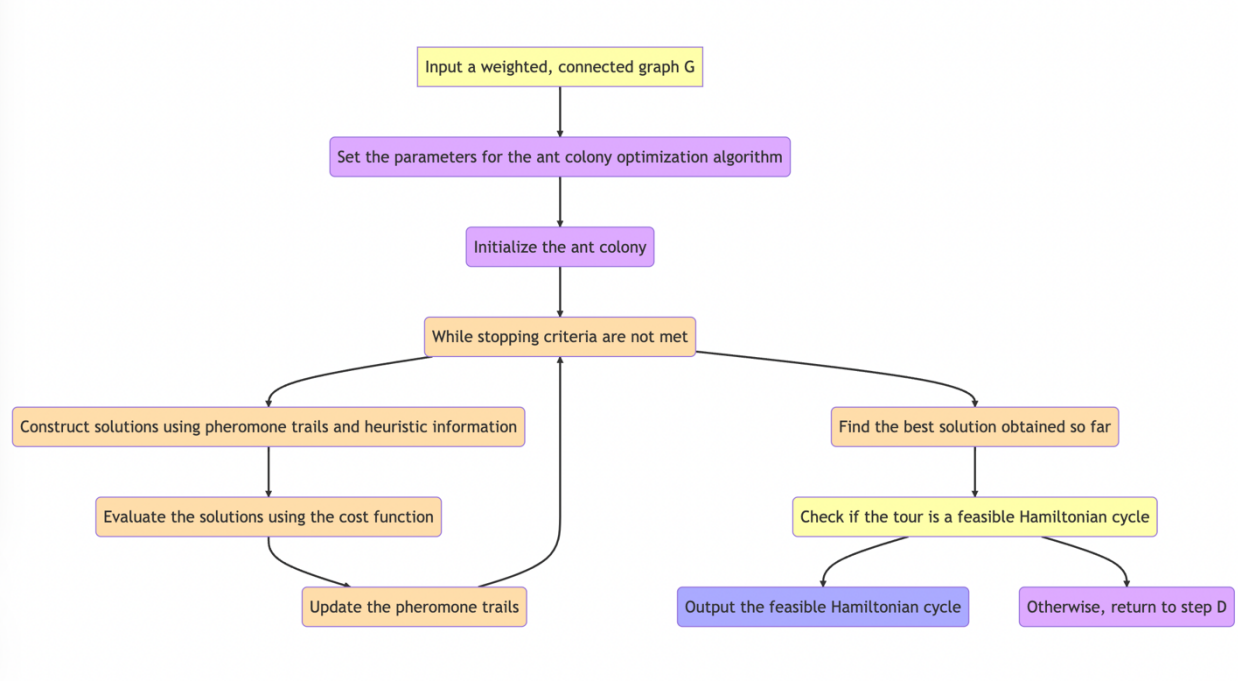
Overall, the goal of Two Opt Optimization on Christofides Algorithm is to generate a globally optimal solution to the TSP problem by iteratively improving the current solution through the maintenance of these invariants.

Flowcharts

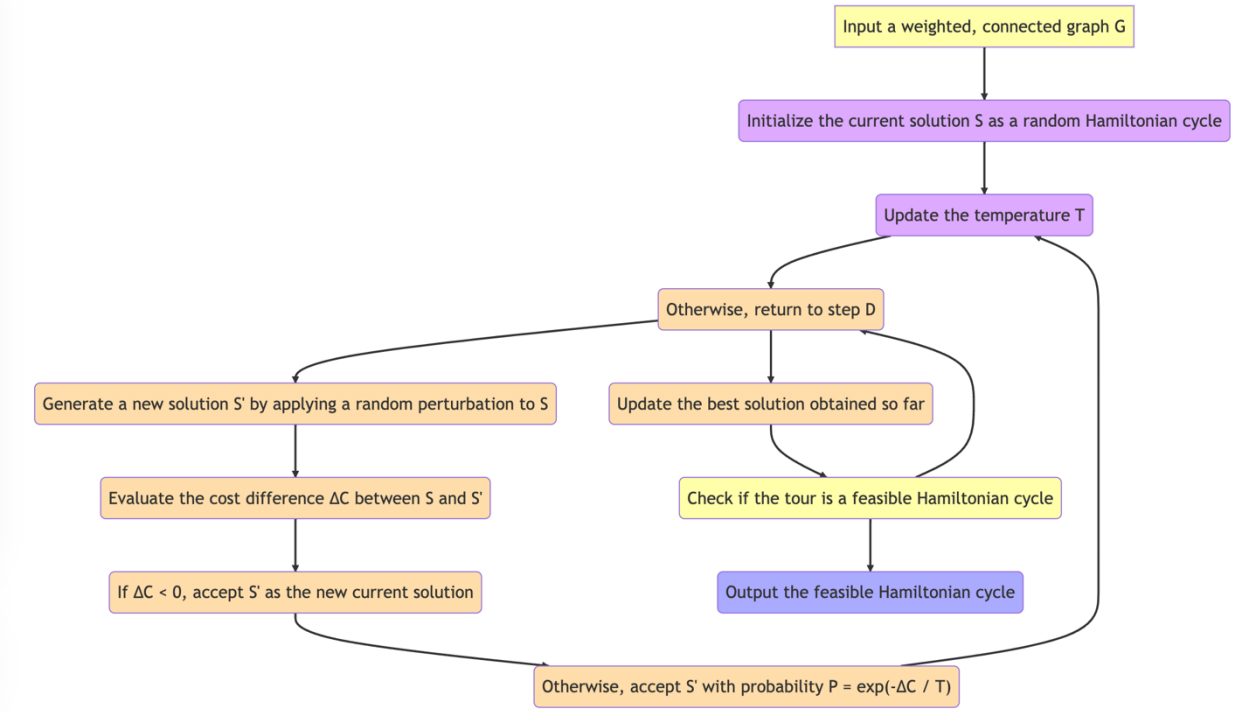
Flowchart: Christofides Algorithm



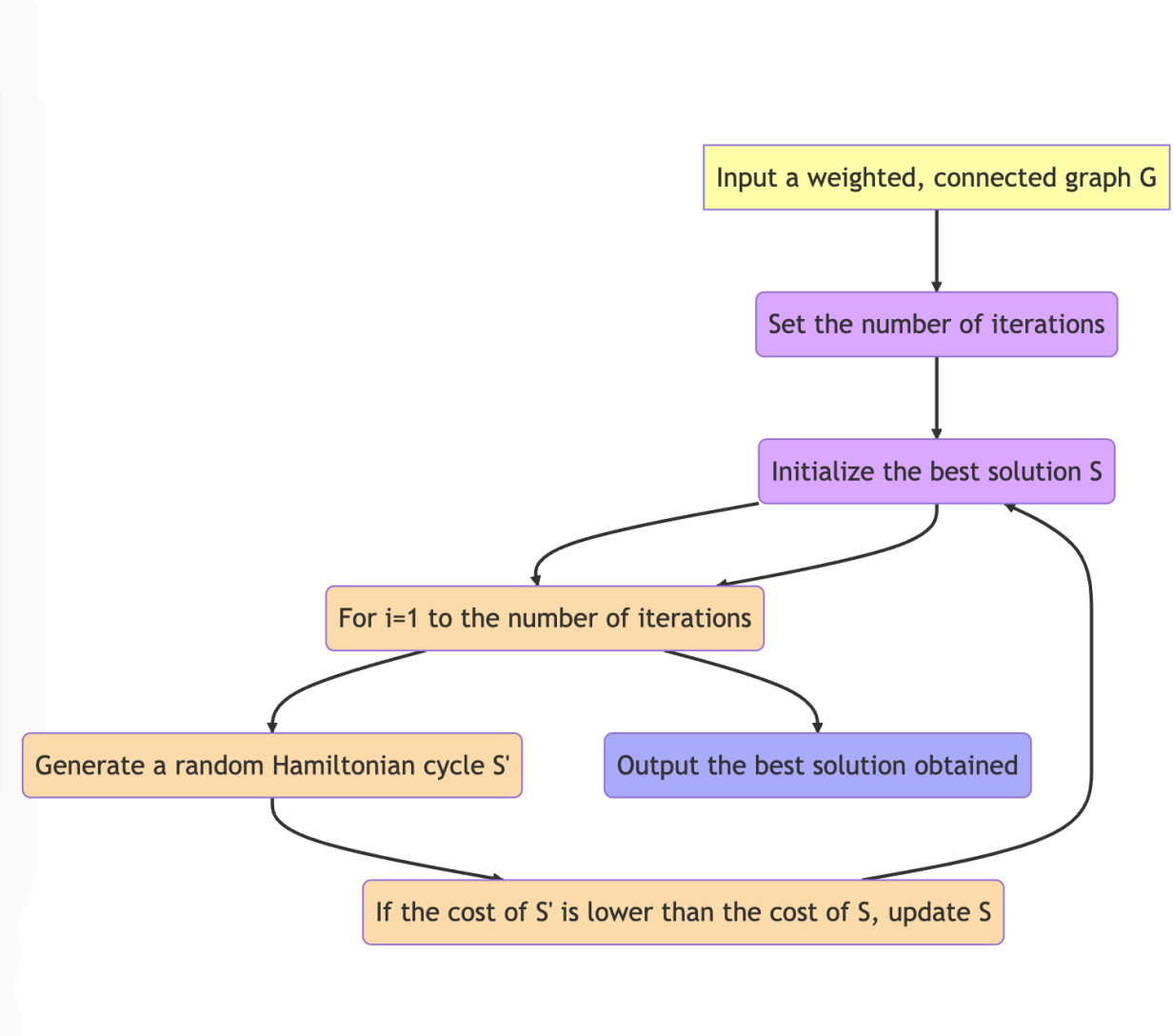
Flowchart: Ant colony Optimization on Christofides Algorithm



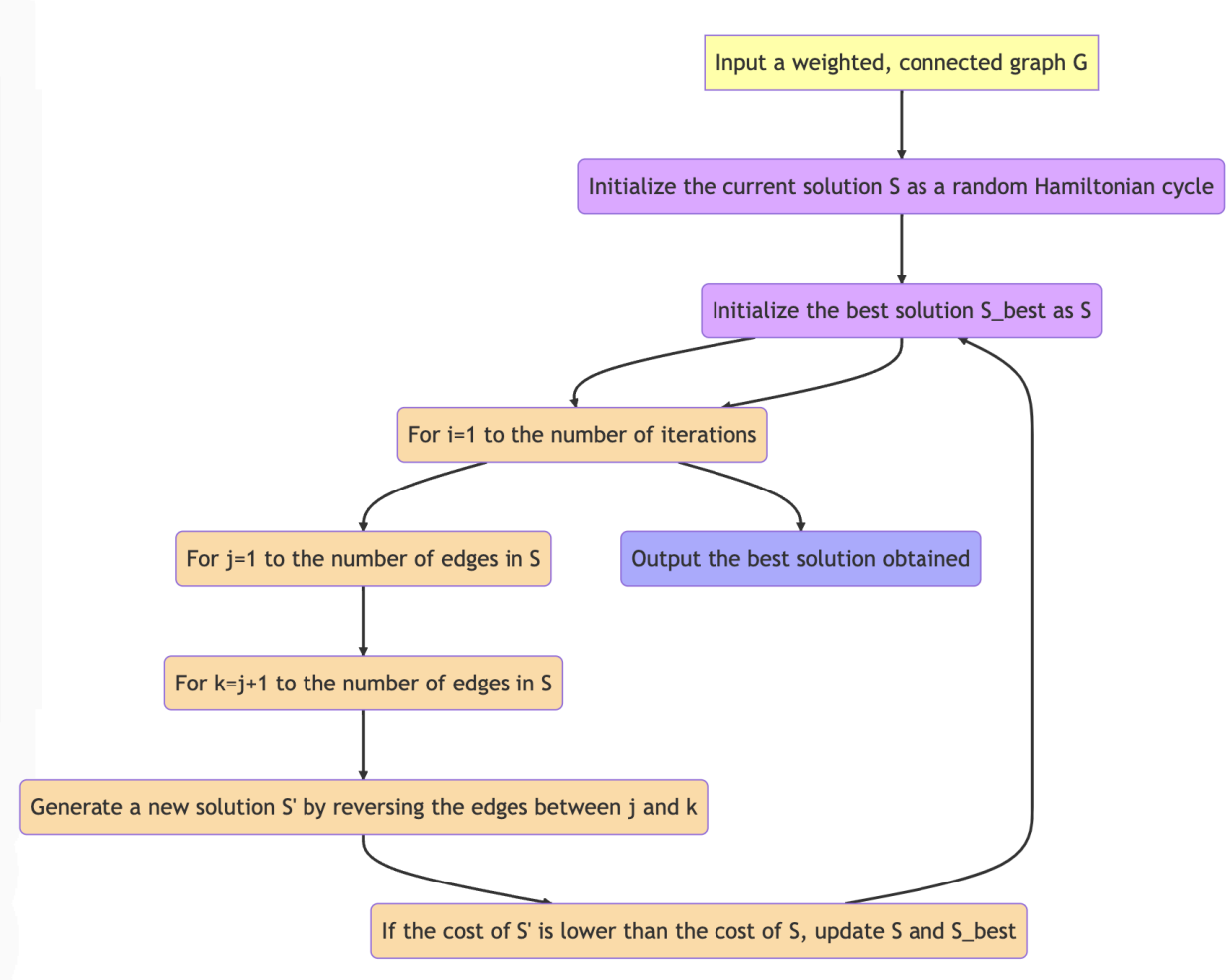
Flowchart: Simulated Annealing on Christofides Algorithm



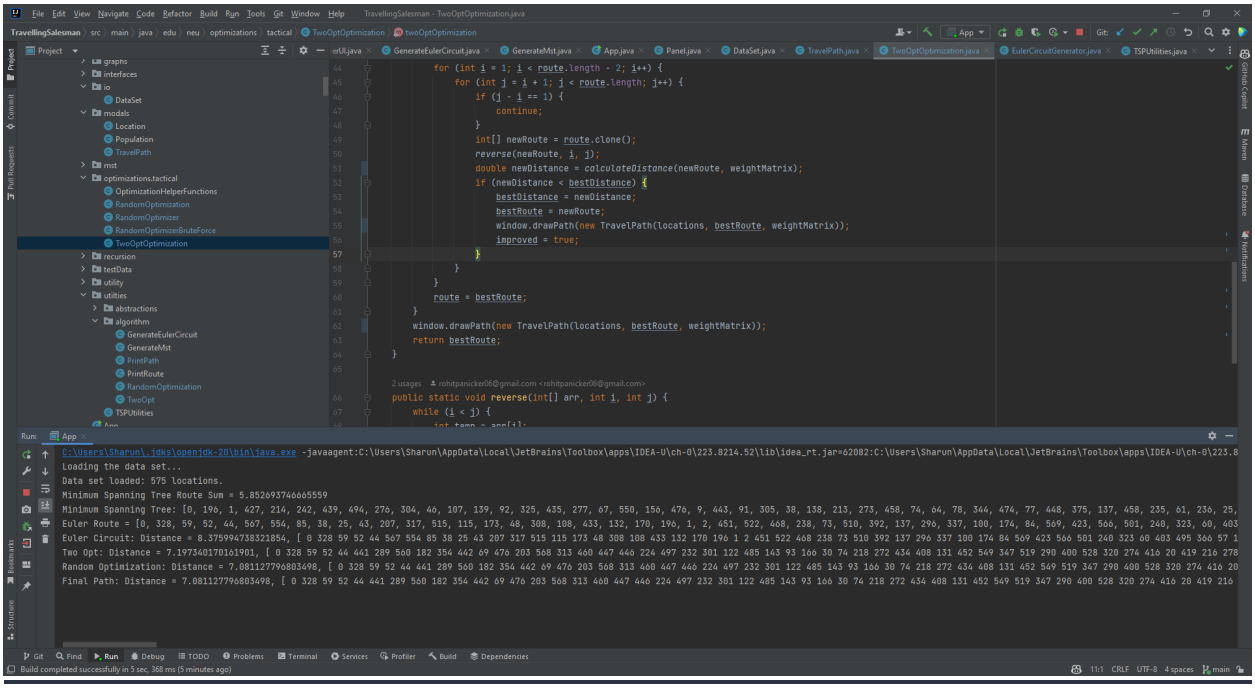
Flowchart: Random Optimization on Christofides Algorithm



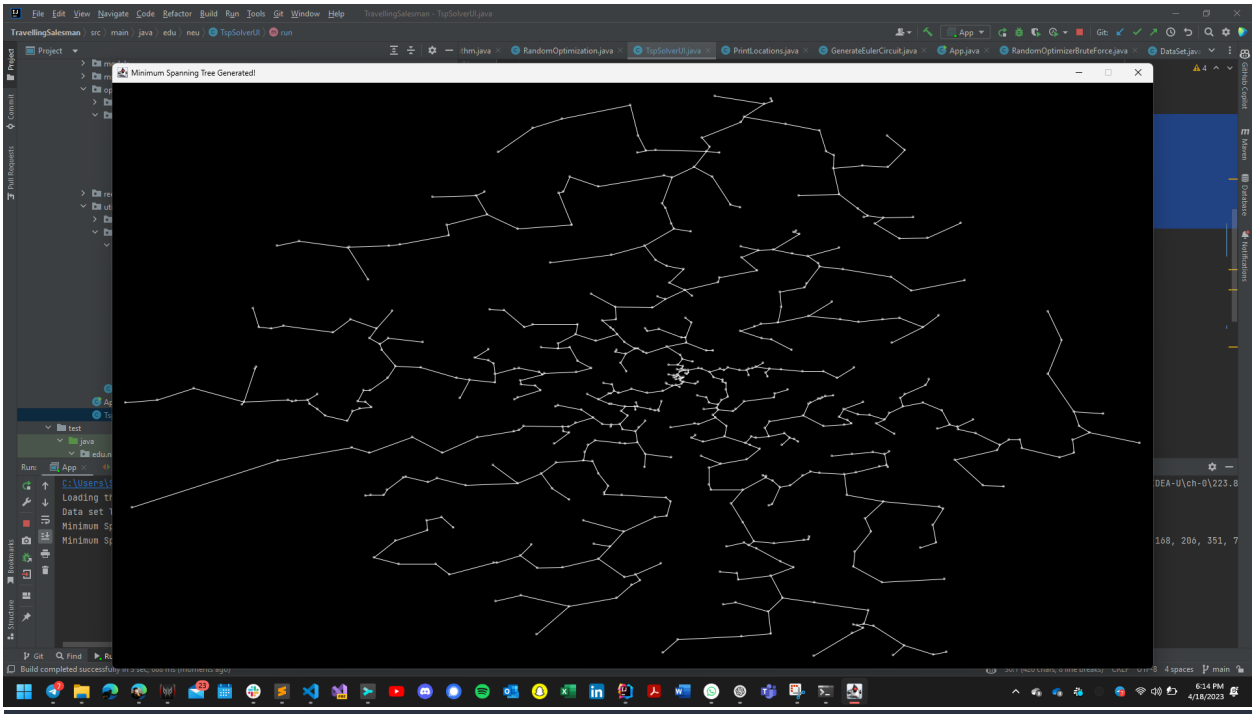
Flowchart: 2 Opt Optimization on Christofides Algorithm



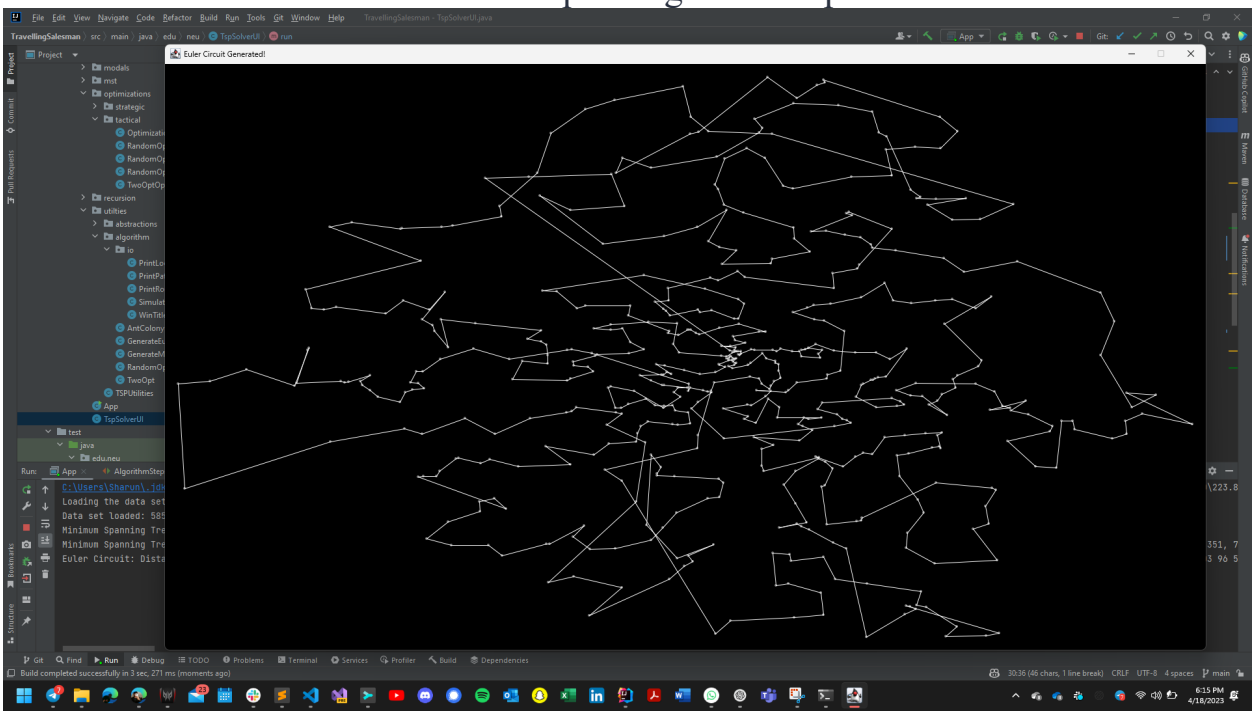
Running Path Output



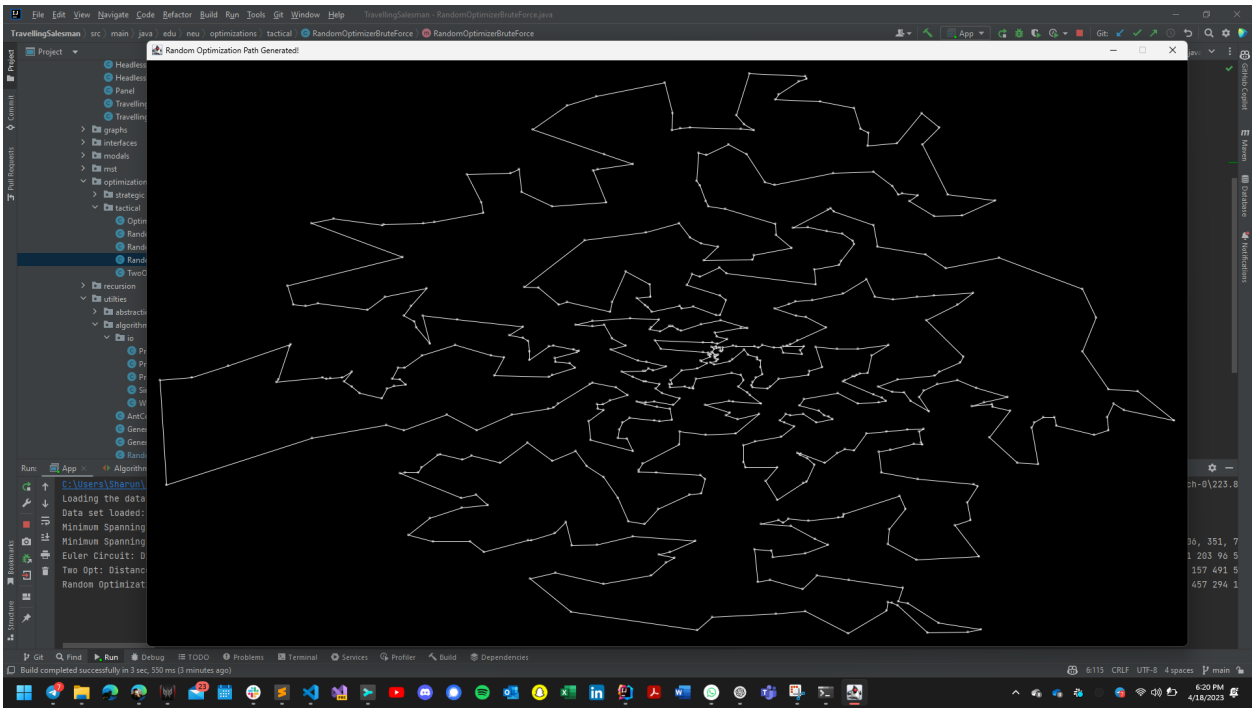
UI Outputs



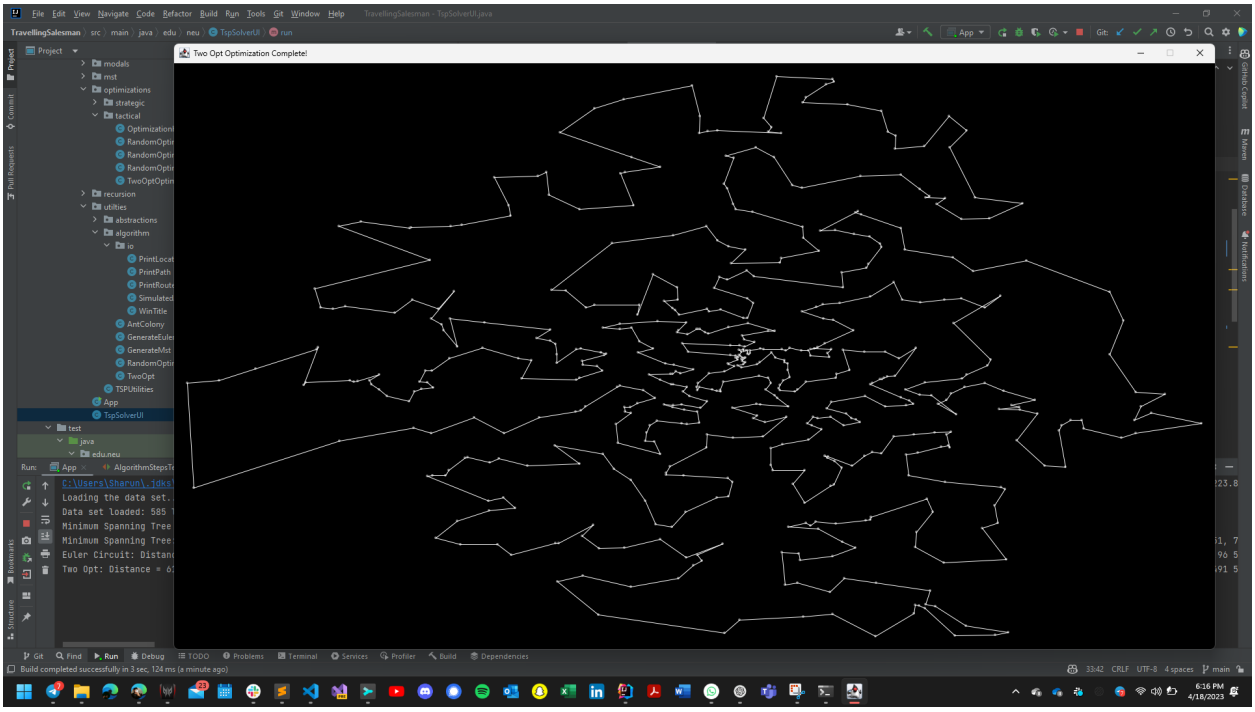
Minimum Spanning Tree Output



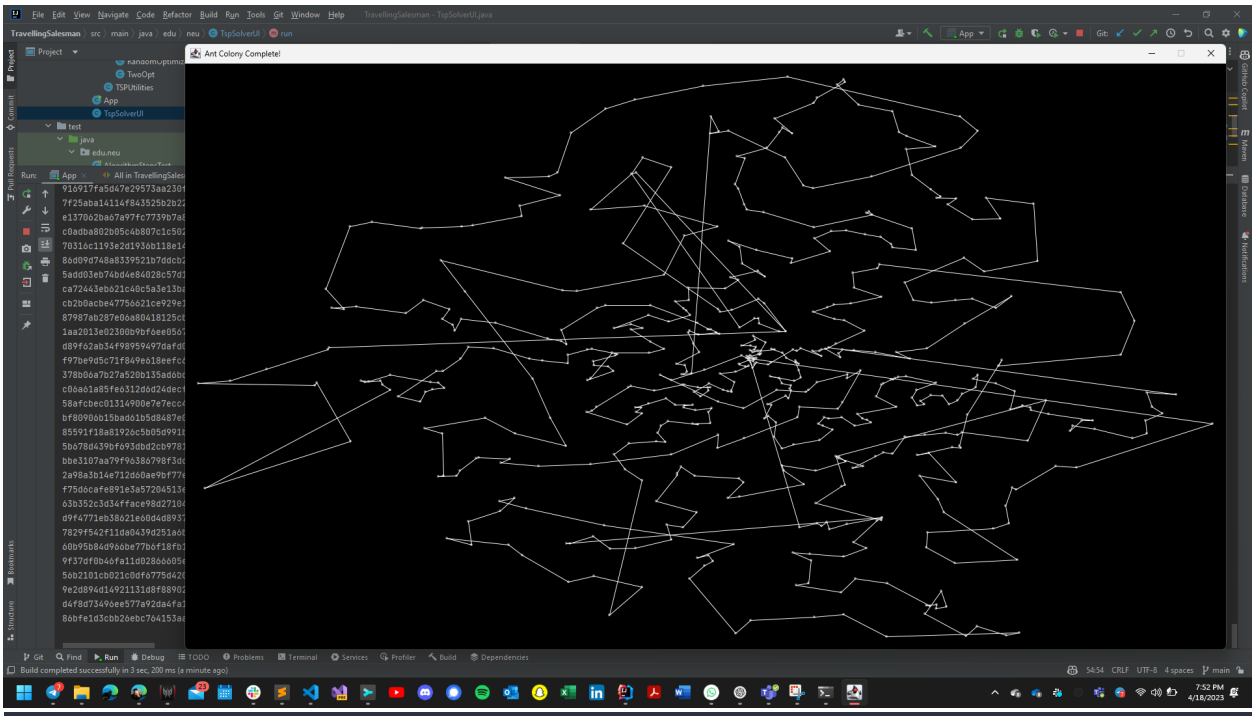
Euler Circuit Output



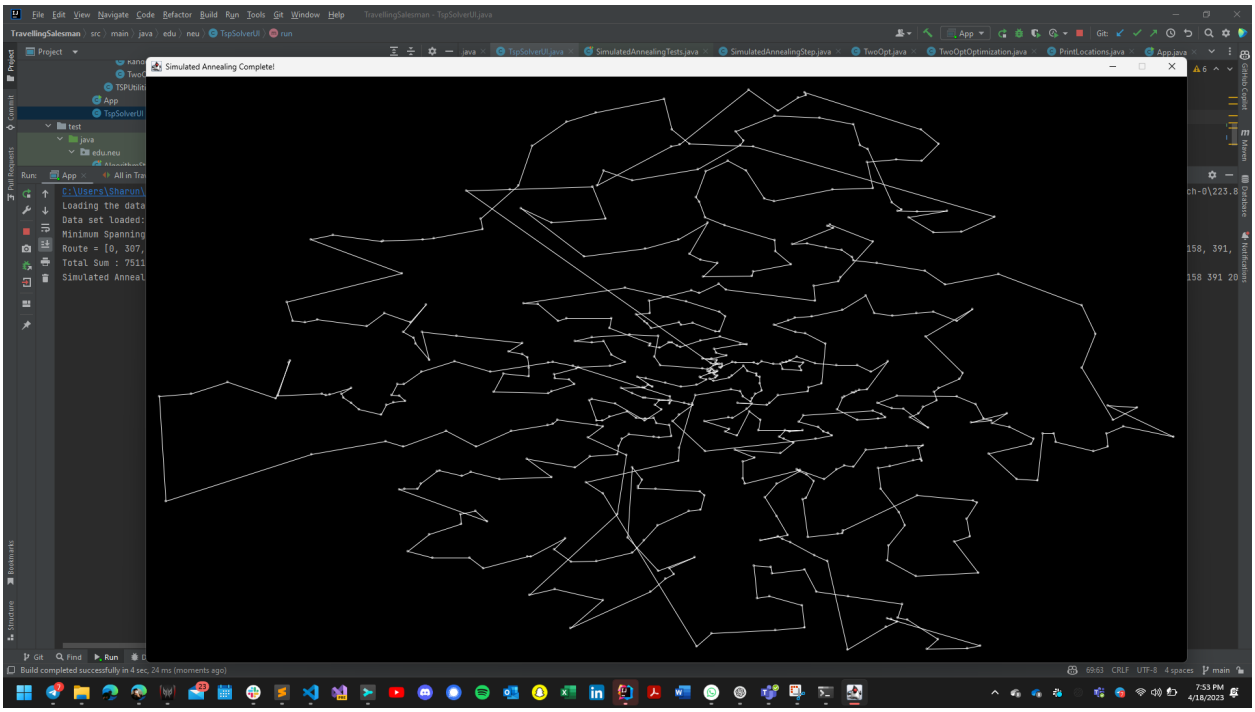
Random Optimization Output



Two Opt Optimization Output



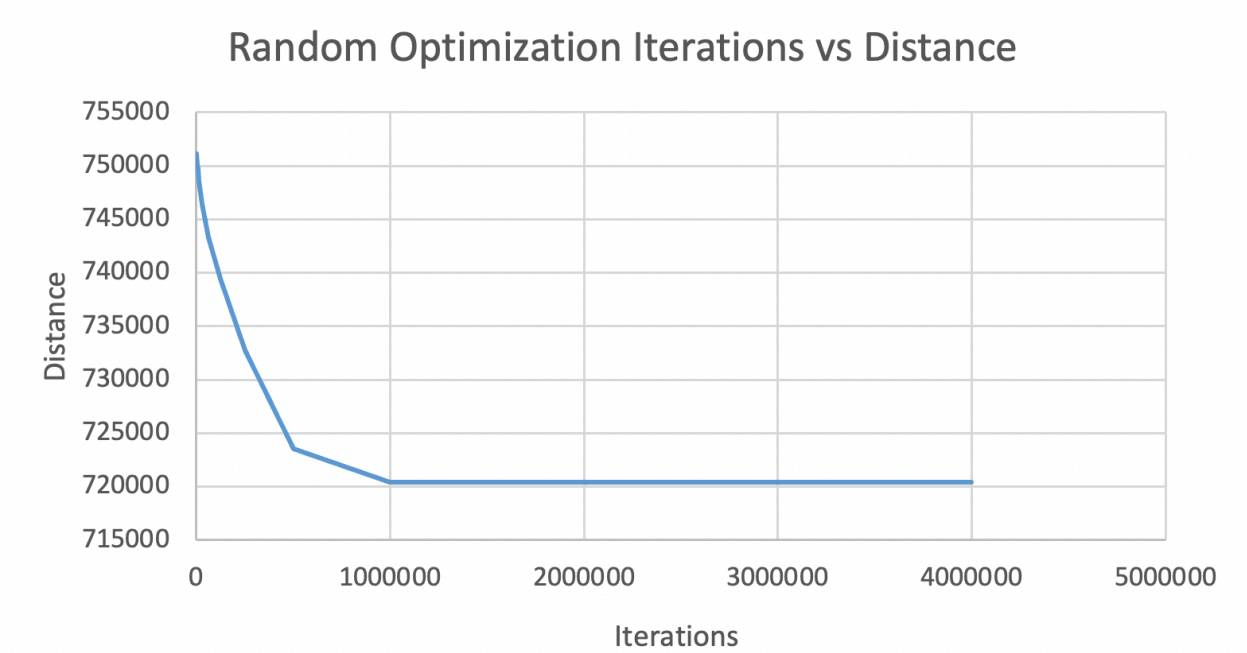
Ant Colony Optimization Output



Simulated Annealing Output

Observations and Graphical Analysis

1. Random Swapping:

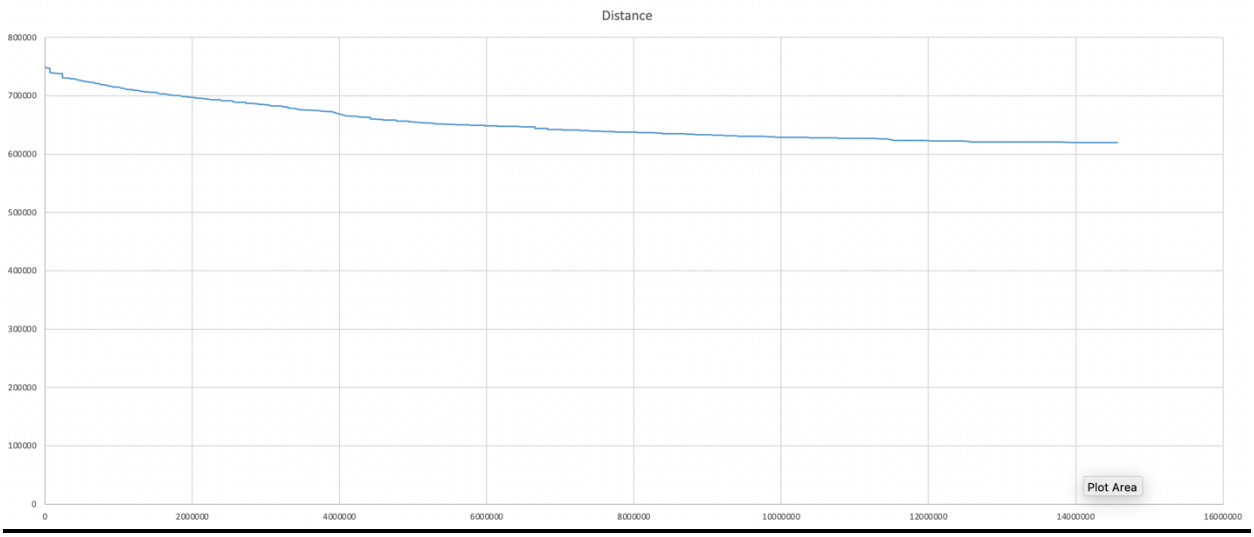


For the random swapping optimization technique, the graph is plotted by increasing the number of iterations and noting the distance from the optimization technique.

The graph points we noted are as follows for the random swapping optimization:

Iterations	Distance
122	751122.4407
244	751122.4407
488	751122.4407
976	751122.4407
1953	751122.4407
3906	750053.0943
7812	749818.9776
15625	748602.8608
31250	746428.4734
62500	743343.6213
125000	739425.2675
250000	732744.2363
500000	723575.4511
1000000	720431.4358
2000000	720431.4358
4000000	720431.4358

2. 2-Opt Optimization:



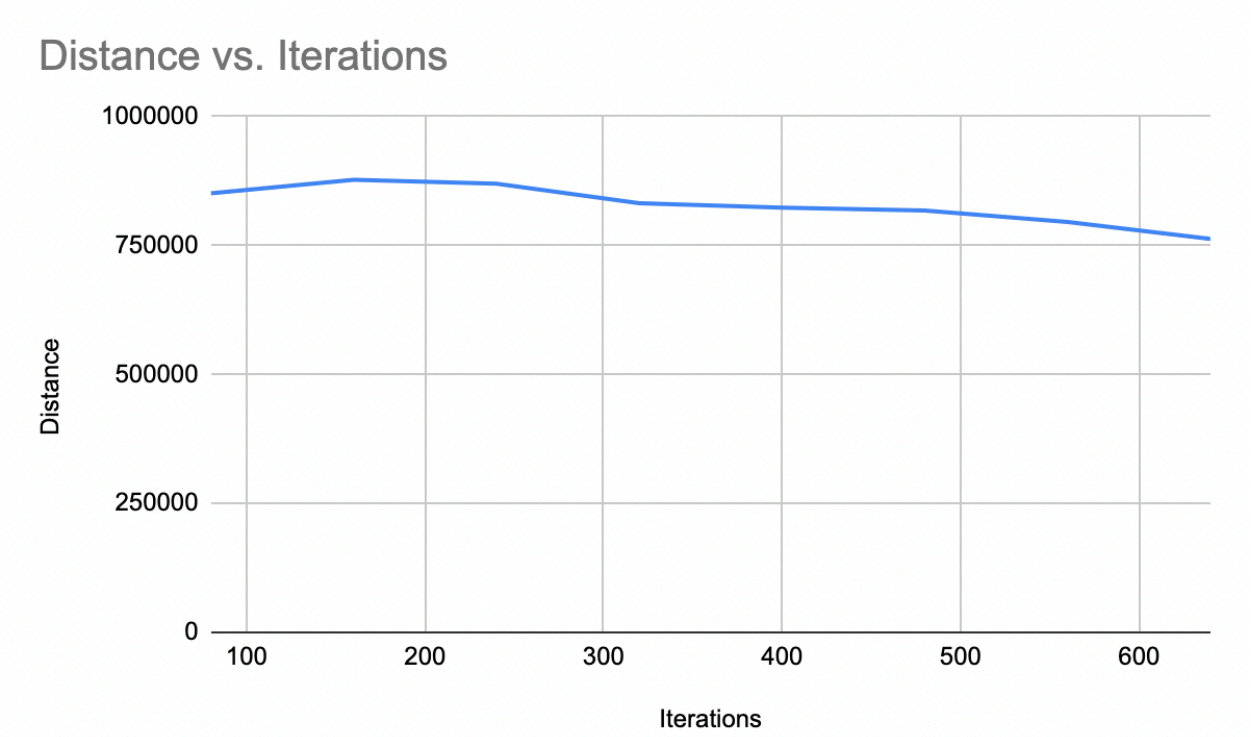
This is the graph we observed for the 2 Opt Optimization technique on the Christofides Algorithm. The time complexity for this is $O(n^2)$.

This graph is drawn by increasing iterations and noting the distance. The sample in data we noted is as follows:

The sample in data points noted are as follows:

Iteration	Distance
118	749980.3853
571	749006.7408
5322	748886.8199
9197	747458.9762
62808	746988.994
63734	739315.6138
170353	738173.5585
207183	737799.9555

3. Ant Colony Optimization:



For ant colony optimization, we have changed the number of iterations for testing the performance in calculating distance and the following values are given as parameters:

Alpha – 1.0: It is used to determine the weight and pheromone trail is used while ant goes to the next node of the graph. We provided value 1.0 to alpha.

Beta – 5.0: It is used to determine the weight and heuristic information is used while ant goes to the next node of the graph. We provided value 5.0 to beta.

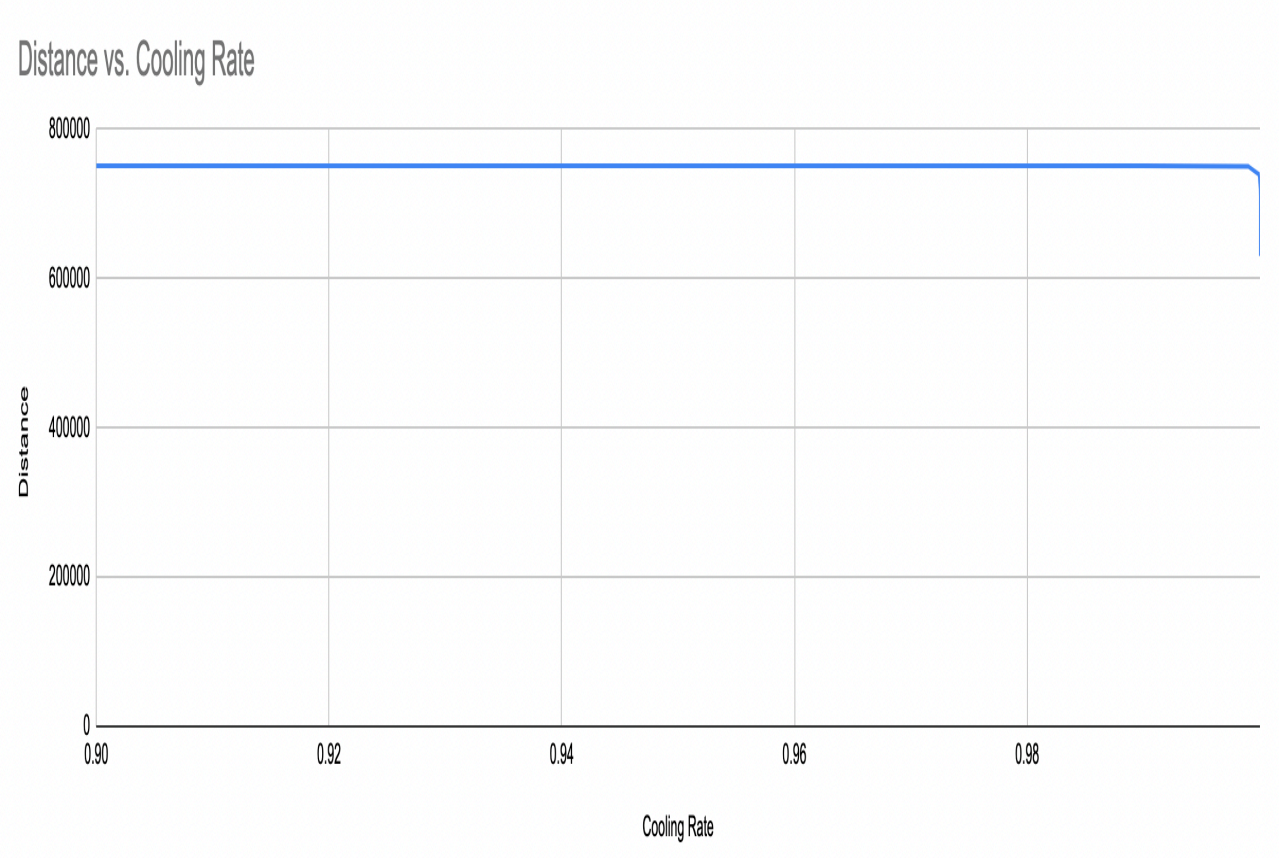
Evaporation – 0.5: It is rate of diminishing the pheromone trail while performing ant colony optimization. Usually, it varies between 0.1 and 0.7. We chose 0.5 as evaporation rate.

Initial Pheromone Level – It is the initial value of phenome level that an ant puts on the edge in graph. Higher value means the pheromone is kept more on that edge. We have given 0.1 as the initial pheromone level.

The values are as follows for the ant colony optimization:

Iterations	Distance
80	851370.129
160	877458.29
240	869671.696
320	832190.79
400	823554.436
480	818038.918
560	795778.215
640	762934.704

Simulated Annealing Performance Testing Graph:



The cooling rate is varied to perform testing on the simulated annealing optimization on the christofides algorithm.

The accuracy increases when cooling rate is increased. The below are the observations we have made for the Simulated Annealing optimization on the Christofides algorithm.

Cooling Rate	Distance
0.9	751122.44
0.99	751122.44
0.999	750581.77
0.9999	739917.994
0.99999	720644.898
0.999999	710681.195
0.9999999	630317.251

However, we have tested this for more than 345 more iterations apart from the table screenshot given above. And the graph for all these iterations is attached above.

Christofides Algorithm:

The Christofides algorithm is a heuristic algorithm used to find an approximate solution to the traveling salesman problem. It is based on finding a minimum weight perfect matching of the input graph and then constructing a Eulerian tour by adding edges to the matching. Then, the algorithm finds a Hamiltonian cycle by traversing the Eulerian tour while skipping already visited vertices.

One way to visualize the performance of the Christofides algorithm is by plotting the approximation ratio as a function of the input size. The approximation ratio is defined as the ratio of the weight of the Hamiltonian cycle found by the algorithm to the weight of the optimal Hamiltonian cycle. A ratio of 1 indicates that the algorithm found the optimal solution, while a ratio greater than 1 indicates that the algorithm found a suboptimal solution. By plotting the approximation ratio as a function of the input size, we can see how the performance of the algorithm changes as the input size increases. We can also compare the performance of the Christofides algorithm to other heuristic algorithms and exact algorithms.

Simulated Annealing:

Simulated annealing is a stochastic optimization technique inspired by the annealing process in metallurgy. It is used to find an approximate solution to optimization problems by iteratively improving a candidate solution. The algorithm works by randomly perturbing the current solution and accepting the perturbation if it improves the objective function. The probability of accepting a perturbation that worsens the objective function decreases as the algorithm progresses, mimicking the cooling process in metallurgy.

To apply simulated annealing to the Christofides algorithm, we can start with the approximate solution obtained by the Christofides algorithm and use simulated annealing to iteratively improve the solution. The perturbation can be achieved by swapping two edges in the Hamiltonian cycle. The acceptance probability can be determined based on the change in the weight of the Hamiltonian cycle.

Ant Colony Optimization:

Ant colony optimization is a metaheuristic algorithm inspired by the foraging behavior of ants. It is used to find an approximate solution to optimization problems by iteratively constructing solutions based on the pheromone trails left by the ants. The algorithm works by constructing a solution by probabilistically choosing edges based on the pheromone levels and the heuristic information, which guides the search towards promising regions of the search space. After constructing a solution, the pheromone levels are updated based on the quality of the solution.

To apply ant colony optimization to the Christofides algorithm, we can use the pheromone trail to bias the selection of edges in the construction of the Hamiltonian cycle. The heuristic information can be based on the distance between vertices or the weight of the edges. After constructing a Hamiltonian cycle, the pheromone levels can be updated based on the length of the cycle.

Random Optimization:

Random optimization is a simple optimization technique that involves generating random solutions and selecting the best one. It is often used as a baseline to compare the performance of more sophisticated optimization techniques.

To apply random optimization to the Christofides algorithm, we can generate random Hamiltonian cycles and select the one with the lowest weight. The number of random solutions generated can be determined based on the input size and the desired level of accuracy.

2-opt Optimization:

2-opt optimization is a local search optimization technique used to improve an existing solution by iteratively removing two edges from the solution and reconnecting the resulting paths to form a new solution. The algorithm continues until no further improvements can be made.

To apply 2-opt optimization to the Christofides algorithm, we can start with the Hamiltonian cycle obtained by the Christofides algorithm and apply the 2-opt optimization to improve the solution. In each iteration, we can remove two edges from the Hamiltonian cycle and reconnect the resulting paths to form a new cycle. If the new cycle has a lower weight than the previous one, we can update the solution. The algorithm continues until no further improvements can be made.

Overall, these optimization techniques can be used to improve the performance of the Christofides algorithm by finding better solutions or improving existing solutions. Graphical analysis can help us understand how the performance of the algorithm changes as the input size increases and how it compares to other algorithms. Simulated annealing, ant colony optimization, random optimization, and 2-opt optimization are just a few examples of techniques that can be applied to the Christofides algorithm, and there may be other optimization techniques that can also be effective.

Observed Performance Outputs:

Results and Mathematical Analysis

Mathematical Analysis:
To create the distance matrix, we used the Haversine formula to determine the separation between the input edges. The distance between two places on the surface of a sphere can be calculated using the Haversine formula, which is a mathematical formula used in navigation.

Given two places latitudes and longitudes in our dataset, it is frequently used to determine their distance from one another.

The law of haversines, which states that for a triangle on the surface of a sphere, each angle's haversine is equal to the haversine of the sum of the other two angles less the product of the haversines of those angles, is the source of the Haversine formula.

The formula employs the trigonometric functions ‘sin’ and ‘cos’ to convert the latitudes and longitudes to angles, and the arcsine function to determine the angle between the two points. The formula makes the imperfectly accurate but widely regarded as accurate enough assumption that the object is spherical in shape.

The Haversine formula is given by:

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

where, d is the distance between the two points (in the same units as r, typically kilometers or miles)

r is the radius of the sphere (in the same units as d)

lat1 and lat2 are the latitudes of the two points (in degrees)

lon1 and lon2 are the longitudes of the two points (in degrees)

Simulated annealing is a process where an initial solution is used as a starting point, then iteratively new solutions are generated by making minor random changes to the first solution. The algorithm then analyzes the objective function for each potential solution and, using a probability distribution, determines whether to accept or reject the new answer. Based on the difference between the objective function values of the current and alternative solutions as well as a temperature parameter that decreases over time in accordance with a cooling schedule, the Metropolis-Hastings algorithm calculates the probability for simulated annealing. The cooling schedule is frequently designed to find a compromise between exploitation and exploration.

P(x',x,k,T) = exp[-(f(x') - f(x))/(kT)]

f(x) and temperature T at iteration k, and is the mathematical formula for the acceptance probability of a candidate solution.

When x is the existing solution, x' is a potential solution, k is a constant, T is the current temperature.

The Ant Colony Optimization technique is one of the optimization techniques we have utilized to obtain the best TSP solution. The Ant colony algorithm determines the pheromone trail levels for each edge in the graph and updates them based on how well the ants solutions work. To determine the likelihood that an ant would select a certain edge, the pheromone level and a assumed value, which denotes the edge's desirability based on elements like its length or cost and are combined.

The formula appears as follows:

p(i,j) is equal to [(i,j) (i,j)] / [(i,k) (i,k)]

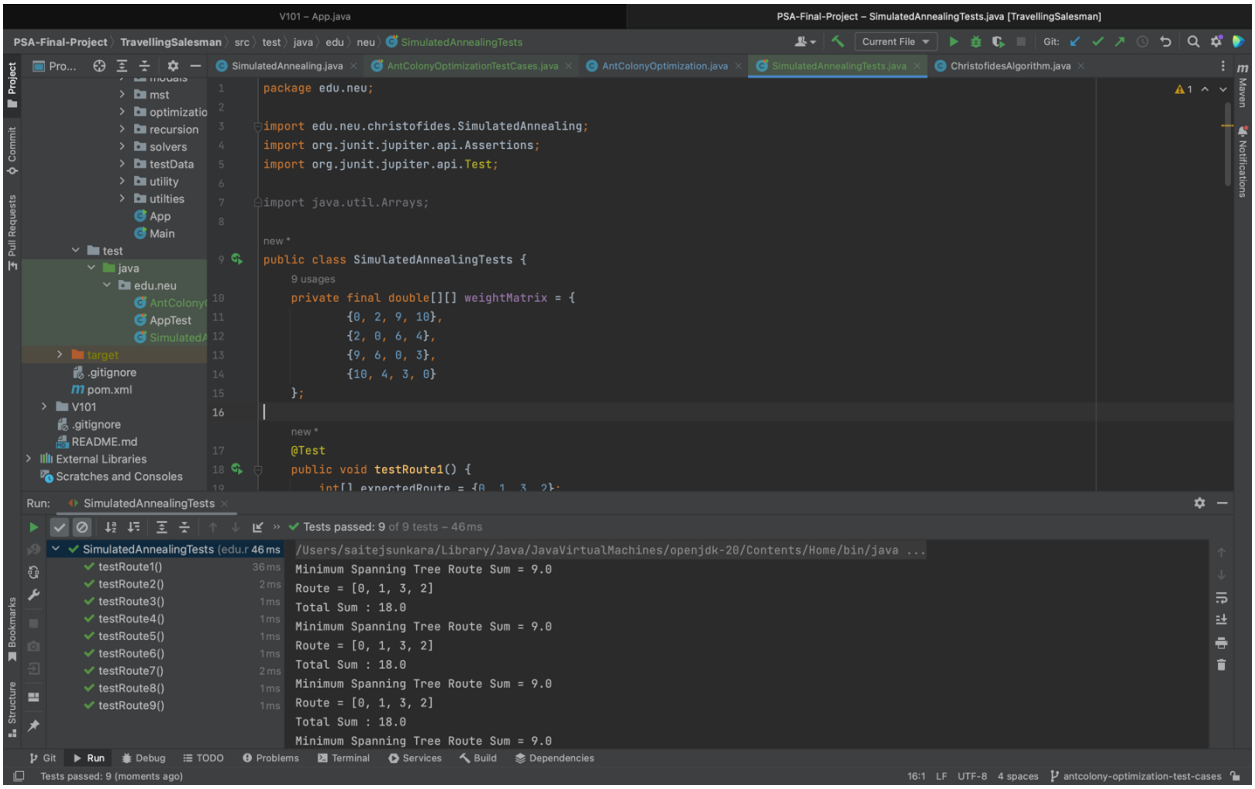
where p(i,j) is the likelihood that an ant would move from node i to node j, (i,j) is the pheromone level on that edge, (i,j) is the edge's heuristic value, and and are parameters that regulate the relative weights of the pheromone and heuristic values, respectively. The probability of all the edges leaving node i are totaled in the denominator.

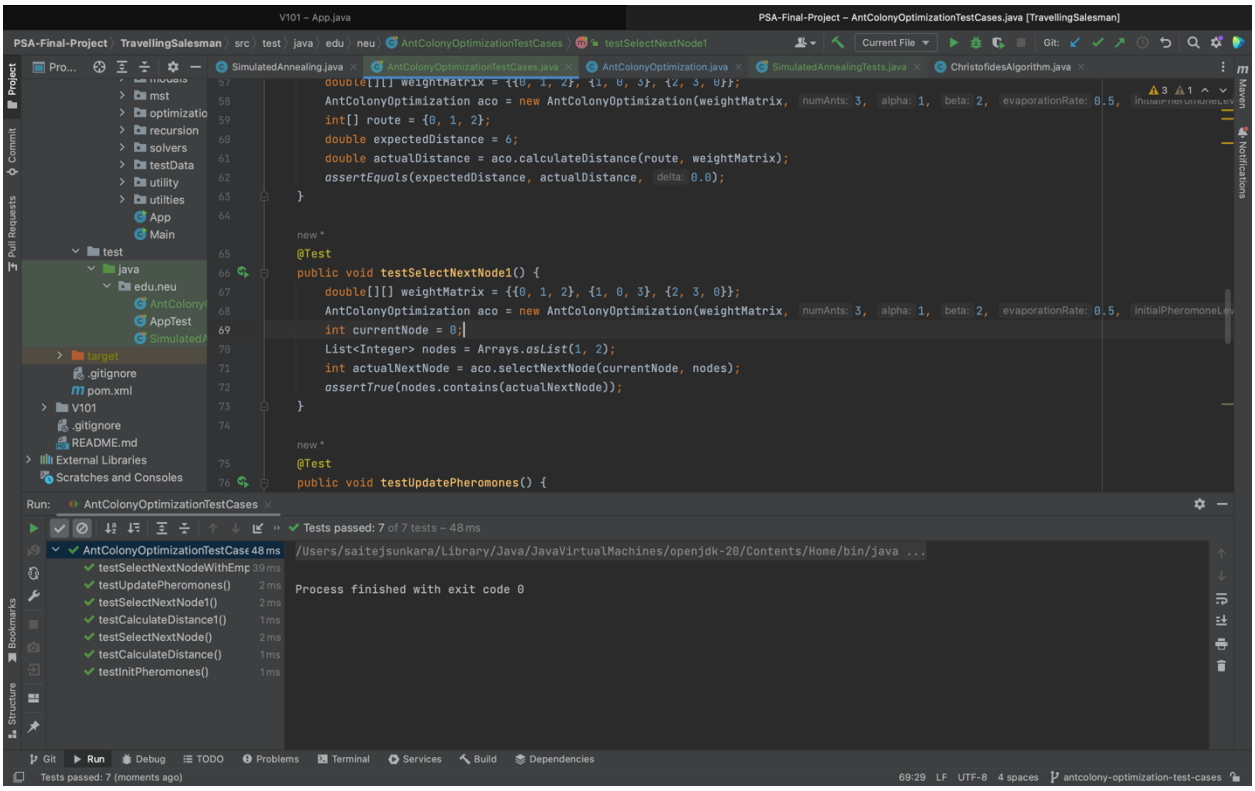
In order to determine the MST (minimum spanning tree), Prims' Algorithm is utilized.
The Hungarian Algorithm was used to determine the least weight perfect matching.
TSP is solved using the Christofides algorithm.

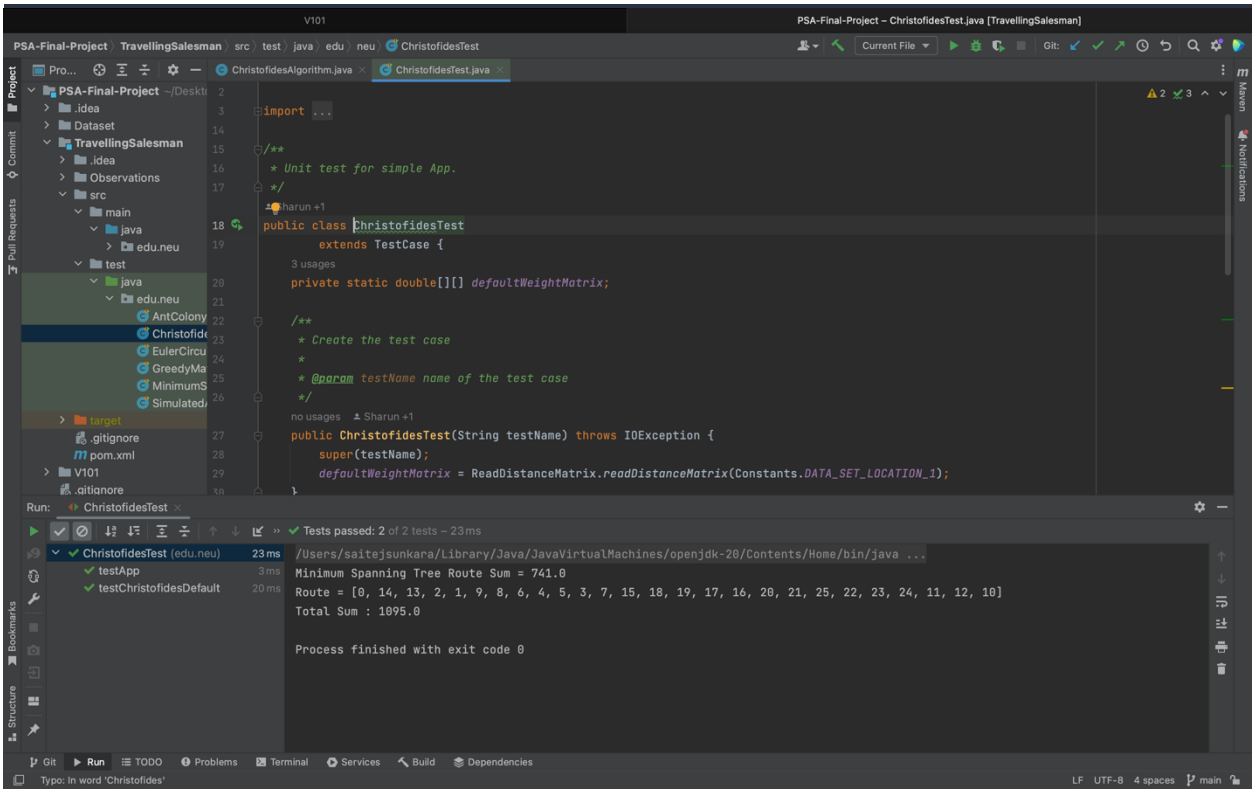
To obtain the TSP's optimal solution, we have applied optimizations such as 2 opt and random swap.

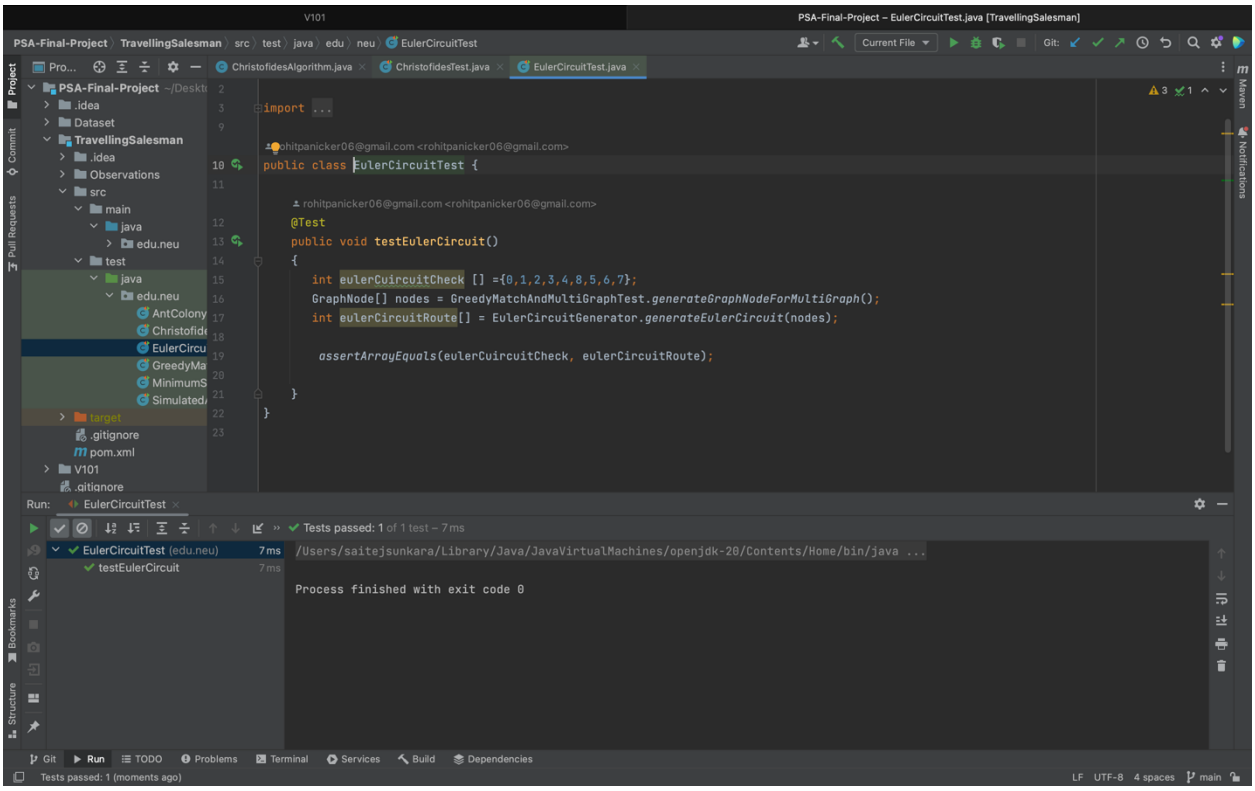
It might be challenging to estimate the precise time complexity for optimization methods like Simulated Annealing and Ant Colony Optimization because it depends on the input parameters.

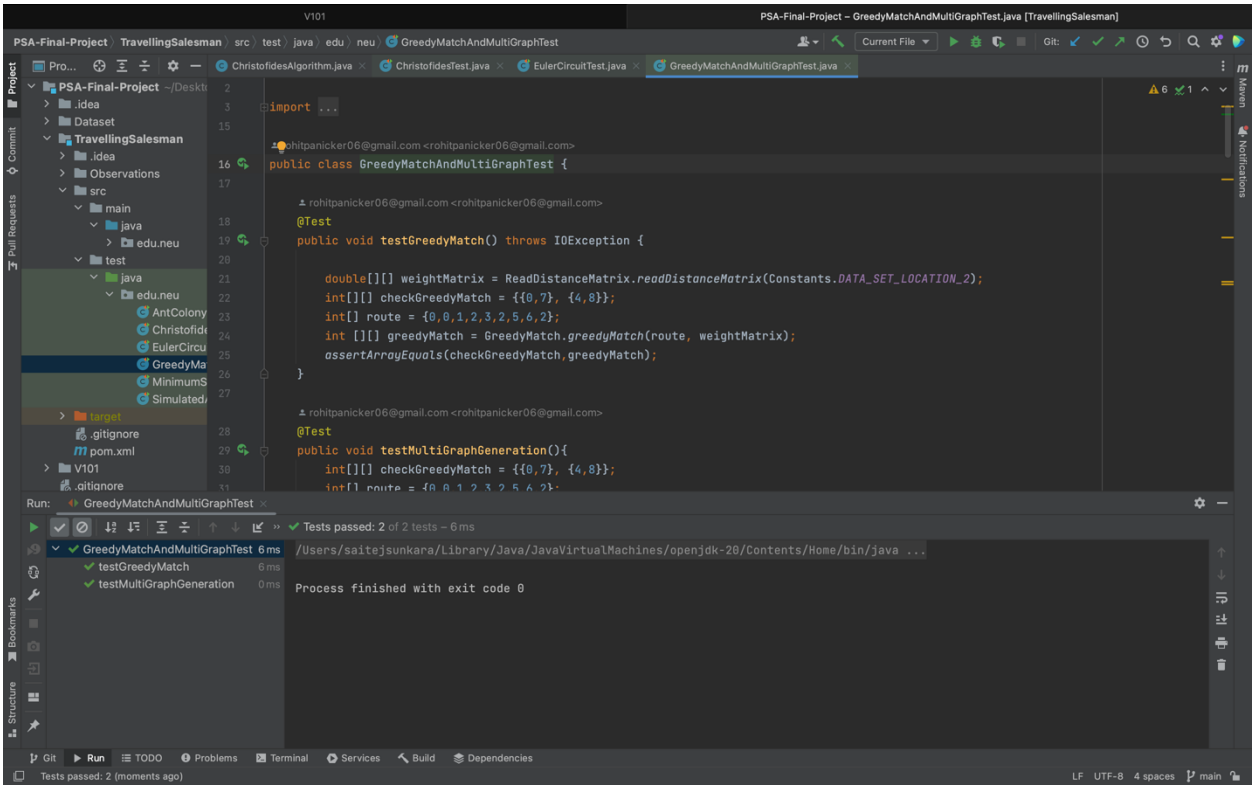
Unit Tests


V101 – App.java
PSA-Final-Project – SimulatedAnnealingTests.java [TravellingSalesman]
PSA-Final-Project 〉 TravellingSalesman 〉 src 〉 test 〉 java 〉 edu 〉 neu 〉 SimulatedAnnealingTests
Current File
Git:
SimulatedAnnealing.java
AntColonyOptimizationTestCases.java
AntColonyOptimization.java
SimulatedAnnealingTests.java
ChristofidesAlgorithm.java
package edu.neu;
import edu.neu.christofides.SimulatedAnnealing;
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import java.util.Arrays;
new *
public class SimulatedAnnealingTests {
9 usages
private final double[][] weightMatrix = {
{0, 2, 9, 10},
{2, 0, 6, 4},
{9, 6, 0, 3},
{10, 4, 3, 0}
};
new *
@Test
public void testRoute1() {
Run:
SimulatedAnnealingTests
Tests passed: 9 of 9 tests – 46 ms
SimulatedAnnealingTests (edu.r 46 ms
testRoute1() 36 ms
testRoute2() 2 ms
testRoute3() 1 ms
testRoute4() 1 ms
testRoute5() 1 ms
testRoute6() 1 ms
testRoute7() 2 ms
testRoute8() 1 ms
testRoute9() 1 ms
/Users/saitejsunkara/Library/Java/JavaVirtualMachines/openjdk-20/Contents/Home/bin/java ...
Minimum Spanning Tree Route Sum = 9.0
Route = [0, 1, 3, 2]
Total Sum : 18.0
Minimum Spanning Tree Route Sum = 9.0
Route = [0, 1, 3, 2]
Total Sum : 18.0
Minimum Spanning Tree Route Sum = 9.0
Route = [0, 1, 3, 2]
Total Sum : 18.0
Minimum Spanning Tree Route Sum = 9.0
Git
Run
Debug
TODO
Problems
Terminal
Services
Build
Dependencies
Tests passed: 9 (moments ago)
16:1
LF
UTF-8
4 spaces
antcolony-optimization-test-cases


V101 – App.java
PSA-Final-Project – AntColonyOptimizationTestCases.java [TravellingSalesman]
PSA-Final-Project 〉 TravellingSalesman 〉 src 〉 test 〉 java 〉 edu 〉 neu 〉 AntColonyOptimizationTestCases 〉 testSelectNextNode1
Current File
Git:
SimulatedAnnealing.java
AntColonyOptimizationTestCases.java
AntColonyOptimization.java
SimulatedAnnealingTests.java
ChristofidesAlgorithm.java
double[][] weightMatrix = {{0, 1, 2}, {1, 0, 3}, {2, 3, 0}};
AntColonyOptimization aco = new AntColonyOptimization(weightMatrix, numAnts: 3, alpha: 1, beta: 2, evaporationRate: 0.5,
int[] route = {0, 1, 2};
double expectedDistance = 6;
double actualDistance = aco.calculateDistance(route, weightMatrix);
assertEquals(expectedDistance, actualDistance, delta: 0.0);
}
new *
@Test
public void testSelectNextNode1() {
double[][] weightMatrix = {{0, 1, 2}, {1, 0, 3}, {2, 3, 0}};
AntColonyOptimization aco = new AntColonyOptimization(weightMatrix, numAnts: 3, alpha: 1, beta: 2, evaporationRate: 0.5,
int currentNode = 0;
List<Integer> nodes = Arrays.asList(1, 2);
int actualNextNode = aco.selectNextNode(currentNode, nodes);
assertTrue(nodes.contains(actualNextNode));
}
new *
@Test
public void testUpdatePheromones() {
Run: AntColonyOptimizationTestCases
Tests passed: 7 of 7 tests – 48 ms
AntColonyOptimizationTestCase 48 ms
testSelectNextNodeWithEmp 39 ms
testUpdatePheromones() 2 ms
testSelectNextNode1() 2 ms
testCalculateDistance1() 1 ms
testSelectNextNode() 2 ms
testCalculateDistance() 1 ms
testInitPheromones() 1 ms
/Users/saitejsunkara/Library/Java/JavaVirtualMachines/openjdk-20/Contents/Home/bin/java ...
Process finished with exit code 0
Git Run Debug TODO Problems Terminal Services Build Dependencies
Tests passed: 7 (moments ago)
69:29 LF UTF-8 4 spaces antcolony-optimization-test-cases


V101
PSA-Final-Project – ChristofidesTest.java [TravellingSalesman]
PSA-Final-Project 〉 TravellingSalesman 〉 src 〉 test 〉 java 〉 edu 〉 neu 〉 ChristofidesTest
Current File
Git:
ChristofidesAlgorithm.java
ChristofidesTest.java
import ...
/**
* Unit test for simple App.
*/
Sharun +1
public class ChristofidesTest
extends TestCase {
3 usages
private static double[][] defaultWeightMatrix;
/**
* Create the test case
*
* @param testName name of the test case
*/
no usages Sharun +1
public ChristofidesTest(String testName) throws IOException {
super(testName);
defaultWeightMatrix = ReadDistanceMatrix.readDistanceMatrix(Constants.DATA_SET_LOCATION_1);
}
Run: ChristofidesTest
Tests passed: 2 of 2 tests – 23 ms
ChristofidesTest (edu.neu) 23 ms
testApp 3 ms
testChristofidesDefault 20 ms
/Users/saitejsunkara/Library/Java/JavaVirtualMachines/openjdk-20/Contents/Home/bin/java ...
Minimum Spanning Tree Route Sum = 741.0
Route = [0, 14, 13, 2, 1, 9, 8, 6, 4, 5, 3, 7, 15, 18, 19, 17, 16, 20, 21, 25, 22, 23, 24, 11, 12, 10]
Total Sum : 1095.0
Process finished with exit code 0
Git Run TODO Problems Terminal Services Build Dependencies
Typo: In word 'Christofides'
LF UTF-8 4 spaces main


PSA-Final-Project – EulerCircuitTest.java [TravellingSalesman]
PSA-Final-Project 〉 TravellingSalesman 〉 src 〉 test 〉 java 〉 edu 〉 neu 〉 EulerCircuitTest
ChristofidesAlgorithm.java
ChristofidesTest.java
EulerCircuitTest.java
import ...
rohitpanicker06@gmail.com <rohitpanicker06@gmail.com>
public class EulerCircuitTest {
rohitpanicker06@gmail.com <rohitpanicker06@gmail.com>
@Test
public void testEulerCircuit()
{
int eulerCuircuitCheck [] ={0,1,2,3,4,8,5,6,7};
GraphNode[] nodes = GreedyMatchAndMultiGraphTest.generateGraphNodeForMultiGraph();
int eulerCircuitRoute[] = EulerCircuitGenerator.generateEulerCircuit(nodes);
assertArrayEquals(eulerCuircuitCheck, eulerCircuitRoute);
}
}
Run: EulerCircuitTest
Tests passed: 1 of 1 test – 7 ms
EulerCircuitTest (edu.neu) 7 ms
testEulerCircuit 7 ms
/Users/saitejsunkara/Library/Java/JavaVirtualMachines/openjdk-20/Contents/Home/bin/java ...
Process finished with exit code 0
Tests passed: 1 (moments ago)


PSA-Final-Project – GreedyMatchAndMultiGraphTest.java [TravellingSalesman]
PSA-Final-Project 〉 TravellingSalesman 〉 src 〉 test 〉 java 〉 edu 〉 neu 〉 GreedyMatchAndMultiGraphTest
ChristofidesAlgorithm.java
ChristofidesTest.java
EulerCircuitTest.java
GreedyMatchAndMultiGraphTest.java
import ...
rohitpanicker06@gmail.com <rohitpanicker06@gmail.com>
public class GreedyMatchAndMultiGraphTest {
rohitpanicker06@gmail.com <rohitpanicker06@gmail.com>
@Test
public void testGreedyMatch() throws IOException {
double[][] weightMatrix = ReadDistanceMatrix.readDistanceMatrix(Constants.DATA_SET_LOCATION_2);
int[][] checkGreedyMatch = {{0,7}, {4,8}};
int[] route = {0,0,1,2,3,2,5,6,2};
int [][] greedyMatch = GreedyMatch.greedyMatch(route, weightMatrix);
assertArrayEquals(checkGreedyMatch,greedyMatch);
}
rohitpanicker06@gmail.com <rohitpanicker06@gmail.com>
@Test
public void testMultiGraphGeneration(){
int[][] checkGreedyMatch = {{0,7}, {4,8}};
Run: GreedyMatchAndMultiGraphTest
Tests passed: 2 of 2 tests – 6 ms
GreedyMatchAndMultiGraphTest 6 ms
testGreedyMatch 6 ms
testMultiGraphGeneration 0 ms
/Users/saitejsunkara/Library/Java/JavaVirtualMachines/openjdk-20/Contents/Home/bin/java ...
Process finished with exit code 0
Tests passed: 2 (moments ago)

CONCLUSION:

In this paper, we have analyzed and investigated the effectiveness of Christofides algorithm and various strategical and tactical optimization techniques in generating a highly optimized tour for Travelling Salesman Problem.

As mentioned at the very start of this report, we were able to generate a tour with 3/2 times the optimal path. We were also able to further optimize the path by using strategical and tactical optimization techniques such as Random Swapping technique, Ant Colony technique, simulated annealing and 2-opt optimization technique.

While performing various experiments we observed that the combination of 2-opt with Random Optimization technique gave a better result than other optimizations.
We also implemented a live UI which displayed the live swapping of vertices and edges.

While there are several other solution spaces which can be explored through a variety of combination of different optimization techniques, this paper provides a huge depth for initial practitioners and for supporting future exploration techniques.

References

Springer, William M. “Review of the Traveling Salesman Problem: A Computational Study by Applegate, Bixby, Chvátal, and Cook (Princeton University Press).” *ACM SIGACT News*, vol. 40, no. 2, 2009, pp. 30–32., https://doi.org/10.1145/1556154.1556162.

Johnson, R., and M. G. Pilcher. “The Traveling Salesman Problem, Edited by E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B Shmoys, John Wiley & Sons, Chichester, 1985, 463 Pp.” *Networks*, vol. 18, no. 3, 1988, pp. 253–254., https://doi.org/10.1002/net.3230180309.

Worst-Case Analysis of a New Heuristic for the Travelling - Researchgate. https://www.researchgate.net/journal/SN-Operations-Research-Forum-2662-2556/publication/235067784_Worst-Case_Analysis_of_a_New_Heuristic_for_the_Traveling_Salesman_Problem/links/6220f41ce474e407ea1fc8a4/Worst-Case-Analysis-of-a-New-Heuristic-for-the-Traveling-Salesman-Problem.pdf?_sg%5B0%5D=started_experiment_milestone&_sg%5B1%5D=started_experiment_milestone&origin=journalDetail.

Wong, Richard T. “Combinatorial Optimization: Algorithms and Complexity (Christos H. Papadimitriou and Kenneth Steiglitz).” *SIAM Review*, vol. 25, no. 3, 1983, pp. 424–425., https://doi.org/10.1137/1025101.

A Method for Solving Traveling-Salesman Problems - JSTOR. https://www.jstor.org/stable/167074.

Dorigo, Marco, and Thomas Stützle. “Ant Colony Optimization.” 2004, https://doi.org/10.7551/mitpress/1290.001.0001.

Simulated Annealing and Boltzmann Machines: A Stochastic Approach to ... https://epubs.siam.org/doi/10.1137/1033080.