Program Structures and Algorithms
Spring 2023
INFO6205 Team Project

THE TRAVELING SALESMAN PROBLEM

Team Members:

1. Venkatesha Matam from Section-3 (NUID: 002740702)
2. Naga Venkata Nishanth Sayana from Section-1 (NUID: 002930970)

Introduction:

Aim: The goal of this project is to explore and evaluate various heuristic algorithms for solving the Traveling Salesman Problem (TSP), such as 2-opt, 3-opt, ant colony, and simulated annealing algorithms. The main goal is to find the shortest path that visits every vertex in each graph exactly once and returns to the starting vertex, as well as to compare the performance and effectiveness of the implemented algorithms. The report will describe the implementation and operation of each algorithm, compare their performance in terms of solution quality and execution time, and discuss any trade-offs, limitations, or drawbacks. The report will also compare the results of the implemented algorithms to known optimal solutions or other state-of-the-art TSP algorithms, to provide insights into the TSP problem-solving approach and potential avenues for improvement.

Approach:

Approach used to find the TSP and its weight:

Implemented Christofides algorithm:

1. Represent the problem as a graph: The first step is to represent the TSP problem instance as an undirected graph, with the vertices representing the cities to be visited and the edges representing the distances between them. The weights of the edges should be set to the distances between the corresponding vertices.
2. Find a minimum spanning tree (MST): Using a minimum spanning tree algorithm such as Prim's or Kruskal's, find the MST of the graph.

3. Find vertices with odd degree: Identify the vertices in the MST that have odd degree (i.e., an odd number of edges connected to them).
4. Find a minimum-weight perfect matching: Construct a complete graph using the odd-degree vertices from step. Then, find a minimum-weight perfect matching in this complete graph using greedy matching.
5. Combine the MST and perfect matching: Combine the MST and the minimum-weight perfect matching to form a connected, non-Eulerian graph.
6. Find a Eulerian circuit: Find a Eulerian circuit in the graph from step 5, starting and ending at the same vertex.
7. Shortcut the circuit: Remove duplicate vertices from the Eulerian circuit to form a Hamiltonian circuit (i.e., a path that visits every vertex exactly once).
8. Calculate the total weight: Calculate the total weight of the Hamiltonian circuit to obtain the solution to the TSP problem instance.

Approaches used to optimize the TSP:

Tactical:-

- **2-opt :** 2-opt is a local search optimization that can be applied to improve an existing solution to the TSP. It involves iterating through pairs of edges in the path and checking if swapping them will result in a shorter path. If a shorter path is found, the swap is made, and the process is repeated until no more improvements can be made.
- **3-opt :** 3-opt is a more sophisticated local search optimization that involves considering three edges at a time and exploring different combinations of swapping them to find shorter paths. The 3-opt algorithm can be more effective than 2-opt, but it can also be more computationally expensive due to the larger number of possible swaps.

Strategic:-

- **Ant colony optimization :** Ant colony optimization is a metaheuristic optimization that is inspired by the behavior of ants in finding the shortest path between their nest and a food source. The algorithm involves simulating the behavior of ants laying pheromone trails on paths, and iteratively updating the pheromone levels to bias future ant movements towards better paths. The ant colony optimization algorithm can be effective in finding good solutions to the TSP, but it can also be computationally expensive due to the need to simulate multiple ant movements.
- **Simulated annealing :** Simulated annealing is a metaheuristic optimization that is inspired by the process of annealing in metallurgy. The algorithm involves starting with a high temperature, which allows for more exploratory

movements in the solution space, and gradually cooling the temperature, which makes the algorithm more likely to converge on a good solution. Simulated annealing can be effective in finding good solutions to the TSP, but it can also be computationally expensive due to the need to simulate many iterations and temperature changes.

Program:

Data Structures and Classes:

1. **Edge**: This class represents an edge in a graph, with fields for the source and destination vertices, and the weight of the edge. The class also includes methods for getting and setting the values of these fields.
2. **Graph**: This class represents a graph, with a map of vertices to their associated edges. The class includes methods for adding vertices and edges to the graph, getting the degree of a vertex, and obtaining the graph as a map.
3. **Vertex**: This class represents a vertex in a graph, with a unique ID and a list of adjacent edges. The class includes methods for getting and setting the ID and list of adjacent edges, as well as for getting the degree of the vertex.
4. **PrimsMST**: This class contains an implementation of Prim's algorithm for finding the minimum spanning tree of a graph. The class includes a method for executing the algorithm, which takes a graph as input and returns the minimum spanning tree as a map of vertices to their associated edges.
5. **GreedyMatching**: This class contains an implementation of a greedy algorithm for finding a minimum-weight perfect matching in a graph. The class includes a method for executing the algorithm, which takes a graph and a set of vertices as input and returns a list of edges representing the perfect matching.
6. **MultiGraph**: This class contains methods for forming a multigraph from a minimum spanning tree and a minimum-weight perfect matching. The class includes a method for executing the algorithm, which takes the minimum spanning tree, the perfect matching, and a start vertex as input, and returns a map representing the multigraph.
7. **TSPSolver**: The TSPSolver class provides a method for solving the TSP problem by generating a Eulerian tour, obtaining a Hamiltonian circuit, and selecting the optimal start vertex based on the total distance of the circuit. The class contains a solve method that takes a graph and a multigraph as input and returns a list of vertices representing the optimal tour. Additionally, it

includes a method for obtaining a Hamiltonian circuit from a Eulerian tour and a method for calculating the total distance of a circuit.

8. **TwoOpt**: This class contains an implementation of the 2-opt local search optimization for improving an existing TSP solution. The class includes a method for executing the algorithm, which takes a tour as input and returns an improved tour.
9. **ThreeOpt**: This class contains an implementation of the 3-opt local search optimization for improving an existing TSP solution. The class includes a method for executing the algorithm, which takes a tour as input and returns an improved tour.
10. **AntcolonyOptimization**: This class contains an implementation of the ant colony optimization metaheuristic for solving the TSP problem. The class includes a method for executing the algorithm, which takes a graph as input and returns a list of vertices representing the optimal tour.
11. **Simulated Annealing**: This class contains an implementation of the simulated annealing metaheuristic for solving the TSP problem. The class includes a method for executing the algorithm, which takes a graph as input and returns a list of vertices representing the optimal tour.
12. **FileIO**: This class contains methods for reading and writing TSP problem instances to and from CSV files. The class includes methods for reading a graph from a CSV file, as well as for writing a graph to a CSV file.
13. **FormGraph**: This class contains a method for forming a disconnected graph from a set of vertices. The class includes a method for executing the algorithm, which takes a set of vertices as input and returns a map representing the disconnected graph.
14. **GraphUtils**: This class contains utility methods for working with graphs, such as finding vertices with odd degree, creating a subgraph, finding a perfect matching, and calculating the total distance of a tour. The class includes methods for executing each of these tasks, as well as methods for getting the degree of a vertex.

Algorithm:

1. **Read the input data from a CSV file.**

Input: CSV file containing coordinates of points and their ID's

Output: Parsed data in the form of a connected graph

Pseudocode:

```
Read the CSV file and extract the coordinates and calculate distances of each point.
Create a vertex for each point and add it to the graph.
Create an edge between each pair of vertices with their respective edge weights.
Return the connected graph.
```

2. **Form a connected graph from the input data.**

Input: Parsed data from Step 1

Output: Connected graph

Pseudocode:

```
Create an empty graph.
For each vertex, add it to the graph.
For each edge, add it to the list of edges for its corresponding vertex in the graph.
Return the connected graph.
```

3. **Obtain a minimum spanning tree (MST) using Prim's algorithm.**

Input: Connected graph from Step 2

Output: MST

Pseudocode:

```
Choose a random vertex as the starting vertex.
Create a set of visited vertices initialized with the starting vertex.
Create a priority queue and add all edges that connect the starting vertex to other vertices.
While the priority queue is not empty:
    a. Pop the edge with the minimum weight from the priority queue
    b. If the destination vertex of the edge is not visited:
        i. Add the edge to the MST.
        ii. Add the destination vertex to the set of visited vertices.
```

```
iii. Add all edges that connect the destination vertex to other unvisited vertices to the priority queue.
Return the MST
```

4. **Identify vertices with odd degree in the MST and form a multigraph by adding perfect matchings between them.**

Input: MST from Step 3

Output: Multigraph

Pseudocode:

```
Find all vertices in the MST with an odd degree.
Pair up the odd-degree vertices
For each pair of vertices, find the shortest edge that connects them.
Add the shortest edge to the multigraph to form a perfect matching between the two vertices.
Return the multigraph.
```

5. **Solve the TSP problem using the multigraph by generating a Eulerian tour, obtaining a Hamiltonian circuit, and selecting the optimal start vertex based on the total distance of the circuit.**

Input: Multigraph from Step 4

Output: Optimal TSP tour

Pseudocode:

```
Choose a starting vertex in the multigraph.
Generate a Eulerian tour of the multigraph starting from the chosen vertex.
Obtain a Hamiltonian circuit from the Eulerian tour.
Calculate the total distance of the Hamiltonian circuit.
Choose the starting vertex that gives the smallest total distance as the optimal start vertex.
Return the Hamiltonian circuit as the optimal TSP tour.
```

6. **Use various optimization techniques, such as 2-opt, 3-opt, ant colony optimization, or simulated annealing, to improve the solution obtained in Step 5**

Input: Optimal TSP tour from Step 5

Output: Optimized TSP tour

Pseudocode:

```
Initialize the current tour as the optimal TSP tour from Step 5
Repeat until no further improvement is possible:
a. Apply the selected optimization technique to the current tour.
b. If the new tour is better than the current tour, update the current tour.
Return the final optimized TSP tour.
```

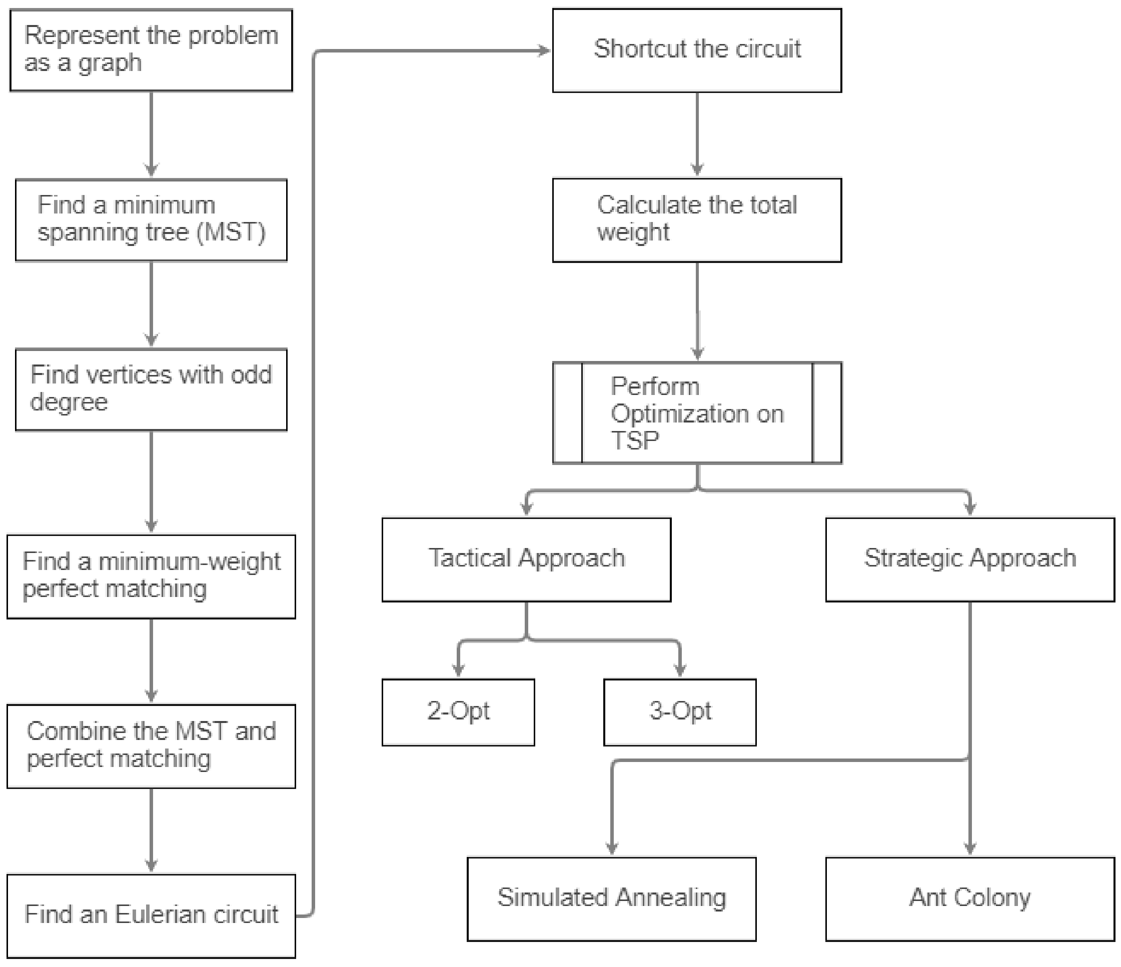
Invariants:-

- **Connected graph**: The input data is parsed to form a connected graph. This means that there is a path between any two vertices in the graph. This condition is maintained throughout the algorithm because all subsequent steps of the algorithm rely on the graph being connected. If the graph were not connected, the algorithm would not be able to find a solution to the TSP problem.
- **Minimum spanning tree**: The minimum spanning tree is obtained using Prim's algorithm, which starts at a random vertex and adds edges to the tree that have the minimum cost. This ensures that the total cost of the edges in the tree is minimized. This condition is an invariant because the MST is used to form the multigraph in the next step, and all subsequent steps of the algorithm rely on the MST being correctly formed.
- **Multigraph with even degree vertices**: The odd degree vertices in the MST are paired up and connected by adding perfect matchings to the multigraph. This ensures that all vertices in the multigraph have an even degree, which is necessary for finding a Eulerian tour. This condition is maintained throughout the algorithm because all subsequent steps of the algorithm rely on the multigraph having vertices with even degrees.
- **Eulerian tour**: A Eulerian tour is a path that traverses every edge of a graph exactly once. The multigraph obtained in step 3 is used to generate

a Eulerian tour, which is necessary for finding a Hamiltonian circuit. This condition is maintained throughout the algorithm because all subsequent steps of the algorithm rely on the Eulerian tour being correctly formed.

- **Hamiltonian circuit**: A Hamiltonian circuit is a path that visits every vertex in a graph exactly once. The Hamiltonian circuit is obtained from the Eulerian tour obtained in step 4 by removing duplicate vertices. This condition is maintained throughout the algorithm because the total distance of the Hamiltonian circuit is used to find the optimal start vertex.

Algorithm Flowchart: -


Represent the problem as a graph
Find a minimum spanning tree (MST)
Find vertices with odd degree
Find a minimum-weight perfect matching
Combine the MST and perfect matching
Find an Eulerian circuit
Shortcut the circuit
Calculate the total weight
Perform Optimization on TSP
Tactical Approach
Strategic Approach
2-Opt
3-Opt
Simulated Annealing
Ant Colony

Observations and Graphical Analysis: -

We have recorded the observations we made at different stages of our implementation, which are listed below:

1)**MST**: Our initial implementation of the first step in our algorithm involved using

Prim's Algorithm, where we had to iterate through all edges to identify the next minimum edge. However, this approach proved to be time-consuming. Therefore, we modified our implementation by utilizing a priority queue, which helped reduce our time complexity from linear (n) to logarithmic (log n). This change significantly improved our ability to find a minimum spanning tree efficiently.

2)**Minimum Perfect Matching**: Once we had found the MST, we identified the odd degree vertices that were part of it. Our first approach to finding the minimum perfect matching edges was to use Blossom's algorithm, but we found it to be quite complex. As an alternative, we adopted a greedy matching approach where we formed all possible edges between the odd vertices and sorted them based on weight. We then selected the minimum weighted edges with unique vertices in each iteration.

3)**Eulerian path**: Our algorithm involves traversing all edges and vertices in a depth-first search using a stack data structure. We iteratively perform this search until we have visited every edge and vertex, and our output is a list of the visited vertices.

4)**Hamiltonian Circuit**: We have eliminated any duplicate vertices from the list of vertices obtained from the Eulerian path to form a Hamiltonian circuit. To ensure that we obtain a TSP tour with the minimum possible weight, it is important to execute the Eulerian path finding and Hamilton circuit finding algorithms from different starting points. This approach allows us to explore all the possible Hamilton circuits present in the graph and identify the one with the lowest weight.

5)**Two Opt Optimization:** To optimize the Two-opt technique, instead of continuously swapping and reversing the list of vertices, a delta function is used to calculate the difference between the lengths of the old and new edges after swapping. This avoids the need to check the length of each tour in every iteration, which can be time-consuming. The delta function allows for faster convergence to an optimal solution and identifies which swaps result in shorter tours.

The formula to calculate the length difference between the current and new edges after swapping is as follows:

lengthDelta = -dist(route[v1], route[v1+1]) - dist(route[v2], route[v2+1]) + dist(route[v1+1], route[v2+1]) + dist(route[v1], route[v2])

Here, (v1,v1+1) and (v2,v2+1) are the edges being swapped, and (v1,v2) and (v1+1,v2+1) are the new edges that are formed after the swap. The formula calculates the length of the new edges and subtracts the length of the edges being removed to obtain the difference in length between the current tour and the new tour. If the resulting value is negative, the swap is accepted, and the search continues for an optimal solution. If it is positive or zero, the swap is rejected, and the algorithm moves on to the next iteration.

This has significantly reduced our processing time. The processing time has come down from minutes to a few seconds.

6)**Three Opt optimization:** To overcome the high processing time associated with the Three Opt optimization technique due to its O(n^3) time complexity, a delta function is used to compute and compare tour values instead of directly swapping three edges. This eliminates expensive operations and enables efficient comparison between the old and new edges. By using a delta function, Three Opt optimization can be performed more efficiently for large datasets.

The formula used in the Three Opt optimization technique to calculate the difference in length between the current and new tours after swapping edges is more complex than that of the Two-opt technique. It involves calculating the length of multiple new edges that are formed after swapping three edges in the route. The formula is as follows:

lengthDelta = -dist(route[i], route[i+1]) - dist(route[j], route[j+1]) - dist(route[k], route[k+1]) + dist(route[i], route[j]) + dist(route[i+1], route[k]) + dist(route[j+1], route[k+1])

Here, i, j, and k are the indices of the edges being swapped, and the resulting lengthDelta is the difference in length between the current tour and the new tour. The formula subtracts the length of the edges being removed (route[i] to route[i+1], route[j] to route[j+1], and route[k] to route[k+1]) from the total length of the edges being added (route[i] to route[j], route[i+1] to route[k], and route[j+1] to route[k+1]).

If lengthDelta is negative, the new tour is shorter than the current tour, and the swap is accepted. If it is positive or zero, the swap is rejected, and the search continues for a better solution.

7)**Simulated Annealing:** Simulated Annealing is a TSP optimization algorithm that uses an optimal temperature and cooling factor to regulate the optimization process. The algorithm gradually reduces the temperature towards a value less than one, allowing it to accept tours with costs less than the current tour. Initially, random swapping was used to optimize the tour, but the Two Opt optimization technique was found to be more effective. Two Opt optimization involves swapping two edges to obtain a better tour.

The optimal values of temperature and cooling factor our implementation is mentioned below.

Temperature=1000 & Cooling factor=0.001

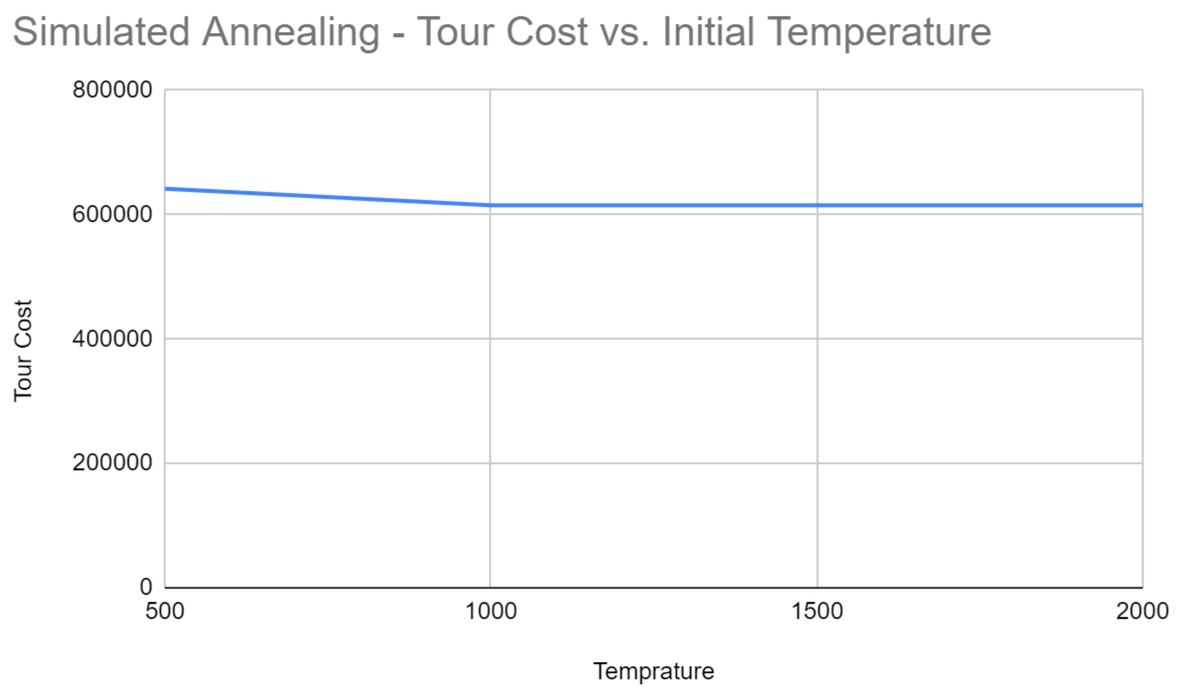
The formula for determining the new temperature in the Simulated Annealing algorithm is:

new_temperature = current_temperature * cooling_factor

The cooling factor affects the probability of accepting new solutions with higher energy and is based on the difference in energy and the current temperature.

probability = (currentDistance - newDistance) / temperature

The equation presented is a simplified version of the formula used in simulated annealing, a probabilistic approach for discovering the global minimum of a function. This equation computes the probability of accepting a new solution by evaluating the difference between the objective function value and the current "temperature" of the system.



Our experimentation involved setting the initial temperature for the simulated annealing algorithm to a range of 500 to 2000, incrementing in intervals.

According to the results obtained, it appears that the simulated annealing algorithm yields better optimized values as the initial temperature is increased. However, beyond a certain temperature, no further improvement in the tour cost is observed and the value remains constant, leading to a cessation of improvement.

8)**Ant Colony:** Initially, we ran a set of ants through an iterations loop, updating the pheromone matrix after the ant loops completed, but this process was found to be inefficient and time-consuming, with no significant results. In response, we adjusted our approach by updating the pheromone matrix after each ant travels in the loop, resulting in improved efficiency.

We noticed that the algorithm was taking a significant amount of time to complete, but it was still generating results without getting stuck. However, we observed that the algorithm was not able to efficiently move towards optimal solutions, resulting in expensive tours that did not achieve satisfactory optimization.

We also conducted experiments with several crucial factors in Ant Colony Optimization, including the number of ants, alpha (importance of pheromone), beta (importance of quality), rho (evaporation factor), and the number of iterations necessary for optimal outcomes. Despite altering and testing various parameter values, we were unable to achieve an optimized result. Our final parameter values were as follows:

$\alpha = 1$, $\beta = 2$, $\rho = 0.5$, no. of ants=50, no. of iterations=1000

The following are some of the mathematical derivations and formulae used in ACO:

Pheromone trail update formula: The pheromone trail update formula is used to update the pheromone trail after an ant completes a tour. It is given by:

$$\tau_{ij} = (1 - \rho)\, \tau_{ij} + \sum(\Delta\tau_{ij}^k)$$

Where **τ_{ij}** is the pheromone level of edge (i,j), ρ is the pheromone evaporation coefficient, and **$\Delta\tau_{ij}^k$** is the amount of pheromone deposited by ant k on edge (i,j).

Ant probability formula: The probability that an ant k chooses to move from city i to city j is given by:

$$p_{ij}^k = \{\tau_{ij}^\alpha * \eta_{ij}^\beta\} / \sum((\tau_{il}^\alpha * \eta_{il}^\beta))$$

Where τ_{ij} is the pheromone level of edge (i,j), η_{ij} is the heuristic value of edge (i,j), α is the pheromone exponent parameter, and β is the heuristic exponent parameter.

Heuristic information formula: The heuristic information formula is used to calculate the heuristic value of an edge (i,j). It is given by:

$$\eta_{ij} = 1 / d_{ij}$$

Where d_{ij} is the distance between city i and city j.

Pheromone evaporation formula: The pheromone evaporation formula is used to calculate the pheromone level after evaporation. It is given by:

$$\tau_{ij} = (1 - \rho)\, \tau_{ij}$$

Where τ_ij is the pheromone level of edge (i,j), and ρ is the pheromone evaporation coefficient.

Ant movement formula: The ant movement formula is used to move an ant from city i to city j. It is given by:

L_k = ∑(d_ij)

Where L_k is the length of the tour taken by ant k, and d_ij is the distance between city i and city j.

These formulas are some of the key elements of ACO that are used to update the pheromone trail, calculate probabilities, and move the ants.

Results and Mathematical Analysis: -

The first table shows the results of applying algorithms to solve two graph problems - the Minimum Spanning Tree (MST) and the Traveling Salesman Problem (TSP). The total cost column shows the sum of the weights of the edges in the MST and TSP, while the algorithm column indicates the specific algorithm used to generate the MST and the TSP.

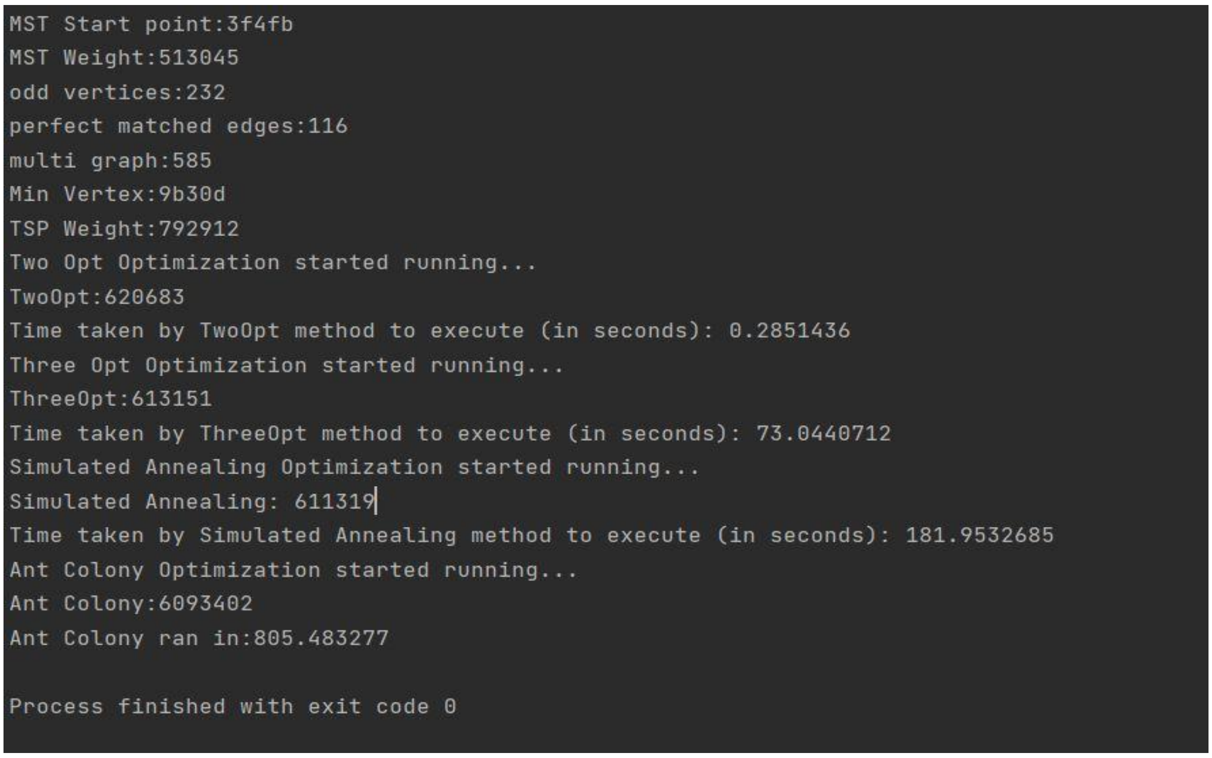
Graph Problems	Total Cost	Algorithm
Minimum Spanning Tree	513,045	Prim's
TSP	792,912	Christofides

In this table, we compare the performance of four optimization methods - TwoOpt, ThreeOpt, Simulated Annealing, and Ant Colony - in terms of their effect on total cost and execution time. The total cost column shows the total cost of the optimized solution, while the execution time column shows the time taken by the optimization method to find the solution. The type of optimization column indicates whether the method is tactical or strategic in nature.

Tabulation of results:

Optimization Method	Total Cost	Execution Time (seconds)	Type of Optimization
TwoOpt	620,683	0.2851436	Tactical
ThreeOpt	613,151	73.0448712	Tactical
Simulated Annealing	611,319	181.9532685	Strategic
Ant Colony	6,093,402	805.483277	Strategic

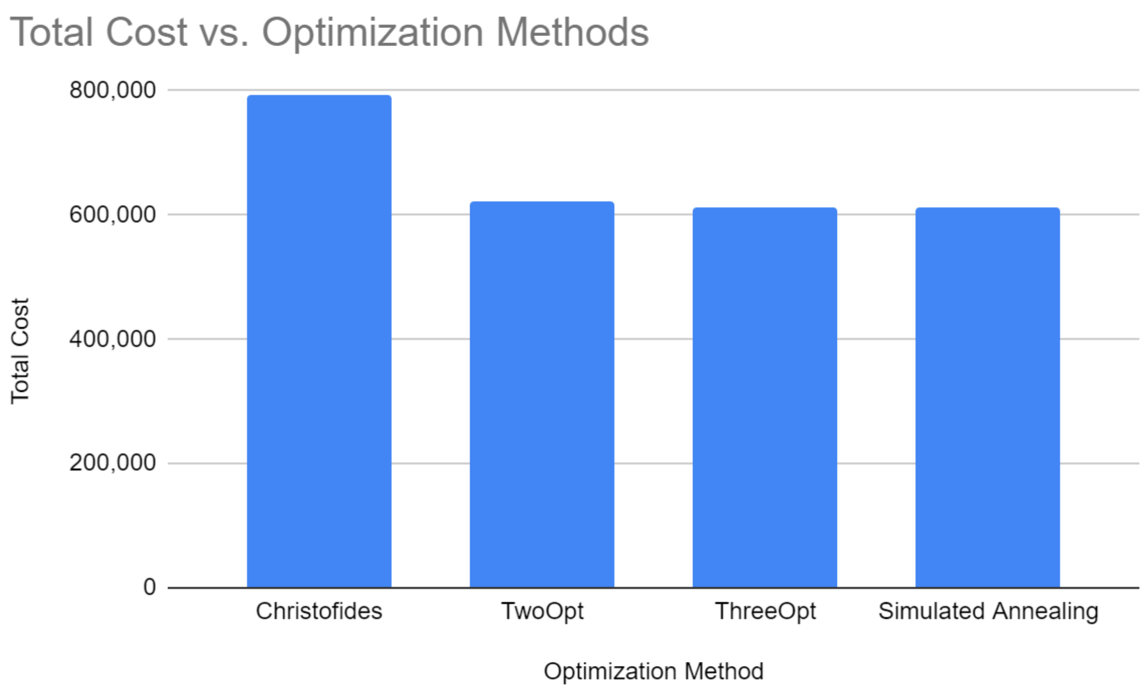
Results printed on Console:



```
MST Start point:3f4fb
MST Weight:513045
odd vertices:232
perfect matched edges:116
multi graph:585
Min Vertex:9b30d
TSP Weight:792912
Two Opt Optimization started running...
TwoOpt:620683
Time taken by TwoOpt method to execute (in seconds): 0.2851436
Three Opt Optimization started running...
ThreeOpt:613151
Time taken by ThreeOpt method to execute (in seconds): 73.0440712
Simulated Annealing Optimization started running...
Simulated Annealing: 611319
Time taken by Simulated Annealing method to execute (in seconds): 181.9532685
Ant Colony Optimization started running...
Ant Colony:6093402
Ant Colony ran in:805.483277

Process finished with exit code 0
```

Graphical comparison of Tour Costs:



Let’s calculate the ratios for each optimization method with respect to the MST weight and then find the percentage increase.

Ratios:

TwoOpt Ratio = TwoOpt Weight / MST Weight = 620,683 / 513,045 = **1.20**
ThreeOpt Ratio = ThreeOpt Weight / MST Weight = 613,151 / 513,045 = **1.19**
Simulated Annealing Ratio = Simulated Annealing Weight / MST Weight = 611,319 / 513,045 = **1.19**
Ant Colony Ratio = Ant Colony Weight / MST Weight = 6,093,402 / 513,045 = **11.87**

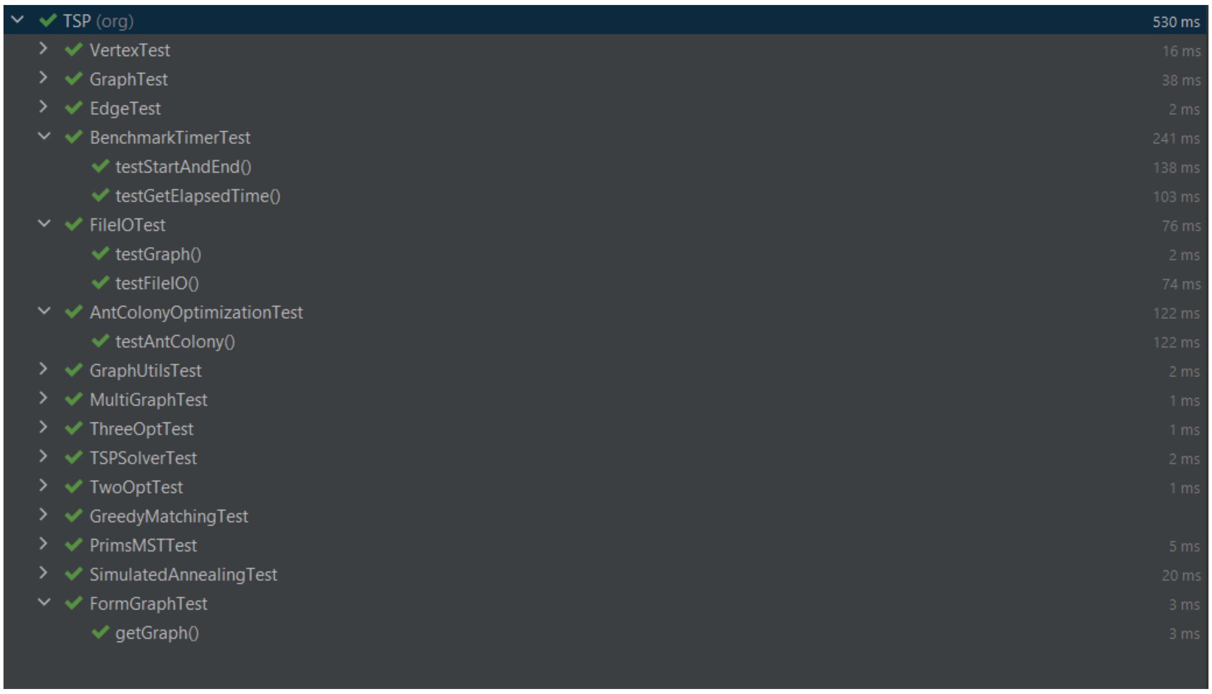
Percentage Increase (w.r.t MST) :

TwoOpt % Increase = (TwoOpt Ratio - 1) * 100 = **20.97%**
ThreeOpt % Increase = (ThreeOpt Ratio - 1) * 100 = **19.51%**
Simulated Annealing % Increase = (Simulated Annealing Ratio - 1) * 100 = **19.17%**
Ant Colony % Increase = (Ant Colony Ratio - 1) * 100 = **1,087.92%**

Optimization Method	**Ratio (wrt MST)**	**% increase (wrt MST)**	**Type of Optimization**	**Characteristics**
TwoOpt	1.20	20.97%	Tactical	Local search; pairwise exchanges of edges; less computationally intensive
ThreeOpt	1.19	19.51%	Tactical	Local search; triple wise exchanges of edges; more complex than TwoOpt
Simulated Annealing	1.19	19.17%	Strategic	Global search; probabilistic acceptance of non-improving solutions; uses temperature parameter
Ant Colony	11.87	1,087.92%	Strategic	Global search; inspired by ant behavior; pheromone-based path updates

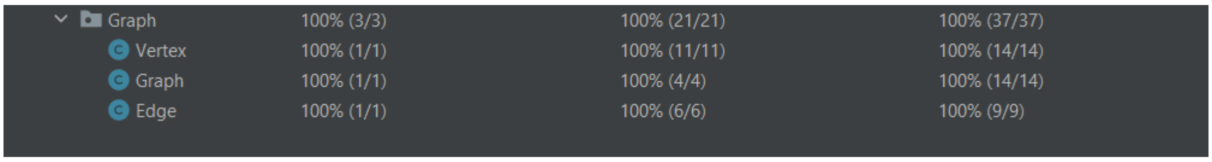
Unit Tests:-

All unit tests have passed -



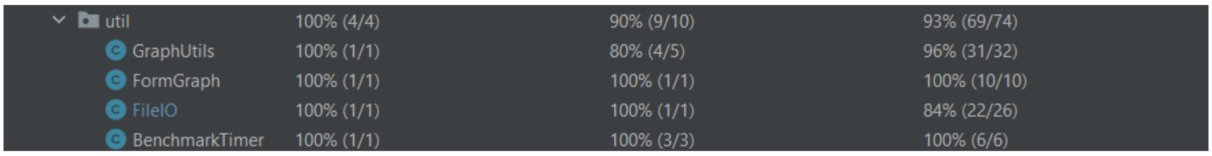
```
✓ TSP (org)                                   530 ms
  ✓ VertexTest                                 16 ms
  ✓ GraphTest                                  38 ms
  ✓ EdgeTest                                    2 ms
  ✓ BenchmarkTimerTest                        241 ms
    ✓ testStartAndEnd()                       138 ms
    ✓ testGetElapsedTime()                    103 ms
  ✓ FileIOTest                                 76 ms
    ✓ testGraph()                               2 ms
    ✓ testFileIO()                             74 ms
  ✓ AntColonyOptimizationTest                 122 ms
    ✓ testAntColony()                         122 ms
  ✓ GraphUtilsTest                              2 ms
  ✓ MultiGraphTest                              1 ms
  ✓ ThreeOptTest                                1 ms
  ✓ TSPSolverTest                               2 ms
  ✓ TwoOptTest                                  1 ms
  ✓ GreedyMatchingTest
  ✓ PrimsMSTTest                                5 ms
  ✓ SimulatedAnnealingTest                     20 ms
  ✓ FormGraphTest                               3 ms
    ✓ getGraph()                                3 ms
```

Unit tests coverage for org.TSP.Graph –



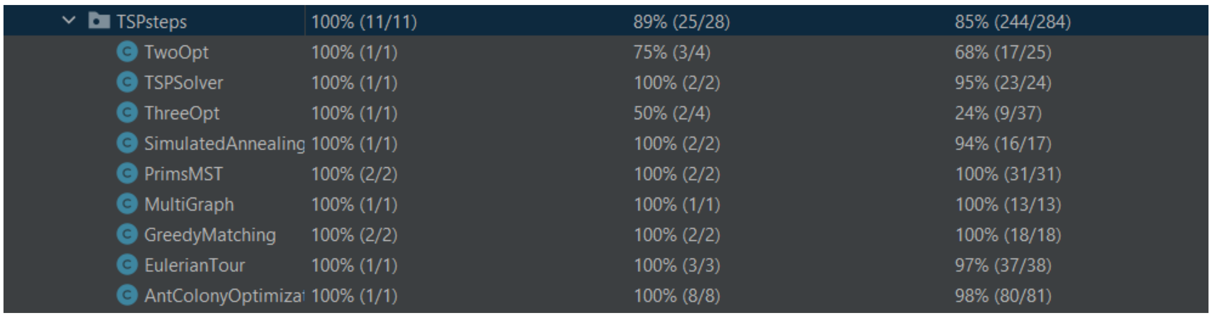
```
Graph          100% (3/3)     100% (21/21)     100% (37/37)
  Vertex       100% (1/1)     100% (11/11)     100% (14/14)
  Graph        100% (1/1)     100% (4/4)       100% (14/14)
  Edge         100% (1/1)     100% (6/6)       100% (9/9)
```

Unit tests coverage for org.TSP.Util –



```
util              100% (4/4)     90% (9/10)      93% (69/74)
  GraphUtils      100% (1/1)     80% (4/5)       96% (31/32)
  FormGraph       100% (1/1)     100% (1/1)      100% (10/10)
  FileIO          100% (1/1)     100% (1/1)      84% (22/26)
  BenchmarkTimer  100% (1/1)     100% (3/3)      100% (6/6)
```

Unit tests coverage for org.TSP.TSPSteps –



TSPsteps	100% (11/11)	89% (25/28)	85% (244/284)
TwoOpt	100% (1/1)	75% (3/4)	68% (17/25)
TSPSolver	100% (1/1)	100% (2/2)	95% (23/24)
ThreeOpt	100% (1/1)	50% (2/4)	24% (9/37)
SimulatedAnnealing	100% (1/1)	100% (2/2)	94% (16/17)
PrimsMST	100% (2/2)	100% (2/2)	100% (31/31)
MultiGraph	100% (1/1)	100% (1/1)	100% (13/13)
GreedyMatching	100% (2/2)	100% (2/2)	100% (18/18)
EulerianTour	100% (1/1)	100% (3/3)	97% (37/38)
AntColonyOptimizat	100% (1/1)	100% (8/8)	98% (80/81)

All classes in the project have been tested in isolation using unit tests, resulting in an overall project coverage of 94% for classes, 91% for methods, and 80% for lines. This means that each component of the system was tested individually, without any external dependencies or interactions. Unit tests were written for all classes except for the main method, which is typically not unit tested.

Conclusion:-

Based on our analysis of various algorithms and optimization techniques for the Traveling Salesman Problem (TSP), we have concluded that the use of optimization techniques such as TwoOpt and ThreeOpt can significantly reduce the total cost of TSP compared to the standard TSP solution. Specifically, TwoOpt technique resulted in a significant decrease in the total cost compared to the standard TSP solution, with a time complexity of O(n^2) and a relatively shorter execution time. Meanwhile, the ThreeOpt method, with a higher time complexity of O(n^3), also led to a significant decrease in the total cost compared to the standard TSP solution but required a longer execution time. In summary, the use of optimization techniques such as TwoOpt and ThreeOpt can offer significant improvements in the cost optimization of TSP, with TwoOpt being more efficient in terms of execution time while ThreeOpt also offers great optimization results at the cost of longer execution times.

For strategic optimization techniques, Simulated Annealing showed better performance in cost reduction than Ant Colony Optimization. The choice of optimization technique should be based on factors such as solution quality, problem size, and available computational resources. Tactical methods like TwoOpt or ThreeOpt are suitable for smaller problem instances or limited resources, while strategic methods like Simulated Annealing can be considered for larger instances or more thorough exploration of the solution space.

References:-

Traveling Salesman Problem. In Wikipedia. https://en.wikipedia.org/wiki/Traveling_salesman_problem

Graph Data Structure and Algorithms. In GeeksforGeeks. https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/

Prim's Minimum Spanning Tree Algorithm. In GeeksforGeeks. https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/

Eulerian Path and Circuit. In GeeksforGeeks. https://www.geeksforgeeks.org/eulerian-path-and-circuit/

Two-Opt Algorithm. In Wikipedia. https://en.wikipedia.org/wiki/2-opt

Three-Opt Algorithm. In Wikipedia. https://en.wikipedia.org/wiki/3-opt

Ant Colony Optimization Algorithm. In Baeldung. https://www.baeldung.com/java-ant-colony-optimization

Simulated Annealing Algorithm. Medium Article. https://medium.com/ai-techsystems/simulated-annealing-580f73bd807a