INFO 6205 - Program Structure and Algorithms

Spring 2023 Project : Traveling Salesman using Christofides Algorithm

Team 34:

1. Neha Rastogi (NUid : 002709191)
2. Harshini Venkata Chalam (NUid : 002934047)

GitHub Repository Link - https://github.com/rastogi-neha/INFO6205-TSP

1. INTRODUCTION

The Travelling Salesman Problem (TSP) is a classic optimization problem that aims to find the shortest possible route that a salesman can take to visit a set of cities and return to the starting point. There are several algorithms to solve the TSP, and one of the most popular ones is the Christofides Algorithm.

The problem was first introduced by William Rowan Hamilton in 1835, and it has since then been a topic of active research in various fields. Because it is extremely complex, there is no known exact algorithm that can solve it quickly. Instead, heuristic and approximation algorithms have been created, such as the Christofides algorithm, which produces solutions that are not optimal but still useful within a reasonable time when compared to the brute force approach that involves examining all routes making it infeasible for even moderately sized instances of the problem. The Christofides algorithm guarantees that the solution is within a factor of 3/2 of the optimal solution and thereby is particularly useful.

The TSP is used in a variety of real-world situations, including optimising delivery routes, drilling circuit boards, and scheduling issues in computer science. This project aims to implement the TSP using the Christofides algorithm and optimise the cost using several optimization techniques.

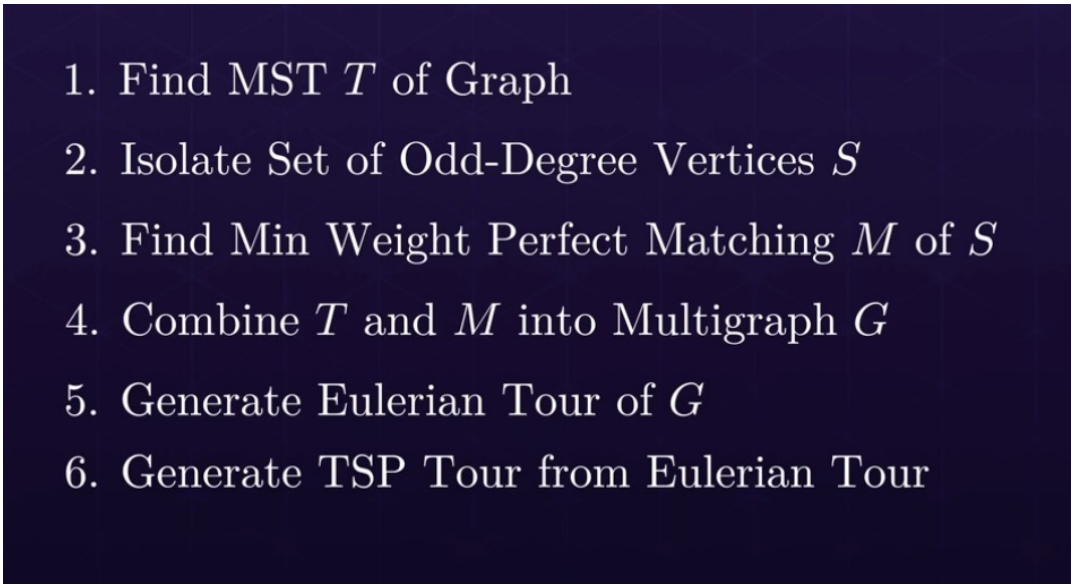
Aim

The project involves implementing TSP using Christofides algorithm and then optimising the cost using the tactical and strategic optimization schemes 2-opt, 3-opt, simulated annealing, and ant colony optimization. To evaluate the effectiveness of each optimization method, unit test cases are executed, and the cost is mathematically analysed for several scenarios. These

tests have provided insights into each optimization method's effectiveness in reducing the cost of the TSP.

Approach

The approach to implement Christofides algorithm,



- The program reads the input data from a CSV file that contains the coordinates of different cities. The Haversine distance is then used to calculate the distance matrix between every pair of cities.
- A graph is initialised with the number of vertices and edges. Then it finds n Eulerian cycle using the Eulerian cycle algorithm. A Eulerian cycle is a cycle that visits every edge exactly once.
- After finding the Eulerian cycle, the code removes duplicate vertices from the cycle and creates a new graph with only the odd-degree vertices from the original graph. It then finds a minimum spanning tree (MST) for this new graph using the Prims's algorithm.
- Then we find the minimum weight perfect matching between pairs of odd-degree vertices in the MST using the nearest neighbour algorithm. The minimum weight perfect matching connects every pair of odd-degree vertices with the minimum possible distance.
- Finally, the code merges the MST and the minimum weight perfect matching to create a Hamiltonian cycle that visits every vertex exactly once.

- The TSP tour along with the total cost is printed as output. This method provides a suboptimal solution to the TSP problem, but it is computationally efficient and provides a reasonable approximation to the optimal solution. The effectiveness of this method is further improved by tuning the optimization methods and adjusting the input parameters based on the specific TSP problem being solved.
- We have employed several optimization methods, such as 2-Opt, 3-Opt, Simulated Annealing, and Ant Colony Optimization, to refine the outcome. These optimization methods aim to improve the solution by iteratively modifying the tour to reduce its total distance travelled.

The approach to implement 2-opt,

- The 2-opt algorithm used is a local search heuristic that aims to improve the TSP initial tour by iteratively swapping pairs of edges in the tour to reduce the total cost of the tour. If the new tour is shorter, then the tour is updated by reversing the order of edges. When no further improvements can be made, and the final tour cost is returned.

The approach to implement 3-opt,

- The 3-opt algorithm is an iterative optimization method that starts with an initial tour and iteratively attempts to improve it by exchanging three edges at a time. The algorithm does this by selecting three edges and checking if swapping them results in a shorter tour. If it does, then the edges are swapped, and the process continues until no further improvement is possible.

The approach to implement Simulated Annealing,

- Simulated annealing is a probabilistic technique for optimization. The basic idea is to start with the TSP generated through Christophides and then iteratively make small modifications to it, gradually changing the temperature and cooling factor, which controls the likelihood of accepting worse solutions as the algorithm progresses. The algorithm terminates when the temperature reaches a predefined threshold. If the cost of the new solution is worse than the current solution, we accept the new solution with a probability that depends on the temperature and the magnitude of the cost increase.
- The approach to implement Ant Colony Optimization,

- The approach involves constructing a graph where the nodes represent the cities to be visited, and the edges represent the distances between them. A colony of ants is then allowed to explore the graph by constructing a tour of the cities. During the exploration, the ants lay pheromone trails on the edges they traverse. The amount of pheromone deposited is proportional to the quality of the solution found.
- The pheromone trails evaporate over time, and the ants use a probabilistic rule to decide which city to visit next based on the amount of pheromone deposited on each edge and the distance to the next city. The ants bias their choices towards edges with higher pheromone levels, which encourages exploration of the graph while still exploiting good solutions.
- The process is repeated over several iterations until a satisfactory solution is found or the algorithm reaches the maximum number of iterations allowed. At each iteration, the pheromone levels on the edges are updated based on the quality of the solutions found. The best solution found over all iterations is returned as the output of the algorithm.

2. PROGRAM

Data Structures & Classes

1. `FileReading` class: The class contains a single static method readingDataPoints which takes a String parameter file representing the file path and returns a List of String arrays. Each String array contains the values for one row of the CSV file. The method uses a FileReader to read the file, then passes the FileReader to a CSVReaderBuilder to create a CSVReader object. The withSkipLines method is called on the CSVReaderBuilder to skip the first line (header row) of the file. The resulting CSVReader object is used to call the readAll method which returns a List of String arrays representing all rows in the file, excluding the header row.

2. `TravellerData` class: The ‘TravellerData’ class is used to generate a list of cities that will be traversed by the salesman in the Travelling Salesman Problem (TSP). The class contains several methods for managing the list of cities and calculating distances between them. City class has fields for the city name, latitude, longitude, and an index representing the city's position in the list. The class has a single private field cities which is an ArrayList of City

objects. This field is used to store the list of cities that will be traversed. The ‘addCity’ method is used to add a single City object to the cities list. The ‘getCity’ method is used to retrieve a City object from the cities list by index. The putCitiesInList method is used to populate the cities list from a List of String arrays that is returned by the ‘readingDataPoints’ method in the ‘FileReading’ class. Each String array represents a row of data from the CSV file, and the method creates a new City object for each row and adds it to the cities list. The ‘getNumberOfCities’ method returns the number of cities in the cities list. The ‘getDistance’ method calculates the distance between two City objects using the Haversine formula, which is an approximation of the distance between two points on a sphere (in this case, the Earth's surface). The method takes two City objects as arguments and returns a double value representing the distance between them in metres.

3. `City` class: The City class represents a city in the Travelling Salesman Problem (TSP). The class has four fields, cityName, x, y, and index, which represent the name of the city, its x and y coordinates, and its position in the list of cities.The class provides getters and setters for each field to allow access to and modification of the city's attributes.The class has a constructor that takes four arguments: the city name, x and y coordinates, and the index representing the city's position in the list. This method returns a string containing the city's name, x and y coordinates, and index, formatted as a JSON object.The City class is used by the TravellerData class to create a list of cities that will be traversed by the salesman. Each City object represents a single city in the list.

4. `Edge` class: The Edge class represents an edge in the Travelling Salesman Problem (TSP). The class has three private fields, src, dest, and edgeWeight, which represent the source vertex of the edge, the destination vertex of the edge, and the weight of the edge, respectively. The class implements the Comparable interface and overrides the compareTo() method to allow for sorting of Edge objects based on their weight. The compareTo() method compares the weight of the current edge with the weight of the edge passed as an argument, and returns a negative, zero, or positive integer if the weight of the current edge is less than, equal to, or greater than the weight of the argument edge, respectively.
The Edge class is used in the Graph class to represent the edges of the TSP graph. The edges connect vertices that represent the cities in the TSP problem, and the weight of each edge is calculated as the distance between the two cities that the edge connects.

5. `Graph` class: It uses various data structures including arrays, linked lists, and array list to represent and manipulate the graph. This class represents a graph and has the following properties: number of vertices, number of edges, and a list of edges. It also has several methods for performing operations on graphs, such as finding the minimum spanning tree using Prim's algorithm, finding, and adding perfect matches, and finding the Eulerian cycle.

- v: number of vertices in the graph.
- e: number of edges in the graph.
- adjList: an adjacency list representation of the graph, where each element in the array is a linked list containing the vertices adjacent to the corresponding vertex in the graph.
- eulerianCircuit (array list): a list to store the vertices in the order of an Eulerian circuit, which is a path that traverses every edge in the graph exactly once.

The class has the following methods:

- Graph(int v, int e) (constructor): initialises the number of vertices and edges in the graph and creates an empty adjacency list.
- Graph(int numOfVertices) (constructor): initialises the number of vertices in the graph and creates an empty adjacency list.
- addEdge(Integer u, Integer v): adds an undirected edge between vertices u and v to the graph by adding v to the adjacency list of u and vice versa.
- removeEdge(Integer u, Integer v): removes an undirected edge between vertices u and v from the graph by removing v from the adjacency list of u and vice versa.
- isValidNextEdge(Integer u, Integer v): checks if adding an edge between vertices u and v would create a new Eulerian circuit in the graph. This is done by performing a depth-first search (DFS) from u before and after the edge is added and comparing the number of vertices visited in both searches.
- dfsCount(Integer s, boolean[] isVisited): performs a DFS from vertex s and returns the number of vertices visited.
- eulerianCycle(): finds an Eulerian circuit in the graph by starting at a vertex with an odd degree (if one exists) and recursively adding edges to the circuit using isValidNextEdge() until all edges have been traversed. The resulting circuit is stored in eulerianCircuit.
- findAndAddPerfectMatches(Edge[] mst, List<City> cities): finds a set of edges that, when added to the minimum spanning tree mst, creates a graph where all vertices have an even degree. This is done by finding all vertices with an odd

degree, finding the nearest neighbour for each of these vertices, and creating edges between each pair of nearest neighbours. The resulting edges are then added to mst to create a new set of edges that satisfy the even degree property. The resulting set of edges is returned.

6. `TwoOpt` class: The TwoOpt class implements the 2-opt algorithm for solving the travelling salesman problem (TSP). It has two methods: do2Opt and twoOptimization.
The do2Opt method takes an ArrayList of City objects tspTour and two indices i and j, and performs a 2-opt swap on the tour segment between the cities at indices i and j. It uses the Collections.reverse method to reverse the sublist of cities between indices i+1 and j+1 in the tspTour list.
The twoOptimization method takes an ArrayList of City objects tspTour and the current tour distance currentDistance, and performs the 2-opt algorithm on the tspTour to improve the tour. It uses a TravellerData object td to calculate the distances between cities. It loops through all pairs of cities in the tspTour and checks if a 2-opt swap between them would result in a shorter tour length. If so, it calls the do2Opt method to perform the swap and updates the tour distance. It continues looping until no further improvements can be made.

7. `ThreeOpt` class: The data structure used in this code is an ArrayList of City objects, where a City object represents a point on the TSP tour.
The ‘reverseSegmentIfBetter’ method takes in the ArrayList representing the TSP tour and three indices i, j, and k, and performs a 3-opt move on the tour by reversing a segment of cities if it results in a shorter tour distance. The method returns the change in distance resulting from the move. The ‘threeOptimization’ method takes in the ArrayList representing the TSP tour and repeatedly applies the ‘reverseSegmentIfBetter’ method to all possible segments of the tour until no further improvements are possible. The method returns the distance of the optimised TSP tour.The allSegments method generates all possible segments of the TSP tour using nested loops and returns an ArrayList of integer arrays representing the start, middle, and end indices of each segment.
The calculate ‘ThreeOptDistance’ method takes in the ArrayList representing the TSP tour and calculates the total distance of the tour by summing the distances between each adjacent pair of cities in the tour and adding the distance between the last and first cities in the tour.

8. `SimulatedAnnealing` class: The simulatedAnnealing() method takes an ArrayList of City objects (representing a tour) and returns the best tour found by the algorithm. The algorithm is implemented using a loop that performs iterations until the temperature is cooled down to a specified minimum value. In each iteration, the algorithm generates a new solution by randomly swapping two cities in the current tour. The algorithm then decides whether to accept the new solution or not based on a probability that depends on the cost (distance) of the new solution and the current temperature. The algorithm also updates the best solution found so far and cools down the temperature.The calculateTotalDistanceAnnealing() method takes an ArrayList of City objects (representing a tour) and returns the total distance of the tour.The acceptanceProbability() method takes the current cost, the new cost, and the current temperature and returns a probability that depends on the difference between the current cost and the new cost and the current temperature.

9. `AntColony` class: This class implements the ant colony optimization algorithm for the Travelling Salesman Problem. It contains various parameters for the algorithm, such as the number of ants, the alpha and beta parameters (which control the relative importance of the pheromone trail and the distance between cities), and the evaporation rate of the pheromone trail. The run method of the AntColony class is the main loop of the algorithm. In each iteration, the algorithm constructs a solution using a set of artificial ants, updates the pheromone trail based on the solutions found, and performs pheromone trail evaporation.

10. `Main` class: This is the main class that brings all the other classes together to solve the Travelling Salesman Problem using Christofides Algorithm and then optimising the solution using various optimization algorithms. It also has several helper methods for post-processing the result set and displaying the solution.

Algorithm

We broke down the problem into smaller subproblems and started dealing with them one by one.

Step 1 : Take input from the CSV files using OpenCSV API and check for proper filepath and reading it in according to the datatypes necessary.

Step 2 : Created a class called City which stores the latitude, longitude, crimeID and adds an additional column called index which helps in using the cities as nodes for the Graph.

Step 3 : Constructed a 2D matrix to store the distances between cities which will serve as the cost for choosing Edges between Nodes.

Step 4 : Now that we have our prerequisites, we initialise a graph and a corresponding adjacency matrix to store chosen edges.

Step 5 : A Minimum Spanning Tree is required to determine the lower bound of an optimal TSP which is constructed in our case using Prim's algorithm since it is more suited for a denser graph with multiple edges.

Step 6 : The MST may have multiple edges from the graph since its purpose is to get a connected graph therefore we need to determine and isolate vertices having odd degrees since they will be used for constructing a TSP tour.

Step 7 : We find the Perfect matching edges based on the Greedy approach of Nearest Neighbours thus obtaining a secondary graph.

Step 8 : Combine the MST and the secondary graph to create a Multigraph which is utilised for further processing.

Step 9 : The Multigraph is used to construct a Eulerian tour, a path which visits each edge but might have repeating nodes in it.

Step 10 : Next we remove the repeating nodes and convert the Eulerian tour to a Hamiltonian circuit. The current tour starts and ends at the same node and visits each city once in the minimum possible cost.

Step 11 : The tour obtained needs to undergo optimizations to better the current cost that we are obtaining. We implemented Tactical Optimizations namely 2 Opt which swaps links between 2 nodes at a time and if a lower cost is obtained then chooses that.

Step 12 : 3 Opt follows a similar operation but instead of considering 2 nodes, it considers 3 nodes and tries to find a better set of edges which result in a lower cost.

Step 13 : For Strategic Optimization, Simulated Annealing randomly swaps links between 2 cities to find an optimum path however, this was performing poorly so we switched out the swapping algorithm to 2 Opt instead.

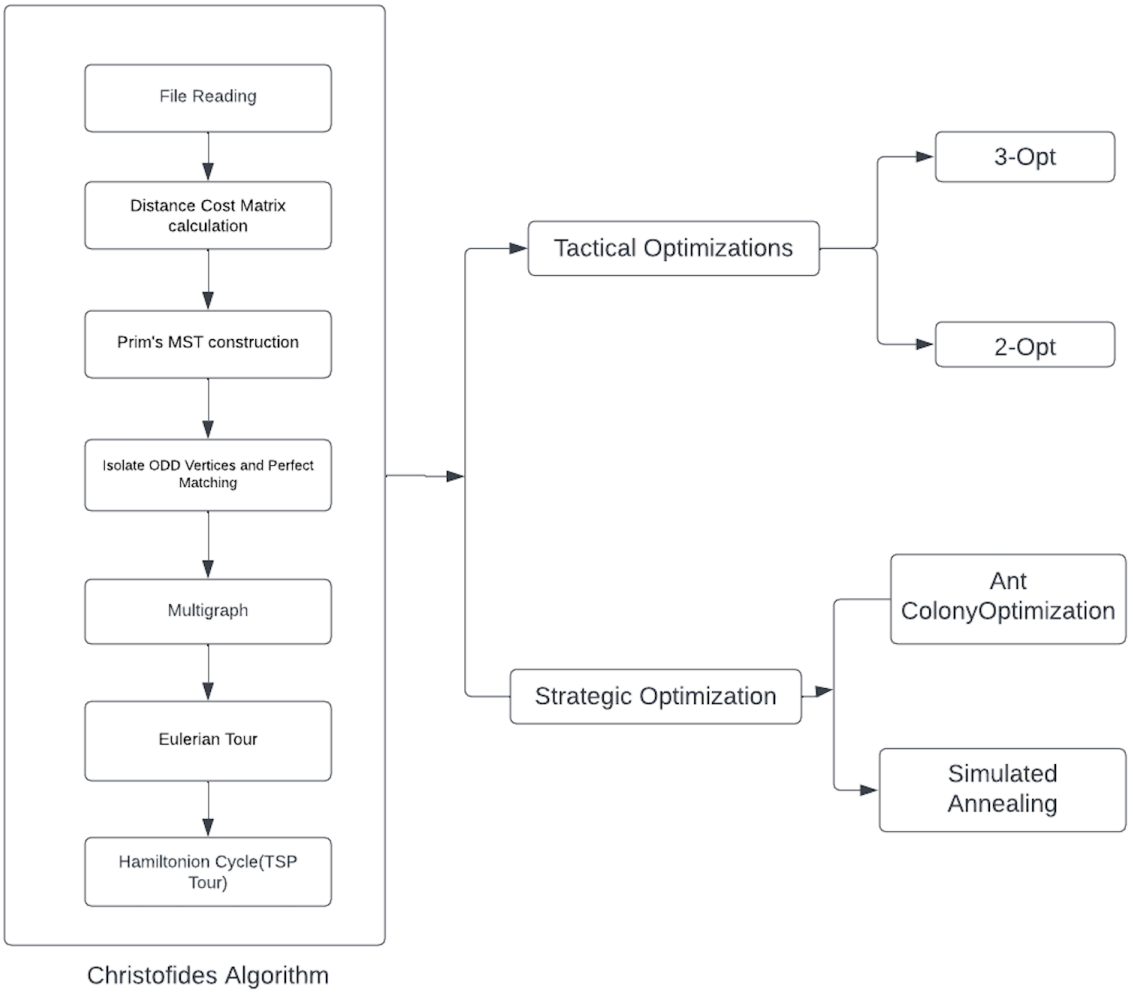
Step 14 : Finally, we applied our last Optimization, the Ant Colony Optimization which uses the concept of ants, their pheromones and their ability to determine best paths based on shared knowledge.

Invariants

- **Number of cities:** The number of cities to be visited in the TSP problem is always fixed and remains constant throughout the algorithm.
- **Distance matrix:** The distance cost matrix is a symmetric matrix, where the diagonal elements are set to infinity. The distance between two cities remains constant and doesn't change throughout the algorithm.
- **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. It has N-1 edges for a graph having N vertices. This ensures that the total cost of the edges in the tree is minimised. 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.
- **Perfect matching:** The algorithm finds a set of perfect matching edges in the minimum spanning tree, which is unique for the given graph.
- **Eulerian cycle:** The algorithm finds an Eulerian cycle in the multigraph formed by combining the minimum spanning tree and perfect matching edges. The Eulerian cycle is unique for the given multigraph.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 the subsequent steps of the algorithm rely on the multigraph having even degree vertices.

- **Hamiltonian cycle:** The algorithm removes repeated cities in the Eulerian cycle to obtain a Hamiltonian cycle, which is unique for the given TSP problem.A Hamiltonian cycle is a path that visits every vertex in the graph exactly once. This is obtained by traversing the Eulerian tour, skipping vertices that have already been visited until a cycle is formed. This condition is an invariant because the final output of the algorithm is a Hamiltonian cycle, which satisfies the TSP problem.

3. FLOWCHART



4. OBSERVATIONS & GRAPHICAL ANALYSIS

Below are various observations while implementing the algorithms and optimizations at each stage ,

1. **MST using Christofides algorithm**:

   Inorder to create a MST we have to calculate the cost distance matrix ,which is used throughout the algorithm ,initially we used Euclidean distance formula but as the data point were longitudes and latitudes we had to change it to Haversine distance and that made a difference in the MST that we obtained .We implemented Kruskal and Prim's to compare the difference and come down to the algorithm that could be used.We implemented Prim's based on those observations .The eulerianCycle() method first checks if there is a vertex with an odd degree in the graph, which is a necessary condition for the existence of an Eulerian cycle. If such a vertex exists, then the eulerUtil() method is called to recursively find an Eulerian cycle starting from that vertex. To make Minimum Perfect Matching and Odd vertices were correct we tested for fewer data points.We used a greedy matching strategy, creating all potential connections between the odd vertices, then organising them by weight. Then, in each iteration, we choose the least weighted edges with distinct vertices.

2. **2-Opt:** To improve the Two-opt approach, a delta function is utilised to determine the difference between the lengths of the old and new edges following swapping rather than repeatedly swapping and reversing the list of vertices. This eliminates the need for time-consuming, iterative length checks of each tour. The delta function shows which swaps have shorter tours and speeds up convergence to an ideal solution.The do2Opt method performs the actual 2-opt swap, i.e., it reverses the order of the cities between indices i and j in the tour. The twoOptimization method implements the 2-opt algorithm itself. It takes the TSP tour and the current distance of the tour as input, and returns the improved distance after applying the 2-opt swaps.The algorithm works by iterating over all pairs of cities in the tour, and checking if swapping them would reduce the total length of the tour. If a swap is found that reduces the length of the tour, the do2Opt method is called to perform the actual swap, and the current distance of the tour is updated accordingly. The loop continues until no further improvements can be found.

   Overall, the 2-opt algorithm provides efficient optimization of TSP.

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])

3. **Three Opt optimization:**

To improve further upon the previous approach we implemented the Three Optimization which subtracts the cost of the existing edges and replace that with the new plausible edges and in the scenario that it performs an optimization we can reverse the links between the segment in that region. The delta factor ensures that we only use the better costing path which is present in the ThreeOptimization function which makes further calls to allSegments and reverseSegmentsIfBetter functions.

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])

4. **Simulated Annealing:**

   Based on the temperature and cooling factor we observed that the optimization costs were varying .And using the below acceptance probability we were able to update the new optimised tour on each iteration.The iteration swerve based on the cool down temperature formula .

   Acceptance probability -(currentCost - newCost) / temperature

   temperature *= (1-coolingRate)

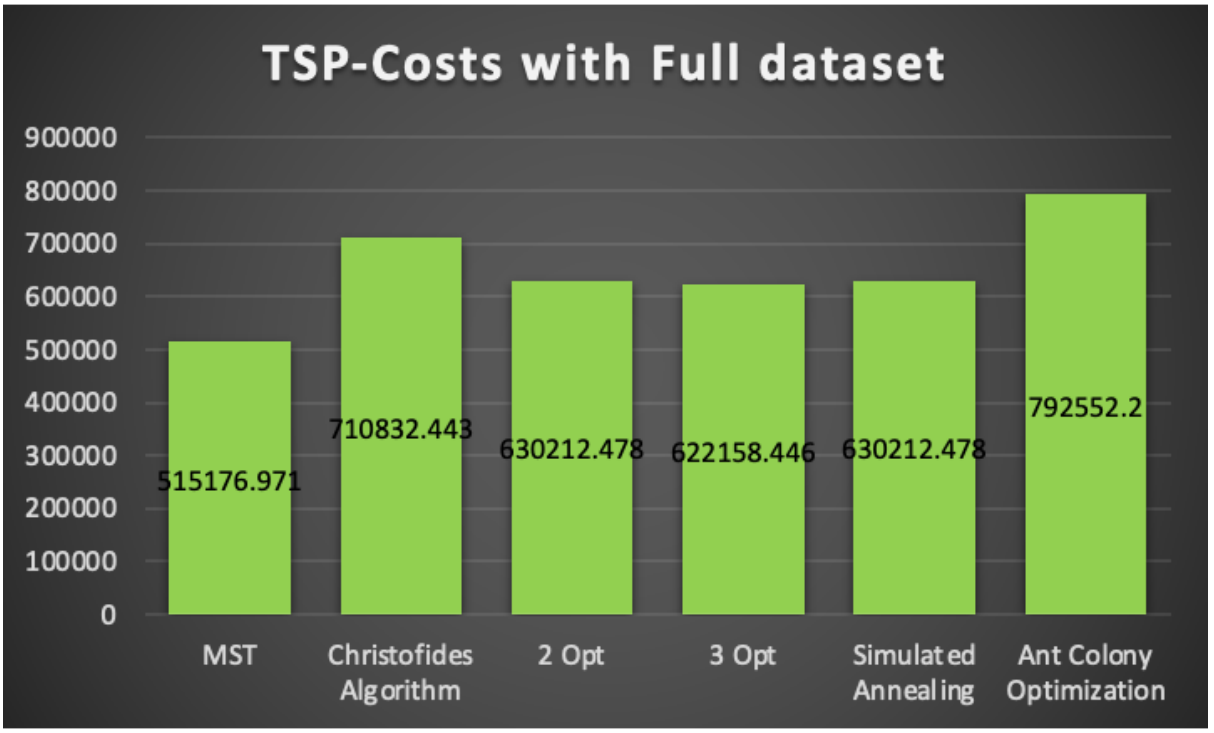
   We observed better results when we combined 2-opt and simulated annealing .

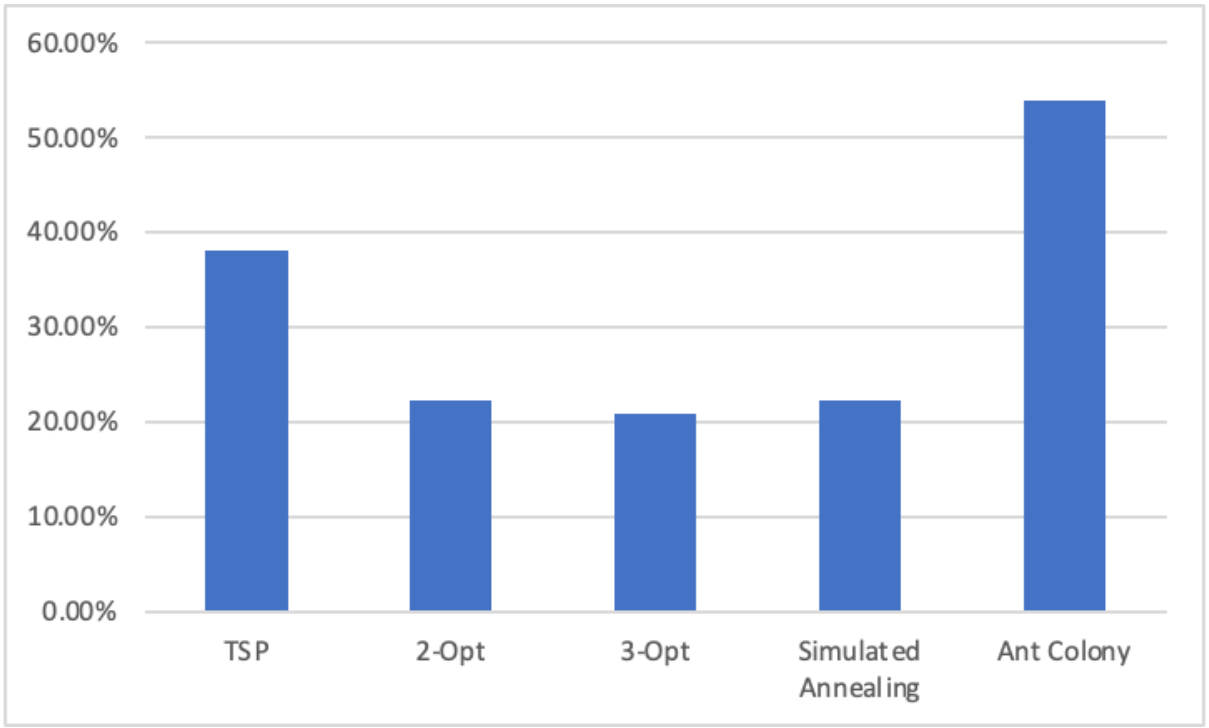
5. **Ant Colony:**

   On changing the below values,

   α= 0.25, β= 10, ρ=0.5, no. of ants=100, no. of iterations=100

   We observed that the optimised tour cost was changing for every run .But since it was taking time more than 60 minutes as we increased the no of iterations and no of ants we could not completely test all the scenarios due to which the final tsp cost is not fully optimized.





Optimization percentage compared to TSP

As is evident from the above two graphs we see that the performance by 3 Opt is the best followed by 2 Opt and Simulated Annealing which is almost comparable. Ant Colony is performing the worst with the current parameters.

5. RESULTS & MATHEMATICAL ANALYSIS

All costs shown are in metres.

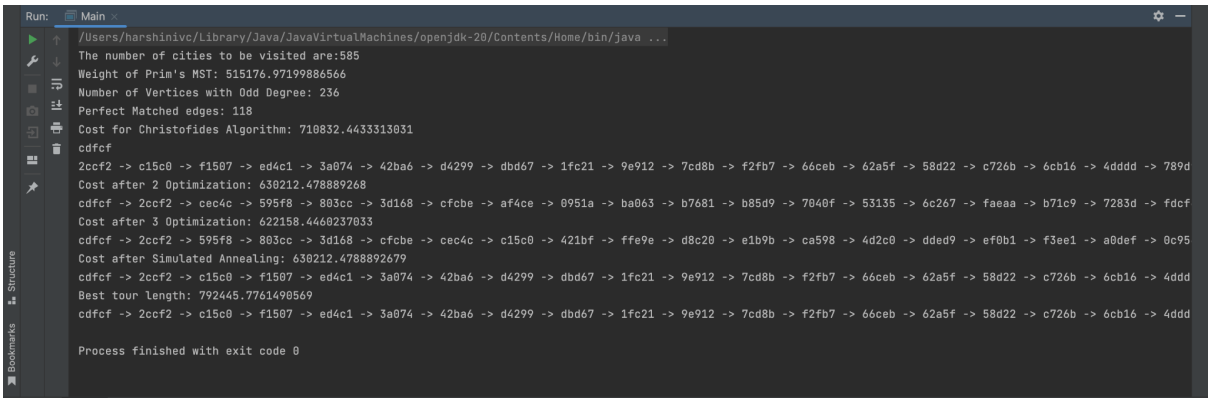
Graph Problems	**Total Cost**	**Algorithm**
Minimum Spanning Tree	515176.971	Prim's MST
TSP	710832.443	Christofides

Optimization Method	**Total Cost**	**Type of Optimization**
2-Opt	630212.478	Tactical
3-Opt	622158.446	Tactical
Simulated Annealing	630212.478	Strategic
Ant Colony Optimization	792552.20	Strategic

Optimization Method	**% increase (wrt MST)**	**Type of Optimization**
TSP	37.98%	Christofides Algorithmic
2-Opt	22.33%	Tactical
3-Opt	20.76%	Tactical
Simulated Annealing	22.33%	Strategic
Ant Colony	53.84%	Strategic

Optimization Method	**% Optimised (wrt Christophides)**	**Type of Optimization**

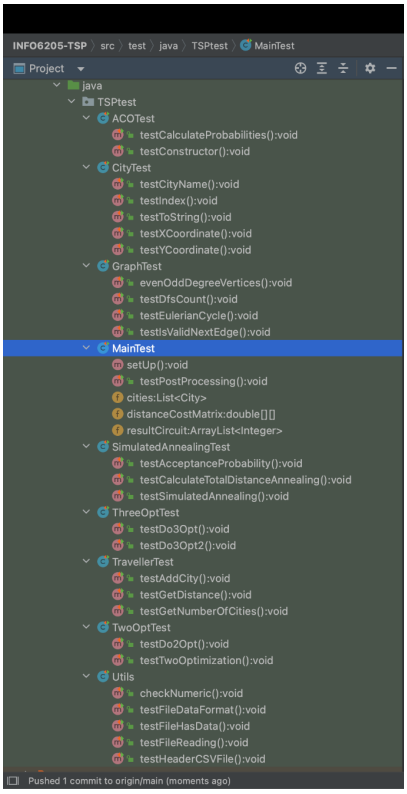
2-Opt	11.34%	Tactical
3-Opt	12.47%	Tactical
Simulated Annealing	11.34%	Strategic
Ant Colony	-11.49%	Strategic



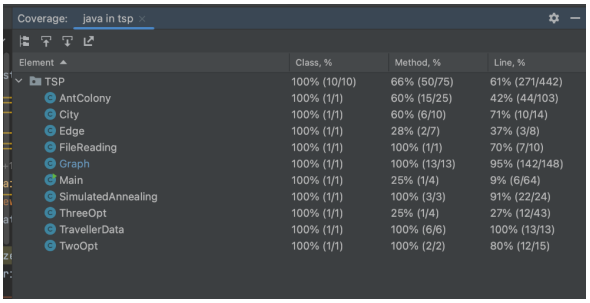
6. UNIT TESTS

We have incorporated a total of 27 test cases to check the various code files.

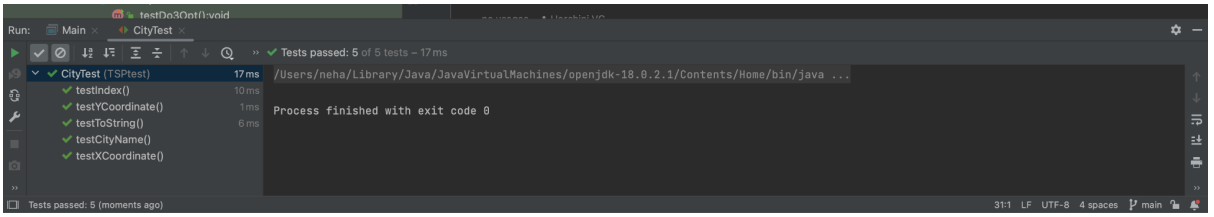
TEST CASES AT A GLANCE

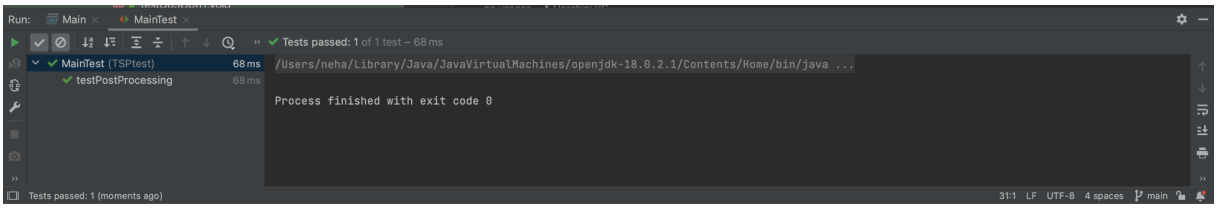


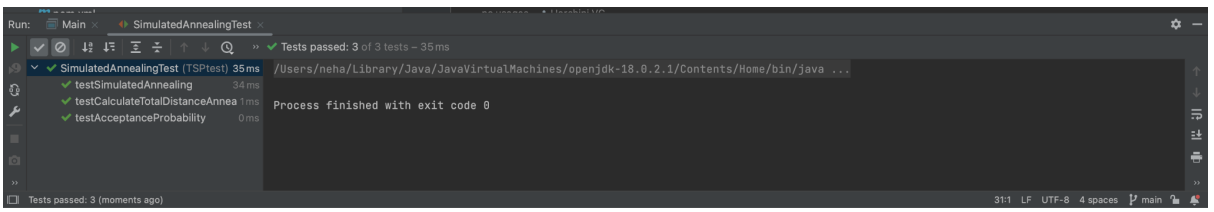
COVERAGE

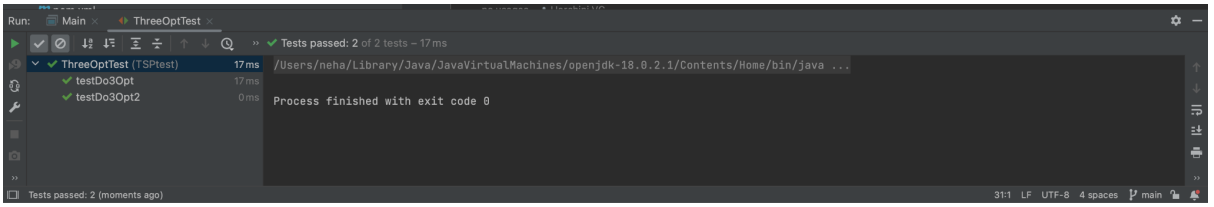


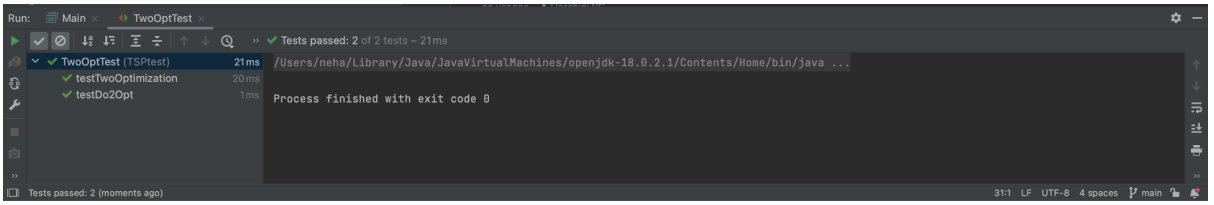
INDIVIDUAL TEST CASES FOR CLASSES











7. CONCLUSION

An age-old problem with real world impacts and far-reaching benefits on obtaining an optimised solution, the Travelling Salesman Problem seemed abundantly easy when we set out to solve it but became progressively complicated.

Using the Christofides Algorithm, we attempted to solve this NP-Hard problem in $O(N^3)$ time complexity. Using Prim's to calculate the MST was beneficial and provided a very optimised calculation of the cost. The Christofides algorithm is a good multi-step process which at each step logically reduces and chooses least cost edges to traverse the cities with a 37.98% optimised TSP tour is obtained.

The optimizations are beneficial but the Strategic Optimizations though useful could not provide as much of an improvement. The Ant Colony Optimization definitely proposes a very intuitive and nature proven way but it failed to provide any improvement in our case. We did learn that keeping the ants within a range of 100 to 500 yields the lowest cost path. Further fine tuning the remaining parameters such as pheromone matrix and evaporation factor may have resulted in a more optimised solution but as of now the cost path could be better.

Among the Tactical Optimizations, 3 Opt has performed slightly better than 2 Opt. with a 2% more efficient outcome. However, allowing it to run longer and many more iterations may result in an even better outcome. The 2 Opt is very fast and provides a good result in a very short time.

Among all the optimizations the reduction shown by 3 Opt is the most prominent and relevant. The tour generated by Christofides is nearly 1.37 times(710832.443/515176.971) that of MST which is well within range and quite optimal.

References

Simulated Annealing -
https://stackabuse.com/simulated-annealing-optimization-algorithm-in-java/
https://www.baeldung.com/java-simulated-annealing-for-traveling-salesman
https://stackoverflow.com/questions/17281954/simulated-annealing-tsp

Ant Colony Optimization -
https://strikingloo.github.io/ant-colony-optimization-tsp
https://www.hilarispublisher.com/open-access/solving-traveling-salesmen-problem-using-ant-colony-optimizationalgorithm-2168-9679-1000260.pdf
https://www.baeldung.com/java-ant-colony-optimization

2-opt-
https://en.wikipedia.org/wiki/2-opt
https://www.technical-recipes.com/2017/applying-the-2-opt-algorithm-to-travelling-salesman-problems-in-java/
https://www.keiruaprod.fr/blog/2021/09/15/traveling-salesman-with-2-opt.html

3-opt-
https://en.wikipedia.org/wiki/3-opt
http://matejgazda.com/tsp-algorithms-2-opt-3-opt-in-python/

TSP-
https://www.geeksforgeeks.org/traveling-salesman-problem-tsp-implementation/
https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_travelling_salesman_problem.htm