INFO 6205 Spring 2023 Project

Traveling Salesman

Aim:

The aim of this project is to develop an efficient solution for the Traveling Salesman Problem (TSP) using the Christofides algorithm and optimization methods such as random swapping, 2-opt and/or 3-opt improvement, simulated annealing, ant colony optimization, genetic algorithms, etc. The goal is to find the shortest tour of n cities (or points in a two-dimensional space) with improved time complexity compared to the deterministic solutions, which have a complexity of $O(k!)$.

Approach:

1. Understand the Christofides algorithm: The first step of the project will be to gain a deep understanding of the Christofides algorithm, which is a popular heuristic algorithm for solving the TSP. This will involve studying the algorithm's theoretical foundations, including the concepts of minimum spanning trees, Eulerian circuits, and the algorithm's steps for finding an approximate solution.

2. Implement the Christofides algorithm: Next, the Christofides algorithm will be implemented in a programming language of choice. This will involve coding the steps of the algorithm, including the construction of a minimum spanning tree, finding an Eulerian circuit, and modifying it to obtain an approximate TSP tour.

3. Optimize the solution: To improve the solution obtained from the Christofides algorithm, various optimization methods will be applied. This will include tactical optimization methods such as random swapping, 2-opt and/or 3-opt improvement, which involve making small adjustments to the tour to potentially find a shorter path. Additionally, strategic optimization methods such as simulated annealing, ant colony optimization, genetic algorithms, etc., will be explored to further refine the solution.

4. Evaluate and compare solutions: The performance of the optimized solution will be evaluated and compared against the results obtained from the basic Christofides algorithm. This will involve measuring the tour length, time complexity, and solution quality, and comparing them to determine the effectiveness of the optimization methods.

5. Fine-tune and optimize: Based on the evaluation and comparison results, the solution will be fine-tuned and optimized further. This may involve tweaking parameters, adjusting algorithms, or exploring additional optimization methods to achieve the best possible solution for the TSP.

6. Documentation and reporting: Finally, the project findings, including the implemented Christofides algorithm, optimization methods, evaluation results, and any insights gained, will be documented and reported. This will provide a comprehensive overview of the project and its outcomes for further analysis and future research.

Requirements to run the project

- Java SE
- NodeJS
- React

Steps to run the project

1. Backend

To run the program only in java run Driver.java file. This will display the results of the project in your IDE terminal.

2. Visualization

a. Our program runs on SpringBoot in backend and React as frontend.
b. To run the project with UI, run TspSpringApplication.java
c. For frontend, open UI folder in the terminal and run “npm i” to install the node modules.
d. To run the UI run ‘npm start’ and open localhost:3000.
e. Our backend is running on localhost:8080, if front end fails to run you can the view the tour array on localhost:8080/christofides, localhost:8080/two-opt, localhost:8080/three-opt, localhost:8080/ aco, localhost:8080/sa
f. Once localhost:3000 is opened **ONLY** use the navbar to navigate between the 5 algorithm visualizations.
g. Christofides and Two opt return data almost immediately and start showing the tour on our map. However, Simulated Annealing and Ant Colony Optimisation take almost 2-3 minutes to run. Three-opt takes 15-20 minutes to run. It is recommended to not switch algorithm selection before the algorithm fully prints the route on the map. Preferred way to run the visualization is to start with Christofides -> two opt -> Ant Colony -> Simulated Annealing -> Three opt

Note: Please ignore the pop up box which says “this page can’t load google maps correctly”. It will disappear if we click ok. It is appearing due to a billing issue with google api.

Program

A) Ant Colony Optimization(ACO)

(ACO.java)

- Data Structures:

  2D arrays: distanceMatrix and pheromoneMatrix are 2D arrays used to represent the distance between cities and the pheromone levels on the edges between cities, respectively.

- Classes:

  ACO: This class represents the Ant Colony Optimization (ACO) algorithm for solving the Traveling Salesman Problem (TSP). It contains methods for initializing and updating pheromone matrix, constructing tours, choosing next cities, calculating tour lengths, and solving the TSP using ACO.

- Algorithms:

  Ant Colony Optimization (ACO): The ACO algorithm is used to solve the Traveling Salesman Problem (TSP) in this code. ACO is a probabilistic algorithm inspired by the foraging behavior of ants, where ants cooperate to find the shortest path between cities by depositing and following pheromone trails on the edges between cities.

- Invariants:

  Pheromone Invariant: The pheromone matrix is updated at each iteration of the ACO algorithm based on the tours of all ants. The pheromone values are evaporated and updated according to the ant's tour lengths and the evaporation factor.

  Tour Invariant: At each iteration, the best tour found so far is updated based on the tour lengths calculated for each ant's constructed tour. The best tour and its length are stored and updated if a shorter tour is found.

(Ant.java)

- Data structures:

  Arrays: Arrays are used to store the visited cities and the tour of the ant, with boolean[] visited and int[] tour respectively.

- Classes:

  Ant: A class representing an ant in an Ant Colony Optimization (ACO) algorithm. It has private member variables for the number of cities (numCities), the initial city (initialCity), the current city (currentCity), an array to keep track of visited cities (visited), an array to store the tour of the ant (tour), and the total length of the tour (tourLength). It also has methods to visit a city, get the current city, check if a city has been visited, get the tour, get the tour length, get the initial city, and calculate the remaining number of cities.

- Algorithms:

  Ant Colony Optimization (ACO): The code is implementing the Ant class as part of an ACO algorithm.

- Invariants:

  Visited cities: The visited array is used to keep track of cities visited by the ant, ensuring that a city is visited only once during the construction of the tour.

  Remaining cities: The remainingCities() method calculates the remaining number of cities that have not been visited by the ant, ensuring that the ant visits all cities exactly once during the construction of the tour.

B) Christofides Algorithm:

(Christofides.java)

- Data structures:

  double[][] graph: a two-dimensional array representing the input graph.

  double[][] minimumSpanningTree: a two-dimensional array representing the minimum spanning tree.

  List<Integer> oddDegreeVertices: a list of integers representing the vertices with odd degrees in the minimum spanning tree.

  double[][] perfectMatching: a two-dimensional array representing the perfect matching.

  double[][] combinedGraph: a 2D array representing the combined grap

- Classes:

  Christofides: the main class that implements the Christofides algorithm.

- Algorithms:

  Prim's algorithm: used to find the minimum spanning tree of the input graph.
  Greedy algorithm: used to find a perfect matching for the odd-degree vertices.
  Eulerian Tour algorithm: used to find the Eulerian tour of the combined graph.
  Hamiltonian Cycle: used to convert the Eulerian tour to a Hamiltonian cycle.
  2-Opt algorithm: used to improve the initial solution (Hamiltonian cycle) using the combined graph.

- Invariants:

  At the end of each step, the resulting graph must still be a valid graph, meaning that it should not have negative edges, loops or disconnected components.

  The number of odd-degree vertices in the minimum spanning tree must be even, since the sum of the degrees of all vertices in a graph must be even. If the number of odd-degree vertices is odd, then the algorithm should throw an error.

C) *Simulated Annealing*

(SimulatedAnnealing.java)

- Data Structures:

  2D array (double[][]): Represents the distance matrix, where distanceMatrix[i][j] represents the distance between city i and city j.

  List (ArrayList<Integer>): Represents the current tour, initial tour, new tour, and best tour, which are lists of city indices representing the order of cities in the current tour.

- Classes:

  SimulatedAnnealing: Contains the implementation of the Simulated Annealing algorithm for solving the Traveling Salesman Problem.

- Algorithms:

  Simulated Annealing: An optimization algorithm used to find an approximate solution to combinatorial optimization problems like the Traveling Salesman Problem. It involves iteratively perturbing the current solution (tour) by swapping two cities and accepting the new solution with a certain probability determined by the current temperature and the difference in tour distances (delta), aiming to escape local optima and explore the solution space for potentially better solutions.

- Invariants:

  Distance Matrix: The distance matrix is assumed to be a symmetric matrix, where distanceMatrix[i][j] = distanceMatrix[j][i] represents the distance between city i and city j.

  City Indices: City indices are assumed to start from 0 and go up to n-1, where n is the total number of cities.

  Tour Representation: The tour is represented as a list of city indices, where the order of cities in the list represents the order of cities in the tour. The tour is assumed to be a complete tour, i.e., it starts and ends at the same city.

  Initial Tour: The initial tour provided during object creation is assumed to be a valid tour, i.e., it includes all city indices exactly once.

  Randomness: The algorithm uses the java.util.Random class for generating random numbers for perturbing the tour and accepting new solutions probabilistically. The randomness is assumed to be uniformly distributed and adequately random for the algorithm's purposes.

  Cooling Rate and Initial Temperature: The cooling rate and initial temperature values are assumed to be appropriate and carefully chosen for the specific problem instance to achieve a good trade-off between exploration and exploitation in the solution space.

D) Three-Opt Algorithm

(ThreeOpt.java)

- Data structures:

  List<Integer> tour: Represents the tour as a list of integers, where each integer represents the index of a city in the tour.

  double[][] distances: Represents the distance matrix or graph of the cities, where distances[i][j] stores the distance between city i and city j.

- Classes:

  None. This code consists of a single class, ThreeOpt, which contains static methods for implementing the 3-opt algorithm.

- Algorithms:

  3-opt algorithm: This algorithm is used to optimize a tour by considering all possible combinations of 3 edges and their possible swaps. It is implemented in the ThreeOpt.threeOptAlgorithm() method, which takes a tour and a distance matrix as input,

and iteratively applies the 3-opt swaps to improve the tour until no further improvement is possible.

- Invariants:

  The tour variable should always represent a valid tour, where each city is visited exactly once and the tour returns to the starting city.

  The distances matrix should always represent a valid distance matrix or graph, where distances[i][j] stores the distance between city i and city j.

  The calculateTourDistance() method should always return a valid total distance of the tour, calculated based on the distances matrix.

  The threeOptAlgorithm() method should always return a valid optimized tour after applying the 3-opt swaps.

E) *Two-Opt Optimization:*

(TwoOpt.java)

- Data Structures:

  int[]: Arrays of integers are used to represent tours and distances in the form of city indices.

  double[][]: 2D arrays of doubles are used to represent the distance matrix, where distanceMatrix[i][j] represents the distance between city i and city j.

- Classes:

  TwoOpt: This class implements the 2-opt algorithm for solving the TSP. It has methods for performing the 2-opt optimization and reversing a tour.

- Algorithms:

  2-opt Algorithm: The 2-opt algorithm is a local search algorithm commonly used for solving the TSP. It iteratively swaps pairs of edges in the tour to find shorter paths, and continues until no further improvement is possible.

- Invariants:

  The distances in the distance matrix should be non-negative and represent the distances between cities. Negative distances or missing distances may result in incorrect tour optimizations.

The tour should be a valid permutation of city indices, where each city appears exactly once in the tour.

The distance matrix should be symmetric, where distanceMatrix[i][j] should be equal to distanceMatrix[j][i] for all i and j, representing that the distance between city i and city j is the same in both directions.

F) Minimum Spanning Tree

(MST.java)

- Data Structures:

  2D Array (double graph[][]): Represents the weighted adjacency matrix of the input graph. It stores the edge weights between vertices, where graph[i][j] represents the weight of the edge from vertex i to vertex j. A value of 0 indicates no edge between the vertices.

  1D Arrays (double key[], boolean mstSet[], int parent[]): These arrays are used to keep track of the properties of vertices during the execution of the algorithm.
  key[]: Represents the minimum key value required to include a vertex in the MST.
  mstSet[]: Represents whether a vertex is already included in the MST or not.
  parent[]: Represents the parent of each vertex in the MST.

- Classes:

  MST: Represents the main class that contains the implementation of Prim's algorithm for finding the minimum spanning tree of a graph. It has three methods:
  minKey(double key[], boolean mstSet[]): Finds the vertex with the minimum key value.
  printMST(int parent[], double graph[][]): Prints the MST distance.
  primMST(double graph[][]): Implements Prim's algorithm to find the MST.

- Algorithm:

  Prim's Algorithm for Minimum Spanning Tree: An algorithm to find the minimum spanning tree of a graph. It initializes key values, mstSet, and parent arrays, and iteratively selects vertices with minimum key values, updates key values and mstSet, and finds the parent vertices until all vertices are included in the MST.

- Invariants:

  key[] always represents the minimum key value required to include a vertex in the MST.

mstSet[] always represents whether a vertex is already included in the MST or not. parent[] always represents the parent of each vertex in the MST.

The graph represented by graph[][] is assumed to be an undirected, weighted graph, where graph[i][j] represents the weight of the edge from vertex i to vertex j. The weights are assumed to be non-negative.

G) Traveling Salesman Problem (TSP)

(TSP.java)

- Data structures:

  double[][] graph: A 2D array representing the graph or distance matrix read from the "teamproject.csv" file.

  List<Integer>: Lists are used to store the initial tour, Christofides solution, 2-opt solution, 3-opt solution, ACO solution, and best tour found by Simulated Annealing algorithm.

- Classes:

  TSP: Represents the main class that implements the Traveling Salesman Problem (TSP) solution. It contains methods for reading the graph from a file, building an initial solution, and applying various TSP algorithms such as Christofides, 2-opt, 3-opt, Ant Colony Optimization (ACO), and Simulated Annealing.

- Algorithms:

  Christofides algorithm: This algorithm is used to find an approximate solution to the TSP. It is applied using the Christofides.applyChristofidesAlgorithm() method.

  2-opt algorithm: This algorithm is used to optimize an initial tour by swapping pairs of edges to shorten the total tour distance. It is applied using the TwoOpt.twoOpt() method

  3-opt algorithm: This algorithm is used to further optimize a tour by considering all possible combinations of 3 edges and their possible swaps. It is applied using the ThreeOpt.threeOptAlgorithm() method.

  Ant Colony Optimization (ACO) algorithm: This algorithm is used to find a solution to the TSP based on the behavior of ants. It is applied using the ACO.solve() method.

Simulated Annealing algorithm: This algorithm is used to find a solution to the TSP by simulating the annealing process of a material. It is applied using the SimulatedAnnealing.solve() method.

- Invariants:

  The graph variable should always represent a valid distance matrix or graph read from the "teamproject.csv" file.

  The methods should return valid solutions as lists of integers representing the tour or path of the TSP. The returned tours should be valid tours that visit all cities exactly once and return to the starting city.

H) (ReadCoordinates.java)

- Data Structures:

  List<double[]>: Used to store the coordinates read from the CSV file as double arrays representing longitude and latitude

  double[][]: Used to represent the graph with distances between pairs of coordinates.

- Classes:

  ReadCoOrdinates: Represents the class itself that contains the methods for reading input from a CSV file and calculating distances between coordinates.

- Algorithms:

  Distance Calculation: The distance() method uses the haversine formula to calculate the distance between pairs of coordinates on the Earth's surface. This is a common algorithm used to calculate distances between points on a sphere or an ellipsoid, such as the Earth.

- Invariants:

  CSV file format: The code assumes that the input CSV file has lines with comma-separated values where the first value is skipped as it is considered as a header.

  Coordinate order: The code assumes that the coordinates in the CSV file are given in the order of longitude and latitude, i.e., the second and third values in each line respectively.

  Graph representation: The code assumes that the graph representation is an adjacency matrix (double[][]) where the distances between pairs of coordinates are stored based on their indices in the coordinates list. The distances are calculated using the distance() method.

I) *(TourDistance.java)*

- Data Structures:

  double[][]: Used to represent the graph with distances between pairs of vertices.
  double[]: Used to store the coordinates of each vertex as an array of doubles.

- Classes:

  TourDistance: Represents the class itself that contains static methods for calculating tour distances and coordinate distances.

- Algorithms:

  Tour Distance Calculation: The calculateTourDistance() method calculates the total distance of a given tour in a graph by summing up the distances between consecutive vertices based on the provided adjacency matrix (graph).

  Actual Tour Distance Calculation: The getActualTourDistance() method calculates the total distance of a given tour by summing up the distances between consecutive vertices based on the provided GraphInfo object, which contains the coordinates of the vertices. The getCoordDistance() method is used to calculate the distance between two coordinates using the Euclidean distance formula.

  Coordinate Distance Calculation: The getDistance() method calculates the distance between two coordinates on the Earth's surface using the haversine formula, which takes into account the curvature of the Earth.

- Invariants:

  Graph representation: The code assumes that the graph representation is an adjacency matrix (double[][]) where the distances between pairs of vertices are stored.

  Coordinate format: The code assumes that the coordinates are represented as arrays of doubles with the first value representing the longitude and the second value representing the latitude.Earth's radius: The code assumes a constant value of 6371 kilometers for the Earth's radius in the haversine formula.

J) (TspSpringApplication.java)

- Data Structures:

  List<Integer>: Used to represent the output data, which is a list of integers representing the sequence of cities in the optimal TSP route.

  double[][]: Used to represent the graph data, which is a 2D array of doubles representing the distances between cities in the TSP problem.

- Classes:

  TSP: This class is being used to solve the TSP problem. It contains methods for different TSP solving algorithms, such as tsp(int algorithmType) which takes an integer parameter representing the algorithm type (e.g., 0 for Christofides, 1 for 2-opt, etc.) and returns a list of integers representing the optimal TSP route.

- Algorithms:

  Christofides algorithm: It is a heuristic algorithm for solving the TSP problem that uses a combination of minimum spanning tree, perfect matching, and cycle merging.

  2-opt algorithm: It is a local search algorithm for improving an existing solution by swapping pairs of edges in the current solution to potentially find a shorter route.

  3-opt algorithm: It is a more advanced version of the 2-opt algorithm that considers three edges instead of two, allowing for more complex swaps to potentially find a shorter route.

  Ant Colony Optimization (ACO) algorithm: It is a metaheuristic algorithm inspired by the behavior of ants in finding optimal routes to food sources. It uses pheromone trails and heuristics to guide the search for the optimal TSP route.

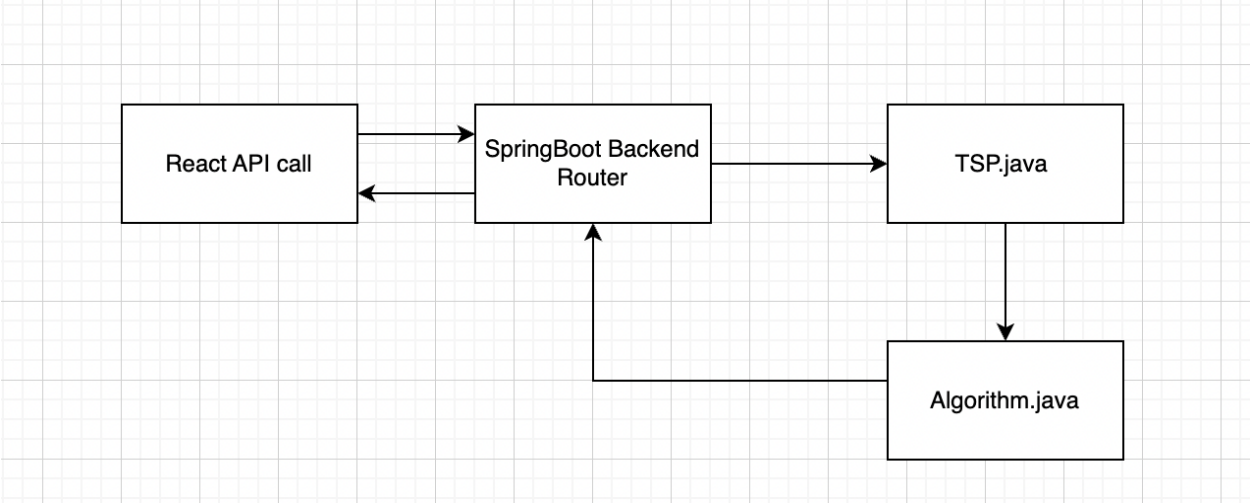
  Simulated Annealing (SA) algorithm: It is a metaheuristic optimization algorithm that uses a random search and probability-based decision-making approach to escape local optima and find the optimal TSP route.

- Invariants:

  @CrossOrigin(origins = "http://localhost:3000"): This annotation is used to specify the allowed origins for cross-origin requests to these endpoints. It allows requests from the specified origin (http://localhost:3000) to access these endpoints, which may be useful for enabling cross-origin requests from a client-side web application running on a different domain.

  @GetMapping: This annotation is used to map HTTP GET requests to the corresponding endpoint methods in the MyController class, allowing for handling of GET requests and returning appropriate responses.

Flow Charts (inc. UI Flow)



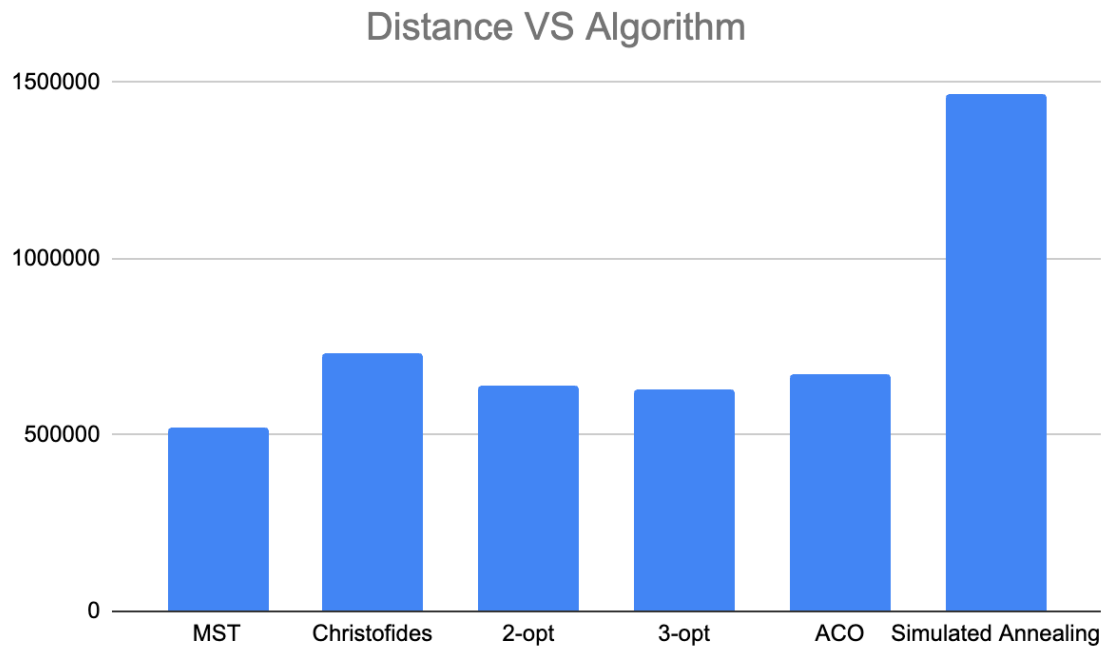
- The Spring Boot application starts by executing the main method in the TspSpringApplication class.
- The application initializes and sets up the Spring Boot framework with the @SpringBootApplication annotation, which includes the default configuration and enables components scanning.
- The MyController class is defined within the TspSpringApplication class and acts as a RESTful web service that handles incoming HTTP GET requests.
- The @CrossOrigin(origins = "http://localhost:3000") annotation allows cross-origin requests from the http://localhost:3000 origin to access the endpoints in the MyController class.
- The @GetMapping annotation maps specific HTTP GET requests to the corresponding endpoint methods in the MyController class.
- When a request is received at any of the mapped endpoints (e.g., /two-opt, /distance, /three-opt, etc.), the corresponding endpoint method is executed.
- Inside each endpoint method, the TSP class is instantiated and the appropriate method for solving the TSP problem is called based on the algorithm type passed as a parameter (e.g., 2-opt, 3-opt, Christofides, ACO, SA).
- The TSP class contains implementations of the TSP solving algorithms, such as 2-opt, 3-opt, Christofides, ACO, and SA.
- The TSP solving algorithm computes the optimal TSP route based on the input data, which includes the graph data (distances between cities) and the algorithm parameters.

- The optimal TSP route is returned as a list of integers representing the sequence of cities in the route.
- The list of integers representing the optimal TSP route or other output data (e.g., graph data) is returned as the HTTP response from the endpoint method.
- The HTTP response is sent back to the client that made the original request (e.g., a web browser or another client-side application), which can then process the response and display the results to the user.

Observations

MST	Christofides	2-opt	3-opt	ACO	Simulated Annealing
523434	733855	641553	630328	673525	1467973

Graphical Representation



Results & Mathematical Analysis

The Travelling Salesman Problem (TSP) is a well-known optimization problem in computer science and mathematics. The problem is defined as follows: given a list of cities and the distances between them, find the shortest possible route that visits each city exactly once and returns to the starting city.

To analyze the TSP mathematically, we can use graph theory and linear programming.
Let G = (V, E) be an undirected complete graph, where V is the set of cities and E is the set of edges between cities. Each edge e = (i,j) in E has a weight w(e) representing the distance between city i and city j.

We can represent the TSP as an integer linear program (ILP) as follows:-
minimize $\sum w(e)x(e)$ (1)
subject to:
$\sum x(e) = 2$ for each vertex $i \in V$ (2)
$\sum x(e) \leq |V| - 1$ for each subset $S \subseteq V$, $|S| \geq 2$, where S does not contain the starting city (3)
$x(e) \in \{0, 1\}$ for each edge $e \in E$ (4)

In the above ILP, x(e) is a binary variable that takes the value 1 if edge e is included in the tour and 0 otherwise. Constraint (2) ensures that each city is visited exactly once, and constraint (3) ensures that the tour is connected and does not contain any subtours.

The TSP is an NP-hard problem, which means that there is no known polynomial-time algorithm that can solve it exactly for all instances. However, there are several approximation algorithms that can provide solutions that are guaranteed to be within a certain factor of the optimal solution. We are implementing four of such algorithms.

xx	MST	Two Opt	Three Opt	Christofides	ACO	Simulated Annealing
Tour Distance	523434	641553	630328	733855	673525	1467973
Ratio w MST	1	1.226	1.204	1.402	1.287	2.805

The algorithms two opt three opt and ACO are pretty much in the ballpark and take around 25% more distance than Minimum Spanning Tree. But the other two algorithms take a little more time to run.

The **time complexity** of the algorithms used to solve the traveling salesman problem depends on the specific algorithm and its implementation. Here are the time complexities for some common algorithms used to solve the traveling salesman problem:

Brute Force: The brute force algorithm for TSP is to generate all possible tours and find the shortest one. The time complexity of this algorithm is O(n!), where n is the number of cities.

Christofides Algorithm: The Christofides algorithm is a heuristic algorithm that starts by finding a minimum spanning tree of the graph, then creates a minimum-weight perfect matching on the odd-degree vertices, and finally finds an Eulerian tour of the resulting graph. The time complexity of this algorithm is O(n^3).

2-opt Algorithm: The 2-opt algorithm is a local search algorithm that repeatedly removes two edges from the tour and reconnects them to create a new tour, until no improvement can be made. The time complexity of this algorithm is O(n^2).

3-opt Algorithm: The 3-opt algorithm is an extension of the 2-opt algorithm that considers three edges at a time instead of two. The time complexity of this algorithm is O(n^3).

Simulated Annealing: The simulated annealing algorithm is a global optimization algorithm that starts from an initial solution and iteratively tries new solutions by randomly perturbing the current solution and accepting or rejecting the new solution based on a probability distribution. The time complexity of this algorithm depends on the number of iterations and the time taken to evaluate the objective function.

Ant Colony Optimization: The ant colony optimization algorithm is a metaheuristic algorithm that mimics the behavior of ants searching for food. The time complexity of this algorithm depends on the number of iterations, the number of ants, and the time taken to evaluate the objective function.

It is important to note that these time complexities are only approximate and may vary depending on the implementation and input size.

Unit tests

- *2-Opt:*

The first test, testTwoOpt(), tests the TwoOpt algorithm with a small input. It initializes a 2D array distances representing the distances between cities, and an array tour representing the initial tour.

It then calls the TwoOpt.twoOpt() method with tour and distances as input and stores the result in the actual array. Next, it calculates the tour distance of the actual tour using the TourDistance.calculateTourDistance() method and stores it in the actualDistance variable. Finally, it uses JUnit's assertions to check that the actualDistance is equal to 8.0 and that the expected tour is equal to the actual tour.

The second test, testTwoOptWithLargerInputs(), tests the TwoOpt algorithm with a larger input. It follows a similar pattern as the first test, but with a larger input array distances and a different tour array. It also uses JUnit's assertions to check that the expected tour is equal to the actual tour and that the actualDistance is equal to 12.0.

- *3-Opt:*

The code contains three test methods: testThreeOpt(), testThreeOptWithLargerInputs(), and testThreeOptEmptyTour(), each testing different scenarios.

1. testThreeOpt():

This test method tests the threeOptAlgorithm() method with a small input, where the distances matrix has 5x5 dimensions and the tour list has 5 cities. The expected result is a tour list with a certain order of cities, and the expected total distance of the tour is 8.0. The method uses assertEquals() to check if the actual distance and the expected distance are equal, and assertArrayEquals() to check if the actual tour and the expected tour are equal.

2. testThreeOptWithLargerInputs():

This test method tests the threeOptAlgorithm() method with a larger input, where the distances matrix has 7x7 dimensions and the tour list has 7 cities. The expected result is a tour list with a certain order of cities, and the expected total distance of the tour is 12.0. The method uses assertEquals() to check if the actual distance and the expected distance are equal, and assertArrayEquals() to check if the actual tour and the expected tour are equal.

3. testThreeOptEmptyTour():

This test method tests the threeOptAlgorithm() method with an empty tour list. The expected result is an empty tour list. The method uses assertArrayEquals() to check if the actual tour and the expected tour are equal.

- *Christofides:-*

The test class contains three test methods:

1. testChristofides: This method tests the applyChristofidesAlgorithm method with a small input consisting of a 5x5 distance matrix and a tour. It checks if the output tour and distance match the expected values. The expected tour is {3, 1, 0, 4, 2} and the expected distance is 10.0.

2. testChristofidesLargerInput: This method tests the applyChristofidesAlgorithm method with a larger input consisting of a 7x7 distance matrix and a tour. It checks if the output tour and distance match the expected values. The expected tour is {4, 2, 1, 3, 0, 5, 6} and the expected distance is 16.0.

3. testChristofidesTour: This method tests the applyChristofidesAlgorithm method with a different input consisting of a 4x4 distance matrix and a tour. It checks if the output tour and distance match the expected values. The expected tour is {3, 2, 1, 0} and the expected distance is 95.0.

- *Ant Colony Optimization:-*

The testAntColony() method tests the AntColony class with a smaller input matrix of distances and expected tour. It creates an instance of the ACO class with specific parameters, solves the TSP using ACO algorithm with 100 iterations, retrieves the best tour and calculates the tour distance. It then compares the calculated tour distance with the expected tour distance of 8.0 and compares the actual tour with the expected tour using assertEquals() and assertTrue() assertions.

The testAntColonyLargerInput() method tests the AntColony class with a larger input matrix of distances and expected tour. It follows the same logic as the testAntColony() method but with a larger input. It creates an instance of the ACO class with specific parameters, solves the TSP using ACO algorithm with 100 iterations, retrieves the best tour and calculates the tour distance. It then compares the calculated tour distance with the expected tour distance of 12.0 and compares the actual tour with the expected tour using assertEquals() and assertTrue() assertions

- *Simulated Annealing:-*

In the first test case testSimulatedAnnealing(), a smaller input with a 5x5 distance matrix is used. The test initializes a SimulatedAnnealing solver with the distance matrix and an initial

tour, and then calls the solve() method with specific parameters for the number of iterations, temperature reduction factor, and cooling rate. After obtaining the solution, it calculates the total distance of the resulting tour using the TourDistance.tourDistance() method and asserts that the actual distance matches the expected distance of 8.0 using assertEquals().

In the second test case testSimulatedAnnealingLargerInputs(), a larger input with a 7x7 distance matrix is used. The test follows a similar approach as the first test case but with a larger input. It initializes a SimulatedAnnealing solver with the distance matrix and an initial tour, calls the solve() method with specific parameters, calculates the total distance of the resulting tour, and asserts that the actual distance matches the expected distance of 12.0 using assertEquals().

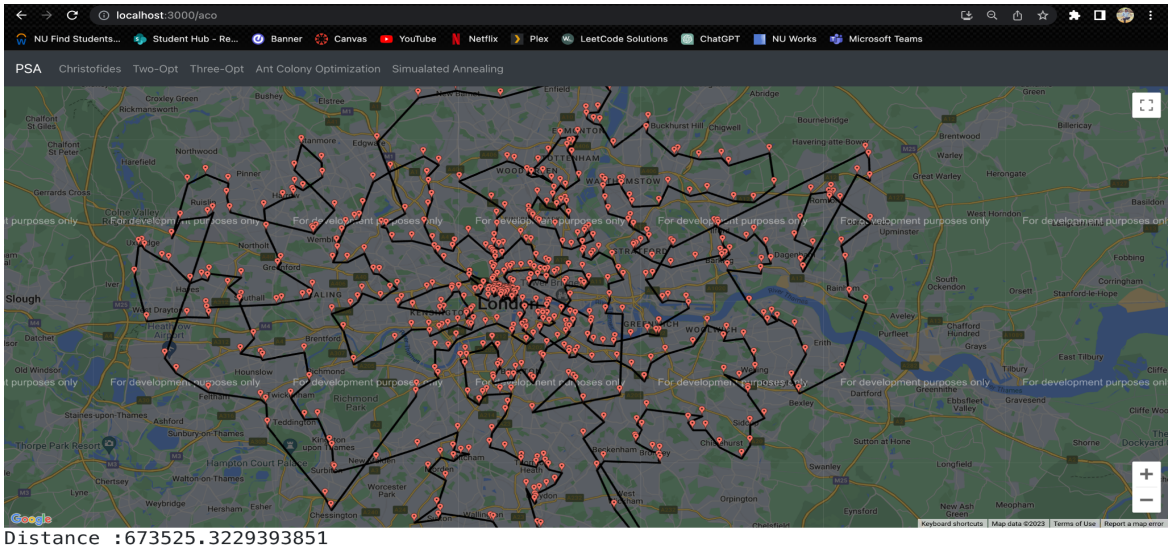
- *TspSpringApplicationTest.java:-*

The @SpringBootTest annotation indicates that this test case is an integration test for a Spring Boot application, and it will load the entire application context along with all the beans and configurations defined in the application.

The contextLoads() method is an empty test method that serves as a placeholder for testing the context loading of the Spring Boot application. If the application context is loaded successfully without any issues, this test case will pass without any assertions or actual test logic.

- **<u>Conclusion</u>**

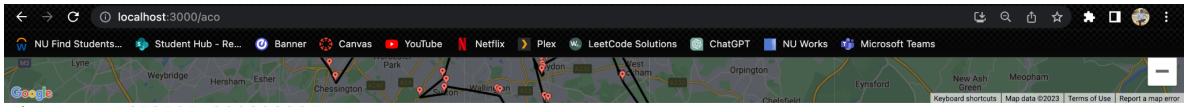
ACO Algorithm:




```
Distance :673525.3229393851
```

#	Crime ID	Lat	Long
482	d399d09ca411c382223266b81b996bf604b077380ced6369dd0eaa315c895134	51.535076	-0.189556

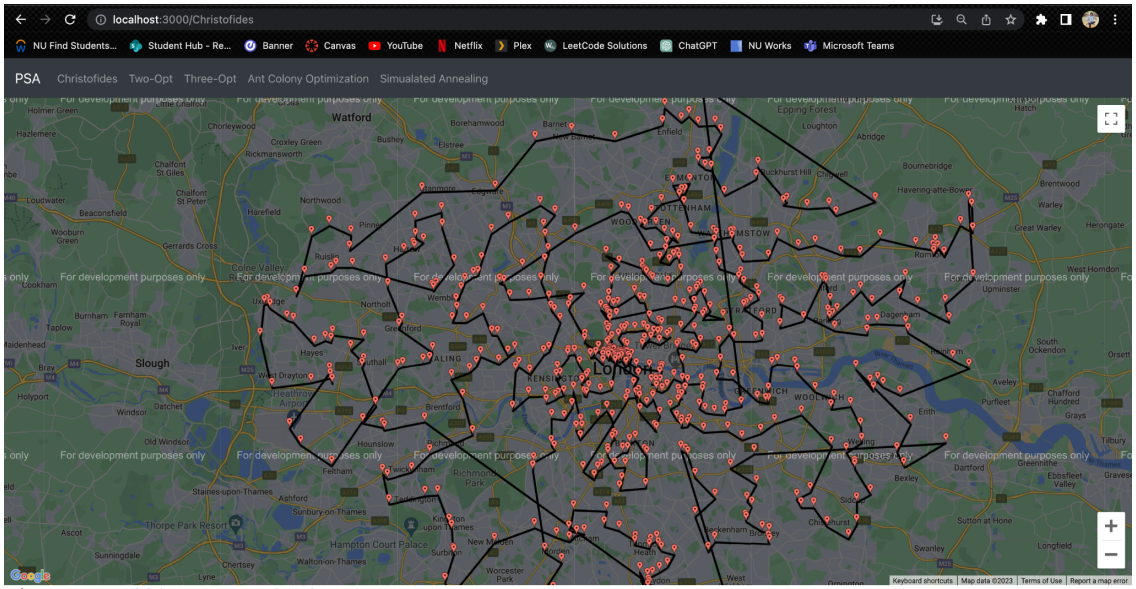
Distance Matrix for ACO:



Distance :673525.3229393851

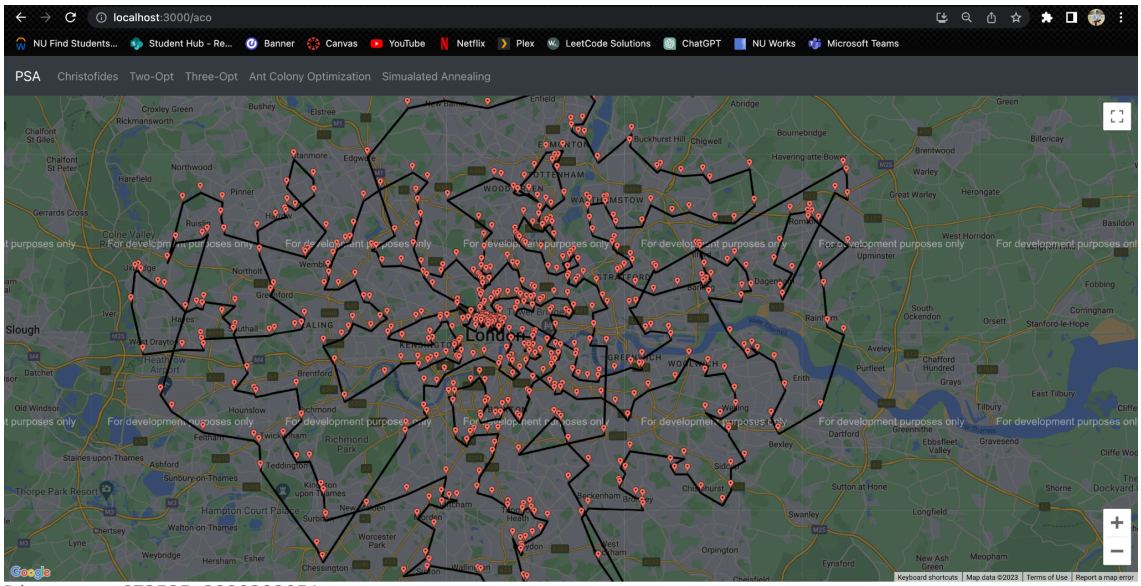
#	Crime ID	Lat	Long
482	d399d09ca411c382223266b81b996bf604b077380ced6369dd0eaa315c895134	51.535076	-0.189556
534	e72856078d6b086aab599d95e84a21a1eb7a63d4d77c424778bafeaa27eba3d9	51.540267	-0.195551
6	6f98975ecfe326d5e13a691623426c80d43b20470dc2d15e62e192fe0567ce6	51.542493	-0.198751
271	7cc2c259c12d406e1feb9fa4bccc210186d9892873407824f043fd6ff8367275	51.548631	-0.200903
574	f93d765223b26b6145287537c91e12bc4e760bff475fe2e10be10d08f49ba8bd	51.559805	-0.208901
408	b22c788ff0c84a0b0af53c831cde61e9a5f32d2347ac25f5d034bcea68ec9b4d	51.571318	-0.204957
118	34db30939ab439eadec0a3cb8d6b7e1a3518cae87661695e4ca3dd94a5336c1d	51.577793	-0.201469
147	411ed269c947b00ec9c9d161d33b9187fb691cb164bbf99e37cd2b2d5e8e71b5	51.586493	-0.204648
434	bc1c9fa5e2e5735508523f78bacf6fdfad3c9f120056875e132591d877dd9536	51.59271	-0.206048
264	79653e4466600f4cbeb0af8659c1dc739d599036331f5d38b49d627bbc71512a	51.601123	-0.194685
417	b686e61d354a34319c1f930f7f411c1d3d1d7cc9a3a2e22a610c5078611ceab2	51.606807	-0.209406
194	55f8739f3c7d0ae37d5997f9e2c648034b3f2061258af27bcc7a6431997cfec3	51.596899	-0.24128
249	743a121c3c3031fddb330c2e8f6a62bfbd22c9e2b35573d63f288528dc9cce32	51.589555	-0.258056
266	7a86493b2d81292215706 92effe8c66fac5b93316b9a37bc8464750f5fd861e4	51.572221	-0.26804

Christofides Algorithm:



Distance :733855.7114543049

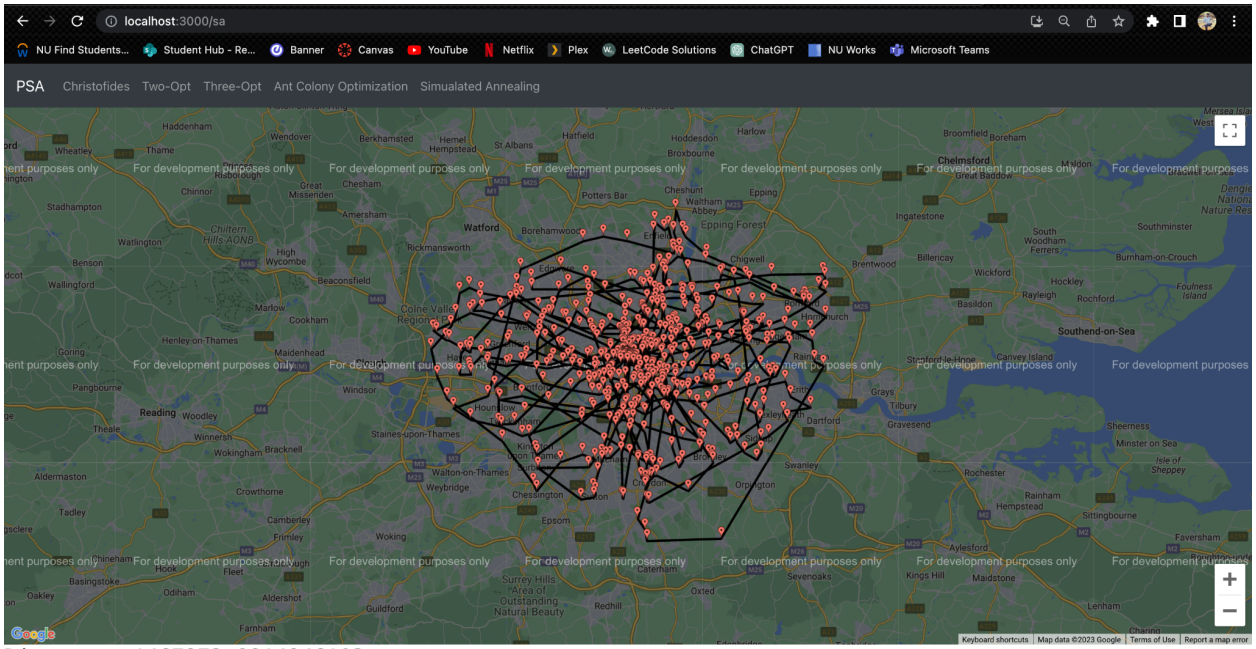
Ant Colony Optimisation




```
Distance :673525.3229393851
```

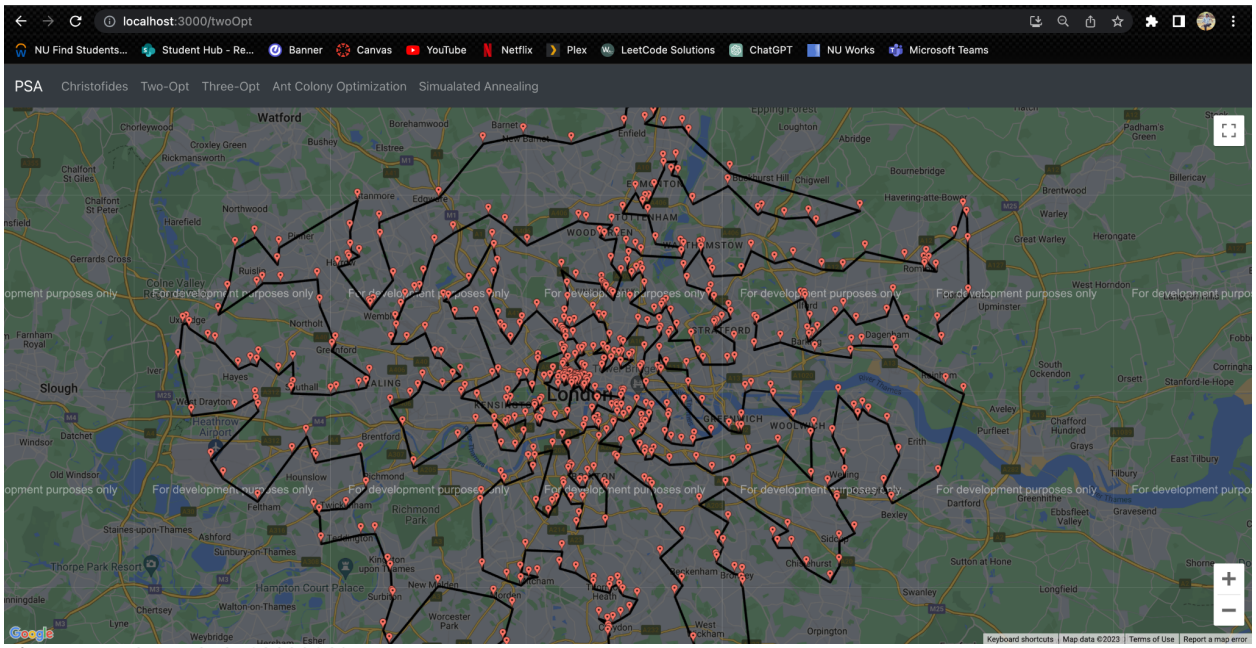
#	Crime ID	Lat	Long
482	d399d09ca411c382223266b81b996bf604b077380ced6369dd0eaa315c895134	51.535076	-0.189556

Simulated Annealing:




```
Distance :1467973.8814946103
```

Two opt




```
Distance :641553.3403932062
```

- **References**

- http://www.cs.cornell.edu/courses/cs681/2007fa/Handouts/christofides.pdf
- https://www.youtube.com/watch?v=wsEzZ4F_bS4&ab_channel=Rudidev
- https://towardsdatascience.com/around-the-world-in-90-414-kilometers-ce84c03b8552#:~:text=The%202%2Dopt%20algorithm%20works,route%20and%20repeats%20the%20steps
- https://medium.com/thelorry-product-tech-data/ant-colony-optimization-to-solve-the-travelling-salesman-problem-d19bd866546e
- https://www.youtube.com/watch?v=XUWm4ALFtyk&ab_channel=Dr.KuppusamyP