INFO 6205 Spring 2023 Project

Traveling Salesman Problem

Section-01

Team

Venkat Pavan Munaganti - 002722397

Ujjanth Arhan - 002108348

Raja Shekar Reddy Siriganagari - 002762145

Github Repository

https://github.com/siriganagari-raja-shekar/psa-final-project-spring2023

Best tour cost

605864.9369014174 meters

Table of Content

Introduction

Aim:

The Travelling Salesman Problem aims to find the shortest route for traveling between cities where you start at a city and visit all other cities exactly once and return back to the starting point. This is an NP-hard problem in computer science and no algorithm has been found to compute an optimal solution for this problem in polynomial time. Our aim is to find an approximate solution for the TSP problem.

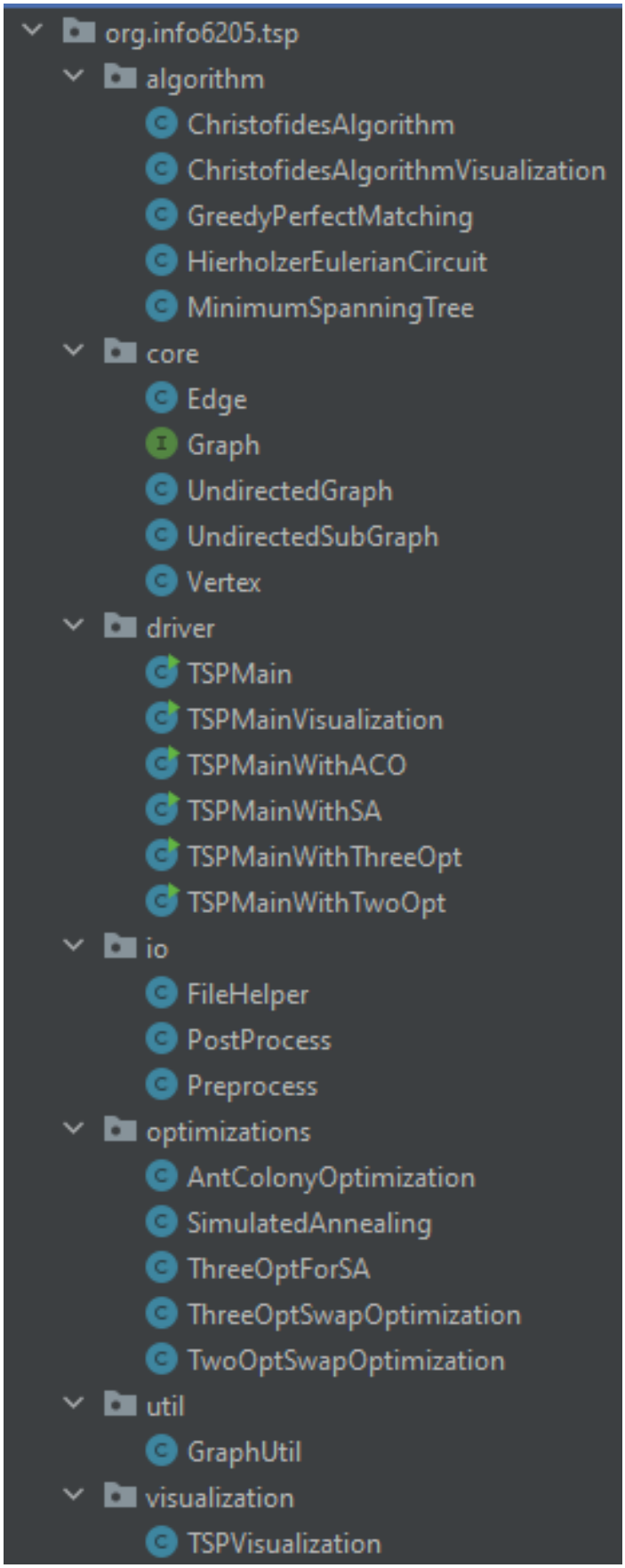
Approach:

We are using the Christofides Algorithm to approach the TSP problem. This gives us an approximate solution on which we are applying the following optimizations:

- 2-opt swap
- 3-opt swap
- Simulated Annealing
- Ant Colony Optimization

Data Structures and classes:

The Christofides Algorithm requires various implementations of a graph at different stages of the algorithm (like undirected weighted graphs, multigraphs). We are primarily working with undirected graphs in this problem and so we have wrapped all our implementations of the graph data structure into an undirected graph implementation for ease of use. An adjacency list is used to emulate a graph. The following list of classes are used as part of the project:


org.info6205.tsp
algorithm
ChristofidesAlgorithm
ChristofidesAlgorithmVisualization
GreedyPerfectMatching
HierholzerEulerianCircuit
MinimumSpanningTree
core
Edge
Graph
UndirectedGraph
UndirectedSubGraph
Vertex
driver
TSPMain
TSPMainVisualization
TSPMainWithACO
TSPMainWithSA
TSPMainWithThreeOpt
TSPMainWithTwoOpt
io
FileHelper
PostProcess
Preprocess
optimizations
AntColonyOptimization
SimulatedAnnealing
ThreeOptForSA
ThreeOptSwapOptimization
TwoOptSwapOptimization
util
GraphUtil
visualization
TSPVisualization

The *core* package consists of the data structures used as part of this project. The description of each of these APIs is as follows:

- **Graph API:** The base interface for any graph implementation. Contains useful methods required to achieve the current aim of the project.
- **Vertex:** Represents the node in a graph. Contains an id and the coordinates of the node(latitude and longitude in our case)
- **Edge:** Represents an edge in the graph connecting two vertices. This edge is **weighted** by default. Contains information about the source and destination vertices along with the weight of the edge. Has useful methods to calculate the weight depending on the coordinates of the vertices passed
- **UndirectedGraph:** Implements the Graph API and uses an adjacency list internally to represent the graph. The adjacency list is in the form of a map containing the mapping between vertices and their adjacent edges. This graph implementation assumes that if there is an edge between the source and the destination in the source vertex's adjacency list there is bound to be a corresponding edge from the destination to source in the destination vertex's adjacency list. This graph implementation also serves as a multigraph allowing duplicate edges between the same source and destination.
- **UndirectedSubGraph:** This graph implementation is useful when you want to extract a certain number of vertices from an original graph along with the edges corresponding to only those vertices. This is a useful implementation when you have to extract the odd degree vertices and their edges for the perfect matching step in our algorithm. This class extends the **UndirectedGraph** class.

The *algorithm* package consists of the implementation of the Christofides algorithm and different steps used as part of it. Following is a description of these classes:

- **MinimumSpanningTree:** Prim's minimum spanning tree implementation is used for this part. Takes in a graph input and returns a new graph representing the minimum spanning tree. It is guaranteed to be correct.
- **GreedyPerfectMatching:** This is a greedy implementation to find the perfect matching for a given graph. It **does not guarantee** an optimal solution and is theoretically proven to be at least twice than the cost of the optimal solution. We had to resort to this as implementing the minimum weight perfect matching using algorithms like Hopcroft-Karp has been difficult.
- **HeirholzerEulerianCircuit:** Implementation of the Hierholzer algorithm for finding out a Euler circuit from a multigraph. We may get different Euler circuit's each time depending upon the order in which the adjacent edges for the vertices are retrieved. Helps instill a randomness to the algorithm and encourages different solutions.

- **ChristofidesAlgorithm:** A driver class which takes in a graph input and passes it along to the above classes. It returns a final TSP tour after removing the duplicates from the Euler circuit.

The *optimizations* package consists of all strategic and tactical optimizations that we have applied for this project. The description of those classes is as follows:

- **TwoOptSwapOptimization:** This class implements the 2-opt swap algorithm. Takes in a TSP tour as input and returns an optimized tour as output. Essentially follows a brute force approach of switching edges but has some checks in place to improve its performance.
- **ThreeOptSwapOptimization:** This class implements the 3-opt swap algorithm. A brute force approach which takes longer to run than the 2-opt swap but is practically proven to provide better results than 2-opt.
- **ThreeOptSwapForSA:** A slightly modified implementation of the ThreeOptSwapOptimization for inclusion in the Simulated Annealing algorithm.
- **SimulatedAnnealing:** An implementation of the simulated annealing approach which internally uses the 3-opt swap. Takes much faster to run than the traditional 3-opt and is consistently proven to find better solutions in the search space by moving away from the local minima.
- **AntColonyOptimization:** A custom implementation of the ant colony simulation. This practically performs better than all the optimization algorithms present for the TSP. Our implementation takes a graph as an input and returns an optimized tour. This algorithm is a little slower than SimulatedAnnealing in terms of performance.

The *visualization* package contains classes that are aid in the simulation of the graph during different stages of the Christofides Algorithm. The description of the classes is as follows:

- **TSPVisualization:** Uses JUNG library's features to visualize the simulation. Different metrics and layouts are used to visualize different stages of the algorithm. A single class containing all the visualization related code

The *io* package contains classes that help in reading and converting datasets to required objects for the algorithm. The description of the classes are as follows:

- **FileHelper:** Contains useful functions for file I/O
- **Preprocess:** Responsible for converting the input dataset to a Graph object
- **Postprocess:** Responsible for converting the final TSP tour to the exact form as the initial input dataset. Writes the final tour to a new .csv file.

The *util* package has classes that contain static methods used by various classes across the project. List of these classes are:

- **GraphUtil:** Contains useful static methods to calculate tour distance, distance between vertices, converting graphs to required formats for visualization etc.

The *driver* package contains the main driver classes for the whole project. The description of the classes is as follows:

- **TSPMain:** Driver class for running Christofides algorithm for a set number of times and giving the best tour out of it,
- **TSPMainWithTwoOpt:** Driver class for running Christofides algorithm and sending each tour to optimization to the **TwoOptSwapOptimization** class for a set number of times and giving the best tour out of it.
- **TSPMainWithThreeOpt:** Driver class for running Christofides algorithm and sending each tour to optimization to the **ThreeOptSwapOptimization** class for a set number of times and giving the best tour out of it.
- **TSPMainWithSA:** Driver class for running Christofides algorithm and sending each tour to optimization to the **SimulatedAnnealing** class for a set number of times and giving the best tour out of it. **This is our main driver class and gives us the best results in the least amount of time. Run this class to get the final result. If it is taking too long decrease the number of iterations to 10. The total time to run 15 iterations is approximately 15min.**
- **TSPMainWithACO:** Driver class for running Christofides algorithm and sending each tour to optimization to the **AntColonyOptimization** class for a set number of times and giving the best tour out of it.

Christofides Algorithm

The Christofides algorithm builds on the idea that the best tour is closer to the minimum spanning tree and contains most of the edges that are included in the minimum spanning tree.

The steps involved in the algorithm are as follows:

1. Find out a minimum spanning tree(MST) for the given graph
2. Extract all the odd degree vertices in the minimum spanning tree
3. Find out a perfect matching for all these odd degree vertices
4. Add all the edges in the perfect matching to the MST to form a multigraph
5. Find a Euler tour starting from any source vertex in the multigraph
6. Convert the Euler tour to a TSP tour by removing the duplicate vertices

Let's take a deep dive into each of these steps and the algorithms used to implement them and later we'll come back to the analysis of Christofides Algorithm.

Prim's algorithm for Minimum Spanning Tree(MST):

The Prim's algorithm is the most efficient algorithm to find out the minimum spanning tree for a dense graph and is theoretically proven that it gives the most optimal tree. A minimum spanning tree here is a tree that contains all the vertices and the least number of edges required to include these vertices(i.e number of edges = number of vertices - 1)

The algorithm for Prim's modified for our implementation is as follows:

1. Start with a random arbitrary vertex 'v'
2. Add all vertices to the 'MST'
3. Add all the adjacent edges of the vertex 'v' to a priority queue 'Q' and add 'v' to the visited set.
4. Repeat until 'Q' is empty or all vertices are added to visited set:
    a. Remove min-weight edge 'e' from 'Q' and add 'e' to the MST
    b. If both the vertices in 'e' are visited skip the current iteration and continue
    c. Else add all adjacent edges of both 'e.source' and 'e.destination' to 'Q'
    d. Add both 'e.source' and 'e.destination' vertices to visited set
5. Return the 'MST'

This algorithm runs with a worst case time complexity of O(E log V). For completely dense graphs like our current graph the number of edges E = V^2 so the worst case time complexity is

equal to O(V log V). This is a much better approach for finding MSTs for a dense graph, whereas for sparse graphs it's better to use Kruskal's algorithm.

Greedy Perfect Matching:

A perfect matching for a graph is a set of edges such that each vertex in the graph is only included in one edge. As you can see this is possible only when the number of vertices is even as an edge contains two vertices. The aim is to find a perfect matching where the sum of weight of the edges is the minimum. Various algorithms exist that help us to find the minimum weight perfect matching in polynomial time. Some of them are:

1. Blossom Algorithm
2. Blossom V Algorithm
3. Hungarian Algorithm (Hopcroft-Karp Algorithm)
4. Maximum Weighted Bipartite Matching Algorithm

However these algorithms are complex to implement and require a thorough understanding of a lot of concepts to work with. For the sake of this project we are taking a more greedy approach and finding out a greedy perfect matching based on sorting the weights of all the edges in the graph. The algorithm that we used for our implementation is as follows:

1. Sort all the edges 'E' in the graph 'G'
2. Initialize a visited vertex set 'S' to empty and a perfect matching 'P'
3. Repeat until the 'S' contains all the vertices:
    a. Select an edge 'e' from 'E' in order of their weights
    b. If atleast one edge in 'e' is visited skip the current iteration
    c. Else add edge 'e' to the perfect matching 'P'
4. Return the perfect matching

This algorithm works with a time complexity of O(E log E) as the major chunk of time is taken to only sort all the edges. Although the perfect matching return is not the optimal, it turns out to be a very good starting point for us to continue with the Christofides Algorithm and its optimizations.

Hierholzer Eulerian Circuit Algorithm:

This algorithm follows a DFS approach to find a Euler circuit where each node of the graph is visited and each adjacent edge is traveled if the graph is visited until there are no more edges to visit. The algorithm implemented for our project is as follows:

1. Select any random start vertex ‘v’
2. Initialize an empty Euler circuit
3. DFS(‘v’):
    a. Add ‘v’ to the euler circuit
    b. For every adjacent edge ‘e’ for ‘v’:
        i. If ‘e’ is already visited skip the current iteration
        ii. Else DFS(‘edge.destination’)
4. Return the Euler circuit

At the end of the algorithm we have a list of vertices representing the Euler tour in order of the vertices visited. To get a TSP tour use the GraphUtil.getTSPTour() method which converts the Euler tour to a TSP tour by removing the duplicates.

Invariants:

There are many invariants while constructing a TSP tour from the original graph using the Christofides algorithm. These invariants can be used as tests to see if we are approaching the correct solution. Some of these invariants are:

- The graph is complete. Every edge is connected to every other edge in the graph. This is an important invariant as we can find the distance between vertices at any point in the algorithm without maintaining a reference to their edge weights which may be harder to retrieve
- The graph satisfies the triangle inequality i.e. for any three vertices A, B, and C in the graph, the distance between A and C is no greater than the sum of the distances between A and B and between B and C.
- The MST is guaranteed to have an even number of odd degree vertices irrespective of the start vertex and is guaranteed to be unique always.
- The total weight of edges from the perfect matching is guaranteed to be at most half the weight of the MST
- The resulting Eulerian graph should have all even degree vertices
- The resulting tour must be at most 1.5 times the cost of the optimal solution(which can’t be tested)

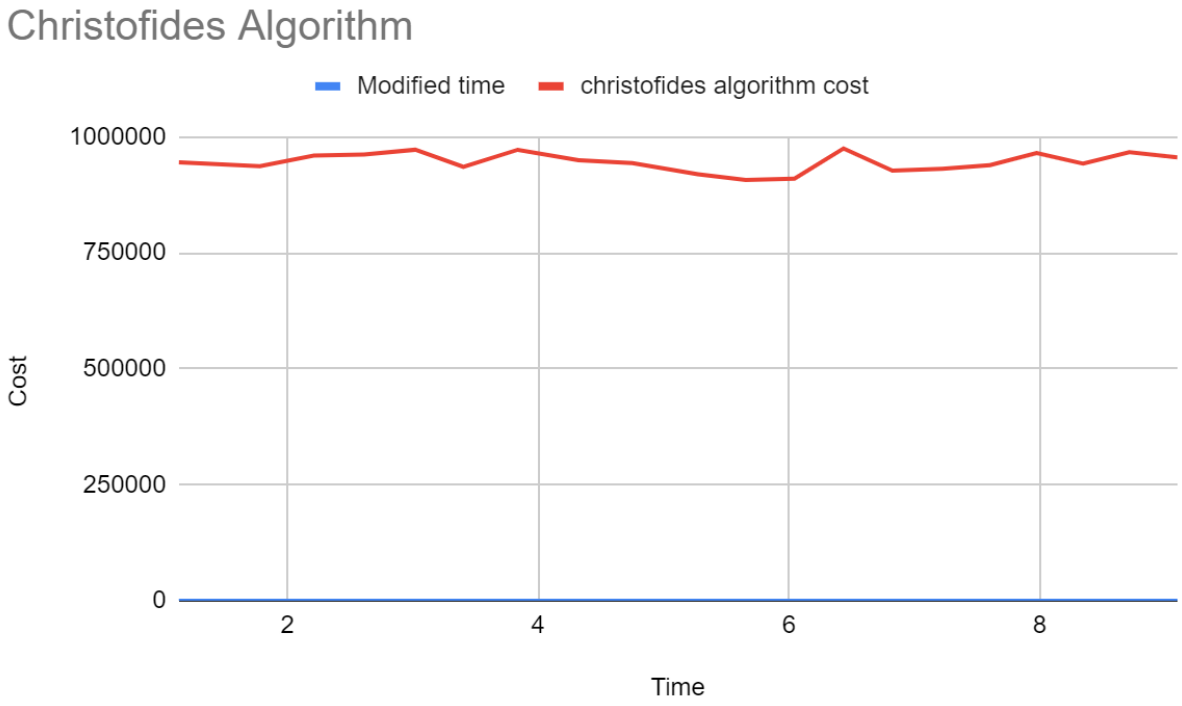
Results and Analysis of algorithms

Analysis of Christofides Algorithm:

- Starting with the MST we slowly build on to an approximate solution for the TSP. The MST is guaranteed to be the lower bound for the TSP as there is no other tree that is lower than the MST in terms of cost and the TSP tour has one more edge than the MST. So our goal is to achieve a tour whose cost is as close to the MST as possible.
- Once the MST is formed using the Prim's algorithm we have a tree now that contains all even degree vertices or an even number of odd degree vertices. It is theoretically proven that an MST is sure to have an even number of odd degree vertices if present. So these vertices will serve as a starting point for our next step
- All the odd degree vertices are extracted along with their edges by creating an UndirectedSubGraph and that is passed to the GreedyPerfectMatching class to get a perfect matching. Once you get the perfect matching we move on to adding all these edges to the MST.
- Until this point our algorithm is sure to give the same solution for the same dataset. There is no randomized factor that will change the solution for these steps and until this step all algorithms are proven to give an optimal solution.
- The next step of finding a Euler tour introduces a factor of randomization into the algorithm. We are starting from every vertex in the graph and are trying to generate a Euler tour by selecting adjacent vertices for the vertex. As we are using all the vertices as start points and the adjacency list stores the adjacent edges in a Set(which does not guarantee order) the tour configuration changes for each start vertex. The best tour is saved after each iteration and returned in the end.
- Time complexity at each step is:
    - Minimum spanning tree: O(V log V)
    - Greedy Perfect Matching: O(E log E)
    - Hierholzer Euler Tour: O(E)
- The total time complexity amounts to O(E log E) for the current implementation of this algorithm. This is not the most efficient implementation of Christofides as the approximation factor for the algorithm is shown to be 3/2 which makes it at most 1.5 times the optimal solution. But as we haven't implemented the minimum weight perfect matching the approximate solution is a little more than that.
- The problem with the TSP is that we can't test how good our algorithm is as we don't know the optimal solution for the given problem with large number of vertices
- Better algorithms like the Hungarian Algorithm for perfect matching can be used which would increase the total time complexity of the algorithm to O(E log E + V^3) but will give us a much better solution than the current one.

- With the current implementation the algorithm was run a 100 times for the *teamprojectfinal.csv* it produced a best tour costing 865383 which is ~1.68 times the Minimum Spanning tree which might be very close to ~1.5 times the optimal solution.

Here is a graph of the algorithm run for a particular number of iterations:



Optimizations

Tactical Optimizations:

Two-opt swap:

The 2-opt swap is a local optimization algorithm used to improve the tour cost of TSP by running a brute force approach and swapping edges to decrease the cost of the tour. This is a very fast and powerful algorithm in polynomial time and is consistently shown to improve the tour obtained from a regular Christofides algorithm run. Some of the characteristics of 2-opt swap observed as part of this project are:

- The 2-opt swap improved the quality of the TSP tour by removing the crossing edges in a tour and replacing them with non-crossing edges. By doing so, the algorithm shortened the tour length and reduced the total distance traveled.
- It is computationally efficient and can be used to improve the quality of the TSP tour in a short amount of time. In fact, the algorithm has a time complexity of O(n^2), where n is the number of vertices in the TSP tour.
- The 2-opt swap is a local optimization algorithm, meaning that it only considers the neighboring edges of a tour when optimizing it. Therefore, the algorithm is not guaranteed to find the global optimal solution of the TSP, but rather a locally optimal solution. So the algorithm consistently builds on the local solution that we got from the Christofides algorithm and can provide not so efficient solutions if it gets stuck in a local optimum generated from the initial TSP tour.
- Multiple iterations of the 2-opt swap algorithm have shown to improve the quality of the TSP tour further. The algorithm can be run multiple times, starting from different random initial tours or different starting points in the tour, to find a better locally optimal solution. For example, let's look at the results for the *teamprojectfinal.csv* for two-opt applied on 10 different runs of Christofides algorithms

Initial cost	Cost after 2-opt
949897	628165
912264	630960
942081	638629
960370	624781
928614	634487

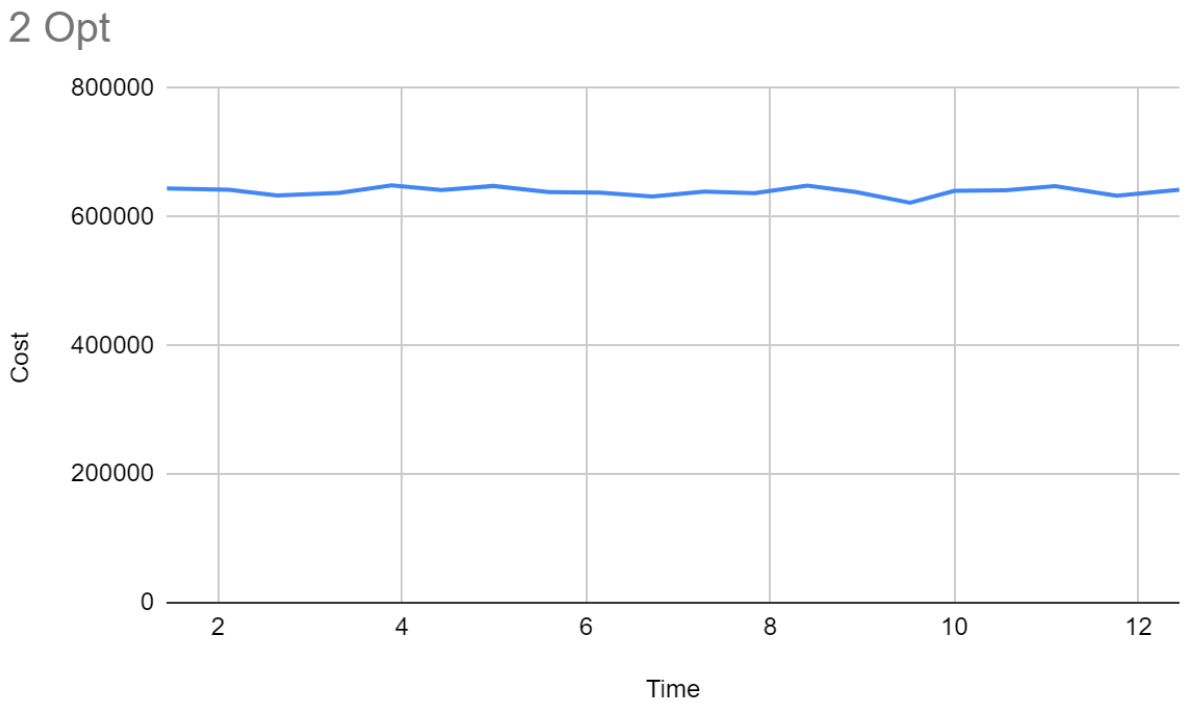
933005	635615
953934	644061
936709	645995
929780	636081
923222	624826

The total cost of the MST in this case is 513326 which makes the best tour **just 21%** above the MST cost which is a very good solution for a local optimization algorithm. We have used an efficient implementation of the 2-opt swap which switches edges only if it finds a better solution. The pseudo-code for the algorithm is as follows:

```
current_cost = calculateTotalDistance(existing_route)
repeat until no improvement is made {
    improvementFound = false
    start_again:
    for (i = 0; i <= number of nodes eligible to be swapped - 1; i++)
{
        for (j = i + 1; j <= number of nodes eligible to be swapped;
j++) {
            v1=i, v2=i+1, v3=j, v4=j+1
            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])
            if (lengthDelta < -1.0) {
                new_route = do2OptSwap(existing_route,i,j)
                currentCost -= lengthDelta
                improvementFound=true
            }
        }
    }
}
```

Although the 2-opt swap is a very good algorithm it is not very consistent in finding good results and shows a lot of variations depending on the initial tour. For an improvement over the 2-opt we have implemented the 3-opt tour.

Here is a graph showing the 2-opt swap optimization costs over a certain number of iterations:



Three-opt swap:

The 3-opt swap is an improvement over the 2-opt swap which considers three edges to swap instead of 2. As inferred it offers a better solution than 2-opt but takes much larger time to run.

Some of the characteristics of 3-opt are:

- 3-opt is also a local optimization algorithm which follows a brute-force approach similar to 2-opt. It applies small optimizations by selecting three edges from the TSP tour. Once the difference in cost is calculated depending on the swaps, if the cost is seen to be reduced the swaps are performed.
- As the 3-opt considers more moves than the 2-opt its search space is more wide than the 2-opt and provides consistently better results in a specific range.

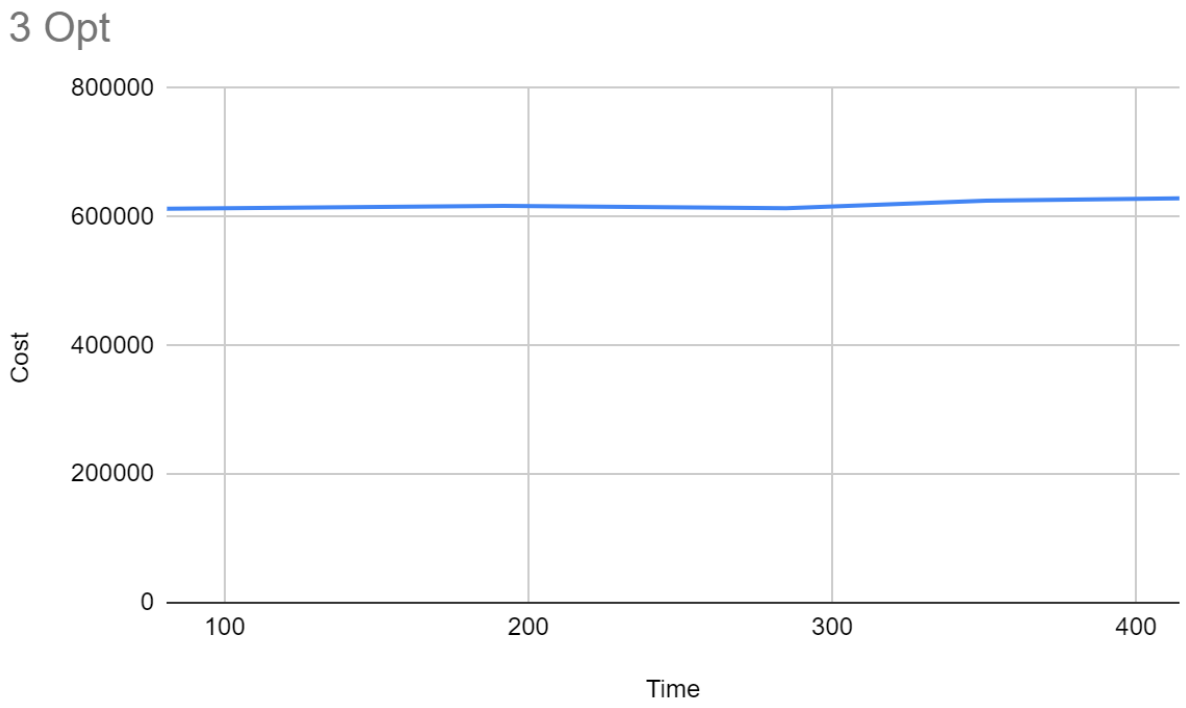
- Each swap in the 3-opt algorithm is potentially much larger than the 2-opt swap as it considers more edges. So you find significant improvement of the tour cost even for less number of swaps and iterations.
- 3-opt can sometimes escape the local optima too giving us a wider search space for the solution.
- In addition to improving existing tours 3-opt can also be used to generate initial tours by modifying the algorithm a bit.
- The time complexity of 3-opt is O(n^3) where n is the number of vertices.
- By itself 3-opt is very slow but can be wired into other branch-and-bound algorithms to improve the performance and find better results in a much lesser time.

Here are the results of the 3-opt swap on 10 initial tours from the Christofides algorithm:

Initial cost	**3-opt cost**
949846	623951
927462	630404
939799	629287
903482	623690
948004	621923
939809	623770
996173	628037
964788	632166
930943	627927
975683	628284

Although the 3-opt provides better solutions it took a huge amount of time to generate these solutions. On an average per run it took ~64 sec for the *teamprojectfinal.csv* dataset.
It's clear that it is better to run 3-opt in some strategic algorithms like Simulated Annealing and local search for Ant Colony Optimization. We have wired our current 3-opt implementation into Simulated Annealing which is helping us perform much faster and get better results. The algorithm is a modification of the code in this page.

Here is a graph showing 3-opt runs for a certain number of iterations:



Strategic Optimizations:

Simulated Annealing:

Simulated annealing is a stochastic optimization algorithm that is often used to solve combinatorial optimization problems. It is inspired by the annealing process in metallurgy, where a material is heated and then slowly cooled to a low temperature to remove defects and improve its properties.

The simulated annealing algorithm starts with an initial solution, which is typically generated randomly or using a heuristic. The algorithm then iteratively improves the solution by making small perturbations and accepting or rejecting the new solution based on a probability distribution that depends on the current temperature. The temperature decreases over time according to a predefined cooling schedule, which controls the balance between exploration and exploitation of the search space.

Pseudocode for Simulated Annealing:

1. Initialize number of iterations, initial temperature, and cool down rate (gamma)
2. Run the optimization until number of iterations are 0
3. For each iteration, calculate the new candidate solution

a. If the candidate solution is better than current best solution accept it
b. If it is not the better than current best solution, accept it with some random probability over negative exponent of cost delta divided by temperature

$$P(x \to x') = \begin{cases} 1, & if\ \Delta y \leq 0 \\ e^{\frac{-\Delta y}{T}}, & \Delta y > 0 \end{cases}$$

c. Reduce the temperature with the cool down rate or gamma
d. Repeat the process

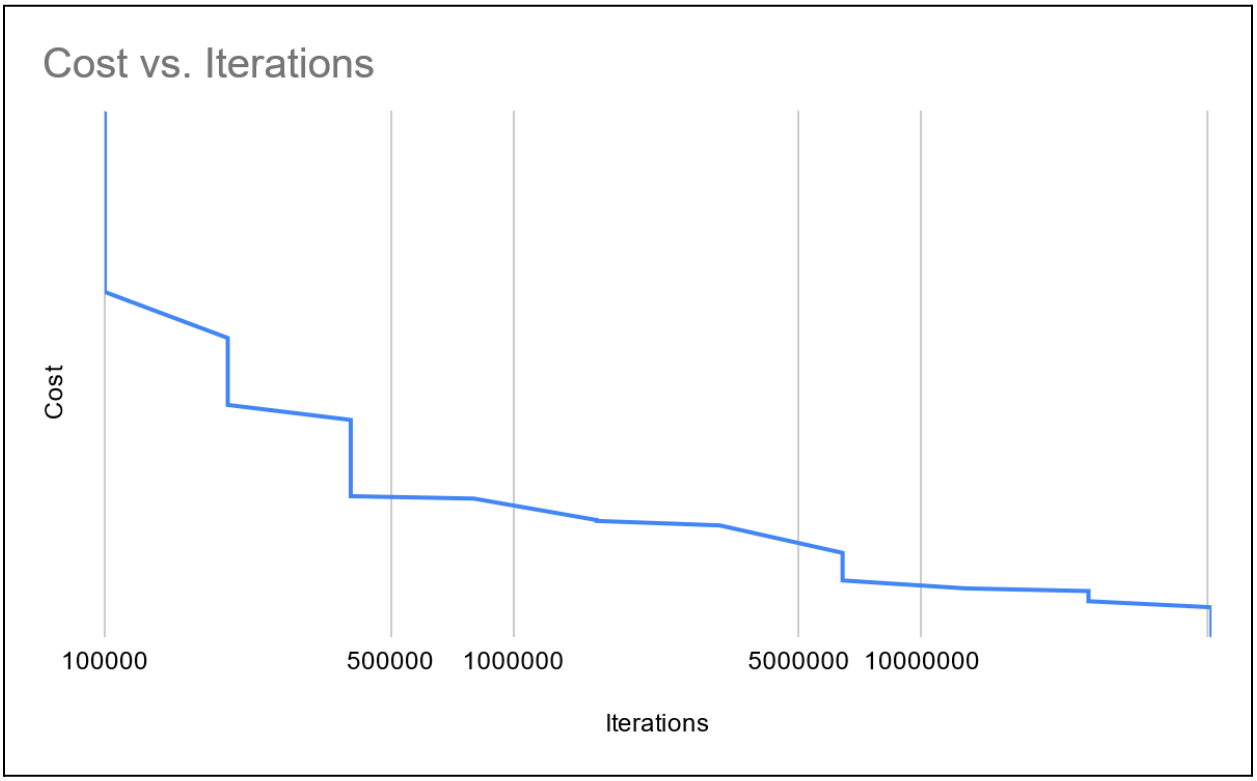
4. Return the best solution

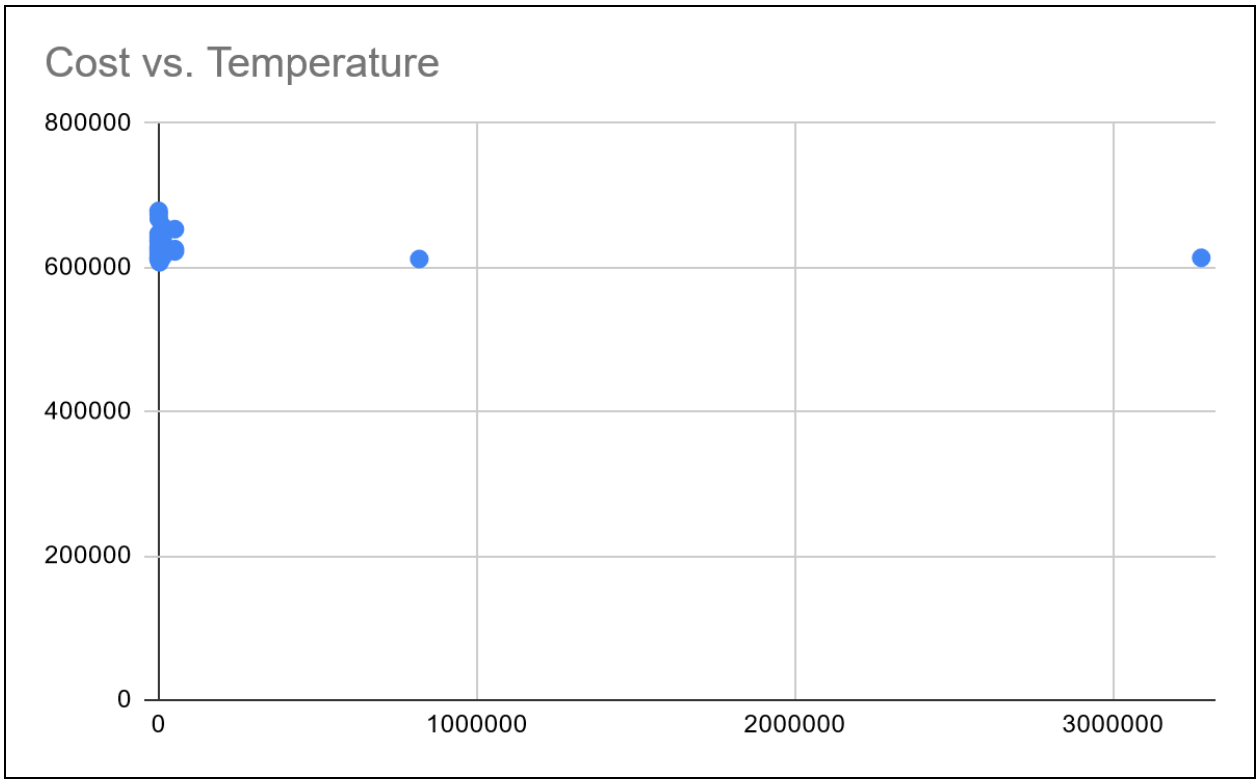
The performance of simulated annealing is heavily dependent on choice of parameters i.e, number of iterations, temperature and cooling down (gamma). The parameters for simulated annealing vary from dataset to dataset, and there are not standard parameters values that are suitable for all use cases. For this particular crime data set we have used a doubling method to find out the parameters that give the best results. Below are the observations and cost results in finding the optimal parameters,

1. The cost of the TSP tour is optimum for higher number of iterations
2. The cost is optimal when the temperature is low, this means that the search space of the problem is relatively well behaved allowing the temperature to converge quickly

Iterations	Temperature	Gamma	Cost
100000	100	0.1	678559
100000	400	0.1	674011
100000	800	0.1	667225
100000	12800	0.9	657069
100000	51200	0.9	653152
200000	100	0.1	646824
200000	100	0.5	641986
200000	400	0.5	639339
200000	800	0.9	637979
200000	12800	0.5	637731
400000	100	0.1	635711
400000	200	0.1	628411
400000	1600	0.5	626794
400000	51200	0.5	625538
800000	100	0.5	625222

1600000	800	0.1	622390
1600000	51200	0.1	622272
3200000	200	0.5	621687
6400000	200	0.1	618082
6400000	400	0.1	615495
6400000	12800	0.5	614489
12800000	3276800	0.9	613459
25600000	100	0.5	613110
25600000	819200	0.9	611794
51200000	100	0.9	611012
51200000	3200	0.5	607175





From the above findings and graphs, it is evident that the cost of the tour is decreasing when the number of iterations are high.
The scatter plot between the temperature and cost shows that the algorithm is performing better when the temperature values are low, which also means that the search space is well behaved and reaches global optimum really quickly.

Improvements and final thoughts:

1. General simulated annealing works by randomly swapping two edges. In this project we have implemented simulated annealing and 3-opt swap in combination. This gave better results compared to general simulated annealing and 3-opt optimization in lesser runtime.
2. This simulation can be optimized more by trying out different parameters using more efficient parameter tuning techniques.
3. Overall, the simulated annealing optimization we implemented is giving better results in less times compared to other optimizations techniques. On average the optimization gives a tour with a cost around 610K to 620k meters, which is 18-20% more than the MST cost, in 44 to 50 seconds.

Ant Colony Optimization:

Ant colony optimization is an attempt at solving optimization problems by taking inspiration from ants. Ants deposit pheromone along their trail to food which influences other ants near it to take that trail. The stronger the trail, the more probability that the ants will use that route leading to an optimized path to the destination. The process works by using probability in choosing the next node to go to, from the source node instead of complete randomness or fixed formulae.

The process can be mainly divided into three parts:

a. Probability Matrix - The matrix of nodes holding probability values of that specific node being picked. This value is computed as a function of distance and reward (pheromone). The matrix is directly dependent on:
   i. The ratio of the inverse of the distance to the inverse of the sum of distance. Shorter distances are better than longer ones.
   ii. The reward or pheromone value of the corresponding node. The stronger the trail the better the path is.
   iii. The alpha value which indicates the magnitude to which the distance matrix influences the probability matrix.
   iv. The beta value which indicates the magnitude to which the reward matrix influences the probability matrix.

   This can be represented in the form of the formula:

$$P(u,\ v)\ =\ \frac{[D(u,v)^{-1}]^{\ \alpha}.[R(u,v)]^{\ \beta}}{\Sigma_{\ j\epsilon adj(u)}\ [D(u,j)^{-1}]^{\ \alpha}.[R(u,j)]^{\ \beta}}$$

   Where P is the probability matrix, D is the distance matrix holding the distance from each vertex to every other vertex. R is the reward matrix holding the pheromone trail from each vertex to every other vertex. α is the value determining the magnitude that the distance on the probability matrix has. β is the value determining the magnitude that the reward matrix has on the probability matrix.

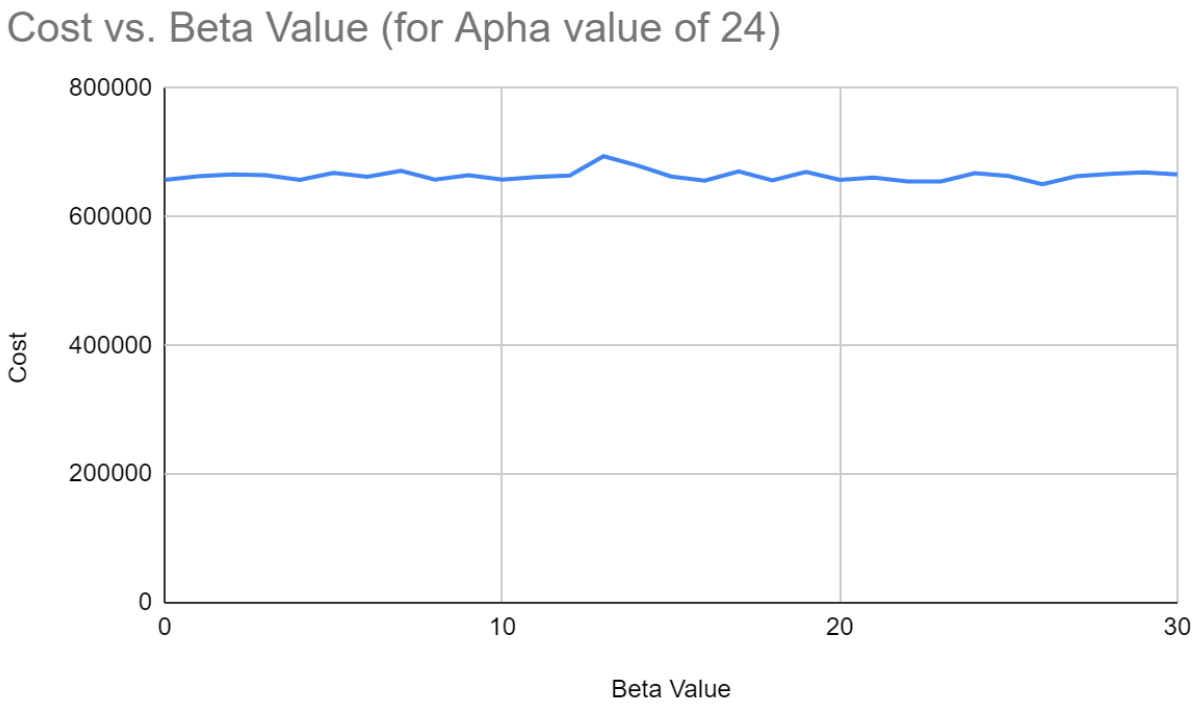
b. Reward Matrix or Pheromone Matrix - This matrix mimics the pheromone trail left by ants. Pheromone trail is the path taken by the ant from the source to the destination. This is being represented by the edges of the matrix. The stronger the pheromone trail, the better the path - is the consideration. This plays a role in calculation of the probability matrix. For each path that the ant takes, the reward for that trail is updated as an inverse to the total distance of the trail, which ensures that shorter paths are given

higher rewards than longer paths.

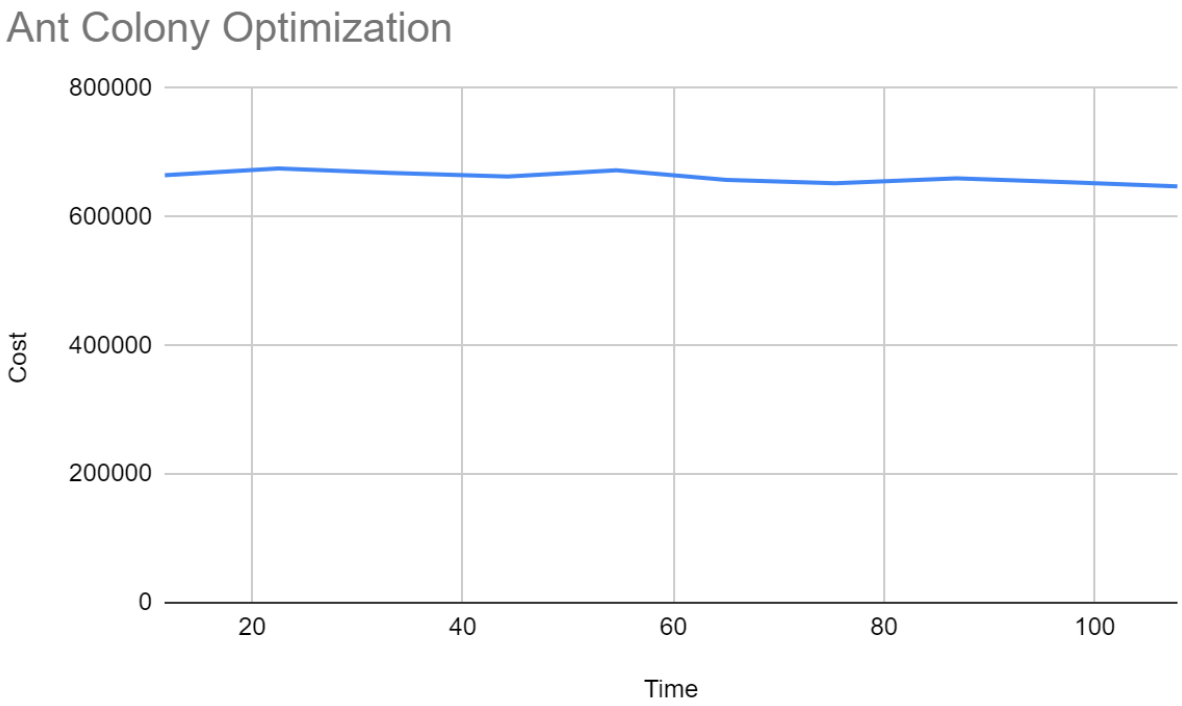
$$R(u,\ v)\ <-\ R(u,\ v)\ +\ \frac{1}{tour\ cost}\ if\ (u,\ v)\ \epsilon\ tour$$

c. Evaporation or decay - If the entire process of the optimization is repeated for a large number of iterations, the pheromone trail could become overtly biased to the paths that have already been explored, preventing new paths from being explored. To prevent this, we ensure the reward values evaporate or reduce to increase the probability of searching newer paths. The magnitude to which the decay occurs is set by a sort of trial and error process. The value we settled on is a constant value times the difference in cost of pheromone trail of the previous trail to the current one.
d. Local Optimization - Once a trail has been found, that trail is locally optimized by using 2 opt to get a better value which is then considered as the trail. This trail is then used to update the reward matrix instead.

After testing all the combinations of alpha and beta for a range from 0 to 30. The values we settled for were an alpha value of 24 and beta value of 25.



Here is a graph showing the ant colony optimization run for a particular number of iterations:


Ant Colony Optimization
800000
600000
400000
200000
0
Cost
20
40
60
80
100
Time

Visualization

The visualization for TSP is developed using the Java JUNG library. JUNG is a library useful for modeling, Analyzing and Visualizing the graph data structures. The visualization part of the JUNG library is based on Java Swing GUI.

The visualization of the TSP is divided into steps in Swing GUI,

1. Generation of MST
2. Highlighting the Odd degree vertices
3. Highlighting the edges after the greedy perfect matching
4. Visualizing the Eulerian tour
5. Visualizing the TSP tour
6. Visualizing the simulated annealing tour

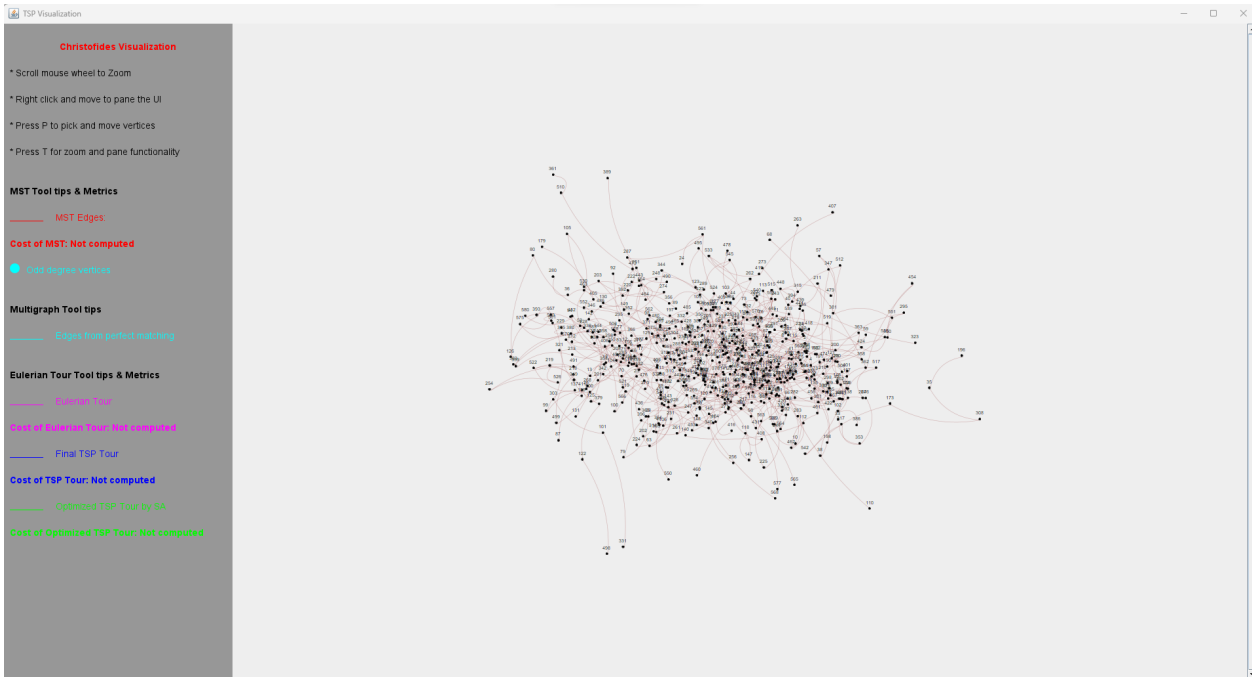
Since the data set is big and hard to plot on a GUI, we have used zoom and pane functionalities from JUNG library to better navigate the graph. We used a layout technique called Kamada-Kawai layout algorithm from JUNG library, which self organizes the graph as new edges and nodes are getting added. This layouting technique may take some additional time to organize the UI but the end result is better to navigate and identify the tour easily.

The GUI also contains an overlay panel which contains the information about all the vertex colors and edge colors to identify and inform the process of the algorithm. The cost of the MST, Eulerian tour, TSP tour and optimized TSP tour is also added to the panel as the algorithm progresses.

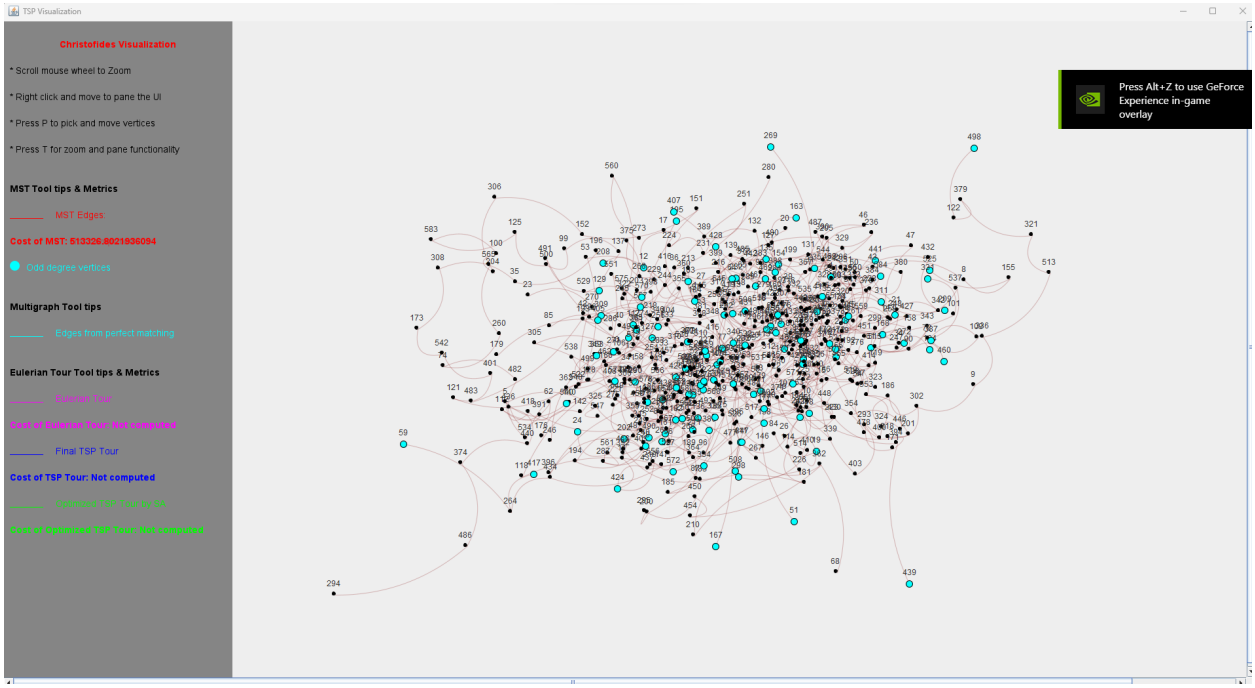
The entry point to the visualization is available in *org.info6250.tsp.driver* package. `TSPMainVisualization` class contains the code to launch and trigger the GUI visualization. The GUI code is present in the `TSPVisualization` class under *org.info 6250.tsp.visualization* package.

Snapshots:

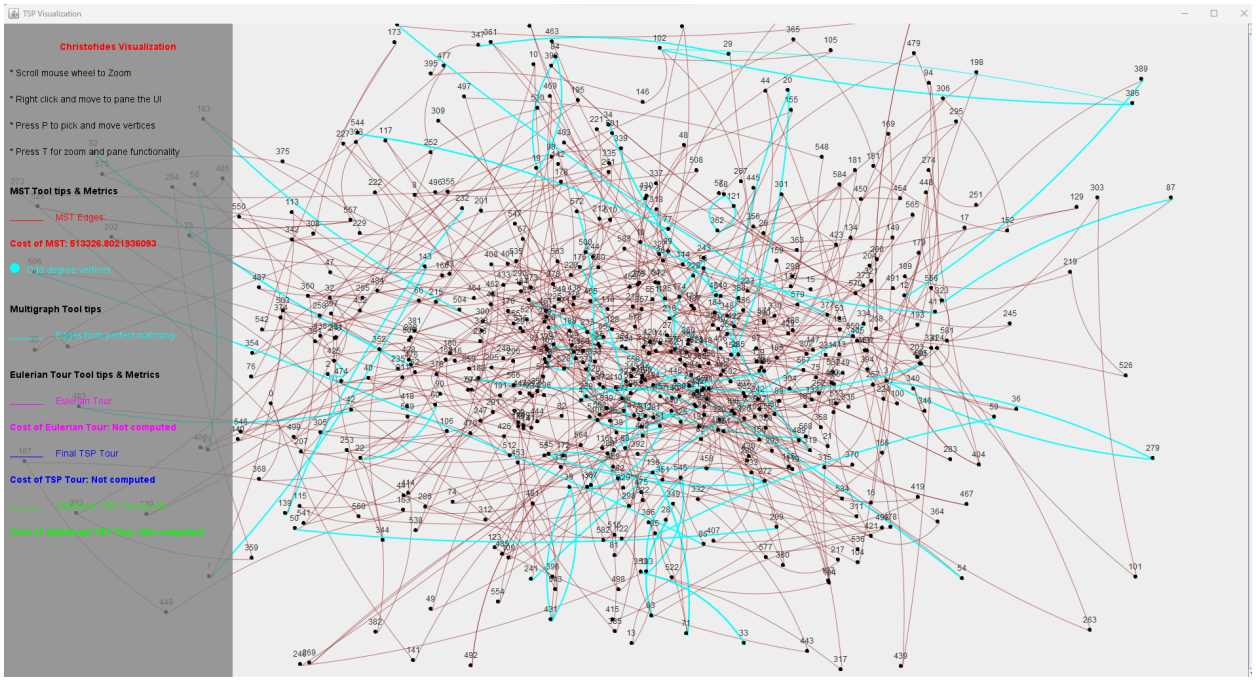
1. MST visualization:



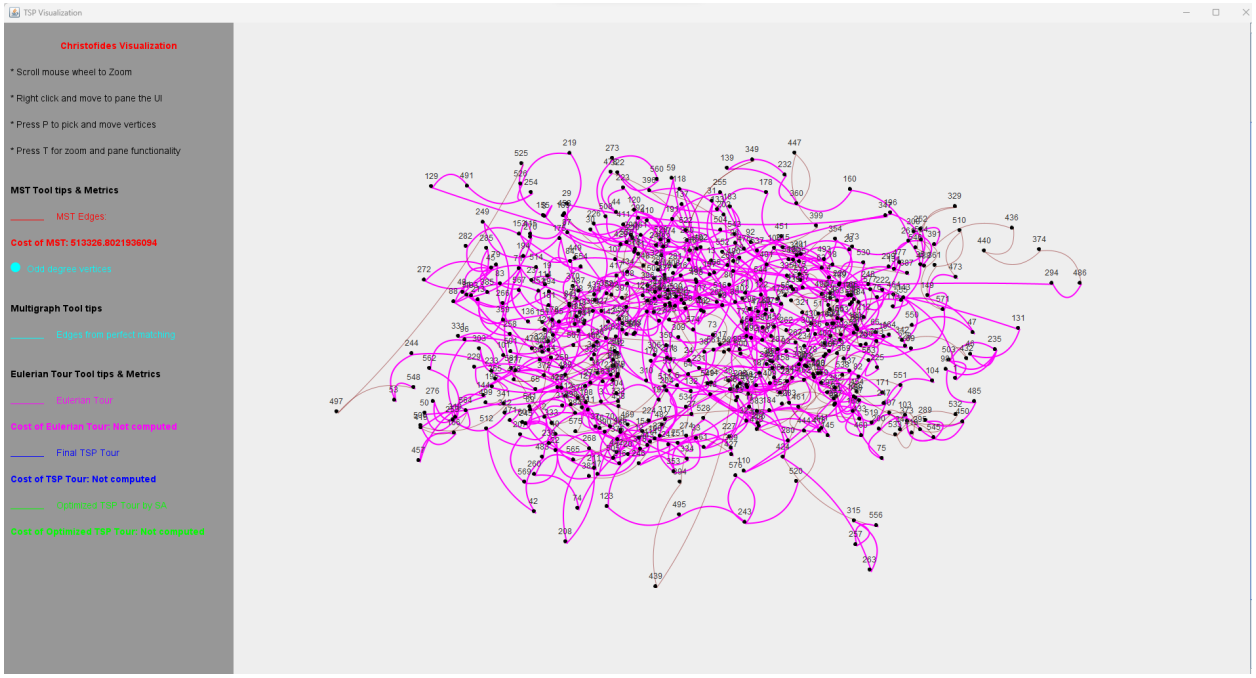
2. Odd degree vertices:



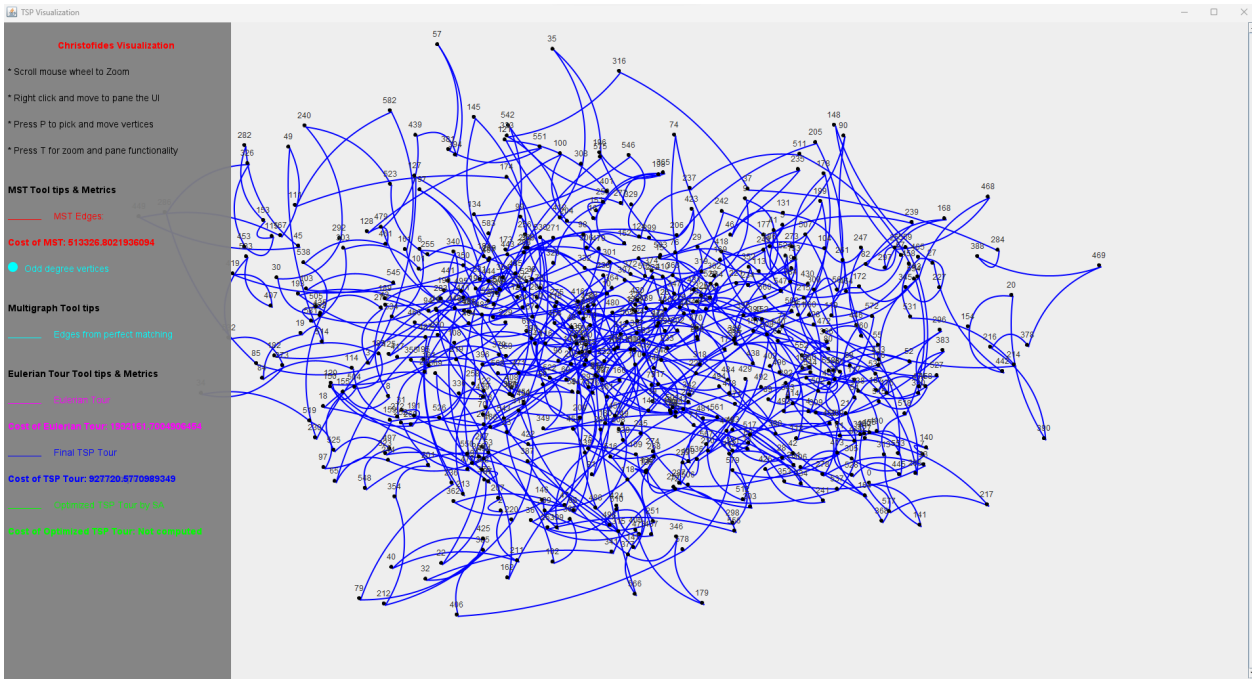
3. Edges after greedy perfect matching:



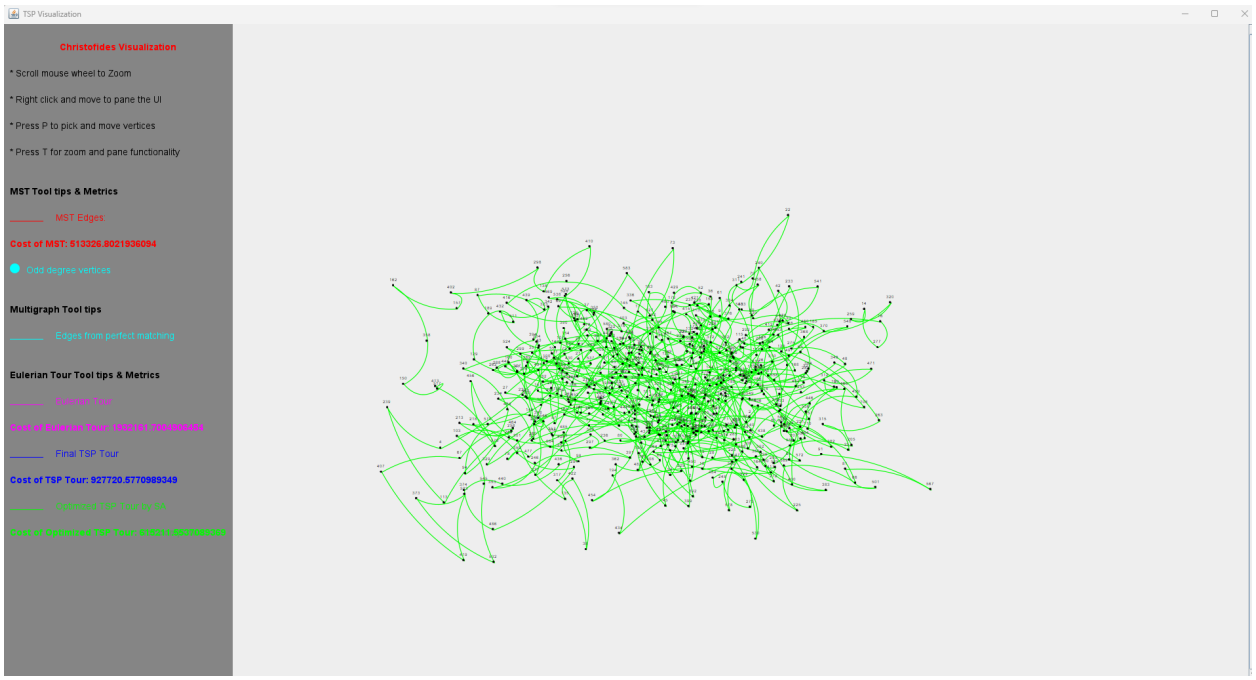
4. Eulerian tour:



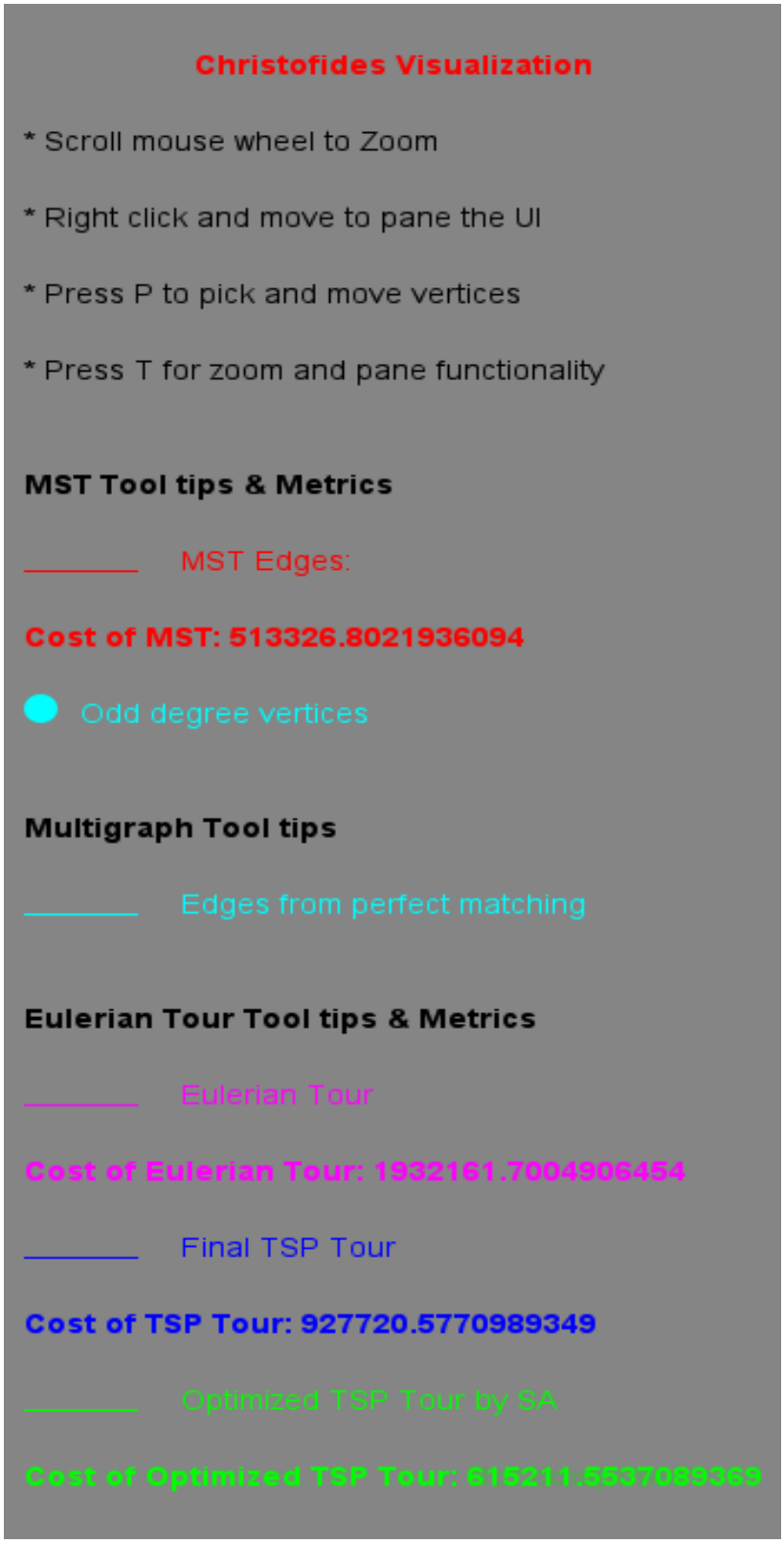
5. TSP Tour:



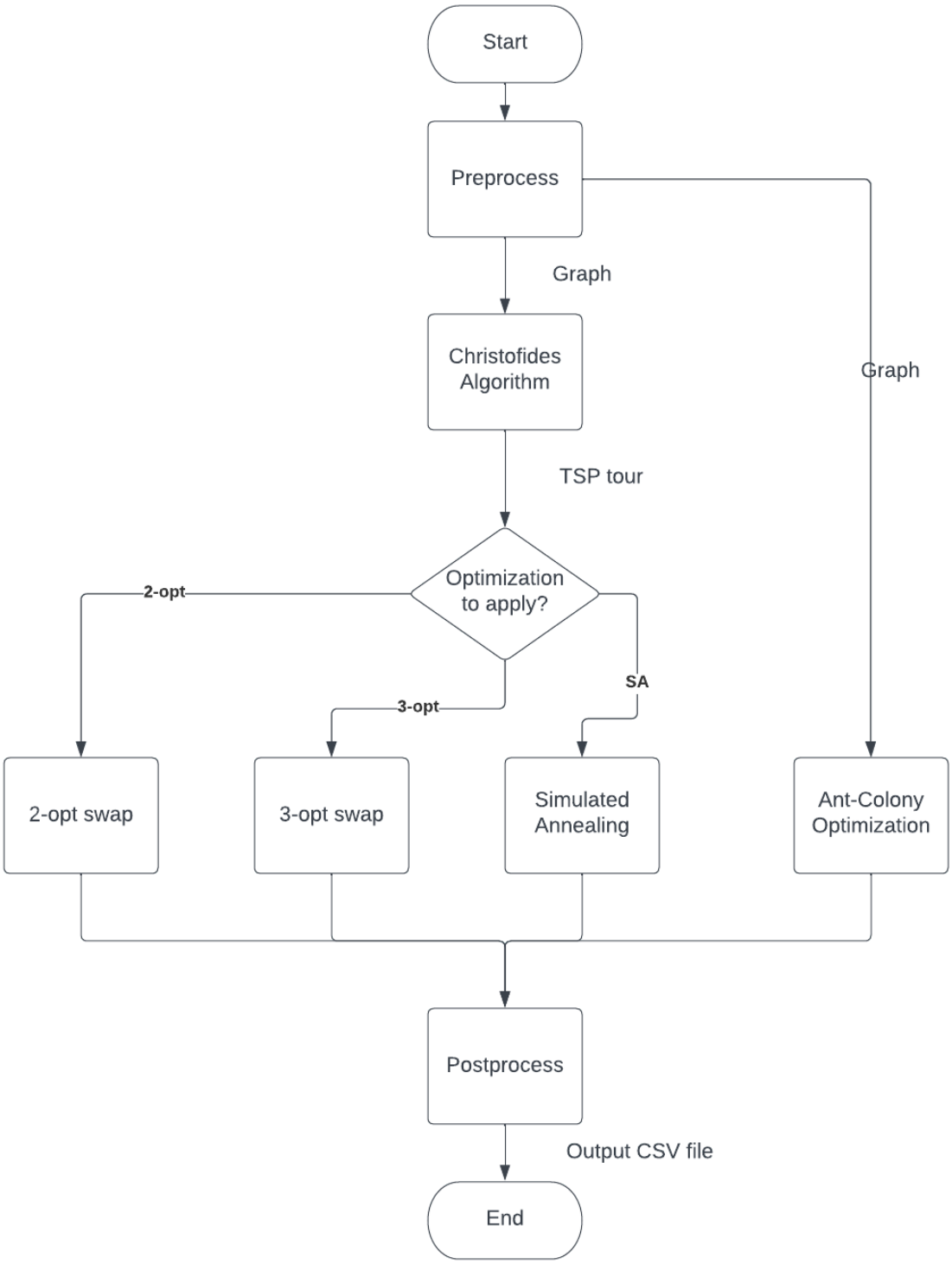
6. TSP tour after applying simulated annealing optimization:

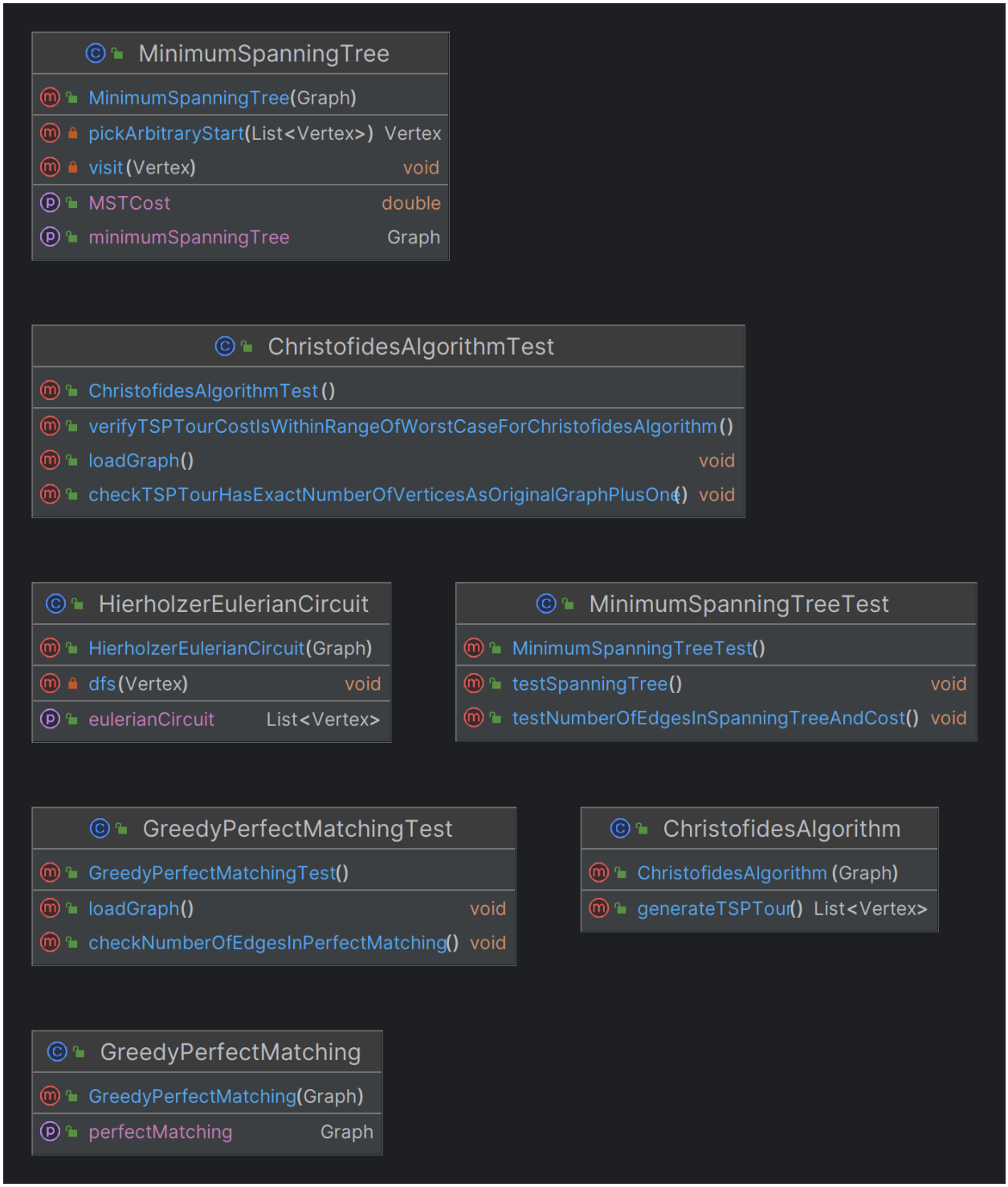


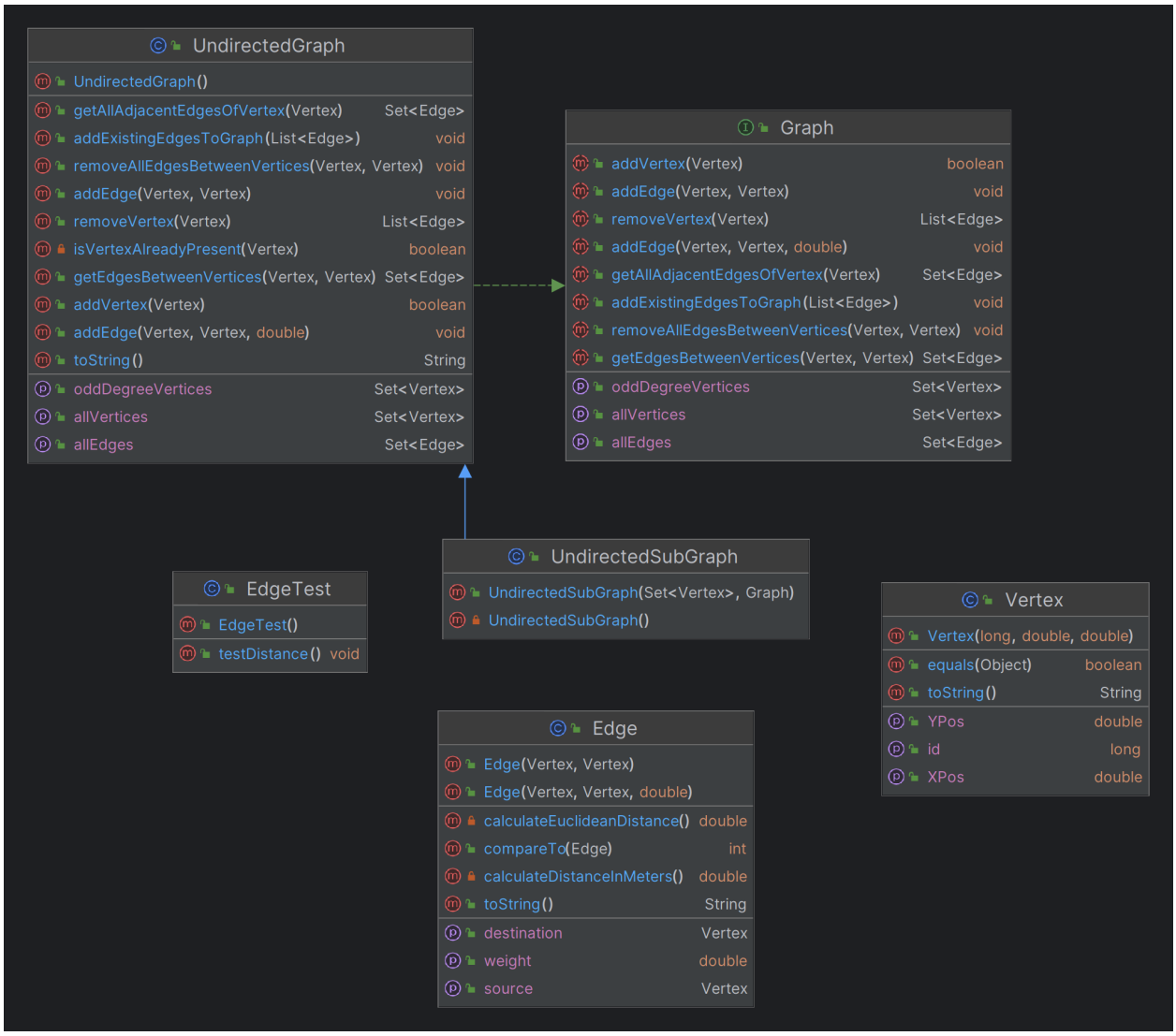
7. Metrics and tooltips overlay on left side of the GUI

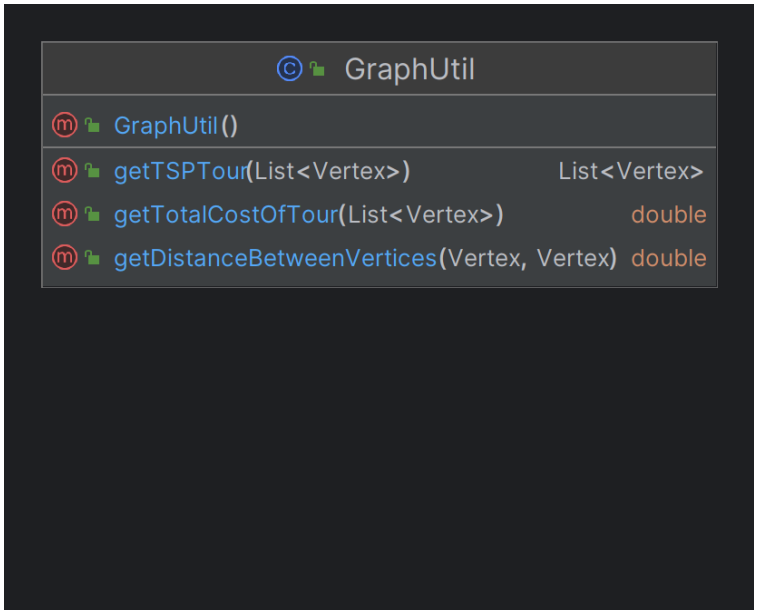


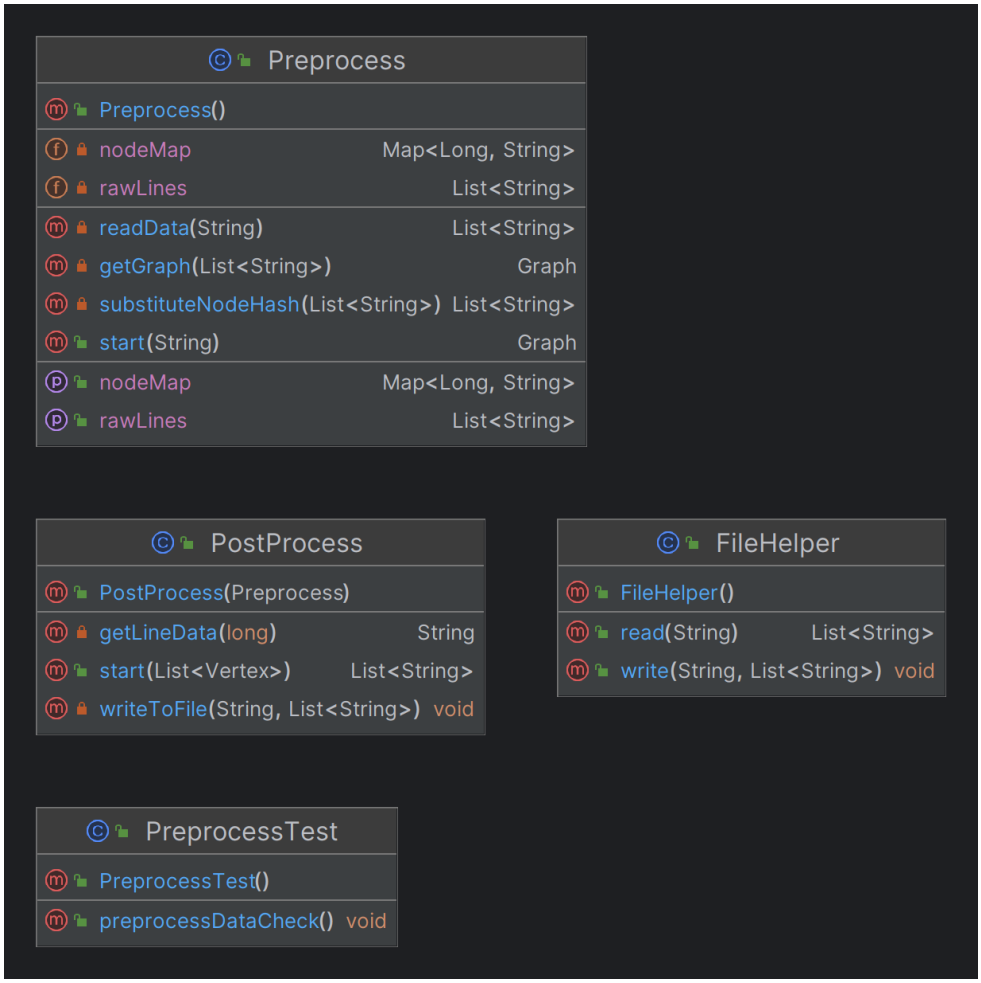
Class Diagrams and Flowcharts

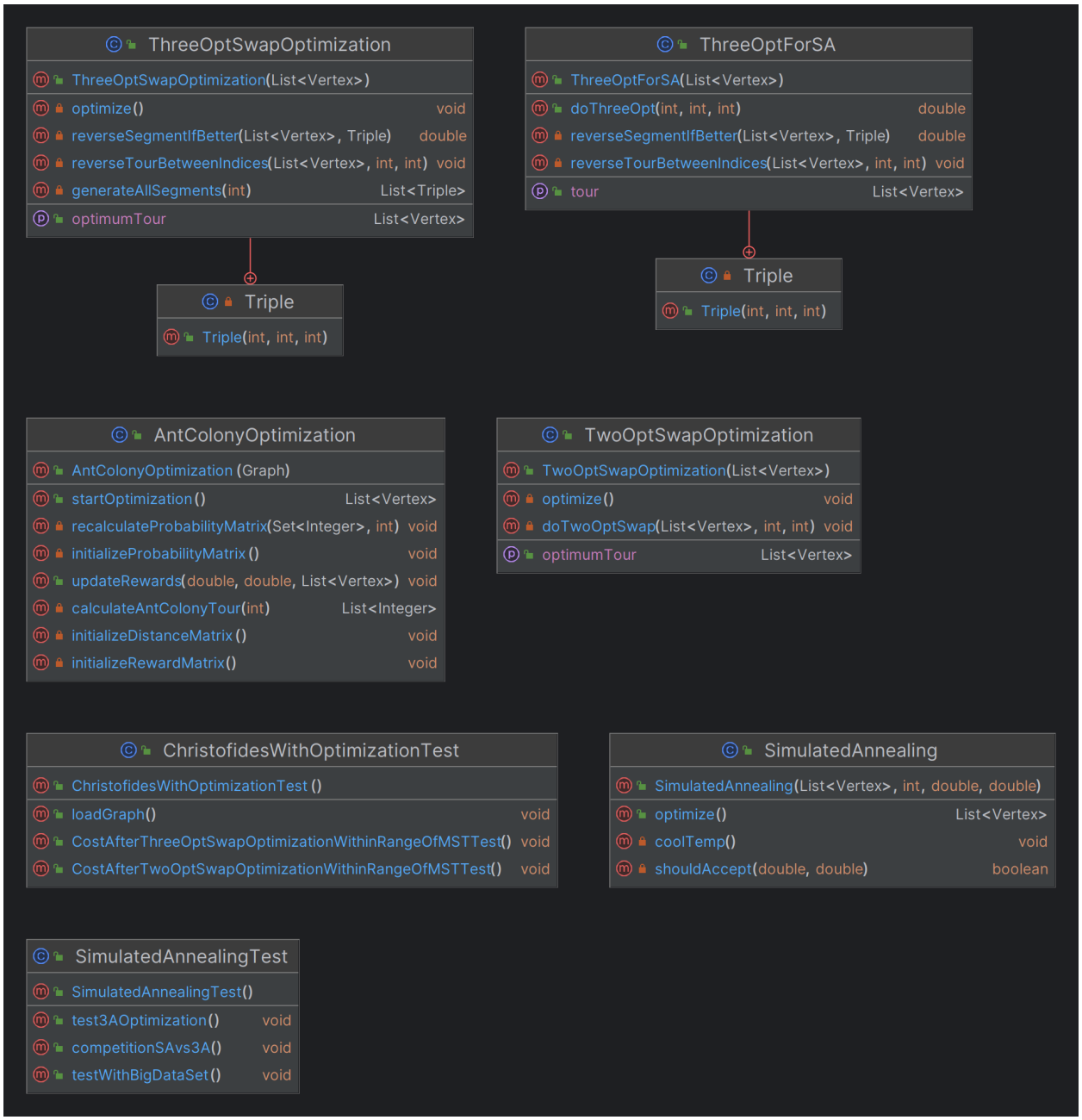



MinimumSpanningTree
MinimumSpanningTree(Graph)
pickArbitraryStart(List<Vertex>) Vertex
visit(Vertex) void
MSTCost double
minimumSpanningTree Graph
ChristofidesAlgorithmTest
ChristofidesAlgorithmTest()
verifyTSPTourCostIsWithinRangeOfWorstCaseForChristofidesAlgorithm()
loadGraph() void
checkTSPTourHasExactNumberOfVerticesAsOriginalGraphPlusOne() void
HierholzerEulerianCircuit
HierholzerEulerianCircuit(Graph)
dfs(Vertex) void
eulerianCircuit List<Vertex>
MinimumSpanningTreeTest
MinimumSpanningTreeTest()
testSpanningTree() void
testNumberOfEdgesInSpanningTreeAndCost() void
GreedyPerfectMatchingTest
GreedyPerfectMatchingTest()
loadGraph() void
checkNumberOfEdgesInPerfectMatching() void
ChristofidesAlgorithm
ChristofidesAlgorithm(Graph)
generateTSPTour() List<Vertex>
GreedyPerfectMatching
GreedyPerfectMatching(Graph)
perfectMatching Graph


UndirectedGraph
UndirectedGraph()
getAllAdjacentEdgesOfVertex(Vertex) Set<Edge>
addExistingEdgesToGraph(List<Edge>) void
removeAllEdgesBetweenVertices(Vertex, Vertex) void
addEdge(Vertex, Vertex) void
removeVertex(Vertex) List<Edge>
isVertexAlreadyPresent(Vertex) boolean
getEdgesBetweenVertices(Vertex, Vertex) Set<Edge>
addVertex(Vertex) boolean
addEdge(Vertex, Vertex, double) void
toString() String
oddDegreeVertices Set<Vertex>
allVertices Set<Vertex>
allEdges Set<Edge>
Graph
addVertex(Vertex) boolean
addEdge(Vertex, Vertex) void
removeVertex(Vertex) List<Edge>
addEdge(Vertex, Vertex, double) void
getAllAdjacentEdgesOfVertex(Vertex) Set<Edge>
addExistingEdgesToGraph(List<Edge>) void
removeAllEdgesBetweenVertices(Vertex, Vertex) void
getEdgesBetweenVertices(Vertex, Vertex) Set<Edge>
oddDegreeVertices Set<Vertex>
allVertices Set<Vertex>
allEdges Set<Edge>
UndirectedSubGraph
UndirectedSubGraph(Set<Vertex>, Graph)
UndirectedSubGraph()
EdgeTest
EdgeTest()
testDistance() void
Vertex
Vertex(long, double, double)
equals(Object) boolean
toString() String
YPos double
id long
XPos double
Edge
Edge(Vertex, Vertex)
Edge(Vertex, Vertex, double)
calculateEuclideanDistance() double
compareTo(Edge) int
calculateDistanceInMeters() double
toString() String
destination Vertex
weight double
source Vertex


GraphUtil
GraphUtil()
getTSPTour(List<Vertex>) List<Vertex>
getTotalCostOfTour(List<Vertex>) double
getDistanceBetweenVertices(Vertex, Vertex) double

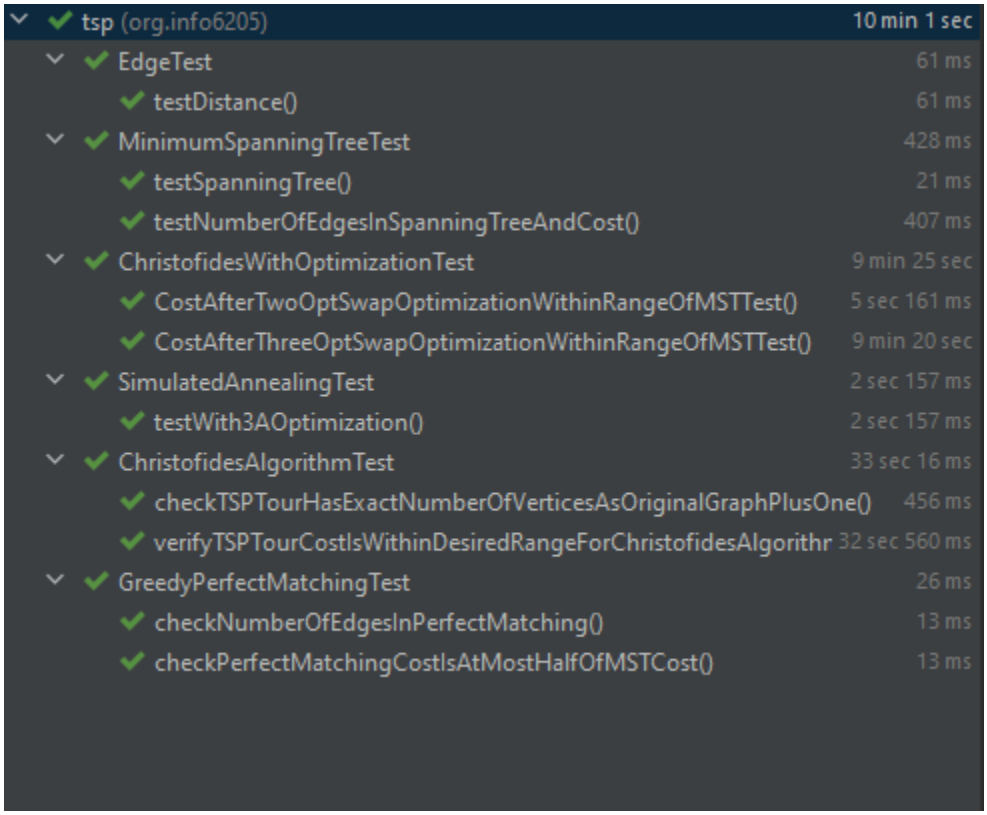

Preprocess
Preprocess()
nodeMap Map<Long, String>
rawLines List<String>
readData(String) List<String>
getGraph(List<String>) Graph
substituteNodeHash(List<String>) List<String>
start(String) Graph
nodeMap Map<Long, String>
rawLines List<String>
PostProcess
PostProcess(Preprocess)
getLineData(long) String
start(List<Vertex>) List<String>
writeToFile(String, List<String>) void
FileHelper
FileHelper()
read(String) List<String>
write(String, List<String>) void
PreprocessTest
PreprocessTest()
preprocessDataCheck() void


ThreeOptSwapOptimization
ThreeOptSwapOptimization(List<Vertex>)
optimize() void
reverseSegmentIfBetter(List<Vertex>, Triple) double
reverseTourBetweenIndices(List<Vertex>, int, int) void
generateAllSegments(int) List<Triple>
optimumTour List<Vertex>
Triple
Triple(int, int, int)
ThreeOptForSA
ThreeOptForSA(List<Vertex>)
doThreeOpt(int, int, int) double
reverseSegmentIfBetter(List<Vertex>, Triple) double
reverseTourBetweenIndices(List<Vertex>, int, int) void
tour List<Vertex>
Triple
Triple(int, int, int)
AntColonyOptimization
AntColonyOptimization (Graph)
startOptimization() List<Vertex>
recalculateProbabilityMatrix(Set<Integer>, int) void
initializeProbabilityMatrix() void
updateRewards(double, double, List<Vertex>) void
calculateAntColonyTour(int) List<Integer>
initializeDistanceMatrix() void
initializeRewardMatrix() void
TwoOptSwapOptimization
TwoOptSwapOptimization(List<Vertex>)
optimize() void
doTwoOptSwap(List<Vertex>, int, int) void
optimumTour List<Vertex>
ChristofidesWithOptimizationTest
ChristofidesWithOptimizationTest ()
loadGraph() void
CostAfterThreeOptSwapOptimizationWithinRangeOfMSTTest() void
CostAfterTwoOptSwapOptimizationWithinRangeOfMSTTest() void
SimulatedAnnealing
SimulatedAnnealing(List<Vertex>, int, double, double)
optimize() List<Vertex>
coolTemp() void
shouldAccept(double, double) boolean
SimulatedAnnealingTest
SimulatedAnnealingTest()
test3AOptimization() void
competitionSAvs3A() void
testWithBigDataSet() void

Unit Tests

We have written test cases based on the invariants across the Christofides algorithm and made sure that with every step of the project these test cases didn't fail.

For all the tours that we got after optimizations we made sure that the cost of the tour is in the range **10-25%.** Here is a list of all the test cases that we wrote as part of the project.



Conclusion

- In conclusion, the Travelling Salesman problem (TSP) is a classic optimization problem that involves finding the shortest possible route that visits all the given cities and returns to the starting point. In this report, we have explored the use of the Christofides Algorithm as a heuristic approach to solve TSP, followed by the application of various optimization techniques such as 2-opt, 3-opt, Simulated Annealing and Ant Colony Optimization to further refine the solution.
- Our experimental results have shown that the initial Christofides Algorithm was able to provide a reasonably good solution in most cases. However, the application of optimization techniques has helped to further improve the quality of the solutions obtained. The 2-opt and 3-opt algorithms were particularly effective in optimizing the solution by iteratively swapping edges and segments of the tour, respectively. Simulated Annealing and Ant Colony Optimization also provided good results, albeit with longer computation times.
- In summary, the results of our experiments demonstrate the effectiveness of combining the Christofides Algorithm with various optimization techniques for solving TSP. While each optimization technique has its strengths and weaknesses, their combination can produce superior solutions that are near-optimal in a reasonable amount of time. What algorithm is best suited for you is heavily dependent on your usecase. Future research may explore the application of other optimization techniques or hybrid approaches to further improve the performance of TSP solvers.
- Best Tour: **605864.9369014174 meters**

  33392-->36473-->47152-->59728-->60381-->63272-->64087-->64764-->65977-->68683-->79491-->58196-->77558-->72571-->70824-->73918-->75269-->72656-->65102-->57771-->54558-->54402-->51104-->52902-->54578-->56032-->50118-->45342-->54052-->53902-->36887-->38552-->42136-->37288-->27833-->23768-->20779-->50993-->50033-->00588-->97809-->93203-->91327-->82392-->47976-->78737-->67667-->46836-->47824-->81394-->83548-->87124-->48573-->85777-->49214-->08069-->09645-->08231-->03993-->92689-->87816-->48199-->78192-->73036-->66898-->62846-->54274-->45295-->35474-->25167-->21945-->41387-->12869-->06986-->00999-->99089-->05853-->21468-->26741-->31078-->21483-->10088-->02299-->11236-->24692-->29562-->45563-->44692-->55257-->59266-->48347-->70735-->61387-->55522-->46382-->61078-->72883-->49026-->95556-->98757-->92948-->97058-->06217-->27701-->33833-->34355-->28745-->33407-->54511-->51236-->06831-->35373-->46169-->61719-->90272-->05036-->09793-->94358-->75535-->79435-->83216-->79711-->77486-->76028-->81522-->61826-->50413-->46108-->54352-->49652-->58158-->55981-->60038-->60038-->47257-->40987-->38044-->40042-->36435-->53002-->53602-->45323-->46578-->31775-->24358-->23687-->53602-->43293--

>43293-->44443-->44171-->46676-->49348-->53159-->57658-->60526-->56367-->73667
-->86857-->58022-->82419-->57646-->75913-->86251-->08904-->00363-->05844-->0476
9-->08598-->07134-->88406-->87939-->58538-->76901-->72725-->68231-->75467-->830
08-->87324-->88911-->90878-->58423-->58724-->89026-->59382-->93061-->94524-->03
411-->02281-->01323-->60794-->19443-->13718-->10971-->06501-->61388-->20426-->3
2317-->36226-->36943-->28983-->62818-->12721-->22939-->30544-->50752-->56005-->
55359-->57767-->61535-->79836-->66585-->56002-->46539-->51448-->64643-->16224--
>96899-->89555-->72221-->70846-->53863-->59548-->63174-->56381-->58988-->58235
-->49418-->49724-->59805-->68193-->71318-->77793-->86493-->59271-->06807-->0112
3-->89891-->76143-->70207-->65118-->57589-->52087-->54757-->30455-->29674-->298
49-->28633-->28645-->25296-->51968-->.5186-->13777-->51401-->14445-->13243-->13
361-->13848-->13064-->11703-->11703-->10942-->10942-->09158-->10952-->12136-->1
2513-->51179-->51179-->13406-->14246-->14492-->15087-->15684-->16166-->13708-->
13114-->12078-->09097-->12336-->14052-->51335-->17684-->14195-->13827-->14684--
>15088-->16058-->19946-->21028-->23786-->27637-->29035-->30394-->34753-->37349
-->38136-->39425-->40595-->40595-->38028-->44881-->44171-->49934-->45695-->4311
7-->53632-->46295-->54276-->48631-->42493-->40267-->35076-->25009-->25009-->226
69-->15434-->15267-->13647-->50835-->00987-->99842-->97988-->97252-->96827-->93
505-->49012-->89662-->91604-->93621-->95541-->90825-->90825-->87966-->87289-->8
9205-->88618-->85535-->47953-->83491-->88202-->48753-->77401-->69029-->68077-->
53099-->54287-->62606-->60693-->55502-->45539-->53122-->54757-->62029-->62029--
>46327-->64088-->75576-->81028-->47597-->68986-->62958-->46891-->73158-->82585
-->75383-->47611-->59749-->56633-->44431-->44936-->80916-->89693-->90054-->8503
8-->87984-->89064-->90506-->92965-->01376-->50062-->05474-->17255-->51525-->031
76-->07942-->04937-->09398-->14824-->51742-->27028-->27366-->14506-->08399-->04
266-->00377-->88386-->46613-->54398-->73944-->46912-->46912-->70774-->73328-->8
3613-->07422-->10221-->18214-->24856-->34988-->31479-->06716-->07893-->09932-->
10896-->16965-->.5353-->35407-->41065-->41678-->46966-->54333-->55153-->56456--
>92759-->02816-->14863-->97678-->86772-->83803-->81095-->79667-->57771-->70924
-->56292-->68804-->79181-->88571-->94505-->88228-->68875-->66119-->63817-->5148
6-->43677-->46608-->46064-->25812-->36406-->35191-->21135-->23934-->27333-->522
74-->04444-->00839-->97717-->97865-->49506-->70361-->59853-->47195-->42223-->43
917-->23432-->43194-->28772-->29261-->11621-->40768-->93407-->92837-->75174-->6
7196-->00022-->08918-->93796-->58411-->36049-->64989-->69471-->87306-->00562-->
03237-->07393-->04326-->05141-->51.42-->43817-->38791-->43638-->45662-->46032--
>44613-->33396-->36987-->32096-->41913-->09953-->07325-->00817-->98513-->96268
-->99107-->01212-->96996-->94462-->92517-->84626-->81824-->78988-->81843-->7812
4-->74063-->72592-->71195-->54449-->55152-->59052-->59704-->36951-->23819-->120

91-->14603-->67486-->94167-->00629-->16061-->27032-->39813-->52926-->54714-->56574-->70924-->73079-->47072-->72736-->75251-->81773-->85008-->88158-->88404-->91347-->92274-->97123-->98139-->00446-->94268-->91926-->93347-->97367-->97784-->96888-->97037-->02367-->04168-->05414-->05386-->06374-->01201-->06998-->08087-->08409-->10373-->51163-->13075-->.5194-->22758-->52355-->28176-->27008-->26563-->52166-->19775-->18422-->09785-->18735-->17182-->21582-->29254-->30061-->26132-->24979-->26984-->32452-->33392

References

- YouTube. (2022, July 26). *The traveling salesman problem: When good enough beats perfect*. YouTube. Retrieved April 16, 2023, from https://www.youtube.com/watch?v=GiDsjIBOVoA
- Wikimedia Foundation. (2023, March 2). *2-opt*. Wikipedia. Retrieved April 16, 2023, from https://en.wikipedia.org/wiki/2-opt
- Wikimedia Foundation. (2023, March 27). *3-opt*. Wikipedia. Retrieved April 16, 2023, from https://en.wikipedia.org/wiki/3-opt
- Wikimedia Foundation. (2023, January 28). *Christofides algorithm*. Wikipedia. Retrieved April 16, 2023, from https://en.wikipedia.org/wiki/Christofides_algorithm
- Wikimedia Foundation. (2023, February 17). *Simulated annealing*. Wikipedia. Retrieved April 16, 2023, from https://en.wikipedia.org/wiki/Simulated_annealing
- Liang, F. (2020, April 21). *Optimization techniques - simulated annealing*. Medium. Retrieved April 16, 2023, from https://towardsdatascience.com/optimization-techniques-simulated-annealing-d6a4785a1de7
- M. Dorigo, M. Birattari and T. Stutzle, "Ant colony optimization," in *IEEE Computational Intelligence Magazine*, vol. 1, no. 4, pp. 28-39, Nov. 2006, doi: 10.1109/MCI.2006.329691. URL: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4129846&isnumber=4129833
- Jrtom. (n.d.). *Jrtom/Jung: Jung: Java universal network/graph framework*. GitHub. Retrieved April 16, 2023, from https://github.com/jrtom/jung
- Jgrapht. (n.d.). *JGRAPHT/jgrapht: Master repository for the JGRAPHT project*. GitHub. Retrieved April 16, 2023, from https://github.com/jgrapht/jgrapht