INFO 6205 Spring 2023 Project
Traveling Salesman

Team Members

Name	NUID	Email
Akash Bharadwaj	002787011	bharadwajkarthik.a@northeastern.edu
Mehul Natu	002743870	natu.m@northeastern.edu
Rishi Desai	002751030	desai.ris@northeastern.edu

Introduction

Aim:
The aim of this report is to apply the Traveling Salesman problem (TSP) to a real-world scenario, specifically, to determine the shortest path for visiting crime data points in London. The TSP is a classic optimization problem in computer science, which involves finding the shortest possible tour of a given set of n cities, where each city is visited exactly once, and the tour ends in the starting city.

Approach:

In this report, we will use the Christofides algorithm as a solution to the TSP. The Christofides algorithm guarantees a tour that is at most 1.5 times longer than the optimal solution. We will explain the steps involved in the algorithm and analyze its time and space complexity.
In addition, we will explore various optimization techniques that can be applied to the Christofides algorithm to further improve its performance.
These techniques include:

- 2-opt: Swapping pairs of edges to create shorter paths
- 3-opt: Swapping triplets of edges to create shorter paths
- Ant Colony Optimization: Using a colony of artificial ants to search for good solutions
- Simulated Annealing: A probabilistic optimization technique that iteratively improves a candidate solution.

We will apply the Christofides algorithm and the optimization techniques to the crime data points in London to determine the shortest path for visiting each point and evaluate the effectiveness of each optimization technique in terms of tour length and execution time. Finally, we will discuss the trade-offs between accuracy and efficiency for each technique.

Processes and Procedures :

Christofides -

1. ***Step 1*** - to create an MST from the given nodes and edges. Initially we are using a complete graph for this process. For creating MST we are using Prim's eager algorithm. It works by storing the weight of the smallest edge to a node in a MinHeap. For this we used [Reference 1] IndexedMinPQ which would help us to update the values in the MinHeap in O(log n) time.

   Prims Eager Algorithm –

   - Step 1 - to take an arbitrary node and all neighbors of the node in the MinIndexedPQ along with their weight and current node as the node which has the smallest possible edge to the node.

   - Step 2 - Now we will remove the edge with the lowest weight and add that edge to the MST if not present already. Then for node which was not present in MST we would we would get all its neighbors and check if the edges to these neighbors are present in the MinIndexedPQ, if present then check if the new edge has lower weight than the already present edge if so then change its value to the new edges value. Also if not present then just add.

   - Repeat the second step till the MinIndexedPQ is empty and voilà we got all the edges to theMST.

   - Then create the mst from all the edges

2. To take all the nodes from the graph which have odd edges.
3. To create Minimum Weight Perfect Matching for these nodes. And since the number of nodes is always going to be even, since a three can have only an even number of odd edges. And our MST is a tree.
   - For this step the optimal method should be to apply Blossom's algorithm or its variation, but it was too complex for us to implement. So, we went ahead with a brute force approach
   - We take an arbitrary node and connect it to the closest node which has not already been connected, and repeat the process till we get all the nodes connected.
   - This process looks right but this will bite us later while optimizing TSP
4. Next step is to merge these edges with MST.
5. And then do a Eulerian tour for the same and save the ordering of nodes visited

- First approach was to cover all the edges
- Second approach was to cover all the vertices
- Switching between them did not provide any change in the process
- And to keep things random we always started from an arbitrary node generated by Math.random on random.nextInt()

6. And the last step is to connect nodes in the order saved from the Eulerian tour and remove those nodes which have already been connected. This will give us our desired TSP

2-Opt -

Process

- Step 1 - to taken any two edges break them
- Step 2 - to check if creating valid edges from the 4 nodes we got by breaking the two edges would result in a small total weight of the TSP.
- If so then replace them

Whole process

- To check for each node pair and if we find any changes then repeat the whole process till we are not able to get any change
- We took the implementation reference from - https://en.wikipedia.org/wiki/2-opt

3-Opt -

Process

- Step 1 - to taken any three edges break them
- Step 2 - to check if creating valid edges from the 6 nodes we got by breaking the three edges would result in a small total weight of the TSP.
- If so then replace them
- Whole process
- To check for each triplets of node with the other node and if we find any changes then repeat the whole process till we are not able to get any change
- We took the implementation reference from - https://en.wikipedia.org/wiki/3-opt

Simulated Annealing-

- Simulated annealing is basically exploring even the routes which seems to be worse than current one in search for global minima

- Here what we do is take a starting temperature as 1 and decrease it by (1 - coolingRate) times this will exponentially decrease the temperature and we want to do this to explore different seem to be worse soulution than current.
- Any solution is accepted if the solution is better than the current solution or
- e ((current solution – new solution)/ temperature) > Random between (0, 1)
- We are doing this so that in initial stages when temperature is near 1 we would explore what seems to be a worse solution and later on stages when temperature starts decreasing we do not accept such solution.
- tempEquliburm is constant is also a factor which is basically how many time do we need to try different solutions for the same temperature we got the idea for this from - [https://www.cs.ubc.ca/~hutter/previous-earg/EmpAlgReadingGroup/TSP-JohMcg97.pdf]
- increaseInTempEquliburmCount - this idea we introduce ourselves so as to check more solutions at later on stages to give ample opportunity to try new solutions when temperature is low. What this does is just adds to temEquliburm at every temperature change so for lower temperature tempEquliburm is hight thus giving more chance to search

Process

- First we set a temperature, cooling rate, max Iterations, temperature equilibrium count and increaseInTempEquliburmCount
- Then we do a while till temperature not zero or number of iteration has not been competed
- Then we try different solution on that temperature and then improve or go to a worse solution
- Above step is repeated till the count for that temperature has reached the tempEquliburm
- Once reached then we decrease the temperature by (1 - coolingRate) and then increase the tempEquliburm with increaseInTempEqulibriumCount
- Repeating the whole process till temperature is 0 and max iteration have reached

Ant-Colony: (Explained in Program section)

Program

2-Opt, 3-Opt, Simulated Annealing : ***(Documented in codebase)***

Ant Colony Optimization:

- Classes:
    1. **Location**: A class used to represent a crime data point - consisting of the crimeId, latitude and longitude which makes up a 'location'

    2. **Ant:** A class used to represent information about a specific tour path; in the AntColony optimization, each 'ant' has a one-to-one mapping with a tour permutation.

    3. **AntColony**: A class which is used to encapsulate all locations and the distances between them; this class is composed of a 'graph' adjacency matrix to represent edge weights; a 'reward' adjacency matrix used to represent the lure factor for future ants to follow previously found short distance tour edges; and the travelingSalesman method which together with other helper methods, calculates the minimum tour cost accounting for each traveling ant.
- Algorithms:

  The defining algorithms for the Ant Colony optimizations occur in the ***getWeightage***, ***generateProbabilities***, ***generateRandomVertex*** and ***alterRewardMatrix*** methods in the AntColony class.

    1. ***getWeightage(int x, int y)***: a helper method that calculates the 'desirability' factor for an ant to traverse the edge between vertices 'x' and 'y'; this is crucial in determining the probability for an ant to select a certain edge to traverse for the shortest tour. 'Desirability' or 'weightage' is inversely proportional to distance between two edges and directly proportional to 'reward' for traversing the edge calculated from previous tours.

    2. ***generateProbabilities(int currentVertex, Ant ant):*** This is used to generate the probabilities for an Ant – 'ant' at a given vertex - 'currentVertex' – to choose the next adjacent edge from 'currentVertex' to traverse. These probabilities are generated for each edge using the ratio: $\frac{getWeightage(edge_i)}{\sum_{j \neq i} getWeightage(edge_j)}$.

    3. ***generateRandomVertex(Ant ant, int currentVertex, double[][] probabilities)***: Given an Ant 'ant', its current position in the tour – 'currentVertex', and the probabilities for selection of an adjacent edge – 'probabilities', this function generates a random number and selects the edge for traversal based on the random number.

4. ***alterRewardMatrix(Ant[] ants):*** Given all the ants in the colony that have travelled, this function updates the 'reward' or lure factor for those edges which were traversed by each ant, in order to lure future ants towards edges which result in a shorter overall distance for the tour.

- Invariants:

1. ***Global pheromone update:*** The reward matrix is always be updated or 'altered' ***after*** all ants in a generation have travelled, to create a 'lure' bias towards a shorter overall path for the next generation of ants.

2. ***Pheromone Trails***: Ants deposit 'pheromones' on each edge or in the context of our code, they update the reward factor for the edge they chose to traverse based on the quality of that solution predicted after having chosen those edge; in our case, this 'quality' is determined by the 'weightage' function which represents 'desirability'.

3. ***Heuristic function to choose next edge to traverse:*** The ***generateRandomVertex*** probabilistic function is our heuristic function to choose the next most favorable edge for an ant on a tour.

Flowcharts and UI

We have included 2 User interfaces for visualization diversity. ***Christofides, 2-Opt, 3-Opt*** and ***Simulated Annealing*** are displayed using '***Java Swing'*** which plots on cartesian scale, and the ***Ant Colony solution*** to the TSP is displayed using a real life presentation on the browser using HERE maps. It represents the coordinates on the actual world map.

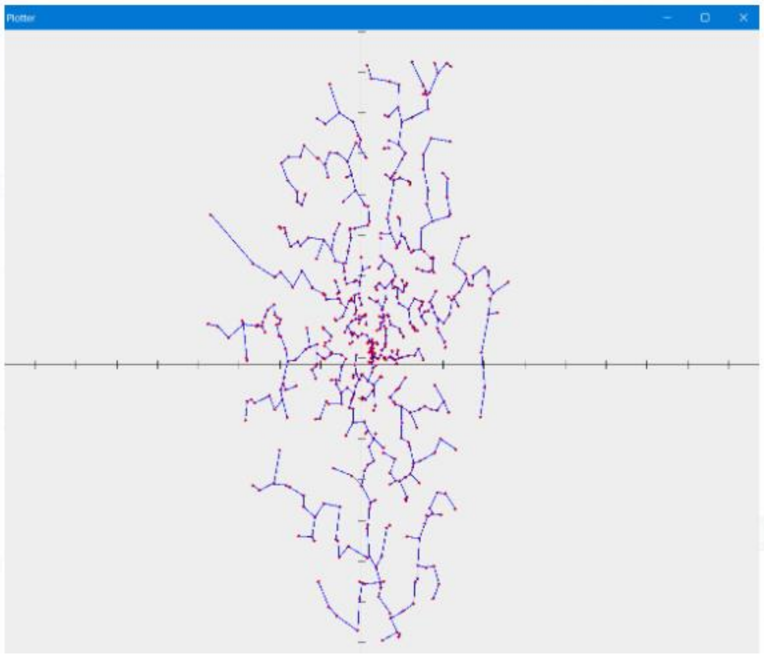


Figure 1: MST

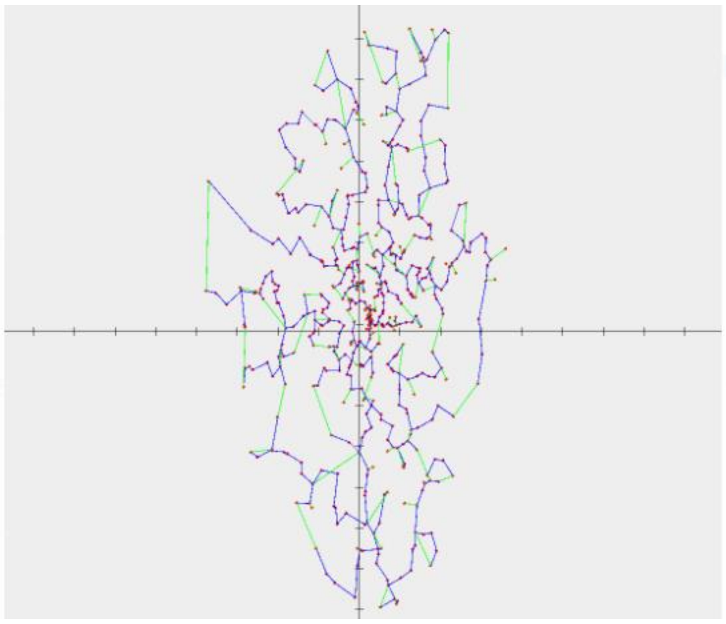


Figure 2: MST with perfect matching

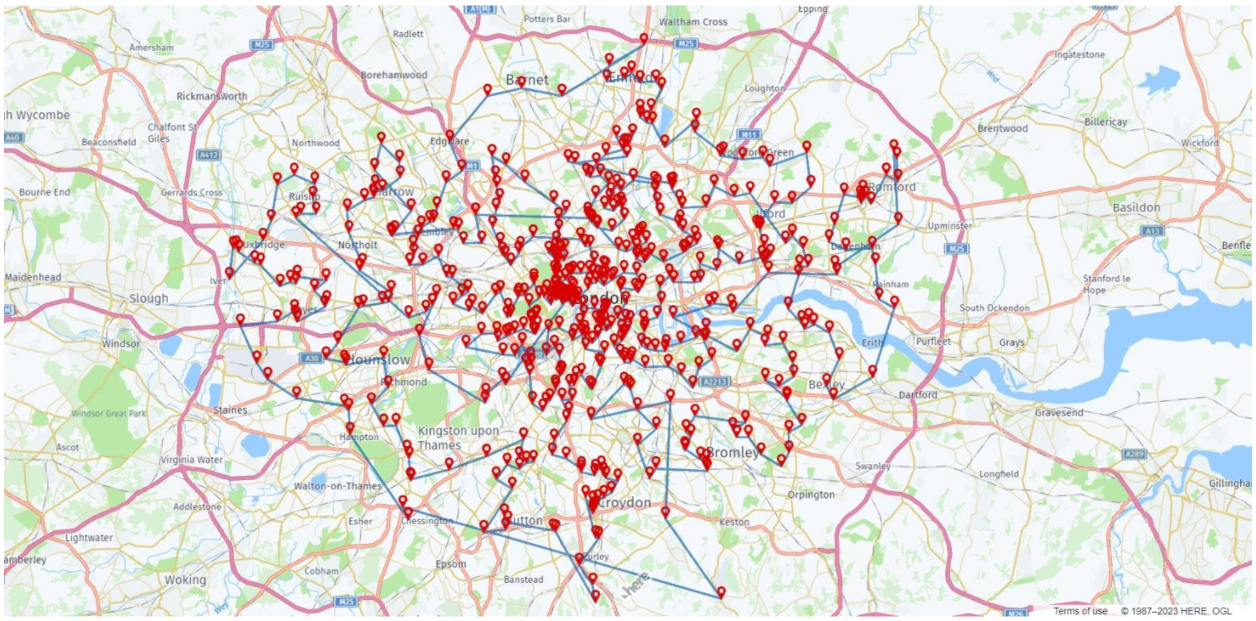


Figure 3: Ant Colony - Optimal Tour (zoomed out)

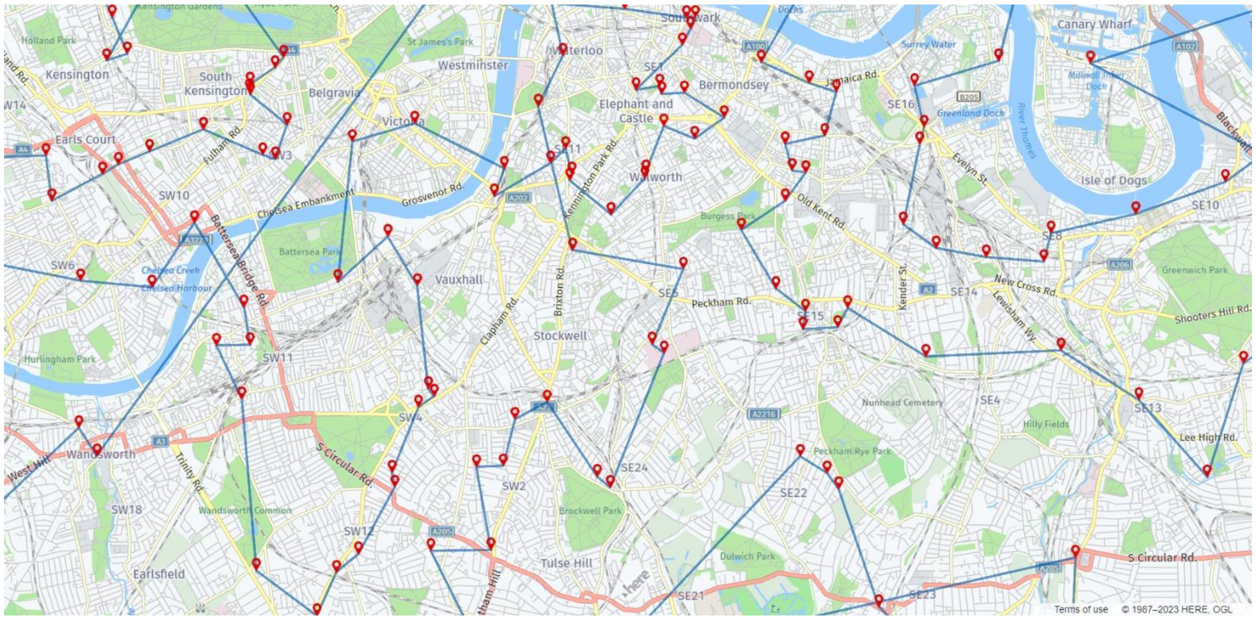


Figure 4: Ant Colony - Optimal Tour (zoomed in)

Observation & Graphical Analysis

AntColony:

The heuristic for minimum tour cost generated from the AntColony optimization allows for a few degrees of freedom in terms of parameters that can be varied such as:

1. ***Number of Ants*** **–** Number of ants that travel in each generation for more precision in minimum cost.
2. ***Generations –*** *The number of trials to conduct; crucial for this optimization as feedback to future generations from global pheromone updates is what produces better results.*
3. ***Alpha*** – (reward sensitivity; the reward value is raised to the power alpha)
4. ***Beta*** – (distance sensitivity - attraction of ants towards closer distance edges; inverse distance is raised to beta)
5. ***Evaporation Rate*** – (Pheromones are decayed after each generation by this number so that a single local minimum doesn't dominate all results and there is scope for better results by tolerating possible short-term worse results)

Hence, graphical analysis takes these parameters into consideration, and we attempt to find the best combination of these parameters which results in the lowest total tour distance.

Generations = 100, Number of Ants = 100,			
Alpha	Beta	Evaporation Rate	Min Tour Cost(m)
1.0	5	0.1	681883.4081709614
1.0	5	0.2	682254.6538168394
1.0	5	0.3	690347.8367897691
1.0	7	0.1	688710.816802817
1.0	7	0.2	684806.7929449955
1.0	7	0.3	681701.2445350608
1.0	9	0.1	676483.9833331941
1.0	9	0.2	687226.5425478045

1.0	9	0.3	682633.4298107352
1.5	5	0.1	673866.7023713198
1.5	5	0.2	665664.1696853424
1.5	5	0.3	666274.9133081606
1.5	7	0.1	666661.4714950342
1.5	7	0.2	674560.4069215985
1.5	7	0.3	677052.1605714431
1.5	9	0.1	675714.6272952942
1.5	9	0.2	678247.6237869023
1.5	9	0.3	675027.5167903384
2.0	5	0.1	679155.2959771959
2.0	5	0.2	680212.3268328843
2.0	5	0.3	668436.525020508
2.0	7	0.1	666151.357245922

2.0	7	0.2	672097.3469282889
2.0	7	0.3	672696.5390895676
2.0	9	0.1	678109.4297143553
2.0	9	0.2	679819.3971629858
2.0	9	0.3	674765.19010161

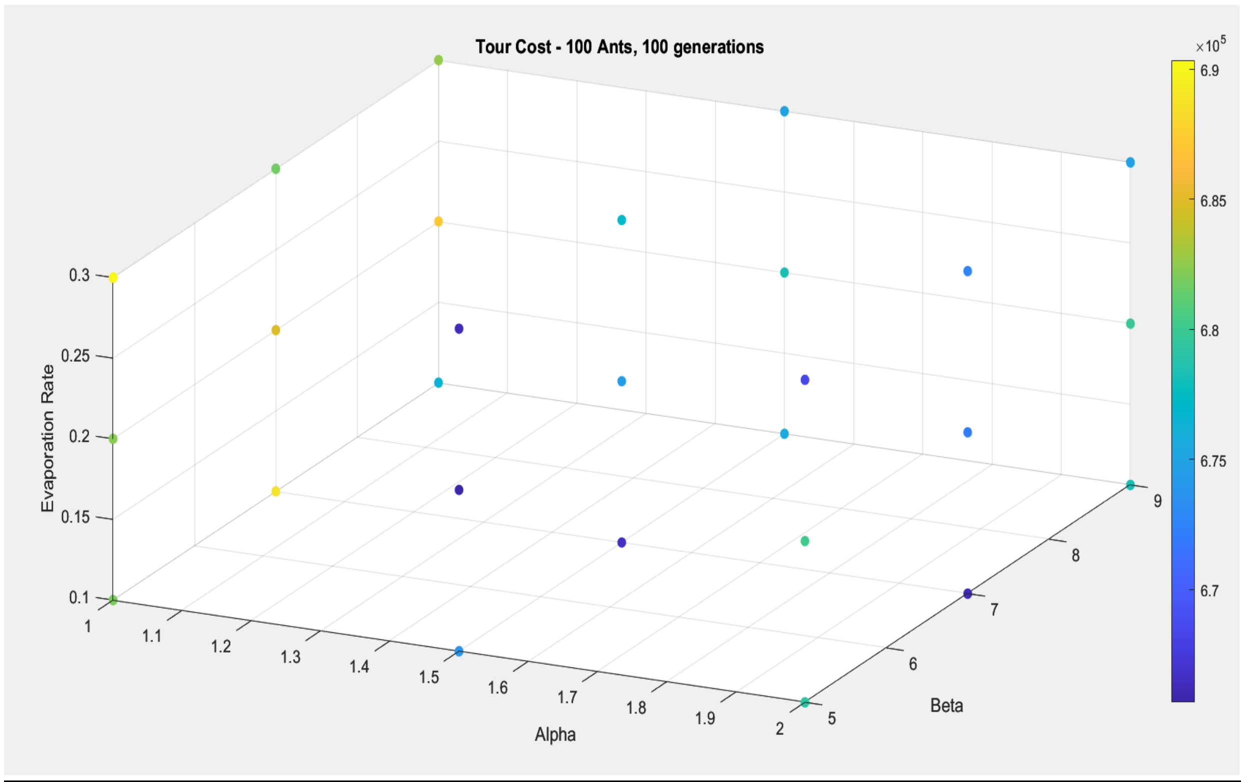
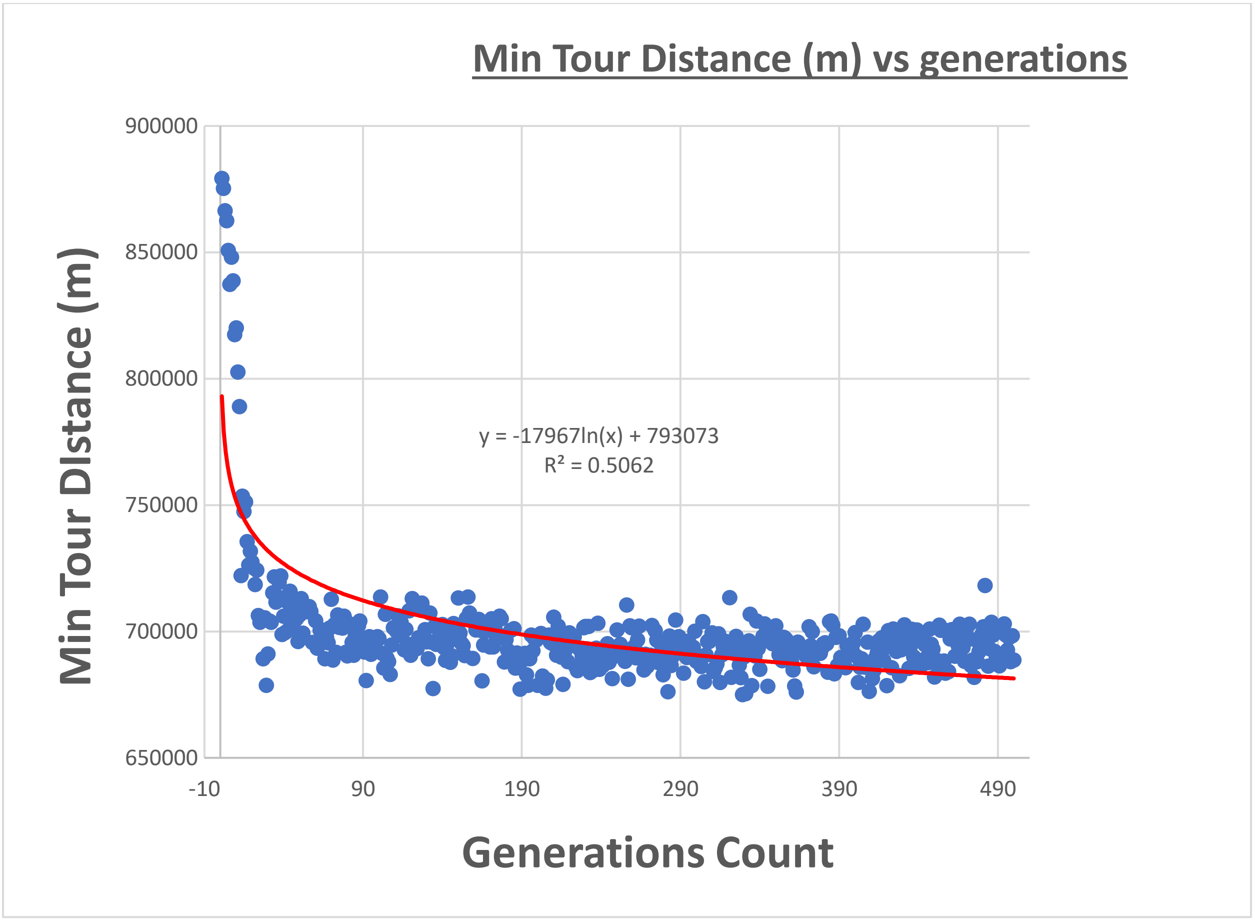


Figure 5: Heat map of Minimum Tour Distance (km) varied with alpha, beta, evaporate for 100 ants and generations

The minimum tour cost is in purple as indicated by the scatter plot, and occurs around **alpha = 1.5, beta = 5, evaporate = 0.3**)

The ***minimum tour distance*** (meters) vs ***generation*** **count** is now measured for these fixed parameter values **(alpha = 1.5, beta = 5, evaporate = 0.3**), to see the optimal performance of the algorithm as generation count increases. This is done for 50 ants over 500 generations, and hence the direct plot is shown:



Precursor to graphical Analysis for other TSP solving techniques:

Christofides -

Observations –

- MST we got around 513 km
- For the small data set we were ratio of TSP/ MST as - 1.5
- For the larger data set with 575 points removing duplicate we were getting the Ratio as 1.68
- Which should not happen since the upper bound for Christofides is 1.5
- So, the problem was the Minimum Weight Perfect Matching.

- We do not using optimal approach for the same, to improve this what we did was to iterate through all the edges and check if break two edges and connecting them in a different way would lead us to decrease the weight, just like an 2 OPT
- And this process worked now we were getting results of about - 1.4 in the small data set and 1.56 in the larger data set still not inside thye 1.5 bound. But it was good enough for us to move forward

2-Opt

- ***Observations -***
  - We were able to get the results down from 781 km to 650 km by checking each possible index for the tour.

3-Opt

- ***Observations*** -
  - We were able to get the results down from 781 km to 670 km by checking each possible index for the tour.

  - The implementation present in the wiki for the same is not totally correct; it only has one true 3 opt combination. What do we mean by True Three opt, by true three opt we mean that all the edges which are selected for breaking should be not be kept or we should remove those edges totally not only any two of them which what was happening the implementation of the three Opt we took from wiki

  - Now we also read a paper to do Three opt in subcubit time [https://www.mdpi.com/1999-4893/13/11/306] which helped us understand true three opt, now we implemented the true three Opt which could have only two ways to do so all the others are derivation from those two ways only. Which affected our three opt-in simulated annealing and helped us to improve our performance. We wished to implement this paper but we were not able to.

Simulated Annealing-

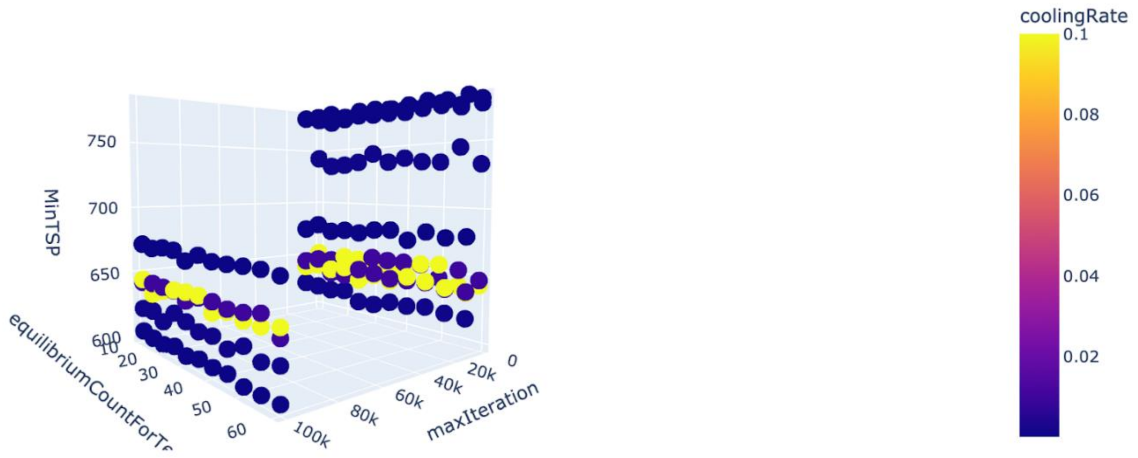
- ***Observation -***
  - Since this is a random process, we do not get consistent results for a 2-Opt and 3-Opt run. They fluctuate quite a lot
  -

Graphical Analysis -

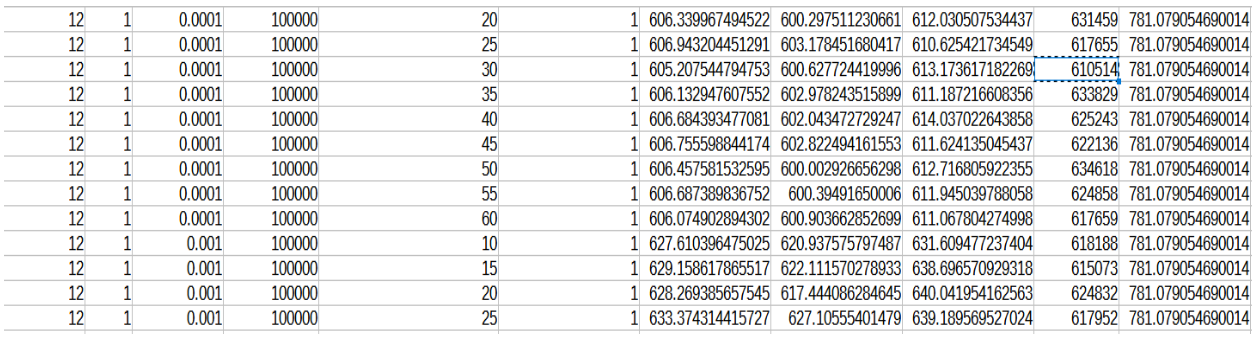
1. First Benchmark - Parallelized 2 Opt –

a. We changed cooling rate from - [0.00001, 0.0001, 0.001, 0.01, 0.1]
b. maxIteration - [1000, 100000] incrementing with a multiple of 10
c. Equilibrium count for temp : 10 to 60

(Min TSP graphed on z axis)



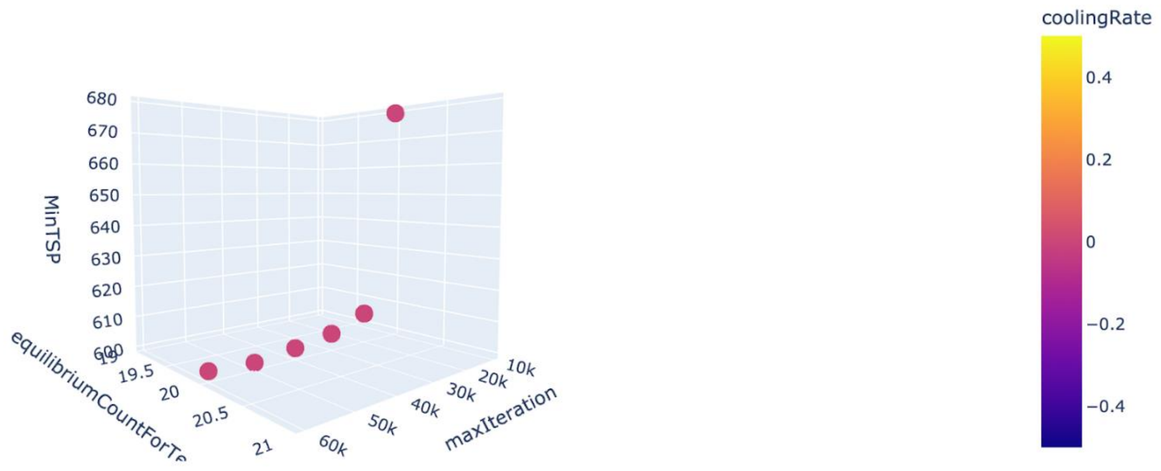
Here we can see for cooling rate 0.0001 we are getting min tsp around 600000 meters and maxIteration to be around 100000, and this took nearly 610514 temp which we can see from this data as well



12	1	0.0001	100000	20	1	606.339967494522	600.297511230661	612.030507534437	631459	781.079054690014
12	1	0.0001	100000	25	1	606.943204451291	603.178451680417	610.625421734549	617655	781.079054690014
12	1	0.0001	100000	30	1	605.207544794753	600.627724419996	613.173617182269	610514	781.079054690014
12	1	0.0001	100000	35	1	606.132947607552	602.978243515899	611.187216608356	633829	781.079054690014
12	1	0.0001	100000	40	1	606.684393477081	602.043472729247	614.037022643858	625243	781.079054690014
12	1	0.0001	100000	45	1	606.755598844174	602.822494161553	611.624135045437	622136	781.079054690014
12	1	0.0001	100000	50	1	606.457581532595	600.002926656298	612.716805922355	634618	781.079054690014
12	1	0.0001	100000	55	1	606.687389836752	600.39491650006	611.945039788058	624858	781.079054690014
12	1	0.0001	100000	60	1	606.074902894302	600.903662852699	611.067804274998	617659	781.079054690014
12	1	0.001	100000	10	1	627.610396475025	620.937575797487	631.609477237404	618188	781.079054690014
12	1	0.001	100000	15	1	629.158617865517	622.111570278933	638.696570929318	615073	781.079054690014
12	1	0.001	100000	20	1	628.269385657545	617.444086284645	640.041954162563	624832	781.079054690014
12	1	0.001	100000	25	1	633.374314415727	627.10555401479	639.189569527024	617952	781.079054690014

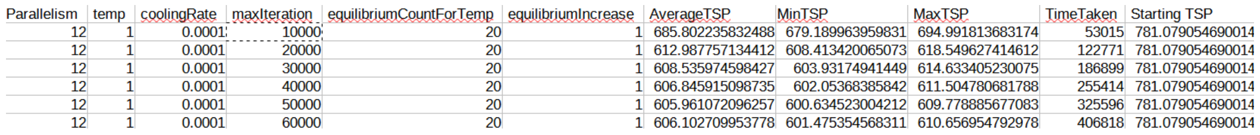
This data is present is the file named - 2Opt_bechmarking_2

- Second Bench Mark - Parallelized 2 Opt -
    - So we started digging further what if the TSP becomes constant after 200000 maxIteration only
    - We kept cooling rate to - 0.0001 as from previous observation this performs best
    - maxIteration - [10000, 60000] incrementing with of 10000
    - Graph for the Min TSP - on z axis



- Second benchmark observation - that indeed we are reaching min tsp of around 600 - 610

Also, time can be observe in this -



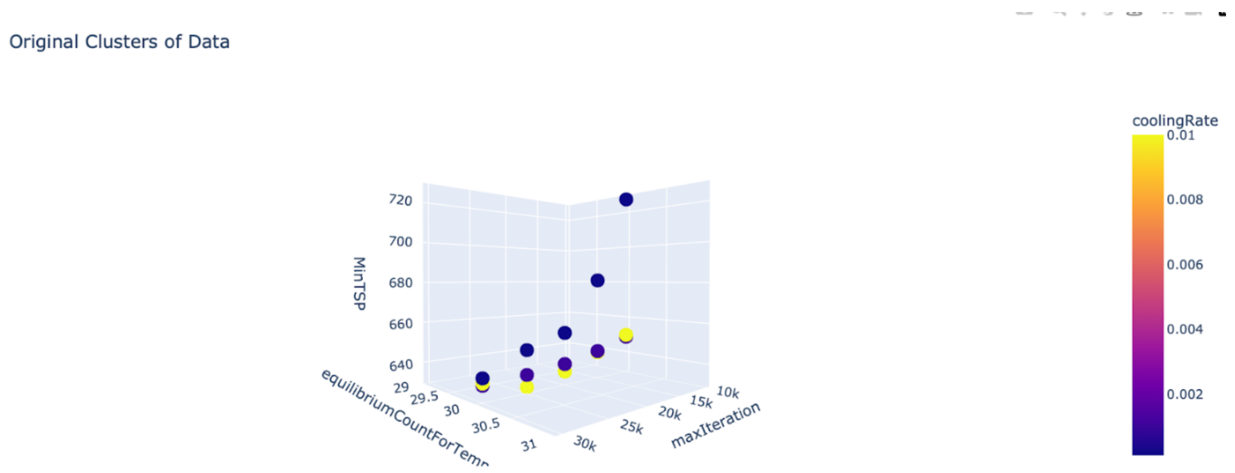
Parallelism	temp	coolingRate	maxIteration	equilibriumCountForTemp	equilibriumIncrease	AverageTSP	MinTSP	MaxTSP	TimeTaken	Starting TSP
12	1	0.0001	10000	20	1	685.802235832488	679.189963959831	694.991813683174	53015	781.079054690014
12	1	0.0001	20000	20	1	612.987757134412	608.413420065073	618.549627414612	122771	781.079054690014
12	1	0.0001	30000	20	1	608.535974598427	603.93174941449	614.633405230075	186899	781.079054690014
12	1	0.0001	40000	20	1	606.845915098735	602.05368385842	611.504780681788	255414	781.079054690014
12	1	0.0001	50000	20	1	605.961072096257	600.634523004212	609.778885677083	325596	781.079054690014
12	1	0.0001	60000	20	1	606.102709953778	601.475354568311	610.656954792978	406818	781.079054690014

This data is present in the file named - **2Opt_bechmarking_3**

- Which is near 180 seconds so we decided to go on with this settings
- maxIteration - 2600
- Equilibrium count - 20
- Increase in equilibrium count 1
- Cooling rate 0.0001
- Temperature - 1
- Parallelism 12

Graphical Analysis for the 3Opt

- After random testing we ran three opt with the variables -
    - We changed cooling rate from - [0.0001, 0.001, 0.01]
    - maxIteration - [10000, 25000] incrementing with 5000
    - Equilibrium count for temp - 30
    - Graph for the Min TSP - on z axis



This gave us results near 640 km to 756 km and with time - 257 seconds to 585 seconds which is worse than the 2 Opt simulated annealing

Parallelism	temp	coolingRate	maxIteration	equilibriumCountForTemp	equilibriumIncrease	AverageTSP	MinTSP	MaxTSP	TimeTaken	Starting TSP
12	1	0.01	10000	30	1	656.052016537263	650.135120099703	661.621111561886	149132	781.079054690014
12	1	0.001	10000	30	1	657.423496050331	649.076460275589	670.654416752176	155471	781.079054690014
12	1	0.0001	10000	30	1	756.187402446591	726.856966831546	773.153466617683	159282	781.079054690014
12	1	0.01	15000	30	1	649.13256172415	642.952824392729	654.80175795839	257116	781.079054690014
12	1	0.001	15000	30	1	649.071526588236	643.550944590109	653.16267365892	259524	781.079054690014
12	1	0.0001	15000	30	1	691.589540397767	681.071438778703	703.39346423481	253637	781.079054690014
12	1	0.01	20000	30	1	645.299280799645	636.029898599684	657.061729602687	351845	781.079054690014
12	1	0.001	20000	30	1	647.647459695923	639.718711474421	652.951758059043	357293	781.079054690014
12	1	0.0001	20000	30	1	668.319151974847	655.169485519727	676.066484904212	357194	781.079054690014
12	1	0.01	25000	30	1	639.413205567735	632.160247526361	646.71428622263	483418	781.079054690014
12	1	0.001	25000	30	1	645.698638598892	637.786050098772	651.737107015938	482986	781.079054690014
12	1	0.0001	25000	30	1	654.338602239955	649.212667759171	659.190778434898	478175	781.079054690014
12	1	0.01	30000	30	1	641.64126295714	637.667383360879	648.510388437446	578580	781.079054690014
12	1	0.001	30000	30	1	642.046970878747	636.70266661293	653.612789623051	582469	781.079054690014
12	1	0.0001	30000	30	1	647.655574178411	639.840200569548	655.655131173116	585664	781.079054690014

Results and Mathematical Analysis

Ant-Colony:

From the graphical analysis and benchmarking, it is found that the minimum overall tour distance found through Ant Colony optimization is **675002.61 m,** with parameters **alpha = 1.5, beta = 5, evaporate = 0.3**.

The equation of **the minimum tour cost logarithmically decreases with increasing generation count, as can be seen by the graph equation.**

2-Opt, 3-Opt, Simulated Annealing:

- We first rant simulated annealing of random variables and then we got a gist which variable should be around what values to we ran benchmarking near those values.
    - 2 Opt -
        - Temperature - 1
        - coolingRate - 0.0001
        - maxIteration - 20000 - 30000
        - tempEquliburm - 20 - 30
        - increaseInTempEquliburmCount - 1
        - This gave a good results in terms of time we got to around 5 minutes with getting our TSP/MST ratio - 1.15 - 1.2
    - 3 Opt -
        - Temperature - 1
        - coolingRate - 0.001
        - maxIteration - 10000 - 30000
        - tempEquliburm - 20 - 30
        - increaseInTempEquliburmCount - 1
        - This gave us okay results in terms of time we got to around 5 minutes with getting our TSP/MST ratio - 1.25
        - This we were getting pure three opt but the one implemented from wiki did not work for us
        -

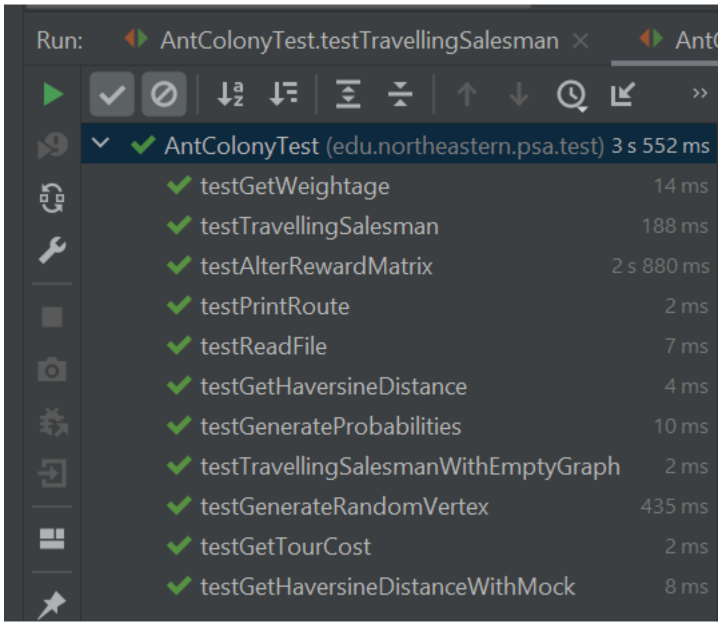
Results - (On the final data set)

- Simple 3 opt - (TSP:633528 m),(Time Taken: 365 seconds) (Ratio: 1.2341637466371382)
- Simple Pure 3 opt - (TSP: 649 km),(Time Taken: more than 5 minutes) (Ratio TSP/MST: 1.26)
- Parallelized Simulated 3 opt - (TSP: 649793),(Time Taken: 161 seconds)(Ratio TSP/MST: 1.2658501951115255)
- Simple 2opt - (TSP: :645698 m),(Time Taken: 2 seconds)(Ratio TSP/MST: 1.25)
- Parallelized 2opt: (TSP:593324 m),(Time Taken: 196)(Ratio TSP/MST: Ratio1.1558425994327923)
- Parallelized 2 Opt proceeding with Simple 3 Opt: (TSP: 596056),(Time Taken: 469 seconds)(Ratio TSP/MST: 1.1611662530700548)

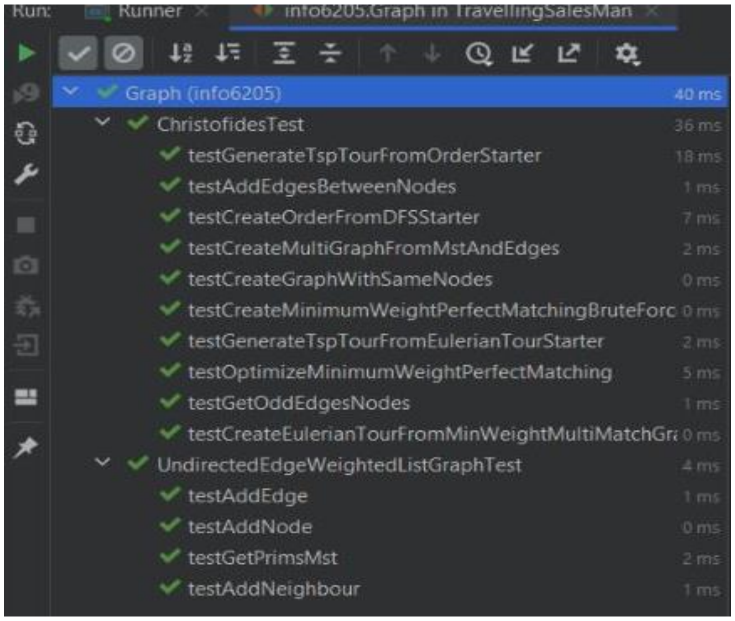
Unit Tests

Unit Tests are written for the '***AntColony***' class to test all the ***algorithmic methods of concern.*** (Getters and Setters are ignored as they are trivial)

Junit Testing framework is used extensively for this purpose along with ***Mockito*** to completely and partially mock/spy on dependent objects to test all class methods.



```
Run:   AntColonyTest.testTravellingSalesman ×   Ant
AntColonyTest (edu.northeastern.psa.test) 3 s 552 ms
    testGetWeightage                             14 ms
    testTravellingSalesman                      188 ms
    testAlterRewardMatrix                   2 s 880 ms
    testPrintRoute                                2 ms
    testReadFile                                  7 ms
    testGetHaversineDistance                      4 ms
    testGenerateProbabilities                    10 ms
    testTravellingSalesmanWithEmptyGraph          2 ms
    testGenerateRandomVertex                    435 ms
    testGetTourCost                               2 ms
    testGetHaversineDistanceWithMock              8 ms
```



```
Run:   Runner ×   info6205.Graph in TravellingSalesMan ×
Graph (info6205)                                                40 ms
  ChristofidesTest                                              36 ms
    testGenerateTspTourFromOrderStarter                         18 ms
    testAddEdgesBetweenNodes                                     1 ms
    testCreateOrderFromDFSStarter                                7 ms
    testCreateMultiGraphFromMstAndEdges                          2 ms
    testCreateGraphWithSameNodes                                 0 ms
    testCreateMinimumWeightPerfectMatchingBruteForc              0 ms
    testGenerateTspTourFromEulerianTourStarter                   2 ms
    testOptimizeMinimumWeightPerfectMatching                     5 ms
    testGetOddEdgesNodes                                         1 ms
    testCreateEulerianTourFromMinWeightMultiMatchGr              0 ms
  UndirectedEdgeWeightedListGraphTest                            4 ms
    testAddEdge                                                  1 ms
    testAddNode                                                  0 ms
    testGetPrimsMst                                              2 ms
    testAddNeighbour                                             1 ms
```

Unit Test Source Codes:

```java
package edu.northeastern.psa.test;

import static org.junit.Assert.*;
import static org.mockito.Mockito.*;

import edu.northeastern.psa.Ant;
import edu.northeastern.psa.AntColony;
import org.junit.Before;
import org.junit.Test;
//import org.junit.jupiter.api.BeforeEach;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

import java.io.ByteArrayOutputStream;
import java.io.IOException;
import java.io.PrintStream;
import java.util.*;

public class AntColonyTest {

    @Test
    public void testGetHaversineDistance() {
        AntColony antColony = new
AntColony("src/edu/northeastern/psa/test/test.csv");
        double lat1 = 42.3601;
        double lat2 = 40.7128;
        double lon1 = -71.0589;
        double lon2 = -74.0060;
        double expectedDistance = 306.96;
        double distance = antColony.getHaversineDistance(lat1, lat2, lon1,
lon2);
        assertEquals(expectedDistance, distance, 1);
    }

    @Test
    public void testGetHaversineDistanceWithMock() {
        AntColony antColony = new
AntColony("src/edu/northeastern/psa/test/test.csv");
        double lat1 = 42.3601;
        double lat2 = 40.7128;
        double lon1 = -71.0589;
        double lon2 = -74.0060;
        double expectedDistance = 306.96;

        AntColony antColonyMock = mock(AntColony.class);
        when(antColonyMock.getHaversineDistance(lat1, lat2, lon1,
lon2)).thenReturn(expectedDistance);

        double distance = antColonyMock.getHaversineDistance(lat1, lat2,
lon1, lon2);
        assertEquals(expectedDistance, distance, 0.01);
    }

    @Test
    public void testReadFile() throws IOException {
        AntColony antColony = new
AntColony("src/edu/northeastern/psa/test/test.csv");
        int expectedLocationCount = 3;
        assertEquals(expectedLocationCount, antColony.getCoords().size());
    }

    @Test
    public void testTravellingSalesmanWithEmptyGraph() {
        AntColony antColony = new
AntColony("src/edu/northeastern/psa/test/empty-file.csv");
        double[][] graph = antColony.getGraph();
        assertEquals(0, graph.length);
    }

    @Test
    public void testGetWeightage() {
        AntColony antColony = new AntColony();
        antColony.setAlpha(1);
        antColony.setBeta(5);
        antColony.setGraph(new double[][]{
                {0, 1, 4},
                {1, 0, 5},
                {4, 5, 0}
        });
        antColony.setRewardMatrix(new double[][]{
                {1, 1.1, 1.2},
                {1.1, 1, 1.3},
                {1.2, 1.3, 1}
        });

        double expectedWeightage = 0.000416;

        double actualWeightage = antColony.getWeightage(1, 2);
        assertEquals(expectedWeightage, actualWeightage, 0.01);
    }

    @Test
    public void testPrintRoute() {
        AntColony antColony = new AntColony();
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
        System.setOut(new PrintStream(outputStream));

        List<Integer> route = new ArrayList<>();
        route.add(2);
        route.add(1);
        route.add(4);

        // Call the function that prints to the console
        antColony.printRoute(route);

        // Get the printed output
        String printedOutput = outputStream.toString().trim();

        // Define the expected output
        String expectedOutput = "2-->1-->4-->2";

        // Compare the printed output to the expected output
        assertEquals(expectedOutput, printedOutput);
    }

    @Test
    public void testGetTourCost() {
        AntColony antColony = new AntColony();
        antColony.setGraph(new double[][]{
                {0, 1, 4},
                {1, 0, 5},
                {4, 5, 0}
        });

        Ant ant = new Ant();
        List<Integer> visitedLocations = new ArrayList<>();
        visitedLocations.add(2);
        visitedLocations.add(1);
        visitedLocations.add(0);
        ant.setVisitedLocationsInOrder(visitedLocations);

        double actualCost = antColony.getTourCost(ant);
        assertEquals(10, actualCost, 0);
    }

    @Test
    public void testGenerateRandomVertex() {
        AntColony antColony = new AntColony();
        RandomNumberGenerator randomMock = mock(RandomNumberGenerator.class);
        when(randomMock.nextDouble()).thenReturn(0.5);
        AntColony.setR(randomMock);
        double[][] probabilities = new double[][]{
                {0.4, 1},
                {0.6, 2},
                {0.8, 3}
        };

        Ant mockAnt = mock(Ant.class);
        when(mockAnt.getVisited()).thenReturn(new HashSet<>());
        when(mockAnt.getVisitedLocationsInOrder()).thenReturn(new
ArrayList<>());


        int expectedSelectedLocation = 2;
        int actualSelectedLocation = antColony.generateRandomVertex(mockAnt,
1, probabilities);
        assertEquals(expectedSelectedLocation, actualSelectedLocation);

    }

    @Test
    public void testGenerateProbabilities() {
        AntColony antColony = new AntColony();

        AntColony spyColony = spy(antColony);
        spyColony.setGraph(new double[][]{
                {0, 1, 4},
                {1, 0, 5},
                {4, 5, 0}
        });

        doReturn(0.2).when(spyColony).getWeightage(1, 0);
        doReturn(0.8).when(spyColony).getWeightage(1, 2);


        Ant a = new Ant();
        a.setVisited(new HashSet<>());
```

```java
        double[][] actualProbabilities = spyColony.generateProbabilities(1,
a);

        double[][] expectedProbabilities = new double[][]{ {0.2, 0}, {1, 2},
{0, 0} };
        assertArrayEquals(expectedProbabilities, actualProbabilities);
    }

    @Test
    public void testAlterRewardMatrix(){
        AntColony antColony = new AntColony();
        double[][] rewardMatrix = new double[][]{
                {1, 1, 1},
                {1, 1, 1},
                {1, 1, 1}
        };
        antColony.setRewardMatrix(rewardMatrix);

        Ant mockAnt1 = mock(Ant.class);
        when(mockAnt1.getVisitedLocationsInOrder()).thenReturn(new
ArrayList<>(Arrays.asList(0, 1)));

        Ant mockAnt2 = mock(Ant.class);
        when(mockAnt2.getVisitedLocationsInOrder()).thenReturn(new
ArrayList<>(Arrays.asList(1, 2)));

        AntColony spyColony = spy(antColony);

        doReturn(1.0).when(spyColony).getTourCost(mockAnt1);
        doReturn(1.0).when(spyColony).getTourCost(mockAnt2);

        spyColony.alterRewardMatrix(new Ant[]{mockAnt1, mockAnt2});

        double[][] expectedRewardMatrix = new double[][]{
                {1, 2.9, 0.9},
                {2.9, 1, 2.9},
                {0.9, 2.9, 1},
        };

        assertArrayEquals(expectedRewardMatrix, spyColony.getRewardMatrix());


    }

    @Test
    public void testTravellingSalesman() {
        AntColony antColony = new AntColony();
        antColony.setGraph(new double[][]{
                {0, 3, 5, 4},
                {3, 0, 7, 2},
                {5, 7, 0, 9},
                {4, 2, 9, 0},
        });

        antColony.setRewardMatrix(new double[][]{
                {1, 1, 1, 1},
                {1, 1, 1, 1},
                {1, 1, 1, 1},
                {1, 1, 1, 1},
        });

        double actualMinCost = antColony.travellingSalesman(100,100, new
ArrayList<>());
        double expectedMinCost = 18;

        assertEquals(expectedMinCost, actualMinCost, 0.01);

    }

}
```

Conclusion

Best Results –

- We were getting consistent results from the Christofidies and Parallelized 2opt with a consistent ratio of (TSP/MST) 1.15-1.16 withins 196 - 290 seconds, and hence this was the best algorithm for this task
- This could be improved further with the Blossom Algorithm for perfect matching of MST, so that the ratio TSP/MST could be within 1.5, as predicted by the mathematical models.
- Ant Colony seemed to provide a strategic simplicity in implementation and provided good results of TSP/MST ratio of 675002.61/ 513000 ~ **1.31**

References

- The Traveling Salesman Problem: A Case Study in Local Optimization
- Finding the Best 3-OPT Move in Subcubic Time
- 3-opt - Wikipedia
- 2-opt - Wikipedia
- IndexMinPQ.java