TÉCNICO LISBOA

OPTIMIZATION & DECISION

INSTITUTO SUPERIOR TÉCNICO

2^{nd} SEMESTER - $2021/2022$

Traveling Salesman Problem

Group 6:

Catarina Alexandra Saleiro Rodrigues	92434
Guilherme Miguel Viegas Rodrigues	92632
Filipa Barros Costa	92626

Professor:

Susana Vieira

April 22, 2022

Contents

1 Introduction

In this project the traveling salesman problem will be discussed and present three algorithms to approximate the optimal solution will be shown.

The traveling salesman problem [11] is a classic problem in mathematics, operations research and optimization. The original problem involves a traveling salesman who is required to visit each of n cities, indexed by 1,..,n. This way, he leaves from a "base city" indexed by 0, visits each of the n other cities exactly once, and returns to city 0, in one tour (by a tour we mean a succession of visits to cities without stopping at city 0). The question is which order should the salesman visit the cities to make the shortest possible trip, that is, it is required to find such an itinerary which minimizes the total distance traveled by the salesman.

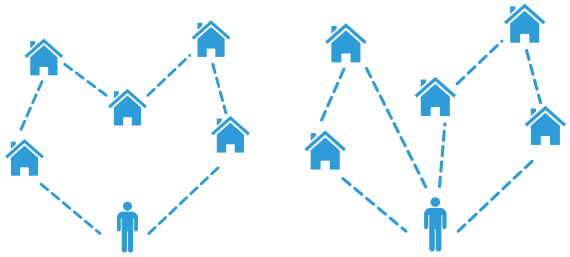


Figure 1: Traveling salesman problem (on the left); Traveling salesman problem with 1 sub-tour (on the right).

It is important to notice that no sub-tours (when the route is broken into multiple trips) are allowed.

This problem is intriguing because it applies to a variety of other interesting questions with real-world implications. For example, a package delivery company may want to plan the shortest route to deliver the day's packages in order to save time and fuel; a school district may have similar concerns when planning the best route for school buses to transport students to and from school; and in drone flight planning, choosing the optimal path through the desired waypoints can reduce time or increase the area that can be covered in a single flight. Of course, each of these challenges has its own set of considerations, but at their basis, they all can be viewed as solving the traveling salesman dilemma.

Unfortunately, the traveling salesman is not always an easy problem to solve, since it is NP-complete. If a problem is NP-complete [3], one is unlikely to find a polynomial-time algorithm for solving it exactly. However, it may still be possible to obtain near-optimal solutions in polynomial time. In practice, near-optimality is frequently sufficient. This way, the polynomial-time approximation algorithms that will be studied for this problem are: Ant Colony Optimization Algorithm (ACO), Genetic Algorithm and an heuristic one: Nearest Neighbour Algorithm.

2 Problem Description

In the traveling salesman problem, it is given a complete undirected graph $G = (V, E)$ that has a nonnegative integer cost c_{ij} associated with each edge $(i, j) \in E$, and the objective is to identify an hamiltonian cycle of G with the lowest cost.

This way, the vertices correspond to cities and the edges relate to the path between those cities. When modeled as a complete graph, paths that do not exist between cities can be modeled as very large cost edges [9]. Minimizing the total cost of the Hamiltonian cycle is the same as finding the shortest path that passes through each city just once. When the distance between cities is equal in both directions ($c_{ij} = c_{ji}$), then one has the symmetric traveling salesman problem to deal with.

2.1 Model Description

The model described bellow has some inputs:

- V is the set of vertices/cities demanded to visit;
- E is the set of edges/paths between each city;
- c_{ij} is the cost/distance between each pair of cities i and j;
- n is number of cities the salesman is required to visit;

- y_{ij} is a binary decision variable such that:

$$y_{ij} = \begin{cases} 1, & \text{if city j is visited immeditely after city i} \\ 0, & \text{otherwise} \end{cases}$$

which indicates if a city j is immediately visited after city i.

The objective of this model is to find the shortest route that visits each city exactly once and returns to the origin city. Let c_{ij} (i≠j = 0, 1,..., n) be the distance covered in traveling from city i to city j.
Thus, the TSP model is the following [9]:

$$\text{Minimize} \sum_i \sum_j c_{ij} * y_{ij} \tag{1}$$

Subject to

$$\sum_{i<k} y_{ik} + \sum_{j>k} y_{kj} = 2, k \in V \tag{2}$$

$$\sum_i \sum_j y_{ij} \leq |S| - 1, S \subset V, 3 \leq |S| \leq n - 3 \tag{3}$$

$$y_{ij} \in 0, 1 \forall i, j \in E \tag{4}$$

The objective function (1) minimizes the total distance traveled by the salesman, by adding the distances associated with each chosen path to be driven by. Note that although we could only focus in minimizing (5) (since if y_{ij} is 1, then y_{ji} needs to be 0), it is important to have all the possible solutions available for the algorithm to go through, and that is already assured in the constraints.

$$\text{Minimize} \sum_i \sum_{j>i} c_{ij} * y_{ij} \tag{5}$$

The constraint (2) indicates that after visiting a city i, the salesman must visit only one city next, and when visiting a city, the salesman must have come from only one city.

The constraint (3) eliminates all solutions containing subtours, as for every proper set of nodes ($S : 3 \leq |S| \leq n - 3$) it forbids the number of the selected paths to be equal to or larger than the number of cities in S, ensuring that a tour is fully connected.

Finally, the constraint (4) defines the domain of the variable y_{ij}.

2.2 Implementation of the Model

3 Data

Two sets of data were used in this project, taken from a website provided by the faculty. One was asked to choose two benchmarks of TSP problems from the database, one with about 100 nodes and one with at least 200 nodes. So it was decided to choose ch130, corresponding to the dataset with about 100 nodes and tsp225, with more than 200 nodes.

So, one will use these two benchmarks to solve the TSP. This way, the only optimization criterion is the distance to complete the journey. The optimal solution to these problems is known, it is 6110 to ch130 and 3916 to tsp225.

4 Methods and Algorithms

4.1 Genetic Algorithm

Nature uses a series of mechanisms that have resulted in the creation of new species that are more adapted to their environments. The genetic algorithm is a classical evolutionary algorithm that is based on randomness and was inspired by Charles Darwin's idea of natural evolution. To put it another way, this algorithm mimics the natural selection process, in which the fittest individuals are chosen for reproduction in order to produce offspring for the next generation. In this method, random changes (such as mutations and crossovers) are applied to current solutions to generate new ones [5].

This way, since GA makes slight changes to its solutions slowly until getting the best solution, it is a slow gradual process [5].

Some parameters in the genetic algorithm should be adjusted in order to produce accurate results. In this project, the various steps that make up the overall framework of a genetic algorithm are provided: encoding, selection method, crossover and mutation operators, and their probabilities, as well as the stopping test.

For each of these steps, there are several possibilities. The freedom to choose between these many options allows us to develop multiple genetic algorithm versions. Following that, we concentrate our efforts on finding a solution to the combinatorial problem: to find what are the best conditions for producing a useful genetic algorithm variation to solve the Traveling Salesman Problem.

4.1.1 The approach

- **Gene:** a city (represented as (x, y) coordinates);
- **Individual:** a single route/solution;
- **Population:** a collection of possible routes/solutions;
- **Parents:** two routes that are combined to create a new route;
- **Mating pool:** a collection of parents that are used to create the next population/set of solutions;
- **Fitness:** a function that tells how good/short each route is (in this case, it will be the inverse of the total distance of a route);
- **Elitism:** a way to carry the best individuals into the next generation;
- **Breed/Crossover:** a way to combine two different parents (routes) to create a new route (in this case, 2 types of crossover will be implemented);
- **Mutation:** a way to introduce variation in the population by randomly mutating individuals in a population (in this case, 4 types of mutation will be implemented);

In this project, the Genetic Algorithm will proceed in the following steps:

Initialization:

- An initial population (set of possible paths) is generated. Each path is chosen randomly, that is, cities are randomly connected to each other.

Iterate until $stop\_criteria$**:**

1. Each individual in the population is ordered by the inverse of its distance (fitness), in descending order;
2. From this decreasing order, is it possible to have probabilities associated with each individual in the population. The individuals whose fitness is greater, have a greater probability of being selected;
3. Next, it is necessary to choose the new generation. First, the *eliteSize* best solutions (elite) are chosen;
4. The remaining places in the population must be filled (let it be called subset_cross). These remaining places will be chosen randomly, having in mind that the individuals with a **higher probability** have **more chance of being chosen**. Note that as these individuals (from the subset_cross) will be crossovered, it is possible to choose elite elements again;
5. The individuals from the subset_cross are crossovered;
6. Having this new population (elitism + crossover), mutation is performed on some chosen individuals according to a *mutationRate*;
7. A new population was created, go back to point (1).

The $stop\_criteria$ used will be the number of generations (fixed at 250 generations), but if the solution does not improve after 10 generations, the algorithm is allowed to stop.

4.1.2 Representation of the solution - Path Representation

The path representation is perhaps the most natural representation of a tour. A tour is encoded by an array of integers representing the successor and predecessor of each city ([2]). For example, a tour $3 \rightarrow 5 \rightarrow 2 \rightarrow 9 \rightarrow 7 \rightarrow 6 \rightarrow 8 \rightarrow 4$ can be represented simply as (3 5 2 9 7 6 8 4).

4.1.3 Crossover

In genetic algorithms, the search for the best solution depends mainly on the creation of new individuals from the old ones ([7]). This way, the process of crossover ensures the exchange of genetic material between parents and thus creates chromosomes that are more likely to be better than the parents.

In an attempt to provide diversity in the population, there are many crossover strategies in the literature, thus the remaining question is which strategy to adopt.

In this project, the path representation will be used, which is the most natural and legal method to represent a tour ([8]).

Since the ordered and cycle crossover operators are the most commonly utilized in literature ([8]), these operators will be implemented in this project.

4.1.4 Ordered Crossover (OX)

The Ordered Crossover method is presented by Goldberg [6], and it is considered one of the best genetic operators used in the resolution of the traveling salesman problem [2]. In ordered crossover, a subset of the first parent string is randomly selected and the rest of the route is filled with genes from the second parent in the order they appear, without duplicating any of the cities already chosen from the first parent. See the illustration 2:

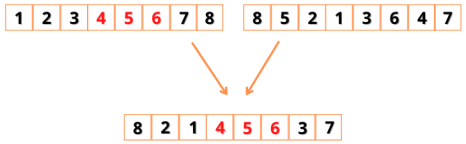


Figure 2: Illustration of ordered crossover

4.1.5 Cycle Crossover (CX)

The cycle crossover (CX) operator was firstly proposed by Oliver et al. [12].

Starting by choosing if the first gene is from the first or the second parent, in our implementation we assume it comes from parent 1, since the parents are randomly assigned.

Next, the algorithm is illustrated.

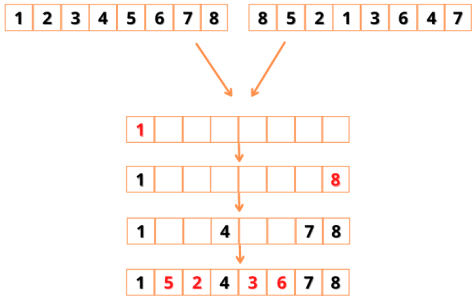


Figure 3: Illustration of cycle crossover

Now, every bit in the offspring should be taken from one of its parents with the same position, which means we don't have a choice, thus bit 8 must be evaluated next, since the bit of the second parent that is in the same position as bit 1 is bit 8.

In first parent, the bit 8 (from the second parent) is at 8th position, thus the child will have bit 8 in the last position.

Next, the bit of the second parent that is in the 7th position is the bit 7. In the first parent, the bit 7 is on the 6th position, thus the child will have bit 7 in the 6th position.

The algorithm stops once a repeated beat is reached. For example, the bit of the second parent that is in the same position as the bit 4 of the child is bit 1, which already is in the child.

Finally, the remaining blank positions are filled with the bits of those positions which are in second parent.

4.1.6 Mutation

The mutation operator is crucial in the Genetic Algorithm, since it encourages the GA to explore new areas of the search space by causing random changes in the chromosomes that reproduction and crossover alone could not entirely ensure. As a result, mutation guarantees that no key characteristics are permanently lost, preserving the diversity of individuals [4].

The mutation, in each individual, will be carried out according to the mutation probability $mutationRate$, which is usually a low probability [1]

In the last decades, several mutation operators for TSP are proposed by researchers. The four most known existing mutation operators [4] are: Inversion mutation, Displacement mutation, Exchange/Swap mutation and Insertion mutation.

In this project, these four different types of mutation will be implemented.

4.1.7 Exchange/Swap/Twors Mutation

Exchange mutation selects two positions at random and swaps the cities on these positions.

Figure 4: Illustration of Swap Mutation

4.1.8 Inversion/Reverse Sequence Mutation

Inversion Mutation selects two positions within a chromosome/tour and then inverts the substring between these two positions.

Figure 5: Illustration of Inversion Mutation

4.1.9 Insertion Mutation

Insertion Mutation selects a city at random and inserts it at a random position. If the random city is the city identified as "2" and the random position is the 3rd position, then the mutated tour will be like 6:

Figure 6: Illustration of Insertion Mutation

4.1.10 Displacement Mutation

Displacement Mutation selects a subtour at random and inserts it at a random position outside the subtour. If the random subtour is the the one in red and the random position is the 3rd position, then the mutated tour will be like 7:

Figure 7: Illustration of Displacement Mutation

4.1.11 Experimental setup

The benchmarks for the TSP are described in section 3. Since the algorithm depends on randomness, all the simulations were carried out for 5 runs, and the objective function mean values, average number of function evaluations and average execution time of successful runs are recorded. It would be better to perform more than 5 runs, however due to computational time it was not possible.

4.1.12 Results and discussion

In the beginning of the choice of the best hyper-parameters, the population size was set at 80, 150, or 300 solutions. Either 30 or 60 was chosen as the elite size. 0.0001, 0.001, 0.01, 0.1 was chosen as the mutation rate likelihood. Four different forms of mutation and two types of crossover were also tested, including swap, invert, insert, and displacement mutation and order and cycle crossover. The number of generations was fixed at 250.

With this, $192 * 5$ runs were made for each problem, and a statistical analysis was conducted. From this statistical analysis, the following conclusions were made:

- For problem 1, the **minimum distance** achieved was 15955, and for the problem 2 was 16809.
- The standard deviation is high (about 7000) for both problems, indicating that **the distance is highly reliant on the hyper-parameters** used.
- The mean and median number of generations for both scenarios is around 150, indicating that the algorithm stops in its vast majority before reaching 250 generations. The algorithm may have become trapped in a local minimum.
- The number of generations and the population size are the hyper-parameters that are more correlated (and with a negative correlation) with the distance for both problems, according to a correlation study 8. This indicates that **when the population and the number of generations grows, the total tour distance may decrease**.

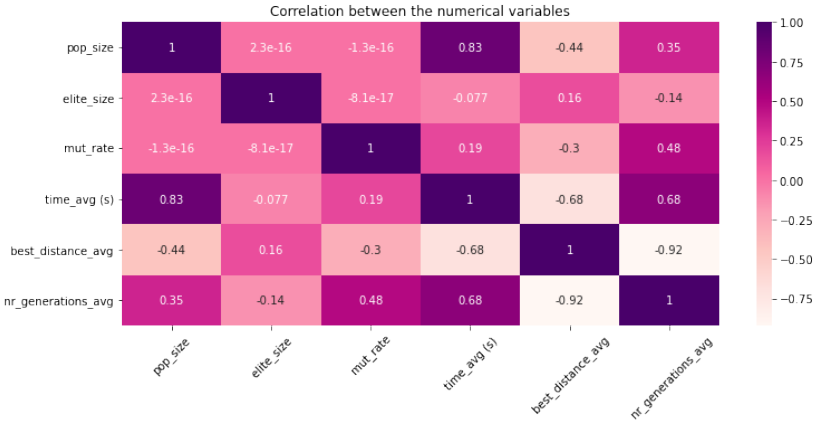


Figure 8: Correlation between the numerical variables, for problem 1 (problem 2 is similar, on appendix 19)

It is also feasible to deduce that **the higher the mutation rate, the more generations the algorithm runs**, which makes sense because increasing the mutation rate increases the exploration of potential solutions, therefore the algorithm takes longer to become trapped in a local minimum.

- It was also discovered that **the mutation type did not appear to influence the distance**, as the p-value for both problems was less than 5% when a non-parametric test was performed on the equality of the means of distances across the four different forms of mutation.
- On another hand, it was found out that **the crossover type highly affects the distance**, since the p-value was around 90%, for both problems.
- Finally, the distance values for each of the hyper-parameters were investigated and are shown in the appendix 1 and 2 . As a result, it is possible to have a better understanding of how each of the hyper-parameters affects the distance. Based on the results of this study, it was determined that **the order crossover produces the best outcomes for both problems**, as it has the lowest standard deviation, distance mean, and distance median. According to the same logic, **the population size of 300 was the best**.
- In terms of mutation rate, aside from achieving a minimal distance value with 1%, it is possible to obtain a lowest standard deviation, as well as a lowest mean, median, and maximum distance, with a mutation rate of 10%. Because it was seen in the correlation plot 8 that raising the mutation rate results in a decrease in the distance value, **the mutation of 10% appears to be the best**.
- The elite size and mutation type does not seem to affect the distance results.

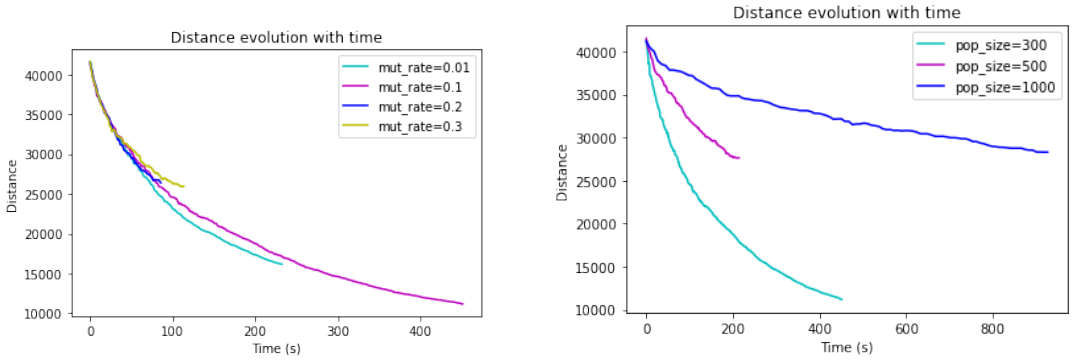
This way, **the best choice of hyper-parameters** was:

- **Problem 1:** $elite\_size = 30, mutation\_rate = 10\%, population\_size = 300$, order crossover and invert mutation;
- **Problem 2:** $elite\_size = 30, mutation\_rate = 10\%, population\_size = 300$, order crossover and displacement mutation.

Finally, it was decided to run the algorithm once more, but only using the best hyper-parameters obtained in the previous analysis, with the number of generations increased to 2000 and the pop-size ranged between 300 and 500. A population of 1000 individuals was also tested, but it took a long time and produced poor results.

For the ch130 benchmark, four distinct mutation rates were tested, including 0.01, 0.1, 0.2, and 0.3. However, because the best results were again obtained with 0.01 and 0.1, only 0.01 and 0.1 were explored for the tsp225 benchmark.

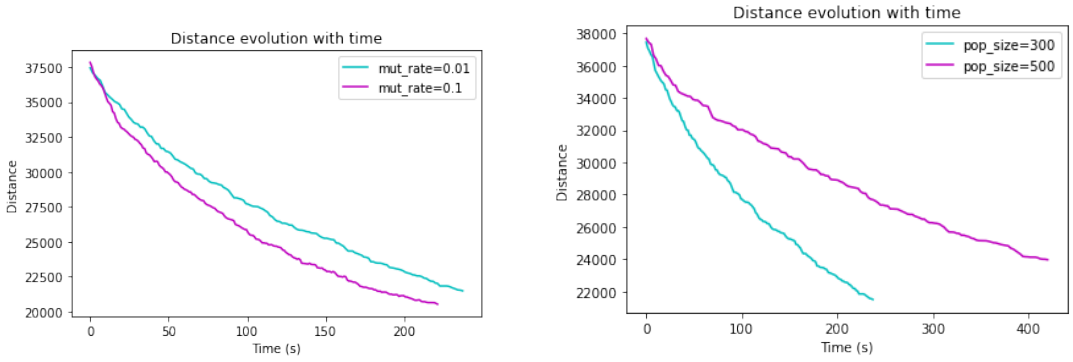
The results were the following:



(a) Varying mut_rate (b) Varying pop_size

Figure 9: Travelled distance VS time for ch130 dataset with varying mut_rate and pop_size.

The minimum distance achieved was 10075.2, which comparing with the optimal solution (6110) makes a relative error of 65%. In figure 9 it is possible to see that a smaller population size made the generations faster which resulted in a quicker optimization. In terms of mutation rates, as compared to 0.01 and 0.1, the higher ones (0.2 and 0.3) stopped enhancing the solution too quickly. This supports the idea that the mutation rate should not be too high, because an increased mutation rate makes it harder to preserve good solutions in the population.



(a) Varying mut_rate (b) Varying pop_size

Figure 10: Travelled distance VS time for tsp225 dataset with varying mut_rate and pop_size.

In the tsp225 dataset, the best solution obtained was 13715.6, which makes a relative error of 71%.

4.2 Ant Colony Optimization Algorithm

The Ant Colony Optimization Algorithms (**ACO**) is a probabilistic technique used for solving computational/ optimization problems, mainly ones that can be reduced to finding optimal paths through graphs. Here, artificial "ants" move through the graph, recording their paths and harvesting information, so that, in later iterations, other "ants" may be able to construct better and better solutions, using that information. This constructive aspect is precisely what differs **ACO**

algorithms from others which results in a more dynamic and adaptive way of solving such problems (even being able to adjust to changes in real time). It is heavily inspired in the behaviour of real ants, particularly in their method of directing others towards more favorable paths by realising pheromones while exploring the environment. In fact, this "pheromone release" is a major component of such algorithms, as we'll explore later.

Despite the overall good performance of the main algorithm several variations have been developed, with computational advantages. These are mainly achieved through:

- the limitation of its parameters / the limitation of the importance of the ants - **MMAS** and **ASrank**, respectively.
- the over prioritization of certain parameters in favor of others or of certain ants - **ACS** and **Elitist ant system**, respectively
- the subdivision of the problems in a recursive manner until finding a solution that is good enough - **Recursive ant colony optimization**.

In this paper it'll only be implemented the original algorithm of the **ACO** without any modification present. This is due to two reasons: firstly, the advantages that these modifications present can be achieved on the original algorithm through careful and precise choice of parameters. Lastly, each problem is its own and some algorithms, despite being better suited for specific situations, might be completely unadjusted to others. This way, by choosing the original algorithm we avoid this issue, since it is a more general and universal method of solving such problems.

4.2.1 Algorithm

The usual algorithm of the **ACO** is

while not terminated **do**
  generateSolutions()
  comparePaths()
  pheromoneUpdate()
  repeat
end while

There are three main steps in **ACO** algorithm:

- **generateSolutions()** - solutions are created in a stochastic way: at each node each "ant" has an array of different edges to move to, with different levels of pheromones and lengths - which will determine the probability of that path being chosen. This way, despite previous information of better solutions still influencing the results of later iterations, these will still be obtained in a stochastic way, allowing a more diverse list of solutions, as well as new paths, some of which might actually improve the good results that were already obtained.
- **comparePaths()** - solutions are compared and analyzed. It is verified if the level of optimization is already acceptable, and which paths are better (and which ones are going to be followed next, despite this part being primarily resolved by the next step). It is a optional step, mainly used by the more modern **ACO** algorithms.
- **pheromoneUpdate()** - the pheromone levels of each edge are updated. Mimicking what happens with real ants, where the pheromones deposited dissipate with time, in the **ACO** they are periodically updated, changing with each iteration. Adding to the dissipation, pheromones are also deposited by each ant that went through it. The amount added is inversely proportionate to the length of each edge, thus rewarding the shorter ones.

4.2.2 Formulas

As was previously explained, the **ACO** is a probabilistic iterative method of finding solutions. Because of this there are two main formulas for this algorithm: one for the **edge selection** and another for the **pheromone update**. Both of this formulas have different parameters that can be altered in accordance to our preferences/priorities: if we either want to value more the length of the edges or the amount of pheromones, how much should the pheromones dissipate, etc.

Edge selection Since each ant constructs the solutions in a stochastic manner, in order to progress through the graph, it is important to determine the probability of crossing a particular edge. This probability mainly depends on its length and level of pheromones. This way, let the ant be at node k and let $\{y_1, ..., y_n\}$ be the list of possibles nodes to transverse too (in the **TSP** each node can only be crossed once). The probability of going from x to y_i is

$$p_{xy_i} = \frac{(\tau_{xy_i})^\alpha (\eta_{xy_i})^\beta}{\sum_{i=1}^{n} (\tau_{xy_i})^\alpha (\eta_{xy_i})^\beta}$$

where τ_{xy_i} is the amount of pheromone deposited, η_{xy_i} is the desirability to cross this particular edge (usually, $\eta_{xy_i} = 1/d_{xy_i}$, where d_{xy_i} is the distance between the two nodes), α is a positive parameter used the control the influence of the amount of pheromone and β to control the influence of η_{xy_i}. Evidently

$$\sum_{i=0}^{n} p_{xy_i} = 1.$$

Pheromone Update After each ant has constructed a solution the pheromone levels need to be updated, so that in later iterations better solutions may be constructed. Firstly, the levels that were previously on the edge partially dissipate. Then, more pheromones are added, associated to the ants that passed through it. This way, optimal edges see their pheromone levels increased, while non-optimal decrease. An example of a pheromone update formula, assuming that n ants cross it, for the edge between x and y is

$$\tau_{xy} \longleftarrow (1-\rho)\tau_{xy} + \sum_{i=1}^{n} \Delta\tau_{xy}^{k}$$

where ρ is the **pheromone dissipation coefficient** and $\Delta\tau_{xy}^{k}$ is the amount of pheromone deposited by the k'th ant. The value of the latter parameter is inversely proportionate to the length of the edge (weight):

$$\Delta\tau_{xy}^{k} = \frac{Q}{L_k}$$

with L_k being the length of the edge and Q a constant that can be adaptable.

4.2.3 ACO on the Travelling Salesman Problem

Since the **TSP** is a problem that can be reduced to finding optimal paths in graphs, it's natural that the **ACO** would be eventually applied to it. In fact, even from the very creation of the **ACO** that there has been a close relationship of it with the **TSP**. In fact, it was initially applied to the **TSP** and subsequent improvements are always tested against it to measure their quality. This is due to

- **i)** the ACO being easily applied to it;
- **ii)** it is an NP-hard optimization problem, perfect for the **ACO** algorithms;
- **iii)** it is a standard test-bed for new algorithmic ideas and a good performance on the TSP is often taken as a proof of their usefulness.

The application of the **ACO** on the **TSP** is rather intuitive: not onky because of the reasons just presented, but also because the TSP (in our case, symmetric) is reduced to a problem with a complete symmetric graph, which makes it that much easir. This way, for each pair of nodes (citys) (i, j) a pheromone strength τ_{ij} is associated to it, where τ_{ij} (t) is a numerical information which is modified during the run of the algorithm. In our case, we always have $\tau_{ij} = \tau_{ji}$. Following the algorithm presented previously, the implementation of the ACO to the TSP is as follows: After creating m instances of ants, each of these is placed on a randomly chosen city, after which it iteratively applies at each city a state transition rule, choosing an unvisited city, following the formula above, and repeating the process at each new node, until every city is visited. Thus, each ant constructs its own solution, depositing pheromones along the edges crossed. After the pheromones are correctly updated a new iteration begins and the process is repeated. At each iteration the best solution is analysed to check if it already respects the solution thresh hold (or if it already is the optimal solution).

4.2.4 Experimental Setup

Both data sets had a list with the coordinates of each city, so we only had to create a matrix with the distances between them. Since this is a complete and symmetric graph (typical of symmetric **TSP**), the matrix will be $n \times n$ and symmetric, where n is the number of cities. There will be no 0 entries in the matrix, since the cities are all different and the main diagonal will have very large numbers to avoid division by 0. The function we used is the following:

```
AntColonyOptimizer(ants=n, evaporation_rate=p, intensification=m, alpha=c1, beta=c2,
                                beta_evaporation_rate=s, choose_best=l)
```

it has several different parameters that are important to understand:

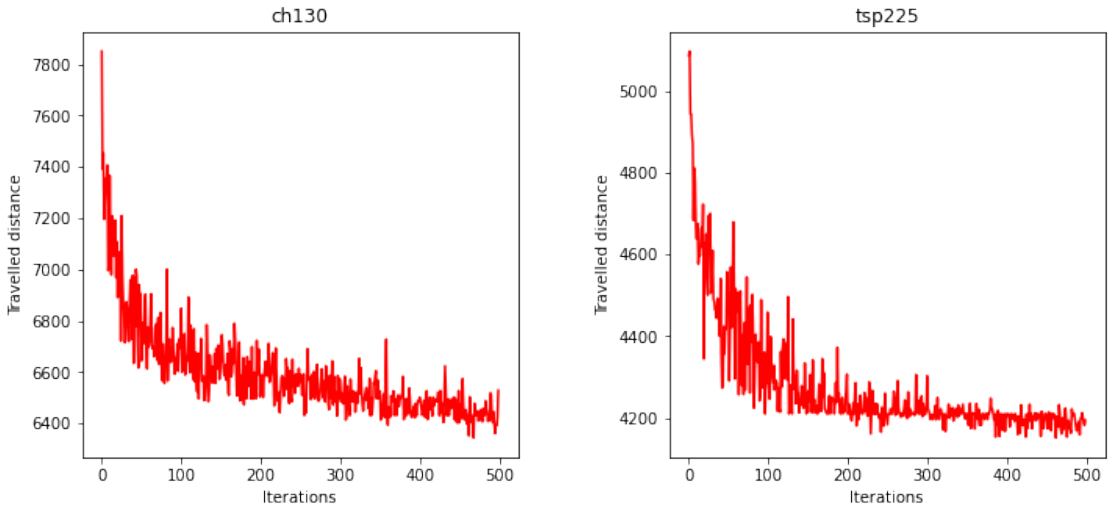
- $ants = n$ - the number of ants that are going to run through the cities, creating several solutions. The bigger the number, more solutions will be generated, more probable of reaching better/optimal solutions. However, a large enough number will result in a over saturation of the graph with pheromones, which might result in the stagnation of results.
- $evaporation\_rate = p$ - the percentage of the pheromones on each edge that evaporates after each iteration. The bigger the percentage, the less dependant later iterations are of the previous, so the solutions are more random.
- $intensification = m$ - constant added to the best path found (until that iteration). The bigger the most later iterations will resemble that solution. However, it also leads to lack of diversity in the solutions, making it harder to find the optimal (or better) paths.
- $alpha = c_1$ and $beta = c_2$ - constants powered to the pheromones and inverse of the length of each edge, respectively, in the formula for the probability of crossing each edge. The bigger this variables are the more weight their respective bases will have to the probabilities. However, if they are very different it will render the smallest one (and, consequently, its base) useless in the determination of the probabilities, so it needs to be balanced out.
- $beta\_evaporation_rate = s$ - totally optional parameter used to reduce the value of the $beta$ throughout the whole process. This way, although at the beginning of the process smaller edges are prioritized, as the pheromone levels start to increase, so their importance will. This allows for a more diverse construction of solutions.
- $choose\_best = l$ - probability of each ant to choose the best path already known. The bigger it is the smaller are the the array of different solutions, despite being better on average.

This function creates the ants and sets up the functions for the edge selection and pheromone update of the edges which is later fitted to our specific problem: to our matrix of distances and the iterations desired. Because of the randomness of this algorithm a large number of iterates (500) will be chosen for the runs of the function, so that the actual quality of the algorithm may be visible. Here, we're not only going to test the algorithm with regular values on the parameters (to see how good it is in a "normal" setup), but also going to test the algorithm against the data sets with small variations on its parameters (and compare the results). This way, we're going to understand how the parameters work (and the relations between them) in an attempt to achieve the optimal solution. It is important however to highlight that there is a correlation between all these parameters that might not get explored since we're going to vary them individually, while the others are constant.

4.2.5 Results and discussion

We are going to apply now the function to both data sets, storing the solutions obtained (at each iterate) as well as the best found yet, changing routinely its parameters not only to obtain different sets of solutions and approximations, but also to verify which ones are more important and have good correlations. It is important, however, to note that these parameters should be handled carefully and even small changes in its values may result in a complete lack of good solutions, especially if changed with others whom they have no synergy with. Even more than that (and more probable) is the stagnation of the solutions: even though the edge selection is stochastic, overvaluing the parameters may cause the probabilities to approximate to 1 (and 0 for the other edges) and consequently every ant builds the same solution, rendering the iterations useless, since they won't create any new solutions.

Firstly, we'll apply the algorithm with regular parameters to both data sets to check if it already constructs good enough solutions, and if the precision is good (that is, if the variation between iterates is large or slim):



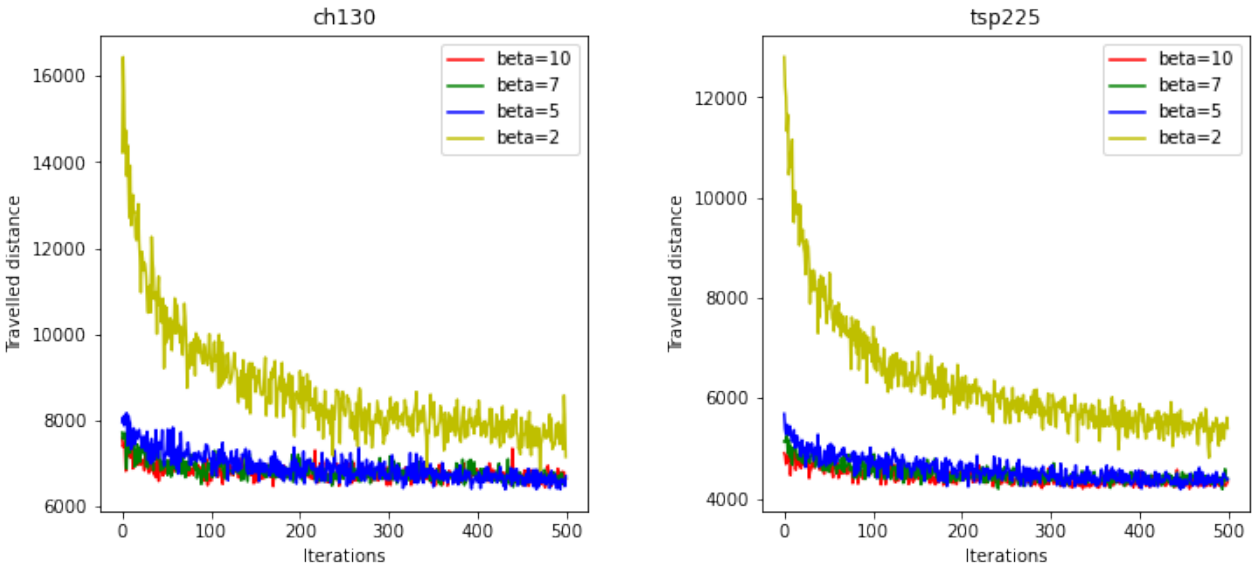
(a) ch130 dataset

(b) tsp225 dataset

Figure 11: Travelled distance VS Iterations for both data sets

The constructive aspect of the algorithm is visible in both graphics since as more iterates occur, better solutions are obtained. However, the model appears to fit the $tsp225$ data set much better: in spite of similar best solutions overall (more or less 300 units of distance different from the optimal), the second graph exhibits a logarithmic behaviour - normal within this types of problems and algorithms. Not only this, but the interval of solutions of the first problem remains very large, independently of the iterate we're on. This might be due to a lack of pheromone concentration, which can be solved by either creating more ants, or by lowering the evaporation rate. On the other hand, the second algorithm seems balanced in this aspect, needing smaller tweaks for it to improve.

Before fixing this problem, however, we'll first solve the large values obtained in the first iterations. In order to do this we'll vary the $beta$ parameter (since it will augment the attractiveness of each edge - associated to its length - which, in turn, will result in the prioritization of the smaller edges, from the very beginning): we'll calculate the best solutions for 4 different values of this parameter, combining all of them in the same graph so that we can compare them better. This way, it'll be clear which value is better for each data set, which will allow us to obtain better solutions from here on out:



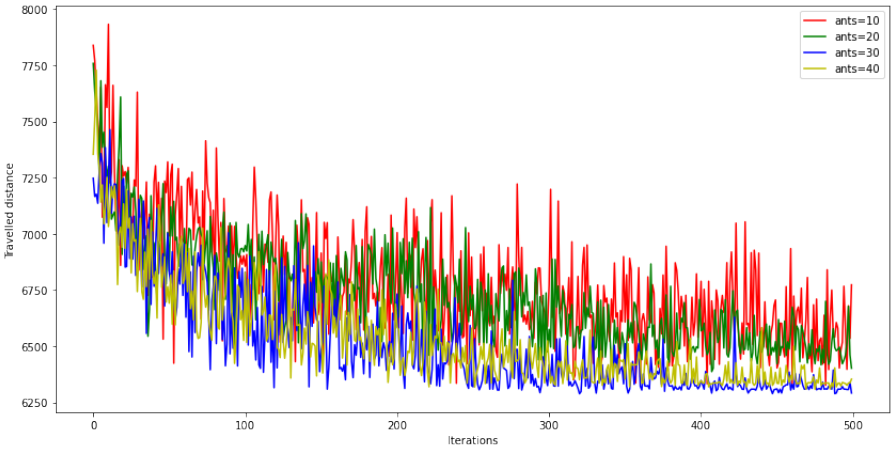
(a) ch130 dataset

(b) tsp225 dataset

Figure 12: Travelled distance VS Iterations for both data sets with varying $betas$

Only for $beta = 2$ we have the large values for solutions on the first iterations (and for the others as well), while for the other three values we see no difference in the results. Because of this we'll settle for $beta = 7$ since it is the middle

value used, although any other value on the $[5, 10]$ interval (at least) would be good. Now we'll check which number of ants creates the better solutions (and is better from a computational point of view):



Despite all algorithms beginning in a similar fashion there is a notorious difference in the values at higher number of iterates: the lower number of ants produces worse solutions (which is expected, since less paths are taken). However, the best results are achieved with only 30 ants instead of 40. This might have two reasons behind it:

- **1)** Since it is a probabilistic algorithm it might just be an instance where the third algorithm achieved better solutions. However, the large number of iterations discredits this reason.
- **2)** The more ants, the more pheromones are deposited in the edges that are crossed. In the case of the 40 ants the amount of pheromones might be high enough to cause a positive feedback situation that induces stagnation of the results, since the ants are always crossing the same edges. This way, the results don't vary beyond a certain point, which (if it's not optimal enough) may result in the worse solutions.

It is important to note, although, that the difference is marginal and not very determinant for the final solution. The same exact thing is visible in the $tsp225$ data set as it is visible in the figure 20. To check if in fact the best number of ants was around the 30 mark we plotted the best solution obtained by 100 iterates of the algorithm (on the $ch130$ data set) against the number of ants used in the search of such a result:

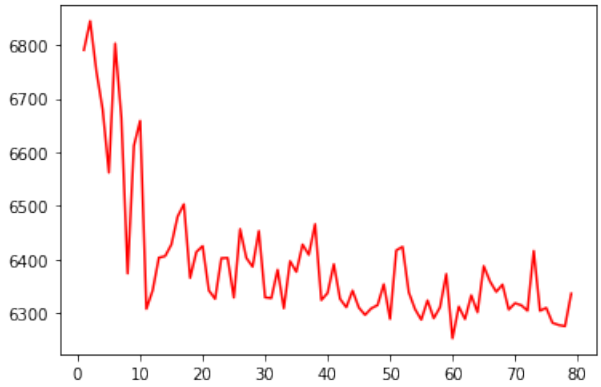
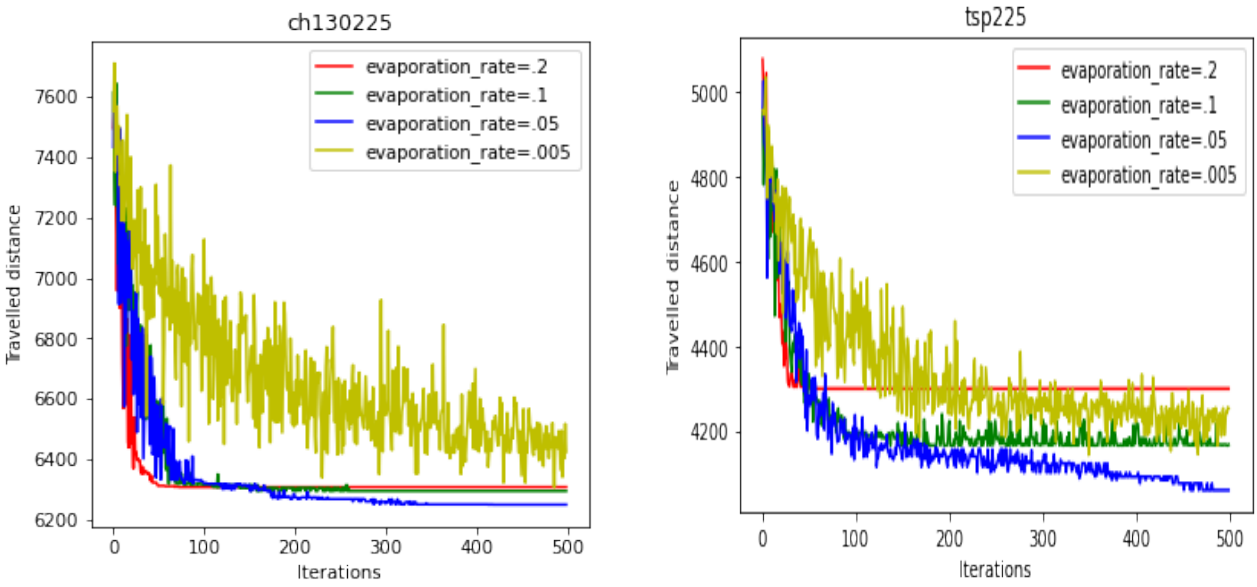


Figure 13: Best solution obtained vs number of $ants$ for the $ch130$ data set

It is apparent that for smaller number of ants the best solutions obtained are quite far from the optimal solution, and from around the 20 thresh hold (despite a small improvement) there isn't a big difference between algorithms, specially considering the computational resources that the bigger numbers require. It is important to note that this graphic is flawed (since it does not take into account the iterates but the best result obtained, which is even more random) and should not be used individually as proof of the importance of the number of ants. However, this graphic allied to the two previous shown does indicate a tendency for the number of ants not mattering from a certain point, at least with the parameters being used. Because of this we'll settle with 30 ants from now on, on any algorithm used.

Now we'll analyse the influence of the $evaporation\_rate$ parameter on the solutions:



(a) ch130 dataset

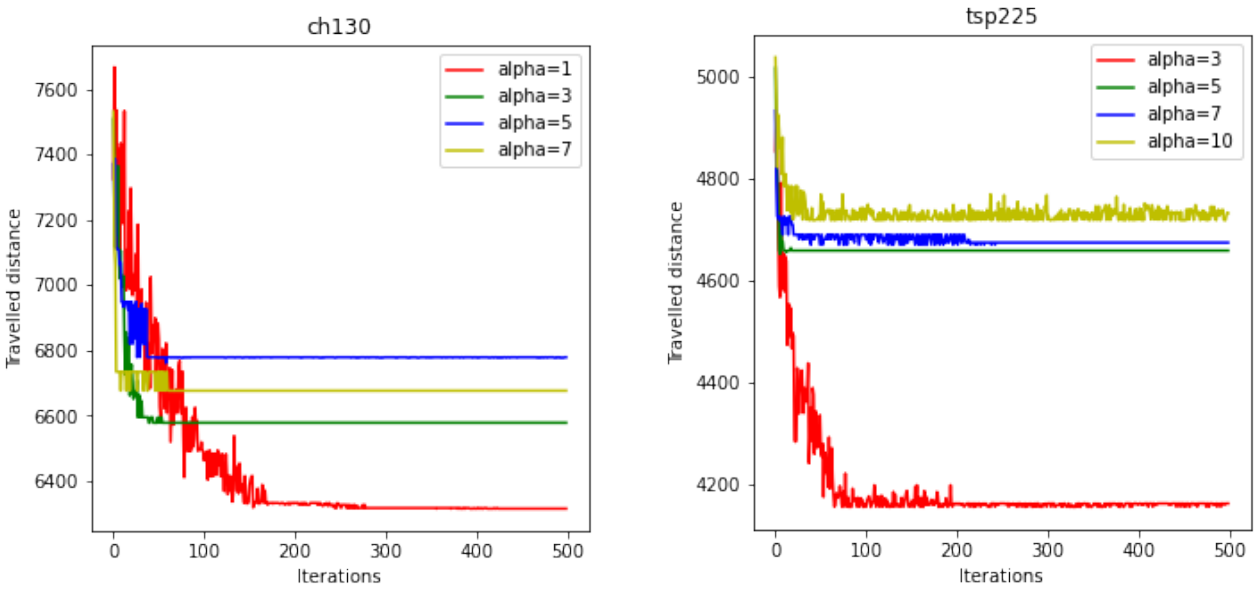
(b) tsp225 dataset

Figure 14: Travelled distance VS Iterations for both data sets with varying $betas$

Here we can see how much of a factor the evaporation rate actually is: both small and big values present issues, thus being hard to balance this parameter, in an attempt to achieve the best possible solutions. On one hand, for the smallest value, $evaporation\_rate = 0.005$, the solutions obtained are the farthest from the optimal (and the less precise): since almost all pheromones present at all edges remain after each iterate (including the worst ones), all of them will have high probability of being chosen, creating a cycle of edge crossing and pheromone release that won't end since they aren't cleared. Because of this there will be a large sample of solutions obtained, that won't shorten with time, creating a varied (albeit imprecise and inexact) group of solutions. On the other, for all other values of the parameter it occurs the first case of stagnation of solutions, caused by the positive feedback phenomenon present in the **ACO** algorithms. Since the pheromones evaporate on a larger scale after each iterate, only the best edges will have large enough concentration to actually influence the probabilities of crossing, which will result in more ants crossing these edges, and more pheromones being released, and so on. This will result in only a select few edges actually being traversed and, consequently, the stagnation of the results. Despite all this, it is visible in both algorithms that the best solutions occur when $evaporation\_rate = 0.05$.

Note: These graphics also illustrate that the second data set is much better suited for the **ACO** algorithm used, something visible from the very first graphics here presented. As it was relayed at the beginning of this chapter, different data sets have different properties, and different optimal ways to approach them.

Finally, we test for the values of the $alpha$ parameter:



(a) ch130 dataset

(b) tsp225 dataset

Figure 15: Travelled distance VS Iterations for both data sets with varying $alphas$

It is observable that despite being a crucial parameter, the setup of the function (with all the other pre-established values for the other parameters) renders the action of the $alpha$ almost useless, since it stagnates for any value of it. Even though for smaller $alpha$ values the solutions found are actually quite good, this could be due to stagnating later in the iterations (which is, obviously, still preferable and optimal). Because of this, despite finding good solutions and at least understanding how it functions, we still want to check how the algorithm would do with different values of parameters. This way: we not only lowered the $beta$ value back to 1 but also increased the number of ants and the evaporation rate (to impede stagnation) in order to enhance the importance of the pheromones in the algorithm. These were mainly tested in the $ch130$ data set since it presented more problems. However, independently of the combinations of values of the parameters we tried, stagnation would almost always occur (sometimes sooner, sometimes later):

- Firstly, we tried to just put a bigger number of ants with a large $alpha$ value, however due to the still small evaporation rate this didn't produce the results wanted - figure 21
- Then, we tested various $evaporation\_rate$ values against the same alpha and ants values, with no avail: it still stagnated every time - figure 21.
- Finally, we tried to change the other non-main parameters to see if it would yield any result but it stagnated as well and very far from the optimal value - figure 22.

Other combinations of variables were tested and since they didn't yield great results (mainly stagnation after a few iterates) we didn't find them noteworthy of including in this paper (although they can be checked at the jupyter file). Being a meta heuristic algorithm it is expected to have a large array of solutions close to the optimal one. This was, in fact, achieved with different sets of values for the variables for both data sets - achieving results 200 units of distance close to the optimal ones. Because this type of algorithms (and, specifically, the **ACO**) are probabilistic there is a degree of randomness associated to the results. The same set of parameters could produce great solutions on one instance, and the very next instance yield terrible results. To shield us from this type of errors we ran every single case at least three times.

4.3 Nearest Neighbor Algorithm

The Nearest neighbor is an heuristic algorithm and is one of the first algorithms that comes to mind while trying to solve the traveling salesman problem (TSP). Nearest neighbor algorithm requires a salesman to arrange a tour of cities that is of minimal length. In this heuristic, the salesman starts in one city, then moves on to the next closest city, and so on, taking care not to visit the same city again. The salesman returns to the starting city after visiting all of the cities. Nevertheless, the heuristic algorithms can obtain good solutions but it cannot be guaranteed that the optimal solution will be found.

The steps are (repeated for all the cities):

- 1. Select a city as current city;
- 2. Find out the shortest edge connecting the current city and an unvisited city;
- 3. Set the new city as current city;
- 4. Mark the previous current city as visited;
- 5. If all the cities are visited, then stop;
- 6. Go to step 2.

One can obtain the best result by running the algorithm over again for each vertex and repeat it for n times. However, Rosenkrantz, Stearns, and Lewis' theorem reveals that this approximation approach may have substantially worse behavior:

Theorem : For all m-city instances I of the traveling salesman problem with triangle inequality,

$$NN(I) \leq \frac{1}{2}(\lceil log_2 m \rceil + 1)OPT(I) \tag{6}$$

Furthermore, for arbitrarily large values of m, there exist m-city instances for which

$$NN(I) > \frac{1}{3}(log_2(m+1)\frac{4}{3})OPT(I) \tag{7}$$

where OPT(I) represents the length of the optimal tour for all m-city instances I.

This theorem's major point can be summarized as follows: $NN = \infty$, is not a promising assurance. As a result, $m \to \infty \implies$ NN(I) $\to \infty$. [10]

Finally, the Nearest neighbor algorithm definitely has a lot of room for improvement, once the solution produced by the algorithm is not the optimal one. As the algorithm is a greedy algorithm it misses out some of the shorter routes, which can be detected by human insight easily. So, the nearest neighbour algorithm does not give the feasible solution. Since the tours quality might depend on the starting city chooses, a better result can be obtained by repeating the procedures for different starting city, which is going to be done in this project.

4.3.1 Two Directional Nearest-Neighbour Algorithm

A different modification of the Nearest neighbor algorithm is proposed in this section. The steps of the algorithm are as following:

- 1. Choose an arbitrary vertex in the graph;
- 2. Visit the nearest unvisited vertex to this vertex;
- 3. Visit the nearest unvisited vertex to these two vertices and update the end vertices;
- 4. Is there any unvisited vertex left? If yes, go to step 3;
- 5. Go to the end vertex from the other end vertex.

This algorithm generally gives better results than the original Nearest Neighbor algorithm, as one will see when the the results are later commented on. [13]

4.3.2 IorNN

Another modification can be done to the Nearest neighbor algorithm. IorNN algorithm is a new hybrid method based on the NN and 2NN algorithms. By combining the NN and 2NN algorithms with a contribution ratio k, the IoNN is able to take benefit of both. This algorithm starts with a node that is randomly picked. The NN or 2NN method is then used to select the next nodes from the list of unvisited nodes. If the NN method is used to select a number of nodes, the remaining unvisited nodes will be selected using the 2NN algorithm, and vice versa.

The parameter k determines the contribution ratios of the algorithms. The suggested algorithm is the 2NN algorithm if $k = 0$; similarly, the proposed algorithm is the NN algorithm if $k = n$. When k is in the range $(0, n)$, the NN algorithm is used for the first k nodes and the 2NN algorithm for the remaining nodes. [13]

4.3.3 Results

Nearest Neighbor Algorithm (NN) To test the Nearest neighbor algorithm, the distance used was the Euclidean distance, since the EDGE_WEIGHT_TYPE of the problems is in EUC_2D. So, one implemented a function that receives the number of vertices of the dataset and the dataset. In this case, the function was tested for ch130 (referred to as problem1) and tsp225 (referred to as problem2). For the case of problem1 the code was run 130 times, which corresponds to the number of problem vertices, where in each iteration the initial vertex changed, that is, in the end all problem vertices were used as initial vertices. The same was done for problem2, running 225 times. The output of the Nearest neighbor function then returns the average minimum distance obtained after testing the function for each different starting vertex, along with the average time the algorithm takes to run. In addition the path that corresponds to the "best" is returned, since it is the path with the shortest distance traveled. For ch130 dateset the results obtained are shown below:

```
* Nearest Neighbor Algorithm *
Minimum distance travelled: 6651.164391740215 and running time of: 25.7793 ms. The starting node,
    in this case, is the 76 node
The average travelled distance of the 130 iterations is: 7212.822859218781 with the respect
    average running time of: 27.681199999999997 ms
```

And for the tsp225 dataset:

```
* Nearest Neighbor Algorithm *
Minimum distance travelled: 4536.261133090816 and running time of: 97.0707 ms. The starting node,
    in this case, is the 20 node
The average travelled distance of the 225 iterations is: 4740.772121800176 with the respect
    average running time of: 106.56102088888889 ms
```

As it can be seen above, the minimum travelled distance obtained for the dataset ch130 was approximately 6651 and the optimal one is 6110. For the tsp225 dataset the minimum distance obtained was approximately 4536 and the optimal one is 3916.

So, in both cases, the distance travelled obtained was reasonably close to the optimal distance. This algorithm provides a sub optimal solution, which means that the path obtained together with the distance travelled are not the optimal ones. Since it is a deterministic algorithm, which is, it is purely determined by its inputs, where no randomness is involved in the model, it was only necessary to run the algorithm once for each initial node. With the same inputs the algorithm always gives the same outputs. This is a simple algorithm that is easy to implement and doesn't take too much time running, so it can be an option when talking about getting a suboptimal solution.

Two directional Nearest Neighbor Algorithm (2NN) As with the Nearest neighbor algorithm, the modification of the NN algorithm was applied to the ch130 and tsp225 datasets. The only difference from one to the other, is that, in this algorithm, the vertices that will be used to calculate the distances between cities are the end vertices.

The results obtained for ch130 dateset are:

```
* Two Directional nearest Neighbor Algorithm *
Minimum distance travelled: 6625.5961500967105 and running time of: 35.0204 ms. The starting node,
     in this case, is the 63 node
The average travelled distance of the 130 iterations is: 6949.04703851558 with the respect average
     running time of: 43.47977692307692 ms
```

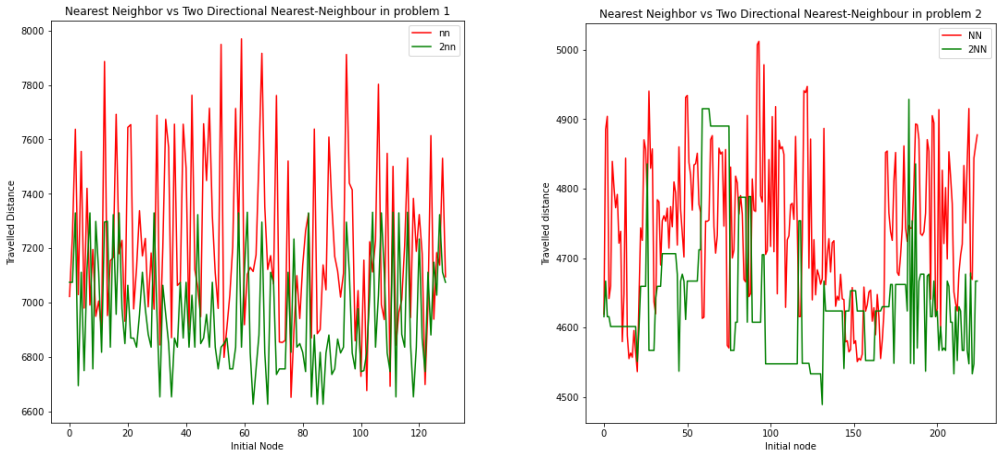
And for the tsp225 dataset:

```
* Two Directional Nearest Neighbor Algorithm *
Minimum distance travelled: 4488.758685550873 and running time of: 213.577 ms. The starting node,
    in this case, is the 131 node
The average travelled distance of the 225 iterations is: 4643.358113077188 with the respect
    average running time of: 180.88190533333335 ms
```

As it can be seen above, the minimum travelled distance obtained for the dataset ch130 was approximately 6626 and the optimal one is 6110. For the tsp225 dataset the minimum distance obtained was approximately 4489 and the optimal one is 3916.

So, in both cases, the travelled distances obtained were slightly better than the ones obtained in the Nearest neighbor algorithm, which are now a bit closer to the optimal distance. Once again, the path obtained together with the distance travelled are not the optimal ones. In terms of the running times, as one might expect, this algorithm takes more time to run than the other one, since in each iteration of the algorithm it compares the distances between the two end nodes and the nearest city to each one.

Nearest Neighbor vs Two directional Nearest Neighbor Algorithm Below, in figure 16 are the plots that show the the difference between the travelled distances for different starting nodes in each algorithm for each problem.

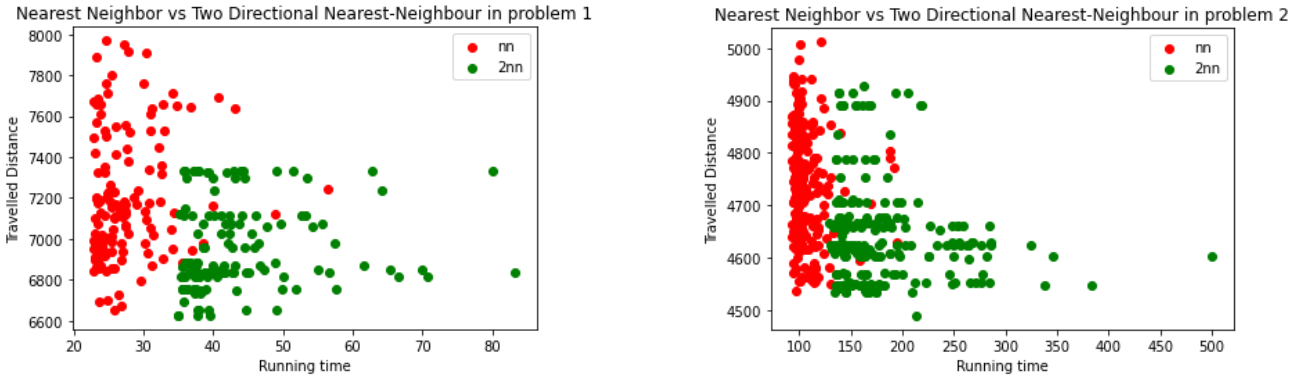


(a) ch130 dataset (b) tsp225 dataset

Figure 16: Travelled distance VS Initial node with the two algorithms NN and 2NN

Taking a closer look to the plots above, the behaviour of the NN (represented in red in the plots) is worse than the behaviour of the 2NN (represented in blue). With the ch130 dataset the minimum travelled distance obtained was 6651, approximately, with the Nearest neighbor algorithm. On the other hand, the minimum travelled distance obtained with the modified version of this algorithm was 6626, approximately. With the tsp225 dataset the minimum travelled distance obtained with the NN algorithm was 4536, approximately, and with the modified version was 4489, approximately. As it can be seen in the plots above, with both datasets, the oscillations are larger with the NN algorithm than with the other one, which means that in each iteration the travelled distance, in the case of NN, is further away from the optimal distance than in the case of 2NN. This happens because the modified algorithm uses more information for the construction of the path and distance traveled, which means that, in this case, when one want to update the path it will be seen which of the unvisited nodes is closer to each of the extremities of the path already built and then update the path with the city that is closer to either one of the nodes. That is, there is a comparison of distances between two nodes already visited and other two not yet visited, which does not happen in the NN, where one only compare distances between one node already visited and another that has not yet been visited.

Taking a look to the figure 17 below, one can observe the differences between the two algorithms when talking about the travelled distances versus the time that the algorithm takes to run, for different starting vertices.

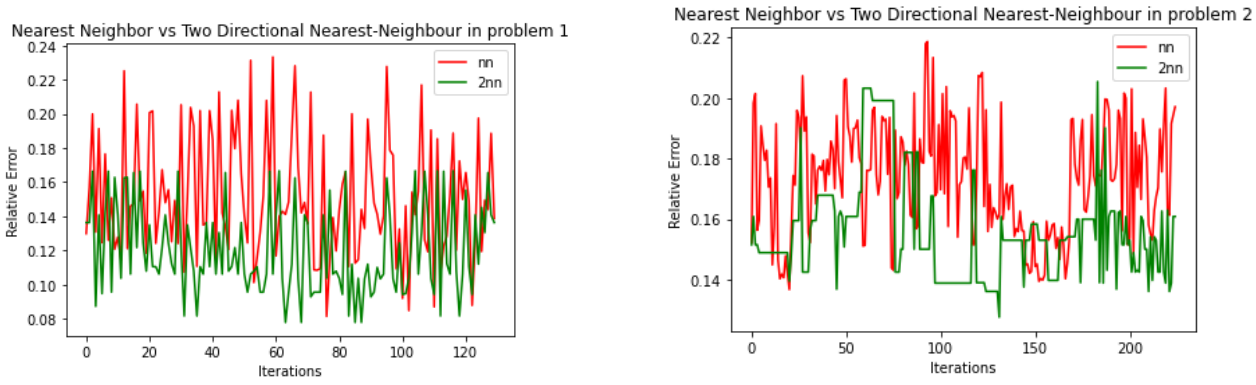


(a) ch130 dataset (b) tsp225 dataset

Figure 17: Travelled distance VS Running time with the two algorithms NN and 2NN

With the ch130 dataset the minimum and maximum running times obtained were 22.8306 ms and 56.4864 ms, respectively, with the NN algorithm. On the other hand, the minimum and maximum running times obtained with the modified version of this algorithm were 35.0198 ms and 83.2853 ms, respectively. With the tsp225 dataset the minimum and maximum running times obtained with the NN algorithm were 92.7407 ms and 194.2469 ms, respectively, and with

the modified version were 129.5476 ms and 209.576 ms, respectively. As it can be seen in the plots above, in the case of 2NN it is noticeable that the algorithm, in general, takes, sometimes, twice as long to run than the NN algorithm. However, since it produces better results and the running times are not that high, if one have to choose between the two algorithms, will clearly choose the 2NN. Thus, the "best" solution is the minimum distance obtained for each dataset with the 2NN, once the running times of this algorithm are low. Therefore, for the ch130 dataset the "best" travelled distance is approximately 6626 and for the tsp225 dataset is 4489. However the "best" path and travelled distance obtained with this algorithm is a suboptimal solution and is still far away from the optimal ones.



(a) ch130 dataset

(b) tsp225 dataset

Figure 18: Relative Error VS Iterations with the two algorithms NN and 2NN

As it can be seen in figure 18 the relative errors of the NN algorithm are worse than the 2NN algorithm, with both algorithms. The NN algorithm, generaly, has higher percentages of error than the other one, which is well visible in both plots. Considering the minimum distances obtained for each algorithm, with both datasets, the relative error obtained with the NN algorithm was 13.67% with the tsp225 dataset and 8.13% for the ch130 and with the 2NN algorithm was 7.79% for the ch130 dataset and 12.76% for the tsp225, which shows, once again that the 2NN algorithm performs slightly better.

If the datasets were tested with the IorNN algorithm presented before, in terms of accuracy and ability to find better solutions, it would have performed better or equivalent to the NN and 2NN algorithms. The IorNN has the potential to be used as a construction heuristic in other techniques because it can reduce the percentage error of the NN and 2NN algorithms.

5 Conclusion and future work

The Travelling salesman problem can be solved using heuristic algorithms such as Nearest Neighbor, which provide suboptimal results, and meta heuristics algorithms such as Genetic algorithm and Ant Colony. To this problem, this project compares the Genetic algorithm, Ant colony, and Nearest neighbor method (and a modified version of it). The comparison is based on the number of cities visited by the algorithms, and then the distance travelled and the execution time are used to determine which one is the best algorithm.

When talking about the Nearest neighbor algorithm, every node is serving as a corresponding initial node in relation to the next closest node, implying that nodes are not independent of one another. The solution produced by this algorithm is not the optimal one. As the algorithm is a greedy algorithm it misses out some of the shorter routes. So, the Nearest neighbour algorithm does not give the feasible solution. The exact same thing happens with the modified version of NN, despite presenting, in general, better results that the original version of it, the solutions obtained continue not be the optimal ones.

The Ant Colony Optimization algorithm used yielded good results, particularly for the second data set. The constructive approach used allowed us to not only obtain better and better solutions with the changes being made, but also to understand the importance and influence of each parameter in the construction of the solution. Because of this we were able to identify that, despite the pheromones still largely improving the solutions, the better solutions were obtained when a bigger focus was drawn on the attractiveness of the edges. Our inability to obtain great results using the pheromones as the main contributor might have been caused by either a poor balancing job with the parameter values, or two data sets that were actually better suited for the other method. In spite of this, we obtained good to great solutions. Better ones could be obtained with more tuning of the values, and with greater computational resources.

The results obtained with the Genetic Algorithm were unsatisfactory since the relative errors were too big. To improve the results obtained, it would be beneficial to do a narrower search on the hyper parameter values in future work, by trying combinations of values near the best solution so far, which was not achievable due to computational time constraints. Indeed, when the hyper-parameter values were fine-tuned a second time, the results were noticeably better. Combining the nearest neighbor heuristic with the genetic algorithm when selecting the initial population is another intriguing option. Rather than picking the initial population at random, use a nearest neighbor solution.

All three algorithms used are commonly applied to these types of problems so good results were expected, mainly by the ACO and the Genetic algorithm, followed closely by the NN (and 2NN). This was in fact what happened, despite the poor performance of the Genetic algorithm: both the ACO and the NN algorithms achieved good results, close to the optimal solutions, with a slight advantage to the first one - which was expected since it is a non-deterministic meta heuristic that can be improved upon. As far as the Genetic algorithm is concerned, in spite of not showing the results intended, the work done throughout the project was really promising, showing that with more computational resources and a narrower search of the hyper parameters, good to great solutions would be eventually obtained. Finally, it is important to note that there are a variety of TSP algorithms that may be compared to these to determine which method produces the greatest optimal outcomes under a specific set of conditions and data.

References

[1] [n.d.]. *Genetic Algorithms - Mutation*. https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm

[2] Otman Abdoun, Jaafar Abouchabaka, and Chakir Tajani. 2012. Analyzing the performance of mutation operators to solve the travelling salesman problem. *arXiv preprint arXiv:1203.3099* (2012).

[3] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2022. *Introduction to algorithms*. MIT press.

[4] Kusum Deep and Hadush Mebrahtu. 2011. Combined mutation operators of genetic algorithm for the travelling salesman problem. *International Journal of Combinatorial Optimization Problems and Informatics* 2, 3 (2011), 1–23.

[5] Ahmed Gad. 2018. *Introduction to Optimization with Genetic Algorithm*. https://towardsdatascience.com/introduction-to-optimization-with-genetic-algorithm-2f5001d9964b

[6] David E Goldberg. 2006. *Genetic algorithms*. Pearson Education India.

[7] Ahmad BA Hassanat and Esra'a Alkafaween. 2017. On enhancing genetic algorithms using new crossovers. *International Journal of Computer Applications in Technology* 55, 3 (2017), 202–212.

[8] Abid Hussain, Yousaf Shad Muhammad, M Nauman Sajid, Ijaz Hussain, Alaa Mohamd Shoukry, and Showkat Gani. 2017. Genetic algorithm for traveling salesman problem with modified cycle crossover operator. *Computational intelligence and neuroscience* 2017 (2017).

[9] Cornell University Jessica Yu. 2014. *Traveling Salesman Problem*. https://optimization.mccormick.northwestern.edu/index.php/Traveling_salesman_problems

[10] Gözde Kizilateş and Fidan Nuriyeva. 2013. On the nearest neighbor algorithms for the traveling salesman problem. In *Advances in Computational Science, Engineering and Information Technology*. Springer, 111–118.

[11] Clair E Miller, Albert W Tucker, and Richard A Zemlin. 1960. Integer programming formulation of traveling salesman problems. *Journal of the ACM (JACM)* 7, 4 (1960), 326–329.

[12] IM Oliver, DJd Smith, and John RC Holland. 1987. Study of permutation crossover operators on the traveling salesman problem. In *Genetic algorithms and their applications: proceedings of the second International Conference on Genetic Algorithms: July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, MA*. Hillsdale, NJ: L. Erlhaum Associates, 1987.

[13] Lilysuriazna Raya and Safaa Najah Saud. 2020. A COMPARATIVE STUDY BETWEEN THE NEAREST-NEIGHBOUR ALGORITHM AND ITS VARIANTS FOR SOLVING THE EUCLIDEAN TRAVELING SALESMAN PROBLEM. *PalArch's Journal of Archaeology of Egypt/Egyptology* 17, 10 (2020), 938–945.

6 Appendix

6.1 Genetic Algorithm

6.1.1 Correlation matrix of tsp225 benchmark

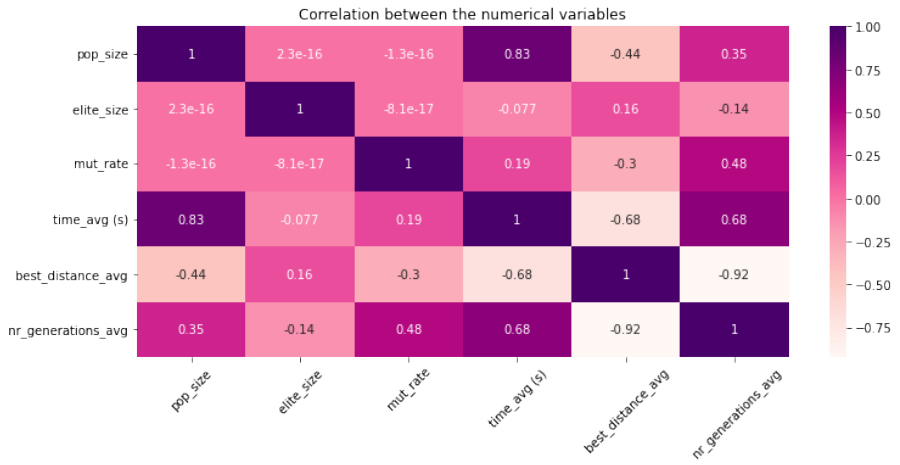


Figure 19: Correlation between the numerical variables, for problem 2

6.1.2 Statistical Measures, nr. generations=250

Statistical measures for each hyperparameter, when the nr. generations=250						
Hyperparameter	**Hyperparameter value**	**Min.**	**Max.**	**Mean**	**Median**	**Standard Deviation**
Crossover Type	**Order**	**15955.000**	37573.200	23127.204	21381.800	5333.527
	Cycle	19673.200	39590.000	32146.690	33299.300	5670.716
Mutation Type	**Swap**	16724.000	39239.800	27965.200	26200.600	6908.398
	Invert	**15955.000**	39590.000	27187.575	25625.800	7573.342
	Insert	16319.200	39100.800	27517.554	25231.800	6960.366
	Displacement	16769.400	39517.800	27877.458	25734.500	7187.516
Population Size	**80**	19441.400	39590.000	32370.575	34239.300	6116.485
	150	16813.400	36350.200	26354.978	23319.800	6593.676
	300	**15955.000**	34111.000	24185.288	21249.100	5995.383
Elite Size	**30**	16563.200	38035.600	26508.398	24688.800	6716.880
	60	**15955.000**	39590.000	28765.496	31969.000	7349.606
Mutation Rate	**0.0001**	19518.800	39590.000	30091.417	33033.400	6560.703
	0.001	18285.400	39517.800	29716.671	32029.000	6848.718
	0.01	**15955.000**	38529.400	27550.175	29076.200	7232.945
	0.1	16319.200	37670.800	23189.525	21402.900	5704.143

Table 1: Statistical measures for each hyperparameter, when the number of generations if fixed at 250 generations, for ch130 dataset (problem 1), for Genetic Algorithm

Statistical measures for each hyperparameter, when the nr. generations=250						
Hyperparameter	**Hyperparameter value**	**Min.**	**Max.**	**Mean**	**Median**	**Standard Deviation**
Crossover Type	**Order**	**16809.600**	34233.400	22386.848	20835.100	4720.340
	Cycle	21781.800	36421.800	30592.637	31583.500	4231.855
Mutation Type	**Swap**	18045.200	35548.600	26578.083	25351.400	5995.148
	Invert	17010.000	35828.200	26488.038	25345.300	6304.854
	Insert	17845.000	35834.400	26574.542	25240.900	5975.352
	Displacement	**16809.600**	36421.800	26318.308	25783.900	6209.671
Population Size	**80**	18941.600	36421.800	30533.347	32790.900	5205.452
	150	17711.000	33769.200	25448.647	23229.400	5512.613
	300	**16809.600**	31812.600	23487.234	21812.900	5248.999
Elite Size	**30**	17711.000	35105.200	25500.965	24110.800	5623.658
	60	**16809.600**	36421.800	27478.521	30635.600	6372.645
Mutation Rate	**0.0001**	18352.800	35858.400	28108.138	30722.200	5863.166
	0.001	18195.400	35832.000	28135.867	30969.900	6118.777
	0.01	**16809.600**	36421.800	26227.208	27703.700	6444.579
	0.1	17711.000	34987.000	23487.758	22128.400	4683.809

Table 2: Statistical measures for each hyperparameter, when the number of generations if fixed at 250 generations, for tsp225 dataset (problem 2), for Genetic Algorithm.

6.2 Ant Colony Algorithm

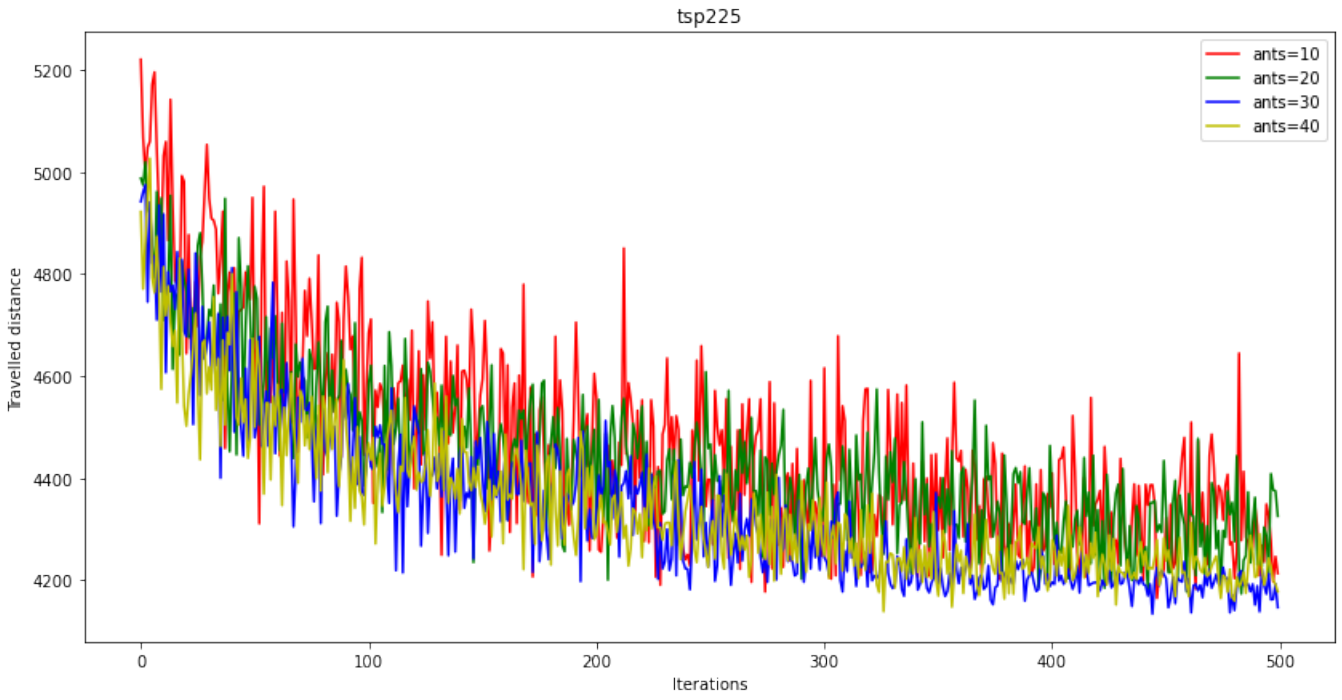
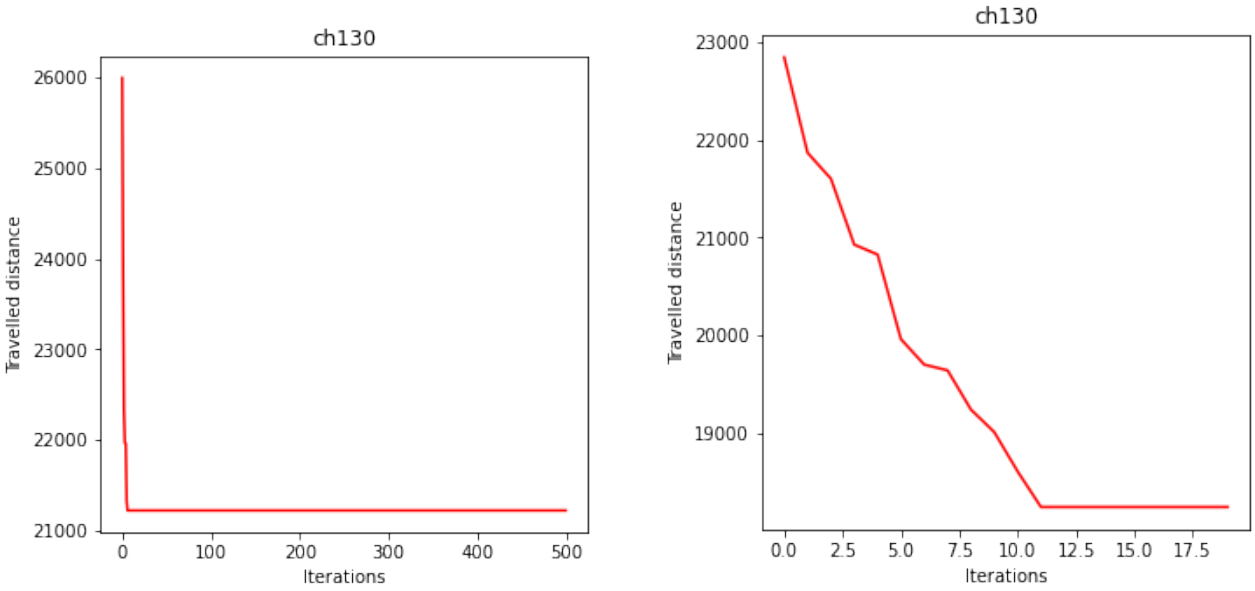


Figure 20: Travelled distance VS Iterations for the $tsp225$ data set with varying number of $ants$



(a) ch130 dataset

(b) tsp225 dataset

Figure 21

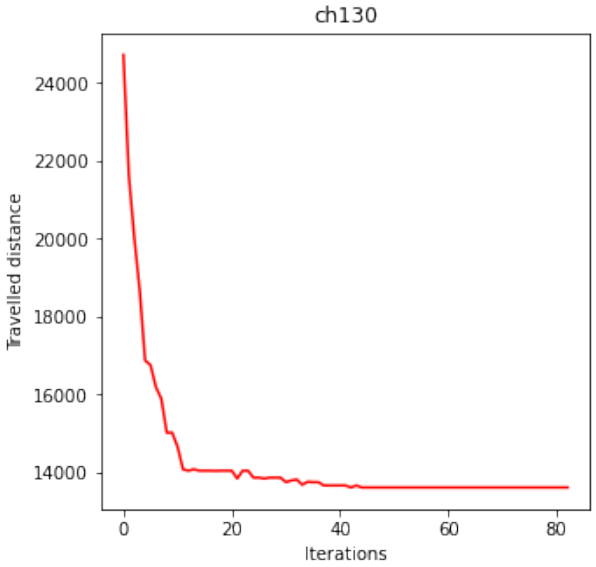


Figure 22: Travelled distance VS Iterations for the $ch130$ data set for varying $alphas$, $ants$ and $evaporation\_rate$