University of West Attica
Faculty of Engineering
Department of Electrical and Electronics Engineering
Sector of Telecommunications, Informatics and Signal Processing
TelSiP Research Lab

Solving the Travelling Salesman Problem with Real World data

Bachelor's Thesis
of
Elefterios Gryparis

Supervisor: Grigorios Koulouras
Assistant Professor

May 26, 2020

Abstract

Nowadays, Traveling Salesman Problem (TSP) attracts much attention due to the huge interest of Logistics Companies to reduce cost and maximize time efficiency to be more effective and competitive from opponents. From the late '50s, researchers try to define the problem and create algorithms for solving that NP-hard problem. In definition, the TSP is the challenge of finding the shortest route for a single vehicle to traverse in order to deliver goods to a given set of customers. The objective of the TSP is to decrease the total route cost. This bachelor's thesis will try to solve the TSP by using different algorithms to compare them, as a result, to find an optimal algorithm for it. Afterward, the problem will be transferred to Real World occasions by using Google's APIs for accessing data as polar coordinates and distances or durations from point to point. Finally, the application will export a map powered by Google with the optimal route plot on it, to make this more handheld for drivers.

Περίληψη

Στην σημερινή εποχή, το πρόβλημα του πλανόδιου πωλητή (TSP) προσελκύει μεγάλη προσοχή λόγω του τεράστιου ενδιαφέροντος που δείχνουν οι λογιστικές εταιρείες στο να μειώσουν το κόστος και να μεγιστοποιήσουν την αποδοτικότητα του χρόνου για να είναι πιο αποτελεσματικοί και ανταγωνιστικοί. Από τα τέλη της δεκαετίας του '50, ερευνητές προσπαθούν να ορίσουν το πρόβλημα και να δημιουργήσουν αλγόριθμους για την επίλυση αυτού του NP-Hard προβλήματος. Εξ ορισμού, το TSP είναι η πρόκληση της εύρεσης της συντομότερης διαδρομής όπου ένα όχημα καλείται να διασχίσει, προκειμένου να παραδοθούν αγαθά σε ένα συγκεκριμένο σύνολο πελατών. Ο στόχος του TSP είναι η μείωση του συνολικού κόστους διαδρομής. Αυτή η πτυχιακή εργασία θα προσπαθήσει να λύσει το TSP χρησιμοποιώντας διαφορετικούς αλγόριθμους με αποτέλεσμα να αποφανθεί τον βέλτιστο αλγόριθμο για αυτό το σκοπό. Εν συνεχεία, το πρόβλημα θα μεταφερθεί στην περίπτωση του πραγματικού κόσμου χρησιμοποιώντας API της Google για την πρόσβαση σε πραγματικά δεδομένα, οπως πολικές συντεταγμένες και αποστάσεις από σημείο σε σημείο ενδιαφέροντος. Τέλος, η εφαρμογή θα εξάγει έναν χάρτη που υποστηρίζεται από την Google με την λυση που εχει εξάγει το προγραμμα σχεδιασμένη σε αυτόν, ωστε να γίνει πιο φιλικο προς τον χρήστη.

Acknowledgments

I would first like to thank most, my thesis advisor, Dr. Grigorios E. Koulouras of the Department of Electrical and Electronic Engineering at the University of West Attica. He was always very helpful and guiding whenever I ran into a trouble spot or had a question about my research or writing. The experience of working next to a Professor at this academic level will always be one of the most exciting experiences in my career.

I would also like to thank Dr. Evangelos Zervas and Dr. Alexandros Alexandridis for the noticeable contribution that they made for my research. Without their passionate participation and their deep knowledge, this thesis project could not have been successfully conducted.

Finally, I owe a huge thanks to my family and to the people who stand by my side and strengthen me, from the beginning of this period of my life.

Author

Gryparis Eleftherios

Solving the Travelling Salesman Problem with Real World data

Gryparis Eleftherios
grypelev@gmail.com

February, 2019

Contents

List of Figures

List of Tables

List of Acronyms

TSP: Travelling Salesman Problem

sTSP: Symmetric Travelling Salesman Problem

aTSP: Asymmetric Travelling Salesman Problem

BF: Brute Force

DP: Dynamic Programming

SA: Simulated Annealing

GA: Genetic Algorithm

PMX: Partially Mapped Crossover Operator

OX: Order Crossover Operator

CX: Cycle Crossover Operator

API: Application Programming Interface

JSON: JavaScript Object Notation

GIS: Geographic Information System

Chapter 1

Introduction

1.1 Problem Definition

Since 1930, when Karl Menger first formulated the Traveling Salesman Problem (as the Messenger Problem) [1] researchers around the world have given a large amount of attention to this problem. The Traveling Salesman Problem (TSP) is the problem of finding the shortest route by which a person (or vehicle) may traverse a set of nodes only once and return to the starting node. As an example, a driver of a courier company has to serve packages to a set of customers located in different parts of town and returns to the depot whence he began. The schedule of the prioritization has to be efficient, minimal cost of time and waste of human resources. The TSP in small sizes with few nodes may seem an easy problem for a human to solve, but the fact that nodes could increase, possible tours could grow exponentially, make the problem tough for a human brain to solve. That growth ratio makes the TSP an NP-Hard problem and makes it one of the most famous optimization problems. Since the late '50s, researchers have tried to determine an optimal solution for the TSP by creating various algorithms and benchmarks in order to compare them. A lot of variations of the TSP have been studied, such as Dynamic TSP or TSP with Time Windows, but this thesis work is going to focus on the main TSP with some adds-on of Real-World data.

1.2 Aims and Objectives

1.2.1 Aims

The main aim of this thesis is to introduce another open-source up-to-date tutorial for solving the Travelling Salesman Problem. his tutorial will be comprised by six different algorithms for the TSP, as it aims to emphasise the best algorithm by comparison. Further, one more goal is to introduce a proper and easy way to handle Google APIs for accessing Real-World data; by hopes, that it will make this tutorial interesting and differentiated as it transfers the problem in real life conditions.

1.2.2 Objectives

As the Travelling Salesman Problem is one of the most thoroughly studied optimization problems, there is a significant number of scientific papers and publications on it, which carves a path for researching by Literature Review. Some of the most popular and important papers will become the theoretical foundation of this Thesis work. The most important algorithms which are studied will be applied to a notebook environment, the Google Colab, which afforrds the advantages of an online service. The source code will be written in Python 3 and executed by Google's resources online. A practical comparison of 6 eligible algorithms, will give a measure of their applicability. After this, the chosen algorithm will be implemented real life parameters. The Real-World data will be fetched from Google's GIS services via Google APIs' communication protocol. More specifically, Geocode API will be used for polar coordinates and Distance Matrix API for the distances between them. Subsequently, when an optimal solution is given by the software, the directions of the route can be fetched from Directions API. These directions given, one has obtained a complete solution to the Real-Worl TSP. Finally, it would be helpful to plot the optimal route on an interactive map to reinforce the guidance, accomplished by the Maps Javascript API.

1.3 Contribution to Knowledge

This Thesis, will aim to be an interesting and introductory tutorial for other researchers who want to dive into the Travelling Salesman Problem or generally into optimization problems that concern Transportations and Logistics. It will clearly state the advantages and disadvantages of each algorithm that is used, for any occasion examined. Furthermore, the tutorial will contain a guide for manipulating APIs which is a basic way to communicate and interact with online services.

1.4 Thesis Contents

In the Chapter, a report has been given of the existence of the public interest towards the TSP. Also, the Aims and Objectives of this Thesis were discussed. Chapter 2 will be a literature review around the Travelling Salesman Problem and the algorithms under examination, with a flowchart on each. Chapter 3 will include this Thesis' implementations of the algorithms. Chapter 4 will contain the comparison of the algorithms and a discussion about the results. In Chapter 5, the application will be transferred to a Real-World problem. Finally, Chapter 6 will include this Thesis' survey, a discussion of conclusions, thoughts and suggestions of potential future work.

Chapter 2

Literature Review

2.1 Travelling Salesman Problem in more detail

The Travelling Salesman Problem is one of the most popular optimization problems around the globe. From colossal logistic companies to smaller post offices and to shops with delivery services there is a common aim, profit. As an organization is build around profit, has to reduce the cost of delivery planning and maximize the efficiency of its schedule, in order to serve more customers on a daily bases. The TSP seems to be an easy to solve problem when faced with only a few customers. Nevertheless, its complexity grows exponentially with the number of customers: all possible routes are the factorial number of the customers n which have to been served.

One way to define the complexity of this problem is provided by Computational Complexity Theory, which, in rough terms, is the study of classifying computational problems according to their inherent difficulty. The main issue that exists is Time Complexity, which is practically the amount of time that is needed to solve a particular problem. There is a mathematical notation that describes the limiting behavior of a function when its argument tends towards a particular value or infinity, Big O Notation [2], which characterizes functions according to their growth rates. In Computer Science, Big O is used to classify algorithms according to how their running time or space requirements change as the input size changes. In more detail, Big O comes with an x value in brackets, such as $O(1)$ or $O(n^2)$, which expresses the ratio of input values and solving time. Figure 2.1 shows how the Time Complexity grows for different values of x, from the constant $O(1)$ to exponential $O(n!)$.

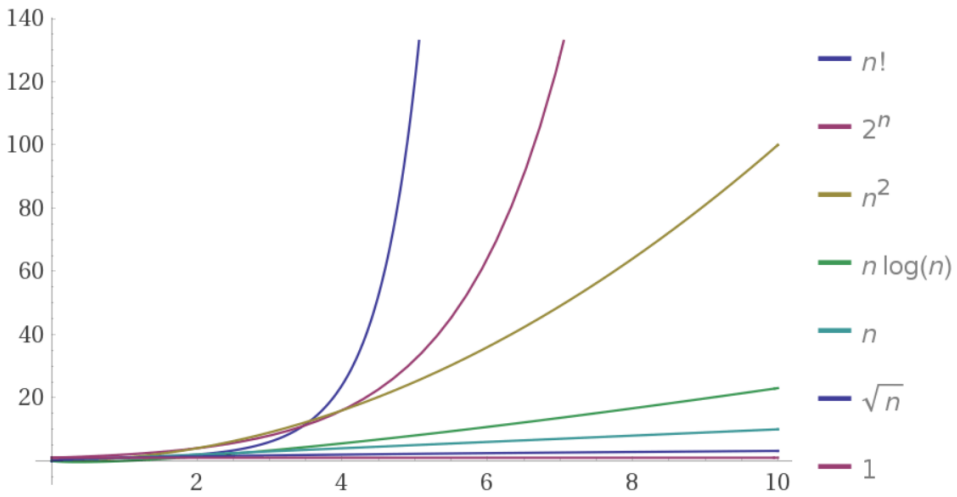

Figure 2.1: Time Complexities

The previous discussion about Time Complexity and Big O Notation allows the introduce of P and NP problems [3]. This perspective defines a problem as solvable by an algorithm in polynomial time (P) or in non-deterministic polynomial time (NP). Class P contains all decision problems that can be solved by a deterministic Turing Machine using a polynomial amount of computation time. Polynomials are functions involving n or n^k (where k is a predefined constant). NP Class contains decision problems that are not easily solved, but one may check the validity of a proposed solution in polynomial time. So their time complexities are exponentials with $O(2^n)$ or $O(n!)$. The theory behind this says that P are part of NP problems. If an appropriate algorithm could be found and somehow solves an NP problem in polynomial time then the problem goes to P problems. An interesting question about P vs. NP is if every NP problem can be solved by an algorithm in polynomial time, and thus become P. This question is an NP problem itself. All NP problems are not the same, and some problems, such as Travelling Salesman Problem, are called NP-Hard, due to its hardness and complexity. Figure 2.2 shows the Euler diagram for P different from NP and P equals NP.

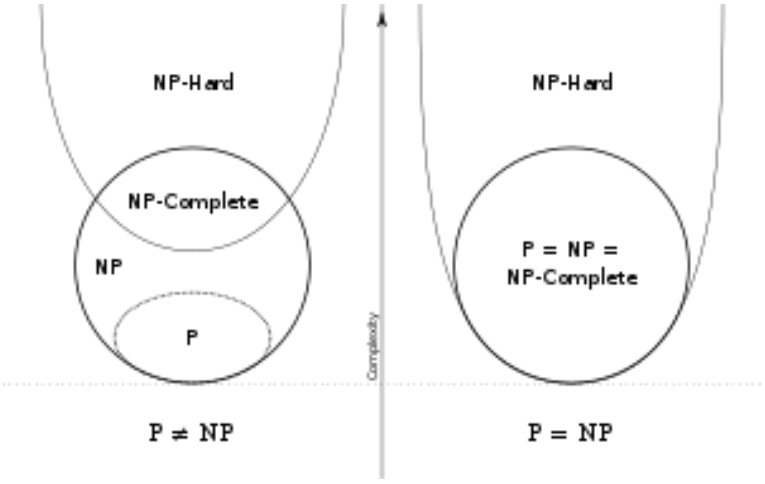

Figure 2.2: Euler's diagram, P vs NP - Wikipedia

As this thesis discusses the theoretical background of that type of problem, it could dive further into this. There are two main classifications of the TSP, which depend on the distances from node to node. If the distances are euclidean i.e. the distance from node i to node j equals that from j to i, thus creating a symmetric distance matrix, in turn giving an undirected graph $G = (V, E)$. Hence, the TSP is symmetric and can be defined as sTSP. On the other hand, when a distance from i to j differs from the one in the opposite direction, thus creating an asymmetric distance matrix, which resulting a directed graph $G = (V, A)$, making the problem an aTSP. The set $V = 1, 2, \ldots, n$ is the vertex set, $E = (i, j) : i, j \in V, c_{ij} = c_{ji}$ is the set of (directed) edges and $A = (i, j) : i, j \in V, c_{ij} \neq c_{ji}$ is an arc set. A square distance matrix $C = (c_{ij})$ is defined with all the distances in it. The objective of the TSP [4] is described by the mathematical type 2.1, which gives the minimum possible cost

$$min_{\pi} \sum_{i \in N} c_{i\pi(i)}, \tag{2.1}$$

where π runs over all cyclic permutations of N; $\pi^k(i)$ is the kth city reached by the salesman from city i.

As an example, the solution of the TSP for 7 nodes and a constant starting point would be a list (2.2) of numbers starting from the depot and returns to it. So if the depot took number 1 as a sign, the other 7 nodes to would have to pick a number from 2 to 8.

$$\begin{bmatrix} 1 & 3 & 6 & 2 & 4 & 5 & 8 & 7 & 1 \end{bmatrix} \tag{2.2}$$

Figure 2.3 is the nodes from above, connected in the optimal order. All nodes visited once and return back to the starting node. The fact that the TSP is symmetrical, the direction of the order does not matters.

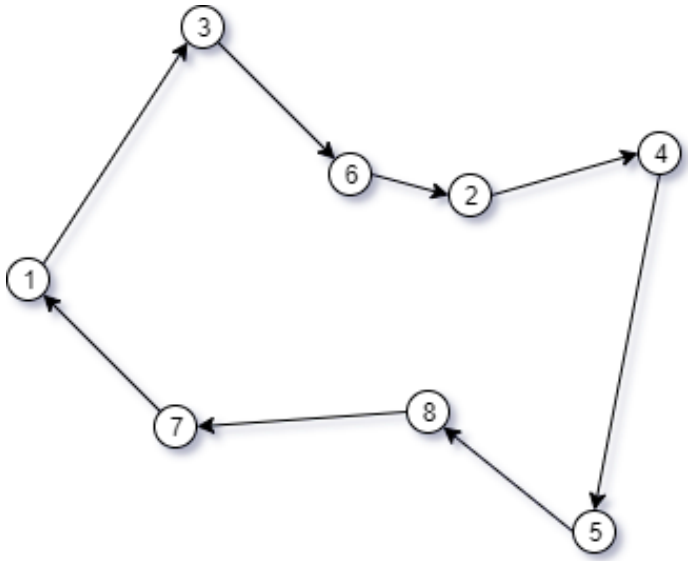

Figure 2.3: TSP optimal route of 8 nodes

Considering all the above, the TSP is completely defined. In the following subsection a definition of an algorithm is given and 6 algorithms capable of solving the TSP are presented.

2.2 Algorithms for the TSP

In a nutshell, an algorithm is a clear and precise description of a series of separate instructions - steps, in order to solve a specific problem in some way. This subsection introduces three types of algorithms, Exact Algorithms, Heuristic Algorithms, and Meta-Heuristic Algorithms. These three types differ from each other in the way that they look for a solution to a problem. The optimality of a solution depends on their behavior, which also has a noticeable impact on the time to completion. In our days, as the problems become more complex and our expectations of running time become more demanding, the optimal algorithm needs to have a high accuracy rate in the least amount of running time possible.

2.2.1 Exact Algorithms

Exact Algorithms are the algorithms that always give the optimal solution, by comparing all possible ones. As the possible solutions of the TSP are the factorial of the input nodes n, the process of examining all the given solutions can be extremely time consuming. Resulting in an operations with much higher time complexity, space complexity and power consumption.

2.2.1.1 Brute Force Method

Brute force is a simple and basic algorithm. It calculates the cost of every possible solution, keeping only the smallest. As the amount of possible solutions grows exponentially, the calculation time grows at the same rate. The table 2.1 shows the number of possible tours for different numbers of input nodes and the time required to solve each problem, for a theoretical system which solves a 3-customer-TSP in about 0.5(*msec*).

The failing of Brute Force becomes apparent at 10 nodes. At this point, the algorithm needs approximately 5 minutes to find the optimal answer. For commercial purposes, this amount of time makes the algorithm useless. To demonstrate how dramatically this algorithm fails, with 20 nodes as input, the algorithm would need almost a million years for the solution.

Figure 2.4, is the flowchart of Brute Force Algorithm which all the methods and conditions that are involved. (A flowchart provides an effective way of understanding an algorithm.)

Nodes	Possible Tours	Approx. Time (sec)
3	6	$5.0\ 10^{-4}$
4	24	$2.0\ 10^{-3}$
5	120	$1.0\ 10^{-2}$
6	720	$6.0\ 10^{-2}$
7	5040	$4.2\ 10^{-1}$
8	40320	$3.4\ 10^{0}$
9	362880	$3.0\ 10^{1}$
10	3628800	$3.0\ 10^{2}$
11	39916800	$3.3\ 10^{3}$
12	479001600	$3.9\ 10^{4}$
13	6227020800	$5.2\ 10^{5}$
14	87178291200	$7.3\ 10^{6}$
15	1307674368000	$1.1\ 10^{8}$
16	20922789888000	$1.7\ 10^{9}$
.	.	.
.	.	.
20	$2.4\ 10^{18}$	$2.0\ 10^{14}$

Table 2.1: Growth ratio of calculation time of the Brute Force

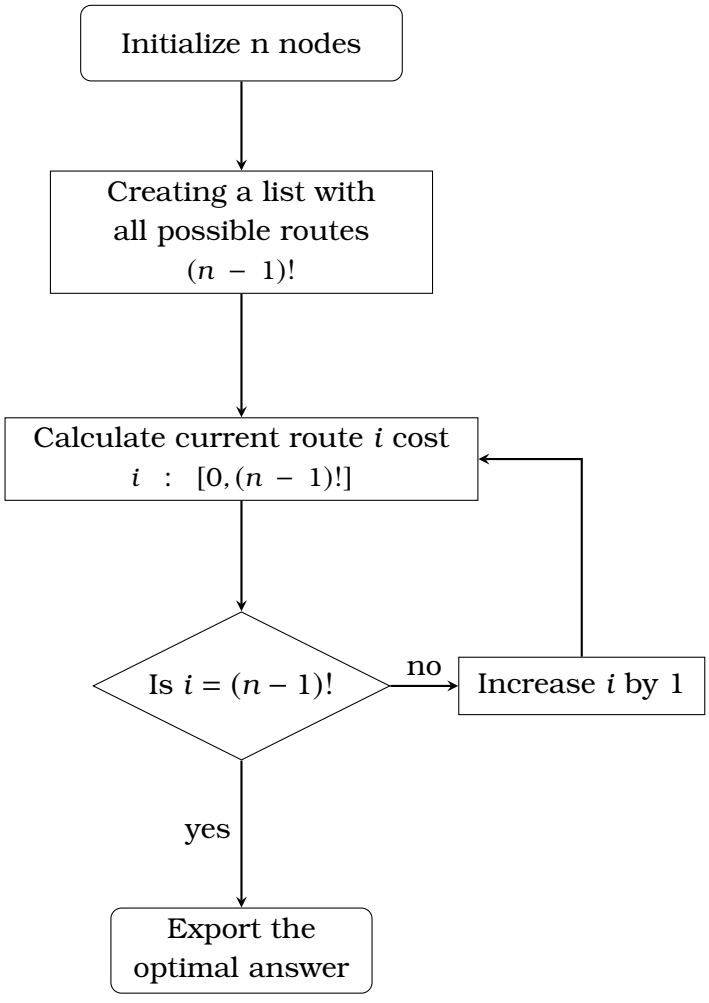

Figure 2.4: Brute Force flowchart

2.2.1.2 Dynamic Programming

Another exact approach to solving the TSP is the Dynamic Programming (DP) formulation presented by Held and Karp [5] in 1962. The main idea of DP is to save the results of computing and by combining them, avoid recomputing. This effectively separates the main problem into smaller sub-problems. The same process works on the sub-problems, creating further sub-problems, and so on. In more detail, DP applied to TSP, creates a tree with all possible tours as shown in figure 2.5, then starting from the bottom, calculates and saves the value of the cost from node to node. For every sub-problem whose cost it has to compute, the algorithm draws from its saved values in case of already calculated route-costs, eliminating the need to recompute them. For instance, let's say that DP has already computed the cost of the tour [1, 5, 4] (1st column on the left of figure 2.5), when it is needed to recompute it (7th column on the left), it pulls this value from the saved ones. Owing to that fact, Dynamic Programming is an exact approach but conserves a lot of time, compared to Brute Force.

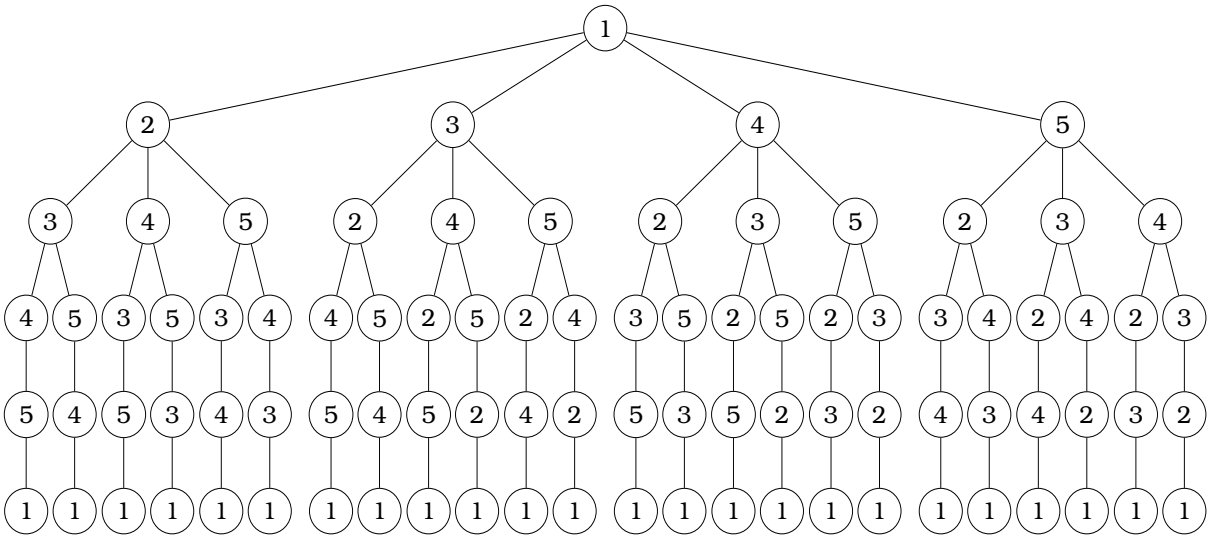

Figure 2.5: Tree of possible routes

There are $n - 1$ possible nodes and 2^n possible subsets, and for each call performs at most $O(n)$ work. So the time complexity of DP on the TSP is $O(2^n n^2)$, which is exponential, and makes DP faster than Brute Force, which has factorial time complexity. The following table 2.2 shows the approximate solving time for an increasing number of nodes. The 3-nodes problem's solving time is assumed at 0.5($msec$), based on this, the rest of the values for higher numbers of n is calculated.

Nodes	Possible Tours	Approx. Time (sec)
3	6	$0.5\ 10^{-3}$
4	24	$1.8\ 10^{-3}$
5	120	$5.6\ 10^{-3}$
6	720	$1.6\ 10^{-2}$
7	5040	$4.4\ 10^{-2}$
8	40320	$1.1\ 10^{-1}$
9	362880	$2.9\ 10^{-1}$
10	3628800	$7.1\ 10^{-1}$
11	39916800	$1.7\ 10^{0}$
12	479001600	$4.1\ 10^{0}$
13	6227020800	$9.6\ 10^{0}$
14	87178291200	$2.2\ 10^{1}$
15	1307674368000	$5.1\ 10^{1}$
16	20922789888000	$1.2\ 10^{2}$
.	.	.
.	.	.
20	$2.4\ 10^{18}$	$2.9\ 10^{13}$

Table 2.2: Growth ratio of calculation time of the Dynamic Programming

Solving time has significantly decreased as compared to Brute Force. The 10 nodes problem is calculated in about 0.7(*sec*): Dynamic Programming is a useful algorithm for smaller-sized problems, up to 13 nodes, as it can solve them to optimality within a few seconds. Figure 2.6 is the flowchart of Dynamic programming Algorithm which gives the commands and conditions involved.

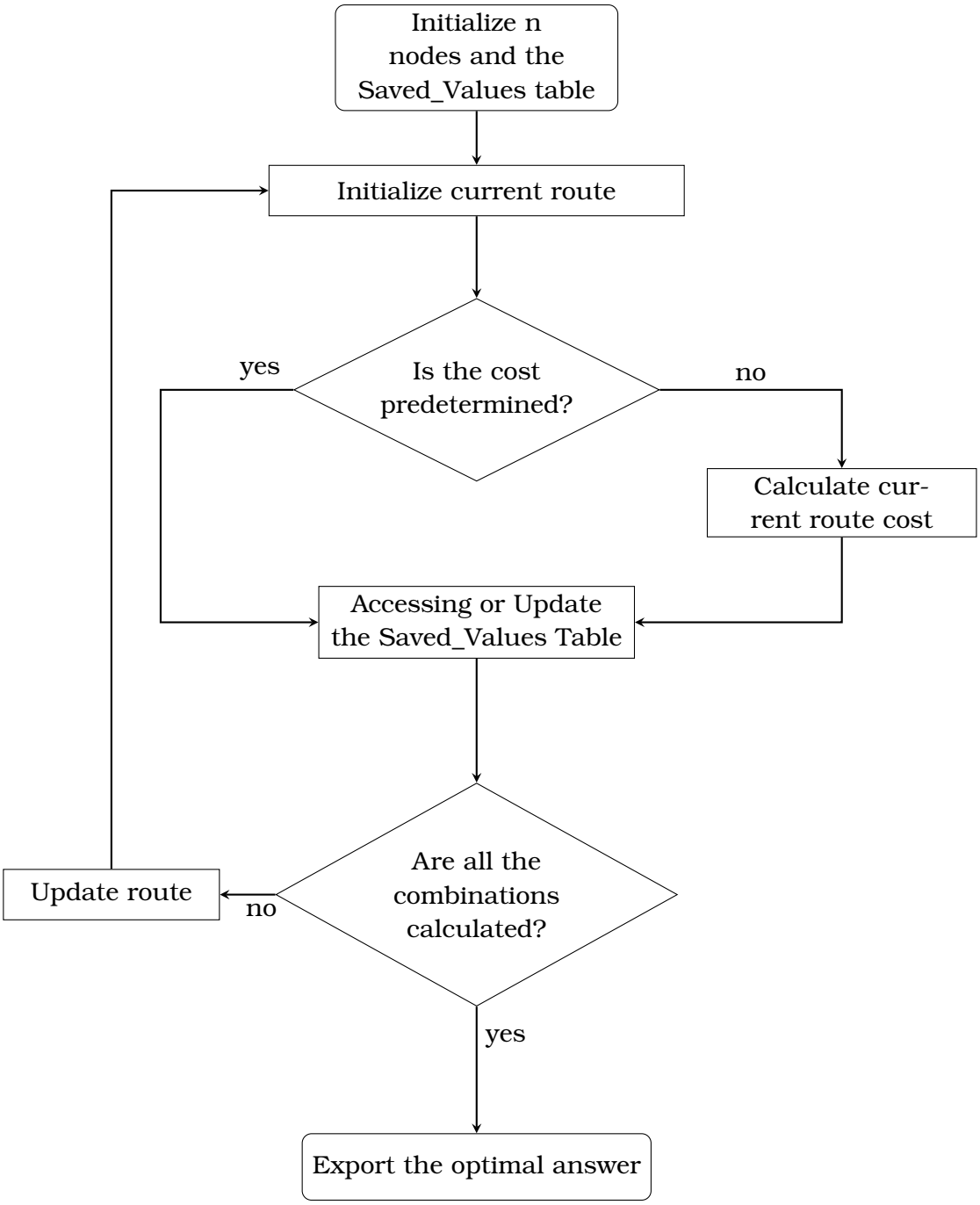

Figure 2.6: Dynamic Programming flowchart

2.2.2 Heuristic Algorithms

Heuristic Algorithms give a solution to a problem, but are no longer guaranteed to find the optimal one. This type of algorithms is classified as an approximate algorithm, as opposed to the exact algorithms above. The behavior of a heuristic is to search locally for the best answer (e.g. Hill Climbing [6]), which makes them susceptible to local minima traps, i.e. returning an answer which is not the global minimum. Their advantage is solving time, as they need a few seconds to solve large-scale problems. Figure 2.7 is the plot of a function, which could be the cost function of the TSP, with its local and global minimum marked.

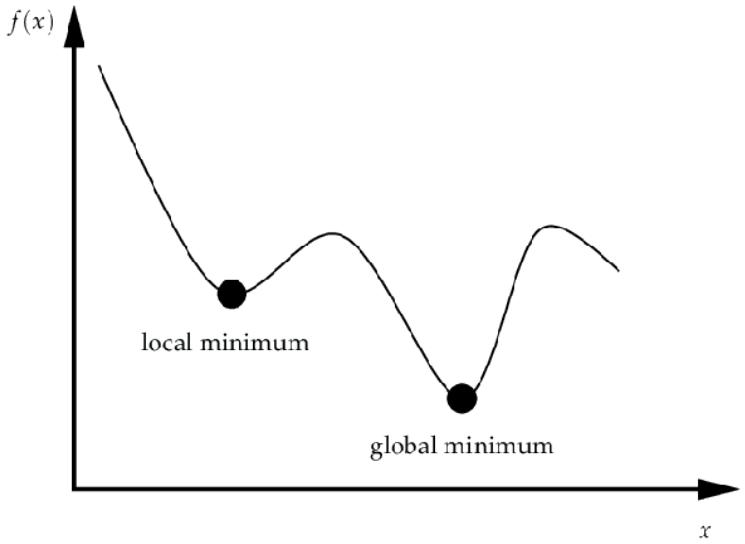

Figure 2.7: Local vs Global minimum - Quora

2.2.2.1 Greedy Algorithm

A simple Heuristic algorithm is the Greedy algorithm [7]. Its name makes known its biggest disadvantage: Greedy algorithm picks the nearest node from the node it begins. This means it always misses the opportunity of finding better paths to reduce the total cost. The Time Complexity of this algorithm is $O(n * log(n))$: n times of finding the nearest node according to the distance matrix, ($log(n)$) by merge sort. The following table 2.3, based on the discussed behavior, shows the calculation time growth ratio for different input nodes, followed by the flowchart of Greedy algorithm, figure 2.8.

Nodes	Approx. Time (sec)
3	$0.5\ 10^{-3}$
4	$0.8\ 10^{-3}$
5	$1.2\ 10^{-3}$
6	$1.6\ 10^{-3}$
7	$2.1\ 10^{-3}$
8	$2.5\ 10^{-3}$
9	$3.0\ 10^{-3}$
10	$3.5\ 10^{-3}$
11	$4.0\ 10^{-3}$
12	$4.5\ 10^{-3}$
13	$5.1\ 10^{-3}$
20	$9.1\ 10^{-3}$
21	$9.7\ 10^{-3}$
22	$1.0\ 10^{-2}$
50	$3.0\ 10^{-2}$
75	$5.0\ 10^{-2}$
100	$7.0\ 10^{-2}$
140	$1.1\ 10^{-1}$

Table 2.3: Growth ratio of calculation time of the Greedy algorithm

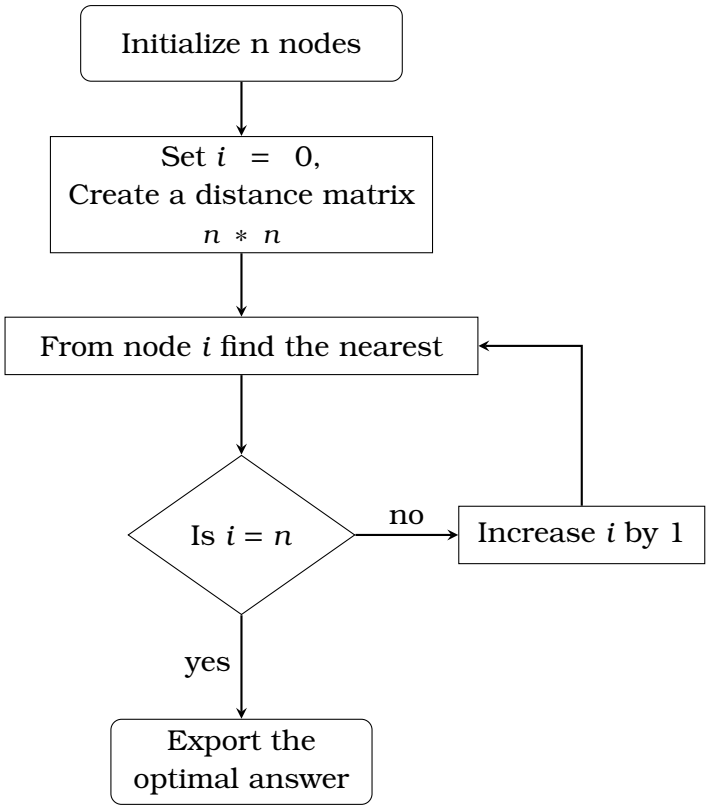

Figure 2.8: Greedy algorithm flowchart

Greedy algorithm cannot stand on its own, the solutions it provides are, in most cases, far from optimal. However, Greedy algorithm could be used in some applications in combination with other algorithms, in order to create hybrid algorithms.

2.2.2.2 2-Opt Algorithm

2-Opt is a very popular algorithm for solving the TSP, and it first proposed by Croes in 1958 [8]. Its main idea is by starting with a random route, search locally for better solutions. At figure 2.9 2-Opt forced on a possible route, which crosses over itself, in order to reorder it. The algorithm investigates if a sub-tour (from node b to e) swaps with another sub-tour (from node c to f), reduce the cost of the total route. When the result is positive the 2-Opt replace its answer with the current best one. If the result is negative, the algorithm tries different pair of sub-tours.

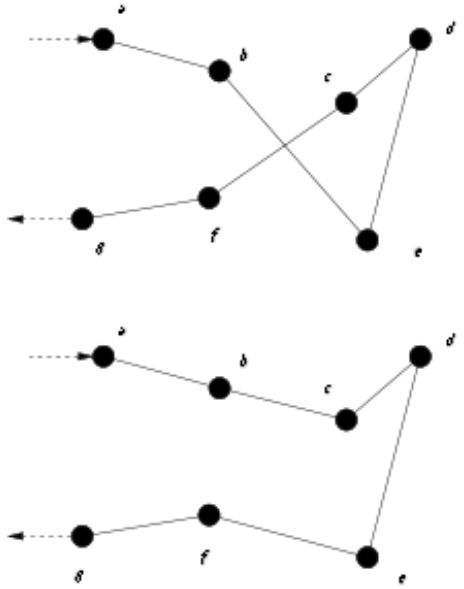

Figure 2.9: 2-Opt forced on a route

It belongs to K-Opt algorithms, which their name comes from k points optimization. Another K-Opt algorithm that is widely known is the 3-Opt, which picks 3 sub-tours of the route for examination. The 2-Opt can really improve a route in reasonable time, but as it searches locally, often get trapped into local minimum solutions (figure 2.7). A proposal way to overcome this issue is to run the algorithm several time with different starting route. That is may guide the algorithm close or even to the optimal solution. Figure 2.10 shows the flowchart of the 2-Opt.

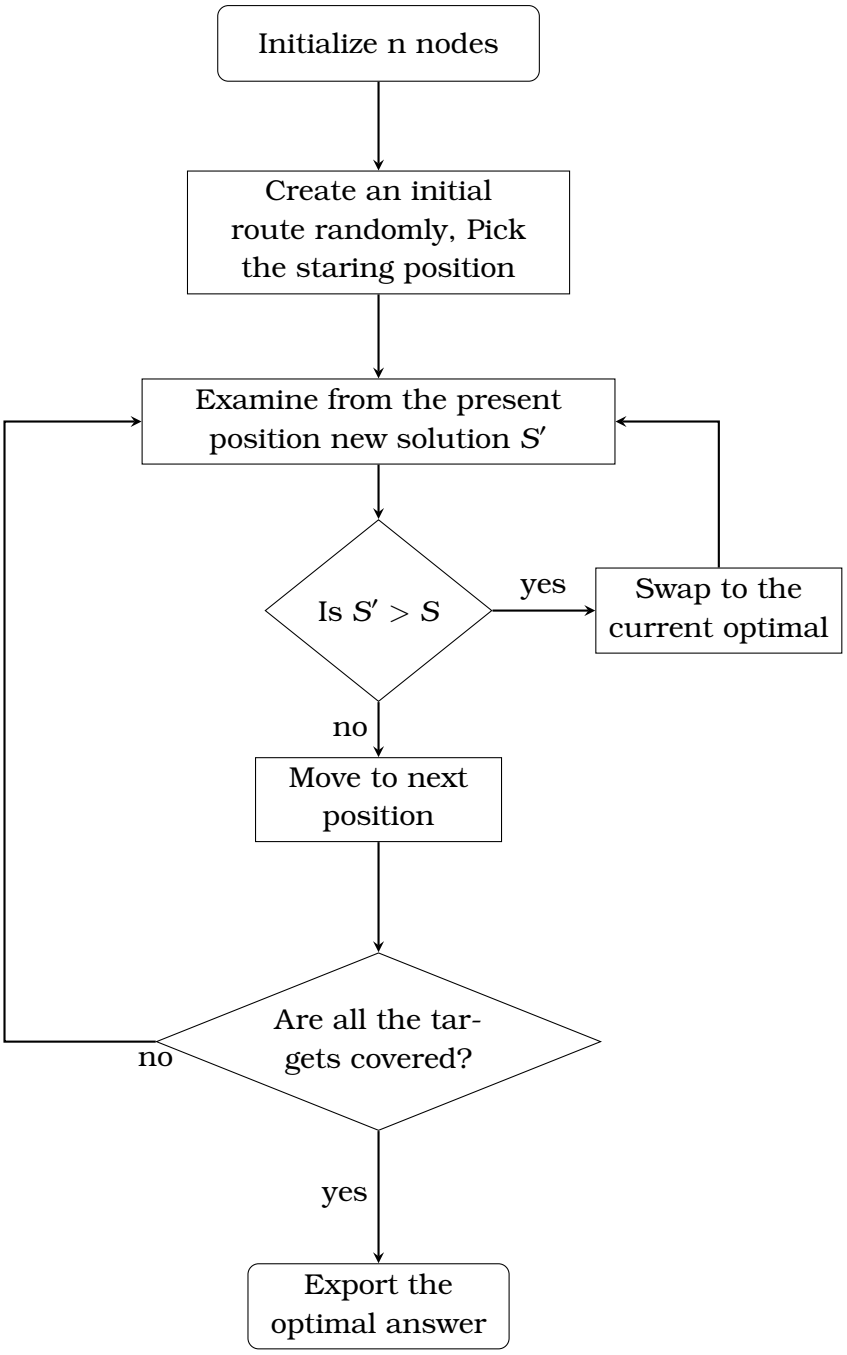

Figure 2.10: 2-Opt algorithm flowchart

Regarding the Time Complexity, 2-Opt is in $O(n^3)$. At each position, the algorithm tests all the possible combinations which that give n^2, and that process repeated for all the n positions, so they multiplied. Follows the table 2.4 which shows the growth ratio of the approximate time that calculation process needs. The calculation time of 3 customer's nodes set at $0,5(msec)$, based on the previous perspective.

Nodes	Approx. Time (sec)
3	$0.5\ 10^{-3}$
4	$1.2\ 10^{-3}$
5	$2.3\ 10^{-3}$
6	$4.0\ 10^{-3}$
7	$6.4\ 10^{-3}$
8	$9.5\ 10^{-3}$
9	$1.4\ 10^{-2}$
10	$1.9\ 10^{-2}$
11	$2.5\ 10^{-2}$
12	$3.2\ 10^{-2}$
13	$4.1\ 10^{-2}$
20	$1.5\ 10^{-1}$
21	$1.7\ 10^{-1}$
22	$2.0\ 10^{-1}$
50	2.3
75	7.8
100	$1.9\ 10^{1}$
140	$5.1\ 10^{1}$

Table 2.4: Growth ratio of calculation time of the 2-Opt

2.2.3 Meta-Heuristic Algorithms

In science, researchers usually inspired by the way that nature acts and reacts. Due to that fact, Meta-Heuristic algorithms introduced, from the late '50s, as higher-level heuristics which could escape from being trapped into local optimal solutions. Escaping a local minimum may be a complicated operation. To overcome this, the algorithm may accept temporary solutions, sometimes worse than the current, in case of finding new paths, which could lead to the global optimum. They can also evolve or mutate some good solutions, during the time of processing the problem, in order to be more effective and expeditious. Meta-Heuristic algorithms at the '50s were not widely known, until 1963 Ingo Rechenberg and Hans-Paul Schwefel developed evolutionary strategies. In 1966, L. J. Fogel developed evolutionary programming, to contribute to Meta-Heuristics.

Meta-Heuristics cannot guarantee that a globally optimal solution can be found, but they increase that probability by the efficient exploring of the search space. A further property of Meta-Heuristic algorithms is that they are not problem-specific so they can transfer to many other optimization problems. The two most famous algorithms that this Thesis presents, is Simulated Annealing and Genetic Algorithm.

2.2.3.1 Simulated Annealing Algorithm

Simulated Annealing is a Meta-Heuristic algorithm aiming the global extremum in optimization problems. First introduced by Kirkpatrick, Gelatt Jr. and Vecchi in 1983 [9]. The Simulated Annealing is inspired by the metal annealing process. Annealing is the controlled cooling after a heat treatment, which alerts the physical and chemical properties of a material. Due to changes in its structure, the material's ductility and hardness will be affected as its crystal size increased and its defects decreased. In the algorithm, an initial high temperature is set and then reduced exponentially by a cooling rate, alpha. As the temperature is still high, the probability of choosing a current solution that is worse than the current is high too. Conversely, that probability is reduced as the temperature decreased. That makes the algorithm able to have the energy of jumping out of a local optimum.

In more detail, the Simulated Annealing algorithm first generates a random solution x_0 (a random route) of the problem. Then creates a new evolutionary solution x_{new} based on the current x_0. Several methods of creating the x_{new} have been published. A basic method is to reverse a random range of the route. As an example, at 2.3 is the initial route. The fourth index picked randomly, as such the length of the sub-route that will be reversed, to produce the route 2.4.

$$\begin{bmatrix} 1 & 3 & 6 & 2 & 4 & 5 & 8 & 7 & 1 \end{bmatrix} \tag{2.3}$$

$$\begin{bmatrix} 1 & 3 & 6 & 8 & 5 & 4 & 2 & 7 & 1 \end{bmatrix} \tag{2.4}$$

Afterwards, the algorithm compares the cost (2.5) of x_{new} with x_0. If the difference Δf between them is positive ($f(x_{new}) < f(x_0)$), the SA replaces the current route x_0 with the new route x_{new}.

$$\Delta f = f(x_0) - f(x_{new}) \tag{2.5}$$

Else if the Δf is negative the algorithm forces the metropolis rule on the x_{new}. Metropolis rule introduced by Metropolis et al. in 1953 [10]. The (2.6) gives the mathematical equation of Metropolis law which take numbers in [0,1]. That p compared with a random float number, also in [0,1], in order to decide if the examined solution will be accepted or not.

$$p = exp\left(\frac{\Delta f}{T}\right) \tag{2.6}$$

where T is the current temperature, Δf is the difference of the objective functions.

As the iterations passes and the temperature decreased, the probability p reduced, so the algorithm be less capable of accepting new solutions. As a result, the algorithm at the last iterations can only accept better solutions.

As the problem gets more complicated, the number of iterations should be increased. The Simulated Annealing algorithm uses two loops, one into the other. The inner loop creates the x_{new} and compares it with the x_0, to decide about the

selection. The outer loop reduces the temperature of the algorithm, and it stops when it reaches the desired one. The size of the outer loop is the times that alpha could multiplies the temperature until is equal to the final temperature. The size of the inner loop is proposed to be 3 or 4 times the number of the n nodes.

The calculation time of the Simulated Annealing estimated by the number of iterations of the inner loop, times the number of iterations of the outer loop. As the problem gets more complicated, only the amount of inner loop will be increased. That makes the algorithm to have linear growth rate and Time Complexity. Let's say, that a TSP with 3 customer nodes, needs 1(sec) to be calculated. The table 2.5 shows the approximate calculation time. The number of the outer loop set to 1400 and the inner loop to 4 times the n nodes. Follows the flowchart of the Simulated Annealing in figure 2.11.

Nodes	Approx. Time (sec)
3	1.0
4	1.3
5	1.7
6	2.0
7	2.3
8	2.7
9	3.0
10	3.3
11	3.7
12	4.0
13	4.3
20	6.7
21	7.0
22	7.3
50	16.7
75	25.0
100	33.3
140	46.7

Table 2.5: Growth ratio of calculation time of the Simulated Annealing

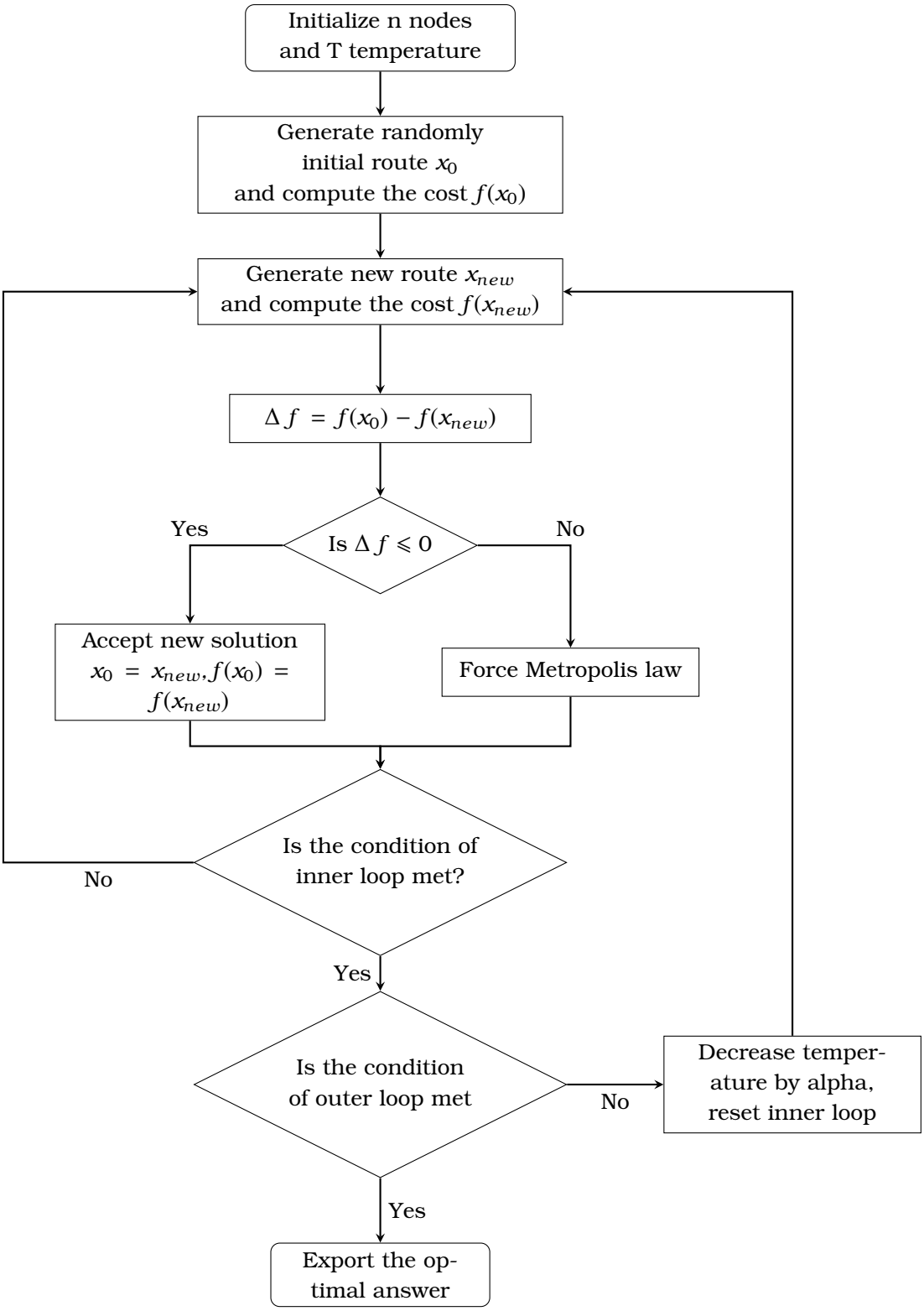

Figure 2.11: Simulated Annealing algorithm flowchart

2.2.3.2 Genetic Algorithm

Genetic Algorithm is another meta-heuristic that is widely used for optimization problems. GAs inspired by the natural selection and the survival of the fittest. First introduced by J. Holland in 1960 based on Darwin's theory of evolution. In 1988, David E. Goldberg and his professor J. Holland introduced a more completed version of Genetic Algorithms [11]. All meta-heuristic algorithms, but especially GA was the introduction to Artificial Intelligence. The concept of GAs is inspired by biology, and especially from chromosomes. Chromosomes interact with themselves or with other chromosomes in many ways, in order to breed. Genetic Algorithms patterned two main interactions of chromosomes, mutation and crossover, aiming to reproduce its solutions.

Extensively, the Genetic Algorithm at the first step creates a population of random solutions for a specific problem. The amount of that population depends on the application and the pre-scheduling. A general perspective for the TSP, is that a population of around 200 solutions will be enough for most of the problems. Therefore, every single solution of the population, computed by the cost function. Then, the population has to take part in the production of new solutions. So the first population becomes parents in order to create the children population. Before this process, the algorithm must judge every parent by its fitness value. That judgment is part of the survival of the fittest. Parents with less cost (better fitness) have higher probability of being in reproduction. On the other hand, worse parents must have a smaller probability of reproduction. That concept should not be absolutist, is necessary that even the worst solution must have some chances of being a parent. That is how GA could be led to a global optimum solution.

The selection's method of parents has several variations. Each one differs from the other in the way of assigning the probabilities to parents. There are some methods which are optimistic, as they left the selection to chance. oppositely there some methods which decide very strictly, as they exclude many solutions from breeding. Some of the most well-known methods following below:

- Fitness Proportionate Selection
  Also known as roulette wheel selection. Parents take a probability which is proportional to its fitness value. The better fitness the higher chances for breeding. Think about a roulette wheel with different sizes for each pie. Every possible parent gets a pie on the wheel. Then a random number on the length of that wheel, chooses the parent.

- Elitism Selection
  Elitism is a method which allows the best few solutions to pass through the next generation unchanged, without a kind of mutation. The amount of parents that will be picked as elites usually is about two to five percent of the amount of population.

- Tournament Selection
  At this method, a certain number of parents each time picked randomly from the population. These picked parents are judged by the algorithm. Only the best one will be a parent of the next generation. That gives small possibilities to the parents with lower fitness value. That process must be repeated for every parent

- Rank Selection
  This method decides the parents due to their ranking, and not on their fitness value. So it does not matter what is the cost of each solution, but which is the best one at that time. That is may characterize it as a greedy method, as it could give a high difference in probability on two good solutions with almost the same cost value. Oppositely, it could give equal chances to two parents with a remarkable difference in fitness.

The selection's methods that mentioned before, could be used in any application combinatorially too. As an example, an application at the first 80% percent of generations could use the Roulette-Wheel selection, and at the rest 20% could use the Rank selection. At the same application, the GA could pick the elites from each generation in order to transfer them unmutated into the next.

The selection in Genetic Algorithms is the way of the evolution through generations. After the creation of the first population, the parents (named also as chromosomes) which have been chosen must be bred in order to create their children (named as offsprings). The main way of reproduction is by the Crossover operators. By doing a crossover, two parents have to been picked, to create one or more offsprings. There are a lot of crossover operators, with the most popular following:

- Partially Mapped Crossover Operator (PMX)

PMX operator first suggested by Golberg and Lingle at 1985 [12]. First, it selects at random two cut points along the parents.

$$P_1 = (1\ 2\ 3\ 4\ |\ 5\ 6\ 7\ |\ 8\ 9\ 1)$$

$$P_2 = (1\ 8\ 6\ 3\ |\ 5\ 2\ 9\ |4\ 7\ 1)$$

Secondly, create two offsprings and copies the cut content of parent 1 to offspring 2. Respectively, it does the same process for parent 2. As a reminder, the parents both start and end at point 1, which is the starting point and it can not change. So, two offsprings are created:

$$O_1 = (1\ -\ -\ -\ |\ 5\ 2\ 9\ |\ -\ -1)$$

$$O_2 = (1\ -\ -\ -\ |\ 5\ 6\ 7\ |\ -\ -1)$$

Finally, copies the rest of the chromosome's information. If PMX operator tries to copy a node which is already satisfied, replaces it with the first possible of the cut content. The two final offsprings are:

$$O_1 = (1\ 6\ 3\ 4\ |\ 5\ 2\ 9\ |\ 8\ 7\ 1)$$

$$O_2 = (1\ 8\ 2\ 3\ |\ 5\ 6\ 7\ |\ 4\ 9\ 1)$$

- Order Crossover Operator (OX)

The Order crossover was proposed by Davis at 1985 [13]. It can be considered as an alternative of PMX operator. As the previous perspective, two random cut points being picked

$$P_1 = (1\ 2 \mid 3\ 4\ 5 \mid 6\ 7\ 8\ 9\ 1)$$

$$P_2 = (1\ 8 \mid 6\ 3\ 5 \mid 2\ 9\ 4\ 7\ 1)$$

Then, the two offsprings can be produced:

$$O_2 = (1\ - \mid 3\ 4\ 5 \mid - - - - 1)$$

$$O_1 = (1\ - \mid 6\ 3\ 5 \mid - - - - 1)$$

As the last step, start filling the offspring from the 6th element by copying the content from the second crossover point of parent 2 to the parent 1. If a node already took place to skip it and continue to the next one. The two final offsprings would be:

$$O_1 = (1\ 6 \mid 3\ 4\ 5 \mid 2\ 9\ 7\ 8\ 1)$$

$$O_2 = (1\ 4 \mid 6\ 3\ 5 \mid 7\ 8\ 9\ 2\ 1)$$

- Cycle Crossover Operator (CX)

Cycle crossover published by Oliver et al. at 1987 [14]. Is a bit confusing operation for beginners. The main idea is to separate the process to cycles. Lets the two parents will be:

$$P_1 = (1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 1)$$

$$P_2 = (1\ 8\ 6\ 3\ 5\ 2\ 9\ 4\ 7\ 1)$$

In the beginning, the first cycle has to be created. A cycle goes from 1st element of parent 1 to the element at the same position of parent 2. Then, the cycle continues to the element of parent 1 that the 1st element of parent 2 shows. When a cycle returns to an already visited element of parent 1, the cycle closes. Alternative cycles will be created until all the nodes covered. For instance, the 2nd element of P_1 (first possible for the process) is '2', and the cycle has to goes vertically to the same position in P_2, which is the node '8'. Then the cycle has to continue at the position in P_1 in which '8' is. That is the 8th position, is noticeable to clear that there is no relation in the index and in the information, as the parents picked randomly. So the first cycle will be:

$$2 \rightarrow 8 \rightarrow 8 \rightarrow 4 \rightarrow 4 \rightarrow 3 \rightarrow 3 \rightarrow 6 \rightarrow 6 \rightarrow 2$$

Another representation of the cycle could be, the index oriented method. It shows the index of each parent

$$P_1, 2 \rightarrow P_2, 2 \rightarrow P_1, 8 \rightarrow P_2, 8 \rightarrow P_1, 4 \rightarrow P_2, 4 \rightarrow P_1, 3 \rightarrow P_2, 3 \rightarrow P_1, 6 \rightarrow P_2, 6$$

In he same way, cycle 2 is:

$$5 \rightarrow 5$$

The remaining cycle is:

$$7 \rightarrow 9 \rightarrow 9 \rightarrow 7$$

So the first offspring takes the first cycle's elements that present the parent 1:

$$O_1 = (1\ 2\ 3\ 4 - 6 - 8 - 1)$$

From the 2nd cycle the first offspring would be:

$$O_1 = (1\ 2\ 3\ 4\ 5\ 6 - 8 - 1)$$

At the time the chromosome looks the same with its parent. So for the last step, the 3rd cycle will add the elements of the 2nd parent. Final O_1 will be:

$$O_1 = (1\ 2\ 3\ 4\ 5\ 6\ 9\ 8\ 7\ 1)$$

Accordingly, the O_2 will be:

$$O_1 = (1\ 8\ 6\ 3\ 5\ 2\ 7\ 4\ 9\ 1)$$

Another method of creating the new population is by Mutation. This method needs only one parent for breeding an offspring. Some GA applications could use Mutation as a secondary option, right after the Crossover operation. In literature several methods of Mutation have been noticed already, with the most popular following:

- Swap Mutation

This method is a very simple operation. It is accomplished by swapping 2 or more nodes of a specific parent. For example, bellow at the P parent is supposed to node '6' will change positions with node '7'. As a result, the O offspring produced.

$$P = (1\ 8\ \underline{6}\ 3\ 5\ 2\ \underline{7}\ 4\ 9\ 1)$$

$$O = (1\ 8\ \underline{7}\ 3\ 5\ 2\ \underline{6}\ 4\ 9\ 1)$$

- Scramble Mutation

That type of Mutation picks a range of the P parent, and shuffles it randomly in order to create the O offspring:

$$P = (1\ 2\ |\ 3\ 4\ 5\ 6\ 7\ |\ 8\ 9\ 1)$$

$$O = (1\ 2\ |\ 4\ 6\ 7\ 3\ 5\ |\ 8\ 9\ 1)$$

- Inversion Mutation

In the same way as before, that operator picks a random subset in the P parent. Then reverse that subset and replace it to the chromosome, to produce the O offspring:

$$P = (1\ 2 \mid 3\ 4\ 5\ 6\ 7 \mid 8\ 9\ 1)$$

$$O = (1\ 2 \mid 7\ 6\ 5\ 4\ 3 \mid 8\ 9\ 1)$$

One more possible way of producing new chromosomes is by creating new completely random solutions. If a GA only chooses that way, the chromosomes will stumble upon randomness. As a result, there will be no improvement.

In conclusion, there are a lot of types of creating a new generation. Each one has its benefits, so the best-case scenario would be to combine all of them in the right way. That part belongs to statistics, which have to analyze the percentages that each method will take, through benchmarks and tests. The widely accepted perspective through rough research on Genetic Algorithms used in the TSP is to give 65% to 75% to crossover operators, a 25% to 35% to mutation operators and a 2% or less to the random impact.

After the appropriate production of the next generation, GA has to compute the cost of each children. This children's generation will become the next parents. That procedure must continues until the optimal answer is found. Follows the figure 2.12, which presents a recommended flowchart of the Genetic Algorithm.

Concerning the calculation time that a GA needs, is the amount of the population that you picked times the number of generations that you decide. As there are many variants and methods of select or breed chromosomes, the Time Complexity has a significant difference from application to application.

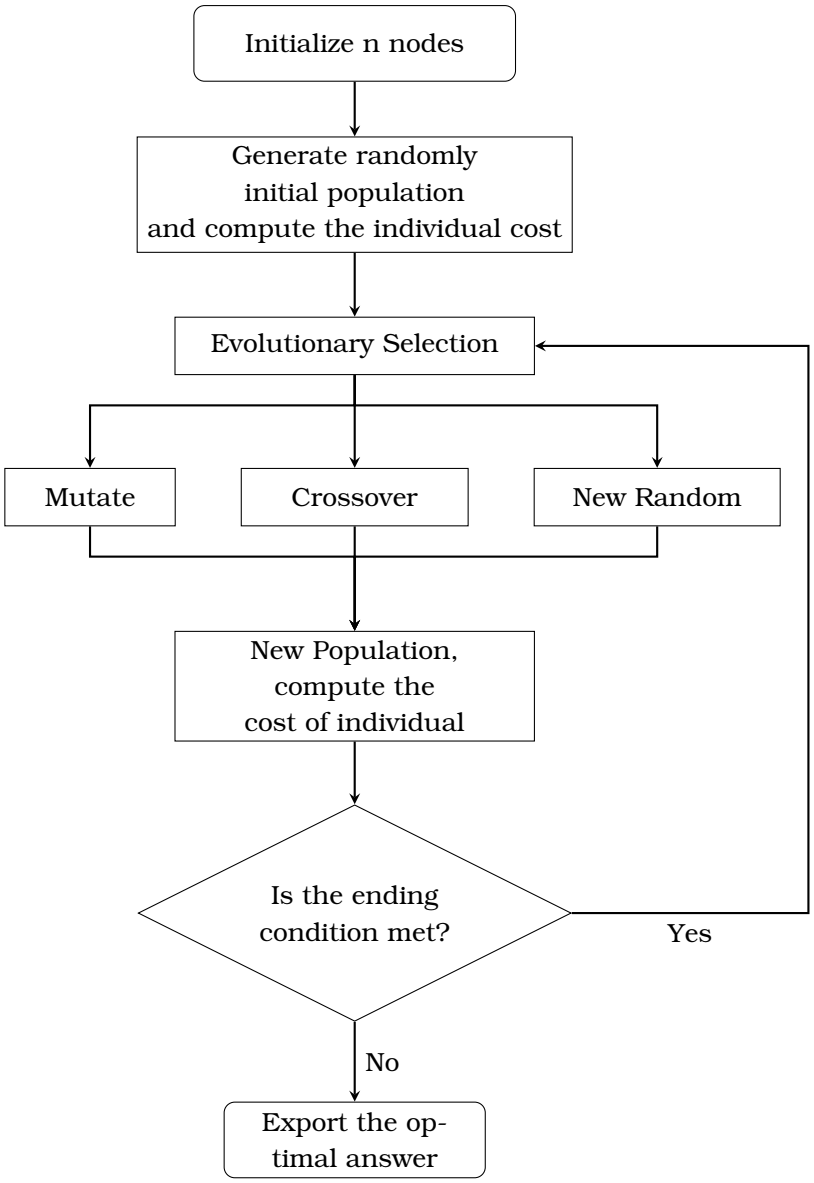

Figure 2.12: Genetic Algorithm flowchart

2.3 Summarize

In this chapter a Literature Review of the most important algorithms used for the TSP took place. Each algorithm explained decently, in order to be understanding and let us continue to the implementation of them. Therefore, in the next chapter, the Thesis will introduce a short description of each algorithm's implementation. There will be also an example of a TSP for each algorithm.

Chapter 3

Algorithms' Implementation

3.1 Introduction

In this chapter, Thesis introduces the algorithms' implementations that mentioned in the previous chapter. The environment that is used for this application, is a free Jupyter Notebook which is provided by Google Colaboratory. That software needs no setup, as it completely compiled in the cloud. That gives the opportunity of making benchmarks with constant computing power. The programming language that is used, is Python 3, due to its compatibility with the Google Colab and to high interest of its community for open-source software. All the algorithms will be tested in specific problems, which will remain unchanged from algorithm to algorithm.

The program will use a csv file of 140 nodes with coordinates from 0 to 1. As the problem needs to be more complicated the application will increase the number of nodes that will be taken from the list. So, the distances will be Euclidean and the distance matrix will be a symmetric one. Therefore, the TSP will be an sTSP.

The code of each algorithm is uploaded to Github.

3.2 Exact Algorithms

3.2.1 Brute Force Method

Brute Force algorithm produces a list in size $(n-1)!$ with all possible tours. That will be done with the permutations function of itertools module. Then, the algorithm computes the cost of the individual, and saves the shortest tour. When the whole list have been checked the algorithm is ready to export the optimal solution.

In this example the first 10 nodes of the csv file, will be used for the TSP. Always the first node (0.12996464, 0.82823386) will be the Depot, labeled as the red diamond. The rest of the nodes will be plotted as the blue circles. The scatter is at figure 3.1.

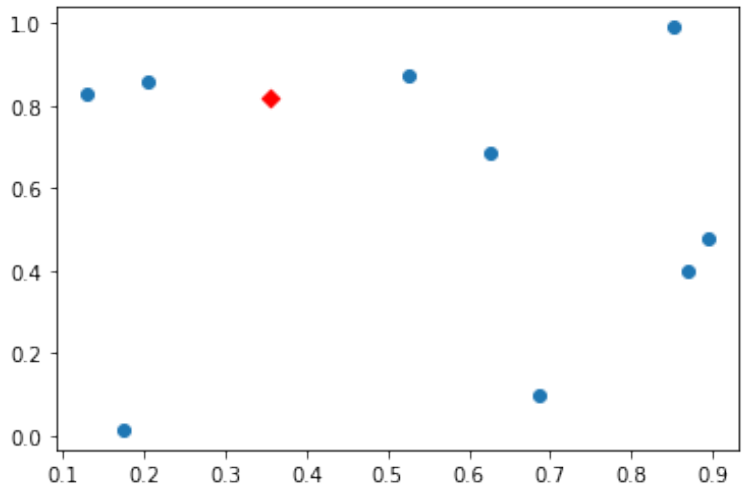

Figure 3.1: Scatter of 10 nodes

The distance matrix has to be created. The function 'Distance' will return the Euclidean distance of a set of coordinates to another.

```
def Distance(x1, y1, x2, y2):
    return sqrt((x1-x2)**2+(y1-y2)**2)

length = len(Coordinates)
Distance_Matrix = np.zeros((length, length))
for i in range(length):
  for j in range(length):
    Distance_Matrix[i][j] = Distance(Coordinates[i][1],  Coordinates[i
   ][2], Coordinates[j][1], Coordinates[j][2])
```

Afterwards, the list "All_Possible_Tours" with all the combinations has to be produced by 'permutations' function, which takes as input the first possible tour [2,3,4,5,6,7,8,9,10] taken by 'First_Tour' function. That will give a list in size of $(n-1)!$. One more important things is, that with the 'timer' function, the whole process must be timed, in order to benchmark the algorithms.

```
start = timer()

firstpossible = First_Tour()
All_Possible_Tours=list(permutations(firstpossible))
```

The algorithm will use a for-loop to scan the list "All_Possible_Tours", and compute each solution with the 'FindCurrentCost' function about its cost. If the cost is less than the best current, the shortest tour will be changed.

```
Shortest_Tour_Cost = 9999

for i in range(len(All_Possible_Tours)):
    CurrentCost=FindCurrentCost(list(All_Possible_Tours[i]))
    if CurrentCost < Shortest_Tour_Cost:
        Shortest_Tour_Cost = CurrentCost
        Shortest_Tour = list(All_Possible_Tours[i])
```

```
20          Shortest_Tour.insert(0,1)
21          Shortest_Tour.append(1)
22
23  end = timer()
```

Follows the optimal route and its cost for the problem of 10 nodes. The calculation time was 2.2188(*sec*). Figure 3.2 shows the optimal answer plotted.

$$(1\ 6\ 9\ 3\ 5\ 7\ 10\ 4\ 2\ 8\ 1) = 3.246$$

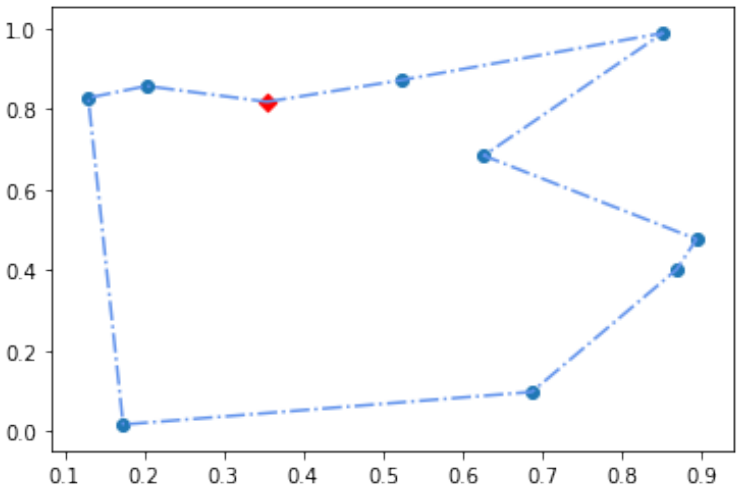

Figure 3.2: Brute Force tour of 10 nodes

3.2.2 Dynamic Programming

Dynamic Programming as is an exact algorithm, is going to give the optimal solution. From the chapter 2, the operation of that algorithm is known.

The Dynamic Programming deal with a TSP of 15 nodes, scattered at figure 3.3.

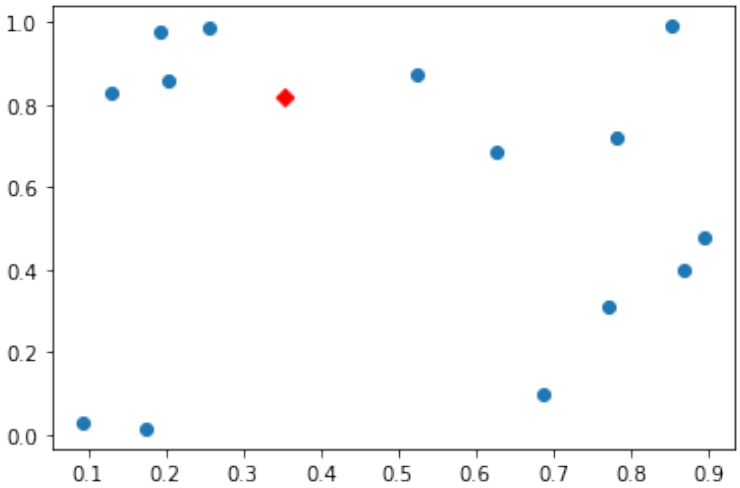

Figure 3.3: Scatter of 15 nodes

The DP algorithm firstly creates the "Saved_Values" dictionary, in which all the computed sub-tours will be saved. The computation process will be completed by the 'get_minimum' function. When the minimum cost is found, the algorithm the algorithm creates the "Shortest_Tour" list and appends it the optimal tour.

```
n = len(Coordinates)
Saved_Values = {}
p = []
start = timer()
for x in range(1, n):
  Saved_Values[x + 1, ()] = Distance_Matrix[x][0]

get_minimum(1, tuple(range(2,len(Distance_Matrix)+1)))

#Shortest_Tour list creation
solution = p.pop()
Shortest_Tour = [1]
Shortest_Tour.append(solution[1][0])

for x in range(n - 2):
  for new_solution in p:
    if tuple(solution[1]) == new_solution[0]:
      solution = new_solution
      Shortest_Tour.append(solution[1][0])
      break

Shortest_Tour.append(1)
```

The 'get_minimum' function takes as input the depot node '1' and a tuple with the rest of the nodes, aiming to compute all the combinations. At the time when a pre-calculated combination is going to recomputed, the algorithm takes its value from the "Saved_Values" dictionary.

```
def get_minimum(k, a):

    if (k, a) in Saved_Values:
        return Saved_Values[k, a]

    values = []
    all_min = []
    for j in a:
        set_a = deepcopy(list(a))
        set_a.remove(j)
        all_min.append([j, tuple(set_a)])
        result = get_minimum(j, tuple(set_a))
        values.append(Distance_Matrix[k-1][j-1] + result)

    Saved_Values[k, a] = min(values)
    p.append(((k, a), all_min[values.index(Saved_Values[k, a])]))

    return Saved_Values[k, a]
```

The tour that DP algorithm gave as the optimal answer is at the figure below (3.4). The cost of the tour is 3.603. The calculation time that DP need was 6.55(*sec*).

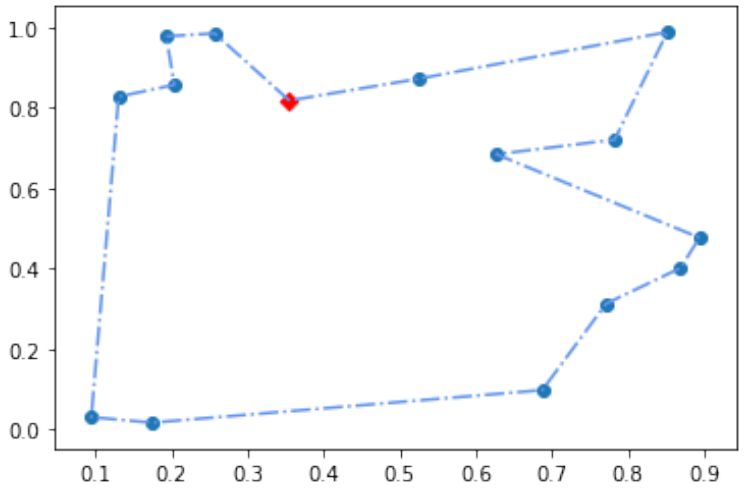

Figure 3.4: Dynamic Programming tour of 15 nodes

3.3 Heuristic Algorithms

3.3.1 Greedy Algorithm

The greedy algorithm chooses the next visited node about the distance that is needed to cover from the current node. So, the algorithm begins from the 1st line of the distance matrix, which presents the depot, and picks the nearest node, let's say the 3rd node. The algorithm has to set the 3rd column to infinity, with the aim of not visiting this node again from another node. So, the algorithm is at row 3 of the distance matrix and searches the nearest node. The 1st node it cannot revisited, as the cost now is huge.

At this example, the TSP has 30 nodes:

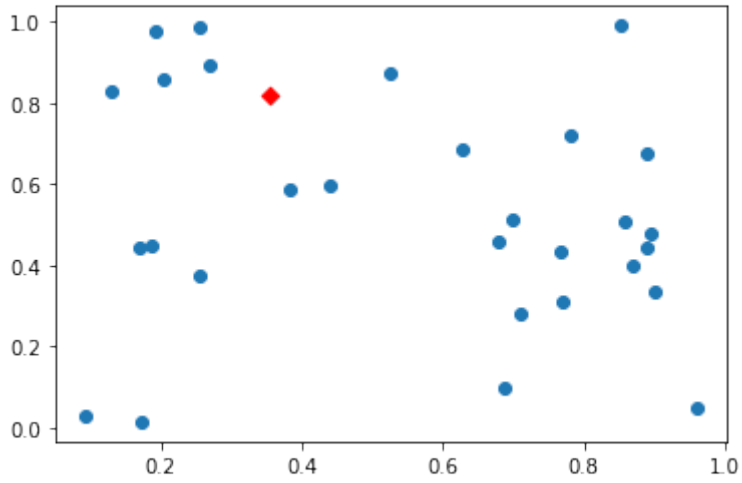

Figure 3.5: Scatter of 30 nodes

Firstly, the matrix "dis" that will contain all the distances has to be initialized. The diagonal of the "Distance_Matrix" was zero, therefore, the algorithm will choose that as the next node. For this reason the diagonal has to set to infinity.

```
n = len(Coordinates)
dis=np.full((n,n),float(9999999))
for i in range(n):
    for j in range(n):
        if i == j:continue
        else:
            dis[i,j]=Distance_Matrix[i][j]

dis[:,0]=9999999
```

At last, the 1st node appended to the "Greedy_Tour", as the route has to finish at the starting point.

Then, a while loop is set, to scan the "dis" matrix. When the algorithm find the smallest cost, takes its index, as it shows the number of the node, and appends it to the "Greedy_Tour" list. If a new node have been picked, its column has to set to infinity. At the time that all nodes have been satisfied, the counter equals $n - 1$ and the while loop breaks.

```
Greedy_Tour=[1]
i=0
counter=0

while True:
    idx=np.where(dis == min(dis[i,:]))
    if len(idx[1]) == 1:
        Greedy_Tour.append(int(idx[1]+1))
        dis[:,int(idx[1])]=9999999
        i=int(idx[1])
    else:
        Greedy_Tour.append(int(idx[1][0]+1))
        dis[:,int(idx[0][1])]=9999999
        i=int(idx[0][1])
    del idx

    counter+=1
    if counter == n-1:
        break

Greedy_Tour.append(1)
```

The route that it has given by the Greedy algorithm, is plotted in figure 3.6. The cost of the route by is 5.648 and the calculation time was 2.2(*msec*).

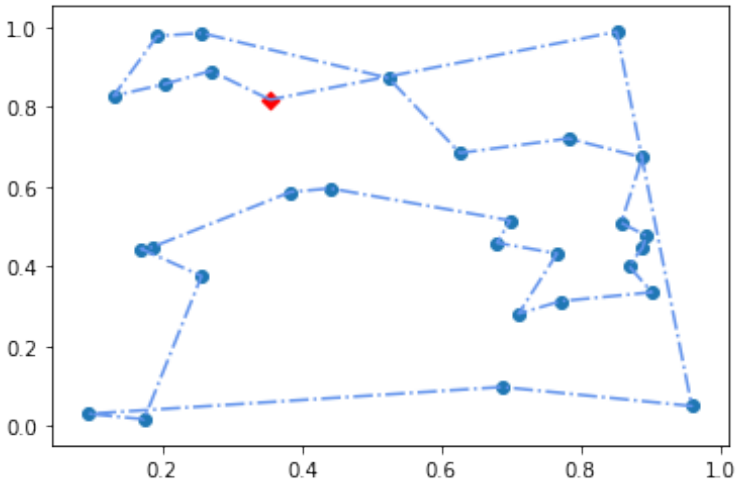

Figure 3.6: Greedy tour of 30 nodes

3.3.2 2-Opt Algorithm

From Chapter 2, is clear that the 2-Opt algorithm is a local-search heuristic. It has to begin with an initial tour as a solution. At the time of the operation, the algorithm tries to fix the tour, aiming to reduce the cost. The program stops when no more improvement could be, at the current solution. It is important to notice that 2-Opt algorithm as is a local-search heuristic, gets trapped into local optimum very often. As a result, every time that the algorithm runs, the answer changes, so the optimal answer remains uncertain.

At this example, 40 nodes will be used for solving the TSP. Figure 3.7 shows the scatter plot of the coordinates.

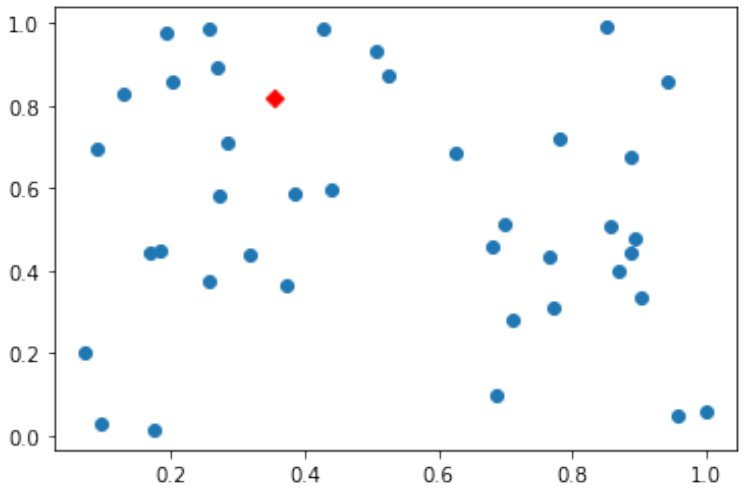

Figure 3.7: Scatter of 40 nodes

The algorithm takes from the 'First_Tour' function a possible tour. Then, a while loop runs until the "improved" remains as 'False' at the end of the loop. That would mean that no further improvements could be. Inside the while loop, 2 for loops scan the route, node to node, for possible improvements. The reverse method is held for improving the tour. Any subset of the solution checked if by reversing it, the cost get decreased.

```
start = timer()

route = First_Tour()
best = route
improved = True
while improved:
    improved = False
    for i in range(1,len(route)-2):
        for j in range(i+1, len(route)):
            if j-i == 1: continue
            new_route = route[:]
            new_route[i:j] = route[j-1:i-1:-1]
            if FindCurrentCost(new_route) < FindCurrentCost(best):
              best = new_route
              improved = True
    route = best

end = timer()
```

The result that 2-Opt gave, is the tour at figure 3.8. The cost of this tour is 6.214 and the calculation time was 0.87(*sec*). The tour is may not the optimal solution of that problem. Several possible tours could be given as an answer by 2-Opt.

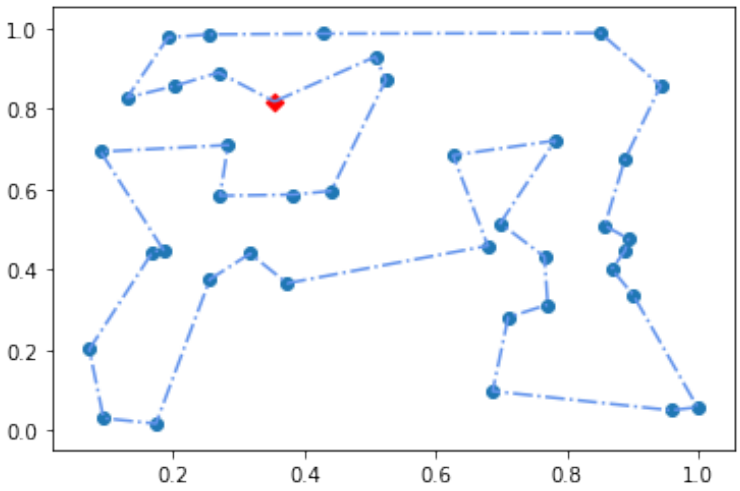

Figure 3.8: 2-Opt tour of 40 nodes

3.4 Meta-Heuristic Algorithms

3.4.1 Simulated Annealing Algorithm

For the implementation of the Simulated Annealing 2 loops was set. The outer loop, which depends on the temperature, runs for 2470 times. That has come from the initial temperature which was at 20, the minimum temperature which was at 0.001 and the alpha 0.996. The temperature is decreased exponentially by multiply itself with alpha, figure 3.9. The inner loop was set at 4 times the amount of nodes, which at the example of 40 nodes gives 160 times. So the algorithm runs for 395200 times in total.

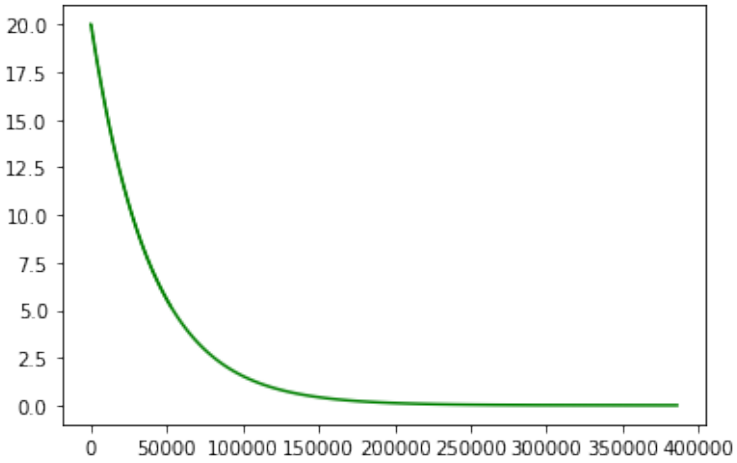

Figure 3.9: Temperature decreasing

The TSP that SA forced to solve was the same as before, figure 3.10.

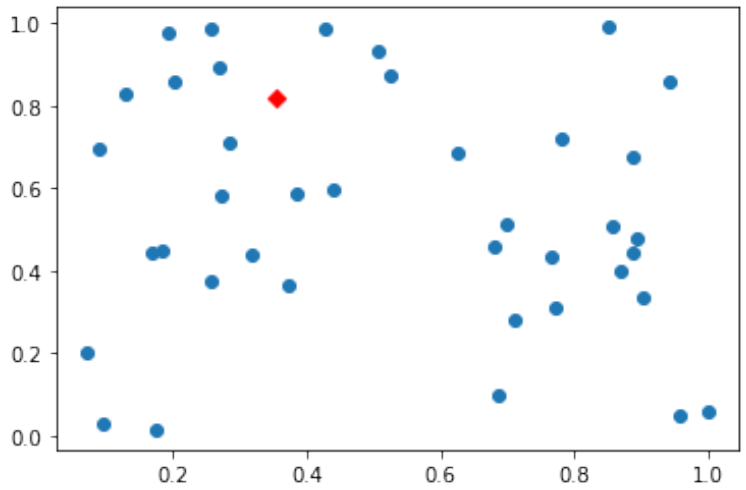

Figure 3.10: Scatter of 40 Nodes

The first tour X_0 and its cost Fx_0 has to be initialized before entering the while loop. As the algorithm enters into the two loops, the new tour X_{new} and its cost Fx_{new} has to be computed. If the difference between them is positive ($Df > 0$) the X_{new} take the place of the X_0. Else metropolis rule take part in the algorithm. That process continues until the temperature is equal or below of the minimum temperature.

```
Xo = FirstTour[:]
Fxo=FindCurrentCost(FirstTour)

while True:
    for i in range(len(Coordinates)*4):
        Xnew=NewTourRev(Xo[:])
        Fxnew=FindCurrentCost(Xnew)

        Df=Fxo-Fxnew

        if Df > 0:
            Xo=Xnew[:]
            Fxo = Fxnew
        else:
            p = exp(Df/Temperature)
            ran=random()
            if ran < p:
                Xo=Xnew[:]
                Fxo = Fxnew

    Temperature*= alpha
    if Temperature <= mintemp:break
```

The 'NewTourRev' function picks a sub-tour of $X - 0$, and reverse it in order to generate the X_{new}.

```
def NewTourRev(seq):
    tmp=seq[0]
    del seq[0],seq[-1]
    i = randint(0,  len(seq)- 2)
    j = randint(i+2, len(seq))
    seq[i:j] = reversed(seq[i:j])
    seq = [tmp] + seq + [tmp]
    return seq
```

A very interesting graph, as it helps with the understanding of the SA's efficiency is at figure 3.11. That figure shows the "p" values when a worse solution examined by the metropolis rule (equation 2.6). The "p" takes numbers close to 1 when the temperature is high, so its more possible to swap your current solutions with one with higher cost. When the temperature is close to the minimum temperature, the "p" goes close to zero.

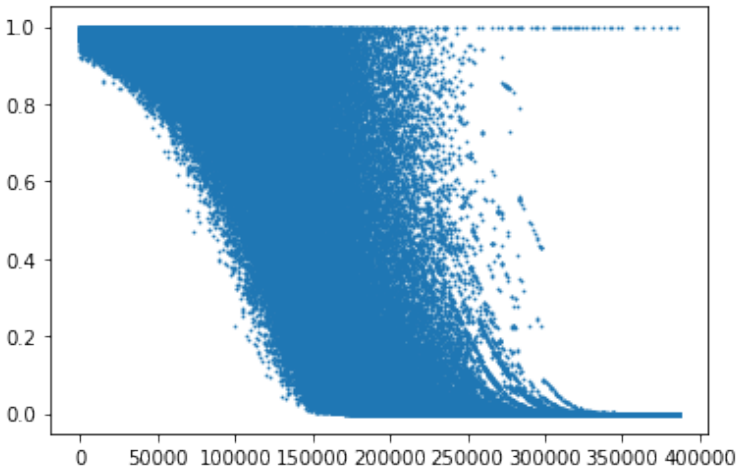

Figure 3.11: Acceptance probability by Metropolis rule

In more detail, at figure 3.12, is with red color the "p" values when the algorithm accepted a new solution, and with blue, the "p" values when rejected. That figure makes obvious the algorithm's philosophy.

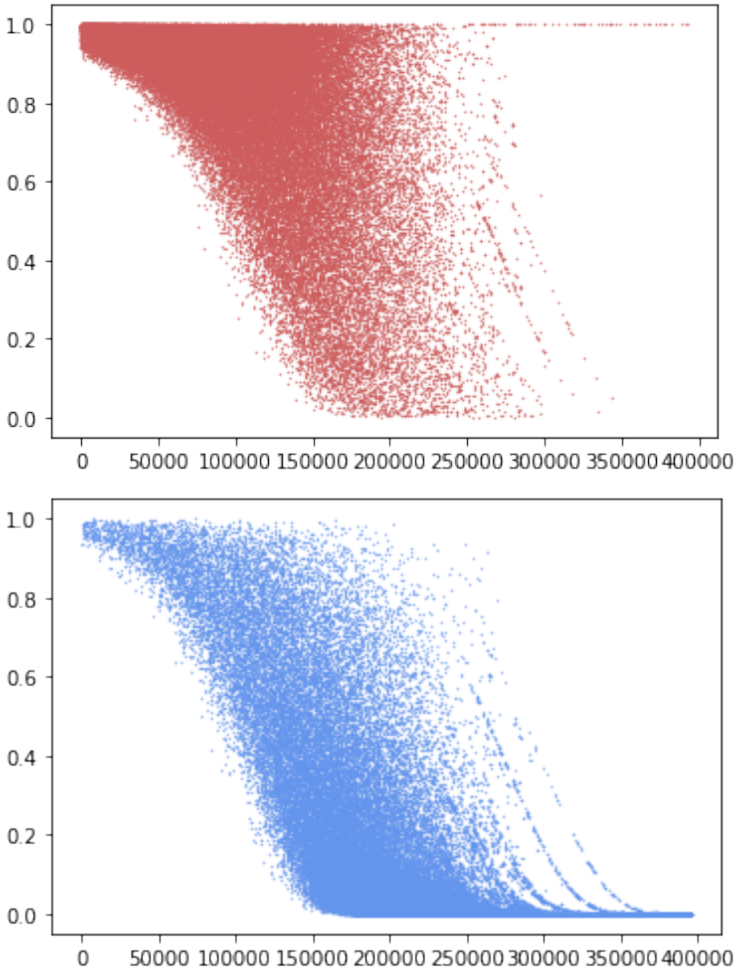

Figure 3.12: Metropolis Rule, Acceptance Rejection

The cost function is at figure 3.13. The red graph is the shortest tour that SA had found. The blue graph shows the cost of each X_{new} that SA creates. The amplitude of the graph in the first 100000 iterations is high. Oppositely, at the last 100000 iterations, the amplitude had been decreased. At this time, the algorithm does not have the energy of jumping a local optimum, so it becomes more stable and generates new solutions with the behavior of local search.

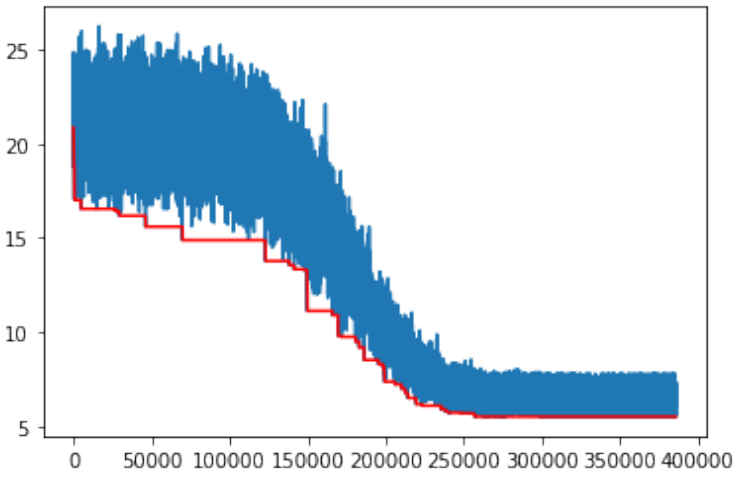

Figure 3.13: Simulated Annealing cost function

The best tour that SA found is at figure 3.14. Its cost was 5.500 and the calculation time took 8.4(*sec*)

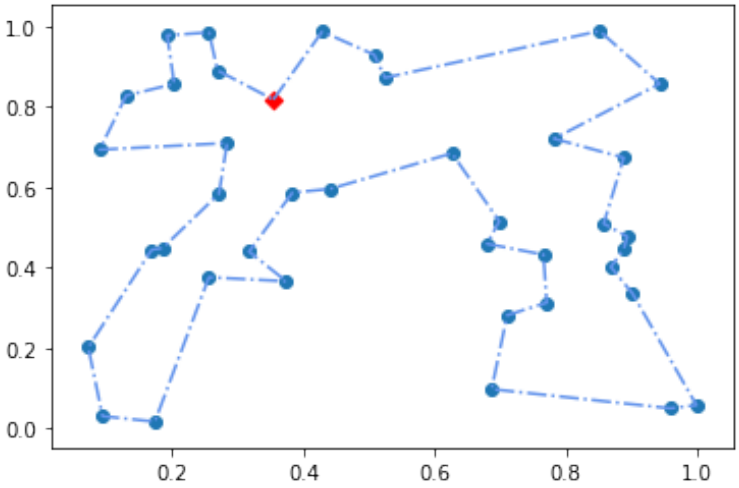

Figure 3.14: Simulated Annealing tour of 40 Nodes

3.4.2 Genetic Algorithm

At the implementation of the Genetic Algorithm, the size of the population selected at 150 and was executed for 1000 generations. The PMX operator used as the crossover method. As for the mutation, was chosen the Inversion mutation operator. The crossover operator had the 75% probability, the mutation operator had the 24% probability and the remaining 1% belonged to the random factor of creating new chromosomes. In figure 3.15 is the histogram of the methods of producing new chromosomes.

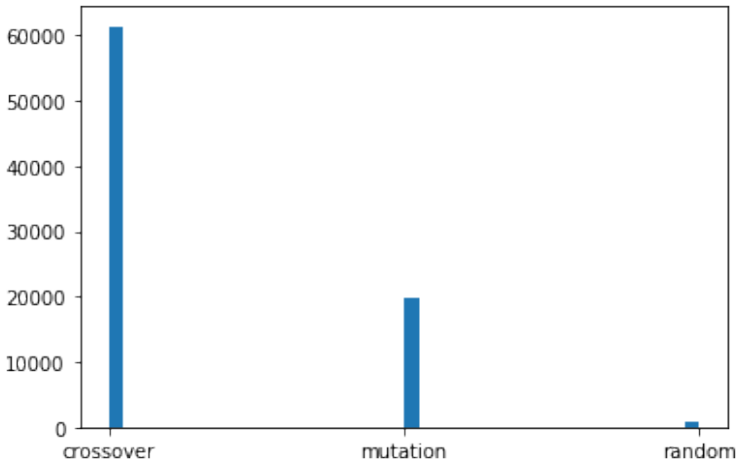

Figure 3.15: New chromosome method histogram

The selection method that was used is the Fitness Proportionate Selection (also known as Roulette Wheel Selection), in combination with the Elitism Selection, which chose the 5% of the parent population passing to the next generation as elites.

The TSP that was chosen was the same as before, figure 3.16:

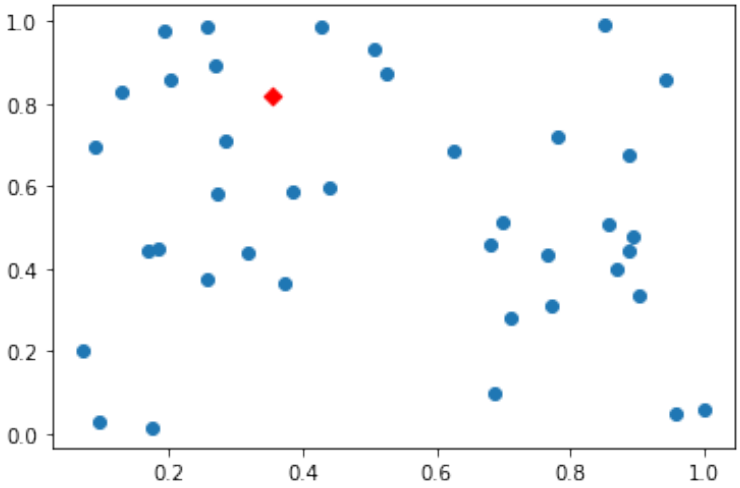

Figure 3.16: Scatter of 40 nodes

The initialization of the GA follows:

```
Population = 150
Generations = 1000

elitism = 5 # of 100

# Propabilitties
Crossover_Propabillity = 75
Mutation_Propabillity = 24
New_Random_Propabillity = 100 - Crossover_Propability -
    Mutation_Propabillity
```

Before entering the main loop of the GA, the first population has to be created randomly. Next, the cost of each chromosome has to be computed. The 'First-Tour' function returns a possible solution of the TSP. The 'FirstParents' function takes as input the "FirstTour" and creates a list of chromosomes with the size of the "Population", named as Parents. Moreover, 'CostofList' returns a list, which includes the cost of every individual chromosome.

```
FirstTour=First_Tour()
Parents=FirstParents(Population,FirstTour[:])
Parents_Cost_List=CostofList(Parents[:])
```

Therefore, the GA could begin. The first operation that GA does, is to create the "Children" list by 'Elitism' function. That list contains 7 chromosomes (5% of 150) with the best fitness value from all the population. Then, the 'RoulleteWheelSelectio02nArray' function, take the list of the costs of the parents in order to give a proportionate range, depended on the cost of each parent. At that time, the while loop begins by picking an integer from 1 to 100. That will decide about which operation would take place for breeding new population.

```
start = timer()
for rnd in range(Generations):
  Children = Elitism(Parents, Parents_Cost_List, elitism)
  Proporsional_Selection_Array = RoulleteWheelSelectionArray(
    Parents_Cost_List)
  while True:
    randomnum=randint(1,100)
```

The 'Elitism' function is:

```
def Elitism(lst, lstcost, percentage):
  elite = []
  num = (len(lst)*(percentage))//100

  for i in range(num):
    idx=lstcost.index(min(lstcost))
    elite.append(lst[idx])
    lstcost[idx]=99999999

  return elite
```

The 'RoulleteWheelSelectionArray' function returns a 150×3 matrix. The first and the second column, is the begining and the ending of the range of the normalized cost. The third column is the size of that range. The greater range gives more possibilities to the parent for breeding. The whole function is:

```
def RoulleteWheelSelectionArray(costs):
    n = len(costs)
    minim=min(costs)
    maxim=max(costs)
    reverse=np.zeros(n)
    normal=np.zeros(n)
    prop_sel=np.zeros((n,3))

    for i in range(n):
        reverse[i]=(maxim-costs[i])+minim
    summ=sum(reverse)
    prop_sel[0,0]=0
    for i in range(n-1):
        normal[i]= reverse[i]/summ
        prop_sel[i,1] = normal[i] + prop_sel[i,0]
        prop_sel[i,2] = normal[i]
        prop_sel[i+1,0] = prop_sel[i,1]

    normal[n-1]= reverse[n-1]/summ
    prop_sel[n-1,1] = normal[n-1] + prop_sel[n-1,0]
    prop_sel[n-1,2] = normal[n-1]

    return prop_sel
```

The decision about which the method of breeding takes place. If the number "randomnum" is 1, a new random chromosome will be appended in the "Children" list. Else if the "randomnum" is at [2,25], a new chromosome will be created by the 'Mut_Inversion' operator. Else if the "randomnum" is at the rest [26,100], two chromosomes will be created by the 'Crossover_PMX' function.

```
    if randomnum <= New_Random_Propabillity:
      Children.append(NewRandom(FirstTour[:]))

    elif randomnum > New_Random_Propabillity and randomnum <= (
   Mutation_Propabillity + New_Random_Propabillity) :
      idx=pickone(Proporsional_Selection_Array)
      chromosome = Parents[idx]
      Children.append(Mut_Inversion(chromosome[:]))

    elif randomnum > (Mutation_Propabillity + New_Random_Propabillity):
      idx1=pickone(Proporsional_Selection_Array)
      while True:
        idx2=pickone(Proporsional_Selection_Array)
        if idx2 != idx1:break
      kid1, kid2 = Crossover_PMX(Parents[idx1][:], Parents[idx2][:])
      Children.append(kid1)
      if len(Children) == Population:break
      Children.append(kid2)
```

The 'Mut_Inversion' function create a single offspring from a single parent which have been picked from the 'pickone' function. The Inversion method already discussed at Chapter 2. The function's implementation is:

```
def Mut_Inversion(parent):
    tmp=parent[0]
    del parent[0],parent[-1]

    i = randint(0,  len(parent)- 2)
    j = randint(i+2, len(parent))

    parent[i:j] = reversed(parent[i:j])

    parent.insert(0,tmp)
    parent.append(tmp)

    return parent
```

The 'Crossover_PMX' takes as input two parents, aiming to return two children. The process of that operator already discussed at Chapter 2. Follows the code:

```
def Crossover_PMX(par1,par2):
  del par1[0],par2[0],par1[-1],par2[-1]
  i = randint(0,  len(par1)- 3)
  j = randint(i+3, len(par1))

  tmp1 = par1[i:j]
  tmp2 = par2[i:j]

  off2 = [None] * len(par1)
  off1 = [None] * len(par1)

  off1[i:j]=par2[i:j]
  off2[i:j]=par1[i:j]

  for k in range(len(par1)):
    if off2[k] == None:
      if  not par2[k] in off2:
        off2[k] = par2[k]
      else:
        for x in tmp2:
          if not x in off2:
            off2[k] = x
          else:continue
    else:continue
  off2 = [1] + off2 + [1]

  for k in range(len(par1)):
    if off1[k] == None:
      if  not par1[k] in off1:
        off1[k] = par1[k]
      else:
        for x in tmp1:
          if not x in off1:
            off1[k] = x
```

```
35            else:continue
36      else:continue
37    off1 = [1] + off1 + [1]
38
39    return off1,off2
```

When the size of the "Children" list become as the population the while loop breaks. The children become parents and another generation begins.

```
57      if len(Children) == Population:break
58
59    # Children -> Parents
60    Parents = Children[:]
61    Parents_Cost_List = CostofList(Parents)
```

A way to inspect the evolution of the GA, is by looking at the minimum cost at each step. So, at every generation the cost of the best chromosome, saved in order to create the Fitness graph, figure 3.17. The cost begins around 17, by a random solution. As the GA evolves its chromosomes the cost reduces exponentially until the first 400 generations. Then for the rest 600 generations, the TSP is close to being solved to optimality. Only a few better solutions were given by the GA at this point.

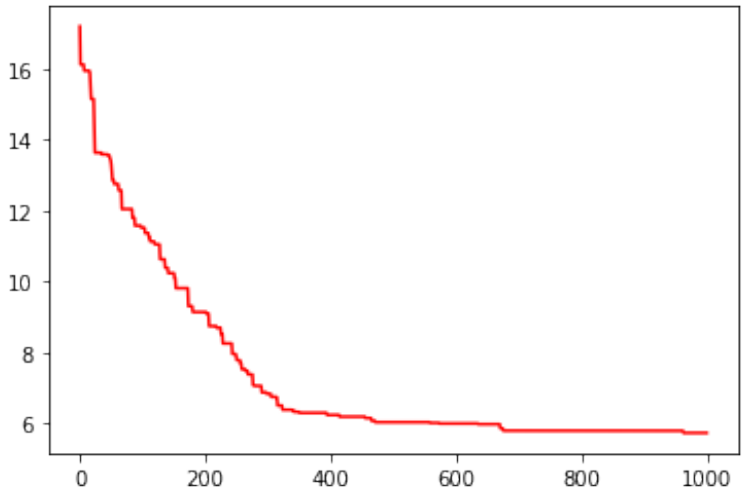

Figure 3.17: Genetic Algorithm Fitness Function

The result of the GA for the TSP of 40 nodes, is a tour with a cost at 5.755 at 13.73(*sec*). The tour is plotted at figure 3.18.

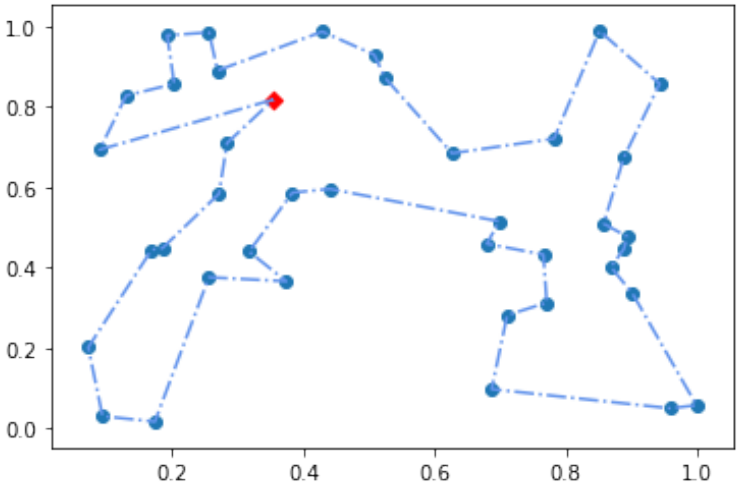

Figure 3.18: Genetic Algorithm tour of 40 nodes

3.5 Summarize

In this chapter, the implementations of each algorithm held. Every algorithm was set to specific TSP with a different number of nodes. The results differ from application to application. Further, the main and more important functions of each algorithm were included. Also, at the meta-heuristic algorithms, more graphs about the cost function and their operation methods were pinned. In the next chapter, all the algorithms will be compared to each other, in the calculation time, in the shortest tour that they found and how accurate were been.

Chapter 4

Comparison of the Algorithms

4.1 Algorithms Comparison

In this section, the algorithms that this Thesis presents will be compared in the calculation time and in their giving results of the TSP that they forced to solve. Each algorithm solves every TSP 100 times, in order to compute mean values of the calculation time, the best answer that they found and the accuracy rate that they provide on the solution. The accuracy term is the percentage of finding the best tours among 100 times. As is mentioned before, the coordinates for each problem held by a list, so as the problem needs to be more complicated, more nodes will be taken from that list for the TSP. In the beginning, the thesis benchmarks the two exact algorithms, which had the biggest time complexity.

Firstly the Brute Force method, as it gave only the optimal answer in each TSP, is 100% accurate. So the mean value of the costs of 100 times was the shortest tour's cost. The disadvantage of this algorithm is the time and space complexity as they both grow exponentially. Table 4.1 shows, that for the TSP of 12 nodes, 242(*sec*) needed to solve that problem. When the algorithm forced to solve the 13 nodes TSP, the Google Colab environment runs out of RAM, with the approximate calculation time would be around 40 minutes. So the Brute Force stops at the 12 nodes TSP

Num of Coordinates	Time (sec)	Minimum Cost
4	0,000033	2.15 (100%)
5	0,000115	2.535 (100%)
6	0,000452	2.625 (100%)
7	0,003309	2.644 (100%)
8	0,024326	2.655 (100%)
9	0,205632	3.167 (100%)
10	1,977503	3.246 (100%)
11	20,920176	3.255 (100%)
12	242,315981	3.432 (100%)

Table 4.1: Brute Force Benchmark

The Dynamic Programming created in order to decrease the calculation time of the Brute Force method. According to table 4.2 the DP algorithm has greater calculation time than the BF until the first 7 nodes. That is because the algorithms differ from one another in their perspective, as the DP algorithm is more complicated in its way of solving a problem. The advantage of the DP from the BF is more clear from the 8 nodes TSP. As its time complexity is smaller from exponential, the Dynamic Programming becomes faster. DP's final set of nodes is the 19 nodes problem, in which the calculation time was 200(*sec*). As is expected, these two algorithms did not have any difference in their answers about the shortest tour, as they are both exact methods.

Num of Coordinates	Time (sec)	Minimum Cost
4	0,0006	2.15 (100%)
5	0,0014	2.535 (100%)
6	0,0019	2.625 (100%)
7	0,0046	2.644 (100%)
8	0,0096	2.655 (100%)
9	0,0239	3.167 (100%)
10	0,0612	3.246 (100%)
11	0,1663	3.255 (100%)
12	0,5061	3.432 (100%)
13	1,1625	3.490 (100%)
14	2,8453	3.557 (100%)
15	6,7468	3.603 (100%)
16	15,871	3.646 (100%)
17	37,0488	3.665 (100%)
18	83,8431	3.843 (100%)
19	199,6798	3.938 (100%)

Table 4.2: Dynamic Programming Benchmark

The Greedy algorithm results are presented at table 4.3. This algorithm has significant small calculation time. Its solutions, about the TSP, were greater than the already known ones, even from the 4-nodes TSP. Only the 7-nodes TSP solved by giving the optimal answer, which made itself by chance.

Num of Coordinates	Time (sec)	Minimum Cost
4	0,000041	2,737 (100%)
5	0,000095	2,758 (100%)
6	0,000064	3,209 (100%)
7	0,000093	2,644 (100%)
8	0,000111	2,882 (100%)
9	0,000133	3,862 (100%)
10	0,000156	3,851 (100%)
11	0,000171	3,860 (100%)
12	0,000170	3,956 (100%)
13	0,000197	4,046 (100%)
14	0,000243	4,163 (100%)
15	0,000255	4,188 (100%)
16	0,000277	4,233 (100%)
17	0,000354	4,252 (100%)
18	0,000334	4,274 (100%)
19	0,000355	4,369 (100%)
20	0,000400	4,378 (100%)
22	0,000517	4,459 (100%)
25	0,000673	5,642 (100%)
35	0,001103	6,450 (100%)
45	0,001772	7,235 (100%)
60	0,003109	7,463 (100%)
75	0,004958	8,482 (100%)
90	0,007142	9,067 (100%)
110	0,011298	10,070 (100%)
140	0,019623	11,380 (100%)

Table 4.3: Greedy Algorithm Benchmark

The 2-Opt algorithm is the first heuristic that tries to solve big size TSP and gives reasonable solutions. Figure 4.4 shows the results of the benchmark. This algorithm solves the 45 node TSP in about 1(*sec*). Its giving answers are fair enough, and in some cases the optimal. In more detail. figure 4.4 makes obvious that the algorithm until the 9 nodes give with 100% probability the optimal answer. From the 10 nodes, the 2-Opt became less accurate. At the 60 nodes TSP, the algorithm succeed to find that solution only once, which is not certain if it is the optimal answer. Surprisingly, the algorithm until the 19 nodes problem gave the optimal answer that is known from the DP (see fig. 4.2), even with 20% accuracy. The 2-Opt algorithm after the 90 nodes starts to procrastinate, in order to make him useless for this scale of problem.

Num of Coordinates	Time (sec)	Minimum Cost	Mean Cost
4	0,000045	2.15 (100%)	2,150
5	0,000115	2.535 (100%)	2,535
6	0,002250	2.625 (100%)	2,625
7	0,000502	2.644 (100%)	2,644
8	0,000863	2.655 (100%)	2,655
9	0,001625	3.167 (100%)	3,167
10	0,002714	3.246 (99%)	3,249
11	0,003965	3.255 (98%)	3,264
12	0,005698	3.432 (91%)	3,459
13	0,007634	3.490 (67%)	3,522
14	0,010811	3.557 (65%)	3,580
15	0,013882	3.603 (55%)	3,660
16	0,019162	3.646 (70%)	3,707
17	0,023541	3.665 (65%)	3,750
18	0,030765	3.843 (35%)	3,860
19	0,037522	3.938 (21%)	4,053
20	0,045353	4.001 (37%)	4,063
22	0,071704	4.082 (15%)	4,153
25	0,116553	4.155 (20%)	4,288
35	0,461513	5.205 (5%)	5,409
45	1,282499	5.841 (2%)	6,051
60	4,061360	6.757 (1%)	6,992
75	10,814159	7.625 (1%)	7,932
90	23,183003	8.276 (1%)	8,513
110	53,360063	9.115 (1%)	9,274
140	134,450488	9.718 (1%)	10,152

Table 4.4: 2-Opt Algorithm Benchmark

The Simulated Annealing is the first meta-heuristic which forced to benchmark. In table 4.5 the results of this algorithm are listed. The SA solved even the 140 nodes TSP in 30 seconds and gave better solutions from the 2-Opt. From the table, SA makes obvious its big advantage of finding better solutions in each TSP. Until the 25-nodes TSP, which means $62 * 10^{22}$ possible solutions, the algorithm gave the solution with 100% probability, which may is the optimal one. The algorithm continues to give better solutions than the 2-Opt at the rest of the benchmark.

Num of Coordinates	Time (sec)	Minimum Cost	Mean Cost
4	3,534196	2.150 (100%)	2,150
5	3,750613	2.535 (100%)	2,535
6	3,969361	2.625 (100%)	2,625
7	4,078313	2.644 (100%)	2,644
8	4,421785	2.655 (100%)	2,655
9	4,573957	3.167 (100%)	3,167
10	4,881223	3.246 (100%)	3,246
11	5,087911	3.255 (100%)	3,255
12	5,250414	3.432 (100%)	3,432
13	5,462118	3.490 (100%)	3,490
14	5,635319	3.557 (100%)	3,557
15	5,848986	3.603 (100%)	3,603
16	5,988934	3.646 (100%)	3,646
17	6,285943	3.665 (100%)	3,665
18	6,495266	3.843 (100%)	3,843
19	6,693107	3.938 (100%)	3,938
20	6,883251	4.001 (100%)	4,001
22	7,268323	4.082 (100%)	4,082
25	7,849679	4.155 (100%)	4,155
35	9,784093	5.205 (45%)	5,218
45	11,81127	5.764 (18%)	5,809
60	14,70318	6.569 (5%)	6,676
75	17,83683	7.371 (2%)	7,526
90	20,75842	7.905 (1%)	8,143
110	24,86028	8.693 (1%)	8,957
140	30,81813	9.450 (1%)	9,835

Table 4.5: Simulated Annealing Benchmark

After the Simulated Annealing, the Genetic Algorithm has to be forced in the benchmarks.At the beginning, the algorithm presented as a better solution from the SA, as until 14 nodes gave with 100% the optimal answers in less time. Then, this accuracy rate decreased till the 35 nodes, which have been equal to 1%. In the previous examined problem with 25 nodes, the Simulated Annealing was 100% accurate. Nevertheless, the GA succeeded to give the best-known answer, even once, until the 25-nodes problem in less time, but the mean value of the costs is higher as the algorithm is less accurate. The benchmark went worse for the GA from the 60 nodes and beyond, where the costs have a big difference from the SA even from the 2-Opt. The algorithm's solution of the 140-nodes TSP was the cost of 33.9 in 73(*sec*). Which means that the GA needs more time to solve problem in this scale.

Num of Coordinates	Time (sec)	Minimum Cost	Mean Cost
4	3,474392	2.15 (100%)	2,150
5	3,551503	2.535 (100%)	2,535
6	3,570737	2.625 (100%)	2,625
7	3,625181	2.644 (100%)	2,644
8	3,675830	2.655 (100%)	2,655
9	3,722495	3.167 (100%)	3,167
10	3,794725	3.246 (100%)	3,246
11	3,839345	3.255 (100%)	3,255
12	3,892131	3.432 (100%)	3,432
13	3,952425	3.490 (100%)	3,490
14	4,001136	3.557 (100%)	3,557
15	4,076643	3.603 (86%)	3,624
16	4,177042	3.646 (81%)	3,679
17	4,220813	3.665 (64%)	3,690
18	4,313189	3.843 (34%)	3,861
19	4,376954	3.938 (27%)	3,975
20	4,460556	4.001 (19%)	4,098
22	4,651634	4.082 (15%)	4,120
25	4,903328	4.155 (9%)	4,326
35	6,224841	5.387 (1%)	5,836
45	7,854988	6.444 (1%)	7,432
60	11,74672	9.721 (1%)	11,146
75	17,37416	13.479 (1%)	15,398
90	25,30950	17.895 (1%)	19,986
110	40,03283	24.279 (1%)	26,415
140	73,02712	33.950 (1%)	36,871

Table 4.6: Genetic Algorithm Benchmark

From the previous results, its important to make clear the time complexity of each algorithm. Figure 4.1 shows the growth ratio of each algorithm in one chart. It is obvious how the calculation time of the 2 exact methods, increased so fast and early. Concerning the 2-Opt, SA and GA, they pass the 100-nodes problem with almost the same amount of time, but after that, both 2-Opt and GA increase their calculation time rapidly. Figure 4.1 is a clear state of making the Simulated Annealing, the algorithm with the best ranking among the others, based on the time complexity.

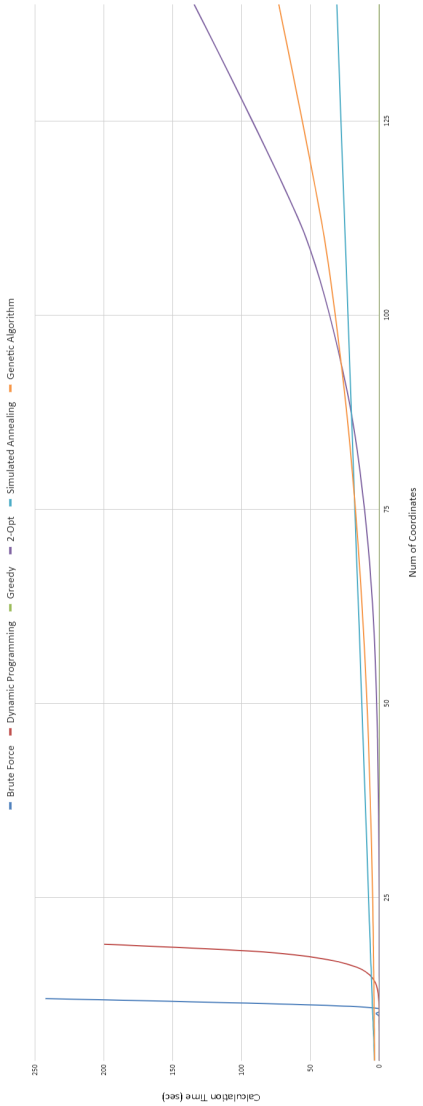

Figure 4.1: Algorithm's calculation time

Another important state that makes the SA the winning algorithm of this comparison, is its capability of finding the best answer among the others. Is unsure if a solution, given by SA, is the optimal one, but is the one with the smallest cost known. Until the 25 nodes TSP all the algorithms succeed to find the optimal solution even once. From there and after they start to have a deviation in their solutions. According to table 4.7, the figure 4.2 is created. That figure shows the large difference of the GA, which makes it impractical for the TSP. The 2-Opt gives a greater cost than the SA, even with a small difference.

Num of Coordinates	2-Opt Algorithm	Simulated Annealing	Genetic Algorithm
35	5.205	5.205	5.387
45	5.841	5.764	6.444
60	6.757	6.569	9.721
75	7.625	7.371	13.479
90	8.276	7.905	17.895
110	9.115	8.693	24.279
140	9.718	9.450	33.950

Table 4.7: Cost Comparison of 2-Opt, SA and GA

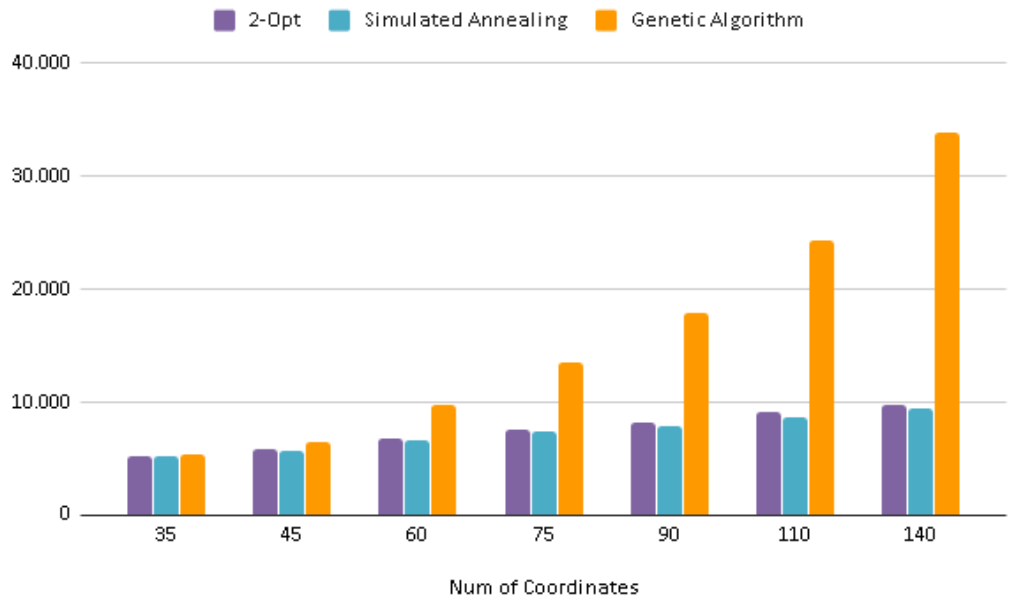

Figure 4.2: Comparing the SA, GA and 2-Opt

4.2 Summarize

In this chapter, a comparison of the algorithms that this thesis introduces, held in. That operation made the Simulated Annealing the optimal algorithm for the TSP in every category. Therefore in the next chapter, this thesis will transfer the theoretical TSP into Real-World data. The necessary changes in the implementation of the SA must be done. The transfer will be accomplished by accessing data from Google via its APIs.

Chapter 5

TSP on Real-World Data

5.1 TSP in Real Data via APIs

For the purpose of transferring the TSP in the Real-World problem, this project should access data that concern real occasions and civil attributes. In order to actualize that, the thesis should make the use of an API. An API will carry out the contact between the project's software and the Google's servers. For instance, the software makes a request in a specific format written in Python, and Google's server will send back a JSON file with all the information that is asked for. That service provided by a bunch of companies in different pricing. The decision which picked Google as the provider, is the fact that it gives a year of free trial services. Google also is a very reliable provider in GIS services in almost every country in the world.

On the previous implementation, the coordinates that this thesis used were between 1 to 0, randomly created. From now on, the user will give a csv file with all the addresses in it, and the software will convert it to a DataFrame with the Pandas module. Then it will request Google for the polar coordinates of each. That operation will be done by the Geocoding API, in the following format.

```
df = pd.read_csv("Addresses.csv")
df["LAT"] = None
df["LON"] = None
for i in range(len(df)):
  geocode_result = gmaps.geocode(df.iat[i,0])
  try:
      df.iat[i, df.columns.get_loc("LAT")] = geocode_result[0]["geometry"
    ]["location"]["lat"]
      df.iat[i, df.columns.get_loc("LON")] = geocode_result[0]["geometry"
    ]["location"]["lng"]
  except:
      lat = None
      lon = None
```

As an example, the "Addresses.csv" will contain 24 customer addresses in London plus the address of the depot which set randomly to be at 11 Porlock St London. As the "df" filled with the polar coordinates, which scattered in figure 5.1, the application has to create the distance matrix to continue its operation.

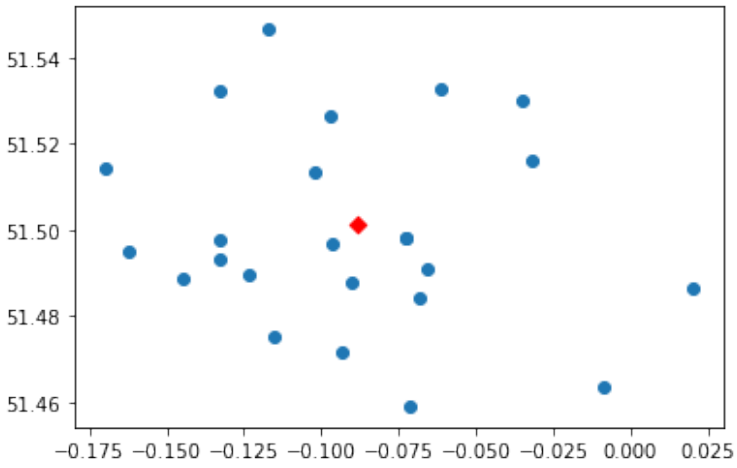

Figure 5.1: Scatter of 25 coordinates in London

In order to create the distance matrix, the Distance Matrix API will be used. Google gives the ability to the user to choose the orientation of the matrix, depending on the distance or on the duration between nodes. Moreover, it gives the ability to retrieve the duration with the traffic information in it, but this project will remain to the normal TSP without the dynamism of the traffic. So for this example will choose the duration oriented distances.

```
12 Distance_Matrix = np.zeros((len(df),len(df)))
13 for i in range(len(df)):
14   for j in range(len(df)):
15     dist = gmaps.distance_matrix( df.iat[i, df.columns.get_loc("ADDRESS")
     ], df.iat[j, df.columns.get_loc("ADDRESS")])
16     Distance_Matrix[i,j] = dist["rows"][0]["elements"][0]["duration"]["
     value"]
```

Therefore, the app is will use the SA algorithm to solve that problem as in chapter 3. The initial temperature set to 40000, the minimum to 50 and the alpha was 0.997. Their combination with the inner loop, which was 4 times the number of coordinates, they made around 220000 iterations. After 3.5(*sec*) the algorithm gave a tour with a cost at 13387. That cost is in seconds, so alternative the cost could be 3 hours and 43 minutes. The shortest tour is plotted below in figure 5.3.

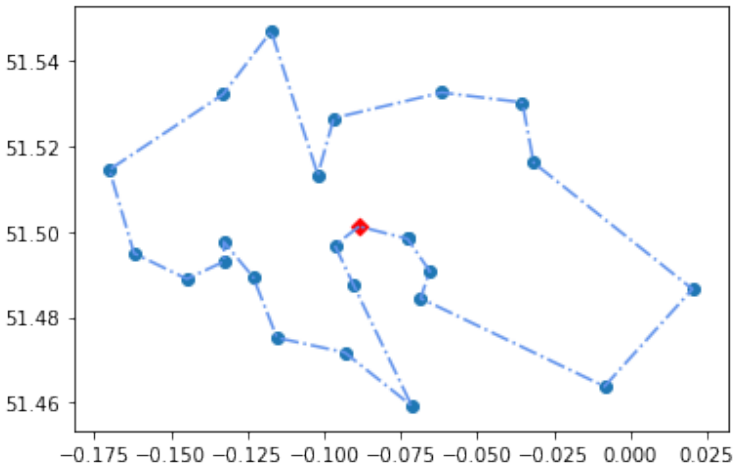

Figure 5.2: Optimal tour of 25 nodes in London

The project could be finished after this step, as the optimal tour was given, but it would be more useful to create an interactive map for easier manipulation. That can be accomplished by the Maps JavaScript API. This API takes as input the optimal tour and returns a map as an Html file plotted on Google Maps. The optimal tour has to described by a polyline that can be taken from Directions API. So the application would send requests from every address to the next one, in order to create the polyline tour. This polyline tour contains all the polar coordinates that a graph has to follow aiming to plot the shortest tour on the Google Map. The final result of this process is plotted in figure 5.3.

Figure 5.3: Optimal tour of 25 nodes in London - Google Map

In the previous map, the tour that had been selected from the SA plotted in high detail. It is possible to interact on this map, as you can zoom in on specific parts to understand better the tour guide. In figure 5.4, some parts of the tour plotted as a better description with more details of the tour.

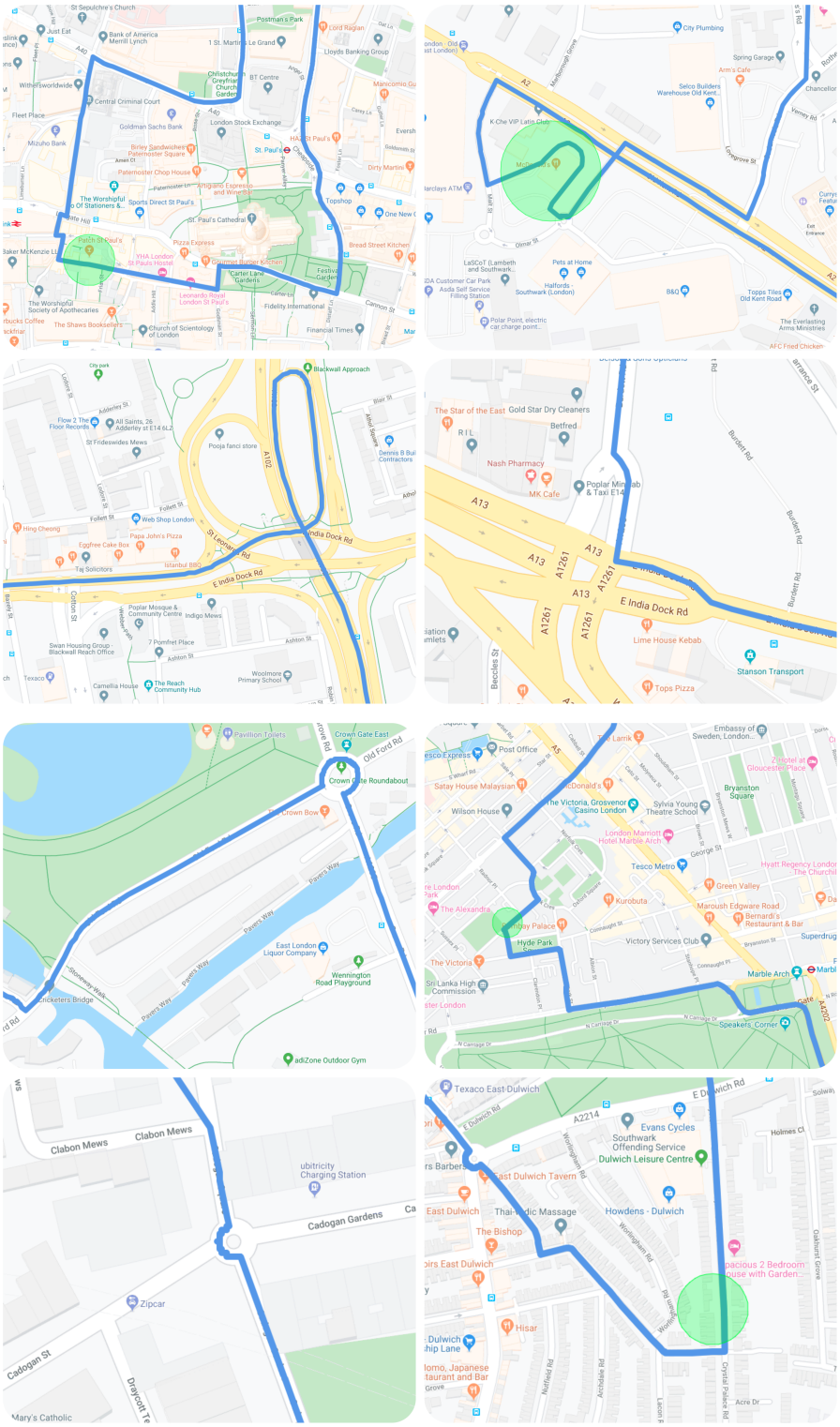

Figure 5.4: Detailed information of tour

5.2 Summarize

In this chapter, the application transfers its occasion in the real-world problem. The necessary data have been taken from Google, by using API methods. The results seem reasonable and the tour is decent. In the last chapter, this project is going to summarize the basic ideas and the conclusions that came from the previous work.

Chapter 6

Discussion - Conclusion

6.1 Thesis Survey

The main reason that makes this Thesis project important and unique, is the process of benchmarking the most famous algorithms that researchers apply to solve the TSP. An extra advantage of this work is that the algorithm applied to the Real-World problem by using Google APIs to give a result that more people could understand and use it on their daily base. This project will be uploaded for free on GitHub aiming to be an open-source project that can easily be accessed by everyone who wants to evolve with Logistic Engineering.

6.2 Results Discussion

At the beginning of the project, six of the most famous algorithms for solving the TSP have been chosen. The expected result of each one was more or less known. This Thesis confirmed that results one more time. More specifically, the two exact algorithms, the Brute Force and the Dynamic Programming, gave the optimal solution among the others in every problem that they applied. The main disadvantage of them was the execution time. That made these two algorithms incapable to use them for problems bigger than 15 customer nodes.

For this purpose, this Thesis involves two other Heuristic algorithms. The 2-Opt and the Greedy algorithm, applied to the TSP to find solutions more quickly. The Greedy algorithm was the fastest one, but the given results were unable of using them. Nevertheless, this algorithm is not completely useless, it can be used in combination with other algorithms for some improvements. On the other hand, 2-Opt turned out as a valuable method for solving the TSP. It could export pretty reasonable results in a small amount of time. Benchmarks show that the 2-Opt could not find the optimal answer in a high accuracy rate even for problems with 10 nodes. Although, it could be used by someone who needs a very fast method for solving the TSP.

Furthermore, this project examined two Meta-Heuristic algorithms, the Simulated Annealing, and the Genetic Algorithm. The Genetic Algorithm succeed to export the optimal answer with one hundred percent chance till the 14-nodes TSP. Afterward, this percentage falls rapidly, which is make the GA unstable and non reliable for solving the TSP. That may change and improved by using different and more efficient types of mutations and crossovers. Finally, this Thesis faced the winning algorithm of this procedure, the Simulated Annealing. Before this project started, the SA seemed to be the most effective algorithm for the TSP

due to the literature review. This Thesis confirmed the previous opinion, as the Simulated Annealing become the winning algorithm of the benchmarks. It succeeds to find a specific answer with a 100% chance, which seems to be optimal, till the 25-nodes problem. That benefit gets more advantages as its execution time was under 8 seconds. Which is a pretty reasonable amount of time for solving a 25-nodes TSP to optimality.

6.3 Future Work

As it concerns, the future work of this Thesis could be an improvement at the algorithms that were used. Such as trying different and more efficient ways of mutation and crossovers at the Genetic Algorithm. Moreover, it would be possible to improve this algorithm by changing the selection method. Simulated Annealing could also get some improvements by changing the method of creating new neighbor routes.

This project dealt with the most famous Logistic problem, but it is also the most basic problem. As a possible future work, the problem could be more dynamic. Customer nodes could provide a specific time window that goods could deliver to them. Traffic situations could influence the current solution. With a smart program is possible to download the real situation of the traffic and reroute the answer that a vehicle has to visit in order to avoid roads with huge traffic or collisions. Another future work that this Thesis could produce, is to include the ability of drivers to pick up packages at the time of the travel and deliver them directly.

One more interesting future work that this application could be transferred is the Vehicle Routing Problem (VRP). The VRP is similar to TSP with the difference that more than one vehicle is available to deliver the goods. The nodes have to separate to each vehicle efficiently. That situation is more look-alike to the reality which a store with 2 to 3 vehicles has to serve customers among a district.

Bibliography

[1] K. Menger, "Das botenproblem," *Ergebnisse eines mathematischen kolloquiums*, vol. 2, pp. 11–12, 1932.

[2] D. E. Knuth, "Big omicron and big omega and big theta," *ACM Sigact News*, vol. 8, no. 2, pp. 18–24, 1976.

[3] S. Cook, "The p versus np problem," *The millennium prize problems*, pp. 87–104, 2006.

[4] J. K. Lenstra and A. R. Kan, "Some simple applications of the travelling salesman problem," *Journal of the Operational Research Society*, vol. 26, no. 4, pp. 717–733, 1975.

[5] M. Held and R. M. Karp, "A dynamic programming approach to sequencing problems," *Journal of the Society for Industrial and Applied mathematics*, vol. 10, no. 1, pp. 196–210, 1962.

[6] S. M. Goldfeld, R. E. Quandt, and H. F. Trotter, "Maximization by quadratic hill-climbing," *Econometrica: Journal of the Econometric Society*, pp. 541–551, 1966.

[7] J. L. Bentley and J. Saxe, "An analysis of two heuristics for the euclidean traveling salesman problem," in *Proc. 18th Annual Allerton Conference on Communication, Control, and Computing*, 1980, pp. 41–49.

[8] G. A. Croes, "A method for solving traveling-salesman problems," *Operations research*, vol. 6, no. 6, pp. 791–812, 1958.

[9] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[10] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *The journal of chemical physics*, vol. 21, no. 6, pp. 1087–1092, 1953.

[11] D. E. Goldberg and J. H. Holland, "Genetic algorithms and machine learning," 1988.

[12] D. E. Goldberg, R. Lingle *et al.*, "Alleles, loci, and the traveling salesman problem," in *Proceedings of an international conference on genetic algorithms and their applications*, vol. 154. Lawrence Erlbaum, Hillsdale, NJ, 1985, pp. 154–159.

[13] L. Davis, "Applying adaptive algorithms to epistatic domains." in *IJCAI*, vol. 85, 1985, pp. 162–164.

[14] I. Oliver, D. Smith, and J. R. Holland, "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., 1987.