UNIVERSITAT DE BARCELONA

Facultat de Matemàtiques i Informàtica

GRAU DE MATEMÀTIQUES

Treball final de grau

Different approaches to Travelling Salesman Problem

Autor: Roger Nogales Gine

Director: Sr. Eloi Sans Gispert

Realitzat a: Departament de Matemàtiques i Informàtica

Barcelona, January 25, 2022

Abstract

This thesis is a comparison of some to solve the Travelling Salesman Problem. The approach for the analysis by coding and compering the result. Starting with the most primitive Brute Force Algorithm and going through more creative and innovative ones, there is presented how do they work, a pseudo-code the mathematical theory hidden behind each of them. The objective is to compare and evaluate the pros and cons on different scenarios of the Salesman Travelling Problem.

2010 Mathematics Subject Classification. 11G05, 11G10, 14G10

Acknowledgements

During the last months it has been highly important the guidance and support of Sr. Eloi Sans, the director of this thesis. My most sincere words of thanks for his time and the confidence he has shown on me.

Contents

Abstract **iv**

1 Introduction **1**
1.1 Motivation 1
1.2 Problem Statement and variants 1
1.3 Objective 3
1.4 Report Layout 3

2 Foundations **5**
2.1 P, NP, NP-Complete and NP-Hard 5
2.2 Markov Chains 7

3 Algorithms **11**
3.1 Brute Force Algorithm 11
3.2 Nearest-Neighbour Algorithm 12
3.3 K-Opt Algorithms 14
3.4 Markov Chain Monte Carlo Algorithm 18
3.5 Simulated Annealing Algorithm 24
3.6 Ant Colony Optimization 28
3.7 Genetic Algorithms 32

4 Experimental results **37**

5 Conclusion **39**

Bibliography **41**

Chapter 1

Introduction

1.1 Motivation

The Travelling Salesman Problem is a well known problem often introduced in Graph Theory. As we will see in the next section, due to the simplicity of its statement but at the same time its difficulty to be solved, this problem have been a very popular target to test new and revolutionary ways to approach optimization problems. The idea of reviewing, programming and comparing some of the most popular algorithms was brought by Eloi and I immediately liked it. Not only because I would have a perfect opportunity to learn Python but also learn about Artificial Intelligence, an another interesting topic I was really interested about.

1.2 Problem Statement and variants

The Travelling Salesman Problem or TSP was first formulated in 1930 and is one of the most intensively studied problems in optimization. Although its statements are very simple, the problem have a lot of different variations. The classic Travelling Salesman Problem state that given $n \geq 1$ cities with all distances between city pairs known, the objective is to visit each city exactly once and return to the starting city in such a way that the total distance is minimum. "Distance", d_{ij}, is the cost associated with the travel from city i to city j. Since all cities are visited only once, the starting city can be any of the n cities without affecting the optimal solution. Thus, there are $(n-1)!$ possible solutions or tours for any n dimensional problem.
Using G. Dantzig, R. Fulkerson and S. Johnson [2] formulation, given a network of $n \geq 2$ nodes and the distance c_{ij} associated with each arc (i, j) joining two cities $0 < i, j \leq n$. A salesman who begins the trip in node 1, which is the depot, must

visit each node exactly once and return to the node 1. The problem is to find the shortest directed tour for visiting n nodes. The basic formulation for the TSP is as follows:

Let

$$\mathrm{x}_{ij} = \begin{cases} 1, & \text{the path goes from city } i \text{ to city } j \\ 0, & \text{otherwise} \end{cases}$$

Want to Minimize

$$\textstyle\sum_{(i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij} \tag{1}$$

subject to

$$\textstyle\sum_{i=1, i \neq j}^{n} x_{ij} = 1, j = 1, ..., n; \tag{2}$$

$$\textstyle\sum_{j=1, j \neq i}^{n} x_{ij} = 1, i = 1, ..., n; \tag{3}$$

$$\textstyle\sum_{i \in Q} \sum_{j \neq i, j \in Q} x_{ij} \leq |Q| - 1, \forall Q \nsubseteq \{1, ..., n\}, |Q| \geq 2 \tag{4}$$

The last constraint of the DFJ formulation ensures no proper subset Q can form a sub-tour, so the solution returned is a single tour and not the union of smaller tours.

As mentioned previously, there are a lot of different variations of the classic TSP. For example the asymmetric variation, where $d_{ij} \neq d_{ji}$. This is an important difference, since some of the algorithms that we will further see use reformulation operators in order to prevent itselves from being trapped at a local minimum, so symmetrical it is needed in those cases. Another interesting variation is the "M-Salesman Traveling Salesman Problem", deeply developed by Dennis Francis [8], where given m salesmen and n cities, the objective is to find m tours or "sorties" with minimum total travel, whereby all the cities are visited exactly once, each salesman visits at least one city, and all salesmen return to their common or "home" city. There are (n+m-2)! possible solutions to this problem. Another variations exists such the ones studied by Fredrick [4] where visiting the same city more than once is allowed, others where the salesman has to return to the home city every k cities visited, etcetera.

The Travelling Salesman Problem also can be applied to real-world situations, e.g., a real traveling salesman route, a school bus route, a job-shop machine schedule for a given set of repeated operations, a garbage truck route, and so on. The M-

Salesman problem, for instance, also has real applications, printing press scheduling for multi-edition periodicals, and bank messenger routing where there are m messengers and n branch banks are some examples of its applications.

1.3 Objective

The objective of this thesis is to compare 8 different algorithms that solve the Traveling Salesman Problem with their own particular method . Those will be the Force Brute Algorithm, the Nearest-Neighbour Algorithm, the family of K-Opt Algorithms (in particular the 2-Opt and the 3-Opt), and 4 Markov Chain based algorithms: a simple Monte Carlo Markov Chain Algorithm, the Simulated Annealing algorithm, the Ant Colony Optimization Algorithm and a Genetic Algorithm. To do so, they will be coded into Python language and evaluated multiple times under the same circumstances to secure impartiality.
Different initial scenarios will be presented to the algorithms in order to detect when ones are better than the others.

A part from comparing the results, another objective is to understand how each one of those algorithms work and why they lead to an optimal solution. The mathematics fundamentals behind them will be described at each algorithm section.

1.4 Report Layout

There will be four different comparisons depending on the number of cities $n \in \{10, 20, 30, 40, 50, 75, 100, 125, 150\}$, where all the algorithms will be run several times in order to get a representative mean of the computation time and the quality of the solutions.
The code is presented in a zip folder and it have a main plus 15 tabs. On each one, an algorithm is presented with its code and a brief explanation of its working procedure, its functions and its parameters.

Chapter 2

Foundations

2.1 P, NP, NP-Complete and NP-Hard

In theoretical computer science, the classification and complexity of common problem definitions have two major sets; P which is Polynomial time and NP which Non-deterministic Polynomial time. There are also NP-Hard and NP-Complete sets, which we use to express more sophisticated problems. In the case of rating from easy to hard, we might label these as easy, medium, hard, and finally hardest.

The first set of problems, p, are polynomial algorithms that we can solve in polynomial time, like logarithmic, linear or quadratic time. If an algorithm is polynomial, we can formally define its time complexity as:

$$T(n)=O(C*n^k)$$

where C > 0 and k > 0 are constant and n is input size of the problem. In general, for polynomial-time algorithms K is expected to be less than n. An example of this kind of problems, we have all the basic mathematical operations: addition, subtraction, division, multiplication. All the problems in this category have a complexity of O(n^k) for some k. Of course, we donât always have just one input n but, so long as each input is a polynomial, multiplying them will still be a polynomial.

The second set of problems, NP, cannot be solved in polynomial time. However, they can be verified (or certified) in polynomial 105 time.
Is expected these algorithms to have an exponential complexity, which is defined as:

$$T(n)=O(C_1 * k^{C_2 * n})$$

where $C_1 > 0$, $C_2 > 0$, k > 0, and C_1, C_2 and k are constant (again n is the input size). T(n) is a function of exponential time when at least $C_1 = 1$ and $C_2 = 1$. As a result, we got O(k^n). Graphs isomorphisms problems belongs to this category.
Formally, we also state that these problems must be decision problems (i.e. have a yes or no answer) though note that practically speaking, all function problems can be transformed into decision problems. This distinction helps us to nail down what we mean by verified. To speak precisely, then, an algorithm is NP if it canât be solved in polynomial time and the set of solutions to any decision problem can be verified in polynomial time.

The next set, NP-Complete, is very similar to the previous set. In fact, all NP-Complete problems are indeed NP but are among the hardest in the set. What makes them different from other NP problems is a useful distinction called completeness. For any NP problem that is complete, there exists a polynomial-time algorithm that can transform the problem into any other NP-complete problem. This transformation requirement is also called reduction.

The last set, NP-Hard algorithms, contains the hardest, most complex problems in computer science. They are not only hard to solve but are hard to verify as well. In fact, some of these problems are not even decidable. These algorithms have a property similar to ones in NP-Complete which is the fact that they can all be reduced to any problem in NP. Because of that, a NP-Hard problems are, at least, as hard as any other problem in NP. And it is possible to a problem to be both in NP and NP-Hard.

This is the category for the problem of this thesis, the Travelling Salesman Problem. As said, since it is possible for a problem to be both in NP and NP-Hard, this characteristic has led to a debate about whether or not Traveling Salesman is indeed NP-Complete or NP-Hard. Since NP and NP-Complete problems can be verified in polynomial time, proving that an algorithm cannot be verified in polynomial time is also sufficient for placing the algorithm in NP-Hard. And since proving that a tour is indeed the solution for the TSP is not easy or polynomial timed, define TSP as a NP-Hard problem.

Further theory and proof's con be found in Wenhong Tian [6].
Summarizing, P problems are quick to solve, NP problems are quick to verify but slow to solve, NP-Complete problems are also quick to verify, slow to solve and can be reduced to any other NP-Complete problem, and finally NP-Hard problems are slow to verify, slow to solve and can be reduced to any other NP problem.

P versus NP problem is one of seven Millennium Prize Problems in mathematics that were stated by the Clay Mathematics Institute in 2000. In fact, showing that this is truth would have devastating consequences for cryptography and also, for example, it would transform mathematics by allowing a computer to find a formal proof of any theorem that has a proof of reasonable length, since formal proofs can easily be recognized in polynomial time.

2.2 Markov Chains

Some of the algorithms and methods that will be further introduced and used to solve the TSP are based on Markov Chain. Markov Chain o Markov Process is a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event. Since this is a huge branch in the stochastic theory, we wont get in detail into it. The idea of this section is to understand the basics of Markov Chains and why, thanks to it, some of the algorithms studied in this thesis work. We will be using Matthew Richey [7] and Olivier Martin, Steve W. Otto and Edward W. Felten [3] for the following theory section. Further concepts and proofs can be found there.

Before we define Markov chains, we must define what a stochastic process is. A discrete-time stochastic process (DTSP) is a sequence of random variables $X_0, X_1, X_2, ...$ where X_t is the value at time t. An easy example would be the number of people that goes into a shop every day: X_0 is the number of people who came on the first day, X_1 on the second, and so on.

Now, given a finite state (configuration) space $S = 1, 2, ..., N$, a Markov Chain is a stochastic process defined by a sequence of random variables, $X_i \in S$, for i = 1, 2,... such that

$$\text{Prob}(X_{k+1} = x_{k+1} | X_1 = x_1, ..., X_k = x_k) = Prob(X_{k+1} = x_{k+1} | X_k = x_k).$$

In other words, the probability of being in a particular state at the (k + 1)st step only depends on the state at the kth step. We only consider Markov chains for which this dependence is independent of k (that is, time-homogeneous Markov chains). This gives an NxN transition matrix P = (p_{ij}) defined by

$$\text{p}_{ij} = Prob(X_{k+1} = j | X_k = i),$$

where for i = 1,2,...,N, $\sum_{j=1}^{N} p_{ij} = 1$

The (i, j)-entry of the Kth power of P gives the probability of transitioning from state i to state j in K steps.
Two desirable properties of a Markov chain are: It is irreducible: for all states i and j, there exists K such that $(P^K)_{i,j} \neq 0$.And it is aperiodic: for all states i and j, gcd{K : $(P^K)_{i,j} \geq 0\} = 1$.

The stationary distribution of a Markov Chain with n states (if one exists), is the n-dimensional row vector π (representing a probability distribution: entries which are non negative and sum to 1), such that

$$\pi P = \pi$$

Intuitively, it means that the belief distribution at the next time step is the same as the distribution at the current. This typically happens after a long time (called the mixing time) in the process, meaning after lots of transitions were taken.

An irreducible, aperiodic Markov chain must have a unique distribution π = $(\pi_1, \pi_2, ..., \pi_N)$.

We say that the Markov chain is stable on the distribution π, or that π is the stable distribution for the Markov chain. Markov chain methods depend on the observation: If π is the stable distribution for an irreducible, aperiodic Markov chain, then we can use the Markov chain to sample from π.

Since the set of all tours in the TSP is a finite set, the Markov chain can be characterized by a transition matrix T, where the matrix element T_{mn} is the probability to go from tour n to m. In practice, the selection of m requires random numbers. Given a starting tour, the application of T produces a sequence, or chain of tours. After some transients, usually the, so called memory, of the starting point decays, and tours appear with a limiting probability distribution P . P depends on the matrix T , and the goal is to find T's which lead to P(C) large for tours C of short length. This is called biased sampling, and it leads to sampling the tours of interest more efficiently.

If C′ is the optimal tour, is P (C′) $\neq$ 0? In the case of simulated annealing, the distribution P is known because T satisfies detailed balance. In particular, the probability of all tours is non-zero (the Markov chain is ergodic) and P(C) depends only on the length of C. For general Markov chains, (i.e., for general choice of the

matrix T), very little can be said of the probability distribution P. It is plausible nevertheless that within local-opt tours, our Markov chain is ergodic, and all our runs are consistent with this.

Chapter 3

Algorithms

3.1 Brute Force Algorithm

The first algorithm that we will see in this chapter and probably the most primitive and less optimized one is the Brute Force Algorithm. As its name describes, this algorithm lacks of any kind of optimal procedure in it and only does one thing: calculate a tour distance. In fact, it calculates all the possible tours and finally return the shortest one. The problem with that procedure is that is not often possible since for a large number of cities, the amount of possible different tours that the salesman can take scale very fast and ,consequently, the computational time of this algorithm.

Algorithm 1 Brute Force Algorithm

Number_of_tours ← factorial(n)
List_of_tours ← permutation(n,n)
Best_tour ← List_of_tours[0]
Best_length ← Length(List_of_tours[0])
for $i = 1, 2, .., n$ **do**
  if Length(List_of_tours[i]) < Best_length **then**
    Best_length ← Length(List_of_tours[i])
    Best_tour ← List_of_tours[i]
  end if
end for
return Best_tour, Best_length

We can see that, yet the algorithm will always guarantee the best solution for

Table 3.1: Brute Force Algorithm time

cities	total tours calculated	time (s)
5	120	0.01
8	40320	0.25
10	3628800	24.32
11	39916800	541.04

the problem, it can only really work with tours with few cities on it. As n get bigger, n! grows worse than exponentially. If we wanted to use brute-force to solve a 20-city problem, then we would need to evaluate 121645100408832000 different permutations, and that is still nothing compared to the 99! different permutations needed to search through for a 100-city problem.
In the code presented, there is bounded to 11 the number of cities that this algorithm can afford. Otherwise, it will return zeros.

3.2 Nearest-Neighbour Algorithm

This second algorithm use a very human way of thinking in order to approach the problem. We have seen that it is not feasible to compute for more than 11 cities in basic computers, so the idea of making decisions (and if possible, good ones) at some points of the tour comes to mind. Specifically, this algorithm replicate the behaviour of the salesman if he decides to move, on each step of the tour, to the next closest city.

This is a decision base algorithm but in fact, during all the journey the salesman have to make only 2 decisions and both before the journey even begins. The first one which he will have to follow the rest of the tour is: from one city move to the closest and not visited next city, this is, the way the salesman is going to approach the problem. The second decision is at which city is he going to start. This is a non-trivial decision because different starting cities, which the same strategy, leads to different results. Since the algorithm such as the salesman don't know a priory with city is the best one to start, it have to run the algorithm once for every starting city and storing the best solution during the process.

At this point I would like to introduce the difference between exact algorithm and greedy algorithms. The exact algorithms, such as the Brute Force Algorithm

introduced in the last chapter, are the ones that guarantee to find the exact optimal solution to the problem, no matter the time needed to do so. In the other hand, the heuristic algorithm do not guarantee that, but they are designed to run quickly. A subgroup of that class are the greedy algorithms, which are characterized by making the locally optimal choice at each stage. In other words, a greedy strategy does not produce an optimal solution, but can yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.
If we try to find an approximate solution to an NP-hard problem using heuristics, we need to compare the solutions using computational experiments. There is a number called domination number that compares the performance of heuristics. A heuristic with higher domination number is a better choice than a heuristic with a lower domination number. The following definition and theorem, proof can also be found there, come from Gregory Gutin, Anders Yeo, AlexeyZverovich [9]:

The domination number, $d(n)$, for the TSP of a heuristic A is an integer such as that for each instance I of the TSP on n vertices A produces a tour T that is not worse than at least $d(n)$ tours in I including T itself.

Observe that any exact algorithm for the TSP, such as the Brute Force Algorithm, has a domination number of $\frac{(n-1)!}{2}$.

In fact, as explained in Abraham Punnen, Franois Margot and Santosh Kabadi [6], for any heuristic A algorithm for the TSP, the domination number $d(n)$ exists and it is at least one, as the problem is always feasible. The definition extends directly to any minimization combinatorial problem with a feasible set F,and its proved that if $d(n)=|F|$then A is an exact algorithm producing an optimal solution.Thus the goal is to develop heuristic algorithms with domination number close to $|F|$.

Theorem 1. *Let $n \geq 4$. The domination number of the Nearest-Neighbor Algorithm for the Traveling Salesman Problem is, at most, 2^{n-3}.*

Proof. Proof can be found on Abraham Punnen, Franois Margot and Santosh Kabadi [6] □

The pseudo code for the Nearest-Neighbour Algorithm go as follows:

Algorithm 2 Nearest-Neighbour Algorithm

```
for k = 0 to n-1 do
    tour[0] ← k
    for j = 1 to n-1 do
        for i = 0 to n-1, i ∉ tour do
            d(j,i) ← distance between city j and city i
        end for
        best_i ← minimum(d(j,i))
        tour[j]=best_i
    end for
    for i = 0 to n-1, i ∉ tour do
        tour[n] = i
    end for
end for
return tour
```

Table 3.2: Nearest-Neighbour Algorithm

cities	total tours calculated	time (s)
10	1	0.01
100	1	1.18
250	1	41.13
500	1	664.45

3.3 K-Opt Algorithms

The tour improvement framework includes a set of operations that can be used to convert one tour to another. Formally, the tour improvement algorithm tries to reduce the cost of an initial sub-optimal tour repeatedly until no improvement can be made. Particularly, the k-opt algorithm is the most popular heuristic method for the TSP, where k means the number of edge exchanges/re-permutation within certain neighborhoods. However, existing k-opt approaches are not adaptive, and the number k is either fixed or sequentially probed following certain probing order. In practice, the most widely used tour improvement algorithms are probably the 2-opt and the 3-opt (k is fixed to be 2 and 3, respectively.) The 2-opt or 3-opt methods are advantageous in that, the implementation is relatively easy and the methods can be terminated at any point (anytime algorithms).

Figure 3.1: 2-Opt

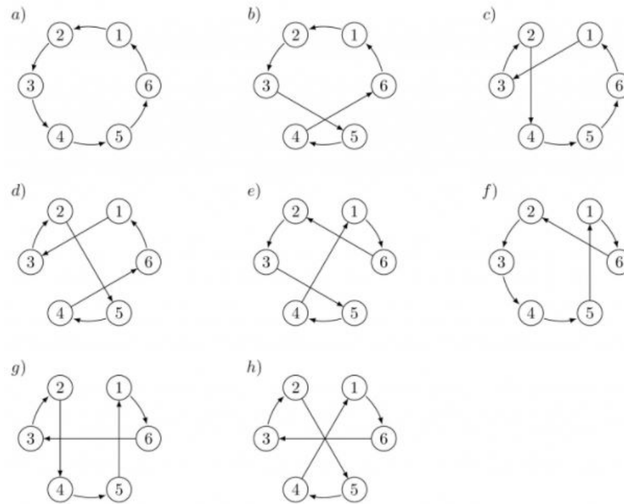

Figure 3.2: 3-Opt

Figure 3.3: Figures used from matejgazda.com

For example, the 2-opt algorithm is simply an iterative approach where in each iteration it strategically replaces two old edges with two new ones, so that all edges still form a tour (the total number of edges remains the same) and the operation decreases the tour cost. The 3-opt method is quite similar to the 2-opt, with the only difference that there are 3 edges needed to be appropriately replaced in each iteration and the number of 3-edge combinations is larger than that of the 2-opt. There are also 4-opt, 5- opt, etc. However, the complexity of manipulating the edge replacements increases exponentially as the number k grows, thus in practice it is very rare to see implementations of 4-opt and those opts with k greater than 4.

In each step, 2-Opt Algorithm deletes two edges thus creating 2 subtours and reconnects it with opposite edges (Figure 1) in case that the replacement reduces the length of the tour. One 2-Opt move has time complexity of $O(n^2)$. This is because in one 2-Opt move, in the worst case, we need to check for one broke edge to

(nâ2) other edges that could be broken to improve the tour - thus the O(n). Since we need to check it for all the edges, therefore we get the equation $nO(n)$=O(n^2). 3-opt Algorithm works in the similar fashion. Instead of breaking the tour by removing 2 edges, it removes three edges, therefore breaking the tour into 3 separate sub-tours that can be reconnected in 8 possible ways (including the original tour) as shown on Figure 2. The 3-Opt Algorithm have a time complexity of O(n^3).

In terms of domination number, we have seen that is a good way to compare heuristic algorithms so as Abraham Punnen, Franois Margot and Santosh Kabadi [6] proved in their paper, we have those two following theorems involving 2-Opt and 3-Opt (proofs can be found there):

Theorem 2. *The domination number of 2-Opt is at least* $\frac{(n-2)!}{2}$ *when n is even and* $(n-2)!$ *when n is odd.*

Proof. Proof can be found on Abraham Punnen, Franois Margot and Santosh Kabadi [6] □

Theorem 3. *The domination number of 3-Opt is at least* $\frac{(n-2)!}{2}$.

Proof. Proof can be found on Abraham Punnen, Franois Margot and Santosh Kabadi [6] □

The pseudo code for both of this algorithms go as follows:

Algorithm 3 2-Opt Algorithm

```
input a tour to improve tour_to_improve
min_length ← length(tour_to_improve)
best_tour ← tour_to_improve
for i=1,2,...n-1 do
    for j=i+2,...,n do
        if distance(i,i+1) + distance(j,j+1) > distance(i,j) + distance(i+1,j+1) of the
tour_to_improve then
            best_tour[i+1,j+1] ← reversed(tour_to_improve[i+1,j+1])
            tour_to_improve ← best_tour
        end if
    end for
end for
return best_tour
```

And the table with the results:

Algorithm 4 3-Opt Algorithm

```
input a tour to improve tour_to_improve
min_length, current_length ← length(tour_to_improve)
best_tour, current_tour ← tour_to_improve
while improvement == True do
    for i = 1,2,....,n-5 do
        for j = i+2,...,n-3 do
            for k = j+2,...,n do
                swap all possible 3 cities of the tour as in Figure 3.2 of current_tour
                if length(swapped_tour) < min_length then
                    min_length ← length(swapped_tour)
                    best_tour ← swapped_tour
                end if
            end for
        end for
    end for
    if best_tour == current_tour then
        Improvement = False
    else
        current_tour ← best_tour

        return best_tour
```

Table 3.3: 2,3-Opts Results obtained from a random tour

cities	2-Opt Improvement	3-Opt Improvement	time difference
10	20,95%	41,75%	0,08
25	9,40%	69,02%	3,26
50	6,15%	78,92%	117,73
75	3,98%	81,25%	987,51

Table 3.4: 3-Opts Results from Nearest-Neighbour solution

cities	3-Opt Improvement	time difference
20	9,56%	0,31
30	8,49%	2,41
40	11,59%	9,09
50	13,66%	25,81
65	15,23%	69,68

3.4 Markov Chain Monte Carlo Algorithm

The Monte Carlo Markov Chain (MCMC) is a method that makes it possible to efficiently sample from a large combinatorial set according to a desired probability distribution and can be used for function optimization, as for example in our case the length of a tour. In fact, this method facilitates the development of simple, efficient, and general solutions to whole classes of decision problems.

Decision and optimization problems involving graphs arise in many areas of artificial intelligence, including probabilistic networks, robot navigation, and network design. Many such problems, as we have seen with TSP, are NP-complete; this has necessitated the development of approximation methods, most of which are very complex and highly problem specific.

Suppose that Ω is a large, finite combinatorial set and that $f\colon \Omega \to \mathbb{R}$ is a function defined on Ω. We want to find a solution x $\in \Omega$ such that $f(x)$ is minimal or maximal, depending on the problem we are optimizing. The MCMC method requires that we define an undirected, connected graph Λ on all possible solutions to the problem and a set of one or more moves, i.e., relatively simple operations that transform one element of Ω to another. Each vertex of Λ represents a member

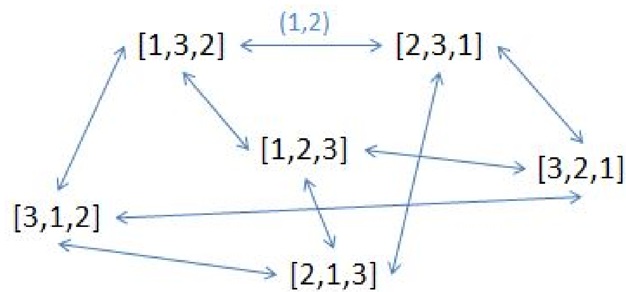

Figure 3.4: 3-city tours and 2 element permutation operation

of Ω, i.e., one possible solution, and each edge represents a move. Λ is called a neighborhood structure because the neighbor vertices of any vertex x correspond to the set of solutions reachable via a single move from the solution represented by x.
And here is the interesting part, because by representing each vertex of Λ with a set of features and each edge as a change to one feature, we can simulate a Markov chain through the space of possible solutions Ω). A random walk on the vertices of Ω then represents a sequence of solutions in which each solution differs from the previous solution by at most one feature (solutions can repeat). This sequence is a Markov chain because at each step in the sequence, the choice of the next solution depends only on the current solution and not on any previous ones.

Summarizing, the MCMC approach to optimization and decision problems involves simulating the Markov chain for some number of steps T, beginning with an arbitrary initial solution, and then either outputting the best solution seen so far or outputting whether a solution has been found. Due to the Markov property, seen in Markov Chain Theory chapter, any algorithm based on generating a randomized sequence of solutions does not need to maintain a data structure for the entire graph Λ luckily since most of the cases, this would be impossible given the large size of Ω.

Introducing some bias and probabilities into the random walk produces better solutions. This usually means always accepting transitions of the Markov chain to states with higher f value but occasionally accepting transitions to states with lower f value. In particular, suppose degree(x) denotes the degree of vertex x in Λ, and suppose D(Λ) denotes an upper bound on the maximum degree. Then, as described in Timothy Huang, Yuriy Nevmyvak [10], the transition from a current vertex x to the next vertex y is specified as follows:

Algorithm 5 Markov Chain Monte Carlo Algorithm

I. With probability 0.5 let y=x, otherwise II.
II. Select y according to the distribution:

$$Pr(y) = \begin{cases} & \text{if y is a neighbour of x} \\ & \text{if y=x} \\ 0, & \text{otherwise} \end{cases}$$

III.With probability min{1,$\alpha^{f(y)-f(x)}$}, transition to y; otherwise y←x.
return y =0

Note that in the algorithm, $\alpha \geq 1$; this ensures that transitions to neighbors with lower f values are always accepted and that transitions to neighbors with higher f value are rejected with probability (1 - $\alpha^{f(y)-f(x)}$) < 1. The last inequality holds since f(y) - f(x) < 0 when the state denoted by y is a less desirable state; thus, 0 < $\alpha^{f(y)-f(x)}$ < 1 and (1 - $\alpha^{f(y)-f(x)}$) < 1.

To show that the resulting Markov chain converges to a stationary distribution, we note the following properties: First, since Λ is connected by definition, the Markov chain is irreducible, i.e., any state can eventually be reached from any other state. Second, since all self-loop probabilities are required to be non-zero, the chain is aperiodic and hence ergodic, i.e., there are no fixed cycles through which states will alternate. The probability distribution of this chain can then be defined as

$$\pi_\alpha = \frac{\alpha^{f(x)}}{Z(\alpha)}, \text{ for all } x \in \Omega$$

where Z(α) is a normalizing constant that ensures π_α is a probability distribution. Finally, we note that the chain is also reversible, i.e., it satisfies the detailed balance condition:

$$\pi_\alpha(x)P(x,y)=\pi_\alpha(y)P(y,x), \text{ for all } x,y \in \Omega.$$

These conditions guarantee that the Markov chain converges to the stationary distribution π_α. Detailed definitions of each requirement and proofs of convergence can be found in Richard L. Smith and Luke Tierney [11]. A Markov chain of this form is known as a Metropolis process.

Notice also that the parameter α influences the rate at which the algorithm finds better solutions and the ability of the algorithm to move beyond local maximum in the solution space. Lower values of α smooth out the distribution π_α and help keep the chain from getting stuck in local maximum; higher values of α make

the distribution π_α more peaked around optimal solutions and help find better solutions more quickly. At the two extremes, we have an unbiased random walk on the graph Λ when $\alpha = 1$ and a greedy search when $\alpha = \inf$. Hence, some intermediate value for α is usually preferred. Jerrum and Sinclair provide a detailed analysis of the Metropolis algorithm at α a in Jerrum and Sinclair[13]. Varying α while the process is simulated results in a simulated annealing algorithm, which we will see further on.

For the code we present for this method, we used the same idea as Timothy Huang, Yuriy Nevmyvak [11], but since the algorithm is attached to a number of iterations, is not interesting to us the idea of not moving around on each iteration, this is, letting $y \leftarrow x$ at any time. We will focus on moving around the vertices of Λ only taking care of the probability to move to the next one, and this probability will be the same for all possible neighbours of the current state. In other words: on each iteration we will move towards a neighbour state (i.e solution) with the same probability. Then, if this next state result to be better, we will always stay there and move to the next iteration. If it is not the case, there would be still a little chance to remain there. If still is not the case, then finally we will keep at the same beginning state.

The operation we use to move towards a neighbour is the permutation of a subtour belonging of length desired. As shown in Figure X, all tours that can be obtained from a permutation of a subtour of length desired form all the possible next states to move towards. With that in mind, our proposed code need to have defined beforehand the number of iterations, the permutation number (p) , i.e the length of the subtour that we are going to swap and the alpha value already defined. The pseudo code goes as following:

Algorithm 6 Simple MCMC Algorithm

```
for i = 1,2,..,n do
    Current_solution ← random or defined starting state
    Next_solution ← Current_solution
    Start_subtour ← random(0,n)
    Sub-tour_to_swap ← Current_solution[Start_subtour][Start_subtour + p]
    Sub-tour_swapped ← Sub-tour_to_swap randomly swapped
    Next_solution[Start_subtour][Start_subtour + p] ← Sub-tour_swapped
    if length_next < length_current then
        Current_solution ← Next_Solution
    else
        With probability α^(length(next)−length(curr)), Current_solution ← Next_Solution
    end if
end for
return Current_solution
```

We have experimented with the values of α and the permutation number value p (i.e the length of the sub-tour that is swapped).
First and in order the find the optimal, or at least a good estimation, value for α, we fixed the permutation number p to 5. This is, to move from one vertex to another of the meta-graph, we have to perform a 5 cities sub-tour permutation. Also, we fixed n = 150 the number of cities and a number of iteration of 2000. Then we run the experiment 50 times for each value of α, each time starting in the same initial tour. The following tables shows the result:

Table 3.5: Finding optimal α for MCMC

α	**iterations**	**average tour length**	**average time**
1	2000	78604,76	0,32
3	2000	44579,23	0,35
5	2000	44888,79	0,34
7	2000	44688,35	0,31
9	2000	44795,68	0,34
11	2000	44702,61	0,32
13	2000	44650,66	0,32
15	2000	44982,06	0,31
17	2000	44883,29	0,31
19	2000	45084,38	0,33
21	2000	44566,84	0,37
23	2000	44721,88	0,35
25	2000	44709,66	0,37
27	2000	45129,29	0,31
29	2000	44604,14	0,31
31	2000	44687,69	0,39
33	2000	44733,70	0,39
35	2000	44401,07	0,34
37	2000	45017,48	0,34
39	2000	44399,22	0,35
41	2000	44553,64	0,35
43	2000	44887,65	0,34
45	2000	44880,52	0,34
47	2000	45008,85	0,35
49	2000	44942,46	0,36

We can see that, both in terms of average length and time, almost all α gave similar values. Since the compute time is less than a second, it wound be relevant for the decision. And sorting by increasing average distance we get that the better values for α are:39, 35, 41, 21, 3, 29... (depending of the number of cities n, we will pick one or another in that order).
Now, fixing α = 39, the same 2000 iterations and n = 150, we will try to find out the best value for the parameter p:

Table 3.6: Finding optimal p for MCMC

p	**iterations**	**average tour length**	**average time**
2	2000	64409,63	0,30
3	2000	52254,95	0,33
4	2000	45611,91	0,40
5	2000	44381,81	0,33
6	2000	43456,99	0,33
7	2000	43487,71	0,33
8	2000	45254,26	0,34
9	2000	45931,84	0,44
10	2000	46133,48	0,36
11	2000	48267,33	0,35
12	2000	49000,90	0,36
13	2000	49683,98	0,36
14	2000	50622,34	0,37
15	2000	51440,70	0,49
16	2000	51821,75	0,53
17	2000	52596,77	0,47
18	2000	53491,98	0,39
19	2000	53471,28	0,38
20	2000	54228,75	0,38

As before, the average time is so low that will be irrelevant for the choosing of p. Sorting the average tour length values in a increasing order, we find that the p that produce shortest tours are: 6, 7, 5, 8, 4, 9... (curiously, 2 is the worst value by far).

With that, we will be using the MCMC algorithm with values $\alpha = 39$ and $p = 6$ from now on.

3.5 Simulated Annealing Algorithm

Simulated annealing is so named because of its analogy to the process of physical annealing with solids, in which a crystalline solid is heated and then allowed to cool very slowly until it achieves its most regular possible crystal lattice config-

uration (i.e., its minimum lattice energy state), and thus is free of crystal defects. If the cooling schedule is sufficiently slow, the final configuration results in a solid with such superior structural integrity. Simulated annealing establishes the connection between this type of thermodynamic behavior and the search for global minimum for a discrete optimization problem. Furthermore, it provides an algorithmic means for exploiting such a connection.

At each iteration of a simulated annealing algorithm applied to a discrete optimization problem, the objective function generates values for two solutions (the current solution and a newly selected solution) are compared. Improving solutions are always accepted, while a fraction of non-improving (inferior) solutions are accepted in the hope of escaping local optima in search of global optima. Unlike the MCMC algorithm seen in the last chapter where on each step we had a probability of moving to a worst state depending on the actually difference between both states, here the probability of accepting non-improving solutions depends on a temperature parameter, which is typically non-increasing with each iteration of the algorithm. The key algorithmic feature of simulated annealing is that it provides a means to escape local optima by allowing hill-climbing moves (i.e., moves which worsen the objective function value). As the temperature parameter is decreased to zero, hillclimbing moves occur less frequently, and the solution distribution associated with the inhomogeneous Markov chain that models the behavior of the algorithm converges to a form in which all the probability is concentrated on the set of globally optimal solutions (provided that the algorithm is convergent; otherwise the algorithm will converge to a local optimum, which may or not be globally optimal).

The proof of this convergence approach for the simulated annealing algorithm is based on inhomogeneous Markov chain theory and can be found on S. Anily and A. Federgruen [5].

The acceptance probability of moving from state i to state j is defined by (8):

$$\mathrm{A}_{ij} = \begin{cases} exp(\frac{length(y)-length(x)}{T}), & \text{if length(y) > length(x)} \\ 1, & \text{otherwise} \end{cases}$$

Our proposed code for this algorithm needs just 4 initial values to run: an initial temperature which needs to be big so it can start decreasing from this point, a minimum temperature value which will be a threshold to stop the algorithm, a maximum number of iterations allowed and a value of alpha similar to the MCMC we have already seen. Moving from one state to another is defined as applying

2-opt algorithm to a subtour of the current solution of random length (< n). Then an acceptance function will decide if move or not to that next state depending on the probability defined above.
The pseudo code is as follows:

Algorithm 7 Simulated Annealing Algorithm

```
current_solution ← random_tour
minimum_length ← length(current_solution)

while temp > min_temp and iter < iter_max do
    next_solution ← 2-Opt applied to a subtour of current_solution
    if length(next_solution) < length(current_solution) then
        current_solution ← next_solution
    else
        current_solution ← next_solution with probability exp((length(y)−length(x))/T)
end if

Decrease temperature

return current_solution
```

Anticipating the results, this algorithm have proven to be very efficient, providing the best solution on almost every experiment and also very fast in terms of time.

In order to choose the best way to decrease the temperature value on each iteration, we have tried different approaches to it. The first one simply will multiply by a constant $\beta \in (0, 1)$ and we will try three different values for it. The second and the third way of decreasing will be following a lineal and a logarithmic function. On each number of cities it has been tried 25 times each way, and having 1000 and 0 as initial and lowest temperature value.

Table 3.7: Simulated Annealing Algorithm

cities	alpha	improvement	iterations	time
25	0,79	44,58%	109	0,01
25	0,89	52,91%	219	0,01
25	0,99	65,32%	2522	0,09
25	lineal	52,91%	2001	0,07
25	logarithmic	64,64%	3000	0,11
-	-	-	-	-
50	0,79	30,75%	109	0,01
50	0,89	42,62%	219	0,01
50	0,99	69,99%	2522	0,15
50	lineal	56,85%	2001	0,12
50	logarithmic	75,70%	3000	0,19
-	-	-	-	-
100	0,79	23,79%	109	0,01
100	0,89	35,38%	219	0,03
100	0,99	72,87%	2522	0,32
100	lineal	52,35%	2001	0,24
100	logarithmic	74,22%	3000	0,36
-	-	-	-	-
150	0,79	15,83%	109	0,02
150	0,89	26,38%	219	0,04
150	0,99	68,17%	2522	0,5
150	lineal	45,43%	2001	0,36
150	logarithmic	70,53%	3000	0,54

We can see that, in terms of time, all the runs take less than a second to compute so it almost won't be important when choosing the best fit for alpha. Also we notice that the improvement goes hand by hand with the number of iterations, so for this algorithm from now on we will work with the logarithmic decrease function for the temperature which we can see give a better improvement of the length of the tour.

3.6 Ant Colony Optimization

Ant colony optimization (ACO) is a discrete combinatorial optimization algorithm based on the foraging behavior of ants. Over a period of time ants are able to determine the shortest path from their home to a food source. This shortest-path-finding process of the colony can be viewed as a form of swarm intelligence. This process is achieved by the colonyâs accumulation of information about the surrounding area, which is communicated to the individual ants in the form of trails of pheromone, a chemical substance laid by the ants themselves. Isolated ants essentially wander randomly until they come across a previously laid pheromone path, which they will, by instinct, be more inclined to follow as opposed to continuing to wander randomly. As an ant traverses the path, it too lays pheromone, thus reinforcing the existing pheromone strength of the current path and, hence attracting further ants to follow it. Gradually over time, the shorter paths between destinations will increase in pheromone intensity due to lower traverse times, and so the colony gradually determines the optimum route between the destinations. This phenomenon is best illustrated in the figure below:

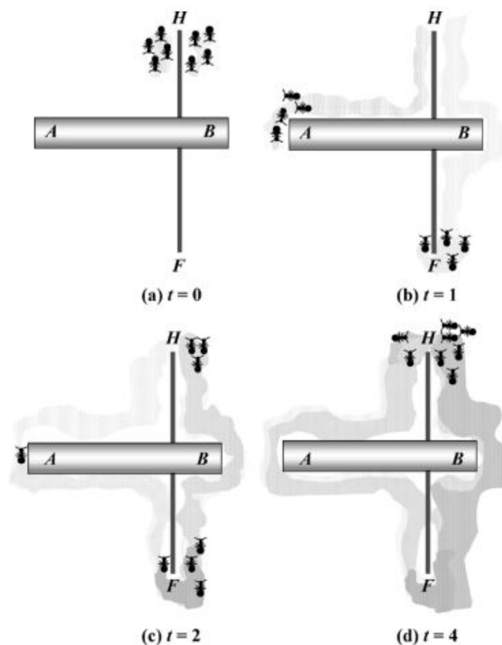

Figure 3.5: Example of evolution of pheromone trails, Figure from [x]

In Figure 3.5, a system comprising a colony home (H), a food source (F), and an obstacle (A,B) is depicted. The obstacle is placed such that there are two paths of

unequal lengths, and it is assumed that path HAF takes two time steps to traverse and path HBF takes a single time step to traverse. At time t = 0, [Fig. 3.5(a)], eight ants are placed into the system with the objective of getting food from F back to H. As their initial selection of the path they take is random, it is assumed that four ants select each of the two alternate paths.
At t = 1, [Fig. 3.5(b)], the ants that traversed HBF have acquired food and begin to journey back to H. As there is existing pheromone on FBH the ants have a higher probability of utilizing this path, consequently three ants select FBH and one ant selects FAH. The ants that are traversing HAF are only half way along this path.
At t = 2, [Fig. 3.5(c)], the three ants that traversed FBH are home again while the ant that embarked on FAH is only half way along this path. The four ants that were traversing HAF have made it to and embark on their journey back to via either FAH or FBH. Path FBH has a greater amount of pheromone on it (note that the pheromone intensity is represented by the darkness of the path) as it has been traversed seven times, whereas FAH has only been traversed five times. Consequently, by probability, three ants select path FBH and one ant selects FAH.
At t=4, [Fig. 3.5(d)], all ants have returned to H (note that the ant that embarked on FBH at t = 2 arrived at H at t = 3). From Fig. 3.5(d), it is seen that the shorter path HBF has a greater amount of pheromone on it as it has been traversed ten times in total, while the longer path HAF has been traversed only six times. Future ants entering the system will have a higher probability of selecting path HBF. In the pattern illustrated here the operation of swarm intelligence to determine the shortest path is seen.
In addition to the positive feedback strategy illustrated in the example, the pheromone trails also decay with time. This means that paths that are not regularly given additional pheromone will eventually decay to zero intensity. This decaying quality of pheromone also aids in the ability of the ant colony to find the shorter paths. The longer paths, which receive less pheromone, will decay more rapidly enabling shorter paths to have a higher probability of being selected.

Thus, applying Ant Colony Optimization to our TSP problem goes as follows: At each time t, an ant in city i has to choose the next city j it goes to, out of those cities that it has not already visited. The probability of picking a certain city j is biased by the distance between i and j and the amount of pheromone on the edge between these two cities. Let Π_{ij} denote the amount of pheromone (also called trail) on the edge between cities i and j and let η_{ij} be the visibility of j from i defined as:

$$\eta_{ij} = \frac{1}{distance(i,j)}$$

Then the bigger the product of Π_{ij} and η_{ij} , the more likely it is that j will be

chosen as the next city. The trail and visibility are now weighted by parameters α and β and we arrive at the following formula (where $p_{i,j}$ (t) is the probability of choosing city j from city i at time t and allowed_k is the set of cites that are still allowed (unvisited) for ant k:

$$p_{ij}(t) = \begin{cases} \frac{\Pi_{ij}^{\alpha}\eta_{ij}^{\beta}}{\sum_{k \in allowed_k} \Pi_{ik}^{\alpha}\eta_{ik}^{\beta}}, & \text{if j} \in \text{allowed_k} \\ 0, & \text{otherwise} \end{cases}$$

This probabilistic choice however does not guarantee that the optimal solution will be found. In some cases, a slightly worse than optimal solution may be found at the very beginning and some sub-optimal arcs may be reinforced by deposited pheromones. This reinforcement can lead to stagnation behaviour resulting in the algorithm never finding the best solution.

After all of the ants have completed their tours, the trail levels on all of the arcs need to be updated. The evaporation factor Ï ensures that pheromone is not accumulated infinitely and denotes the proportion of âoldâ pheromone that is carried over to the next iteration of the algorithm. Then for each edge the pheromones deposited by each ant that used this edge are added up, resulting in the following pheromone level-update equation:

$$\Pi_{ij}(new) = p * \Pi_{ij}(old) + \sum_{k=1}^{m} \triangle\Pi_{ij}^{k}$$

where m is the total number of ants on the system, and $\triangle\Pi_{ij}^{k}$ is the amount of pheromone deposited by ant k onto the edge from i to j at the current iteration. This amount is based on a constant Q, which is the total amount of pheremome that can be distributed on each iteration, divided by the length of the tour found by ant k denoted L^k.

$$\triangle\Pi_{ij}(t) = \begin{cases} \frac{1}{L_{length_of_best_tour}(t)}, & \text{if arc (i,j)} \in \text{best_tour} \\ 0, & \text{otherwise} \end{cases}$$

And for the first iteration, since the ants have no pheromone to follow, we will be using:

$$\Pi_0 = (n * L_{nn})^{-1},$$

where n is the number of cities and L_{nn} is the tour length produced by the nearest neighbor heuristic.

For the rest of the parameters, we will be using Dorian Gaertner and Keith

Clark [14], where suggest different kind of optimal parameters based on experimental tests. In our case, we will be picking α, the parameter that controls the influence of distance, equal to 1; β, the parameter that controls the influence of pheromone, equal to 5; the evaporation rate parameter p equal to 0.5; a number of maximum iteration set as 40 and the constant Q equal to the total number of pheromones equal to 100.

The pseudocode for the Ant Colony Algorithm goes as follow:

Algorithm 8 Ant Colony Optimization

```
input algorithm parameters
initialize pheromone matrix
for iteration < iter_max do
    for all ants k=1,...,m do
        Create tour_k based on the actual pheromone matrix
    end for
    Best_tour ← evaluate and find the best of all m tours
    Upgrade pheromone matrix according to the algorithm
end for
return Best_tour
=0
```

And the following table shows the results of this algorithm, where for every n cities it have been run 20 times and computed its mean:

Table 3.8: Ant Colony Optimization

cities	iterations	improvement from a random tour	time (s)
25	20	65,15%	0,72
25	30	68,55%	1,14
25	40	68,97%	1,59
-	-	-	-
50	20	74,28%	6,30
50	30	76,59%	9,38
50	40	73,52%	13,41
-	-	-	-
100	20	82,70%	58,54
100	30	82,18%	87,91
100	40	82,35%	115,54
-	-	-	-
150	20	85,21%	224,80
150	30	85,80%	330,97
150	40	85,81%	446,06

Â¡We can see that doing 30 iterations on this method give almost the same result as doing 40, but with nearly 26% less time. For this reason, we will be using the Ant Colony Optimization with 30 iterations because it have a better improvement/time ratio.

3.7 Genetic Algorithms

In nature, there exist many processes which seek a stable state. These processes can be seen as natural optimization processes. Over the last 30 years several attempts have been made to develop global optimization algorithms which simulate these natural optimization processes. Some of these attempts have resulted in the already seen optimization methods: Simulated Annealing, based on natural annealing processes; Ant Colony Optimization, based on the ants behaviour and the one we will see in this chapter, Evolutionary Computation, based on biological evolution process.
Genetic Algorithms were introduced in 1975 and belong to one of the Evolutionary Computation's branches. In these algorithms the search space of a problem is represented as a collection of individuals and the purpose of using a genetic

algorithm is to find the individual from the search space with the best âgenetic materialâ. The quality of an individual is measured with an evaluation function. The part of the search space to be examined is called the population. Roughly, a genetic algorithm works as follows:

Algorithm 9 Abstract Genetic Algorithm

```
Make initial population at random
while Not stop do
    Select parents from the population
    Produce children from the selected parents
    Mutate the individuals
    Extend the population adding the children to it
    Reduce the population
end while
return the best individual found
```

First, the initial population is chosen, and the quality of this population is determined. Next, in every iteration parents are selected from the population. These parents produce children, which are added to the population. For all newly created individuals of the resulting population a probability near to zero exists that they will mutate, i.e. that they will change their hereditary distinctions. After that, some individuals are removed from the population according to a selection criterion in order to reduce the population to its initial size. One iteration of the algorithm is referred to as a generation.
The operators which define the child production process and the mutation process are called the crossover operator and the mutation operator respectively. Mutation and crossover play different roles in the genetic algorithm. Mutation is needed to explore new states and helps the algorithm to avoid local optima. Crossover should increase the average quality of the population. By choosing adequate crossover and mutation operators, the probability that the genetic algorithm results in a near-optimal solution in a reasonable number of iterations is increased.
As you can imagine, there are a lot of different crossover and mutation operators and it is very difficult to determine which ones are better than the others. For this algorithm, we will be using Omar M.Sallabi and Younis El-Haddad [z] paper's idea and using theirs operators.
The first approach to the genetic algorithm turned not to be very accurate (it can be found in the code folder). This method kind of looks very intuitive and easy to implement by only looking at its abstract pseudo code, but in reality you have to be very precise and make sure that on each generation a large range of search

space is discovered, no loops of population are produced and secure some random noise is added to the actual population.
Returning to Omar M.Sallabi and Younis El-Haddad [15], it introduces a new crossover technique for genetic algorithms in order to perform the Improved Genetic Algorithm (IGA).
Their idea for this method is to start by a population of only 2 individuals. Then, a Swapped Inverted Crossover (SIC) is applied to them in order to produce 12 children. Afterwards, 10 copies for each one of those 12 new children are made, where on each one of the copies a different mutation method is applied. All this produce the population which will be analysed, this is to say, a total of 134 individuals. A fitness function will calculate all their lengths and the best two individuals will be selected. Finally, a partial local optimal mutation operation for the next generation is applied and also a population reformulates operation. This techniques ensures that new cities will be at the middle part of the cycle, ready for a possible improvement and not stuck at the sides of the tour.

So going into detail on each part of this method, as said It starts with 2 individuals called parents and a Swapped Inverted Crossover is applied to them. The main idea of the SIC is to backtrack different ways to search for better tours and can be applied with a one or two point crossover, or both, as is done here. With the Two Point SIC, two random cut points (p1 and p2) are defined in order to cut the parents tour in three parts: a head, contain (1,2,...,p1-1), the middle containing (p1,...,p2), and the tail containing (p2+1,...,n). Then the head and tail of each parent are flipped, and then the head of the first parent is swapped with the tail of the other parents, and vice versa. Applying this process will create two children: O1 and O2, as seen in the Figure below. For the other 10 children, O3 to O12, the One point SIC will be applied. With almost the same idea, cutting one part of one parent and replacing it with another part of the second parent is how the rest of the children are born.

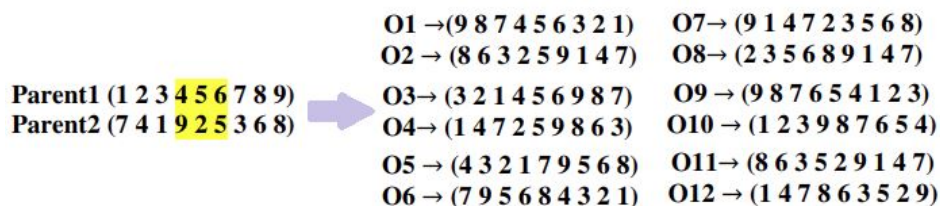

Figure 3.6: Children's birth from 2 parents, details can see in [z]

Then, a rearrangement operation is done to the newborn children, were it finds

the longest distance between two cities of the tour and swap them for three different cities located on three different positions on the tour (beginning, middle, and the end). The best position, plus the original position will be accepted. Since this operation works in random matter, it may not achieve any improvement after several iterations, but it may take big jump and improve the result.
After that, the multi mutation operator is applied to the children. It produces 10 copies of each one with different city's swaps on it, so the mutations still got the genetic of the parents but with different permutations.
Now we already have all the population of the new generation created and a fitness function will determine all their length and choose the best 2 tours, that will be used as the parents for the next generation. But before that, 2 more operations have to be applied to both of them: the partial local optimal mutation operation (PLOO), which will apply the 2-opt algorithm to a subtour of one of the parents in order to produce a local minimal on it; and the population reformulates operation, which is no other than just a reformulate of the city's indexes of the tour.

The pseudo code for the Improved Genetic Algorithm is the following:

Algorithm 10 Improved Genetic Algorithm

```
input number of parents, number of maximum iterations, number of children
per generation, mutation number
initialize population matrix
current_parents ← random_tours
for iteration t=1,2,...,iter_max do
    new_children ← SIC(current_parents)
    rearranged_children ← rearrange_oper(new_children)
    mutated_children ← multi_mutation_oper(rearranged_children)
    population_matrix ← current_parents,rearranged_children,mutated_children
    new_parents ← select_parents(population_matrix)
    optimal_new_parents ← PLOO(new_parents)
    current_parents ← reformulate_oper(optimal_new_parents)
end for
return current_parents
```

In order the find an optimal number of iterations ([z] suggest 2000), we will try different values and compare the efficiency for them:

Table 3.9: Genetic Algorithm iteration comparison

cities	iter	improvement vs random	improvement vs NN	time diff vs NN
25	500	68,11%	8,90%	2,63
25	1000	67,19%	7,43%	5,17
25	2000	68,87%	11,14%	10,63
25	3000	67,29%	7,64%	16,14
-	-	-	-	-
50	500	74,12%	11,22%	5,66
50	1000	75,23%	12,10%	12,67
50	2000	77,06%	8,86%	20,54
50	3000	77,45%	7,52%	30,69
-	-	-	-	-
100	500	84,21%	10,33%	10,07
100	1000	84,91%	7,09%	22,03
100	2000	84,60%	7,92%	47,79
100	3000	85,26%	8,45%	72,32
-	-	-	-	-
150	500	86,61%	8,73%	12,53
150	1000	86,40%	5,86%	32,58
150	2000	86,89%	7,97%	79,36
150	3000	86,59%	8,29%	117,18

Chapter 4

Experimental results

In this chapter, all the algorithms seen will be compared to each order in different scenarios in order to know which one performs better. The algorithms that use parameters will be using the bests and optimal ones found on their own section.

Table 4.1: Comparison of the algorithms in terms of quality

n	BF	NN	3-Opt	MCMC	SA	AC	GA
10	100%	93,77%	96,94%	98,56%	99,29%	98,87%	100%
20	0%	86,80%	98,69%	69,31%	96,49%	95,28%	99,23%
30	0%	90,48%	98,69%	38,04%	93,67%	92,84%	99,14%
40	0%	84,59%	99,08%	5,49%	88,99%	89,54%	98,80%
50	0%	85,79%	99,94%	2,34%	82,67%	88,86%	97,13%
75	0%	88,16%	0%	1,98%	67,31%	85,60%	100%
100	0%	87,28%	0%	1,67%	27,75%	88,65%	100%
125	0%	90,36%	0%	1,34%	5,26%	90,03%	100%
150	0%	92,99%	0%	1,23%	4,97%	94,75%	99,82%

Table 4.2: Comparison of the algorithms in terms of computation speed, in seconds

n	BF	NN	3-Opt	MCMC	SA	AC	GA
10	25,03	0,01	0,01	0,06	0,06	0,08	5,31
20	0,00	0,01	0,91	0,09	0,12	0,77	10,57
30	0,00	0,01	7,74	0,09	0,13	2,19	12,76
40	0,00	0,04	36,69	0,11	0,16	5,11	16,76
50	0,00	0,09	121,60	0,14	0,21	10,27	21,15
75	0,00	1,46	0,00	0,59	0,89	46,87	133,23
100	0,00	5,01	0,00	1,29	1,5	128,72	172,97
125	0,00	11,05	0,00	1,03	1,63	274,89	231,94
150	0,00	20,94	0,00	1,17	2,26	452,72	289,39

The column's labels are BF for Brute Force algorithm, NN for nearest neighbour algorithm, SA for Simulated Annealing, AC for Any Colony and GA for Genetic Algorithm.
At the first table, there are shown the results of the comparison in terms of quality of the solution. This is to say, the best tour produced by all algorithm on each iteration gets a 100%, and then the other tours are compared with that tour.
The second table shows the time average, in seconds, which each algorithm produce a solution depending on the number of cities n.
For each n shown, all algorithms were run at least 5 times and computed its average of quality and time.

Chapter 5

Conclusion

It have been proved that Brute Force algorithm always returns the best solution for the TSP but it is limited to a number of 11 cities, otherwise its computational time is extremely large. In order to solve TSP for $n \geq 11$, heuristics methods such 2-opt and 3-opt are used because they work very efficient when $n \leq 50$ as shown in Figures 4.1 and 4.2. With n bigger than 50 its computational time increase faster than exponentially and it is when the presented Markov Chain's based algorithm take a step forward. Ant Colony Optimization and Genetic Algorithm both produces good quality solutions with almost the similar computation speed, but it is the Genetic Algorithm that almost always guarantees to find the best solution among the others.

Bibliography

[1] Chandra, Barun Karloff, Howard Tovey, Craig. *New Results on the Old k -opt Algorithm for the Traveling Salesman Problem*, SIAM J. Comput., 28, (1999), 1998-2029.

[2] G. Dantzig, R. Fulkerson and S. Johnson, *Solution of a Large-Scale Traveling-Salesman Problem*, INFORMS , **88**, no. 1, (1954).

[3] Martin, Olivier and Otto, Steve and Felten, Edward. *Large-Step Markov Chains for the Traveling Salesman Problem*, Complex Systems, 5, (1997),

[4] Ben-Ameur, Walid. *Computing the Initial Temperature of Simulated Annealing*, Computational Optimization and Applications, 29, (2004), 369-385.

[5] S. Anily and A. Federgruen. *Simulated Annealing Methods with General Acceptance Probabilities*, Journal of Applied Probability, 24, (1987), 657-667

[6] Wenhong Tian. *On the Classification of NP Complete Problems and Their Duality Feature*, International Journal of Computer Science Information Technology, 10, (2018),

[7] Matthew Richey. *The Evolution of Markov Chain Monte Carlo Methods*, Published Online, (2017), 383-413

[8] Dennis Francis Roerty. *M-SALESMAN BALANCED TOURS TRAVELING SALESMAN PROBLEM WITH MULTIPLE VISITS TO CITIES ALLOWED*, Thesis, (1974).

[9] Gutin, Gregory and Yeo, Anders and Zverovich, Alexey. *Traveling Salesman Should not be Greedy: Domination Analysis of Greedy-Type Heuristics for the TSP*, Discrete Applied Mathematics, (2002), 117, 81-86

[10] Timothy Huang, Yuriy Nevmyvaka. *A Practical Markov Chain Monte Carlo Approach to Decision Problems* Department of Mathematics and Computer Science, (2001).

[11] Richard L. Smith and Luke Tierney. *EXACT TRANSITION PROBABILITIES FOR THE INDEPENDENCE METROPOLIS SAMPLER*, (1996).

[12] Timothy Huang, Yuriy Nevmyvaka. *A Practical Markov Chain Monte Carlo Approach to Decision Problems* Department of Mathematics and Computer Science, (2001).

[13] Mark Jerrum Alistair Sinclair. *THE MARKOV CHAIN MONTE CARLO METHOD: AN APPROACH TO APPROXIMATE COUNTING AND INTEGRATION*,Online, 12, (1996). 482-520

[14] Gaertner, Dorian and Clark, Keith. *On Optimal Parameters for Ant Colony Optimization Algorithms.* Proceedings of the 2005 International Conference on Artificial Intelligence, ICAI'05, 1, (2005). 83-89

[15] Sallabi, Omar and Elhaddad, Younis. *An Improved Genetic Algorithm to Solve the Traveling Salesman Problem.* (2009).