République Algérienne Démocratique et Populaire

الجمهورية الجزائرية الديمقراطية الشعبية

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

وزارة التعليم العالي والبحث العلمي

ECOLE NATIONALE SUPÉRIEURE D'INFORMATIQUE

المدرسة الوطنية للإعلام الآلي

(المعهد الوطني للتكوين في الإعلام الآلي سابقا)

École nationale Supérieure d'Informatique

ex. INI (Institut National de formation en Informatique)

Deuxième Année du Cycle Supérieur (2CS)

Combinatorial Optimisation Assignement Report

The Traveling Salesman Problem Exact and Approximate Algorithms

Made by :
ALIOUSALAH Mohamed Nassim
BELGOUMRI Mohammed Djameleddine
MELIANI Abdelghani
AÏT MEZIANE Mohamed Amine
LOUCIF Taha Ammar

Supervised by :
Mme. BESSEDIK M
M. KECHID A

June 4, 2022

Contents

List of Figures

List of Algorithms

Introduction

The traveling salesman problem (which will be denoted by TSP for brevity's sake) is a classic problem in computer science. It is a typical example of a combinatorial optimization problem, that is, an optimization problem with a *discrete* solution space.

In its simplest form, the TSP asks the following question: "A salesman wants to take *the best*[1] *possible itenerary* between a set of cities, every city must be visited exactly once, and the salesman must start and finish at the same city. How can he find this itenerary?"

It is not dificult to see the practical use of solving the TSP. In fact, many important problems like vehicle routing, scheduling, array clustering [7], and circuit design [6] can be *expressed*[2] as TSP instances.

Furthermore, the TSP is of particular theoretical interest to complexity theory researchers, as its decision variant a member of a very important family of decision problems called NP-complete problems.

In this document, we will introduce the TSP, investigate some of its properties and applications, and propose a few algorithms for solving it.

[1]Usually "best" means shortest.

[2]Formally speaking, these problems can be *reduced* to TSP.

Chapter 1

General Concepts

1.1 Computational Complexity

Throughout this document, we will often find ourselves in need of a method to objectively measure the difficulty of a problem or the efficiency of an algorithm. Fortunately, there exists an entire branch of theoretical computer science that addresses these very questions: *the theory of computational complexity*.

The theory of computational complexity formalizes the intuitive concept of the *difficulty* of a problem. Quite reasonably, this discipline relies on the premise that a problem is as difficult as it is to perform its most efficient solution, or, to use technical terms, to *execute the most efficient algorithm* that solves the problem.

1.1.1 Decision Problems and Complexity Classes

For historical (and technical) reasons, most of the work in this branch has been done around a special type of problem called *decision problems*. A decision problem is a problem that has a binary answer, that is, given an instance (or an input) of the problem, we compute an answer (or output) that is an element of some preknown set with cardinality 2. The sets $\{0, 1\}$, $\{\texttt{false}, \texttt{true}\}$, and $\{\texttt{no}, \texttt{yes}\}$, are common examples of such a set, the former of which will be used in the rest of this discussion.

A decision problem can therefore be defined as the problem of evaluating some *computable*[1] function $f : S \to \{0, 1\}$ where S is some set of inputs.

These functions can be mapped to decidable subsets of S by associating every such a set P with its characteristic function $\mathbb{1}_P$. This correspondence is what motivates Definition 1 of decidable problems.

Definition 1 (Decision Problem)**.**
A decision problem is a pair $X = \langle S, P\rangle$ where S is a countable set called the *instance space* and $P \subset S$ is called the set of *positive instances.* We say the problem X is decidable iff P is decidable (i.e. if $\mathbb{1}_P$ is computable).

To better understand Definition 1, we consider Examples 1.1, and 1.2, the latter of which is of particular historical significance in the context of complexity theory.

Example 1.1 (A Number of Decision Problems)**.**

- PRIME is the problem $\langle \mathbb{N}, P\rangle$ where P is the set of all prime numbers.
- HAM, or the *hamiltonicity problem* is the problem $\langle S, P\rangle$ where S is the set of all undirected graphs and P is the set of all hamiltonian graphs.
- CLIQUE. An instance of this problem is a pair $\langle G, k\rangle$ where G is a graph and $k \in \mathbb{N}$. Such an instance is positive iff G has a clique of size k.
- PAIR or the parity problem is the problem $\langle \{0, 1\}^*, P\rangle$ where

$$P = \big\{w \in \{0, 1\}^* \big| |w|_1 \equiv 0 \bmod 2\big\}$$

Example 1.2 (SAT)**.**
The *satisfiability problem of boolean logic*, or SAT, is the problem $\langle S, P\rangle$ where S is the set of all formulae of boolean logic and P is the set of all *satisfiable* formulae. A formula $\varphi \in S$ is satisfiable iff it has a satisfying assignment or a *model*, that is, if its negation is not a tautology. For instance, the formula φ below is satisfiable (by the assignment $x_1 = 0, x_2 = 0, x_3 = 1$ for example) while ψ is not.

$$\varphi = ((x_1 \vee \neg x_2) \wedge x_3) \vee (x_2 \wedge \neg x_3)$$
$$\psi = \neg(x_1 \vee \neg x_2) \wedge \neg(x_2 \vee \neg x_1)$$

[1]The model of computation is not important when defining decision problems, but it becomes so when discussing their complexity. The turing machine is the model we will use throughout this work.

The complexity of a decision problem is given by two pieces of information, the first being its time complexity (intuitively, this is the runtime of the *fastest* turing machine deciding the problem), and the second its space complexity (the *minimal* number of distinct visited cells on the tape of turing machine deciding the problem). The rigorous definitions of these quantities are given in Appendix **??**. We will use the notions of *algorithms*, *runtime*, and *memory usage* as intuitive analogues of Turing machines, runtime and space complexity respectively.

As is common when discussing complexity, we will sort problems in a hierarchy of *complexity classes*. These complexity classes are based on the asymptotic behavior of the time and space complexities instead of the exact runtime or memory usage of a particular algorithm solving a particular problem. A few important complexity classes are given by Definition 2 [2].

Definition 2 (Most Important Complexity Classes)**.**
Let $f : \mathbb{N} \to \mathbb{R}_+$ be a function. We define the following classes of problems:

- $\mathsf{TIME}(f(n))$ is the set of problems X for which there exists a deterministic Turing machine $\mathcal{M}$ that decides X such that $t_{\mathcal{M}}(n) = O(f(n))$.
- $\mathsf{NTIME}(f(n))$ is the set of problems X for which there exists a nondeterministic Turing machine $\mathcal{M}$ that decides X such that $t_{\mathcal{M}}(n) = O(f(n))$.

From these two, we can define the following classes:

$$\mathsf{P} = \bigcup_{k\in\mathbb{N}} \mathsf{TIME}\left(n^k\right) \qquad \mathsf{NP} = \bigcup_{k\in\mathbb{N}} \mathsf{NTIME}\left(n^k\right)$$
$$\mathsf{EXPTIME} = \bigcup_{k\in\mathbb{N}} \mathsf{TIME}\left(2^{n^k}\right) \qquad \mathsf{NEXPTIME} = \bigcup_{k\in\mathbb{N}} \mathsf{NTIME}\left(2^{n^k}\right)$$

The list given by Definition 2 is of course far from complete. In fact, a very easy way to extend it is to add for every class C its *dual* class co-$C := \{\overline{X} | X \in C\}$ of complements of problems in C. It is however largely sufficient for the purposes of this investigation. In fact, we will mostly be dealing with the classes P and NP exclusively.

It is quite straightforward to verify the following inclusions between the classes defined thus far:

$$\begin{array}{ccccccc} \mathsf{P} & \subset & \mathsf{NP} & \subset & \mathsf{EXPTIME} & \subset & \mathsf{NEXPTIME} \\ \mathsf{P} & \subset & \text{co-}\mathsf{NP} & \subset & \mathsf{EXPTIME} & \subset & \text{co-}\mathsf{NEXPTIME} \end{array}$$

However, it is exceedingly difficult to prove strict inclusion for most pairs. As a matter of fact, the problem of determining whether $\mathsf{P} = \mathsf{NP}$ is an open one, as well as that of finding the intersection between C and co-C for $C \in \{\mathsf{NP}, \mathsf{NEXPTIME}\}$. The Venn diagram of Figure 1.1 shows the known inclusions (opting for strictness for unknown pairs).

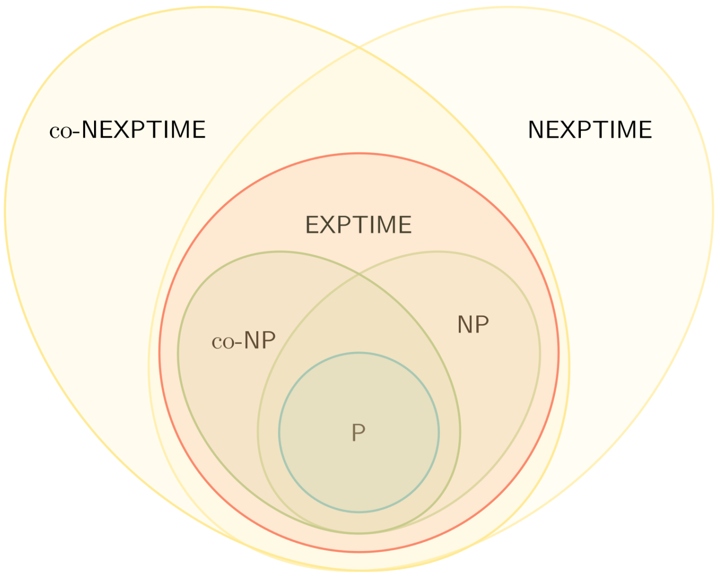

Figure 1.1: A Venn diagram of the classes defined so far.

1.1.2 Reducibility and Completeness

As we have seen in the previous section, it is currently unknown whether $\mathsf{P} = \mathsf{NP}$. As far as we know, it could be as easy to decide a problem with a deterministic machine as it is to decide it with a nondeterministic one. We don't know if problems in NP are *strictly harder* than those in P (even though we suspect them to be). This is hardly surprising given the vastness of NP. In order to prove $\mathsf{P} = \mathsf{NP}$, one must find a polynomial time algorithm for *every* problem in NP. And in order to prove that $\mathsf{P} \neq \mathsf{NP}$, one must prove that for at least one problem in NP, *all algorithms* are not polynomial time (this is admittedly an easier task than proving equality).

The point above is valid for all pairs of classes for which we have one inclusion but are uncertain of the other (NP and $\mathsf{EXPTIME}$ for example). The notion of completeness simplifies the task of proving inequality of two classes by reducing it to the task of proving that *one particular* problem (intuitively thought of as the hardest problem) in the supposedly bigger class is in deed a member of the smaller one. These *most difficult problems* are what the concept of completeness aims to formalize. In order to define them, we must first address the question of how to compare the difficulty of two problems, which is exactly what the relation of *reducibility* introduced in Definition 3 allows us to do.

Definition 3 (Polynomial Reduction)**.**
Let $X = \langle S_1, P_1 \rangle$ and $Y = \langle S_2, P_2 \rangle$ be two decision problems. A *polynomial reduction* of X to Y is a function $f : S_1 \to S_2$ computable by a deterministic Turing machine in polynomial time such that:

$$\forall i \in S_1 \; i \in P_1 \iff f(i) \in P_2$$

If such a reduction exists, X is said to be *polynomially reducible* to Y, which we denote by $X \preceq_{\mathsf{P}} Y$.

One can show that $\preceq_{\mathsf{P}}$ is both transitive, and reflexive (a preorder on decision problems) without too much difficulty. Furthermore, the relation $\preceq_{\mathsf{P}}$ can be shown to satisfy Proposition 1.1 which will become very important once NP-completeness is defined.

Proposition 1.1.
Let X and Y be two decision problems. If $X \preceq_{\mathsf{P}} Y$ and $Y \in \mathsf{P}$, then $X \in \mathsf{P}$.

Proof.
The composition of the Turing machine that computes the polynomial reduction of X to Y and the one that decides Y in polynomial time is a polynomial time Turing machine that decides X. □

We now define NP-completeness.

Definition 4 (NP-hardness, NP-completeness)**.**
A problem X is NP-hard if and only if :

$$\forall Y \in \mathsf{NP} \; Y \preceq_{\mathsf{P}} X$$

A problem X is NP-complete if and only if X is NP-hard and $X \in \mathsf{NP}$.

Under our intuitive interpretation of $\preceq_{\mathsf{P}}$, an NP-hard problem is a problem that is at least as difficult as any problem in NP. An NP-complete one is then the hardest problem in NP. It stands to reason then that if an NP-hard problem X is in P, we would have $\mathsf{NP} \subset \mathsf{P}$ and hence $\mathsf{P} = \mathsf{NP}$. This is in deed the case as affirmed by Corollary 1.2.

Corollary 1.2. *If $\mathsf{P} \cap \mathsf{NP}-$hard $\neq \emptyset$, then $\mathsf{P} = \mathsf{NP}$*

Proof.
Let X be an NP-hard problem that is also in P, and let $Y \in \mathsf{NP}$. By definition of NP-hardness, $Y \preceq_{\mathsf{P}} X$, and by Proposition 1.1 and the fact that $X \in \mathsf{P}$, $Y \in \mathsf{P}$. Therefore, $\mathsf{NP} \subset \mathsf{P}$, and hence $\mathsf{P} = \mathsf{NP}$. □

The same intuitive analysis that led us to Corollary 1.2 suggests that if a problem is harder[2] than an NP-hard problem, then it must be NP-hard as well. This is once again correct as affirmed by Proposition 1.3.

Proposition 1.3.
Let X and Y be two decision problems. If X is NP-hard and $X \preceq_{\mathsf{P}} Y$, then Y is NP-hard.

Proof.
Let X, Y, Z be decision problems such that X is NP-hard and $X \preceq_{\mathsf{P}} Y$, and $Z \in \mathsf{NP}$. By NP-hardness of X, $Z \preceq_{\mathsf{P}} X$, and by transitivity of $\preceq_{\mathsf{P}}$ we have $Z \preceq_{\mathsf{P}} Y$. Y is then NP-hard. □

Although the idea of NP-completeness is promising, it is not immediately obvious how it makes approaching P vs NP easier. It would seem that proving a problem is NP-hard is as difficult as solving P vs NP. This is once more due to the unfathomable vastness of NP. Fortunately, this initial impression proved wrong. We know of many NP-hard and NP-complete problems. The first problem to be proven NP-complete is the problem SAT we invoked in Example 1.2. This has been done by Stephen Cook and Leonid Levin who proved Theorem 1.4 in 1971. A proof of this theorem is given in [3].

Theorem 1.4 (Cook Levin)**.**
SAT is NP-complete.

[2]In the $\preceq_{\mathsf{P}}$ sense.

Now that we have one problem that we know is NP-complete, it becomes significantly easier to prove that a given problem is NP-hard. Using Proposition 1.3, we can show any problem is NP-hard by reducing a known NP-hard problem to it. This is what we will do in the following example by proving 3-SAT is NP-complete.

Example 1.3.
An instance of 3-SAT is a conjunction of clauses with at most 3 literals each and is positive iff it is satisfiable. To show that 3-SAT is NP-complete, we will reduce SAT to it. It is technically necessary to show that 3-SAT $\in$ NP too, but this is easy given that it is a subproblem of SAT which is already known to be in NP. We will therefore focus on proving it NP-hard.

To reduce SAT to 3-SAT, we must find for every formula φ of boolean logic a formula φ' in 3CNF, with size polynomial in that of φ, such that φ' is satisfiable iff φ is satisfiable. We start by considering the *syntax tree* of φ, for example, the syntax tree of the formula

$$\varphi = ((x_1 \vee \neg x_2) \wedge x_3) \vee (x_2 \wedge \neg x_3)$$

is shown in Figure 1.2. We label each internal node of this tree with a new variable (the leaves are associated with the variables of φ), this produces the tree of Figure 1.3.

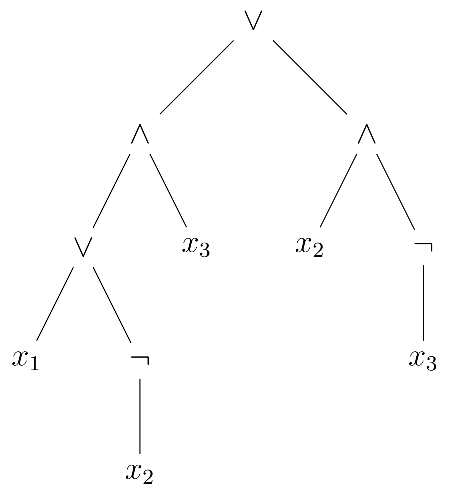

Figure 1.2: Syntax tree of the formula φ

Each new variable x_i is then associated with an equivalence e_i according to the following rules:

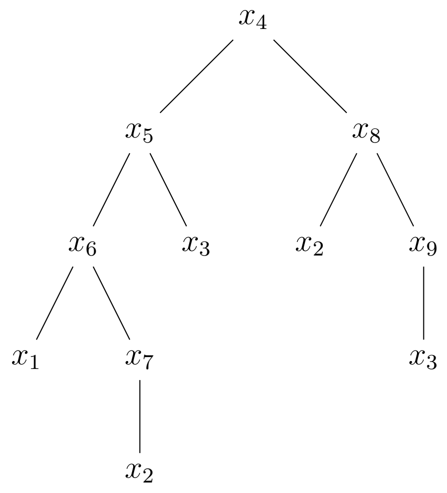

Figure 1.3: Syntax tree of Figure 1.2 labeled with new variables.

- If the node labeled by the variable x_i is $\vee$, we associate it with the formula $x_i \leftrightarrow x_j \vee x_k$ where x_j and x_k are the children of x_i.
- If it is labeled with $\wedge$, we associate it with the formula $x_i \leftrightarrow x_j \wedge x_k$.
- If it is labeled with $\neg$, we associate with the formula $x_i \leftrightarrow \neg x_j$.

Finally, by taking the conjunction[3] $\bigwedge\limits_{i>n} e_i$ of the equivalences thus obtained, and replacing equivalences with their clausal forms given by the following tautologies:

$$\begin{aligned} x \leftrightarrow \neg y &\equiv (x \vee \neg y) \wedge (\neg x \vee y) \\ x \leftrightarrow y \wedge z &\equiv (x \vee \neg y \vee \neg z) \wedge (\neg x \wedge y) \wedge (\neg x \wedge z) \\ x \leftrightarrow y \wedge z &\equiv (\neg x \vee y \vee z) \wedge (x \wedge \neg y) \wedge (x \wedge \neg z) \end{aligned}$$

We get a formula φ' in 3CNF, which is logically equivalent toφ (and is a fortiori satisfiable iff φ is satisfiable). Furthermore, since the size of the syntax tree is proportional to the size of φ, so is the size of φ' (i.e. this reduction is in deed polynomial). It then follows that SAT $\preceq_{\mathsf{P}}$ 3-SAT, and by Proposition 1.3, that 3-SAT is NP-complete.

Using similar techniques, countless problems have been shown to be NP-complete. Examples include CLIQUE (the problem of deciding whether a graph has a k-clique

[3] n being the number of variables in φ

for $k \in \mathbb{N}$), VERTEX-COVER (the problem of deciding whether a graph has a covering set of vertices with cardinality k for $k \in \mathbb{N}$), and crucially for our study, HAM (mentioned in Example 1.1).

1.1.3 Optimization Problems and Their Classification

Despite their ubiquity, decision problems are not always powerful enough to model a given situation. Combinatorial optimization extends them with a much more general class of problems. Instead of finding a binary answer, a combinatorial optimization problem (or simply, an optimization problem) seeks an *optimal* solution in a discrete *space of options.* The optimality of a solution is judged with respect to an *objective function.*

Definition 5 (Optimization Problem)**.**
An optimization problem is a quintuplet $A = \langle X, S, F, \mu, g \rangle$ where

(i) X is a set of *instances*,

(ii) S is a set of *solutions*,

(iii) $F : X \to \mathcal{P}(S)$, for $x \in X$, $F(x)$ is called the set of feasible solutions of instance x,

(iv) $\mu : \big\{(x, s) \big| x \in X \wedge s \in F(x)\big\} \to \mathbb{R}_+$, $\mu(x, s)$ is called the *measure* of the solution s to instance x,

(v) $g \in \{\min, \max\}$ is the *goal* of the problem.

The following notation will be used throughout the rest of this document to refer to optimization problems

$$A = \underset{s \in F(x)}{g} \{\mu_x(s)\}$$

If $g = \min$, the problem is called a *minimization problem*, otherwise, it is called a *maximization problem.*

For an instance x, we define the *objective function* $\mu_x : F(x) \to \mathbb{R}_+, s \mapsto \mu(x, s)$, the *optimum* $\mathrm{opt}(x) := g \big\{\mu(x, s) \big| s \in F(x)\big\}$, and the set of *optimal solutions* $\mu_x^{-1}(\{\mathrm{opt}(x)\})$. Such an instance is said to be *solvable* if $F(x) \neq \emptyset$.

By *solving* A, we mean finding a computable function $f : X \to S$ (also called a solver) such that for every solvable $x \in X$, $f(x) \in F(x)$. If f has the additional property that for every solvable $x \in X$, $f(x)$ is an optimal solution to x, we say that f is an *exact* solver of A.

To understand further clarify Definition 5, we will consider a few examples of optimization problems.

Example 1.4.
LP or Linear Programming is the problem defined by:

- **The instance space**: An instance is a triplet (A, b, c) where $b, c \in \mathbb{R}^n$, and $A \in \mathcal{M}_n(\mathbb{R})$ for some $n \in \mathbb{N}$.
- **The solution space**: A solution is a vector $x \in \mathbb{R}^n_+$.
- **Feasible solutions (constraints)**: A solution x is feasible iff $Ax \leq b$.[4]
- **Objective function**: $\mu_{A,b,c}(x) = {}^t cx$.
- **Goal**: LP can be either a maximization or a minimization problem.

The maximization variant of LP can then be denoted by

$$\max_{Ax \leq b} \left\{ {}^t cx \right\}$$

Or

$$\begin{aligned} \max\ & {}^t cx \\ \text{where } & Ax \leq b \end{aligned}$$

To every optimization problem A, we can associate a decision problem B. An instance of B is a pair $\langle x, m \rangle$, and is positive iff $\text{opt}(x) = m$. In this fashion we can use the complexity classes of decision problems to define analogues for optimization problems.

Definition 6 (NPO)**.**
An optimization problem A is in NPO (NP optimization) iff the decision problem associated to A is in NP.

[4]Inequality is taken componentwise.

It stands the reason that the problems in NPO can also be divided on the basis of whether their decision variants are NP-complete, this is however not a very fruitful classification.

Instead, we classify NPO problems based on the difficulty of approximating them (since they can't be easily solved under the hypothesis $\mathsf{NP} \neq \mathsf{P}$). The very bottom of this hierarchy are problems with decision variants in P.

Immediately above, we find problems that can be approximated to an arbitrary precision in polynomial time, this is captured by Definition 7.

Definition 7 (PTAS)**.**
A Polynomial Time Approximation Scheme (PTAS) for an optimization problem $A = \langle X, S, F, \mu, g\rangle$ is a computable function $f : X \times \mathbb{R}_+ \to S$ such that

$$\forall\varepsilon > 0, \forall x \in X, \left|\frac{\mathrm{opt}(x) - \mu_x(f(x,\varepsilon))}{\mathrm{opt}(x)}\right| < \varepsilon$$

and whose runtime is polynomial in the size of the instance.

An example of an NPO problem (and one whose decision variant is NP-complete in fact) who has a PTAS is $\mathsf{KNAPSACK}$. Immediately above PTAS problems, we find problems who can be approximated to some minimum (but not arbitrary) degree of precision.

Definition 8 (APX)**.**
An optimization problem is APX iff it has a constant-factor approximation algorithm i.e. a solver f with the propriety

$$\exists r > 1, \forall x \in X, f(x) \leq r \cdot \mathrm{opt}(x)$$

A problem is APX-complete iff all APX problems are PTAS-reducible to it.

It is clear that All PTAS problems are APX, but the opposite inclusion is equivalent to $\mathsf{P} = \mathsf{NP}$. Similarly, a problem is NPO-complete iff all of NPO is PTAS-reducible to it.

Chapter 2

The Traveling Salesman Problem

2.1 A Brief History

The mathematical study of the problem we have been referring to as the Traveling Salesman Problem most likely began in the 1930s [6]. However, the origins of that terminology remain unclear.

What is outside the realm of debate, is the fact that the problem itself has been popularized among mathematicians thanks to the efforts of mathematician Merrill Flood [1]. He introduced it to his colleagues at the RAND corporation[1], who interned made it popular in the wider world of operations research to the point that it became the archetypical example of a hard combinatorial optimization problem.

Although, it should be noted that in 1831, a non-mathematical text by the title *"Der Handlungsreisende, wie er sein soil und was er zu thun hat, um Auftrage zu erhalten und eines glucklichen Erfolgs in seines Geschdften gewiss zu sein"*, which translates to *"The Traveling Salesman, how he should be and what he should do to get Commissions and to be Successful in his Business. By a veteran Traveling Salesman"* [7, 8]. In the last chapter, one can read

> By a proper choice and scheduling of the tour, one can often gain so much time that we have to make some suggestions ... The most important aspect is to cover as many locations as possible without

[1]Research and Development, An American think tank created in 1948.

visiting a location twice ...

In 1954, the paper “Solution of a Large-Scale Traveling-Salesman Problem” [4] which caused interest in the TSP to explode.

2.2 Motivation

2.3 Formal Statement

Chapter 3

Exact Solutions

Having stated TSP, it is natural to consider its solvability and, if so, its solution. It is quite straightforward to show that TSP is solvable. In fact, three exact solvers will be presented in this chapter.

3.1 The Naive Approach

3.1.1 The Algorithm

In this section, we will examine the simplest approach to solving TSP. It simply consists of a brute-force search. More specifically, it is an exhaustive search of the solution space (i.e., the set of all possible tours).

It is easy to see that this indeed is an exact solver, and that TSP is a fortiori solvable, as is by the same reasoning any problem with a finite solution space . The pseudocode for this approach is given by Algorithm 3.1.

3.1.2 Performance Analysis

Since this is an exact algorithm, it makes no sense to talk about the quality of the solutions it produces, they are always optimal. However, it is possible to assess the performance of the algorithm by examining its computational complexity.

ALGORITHM 3.1: Naive TSP

```
Input: instance
// An instance of the TSP
begin
    $n \leftarrow$ size($instance$) $- 1$
    best_length $\leftarrow +\infty$
    foreach $\sigma \in S_n$ do
        if path_length($instance, \sigma$) $<$ $best\_path$ then
            best_path $\leftarrow \sigma$
            best_length $\leftarrow$ path_length($instance, \sigma$)
        end
    end
end
Output: best_path
```

From this point of view, the naive algorithm is very inefficient. In fact, the runtime of Algorithm 3.1 is proportional to $|S_n|$ which gives a runtime of $\Theta(n!)$. This is completely unfeasible for even moderate instances (for 15 cities, we get something on the order of 8.7×10^{10}), a more efficient algorithm is required.

3.2 Branch and Bound Methods

3.3 Dynamic Programming

ALGORITHM 3.2: Dynamic Programming TSP

```
   Input: instance :An instance of the TSP
1  begin
2      $n \leftarrow$ size(instance)
3      for $k \in [\![1, n-1]\!]$ do
4          best$[\{k\}, k] \leftarrow$ distance(0, k)
5      end
6      for $s \in [\![1, n-1]\!]$ do
7          forall $S \subset \mathcal{P}_s([\![1, n-1]\!])$ do
8              forall $k \in S$ do
9                  $best[S,k] \leftarrow \min\{best\,[S - \{k\}, m] + \texttt{distance}(m,k) : m \in S - \{k\}\}$
10             end
11         end
12     end
13     optimal $\leftarrow \min_{k>0}\{best[[\![1, n-1]\!], k] + \texttt{distance(1, k)}\}$
14 end
   Output: optimal
```

Chapter 4

Approximate Solutions 1 (Heuristics)

Despite them being exact, the algorithms we introduced so far are not always practical to use on a real world problem. This is due to their computational expense. No efficient exact algorithm known for the TSP. In fact, under the assumption $\mathsf{P} \neq \mathsf{NP}$, no such algorithm exists. A reasonable alternative to consider is finding an efficient *approximate* algorithm.

Approximate methods sacrifice the exactness of the solution for efficiency. The extent to which we can tolerate loss of accuracy is however limited. A constant time solution is useless if it gives a sufficiently large error.

4.1 Nearest Neighbor

All approximate methods are based on a compromise between accuracy and time. Nearest Neighbor being one such method, it is built on one such compromise. It places a higher emphasis on runtime than accuracy. In deed, we will see that Nearest Neighbor is the fastest among the algorithms we will consider. Correspondingly, it also gives the most mediocre solutions.

4.1.1 The Algorithm

Nearest Neighbor is what is called a *construction heuristic*, which means it gives a solution that is feasible by construction, as opposed to *improvement heuristics* which start with a feasible solution and then improve it to get a (locally) optimal solution.

It is also a *greedy* algorithm. Greedy algorithms are iterative algorithms that take the optimal action at each iteration. The choice of action is based only on the information available at the current state. No backtracking is performed. They are therefore seldom able to find optimal solutions.

Nearest Neighbor in particular explores one branch of the search tree traversed by Branch and Bound. It is in deed equivalent to the first iteration of Branch and Bound. What it does is simply take the shortest edge to a city that has not been visited yet until no such city remains. This is exactly what Algorithm 4.1 does.

ALGORITHM 4.1: Nearest Neighbor

```
Input: instance
// represented as an adjacency matrix
begin
    path ← [0]
    unvisited ← cities − [origin]
    while unvisited ≠ ∅ do
        last ← path[-1]
        forall city in nearestNeighbors(last) do
            if city ∉ unvisited then
                break
            end
            path ← path + [city]
            visited ← visited - [city]
        end
    end
    Output: path
end
```

4.1.2 Performance Analysis

In order to judge the performance of Algorithm 4.1, two factors must be considered: runtime complexity and solution quality.

On the first measure, Nearest Neighbor does exceedingly well. Not only is

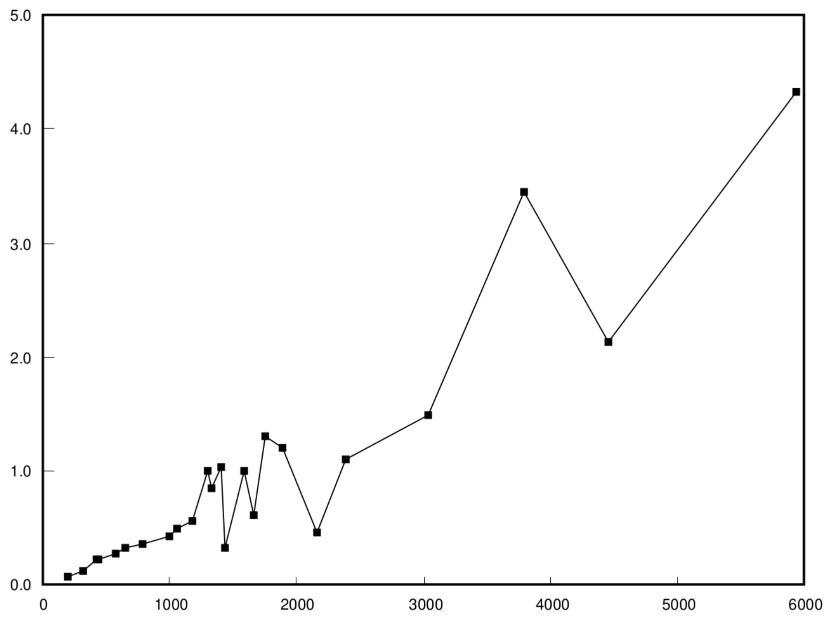

Figure 4.1: Average runtime in seconds of Nearest Neighbor

its runtime polynomial in the number of cities, the degree of the polynomial is remarkably low. In fact, it can be shown that the runtime of Nearest Neighbor is $\Theta(n^2)$, n being the number of cities (see Figure 4.1), which is impressive given the difficulty of TSP. This is because the main while loop is executed n times, and each call to `nearestNeighbors` runs in $\Theta(n)$ time.

On the second however, it is difficult to come up with an algorithm that gives worse solutions than Nearest Neighbor. It is, in a certain sense, the worst possible solver of TSP. This assessment is made rigorous by Proposition 4.1.

Proposition 4.1 (Anti optimality of Nearest Neighbor).
Let $n \in \mathbb{N}$, NN be a Nearest Neighbor solver.

(i) $\forall r \in [1, +\infty[$*, there exists an instance x of TSP with at least n cities such that* $\mu_x(\mathrm{NN}(x)) \geq r \cdot \mathrm{opt}(x)$

(ii) There exists an instance x of TSP with at least n cities such that NN gives the worst tour of x.

Proof. See [7]. □

Worse still, (ii) of Proposition 4.1 persists even when we restrict our interest to the metric TSP (i.e. if we suppose the triangle inequality). This is not the case for almost any other heuristic.

Intuitively, this is because despite the optimality of the first few edges of the nearest neighbor tour, the later edges can be very costly and can't be avoided due to the lack of backtracking.

4.2 Improvement Heuristics 2-OPT

4.2.1 The Algorithm

Improvement heuristics are the polar opposite of construction heuristics. They need an initial solution to start from. The solution is then modified to get a better neighboring solution.

In particular, 2-OPT takes a tour and swaps two edges to reduce the tour length. It keeps doing this until no such edge pair can be found. Algorithm 4.2 describes this process.

4.2.2 Performance Analysis

From a runtime perspective, 2-OPT is a somewhat decent heuristic. Its average case runtime is $O(n^2)$, but its worst case runtime is exponential, Figure 4.2 shows the observed runtime of 2-opt vs the size of the instance. This makes it generally too expensive to use on its own, but it is very useful when combined with other heuristics.

The solutions it gives by contrast are of very high quality. Very tight upper bounds on the 2-OPT tour's length can be proven under the assumption of the triangle inequality [5], and even better bounds can be achieved. This is formalized by Proposition 4.2.

Proposition 4.2 (Bounds on 2–OPT)**.** *If x is a TSP instance, then*

$$\mu_x(2OPT(x)) \leq \sqrt{\frac{n}{2}}\mathrm{opt}(x)$$

ALGORITHM 4.2: 2–OPT

```
Input: instance  // An instance of TSP
Input: starting_tour  // A tour of the instance
begin
    repeat
        exit gets true
        foreach $i \in [\![0, n-1]\!]$ do  // For evry city in the tour

            foreach $j$ in $\in [\![0, n-1]\!] - \{i-1, i, i+1\}$ do  // For evry city non-adjacent to i in the tour

                if distance($i$, $i+1$) + distance($j$, $j+1$) < distance($i$, $j$) + distance($j+1$, $i+1$) then  // If the swap is benificial

                    swap($i$, $i+1$, $j$, $j+1$)  // Swap the edges

                    exit gets false  // Keep looping

                end
            end
        end
    until exit
end
```

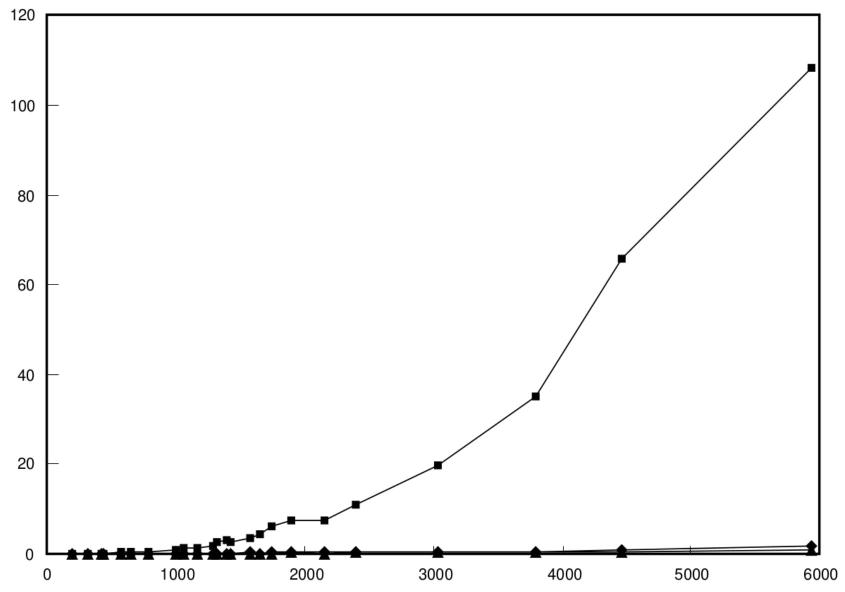

Figure 4.2: Average runtime in seconds of Nearest Neighbor initialized 2–OPT

Chapter 5

Approximate Solutions 2 (Metaheuristics)

5.1 Simulated Annealing

5.2 Genetic Algorithm

ALGORITHM 5.1: Simulated Annealing TSP

```
1  Suppose you have a combinatorial optimization problem with just five elements, and
     where the best ordering/permutation is [B, A, D, C, E]. A primitive approach would
     be to start with a random permutation such as [C, A, E, B, D] and then repeatedly
     swap randomly selected pairs of elements until you find the best ordering. For
     example, if you swap the elements at [0] and [1], the new permutation is [A, C, E, B,
     D]. This approach works if there are just a few elements in the problem permutation,
     but fails for even a moderate number of elements. For example , a banchmark with n
     = 20 cities, there are 20! possible permutations = 20 * 19 * 18 * . . * 1 =
     2,432,902,008,176,640,000. That's a lot of permutations to examine.
2  make a random initial guess set initial temperature loop many times swap two
     randomly selected elements of the guess compute error of proposed solution if
     proposed solution is better than curr solution then accept the proposed solution else if
     proposed solution is worse then accept the worse solution anyway with small
     probability else don't accept the proposed solution end-if reduce temperature slightly
     end-loop return best solution found
   Input: instance: an instance of TSP, initial_temperature, cooling_rate,
          final_temperature
3  begin
       // Initialize the temperature
4      path ← random(size(instance))
5      cost ← path_length(path)
6      temperature ← initial_temperature
7      while $temperature > final_temperature$ do  // Simulated Annealing
8
           // Generate neighbor path by swapping two cities
9          nodes_to_swap ← random(0, size(instance), 2)
10         new_path ← copy(path)
11         swap(new_path,nodes_to_swap)
           // Accept new path if it is better or if it is accepted with a probability
12         difference ← path_length(new_path) - cost; if $difference < 0$ or
             random(0,1) < $\exp(-difference/temperature)$ then
13             path ← new_path
14             cost ← path_length(path)
15         end
           // Cool down
16         temperature ← $temperature * cooling\_rate$
17     end
18 end
   Output: path
```

ALGORITHM 5.2: Genetic Algorithm TSP

This section describes a simple genetic algorithm. The vocabulary will probably look a little bit ”esoteric” to the OR specialist, but the aim of this section is to describe the basic principles of the genetic search in the most straightforward and simple way. At the origin, the evolution algorithms were randomized search techniques aimed at simulating the natural evolution of asexual species . In this model, new individuals were created via random mutations to the existing individuals. Holland and his students extended this model by allowing ”sexual reproduction”, that is, the combination or crossover of genetic material from two parents to create a new offspring. These algorithms were called ”genetic algorithms” , and the introduction of the crossover operator proved to be a fundamental ingredient in the success of this search technique.

Basically, a genetic algorithm operates on a finite population of chromosomes or bit strings. The search mechanism consists of three different phases: evaluation of the fitness of each chromosome, selection of the parent chromosomes, and application of the mutation and recombination operators to the parent chromosomes. The newchromosomes resulting from these operations form the next generation, and the process is repeated until the system ceases to improve. In the following, the general behavior of the genetic search is first characterized. Then, section 2.2 will describe a simple genetic algorithm in greater detail. First, let us assume that some function f(x) is to be maximized over the set of integers ranging from 0 to 63. In this example, we ignore the obvious fact that such a small problem could be easily solved through complete enumeration. In order to apply a genetic algorithm to this problem, the variable x must first be coded as a bit string. Here, a bit string of length 6 is chosen, so that integers between 0 (000000) and 63 (I 11111) can be obtained. The fitness of each chromosome is f(x), where x is the integer value encoded in the chromosome. Assuming a population of eight chromosomes, it is possible to create an initial population by randomly generating eight different bit strings, and by evaluating their fitness, through f, as illustrated in figure 3. For example, chromosome 1 encodes the integer 49 and its fitness is f(49) = 90.

5.3 Genetic Algorithm’s steps

1. Create an initial population of P chromosomes (generation 0). 2. Evaluate the fitness of each chromosome. 3. Select P parents from the current population via proportional selection (i.e.,the selection probability is proportional to the fitness). 4. Choose at random a pair of parents for mating. Exchange bit strings with the one-point crossover to create two offspring. 5. Process each offspring by the mutation operator, and insert the resulting offspring in the new population. 6. Repeat steps 4 and 5 until all parents are selected and mated (P offspring are created). 7. Replace the old population of chromosomes by the new one. 8. Evaluate the fitness of each chromosome in the new population. 9. Go back to step 3 if the number of generations is less than some upper bound. Otherwise, the final result is the best chromosome created during the search.

Input: instance: an instance of TSP, population_size,
mutation_rate,
max_generations,
elitism: boolean

```
begin
    // Initialize the population
    population ← {random(size(instance)) ∗ population_size}
    // Evolve the population
    for generation ∈ ⟦1, max_generations⟧ do
        // Selection
        if elitism then  // select best

            selected ← argmin(population, .1, key=path_length())
        else  // select randomly

            selected ← random(size(population), .1)
        end
        // Crossover (and mutation)
        for i ∈ ⟦1, population_size⟧ do
```

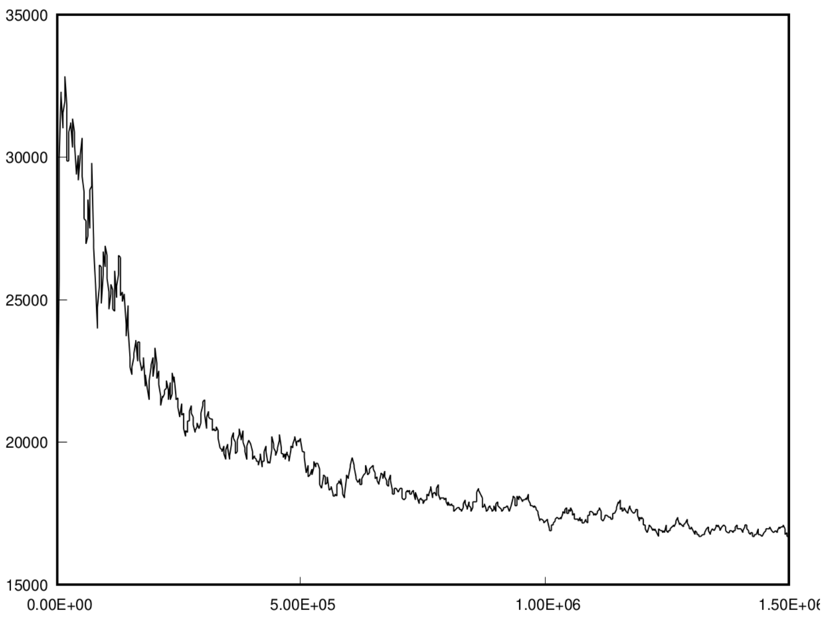

Figure 5.1: Path length vs initial temperature

Bibliography

[1] David L. Applegate et al. *The traveling salesman problem a computational study.* Princeton University Press, 2006.

[2] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach.* 2007.

[3] Carton, Olivier Perrin, and Dominique. *Langages formels Calculabilité et complexité Cours et exercices corrigés.* (fr). Vibert, 2014.

[4] G. Dantzig and R. Fulkerson. "Solution of a Large-Scale Traveling-Salesman Problem". In: (1954).

[5] Stefan Hougardy, Fabian Zaiser, and Xianghui Zhong. "The Approximation Ratio of the 2-Opt Heuristic for the Metric Traveling Salesman Problem". In: (2020).

[6] Gerhard Reinelt. *The Traveling Salesman Computational Solutions for TSP Applications.* Springer, 1994.

[7] *THE TRAVELING SALESMAN PROBLEM A Guided Tour of Combinatorial Optimization.* John Wiley & Sons Ltd., 1985.

[8] B. F. Voigt. *Der Handlungsreisende – wie er sein soll und was er zu tun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiß zu sein – von einem alten Commis-Voyageur.* 1831.