République Algérienne Démocratique et Populaire
الجمهورية الجزائرية الديموقراطية الشعبية
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
وزارة التعليم العالي والبحث العلمي

المدرسة الوطنية للإعلام الآلي
(المعهد الوطني للتكوين في الإعلام الآلي سابقا)
École nationale Supérieure d'Informatique
ex. INI (Institut National de formation en Informatique)

**2nd Year of Higher Cycle**

**2021 - 2022**

# 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

May 5, 2022

# Contents

# List of Figures

# 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 [3], and circuit design [2] 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.

# Symbols and Notation

# Chapter 1

# General Concepts

## 1.1 Computational Complexity

Throughout this document, we will often find ourselevs 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 $\langle G, k\rangle$ wher $G$ is a graph and $k \in \mathbb{N}$. Such an instance is positive iff $G$ has a clique of size $k$.
- PAIR or the pairity problem is the problem $\langle \{0, 1\}^*, P\rangle$ where

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

**Example 1.2** (SAT)**.**
The *satesfiability 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 satesfiable iff it has a satisfying assignment or a *model*, that is, if its negation is not a tautology. For instance, the formula $\varphi$ below is satesfiable (by the assignment $x_1 = 0, x_2 = 0, x_3 = 1$ for example) while $\psi$ 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 A. We will use the notions of *algoorithms*, *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 behaviour 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 [1].

**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 probelms $X$ 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 probelms $X$ 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 $\mathsf{P}$ and $\mathsf{NP}$ exclusively.

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

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

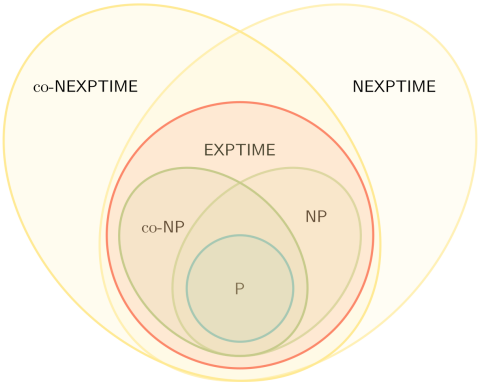


Figure 1.1: A Venn diagram of the classes defined so far.

However, it is exeedingly 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).

### 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 to decide it with a nondeterministic machine. We don't know if problems in $\mathsf{NP}$ are *strictly harder* thans those in $\mathsf{P}$ (even though we suspect that they are). This is hardly surprising given the vastness of $\mathsf{NP}$. In order to prove $\mathsf{P} = \mathsf{NP}$, one must find a polynomial time algorithm for *every* problem in $\mathsf{NP}$. And in order to prove that $\mathsf{P} \neq \mathsf{NP}$, one must prove that for at least one problem in $\mathsf{NP}$, *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 ($\mathsf{NP}$ and $\mathsf{EXPTIME}$ for example). The notio of complexity tries to simplify 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 that class) of the supposedly bigger class is actually in 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 attempts 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 quasi-order on decision problems) without too much difficulty. Furthermore, the relation *ple* can be shown to satisfy Proposition 1.1 which will become very importent once NP-completeness is defined.

**Proposition 1.1.**
*Let $X = \langle S_1, P_1 \rangle$ and $Y = \langle S_2, P_2 \rangle$ 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 are now ready to define the notion of 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 $P \cap NP-\text{hard} \neq \emptyset$, then $P = NP$*

*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 $\mathsf{P} = \mathsf{NP}$. □

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 to be a false one, since we know of many NP-hard and NP-complete problems. The first problem proved to be NP-complete is the problem SAT that we invoked in Example 1.2, and that was proven to be NP-complete in 1971.

**Theorem 1.3** (Cook Levin)**.**
*SAT is NP-complete.*

*Proof.*
The proof is twofold. First, we know that $\mathsf{SAT} \in \mathsf{NP}$, and second, that it is NP-hard. The first part is more or less trivial, the second by comparaison is very technical. It requires finding a polynomial reduction of every single problem in NP to SAT.

1. $\underline{\mathsf{SAT} \in \mathsf{NP}:}$

□

# Chapter 2

# Problem Statement

## 2.1 History

The first use of the term 'traveling salesman problem' in mathematical circles may have been in 1931-32, as we shall explain below. But in 1832, a book was printed in Germany entitled 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. Von einem alien Commis- Voyageur ("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').

Although devoted for the most part to other issues, the book reaches the essence of the TSP in its last chapter: '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 visiting a location twice ...' [Voigt, 1831; MiMer-Merbach, 1983].

# Chapter 3

# Branch and Bound

## 3.1 Motivation

The direct method as we have seen is, despite the simplicity of its implementation, unrealistically slow for even very small instances.

Faster exact algorithms exist, but none of them is polynomial since TSP is NP-complete. In fact, under the assumption $P \neq NP$, no polynomial solution exists.

Branch and Bound is one such algorithm that we will dedicate the rest of the chapter to.

## 3.2 The idea of Branch and Bound

The idea of Branch and Bound is to eliminate certain branches from the search space to decrease runtime.

This is done by computing a *lower bound* and *upper bound* for every branch, and then pruning branches that are garanted to be worse than the best known solution.

## 3.3 The implementation

# Appendix A

# Computability

# Appendix B

# Test

# Bibliography

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

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

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