Operations Research 2

Computer Engineering

Università degli Studi di Padova

Algorithms for the Symmetric Travelling Salesman Problem

Authors:

Bolzonello Enrico (ID. 2087644) enrico.bolzonello@studenti.unipd.it
Vendramin Riccardo (ID. 2087923) riccardo.vendramin.1@studenti.unipd.it

Supervisor:

Prof. Matteo Fischetti

Academic Year 2023/2024

Contents

List of Figures

List of Tables

1 Introduction

The Traveling Salesman Problem (TSP) stands as one of the most renowned and extensively studied combinatorial optimization challenges in the realm of computer science, operations research, and applied mathematics. At its core, the TSP revolves around finding the shortest possible route that a salesman must take to visit a given set of cities exactly once and then return to the original city. Despite its seemingly straightforward premise, the TSP's complexity escalates rapidly as the number of cities increases, leading to its classification as an NP-hard problem. Originating from the realm of logistics and operations, the TSP has garnered widespread attention due to its practical applications in various domains, including transportation planning, network design, and even DNA sequencing. Over the years, researchers have developed a plethora of algorithms, heuristics, and mathematical formulations aimed at tackling the TSP, each offering unique insights into its intricate nature and pushing the boundaries of computational optimization. This paper explores and compares different methods to solve the Traveling Salesman Problem and was developed during the Operations Research 2 course in the academic year 2023/24 at the University of Padua under the supervision of professor Matteo Fischetti.

This chapter introduces the TSP problem, it's history and formulations. Chapter 2 will discuss Heuristics, in particular the Nearest Neighbour heuristic and the 2-Opt refinment heuristic. Chapter 3 will discuss Metaheuristics, with a focus on Tabu Search and Variable Neighborhood Search.Chapter 4 is dedicated to exact models developed with CPLEX, which are Benders' Loop method and Branch&Cut. Chapter 5 considers Matheuristics, which are Hard Fixing and Local Branching. At the end, Chapter 6 conclude the report with summary and some observations.

1.1 Description

The problem asks the following question:

> Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

More formally, given a graph $G = (V, E)$ and $c : E \to \mathbb{R}^+$ function that assigns to each edge $\{i, j\} \in E$ the cost $c(e) = c_e$, the task is to find a Hamiltonian circuit[1], or tour, of minimum total cost.

In this paper, we consider an undirected complete graph and use Euclidean distance as the measure for cost.

[1] A Hamiltonian cycle (or Hamiltonian circuit) is a cycle that visits each vertex exactly once

1.2 History

Before mathematical formalization, the traveling salesman problem received significant attention due to its natural interpretation, as said by Dantzig, Fulkerson and Johnson [8]:

> It appears to have discussed informally among mathematicians at mathematics meetings for many years.

Detecting the origin is hard, but the first transcript mentioning the problem dates back to 1832 with a manual for the successful traveling salesman called *Der Handlungsreisende — wie er sein soll und was er zu thun hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiß zu sein — von einem 38 alten Commis-Voyageur*[2] and suggets five tours through Germany [23].

The formal beginning of studies on complete circuits can be attributed to Thomas Penyngton Kirkman. Although he believed he had originated the problem, he was actually inspired by William Rowan Hamilton's dodecahedron game, called *The Icosian Game*. In this game, a player has to place the whole or part of a set of twenty numbered pieces upon the points, as in Figure 1, to always proceed along the lines and also to fulfil other conditions. In particular, the first problem was to find a circuit passing just once through each vertex of the graph. There was another version of this game, involving a solid dodecahedron, and known as *The Traveller's Dodecahedron*, where each vertex represented important places [4].

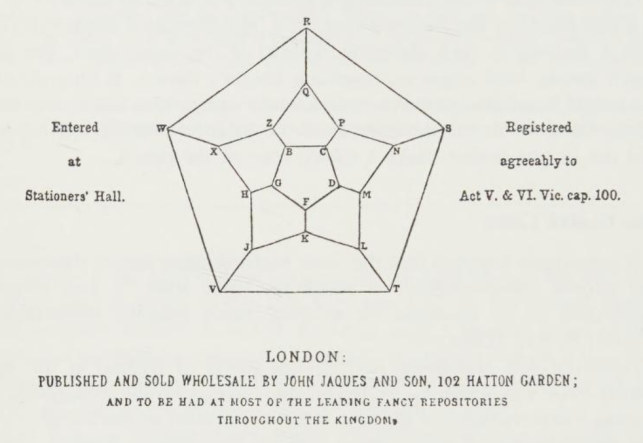


Figure 1: Diagram of the Icosian Game

The general form of the TSP has been first studied by Karl Menger, who defined the problem, considers the nearest-neighbour heuristics and observes the non-optimality of the greedy approach.

In the 30s, while solving a school bus routing problem, Flood mentioned connections between TSP and Hamiltonian games and paths in graphs. Popularity of the problem increased after the interest of the RAND Corporation,

[2] "The traveling salesman — how he should be and what he has to do, to obtain orders

which offered a prize "for a significant theorem bearing on the TSP". RAND's researcher Julia Robinson report is, as far as it is known, the earliest mathematical reference using the term 'traveling salesmen problem'.
While US researches tackled the TSP as an optimization problem, in the other side of the world statistician P. C. Mahalanobis took on the problem from a different mathematical point of view. He made a connection to the TSP while carrying out sample surveys of jute plantations, since one of the major component in the cost of the operation was to move men and equipment between sample areas. Thanks to its statistical background, he focused on making statistical estimates of the lengths of the optimal tours. His research, while not optimal, will serve as a baseline for future development and lead to the Beardwood–Halton–Hammersley Theorem [2], which implies that as n gets large, the distribution of tour lengths will spike around a particular number called β, the TSP constant.

Significant progress was made by Dantzig, Fulkerson and Johnson [8] in a paper which introduced several methods for solving the TSP and shows the importance of *cutting planes* for combinatorial optimization. The paper will be called "one of the principal events in the history of combinatorial optimization" [17].

In 1972, Richard M. Karp showed that the Hamiltonian cycle problem was NP-complete, which implied the NP-hardness of TSP.

In 1976, Christofides proposed what is now known as the Christofides algorithm which yields a solution that is at most $3/2$ times worst than the optimal solution, result unbeaten until 2011 when a slightly better algorithm (but unfeasible to implement in practice) was proposed.

As of today, the largest instance with a known optimal solution is a tour through 85,900 cities, a significant advancement from the 49-city problem solved in 1954. This research was conducted by Applegate, Bixby, Chvátal, Cook, Espinoza, Goycoolea, and Helsgaun [1].

and to be sure of a happy success in his business — by an old traveling salesman"

1.3 Formulations

We will now discuss the various formulations of the standard Traveling Salesman Problem (TSP): the initial one being the classical formulation, followed by the compact formulations.

1.3.1 Dantzig-Fulkerson-Johnson (DFJ)

This is the classical formulation of the standard TSP due to Dantzing, Fulkerson and Johnson [8]:

$$
\begin{aligned}
\min \quad & \sum_{i=1}^{n}\sum_{i=1}^{n} c_{ij}x_{ij} \\
\text{s.t.} \quad & \sum_{i=1}^{n} x_{ij} = 1, \quad j = 1, ..., n \\
& \sum_{j=1}^{n} x_{ij} = 1, \quad i = 1, ..., n \\
& \sum_{i\in S}\sum_{j\in S} x_{ij} \leq |S| - 1, \quad S \subset V : S \neq \emptyset \\
& x_{ij} \in \{0, 1\}, \quad i, j = 1, ..., n
\end{aligned}
\tag{1}
$$

where x_{ij} is equal to 1 if and only if arc (i, j) is in the optimal tour. A problem of this formulation is that the subtour elimination constraints are exponential in number.
The solver generates solutions containing subtours when the subtour elimination constraints are not applied, as depicted in Figure 2.

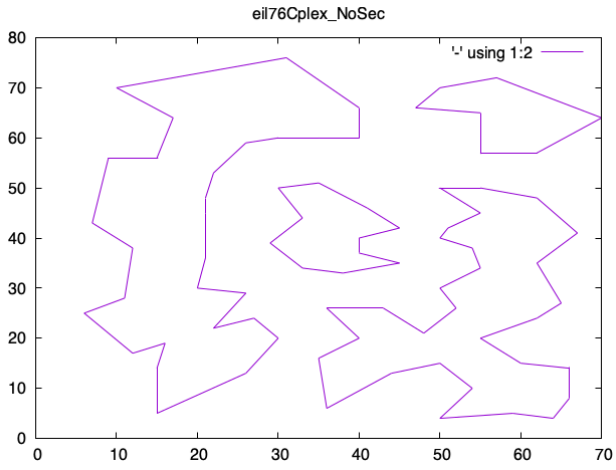


Figure 2: Plot of the optimal solution of the Eil76 dataset without Subtour Elimination Constraints (SEC)

1.3.2 Miller-Tucker-Zemlin (MTZ)

The first compact formulation we consider is the following, due to Miller, Tucker and Zemlin [18]:

$$\begin{aligned}
\min \quad & \sum_{i=1}^{n}\sum_{i=1}^{n} c_{ij}x_{ij} \\
\text{s.t.} \quad & \sum_{i=1}^{n} x_{ij} = 1, \quad j = 1, ..., n \\
& \sum_{j=1}^{n} x_{ij} = 1, \quad i = 1, ..., n \\
& u_i - u_j + (n-1)x_{ij} \leq n-2, \quad i, j = 2, ..., n \\
& x_{ij} \in \{0, 1\}, \quad i, j = 1, ..., n
\end{aligned} \tag{2}$$

where $u_i, i = 2, ..., n$ is an arbitrary real number representing the order of vertex i in the optimal tour. The original formulation is without any bound on variables u_i; simple bounds have been introduced later but it has been shown to not affect the LP bound and increasing computing time [22]. The MTZ formulation is compact, with only $O(n^2)$ variables and $O(n^2)$ constrains, but its LP relaxation yields an extremely weak lower bound, much weaker than that of the DFJ formulation [20].

1.3.3 Time-Staged (TS)

Historically, the next compact formulation is the Time-Staged, independently proposed by Vajda [9] and Houck, Picard and Vemuganti [15].
Let r_{ij}^k be a binary variable taking the value 1 if and only if the edge $\{i, j\}$ is the k-th edge to be traversed in the tour in the direction from i to j.

$$\begin{aligned}
\min \quad & \sum_{i=2}^{n} c_{1i}r_{1i}^1 + \sum_{k=2}^{n-1}\sum_{i,j=2}^{n} c_{ij}r_{ij}^k + \sum_{i=2}^{n} c_{i1}r_{i1}^n \\
& \sum_{j=1}^{n} r_{1j}^1 = 1 \\
& \sum_{j=2}^{n} r_{j1}^n = 1 \\
& \sum_{k=1}^{n-1}\sum_{j\neq i} r_{ji}^k = 1, \quad 2 \leq I \leq n \\
& \sum_{j\neq i} r_{ji}^k = \sum_{j\neq i} r_{ji}^{k+1}, \quad 2 \leq i \leq n, 1 \leq k \leq n-1 \\
& r_{ij}^k \in \{0, 1\}, \quad 1 \leq i, j, k \leq n, i \neq j
\end{aligned} \tag{3}$$

It has $O(n^3)$ variables and $O(n^2)$ constraints. The associated lower bound is intermediate in strength between the MTZ and DFJ bounds [20].

1.3.4 Gavish-Graves (GG)

GG [12] is a single-commodity flow formulation, meaning that the salesman carries $n-1$ units of a commodity when he leaves node 1 and delivers 1 unit of this commodity to each other node.

$$\begin{aligned}
\min \quad & \sum_{i=1}^{n}\sum_{i=1}^{n} c_{ij}x_{ij} \\
\text{s.t.} \quad & \sum_{i=1}^{n} x_{ij} = 1, \quad j = 1, ..., n \\
& \sum_{j=1}^{n} x_{ij} = 1, \quad i = 1, ..., n \\
& \sum_{j=1}^{n} g_{ji} - \sum_{j=2}^{n} g_{ij} = 1, \quad i = 2, ..., n \\
& 0 \leq g_{ij} \leq (n-1)x_{ij}, \quad i, j = 1, ..., n, j \neq i \\
& x_{ij} \in \{0, 1\}, \quad i, j = 1, ..., n
\end{aligned} \tag{4}$$

The GG formulation has $O(n^2)$ variables and $O(n)$ constraints. The associated lower bound is intermediate in strength between the MTZ and TS bounds [14].

1.3.5 Multi-Commodity Flow (MCF)

In this formulation, we image that the salesman carries $n-1$ commodities, one unif of each for each customer. Define the additional continuous variable f_{ij}^k, representing the amount of the kth commodity passing directly from i to j.

$$\begin{aligned}
\min \quad & \sum_{i=1}^{n}\sum_{i=1}^{n} c_{ij}x_{ij} \\
\text{s.t.} \quad & \sum_{i=1}^{n} x_{ij} = 1, \quad j = 1, ..., n \\
& \sum_{j=1}^{n} x_{ij} = 1, \quad i = 1, ..., n \\
& \sum_{j=1}^{n} g_{ji} - \sum_{j=2}^{n} g_{ij} = 1, \quad i = 2, ..., n \\
& 0 \leq f_{ij}^k \leq x_{ij}^k, \quad k = 2, ..., n; \{i, j\} \subset \{1, ..., n\} \\
& \sum_{i=2}^{n} f_{1i}^k = 1, \quad k = 2, ..., n \\
& \sum_{i=1}^{n} f_{ik}^k = 1, \quad k = 2, ..., n \\
& \sum_{i=1}^{n} f_{ij}^k - \sum_{i=2}^{n} f_{ji}^k = 0, \quad k = 2, ..., n, j \in \{2, ..., n\} \backslash \{k\} \\
& x_{ij} \in \{0, 1\}, \quad i, j = 1, ..., n
\end{aligned} \tag{5}$$

The MCF formulation has $O(n^3)$ variables and $O(n^3)$ constraints, but its lower bound is equal to the DFJ bound [20].

1.4 Development Stack

This section outlines the hardware and software utilized for the project.

1.4.1 Hardware

All our algorithms are run on our personal computers, namely:

- Apple MacBook Air 2020: Chip M1, 8GB RAM, 256GB SSD
- Lenovo Legion 5: AMD Ryzen 4700H, RAM 16GB, Nvidia GeForce RTX2060 6GB

1.4.2 Software

Within this section, we provide a quick overview of the external software and libraries integrated into our project. We elaborate on their roles, functionalities, and contributions to our overall development endeavor.

TSPLIB
TSPLIB [21] is a library of sample instances for the TSP (and related problems) from various sources and of various types. In our application, we focused on the data for the symmetric TSP problem with Euclidean distance, since it is the only type currently supported.

GnuPlot
GnuPlot[3],a versatile plotting utility widely employed in scientific and engineering domains, was used to visualize the obtained solution, serving a valuable tool especially in the debugging phase.

IBM ILOG CPLEX
To solve the Traveling Salesman Problem (TSP) exactly, we employ CPLEX [6], with which we can formulate the TSP as an optimization model and find the optimal solution using its advanced and well debugged algorithms and heuristics. This ensures that we obtain the best possible solution to the TSP, guaranteeing accuracy and reliability in our results. CPLEX is a high-performance optimization software package developed by IBM. It is designed to solve complex linear programming (LP), mixed integer programming (MIP), and quadratic programming (QP) problems efficiently. The CPLEX Model is a representation of the optimization problem formulated using the CPLEX Optimization Programming Language (OPL) or through APIs available for various programming languages, including C. The model specifies the objective function to be optimized, along with the constraints and variables of the problem.

[3] `http://www.gnuplot.info/`

Concorde
Concorde[4] is a computer code for the symmetric traveling salesman problem (TSP) and some related network optimization problems. The code is written in the ANSI C programming language and it is available for academic research use. It supports QSopt and CPLEX as an LP solver. The code dates back to 2003, when CPLEX was still in version 8.0, while at the time of writing this report the version 22.1.1 is used. For the purpose of the course, we used a minimal version of Concorde with the functions we needed.

[4] `https://www.math.uwaterloo.ca/tsp/concorde.html`

1.5 Code Structure

In this section we describe how the code base is structured, each section is a folder and paragraphs are the individual files. The code is available at the following GitHub repository: `https://github.com/enricobolzonello/TravellingSalesmanOptimization`

1.5.1 Algorithms

heuristics Contains utilities and functions for the following algorithms:

- `Greedy`, described in Section 2.1
- `Greedy_iterative`, described in Section 2.1.1
- `Greedy_2opt`, which is greedy iterative combined with 2opt refinment algorithm, described in Section 2.2.1
- `ExtraMileage`

refinment Contains utilities and functions for the following algorithms:

- `2opt`, described in Section 2.2.1

metaheuristic Contains utilities and functions for the following algorithms:

- `TabuSearch`, described in Section 3.1
- `VNS`, described in Section 3.2

cplex_model Contains general utilities and functions to work with CPLEX and the following algorithms:

- `Nosec`, basic tsp model without subtour elimination constraints and solved with CPLEX MIP solver
- `BendersLoop`, described in Section 4.1
- `BranchAndCut`, described in Section 4.2

matheuristics Contains utilities and functions for the following algorithms:

- `HardFixing`, described in Section 5.1
- `LocalBranching`, described in Section 5.2

1.5.2 Utils

errors Contains utilities and functions to manage verbosity levels, error types and logging

plot Contains utilities and functions to plot TSP solution

utils Contains general utilities used throughout the code in different situations

1.5.3 tsp

Contains settings, parameters and general functions about TSP instance and its initialization

1.5.4 mincut

Contains a reduced version of Concorde, with utilities and function to address the generation of cuts for fractionary solutions

1.5.5 main

Contains the entry point of our solver and manage the algorithm execution

2 Heuristics

Given its combinatorial nature, the TSP is classified as NP-hard, meaning that as the number of cities increases, the computational effort required to solve the problem grows exponentially, so finding an exact solution for large instances becomes impractical. To address this, heuristic algorithms offer approximate solutions that are computationally feasible, trading off optimality for efficiency. Heuristics do not guarantee the optimal solution but aim to find good enough solutions within reasonable time frames.

In this section we describe a constructive heuristic, called in literature Nearest Neighbor, and a simple refinement heuristic, named 2opt.

2.1 Nearest Neighbor

One of the first and simplest heuristics is the Nearest Neighbor heuristic, which is a $O(n^2)$-greedy algorithm. Given a starting node, at each step the algorithm greedily chooses the next unvisited neighbour node which has the smallest cost. This step repeats until there are not any nodes to visit, and finally it will close the path connecting the last node to the starting one.
Pseudocode for the described algorithm is in Algorithm 1.

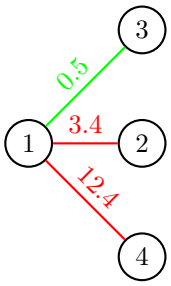


Figure 3: One iteration of NN, in this case node 3 is chosen since it has lowest cost

Algorithm 1 Nearest Neighbor heuristic

Input: set of nodes V with $|V| = N$, starting node $s \in V$
Output: set of N nodes forming an Hamiltonian tour with a cost c

procedure NEARESTNEIGHBOR
  S $\leftarrow \{s\}$ ▷ set of visited nodes
  curr $\leftarrow s$
  cost $\leftarrow 0$
  while there are nodes to visit **do**
    next_node $\leftarrow \arg\min_{v \in V \setminus S} c_{curr,v}$
    add edge to path
    visited $\leftarrow$ visited $\cup$ next_node
    cost $\leftarrow$ cost $+c_{curr,next\_node}$
    curr $\leftarrow$ next_node
  cost $\leftarrow$ cost $+c_{curr,s}$ ▷ last edge

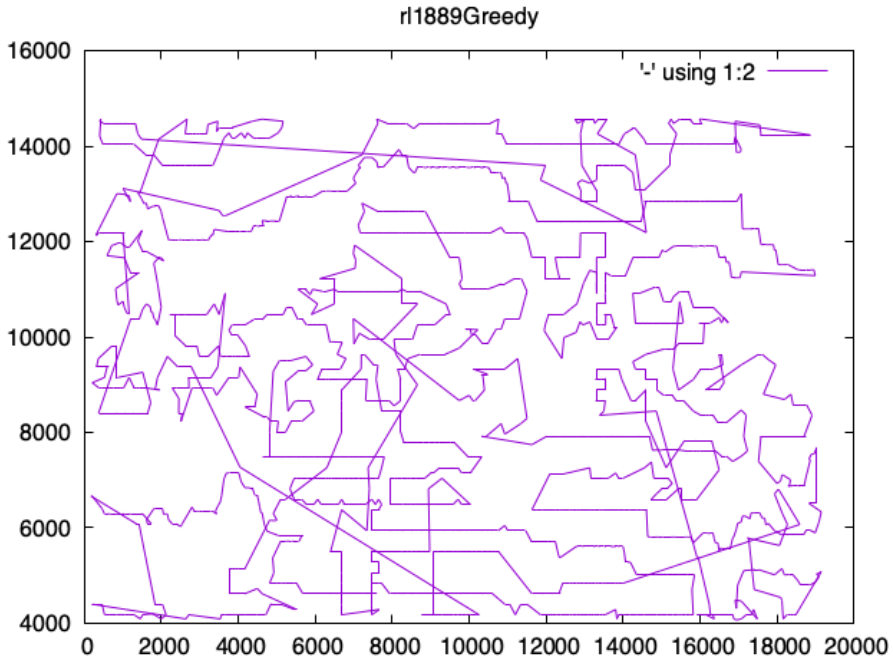


Figure 4: Plot of the solution found by the Nearest Neighbor heuristic on the rl1889 dataset

For this algorithm we are guaranteed that $NN(I)/OPT(I) \leq 0.5 \cdot (\lceil \log_2 N \rceil + 1)$ [16], but it has two major problems:

- The first problem is that since it makes a choice only on the next nearest node, it often happens that the last connection, which closes the tour, may be very long and it obtains a bad solution
- In the case of the Euclidean TSP, it is known that the optimal solution can't have two edges crossing, but the NN heuristics produces solutions with lots of crossings, as can be seen in Figure 4.

To address both issues, we will explore methods to refine these initial solutions in Section 2.2.

2.1.1 All Nearest Neighbor

A natural extension of the nearest neighbor heuristic is the all nearest neighbor, which is just running the nearest neighbor heuristic for all nodes at starting point and picking the best one.

2.2 Refinements Heuristics

Starting from the solution obtained by other heuristic, this group of algorithms make small changes to it to obtain a possible better solution. Note the quality of the obtained solution depends also on the quality of the starting solution, so a good starting solution is needed.

2.2.1 2-opt algorithm

The simple algorithm considered is the 2-opt algorithm, a local search algorithm proposed in 1958 by Croes [7]. The main idea is to select two edges that, if they were swapped, would produce a shorter tour. Trying all pairs of edges is unfeasible, but it is known that in the Euclidean space the optimal tour has no edges that cross.

A 2-opt move consists in finding a pair of nodes (i, j) for which changing their outgoing edge with an new one will reduce the cost of the tour, meaning that we replace $(i, succ_i)$ with (i, j) and $(j, succ_j)$ with $(succ_i, succ_j)$. The gain offered by such a move is calculated as the difference between the old edges and the new ones:

$$\Delta = \underbrace{(c(i,j) + c(succ_i, succ_j))}_{\text{swapped cost}} - \underbrace{(c(i, succ_i) + c(j, succ_j))}_{\text{original cost}} \tag{6}$$

If $\Delta < 0$, it means there is a crossing. Visually, what happens is illustrated in Figure 5.

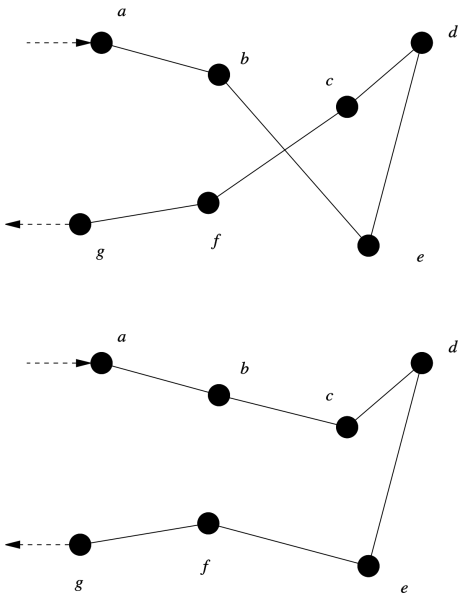


Figure 5: Exchange step of 2opt algorithm
Above: before 2opt move
Below: after 2opt move
Source: `https://en.wikipedia.org/wiki/2-opt`

The algorithm continues until $\Delta \geq 0$, meaning that that are no more moves

that improve the solution and a local optimal solution has been obtained. Pseudocode for the algorithm can be found at Algorithm 2.

Algorithm 2 2-opt pseudocode

Input: set of nodes V with $|V| = N$, valid solution for the TSP
Output: set of N nodes forming an Hamiltonian tour with a cost c

```
1:  procedure 2-OPT
2:      while Δ < 0 do
3:          Δ ← 0                                              ▷ Find best swap
4:          for a ← 0 to n − 1 do
5:              for b ← a + 1 to n do
6:                  temp ← (c_{i,j} + c_{succ_i,succ_j}) − (c_{i,succ_i} + c_{j,succ_j})
7:                  if temp < Δ then
8:                      Δ ← min(Δ, temp)
9:                                                             ▷ Execute best swap
10:         if Δ < 0 then
11:             replace (i, succ_i) with (i, j) and (j, succ_j) with (succ_i, succ_j)
12:             reverse path from j to succ_i
13:             cost ← cost +Δ
```

We chose to run the refinement algorithm on every solution computed by running All-NN, and not only on the final one. Compared to the plot obtained by the Nearest Neighbour heuristic alone in Figure 4, the solution obtained by adding the 2-OPT algorithm is more regular and without crossings, as can be seen in Figure 6.

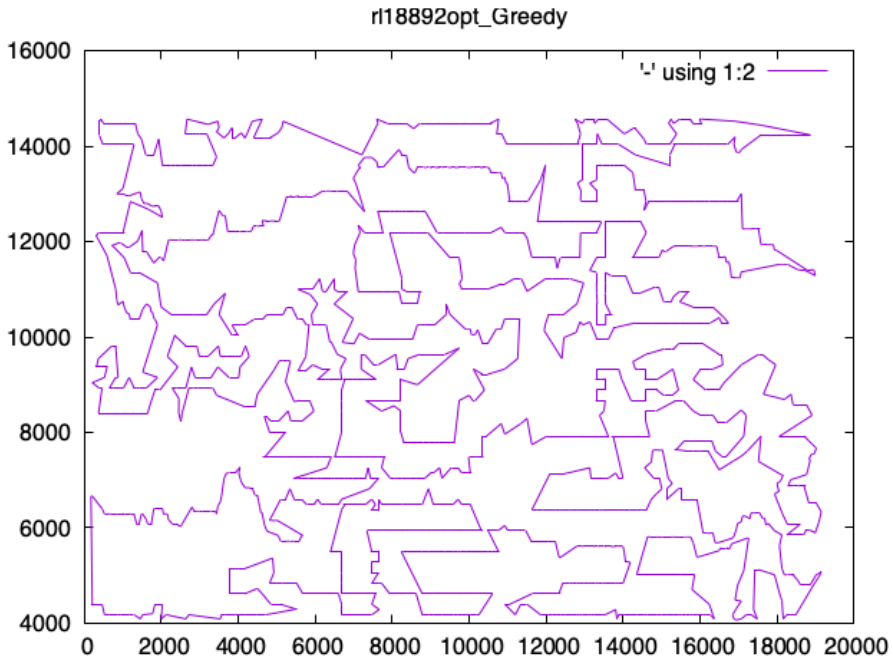


Figure 6: Plot of the solution found by All-NN + 2-OPT on the rl1889 dataset

2.3 Comparative Analysis

The greedy nearest neighbor solution from a chosen node is significantly far from the optimal so it is useless to use as final solution and even as a bound for other algorithms as well. Trying different runs over all or some nodes leads to a better final solution, but the outcome is still not good enough. On the other side, 2opt refinement is a very powerful tool, which improves initially bad solutions by a significant amount, as can be seen in Figure 7, with more than half of the instances with the iterative greedy improving by around 20%.

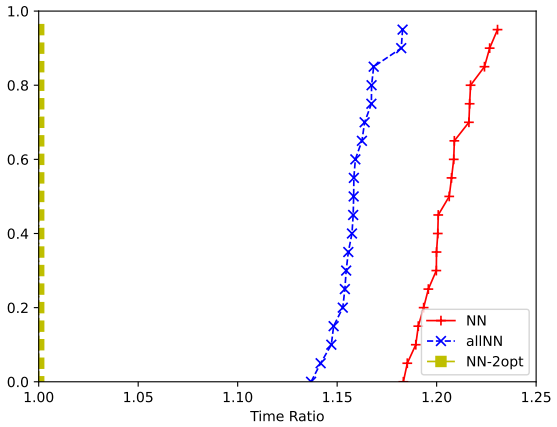


Figure 7: Performance profile of Constructive Heuristics

instance	NN	allNN	NN-2opt
0	415100.00	406038.00	343459.00
1	399055.00	388713.00	336684.00
2	418197.00	390205.00	339833.00
3	405185.00	389304.00	342428.00
4	418674.00	395263.00	341317.00
5	405948.00	393241.00	339470.00
6	404114.00	384501.00	336779.00
7	407487.00	394560.00	341431.00
8	410827.00	391261.00	337790.00
9	407231.00	387438.00	334735.00
10	404432.00	391958.00	339996.00
11	407167.00	391239.00	337525.00
12	412960.00	394896.00	339312.00
13	408388.00	395660.00	340368.00
14	412102.00	403235.00	340891.00
15	406509.00	388464.00	336660.00
16	411116.00	392057.00	335868.00
17	401266.00	386536.00	336979.00
18	408250.00	397300.00	340047.00
19	409549.00	398087.00	341061.00

Table 1: Comparison between heuristics results

3 Metaheuristics

Metaheuristic algorithms attemp to find the best possible solution of an optimization problem by evaluating possible solutions and perform a series of operations on them in order to find different, better solutions. Three classes can be delineated based on the way in which solutions are manipulated:

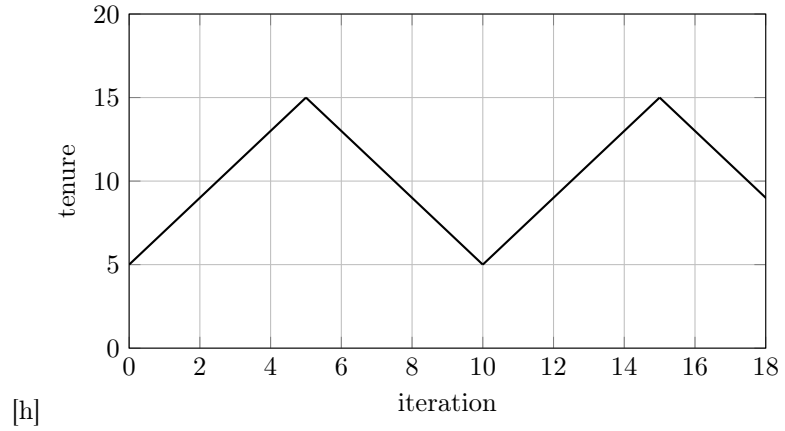
- **Local Search** (LS), finds good solutions by iteratively making changes to the incumbent solution. In each iteration, the incumbent solution is replaced by a solution in its neighborhood, which is the set of solutions that can be derived by a single move. In this category falls both Tabu Search and Variable Neighborhood Search.
- **Constructive**, construct solutions rather than improving complete ones. One example of this is the GRASP algorithm.
- **Population-based**, find good solutions by iteratively selecting and then combining existing solutions from a set, called population. One example in this category is the Genetic algorithms.

3.1 Tabu Search

Tabu Search is a metaheuristics technique based on local search proposed in 1986 by Glover [13]. Starting from a valid solution of the TSP, it operates by iteratively exploring the solution space performing changes in the solution while keeping track of moves that are considered forbidden in a structure called *Tabu List*. This prevents the algorithm from getting stuck in local optima allowing moves which increase the cost of the solution, and encourages exploration of different regions of the solution space. The Tabu Search Algorithm offers a powerful approach for refining solutions to the Travelling Salesman Problem: by intelligently exploring the solution space while avoiding previously explored regions, it can effectively find high-quality solutions.

The performance of the algorithm are strictly dependant on the tenure, which is the size of the Tabu List, that could be managed in different ways. In our implementation, there are different policies:

- *Fixed*
- *Size dependent*, the average tenure is calculated as the average between a maximum and minimum tenure, which are fractions of the number of nodes. The fractions are fixed respectively to 0.25 and 0.125 [24].
- *Random*, again with a maximum and minimum tenure
- *Linear*, starting from the minimum tenure, at each iteration it grows until reaching the maximum. At this point, it will start to descend until the minimum. This process repeats until the end of the algorithm.
- *Sinusoidal*, increasing and decreasing the tenure following a sinusoidal function between a minimum and a maximum set through a percentage of the number of nodes.



[h]

Figure 8: Representation of the Linear Policy

As shown in the performance profile in Figure 9, the fixed-size tabu list performs the best. However, the improvement is minimal and does not lead to a significant impact. The linear and sinusoidal tenure turned out to be the worst policies, one reason could be the hyper-parameters used to set the bounds as a percentage of the nodes.

Another crucial decision is the move used to explore the solution space, which, in our implementation, is a 2-opt move which skips nodes in the Tabu list. As a side note, the tabu list is implemented as a linear array of $n = |V|$ nodes, where element $t[i]$ stores the last iteration during which node i was considered.

Algorithm 3 Tabu Search Algorithm for TSP

Input: set of nodes V with $|V| = N$, valid solution for the TSP
Output: set of N nodes forming an Hamiltonian tour with a cost c

1: **procedure** TabuSearch
2: Initialize a *best_solution* to a starting solution S
3: Initialize a tabu list TL with a capacity L
4: **while** stopping criterion not met **do**
5: Generate a new solution S' from S avoiding moves in TL
6: Update TL with S'
7: **if** S' is better than S **then**
8: Update *best_solution*
9: Update S to S'

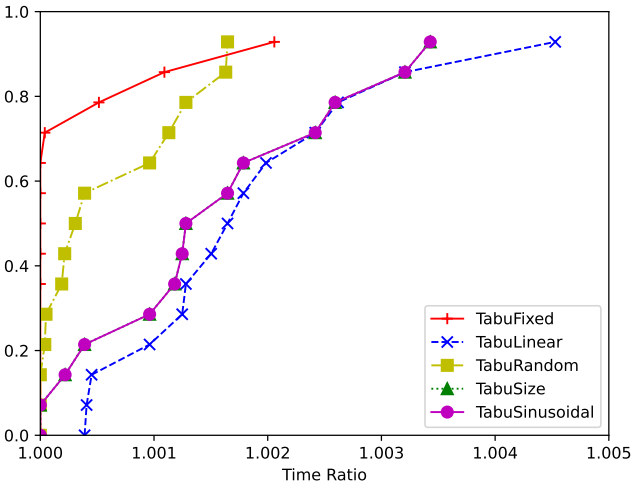


Figure 9: Performance profile of different Tabu Policies

	Fixed	**Linear**	**Random**	**Size**	**Sinusoidal**
rl1304.tsp	265948.00	266068.00	266005.00	266006.00	266006.00
pr1002.tsp	269220.00	269925.00	269659.00	269918.00	269918.00
vm1748.tsp	351069.00	351696.00	351135.00	351696.00	351696.00
rl1323.tsp	279990.00	280093.00	279990.00	279979.00	279979.00
u1060.tsp	236202.00	236557.00	236469.00	236481.00	236481.00
rl1889.tsp	330593.00	330972.00	329914.00	330972.00	330972.00
vm1084.tsp	249518.00	249557.00	249246.00	249557.00	249557.00
u1817.tsp	59972.00	60117.00	59975.00	60117.00	60117.00
nrw1379.tsp	59299.00	59356.00	59356.00	59356.00	59356.00
pcb1173.tsp	60137.00	60214.00	60214.00	60214.00	60214.00
fl1577.tsp	22745.00	22848.00	22752.00	22823.00	22823.00
fl1400.tsp	20543.00	20551.00	20551.00	20551.00	20551.00
d1291.tsp	52423.00	52500.00	52396.00	52396.00	52396.00
d1655.tsp	64982.00	65089.00	65089.00	65089.00	65089.00

Table 2: Policies for tabu search results

3.2 Variable Neighborhood Seach (VNS)

Variable Neighborhood Search (VNS) is a metaheuristic algorithm that explores distant neighborhoods and moves there if there is an improvement. Once in the neighborhood, it applies a local search algorithm like 2-opt to find the local optima. It is based on the following observations:

- a local optimum relative to one neighborhood is not necessarily a local optimum for another neighborhood
- a global optimum is a local optimum with respect to all neighborhoods
- many problems have local optima that are relatively close to each other

This properties suggest that the main components of the algorithm are an *improvement phase*, used to possibly improve a given solution, and a *kick phase*, to hopefully resolve local minima traps. In our implementation, the improvement phase is implemented as the 2opt refinment algorithm described in Section 2.2.1 and the kick as a 3opt on three random edges.
The outline of the algorithm is as follows:

Algorithm 4 Variable Neighborhood Search (VNS)

Input: set of nodes V with $|V| = N$
Output: set of N nodes forming an Hamiltonian tour with a cost c

```
1: procedure VNS
2:     construct initial solution with any heuristic
3:     while stopping criterion not met do
4:         local search with 2opt
5:         if new solution < best solution then
6:             save new solution in best solution
7:         apply kick a random number of times
```

At each iteration, the kick is applied a random number of times between 2 and 10. This functionality enables the algorithm to allow worse solution to then hopefully find a better one, thus breaking the local optimum. This behaviour can be seen graphically in Figure 10.

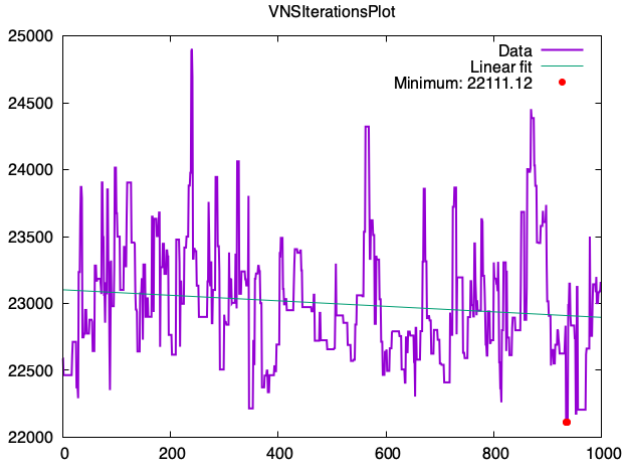


Figure 10: Cost of the solution during VNS iterations (cut to 1000 iterations)

3.3 Comparative Analysis

	Optimal	VNS	GAP %	TabuFixed	GAP %
rl1304.tsp	252948,00	261789,00	3,50	265948,00	5,14
pr1002.tsp	259045,00	267674,00	3,33	269220,00	3,93
vm1748.tsp	336556,00	350472,00	4,13	351069,00	4,31
rl1323.tsp	270199,00	277394,00	2,66	279990,00	3,62
u1060.tsp	224094,00	232071,00	3,56	236202,00	5,40
rl1889.tsp	316536,00	332973,00	5,19	330593,00	4,44
vm1084.tsp	239297,00	248764,00	3,96	249518,00	4,27
u1817.tsp	57201,00	59329,00	3,72	59972,00	4,84
nrw1379.tsp	56638,00	59562,00	5,16	59299,00	4,70
pcb1173.tsp	56892,00	59595,00	4,75	60137,00	5,70
fl1577.tsp	22249,00	22696,00	2,01	22745,00	2,23
fl1400.tsp	20127,00	20451,00	1,61	20543,00	2,07
d1291.tsp	50801,00	52599,00	3,54	52423,00	3,19
d1655.tsp	62128,00	64512,00	3,84	64982,00	4,59

Table 3: Comparison between metaheuristics and optimal solution costs

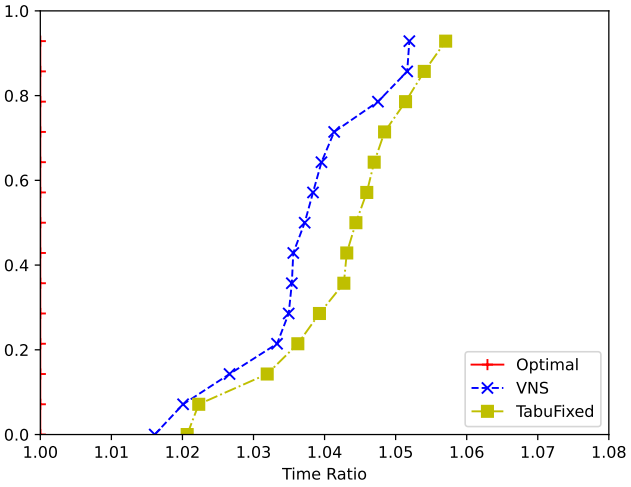


Figure 11: Performance profile between Metaheuristics and optimal solution costs

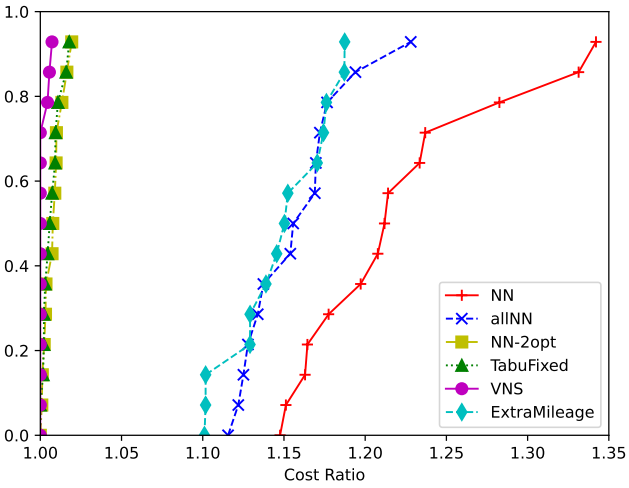


Figure 12: Performance profile between different heuristics and metaheuristics

As shown in the performance profile in Figure 11 and the results in Table 3, Variable Neighborhood Search (VNS) consistently outperforms Tabu Search. The gap relative to the optimal solution is around 3% for VNS and just over 4% for Tabu Search, indicating that both algorithms perform well, if we consider a good heuristic solution a solution within the margin of the 5% from the optimal.

In Figure 12 we can see a comparison between algorithm mentioned so far, tested on a set of 20 randomly generated instances with 120 seconds of time limit. The VNS turned out to be the best, followed by Tabu Search with fixed tenure and Nearest Neighbor with 2opt refinement.

4 Exact models

Within the realm of solving the Traveling Salesman Problem (TSP), exact methods emerge as instruments for finding the optimal solution. We will present several algorithms developed leveraging the robust and optimized basic solver of CPLEX. Since in the definition of CPLEX model we omitted subtour elimination contraints, CPLEX basic solver does not guarantee a valid TSP solution. A first approach to improve its efficacy is Benders' Loop method and further enhance it adding a patching heuristics. Then we will move on the branch and cut technique, employing CPLEX callbacks to refine our approach iteratively.

4.1 Benders' Loop Method

Named after Jacques F. Benders, Benders' Loop Method is a technique to overcome the problem of the exponential number of Subtour Elimination constraints. The name has been given informally since we use the main loop of the much more complex Benders Decomposition method [3], a general technique in mathematical programming for solving problems that have a special block structure. Benders' Loop Method begins by solving a relaxed version of the TSP, where Subtour Elimination Constraints (SECs) are missing, with the CPLEX MIP solver. This initial model does not produce feasible solutions, since it allows subtours as discussed in Section 1.3.1.
To cope with this, for each connected component of the current solution, some constraints are added to the model to tighten the relaxation and the new solution is computed. This process continues iteratively until only one connected component is obtained, which means an optimal feasible solution has been found, or until the time limit is reached. The algorithm just described above can be found at Algorithm 5

Algorithm 5 Bender's Loop method

Input: CPLEX MIP model
Output: set of N nodes forming an Hamiltonian tour with a cost c

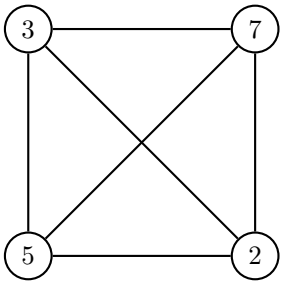
```
1: procedure BendersLoop
2:     initialize CPLEX model
3:     while stopping criterion not met do
4:         solve with MIP solver
5:         if number of components is 1 then
6:             break the loop
7:         add SEC
```

The constraint added is the following:

$$\sum_{e \in E(S)} x_e \leq |S| - 1 \tag{7}$$

where S is a subtour, graphically an example can be found in Figure 13. With our datasets, it adds on average approximately $\frac{2}{5} \cdot n$ constraints, as evidenced by the data presented in Table 4, which is much better than the exponential

number of the complete formulation.



$$x_{37} + x_{27} + x_{25} + x_{35} + x_{23} + x_{52} \leq 3$$

Figure 13: Example of a Subtour Elimination Constraint

4.1.1 Patching heuristic

A notable issue with the method is that each iteration of Benders' Loop, except the last, generates an invalid solution, potentially composed of multiple independent components. This is a problem since if the algorithm runs out of time or the execution is interrupted no solution will be given to the user. To fix it, we can apply iteratively a repairing function, also called *patching heuristic*, to have a valid solution at each iteration.

The concept is to pick two edges which belong to different independent components, break them and connect the nodes of the different components. In this way, the number of independent components decreases by 1. More formally, let k_1, k_2 be two independent sub-components of the solution and let $V_{k_1}, V_{k_2} \subset V$ be its vertices. Let (x, y) be an edge of component k_1 and (u, v) an edge of component k_2. To connect the two components we need to break the two edges and connect them with two edges (x, u) and (y, v). At the end, 2-OPT is applied, since we may have generated a crossing.

To choose what components k_1, k_2 to patch and what edges to break, we loop through all possibilities and we pick the pair that gives the best improvement, with the same method done in 2-OPT. The technique of merging two connected components is also called *gluing*, and repeating this operation until obtaining a single connected component assures to have a feasible TSP solution, even if not optimal.

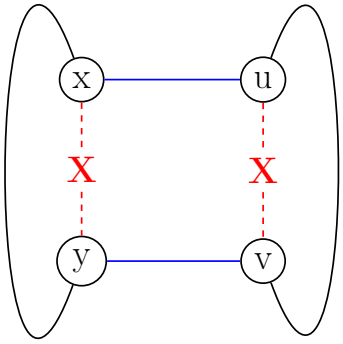


Figure 14: Representation of the patching heuristic

As shown in Figure 15, the Patching heuristic consistently reduced the time required to find the optimal solution. Additionally, it ensured that a valid solution was obtained even if the algorithm reached the time limit. The results above have been obtained running the two configurations with 20 random generated instances with 300 nodes.

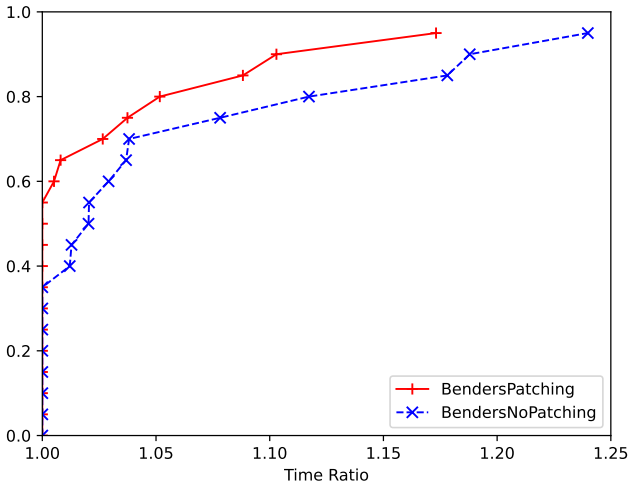


Figure 15: Performance profile between Benders with and without the patching heuristic

dataset	#SEC	nnodes
pr439	59	439
ch150	84	150
d198	69	198
ch130	48	130
kroA100	57	100
berlin52	8	52
eil51	16	51
kroB100	46	100
random	10	30
random	102	300
kroA150	83	150
pr299	89	299
kroB150	50	150
pr107	78	107
kroA200	64	200
a280	51	280
pr264	54	264
pr76	45	76
d493	55	493
pr136	49	136
ch150	59	150
bier127	36	127
kroC100	43	100
pr124	74	124
eil101	18	101
kroB200	79	200
pr226	125	226
pr144	143	144
kroD100	42	100

Table 4: number of SECs added in each dataset

4.2 Branch and Cut

One of the most famous and effective exact techniques to solve MIP problems is the Branch&Cut, proposed in 1991 by Rinaldi and Padberg [19].
This method combines branch-and-bound with cutting planes. The initial problem is relaxed to a Linear Programming (LP) problem, which can be solved optimally using known algorithms such as the simplex algorithm. However, the solution obtained from this relaxation is likely to be non-integer. To address this, a cutting plane algorithm is applied to the variables that are supposed to be integer. The problem is then divided into multiple subproblems, and the process repeats. However, this can lead to an excessively large tree that is impractical to process. Therefore, it is essential to prune the tree based on specific rules:

- solution is integer
- the relaxation is infeasible
- the solution of the relaxation (called upper bound) is greater than the best integral solution found (called lower bound)

Fortunately, we don't need to implement this entire process from scratch. CPLEX employs the Branch and Cut technique in its MIP solver, `CPXmipopt`. This solver is also enhanced with heuristics to close the optimality gap more quickly and different types of cuts.
CPLEX also allows to monitor and control the execution of the solver thanks to *callbacks*, which are user-functions that are called by CPLEX when the solver is in the specified context[5]. For our purposes we used two context:

- `CPX_CALLBACKCONTEXT_CANDIDATE`
- `CPX_CALLBACKCONTEXT_RELAXATION`

The callback functions for these contexts will be described in the following sections.

4.2.1 Candidate callback

The candidate callback is called when the solver has found a new candidate solution for an integer-feasible solution or has encountered an unbounded relaxation. Since our model lacks subtour elimination constraints, the solution might contain multiple independent components, making it invalid for the Traveling Salesman Problem (TSP). If this is the case, we have to reject the solution and patch it by applying new cuts, like we did in Benders' Loop method. CPLEX facilitates these steps with the `CPXcallbackrejectcandidate` function.

[5]To set the callback function, `CPXcallbacksetfunc` is used

4.2.2 Relaxation callback

CPLEX invokes the generic callback in this context when it has found a relaxed solution available, usually it is not integer feasible. In this callback, we employ Concorde[6] to find the independent connected components and to find and add violated cuts.
First of all, we find the connected components. If we find only one connected component, since it is a relaxed solution, it may not be a tsp solution so we detect the violated cuts and add them with `CPXcallbackaddusercuts`. Otherwise, we apply patching like before.

Modified Greedy As an optional flag, we added to possibility to use the information from the fractional x^* to compute a new heuristic solution to possibly make a more informed decision when we choose arcs.
The idea is to modify the costs of all arcs interpreting each value of x^* as a probability:

$$cc_{ij} = c_{ij} \cdot (1 - x^*_{ij}) \tag{8}$$

In this way, we can give more weight to the edges selected by the solver, but also maintaining the distance. This heuristic solution is then posted to CPLEX with `CPXcallbackpostheursoln`.

4.2.3 Comparative Analysis

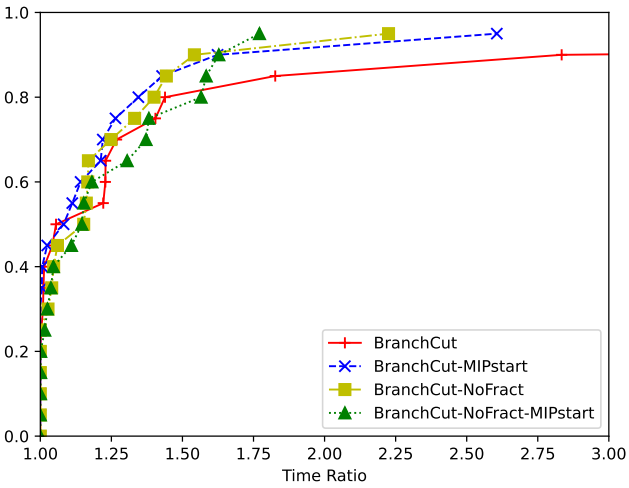


Figure 16: Performance profile between different configurations of branch&cut with 300-350 nodes

To analyze performance of the different configuration of the branch and cut method, we keep the `CPX_CALLBACKCONTEXT_CANDIDATE` active in all algorithms, to assure the finding of a TSP solution. We also exclude the posting of heuristic solution during a relaxation callback, because we notice during the test of code that it did not improve the solution at any iteration. So the four configuration

[6] `https://www.math.uwaterloo.ca/tsp/concorde/index.html`

compared are the basic branch and cut, the branch and cut with warm start, the branch and cut with relaxation callback and the branch and cut with relaxation callback and warm start. For random instances with the number of nodes between 300 and 350, as shown in Figure 16, the configuration using fractional callback and MIP start performs slightly better. It's important to note that all times have been scaled by 30 seconds because some executions were extremely fast. Somewhat surprising, increasing the number of nodes between 400 and 450, changes the performance profile dramatically, as can be seen in Figure 17.

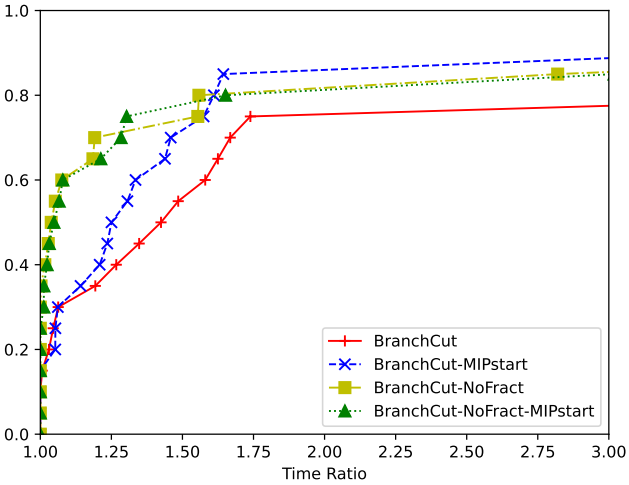


Figure 17: Performance profile between different configurations of branch&cut with 400-450 nodes

These observations underscore the necessity for more rigorous statistical analysis to better understand and predict algorithm performance across different instance sizes, since either configuration could be experiencing an unusual stroke of bad luck.

Nevertheless, we will assume that the fractional callback performs better and we will use this configuration to compare Benders Loop method and Branch-and-Cut.

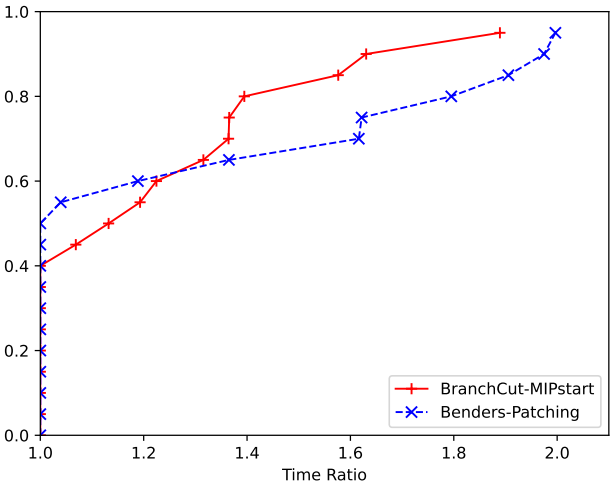


Figure 18: Performance profile between branch&cut and benders loop with 400-450 nodes

As shown in Figure 18 and detailed in Table 5, Branch&Cut outperforms the alternative in most cases, sometimes by a significant margin. Nonetheless, Benders in some instances outperforms Branch&Cut. Again, more runs and a statistical analysis would be needed to crown a definitive winner.

instance	BranchCut	Benders
0	219.28	417.88
1	179.43	290.01
2	75.77	103.42
3	267.44	203.30
4	89.21	56.59
5	48.38	45.27
6	71.72	63.35
7	150.44	79.63
8	126.71	90.85
9	171.54	278.18
10	127.28	103.93
11	248.65	208.42
12	155.94	95.64
13	304.06	600.43
14	350.75	364.60
15	129.53	94.86
16	80.45	95.64
17	331.34	242.88
18	287.86	574.74
19	36.47	65.47

Table 5: Time comparisons between branch&cut and benders loop

Given the previous observations, it is also insightful to examine the behavior of exact models with nodes ranging from 300 to 350. The results, presented in

Figure 19, clearly demonstrate the superiority of the Branch&Cut algorithm in this case.

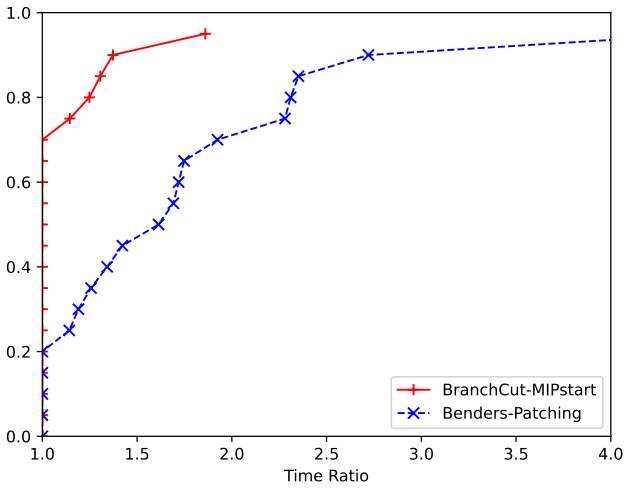


Figure 19: Performance profile between branch&cut and benders loop with 400-450 nodes

5 Matheuristics

As we have seen in the previous sections, exact methods are guaranteed to solve instances to proven optimality, but they need significant computing resources for large instances. On the other hand, heuristics are much faster, but they are not guaranteed to find the optimal solution.
The basic idea of matheuristics is to use the MP solver as a basic tool within the heuristic framework [10]. In this way, we can solve instances of around 1000 nodes within a reasonable time with a better solution than the heuristic.
During the course, we studied two methods—*hard fixing* and *local branching*—both of which involve passing a restricted model, in which certain solutions are arbitrarily excluded, to the MIP solver.

5.1 Hard Fixing

Hard Fixing, or Diving heuristics [5], are a family of MIP heuristics that iteratively fix variables to integer values until a feasible MIP solution is obtained. The name diving comes from the fact that by fixing variables, we fix a path in the branch-and-cut tree, ignoring other branches, and we dive in the free part, where the solver can do its work.
We start from an heuristic solution x_e^H, and we fix some arcs $e \in \tilde{E}$:

$$x_e, \forall e \in \tilde{E} : x_e^H = 1 \tag{9}$$

with $\tilde{E} \subset E = \{e \in E : x_e^H = 1\}$. We repeat this process iteratively, until the time limit is reached.
The only thing to define is how to choose with edges belong to $\tilde{E}$: the easier and most efficient way is to insert the edge with a fixed probability. The pseudocode can be found in Algorithm 6.

Algorithm 6 Hard Fixing

Input: CPLEX MIP model
Output: set of N nodes forming an Hamiltonian tour with a cost c

```
1: procedure HardFixing
2:     compute warm start
3:     while stopping criterion not met do
4:         add mip start
5:         fixing
6:         mip solver and build solution
7:         undo fixing
```

5.2 Local Branching

In Local Branching [11], instead of fixing variables like in Hard Fixing, we try to restrict the search space. In hard fixing, given the heuristic solution x_e^H we have to choose what variables to fix and how many; the intuition behind Local Branching is to only choose *how many*.
The procedure is similar to the local search heuristics, like 2-opt, in the sense that we find the solution in a neighborhood, which is chosen with the introduc-

tion in the model of linear inequalities called *Local Branching cuts*.
For our purposes, the inequality is:

$$\underbrace{\sum_{e:x_e^H=1} x_e}_{\text{n preserved arcs}} \geq n - k \tag{10}$$

where k is a parameter that gives degrees of freedom to the MIP solver. In the original paper the constraint is different:

$$\underbrace{\underbrace{\sum_{e:x_e^H=0} x_e}_{\text{n flip 0-1}} + \underbrace{\sum_{e:x_e^H=1} (1 - x_e)}_{\text{n flip 1-0}}}_{\text{Hamming distance} H(x,x^H)} \leq k \tag{11}$$

which could be interpreted as the Hamming distance between x and x^H and was thought for general problems. In the TSP, since the number of 1s is always n, the number of flips is always the same. Therefore, calculating it twice is redundant. Thus, either the constraint should be set to $2k$ or one of the flip calculations should be eliminated, resulting in an *asymmetric version*.
Regarding on how setting k, when the original paper was written setting it to 5 or 10 worked well, but at the present date, with the higher efficiency of CPLEX, we can bring it higher to 20,30 or 50. The performance profile for different values of fixed k can be found in Image 20.

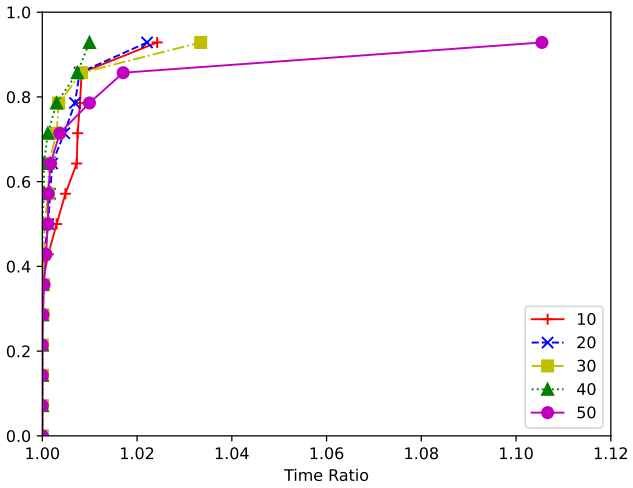


Figure 20: Performance profile of Hyper parameter k tuning

dynamic k An enhancement to the original technique is to change the value k between iterations, so to vary the neighborhood that needs to be considered. The idea is to increase k when we can't find good enough solutions, and decrease it otherwise. The policy we implemented is as follows:

k	**10**	**20**	**30**	**40**	**50**
fl1400.tsp	20551.00	20551.00	20551.00	20551.00	20551.00
nrw1379.tsp	59378.00	59213.00	59294.00	59159.00	59090.00
pcb1173.tsp	59905.00	59900.00	59511.00	59432.00	59478.00
fl1577.tsp	22860.00	22860.00	22860.00	22860.00	22860.00
u1060.tsp	233518.00	235127.00	233587.00	235246.00	233795.00
rl1304.tsp	268109.00	267549.00	270513.00	261762.00	266228.00
vm1084.tsp	247650.00	247007.00	245982.00	245867.00	246774.00
d1655.tsp	65237.00	65260.00	65260.00	65260.00	65260.00
d1291.tsp	52697.00	52697.00	52704.00	52697.00	52704.00
rl1323.tsp	282233.00	281384.00	282233.00	282233.00	311057.00
rl1889.tsp	332442.00	332442.00	332442.00	332027.00	332442.00
u1817.tsp	60706.00	60207.00	60706.00	60806.00	60806.00
pr1002.tsp	270384.00	270384.00	270384.00	270384.00	270384.00
vm1748.tsp	353710.00	351666.00	351090.00	351207.00	351696.00

Table 6: Hyper parameter k results

- keep a stagnation counter to count how many iterations do not improve the solution, if it becomes greater than a threshold, increase k
- if the new solution is better, we have two choices:
  - if the improvement is not much (by default 2%, but can be tuned), we increase k to generate deeper cuts
  - otherwise, we decrease it until it reaches a lower bound

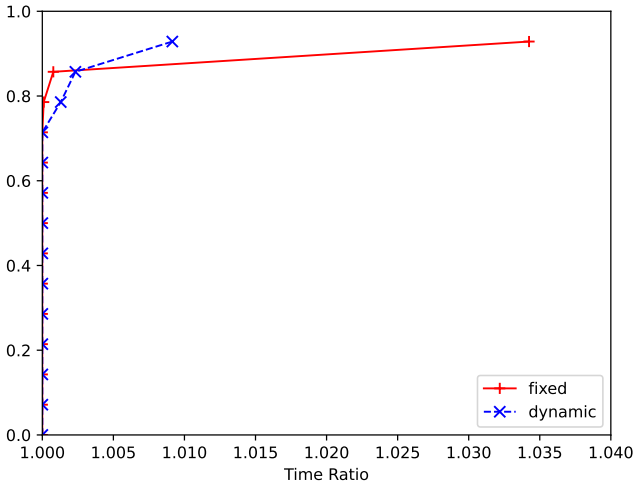


Figure 21: Performance profile dynamic k vs best fixed k

As it can be seen on the performance profile in Figure 21, both approaches perform similarly.

k^* **approach** Instead of hard-setting the value k, we tried an approach based on the following intuition: we can find the minimum value k^* for which the local branching cut does not change the solution and fix k as the value between 0 and k^*. The value k^* is computed in the following way:

$$k^* = \sum_{e:x_e^H=1} (1 - x_e^*) \tag{12}$$

where x^* is the optimal solution of the LP relaxation of the original model.

5.3 Comparative Analysis

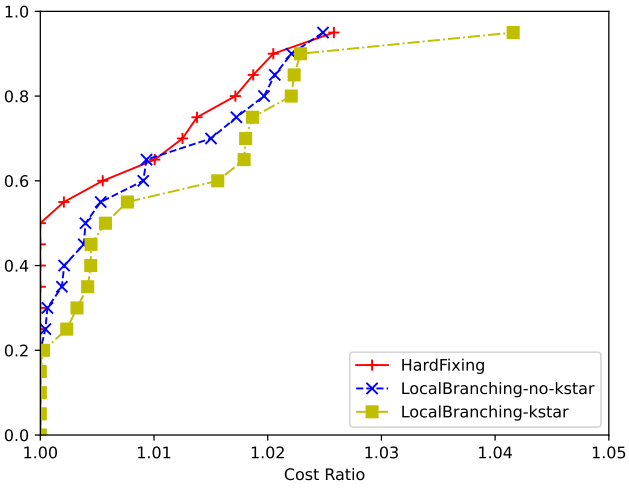


Figure 22: Performance profile of matheuristic algorithms

We test matheuritics algorithms with 20 randomly generated instances of 1000 nodes, with 120 seconds of time limit. As shown in Figure 22, hard fixing generally finds lower-cost solutions. Another significant result is that the dynamic k-star for the local branching does not improve the performance of the method.

6 Conclusions

Heuristics perform exceptionally well with 2000 nodes, whereas exact models struggle even with 400 nodes. However, this speed advantage comes with a trade-off: the best metaheuristic (VNS) has a gap of less than 4% from the optimal solution.

For exact solutions, the best choice, in terms of the time to reach the optimal solution, is Branch&Cut with fractional cuts and a heuristic initial solution.

Matheuristics are effective when exact models cannot find the optimal solution within a feasible time limit, enabling us to solve instances with more nodes by leveraging the CPLEX mathematical model. The best performing matheuristic is Hard Fixing, though it has a small gap compared to Local Branching.

However, these results are not definitive. We acknowledge that our findings lack fine-tuned hyperparameter optimization, which might not make a significant difference but still needs to be considered. Additionally, we did not perform a rigorous statistical analysis, so our runs might be affected by performance variability. Notably, we observed performance differences between our machines.

Both issues could be addressed with access to more computational resources, allowing us to run the algorithms more frequently and for extended periods, thus providing a more complete view.

References

[1] D. L. Applegate, R. E. Bixby, V. Chvatál, and W. J. Cook, *The Traveling Salesman Problem: A Computational Study*, Princeton University Press, 2006.

[2] J. Beardwood, J. H. Halton, and J. M. Hammersley, *The shortest path through many points*, Mathematical Proceedings of the Cambridge Philosophical Society, 55 (1959), p. 299–327.

[3] J. F. Benders, *Partitioning procedures for solving mixed-variables programming problems*, Numerische Mathematik, 4 (1962), pp. 238–252.

[4] N. Biggs, E. K. Lloyd, and R. J. Wilson, *Graph Theory, 1736-1936*, Clarendon Press, USA, 1986.

[5] E. R. Bixby, M. Fenelon, Z. Gu, E. Rothberg, and R. Wunderling, *Mip: Theory and practice — closing the gap*, in System Modelling and Optimization, M. J. D. Powell and S. Scholtes, eds., Boston, MA, 2000, Springer US, pp. 19–49.

[6] I. I. Cplex, *V22.1.1: User's manual for cplex*, (2022).

[7] G. A. Croes, *A method for solving traveling-salesman problems*, Operations Research, 6 (1958), pp. 791–812.

[8] G. Dantzig, R. Fulkerson, and S. Johnson, *Solution of a large-scale traveling-salesman problem*, Journal of the Operations Research Society of America, 2 (1954), pp. 393–410.

[9] G. Enderlein, *Vajda, s.: Mathematical programming. addison-wesley, massachusetts-london 1961; 310 s., geb. 64s*, Biometrische Zeitschrift, 5 (1963), pp. 280–281.

[10] M. Fischetti and M. Fischetti, *Matheuristics*, Springer International Publishing, Cham, 2016, pp. 1–33.

[11] M. Fischetti and A. Lodi, *Local branching*, Mathematical Programming, 98 (2003), pp. 23–47.

[12] B. Gavish and S. C. Graves, *The travelling salesman problem and related problems*, 1978.

[13] F. Glover, *Future paths for integer programming and links to artificial intelligence*, Computers & Operations Research, 13 (1986), pp. 533–549.

[14] L. Gouveia and S. Voss, *A classification of formulations for the (time-dependent) traveling salesman problem*, European Journal of Operational Research, 83 (1995), pp. 69–82.

[15] D. J. J. Houck, J.-C. Picard, M. Queyranne, and R. R. Vemuganti, *Traveling salesman problem as a constrained shortest path problem: theory and computational experience*, technical report, École Polytechnique de Montréal, 1978.

[16] D. S. Johnson and L. A. McGeoch, *The traveling salesman problem: A case study in local optimization*, 2008.

[17] G. Laporte, *A concise guide to the traveling salesman problem*, The Journal of the Operational Research Society, 61 (2010), pp. 35–40.

[18] C. E. Miller, A. W. Tucker, and R. A. Zemlin, *Integer programming formulation of traveling salesman problems*, J. ACM, 7 (1960), p. 326–329.

[19] M. Padberg and G. Rinaldi, *A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems*, SIAM Review, 33 (1991), pp. 60–100.

[20] M. W. Padberg and T.-Y. Sung, *An analytical comparison of different formulations of the travelling salesman problem*, Mathematical Programming, 52 (1991), pp. 315–357.

[21] G. Reinelt, *TSPLIB–a traveling salesman problem library*, ORSA Journal on Computing, 3 (1991), pp. 376–384.

[22] R. Roberti and P. Toth, *Models and algorithms for the asymmetric traveling salesman problem: an experimental comparison*, EURO Journal on Transportation and Logistics, 1 (2012), pp. 113–133.

[23] A. Schrijver, *On the history of combinatorial optimization (till 1960)*, in Discrete Optimization, K. Aardal, G. Nemhauser, and R. Weismantel, eds., vol. 12 of Handbooks in Operations Research and Management Science, Elsevier, 2005, pp. 1–68.

[24] S. Tsubakitani and J. R. Evans, *Optimizing tabu list size for the traveling salesman problem*, Computers & Operations Research, 25 (1998), pp. 91–97.