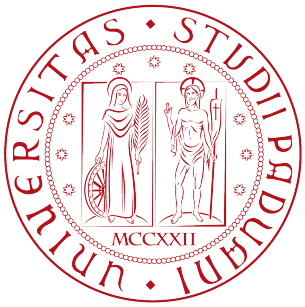


University of Padova

Information Engineering Department

Master's Degree in Computer Engineering

Algorithms for the Travelling Salesman Problem

Operations Research 2 Course

Professor

Prof. Matteo Fischetti

Students

Alberto Valente - 1236460

Enrico Dandolo - 1236009

Academic Year 2020-21

Contents

1

Introduction

1.1 The Travelling Salesman Problem (TSP)

The Travelling Salesman Problem (also known as TSP) can be stated as follows: given a list of cities and the distances between each pair of them, it aims to find the shortest possible route that visits each city exactly once and then returns to the origin city. [1]

It is currently one of the most intensively studied problems in optimization because, even in its purest formulation, it can be applied to a wide range of applications, from plain logistics to the manufacture of microchips, where it contributes to drive down travel times and production costs significantly.

Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing, where the cities represent DNA fragments and their distances are the values of a similarity measure between DNA fragments. In many applications, additional constraints such as limited resources or time windows may be imposed.

Even though the problem is computationally difficult (more details about this in section 1.3), many heuristics and exact algorithms have been developed, so that some instances with tens of thousands of cities can be solved to optimality and even problems with millions of cities can be approximated within a small fraction of 1%. [2]

1.2 Brief history of TSP

Although the exact origins of the Traveling Salesman Problem are unclear, the first example of such a problem appeared in the German handbook *Der Handlungsreisende - Von einem alten Commis-Voyageur* [3], which was used by salesman travelling through Germany and Switzerland in 1832. This handbook was merely a guide, since it did not contain any mathematical language, but it proves that people started to realize that the time one could save from creating optimal paths is not to be overlooked, and thus there is an advantage to figuring out how to create such optimal paths. [4]

The TSP was mathematically formulated in the 1800s by the mathematicians W.R. Hamilton and Thomas Kirkman [5], while the general form of the TSP appears to have been first studied by mathematicians during the 1930s in Vienna and at Harvard. In particular, Karl Menger defined the problem, considered the trivial brute-force algorithm and observed the

non-optimality of the nearest neighbour heuristic. [6]

Remarkable contributions were made in the 1950s, when George Dantzig, Delbert Ray Fulkerson and Selmer M. Johnson from the RAND Corporation [7], wrote what is considered the seminal paper on the subject [8]: they expressed the problem as an integer linear program (see DFJ model in section 1.3.1), devised the cutting plane method and managed to solve an instance with 49 nodes to optimality by constructing a tour and proving that no other tour could be shorter. While this paper [8] did not give an algorithmic approach to TSP problems, the ideas that it proposed were essential for the development of exact solution methods for the TSP, though it would take almost 15 years to find an approach to algorithmically generate these cuts. As well as cutting plane methods, Dantzig, Fulkerson and Johnson employed branch and bound algorithms perhaps for the first time. [9]

Leaping forward to the 1980s Grötschel, Padberg, Rinaldi and others managed to exactly solve instances with up to 2392 nodes, using both cutting planes and branch and bound. [10]

In 1991 Gerhard Reinelt published the TSPLIB, a collection of benchmark instances of varying difficulty, which has been used for comparing results among many research groups. [11]

In the 1990s Applegate, Bixby, Chvátal and Cook developed the Concorde TSP solver [12], which we are going to use ourselves in section 4.3. This program has been used in many recent record solutions: in 2006, Cook and others computed the optimal tour for an instance of 85900 nodes given by a microchip layout problem and it is currently the largest solved TSPLIB instance. For many other instances with millions of cities, solutions can be found that are guaranteed to be within 2-3% of an optimal tour. [13]

1.3 Mathematical formulation of TSP

Proceeding from its well-suited analogy, the TSP can be formalized as the combinatorial optimization problem whose objective is to find the shortest Hamiltonian cycle (a *tour*) in a weighted complete graph, where the graph nodes represent the cities, the arcs are the connections between each pair of cities and their weight is the distance between said cities.

The fact that it is modelled as a complete graph means that each pair of nodes is connected by an arc, so we can assume that a path between two nodes always exists. Furthermore, we look for a Hamiltonian cycle, meaning that the final tour has to start and finish at the same node after having visited all the other nodes only once.

Formally, let $G = (V, A)$ be a directed complete graph with n nodes $V = \{v_1, \ldots, v_n\}$ and let $c : A \to \mathbb{N}^+$ be a symmetric cost function, which represents the weights of the arcs or, equivalently, the distances between adjacent nodes. Taking into account that we are considering a special case of the Asymmetric TSP in which the cost of the arc between two nodes is the same in both directions, the optimization problem of the Asymmetric TSP is

defined as:

$$\begin{cases} \text{INSTANCE:} & < G = (V, A), c > \quad c : A \to \mathbb{N}^+, \quad c(u, v) = c(v, u) \quad \forall u, v \in V \\ \text{QUESTION:} & \text{Is there a tour } \gamma^* =< v_1, \ldots, v_n, v_{n+1} > \text{, with } v_{n+1} = v_1, \text{ of minimal} \\ & \quad cost(\gamma^*) = \sum_{i=1}^{n} c(v_i, v_{i+1})? \end{cases}$$

The decision version of the TSP problem is defined as:

$$\begin{cases} \text{INSTANCE:} & < G = (V, A), c, k > \quad c : A \to \mathbb{N}^+, k \in \mathbb{N}^+ \\ \text{QUESTION:} & \text{Is there a tour } \gamma =< v_1, \ldots, v_n, v_{n+1} > \text{, with } v_{n+1} = v_1, \text{ of} \\ & \quad cost(\gamma) = \sum_{i=1}^{n} c(v_i, v_{i+1}) \leq k? \end{cases}$$

The decision problem related to the TSP is a NP problem, in fact it can be easily proved that TSP is verified in polynomial time given a certificate which could be a Hamiltonian cycle. We can also prove that TSP is a NP-Hard problem by reducing in polynomial time from the HAMILTON problem. An instance of HAMILTON is an arbitrary directed graph $G = (V, A)$, with the related question: is there a simple cycle in G containing all nodes of V?

Since TSP belongs to both NP and NP-Hard classes, then TSP is a NP-Complete problem. As a consequence, supposing P≠NP, no algorithm can solve it in polynomial time.

The traditional lines of attack for the NP-Hard problems are the following:

1. Devising non polynomial algorithms, sometimes suitable for small instances;
2. Restricting the instance set, solving the problem on a subset of instances for which there is a polynomial algorithm;
3. Devising approximation algorithms, returning a feasible solution whose cost is within a given error margin (computable in polynomial time) from the optimal solution;
4. Using *intelligent* exhaustive search methods, e.g. branch-and-bound: try to get the optimal solution while avoiding to examine all feasible solutions (*pruning* strategies); heuristic methods: there is no upper bound on the worst-case running time and no guarantee on the quality of the solution.

As a quick overview on the approximation algorithms, it is important to remember that for the general TSP problem, in which the cost function c is not restricted to be a metric function, if P≠NP there cannot exist a polynomial approximation algorithm with an approximation factor $\rho(|V|)$ computable in polynomial time. On the other hand, if we restrict the cost function c to be a metric function, so with the triangle inequality holding true, e.g. the Euclidean distance, the TSP problem is approximable. A 2-approximation algorithm can be obtained by first determining the Minimum Spanning Tree (MST) T^* of the graph G and then getting a tour γ_{T^*} by visiting in preorder T^*. Moreover, a well known 3/2-approximation algorithm is the Christofides one. It was discovered in 1976 and still stands as the best polynomial time approximation algorithm that has been thoroughly peer-reviewed by the relevant scientific community for the Traveling Salesman problem on general metric spaces. [14]

The fourth point of the above list will be explored in detail in the next sections starting with the Integer Linear Programming (ILP) conventional formulation of Dantzig, Fulkerson and Johnson (DFJ).

1.3.1 Integer Linear Programming: DFJ model

In all our formulations we will consider $V = \{1, 2, ..., n\}$ as the set of n nodes of the TSP instance and the cost function c as related to the Euclidean distance, meaning that it is a metric function for which the triangle inequality holds true.

We define the variables x_{ij} and c_{ij} for $i, j \in V$ as follows:

$$x_{ij} = \begin{cases} 1 & \text{if arc } (i,j) \in A \text{ is a link in the optimal tour} \\ 0 & \text{otherwise} \end{cases}$$

$$c_{ij} = \text{length of the arc } (i,j) \in A.$$

Since arcs that start and end in the same node are not allowed, both variables are not defined in the case that $i = j$.

The objective function to minimize is:

$$\sum_{\substack{i,j \in V \\ i \neq j}} c_{ij} x_{ij}.$$

The Dantzig, Fulkerson and Johnson (DFJ) formulation of TSP is formulated as:

$$min \sum_{\substack{i,j \in V \\ i \neq j}} c_{ij} x_{ij}$$

$$\sum_{\substack{j \in V \\ j \neq i}} x_{ij} = 1 \qquad \forall i \in V \tag{1.1}$$

$$\sum_{\substack{i \in V \\ i \neq j}} x_{ij} = 1 \qquad \forall j \in V \tag{1.2}$$

$$\sum_{\substack{i,j \in S \\ i \neq j}} x_{ij} \leq |S| - 1 \qquad \forall\ S \subset V : \{1\} \notin S, |S| \geq 2 \tag{1.3}$$

$$0 \leq x_{ij} \leq 1 \quad integer \qquad \forall i, j \in V. \tag{1.4}$$

Equations (1.1) and (1.2) are called *Degree Constraints*, because they are limiting, respectively, the outer and inner degree of each node, leading to a path where each node has only one edge starting from it and one ending on it, which is one of the fundamental properties that define a Hamiltonian cycle.

On the other hand, inequalities (1.3) are the *Subtour Elimination Constraints* (SECs), because they state that, for any possible proper subset of nodes from V, we cannot have a number of arcs connecting them that allows for the formation of cycles, so that the returned

solution is definitely a single tour and not the union of smaller tours.

The exponential number of constraints of this formulation makes it unsuitable to be solved in practice, so a couple of alternative *compact* formulations, which involve a polynomial number of constraints, are introduced in Chapter 3.

Another typical procedure to tackle this problem is to apply the so-called Degree Constraints and append only those SECs when they are actually violated. We are focusing on this kind of approach in Chapter 4.

2

Experimental Setup

In this chapter we provide an overview of the structure of our project and of the dataset used in our tests, then we briefly present the set of tools we employed to carry it out successfully.

2.1 Software and tools employed

The main purpose of the project is to study linear optimization techniques and experience with heuristic algorithms while focusing on solving instances of TSP of increasing size. Due to the fact that, as we have extensively discussed is section 1.3, TSP is a computationally hard problem, it should be clear why efficiency is one of our main concerns, so we decided to use the C programming language, as it is recognized to be one of the fastest languages and provides full control on memory management, which is especially significant when dealing with huge data structures.

The main programming tools we used are the IDE Microsoft Visual Studio Community 2019 (version 16.9.1) and CMake (version 3.18.5) to build the project, as together they provide all the functionalities we needed to manage a large project which includes CPLEX, Concorde and other libraries. Both the systems we employed for software development and to run the tests are based on Windows 10.

To allow for tests reproducibility, we devoted a system to perform all of them: it consists of an AMD Ryzen 5 3600 processor with 16GB RAM @ 3200 MHz, and we set *Maximum performance* in Windows power management settings in order to keep the CPU at its maximum frequency for all the time they were running.

CPLEX is a mathematical optimization software developed by IBM. [15] In our project we used the CPLEX Callable Library (version 20.1), which is the API they provide to allow for integration in C projects. All the algorithms we are going to introduce in chapters 3, 4 and 5 make use of the integrated Mixed-Integer Programming (MIP) solver.

Concorde is a program which was developed to solve the symmetric 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. [12] In section 4.3 we use a couple of useful functions of the latest version of its library (which dates back to 2003) together with the aforementioned CPLEX MIP solver to implement a variant of the Branch & Cut method introduced in section 4.2.

We freely decided to use GitHub as a version control system for our codebase. The link to our repository is the following: `https://github.com/AlbyVal97/RO2_TSP`.

Gnuplot is a command-line and GUI program to visualize mathematical functions and data interactively. We used it to draw the tours associated to the solutions returned by the algorithms, both to have a look at how these tours change during their execution or to prove that said solutions are in fact Hamiltonian cycles. Some examples of these plots are shown in chapter 8. In our project we exploited the command-line mode of Gnuplot, which is called by our program by first compiling a *gnuplot_commands* text file with the instructions and then performing a system call with the said file as a parameter.

The performance of all the algorithms described below was analyzed using a performance profiling tool (see chapter 8) which is specific for our needs and is available at:

`http://www.dei.unipd.it/~fisch/ricop/RO2/PerfProf`.

2.2 Source code design overview

The program has been designed in order to manage three different *operating modes*, each with its own set of command-line parameters:

- *DEFAULT* mode: runs a specific algorithm on a single instance provided in input. Depending on the *verbosity level* set by the user, it gives more or less details about the execution and may eventually show the plot of the optimal solution, or of the best solution found within the time limit, if dealing with heuristics. This is the mode we employed while writing code to debug our algorithms;
- *CREATE_INSTANCES* mode: generates a set of random instances with a fixed number of nodes. Both the number of such instances and the amount of their nodes are provided by the user. More details about this can be found in section 2.3;
- *RUN_TEST* mode: runs a sequence of algorithms from a list provided by the user on a set of instances contained in a given folder. Basically, it allows to execute automatically multiple algorithms one immediately after the other and with the lowest verbosity level, so that no computational power is wasted. It also collects the results, which consist in the time needed to reach the optimal solution or the cost of the best solution found, in a *csv* file with the format required by the performance profiling tool. Of course, this mode was employed to run all the tests we needed to get the results shown in chapter 8.

As reported in the *CMakeLists* configuration file, the project can be divided into the Concorde library (called *tsp_concorde_lib*) files and the source files actually belonging to the project itself (simply called *tsp*).

Here we discuss about the structure and the overall content of the latest:

- *main.c* → it is the entry point of the program. It parses the command line and decides how to proceed with the execution depending on the operating mode and parameters handed by the user;

- *tsp.h* and *tsp.c* → contain the main functions implementing all the algorithms solving TSP and define the *TSPopt* function, whose task is to call the correct set of functions according to the method being executed;
- *tsp_utilities.h* and *tsp_utilities.c* → contain all the utility functions called inside *tsp.c* or *main.c* which are not directly related to a specific algorithm, such as the function that parses an instance file;
- *instance.h* → defines some global parameters, the list of models and algorithms implemented and the *instance* structure, which collects all the data about the instance to solve (e.g. the number of nodes) and it is conveniently shared among the functions;
- *convex_hull.h* and *convex_hull.c* → contain a set of functions used uniquely to compute the convex hull of a set of nodes in the heuristic algorithm described in section 6.2.

2.3 Dataset management

All the TSP instances we employed in our tests have been created using the aforementioned *CREATE_INSTANCES* mode. In detail, these are generated randomly by setting, for each one, a different seed, which is chosen from a predefined sequence of seeds. By doing so, the instances we create are deterministic, in the sense that, for example, if we decide to generate 20 instances and then another 40 of the same size, the first 20 will be exactly the same as before. The coordinates of the nodes are always fractional values between 0 and 1. [11]

To run our tests, we created a dataset of instances of increasing size, from a minimum of 50 nodes to a maximum of 2000 nodes, in order to assign to each group of algorithms the set of instances with the most appropriate size. The choice of said size has been made, in the case of exact algorithms, by running small batches of instances of different size until the time limit, which we set to 30 minutes, of just a small number of instances is reached, so that most of the results collected are actually useful data and the difference in performance between the algorithms becomes more clear. The number of instances selected to make the tests statistically relevant is 20.

Some of the tests we performed are *tuning tests*, meaning that their objective is to decide which variant of the same algorithm performs the best. For example, this can be due to a change in the value of a hyperparameter or to a slight difference in approach which we know could lead to significantly different results. The best variant is then tested against the other algorithms of the same group. For this reason, it would be statistically incorrect to use the same set of 20 instances for both the tuning and the actual tests, so we generated a set of 40 instances and then split them among a *validation set* and a *test set*, such that our decision about which variant is the best one is not conditioned by the results on the test set.

3

Compact models

The exponential number of constraints of the conventional DFJ formulation makes it unsuitable to be solved in practice, so a couple of alternative compact formulations, which involve a polynomial number of constraints, are introduced in this chapter.

3.1 Basic model

Considering the DFJ formulation (section 1.3.1) as a starting point, we keep the same variables x_{ij} and c_{ij} and we include in the model the previous objective function and the constraints associated to equations (1.1) and (1.2). Subtour Elimination Constraints (1.3) are not included here, as this model is not intended to be standalone, but to be used as a baseline for all SECs separation methods and matheuristics.

We can rewrite the DFJ formulation in a simpler way considering now an undirected complete graph $G = (V, E)$ and introducing the variables x_e for each $e \in E$, which play the same role of the variables x_{ij} given $e = \{i, j\} \in E$. We do the same for the distance function c. The function $\delta : V \to 2^E$ returns the set of edges incident to a given node.

The Basic model is the following:

$$\begin{cases} min \sum_{e \in E} c_e x_e \\ \sum_{e \in \delta(h)} x_e = 2 \qquad \forall h \in V \\ 0 \leq x_e \leq 1 \quad integer \qquad \forall e \in E. \end{cases}$$

In our implementation of the Basic model, we are considering the Symmetric version of TSP problem, in fact the arcs (i, j) and (j, i) are represented by the same edge $\{i, j\}$ with the same distance c_{ij}. The total number of variables in our Basic model is reduced down to $n(n-1)/2$, thus only the variable x_{ij} with $i < j$ is sufficient to represent the edge $\{i, j\}$.

Instead, the next two formulations of the TSP model which are introduced in sections 3.2 and 3.3 will consider a directed complete graph $G = (V, A)$ with all the n^2 variables x_{ij}. Of course, self-loops are still not allowed, so it holds that $x_{ii} = 0$ for each $i \in V$. For simplicity, in the following sections we will be using the same (i, j) notation for both edges and arcs.

3.2 Miller, Tucker and Zemlin (MTZ) model

As we have stated before, the inequalities (1.3) introduce $O(2^n)$ constraints, which make the DFJ model unsuitable to be solved in practice. The MTZ model is a compact model in which the SECs are reformulated in order to be polynomial in number (specifically $O(n^2)$).

At each node $i \in V$ is assigned a new variable u_i. The variables u_i are used to introduce an increasing sequence number in the optimal tour: starting from the second node ($u_2 = 0$), the value of the sequence is increased by 1 at each node until we get to the last node ($u_n = n-2$). The first node of the tour is a special one, as it always has $u_1 = 0$.

The following statement must hold true to achieve the aforementioned sequence:

$$if \quad x_{ij} = 1 \quad then \quad u_j \geq u_i + 1 \quad \forall i, j \in V \setminus \{1\}. \tag{3.1}$$

In order to express this statement as a CPLEX constraint, one possible solution is to use the so called *BIG-M* trick. Let's include in the model the following constraints:

$$u_j \geq u_i + 1 - M(1 - x_{ij}) \quad \forall i, j \in V \setminus \{1\}, M \gg 0. \tag{3.2}$$

These constraints work as follows: when $x_{ij} = 1$, it turns exactly into the statement (3.1). If $x_{ij} = 0$, it becomes $u_j \geq u_i + 1 - M$, which is basically a useless constraint, because $M \gg 0$ makes the right-hand side of the inequality certainly negative, thus we say that such constraint is *deactivated*. The smallest feasible value we can assign to M is $M = n - 1$. That is because, in the extreme case of u_i reaching its maximum value $u_i = n - 2$, we get $u_j \geq n - 2 + 1 - n + 1 = 0$, which is still a useless constraint, but is also consistent with the lower bound associated to the u_i variables.

To show that these constraints prevent the optimal solution from including subtours, let's suppose we have a solution with more than a subtour and, from these, let's consider one which does not include the starting node. The values of the u variables of this subtour should form an increasing sequence, but this is impossible because, sooner or later, we will need to come back to its starting node, meaning that for the last covered arc (i, j) we would have that $u_j < u_i + 1$, which violates the statement (3.1). This contradiction proves that this model guarantees that the optimal solution cannot be made up of subtours.

As a summary, the MTZ model is formulated as follows:

$$\begin{cases} min \sum_{\substack{i,j \in V \\ i \neq j}} c_{ij} x_{ij} \\ \sum_{\substack{j \in V \\ j \neq i}} x_{ij} = 1 \quad \forall i \in V \\ \sum_{\substack{i \in V \\ i \neq j}} x_{ij} = 1 \quad \forall j \in V \\ u_j \geq u_i + 1 - (n-1)(1 - x_{ij}) \quad \forall i, j \in V \setminus \{1\} \\ 0 \leq x_{ij} \leq 1 \quad integer \quad \forall i, j \in V \\ 0 \leq u_i \leq n-2 \quad integer \quad \forall i \in V \setminus \{1\} \\ 0 \leq u_1 \leq 0. \end{cases}$$

3.2.1 Implementation variants of MTZ model

The implementation of the model we introduced in section 3.2 is called *MTZ_STATIC*. It was implemented using the CPLEX library function *CPXaddrows*, which allows to add, one by one, all the constraints of the model before calling the *CPXmipopt* function, which starts the optimization procedure.

A first variant of *MTZ_STATIC* is the so-called *MTZ_LAZY*, which turns the constraints (3.2) into lazy constraints. According to the related CPLEX documentation [16], *lazy constraints* are a group of constraints that the user can identify as unlikely to be violated. Simply including them in the original formulation could make the LP subproblem of a MIP optimization very large or too expensive to solve. As a consequence, these situations can be handled by setting up a *lazy constraints pool* before MIP optimization begins, using the routine *CPXaddlazyconstraints*. This is exactly what we did in our implementation, because while the MTZ model exhibits a polynomial ($O(n^2)$) number of SECs, they can easily become a huge amount of constraints. Recall that the goal of said pool is to allow the optimization algorithm to perform its computations on a significantly smaller model, in the hope of delivering faster run times. This means that the model starts out small, and then potentially grows as members of the pool are added to the model. Lazy constraints are only (and always) checked when an integer-feasible solution candidate has been identified, and of course, any of these constraints that turn out to be violated will then be applied to the full model.

A second variant is *MTZ_SEC2_STATIC*, which is basically *MTZ_STATIC* with the addition of Subtour Elimination Constraints of degree 2. They correspond to the subset of constraints (1.3) with $|S| = 2$, but can be more easily expressed as follows:

$$x_{ij} + x_{ji} \leq 1 \qquad \forall i < j. \tag{3.3}$$

These constraints states that in a feasible solution we could never have simultaneously $x_{ij} = 1$ and $x_{ji} = 1$, because this would result in an arc between two nodes being covered in both directions, meaning that these two nodes would be visited twice, therefore the solution would clearly no longer be a Hamiltonian cycle.

The third and last variant we implemented is *MTZ_SEC2_LAZY*, whose only difference from the previous one is the fact that it provides both the (3.2) and (3.3) constraints to CPLEX as lazy constraints.

3.3 Gavish and Graves (GG) model

The GG model is a *single-commodity flow formulation*, where subtours are broken by introducing $n(n-1)$ nonnegative variables:

$$y_{ij} = \text{"Flow" in an arc } (i,j) \text{ with } i,j \in V \text{ and } i \neq j.$$

Variables y_{ij} can be interpreted as the number of arcs on the path from arc (i,j) to node 1 in the optimal tour. Starting with a value of commodity equal to $n-1$, at each arc one unit of said commodity is lost, so there is no flow conservation. This procedure is followed only in the optimal tour, so linking constraints are needed to set $y_{ij} = 0$ when $x_{ij} = 0$.

New constraints are added to the model to perform this procedure:

$$\sum_{\substack{i \in V \\ i \neq h}} y_{ih} - \sum_{\substack{j \in V \\ j \neq h}} y_{hj} = 1 \qquad \forall h \in V \setminus \{1\} \tag{3.4}$$

$$y_{ij} \leq (n-1)x_{ij} \qquad \forall i,j \in V, i \neq j. \tag{3.5}$$

The following constraint restricts $n-1$ units of a single commodity to exit from the node 1:

$$\sum_{\substack{j \in V \\ j \neq 1}} y_{1j} = n-1. \tag{3.6}$$

These constraints lead to an optimal solution where subtours are no longer allowed. Indeed, just take a solution in which at least two subtours are present and consider the one without the node 1. In such subtour let α be the value of units of the single commodity at the arc which exits from the start node. At the end of the cycle that value is decreased down to $\alpha - k + 1$ where k is the number of nodes of the subtour, thus the value of the variable y_{ij} at the first and at the last arc violate the constraint (3.4).

As a summary, the GG model is formulated as follows:

$$\begin{cases} min \sum_{\substack{i,j \in V \\ i \neq j}} c_{ij} x_{ij} \\ \sum_{\substack{j \in V \\ j \neq i}} x_{ij} = 1 \quad \forall i \in V \\ \sum_{\substack{i \in V \\ i \neq j}} x_{ij} = 1 \quad \forall j \in V \\ \sum_{\substack{i \in V \\ i \neq h}} y_{ih} - \sum_{\substack{j \in V \\ j \neq h}} y_{hj} = 1 \quad \forall h \in V \setminus \{1\} \\ y_{ij} \leq (n-1) x_{ij} \quad \forall i, j \in V \\ \sum_{\substack{j \in V \\ j \neq 1}} y_{1j} = n - 1 \\ 0 \leq x_{ij} \leq 1 \quad integer \quad \forall i, j \in V \\ 0 \leq y_{ij} \leq n - 1 \quad integer \quad \forall i, j \in V \\ 0 \leq y_{i1} \leq 0 \quad integer \quad \forall i \in V. \end{cases}$$

4

Subtour Elimination Constraints separation methods

Another typical procedure to tackle the exponential number of SECs of the DFJ formulation is to apply the so-called Degree constraints and append only those SECs when they are actually violated. This is the kind of approach we are focusing on in this chapter.

4.1 Benders implementation

The Benders method consists in repeatedly solving the TSP problem corresponding to the Basic model and then applying on-the-fly SECs only when they are violated. For this reason, it is also called *Loop method* and the graph $G = (V, E)$ we are considering here is undirected and complete. Let $E(S)$ be the set of edges inside a connected component $S \subset V$.

The SECs are reformulated as:

$$\sum_{e \in E(S)} x_e \leq |S| - 1 \qquad \forall S \subset V : |S| \geq 3. \tag{4.1}$$

As mentioned before, at first the Benders method finds the optimal solution x^* of the MIP problem related to the Basic model introduced in section 3.1. Notice that such optimal solution could be infeasible due to the presence of subtours. Now let $G' = (V, E')$ be the subgraph obtained from G considering only the edges selected by the optimal solution:

$$E' = \{e \in E : x_e^* = 1\}.$$

Using the algorithm *Find_Connected_Components* (see the pseudocode of Algorithm 1), which is of polynomial complexity, we are able to find all the connected components of G'. Since each connected component is related to a specific violated SEC in (4.1), we add such constraints to the starting model. More in detail, in our implementation their insertion is performed outside the *CPXmipopt* function. As a result, the updated model is solved to optimality to find another optimal solution, which could be infeasible or not, and this procedure is iterated until one of the optimal solutions found turns out to be feasible, which happens when it shows no more than one connected component.

Figure 4.1 summarizes the Benders method procedure we have just discussed, while Algorithm 1 describes the steps to compute the connected components of the subgraph G'.

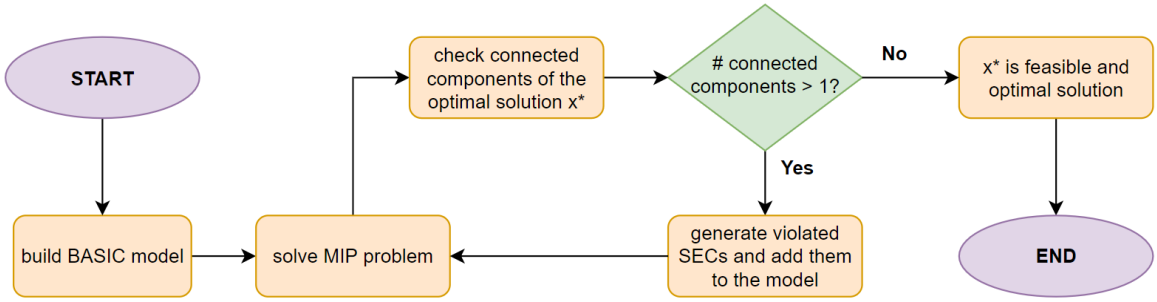


Figure 4.1: Flow Chart of the Benders method.

The main variables and data structures used by the algorithm are the following:

- x^* = optimal solution of the current iteration, identifies the subgraph $G' = (V, E')$;
- $ncomp$ = number of connected components;
- $succ[i]$ = successor of the node $i \in \{1, \ldots, n\}$ inside the connected component;
- $comp[i]$ = index of the connected component containing the node $i \in \{1, \ldots, n\}$.

Algorithm 1: Find_Connected_Components($G = (V, E), x^*$)

Result: Number of connected components (*ncomp*), List of successors (*succ*), Array of indexes of the connected components (*comp*).

```
n ← |V|;
ncomp ← 0;
for i ← 1 to n do
    succ[i] ← −1;
    comp[i] ← −1;
end
for start ← 1 to n do
    if comp[start] ≥ 0 then
        continue;
    end
    ncomp ← ncomp + 1;
    i ← start;
    done ←false;
    while !done do
        comp[i] ← ncomp;
        done ←true;
        for j ← 1 to n do
            if i ≠ j and x*_ij > 0.5 and comp[j] = −1 then
                succ[i] ← j;
                i ← j;
                done ← false;
                break;
            end
        end
    end
    succ[i] ← start;
end
return (ncomp, succ, comp);
```

4.2 Branch & Cut with CPLEX callbacks

The Branch & Cut method is mostly similar to the previous Benders method: the main difference lays on the fact that, while in Benders each iteration corresponds to a call to MIP solver with a model whose SECs are iteratively inserted, in Branch & Cut the MIP solver is called just once and, as a consequence, there is only one branching tree. Again, the starting model is the Basic one introduced in 3.1, so at the beginning it lacks the Subtour Elimination constraints.

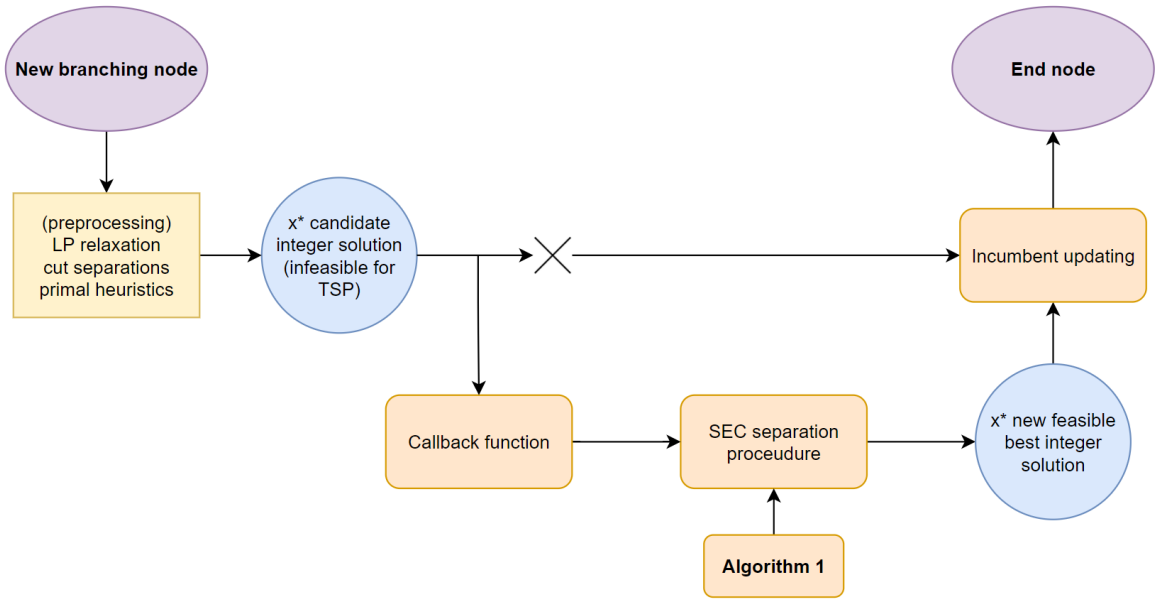


Figure 4.2: Flow Chart of the lifecycle of a CPLEX node in Branch & Cut method.

When solving a node of the branching tree, let's suppose that CPLEX found a new integer solution x^* by applying a sequence of cuts and heuristics to the fractional solution of the continuous relaxation. Notice that x^* is a *candidate solution*, which is feasible for the given partial model but very likely infeasible for the TSP problem.

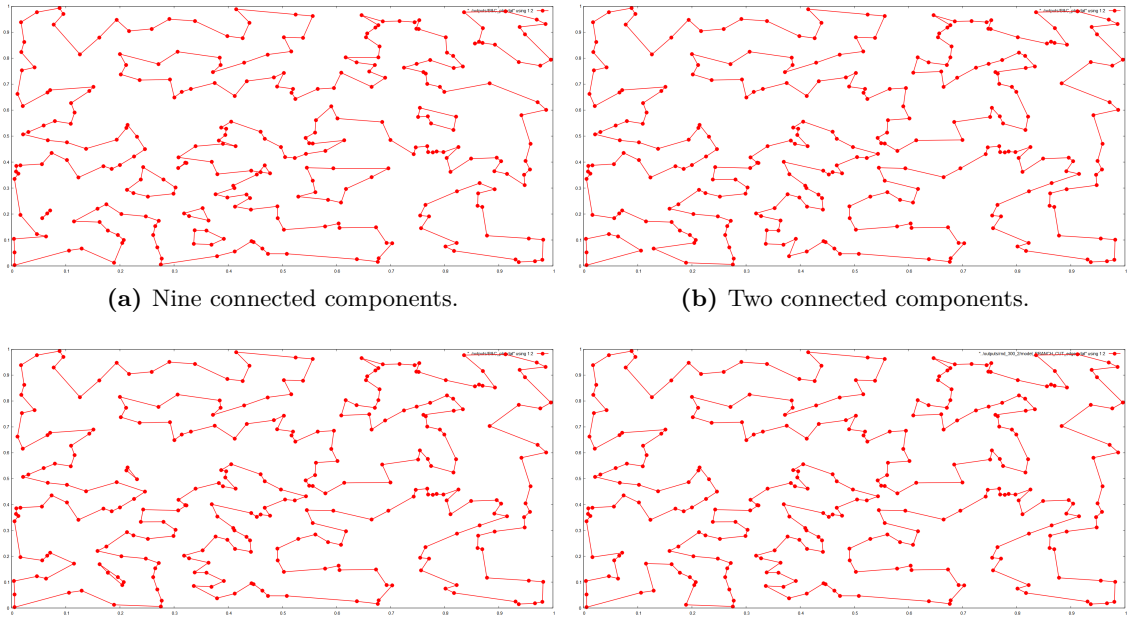
If the cost of said x^* is lower than the best current integer solution (which is called *incumbent*), then the incumbent should be updated accordingly. And that is where CPLEX callbacks comes in handy: before doing that, CPLEX drops such candidate solution and considers it infeasible using the *CPXcallbackrejectcandidate* function.

This detour, represented in Figure 4.2 by the junction and the cross in the center, is made possible by instructing CPLEX (using the *CPXcallbacksetfunc* function) to execute our custom callback function. Inside of this callback function, a SEC separation procedure is called: it finds the connected components of x^* using the Algorithm 1, identifies all the violated SECs and generates the related cuts among the ones in (4.1). Those are basically the same steps we discussed in the Benders method, with the main difference that in Branch & Cut they are performed inside the MIP solver.

At the end of the procedure, the solution will turn out being feasible for the TSP problem and, if it is better then the incumbent, CPLEX proceeds updating the incumbent.

Figure 4.2 represents the Branch & Cut procedure we have described and specifically the lifecycle of a single CPLEX node when a new candidate integer solution is found in such branching node.

Figure 4.3 shows a brief sequence of some integer solutions found during an execution of the Branch & Cut method. Such sequence shows a gradual decrease of the number of connected components as a result of the CPLEX callback being called. At the end (Figure 4.3d) we can see that a feasible solution for the TSP is found.



(a) Nine connected components.

(b) Two connected components.

(c) Three connected components.

(d) One connected component: feasible integer solution achieved.

Figure 4.3: Sequence of some integer solutions during a Branch and Cut execution with SECs applied with a CPLEX callback.

4.3 Introducing Concorde TSP solver

In this section we introduce a variant of the previous Branch & Cut method. We called it *Advanced Branch & Cut* as it consists in a more advanced usage of CPLEX callbacks, where we intercept the solution of the continuous relaxation associated to the ILP problem as soon as it is solved.

More specifically, let x^{LP} be the optimal solution of the continuous relaxation associated to the Basic model, which is, once again, the model we begin with. The idea behind this method is to instruct CPLEX to execute, as soon as it computes the optimal fractional solution x^{LP}, a specific callback function where we add a set of cuts for this fractional solution.

The callback function uses the *CCcut_connect_components* function of the Concorde library to compute the connected components of the graph consisting of the nodes V and of the edges $e \in E$ for which the condition $x_e^{LP} > 0$ holds true. If the number of connected components is greater than 1, then we add to the CPLEX cut pool the violated SECs (4.1) until only one connected component remains.

Once the connected graph is obtained, we employ the *CCcut_violated_cuts* function of the Concorde library to apply a separation procedure and identify a set of cuts violated by x^{LP}. Given $\delta(S) = \{(u, v) \in E : u \in S, v \in V \setminus S\}$ the set of edges associated to the cut-set $(S, V \setminus S)$, these cuts are inequalities with the following form:

$$\sum_{e \in \delta(S)} x_e^{LP} \geq 2 \qquad \forall S \subset V, S \neq \emptyset. \tag{4.2}$$

It means that, for each non-empty cut-set $(S, V \setminus S)$, at least 2 arcs are required to connect the partitions S and $V \setminus S$, otherwise it would be impossible to get a Hamiltonian cycle, which is clearly an underlying condition for an integer feasible solution to be found. After these further cuts are added to the cut pool, CPLEX will autonomously apply these new constraints to the continuous relaxation and solve it again.

Notice that, when we obtain a new integer solution x^*, instead of directly updating the incumbent, all the steps we have introduced in section 4.2 are performed exactly as previously described, so basically the Branch & Cut method works as a subroutine of this method.

4.3.1 Variants of Advanced Branch & Cut

Depending on the frequency we want the Advanced Branch & Cut callback to be called, once CPLEX has computed the optimal solution of the continuous relaxation, we have developed 4 variants of such method:

- *ADVBC_ROOT* → the callback is called only until CPLEX is solving the LP problem associated to the root node of the branching tree;
- *ADVBC_DEPTH_5* → the callback is called until CPLEX is solving the LP problems associated to the nodes of the branching tree of depth up to 5;
- *ADVBC_PROB_50* → the callback can be called at any node with a 50% probability;
- *ADVBC_PROB_10* → the callback can be called at any node with a 10% probability.

The objective of all these variants is to avoid adding SECs to all nodes, which could lead to lots of overhead, while also to prevent, especially at the beginning, to generate useless branches and nodes in the branching tree.

The results of the *tuning test* among these 4 variants of the Advanced Branch & Cut method are shown and discussed in section 8.2.

5

Matheuristics

Matheuristics are optimization algorithms made by the interoperation of metaheuristics and mathematical programming techniques. An essential feature is the exploitation in some part of the algorithms of features derived from the mathematical model of the problems of interest.

5.1 Hard-fixing heuristic

The first heuristic approach we implemented is called *hard-fixing*. Being a heuristic, it is no longer looking for globally optimal solutions, but rather it focuses on getting the best solution in terms of cost within a given time limit. On the other hand it is a matheuristic, meaning that CPLEX still plays a key role in it. A so-called *black box*, which from now on we will refer to as *TSP solver*, is used to improve an integer feasible solution of a given model. This TSP solver is nothing more than one of the exact algorithms we introduced in chapter 4.

The starting point of hard-fixing is, once again, the Basic model. The TSP solver solves it a first time with a small time limit or a node limit. In our implementation, we decided to set the node limit to 0, meaning that the TSP solver is going to process only the root node of the branching tree and then it returns an integer feasible solution x^H, which is the reference solution (also called *incumbent*).

This reference solution is clearly a tour in our graph G, but it is not the optimal one. To move towards the optimum, the hard-fixing approach consists in fixing a certain number of edges of the tour associated to x^H, by changing the lower bound of the respective variables of the ILP model. More precisely, if an edge $e = (i, j)$ is such that $x_e^H = 1$ and e has to be fixed, then the model is going to be modified with: $1 \leq x_{ij} \leq 1$.

Then the updated model is solved by the TSP solver, this time with a small time limit (e.g. 60 seconds) and with the current incumbent provided as the best integer solution from which to start from (this is a well known trick called *warm start*). The objective is to improve such incumbent by exploring its neighborhood, which is made up of all the feasible solutions with a fraction of fixed edges.

Once the small time limit expires, the model is modified again to unfix all the edges. If the new solution returned by the TSP solver is better that the reference one, then the incumbent is updated. Starting from the edge fixing phase, the previous steps are repeated for an amount of time that is provided in input as a global time limit (e.g. 30 minutes). The pseudo-code of the whole procedure is summarized in Algorithm 2.

5.1.1 Implementation details

To decide which edges to fix we simply selected them randomly. Provided as input a parameter k, usually with a value between 0.1 and 0.9, each edge is fixed with a probability of k. Since the value of k is a crucial hyperparameter, we implemented 4 versions of hard-fixing: the first three are characterized by a value of k set respectively to 0.5, 0.7 and 0.9. The last one starts with k set to 0.9, but then decreases to 0.7 and eventually 0.5, each time an improvement of the incumbent does not occur. This last variant of hard-fixing approach is represented in Figure 5.1.

Algorithm 2: Hard_Fixing($G, k, global\_timelimit, small\_timelimit$)

```
Result: Solution (possibly good) for the TSP problem.
*build Basic model*;
*set TSPSolver nodelimit to 0*;
x^H ← TSPSolver();
while global_timelimit > 0 do
    *set current incumbent to x^H*;
    *set TSPSolver timelimit to small_timelimit*;
    for each e ∈ E do
        if x_e^H > 0.5 and RANDOM([0,1]) ≤ k then
            *fix e*;
        end
    end
    x_new^H ← TSPSolver();
    for each e ∈ E do
        *unfix e*;
    end
    if cost(x_new^H) < cost(x^H) then
        x^H ← x_new^H;
    else
        *change k according to the policy used*;
    end
    *update global_timelimit*;
end
return x^H;
```

5.2 Soft-fixing heuristic

The second matheuristic we implemented is called *Local Branching*, but we usually refer to it as *soft-fixing* because, while most of its steps are the same of hard-fixing, its main difference consists in the fact that the edges of the reference solution x^H are fixed in a *softer* way.

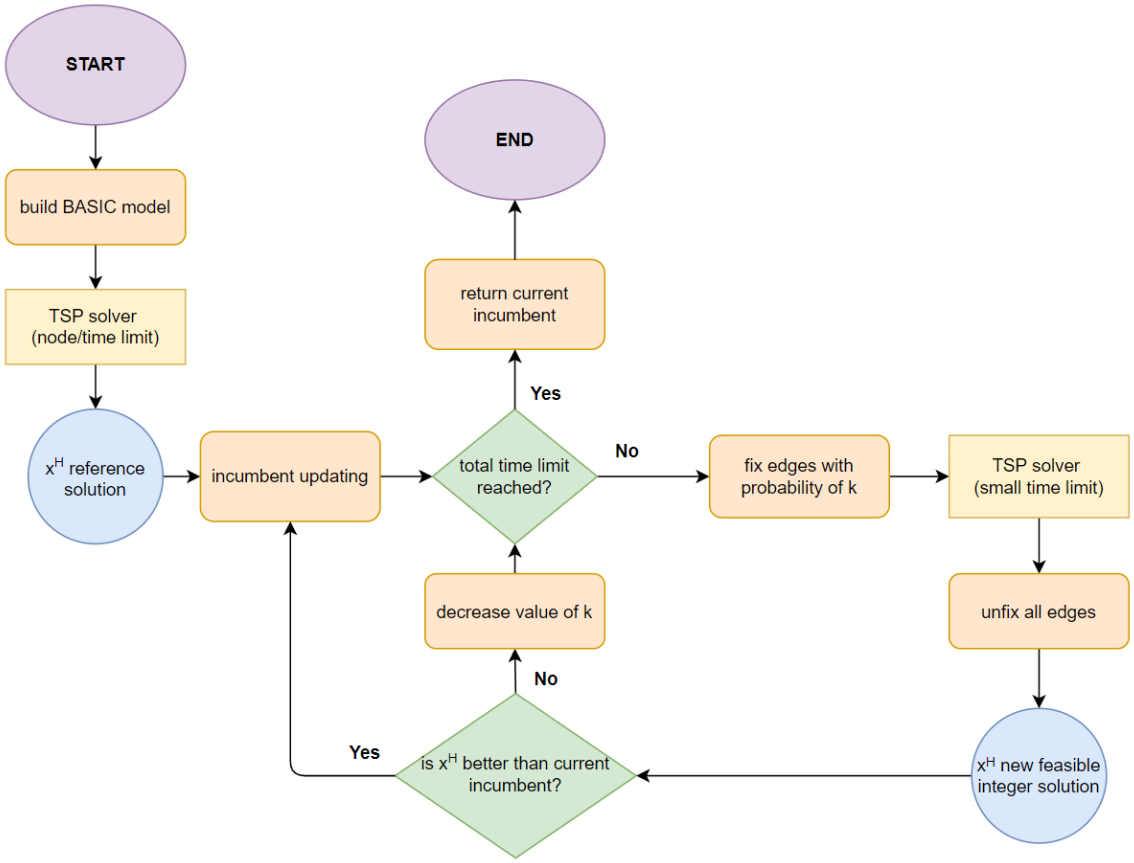


Figure 5.1: Flow Chart of the Hard-fixing heuristics.

To such x^H we associate a neighborhood N with a non-negative radius k defined as

$$N = \{x \text{ feasible solution for TSP such that } H(x, x^H) \leq k\}, \tag{5.1}$$

where H is the Hamming distance between two vectors in $\{0,1\}^{|E|}$, which indicates the number of different bits they have and that is formally defined as follows:

$$H(x, x^H) = \sum_{j:x_j^H=1} (1 - x_j) + \sum_{j:x_j^H=0} x_j. \tag{5.2}$$

If both x and x^H are feasible solutions for the TSP problem, then the number of their elements set to 1 (namely edges belonging to such solutions) must be $n = |V|$, thus

$$\sum_{e \in E} x_e = \sum_{e \in E} x_e^H = n.$$

As a consequence, the Hamming distance can be computed by taking into account only the differences in the bits equal to 1:

$$H(x, x^H) = \sum_{e:x_e^H=1} (1 - x_e) = n - \sum_{e:x_e^H=1} x_e. \tag{5.3}$$

In order for the generic solution x to belong to the neighborhood N, it must hold the condition expressed in (5.1) and therefore

$$\begin{aligned} H(x, x^H) \leq k &\iff n - \sum_{e:x_e^H=1} x_e \leq k \\ &\iff \sum_{e:x_e^H=1} x_e \geq n - k. \end{aligned} \tag{5.4}$$

The inequality (5.4) represents a constraint which narrows the search for TSP solutions down to the neighborhood N of radius k of the incumbent x^H. This constraint is added to the ILP model before calling the TSP solver with the small time limit set. The TSP solver returns a new solution which is possibly the locally optimal solution of N (that is why the method is called Local Search) and the incumbent is updated. The entire procedure is then iterated until the global time limit is reached.

If there is an iteration which leads the new solution obtained by the TSP solver not to improve, then the radius k of N is increased in order to expand the search pool and allow for a wider range of solutions to be evaluated. In our 4 implementations the value of k starts from 3, 5, 7 or 9 and can increase up to 10. The chart in Figure 5.2 and the pseudo-code in Algorithm 3 provide a summary of this soft-fixing approach.

Algorithm 3: Soft_Fixing($G, global\_timelimit, small\_timelimit$)

Result: Solution (possibly good) for the TSP problem.

build Basic model;
set TSPSolver nodelimit to 0;
$x^H \leftarrow TSPSolver()$;
$k \leftarrow 3$;
while $global\_timelimit > 0$ **do**
    set current incumbent to x^H;
    set TSPSolver timelimit to $small\_timelimit$;
    add constraint (5.4) to the model;
    $x^H_{new} \leftarrow TSPSolver()$;
    remove last constraint added;
    if $cost(x^H_{new}) < cost(x^H)$ **then**
        $x^H \leftarrow x^H_{new}$;
    else
        if $k < 10$ **then**
            $k \leftarrow k + 1$;
        end
    end
    update $global\_timelimit$;
end
return x^H;

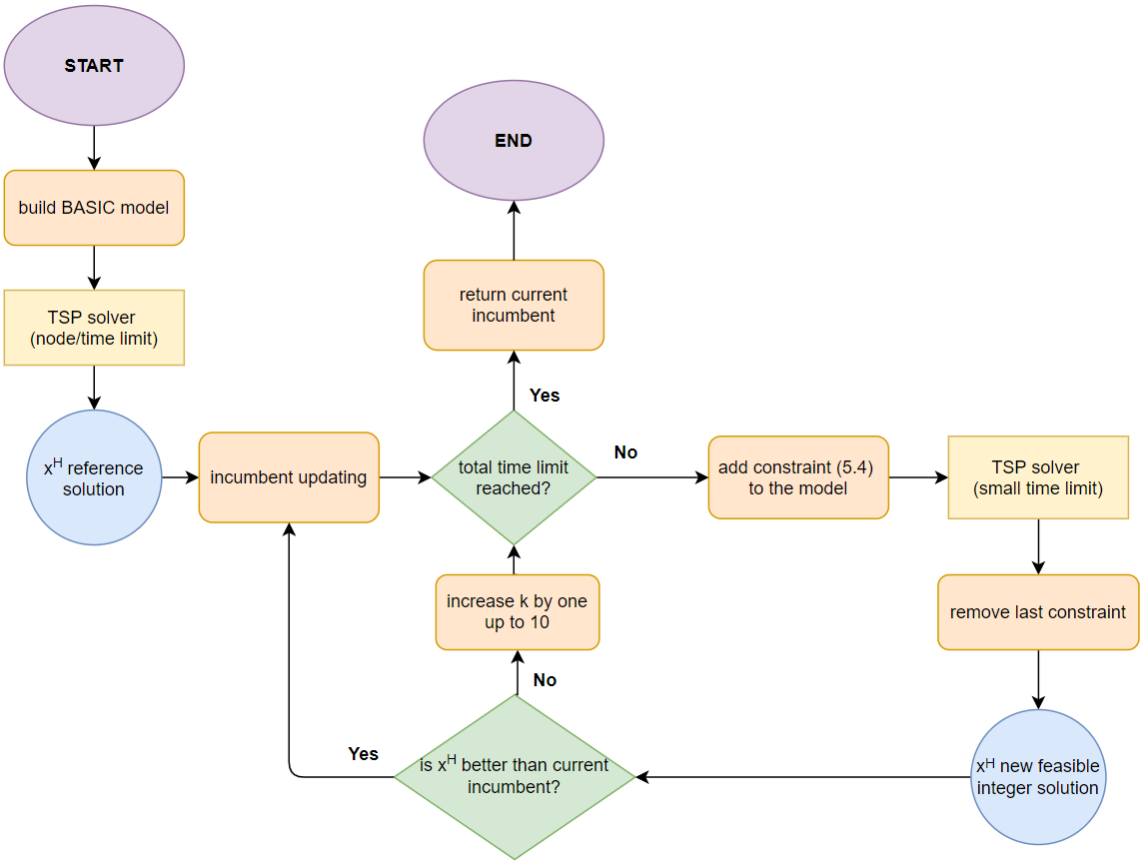


Figure 5.2: Flow Chart of the Soft-fixing heuristics.

6

Constructive heuristics

A constructive heuristic is a type of heuristic method which starts with an empty solution and repeatedly extends the current solution until a complete solution is obtained. It differs from local search heuristics which start with a complete solution and then try to further improve the current solution via local moves.

6.1 Nearest Neighbour heuristic

The Nearest Neighbour (NN) algorithm was one of the first algorithms used to solve the travelling salesman problem starting from an empty solution. [6]

At first the algorithm selects an arbitrary node, sets it as the current node u and marks it as visited. Then, it finds out the shortest edge (u, v) connecting the current node u to another node v and adds it to the final solution. Such node v is set as the current node u and marked as visited. These steps are repeated until all nodes of the graph are visited, then the edge between the first and the last visited node is included to complete the tour.

We say that the NN algorithm implements a *greedy approach*, due to the fact that in each iteration it is going to select the nearest node as the next node of the tour, performing a so-called greedy-choice.

The NN algorithm is trivial to implement and executes quickly, but it can easily miss shorter routes which are usually noticed with human insight, due to its greedy nature. As a general guide, if the last few stages of the tour are comparable in length to the first stages, then the tour is reasonable; if they are much greater, then it is likely that much better tours exist. As a result, in the worst case the algorithm results in a tour that is much longer than the optimal tour.

In our implementation, we are going to select one by one all the possible n nodes of V as the first node of the tour, thus getting n different solutions. The final solution is the one with the lowest cost. The pseudo-code of the NN algorithm can be found in Algorithm 4 and shows clearly that it is $O(n^2)$ in the worst case.

Algorithm 4: Nearest_Neighbour($G = (V, E)$)

Result: Solution (possibly good) for the TSP problem.

$min\_cost \leftarrow \infty$;
for each $e \in E$ **do**
    $x_e^H \leftarrow 0$;
    $x_e \leftarrow 0$;
end
for $start\_node \leftarrow 1$ to n **do**
    $u \leftarrow start\_node$;
    mark u as visited;
    for $i \leftarrow 1$ to $n - 1$ **do**
        find shortest $e = (u, v) \in E$ with v not already visited;
        $x_{uv} \leftarrow 1$;
        mark v as visited;
        $u \leftarrow v$;
    end
    $x_{ustart\_node} \leftarrow 1$;
    if $cost(x) < min\_cost$ **then**
        $min\_cost \leftarrow cost(x)$;
        $x^H \leftarrow x$;
    end
end
return (x^H);

6.2 Extra Mileage approach

In this approach the final solution is built starting from a partial one, that is a cycle of the graph G which does not pass through all of its nodes. A classic example of partial solution, and also the one we chose to start from in this algorithm, is the *convex hull* of the nodes of the graph.

The convex hull of a set of nodes V is the smallest convex set that contains V. It corresponds to a partial cycle $\hat{\gamma}$ touching the external (in a geometrical sense) nodes of V.

Starting from $\hat{\gamma}$, an iteration of the Extra Mileage algorithm consists in searching among the nodes not included in $\hat{\gamma}$ to find the one that leads to the lowest extra cost (or *extra mileage*) with respect to the edges of the given partial solution.

Let $V_{\hat{\gamma}}$ and $E_{\hat{\gamma}}$ be the set of nodes and edges which belong to the partial solution. The extra cost $\delta(u, v, w)$ is defined as the increase of cost of the partial solution $\hat{\gamma}$ that occurs once the node w is added to the solution by substituting the edge (u, v) with the pair of edges

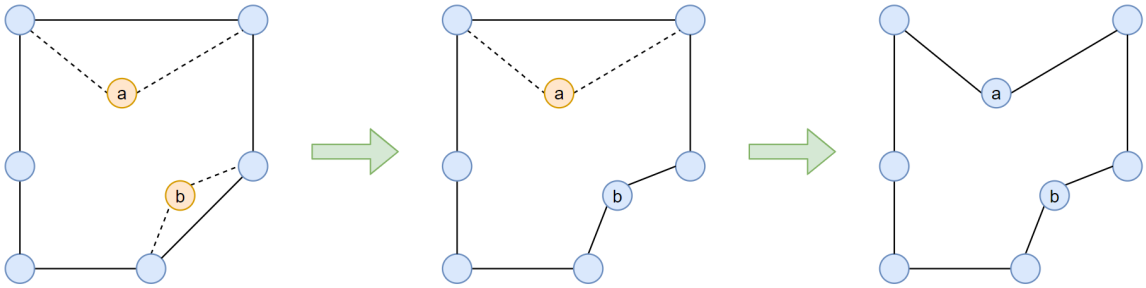


Figure 6.1: Starting from the tour of blue nodes, first we add node b because it leads to a lower *extra mileage* compared to a, then also node a is included to complete the tour.

(u, w) and (w, v). Specifically, it is computed as follows:

$$\delta(u, v, w) = c_{uw} + c_{wv} - c_{uv} \qquad \forall u, v, w \in V : (u, v) \in E_{\hat{\gamma}}, w \in V \setminus V_{\hat{\gamma}}. \tag{6.1}$$

Notice that, thanks to the triangle inequality, which holds true due to the fact that c is a metric function, δ is always positive. Here there is the main idea of the algorithm, which is also explained in Algorithm 5: starting from $\hat{\gamma}$, at each iteration we are going to add to the current partial solution the node which is not already part of the solution and leads to the minimum extra cost δ. Figure 6.1 provides a graphic representation of a couple iterations. This procedure is performed until all nodes of V become part of the solution and so, at the end, we clearly get a feasible solution for the TSP problem.

Algorithm 5: Extra_Mileage($G = (V, E)$)

Result: Solution (possibly good) for the TSP problem.
$\hat{\gamma} \leftarrow ConvexHull(V)$;
Let $V_{\hat{\gamma}}$ and $E_{\hat{\gamma}}$ be the set of nodes and edges in $\hat{\gamma}$;
while $|V_{\hat{\gamma}}| \neq n$ **do**
  $(u, v, w) \leftarrow \arg\min\{\delta(u, v, w) : u, v, w \in V : (u, v) \in E_{\hat{\gamma}}, w \in V \setminus V_{\hat{\gamma}}\}$;
  $V_{\hat{\gamma}} \leftarrow V_{\hat{\gamma}} \cup \{w\}$;
  $E_{\hat{\gamma}} \leftarrow E_{\hat{\gamma}} \setminus (u, v) \cup \{(u, w), (w, v)\}$;
  $\hat{\gamma}$ is modified accordingly;
end
for each $e \in E$ **do**
  if $e \in E_{\hat{\gamma}}$ **then**
    $x_e^H \leftarrow 1$;
  else
    $x_e^H \leftarrow 0$;
  end
end
return (x^H);

6.3 GRASP approach

The Greedy Randomized Adaptive Search Procedure (also known as GRASP) typically consists of iterations made up from successive constructions of a greedy randomized solution. The main idea is to deviate from the deterministic greedy algorithm of the previous sections by introducing some randomized decisions.

We implemented two different versions of this approach:

- *HEUR_GRASP_GREEDY* algorithm → variation of the basic NN algorithm, where the nearest node is not always the one selected to be added to the solution. Instead, sometimes the second or the third one are randomly selected. At each iteration of the NN algorithm, the starting node is chosen randomly among the n possible choices. The only hyperparameter of this algorithm is the probability of discarding a node and, as a result, to take the second or third node drawn, which we set to 10%;
- *HEUR_GRASP_EXTRA_MILEAGE* algorithm → variation of the Extra Mileage algorithm, where in a small amount of iterations (that we set to 1%), instead of selecting the triple of nodes which leads to the minimum extra cost, we draw such triple randomly.

While the basic NN and Extra Mileage algorithms greedy nature makes them unsuitable to provide good reference solutions for other heuristics (e.g. 2-opt refinement, section 6.4.1), the GRASP approach leverages their high performance in terms of number of solutions generated within a limited time interval to return a much better solution.

The main advantage of both versions of the GRASP approach is that the randomness it provides allows such algorithms to run for as long as it is specified by a given time limit and return multiple slightly different solutions. Of course, their output is the best among these solutions.

6.4 Refinement heuristics

The objective of refinement heuristics is to improve a given reference solution x^H and take it closer to the global optimum.

One of the most known techniques is the so called k-opt heuristic, which consists in removing k mutually disjoint edges from the reference solution and then reassemble the remaining fragments into a tour, leaving no disjoint subtours to get a feasible and possibly improved solution for the TSP. In the next section we will introduce the 2-opt heuristic.

6.4.1 2-opt refinement heuristic

Given the reference solution x^H, which could be obtained by using some constructive heuristic, the 2-opt technique consists in a series of iterations in which 2-opt moves are performed on x^H until no further improvement is possible.

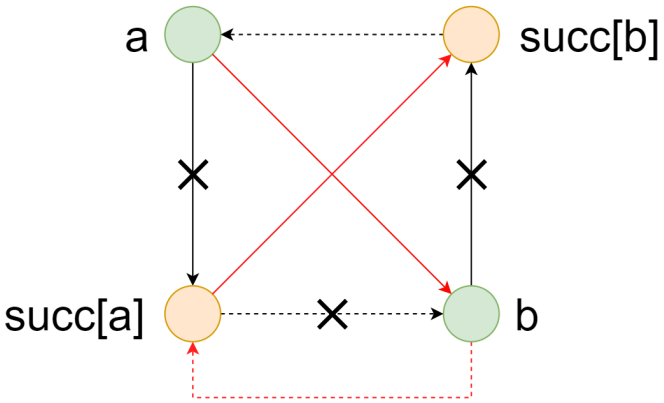


Figure 6.2: Scheme of a 2-opt move: the crossed edges are replaced by the red ones. Notice that the path from node b to succ$[a]$ has been inverted.

Formally, each iteration searches among all pairs of nodes $u, v \in V$ the one which provides to x^H the best possible improvement in terms of cost when we replace the edges $(u, u'), (v, v')$ with $(u, v), (u', v')$, where u', v' refer to the successors of u, v in x^H.

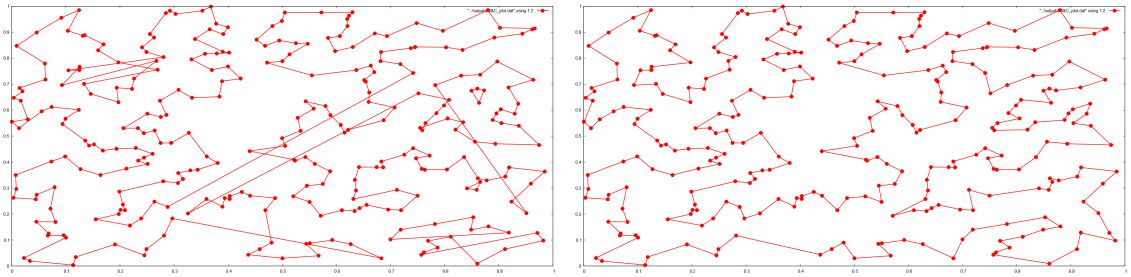
The improvement provided by a 2-opt move is computed as

$$\Delta(u, v) = c_{uv} + c_{u'v'} - c_{uu'} - c_{vv'} \qquad \forall u, v \in V : x^H_{uu'} = 1, x^H_{vv'} = 1, u \neq v', v \neq u'. \tag{6.2}$$

In detail, the algorithm looks for the most negative $\Delta(u, v)$, which corresponds to the pair of nodes that leads to the greatest improvement to the current solution. If no negative $\Delta(u, v)$ is found, then no improvement is possible with a 2-opt move and therefore we say that the algorithm has reached a locally optimal solution x (also called *local optimum*), which is returned as output. Notice that such local optimum will never have crossing edges, because when a solution shows pairs of crossing edges, for the triangle inequality property it can be proven to be not optimal, meaning that we can always find a way to remove the crossing edges and reconnect their extreme nodes so as to obtain a negative value of Δ and thus decrease the overall cost of the path.

An example of an instance where the crossing edges have been removed applying 2-opt refinement can be found in Figure 6.3, while Algorithm 6 provides a summary of the procedure described above.

In our implementation the reference solution x^H is represented with an array of successors *succ*. A 2-opt move requires time $O(n)$ since we have to reverse all successors between v and u' (see Figure 6.2). Searching for the pair u, v among all pairs of nodes $u, v \in V$ requires time $O(n^2)$. As a consequence, the 2-opt algorithm is $O(hn^2)$, where h is the number of iterations performed. If we start with a reference solution obtained by the NN algorithm the number of 2-opt moves is on average $O(n)$. [17]



(a) **(b)**

Figure 6.3: On the left (a), an example of tour with crossing edges. On the right (b), a new solution for the same instance after 2-opt refinement.

Algorithm 6: 2_opt$(G = (V, E), x^H)$

Result: Solution (possibly good) for the TSP problem.

$x \leftarrow x^H$;

$delta\_cost \leftarrow -\infty$;

while $delta\_cost < 0$ **do**

  $(u, v) \leftarrow \arg\min\{\Delta(u, v) : u, v \in V\}$;

  $delta\_cost \leftarrow \Delta(u, v)$;

  $x \leftarrow 2\_opt\_move(G, x, u, v)$;

end

return (x);

6.4.2 Multi-start approach

In the last section we introduced the concept of local optimum as a solution reached at the end of the 2-opt algorithm and no longer improvable by any 2-opt move. The idea behind the multi-start method is to explore in a given time limit multiple local optimum solutions for the TSP problem and return the best one found. Notice that to do this at each iteration the 2-opt algorithm needs a different reference solution x^H as a starting point in order to reach at the end a (possibly new) local optimum.

Different starting solutions can be obtained using as many different constructive heuristics, random solutions or, as we did in our implementation, using the GRASP method, which includes randomized decisions, with a random small time limit.

A detailed description of the multi-start approach is reported in Algorithm 7.

Algorithm 7: Multistart($G = (V, E), timelimit$)

Result: Solution (possibly good) for the TSP problem.

$min\_cost \leftarrow \infty$;

while $timelimit > 0$ **do**

build a reference solution x^H using a constructive randomized heuristic, or a random solution;

$x \leftarrow 2\_opt(G = (V, E), x^H)$;

if $cost(x) < min\_cost$ **then**

$min\_cost \leftarrow cost(x)$;

$x_{best} \leftarrow x$;

end

update $timelimit$;

end

return (x_{best});

6.4.3 Application inside Branch & Cut callback

The same 2-opt refinement heuristic can be used as an additional step inside the callback of the Branch & Cut method (described in section 4.2). The main idea is to use the original method until we get a feasible solution consisting in a single tour, then we submit this solution to the 2-opt algorithm and instruct Cplex to employ the resulting handcrafted improved solution, by using the *CPXcallbackpostheursoln* function. This should speed up the optimization process, due to the fact that it helps removing crossing edges, which, as previously mentioned in 6.4.1, are certainly degrading the quality of the current solution.

7

Metaheuristics

A metaheuristic is a higher-level procedure designed to generate or select a heuristic that may provide a sufficiently good solution to an optimization problem, especially when dealing with incomplete information or limited computation capacity. They manage to sample a subset of solutions which would otherwise be too large to be completely enumerated or explored. Furthermore, metaheuristics do not guarantee that a globally optimal solution can be found on some classes of problems.

7.1 Variable Neighborhood Search (VNS)

In section 6.4.1 we introduced the concept of local optimum, that is a feasible solution x that is no longer improvable using a refinement heuristic like the 2-opt algorithm. A common metaphor compares such local optimum to a solution x locating itself in a valley, where the cost of the solution is linked to the contour (or the altitude) of a mountain chain.

Variable Neighborhood Search (VNS) is a metaheuristic employing local search methods used for mathematical optimization. Their name is due to the fact that research is restricted to a neighborhood: starting from a potential solution to a problem and checking its immediate neighbors, which are the solutions that are similar except for very few minor details, they are expected to find an improved solution between them. All the local search methods share a tendency to become stuck in suboptimal regions (valleys) or on plateaus where many solutions are equally fit.

The basic idea of the VNS is to systematically change the neighborhood both within a *descent phase* to find a local optimum and in the subsequent *perturbation phase* to get out of the corresponding valley.

In this section we introduce the basic strategy of VNS, where the descent phase corresponds to the execution of the 2-opt heuristic and the perturbation phase to the application of a so-called *k-opt kick*.

The general scheme of the VNS heuristic is the following: given a fixed global time limit, a VNS iteration consists of a 2-opt execution in which a local optimum is reached, followed by a k-opt kick to worsen the solution and escape from that local optimum. Once the global time limit expires, the best local optimum is returned as output of the VNS heuristic. This procedure is also summarized in Algorithm 8.

A k-opt kick (worsening move) is performed by removing randomly k edges from the local optimum solution x and then reconnect them in a deterministic way.

Further details about our implementation of the VNS and an overview of the k-opt kick

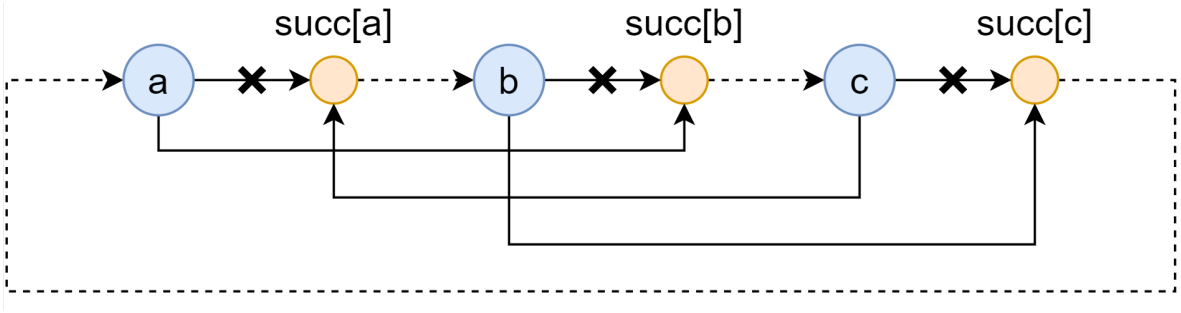


Figure 7.1: Scheme of 3-opt move reconnection pattern.

will be presented in section 7.1.1.

Algorithm 8: VNS($G = (V, E), k, global\_timelimit$)

Result: Solution (possibly good) for the TSP problem.

build a reference solution x^H using a constructive heuristic, or a random solution;

$x_{best} \leftarrow x \leftarrow x^H$;

$min\_cost \leftarrow \infty$;

while $global\_timelimit > 0$ **do**

  $x \leftarrow 2\_opt(G, x)$;

  if $cost(x) < min\_cost$ **then**

    $min\_cost \leftarrow cost(x)$;

    $x_{best} \leftarrow x$;

  end

  $x \leftarrow k\_opt\_kick(x, k)$;

  update $global\_timelimit$;

end

return (x_{best});

7.1.1 k-opt moves overview

In the perturbation phase of our implementation, we decided to apply a 3-opt, 5-opt or 7-opt worsening move with a 50%, 30% and 20% probability respectively. The choice to use up to 7-opt moves is due to the fact that they lead to more *powerful* kicks to the local optimum, which are often needed to escape rather deep valleys. Only k-opt moves with odd values of k have been implemented, because they are easier to implement and faster to be applied, because it is not needed to reverse the list of successors.

The k edges we replace are chosen randomly, but the way in which they are reconnected is deterministic: these configurations are represented schematically in Figures 7.1, 7.2, 7.3.

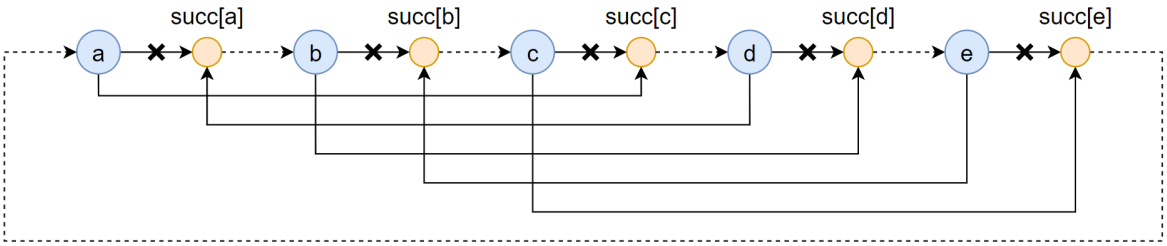


Figure 7.2: Scheme of 5-opt move reconnection pattern.

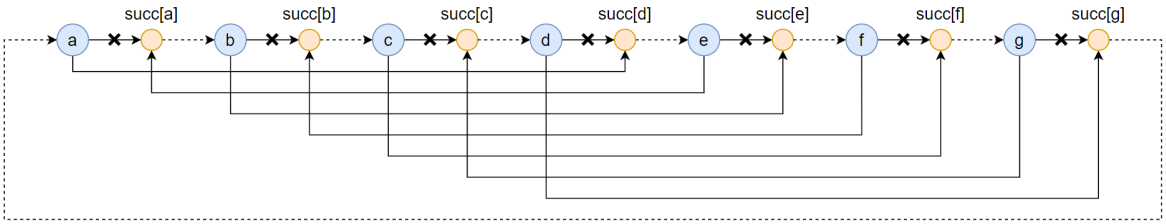


Figure 7.3: Scheme of 7-opt move reconnection pattern.

7.2 Tabu Search

Tabu search is a metaheuristic search method which enhances the performance of local search by relaxing its basic rules. First, at each step worsening moves can be accepted if no improving move is available (e.g. when the search is stuck at a narrow local minimum). In addition, prohibitions (henceforth the term *tabu*) are introduced to prevent the search from coming back to previously visited solutions.

The implementation of tabu search uses data structures that describe the visited solutions or user-provided sets of rules. If a potential solution has been previously visited within a certain short-term period or if it has violated a rule, it is marked as "tabu" (forbidden) so that the algorithm is not allowed to consider the same candidate solution repeatedly.

In our implementation for the TSP problem, a locally optimal solution is reached using the 2-opt heuristic (like in the VNS) while a worsening move, to escape from the valley, always consist in a 2-opt move (*kick*) which chooses two random non-consecutive edges to swap.

Each time a worsening move is performed, the correspondent edges are declared as tabu. As a consequence, the subsequent improving moves cannot touch such edges, thus the algorithm will not fall in a loop by landing on the same local optimum solution. The algorithm will continue worsening the solution until a not tabu improving move is possible. In such case, a possibly new local optimum will be found using, once again, the 2-opt heuristic.

As on output, the tabu search returns the best local optimum solution found within the global time limit (as it happens in the VNS).

From a practical point of view, the tabu moves are managed by using an array *tabu* of size n in which each element represents a node of the graph. When an edge $e = (i, j)$ has to be declared tabu at the iteration number h, the algorithm sets $tabu[i]$ and $tabu[j]$ equal to h. In such a way, all edges connected to the nodes i and j are marked as tabu/forbidden.

Moreover, to be considered as tabu a node u must satisfy the following inequality

$$current\_iteration - tabu[u] \leq tenure, \tag{7.1}$$

where *current_iteration* is the number of the current iteration of the algorithm and *tenure* is a hyperparameter used to specify how many iterations have to be executed in order to *free* a node from the tabu list. In other words, a node which has been declared tabu will stay tabu for *tenure* iterations. We chose a value of the *tenure* parameter which changes over time: specifically, it alternates between $n/10$ and $n/50$ every 500 iterations. A summary of the entire procedure of tabu search can be found in Algorithm 9.

Algorithm 9: Tabu($G = (V, E), global\_timelimit, tenure$)

Result: Solution (possibly good) for the TSP problem.
build e reference solution x^H using a constructive heuristic, or a random solution;
$x_{best} \leftarrow x \leftarrow x^H$;
$min\_cost \leftarrow \infty$;
$curr\_iter \leftarrow 1$;
for $i \leftarrow 1$ to n **do**
  $tabu[i] \leftarrow -1$;
end
while $global\_timelimit > 0$ **do**
  *perform the usual 2-opt procedure with a tabu check (7.1) on the first move, if the move is forbidden then no improvement is allowed, otherwise a new local optimum is reached; *curr_iter* is increased by one at each 2-opt move*;
  $x \leftarrow 2\_opt\_tabu(G, x, tabu, tenure, curr\_iter)$;
  if $cost(x) < min\_cost$ **then**
    $min\_cost \leftarrow cost(x)$;
    $x_{best} \leftarrow x$;
  end
  $u \leftarrow RANDOM(V)$;
  $v \leftarrow RANDOM(V)$;
  $tabu[u] \leftarrow tabu[v] \leftarrow curr\_iter$;
  $x \leftarrow 2\_opt\_move(x, u, v)$;
  $curr\_iter \leftarrow curr\_iter + 1$;
  *update *global_timelimit**;
end
return (x_{best});

7.3 Genetic algorithms

Genetic algorithms are metaheuristics based on the concept of natural selection, which use mechanisms inspired by biological evolution, such as reproduction, mutation, recombination,

and selection. Candidate solutions to the optimization problem play the role of individuals in a population, and the fitness function determines the quality of the solutions.

The main steps of a single iteration (*epoch*) of our genetic algorithm implementation for the TSP are the following:

1. Generation of the starting population (only at the first epoch): a number *pop_size* of random feasible solutions is generated and their fitness value is computed.

2. Parents selection: *pop_size*/10 pairs of solutions are randomly selected to reproduce. Solutions with higher fitness are favored, so that the population quality is going to improve over time;

3. Offspring generation: for each pair of selected parent solutions, one child solution is generated by merging their chromosomes, so as to simulate sexual reproduction. More details on that in section 7.3.1;

4. Population size management: *pop_size*/10 solutions are killed to compensate for the new child solutions and keep the population size to a fixed amount. The best solution (called *champion*) of the last epoch cannot die and it may rarely happen that the killed population members are child solutions of the same epoch (simulates infant mortality);

5. Mutations application: occasionally, a random number of mutations is applied to randomly selected solutions of the population, with the only exception of the current champion. Further details on that are in section 7.3.1.

Analogously to VNS and tabu search, genetic algorithms run multiple iterations and, at the end, return the best champion solution, which usually comes from one of the latest epochs.

7.3.1 Implementation details

The *chromosomes* we mentioned in previous section are simple data structures which consist in the list of nodes of the solution they represent. During the merging process the parents' chromosomes are split in two segments, where the *cutting point* is chosen randomly, and the child solution chromosome is obtained concatenating the first section of one parent and the second section of the other. Of course, this concatenation is followed by a *fixing* phase to make sure that such chromosomes represent a feasible solution.

We want to apply a set of mutations only when the spread between the worst and best solution is too low compared to that of the first epoch: in our implementation we set this hyperparameter to 10%. The number of random mutations applied is limited to the number n of nodes and a mutation consists in swapping 2 random nodes of a solution.

We implemented two variants of the genetic algorithm for TSP: *HEUR_GENETIC* is exactly as described in previous section, while *HEUR_GENETIC_2_OPT* employs a *ratio_2_opt* parameter, which indicates the fraction of both starting and new child solutions to which we apply 2-opt refinement with a small time limit. We set this hyperparameter to 10% and said time limit to 0.5 seconds. The other difference is that the population size

pop_size is set to 1000 for the original HEUR_GENETIC, while it is lowered to 100 for the HEUR_GENETIC_2_OPT. The reason is that each 2-opt refinement takes a considerable amount of time and so, in order to compensate, we set it to work with a relatively small amount of solutions, because by design genetic algorithms need to run lots of epochs to provide high-quality solutions.

8

Results and Discussion

As mentioned in section 2.1, to analyze the performance of the algorithms developed we employed a specific *performance profiling tool*. The plots it produces have to be interpreted as a summary of the results of a series of races, where we pretend that such races are the instances and algorithms are the cars taking part in the competition. As a result, the winning algorithms are the ones that won the highest number of races, considering that for exact algorithms the objective is to get to the optimal solution in the shortest possible time, while for heuristics the competitors have to return the best possible solution within a given global time limit. This difference also affects the measures represented in the plots: while the y axis is always about the fraction of instances which an algorithm ends up winning, the x axis indicates the *Time ratio* for exact algorithms and the *Cost ratio* for metaheuristics. These are both tolerance factors, which are useful to record if an algorithm, while not being able to win in a specific instance, ends respectively within a multiple of the time or with a multiple of the solution cost with respect to the actual winning algorithm.

The reason why such a complex method to compare algorithms is required is that, given the complexity and variability from an instance to another that characterizes the TSP, there is no absolute best algorithm, so our objective is to find the one which is the fastest or provides the best solution in most cases.

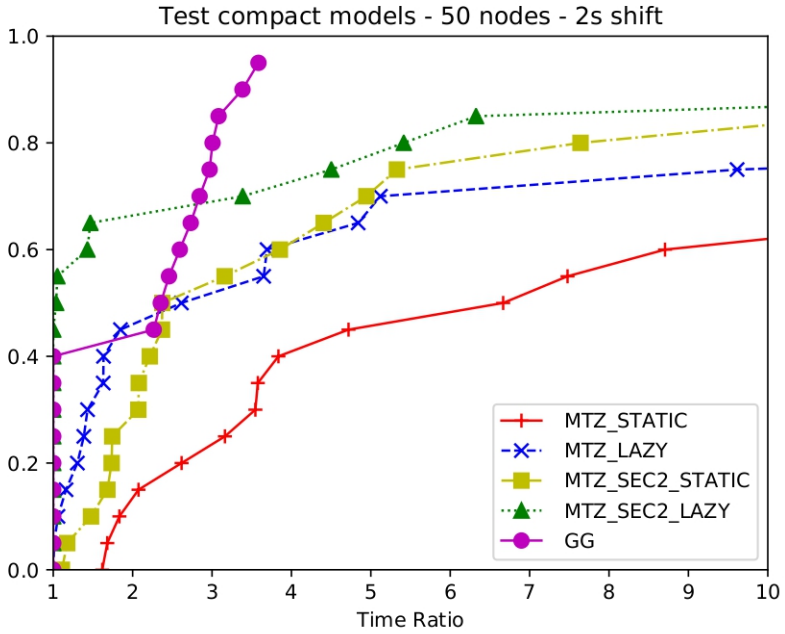
As a general suggestion, the winning algorithm is the one whose plot manages to stay on the left side of the chart for as long as possible. In some cases the plots cross, suggesting that there may be algorithms that usually don't perform best, but lead to more consistent results than others, and so may be preferable for specific applications. In the following sections we are going to discuss such performance profiles for each group of algorithms. All results and relative charts are freely available in our repository under the directory *experimental_results*.

Notice that, in order to allow for test results reproducibility, we run all tests involving CPLEX using it in *deterministic mode* with a fixed seed 123456, meaning that, when solving the same instance, it always follows the same branching path.

We also remark that, while running tests on exact algorithms, we decided to penalize algorithms which reached the 30 minutes time limit in some instances by multiplying their result by a factor of 10, to convey the idea that such a time requirement is not acceptable.

8.1 Compact models comparison

The first test we performed is about the compact models introduced in chapter 3: the models under analysis are all the MTZ variants (see section 3.2.1) and the GG model. Results on a test set of 20 instances of 50 nodes are shown in the following performance profile:

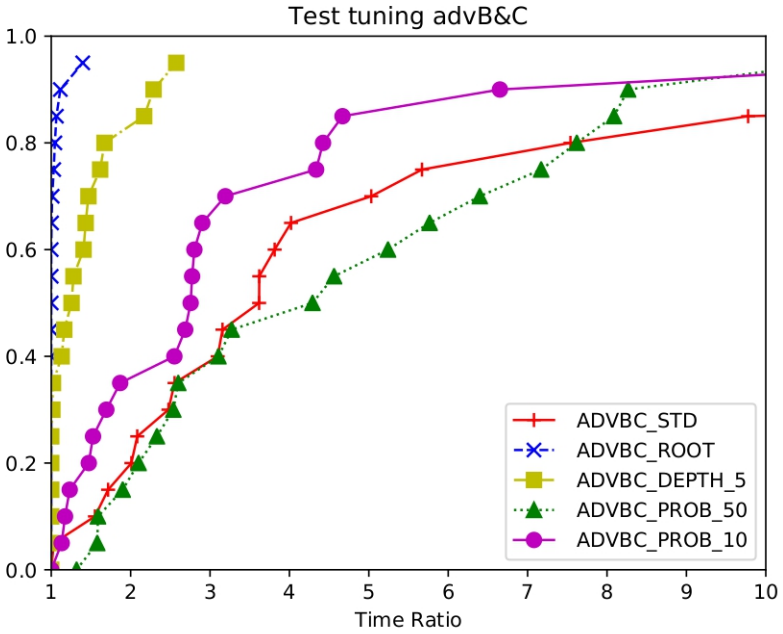


Notice that we applied a 2 seconds shift to make the results more comparable, because while a couple of time limits were reached by MTZ_STATIC and MTZ_LAZY, there are instances where MTZ_SEC2_LAZY required less than a second to reach optimality, leading to other models (especially MTZ_STATIC) suddenly sliding to the right side of the chart. We observe that most of the times the MTZ_SEC2_LAZY model performs significantly better than the others, but there are rare instances where it performs quite bad or even reaches time limit. On the other hand, the GG model never reaches time limit and leads to very consistent results: this is made clear by the fact that its plot is basically vertical, apart from a sharp shift to the right caused by a single instance in which MTZ_SEC2_LAZY performed exceptionally, but in absolute terms the real difference is just about 5 seconds.

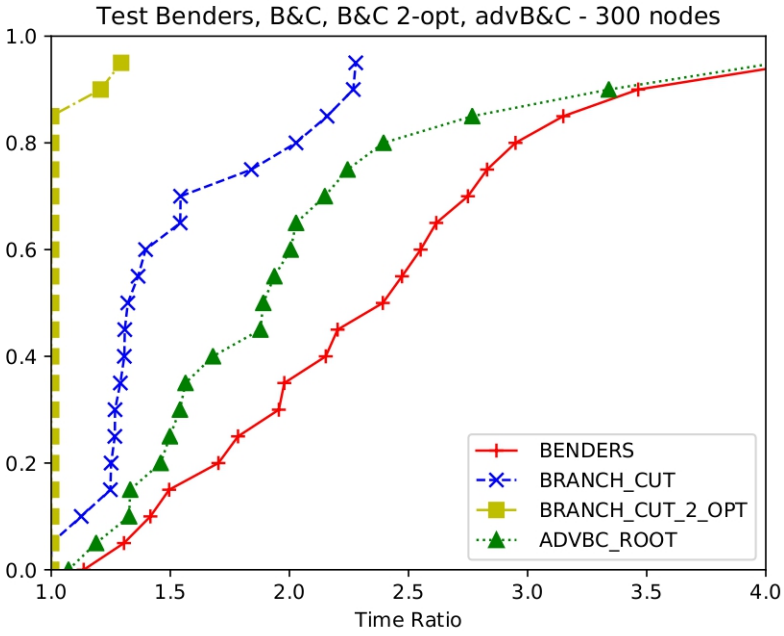
8.2 SECs separation methods comparison

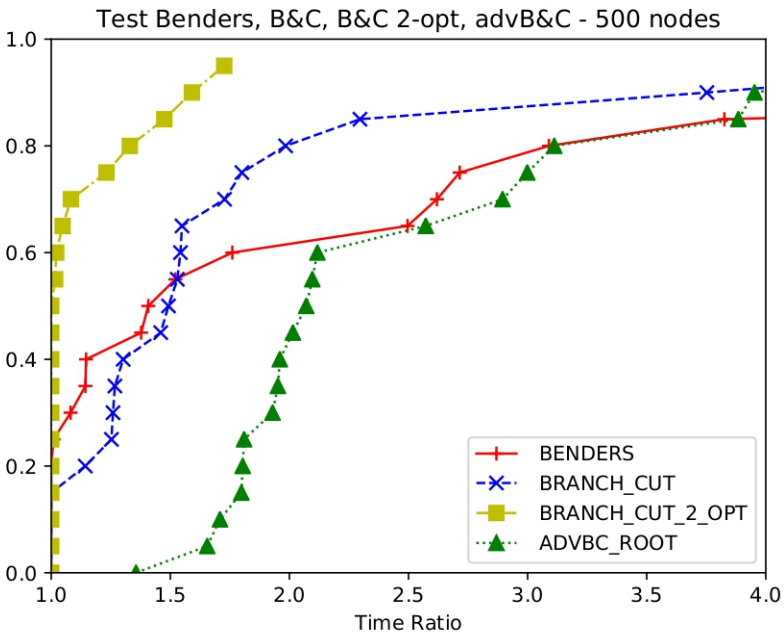
The algorithms introduced in chapter 4 required a two-step analysis: in the first, called *tuning*, we selected the best variant of Advanced Branch & Cut between the ones listed in section 4.3.1, by testing them on the 20 instances (of 300 nodes) of the validation set. The following performance profile clearly states that the winning variant is the ADVBC_ROOT, which is the Advanced Branch & Cut with the callback used only on the root node of the

branching tree (henceforth referred to as Advanced Branch & Cut root).



The second phase consists on the overall test between Benders, Branch & Cut, Branch & Cut with 2-opt (see section 6.4.3) and Advanced Branch & Cut root. We tested them on instances with both 300 and 500 nodes to see if more complex instances could lead to significant differences. From the following pair of charts we notice that Branch & Cut with 2-opt is definitely the best algorithm, while Benders manages to be slightly more consistent and also outperforms Advanced Branch & Cut root when dealing with 500 nodes.

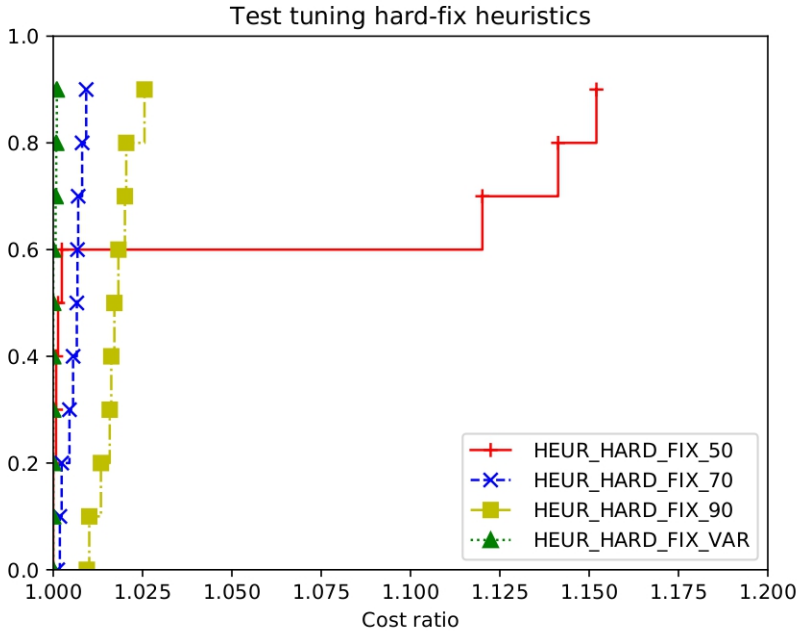


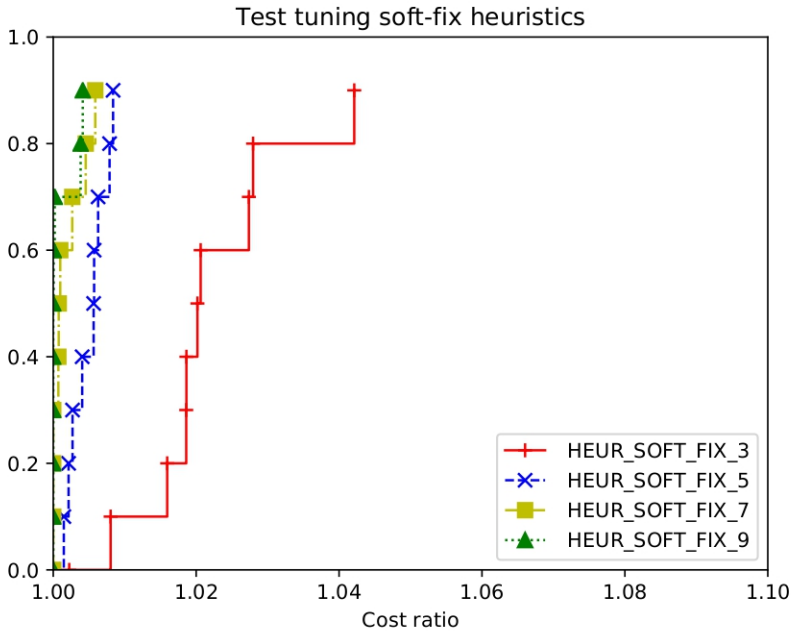


8.3 Matheuristics comparison

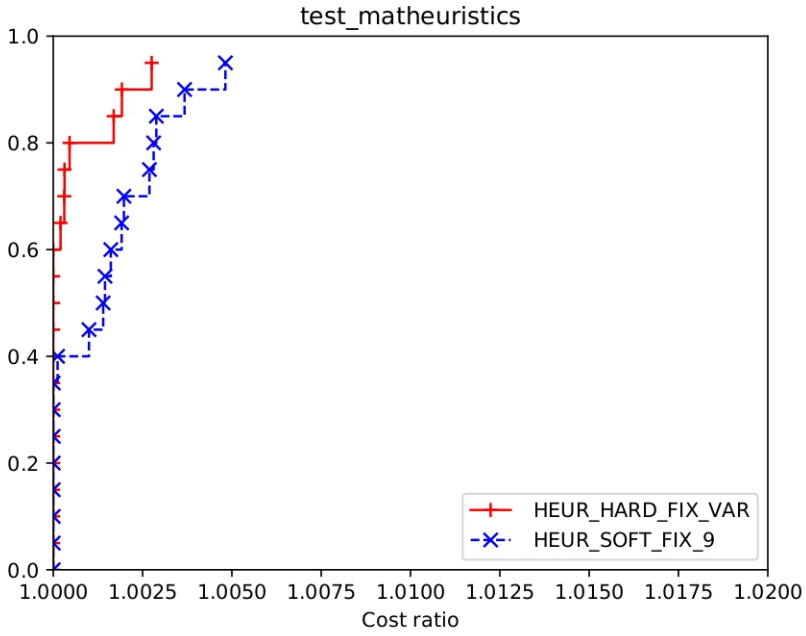
Similarly as before, to compare the performance of the matheuristics we introduced in chapter 5 we start with a tuning test for each algorithm to select the best values of k. These tests are performed on 10 instances of 750 nodes with 15 minutes of time limit and the algorithms employ the Advanced Branch & Cut root as a TSP solver, which is not the best one as proved in the previous section. This is not to be considered an issue since, at least in this first phase, our objective is not to measure performance, but only to select the best value of k.

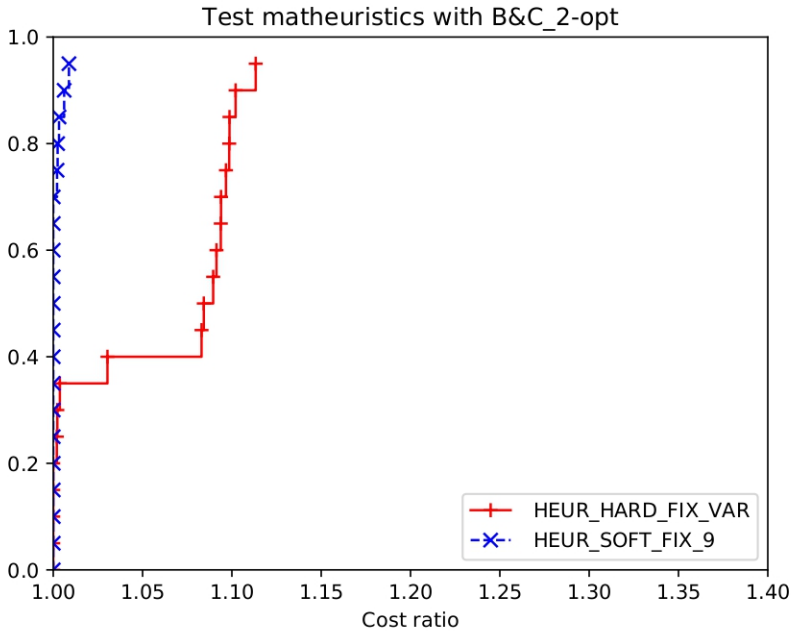
Analyzing the following performance profiles, we can see that HEUR_HARD_FIX_VAR, whose k value gradually decreases from 0.9 to 0.5, is the best variant of the hard-fixing heuristic, while HEUR_SOFT_FIX_9 is the variant to choose for the soft-fixing heuristic. In detail, we notice that also the hard-fixing variant with k set to 0.5 performed well on about 60% of the instances, but fell behind in the remaining part, meaning that the winner ability to change dynamically the percentage of edges to fix is of crucial importance. In soft-fixing it is clear that, given adequately large instances, searching in a small neighborhood (with low value of k) may not be enough to find a competitive solution.





The overall test between HEUR_HARD_FIX_VAR and HEUR_SOFT_FIX_9 has been performed on 20 instances with both 750 and 1000 nodes and a time limit of 30 minutes. The algorithms applied to instances with 750 nodes employ the Advanced Branch & Cut root as TSP solver, while in the second case they use the more performing Branch & Cut with 2-opt. The results reported in the following charts seem conflicting, but it is clear from the reported *Cost ratio* values that in the test on instances with 1000 nodes the difference is much more significant, therefore we can state that the winning method is HEUR_SOFT_FIX_9.





As a final remark, we provide in Figure 8.1 the graph representing the improvement of the current solution cost in hard-fixing heuristic: as expected, the cost decreases very quickly for about 10 minutes, then it plateaus to an almost fixed cost, which corresponds to the solution eventually returned by the algorithm.

8.4 Constructive heuristics comparison

Since most of the constructive heuristics introduced in chapter 6 are not intended to be used standalone, we selected a small set of algorithms which are suitable to be tested on

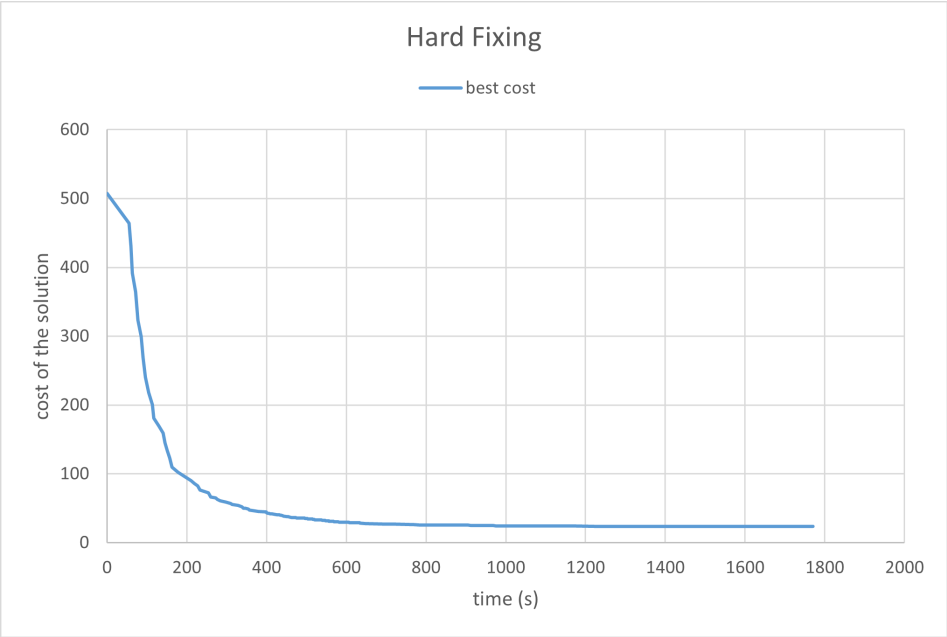
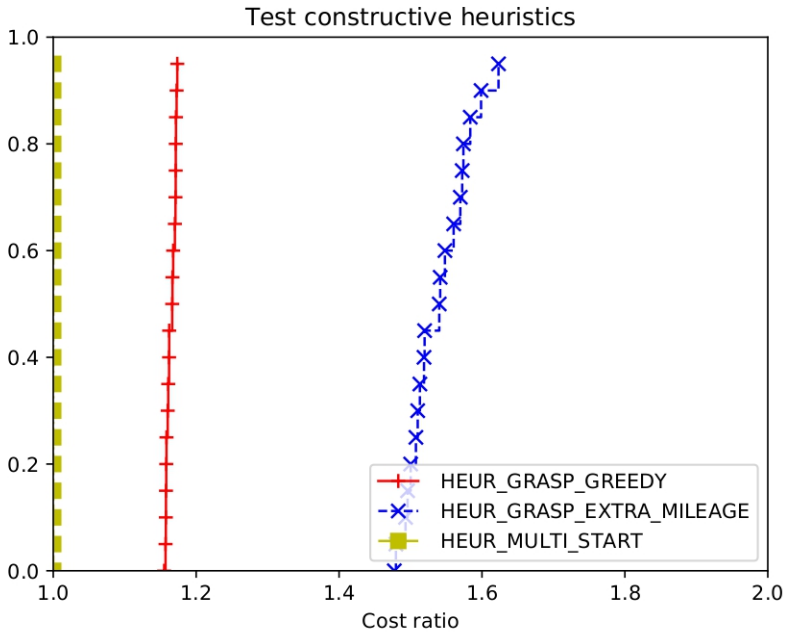


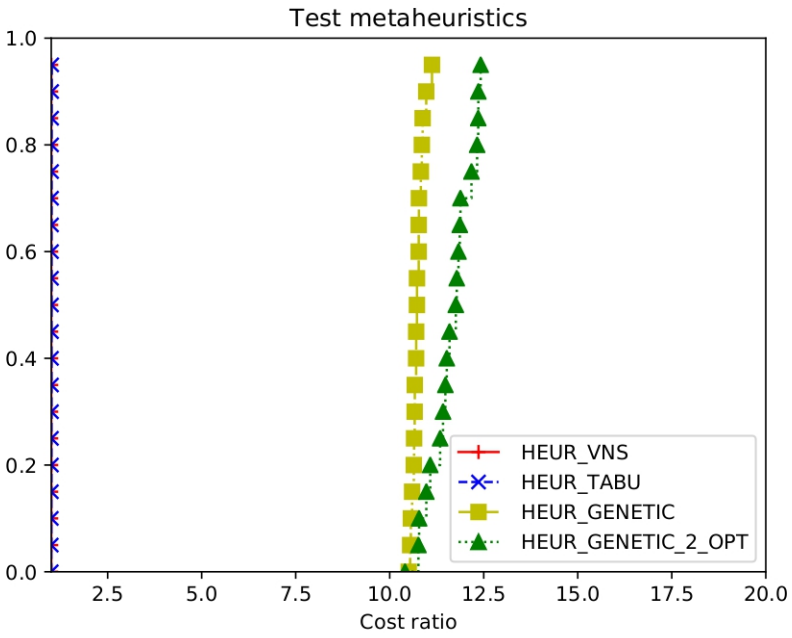
Figure 8.1: Hard-fixing heuristic best solution cost behaviour.

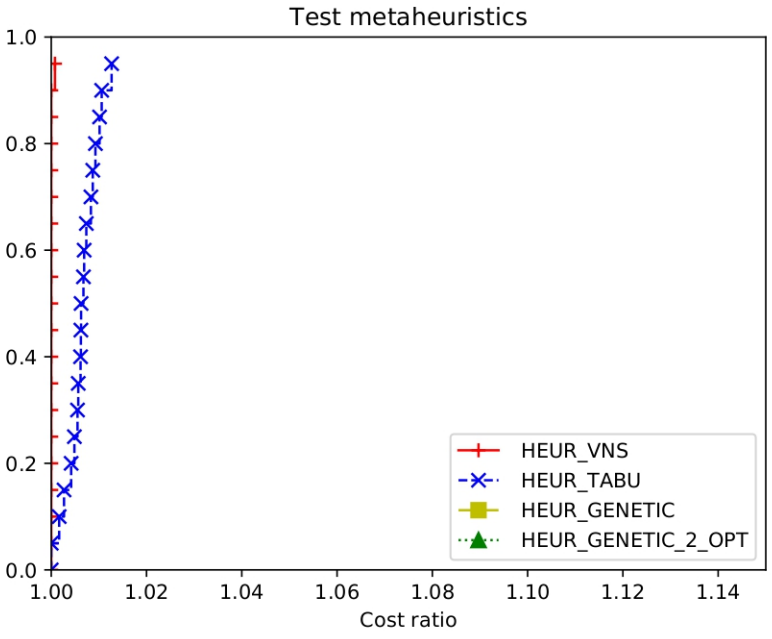
20 instances with 1000 nodes and a small time limit of 10 minutes. The objective is to find out if there is an algorithm that is more effective than the others in providing a good reference solution to perform the so-called *warm start*. Specifically, these algorithms are the two versions of the GRASP approach (see section 6.3) and the implementation of the multi-start approach described in section 6.4.2. The results of the following performance profile clearly confirm that the multi-start approach, while it relies on a smaller number of iterations than the simple GRASP using the fast NN algorithm, is able to provide better quality solutions.



8.5 Metaheuristics comparison

The last group of algorithms we tested are the metaheuristics we presented in chapter 7, which are the VNS, the Tabu search and the two variants of the genetic algorithm described in section 7.3.1. These algorithms have been tested on 20 instances of 2000 nodes with a time limit of 30 minutes. The results are reported in the following pair of charts, where the second plot is obtained by zooming in the leftmost part of the first one, because while VNS and Tabu search exhibit very similar results, with VNS confirming itself as the winner, both variants of the genetic algorithm lead to extremely worse solutions.





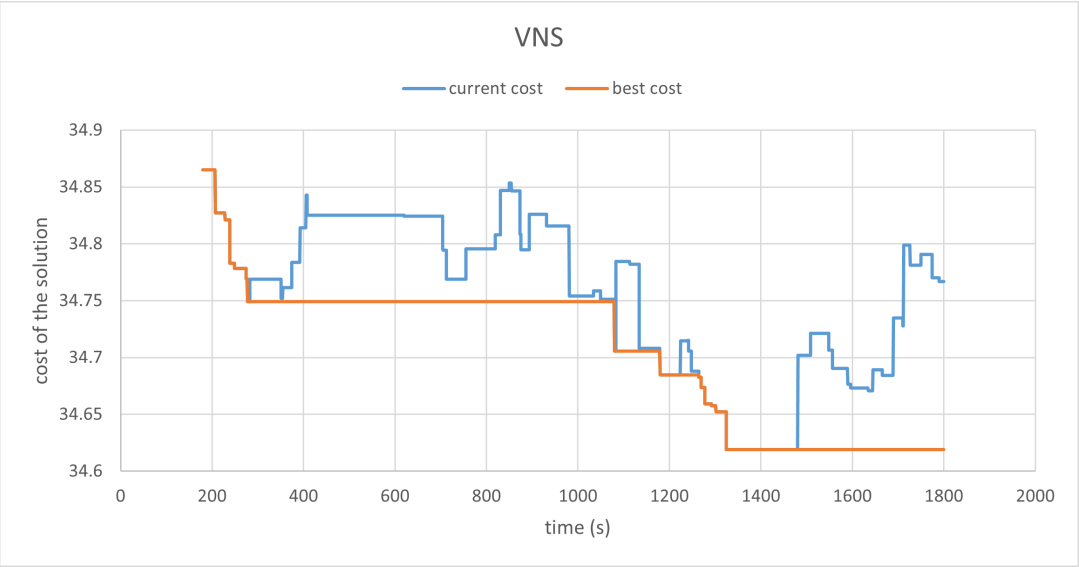


Figure 8.2: VNS metaheuristic best and current solution cost behaviour.

The graph in Figure 8.2 shows how the best and the current solution costs of VNS algorithm change over time (until the 30 minutes time limit is reached). We can see that after the warm start the solution cost is already close to the local optimum reached at the end, but after a series of k-opt moves and 2-opt refinements it manages to improve significantly after about 20 minutes.

The same measures associated to the two genetic algorithm variants are represented respectively in Figure 8.3 and 8.4. We observe that in HEUR_GENETIC the champion cost drops quickly at the beginning and then it plateaus. Of course, with a (considerably) higher time limit the solution returned would have improved, but this pronounced slowdown is likely to be due to the narrowing diversity among the population members, given by the limited amount of random mutations affecting them.

In HEUR_GENETIC_2_OPT there is a much wider difference between the average population cost and that of the champion, because 2-opt refinement allows to have a small fraction of solutions which are considerably better than the others. This also helps maintaining a high rate of improvement, given by the fact that such *elite* group of solutions are very likely going to reproduce with others. In this case, even a slightly higher time limit (e.g. a hour) could have improved the champion solution as much as to become comparable with VNS and Tabu search results.

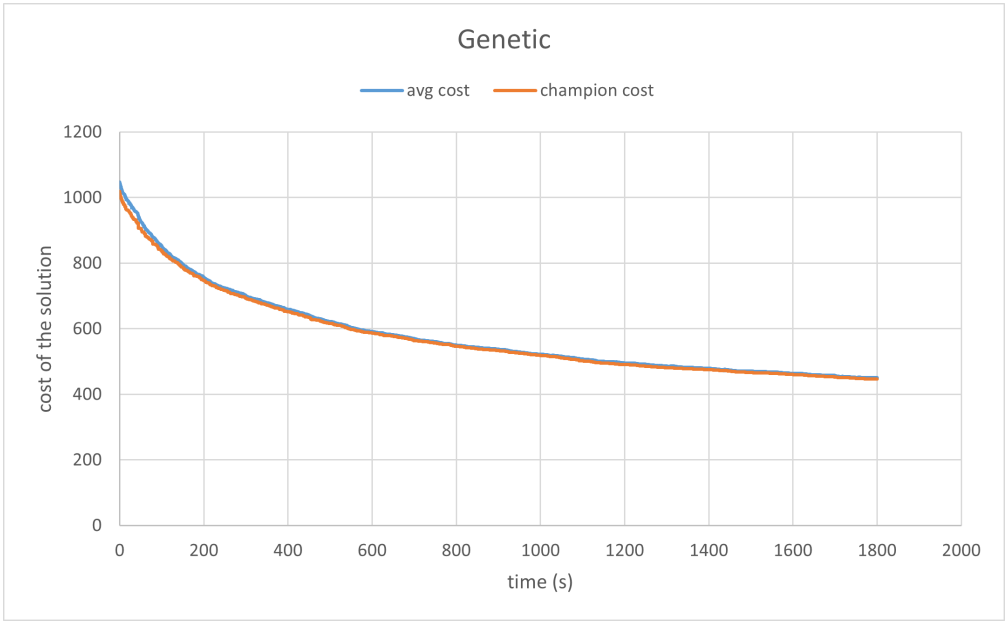


Figure 8.3: HEUR_GENETIC champion and average population fitness (cost) behaviour.

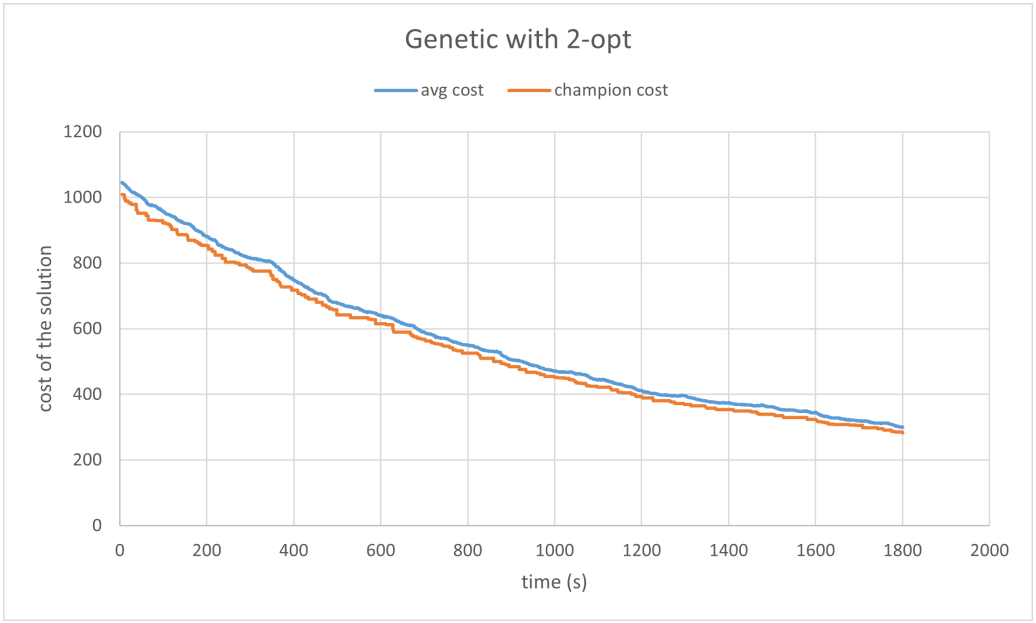


Figure 8.4: HEUR_GENETIC_2_OPT champion and average population fitness (cost) behaviour. N.B. This is not intended to be about the same instance as the graph in Figure 8.3.

9

Conclusions

9.1 Exact algorithms

According to the results discussed in chapter 8, the best performing exact algorithm is the Branch & Cut with 2-opt refinement, which managed to consistently deliver the optimal solution in (much) less than 30 minutes. For this reason, it would very likely be able to solve instances with up to 1000 nodes.

Among the compact models, which are clearly not suitable to solve instances of comparable size, an honourable mention goes to the MTZ model with all lazy constraints (called MTZ_SEC2_LAZY), which for small instances was able to provide optimal solutions in less than a second.

9.2 Heuristics

As regards the heuristic algorithms to solve instances of increasing size (up to 2000 nodes in our case), other than the tests shown in sections 8.3, 8.4 and 8.5, we run a *final test* between VNS and the best soft-fixing variant HEUR_SOFT_FIX_9 on a different set of 20 instances with 2000 nodes: the latter was able to find a solution only twice, because CPLEX often failed to find any integer feasible solution within the time limit. These two results are also about 5% worse than those returned by VNS. As a consequence, we can claim that the VNS metaheuristic is the best performing one.

Among the matheuristics, the soft-fixing heuristic slightly outperforms the hard-fixing one, and this tendency seem to get stronger as we increase the instance size.

One of the most interesting findings is that the 2-opt refinement heuristic was really easy to integrate into unrelated algorithms, such as the multi-start constructive heuristic and the genetic algorithm, and always lead to significant improvements.

On the other hand, our implementations of genetic algorithms proved to be too slow to provide good solutions in a reasonable amount of time, but we are aware that they are potentially extremely powerful and that different implementations can lead to competitive results.

Bibliography

[1] Applegate D.L., Bixby R.M., Chvátal V., Cook W.J., "The Traveling Salesman Problem", 2006, ISBN 978-0-691-12993-8.

[2] http://www.math.uwaterloo.ca/tsp/world (visited on 11/07/21)

[3] https://zs.thulb.uni-jena.de/receive/jportal_jparticle_00248075 (visited on 11/07/21)

[4] Cook, William J., "In Pursuit of the Traveling Salesman: Mathematics at the Limit of Computation.", Princeton, NJ: Princeton UP, 2012.

[5] Biggs N.L., Lloyd E.K., Wilson R.J., "Graph Theory, 1736–1936", Clarendon Press, 1976.

[6] Schrijver Alexander, "On the history of combinatorial optimization (till 1960)". In K. Aardal; G.L. Nemhauser; R. Weismantel, "Handbook of Discrete Optimization", Amsterdam: Elsevier. pp. 1–68, 2005.

[7] https://www.rand.org/about.html (visited on 12/07/21)

[8] Dantzig G., Fulkerson R., Johnson S, "Solution of a Large-Scale Traveling-Salesman Problem.", Journal of the Operations Research Society of America 2, no. 4: 393-410, 1954.

[9] Lawler E.L., "The Travelling Salesman Problem: A Guided Tour of Combinatorial Optimization (Repr. with corrections. ed.)". John Wiley & sons, 1985, ISBN 978-0471904137.

[10] Padberg M., Rinaldi G., "A Branch-and-Cut Algorithm for the Resolution of Large-Scale Symmetric Traveling Salesman Problems", SIAM Review, 33: 60–100, 1991.

[11] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95 (visited on 12/07/21)

[12] https://www.math.uwaterloo.ca/tsp/concorde.html (visited on 12/07/21)

[13] Rego C., Gamboa D, Glover F., Osterman C., "Traveling salesman problem heuristics: leading methods, implementations and latest advances", European Journal of Operational Research, 211 (3): 427–441, 2011.

[14] Christofides N., "Worst-case analysis of a new heuristic for the travelling salesman problem", Technical Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, 1976

[15] https://www.ibm.com/it-it/analytics/cplex-optimizer (visited on 14/07/21)

[16] https://www.ibm.com/docs/en/cofz/12.8.0?topic=techniques-user-cut-lazy-constraint-pools (visited on 13/07/21)

[17] Johnson D.S., McGeoch L.A., "The Traveling Salesman Problem: A Case Study in Local Optimization" in Aarts E.H.L., Lenstra J.K. (eds.), "Local Search in Combinatorial Optimisation", London: John Wiley and Sons Ltd., pp. 215–310, 1997