UNIVERSITY OF PADUA

INFORMATION ENGINEERING DEPARTMENT

MASTER'S DEGREE IN COMPUTER ENGINEERING

Solutions to the Travelling Salesman Problem

Professor:
MATTEO FISCHETTI

Student:
THOMAS PORRO
1237030

Academic year 2021/2022

Contents

Chapter 1

Introduction

In this report I am going to describe, analyze and implement solutions for the Travelling Salesman Problem (hereafter referred to as TSP).
The TSP is a NP-hard problem formulated as follows:

> *Given a list of cities and the distances between them, find the shortest path that connects all the cities and return to the origin one.*

It is used mainly in plain logistics, planning, and as a benchmark for testing optimization problems. However, it can be helpful in many other areas with a slight modification of the formula - such as in DNA sequencing, considering cities as DNA fragments, and astronomy, where they are stars.
Even though it is a NP-hard problem, instances with the dimensions of thousand or even millions of cities can be solved with great precision (around 1%) thanks to many heuristics and exact algorithms.

1.1 Brief history of the problem

The German handbook *Der Handlungsreisende* from 1832 was a guide used by salesman traveling through Germany and Switzerland. Albeit without any mathematical languages, it proves that people were starting to realize that optimal paths could save time and so it can be seen as the first example of TSP. The first TSP mathematical formula was made by Hmail and Kirkman in the XIX century, but it was in the 1930s that the TSP was implemented - mainly in Vienna and at Harvard University. An important leap forward was made in the 1950s, when G. Dantzig, D. R. Fulkerson, and S. M. Johnson expressed the problem as an integer linear program, even if they did not propose an algorithmic solution. They were able to devise the cutting plane method and they solved an instance with 49 nodes - by constructing a tour and proving that no other tour could be shorter. In the 1980s, Grötschel, Padberg, Rinaldi and others figured out instances with up to 2392 nodes, using both cutting planes and branch and bound. 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. In the 1990s Applegate, Bixby, Chvátal, and Cook developed the Concorde TSP solver. Nowadays, this program can run even on mobile devices such as iPads and it 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 this is currently the largest solved TSPLIB instance. For many other instances with millions of cities, today's solutions are guaranteed to be within 2-3% of an optimal tour.

1.2 The mathematical formulation

In this introduction, I used a natural way to describe the problem using the notion of cities and distance travelled. But from a mathematical point of view, the TSP problem can be thought as graph $G = (V, E)$ where $V = \{v_1, \dots, v_n\}$ are the cities described in this introduction (from now on called nodes or vertices of the graph), and E represents the path that connects one node to another and it can be described as $E \subseteq (V \times V)/\{\{i, j\} : i \in V\}\}$ that is the set of edges of the graph.
Another fundamental aspects to be taken in consideration are the concept of distance and its mathematical counterpart, the cost. I assign to each edge a real number that will be used to give weight (or cost) to the path chosen.
Now that I have introduced the main concepts, the TSP problem can be explained also from a mathematical point of view:

> *Given a list of nodes and the distances between them, find the shortest hamiltonian circuit.*

One of the most important formulations is the following one [1], even if it presents slight modifications:

$$x_{ij} = \begin{cases} 1 & \text{if the arc } (i, j) \in A \text{ is chosen in the optimal solution} \\ 0 & \text{otherwise} \end{cases} \tag{1.1}$$

$$min \sum_{(i,j) \in A} c_{ij} x_{ij} \tag{1.2a}$$

$$\sum_{(i,j)\delta^-(j)} x_{ij} = 1, \quad j \in V \tag{1.2b}$$

$$\sum_{(i,j)\delta^+(j)} x_{ij} = 1, \quad i \in V \tag{1.2c}$$

$$\sum_{e \in E_G(S)} x_e \leq |S| - 1, \quad \forall S \subset V \quad , |S| \geq 2 \tag{1.2d}$$

$$x_{ij} \geq 0 \text{ intero}, \quad (i, j) \in A \tag{1.2e}$$

This model is not fully functional since it allows the presence of sub-tours in the solution. In order to avoid them, I am going to add some constraints to the model in the next sections.

Chapter 2

Code setup

Section 1 described the problem and stated that it is NP-hard. The computational effort to obtain an optimal solution to this problem is high. Due to this fact, the implementation will be done using the programming language C which will provide really good efficiency and better memory management than other languages.

All the results obtained in this report were executed on a Linux machine with Ubuntu 20.04 as operative system. The system runs an Intel Core i7 8550u @1.80Ghz with 16Gb of RAM.

To produce this code, lots of software were used. The main IDE in this project is CLion by IntelliJ and CMake is the build software adopted to compile the code. The most important software used is IBM ILOG CPLEX, which is an optimization software package integrated in the code through its Callable Library (API). This software isn't usually free but it was made available free of charge for students that needed it for academic purposes.

During this project, the TSP problem could present some fractional solution, and to deal with them the external library Concorde has been used (furter explaination in section 4.2.1).

The Gnuplot library, which is a command-line driven utility, was adopted to visualize the solution found. The main usage was to check the effect of the code written to the TSP instances. The graphs of this report are generated exploiting the command line and using Gnuplot through a pipe.

For analyzing the efficiency of the algorithms presented, a performance profile tool has been used. This tool is written by Salvagnin (2019) and it is for internal use only.

2.1 Solution and data management

A reference library has been adopted in order to evaluate the method implemented. TSPLIB [5] is a library of sample instances for the TSP (and related problems) from various sources and of various types. This allowed having a starting point on the information needed to solve the problem. Among all the sections contained in the *.tsp* files only, the most important ones are stored into the instance which are:

- DIMENSION: the number of nodes that are in the file;
- EDGE WEIGHT TYPE: the typology of the distance between nodes;
- NODE COORD SECTION: where the coordinates of the nodes are described, usually at the end of the file.

To store the solution in a convenient and useful way, the notion of a successor is introduced. Indeed the solution of the TSP problem is a cycle, so each node has a successor. Considering an array big as the number of nodes in the problem, the index of the array is the nodes number and its content is the number of the next node in the solution. In this way, each node points to its successor and generates the solution.

2.2 CPLEX's variables and constraints

Understanding how CPLEX build and manage constraint is important for a full control over the system.

The whole software works with the concept of rows and columns, which are respectively the constraints and the variables of the problem. Using a simple minimisation problem, where x_i is an integer value, it is:

$$\begin{array}{lllllll} \min & x_1 & + & 3x_2 & + & x_3 & \\ & 2x_1 & & & - & x_3 & \leq 60 \\ & & & 4x_2 & + & 7x_3 & \geq 20 \end{array} \tag{2.1}$$

This problem can be seen as a matrix composed of rows and columns, each column corresponding to a variable and each row corresponding to a constraint that the problem has to satisfy. Whenever it is necessary to add a new variable, for example x_4, it is possible to do so by calling the API and by adding a new column to the problem. The same can be done with a constraint: it can be added by inserting a new row using the callable library.

When a new constraint is added it contains only the right-hand side of the equation/inequation, and all the coefficients of the variables are setted to 0. The constraint will appear as following:

$$\text{*empty*} \qquad \leq 60 \tag{2.2}$$

Inequation 2.2 is the first row of the model described in 2.1. For having the coefficients consistent with the model of this section, it is possible to use the API of CPLEX. In this way it is possible to set the value of the variable x_1 to 2 and x_3 to -1, and to obtain:

$$2x_1 - x_3 \leq 60 \tag{2.3}$$

This operation is done every time a new constraint is added to the instance. The Callable Library has also other methods to add new constraints but this one is the one used during this project.

Chapter 3

Compact models

In this section, I am going to explore the first method used to solve the TSP problem. In particular, I will describe the Miller, Tucker, and Zemlin model (known as MTZ model) and the Gavish and Graves model (known as GG model).

3.1 Basic model

The first model I took in consideration is a lightly modified version of the one presented in section 1.2. This formula still doesn't adopt the Sub-tour Elimination Constraint (SEC) since it is intended to be used jointly with other SECs and with other methods such as matheuristics.
This configuration gives an undirected complete graph $G = (V, E)$. The formulation used in the code is the following:

$$min \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{3.1a}$$

$$\sum_{(i,j)\delta^-(j)} x_{ij} = 1, \quad j \in V \tag{3.1b}$$

$$\sum_{(i,j)\delta^+(j)} x_{ij} = 1, \quad i \in V \tag{3.1c}$$

$$x_{ij} \geq 0 \text{ intero}, \quad (i,j) \in A \tag{3.1e}$$

It is the same model used in section 1.2 but in this case 1.2d is not included.
In this implementation, the graph is symmetric. The arcs (i, j) and (j, i) have the same weight and thus they are represented with the same edge. In this way, the total number of variables are reduced too, considering the starting point not of n^2 but only $\frac{n(n-1)}{2}$.
The result using this model can be seen in figure 3.1.

Since sub-tours are present, this solution cannot be accepted. In the next sections SEC constraints will be added to the formula for avoiding this problem.

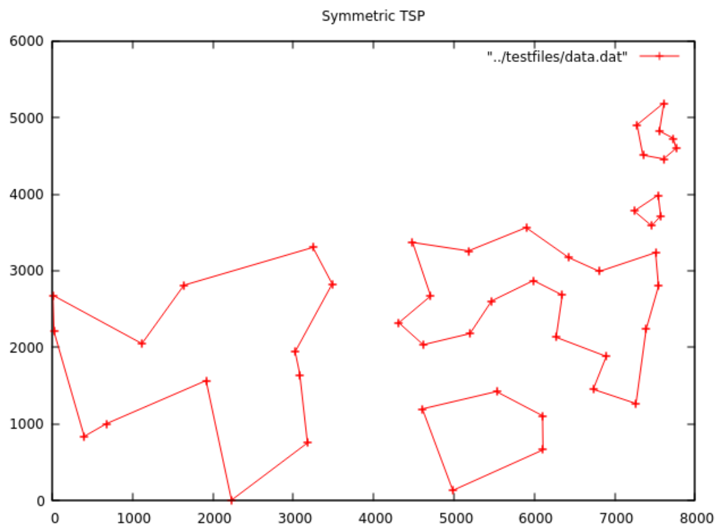


Figure 3.1: The image represent att48.tsp solved with the problem formulation showed in section 3.1

3.2 The Miller, Tucker, and Zemlin model

As said before, the basic model can create loops in the optimal solution.
A first constraint introduced to avoid this situation was described by Dantzig, Fulkerson, and Johnson. The constraint added was:

$$\sum_{i \in S} \sum_{j \in S} x_{ij} \leq |S| - 1; \quad \forall S \subset V : \{1\} \notin S,\ |S| \geq 2,\ i \neq j \tag{3.2}$$

This constraint limits the number of edges in the solution, so that no cycles are allowed in the result. Nonetheless, the usage of this method is infeasible even with a low number of nodes since the amount of constraints needed is exponential ($O(n^2)$).

The model produced by Miller, Tucker, and Zemlin bypasses the exponential SECs by reducing their number to a simple polynomial. Differently from the basic model, the graph obtained in this case is asymmetrical: x_{ij} and x_{ji} can have different weights.

In the new formulation, a new variable called u_i is assigned to each node of the solution. This number represents an increasing sequence number in the optimal tour: starting from the second one ($u_2 = 0$), each node will increase the value by 1 at each following node until the end of the solution ($u_n = n - 2$). The first node is considered special and its value is always set to 0.

This is the new constraint added to the basic model by MTZ:

$$u_i - u_j + n x_{ij} \leq n - 1; \quad i, j \in \{2, \ldots, n\}, i \neq j \tag{3.3}$$

$$0 \leq u_i \leq n - 2; \quad integer \quad i \in V : i > 1 \tag{3.4}$$

The meaning of this formulation is the following: if the arc x_{ij} is selected, then the value of u_j is $u_j \geq u_i + 1$.

3.2.1 Implementation of the model

To express 3.3 as a CPLEX constraint I need to rewrite the inequation in a way called Big-M. This method expects a new variable called M, which allows the system to activate or deactivate the constraint in a simple way. The new constraint will be:

$$u_j \geq u_i + 1 - M(1 - x_{ij}) \tag{3.5}$$

With this approach, the constraint is strictly depending on the value of x_{ij}. If $x_{ij} = 1$, the constraint works like a normal one because the value of $M(1 - x_{ij})$ will be 0, so the meaning of 3.5 will be the same as the one expressed in the previous section. If $x_{ij} = 0$, the right-hand side of the inequation will be certainly negative, making the constraint deactivated and allowing u_j to take up any possible value from 0 to $n - 2$. Between all the values that M can assume, the smallest one is $n - 1$ due to the fact that, in the case of $X_{ij} = 0$, the constraint will be still useful even if u_i reaches its case limit of $n - 2$.

With the application of the Big-M trick, the constraints can be written in the CPLEX environment. There are substantially 3 methods that can be implemented in the framework:

- the use of standard constraints: all the constraints wrote in 3.3 are directly saved into the problem at once. This lead to have $O(n^2)$ constraints active, making the optimization too large or even too expensive to solve;

- the use lazy constraints: as the name suggests, here the constraints are applied lazily. This means they are not always applied to the problem because CPLEX uses them only when necessary. In doing so, a pool of constraints is created and every time an integer optimal solution is found, the violation of every constraint in the pool is checked. If one of them is infringed, this constraint is added permanently to the instance. This method will hopefully make the problem smaller and faster to be solved than the one created with the standard constraints;

- the use of indicator constraints: in the first two cases the Big-M trick is used to trigger a constraint when a particular variable assumes a predetermined value, but this method is not always preferable since it can behave in unstable ways. That is why a good implementation of 3.3 is the usage of the indicator constraints provided by the CPLEX API: this method automatically activates the constraint $u_j \geq u_i + 1$ when the x_{ij} assumes the user passed value.

3.3 The Gavish and Graves model

The second compact model implemented is the one proposed by Gavish and Graves, based on the single commodity flow: the arcs are considered as pipes.
Like in the MTZ model, a new variable is introduced in the problem. It is called "Flow of the arc" and it is represented with the symbol y_{ij}. Compared with the model in section 3.2, the starting value of y_{ij} is $n-1$ and decrease by 1 at each following node in the optimal solution. To implement this proposition the following constraints are added to the instance:

$$y_{ij} \leq (n-1)x_{ij} \tag{3.6}$$

$$\sum_{j \in V; j \neq 1} y_{1j} = n - 1 \tag{3.7}$$

$$\sum_{i,j \in V; i \neq j} y_{ij} - \sum_{j,k \in V; j \neq k} y_{jk} = 1 \tag{3.8}$$

This formula finds an optimal solution without sub-tours.

Chapter 4

Other sub-tour elimination methods

In chapter 3 I described models that cut out any possibility to have some sub-tours in the optimal solution. In this section I use particular techniques to remove the tours in a faster way.

4.1 Benders method

This method is the simplest one presented in this chapter. The basic idea is the insertion of the SECs only when sub-tours are found. This is quite different from the implementation of the lazy constraints of MTZ seen before: the constraints are not activated when one of them is violated, but they are manually added into the instance.
This method uses the basic model described in 3.1 and follows this algorithm:

Algorithm 1 Benders

Input: $G = (V, E), c : E \rightarrow \Re^+$
Output: z^* optimal solution

instance $\leftarrow$ *initializing basic model*
successors, component$\leftarrow$ *initialize arrays*
ncomp $\leftarrow$ 99999
while $ncomp > 1$ **do**
    $z^* \leftarrow$ CPXmipopt
    successors, component, ncomp $\leftarrow$ build_solution(z^*)
    if $ncomp > 1$ **then**
        instance $\leftarrow$ add violated constraints
    end if
end while
return z^*

This algorithm shows the simplicity of the Benders method. The first thing to do is building the basic model inside the instance of the CPLEX environment, and afterwards the arrays that will contain the successors and the components are initialized. These two arrays are n long, so they are big enough to hold all the nodes. The most important function in this algorithm - excluded the optimization (CPXmipopt) - is the function build_solution. It fills the arrays successors and components following this idea: search all the connected components inside the solution, then each *successors*[i] will contain the next node inside the connected component, and *component*[i] will enclose the index of the component of the node i. Then the algorithm checks the number of the connected components saved into ncomp. If more than one loop is present, the algorithm inserts a new constraint into the instance for each component, following this formula:

$$\sum_{e(S)} x_e \leq |S| - 1; \quad \forall S \subset \{2, \ldots, n\},\ |S| \geq 2 \tag{4.1}$$

where x_e are the edges contained inside the connected component S.

4.2 Callback method

In the previous section, the Benders method reaches the optimal solution of the problem through multiple calls of the CPLEX optimizer. This is done by adding the sub-tour elimination constraints if the solution found is composed of various tours. The purpose of this section is to introduce a different approach: I will exploit the branch-and-cut technique through the use of the CPLEX callbacks.
The API by IBM's software grants the use of some callbacks during the optimization process. CPLEX provides a wide range of possibilities such as informational callbacks, query/diagnostic callbacks, and control callbacks. The first ones give the user additional information on the current optimization without affecting the performance or interfering with the solution search space. The second ones access to more detailed information compared to the informational callbacks, but they can affect the overall performance of the problem resolution; the query/diagnostic callbacks are also incompatible with the dynamic search and the deterministic parallel functions. The last ones are the one I am going to use and they allow the user to alter and customize how CPLEX performs the optimization.

During the optimization process, CPLEX will find numerous possible solutions.
Let's assume that a candidate solution x^* is found during the operation. Then, if the cost of this new result is better than the previous one the software will update the current best solution with the last found. Otherwise, the solution is considered infeasible and it is rejected by the system.
The CPXcallbacksetfunc method will set my custom callback that will be used every time a candidate solution is found. The algorithm adopts the same method explained in section 4.1 (build_solution): if more than one connected component is found, a new SEC is added to the instance of the problem and the candidate solution is rejected (through the function CPXcallbackrejectcandidate).

The relevant difference between this implementation and the Benders method is that, while the latter needs the iteration of various optimizations processes, the former rejects the possible solution aforehand provided to the user. The algorithm used here is the following:

Algorithm 2 Callback method

Input: $G = (V, E), c : E \rightarrow \Re^+$
Output: z^* optimal solution

procedure MAIN($G = (V, E), c : E \rightarrow \Re^+$)
    instance ← *initializing basic model*
    instance ← *instance ∪ custom callback*
    $z^* \leftarrow$ CPXMIPOPT
    return z^*
end procedure
procedure CUSTOMCALLBACK(z)
    ncomp ← BUILD_SOLUTION(z)
    if *ncomp* > 1 **then**
        Add SEC and reject candidate solution
    end if
    return
end procedure

4.2.1 Callback on the fractional solutions

In the previous section, the callback was called on each integer candidate solution. This sector will describe the use of the callback even on the fractional solution. The aim is to save computational time by adding new SECs that are probably common in the decision tree.

The procedure defined in this phase is the same in section 4.2, except that this callback is applied in the continuous relaxation of the problem. To implement this kind of operation, I used an external library called Concorde [2]. This library is written in ANSI C and it is one of the most powerful tools on the market: it can solve really large instances of the TSP problem to the optimal solution.
When the callback is called, I use the library mentioned above to compute which SEC I require to exclude the fractional solution from the decision tree.

Chapter 5

Non optimal solutions

In this section I am going to explore a new branch of the TSP resolution, in which the discovery of an optimal solution is no longer important. This development uses matheuristic and heuristic to find a solution with a great approximation to the optimal one, even with instances including millions of nodes. Therefore, I am going to explore how big instances can be solved when classical methods such as MTZ and GG take too long. Moreover, I am going to cover different approach, namely matheuristic methods, heuristic and metaheuristic algoritms.

5.1 Matheuristics

"The objective of a heuristic is to produce a solution in a reasonable time frame that is good enough for solving the problem at hand. This solution may not be the best of all the solutions to this problem, or it may simply approximate the exact solution. But it is still valuable because finding it does not require a prohibitively long time" [3].
The two techniques I am going to delve into are the hard-fixing and the soft-fixing.

5.1.1 Hard-fixing

The main purpose of this method is to try to reduce the search space by cutting down the complexity of the optimization. For achieving this goal, an initial feasible solution is needed - obtained by any approach described in this report.
The central thought of hard-fixing is to get an easier problem by setting some variables of the solution passed to the method - and so by having fewer variables to compute. The variables to be fixed are x_{ij}, $i, j \in V$. This task is done by settling the values to 1, so that the edge will be surely part of the solution. A valid method to choose which path is designated to be fixed is to link each edge to a probability p between 0 and 1, then set randomly - with the probability picked - the value to 1.
Hereupon, the problem will presumably have $p * |E|$ variables fixed and thus the instance

can be solved with a lower effort. Hopefully, this situation will bring to a better solution. Although the choice of using a probability is a good pick, any other method can be used to block the edges.

The performance of this technique is strictly related to the choice of the first feasible solution: if it is not good enough, this approach will get stuck on some non-optimal solution. A good fix for avoiding this is to lower down p whenever the solution is not improved for a predetermined amount of time.

Algorithm 3 Hard-fixing

Input: $G = (V, E)$,$c : E \rightarrow \Re^{+}$, $global\_timelimit$, $iteration\_timelimit$
Output: z hopefully good solution

instance $\leftarrow$ *initializing model (any of this report)*
$z \leftarrow$ CPXmipopt *with nodelimit 0*
$p \leftarrow 0.9$
$i \leftarrow 0$
while $time\_elapsed < global\_time\_limit$ **do**
    if $time\_remaining > iteration\_timelimit$ **then**
        instance $\leftarrow$ *set timelimit to $iteration\_timelimit$*
    else
        instance $\leftarrow$ *set timelimit to $time\_remaining$*
    end if
    instance $\leftarrow$ *hard-fixing with probability p*
    $z_{new} \leftarrow$ CPXmipopt
    if $cost(z_{new}) < cost(z)$ **then**
        $z \leftarrow z_{new}$
        $i \leftarrow 0$
    else
        $i \leftarrow i + 1$
    end if
    if $i = 10$ **then**
        $p \leftarrow p - 0.1$
    end if
    instance $\leftarrow$ *remove hard-fixing*
end while
return z

In this algorithm, there are some variables not mentioned before: the global timelimit and the iteration timelimit. As aforementioned, the hard-fixing is a technique that aims to obtain a good solution in a short amount of time, so the parameters described above are necessary to establish - as the name suggests - the timespan in which the optimization is solved. global_timelimit is the total time reserved to the solver, interation_timelimit is the duration of each optimization in which the hard-fixing is applied.

The initial solution is provided by the solver, limited in the depth of its analysis. Then, algorithm 3 starts to iterate the main process until the global_timelimit is reached. During this phase, the optimization is called several times, in each of which the instance is solved blocking some variables to 1.

5.1.2 Soft-fixing

The technique that will be described is called Local Branching. Since it is a slight modification of the hard-fixing it is also called soft-fixing.
In section 5.1.1 the value of the variables is fixed in a manual way using a probability system. In Local Branching, this operation is performed by adding a new constraint that forces the instance to block a predetermined number of variables, giving to the solver a degree of freedom on which variables to fix and which not.

The main idea of soft-fixing is the Hamming distance: "it measures the minimum number of substitutions required to change one string into the other" [4]. This description can be applied also to vectors. So given two vectors x and $\tilde{x}$ in $\{0,1\}$, the Hamming distance is the number of different bits they have. It can be described in this way:

$$H(x,\tilde{x}) = \sum_{j:\tilde{x}_j=1}(1-x_j) + \sum_{j:\tilde{x}_j=0} x_j \tag{5.1}$$

Considering that the output solution of the solver is a vector in $\{0,1\}$, it is possible to insert into the instance a new constraint that limits the hamming distance between the old solution and the new one. The number of edges active in each solution is forced to be $n = |V|$ and therefore the Hamming distance is computed on the differences in the bits equal to 1. For this reason, it is possible to reduce 5.1 to:

$$H(x,\tilde{x}) = \sum_{j:\tilde{x}_j=1}(1-x_j) = n - \sum_{j:\tilde{x}_j=1} x_j \tag{5.2}$$

The purpose of this method is to limit the diversity of two solutions. For doing so, it is possible to add a new variable k that limits the Hamming distance. Through elementary math, this formulation is built as:

$$H(x,\tilde{x}) \leq k \Rightarrow n - \sum_{j:\tilde{x}_j=1} x_j \leq k \Rightarrow \sum_{j:\tilde{x}_j=1} x_j \geq n-k \tag{5.3}$$

The consequence of 5.3 is to narrow the next iteration of the optimization to a k-neighborhood of the previous one. As the method described in section 5.1.1, the value k can vary if for the optimization doesn't improve the solution.

The only changes between 3 and 4 are the variables used and the constraints added to the instance. In this implementation, k is increased by 1 every time the solution is not improved.

Algorithm 4 Soft-fixing

Input: $G = (V, E)$,$c : E \rightarrow \Re^{+}$, *global_timelimit*, *iteration_timelimit*
Output: z hopefully good solution

instance $\leftarrow$ *initializing model (any of this report)*
$z \leftarrow$ CPXMIPOPT *with nodelimit 0*
$k \leftarrow 2$
$i \leftarrow 0$
while *time_elapsed* $<$ *global_time_limit* **do**
    if *time_remaining* $>$ *iteration_timelimit* **then**
        instance $\leftarrow$ *set timelimit to *iteration_timelimit**
    else
        instance $\leftarrow$ *set timelimit to *time_remaining**
    end if
    instance $\leftarrow$ *soft-fixing using a k-neighborhood*
    $z_{new} \leftarrow$ CPXMIPOPT
    if $cost(z_{new}) < cost(z)$ **then**
        $z \leftarrow z_{new}$
        $i \leftarrow 0$
    else
        $i \leftarrow i + 1$
    end if
    if $i = 10$ **then**
        $k \leftarrow k + 1$
    end if
    instance $\leftarrow$ *remove soft-fixing*
end while
return z

5.2 Heuristics

In this section the concept of heuristic is explored in some of algorithms used to solve the TSP problem. In particular, a set of methods will be presented namely greedy algorithm, extra mileage, and 2-opt optimization.

5.2.1 Greedy algorithm

This type of method is the easiest to understand and implement. This heuristic has its roots on the concept of constructing the shortest path by connecting the nearest nodes.

This algorithm starts from an arbitrary node and chooses its following node by selecting the nearest one from the ones that are not already in the path. For this reason, it is also called Nearest Neighbor.

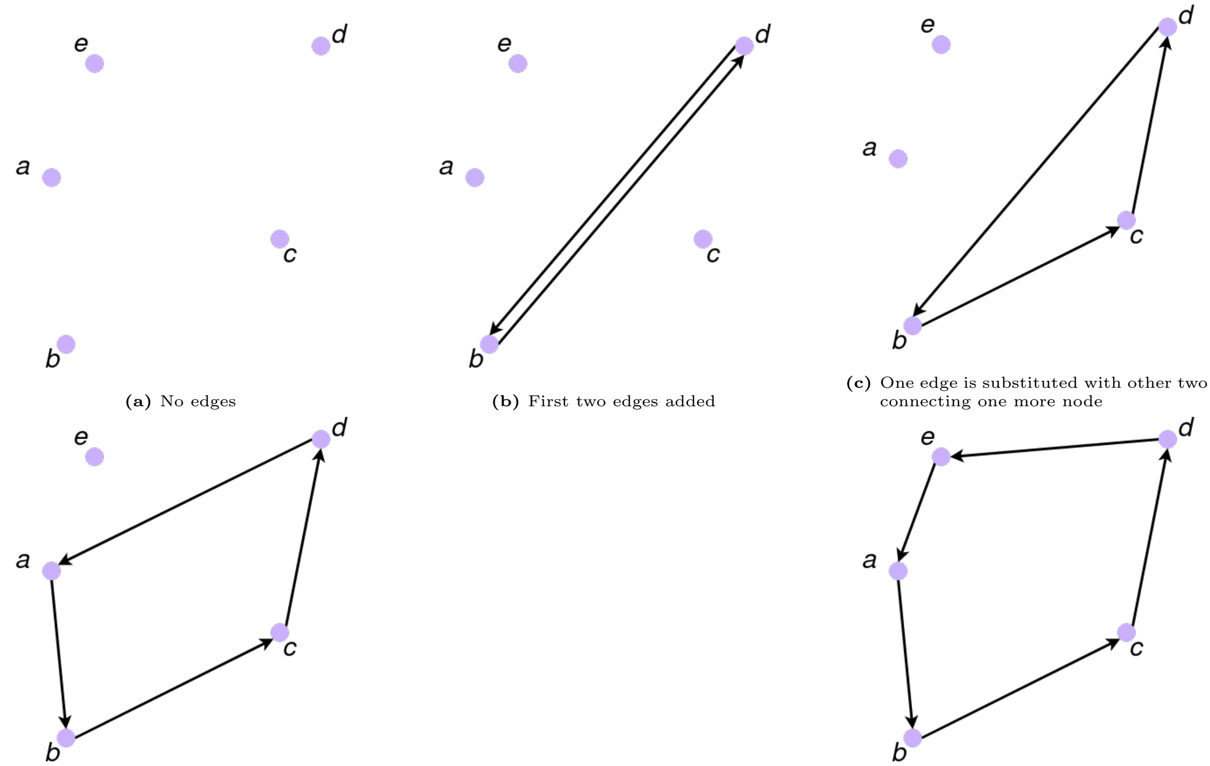
The main side effect is that the shortest path is easily missed since the last nodes - apart from special cases - are far from each othe and in this way the optimal solution is not selected. The algorithm is the following:

Algorithm 5 Greedy

Input: $G = (V, E)$,$c : E \to \Re^+$
Output: z hopefully good solution

$best\_cost \leftarrow +\infty$
$z \leftarrow$ *empty*
for $start\_node$ ←1 *to* n **do**
    $current\_node \leftarrow start\_node$
    while *each node is visited* **do**
        $candidate\_node \leftarrow -1$
        for $i \leftarrow 1$ *to* n **do**
            Finds the nearest node and marks it as visited
            $candidate\_node \leftarrow$ *nearest node*
        end for
        Saves $candidate\_node$ as next node
        $current\_node \leftarrow candidate\_node$
    end while
    if cost(z_{curr})< $best\_cost$ **then**
        $best\_cost \leftarrow cost(z_{curr})$
        $z \leftarrow z_{curr}$
    end if
end for
return z

The classic implementation has a complexity of $O(n^2)$ since each node has to search the nearest node all over the graph. To obtain the best solution possible, in my implementation each node is tested as starting node. In particular, 5 has a complexity of $O(n^3)$ because of the aforementioned method to select the best starting node.



(a) No edges

(b) First two edges added

(c) One edge is substituted with other two connecting one more node

(d) Added one more node

(e) All nodes are connected

Figure 5.1: In this image we can see the process done by extra-mileage to find the solution.

This complexity is quite low but it can be slower than expected with some instances over a great number of nodes.

5.2.2 Extra mileage approach

This approach is more complex than the previous one but in favor of a better solution. The main idea behind this algorithm is very simple and it sets up from an empty solution.

It starts by connecting the farthest nodes in the instance with a cycle, and this path will be the beginning of the solution. Then, the closest node among the active edges is selected and added to the solution by replacing it with two more edges that allow the new node to enter the solution. This process can be seen in figure 5.1, where it is performed with a five nodes.

For inserting the next node, the method used is the triangle inequality, which states that the sum of the lengths of any two sides must be greater than or equal to the length of the remaining side. In doing so with the replacing phase, some cost is added to the solution. Imagine taking into consideration three nodes x, y and z, where x and y are already in the solution and z wants to join them. The mathematical operation to apply is the following:

$$\Delta(x, y, w) = c_{xw} + c_{yw} - c_{xy} \tag{5.4}$$

This value will always be positive thanks to the aforementioned triangle inequality. The algorithm used is the following:

Algorithm 6 Extra mileage

Input: $G = (V, E)$,$c : E \rightarrow \Re^+$
Output: z hopefully good solution
$x, y \leftarrow$ *finds the farthest nodes*
$z \leftarrow \text{ADD}(x, y)$
while $|z| < n$ **do**
  $(x, y, w) \leftarrow \text{argmin}(\Delta(x, y, w) : x, y \in z, w \notin z)$
  $z \leftarrow \text{ADD}(w)$
end while
return z

5.2.3 2-opt refining

This section analyzes a refining technique where the goal is to take an existing solution x and to put it closer to the optimum one.
The k-opt refining consists on rearrange k edges in order to obtain a new solution with a lower cost. In particular, this section describes the 2-opt refining.

To handle this approach effectively, it is applied starting from a solution obtained with a heuristic approach - such as the ones described in section 5.2.1 and 5.2.2. The idea of this algorithm is to remove all the crossing edges and to insert new edges that will decrease the cost of the final solution.

Here too the triangle inequality is used to find the couple of edges that allow the best improvement. In this implementation, the operation is done across four nodes: i, $succ[i]$, j, and $succ[j]$. Since the tour is asymmetric, it has a direction: $succ[i]$ and $succ[j]$ are respectively the following nodes in the path of the nodes i and j.

$$\Delta(i, j) = (c_{i,succ[i]} + c_{j,succ[j]}) - (c_{i,j} + c_{succ[i],succ[j]}) \quad i, j \in V; i \neq succ[j]; j \neq succ[i] \tag{5.5}$$

In 5.5 it is described the correct way to compute the delta of the method. Greater the value, better the improving.

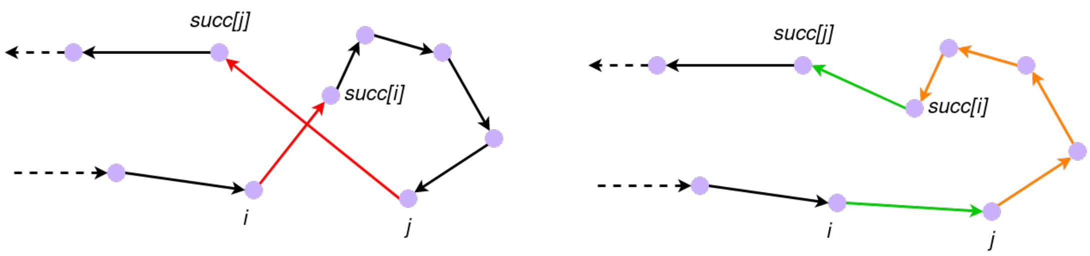


Figure 5.2: The image represent the replacement of two edges during the 2-opt refining.

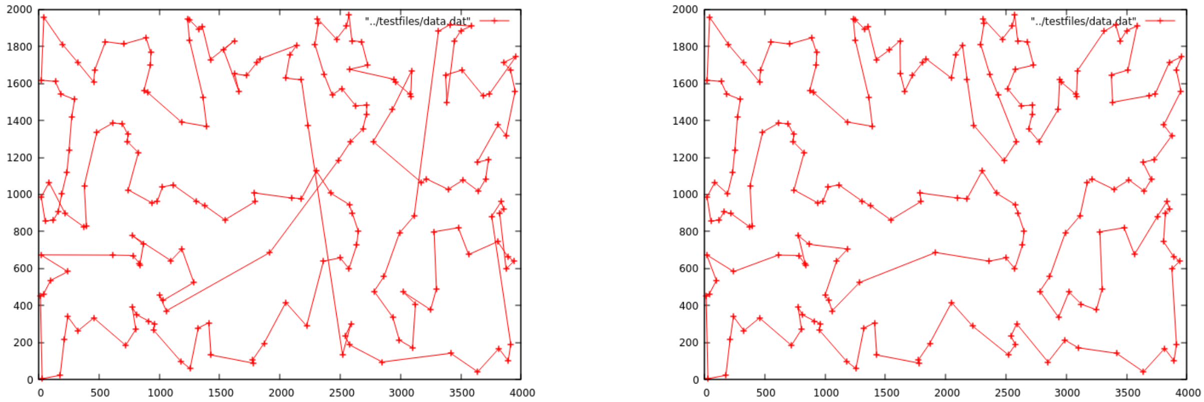


Figure 5.3: The improvement obtained applying to a greedy algorithm (left) the 2-opt refining approach (right).

In figure 5.2 it is possible to look at the replacement of two arcs. It is important to notice that the path from i to $succ[i]$ changes direction after the substitution of the old edges. This is done for maintaining the solution integrity. In particular, this optimization is performed until no more positive $\Delta(i, j)$ are found. If they are not found, the solution is no more improvable by the 2-opt algorithm.

Algorithm 7 2-opt refining

Input: $G = (V, E)$,$c : E \rightarrow \Re^+$
Output: z hopefully good solution

$z \leftarrow$ *find feasible solution with an algorithm*
flag $\leftarrow false$
while $flag == false$ **do**
  flag $\leftarrow true$
  $(i, j) \leftarrow \text{argmax}(\Delta(i, j) : i, j \in V)$
  if $\Delta(i, j) > 0$ **then**
    flag $\leftarrow false$
    $z \leftarrow$ *edge replacement*
  end if
end while
return z

5.3 Metaheuristics

While the heuristic approach seen before was strictly specific to the TSP problem, the concept of metaheuristic is quite different. Its procedure is problem-independent and it does not take advantage of any specificity of the problem. Generally, it is not greedy and it can accept a temporary deterioration of the solution in order to have a bigger search space in which to find a better solution.

In this section, the variable neighborhood search and the tabu search are used.

5.3.1 Variable Neighborhood Search

The variable neighborhood search (VNS) is the first metaheuristic approach presented. Its main purpose is to try to escape from a local optima aiming to find the optimal solution of the problem.

The instance of the problem can be seen as a function with its local minima, local maxima, and so on. During the utilization of some heuristic techniques, it is possible that the process remains stuck in a sub-optimal area. The way in which it tries to escape from this region is done by changing the *neighborhood* of the solution: some data are modified and then a new minimum is searched.

In this report, the adjustment of the neighborhood is performed by a perturbation phase in which is applied a k-opt kick. This change is performed by selecting randomly a k set of edges and then replacing them with others in order to obtain a poor solution. In particular, the procedure done is the following: a 2-opt refining is applied to a reference solution, and when the minimum is reached a kick is given to the solution to worsening the solution. Then, the 2-opt refining approach is used again to find a new minimum. In my implementation, three perturbations are implemented: 3, 5, and 7. An example of implementation can be seen in Algorithm 8.

Algorithm 8 VNS

Input: $G = (V, E)$,$c : E \to \Re^+, k, global\_timelimit$
Output: z hopefully good solution

$z_{curr} \leftarrow z \leftarrow$ *built solution using an heuristc*
$best\_cost \leftarrow$ cost(z)
cycles $\leftarrow$ 0
while $time\_elapsed < global\_timelimit$ **do**
  $z_{curr} \leftarrow$ 2-OPT(z_{curr})
  if cost(z_{curr}) $< best\_cost$ **then**
    z $\leftarrow z_{curr}$
    $best\_cost \leftarrow$ cost(z_{curr})
  end if
  Bigger the value of cycles bigger the kick
  $z_{curr} \leftarrow$ K-KICK($cycles$)
  cycles $\leftarrow$ cycles+1
end while
return z

5.3.2 Tabu search

The tabu search is the second metaheuristic presented in this report. This method improves the local search and avoids falling back in a previous local minimum. To do that the concept of tabu is introduced: a set of banned solutions for preventing the algorithm to return an already visited result.

In this implementation of the tabu search the refining phase is performed by a 2-opt move as it happened in section 5.3.1. The method for escaping from this minimum is to perform a kick like in the previous section. The main difference is that during the VNS it is possible to fall again and again in the previous solution, while using the tabu search this is not possible.

The worsening move is performed by swapping two non-consecutive edges and then declaring them as tabu: they cannot be swapped for a while. This process precludes the possibility of falling into a cycle of deterioration-recover, where the final solution is always the same. This algorithm will continuously worsen the solution until a non-tabu move is performed.

The way in which this method is implemented is the following: each time a worsening action is performed, the nodes selected are declared as tabu and the edges connected with them cannot be changed by any improving moves. If this always happens, the list of tabu nodes becomes so large that no more moves are allowed. To avoid this case, a new variable called *tenure* is introduced: the main idea is to limit how many times a tabu is valid.

To do that, a node tabu (for example i) is declared by inserting into an array the iteration number h, so $tabu\_nodes[i] = h$. This constraint will last for a *tenure* number of times, following this rule:

$$iteration\_number - tabu[i] \leq tenure \tag{5.6}$$

A good choice of *tenure* is crucial to allow the algorithm to perform adequately. The best decision is to make it variable based on the number of iterations already completed. In algorithm 9 can be seen the proceeding of this method.

5.3.3 Genetic algorithms

The last algorithm presented is the genetic one. Its main purpose is to generate a good solution to a problem by emulating natural selection. In particular, this evolution process is composed by different phases: reproduction (with recombination), selection, and mutation.

How this algorithm is going to emulate this idea is: each "generation" (or epoch) of individuals is represented by a set of feasible solutions of the TSP problem, and the next generation will be produced using the individuals of the previous epoch. Since it is a simulation of nature, some of the entities must die because they are too weak to survive.

To be able to perform this approach a random population must be constructed. This set is a group of feasible solutions to the TSP problem, but their costs are far from optimal since they are generated randomly.

Each epoch must go through a sequence of phases that are the following:

Algorithm 9 Tabu search

Input: $G = (V, E)$,$c : E \rightarrow \Re^+$, k, $global\_timelimit$
Output: z hopefully good solution

$z_{curr} \leftarrow z \leftarrow$ *built solution using an heuristc*
$best\_cost \leftarrow \text{cost}(z)$
$iteration\_counter \leftarrow 1$
while $time\_elapsed < global\_timelimit$ **do**
    Performs a 2-opt refining applying the constaraint described in section 5.3.2. The iteration counter is increased each time a move is performed. If no moves are allowed this method does nothing
    $z_{new} \leftarrow \text{TABU-2-OPT}(z_{curr}, tabu, tenure, iteration\_counter)$
    if $\text{cost}(z_{curr}) < best\_cost$ **then**
        z $\leftarrow z_{curr}$
        $best\_cost \leftarrow \text{cost}(z_{curr})$
    end if
    $first\_node \leftarrow RANDOM(|V|)$
    $second\_node \leftarrow RANDOM(|V|)$
    $tabu[first\_node] \leftarrow iteration\_counter$
    $tabu[second\_node] \leftarrow iteration\_counter$
    $z_{curr} \leftarrow \text{2-KICK}(first\_node, second\_node)$
    $iteration\_counter \leftarrow iteration\_counter + 1$
end while
return z

- parent selection: a certain number of pairs of individuals are randomly chosen from the population to be the parents of a child. With the focus to improve the solution cost over time, the individuals with better fitness are advantaged;

- offspring generation: for each pair of parents a new child is generated by combining their chromosomes;

- population management: to maintain constant the number of elements in the population, a set (equal to the number of new children) is killed. As happened during the parents selection, the group is random but the elements with higher fitness have more probability to die. Noticeably, the individual with the best fitness cannot be killed since it has greater chances to survive in nature;

- mutation: a random number of mutations is applied to a random number of elements. As well as the previous point, the champion (individual with the best fitness) is unlikely subject to mutation.

This algorithm follows the same path of the previous ones: it is run for some time and then the best solution found over the epochs is returned.

Implementation details

A little clarification is given on how this approach is implemented. Until this point, the solution was represented by an array of successors: each index of the array was the node number, and its content was the index of the next node in the cycle. To embrace the concept of chromosomes, a new type of representation is generated: the solution array will contain a sequence of indexes which will be the sequence of nodes in the tour. In this way, the combination of chromosomes is much easier to do.

A way to perform this is to select a cutting point in the parents chromosomes and then to choose the first part from one parent and the second one from the other. During this operation, there is the possibility that the second part of the chromosome presents a node already in the first half: in this case, the node is discarded and the merging process continues. Once this operation is finished, the nodes that are not in the child solution (because of the repetition of nodes) are added using the extra-mileage algorithm. This phase is called the fixing phase and even if there is not a corresponding process in nature, it is necessary to have a feasible solution after the generation phase.

The mutation process is not always applied since it is not always present in nature. It is applied only when the difference between the best and the worse fitness is too low. In this way, a possible best solution in the next epochs is generated.

Chapter 6

Experimental results and conclusions

In this section, the models described in this report are compared with the performance profile tool provided. Several runs were carried out to fully test the models for a more accurate conclusion. In particular, the best algorithm is the one that finds the optimal solution in the shortest time for most of the time. While analyzing the metaheuristics approaches it is impossible to compare them in a time scale, so instead of the time the comparison here is done between the cost of the solution obtained.

Since the TSP is a NP-hard problem, finding a winner is not always easy. Sometimes different methods can lead to very similar results, so several factors - such as consistency - are took in consideration.

To discuss, the outcome of the tests is plotted and so it is needed some explanation on how to understand them. While the x-axis portrays the time (or cost) ratio which will show what algorithm performs better, the y-axis shows the fraction of instances that the algorithm ends up winning. In general, the best method is the one that remains more on the left side of the chart.

As explained in section 2.1, the TSPLIB is used only to carry out if the implemented algorithms can reach the optimal value. To run these tests, a series of randomly generated instances are used.

6.1 Compact models

In this section the compact models presented in chapter 3 are evaluated focusing in particular on MTZ and its variants and and GG models. To test them, a set of 20 instance of 50 nodes was generated and was runned with a global timelimit of 30 minutes. The results are visible in figure 6.1.

From this image, it can be stated that the MTZ basic model is far from being the best one, and the best models are GG and MTZ_LAZY - which perform in a comparable way. However, it is possible to see the that GG model deliver a consistent solution of the instance, while MTZ_LAZY has reached the timelimit in some cases.

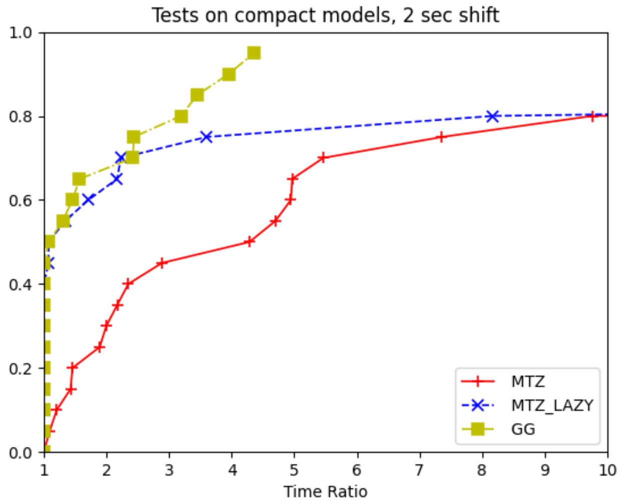


Figure 6.1: The comparison chart of the compact models.

6.2 SEC methods

To test the algorithms presented in section 4, a set of 20 instances with 350 nodes was created.

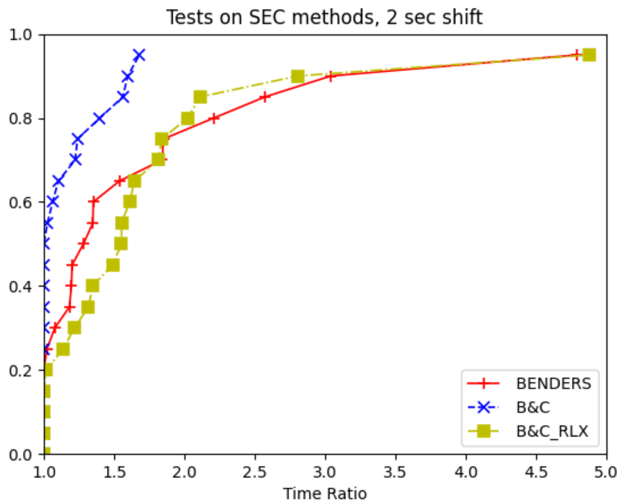


Figure 6.2: The comparison chart of the SEC methods.

The results in figure 6.2 showed clearly that the Branch and Cut method is the best approach of this comparison. The most outstanding outcome is that the B&C (Branch-and-Cut) relaxation method was in a trailing position even compared with the Benders method, while I was assuming that its performance was comparable with the normal B&C.

The meaning of this is the fact that the callback is applied each time a fractional solution is found. This suggests that it is called way more times than in the previous one and this leads to worse performance. For solving this problem, I have done deeper research on this approach. Figure 6.3 shows the B&C relaxation applied with a different probability.

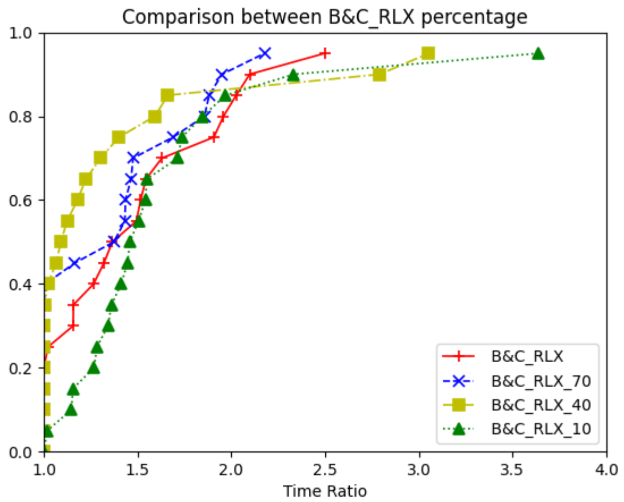


Figure 6.3: The comparison between different B&C relaxation.

This plot shows that the best performing method is the one with a percentage of 40%. In this way is possible to obtain the final chart with the best methods.

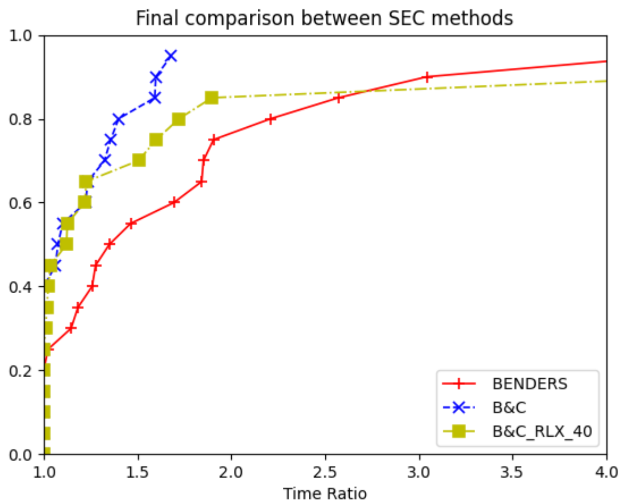


Figure 6.4: Final comparison between different B&C.

This last chart shows that the classic Branch and Cut method and the relaxation one applied with a percentage of 40% are similar, but the one that solves only the integer solution is considered the best among them.

6.3 Matheuristics

The algorithms from section 5.3 are analyzed in this segment. As explained in their relative chapter, for analyzing them properly a starting model has to be applied to whole the system. Having seen the previous results, the best option is choosing the classic Branch-and-Cut method since it is the one that can provide a faster solution to the instances.

To test the algorithms - which are the Hard-Fixing and Soft-Fixing - a set of 20 instances with 1000 nodes is randomly generated, with 30 minutes as time limit. It is essential to remark that for each method the number of edges blocked is variable, and each time a better solution is found this total number decreases. The results are visible in figure 6.5.

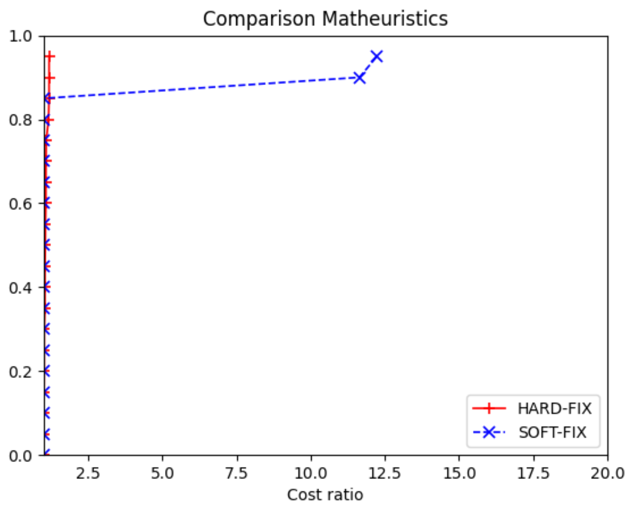


Figure 6.5: The comparison between the Matheuristics.

It is possible to state that - except for particular instances - these two methods are providing the same performances. On the other hand, if we zoom the left side of the previous chart we can see the real difference between the algorithms.

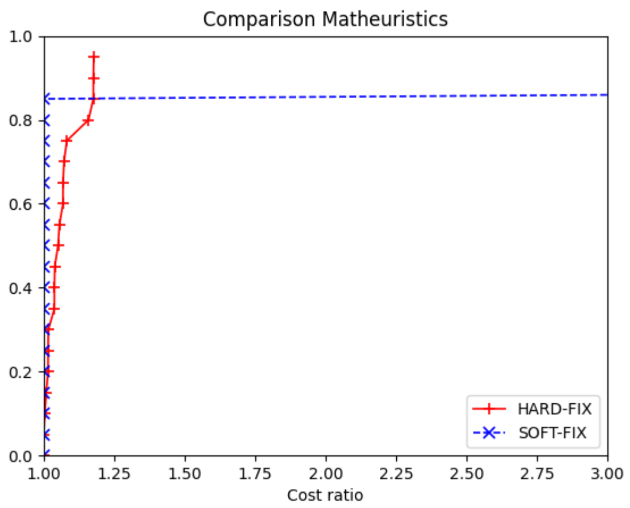


Figure 6.6: The comparison between the Matheuristics zoomed.

In figure 6.6 it is indisputable that the Soft-Fixing approach, with the exclusion of the particular worse results, is the best one of this section.

6.4 Metaheuristics

This is the last comparison between the methods presented in this project. In this section, metaheuristics presented in chapter 5.3 are compared: the VNS, the Tabu Search, and the Genetic algorithms.

To compare them, larger instances than the previous ones are created. A set of 20 instances with 2000 nodes is randomly built and then they are tested with a time limit of 30 minutes. The results are visible in figure 6.7.

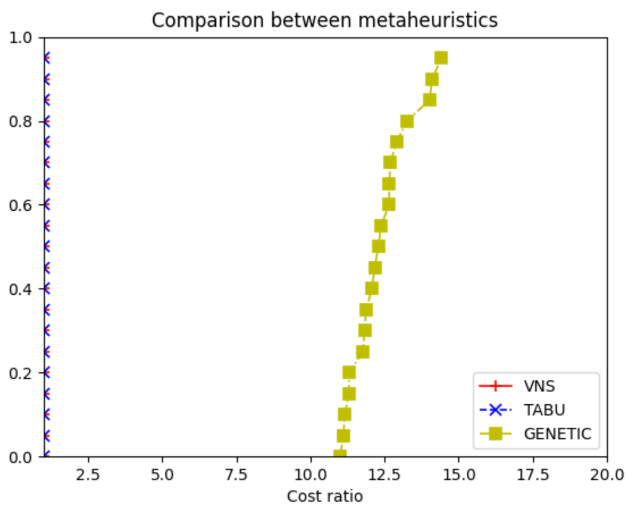


Figure 6.7: The comparison between the Matheuristics.

It is possible to notice that genetic algorithms lead to worst solutions. This is due to the fact that the starting population is totally randomly generated and so, within a small time limit, it is difficult to improve as much as in VNS or Tabu Search.

To see the winner of this section it is necessary to zoom the left side of the chart.

In figure 6.8 it is evident that the winner of this comparison is the VNS.

6.5 Conclusions

Among all the exact algorithms there is no doubt that the best performing one is the B&C method, which reached the optimal solution in the shortest amount of time. Given a sufficient time limit, it should be able to solve larger instances with enough efficiency.

The use of a compact model is infeasible since the effort to compute an optimal solution is enormous and surely not worth it. However in this category, the best performing one is the GG model because it delivers consistently an optimal solution.

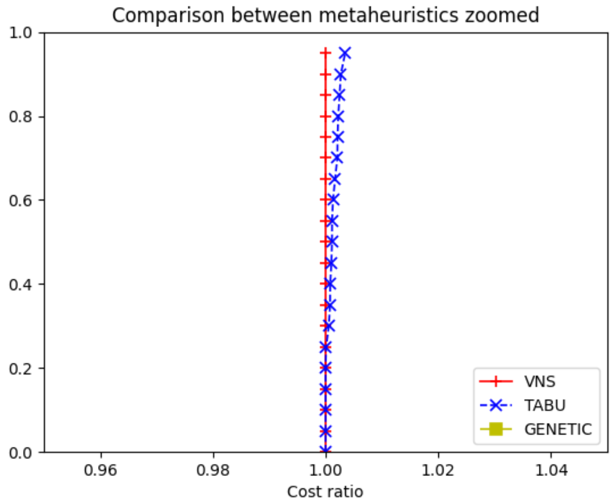


Figure 6.8: The comparison between the Matheuristics.

By all the matheuristic approaches the best-performing one was the Soft-Fixing, which brings the best solution most of the time. The Hard-Fixing method is still usable though because the solutions obtained from it are not too far away from the best ones.

The best metaheuristic approach is the VNS, which always provides the better solution. The use of the Soft-Fixing method is possible even with a higher number of nodes but - given a short amount of time - the solver hardly reaches a feasible solution.

References

[1] Fischetti M., *Lezioni di Ricerca Operativa*, Aprile 2018.

[2] *Concorde TSP Solver*, https://www.math.uwaterloo.ca/tsp/concorde.html, last consultation: 02/02/2022.

[3] *Heuristic (computer science)*, https://en.wikipedia.org/wiki/Heuristic_(computer_science), last consultation: 02/02/2022.

[4] *Hamming distance*, https://en.wikipedia.org/wiki/Hamming_distance, last consultation: 05/02/2022.

[5] *TSPLIB*, http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/, last consultation: 09/02/2022.

[6] Sawik, T. A note on the Miller-Tucker-Zemlin model for the asymmetric traveling salesman problem. *Bulletin Of The Polish Academy Of Sciences Technical Sciences.* **64** (2016,9)

[7] Orman, A. & Williams, H. A Survey of Different Integer Programming Formulations of the Travelling Salesman Problem. (2007)