University of Padova

Department of Information Engineering

Master Degree in Computer Engineering

Operations Research 2

TSP:
A CASE OF STUDY

Andrea Bugin
1180044

Ilie Sarpe
1179806

Academic Year 2018/2019

Abstract

This report is the result of our work done for the course of Operations Research 2 at the Department of Information Engineering (DEI) at the Padova University. During the course we studied, developed, implemented and tested several techniques for the Traveling Salesman Problem (TSP) which is a NP-Hard problem. Given the hardness of the problem, we developed advanced techniques for addressing it, in particular exact algorithms, heuristics, metaheuristics and matheuristics approaches. In this report we present all the techniques we studied, explaining our implementation choices. Moreover we performed an extensive empirical evaluation of all the methods implemented on more than 40 instances from the TSPLIB.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In computer science there are a lot of interesting classes of problems with a very strong impact in real world applications; one of the most fascinating is the class of NP-Hard problems. In a non rigorous way we can say that, it is very unlikely[1] that NP-Hard problems can be solved exactly in an efficient way, where efficient refers to the time of computation that depends on the size of the problem in input.

In this report we are going to present the work done during the operations research course, where we analyzed different methods to obtain a solution of a NP-Hard problem, the Travelling Salesman Problem (TSP).

The TSP can be described as follows: there is a traveling salesman who has to visit a given number of cities; starting from a city, he wants to pass through the others at most one time and then come back to the first one minimizing the overall distance traveled. In literature it is called "tour of minimum distance"; in Chapter 2 such intuitive idea is modeled rigorously. A quite famous example of this problem can be viewed in Figure 1.1 that represents the US with its 48 capitals, we can also see the tour that a salesman has to travel in order to minimize the overall distance and visit all the capitals.

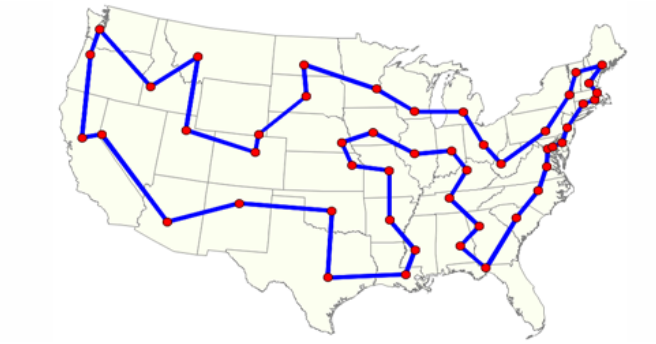


Figure 1.1: An example of the TSP applied to the 48 capitals of the United States.

In the next chapters we are going to present all the work done during more than three

[1] P vs NP is still an open problem.

month, which includes mathematical formulations, implementation and testing phases. In particular, this report is structured as follows:

- in Chapter 2 we will present all the models we have studied and implemented, showing the pros and cons of each model,
- in Chapter 3 we will present a set of algorithms that can solve the TSP, finding the shortest tour and the solution is proved to be the optimal,
- in Chapter 4 we will see other algorithms that can solve the TSP but the solution found is not proved to be optimal,
- in Chapter 5 we will present the results obtained from all the test performed,
- in Chapter 6 we will present the conclusions of our work.

All the source code developed is available at

https://github.com/AndreaB2604/RO2Project.git[2]

[2]For more information to compile and run the code, see the README.md.

Chapter 2

TSP models

In this chapter we are going to present Mixed Integer Linear Programming (MIP) models of the TSP problem. These are the models on which we focused during the course, in particular as we will show in the next chapters we used these models to implement techniques for addressing the TSP problem.

2.1 Conventional formulation (Dantzig, Fulkerson and Johnson)

In the introduction, we mentioned that the natural way to describe the problem is to suppose that a salesman has to travel among a set of given cities, visiting all of them only once, ending in the city were he started from and minimizing the overall distance travelled.

The natural way to model this scenario is to consider a graph $G = (V, E)$ where $V = \{v_1, \ldots, v_n\}$ also denoted, without loss of generality, with $V = \{1, \ldots, n\}$ is the set of the n *nodes* or *vertices* which represents the cities, and $E \subseteq (V \times V) \setminus \{\{i, i\} : i \in V\}$ is the set of *edges* with no self-loops. Intuitively if $\{i, j\} \in E$, then the city $i \in V$ is connected with the city $j \in V$ and since there is no direction the vice versa also holds, i.e., city $j \in V$ is connected to city $i \in V$. Another assumption we make is that the graphs of interest are *complete*, namely $E = (V \times V) \setminus \{\{i, i\} : i \in V\}$.

In this case, since there is not a direction on edges, we say that the graph is *undirected* and consequently *symmetric*. In this report we assume the graphs to be undirected, unless otherwise stated.

The second element we need to build the model is the notion of distance. We define a function $c : E \to \mathbb{R}^+$ that assigns to each edge $e = \{i, j\} \in E$ the cost $c(e) = c_e$. Observe that the function $c(\cdot)$ may represent arbitrary distances or weights in fact; the instances we are going to solve use different distance functions. We highlight the fact that for undirected graphs, travelling from $i \in V$ to $j \in V$ or vice versa has the same cost or distance.

Now the purpose is to find the sequence of edges, or equivalently, of nodes which

forms the tour of minimum cost, also called optimal tour. To address this task, we need to formulate the problem as a mathematical model.

The first model we are presenting comes from [4] with some little adaptations. Consider an undirected graph $G = (V, E)$: first of all we need to introduce some variables for describing the problem, the natural choice is to use:

$$x_e = \begin{cases} 1 & \text{if the edge } e \in E \text{ is chosen in the optimal tour,} \\ 0 & \text{otherwise.} \end{cases}$$

The model is:

$$\min \sum_{e \in E} c_e x_e \tag{2.1.1}$$

$$\sum_{e \in \delta(v)} x_e = 2 \qquad \forall\, v \in V \tag{2.1.2}$$

$$\sum_{e \in E_G(S)} x_e \leq |S| - 1 \qquad \forall\, S \subset V,\, |S| \geq 2 \tag{2.1.3}$$

$$x_e \in \{0, 1\} \qquad \forall\, e \in E \tag{2.1.4}$$

where (2.1.1) represents the objective function that we want to minimize, this corresponds to chose the tour with minimum cost. Equation (2.1.2) states that for each vertex $v \in V$, exactly two edges $e \in \delta(v)$ must be chosen, where $\delta(v) = \{e = \{i, j\} : (e \in E) \wedge ((i = v) \vee (j = v))\}, v \in V$, this corresponds to the fact that a node cannot be visited more than once. Constraint (2.1.4) states that the variables $x_e : e \in E$ are integer $\{0, 1\}$ variables.

With the previous three conditions the model is not complete since it allows the presence of subtours, that is, there can be sets of subtours, that satisfy the previous constrains without forming one single global tour. An example of such case is presented in Figure 2.1. In order to avoid such undesired behaviour, we have to introduce the Constraints (2.1.3), known as Subtour Elimination Constraints (SECs), which state that for each subset $S \subset V$ with $|S| \geq 2$ there can be at most $|S| - 1$ edges selected in $E_G(S) = \{\{i, j\} : i \in S, j \in S, i \neq j\}$; clearly these constraints prevent subtuors. Observe that the overall number of constraints of the model, due to the SECs, is $O(2^n)$, while the overall number of variables is $O(n^2)$, where we recall that $n = |V|$.

2.2 Miller-Tucker-Zemlin model

In this section we are going to present another MIP formulation of the TSP problem. In particular, we want to avoid the presence of subtours without using the Constraints (2.1.3) since, as already noticed, the number of such constraints is exponential in the number of vertices. We want to formulate the problem using only a number of

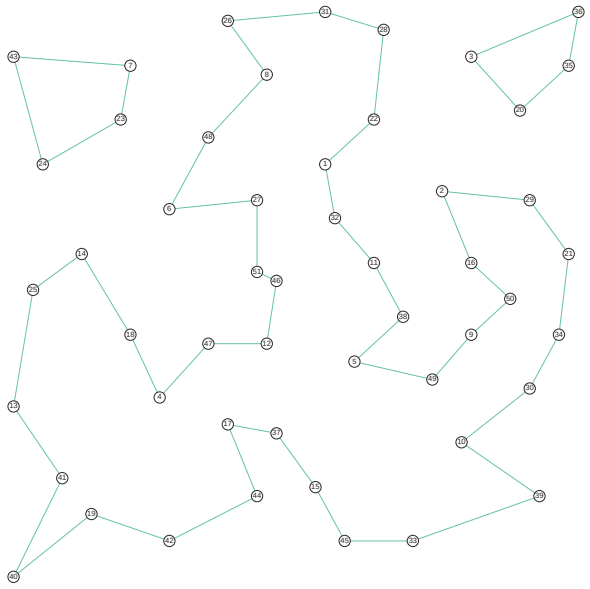


Figure 2.1: Example of solution, in this case the subtour elimination constraints were missing in the model.

constraints that is polynomial in the input size; such models are called in literature *compact models*. One of the most famous compact formulation for the TSP can be found in [9] and in [11].

The original formulation is for *directed* graphs $G = (V, A)$ where V is defined as in the previous section instead $A \subseteq (V \times V) \setminus \{(i, i) : i \in V\}$ is a set of *directed* edges, and in general $(i, j) \neq (j, i) \, \forall \, i, j \in V, \, i \neq j$. This means that also the costs $c\big((i, j)\big) = c_{ij}$ and $c\big((j, i)\big) = c_{ji}$ may differ in general; recall that in the Section 2.1 since the graph was undirected, travelling from $i \in V$ to $j \in V$ or vice versa had the same cost, here this property does not hold in general. As we have done previously, we need to define the variables of the problem, first of all we have:

$$y_{ij} = \begin{cases} 1 & \text{if the edge } (i, j) \in A \text{ is chosen in the solution s.t. } 1 \leq i, j \leq n \\ 0 & \text{otherwise.} \end{cases}$$

These variables are the equivalent of the $x_e : e \in E$ of the previous model, accounting for the fact that now the graph is *directed*.

Another way of accounting for the fact that there cannot be subtours is the following: label each visited node with a number starting from 1 up to n, assigning a label in an incremental way that is, if vertex u has label a $k \in [1, n-1]$ and the edge (u, v) is chosen then we assign to v the label $k + 1$. In this setting there cannot exist an edge chosen between a vertex with a label $2 \leq k_1 \leq n$ and a vertex labelled with $2 \leq k_2 \leq n$ if $k_1 > k_2$, which would imply a closed tour without the presence of all the nodes. In order

to account for such labelling we need to introduce some variables:

$$u_i = \text{position on the } i\text{-th node in the tour with } i = 1, \dots, n.$$

Now we state the model:

$$\min \sum_{i=1}^{n} \sum_{\substack{j=1 \\ i \neq j}}^{n} c_{ij} y_{ij} \tag{2.2.1}$$

$$\sum_{i=1}^{n} y_{ih} = 1 \qquad \forall\, h \in V \tag{2.2.2}$$

$$\sum_{j=1}^{n} y_{hj} = 1 \qquad \forall\, h \in V \tag{2.2.3}$$

$$y_{ii} = 0 \qquad \forall\, i \in V \tag{2.2.4}$$

$$y_{ij} \in \{0, 1\} \qquad \forall\, 1 \leq i \neq j \leq n \tag{2.2.5}$$

$$y_{ij} + y_{ji} \leq 1 \qquad \forall\, 1 \leq i < j \leq n \tag{2.2.6}$$

$$u_i - u_j + M y_{ij} \leq M - 1 \qquad \forall\, 2 \leq i \neq j \leq n \tag{2.2.7}$$

$$u_1 = 1 \tag{2.2.8}$$

$$u_i \in \{2, \dots, n\} \qquad i \in V\,, i > 1 \tag{2.2.9}$$

In (2.2.1) we have the objective function, (2.2.2) and (2.2.3) account for the fact that for each vertex there can be at most one edge incoming and one outcoming respectively, which models the fact that each node except the first one is visited only once. Conditions (2.2.4) and (2.2.5) account respectively for the fact that there cannot be self-loops and each variable must be integer in {0,1}. Since we assume $n > 2$ we should avoid tours with length 2, this is done by constraint (2.2.6) which states that at most one variable between y_{ij} and y_{ji} can be chosen in the solution. Equation (2.2.8) states that $1 \in V$ is the first node, which we observe is not a loss of generality since there must a exists a starting point and changing the starting point does not change the optimal solution in our case due to the fact that the graphs of interest are complete[1], and (2.2.9) accounts for the domain of the labels of each vertex $i \in V$, $i > 1$.

Constraints (2.2.7) model the labelling mechanism we described above. In literature these constraints are known as Big-M constraints. A rewriting of (2.2.7) is $u_j \geq u_i + 1 - M(1 - y_{ij})$, $\forall\, 2 \leq i \neq j \leq n$. If M is sufficiently large then if y_{ij} is not selected then the constraint is always true, otherwise if $y_{ij} = 1$ then the constraint becomes $u_j \geq u_i + 1$. Now it should be clear that this constraint models the logical condition

[1]Recall a complete graph is a graph $G = (V, A)$ where each edge $e \in A$ where $A \subseteq (V \times V) - \{(i, i) : i \in V\}$ exists, thus $A = (V \times V) - \{(i, i) : i \in V\}$.

"$y_{ij} = 1 \Rightarrow u_j \geq u_i + 1$". As explained before, this is another way of preventing subtours, in order to have the constraints work we need to set the value of M, following [11] one can set $M = n - 1$ which is the minimum value in order to obtain the desired behaviour.

Note that the dealing with directed graphs is a generalization of the setting of the previous Section 2.1 where undirected graphs were used. In particular, we can always solve undirected instances with techniques based on directed graphs just by setting $c_{ij} = c_{ji}, \forall 1 \leq i \neq j \leq n$, and in some sense losing the notion of "direction". Observe that the overall number of constraints of this formulation is $O(n^2)$ as we desired, while the overall number of variables is still $O(n^2)$, a well known fact is that this model produces a weak Linear Programming relaxation, which we will see is one of the reasons for which this model is not so used in practice.

2.3 Custom Model

The Miller-Tucker-Zemlin (MTZ) formulation it is not the only compact model available; in particular other formulations can be found in [10]. During the course we developed a custom compact model for undirected graphs $G = (V, E)$ as an exercise, as far as we know this is a completely new formulation.

The variables of the problem are:

$$x_{ij} = \begin{cases} 1 & \text{if the edge } \{i,j\} \in E \text{ is chosen in the tour,} \\ 0 & \text{otherwise.} \end{cases}$$

$$z_{vh} = \begin{cases} 1 & \text{if the node } v \in V \text{ is in position } h \in \{1,\dots,n\} \text{ in the tour,} \\ 0 & \text{otherwise.} \end{cases}$$

Now we state the model:

$$\min \sum_{\{i,j\} \in E} c_{ij} x_{ij} \tag{2.3.1}$$

$$\sum_{\{i,j\} \in \delta(v)} x_{ij} = 2 \qquad \forall\, v \in V \tag{2.3.2}$$

$$x_{ij} \in \{0,1\} \qquad \forall\, \{i,j\} \in E \tag{2.3.3}$$

$$z_{11} = 1 \tag{2.3.4}$$

$$z_{1h} = 0 \qquad \forall\, h \in \{2,\dots,n\} \tag{2.3.5}$$

$$z_{v1} = 0 \qquad \forall\, v \in V \setminus \{1\} \tag{2.3.6}$$

$$z_{vh} \in \{0,1\} \qquad \begin{matrix} \forall\, v \in V \setminus \{1\} \\ \forall\, h \in \{2,\dots,n\} \end{matrix} \tag{2.3.7}$$

$$\sum_{h=2}^{n} z_{vh} = 1 \qquad \forall\, v \in V \setminus \{1\} \tag{2.3.8}$$

$$\sum_{v \in V \setminus \{1\}} z_{vh} = 1 \qquad \forall\, h \in \{2, \dots, n\} \tag{2.3.9}$$

$$\sum_{t=3}^{n-1} z_{it} + x_{i1} \leq 1 \qquad \forall\, i \in V \setminus \{1\} \tag{2.3.10}$$

$$x_{ij} + \sum_{t=2}^{h} z_{it} + \sum_{t=h+2}^{n} z_{jt} \leq 2 \qquad \begin{array}{l} \forall\, i \neq j,\ i, j \in V \setminus \{1\} \\ \forall\, h \in \{2, \dots, n-2\} \end{array} \tag{2.3.11}$$

The x_{ij} variables are the same defined in Section 2.1 on page 3, i.e., $x_{ij} = x_e$ if $e = \{i, j\} \in E$, we remark the fact that the model was designed for undirected graphs although we adopt the notation x_{ij} instead of x_e for ease of description. Notice that since the graph is undirected we have $\{i, j\} = \{j, i\}$ and consequently $c_{ij} = c_{ji}$.

The z_{vh} variables account for the position of each node $v \in V$ in the tour, the position is expressed by $h \in \{1, \dots, n\}$. The idea behind these variables is that similarly to the MTZ constraints, we want to know the position of node $v \in V$ in the tour, this information will be used for preventing the occurrence of subtours in the solution. In (2.3.1) we have the objective function, the constraints (2.3.2), (2.3.3) play the same role of constraints (2.1.2) and (2.1.4) of the model in Section 2.1. Then for convenience and without loss of generality, we set the first node to be the first node visited, this is done in (2.3.4). Since the first node $1 \in V$ is in position one, it must not be in any other position but also no other node $i \in V$, $i > 1$ must be in position one, equations (2.3.5) and (2.3.6) model respectively the above facts. The Equations (2.3.8) state that each node must be in only one position, while the Equations (2.3.9) state that each position must be assigned to only one node; in the literature these are known as *assignment constraints*.

Now we have to express the link between the x_{ij} variables and the z_{vh} ones: we observe that the first node, which is in the first position, has to be connected only with the node in position two and the node in position n, i.e., the nodes $i_1, i_2 \in V$ for which $z_{i_1 2} = 1$ and $z_{i_2 n} = 1$. These facts are modeled by (2.3.10), which states that if the variable of edge x_{i1} is chosen than the node $i \in V$ must be in position n or 2. Finally, we have to say that if the edge $\{i, j\} \in E$ is chosen, namely $x_{ij} = 1$, than the nodes $i, j \in V$ must be in consecutive position in the tour; this is modeled by (2.3.11). In the Constraints (2.3.11), it is stated that if the $x_{ij} = 1$ then the nodes $i, j \in V$ must be consecutive, i.e., there must not be any gap between their positions in the tour, this clearly does not hold for the nodes in position 1 and n otherwise the problem would become infeasible. Note that the overall number of constraints of this formulation is $O(n^3)$.

Chapter 3

Exact Algorithms

In this chapter we describe the algorithms we implemented, based on the the models described in Chapter 2. All the implementations we are presenting are designed for symmetric TSP instances, since the focus of the course was on symmetric problems. All the algorithms in this chapter can solve exactly the TSP problem, since all of them are based on an exact solver that employs the branch-and-cut technique, which in practice in our work was IBM ILOG CPLEX.

3.1 Algorithms based on compact models

The algorithms based on compact models follow the general schema in Algorithm 1, first of all the variables of the model are described (line 1), then the objective function is stated (line 2) and the constraints are added to the model (line 3) as last thing the model is solved and the the result is returned (lines 4-5). This is a very general schema, in the next sections we discuss how these steps are adapted for each of the compact models considered in Chapter 2, in particular we highlight the most difficult steps.

Algorithm 1: Basic schema for an exact algorithm based on compact models

Input: $G = (V, E)$, $c : E \to \mathbb{R}^+$.
Output: z^* optimal solution to TSP on the input graph G.

1 model $\leftarrow$ `* Initialize the variables *`
2 model $\leftarrow$ `* model` $\cup$ `objective function *`
3 model $\leftarrow$ `* model` $\cup$ `constraints *`
4 $z^* \leftarrow$ `* solve optimally the model *`
5 **return** z^*

3.1.1 MTZ Model

In the implementation of the MTZ formulation, during the initialization of the variables (Algorithm 1, line 1) we maintained all the y_{ij}, $i, j = 1, \ldots, n$ which are n^2 variables plus the n variables u_i, $i = 1, \ldots, n$. A lot of attention must be paid implementing such

model since our focus is to solve symmetric instances, thus in the initialization of the objective function one should set $c_{ij} = c_{ji}$, $\forall 1 \leq i \neq j \leq n$, moreover since we do not allow self loops each of the c_{ii}, $i = 1, \ldots, n$ must be set to 0.

In the model implementation the above facts are the trickiest part, but a question may arise from the reader, if there is a way to avoid to load all the $O(n^2)$ constraints of (2.2.7) directly in the model, i.e., do not evaluate them directly but only where they are violated. This is an interesting topic since, lighter models (in terms of number of constraints) in CPLEX, and in general in all the solvers, may perform better than models with a lot of constraints. Thanks to the functions offered by the CPLEX API we chose to implement the (2.2.7) constraints as *lazy* ones, this allows CPLEX to load these constraints dynamically only when these are violated. This is of course a questionable point but we are confident that "lighter" models may result in more efficient algorithms.

3.1.2 Custom Model

In the phase of initialization of the variables, reading the model in Section 2.3 one may ask if it is necessary to have all the $x_{ij}, i, j \in V$ variables which are exactly n^2, although the graph $G = (V, E)$ is undirected and the variables x_{ij} and x_{ji} once fixed $i, j \in V$ refer to the same edge $\{i, j\} \in E$. The answer to this question is that with some attention one can implement the model by reducing the number of the x's variables to $n(n-1)/2$.

First of all the x's variables can be the following ones $x_{ij} : i, j \in V, i < j$, moreover we do not need the variables $x_{ii} : i \in V$ since we are not using them, this reduces the number of variables to exactly $n(n-1)/2$. In performing such reduction, attention must be paid in implementing the constraints in order to avoid mistakes, now we present the crucial observations to implement correctly the algorithm.

First of all in implementing (2.3.10) one should pay attention that the variables $x_{i1} : i \in V \setminus \{1\}$ does not exist in the model, thus the implementation needs to consider the variables $x_{1i} : i \in V \setminus \{1\}$; note that if all the variables: $x_{ij} : i, j \in V, i > j$ are implemented then this step is not needed. As last thing based on the reduction we presented, in implementing (2.3.11) one may think to rewrite the constraint such that it holds $\forall\, i \neq j, i < j, i, j \in V \setminus \{1\}$, this would result in a completely wrong model since the constraint is not preventing a lot of subtours. In such case the constraint has to remain the same, but in considering the variable x_{ij} with $i > j, i, j \in V \setminus \{1\}$ one should swap the indexes of this variable that is, the variable to be considered in the constraint is x_{ji}, this would result in a correct model.

As we pointed out in Chapter 2, this model has $O(n^3)$ constraints, following the idea of the MTZ implementation, we decided to implement all of them as lazy, to have a lighter model that hopefully would result in faster optimization through CPLEX.

3.2 Conventional formulation Model

3.2.1 Loop Method

This first implementation of an algorithm based on the conventional formulation follows the lines 1-2 of Algorithm 1, thus the variables are declared according to the model and the objective function is defined, but then the implementation differs in handling the constraints. We remark that this model is stated for symmetric TSP, so we implemented all the $x_e, e \in E$ variables which are exactly $n(n-1)/2$ variables.

One of the biggest problems of this model is that the formulation has $O(2^n)$ SECs, so it is impossible to implement all the constraints in the model since it would result in a very high execution time and memory usage even for very small graphs. The technique used to handle the SECs is presented in Algorithm 2.

In line 2, we initialize the model with the variables, the objective function and the degree constraints (2.1.2) as already stated. Then we start solving the problem within a `while` loop (lines 3-4), at each iteration we check if the optimal solution[1] has more than one connected component (line 5), i.e., subtours, if so we need to add more constraints to the model to prevent those subtours, thus we add the a SEC for each connected component as reported in (2.1.3). If there is only one connected component (lines 7-8) then we return the optimal solution (line 9). This schema results in a very efficient algorithm since during the first iterations the model is solved very quickly, recall that the number of constraints at the beginning is linear in $n = |V|$, and in general at each iteration the constraints added to the model in line 6 are a small number and prevent a lot of solutions with subtours to the original problem. We remark that the TSP problem is NP-hard so it will always exists an instance that requires exponential computational time, but in practice this is not often the case.

Algorithm 2: Loop method - Basic

Input: $G = (V, E), c : E \to \mathbb{R}^+$.
Output: z^* optimal solution to TSP on the input graph G.

```
1 done ← false
2 model ← * Initialize variables and objective function *
3 while !done do
4     z* ← optimal_solution(model)
5     if components(z*) > 1 then
6         * Add to the model SECs for each connected component *
7     else
8         done ← true
9 return z*
```

The Algorithm 2 has a drawback, in fact we solve to the optimum the TSP problem at each iteration of the while loop, even though the constraints may not be sufficient

[1] Recall that we are using an exact solver so the solution is certified to be optimal.

to achieve the true optimum we desire, i.e., the constraints available at that moment may allow subtours in the solution, but in any case we have to wait the solver to find the optimal solution and this may require a lot in terms of time. One of the possible approaches to address this problem is to limit the execution time of the exact solver at each iteration of the while loop; if the solution is not optimal but has several subtours then we may add the SECs on each component; this may improve the execution time, the technique is presented in Algorithm 3.

Algorithm 3: Loop method - Improved

Input: $G = (V, E), c : E \to \mathbb{R}^+$, DEF_GAP.
Output: $\boldsymbol{z}^*$ optimal solution to TSP on the input graph G.

```
1  done ← false; i ← 1; j ← 1
2  T ← * Vector of execution times sorted increasingly of size t *
3  R ← * Vector of integers opportunely initialized of size r *
4  model ← * Initialize the model without SECs *
5  while !done do
6      z* ← optimal_solution(model, T_i, R_j)
7      gap ← * Gap in % between z* and the LP relaxation solution *
8      model ← * Add SECs to the model if components(z*) > 1 *
9      if (* solution is optimal *) then
10         done ← true
11     else if (components(z*) = 1) ∧(i < t) then
12         i ← i + 1
13         j ← 1
14     else if (components(z*) > 1) ∧(gap ≤ DEF_GAP) then
15         i ← (i = 1)? i : i − 1
16         if i ≤ ⌊t/2⌋ then
17             j ← (j = 1)? j : j − 1
18         else
19             j ← (j = r)? j : j + 1
20     else if (components(z*) > 1) ∧(gap > DEF_GAP) then
21         i ← (i = t)? i : i + 1
22         if i ≤ ⌊t/2⌋ then
23             j ← 1
24         else
25             j ← (j = r)? j : j + 1
26 return z*
```

The schema is based on the one in Algorithm 2 but with some crucial differences. In line 2 we declare a vector T of size t, representing the time limit given to the exact solver

at each iteration, we investigated two different ways of limiting the computational time deterministic and non-deterministic. In fact CPLEX, which is the exact solver employed for all the exact algorithms, has a lot of parameters, some of them allow to set the time limit in seconds or in *ticks* which is a deterministic time. T is sorted increasingly with T_t corresponding to the maximum timelimit available, i.e., no time limit is really applied.

Another important parameter that CPLEX allows to control is the frequency of application of the RINS[2] algorithm, which is a powerful heuristic technique that aims at finding a "good" integer solution. A key observation is that applying RINS often, would result in a very slow run time since it is a powerful tool but, requires non-negligible computational time; if RINS instead is applied rarely then the run would be faster but, the solution may have an objective value much higher than the one obtained with RINS applied often. In line 3 we declare the vector R of size r which contains values of the frequency of application of RINS during the decision tree. The first value corresponds to default RINS, then the values are sorted increasingly with the frequency of application, if the value is lower (which corresponds to a number in position R_j where $j > \lfloor r/2 \rfloor$) then RINS is applied more often.

The initialization of the vectors T and R is a very delicate topic, in fact there is no theoretical guarantee on their setup to obtain bounds on the computational time; moreover the computational time depends on both of them, thus in fixing these parameters the user may use knowledge on the problem but also should pay attention to the algorithm, to be general enough for solving different instances, avoiding in some sense to "overfit".

At each iteration in line 6 we solve the model within a given timeout with RINS set appropriately. In line 8 we check if the solution has more than one connected component, if so then we add the SECs on the current solution, one for each connected component. In line 9 we check the optimality of the solution, if the solution is optimal then we should terminate the loop, clearly the solution is optimal when it is obtained with no limit on the computational time and has only one connected component. In line 11 if we have only one connected component but the time of the computation was bounded, then we cannot be sure of the optimality of the solution, in this case we simply increase the execution time and set RINS to the default value (lines 12-13).

To explain the last part of the algorithm we need to mention a feature available through CPLEX. When CPLEX is used to solve a model, the user can retrieve the best integer solution found but also the gap between the integer solution and the solution of the LP relaxation, i.e., the solution that does not forces the variables to be integer. This gap, although sometimes misleading, it is a very useful information on how close the solution $\boldsymbol{z}^*$ is to the optimal one at that moment. Recall that in our procedure, $\boldsymbol{z}^*$ may be optimal but constraints may miss to the model so the solution may have several connected components. We compute this gap in line 7, then we use it in the following ways:

- If the solution has more then one connected component and the gap is below a

[2]RINS is usually applied every $X \in \mathbb{N}$ nodes of the decision tree, CPLEX allows the user to set the value of X.

DEF_GAP, which acts as a threshold, then we reduce the computation time if the time is not already minimum. Moreover we reduce the frequency of RINS if the time limit is not high, i.e., i is lower than $\lfloor t/2 \rfloor$, otherwise we increase the RINS frequency (lines 14-19).

The idea behind these steps is that if the gap is small, then CPLEX had enough time to compute a solution very near to the optimal one, thus we try to reduce the computation time such that hopefully we may add the constraints much faster. At the same time, we look at the time limit to assign a RINS frequency, in particular if this time is not high then we decrease the application of RINS otherwise we increase it. The idea behind this choice is very simple, if we have more time of computation then we may try to obtain a better solution applying RINS more often.

- Symmetrically to the previous case, if the solution has more then one connected component but the gap is greater than DEF_GAP, then we chose to increment the time of the computation so hopefully we will add more constraints in the next iteration. If the time is not high then we set to the default the RINS frequency of application, otherwise we increase it (lines 20-25). Similar observations to the previous point, on the choices of increasing or decreasing RINS, can be made also in this case.

Clearly the algorithm is correct since we return $\boldsymbol{z^*}$ only when there is one connected component and no upper bound on the time of the computation was applied.

3.2.2 Callback Method

Until now, the algorithms based on the conventional formulation we devised, solve multiple times the TSP problem on relaxed instances, adding the SECs when multiple tours are in the solution, to achieve the true optimum we desire. Another way to add such constraints, is to exploit the branch-and-cut technique used to solve exactly the TSP problem. In particular we observe that when we solve exactly the model without constraints, several integer solutions are found during the decision tree that is, the idea is to add the SECs right when an integer solution with subtours is found. This leads to Algorithm 4 in which we instantiate the model and the objective function as usual (lines 2-3). Then we add to the solver a procedure named *callback* (line 4). The callback is a function that is called by the solver when an integer solution during the decision tree of the branch-and-cut technique is found; the callback checks if there are multiple connected components (line 8), if so it adds the SECs constraints to the model, then the resolution continues (lines 9-11). As last thing we solve the model and return the solution (line 5-6).

With this method we develop only a single decision tree, instead of having multiple trees for solving multiple times different problems such as in the loop method. This hopefully results in a faster computation. Attention must be paid in the implementation of such technique, in particular according to the exact solver employed the implementation may differ much; the crucial point is in adding correctly the constraints. In our

implementation we used CPLEX which allows the user to specify the callback in a very natural way. Moreover this schema can be used in a multi-threading environment which results in a very efficient tecnique.

Algorithm 4: Callback method

Output: z^* optimal solution to TSP on the input graph G.

```
1  Function Main(G = (V, E), c : E → ℝ⁺):
2      model ← * Initialize the variables *
3      model ← * model ∪ objective function *
4      solver ← * initialize the callback function within the solver *
5      z* ← solver(model)
6      return z*
7  Function callback(x, model):
8      comp ← * number of components in the integer solution x *
9      if (comp > 1) then
10         * Add SECs on each connected component to the model *
11     return
```

3.2.3 UserCutCallback Method

The name callback denotes a general class of function that are "called" under some conditions. As a special case, you may look at Algorithm 4 and realize that the special callback described in that algorithm is called when an integer solution is found. Looking at the branch-and-cut technique we decided to try to apply the same idea of "function that is called when a condition is verfied" to fractional solutions found during the exploration of the decision tree by the solver. In particular the idea is to apply the SECs on fractional solutions, namely if these solutions are common in the decision tree then applying immediately these constraints we may save a lot of time of computation. This led us to define the Algorithm 5.

The schema is the same as the Algorithm 4 with the difference that in line 4, also a UsrCutCallback must be defined. This function is used when a fractional solution is found during the decision tree, in particular we call this function only when the dept of the three is less than or equal to 10, this is due to the fact that this function employs several algorithms that are time consuming.

The function checks how many components there are in the actual solution, if there are multiple connected components then it adds the SECs on each connected component (lines 14-15). Recall that the solution is fractional thus even if the solution is connected many variables may have non-integer values, we try to apply the constraints after separating this integer solution. In particular we used flow-based algorithms integrated in the *concorde*[3] package, which can compute in very fast way the global min cut, in particular

[3] `http://www.math.uwaterloo.ca/tsp/concorde.html`

we apply the constraints on each cut $(S, V \setminus S)$ having capacity less than or equal to a cutoff value.

Algorithm 5: UserCutCallback method

```
   Output: z* optimal solution to TSP on the input graph G.
1  Function Main(G = (V, E), c : E → ℝ⁺):
2      model ← * Initialize the variables *
3      model ← * model ∪ objective function *
4      solver ← * initialize the callback functions within the solver *
5      z* ← solver(model)
6      return z*
7  Function callback(x, model):
8      comp ← * number of components in the integer solution x *
9      if (comp > 1) then
10         * Add SECs on each connected component to the model *
11     return
12 Function UsrCutCallback(x, model):
13     comp ← * number of components in the fractional solution x *
14     if (comp > 1) then
15         * Add SECs on each connected component to the model *
16     if (comp = 1) then
17         * Add SECs on the separated fractionary solution *
18     return
```

As the reader may observe, computing the flow is expensive so we decided to limit this callback to the maximum depth of 10 in the decision tree, which is a reasonable trade-off between the number of constraints applied (overall efficiency of the branch and-cut technique) and the time spent in executing flow algorithms. This approach resulting from the combination of the two callbacks results in a very efficient algorithm as we will see in the Chapter 5, where experimental evaluations are reported.

Chapter 4

Non-exact Algorithms

In this chapter we are going to present other techniques that can be used to address the travelling salesman problem; unlike in the previous chapter, where our focus was to develop exact algorithms, in this chapter we focus on the design of algorithms that does not guarantee the optimality of the solution, in contrast these techniques scale to large instances and are much more efficient in the computational time. Many such approaches exists in literature, they can be divided in "class" of algorithms, our focus was on the following categories:

- approximation algorithms,
- matheuristic approaches,
- heuristic and metaheuristic algorithms.

In this chapter we present the different class of algorithms introduced above, discussing the idea behind each algorithm, the techniques we implemented and the choices we made in the design of the techniques.

4.1 Approximation algorithms

An approximation algorithm is defined as follows.

Definition 4.1.1 (Approximation algorithm [2])**.** We say that an algorithm A has an *approximation ratio* of $\rho(n) \geq 1$ if, for any input size n, the cost C_A of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:

$$\max\left\{\frac{C_A}{C^*}, \frac{C^*}{C_A}\right\} \leq \rho(n) \tag{4.1.1}$$

If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a $\rho(n)$-*approximation algorithm.*

Our goal is to find an approximation algorithm with a small approximation ratio for the TSP. First of all we recall the fact that for the general traveling salesman problem[1]

[1]It is called *general* if the cost function $c(\cdot)$ is arbitrary

if $P \neq NP$, then for any constant $\rho(n) \geq 1$, there is no polynomial-time approximation algorithm with approximation ratio $\rho(n)$ (Section 35.2.2 of [2]). Nevertheless we are working with particular instances of the problem which belong to a subclass of TSP, called usually TRIANGLE-TSP a.k.a., *metric* TSP, where the cost function satisfies the triangle inequality, namely:

$$c_{ij} \leq c_{ik} + c_{kj} \qquad \forall\, i, j, k \in V,\, i \neq j \neq k \tag{4.1.2}$$

It can be proved that the TRIANGLE-TSP is still an NP-Hard problem. Now we present the approximation algorithm we used to address the TSP.

4.1.1 A 2-approximation algorithm

A possible approximation algorithm for the TRIANGLE-TSP is presented in 6; it computes the minimum spanning tree (MST) of the graph $G = (V, E)$ (line 2) and then visits the MST with a pre-order visit (line 3). The reason we compute the MST is that it is a lower bound of the optimal tour. The pseudocode follows:

Algorithm 6: 2-approximation algorithm for TRIANGLE-TSP

Input: $G = (V, E)$, $c : E \to \mathbb{R}^+$.
Output: A Hamiltonian cycle H of G which cost is within a factor 2 of the optimal solution of the TSP.

1 $r \leftarrow$ `* Choose a random node in V as root *`
2 $T \leftarrow PRIM\_DIJKSTRA\_MST(G = (V, E), c, r)$
3 $H \leftarrow PREORDER\_VISIT(T)$
4 `* Append` r `to` H `*`
5 **return** H

Now let's analyze the complexity of the Algorithm 6. The pre-order visit at line 3 has a complexity of $O(|V|)$, while the computation of the minimum spanning tree can be executed in $O(|V|^2) = O(|E|)$ using the Prim-Dijkstra algorithm, since the graph is complete; a description of the Prim-Dijkstra algorithm can be found in Section 7.5.2 of [4]. Thus based on the above facts, the overall complexity of the algorithm is $O(|V|^2)$. The next step is to prove that the algorithm returns an approximate solution within a factor two.

Theorem 4.1.1. *The Algorithm 6 is a* 2*-approximation algorithm for TRIANGLE-TSP*

Proof. Let H^* denote an optimal tour for the given set of vertices. We obtain a spanning tree by deleting any edge from a tour, and each edge cost is non negative. Therefore, the weight of the minimum spanning tree T computed in line 2 of Algorithm 6 provides a lower bound on the cost of an optimal tour:

$$c(T) \leq c(H^*) \tag{4.1.3}$$

where we defined the cost function of a subset of edges as

$$c(A) = \sum_{\{u,v\} \in A} c_{uv} \qquad \forall A \subseteq E$$

A full walk of T lists the vertices when they are first visited and also whenever they are returned to after a visit to a subtree. Let us call this full walk W. Since the full walk traverses every edge of T exactly twice, we have $c(W) = 2c(T)$. So combining this last equation and (4.1.3) we have that

$$c(W) \leq 2c(H^*) \tag{4.1.4}$$

Unfortunately, the full walk W is generally not a tour, since it visits some vertices more than once. By the triangle inequality, however, we can delete a visit to any vertex from W and the overall cost does not increase. By repeatedly applying this operation, we can remove from W all but the first visit to each vertex. For example if the full preorder walk is $< a, b, c, b, d, b, a, e, a >$ we can reduce it to $< a, b, c, d, e, a >$, where we removed all the other occurrences in the list of $< a, b >$ except the first one, recalling the property that $c_{ij} \leq c_{ik} + c_{kj}$, $\forall i, j, k \in V$; we observe that we left the last occurrence of the first node. This ordering is the same as that obtained by a preorder walk of the tree T. Let H be the cycle corresponding to this preorder walk. It is a hamiltonian cycle, since every vertex is visited exactly once, and in fact it is the cycle computed by Algorithm 6. Since H is obtained by deleting vertices from the full walk W, we have $c(H) \leq c(W)$. Combining this last inequality and the (4.1.4) we obtain that

$$c(H) \leq 2c(H^*) \tag{4.1.5}$$

which concludes the proof. □

In Figure 4.1 we can see all the steps performed by the Algorithm 6. We have proved that this is a 2-approximation algorithm, but this is not the best known approximation factor, there exists a $\frac{3}{2}$-approximation algorithm, called Christofides algorithm, which we are not going to present since it is beyond the purposes of this report; all the details of the algorithm can be found in [1]. As a an aside note, there exists a theorem which states that, if $P \neq NP$, no algorithm with an approximation factor of less that 123/122 may be achieved for the TRIANGLE-TSP [7].

4.2 Matheuristic approaches

Despite the approximation factor that the approximation algorithms can guarantee, in practice the solution can be way far from the optimum value. For this reason in the next sections we are going to present other methods, that in practice may obtain better solutions than those found by the approximation algorithms. Such techniques we are going to present, unfortunately, do not give any guarantee on the quality of the result, but as we will see in Chapter 5, empirically they result in very powerful techniques.

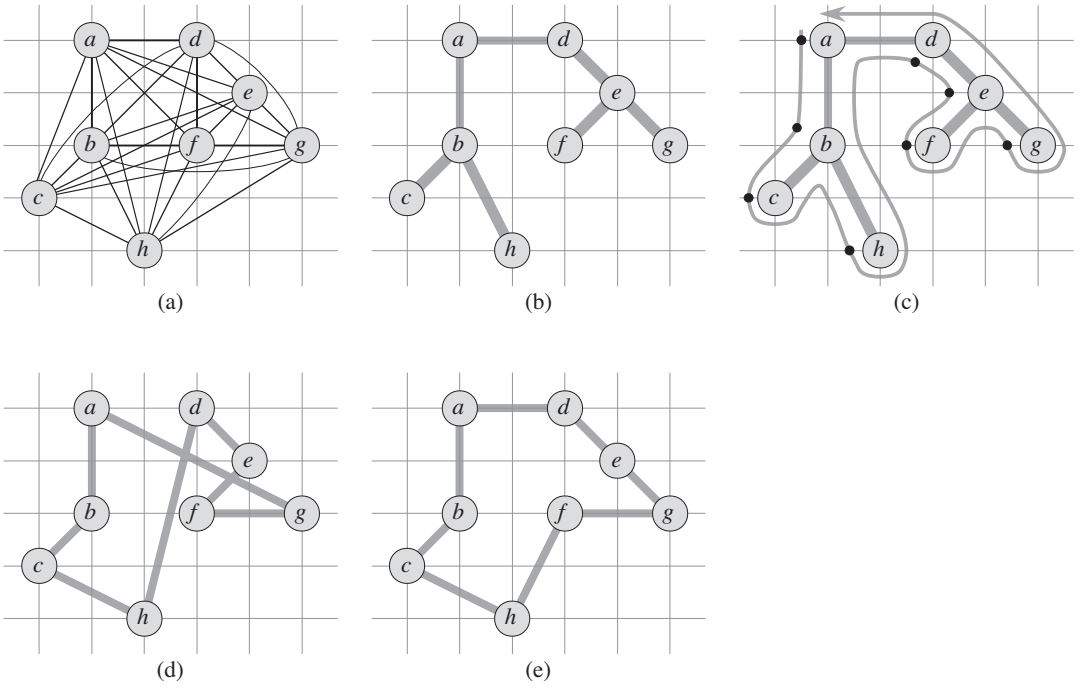


Figure 4.1: Steps of the 2-approximation algorithm for the TSP: starting from the complete graph in (a) to the minimum spanning tree in (b) and finally the pre-order visit in (c) and (d); the optimal tour is shown in (e).

"A *heuristic technique*, often called simply a *heuristic*, is any approach to problem solving or self-discovery that employs a practical method, not guaranteed to be optimal, perfect, logical, or rational, but instead sufficient for reaching an immediate goal. Where finding an optimal solution is impossible or impractical, heuristic methods can be used to speed up the process of finding a satisfactory solution" [12]. As we will present later, in this report we employ the knowledge about the TSP, to design suitable heuristic techniques for addressing it.

In this section, we are going to present the *matheuristic* approaches we developed. Matheuristics are optimization algorithms made by the interoperation of heuristics and mathematical programming (MP) techniques [13]. A matheuristic approach employs the mathematical model of the problem and combines it with some heuristic technique, we highlight that since this technique is still a heuristic there not exists any theoretical guarantee on the quality of the result. The techniques of such class of algorithm we are going to present are: the *hard fixing* and the *local branching*.

4.2.1 Hard Fixing

In a matheuristic approach we need an initial feasible solution. For the TSP problem a feasible solution is any tour of G, for instance a tour obtained by an approximation

algorithm, or even the naive one formed by the sequence of nodes $(1, 2, \ldots, |V|)$[2]. The first tour available will be denoted as the *reference solution*. We recall that each tour has a sequence of $|V|$ edges, which may be regarded as variables in a mathematical formulation of the TSP. The idea behind the *hard fixing* is to fix some variables of the reference solution in the mathematical formulation and then try to solve the new simplified problem, that's why the name "hard fixing". By fixing some variables of the solution, the problem has a fewer number of variables so an exact solver can easily handle such reduced model, solving it to optimality.

The variables to be fixed are x_e, $e \in E$, this is done by fixing their value to 1, which corresponds to fixing some edge of the reference solution in the new solution. In practice to each edge is associated a probability and according to some rule, each edge is chosen to be fixed. A reasonable rule can be to choose a fixed value $0 \leq p \leq 1$, and associate this probability to each edge, then choosing randomly with probability p if the edge is fixed or may vary in the new solution. At the end of this phase there are, on average, $(1-p)|E|$ variables not fixed, this can be proven by associating the random variable Y_e, $e \in E$ to each edge in the *reference solution*. Y_e is a indicator random variable which assumes value 1 if $e \in E$ is chosen to be fixed, then $\mathbb{E}[Y_e] = p$, so the number of fixed values is on expectation $p|E|$ by the linearity of the expectation. After the fixing step, the remaining problem has a lower number of variables to be optimized. Other rules are possible to assign the probabilities, for example based on the edge distance or the normalized distance, in this report we investigated only the one already described.

The hard fixing procedure is described in the Algorithm 7: the model is initialized (line 1), $\boldsymbol{x}_{ref}$, the reference solution, is obtained through the 2-approximation algorithm and passed to the solver as initial solution (lines 2-3). We then start a while loop which continues until a global $TIME\_LIMIT$ is reached (line 4), in the loop we fix some variables of the current solution (line 5) and then the fixed model is solved within a time limit tl (line 6). In our implementation for the step of fixing some edges, we have defined an array of probabilities $\mathcal{P}_t$ which in practice is $[0.9, 0.5, 0.2]$, we choose p starting from $\mathcal{P}_i, i = 1$. Notice that in the first iteration the model has, on average, $0.9|E|$ fixed variables. In the next iterations if the incumbent solution does not improve for at least 10 iterations we decide to choose the next value of the array (lines 10-11), in this way we allow the model to fix less variables to hopefully identify a better solution; however, in general, the greater the number of non fixed variables, the greater the time to solve the problem. As a limit if there are no fixed values then we are solving the original formulation.

Thus the choice of the vector $\mathcal{P}_t$ is a crucial step in the design of such algorithm, we empirically estimated these parameters, but we do not claim that is the best choice, other rules are possible. If the solution does not improve for 10 iterations we also multiply the time limit given to the exact solver by a factor $\alpha > 1$ (line 12) such that solutions with best objective functions may be found, with more computational time available.

[2]From now on we will refer to a *tour* as the compact form $(v_1, \ldots, v_{|V|})$ which represents the sequence of the visited nodes; the tour as sequence of visited edges can be obtained as $(v_1, v_2), \ldots, (v_{|V|-1}, v_{|V|}), (v_{|V|}, v_1)$.

During the loop, if the solution instead improves then we update the incumbent and reset to default the probabilities of each edge to be fixed (lines 7-9). In line 13 we restore the original bounds on the variables fixed in the current iterations, such that the next iteration it is still consistent with the whole procedure. The best solution $\boldsymbol{x}_{ref}$ found during the execution of the loop is returned at it's end (line 14).

Algorithm 7: Hard fixing matheuristic

Input: $G = (V, E)$, $c : E \to \mathbb{R}^+$, $TIME\_LIMIT$, probability values $\mathcal{P}_t$, time limit tl.

Output: A solution for the TSP problem

```
1  model ← * Initialize the model without SECs *
2  x_ref ← * Feasible solution for TSP *; i ← 1
3  * Set x_ref as starting point to the solver *
4  while time elapsed ≤ TIME_LIMIT do
5      model ← * Hard fixing of each variable according to P_i *
6      x_curr ← solution(model, tl)
7      if cost(x_curr) < cost(x_ref) then
8          x_ref ← x_curr
9          i ← 1
10     else if (* no improvement for 10 iterations *) then
11         i ← i + 1
12         tl ← tl * α
13     model ← * Restore the default bounds for each variable *
14 return x_ref
```

4.2.2 Local Branching

The *local branching* approach may be seen as a counterpart of the hard fixing. Proposed in [5], the idea behind the local branching is to add a constraint to the mathematical model, called *local branching constraint*, which forces to fix a certain number of variables, without choosing them explicitly. Recall that in the hard fixing technique, at each iteration starting from a reference solution, we fixed a subset of variables associating to each of them a probability and performing a random experiment, thus fixing in advance some variables. In the local branching technique, no variable has to be fixed in advance, the local branching constraint forces the model itself to fix at most a certain number of variables of the reference solution.

Now we introduce the local branching constraint. Given a reference solution $\boldsymbol{x}_{ref}$ the so called symmetric form of the constraint is:

$$\underbrace{\sum_{e \in E:\, \boldsymbol{x}_{ref}=1} (1 - x_e)}_{\substack{\text{number of variables flipping} \\ \text{their value from 1 to 0}}} + \underbrace{\sum_{e \in E:\, \boldsymbol{x}_{ref}=0} x_e}_{\substack{\text{number of variables flipping} \\ \text{their value from 0 to 1}}} \leq k \tag{4.2.1}$$

The summation on the left accounts for the number of variables which were active, i.e., their value was 1, in $\boldsymbol{x}_{ref}$, and they are not part of a new solution, hence their value is now 0. The second summation accounts for the complementary situation, when a variable had value 0 and now has value 1. The sum of the two summations must be less or equal than k which is the implicit constraint on allowing at most k variables to vary while fixing the others. We observe that high values of k may allow too much freedom to the model hence suitable values must be chosen, we will discuss this topic later. Since in any feasible solution the number of nonzero variables is a constant, we recall in fact that we have exactly $n = |V|$ non zero variables, we can write equivalently the (4.2.1) in the so called asymmetric form:

$$\sum_{e \in E \colon \boldsymbol{x}_{ref}=1} (1 - x_e) \leq k' \tag{4.2.2}$$

which can be rewritten breaking the summation, noting that $\sum_{e \colon \boldsymbol{x}_{ref}=1} 1 = n$, and rearranging we obtain:

$$\sum_{e \in E \colon \boldsymbol{x}_{ref}=1} x_e \leq n - k' \tag{4.2.3}$$

Now we discuss the choice of the parameter k', the value of this parameter defines how many variables we allow to vary in the new solution starting from the $\boldsymbol{x}_{ref}$, high values may allow too much variations and do not respect the idea of a *local* constraint. Since empirically has been proved that values up to fifteen of k' are effective, in our implementation we choose a list of different values, in particular: $K = [3, 5, 10]$.

The procedure is described in Algorithm 8, the model is initialized without the SECs (line 1), the 2-approximation algorithm is used and the solution is set as starting point for the whole procedure (lines 2-3). Then a while loop continues until a general $TIME\_LIMIT$ is reached (line 4), within the loop we add the local branching constraint to the model in the form of (4.2.3) starting with $k' = K_i$, $i = 1$ (line 5). Then the model is solved within a time limit tl; if an integer solution with a lower objective function than the best one available is found, the best solution, i.e., $\boldsymbol{x}_{ref}$, is updated and the value of k' is reset to K_i, $i = 1$ (lines 6-9). If no improvements are made for more that ten iterations then we choose a larger value for k' by increasing the value of i (line 11); this corresponds to allow more variables to vary from the reference solution, due to this choice the model may require much more time to be solved.

The value found with a greater k' may have have a lower objective function, since increasing its value allows more variables to change, thus better solutions may be found in the solution space. Moreover, we increase the time limit tl by multiplying it by a factor $\alpha > 1$, which corresponds to the fact that we allow more computational time to solve the model in the generic iteration (line 6), this is done in order to avoid to stuck in some local optima thus trying to achieve better solutions.

Algorithm 8: Local branching matheuristic

Input: $G = (V, E)$, $c : E \to \mathbb{R}^+$, $TIME\_LIMIT$, K vector of different k', time limit tl.

Output: A solution for the TSP problem

```
1  model ← * Initialize the model without SECs *
2  x_ref ← * Feasible solution for TSP *; i ← 1
3  * Set x_ref as starting point to the solver *
4  while time elapsed ≤ TIME_LIMIT do
5      model ← * Local branching constraint with |V| − K_i as rhs *
6      x_curr ← solution(model, tl)
7      if cost(x_curr) < cost(x_ref) then
8          x_ref ← x_curr
9          i ← 1
10     else if (* no improvement for 10 iterations *) then
11         i ← i + 1
12         tl ← tl ∗ α
13     model ← * Remove the local branching constraint *
14 return x_ref
```

4.3 Heuristic and Metaheuristic approaches

We already introduced the notion of heuristic in Section 4.2. "A *metaheuristic* instead is a higher-level procedure or heuristic designed to find, generate, or select a heuristic that may provide a sufficiently good solution to an optimization problem, especially with incomplete or imperfect information or limited computation capacity. Metaheuristics may make few assumptions about the optimization problem being solved, and so they may be usable for a variety of problems" [14]; thus metaheuristics are very general techniques usable to address different problems. In this chapter we will explain:

- The heuristic approaches we implemented for the TSP, in particular the *Nearest Neighbor Search* and the *2-Opt algorithm*
- How we adapted the general schema of metaheuristics to the TSP, in particular the *Variable Neighborhood Search* and the *Tabu Search*.

4.3.1 Nearest neighbor search (GRASP)

One of the first heuristic approaches that may be designed for addressing the TSP is based on the idea that we may start from some arbitrary node and chose to visit the node that is closest to our actual position and repeat such procedure from the new visited node. Given a graph $G = (V, E)$, the resulting procedure may look as follows [15]:

1. Mark all vertices as unvisited;

2. Select an arbitrary vertex, set it as the current vertes $u \in V$, mark u as visited;
3. Find out the shortest distance connecting the current vertex u and an unvisited vertex $v \in V$;
4. Set v as the current vertex u. Mark v as visited;
5. If all the vertices in V are visited, then terminate. Else, go to step 3.

The complexity in time of such algorithm is clearly $O(n^2), n = |V|$, since the graph is complete. One of the biggest problems of such procedure is that the choices in the last iterations, due to the greedy nature of the algorithm, are very inefficient in terms of costs and far from the optimal solution, i.e., the resulting tour might be improved just by looking at it since usually there are some "crossing" patterns, which can be easily removed, one example of such patterns may be found in Figure 4.2.

The idea to overcome the such greediness is to employ the Greedy Randomized Adaptive Search Procedure (GRASP) technique, which is a general schema that allows to transform deterministic algorithms in randomized ones. We first observe that, starting from some node $u \in V$ and choosing always the shortest distance results in a deterministic algorithm, since given an initial node the outcome it will always be the same. The idea is to randomize such outcome using the GRASP technique, this is done by randomizing the choice of the node to be visited next, given a current node.

In Algorithm 9 it is presented how we employ the GRASP technique for the nearest neighbor algorithm. In line 2 we initialize an empty list which will represent the best tour as sequence of node, i.e., recall the compact representation $(v_1, ..., v_{|V|})$. Since a single execution, with an arbitrary starting node $u \in V$, of the nearest neighbor is efficient, we execute such algorithm with all the available nodes as starting position that is, in line 3 we use each vertex $v \in V$ as a starting position. For each starting position we repeat l times the whole procedure (line 4). The general procedure, initializes a set of non visited nodes N, an empty list representing the current solution $\boldsymbol{x}_{curr}$ (line 7) and the starting node u (line 6). Then a while loop it starts (line 8), this while loop ends when the set of non visited edges N is empty, which corresponds to the fact that $\boldsymbol{x}_{curr}$ has size n so a solution is available, in lines 19-20 a comparison between the cost of $\boldsymbol{x}_{curr}$ and $\boldsymbol{x}_{best}$ is performed and $\boldsymbol{x}_{best}$ is updated consequently if the actual solution has a lower cost. In order to create $\boldsymbol{x}_{curr}$, we need to visit all the unvisited nodes (line 8), we append the current node to $\boldsymbol{x}_{curr}$ (line 9), u is removed from the non visited nodes (line 10). To choose the next vertex to be visited we need $d_u^{(k)}$ which is the obtained by firstly listing all the distances from u to the nodes in N, and then sorting this list by increasing values and $d_u^{(k)}$ corresponds to the k-th value in such list (line 11). The value $d_u^{(k)}$ is used to obtain the set D which represents the set of nodes in N and their distances from the current node, such that their distance is less than or equal to $d_u^{(k)}$ that is, D represents the set of the k nodes and their distances that are "closest" to the current node (line 12). Then we flip a coin, and with probability $1 - p$ we do not apply the GRASP technique (line 13), if so then we simply take the node in D with minimum distance from the current node, and set it as the current node (lines 14-15), then the procedures continues. If instead

GRASP technique is applied then we choose at random from the set D one element, the node from this element will become the next current node (lines 17-18), we recall that the set D contains the k nearest nodes and their distances from the current node, thus even if the closest node is not chosen we may still obtain a good solution. The GRASP procedure may allow the nearest neighbor to perform better during the last iterations since as we explained, the last part of the algorithm is where the nearest neighbor has poor performances in terms of quality of the result.

Algorithm 9: Nearest neighbor with GRASP technique

Input: $G = (V, E)$, $c : E \to \mathbb{R}^+$, an integer $k > 1$
Output: A feasible solution for the TSP problem

```
1  D ← ∅
2  x_best ← * Empty list *
3  foreach v ∈ V do
4      repeat
5          N ← V
6          u ← v
7          x_curr ← * Empty list *
8          while N ≠ ∅ do
9              x_curr ← x_curr.append(u)
10             N ← N − {u}
11             d_u^(k) ← * k-th distance in the sorted list, by increasing
                values, of distances from u to every node in N *
12             D ← {(u', d(u', u)) : u' ∈ N, d(u', u) ≤ d_u^(k)}
13             if FlipBiasedCoin(1 − p) = head then
14                 (u_c, d(u_c, u)) ← arg min_{(u',d(u',u))∈D} {d(u', u)}
15                 u ← u_c
16             else
17                 (u_c, d(u_c, u)) ← uniform(D)
18                 u ← u_c
19         if cost(x_curr) < cost(x_best) ∨ (x_best is empty) then
20             x_best ← x_curr
21     until l iterations are reached
22 return x_best
```

We observe that in the design of the Algorithm 9 several considerations on the parameters k, l, p must be done. We recall that k controls the number of candidates which are then used for the randomized choice if GRASP is applied, if $k \in O(n)$ then we may end up choosing random edges that can be very inefficient, hence we empirically set $k \leq 5$ which is a trade off at obtaining high quality results and randomizing sufficiently the procedure. The number l of repetitions of the procedure for a single starting vertex

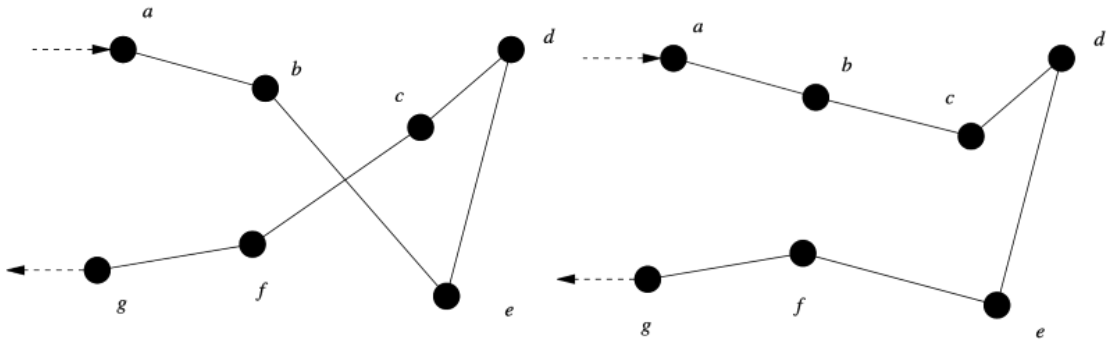


Figure 4.2: Example of "crossing" pattern (left) and it's resolution (right) achieved through the selection of the edges $\{b, c\}, \{e, f\}$ instead of $\{b, e\}, \{c, f\}$.

was set to $l = 10$, this value can be set arbitrary large it depends on how much time of computation is available. Finally the value of p, the probability of applying GRASP was set to 0.5, other values are possible, clearly higher values of p apply GRASP more frequently, one may also investigate to not have this probability fixed but making this larger when $\boldsymbol{x}_{curr}$ has size more than n/c, $c > 1$, thus when more than a fraction of the total nodes has already been visited.

4.3.2 2-Opt Algorithm

The basic idea of the 2-Opt algorithm is to resolve all the *crossing* patterns, an example of such pattern is reported in Figure 4.2, on the left we can see that the tour is $(\dots, b, e, \dots, c, f, \dots)$, observe that the tour on the right $(\dots, b, c, \dots, e, f, \dots)$ where we simply swap the portion of tour between vertices c and e in the tour, results in a lower or equal objective function since the distances are metric. Such intuitive idea leads to the Algorithm 10.

We start from an initial solution, in practice we used the 2-approximation algorithm, (line 1) and within a while loop (line 3) we evaluate all the possible swaps between a pair of nodes in the tour, which corresponds to swapping two edges, if there is a swap that improves the objective function then we perform it (lines 5-10). This is done within a while loop since exchanging some edges may introduce new "crossing" patters, so new iterations may be required to remove all of them. If a solution with a lower cost than the best solution available is found then we update the best solution (lines 11-12), otherwise we end the while loop (lines 13-14) since another iteration will produce the same outcome, i.e., no negative Δ. Δ, as defined in line 8, was found during the current iteration. At the end of the algorithm, the best solution is returned (line 15).

We observe that a swap of two edges starting from an initial solution $\boldsymbol{x}_{ref}$ defines a neighborhood in the solution space, denoted with $\mathcal{N}_2(\boldsymbol{x}_{ref})$ which represents the set of solution $\{\boldsymbol{x} : \boldsymbol{x}$ is a feasible solution for the TSP and $\boldsymbol{x}$ is obtained from $\boldsymbol{x}_{ref}$ swapping 2 edges$\}$. We highlight the fact that the Algorithm 10 may be regarded as a greedy procedure in the solution space where starting from one solution $\boldsymbol{x}_{ref}$ we move to the first

solution in $\mathcal{N}_2(\boldsymbol{x}_{ref})$ that improves the objective function, and we iterate such operation until no improving moves are available, so this also explains why this algorithm may stuck in local optima.

The 2-Opt algorithm is very simple but may improve in a very fast and effective way the value of the objective function, specially if the starting solution is not too far from the optimum, for example it can be used on the output of the 2-approximation algorithm as we also have done.

Algorithm 10: 2-Opt Algorithm

Input: $G = (V, E)$, $c : E \to \mathbb{R}^+$, a starting solution $\boldsymbol{x}$ as sequence of nodes.
Output: A feasible solution for the TSP problem

1 $\boldsymbol{x}_{best} \leftarrow \boldsymbol{x}$
2 done $\leftarrow$ false
3 **while** *!done* **do**
4 $\boldsymbol{x}_{curr} \leftarrow \boldsymbol{x}_{best}$
5 **foreach** $(v_1, v_2) \in (V \times V) \setminus \{(i, j) : i, j \in V, i \geq j\}$ **do**
6 $\delta^+ \leftarrow c((\text{prev}(\boldsymbol{x}_{curr}, v_1), v_2)) + c((v_1, \text{next}(\boldsymbol{x}_{curr}, v_2)))$
7 $\delta^- \leftarrow c((\text{prev}(\boldsymbol{x}_{curr}, v_1), v_1)) + c((v_2, \text{next}(\boldsymbol{x}_{curr}, v_2)))$
8 $\Delta \leftarrow \delta^+ - \delta^-$
9 **if** $(\Delta < 0)$ **then**
10 $\boldsymbol{x}_{curr} \leftarrow \texttt{swap}(\boldsymbol{x}_{curr}, v_1, v_2)$
11 **if** $cost(\boldsymbol{x}_{curr}) < cost(\boldsymbol{x}_{best})$ **then**
12 $\boldsymbol{x}_{best} \leftarrow \boldsymbol{x}_{curr}$
13 **else**
14 done $\leftarrow$ true
15 **return** $\boldsymbol{x}_{best}$

4.3.3 Variable Neighborhood Search (VNS)

The *Variable Neighborhood Search* is the first metaheuristic we are presenting. The basic idea of such technique is to start from a solution which may be regarded as a point in the solution space, then move randomly from this starting point to a solution in it's neighborhood in the solution space, according to some definition of the neighborhood, which may also be *variable*, then optimize the new solution with an heuristic method that converges to a local minimum and then iterate such procedure.

The whole procedure is presented in Algorithm 11, we start from an arbitrary solution $\boldsymbol{x}$ which is also the best solution available at the beginning of the algorithm (line 1). In line 3 a while loop is initialized and it continues until a global time limit TL is reached.

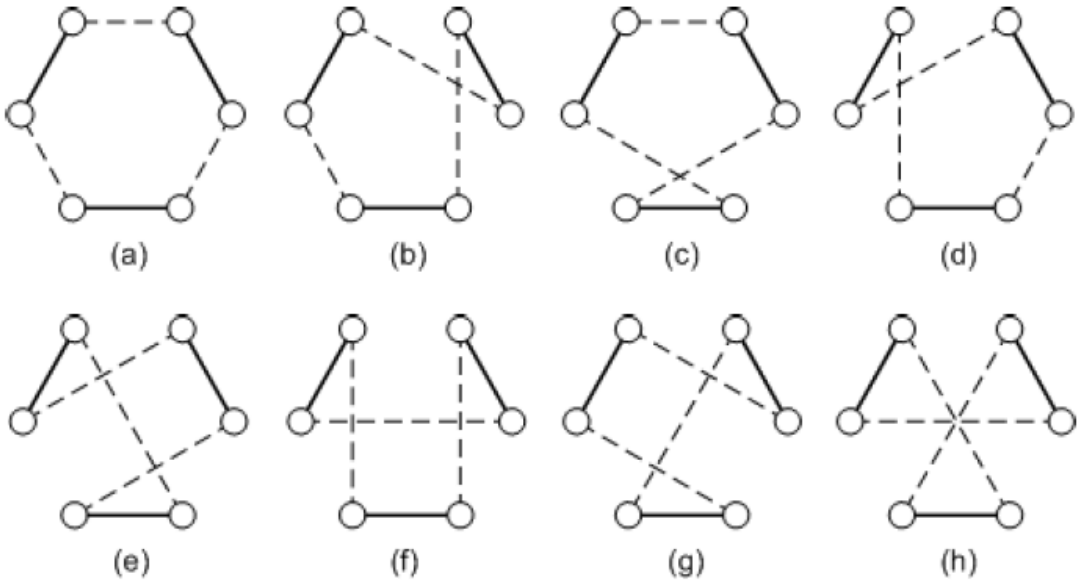


Figure 4.3: All possible swaps in the $\mathcal{N}_3(\boldsymbol{x})$.

In the while loop, at each iteration, we initialize a temporary solution $\boldsymbol{x}_{curr}$ to the best solution seen up to that iteration (line 4). Then we modify $\boldsymbol{x}_{curr}$ to a random point in the k-th neighborhood $\mathcal{N}_k(\boldsymbol{x}_{curr})$ of the current solution, the neighborhoods we defined are the following ones:

- $\mathcal{N}_2(\boldsymbol{x}_{curr}) = \{\boldsymbol{x} : \boldsymbol{x}$ is a feasible solution for the TSP and $\boldsymbol{x}$ is obtained from $\boldsymbol{x}_{curr}$ swapping the order of 2 edges in the tour$\}$. A solution in $\mathcal{N}_2(\boldsymbol{x}_{curr})$ is depicted in Figure 4.2, if $\boldsymbol{x}_{curr}$ is on the left then any swap of the order of two edges, such as the tour on the right, is in $\mathcal{N}_2(\boldsymbol{x}_{curr})$.

- $\mathcal{N}_3(\boldsymbol{x}_{curr}) = \{\boldsymbol{x} : \boldsymbol{x}$ is a feasible solution for the TSP and $\boldsymbol{x}$ is obtained from $\boldsymbol{x}_{curr}$ swapping the order of 3 edges in the tour$\}$. We observe that from all the possible 3 swaps, which are represented in Figure 4.3, we maintained only those swaps which are not achievable through $\mathcal{N}_2(\boldsymbol{x}_{curr})$, thus referring to the Figure 4.3 we allow only swaps: e, f, g and h.

- $\mathcal{N}_4(\boldsymbol{x}_{curr}) = \{\boldsymbol{x} : \boldsymbol{x}$ is a feasible solution for the TSP and $\boldsymbol{x}$ is obtained from $\boldsymbol{x}_{curr}$ swapping for 2 times the order of 2 edges in the tour$\}$.

After the random solution in the neighborhood is chosen (line 5), we optimize this solution, i.e., $\boldsymbol{x}_{curr}$, using the 2-Opt Algorithm (Algorithm 10). If we obtain a solution which improves the global minimum seen up to that iteration, we update $\boldsymbol{x}_{best}$ and reset the value of the neighborhood to $k = 2$, otherwise we increase k.

The idea behind such algorithm is that we do not want to stuck in a local minimum so we employ randomization, in particular choosing a random point in the $\mathcal{N}_k(\boldsymbol{x}_{curr})$ may

be seen as a "jump" in the solution space, for k small the "jump" is short so the solution we obtain may not vary much from $\boldsymbol{x}_{best}$, instead if k is large then we allow the random point to be much more different with respect to $\boldsymbol{x}_{best}$, there it comes the name "variable neighborhood search" from the fact that we control the size k of the neighborhood.

We implemented only three neighborhoods, other choices are possible and we highlight that in the literature $\mathcal{N}_4(\boldsymbol{x}_{curr})$ has different implementations, our idea is that with too many neighborhood we may "jump" too far from the best solution available, i.e., $\boldsymbol{x}_{best}$. Thus with many neighborhoods when k is large we may destroy some properties of $\boldsymbol{x}_{best}$ that we are trying to preserve. We underline the fact that since the solution space is not known, there is no algorithm that always performs better that another, moreover the randomness in the algorithm plays a key role in its execution. We are confident that our choices in implementing such algorithm are general enough to be suitable for solving the TSP with high quality results, moreover this technique is efficient and scalable, so it can be applied when exact algorithms cannot be used due to the large size of the input.

Algorithm 11: VNS algorithm

Input: $G = (V, E)$, $c : E \to \mathbb{R}^+$, a starting solution $\boldsymbol{x}$, a time limit TL
Output: A feasible solution for the TSP problem

```
1  x_best ← x
2  k ← 2
3  while time ≤ TL do
4      x_curr ← x_best
5      x_curr ← x ∈ N_k(x_curr)
6      x_curr ← 2-Opt(x_curr)
7      if c(x_curr) < c(x_best) then
8          x_best ← x_curr
9          k ← 2
10     else
11         (k = 4)? ← k : k + 1
12 return x_best
```

4.3.4 Tabu search

The tabu search is the second metaheuristic we are presenting. The word tabu is used to denote an action that must be avoided; the idea of the tabu algorithm is to explore the solution space through a set of tabu rules; in particular the tabu rules are used to prevent the algorithm to stuck in local optima, as we will explain.

The procedure is presented in Algorithm 12, as in the VNS algorithm, we start from an initial solution which is also the best solution available at the beginning of the algorithm (line 1). We initialize a solution $\boldsymbol{x}_{inc}$ which will represent the incumbent solution, this will be the solution we will modify to try to escape local minimums (line 2). In line 3 the tabu list is initialized, this list contains constraints on the past moves that

lead to the current incumbent solution, so actions that should be avoided in order to not be trapped in a local minimum. For the tabu search it is crucial the definition of *move*. A *move* is represented by a pair of nodes $\bar{m} = (v_s, v_f)$, $v_s \neq v_f$, $v_s, v_f \in V$, in particular given a tour $\boldsymbol{x}_{inc} = (v_1, \ldots, v_s, v_{s+1}, \ldots, v_{f-1}, v_f, \ldots, v_{|V|})$, where without loss of generality we assumed v_s to be visited prior than v_f, we consider $\bar{m}$ a move from $\boldsymbol{x}_{inc}$ to the new tour $\boldsymbol{x}'_{inc}$ if it holds that $\boldsymbol{x}'_{inc} = (v_1, \ldots, v_f, v_{f-1}, \ldots, v_{s+1}, v_s, \ldots, v_{|V|})$ which corresponds to the fact that the edges (v_{s-1}, v_s) and (v_f, v_{f+1}) in $\boldsymbol{x}_{inc}$, after the move become respectively (v_{s-1}, v_f) and (v_s, v_{f+1}), we highlight the fact that this is the same operation to obtain a point in $\mathcal{N}_2(\boldsymbol{x}_{curr})$ in the VNS technique. In order to avoid at the next iteration to perform the inverse move e.g., in the previous example swapping again the vertices v_s, v_f, following [8] we insert in the tabu list the tabu on the node v_s to move to the left in the tour, which for sure prevents the reversion of the move already performed. So the tabu list will contain a list of nodes that cannot swap, through a move, with a node to their left in the tour. It should be clear that the size of the tabu list or *tenure* is a crucial parameter, following [8] we set this parameter to a fixed value which is $|V|/3$, in literature many settings of such parameters were proposed, we believe that simpler solution may lead to more efficient algorithm in terms of computational time.

In line 3 we initialize a variable c which will represent the number of iteration performed from the last improvement of the objective function. A while loop it is started and it continues until a global time limit is reached (line 5). In the generic iteration the following steps are performed:

1. We use the routine `2-OptV2` which is described in algorithm 13, this routine computes the move to be performed at each iteration. In particular all the moves are evaluated and the move which minimizes the Δ' as defined in line 5 of Algorithm 13 is chosen, with some crucial observations:

   - If the move is allowed, i.e., it is not inside the *tabu* list, then it may be returned from the algorithm (lines 7-9 of Algorithm 13);
   - If the move it is inside the tabu list, but if applied to the current solution then we obtain a solution whose cost is less than the cost of the best solution seen up to that moment, then it may be returned from the algorithm (lines 7-9 of Algorithm 13).
   - Otherwise we do not allow the move (lines 10-11 of Algorithm 13).

2. Once the best move $m^* = (v_s, v_f)$ from the current position in the solution space is obtained (line 6 of Algorithm 12), we move to the solution determined by the move (line 7), and we insert in the tabu list the node v_s (line 8), such action will prevent the node v_s to move to the left in the tour at the next iterations, this is done to avoid to loop between moves.

3. If for more than θ iterations the objective function does not increase then we perform a random move, despite if it is or not in tabu list or if the new solution it will be inefficient (lines 9-13). The value of θ has a crucial impact on the quality

of the whole algorithm, we empirically estimated it value to 50, other choices are still possible and it may be interesting to exploit them. We observe that a low value of θ may destroy the original idea of the tabu search, instead high values may transform this step in a negligible one. So based on this facts and some empirical tests our choice was to set its value to 50.

4. If the objective function of the best solution $\boldsymbol{x}_{best}$ has a higher cost than the incumbent then we update consequently the best objective function, and we set c to zero (lines 14-16).

5. If the objective function does not improve then we increase c (lines 17-18).

6. The best solution found during the whole procedure is returned when the time limit is reached (line 19).

The overall complexity of every iteration is $O(n^2|tabu|)$, $n = |V|$ which in our case is $O(n^3)$, although it may look expensive, in practice this results in a very efficient algorithm, the worst case complexity comes from the fact that at each iteration of the foreach in line 2 of Algorithm 13 we may need to look for each $move = (v_1, v_2)$ at each entry in the tabu list (line 7), this does not happens in practice since very unlikely that all the $O(n^2)$ possible moves are sorted by increasing Δ's.

The tabu search is a very general technique, what we presented is our adaptation for the traveling salesman problem, we observe that by changing the definition of move, all the hyperparameters such the tabu length or the value of θ may change completely the whole procedure.

Algorithm 12: Tabu search algorithm

Input: $G = (V, E)$, $c : E \to \mathbb{R}^+$, a starting solution $\boldsymbol{x}$, a time limit TL, a threshold $\theta \geq 1$.

Output: A feasible solution for the TSP problem

$\boldsymbol{x}_{best} \leftarrow \boldsymbol{x} = (u_1, \ldots, u_{|V|})$
$\boldsymbol{x}_{inc} \leftarrow \boldsymbol{x}_{best}$
$tabu \leftarrow$ `* Empty list of size` $|V|/3$ `*`
$c \leftarrow 0$
while $time \leq TL$ **do**
  $m^* = (v_s, v_f) \leftarrow$ `2-OptV2`$(G, c, \boldsymbol{x}_{inc}, \boldsymbol{x}_{best}, tabu)$
  $\boldsymbol{x}_{inc} \leftarrow$ `*` $\boldsymbol{x}_{inc}$ `after the move` m^* `*`
  $tabu \leftarrow tabu$.insert(v_s)
  if $(c > \theta) \wedge (cost(\boldsymbol{x}_{inc}) > cost(\boldsymbol{x}_{best}))$ **then**
    $c \leftarrow 1$
    $r \leftarrow (v_1 =$ `random`(V), $v_2 =$ `random`$(V \setminus \{v_1\}))$
    $\boldsymbol{x}_{inc} \leftarrow$ `*` $\boldsymbol{x}_{inc}$ `after the move` r `*`
    $tabu \leftarrow tabu$.insert(v_1)
  if $cost(\boldsymbol{x}_{inc}) < cost(\boldsymbol{x}_{best})$ **then**
    $\boldsymbol{x}_{best} \leftarrow \boldsymbol{x}_{inc}$
    $c \leftarrow 0$
  else
    $c \leftarrow c + 1$
return $\boldsymbol{x}_{best}$

Algorithm 13: 2-OptV2 Algorithm for Tabu search

Input: $G = (V, E)$, $c : E \to \mathbb{R}^+$, a starting solution $\boldsymbol{x}_{inc}$, the best solution available $\boldsymbol{x}_{best}$, the list $tabu$.

s **Output:** A move (v_s, v_f)
$bestmove \leftarrow (:, :)$; $\Delta \leftarrow \infty$
foreach $move = (v_1, v_2) \in (V \times V) \setminus \{(i, j) : i, j \in V, i \geq j\}$ **do**
  $\delta^+ \leftarrow c((\text{prev}(\boldsymbol{x}_{inc}, v_1), v_2)) + c((v_1, \text{next}(\boldsymbol{x}_{inc}, v_2)))$
  $\delta^- \leftarrow c((\text{prev}(\boldsymbol{x}_{inc}, v_1), v_1)) + c((v_2, \text{next}(\boldsymbol{x}_{inc}, v_2)))$
  $\Delta' \leftarrow \delta^+ - \delta^-$
  if $(\Delta' < \Delta)$ **then**
    if $(move \notin tabu) \vee ((move \in tabu) \wedge (cost(\boldsymbol{x}_{inc}) + \Delta' < cost(\boldsymbol{x}_{best})))$ **then**
      $bestmove \leftarrow (v_1, v_2)$
      $\Delta \leftarrow \Delta'$
    else
      `continue`
return $bestmove$

Chapter 5

Experimental evaluation

In this chapter we will present the extensive experimental evaluation of the methods introduced in Chapters 3 and 4. All the experiments were executed on the *Blade* cluster at the Department of Information Engineering (DEI)[1], the exact solver we used is IBM ILOG CPLEX version 12.8. The experiments were performed on the dataset of 45 instances described in Table 5.1, from the TSPLIB[2].

Instance	**Number of nodes**	**Optimum value**
a280	280	2579
att48	48	10628
berlin52	52	7542
bier127	127	118282
ch130	130	6110
ch150	150	6528
d198	198	15780
eil101	101	629
eil51	51	426
eil76	76	538
gil262	262	2378
gr137	137	69853
gr202	202	40160
gr229	229	134602

Continued on next page

[1] More information on the architecture available at `https://www.dei.unipd.it/node/16542`

[2] Availbale at `https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/`

Instance	Number of nodes	Optimum value
gr96	96	55209
kroA100	100	21282
kroA150	150	26524
kroA200	200	29368
kroB100	100	22141
kroB150	150	26130
kroB200	200	29437
kroC100	100	20749
kroD100	100	21294
kroE100	100	22068
lin105	105	14379
lin318	318	42029
pcb442	442	50778
pr107	107	44303
pr124	124	59030
pr136	136	96772
pr144	144	58537
pr152	152	73682
pr226	226	80369
pr264	264	49135
pr299	299	48191
pr439	439	107217
pr76	76	108159
rat195	195	2323
rat99	99	1211
rd100	100	7910
rd400	400	15281
st70	70	675
u159	159	42080

Continued on next page

Instance	Number of nodes	Optimum value
ulysses16	16	6859
ulysses22	22	7013

Table 5.1: Instances used for the experimental evaluation. We see the name of the instance, the number of nodes and the certified optimum solution of the TSP on the specific instance.

5.1 Compact models

We evaluated the compact models on each instance, and as expected the most of the runs were unable to conclude, due to the large amount of memory necessary to execute such models. We do not report the data since, less than 90 runs over 306 were able to conclude for the *MTZ* formulation-based algorithm and less than 15 over 306 for the *Custom Model*. Thus the MTZ performs way better than the custom model, but still it is hard to extract some statistics since a lot of runs are incomplete or dumped, this may be also due to the constraints on the memory in the *blade* environment. These models are known to have different drawbacks, due to the large number of constraints and the weak relaxations, that leads to a very inefficient and expensive Branch and cut technique both from the running time and the usage of the memory.

5.2 Exact Methods

In this section we report the comparison between the exact methods we implemented, in particular the *Loop improved, Callback, UserCutCallback* methods. For the UserCut-Callback, as reported in Chapter 3 we implemented the API of concorde to separate fractional solutions.

We executed all the models with six different random seeds, $\{12, 893, 39848, 2648948, 201709013, 9837745292\}$, for each random seed we executed the run three times. We observe that the runs were using 4 cores, unfortunately we had not the opportunity to disable some advanced features which can influence the running time, such the hyper-threading. We executed each of the runs with both deterministic time limit (measured in ticks[3]) and non deterministic time limit (measured in seconds). In this chapter we report the results of the deterministic time limit set to 2127600 ticks[4][5], additional material can be found in Appendix C where we reported the results with non deterministic time limit, and tables with results for runs with deterministic time limit.

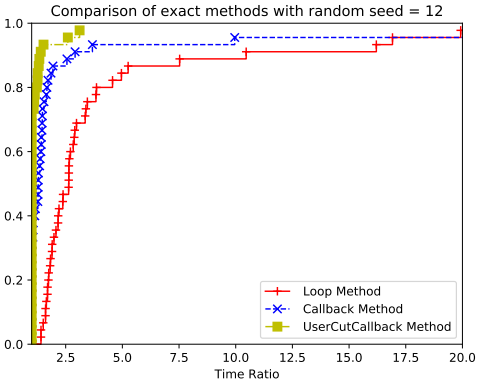
[3] `https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.cplex.help/CPLEX/Parameters/topics/DetTiLim.html`

[4] We choose this number because our limit for all instance was about a hour, so performing some runs as test we found that this number of ticks corresponds more or less to a hour in seconds.

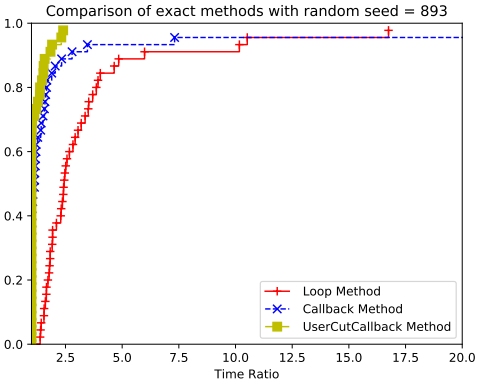
[5] It is a difficult task to estimate the exact number of ticks corresponding to one hour of computation since the *blade* cluster has serveral CPUs with different architectures which have different ticks/s rates.

Let t_1, t_2, t_3 be the times obtained from the three runs once fixed an instance and the random seed, we calculated an average time obtained as $\sum_i t_i/3$ and a geometric mean obtained as $\sqrt[3]{\prod_i t_i}$. Across all the methods, we highlight that the geometric mean and the arithmetic mean are quite similar, except for some rare cases, this happens when all the runs have approximately the same deterministic time which denotes that the runs were executed in a stable environment.

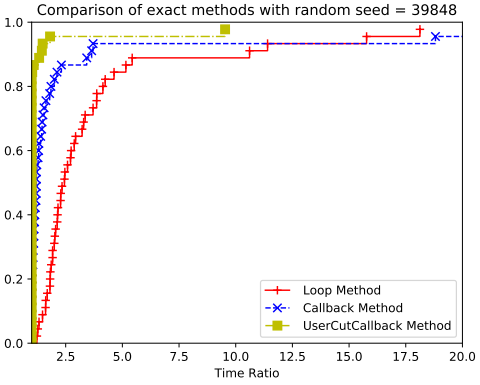
The comparison of the different techniques is done in Figure 5.1, where we compared for each random seed the different methods using the performance profile [3], and the comparison parameter we used is the geometric mean as described above. In the performance profile we have for each method the CDF for the performance ratio as defined in [3]: higher value means that the method performs better in terms of running time, so as we may observe the UserCutCallback is the method which is much more efficient with respect to all the random seeds and the different time ratios. The Callback method performs better than the Loop method, except when it comes to higher time ratios for all the random seeds except for the seed 2648948. We observe that different implementations of the Loop method may improve the running time of the method, instead it it is very interesting that the UserCutCallback outperforms all the other methods even if it employs time expensive routines, based on concorde API. In our implementation of the UserCutCallback method we limited the depth of such callback up to 10 in the decision tree, it would be interesting to understand how the variation of such parameter improves or not the running time.



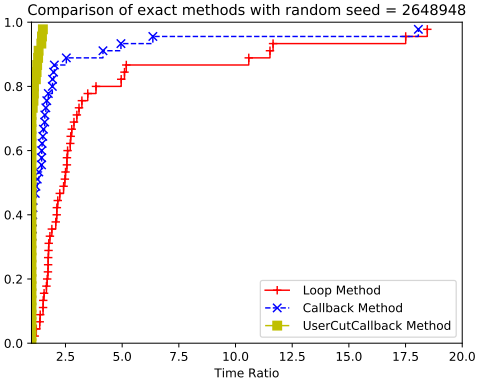
(a) Random seed = 12



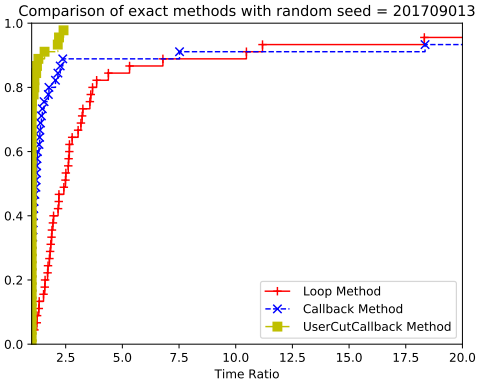
(b) Random seed = 893



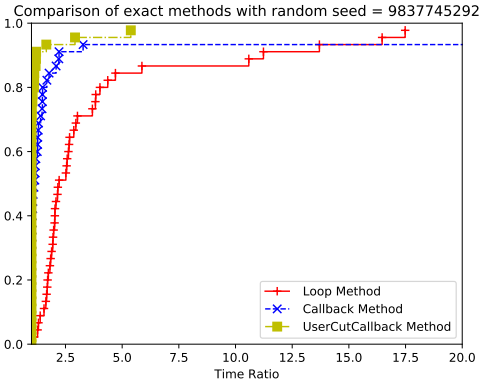
(c) Random seed = 39848



(d) Random seed = 2648948



(e) Random seed = 201709013



(f) Random seed = 9837745292

Figure 5.1: Performance profile of the exact methods with deterministic time limit

5.3 Matheuristics, heuristics and metaheuristics

To evaluate the performance of the matheuristics and the metaheuristic methods we run the methods within a non deterministic time limit, set to 45 minutes, every 15 minutes we extracted the information about the objective function to track the general run of the method on the specific instance. Each table we are going to present, reports the optimum value of the instance, the objective function at 0, 15, 30 and 45 minutes, and the Gap in percentage between the optimal solution and the result at 45 minutes for the different methods. Note that as initial solution we used the 2-approximation algorithm as initial seed, thus the starting objective function is always within a Gap of 100%. The solution at 0 means after the execution of the 2-approximation algorithm; the computational time of the approximation algorithm is obviously greater than zero and in the code we included it to be fair, but even for the instances considered with the greatest number of nodes, it takes few tens of seconds so it is in practice negligible on 45 minutes of computation.

Table 5.2 reports the results of the experimental evaluation of the matheuristics hard fixing and local branching algorithms. In general the hard fixing method achieves better solutions than the local branching, except for some instances such: ch150, pr226, pr264 kroA150, kroA200, rd400. Interestingly on these instances the local branching achieves very good solutions, this may be caused by the randomness in such algorithms but also some problems may have an intrinsic difficulty which makes them suitable to be solved with particular methods. For the instances ulysses16 and ulysses22 there were several issues with the blade cluster, thus the local branching runs are not available.

Table 5.3 reports the results on the metaheuristic techniques VNS and Tabu search. In the comparison of the two techniques we observe that for instances with more than 150 nodes usually the VNS obtains better results in term of gap, for small instances instead the Tabu performs better. The overall performances of the two methods are within at most a 3% of gap with the optimal solution, with a trend of at most 1%, which is a very impressive results, since we recall such techniques do not prove rigorous bound on the quality of the approximation and all the implementation are sequential and not parallel. As an aside note, we observe that in most of the runs where the result is not optimal, for both techniques, the results between 15 and 45 minutes are the same, thus the methods cannot escape the local minimum; to overcome such situation one may think to implement more neighborhoods for the VNS algorithm and change the length of tabu list in the Tabu search algorithm.

We observe that all the techniques have some drawbacks, but this is a natural consequence of the fact that the solution space it is unknown, thus one may launch several techniques on the same instance and at the end keep the best solution achieved; this would probably give satisfactory results and maybe would help to overcome part of the intrinsic hardness of the TSP.

Instance	Optimum value	Hard-Fixing (solution after)					Local Branching (solution after)				
		0 min.	15 min.	30 min.	45 min.	Gap (%)	0 min.	15 min.	30 min.	45 min.	Gap (%)
a280	2579	3486	2579	2579	2579	0.00	3486	3035	2989	2989	15.90
att48	10628	15974	10628	10628	10628	0.00	15974	10628	10628	10628	0.00
berlin52	7542	10564	7542	7542	7542	0.00	10564	7542	7542	7542	0.00
bier127	118282	156184	120996	118282	118282	0.00	156184	120843	118621	118282	0.00
ch130	6110	8533	6110	6110	6110	0.00	8533	6128	6128	6128	0.29
ch150	6528	10083	8706	8631	7320	12.13	10083	6554	6554	6554	0.40
d198	15780	19347	18242	17666	17666	11.95	19347	15871	15809	15793	0.08
eil101	629	890	805	728	639	1.59	890	726	682	682	8.43
eil51	426	603	426	426	426	0.00	603	426	426	426	0.00
eil76	538	706	538	538	538	0.00	706	538	538	538	0.00
gil262	2378	3566	2393	2378	2378	0.00	3566	3112	2809	2765	16.27
gr137	69853	94463	69853	69853	69853	0.00	94463	78842	78842	78842	12.87
gr202	40160	52615	40187	40160	40160	0.00	52615	47436	47068	47068	17.20
gr229	134602	179335	134602	134602	134602	0.00	179335	135563	134797	134658	0.04
gr96	55209	74694	55209	55209	55209	0.00	74694	55736	55291	55291	0.15
kroA100	21282	30713	21282	21282	21282	0.00	30713	22469	21634	21373	0.43
kroA150	26524	40904	38257	35171	35171	32.60	40904	26715	26524	26524	0.00
kroA200	29368	42564	41446	41323	40680	38.52	42564	30154	29811	29368	0.00
kroB100	22141	32654	25034	25034	25034	13.07	32654	22899	22899	22899	3.42
kroB150	26130	38720	29956	28952	28952	10.80	38720	35426	35426	35426	35.58
kroB200	29437	46233	29437	29437	29437	0.00	46233	29791	29791	29791	1.20
kroC100	20749	30964	20749	20749	20749	0.00	30964	23124	22738	21848	5.30
kroD100	21294	32575	21294	21294	21294	0.00	32575	21679	21294	21294	0.00
kroE100	22068	36709	22068	22068	22068	0.00	36709	22068	22068	22068	0.00
lin105	14379	21167	14379	14379	14379	0.00	21167	14379	14379	14379	0.00
lin318	42029	60978	42083	42083	42083	0.13	60978	50001	44771	44193	5.15
pcb442	50778	71540	64322	61427	57061	12.37	71540	62341	62341	62341	22.77

Continued on next page

Instance	**Optimum value**	**Hard-Fixing (solution after)**					**Local Branching (solution after)**				
		0 min.	15 min.	30 min.	45 min.	Gap (%)	0 min.	15 min.	30 min.	45 min.	Gap (%)
pr107	44303	56037	44761	44303	44303	0.00	56037	44347	44303	44303	0.00
pr124	59030	82761	59030	59030	59030	0.00	82761	68451	68451	68451	15.96
pr136	96772	150460	116821	108451	97044	0.28	150460	98694	98694	98496	1.78
pr144	58537	77614	58537	58537	58537	0.00	77614	58537	58537	58537	0.00
pr152	73682	90196	73818	73818	73818	0.18	90196	73682	73682	73682	0.00
pr226	80369	114702	112953	112953	112953	40.54	114702	91485	84802	83056	3.34
pr264	49135	72056	67231	67231	67231	36.83	72056	55169	51004	50471	2.72
pr299	48191	67318	48339	48339	48339	0.31	67318	63560	63560	61776	28.19
pr439	107217	144334	108363	107371	107371	0.14	144334	118238	112018	109912	2.51
pr76	108159	147668	108159	108159	108159	0.00	147668	114009	113238	113238	4.70
rat195	2323	3278	2323	2323	2323	0.00	3278	2981	2869	2762	18.90
rat99	1211	1635	1211	1211	1211	0.00	1635	1531	1239	1235	1.98
rd100	7910	12870	7910	7910	7910	0.00	12870	8042	7910	7910	0.00
rd400	15281	20963	19241	17930	17473	14.34	20963	16488	15810	15731	2.94
st70	675	1020	675	675	675	0.00	1020	675	675	675	0.00
u159	42080	54038	42080	42080	42080	0.00	54038	43304	42388	42080	0.00
ulysses16	6859	8011	6859	6859	6859	0.00	8011	-	-	-	100.00
ulysses22	7013	8506	7013	7013	7013	0.00	8506	-	-	-	100.00

Table 5.2: Results of the Hard-Fixing and Local Branching matheuristics

Instance	Optimum value	Tabu search (solution after)					VNS (solution after)				
		0 min.	15 min.	30 min.	45 min.	Gap (%)	0 min.	15 min.	30 min.	45 min.	Gap (%)
a280	2579	3486	2585	2585	2585	0.23	3486	2607	2607	2607	1.09
att48	10628	15974	10628	10628	10628	0.00	15974	10628	10628	10628	0.00
berlin52	7542	10564	7542	7542	7542	0.00	10564	7542	7542	7542	0.00
bier127	118282	156184	118313	118282	118282	0.00	156184	118616	118616	118616	0.28
ch130	6110	8533	6115	6115	6115	0.08	8533	6124	6124	6124	0.23
ch150	6528	10083	6528	6528	6528	0.00	10083	6533	6533	6533	0.08
d198	15780	19347	15809	15809	15809	0.18	19347	15793	15793	15793	0.08
eil101	629	890	629	629	629	0.00	890	629	629	629	0.00
eil51	426	603	426	426	426	0.00	603	426	426	426	0.00
eil76	538	706	538	538	538	0.00	706	538	538	538	0.00
gil262	2378	3566	2419	2419	2419	1.72	3566	2385	2385	2385	0.29
gr137	69853	94463	69951	69951	69951	0.14	94463	69853	69853	69853	0.00
gr202	40160	52615	40756	40756	40756	1.48	52615	40405	40398	40398	0.59
gr229	134602	179335	136972	136203	136203	1.19	179335	135365	135365	135365	0.57
gr96	55209	74694	55259	55209	55209	0.00	74694	55425	55425	55425	0.39
kroA100	21282	30713	21282	21282	21282	0.00	30713	21282	21282	21282	0.00
kroA150	26524	40904	26590	26590	26590	0.25	40904	26670	26670	26670	0.55
kroA200	29368	42564	29605	29605	29605	0.81	42564	29512	29512	29512	0.49
kroB100	22141	32654	22141	22141	22141	0.00	32654	22199	22199	22199	0.26
kroB150	26130	38720	26194	26147	26147	0.07	38720	26132	26132	26132	0.01
kroB200	29437	46233	29729	29729	29676	0.81	46233	29437	29437	29437	0.00
kroC100	20749	30964	20749	20749	20749	0.00	30964	20749	20749	20749	0.00
kroD100	21294	32575	21294	21294	21294	0.00	32575	21374	21374	21374	0.38
kroE100	22068	36709	22111	22068	22068	0.00	36709	22140	22140	22140	0.33
lin105	14379	21167	14379	14379	14379	0.00	21167	14379	14379	14379	0.00
lin318	42029	60978	43033	43033	43033	2.39	60978	42443	42428	42428	0.95
pcb442	50778	71540	51427	51427	51427	1.28	71540	51387	51312	51300	1.03

Continued on next page

Instance	**Optimum value**	**Tabu search (solution after)**					**VNS (solution after)**				
		0 min.	15 min.	30 min.	45 min.	Gap (%)	0 min.	15 min.	30 min.	45 min.	Gap (%)
pr107	44303	56037	44303	44303	44303	0.00	56037	44347	44347	44347	0.10
pr124	59030	82761	59030	59030	59030	0.00	82761	59030	59030	59030	0.00
pr136	96772	150460	96772	96772	96772	0.00	150460	96785	96785	96785	0.01
pr144	58537	77614	58537	58537	58537	0.00	77614	58537	58537	58537	0.00
pr152	73682	90196	73822	73690	73690	0.01	90196	73682	73682	73682	0.00
pr226	80369	114702	80831	80831	80831	0.57	114702	80369	80369	80369	0.00
pr264	49135	72056	49581	49488	49196	0.12	72056	49135	49135	49135	0.00
pr299	48191	67318	48548	48548	48548	0.74	67318	48230	48230	48230	0.08
pr439	107217	144334	108684	108684	108684	1.37	144334	107370	107370	107370	0.14
pr76	108159	147668	108159	108159	108159	0.00	147668	109085	109085	109085	0.86
rat195	2323	3278	2339	2339	2339	0.69	3278	2329	2329	2329	0.26
rat99	1211	1635	1211	1211	1211	0.00	1635	1212	1212	1212	0.08
rd100	7910	12870	7910	7910	7910	0.00	12870	8042	8042	8042	1.67
rd400	15281	20963	15709	15646	15646	2.39	20963	15341	15341	15341	0.39
st70	675	1020	675	675	675	0.00	1020	680	680	680	0.74
u159	42080	54038	42080	42080	42080	0.00	54038	42080	42080	42080	0.00
ulysses16	6859	8011	6859	6859	6859	0.00	8011	6859	6859	6859	0.00
ulysses22	7013	8506	7013	7013	7013	0.00	8506	7013	7013	7013	0.00

Table 5.3: Results of the Tabu search and VNS metaheuristics

5.4 Combined Approach: VNS and Hard Fixing

In this section we present the last experiment we evaluated. In particular, we used a combined approach of the VNS algorithm and the hard fixing technique. We started from an initial solution obtained from the 2-approximation algorithm, then given a global time limit TL we executed the VNS algorithm with time limit $2/3 * TL$ and the hard fixing technique on the output of the VNS with time limit $1/3 * TL$. The idea is that with the VNS we obtain high quality results, then using the hard fixing technique, which employs an exact solver we may refine such solutions.

The results are reported in Table 5.4, where we see the instances, their optimum value, the results after the VNS and after the hard-fixing step on the last column the final gap between the solution returned and the true optimum is reported. Note that 30 instances over 45 are solved to the optimum, moreover the maximum gap is within 1.88% on the pr439 instance which is a hard instance, as most of the exact solvers cannot solve it within one hour. Based on the above facts and on the results in the table, we can see that such technique it is very useful in practice even with large instances for which exact methods are impractical. We also observe that the global time limit TL was set to 45 minutes, giving more time to such procedure may lead to better solutions, thus we are very confident on such approach.

Instance	**Optimum value**	**Objective function (after)**		**Gap (%)**
		VNS	Hard-Fixing	
a280	2579	2605	2579	0.00
att48	10628	10684	10628	0.00
berlin52	7542	7775	7542	0.00
bier127	118282	119683	119683	1.18
ch130	6110	6155	6155	0.74
ch150	6528	6566	6560	0.49
d198	15780	15783	15783	0.02
eil101	629	640	629	0.00
eil51	426	430	426	0.00
eil76	538	538	538	0.00
gil262	2378	2380	2380	0.08
gr137	69853	69853	69853	0.00
gr202	40160	40418	40418	0.64
gr229	134602	135365	135365	0.57
gr96	55209	55209	55209	0.00
kroA100	21282	21282	21282	0.00
kroA150	26524	26707	26524	0.00
kroA200	29368	29383	29368	0.00
kroB100	22141	22141	22141	0.00
kroB150	26130	26132	26132	0.01

Continued on next page

Instance	**Optimum value**	**Objective function (after)**		**Gap (%)**
		VNS	Hard-Fixing	
kroB200	29437	29624	29624	0.64
kroC100	20749	20749	20749	0.00
kroD100	21294	21374	21294	0.00
kroE100	22068	22140	22068	0.00
lin105	14379	14379	14379	0.00
lin318	42029	42163	42163	0.32
pcb442	50778	51110	51110	0.65
pr107	44303	44303	44303	0.00
pr124	59030	59030	59030	0.00
pr136	96772	98800	98567	1.85
pr144	58537	58537	58537	0.00
pr152	73682	73818	73682	0.00
pr226	80369	80369	80369	0.00
pr264	49135	49135	49135	0.00
pr299	48191	48316	48191	0.00
pr439	107217	109236	109236	1.88
pr76	108159	108274	108159	0.00
rat195	2323	2330	2323	0.00
rat99	1211	1211	1211	0.00
rd100	7910	7910	7910	0.00
rd400	15281	15396	15396	0.75
st70	675	680	675	0.00
u159	42080	42080	42080	0.00
ulysses16	6859	6859	6859	0.00
ulysses22	7013	7013	7013	0.00

Table 5.4: Results of the composition of the VNS and the Hard-Fixing

Chapter 6

Conclusions

During our work we had the opportunity to study different advanced techniques for addressing the TSP. Such problem still fascinates us, since it has an elegant and simple formulation but it hides a lot of mystery and difficulties. We first evaluated exact methods and verified that they are not sufficient to address large instances. Then we evaluated several heuristic techniques; such algorithms are difficult to be designed and no "perfect" algorithm there exists, i.e., no algorithm always performed better than all the other in all the instances, as we showed. Developing and designing such techniques may be defined as *art*, but art is also the result that it may be achieved applying such techniques, Figure 6.1 represents the Mona Lisa, obtained solving an instance with 100K nodes[1].

This Mona Lisa plot was obtained starting from the 2-approximated solution and applying the 2-Opt algorithm once, thus finding a local optimum; the overall time of these two steps was about a hour. This can be a good example of the issues which one faces when trying to solve NP-Hard problems: for instances with hundreds of nodes the same procedure can be applied thousand of times in few seconds (our implementation of the VNS and Tabu search uses the 2-Opt to find better solutions), but instances with thousands of nodes can take hours to apply the same procedure only once.

We conclude observing that, during the months we developed all the techniques reported in this work, we spent entire nights debugging implementing and testing all the techniques but looking back at all we have done we are very happy and proud about our work.

[1] Art tsp are available at `http://www.math.uwaterloo.ca/tsp/data/art/`

Figure 6.1: Solution obtained through the 2-Opt algorithm at the Mona Lisa 100K instance; the plot was obtained using a Python script as explained in the Appendix B

Appendix A

Setup of Cplex in Ubuntu 18.04

In this appendix we present all the information needed in order to configure IBM ILOG CPLEX on your Linux like machine, specifically these steps refer to an Ubuntu distribution (version 18.04).

A.1 Downloading CPLEX

CPLEX is one of the most used solvers for mathematical optimization since it is very efficient and it was the first solver in commerce. Usually if you have to use CPLEX for business you have to contact IBM and to agree for a license fee, but for academic purposes CPLEX is available for free. After checking that your organization (i.e., University) has the permission to use the software you can go to the CPLEX website[1] after registering an signing on with your institutional mail you are allowed to download the software.

There are different versions available: check your operating system and choose consequently, if you are running Linux you have to download a version named similarly to *cplex_studioXXX.linux-x86-64.bin*[2] where *XXX* corresponds to the version, also you may find useful to download the little guide available among the downloads.

A.2 Installing CPLEX

Once the download is completed, open a terminal in that folder and type: `ls -al`, this will display all the files in your folder with the respective permission, be sure that the downloaded file is executable, if not you have to type: `chmod +x <cplex_studio>`, this should change the permissions such that the file becomes executable as shown in figure A.1, you can check it by again executing `ls -al`.

[1] `https://ibm.onthehub.com/WebStore/OfferingDetails.aspx?o=733c3d21-0ce1-e711-80fa-000d3af41938` if you are a student.

[2] From know this will be called just `<cplex_studio>` for brevity.

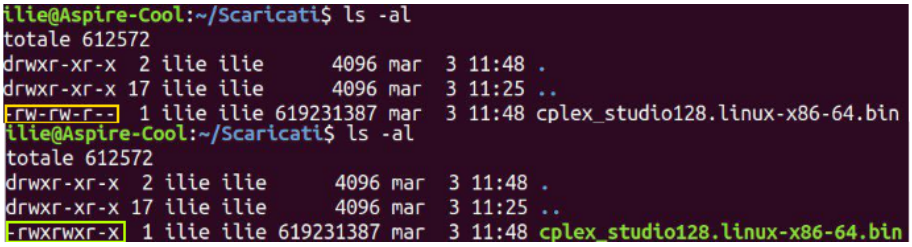


Figure A.1: Before and after the chmod is applied, you can see differences from the yellow rectangle to the green one.

Once you performed these preliminary steps you can install CPLEX by executing `./<cplex_studio>`; during the installation it is required ∼1.8GB of hard disk space. Observe that by default CPLEX asks to be installed in the directory `/opt/ibm/ILOG/CPLEX_<version>`[3] where in our case the `<version>` is `Studio128`. Note that if you want to install CPLEX at that location but you don't have the permission to create that folders (if not already existing) or you can't access that folder, you have to specify a different path during the installation; to maintain the original path you can execute the installation just using `sudo ./<cplex_studio>`, this allows CPLEX to build that path.
After the installation you can check the documentation that explains the content of the folders already installed at `basepath/cplex/readmeUNIX.html` and also test some examples at `basepath/cplex/examples/<machine>/<libformat>` to verify that CPLEX works, noting that in order to execute those examples you need the program *make*[4].
It is possible that if you have installed CPLEX correctly and try to execute the command `cplex` in a terminal you receive a message like "*cplex: command not found*" in that case you can setup CPLEX to be executed from command line just by executing `sudo ln -s basepath/cplex/bin/<architecture>/cplex /usr/bin/cplex` where `<architecture>` if you are using Linux is `x86-64_linux` this should setup a link to the cplex command available from every location.

A.3 Using CPLEX API for C programming

A.3.1 CPLEX Libraries

Once accomplished the previous section you should be able to execute the command `cplex` and solve all the problems given in a certain format file. One may be interested in implementing CPLEX in the source code in order to solve directly from a custom program his optimization problem of interest. Since CPLEX is a proprietary software its source code is not available but we can include the header files (`.h`) and the libraries (`.a`). In order to do this, one can setup a **Makefile** that will help the programmer through the compilation of the source code to link the libraries and the header files need

[3] Let this be `basepath` as a shorthand.
[4] If you are superuser and don't have *make* you can install it by executing `sudo apt install make`

by the compiler in order to be able to use CPLEX correctly. We present the Makefile used during the project that contains all the basic steps needed to link correctly CPLEX to an application in C.

```
# ------------------------
#       Compiler
# ------------------------
CC = gcc

# ------------------------------------------------------------
#       Paths, target, list of all .c files in the directory
#       and list of all the objects we need to create
# ------------------------------------------------------------
TARGET  = main

FILES   = $(wildcard *.c)                # this function lists all .c files
OBJECTS = $(patsubst %.c, %.o, $(FILES)) # substitute file.c -> file.o

CPLEX_LOC = /opt/ibm/ILOG/CPLEX_Studio128/cplex/include/ilcplex
LIB_LOC   = /opt/ibm/ILOG/CPLEX_Studio128/cplex/lib/x86-64_linux/static_pic

# ------------------------
#       Flags and libraries
# ------------------------

LIBS = -L $(LIB_LOC) -lcplex -lm -lpthread -ldl
CFLAGS = -I $(CPLEX_LOC)
RM = rm -f

# ------------------------
#       Targets and rules
# ------------------------

all: $(TARGET)

$(TARGET):$(OBJECTS)
        $(CC) $(CFLAGS) -o $(TARGET) $(OBJECTS) $(LIBS)

%.o: %.c
        $(CC) $(CFLAGS) -c $< -o $@  $(LIBS)
# ---------------------------------------
#       Set the cleanup
# ---------------------------------------
clean:
        $(RM) *.o
        $(RM) $(TARGET)
```

Listing A.1: Makefile for including correctly CPLEX in a C application.

The makefile presented allows the user to specify a target by executing `make <target>` this will generate all the objects `.o` from the `.c` files in the folder and then uses the files already generated to create the executable target. Notice that if you are implementing CPLEX in your source code then you have to specify to the compiler where to find CPLEX libraries; this is accomplished by setting up the right path in `CPLEX_LOC` and `LIB_LOC`, if default paths are used then this Makefile does not need modifications otherwise one should specify his own paths.

A.3.2 CPLEX from C source code

The initial steps necessary to use CPLEX from the C souce code are reported in code A.2, assuming the "`#include <cplex.h>`" is already done. In order for cplex to work correctly there it is the need to initalize correctly the `env` and the `lp`, these steps are reported in the code.

```
int TSPopt(instance *inst)
{
        CPXENVptr env = CPXopenCPLEX(&error);
        // used to check if there are errors while reading data
        CPXsetintparam(env, CPXPARAM_Read_DataCheck, 1);
        CPXsetintparam(env, CPXPARAM_RandomSeed, inst->random_seed);

        CPXLPptr lp = CPXcreateprob(env, &error, "TSP_Model");
        if(VERBOSE > 50)
        {
                // it saves the log of the computation in exec_compact_log.txt
                CPXsetlogfilename(env, "exec_log.txt", "w");
        }

        // build model
        build_model(inst, env, lp);

        // save model
        CPXwriteprob(env, lp, "tsp_model.lp", NULL);
}
```

Listing A.2: C code for performing the basic steps of usage of CPLEX in a C application.

Once the `env` and the `lp` are available they can be used to add constraints to the model, specifying the objective function and at the end for solving the instances. We now present the list of instructions that we find useful to perform such tasks with a brief description[5]:

- `CPXnewcols(...)`: it is used to add variables to the model, specifying the upper and lower bound on the variables and thir type (integer, binary, etc.).
- `CPXnewrows(...)`: it is used to add the constraints of the model, specifying the variables involved, the type of constraint $\{\leq, \geq, =\}$ and the value of the right-hand side.
- `CPXgetnumrows(env, lp)`: gets the current number of constraints.
- `CPXgetnumcols(env, lp)`: gets the number of variables in the current model.
- `CPXaddlazyconstraints(...)`: adds the constrait as a lazy constraint, thus the constraint is not loaded at the beginning of the resolution procedure.
- `CPXmipopt(env, lp)`: used to solve the problem once the model is specified.
- `CPXgetx(...)`: gets the current best integer solution available after the problem was solved with or without a time limit. The solution it will be specified as the

[5]Further documentation can be found at `https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.cplex.help/refcallablelibrary/homepageCrefman.html`

sequence of variables that have value in $\{0, 1\}$ after the resolution (1 if the edge is part of the solution, 0 otherwise), thus a sequence of edges is returned.

- `CPXgetobjval(...)`: gets the objective function's value after the step of resolution.
- `CPXsetintparam(env, CPXPARAM_RandomSeed, random_seed)`: used to specify a random seed for the resolution.
- `CPXaddmipstarts(...)`: used to specify an intial solution to the solver.
- `CPXsetdblparam(env, CPXPARAM_TimeLimit, time_limit)`: used to specify a non deterministic time limit to the solver, it refers to the time in seconds.
- `CPXsetdblparam(env, CPXPARAM_DetTimeLimit, time_limit)`: used to specify a deterministic time limit to the solver, it refers to the time in ticks.
- `CPXgetstat(env, lp)`: used to get the status code at the end of the resolution, there are different codes that express different cases, for example when solution are infeasible, optimal or optimal within a tolerance.
- `CPXgetmiprelgap(...)`: used to obtain the gap between the current solution of the problem and the best lower bound available.
- `CPXsetintparam(env, CPXPARAM_MIP_Strategy_RINSHeur, rins_nodes)`: used to control the number of nodes in which RINS is applied through the parameter `rins_nodes`.
- `CPXgetdettime(...)`: used to obtain the deterministic time in ticks elapsed, from the beginning of the resolution.

A.4 Callbacks

Here we presents some general aspects of the usage of the callbacks from the CPLEX APIs, in particular we refer to the `lazycallbackfunction` and the `usercutcallbackfunction`. There exists other callback functions, the documentation is available at `https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.cplex.help/CPLEX/UsrMan/topics/progr_adv/callbacks_basic/01_cb_title_synopsis.html`.

A.4.1 Lazycallback

The basic code for using a callback is reported in code A.3, after initializing the `env`, the `lp` and adding the variables along the objective function to the model, there it is the need to setup the callback. To use the callback there it is the need to disable a feature called *reduced model* from CPLEX, then the lazy callback is instantiated. For the efficiency of the program, the cores are setted to the maximum number available, since multithreading is used the callback function must be *threadsafe*.

```
...
static int CPXPUBLIC mylazycallback(CPXCENVptr env, void *cbdata,
                 int wherefrom, void *cbhandle, int *useraction_p);

int TSPopt_sec_callback(instance *inst)
{
        ...

        CPXENVptr env = CPXopenCPLEX(&error);

        CPXLPptr lp = CPXcreateprob(env, &error, "TSP");
        build_model(inst, env, lp);

        // let MIP callbacks work on the original model
        CPXsetintparam(env, CPX_PARAM_MIPCBREDLP, CPX_OFF);
        CPXsetlazyconstraintcallbackfunc(env, mylazycallback, inst);
        int ncores = 1;
        CPXgetnumcores(env, &ncores);
        // reset after callback
        CPXsetintparam(env, CPX_PARAM_THREADS, ncores);
        CPXsetintparam(env, CPXPARAM_RandomSeed, inst->random_seed);

        if(CPXmipopt(env, lp))
        {
                print_error("Optimisation failed in TSPopt_sec_loop()");
        }

        ...
}

static int CPXPUBLIC mylazycallback(CPXCENVptr env, void *cbdata,
                 int wherefrom, void *cbhandle, int *useraction_p)
{
        ...
}
```

Listing A.3: C for installing the lazycallback and the callback signature

We recall that the mylazycallback function must add constraints to integer solution which have multiple connected components as described in Chapter 3. Now we list some useful functions that can be used from the callback, and a brief description for each of them.

- `CPXgetcallbacknodex(...)`: gets the solution at the node where the callback was invoked, the solution is returned as a vector with a size equal to those of the overall variables, each variable has a value 1 if the edge is chosen in the solution, 0 otherwise.
- `CPXgetcallbacknodeobjval(...)`: gets the objective function at the current node.
- `CPXgetcallbackinfo(env, cbdata, wherefrom, CPX_CALLBACK_INFO_MY_THREAD_NUM, &mythread)`: gets the thread number which is handling the callback execution.
- `CPXgetcallbackinfo(env, cbdata, wherefrom, CPX_CALLBACK_INFO_BEST_INTEGER, &zbest)`: value of the best integer solution found up to that point.

- `CPXgetcallbacknodeinfo(env, cbdata, wherefrom, 0, CPX_CALLBACK_INFO_NODE_DEPTH, &depth)`: gets the depth of the current node in the decision tree.

A.4.2 UserCutCallback

In order to be able to implement the algorithm based on the UserCutCallback as described at the end of Chapter 3 it is necessary to download concorde[6], then after installing it as reported in the guide, a file `concorde.a` it will be created. From the folder where the file `concorde.a` is available, execute the following command: `cp concorde.a libconcorde.a`. Once this step is performed the makefile we presented can be modified as follows:

```
...
CONCLIBS = -L ${CONCORDELIB}/ -lconcorde
...


all: $(TARGET)

$(TARGET):$(OBJECTS)
	$(CC) $(CFLAGS) -o $(TARGET) $(OBJECTS) $(CONCLIBS) $(LIBS)

%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@ $(CONCLIBS) $(LIBS)

...
```

Listing A.4: Modified makefile to include the concorde library.

Note that `${CONCORDELIB}` is the path to the folder where the file `libconcorde.a` is placed. This will allow the user to use the concorde library through `#include <concorde.h>` from the C source code. The code for using installing and using the usercutcallback is presented in A.5, it is quite similar to the one of the lazycallback with the difference that the call used to instantiate such callback is different. Moreover the function cutcallback, checks if the fractional solution has one or more connected components and respectively the callbacks `doit_fn_concorde` or `separationMultiple` are called. To have all the function working correctly the following concorde functions are needed:

- `CCcut_connect_components (int ncount, int ecount, int *elist, double *x, int *ncomp, int **compscount, int **comps)`: this function verifies, given a fractional solution, how many connected componets it presents.
- `CCcut_violated_cuts (int ncount, int ecount, int *elist, double *dlen, double cutoff, int (*doit_fn) (double, int, int *, void *), void *pass_param)` : it computes the global minimum cut, but calls the doit_fn function for any cut the algorithm encounters that has capacity at most `cutoff`.

```
...
const double EPS = 0.1;
```

[6]Download available at `http://www.math.uwaterloo.ca/tsp/concorde/downloads/downloads.htm`

```
int TSPopt_usr_callback(instance *inst)
{
        ...
        CPXsetintparam(env, CPX_PARAM_MIPCBREDLP, CPX_OFF);
        CPXsetusercutcallbackfunc(env, cutcallback, inst);
        CPXsetlazyconstraintcallbackfunc(env, mylazycallback, inst);
        int ncores = 1;
        CPXgetnumcores(env, &ncores);
        CPXsetintparam(env, CPX_PARAM_THREADS, ncores);// reset after callback
        CPXsetintparam(env, CPXPARAM_RandomSeed, inst->random_seed);
        ...
}

static int CPXPUBLIC cutcallback(CPXCENVptr env, void *cbdata,
              int wherefrom, void *cbhandle, int *useraction_p)
{
        *useraction_p = CPX_CALLBACK_DEFAULT;
        instance* inst = (instance *) cbhandle; // casting of cbhandle

        ...

        if(CCcut_connect_components(inst->nnodes, edgecount, elist, xstar,
                                    &ncomp, &compscount, &comps))
        {
                printf("Error in getting components of solution usercutcallback\n");
                return 1; // y = current y from CPLEX-- y starts from position 0
        }
        if(ncomp > 1)
        {
                // Add constraints on connected components
                int ncuts = separationMultiple(inst, ncomp, compscount,
                                               comps, env, cbdata, wherefrom);
                ...
        }

        if(ncomp == 1)
        {
                cc_instance instCC = {.env = env, .cbdata = cbdata, .wherefrom
                      = wherefrom, .useraction_p = useraction_p, .inst = inst };
                // Separate fractionary solution
                if(CCcut_violated_cuts(inst->nnodes, edgecount, elist, xstar, 2.0
                                       - EPS, doit_fn_concorde, (void*) &instCC))
                {
                        ...
                }
                ...
        }
        ...
}
int doit_fn_concorde(double cutval, int cutcount, int *cut, void *inParam)
{
        ...
}
int separationMultiple(instance *inst, int ncomp, int *compscount,
           int *comps, CPXCENVptr env, void *cbdata, int wherefrom)
{
        ...
}

static int CPXPUBLIC mylazycallback(CPXCENVptr env, void *cbdata,
                 int wherefrom, void *cbhandle, int *useraction_p)
{
```

```
        ...
}
```

Listing A.5: C code for installing the lazycallback along the usercutcallback and the callback signatures

A.5 Saving the solution

For saving the solution on a file we used the following code:

```
void print_plot_subtour(instance *inst, char *plot_file_name)
{
    int i, j, k, l, flag;
    int cur_numcols = xpos(inst->nnodes-2, inst->nnodes-1, inst)+1;
    FILE *file = fopen(plot_file_name, "w");
    fprintf(file, "%d\n", inst->nnodes);
    for(i=0; i<inst->nnodes; i++)
    {
        fprintf(file, "%lf %lf\n", inst->xcoord[i], inst->ycoord[i]);
    }

    fprintf(file, "\nNON ZERO VARIABLES\n");

    for(k=0; k<cur_numcols; k++)
    {
        if(inst->best_sol[k] > TOLERANCE)
        {
            l = inst->nnodes -1;
            flag = 0;
            for(i=0; (i<inst->nnodes-1) && (!(flag)); i++)
            {
                if(k<l)
                {
                    for(j=i+1; j<inst->nnodes; j++)
                    {
                        if(xpos(i, j, inst) == k)
                        {
                            fprintf(file, "x_%d_%d = %f\n", i+1, j+1,
                                    inst->best_sol[k]);
                            flag = 1;
                            break;
                        }
                    }
                }
                else
                {
                    l += inst->nnodes-i-2;
                }
            }
        }
    }
    fclose(file);
}
```

Listing A.6: C code for plotting the solution, assuming the coordinates of node v_i are saved respectively in `(inst->xcoord[i],inst->ycoord[i])`, with $i = 1, \ldots, |V|$ and the optimal solution saved in `inst->bestsol` which is an array of length $|E|$ of {0,1} elements, 0 if the variable is not chose in the solution, 1 otherwise.

A sample of the output is now reported.

```
16
38.240000 20.420000
39.570000 26.150000
40.560000 25.320000
36.260000 23.120000
33.480000 10.540000
37.560000 12.190000
38.420000 13.110000
37.520000 20.440000
41.230000 9.100000
41.170000 13.050000
36.080000 -5.210000
38.470000 15.130000
38.150000 15.350000
37.510000 15.170000
35.490000 14.320000
39.360000 19.560000

NON ZERO VARIABLES
x_1_8 = 1.000000
x_1_14 = 1.000000
x_2_3 = 1.000000
x_2_4 = 1.000000
x_3_16 = 1.000000
x_4_8 = 1.000000
x_5_11 = 1.000000
x_5_15 = 1.000000
x_6_7 = 1.000000
x_6_15 = 1.000000
x_7_12 = 1.000000
x_9_10 = 1.000000
x_9_11 = 1.000000
x_10_16 = 1.000000
x_12_13 = 1.000000
x_13_14 = 1.000000
```

Listing A.7: Sample of output for storing the solution on file. In particular on the first line we have $n = |V|$, then for n lines there are the coordinates x_i and y_i of each node $i = 1, \ldots, n$; after the line `NON ZERO VARIABLES` we have the n variables which correspond to the edges in the format $x\_i\_j = 1.0$, where $i, j \in V, i \neq j$ thus representing the edge $\{i, j\} \in E$.

Appendix B

Plotting the solution

Once solved the TSP problem it is very useful to plot the graph and the optimal solution found, we used a very useful package for Python named `toyploy`[1] which can be installed just by using `pip install toyplot` assuming that `pip` is already installed. Assuming that the output of the main program is saved according the previous section we can use the following script in Python to plot the solution.

```
import sys
import numpy as np
import matplotlib.pyplot as plt
import toyplot # pip install toyploy
import math
import toyplot.png
import toyplot.pdf


def plot_graph(x,y,n, edges_plot):
    coordinates=np.transpose(np.vstack((x,y)))
    layout = toyplot.layout.FruchtermanReingold()
    vstyle = {"stroke":toyplot.color.black}
    vlstyle = {"fill":"white"}
    colormap = toyplot.color.LinearMap(toyplot.color.Palette(["white"]))
    canvas, axes, mark = toyplot.graph(edges_plot,vcoordinates=coordinates,
                                       layout=layout,vcolor=colormap,
                                       vsize=18, vstyle=vstyle, width=1000)
    axes.show = False
    toyplot.pdf.render(canvas, "nodes.pdf")

def init_edges(f, n, edges):
        line = f.readline()
        while (line.strip() != "NON ZERO VARIABLES"):
            line = f.readline()
        for i in range(n):
            line = f.readline()
            chuncks = line.split(" ")
            coords = chuncks[0].split("_")
            edges[i] = [int(str(coords[1])), int(str(coords[2]))]
        edges = edges.astype(int)

def init_nodes(f, n, x, y):
    for i in range(n):
```

[1] Documentation available at `https://toyplot.readthedocs.io/en/stable/`

```
        line = f.readline()
        x[i] = float(line.split(" ")[0])
        y[i] = float(line.split(" ")[1])

def main(toplot):
    f = open(toplot,"r")
    nnodes = int(f.readline())
    xcoord = np.zeros(nnodes)
    ycoord = np.zeros(nnodes)
    edges = np.empty(shape=(nnodes,2), dtype='i4')
    init_nodes(f,nnodes, xcoord, ycoord)
    init_edges(f,nnodes, edges)
    plot_graph(xcoord, ycoord, nnodes, edges)


if __name__ == '__main__':
        main(sys.argv[1])
```

Listing B.1: Python code for plotting the optimal solution.

In order to run the code one should execute `python <solution.txt>` where solution is a file structured as we already presented. The script B.1 receives a file in input then builds the node matrix since each $v \in V$ has two coordinates we save the vectors $\mathbf{x}$ and $\mathbf{y}$ of size $|V|$ each whose coordinates represent respectively the x and the y coordinate for each point. Then the script parses the optimal solution in a matrix **edges** of size $|E| \times 2$. After the parsing is done the script invokes the method `plot_graph` which draws and saves the solution in a file called "node.pdf". An example of the result is presented in figure B.1.

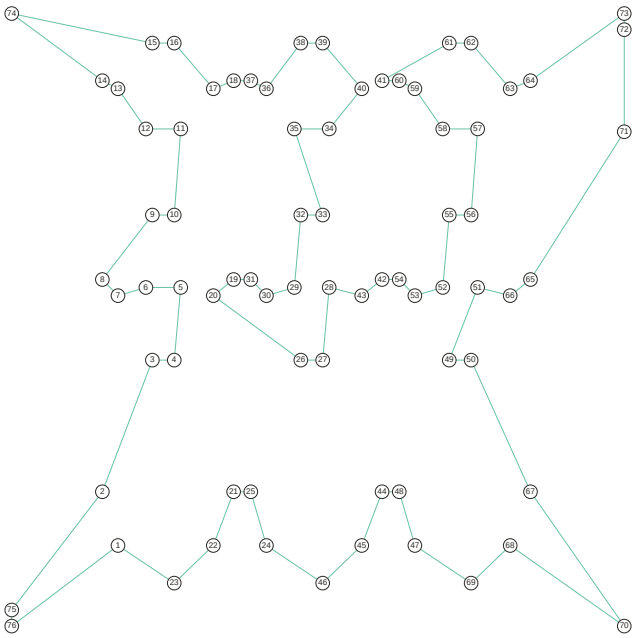


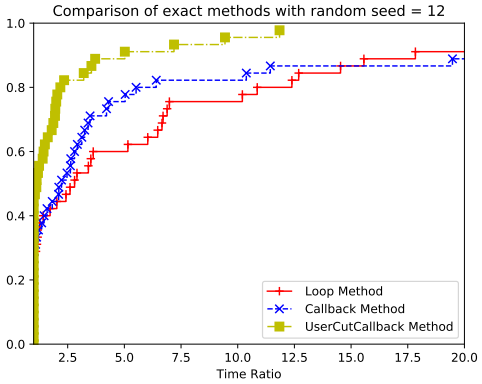
Figure B.1: The result of the Python script to plot the solution

Appendix C

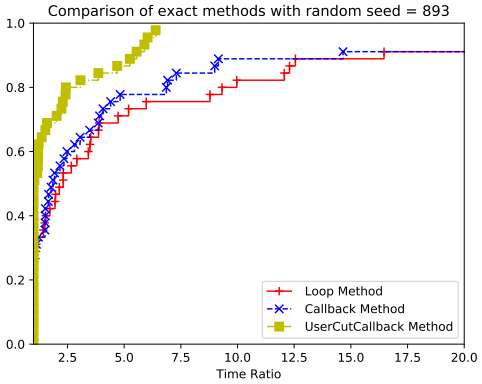
Additional Material

C.1 Non-deterministic results

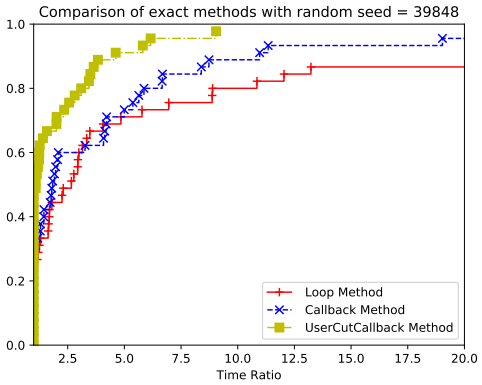
In this appendix, we report the results of the exact methods with a non deterministic time limit and comparison. The average times and the geometric ones have non negligible differences in most of the runs, implying that different CPUs were used to perform the runs, thus it is less meaningful to compare the deterministic time. Moreover some runs exceeded the timeout of 3600 seconds, since some time is needed to recognize that the time limit is expired. Despite all the drawbacks of measuring the deterministic time, in figure C.1 we can see that most of the observations we made for the deterministic time still hold, in particular the usercutcallback method still outperforms all the other methods. The callback method is still preferable to the loop method in all the seeds, except with the seed 201709013 where both techniques achieve approximately the same results.



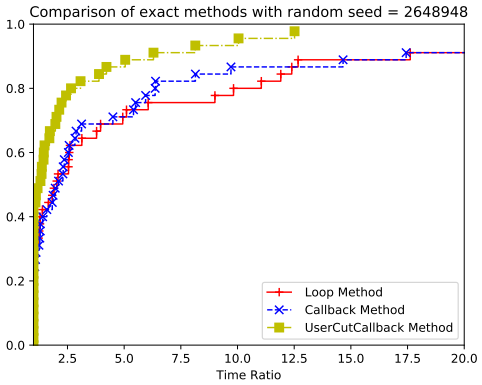
(a) Random seed = 12



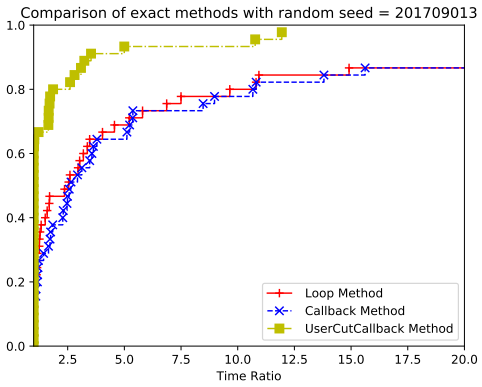
(b) Random seed = 893



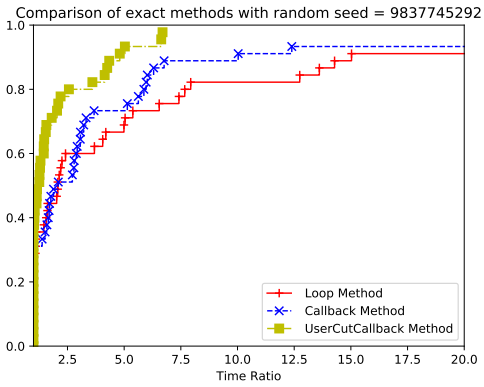
(c) Random seed = 39848



(d) Random seed = 2648948



(e) Random seed = 201709013



(f) Random seed = 9837745292

Figure C.1: Performance profile of the exact methods with non deterministic time limit set to 3600 seconds.

C.2 Deterministic results

In this Appendix we report the tables which refer to the comparison of methods *Loop improved, Callback* and *UserCutCallback* with deterministic time limit set to 2127600 ticks. Each table we are going to present has an average time obtained and a geometric mean obtained as described in Chapter 5, moreover we report a column "optimum" which states if the solution obtained was certified to be the true optimum within the given time limit in at least one run of the three. For the sake of clarity we decided to show the results grouped by random seed and showing three random seed in each table. Tables C.1 and C.2 summarize the results on the different seeds obtained by the improved Loop method. We observe that all the instances, except for two, the pr299 and the pr439 in table C.2, were solved to the optimum in the given amount of deterministic time. Tables C.3 and C.4 report the results on the different random seeds on the Callback method. We observe that some runs were unable to finish due to some issues with the blade cluster, thus we consider them as the instances that exceeded the time limit. In Tables C.5 and C.6 we reported the results of the UserCutCallback method. Across all the methods, we highlight that the geometric mean and the arithmetic mean are quite similar, except for some rare cases, this happens when all the runs have approximately the same deterministic time which denotes that the runs were executed in a stable environment.

Deterministic Time of Loop Method (ticks)									
Instance	**Random Seed**								
	12			893			39848		
	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.
a280	16883.94	16883.93	yes	10947.64	10788.33	yes	9951.67	9951.37	yes
att48	154.24	154.24	yes	154.48	154.48	yes	151.13	151.13	yes
berlin52	19.29	19.29	yes	19.50	19.50	yes	19.29	19.29	yes
bier127	951.51	951.51	yes	918.53	918.53	yes	949.36	949.36	yes
ch130	1114.21	1114.21	yes	1050.52	1050.52	yes	1106.43	1106.43	yes
ch150	2795.25	2794.37	yes	2743.99	2743.91	yes	2786.19	2786.17	yes
d198	15229.24	15228.87	yes	13177.83	13177.82	yes	16039.15	16039.15	yes
eil101	449.87	449.87	yes	455.20	455.20	yes	448.89	448.88	yes
eil51	115.83	115.83	yes	127.51	127.51	yes	127.10	127.10	yes
eil76	114.24	114.24	yes	114.56	114.56	yes	111.40	111.40	yes
gil262	18180.59	18163.35	yes	17240.48	17039.22	yes	17169.28	16968.41	yes
gr137	2190.43	2190.43	yes	2175.54	2175.53	yes	2183.20	2183.20	yes
gr202	8608.97	8608.97	yes	8632.54	8632.54	yes	8557.33	8557.33	yes
gr229	721713.67	90837.39	yes	17990.74	17975.83	yes	721936.37	91915.34	yes
gr96	1239.57	1239.57	yes	1278.35	1278.35	yes	1294.39	1294.37	yes
kroA100	942.31	942.31	yes	1014.20	1014.20	yes	998.16	998.16	yes
kroA150	4968.64	4968.64	yes	4862.90	4862.30	yes	4802.86	4802.85	yes
kroA200	12008.58	12008.58	yes	12438.91	12431.63	yes	13621.20	13613.65	yes
kroB100	1379.22	1379.22	yes	1470.62	1470.62	yes	1465.12	1465.12	yes
kroB150	6526.36	6526.36	yes	6775.21	6775.21	yes	6698.59	6698.59	yes
kroB200	4860.52	4860.52	yes	4823.34	4822.87	yes	4796.02	4795.11	yes
kroC100	1128.13	1128.13	yes	1106.50	1106.50	yes	1100.16	1100.16	yes
kroD100	1068.61	1068.61	yes	1074.46	1074.46	yes	1032.43	1032.43	yes
kroE100	948.13	948.13	yes	953.76	953.76	yes	973.48	973.48	yes

Continued on next page

Deterministic Time of Loop Method (ticks)									
Instance	**Random Seed**								
	12			893			39848		
	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.
lin105	825.26	825.26	yes	794.68	794.68	yes	895.74	895.74	yes
lin318	25832.92	25832.81	yes	29947.31	29947.30	yes	26708.94	26708.94	yes
pcb442	211336.11	211049.09	yes	187494.62	187489.14	yes	184782.29	184664.99	yes
pr107	229.40	229.40	yes	230.64	230.64	yes	232.66	232.66	yes
pr124	3770.23	3770.23	yes	3795.24	3795.24	yes	3891.71	3891.71	yes
pr136	1295.85	1295.85	yes	1396.05	1396.05	yes	1554.39	1554.39	yes
pr144	5331.48	5331.48	yes	5672.50	5672.50	yes	5478.36	5478.36	yes
pr152	3109.40	3109.40	yes	2982.48	2982.48	yes	3051.59	3051.59	yes
pr226	36375.91	36375.91	yes	26655.24	26655.23	yes	26411.49	26350.09	yes
pr264	55659.85	55641.72	yes	60185.59	60181.85	yes	56686.21	56686.21	yes
pr299	56581.90	56560.39	yes	58652.14	58652.14	yes	50164.54	50164.53	yes
pr439	254580.92	254476.96	yes	208531.09	207658.81	yes	225983.84	225983.84	yes
pr76	1463.76	1463.76	yes	1394.13	1394.13	yes	1392.91	1392.91	yes
rat195	11499.07	11499.07	yes	11878.62	11878.62	yes	13989.06	13989.06	yes
rat99	537.41	537.41	yes	565.47	565.47	yes	535.24	535.24	yes
rd100	828.94	828.94	yes	833.94	833.94	yes	830.16	830.16	yes
rd400	61771.53	61771.53	yes	73630.85	72894.69	yes	76726.25	76680.98	yes
st70	233.58	233.58	yes	229.18	229.18	yes	229.83	229.83	yes
u159	1662.93	1662.93	yes	1531.58	1531.58	yes	1541.39	1541.39	yes
ulysses16	3.15	3.15	yes	3.10	3.10	yes	3.32	3.32	yes
ulysses22	8.38	8.37	yes	8.38	8.38	yes	8.39	8.38	yes

Table C.1: Results grouped by the first three random seed of the Loop Method

Deterministic Time of Loop Method (ticks)									
Instance	**Random Seed**								
	2648948			201709013			9837745292		
	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.
a280	6645.57	6645.57	yes	6626.12	6626.12	yes	5284.22	5204.15	yes
att48	62.95	62.93	yes	61.31	61.29	yes	61.55	61.53	yes
berlin52	10.70	10.70	yes	10.70	10.70	yes	10.70	10.70	yes
bier127	872.76	872.76	yes	709711.98	10785.99	yes	884.86	884.37	yes
ch130	1085.10	1083.51	yes	1122.23	1122.05	yes	1479.99	1478.15	yes
ch150	2442.47	2418.51	yes	1761.91	1742.22	yes	1627.14	1627.14	yes
d198	2915.87	2856.03	yes	7946.80	7946.48	yes	6098.27	6098.10	yes
eil101	646.26	646.26	yes	337.58	337.41	yes	205.59	205.59	yes
eil51	97.45	97.45	yes	88.75	88.75	yes	123.63	123.37	yes
eil76	98.13	98.13	yes	31.18	31.17	yes	41.80	41.80	yes
gil262	13500.24	13458.72	yes	7703.26	7703.26	yes	14629.08	14628.15	yes
gr137	1575.84	1568.16	yes	977.87	964.48	yes	1293.16	1290.51	yes
gr202	3140.02	3132.37	yes	3280.24	3228.92	yes	3140.65	3051.51	yes
gr229	8783.74	8782.80	yes	7669.92	7669.92	yes	8041.52	8039.76	yes
gr96	647.51	644.84	yes	653.98	653.19	yes	628.77	628.31	yes
kroA100	731.28	731.28	yes	870.48	867.24	yes	575.64	575.64	yes
kroA150	1783.36	1783.36	yes	1359.38	1359.38	yes	1758.89	1758.89	yes
kroA200	9281.48	9279.09	yes	7698.69	7689.39	yes	9909.49	9643.83	yes
kroB100	557.77	543.96	yes	688.53	666.18	yes	818.52	818.52	yes
kroB150	2586.93	2531.86	yes	2691.50	2670.36	yes	2095.01	2043.03	yes
kroB200	2627.85	2613.76	yes	2817.54	2817.54	yes	3177.34	3151.65	yes
kroC100	469.10	469.10	yes	729.44	729.44	yes	646.90	646.90	yes
kroD100	666.97	666.97	yes	545.03	545.03	yes	240.08	240.08	yes
kroE100	871.65	871.65	yes	705.92	696.22	yes	910.54	905.24	yes

Continued on next page

Deterministic Time of Loop Method (ticks)									
Instance	**Random Seed**								
	2648948			201709013			9837745292		
	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.
lin105	423.22	421.16	yes	373.92	372.85	yes	368.10	368.10	yes
lin318	10534.31	10534.30	yes	14052.29	14052.29	yes	11835.96	11819.67	yes
pcb442	17079.68	16692.81	yes	24626.18	23848.89	yes	21200.19	19659.78	yes
pr107	21.93	21.93	yes	21.93	21.93	yes	21.93	21.93	yes
pr124	1359.10	1351.77	yes	723.45	716.40	yes	952.76	952.76	yes
pr136	918.11	918.11	yes	855.76	855.76	yes	969.50	969.50	yes
pr144	2982.04	2966.22	yes	1822.67	1822.67	yes	2233.80	2231.21	yes
pr152	1616.12	1613.98	yes	1720.65	1720.65	yes	1582.98	1582.98	yes
pr226	6207.99	6207.99	yes	4964.05	4964.05	yes	5162.22	5158.74	yes
pr264	6596.65	6595.38	yes	5857.48	5840.57	yes	6854.13	6854.13	yes
pr299	1475118.32	916659.80	yes	1418846.32	1340866.28	yes	2127600.00	2127600.00	no
pr439	2127600.00	2127600.00	no	1805033.63	1738068.73	yes	1730328.71	1618026.10	yes
pr76	1496.47	1485.58	yes	1031.86	1026.26	yes	1127.90	1127.57	yes
rat195	3739.46	3739.46	yes	5913.86	5829.67	yes	5422.98	5321.63	yes
rat99	347.35	346.82	yes	350.26	349.54	yes	332.56	331.94	yes
rd100	500.83	494.67	yes	463.53	462.62	yes	415.33	415.33	yes
rd400	135024.01	128299.67	yes	1108112.73	913291.53	yes	96465.93	94112.86	yes
st70	210.29	210.13	yes	207.47	207.47	yes	142.67	142.67	yes
u159	1292.76	1291.46	yes	1273.25	1272.54	yes	1724.48	1724.48	yes
ulysses16	0.60	0.60	yes	1.47	1.47	yes	1.47	1.47	yes
ulysses22	2.29	2.29	yes	2.29	2.29	yes	2.33	2.33	yes

Table C.2: Results grouped by the second three random seed of the Loop Method

Deterministic Time of Callback Method (ticks)									
Instance	**Random Seed**								
	12			893			39848		
	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.
a280	6054.89	5953.41	yes	4532.06	4488.03	yes	4604.98	4604.98	yes
att48	61.89	61.87	yes	61.37	61.35	yes	61.84	61.82	yes
berlin52	10.70	10.70	yes	10.91	10.91	yes	10.70	10.70	yes
bier127	838.67	838.67	yes	834.27	834.22	yes	881.18	881.17	yes
ch130	948.07	947.14	yes	972.80	971.23	yes	1344.62	1341.86	yes
ch150	1608.11	1595.61	yes	2536.75	2498.31	yes	1854.27	1806.97	yes
d198	4277.06	4275.78	yes	7409.49	7290.96	yes	3304.16	3304.16	yes
eil101	181.91	181.91	yes	434.07	434.07	yes	534.56	534.56	yes
eil51	81.62	81.62	yes	105.40	105.40	yes	118.27	118.27	yes
eil76	94.43	94.43	yes	32.68	32.68	yes	41.07	41.07	yes
gil262	11815.22	11815.22	yes	11559.06	11559.06	yes	12908.82	12908.82	yes
gr137	1357.09	1345.46	yes	712.34	712.34	yes	806.73	796.72	yes
gr202	3775.98	3753.03	yes	3135.44	3093.87	yes	2997.38	2997.38	yes
gr229	6491.04	6491.04	yes	6981.33	6751.08	yes	10197.45	10129.31	yes
gr96	460.61	457.69	yes	577.04	576.90	yes	631.63	630.38	yes
kroA100	707.61	706.39	yes	613.15	612.56	yes	901.17	896.33	yes
kroA150	1790.53	1752.78	yes	1235.54	1235.54	yes	1989.62	1963.06	yes
kroA200	9545.81	9544.97	yes	10800.38	10718.76	yes	8363.14	8363.14	yes
kroB100	518.25	518.25	yes	436.90	436.90	yes	720.28	717.95	yes
kroB150	2209.15	2193.80	yes	2054.85	2013.11	yes	2239.84	2216.03	yes
kroB200	2240.09	2240.09	yes	3433.60	3371.51	yes	2842.16	2842.16	yes
kroC100	1512.58	1512.58	yes	729.71	729.71	yes	834.92	834.88	yes
kroD100	233.98	233.98	yes	402.03	402.02	yes	307.31	307.31	yes
kroE100	591.19	575.43	yes	602.20	591.56	yes	969.54	967.21	yes

Continued on next page

Deterministic Time of Callback Method (ticks)									
Instance	**Random Seed**								
	12			893			39848		
	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.
lin105	612.16	612.16	yes	383.32	382.02	yes	600.79	600.79	yes
lin318	9879.69	9808.10	yes	13033.67	12956.89	yes	13250.69	13235.48	yes
pcb442	16678.59	16678.59	yes	19044.88	18434.58	yes	17789.58	17789.58	yes
pr107	21.93	21.93	yes	21.93	21.93	yes	21.93	21.93	yes
pr124	1121.67	1121.67	yes	816.35	816.35	yes	1010.83	1005.62	yes
pr136	918.03	918.03	yes	1220.10	1220.10	yes	1228.20	1228.20	yes
pr144	2856.45	2856.44	yes	1833.96	1833.96	yes	1803.18	1803.18	yes
pr152	2693.66	2693.66	yes	1685.67	1685.67	yes	1837.85	1823.80	yes
pr226	4833.02	4833.02	yes	7141.03	7101.57	yes	6210.98	6210.98	yes
pr264	5659.75	5659.75	yes	5422.34	5422.34	yes	8305.26	8305.26	yes
pr299	2127600.00	2127600.00	no	1801259.19	1732394.37	yes	2127600.00	2127600.00	no
pr439	1369208.78	1277587.47	yes	1224076.03	1082661.31	yes	2127600.00	2127600.00	no
pr76	1072.95	1072.95	yes	1040.76	1039.78	yes	989.32	988.74	yes
rat195	6066.95	5982.90	yes	5787.72	5684.80	yes	4005.66	3919.85	yes
rat99	334.10	333.55	yes	341.08	340.31	yes	331.30	330.71	yes
rd100	419.80	419.23	yes	415.98	415.57	yes	628.65	627.95	yes
rd400	75615.82	75510.80	yes	112271.26	104617.99	yes	102600.70	86370.30	yes
st70	177.30	177.08	yes	193.10	193.10	yes	142.47	142.47	yes
u159	1827.45	1827.38	yes	839.17	839.17	yes	1878.50	1878.49	yes
ulysses16	1.47	1.47	yes	1.49	1.49	yes	1.47	1.47	yes
ulysses22	2.29	2.29	yes	2.29	2.29	yes	2.29	2.29	yes

Table C.3: Results grouped by the first three random seed of the Callback Method

Deterministic Time of Callback Method (ticks)									
Instance	**Random Seed**								
	2648948			201709013			9837745292		
	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.
a280	6054.89	5953.41	yes	4532.06	4488.03	yes	4604.98	4604.98	yes
att48	61.89	61.87	yes	61.37	61.35	yes	61.84	61.82	yes
berlin52	10.70	10.70	yes	10.91	10.91	yes	10.70	10.70	yes
bier127	838.67	838.67	yes	834.27	834.22	yes	881.18	881.17	yes
ch130	948.07	947.14	yes	972.80	971.23	yes	1344.62	1341.86	yes
ch150	1608.11	1595.61	yes	2536.75	2498.31	yes	1854.27	1806.97	yes
d198	4277.06	4275.78	yes	7409.49	7290.96	yes	3304.16	3304.16	yes
eil101	181.91	181.91	yes	434.07	434.07	yes	534.56	534.56	yes
eil51	81.62	81.62	yes	105.40	105.40	yes	118.27	118.27	yes
eil76	94.43	94.43	yes	32.68	32.68	yes	41.07	41.07	yes
gil262	11815.22	11815.22	yes	11559.06	11559.06	yes	12908.82	12908.82	yes
gr137	1357.09	1345.46	yes	712.34	712.34	yes	806.73	796.72	yes
gr202	3775.98	3753.03	yes	3135.44	3093.87	yes	2997.38	2997.38	yes
gr229	6491.04	6491.04	yes	6981.33	6751.08	yes	10197.45	10129.31	yes
gr96	460.61	457.69	yes	577.04	576.90	yes	631.63	630.38	yes
kroA100	707.61	706.39	yes	613.15	612.56	yes	901.17	896.33	yes
kroA150	1790.53	1752.78	yes	1235.54	1235.54	yes	1989.62	1963.06	yes
kroA200	9545.81	9544.97	yes	10800.38	10718.76	yes	8363.14	8363.14	yes
kroB100	518.25	518.25	yes	436.90	436.90	yes	720.28	717.95	yes
kroB150	2209.15	2193.80	yes	2054.85	2013.11	yes	2239.84	2216.03	yes
kroB200	2240.09	2240.09	yes	3433.60	3371.51	yes	2842.16	2842.16	yes
kroC100	1512.58	1512.58	yes	729.71	729.71	yes	834.92	834.88	yes
kroD100	233.98	233.98	yes	402.03	402.02	yes	307.31	307.31	yes
kroE100	591.19	575.43	yes	602.20	591.56	yes	969.54	967.21	yes

Continued on next page

Deterministic Time of Callback Method (ticks)									
Instance	**Random Seed**								
	2648948			201709013			9837745292		
	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.
lin105	612.16	612.16	yes	383.32	382.02	yes	600.79	600.79	yes
lin318	9879.69	9808.10	yes	13033.67	12956.89	yes	13250.69	13235.48	yes
pcb442	16678.59	16678.59	yes	19044.88	18434.58	yes	17789.58	17789.58	yes
pr107	21.93	21.93	yes	21.93	21.93	yes	21.93	21.93	yes
pr124	1121.67	1121.67	yes	816.35	816.35	yes	1010.83	1005.62	yes
pr136	918.03	918.03	yes	1220.10	1220.10	yes	1228.20	1228.20	yes
pr144	2856.45	2856.44	yes	1833.96	1833.96	yes	1803.18	1803.18	yes
pr152	2693.66	2693.66	yes	1685.67	1685.67	yes	1837.85	1823.80	yes
pr226	4833.02	4833.02	yes	7141.03	7101.57	yes	6210.98	6210.98	yes
pr264	5659.75	5659.75	yes	5422.34	5422.34	yes	8305.26	8305.26	yes
pr299	2127600.00	2127600.00	no	1801259.19	1732394.37	yes	2127600.00	2127600.00	no
pr439	1369208.78	1277587.47	yes	1224076.03	1082661.31	yes	2127600.00	2127600.00	no
pr76	1072.95	1072.95	yes	1040.76	1039.78	yes	989.32	988.74	yes
rat195	6066.95	5982.90	yes	5787.72	5684.80	yes	4005.66	3919.85	yes
rat99	334.10	333.55	yes	341.08	340.31	yes	331.30	330.71	yes
rd100	419.80	419.23	yes	415.98	415.57	yes	628.65	627.95	yes
rd400	75615.82	75510.80	yes	112271.26	104617.99	yes	102600.70	86370.30	yes
st70	177.30	177.08	yes	193.10	193.10	yes	142.47	142.47	yes
u159	1827.45	1827.38	yes	839.17	839.17	yes	1878.50	1878.49	yes
ulysses16	1.47	1.47	yes	1.49	1.49	yes	1.47	1.47	yes
ulysses22	2.29	2.29	yes	2.29	2.29	yes	2.29	2.29	yes

Table C.4: Results grouped by the second three random seed of the Callback Method

Deterministic Time of UserCutCallback Method (ticks)									
Instance	**Random Seed**								
	12			893			39848		
	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.
a280	4401.49	4401.49	yes	5424.54	5424.31	yes	6122.05	6122.05	yes
att48	58.66	58.66	yes	60.18	60.18	yes	58.54	58.54	yes
berlin52	10.64	10.64	yes	10.85	10.85	yes	10.64	10.64	yes
bier127	590.23	590.23	yes	592.59	592.50	yes	588.76	588.76	yes
ch130	737.35	737.35	yes	759.92	759.92	yes	748.31	748.31	yes
ch150	1769.22	1769.22	yes	1593.81	1590.66	yes	1496.42	1496.42	yes
d198	2897.81	2897.81	yes	3562.04	3562.04	yes	3110.73	3110.73	yes
eil101	156.24	156.24	yes	155.54	155.54	yes	155.96	155.96	yes
eil51	55.92	55.92	yes	55.68	55.68	yes	55.88	55.88	yes
eil76	67.17	67.17	yes	33.13	33.13	yes	41.07	41.07	yes
gil262	7670.38	7652.66	yes	6896.34	6896.34	yes	8021.83	7925.30	yes
gr137	1259.61	1259.61	yes	1126.87	1126.87	yes	1182.43	1182.43	yes
gr202	2891.40	2891.40	yes	2702.36	2702.36	yes	2652.95	2652.95	yes
gr229	4559.47	4559.47	yes	6351.13	6351.13	yes	5070.92	5070.92	yes
gr96	640.88	640.88	yes	509.95	509.01	yes	553.81	553.37	yes
kroA100	555.17	555.17	yes	897.07	896.73	yes	810.80	810.80	yes
kroA150	1995.14	1995.14	yes	2362.27	2359.06	yes	1635.26	1635.26	yes
kroA200	6625.10	6558.17	yes	6457.26	6437.88	yes	6398.84	6396.96	yes
kroB100	679.73	679.71	yes	666.98	666.98	yes	594.71	594.70	yes
kroB150	1925.12	1925.12	yes	1678.38	1678.38	yes	1806.32	1806.32	yes
kroB200	2998.67	2998.67	yes	5180.55	5180.55	yes	711237.73	27088.70	yes
kroC100	592.04	592.04	yes	605.80	605.80	yes	572.22	572.22	yes
kroD100	728.64	728.64	yes	963.97	963.97	yes	556.65	556.64	yes
kroE100	875.79	875.79	yes	819.80	815.46	yes	735.12	735.12	yes

Continued on next page

Deterministic Time of UserCutCallback Method (ticks)									
Instance	**Random Seed**								
	12			893			39848		
	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.
lin105	166.41	166.41	yes	163.98	163.98	yes	164.84	164.84	yes
lin318	12048.61	12044.67	yes	13340.96	13340.96	yes	13012.67	12991.12	yes
pcb442	13025.12	13025.12	yes	24363.62	23669.71	yes	16192.01	16192.00	yes
pr107	21.93	21.93	yes	21.93	21.93	yes	21.93	21.93	yes
pr124	1413.05	1413.05	yes	1502.10	1500.93	yes	1456.44	1456.44	yes
pr136	2382.08	2382.08	yes	2826.67	2825.72	yes	851.96	851.96	yes
pr144	1543.62	1542.35	yes	1609.22	1609.22	yes	1292.27	1290.92	yes
pr152	1638.50	1638.50	yes	1678.35	1678.35	yes	1509.79	1509.79	yes
pr226	5262.85	5262.24	yes	4453.34	4451.48	yes	5687.06	5686.74	yes
pr264	3289.30	3289.30	yes	3593.27	3593.27	yes	3593.95	3593.95	yes
pr299	40499.64	40087.97	yes	37794.40	37633.71	yes	55855.76	54813.90	yes
pr439	128331.14	128320.98	yes	151237.13	148044.99	yes	118189.71	113108.71	yes
pr76	553.98	553.98	yes	728.05	728.05	yes	771.00	771.00	yes
rat195	4355.20	4355.20	yes	4890.40	4856.68	yes	3377.25	3377.25	yes
rat99	421.66	421.61	yes	383.95	383.68	yes	338.60	338.60	yes
rd100	285.75	285.74	yes	352.86	352.86	yes	339.44	339.44	yes
rd400	26302.16	25902.01	yes	30324.53	30169.80	yes	23614.10	23295.11	yes
st70	105.94	105.94	yes	118.75	118.75	yes	119.51	119.51	yes
u159	1419.25	1419.21	yes	1148.97	1148.96	yes	1213.08	1213.08	yes
ulysses16	1.46	1.46	yes	1.48	1.47	yes	1.46	1.46	yes
ulysses22	2.17	2.17	yes	2.17	2.17	yes	2.17	2.17	yes

Table C.5: Results grouped by the first three random seed of the UserCutCallback Method

Deterministic Time of UserCutCallback Method (ticks)									
Instance	**Random Seed**								
	2648948			201709013			9837745292		
	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.
a280	4193.97	4193.97	yes	4281.77	4281.77	yes	6240.54	6240.54	yes
att48	60.53	60.53	yes	58.55	58.55	yes	60.37	60.37	yes
berlin52	10.64	10.64	yes	10.64	10.64	yes	10.64	10.64	yes
bier127	594.44	594.44	yes	587.75	587.75	yes	603.13	603.04	yes
ch130	754.29	750.56	yes	808.63	807.96	yes	832.02	832.02	yes
ch150	1259.56	1259.55	yes	2149.22	2149.15	yes	1591.74	1591.19	yes
d198	3758.60	3756.28	yes	3355.68	3351.50	yes	4113.08	4113.08	yes
eil101	155.60	155.60	yes	156.41	156.41	yes	156.94	156.94	yes
eil51	55.91	55.91	yes	56.12	56.12	yes	55.92	55.92	yes
eil76	67.72	67.72	yes	31.63	31.63	yes	49.47	49.47	yes
gil262	6986.59	6973.98	yes	6349.40	6349.40	yes	9891.86	9817.25	yes
gr137	1044.47	1044.47	yes	1156.48	1156.48	yes	1012.69	1012.69	yes
gr202	2939.38	2939.38	yes	2617.10	2617.10	yes	2843.88	2843.87	yes
gr229	5472.40	5375.99	yes	6207.04	6120.35	yes	6925.64	6876.09	yes
gr96	515.00	514.11	yes	493.96	493.96	yes	483.71	483.71	yes
kroA100	611.07	610.61	yes	941.99	941.99	yes	569.62	568.79	yes
kroA150	1792.09	1787.14	yes	2140.96	2131.80	yes	1832.24	1832.24	yes
kroA200	7950.20	7950.20	yes	7221.79	7221.79	yes	714603.10	51894.31	yes
kroB100	703.90	703.33	yes	747.84	747.50	yes	711.97	711.61	yes
kroB150	1912.26	1912.26	yes	1975.84	1975.83	yes	2295.92	2295.92	yes
kroB200	3926.28	3926.28	yes	2792.48	2792.48	yes	3196.99	3196.99	yes
kroC100	673.25	673.25	yes	613.66	613.66	yes	557.18	557.09	yes
kroD100	834.09	834.07	yes	609.04	609.04	yes	701.21	701.21	yes
kroE100	840.02	840.02	yes	698.21	698.21	yes	858.13	858.13	yes

Continued on next page

Deterministic Time of UserCutCallback Method (ticks)									
Instance	**Random Seed**								
	2648948			201709013			9837745292		
	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.	Avg. Time	Geom. Time	Opt.
lin105	166.00	166.00	yes	165.05	165.05	yes	166.12	166.12	yes
lin318	11701.92	11691.71	yes	11780.10	11773.87	yes	12000.48	12000.48	yes
pcb442	19021.20	18469.48	yes	16228.01	16228.01	yes	17712.96	17712.96	yes
pr107	21.93	21.93	yes	21.93	21.93	yes	21.93	21.93	yes
pr124	1471.41	1471.40	yes	1540.75	1540.75	yes	1576.04	1576.04	yes
pr136	1095.34	1095.34	yes	1887.81	1887.81	yes	772.20	772.20	yes
pr144	1771.07	1771.07	yes	1667.80	1667.80	yes	1592.56	1591.45	yes
pr152	1579.35	1579.35	yes	1559.42	1559.42	yes	1665.04	1665.04	yes
pr226	4065.01	4065.01	yes	4589.43	4589.43	yes	6216.17	6216.08	yes
pr264	3282.28	3282.28	yes	3314.77	3314.77	yes	3284.72	3284.72	yes
pr299	79280.57	74181.76	yes	45637.73	45525.57	yes	62722.07	60137.89	yes
pr439	797761.59	334853.20	yes	943384.34	555751.27	yes	72584.29	72506.24	yes
pr76	935.79	935.79	yes	843.11	842.17	yes	671.40	671.40	yes
rat195	4583.08	4583.07	yes	4283.71	4283.71	yes	5220.78	5195.58	yes
rat99	327.08	325.84	yes	305.70	305.70	yes	369.37	369.21	yes
rd100	341.59	340.72	yes	323.53	323.53	yes	327.14	326.47	yes
rd400	25938.20	25938.20	yes	19729.11	19697.99	yes	28715.65	28715.65	yes
st70	106.35	106.35	yes	119.51	119.51	yes	119.41	119.41	yes
u159	1363.33	1363.29	yes	1283.75	1283.75	yes	1163.53	1163.52	yes
ulysses16	0.60	0.60	yes	1.46	1.46	yes	1.46	1.45	yes
ulysses22	2.17	2.17	yes	2.17	2.17	yes	2.21	2.21	yes

Table C.6: Results grouped by the second three random seed of the UserCutCallback Method

Bibliography

[1] N. Christofides. *Worst-case analysis of a new heuristic for the travelling salesman problem.* Tech. rep. Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, Feb. 1976.

[2] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition.* 3rd. The MIT Press, July 2009. ISBN: 9780262033848. URL: https://mitpress.mit.edu/books/introduction-algorithms-third-edition.

[3] Elizabeth D Dolan and Jorge J Moré. "Benchmarking optimization software with performance profiles". In: *Mathematical programming* 91.2 (2002), pp. 201–213.

[4] M. Fischetti. *Lezioni Di Ricerca Operativa.* Independently Published, 2018. ISBN: 9781980835011. URL: https://books.google.it/books?id=VYaAuQEACAAJ.

[5] M. Fischetti and A. Lodi. *Local branching.* Tech. rep. 1. Sept. 2003, pp. 23–47. DOI: 10.1007/s10107-003-0395-5. URL: https://doi.org/10.1007/s10107-003-0395-5.

[6] Pierre Hansen and Nenad Mladenović. "Variable neighborhood search". In: *Handbook of metaheuristics.* Springer, 2003, pp. 145–184.

[7] Marek Karpinski, Michael Lampis, and Richard Schmied. "New inapproximability bounds for TSP". In: *Journal of Computer and System Sciences* 81.8 (2015), pp. 1665–1677.

[8] Miroslaw Malek et al. "Serial and parallel simulated annealing and tabu search algorithms for the traveling salesman problem". In: *Annals of Operations Research* 21.1 (1989), pp. 59–84.

[9] C. E. Miller, A. W. Tucker, and R. A. Zemlin. *Integer Programming Formulation of Traveling Salesman Problems.* Tech. rep. 4. New York, NY, USA, Oct. 1960, pp. 326–329. DOI: 10.1145/321043.321046. URL: http://doi.acm.org/10.1145/321043.321046.

[10] A.J. Orman and H Paul Williams. *A Survey of Different Integer Programming Formulations of the Travelling Salesman Problem.* Tech. rep. 2007, pp. 91–104. DOI: 10.1007/3-540-36626-1_5.

[11] T. Sawik. *A note on the Miller-Tucker-Zemlin model for the asymmetric traveling salesman problem.* Tech. rep. 3. 2016, pp. 517–520.

[12] Wikipedia contributors. *Heuristic — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Heuristic&oldid=896446911. [Online; accessed 11-May-2019]. 2019.

[13] Wikipedia contributors. *Matheuristics — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/w/index.php?title=Matheuristics&oldid=824685557. [Online; accessed 11-May-2019]. 2018.

[14] Wikipedia contributors. *Metaheuristic — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Metaheuristic. [Online; accessed 10-July-2019]. 2019.

[15] Wikipedia contributors. *Nearest neighbour algorithm — Wikipedia, The Free Encyclopedia*. https://en.wikipedia.org/wiki/Nearest_neighbour_algorithm. [Online; accessed 04-July-2019]. 2019.