University of Padua

Master Degree in Computer Engineering

A.Y. 2019/2020

Operations Research Course Report

Travelling Salesman Problem

Barison Marco

Moratello Matteo

Obetti Christian

Abstract

On this report we present the traveling salesman problem (TSP), an optimization problem where we want to find the best Hamiltonian path on a given graph, and some deterministic and heuristic solutions to resolve it. The goal of our work is to implement some of the most known algorithms in the literature of the operational research and analyze them to discover which are the best ones to resolve this particular optimization problem. Each solution is tested on a dataset of different instances, results are then compared using a performance profile.
The first part of the report is focused on algorithms based on CPLEX, the solver that we used for this research. These are mostly deterministic algorithms, so they work well only with relatively small instances (less than 600-700 nodes). After realizing the performance profile we can conclude that the best deterministic algorithm is the combination of the UserCut and Heuristic Callbacks based on the Generic Callback.
In the second part of the report we analyze generic heuristic solutions (so not based on CPLEX), ideal to resolve very big instances. After our tests we conclude that VNS (variable neighborhood search) is a very good algorithm for the TSP, simulated annealing can also be a valid alternative while the genetic algorithm has very poor performance.

Contents

Chapter 1

Introduction

1.1 Computational complexity theory

First of all we want to start this report with a brief introduction to the computational complexity theory and in particular to the P and NP classes. The computational complexity theory aim to establish how difficult is a certain problem. In order to do this, it is required a system to describe the complexity of the problem that has general validity, so that it is independent from the computer that we are using. For this reason an algorithm is studied with respect to the number of elementary operations that are required to complete it if it's executed by a Turing machine. So for example if we say that an algorithm is $O(n)$ where n is the size of the instance, it means that it requires in the worst case a number of operation to finish that it's linear with respect to the problem size itself.
We need to spend a few words for the Turing machine. For simplicity we can consider a deterministic Turing machine (DTM) the equivalent of a deterministic computer and a nondeterministic Turing machine (NTM) the equivalent of an ideal nondeterministic computer that is able to exploit multiple action at the same time. Please notice that this is a great simplification, an in-depth discussion about Turing machine and the computational complexity theory exceeds the purpose of this introduction, that is only to give a general idea. Now, in an informal way, we can say that a problem belong the P class if exist an algorithm that resolve the associated decision problem in a polynomial time on a deterministic Turing machine. Moreover we say that a problem belong to the NP class (nondeterministic polynomial) if exist an algorithm able to resolve the associated decision problem in polynomial time on a nondeterministic Turing machine.
A DTM can simulate a NTM, but in the worst case this require an exponential number of operation (again there is a theorem that guarantee this, omitted on this report). This means that a problem that belong to the NP class, can require an exponential number of operation to be resolved on a deterministic computer, making it infeasible if the instance is not small.
There are no theoretical proofs that $P \neq NP$ (This is one of the *Millennium Prize Problems*) so this mean that NP problems may be exponential only because we don't

know an efficient algorithm to resolve them, however nowadays this is considered unlikely and it's widely believed that some problems are intrinsically exponential to resolve.
In conclusion there are some problem that are considered even more difficult than the ones that belong to the *NP* class. These are called *NP-hard* and they are of particular interest because lots of problem that have practical application in reality belong to this typology [1].

1.2 Traveling salesman problem

The traveling salesman problem (TSP) consist in finding the Hamiltonian path [1] of lowest cost given an oriented graph $G = (V, A)$. The problem can be also defined in an analogous way on a non oriented graph if the cost of an edge doesn't depend on the direction of the edge itself. This problem is very common in several real life situations, for example a courier that has to deliver parcels on different locations. However TSP is NP-hard and this is the reason why we can obtain exact solutions only for small instances (up to 600-800 nodes with the most advanced algorithms) while we have to look for different heuristic strategies if the instances are large [2].
A possible integer linear programming model can be the following:

$$\min \underbrace{\sum_{(i,j)\in A} c_{ij}x_{ij}}_{\text{circuit cost}} \tag{1.1}$$

$$\underbrace{\sum_{(i,j)\in\delta^-(j)} x_{ij} = 1}_{\text{one edge incoming in j}}, \quad j \in V \tag{1.2}$$

$$\underbrace{\sum_{(i,j)\in\delta^+(j)} x_{ij} = 1}_{\text{one edge outgoing in i}}, \quad i \in V \tag{1.3}$$

$$\underbrace{\sum_{(i,j)\in\delta^+(S)} x_{ij} \geq 1}_{\text{subtour eliminator}}, \quad S \subset V \ : \ 1 \in S \tag{1.4}$$

$$x_{ij} \geq 0 \text{ integer}, (i,j) \in A \tag{1.5}$$

where x_{ij} are the decision variables defined in this way:

$$x_{ij} = \begin{cases} 1 & \text{if the edge } (i,j) \in A \text{ is selected in the optimal circuit} \\ 0 & \text{otherwise} \end{cases}$$

[1] In the mathematical field of graph theory, a Hamiltonian path (or traceable path) is a path in an undirected or directed graph that visits each vertex exactly once.

The 1.2 means that the sum of all the outgoing edges in each vertex must be equals to 1, similar the 1.3 says that the sum of all the edges entering a vertex must be equals to one. So if we consider both 1.2 and 1.3 we have that each vertex must have only one incoming and one outgoing edge.
The 1.4 says that each vertex v must be reachable from the vertex 1, so that the solution must be connected and must visit all the vertices (subtour are not allowed). These constraints are $O(2^n)$, on the following chapters we will show some possible solutions to handle them.

1.3 CPLEX

CPLEX is an optimization software which is capable of solving Mixed Integer Programming (MIP) models (or quadratic convex) even of considerable size. The version used in our implementations is the 12.10.
CPLEX uses branch-and-cut search when solving MIP models. The branch-and-cut procedure manages a search tree consisting of *nodes*. Every node of the tree represents an LP subproblem to be solved. Nodes not already processed are called active and remain in this state until they are reached and solved during the exploration of the branching tree. The branch-and-cut stops when no more active nodes are available or some limit have been reached.
A *branch* is the creation of two new nodes from a parent node. Our reference problem is the TSP which has binary variables. In this case, if the current LP relaxation solution has a variable with a fractional value, the branching operation creates two child nodes: one node with a modified upper bound of 0 (requiring this variable to take only the value 0), and the other node with a modified lower bound of 1. The solution domains of the two child nodes are thus distinct.
In order to limit the size of the solution domain for the continuous LP problem, it is possible to add *cuts* to the model, without eliminating the legal integer solutions. This makes possible to prune many branches of the decision tree, speeding up the resolution of the MIP.

1.4 Performance Profile

To compare our algorithms we used the Performance Profile [3]. For the comparison of algorithms it takes into account the number of problems solved as well as the cost it took to solve them. It scaled the cost of solving the problem according to the best solver for that problem. Given a set of problems P and a set of algorithms S, we define $c_{s,p}$ as the cost of solving problem $p \in P$ by algorithm $s \in S$. If an algorithm can't solve the problem p, we define $c_{s,p} = +\infty$. We assume that at least one algorithm solves problem p. The best algorithm for a given problem is the one that solves it with the least cost, i.e., we define

$$c_{min,p} = \min_{s \in S} c_{s,p}. \tag{1.6}$$

Now we define the relative cost of the algorithm on a problem:

$$r_{s,p} = \frac{c_{s,p}}{c_{min,p}}. \tag{1.7}$$

Notice that $r_{s,p} >= 1$, with $r_{s,p} = 1$ meaning that algorithm s is (one of) the best for problem p. Finally, the performance function of algorithm s is given by

$$P_s(t) = \frac{\#\{p \in P \mid r_{s,p} <= t\}}{\#P}. \tag{1.8}$$

See that $P_s(1)$ is the number of problems such that $r_{s,p} = 1$, that is the number of problems for which algorithm s is one of the best. Furthermore, $P_s(r_{max})$ is the number of problems solved by algorithm s, where

$$r_{max} = \max_{s \in S,\, p \in P} r_{s,p}. \tag{1.9}$$

The value $P_s(1)$ is called the efficiency of algorithm s and $P_s(r_{max})$ is the robustness[2].

In the next chapters we are going to present all our algorithms together with their implementations and tests. In particular, this report is structured as follows:

- in Chapter 2 we will present some Compact models that we have studied and implemented, showing for each the mathematical formulation, a little description and a performance comparison between them;
- in Chapter 3 we will present Exact algorithms which can solve the TSP problem to optimum using the DJF formulation, handling the $O(2^n)$ SECs constraints with different approaches;
- in Chapter 4 we will present two heuristic algorithms based on CPLEX that can handle problems with thousands of nodes;
- in Chapter 5 we will present some TSP solution builders and a variety of metaheuristic algorithms;
- in Chapter 6 we will present the conclusions of our work;
- in Chapter 7 we have collected the Result Tables of the tests submitted to our algorithms.

All the source code we developed is available at:

`https://github.com/Morat96/RO2-homework` [3].

[2]For more informations on Performance Profile visit `http://abelsiqueira.github.io/blog/introduction-to-performance-profile/`

[3]For more informations on how compile and run the code, see the README.

Chapter 2

Compact Models

Here we introduce different formulations for the traveling salesman problem that we implemented. For all the following models we will use the decision variable x_{ij} already defined in Section 1.2.
According to [4], we considered the case of the Asymmetric version of the traveling salesman problem, which can be considered more general than the Symmetric version.
Main advantage of going with these models, is that once the model is written, it can be solved as a "black box" by CPLEX solver.
To improve the time performance in the computation of the optimal solution, we implemented part of constraints about the subtour elimination through *lazy constraints*. This allowed us to improve the performance of the models because this kind of constraints will be applied only once needed.

2.1 Sequential formulation

For this version, formulated by Miller, Tucker and Zemlin (MTZ) in 1960 [5], we introduce the continuous variable

$$u_i = \text{sequence in which point i is visited } (i \neq 1)$$

and the constraint 2.1d.

$$\min \underbrace{\sum_{(i,j)\in A} c_{ij}x_{ij}}_{\text{circuit cost}} \tag{2.1a}$$

$$\underbrace{\sum_{(i,j)\in\delta^-(j)} x_{ij} = 1\,,}_{\text{one edge incoming in j}} \quad j \in V \tag{2.1b}$$

$$\underbrace{\sum_{(i,j)\in\delta^+(j)} x_{ij} = 1\,,}_{\text{one edge outgoing in i}} \quad i \in V \tag{2.1c}$$

$$u_i - u_j + nx_{ij} \leq n-1 \quad \forall i,j \in N - \{1\},\ i \neq j \ \textbf{(Lazy constraint)} \tag{2.1d}$$

Constraints 2.1d ensure that, if the salesman travels from i to j, then the position of node j is one more than that of node i. This allows to have only a polynomial number of variable and constraints. In total, there are $O(n^2)$ variables and $O(n^2)$ constraints. The problem is no more NP-Hard, but the relaxation of the problem is somewhat weak and as one can see in the section where compact models are compared, it yields to decent performances only for small instances.

2.2 Flow-based formulations

2.2.1 Single commodity flow

Provided by Gavish and Graves in 1978 [6], it adds continuous variables:

$$y_{ij} = \text{flow in arch } (i,j)\ i \neq j$$

$$\min \sum_{(i,j)\in A} c_{ij}x_{ij} \tag{2.2a}$$

$$\sum_{(i,j)\in\delta^-(j)} x_{ij} = 1, \quad j \in V \tag{2.2b}$$

$$\sum_{(i,j)\in\delta^+(j)} x_{ij} = 1, \quad i \in V \tag{2.2c}$$

$$y_{ij} \leq (n-1)x_{ij} \quad \forall i,j \in N,\ i \neq j \tag{2.2d}$$

$$\sum_{j,\ j\neq 1} y_{1j} = n-1 \ \textbf{(Lazy constraint)} \tag{2.2e}$$

$$\sum_{i,\ i\neq j} y_{ij} - \sum_{k,\ i\neq k} y_{jk} = 1 \quad \forall j \in N - \{1\} \tag{2.2f}$$

The idea is that whenever one node is chosen, the salesman delivers one unit of commodity flow. In this way, an order between the nodes is picked and subtours are impossible.
This formulation has $O(n^2)$ variables and $O(n)$ constraints. The relaxation of the linear problem works better than the MTZ, as one can observe in the section about the comparison between the compact models.

2.2.2 Two commodity flow

Created by Finke, Claus and Gunn in 1983. It maintains constraint 2.1b and 2.1c (from this point they and the objective function will not be shown in the formulation) and introduce the continuous variables:

$$y_{ij} = \text{flow of commodity 1 in arc } (i,j)\ i \neq j$$

$$z_{ij} = \text{flow of commodity 2 in arc } (i,j)\ i \neq j$$

and add the constraints:

$$\sum_{j,\, j\neq 1} (y_{1j} - y_{j1}) = n - 1 \ \textbf{(Lazy constraint)} \tag{2.3a}$$

$$\sum_{j} (y_{ij} - y_{ji}) = 1 \quad \forall i \in N - \{1\},\ i \neq j \tag{2.3b}$$

$$\sum_{j,\, j\neq 1} (z_{1j} - z_{j1}) = -(n-1) \ \textbf{(Lazy constraint)} \tag{2.3c}$$

$$\sum_{j} (z_{ij} - z_{ji}) = -1 \quad \forall i \in N - \{1\},\ i \neq j \tag{2.3d}$$

$$\sum_{j} (y_{ij} + z_{ij}) = n - 1 \quad \forall i \in N \tag{2.3e}$$

$$y_{ij} + z_{ij} = (n-1)x_{ij} \quad \forall i,j \in N \tag{2.3f}$$

Constraints 2.3a and 2.3b force $(n-1)$ units of commodity 1 to flow into city 1 and 1 unit of commodity to flow out from every other node.
Constraits 2.3c and 2.3d force $(n-1)$ units of commodity 2 to flow out from city 1 and 1 unit of commodity to flow into every other node.
Constraints 2.3e force exactly $(n-1)$ units of commodity in each arc and constraints 2.3f only allow flow in an arc if present.
The formulation has $O(n^2)$ constraints and $O(n^2)$ variables.

2.2.3 Multi-commodity flow

Proposed by Wong and Claus in 1984. Again constraint 2.1b and 2.1c are maintained. It introduce the continuous variable:

$$y_{ij}^k = \text{flow of commodity k in arc } (i,j) \in N - \{1\}$$

and the following constraints:

$$y_{ij} \leq x_{ij} \quad \forall i,j,k \in N,\ k \neq 1 \tag{2.4a}$$

$$\sum_{i} y_{1i}^k = 1 \quad \forall k \in N - \{1\} \tag{2.4b}$$

$$\sum_{i} y_{i1}^k = 0 \quad \forall k \in N - \{1\} \tag{2.4c}$$

$$\sum_{i} y_{ik}^k = 1 \quad \forall k \in N - \{1\} \tag{2.4d}$$

$$\sum_{j} y_{kj}^k = 0 \quad \forall k \in N - \{1\} \tag{2.4e}$$

$$\sum_i y_{ij}^k - \sum_i y_{ji}^k = 0 \quad \forall j,k \in N - \{1\},\ j \neq k \tag{2.4f}$$

Constraints 2.4a only allow flow in an arc which is present. Constraints 2.4b force exactly one unit of each commodity to flow into city 1 and constraints 2.4c does the opposite, preventing commodity to flow out from city 1.
Constraints 2.4d force exactly one unity of commodity k to flow out from city k and constraints 2.4e prevent any of commodity k flowing into city k.
Finally, constraints 2.4f force balance for all commodities at each node, apart from city 1 and for commodity k at city k.
There is a total number of $O(n^3)$ constraints and $O(n^2)$ variables.

2.3 Time staged formulations

We decided also to explore this section of compact models, exploiting CPLEX's solver that just requires the models to be written. Dealing with performance, especially the T1 and T2 were not working good in practice and required too much to compute the optimal solution.

2.3.1 First stage dependent

Introduced by Fox, Gavish and Graves in 1980. Constraint 2.1b and 2.1c are maintained, in addition we have 0-1 integer variables:

$$y_{ij}^t = \begin{cases} 1 & \text{if the edge } (i,j) \text{ is traversed at stage t} \\ 0 & \text{otherwise} \end{cases}$$

and constraints below, for a total of $n(n+2)$ constraints and $n(n-1)(n+1)$ 0-1 variables.

$$\sum_{i,j,t} y_{ij}^t = n \tag{2.5a}$$

$$\sum_{j,t\ t\geq 2} t y_{ij}^t - \sum_{k,t} t y_{ki}^t = 1 \quad \forall i \in N - \{1\} \tag{2.5b}$$

$$x_{ij} - \sum_t x_{ij}^t = 0 \quad \forall i,j \in N,\ i \neq j \tag{2.5c}$$

with the condition

$$y_{il}^t = 0\ \forall t \neq n, \quad y_{ij}^t = 0\ \forall t \neq 1, \quad y_{ij}^l = 0\ \forall i \neq 1, \quad i \neq j \tag{2.5d}$$

2.3.2 Second stage dependent

Provided by Fox, Gavish and Graves in 1980. It uses the same variables of first stage and constraints 2.1b and 2.1c and it adds:

$$\sum_{i,t\ i\neq j} y_{ij}^t = 1 \quad \forall j \in N \tag{2.6a}$$

$$\sum_{j,t\ j\neq i} y_{ij}^t = 1 \quad \forall i \in N \tag{2.6b}$$

$$\sum_{i\ j\neq i} y_{ij}^t = 1 \quad \forall t \in N \tag{2.6c}$$

$$\sum_{j,t\ t\geq 2} t y_{ij}^t - \sum_{k,t} t y_{ki}^t = 1 \quad \forall i \in N - \{1\} \ \textbf{(Lazy constraint)} \tag{2.6d}$$

In totals there are $4n - 1$ constraints and $n(n-1)(n+1)$ 0-1 variables.

2.3.3 Third stage dependent

Again we have the same variables of first stage and constraints 2.1b and 2.1c. In addition, for a total of $2n^2 - n + 3$ constraints and $n(n-1)(n+1)$ 0-1 variables we have:

$$\sum_j y_{1j}^1 = 1 \tag{2.7a}$$

$$\sum_i y_{i1}^n = 1 \tag{2.7b}$$

$$\sum_j y_{ij}^t - \sum_k y_{ki}^{t-1} = 0 \quad \forall i, t \in N - \{1\} \tag{2.7c}$$

2.4 Final comparison between Compact Models

As explained in the previous sections, we decided mainly to focus on MTZ and Flow Based formulations, because of the time required to compute the optimal solution for the dataset we used.
We tested these algorithms on dataset made of instances with a number of nodes up to 80. For bigger instances, the computation of an optimal solution became unfeasible.
The instances we used to test all our implementations, from compact models to heuristic algorithms, are taken from the TSPLIB [7].
From our results shown in Figure 2.1 and 2.2, we observed that FLOW1 outperformed the other algorithms. Decent results were obtained also with MTZ and FLOW2. In our implementation, FLOW3 had poor performances.

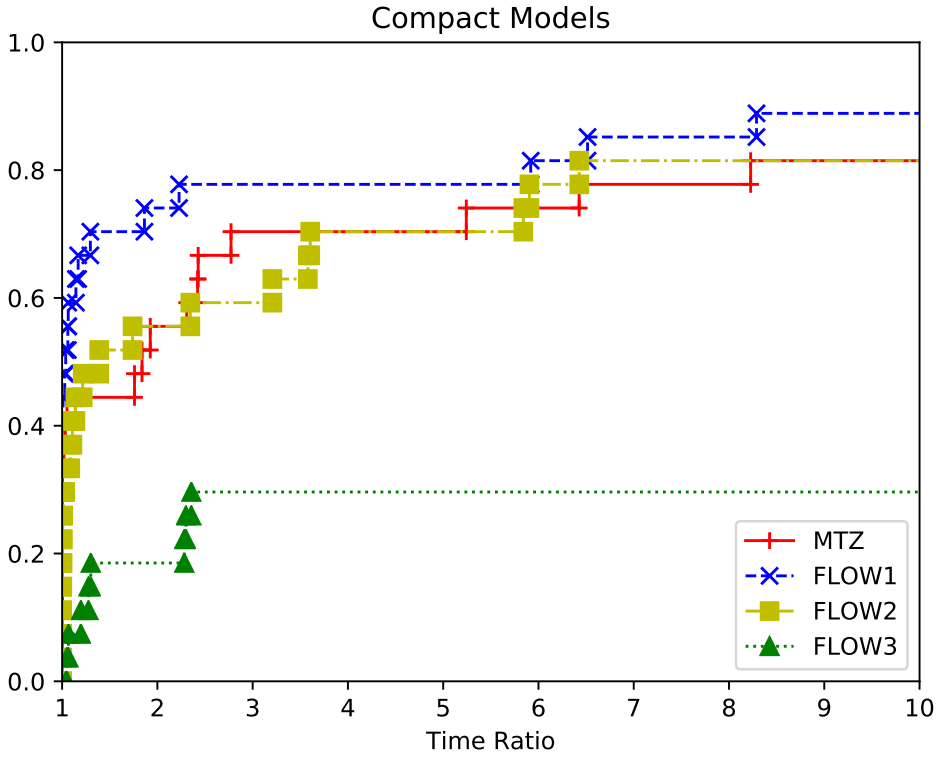


Figure 2.1: Performance profile for compact models with time ratio 10

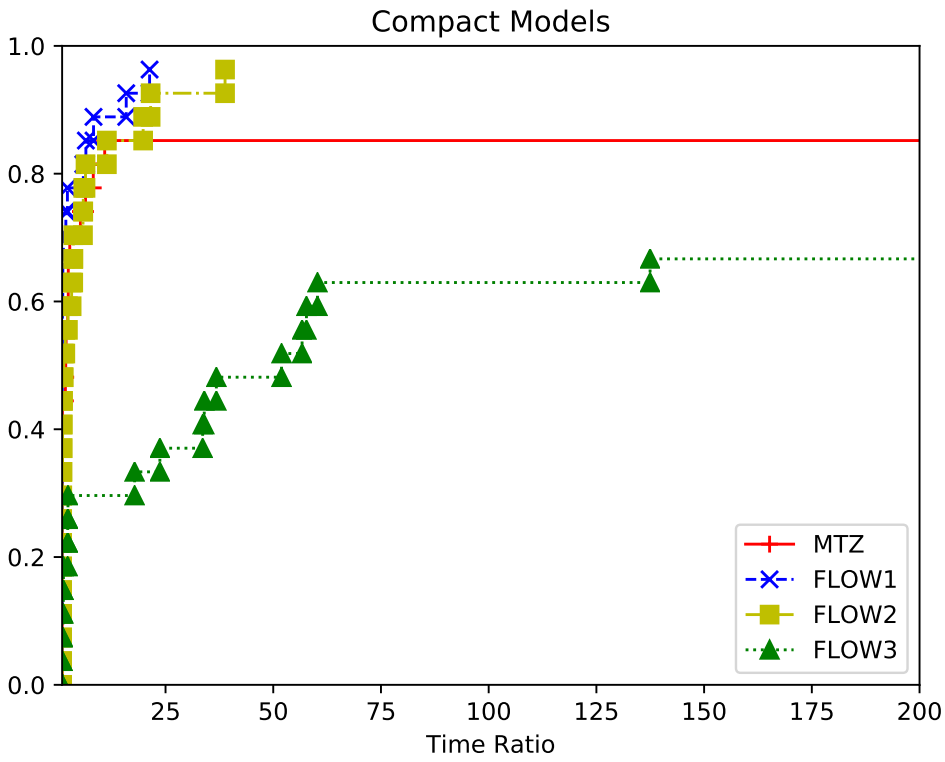


Figure 2.2: Performance profile for compact models with time ratio 200

Chapter 3

Exact Algorithms

On this chapter we will describe exact algorithms based on the Dantzig-Fulkerson-Johnson formulation [8], described in Section 1.2. The main problem of this model is that the formulation has $O(2^n)$ SECs, so it is impossibile to implement all the constraints in the model since it would result in a very high execution time and memory usage even for small graphs.
For this reason, SECs constraints are added only when necessary in the model, in order to discard solutions with cycles. Using this technique, hopefully, only a small part of $O(2^n)$ SECs are added to the model.
The main algorithms that make use of this technique are Loop methods and CPLEX callbacks.

3.1 Loop Methods

On this section we will describe two iterative approaches, originally proposed by [9], in order to solve the DFJ model without adding an exponential number of Cycle constraints.

3.1.1 Simple Loop

The first approach, that we called Simple Loop, is presented in Algorithm 1.
In line 2, we initialize the model with the variables, the degree constraints and the objective function. The algorithm then proceeds solving the problem within a while loop (lines 3-4), and at each iteration we check if the the optimal solution has more than one component (line 5). If so, we add, for each component, the corresponding subtour eliminator. Instead, if the optimal solution has one component (lines 7-8), i.e. a Hamiltonian cycle, then we return the optimal solution found (line 9).
This algorithm turns out very efficient, especially with recent solvers. Its efficiency is due to the fact that the solution is solved very fast, especially in first iterations, and only small number of SECs are added (line 6) at each iteration. These constraints allow to exclude a lot of solutions with subtours of the original problem.
At worst, since the problem is NP-hard, all constraints will be added to the model,

requiring exponential computational time, but in most cases this doesn't happen. It may seem that this method is not very efficient, but with recent and increasingly sophisticated solvers, this method can lead to the optimal solution, even with many nodes, in few seconds.

Algorithm 1 Simple Loop

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 SECs for each connected component to the model** ∗
7: **else**
8: *done* ← *true*
9: **return** z^*

3.1.2 Heuristic Loop

The Algorithm 1 has a drawback, since at each iteration we solve to the optimum the TSP problem, even if the constraints collected until that point, are not sufficient to achieve the true optimum. In other words, we must wait that the solver finds out the optimal solution, even if the constraints avaiable may allow solutions with subtours.
For this reason, we implemented an "heuristic" version of the method seen before. This method is composed of two phases. The first phase is used to collect some SEC constraints and to relax the optimality concept, setting for example an higher value for the solution GAP or limiting the solution in the root node of the branching tree. This phase permits to add useful constraints to the model, without computing the optimal solution in each iteration. In fact, with this phase we can obtain a lot of useful SECs in a short time, especially in the first iterations. This phase is active until the solution found has more than one component, from then on, it will be activated the second phase. In the second phase, the "relaxation" of the solver is removed, solving essentially the problem with the Simple Loop method and this ensure that the solution computed is the true optimal solution.
The Heuristic Loop is presented in Algorithm 2. In line 4, as with the first method, we initialize the model with the variables, the degree constraints and the objective function. At the beginning, the first phase is active (line 2), which means that some CPLEX parameters have to be setted (line 5) in order to obtain a non-optimal solution, speeding up the computational time. In our case, we decided to set two CLEX parameters, that is, the solution's GAP and the NODE lim. The first parameter force CPLEX to stop, when the solution found until that point has a GAP of a

certain value (for example 5%). The second parameter specifies the node of the branching tree until which CPLEX can compute the solution, for example, if the value of the parameter is setted to 0, CPLEX can compute the solution only in the root node. The algorithm then proceeds solving the problem within a while loop (line 6). As in the Simple Loop, at each iteration we check if the solution has more than one component, and if so, we add, for each component, the corresponding sub-tour elimination. Instead, If the optimal solution has one component, we set the flag *second_phase* to true (line 15) and reset the solver settings, thus ensuring that the solution found from now on it will be an optimal solution. During the first phase, we have collected, hopefully, a number of useful SEC constraints in a small amount of time, with respect to found the optimal solution at each iteration. For the second phase, please refer to the Simple Loop (Algorithm 1).

Algorithm 2 Heuristic Loop

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

1: *done* ← *false*
2: *first_phase* ← *true*
3: *second_phase* ← *false*
4: model ← ∗ **Initialize variables and objective function** ∗
5: CPXsetintparam ← ∗ **Set MIP Gap and/or Node lim in order to obtain a non-optimal solution** ∗
6: **while** !done **do**
7: **if** *second_phase* **then**
8: CPXsetintparam ← ∗ **Return to original solver setting in order to obtain an optimal solution** ∗
9: *second_phase* ← *false*
10: z^* ← optimal_solution(model)
11: **if** $components(z^*) > 1$ **then**
12: ∗ **Add SECs for each connected component to the model** ∗
13: **if** $components(z^*) == 1$ & $first\_phase$ **then**
14: *first_phase* ← *false*
15: *second_phase* ← *true*
16: **if** $components(z^*) == 1$ & $!first\_phase$ **then**
17: *done* ← *true*
18: **return** z^*

3.1.3 Final comparison between Loop Methods

The dataset used to compare exact algorithms consists of 30 instances with four different random seeds {201909284, 19, 190696, 19061996}, for a total of 120 runs per algorithm. The instances used are the following:

- a280.tsp
- att48.tsp
- att532.tsp
- berlin52.tsp
- bier127.tsp
- ch130.tsp
- ch150.tsp
- d198.tsp
- gil262.tsp
- kroA100.tsp
- kroA150.tsp
- kroA200.tsp
- kroB100.tsp
- kroB150.tsp
- kroB200.tsp
- kroC100.tsp
- kroD100.tsp
- kroE100.tsp
- lin105.tsp
- lin318.tsp
- pr226.tsp
- rat195.tsp
- pr76.tsp
- pr107.tsp
- pr124.tsp
- pr136.tsp
- tsp225.tsp
- u159.tsp
- rat99.tsp
- rd400.tsp

For each problem, we compare the elapsed time to compute the optimal solution. In Figure 3.1 it is possible to see the performance profile of the Loop Methods we analyzed in this section. The parameter we used are the following:

- **Simple Loop:** no parameters;
- **Heuristic Loop:**
  - **MIP gap tolerance:** Absolute tolerance on the gap between the best integer objective and the objective of the best node remaining. Value used: 0.05;
  - **MIP node limit:** Maximum number of nodes solved before the algorithm terminates. Value used: 0 (only the root node);
  - **MIP solution limit:** Number of MIP solutions to be found before stopping. Value used: 1.

The parameters for the Heuristic Loop are used in the first part of the algorithm, then they are reset to their original values to find the optimal solution. From the graphs we can see that the performance of the two algorithms are very similar. In the first part of the Heuristic Loop, the SECs found are probably not sufficient to save time during the second part of the algorithm. In our opinion, there is no reason to use the Heuristic Loop since, even with three parameters, there is always the risk to overtune the parameters using only few instances.

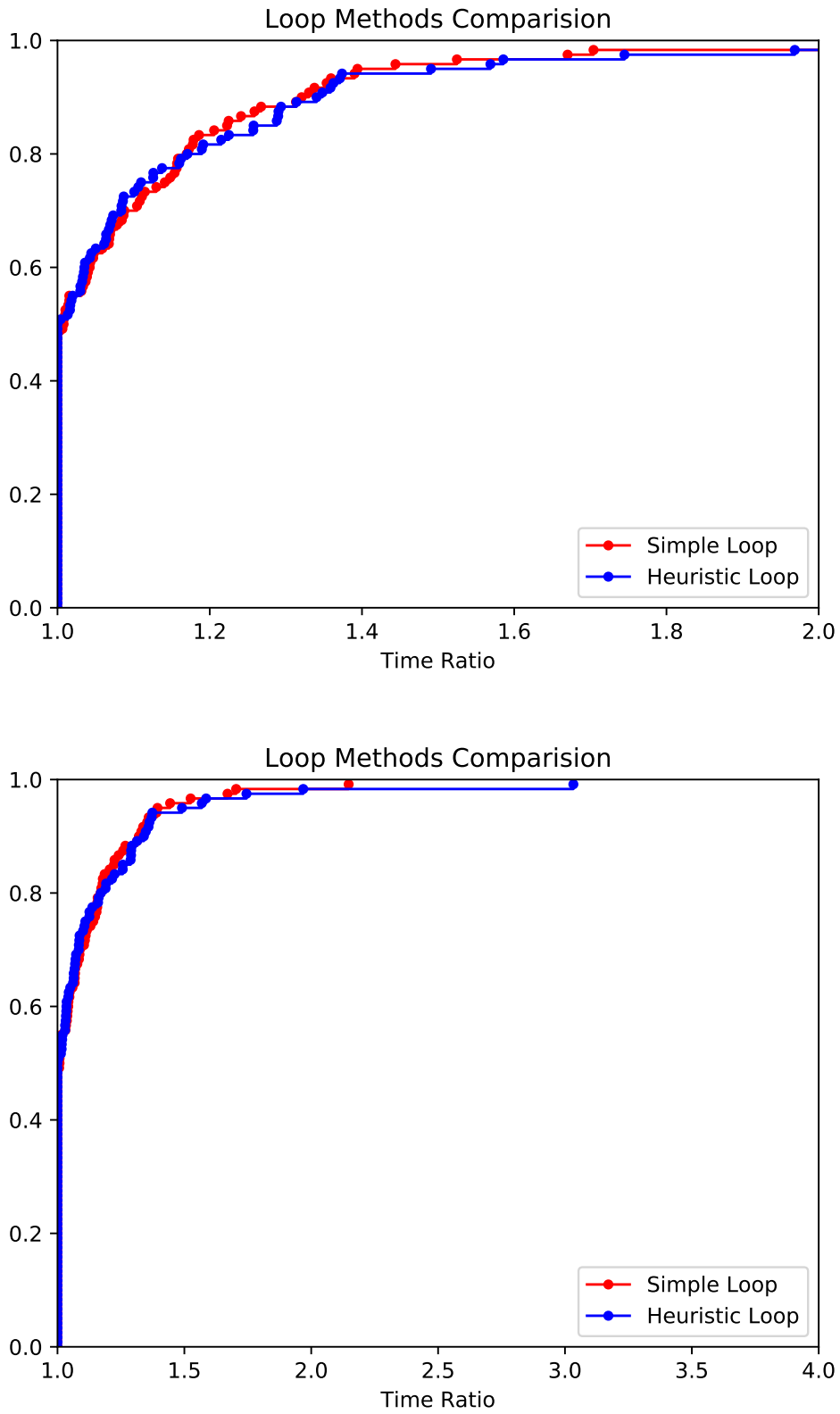


Figure 3.1: On top: detailed view of the performance profile of Loop Methods. On the bottom: full view of the performance profile of Loop Methods.

3.2 Callbacks

On this section we will describe how to solve the DFJ model using CPLEX callbacks. Until now, the algorithms we described resolve the TSP problem with an iterative approach, that is, solve multiple times the problem of a relaxed model, adding the SECs constraints when the solution found has multiple tours. Since CPLEX uses the branch-and-cut technique to solve exactly the model, with this approach multiple decision trees will be created from scratch, losing much of the information from previous trees. Another way to handle the SECs constraints is to exploit the branch-and-cut technique. In particular, when an integer solution is found by CPLEX during the decision tree, we can reject it if contains multiple tours, adding for each of them a SEC constraint. With this method we develop only a single decision tree, and this results, hopefully, in a faster computation. To reach this goal, CPLEX provides a series of callbacks that can be used for adding constraints and solutions on the fly.

3.2.1 Lazy Callback

The lazy callback allows to resolve the TSP problem in a very fast and natural way. As described previously, in order to solve the problem we must reject solutions that have multiple tours, adding SECs constraints on the fly. This leads to Algorithm 3 in which we instantiate the model and the objective function as usual (line 2). Then we instantiate a procedure called LazyCallback (line 3). A LazyCallback is a function called by the optimizer when an integer solution is found during the branch-and-cut algorithm. The task of the callback is to check if the solution found has more than one connected component (line 8) and if it is the case, to reject the solution adding the SECs constraints to the model. After this, the resolution of the model resumes (lines 9 - 11). The routine continues by optimally solving the model and returning the solution found (lines 4 - 5).

Algorithm 3 Lazy Callback

Output: $\boldsymbol{z}$***** optimal solution to TSP on the input graph G

1: **function** TSPOPT($G = (V, E), c : E \to \mathbb{R}^+$)
2: model $\leftarrow$ $*$ **Initialize variables and objective function** $*$
3: optimizer $\leftarrow$ $*$ **Initialize the LazyCallback function within the optimizer** $*$
4: $\boldsymbol{z}$***** $\leftarrow$ optimizer(model)
5: **return** $\boldsymbol{z}$*****
6:
7: **function** LAZYCALLBACK($\boldsymbol{x}$, model)
8: comp $\leftarrow$ $*$ **number of components in the integer solution** $\boldsymbol{x}$ $*$
9: **if** ($comp > 1$) **then**
10: $*$ **Add SECs for each connected component to the model** $*$
11: **return**

3.2.2 UserCut Callback

Another interesting callback is the UserCut which is used to generate SECs constraints on fractional solutions. The idea is pretty the same as the Lazy callback; when a fractional solution is found during the exploration of the decision tree by the solver, the UserCutCallback function checks the solution and adds constraints to the model on the fly. The procedure is described in algorithm 4. In this case both LazyCallback and UserCutCallback must be defined (line 3). In particular, the UserCutCallback function checks how many components there are in the solution (line 14), if there are many connected components then it adds the SECs for each connected component (lines 15 - 16). Instead, when the graph is connected, we apply to the model the violated cuts, looking for sections with a capacity less than a certain threshold. The cutoff value is set to 2.0 - ϵ, with $\epsilon = 0.1$ in our implementation (lines 17 - 18). We used the *Concorde*[1] algorithms in order to fast computing the connected components and violated cuts in fractional solutions. Due to the fact that fractional solutions are more than integer solutions and that the UserCutCallback function employs several time consuming algorithms, the callback is called only when the depth of the decision tree is less than or equal to 10, which is a reasonable tradeoff between number of constraints applied to the model and time spent in executing flow algorithms.

Algorithm 4 UserCut Callback

Output: $\boldsymbol{z}^*$ optimal solution to TSP on the input graph G

1: **function** TSPOPT($G = (V, E), c : E \to \mathbb{R}^+$)
2: model $\leftarrow$ $*$ **Initialize variables and objective function** $*$
3: optimizer $\leftarrow$ $*$ **Initialize the callback functions within the solver** $*$
4: $\boldsymbol{z}^*$ $\leftarrow$ optimizer(model)
5: **return** $\boldsymbol{z}^*$
6:
7: **function** LAZYCALLBACK($\boldsymbol{x}$, model)
8: comp $\leftarrow$ $*$ **number of components in the integer solution** $\boldsymbol{x}$ $*$
9: **if** ($comp > 1$) **then**
10: $*$ **Add SECs for each connected component to the model** $*$
11: **return**
12:
13: **function** USERCUTCALLBACK($\boldsymbol{x}$, model)
14: comp $\leftarrow$ $*$ **number of components in the fractional solution** $\boldsymbol{x}$ $*$
15: **if** ($comp > 1$) **then**
16: $*$ **Add SECs for each connected component to the model** $*$
17: **if** ($comp = 1$) **then**
18: $*$ **Add SECs on the separated fractionary solution** $*$
19: **return**

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

3.2.3 Heuristic Callback

The last callback that we present is the Heuristic which is used to provide to the solver heuristic feasible solutions from fractional or infeasible solutions. In our case, since we are trying to solve the TSP problem, we generate first a tour from integer solutions with multiple connected components and then we add them to the solver using the CPLEX heuristic callback, which permits to add solutions and to automatically check their feasibility. This technique is very useful in the early stages of the branch-and-cut algorithm because, by providing a TSP solution to the solver, we can provide an upper-bound to the optimal solution, in order to cut many branches of the decision tree. This led us to define the Algorithm 5. The main differences with respect to Algorithm 4 are the addition of a new function called HeuristicCallback and a new algorithm to compute a tour starting from an integer solution with multiple components provided in the LazyCallback function. Since we want the algorithm to be thread safe, we must keep track of the index of each thread that produce a specific solution. To do this, for every integer solution with multiple connected components provided by the Lazy callback, we compute a new solution composed by a single tour (i.e. a TSP solution), saving in addition the index of the thread that has computed that solution (lines 11 - 12). The algorithm to compute a tour starting from an infeasible solution is presented in Algorithm 6. The idea is very simple, combine connected components in a single tour optimizing the total length by merging the most nearest components. In particular, the algorithm starts by computing the number of components of the solution (line 2). This value correspond to the iterations that the algorithm must perform in order to merge all the components of the solution (line 3). Inside the while loop, the algorithm compute the minimum delta for each pair of nodes of the graph that are in different components. The delta correspond to the incremental cost of the objective function due to a specific swap of edges. Two cases are possible (lines 6 - 13); in the first case, the edge composed of $(a, a' = succ(a))$ is replaced by (a, b) and the edge (b, b') is replaced by (b', a'), increasing the objective function of $\Delta(a, b)$. In this case, the order of the edges in the second component must be reversed to preserve the correct direction of the graph (Figure 3.2 a - b). The second case is simpler and it is obtained by crossing the edges as in Figure 3.2 c and d. Once the minimum delta is found, the correct swap is applied to the graph (lines 14 - 20) and the number of components is reduced by 1 (line 21 - 22). Concluding the description of algorithm 5, when the HeuristicCallback function is called by the solver, it checks whether a solution has been saved for the current thread (line 24) and, if a solution exists, it adds it to the solver, leaving the latter to verify its feasibility (line 25).

Algorithm 5 Heuristic Callback

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

1: **function** TSPOPT($G = (V, E), c : E \to \mathbb{R}^+$)
2: model $\leftarrow$ $*$ **Initialize variables and objective function** $*$
3: optimizer $\leftarrow$ $*$ **Initialize the callback functions within the solver** $*$

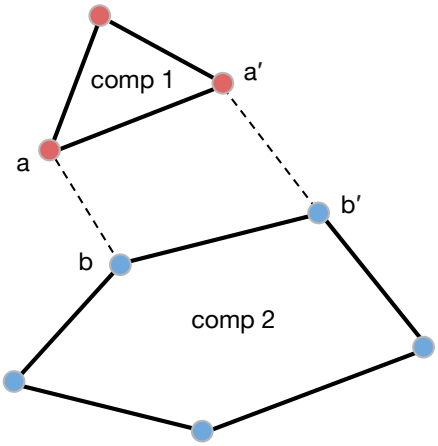
```
    z* ← optimizer(model)
    return z*

function LazyCallback(x, model)
    comp ← * number of components in the integer solution x *
    if (comp > 1) then
        * Add SECs for each connected component to the model *
        x' ← complete_tour(x)
        * Save x' and the thread index *
    return

function UserCutCallback(x, model)
    comp ← * number of components in the fractional solution x *
    if (comp > 1) then
        * Add SECs for each connected component to the model *
    if (comp = 1) then
        * Add SECs on the separated fractionary solution *
    return

function HeuristicCallback(x, model)
    if * there is a solution available in the current thread * then
    * Add the TSP heuristic solution x' to the solver *
    return
```

Algorithm 6 complete tour

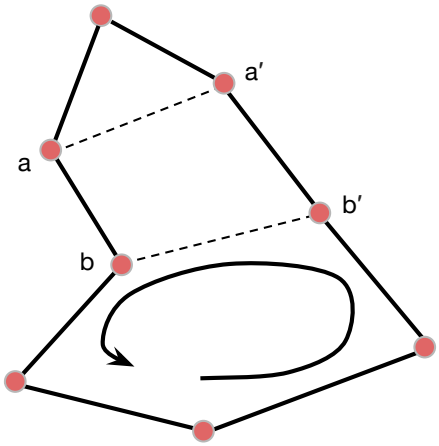
Input: $\boldsymbol{x}$ solution with multiple connected components
Output: $\boldsymbol{x}'$ heuristic TSP solution

```
min ← ∞
ncomp ← components(x)
while (ncomp != 1) do
    for each a, b ∈ V do
        if (comp(a) != comp(b)) then
            Δ(a, b) = dist(a, b') + dist(b, a') − dist(a, a') − dist(b, b')
            if (Δ(a, b) < min) then
                min = Δ(a, b)
                flag ← 0
            Δ(a, b)' = dist(a, b) + dist(b', a') − dist(a, a') − dist(b, b')
            if (Δ(a, b)' < min) then
                min = Δ(a, b)'
                flag ← 1
```

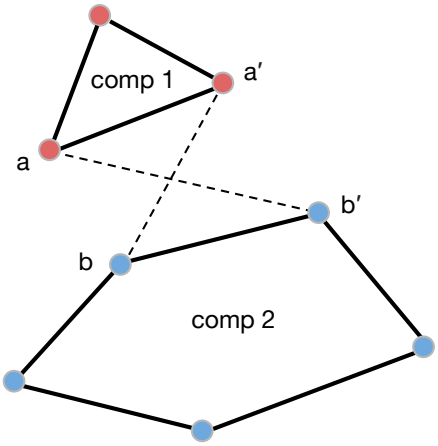
14: **if** $(flag = 1)$ **then**
15: $*$ **reverse the order of edges of the second component** $*$
16: $(a, b) \leftarrow (a, a')$
17: $(b', a') \leftarrow (b, b')$
18: **if** $(flag = 0)$ **then**
19: $(a, b') \leftarrow (a, a')$
20: $(b, a') \leftarrow (b, b')$
21: $*$ **update components of** $\boldsymbol{x}$ $*$
22: ncomp $\leftarrow$ ncomp $- 1$
23: **return** $\boldsymbol{x}$



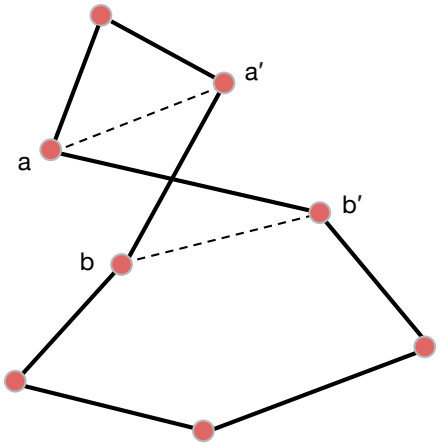
(a) First case: before merging



(b) First case: after merging



(c) Second case: before merging



(d) Second case: after merging

Figure 3.2: The two ways to merge connected components used by the complete_tour algorithm.

Since the overall algorithm is $O(n^2)$, the Heuristic Callback is called only when the depth of the decision tree is less than or equal to 10.

3.2.4 Generic Callback

The Generic Callback is a relatively new function that contains essentially two new features with respect to previous callbacks. The first one is that the callback, during a call to a user-defined function, doesn't stop the execution of the *Dynamic Search* algorithm. The latter is a branch-and-cut based algorithm, which is kept secret by CPLEX and which guarantee better performance than the traditional but robust branch-and-cut algorithm. The second feature is that with a single callback we can define each previous callback using the *context*. The context is a CPLEX structure that defines in which part of the brach-and-cut algorithm the solver calls the callback. An example of the use of Generic Callback is presented in Algorithm 7. This algorithm includes the three callbacks implemented in Algorithm 5, namely, Lazy callback, UserCut Callback and Heuristic Callback. The solver calls the generic callback function with the context CANDIDATE when an integer solution is found (line 8). In this case, we simply collected all function included in the Lazy callback and in the Heuristic callback (lines 9 - 15). On the other hand, after each fractional solution found by the solver the context used is RELAXATION (line 16). In this part of the algorithm, we collected all the function included in the UserCut callback together with the Concorde algorithms (lines 17 - 21).

Algorithm 7 Generic Callback

```
    Output: z* optimal solution to TSP on the input graph G
 1: function TSPopt(G = (V, E), c : E → ℝ⁺)
 2:     model ← ∗ Initialize variables and objective function ∗
 3:     optimizer ← ∗ Initialize the callback function within the solver ∗
 4:     z* ← optimizer(model)
 5:     return z*
 6:
 7: function GenericCallback(x, context, model)
 8:     if (context = CANDIDATE) then
 9:         comp ← ∗ number of components in the integer solution x ∗
10:         if (comp > 1) then
11:             ∗ Add SECs for each connected component to the model ∗
12:             x' ← complete_tour(x)
13:             ∗ Save x' and the thread index ∗
14:         if ∗ there is a solution available in the current thread ∗ then
15:         ∗ Add the TSP heuristic solution x' to the solver ∗
16:     if (context = RELAXATION) then
17:         comp ← ∗ number of components in the fractional solution x ∗
18:         if (comp > 1) then
19:             ∗ Add SECs for each connected component to the model ∗
```

```
20:         if (comp = 1) then
21:             * Add SECs on the separated fractionary solution *
22:     return
```

3.2.5 Final comparison between callbacks

For the final comparison of the callbacks we used the same dataset described in Section 3.1.3, i.e. 30 instances with four random seeds, for a total of 120 problems. For the comparison we used two metrics: time elapsed and nodes solved. The first metric is used to compare the time spent by the algorithms to compute the optimal solution of each problem. The second metric is used to compare the number of nodes solved by the algorithms when running the branch-and-cut algorithm. The number of nodes solved can make us understand if the cuts applied by the callbacks are sufficient to prevent the expansion of numerous nodes of the branched tree during the branch-and-cut algorithm.
On Figure 3.3 it is possible to see the performance profile of the Legacy Callbacks. In CPLEX the legacy callbacks are the previous version of the new generic callback and *Lazy Callback*, *UserCut Callback* and *Heuristic Callback* are part of this category. The first thing we can notice is that *Lazy Callback* is the one that performs worst. The *Heuristic Callback* seems to be performing slightly better than the *UserCut Callback*. The heuristic solutions provided to the solver using the Heuristic Callback together with the cuts computed in fractional solutions using Concorde, can significantly reduce the amount of time to compute the optimal solution for the TSP problem. The number of nodes solved by the three algorithms is also consistent with the times, as you can see in Figure 3.4. This confirms that cuts and heuristic solutions are capable of cutting many branches of the decision tree.
On Figure 3.5 it is possible to see the performance profile of the Generic Callbacks. Also in this case the *Heuristic Callback* outperforms all the other both in terms of time and number of solved nodes, as you can see in Figure 3.6.
In Figure 3.7 we can see the comparison between all the Callbacks. The *Legacy Heuristic Callback* outperforms the others in most of the instances, however it does not do so consistently. In a small percentage of instances it takes a really long time to finish and is outperformed by the *Generic Heuristic Callback*. This means that if we were to look at the percentage of instances solved quickly, the *Legacy Heuristic Callback* is the best method, but the method capable of solving the most of instances within 4 times the time it takes the fastest method to find the optimum is the *Generic Heuristic Callback*. This is followed by the *Generic UserCut Callback* with slightly worse performance and the *Legacy UserCut Callback*.
Lazy Callbacks are the ones that perform worst. The number of nodes solved by each algorithm is shown in Figure 3.8.
On Figure 3.9 it is possible to see the performance profile of the Exact Algorithms. The only new thing to note is that *Loop Methods* perform better than *Legacy Lazy Callback*. All performance profiles are shown on the following pages.

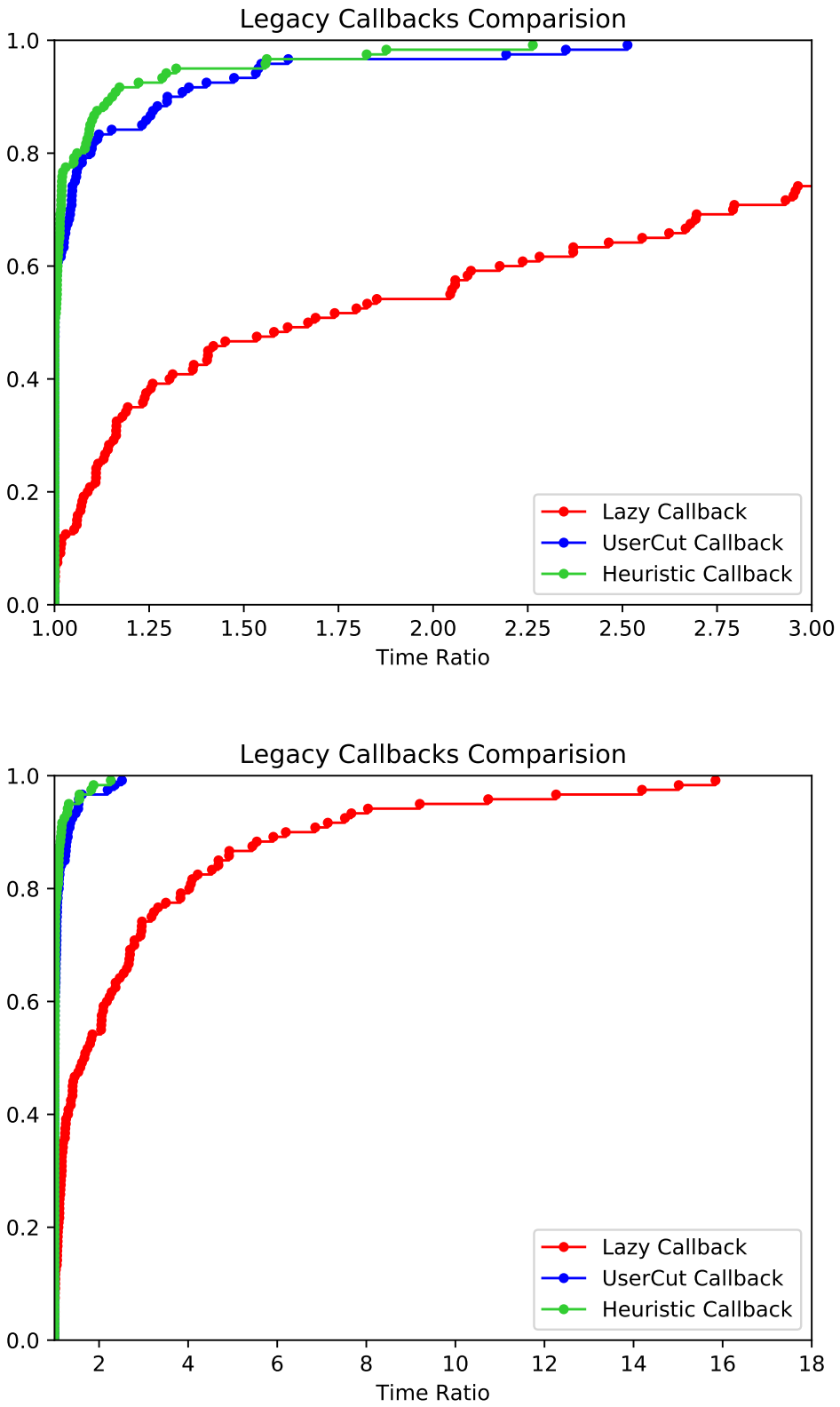


Figure 3.3: On top: detailed view of the performance profile of Legacy Callbacks. On the bottom: full view of the performance profile of Legacy Callbacks.

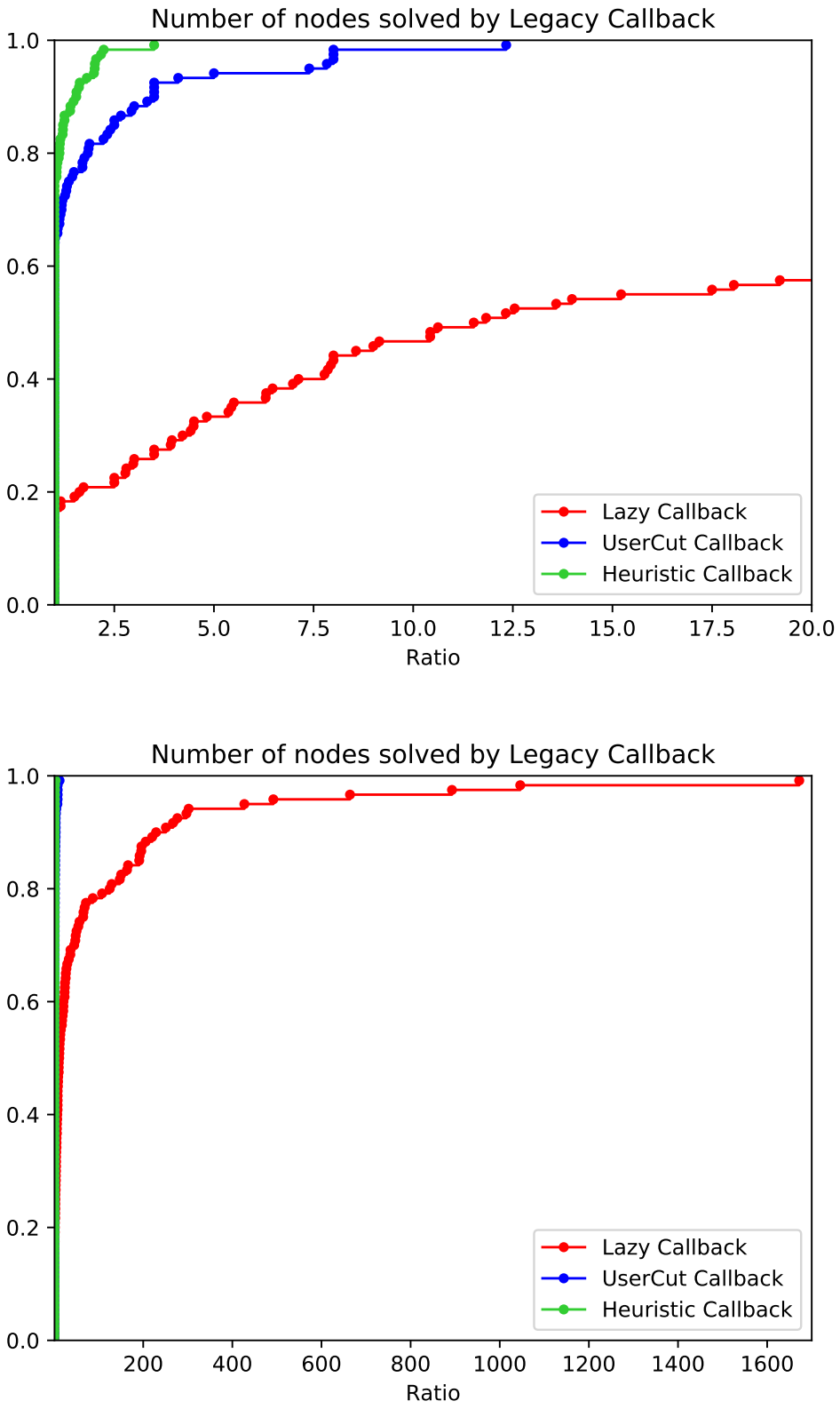


Figure 3.4: On top: detailed view of the performance profile of Legacy Callbacks regarding the number of solved nodes. On the bottom: full view of the performance profile of Legacy Callbacks.

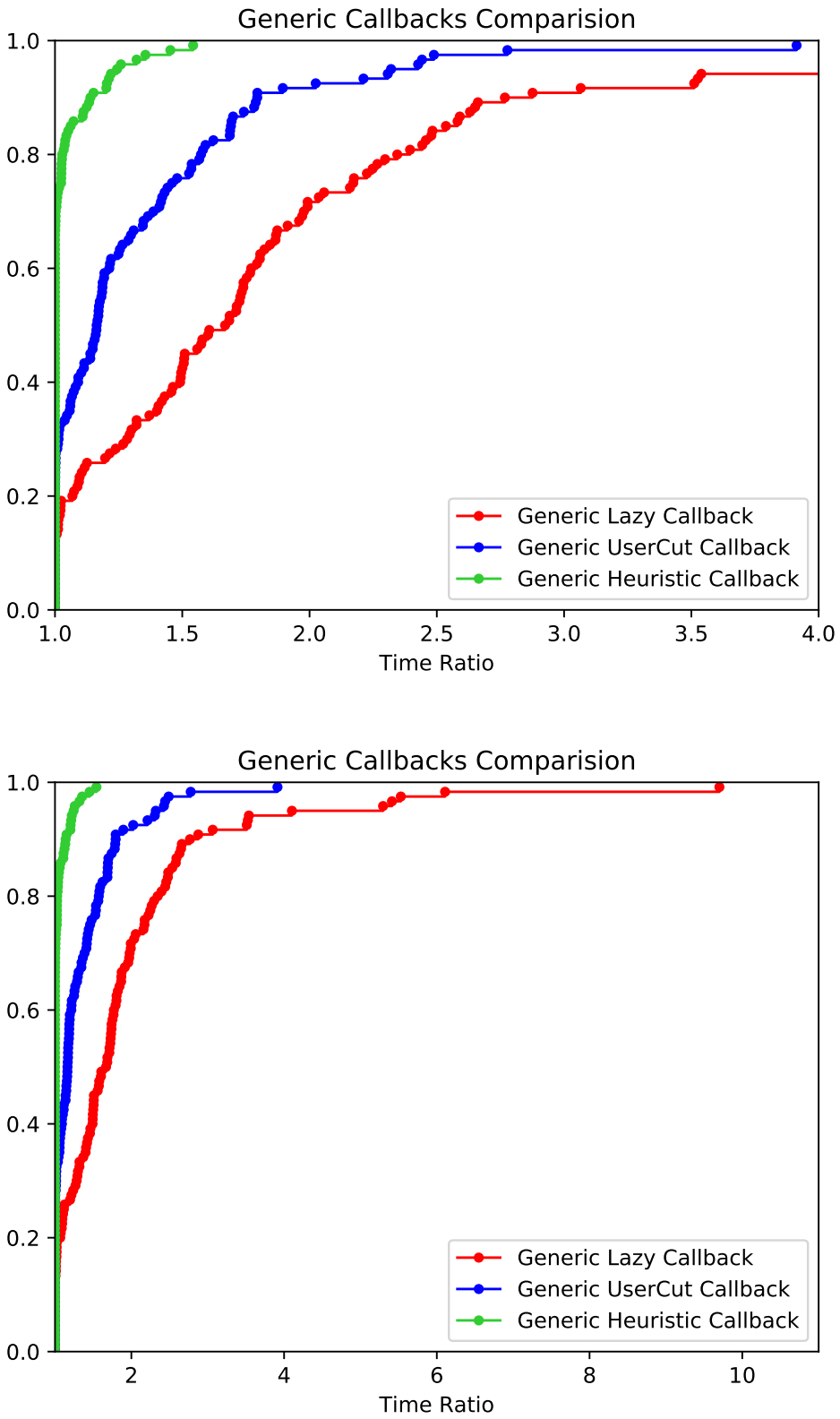


Figure 3.5: On top: detailed view of the performance profile of Generic Callbacks. On the bottom: full view of the performance profile of Generic Callbacks.

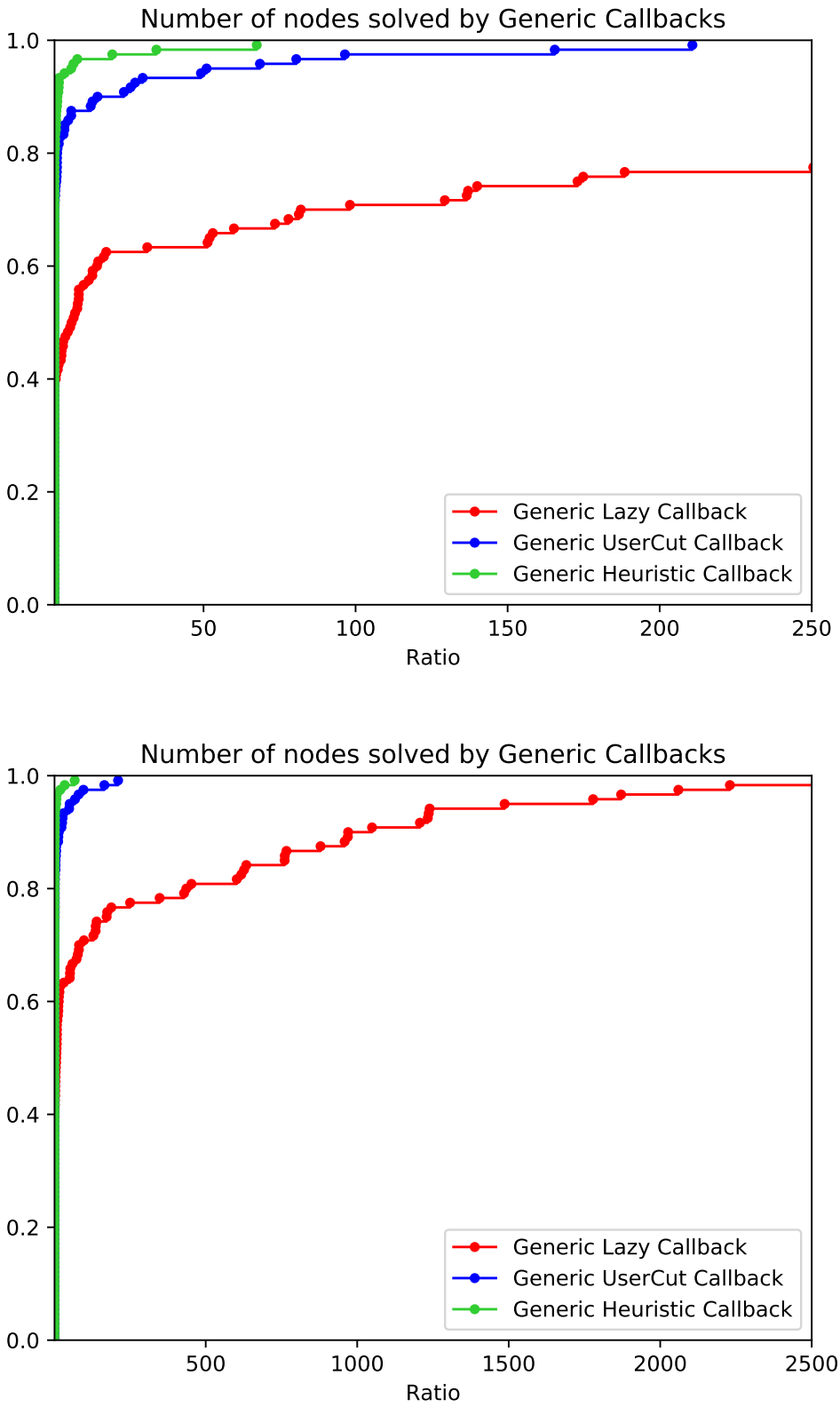


Figure 3.6: On top: detailed view of the performance profile of Generic Callbacks regarding the number of solved nodes. On the bottom: full view of the performance profile of Generic Callbacks.

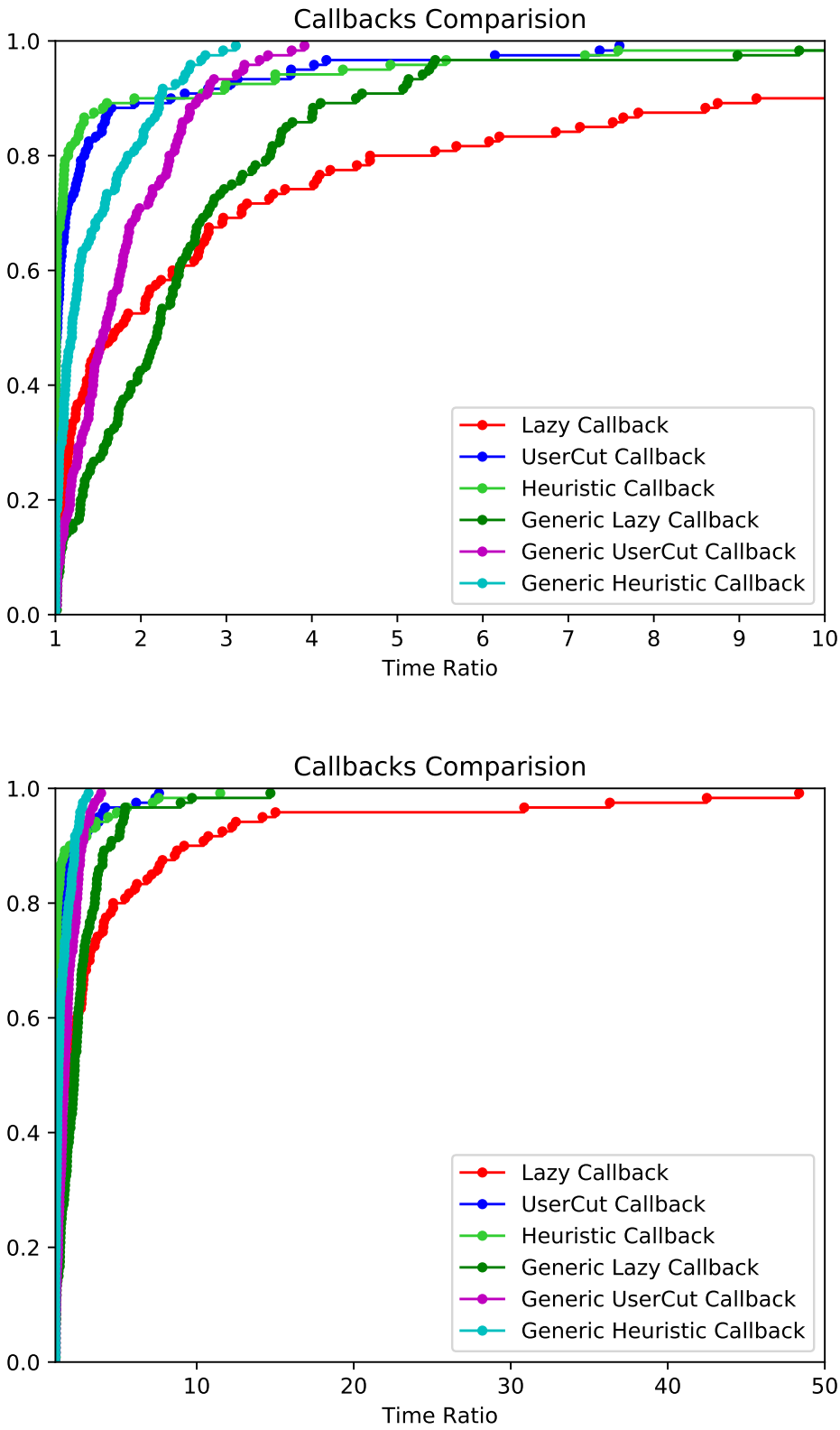


Figure 3.7: On top: detailed view of the performance profile of all Callbacks. On the bottom: full view of the performance profile of all Callbacks.

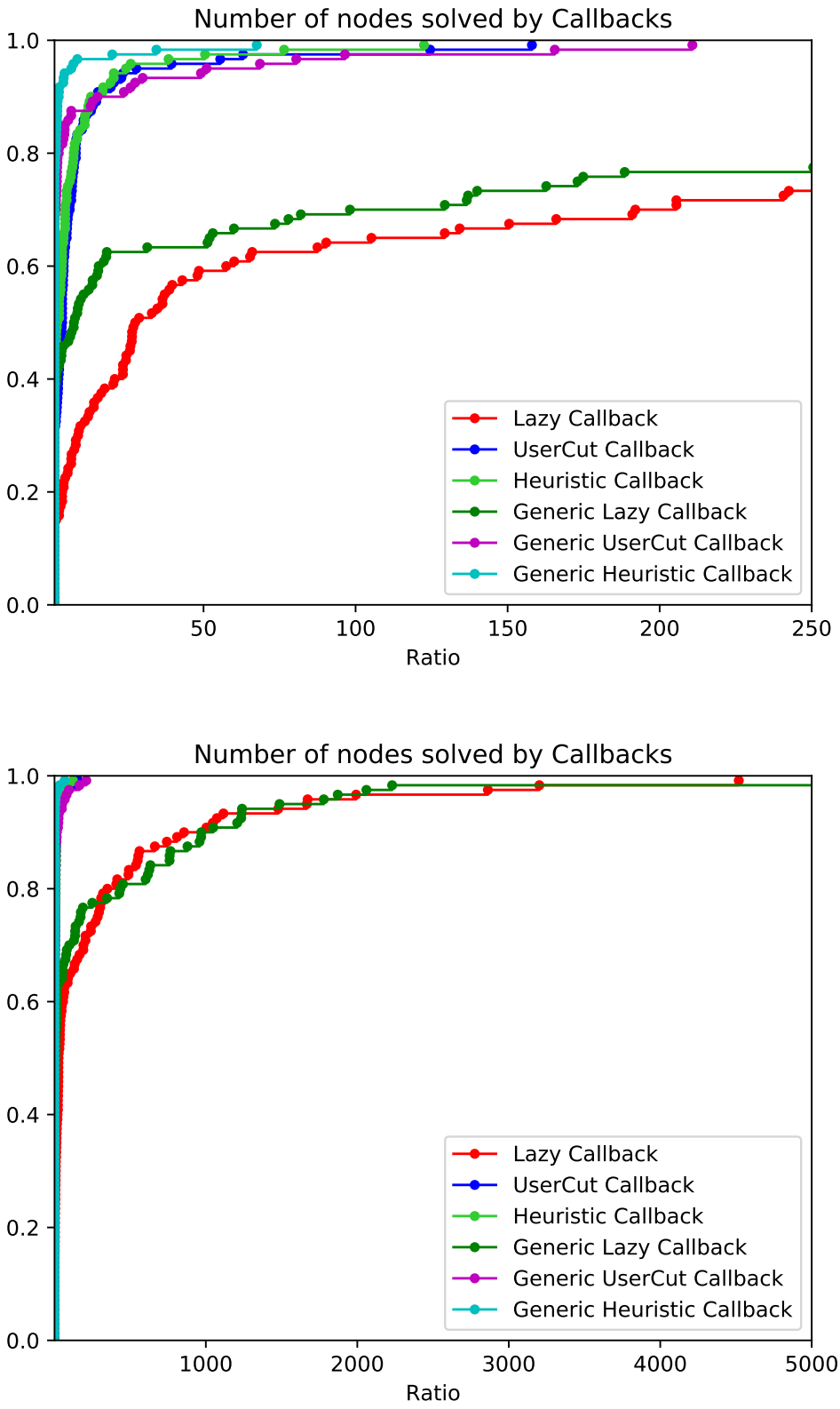


Figure 3.8: On top: detailed view of the performance profile of all Callbacks regarding the number of solved nodes. On the bottom: full view of the performance profile of all Callbacks.

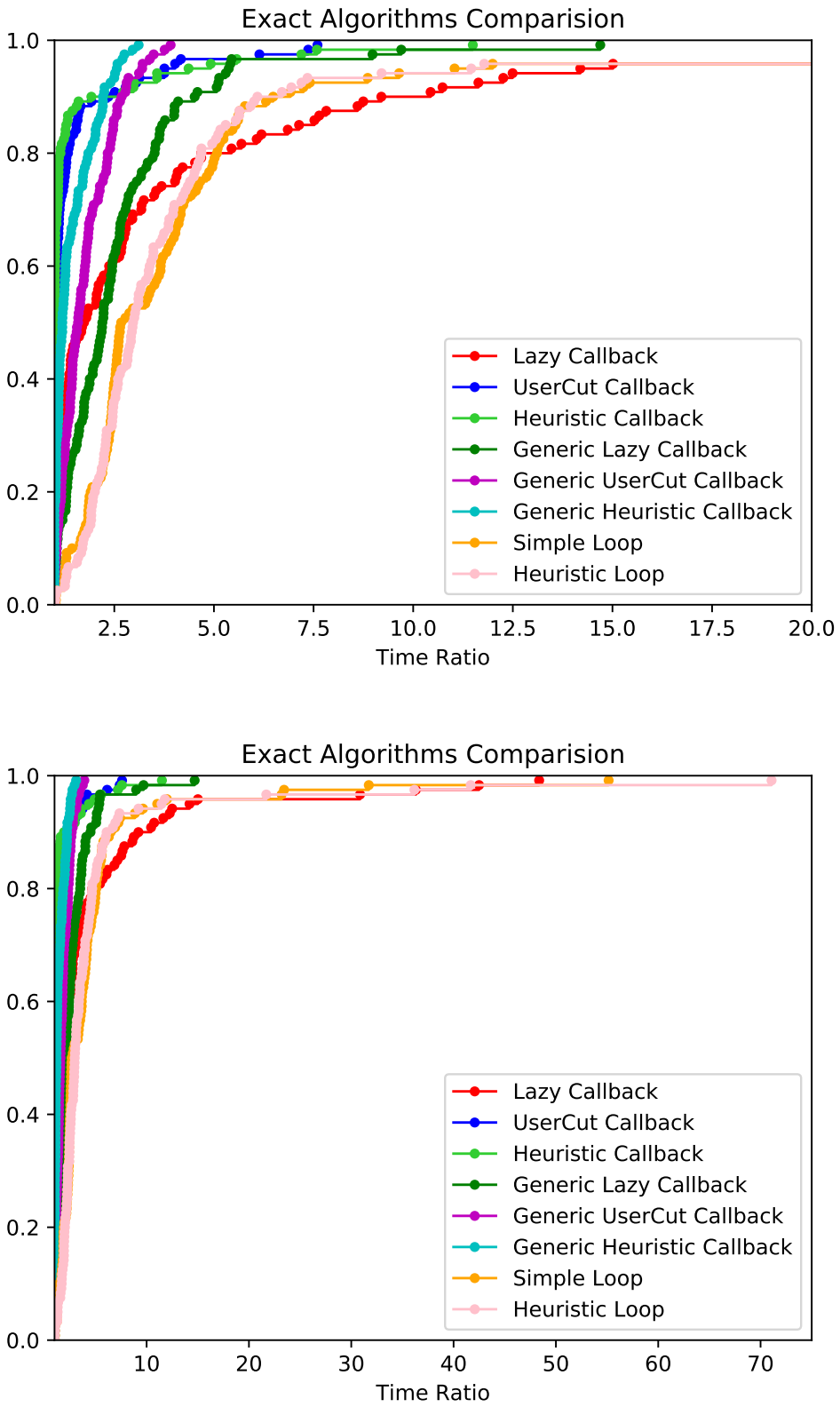


Figure 3.9: On top: detailed view of the performance profile of Exact Algorithms. On the bottom: full view of the performance profile of Exact Algorithms.

Chapter 4

Matheuristics

On this chapter we will illustrate two heuristic solutions for the TSP problem that are based on CPLEX.

4.1 Hard fixing

When we have to resolve a large instance of TSP, CPLEX can require a large amount of time. Moreover in some practical applications we may have to resolve the problem in a certain amount of time that we cannot exceed. So for this reason we can set a time limit for CPLEX, that means that the solver will return the best solution found within that limit. The hard fixing algorithm starts from this point. Let's set a time limit (10 minutes for example) and consider the solution obtained. Of course this solution with high probability is not the optimum solution, but it may contains some edges that belong to it. So the idea is to fix some edges, and pass the new model to CPLEX. In order to do this we can simply work on the bounds of the variables: if we set $1 \leq x(i,j) \leq 1$ the corresponding edge is fixed on the solution. The new model of course is simpler, because we have a smaller amount of variables, so when we resolve it with CPLEX (again with a fixed time limit) it may find a better solution. This procedure can also be iterate until a final time limit (like 1 hour) is reached: each time we release the variables that were fixed on the previous iteration (just reset the bounds to $0 \leq x(i,j) \leq 1$) and we choose a new set of edges to fix. Note that when the variables are fixed, due to the fact that the instance is smaller, CPLEX may resolve that model to its optimum before the time limit exceed. Of course the choice of which edge to fix and their number is quite relevant. In general a good idea is to set a fixing percentage and randomly choose the variables to fix. We can also decide to variate the percentage over the iterations, for example starting with an high value (like 90%) and then decrease it until zero. This mean that on the initial iterations the problem to resolve is very small so we may get a significant improve, while on the last iterations we have a big problem but lot of freedom to refine the solution. Of course this is only an example and other strategies can be implemented.

Our implementation of the Hard Fixing is presented in Algorithm 8. We used two

variables to manage the time limit of the algorithm. The variable *remaining_time* keeps track of the time left to the algorithm to compute the solution. At the beginning the variable assumes the value of the original time limit given by the user (line 2). The variable *time_x_cycle* is the inner time limit and it is initially fixed to *time_limit/5* (line 3). Since with the inner time limit it is very hard that an initial good solution will be computed, we provide to CPLEX a heuristic solution builded with the algorithm GRASP and refined with the 2-OPT algorithm (lines 4-6). The algorithm then proceeds solving the problem within a while loop (line 7). At each iteration, we initially set the inner time limit as CPLEX environment parameter (lines 8 - 11). Then the solution, within the time limit, is computed (line 13) with the time needed to compute it (lines 12 and 14). At this point, the variable *remaining_time* is updated subtracting from it the time taken by the solver (line 15).
The variable *percentage*, initially set to 90% (line 1), is updated only if after two consecutive iterations the solution found doesn't improve of at least the 10%. The update consists in subtracting 15% from the previous percentage of fixed variables (line 18). Then the algorithm proceeds by resetting the lower bound of all variables to 0.0 (line 19), randomly selecting the indices of variables to be fixed and setting the lower bound of them to 1.0 (line 20). This sequence is repeated as long as the user-set time limit is reached.

Algorithm 8 Hard Fixing

Input: $G = (V, E), c : E \to \mathbb{R}$
Output: A solution for the TSP problem

1: *percentage* ← *0.9*
2: *remaining_time* ← *timelimit*
3: *time_x_cycle* ← *timelimit / 5*
4: model ← ∗ **Initialize variables and objective function** ∗
5: first_solution ← ∗ **Heuristic solution with GRASP + 2-OPT** ∗
6: model ← CPXaddmipstarts(first_solution)
7: **while** (*remaining_time* > 0) **do**
8: **if** (*remaining_time* > *time_x_cycle*) **then**
9: *model* ← CPX_PARAM_TILIM(*time_x_cycle*)
10: **else**
11: *model* ← CPX_PARAM_TILIM(*remaining_time*)
12: *t1* ← *second()*
13: $\boldsymbol{x}$ ← solution(model)
14: *t2* ← *second()*
15: *remaining_time* ← *remaining_time* − *(t2* − *t1)*
16: **if** (*remaining_time* <= 0) **then**
17: **break**
18: ∗ **If the solution doesn't improve (+10%) twice in succession then** *percentage* −= 0.15. ∗
19: ∗ **Restore the default bounds for each variable** ∗

20: ∗ **Hard fixing of each variable according to *percentage*** ∗
21: **return *x***

4.2 Local branching

The strategy of this algorithm [10] is similar to the one of hard fixing: each iteration fix some edges in order to give to CPLEX a simpler version of the problem. However, rather then randomly choose the variable to fix, we let CPLEX decide which edges to maintain. We can achieve this with the introduction of a new constraint: let's consider an initial solution H provided by CPLEX within a time limit, let x_e^H be the variables equals to one on this solution, we want the sum of these variables to be greater or equals than a percentage of the total number of edges. More formally:

$$\sum_{e \in E,\, x_e^H = 1} x_e \geq \alpha n$$

where α is the selected percentage and n is the size of the instance. In other word with this constraint we are telling CPLEX that we want a percentage of variables that are in the initial solution to be also in the final solution. At this point we can give the new model to CPLEX that will resolve it within the time limit, automatically deciding which variables to maintain. Of course this is not a valid constraint for the problem, cause we are not sure that the variables equals to one on a solution belong to the optimum.
Again we can iterate the procedure, each time removing the old constrain and generating a new one, based on the new solution, until a final time limit is reached. The fact that the selection of the variables to fix is not random, means that the CPLEX solution is deterministic, so it is not wise to execute two iterations with the same fixing percentage, cause we may resolve two times the same problem obtaining the same solution within the time limit. Finally, another shrewdness that is required, is that this algorithm works only if the fixing percentage is very high, so iterations with percentage of 50-60 or less should be avoided.

Our implementation of the Local branching is presented in Algorithm 9. The implementation is pretty similar to Algorithm 8, for this reason we only emphasise the differences between the two algorithms. At each iteration of the while loop (line 7), the variables of the solution are saved and a new local branching constraint is added to the model, removing the previous one. The known term of the constraint, that is the variable *percentage* in the algorithm, is initially set to "number of nodes ∗ 0.95" (line 1) and is updated at each iteration, but in this case subtracting from the variable *percentage* the value "number of nodes ∗ 0.05" (lines 18 - 21). As in algorithm 8, the sequence is repeated as long as the user-set time limit is reached.

Algorithm 9 Local branching

Input: $G = (V, E), c : E \to \mathbb{R}$
Output: A solution for the TSP problem

percentage $\leftarrow |V| * 0.95$
remaining_time $\leftarrow$ *timelimit*
time_x_cycle $\leftarrow$ *timelimit / 5*
model $\leftarrow$ $*$ **Initialize variables and objective function** $*$
first_solution $\leftarrow$ $*$ **Heuristic solution with GRASP + 2-OPT** $*$
model $\leftarrow$ CPXaddmipstarts(first_solution)
while (*remaining_time* > 0) **do**
    if (*remaining_time* $>$ *time_x_cycle*) **then**
        model $\leftarrow$ CPX_PARAM_TILIM(*time_x_cycle*)
    else
        model $\leftarrow$ CPX_PARAM_TILIM(*remaining_time*)
    t1 $\leftarrow$ *second()*
    $\boldsymbol{x} \leftarrow$ optimal_solution(model)
    t2 $\leftarrow$ *second()*
    remaining_time $\leftarrow$ *remaining_time* $-$ *(t2 − t1)*
    if (*remaining_time* $<= 0$) **then**
        break
    $*$ **If the solution doesn't improve (+10%) twice in succession then** *percentage* $-=$ $(0.05 * percentage)$. $*$
    $*$ **Save the indices of the solution variables.** $*$
    $*$ **If is not the first cycle remove the last row of the model.** $*$
    $*$ **Add the new local branching constraint with** $\alpha n = percentage$. $*$
return $\boldsymbol{x}$

4.3 Final comparison between Matheuristics

The instances used to compare heuristic algorithms are the following:

- d657.tsp
- d1291.tsp
- fl417.tsp
- fl1400.tsp
- fl1577.tsp
- nrw1379.tsp
- p654.tsp
- pcb1173.tsp
- pcb3038.tsp
- pr1002.tsp
- pr2392.tsp
- rl1304.tsp
- rl1323.tsp
- rl1889.tsp
- u724.tsp
- u1060.tsp
- u2152.tsp
- u2319.tsp
- vm1084.tsp
- vm1748.tsp

For each problem i and algorithm j, we compare the Gap computed as

$$gap_{ij}\ (\%) = \frac{objFunc_j(i) - objFunc_{opt}(i)}{objFunc_j(i)} * 100, \tag{4.1}$$

where $objFunc_j(i)$ is the objective function value computed by the algorithm j for the problem i. The method used to compute a solution within the time limit is the *Generic Heuristic Callback*, which is the best exact algorithm we implemented (see Section 3.2.5). The time limit has been set to 30 minutes. In figure 4.1 you can see the performance profile of the Matheuristic algorithms. The Hard Fixing outperforms the Local Branching in most of instances. Even if worse, the Local Branching algorithm achieves a gap within twice the best, so it is considered by us a valid competitor. From our experiments we observed that Matheuristics achieve the optimum in several instances, even in a problem with 2319 nodes. We also noticed that Matheuristic algorithms work very well even with big instances, achieving an objective function value within 5% from optimum. All the numerical results are reported in Section 7.3. The reason for their good performance may be due to the powerful *Generic Heuristic Callback* algorithm which, thanks to Concorde, generates numerous cuts on fractional solutions, and also provides heuristic solutions to the solver thanks to a quick and effective algorithm (see Section 3.2.3). Another reason is that our Matheuristic algorithms exploit all time available by fixing a decreasing number of variables.

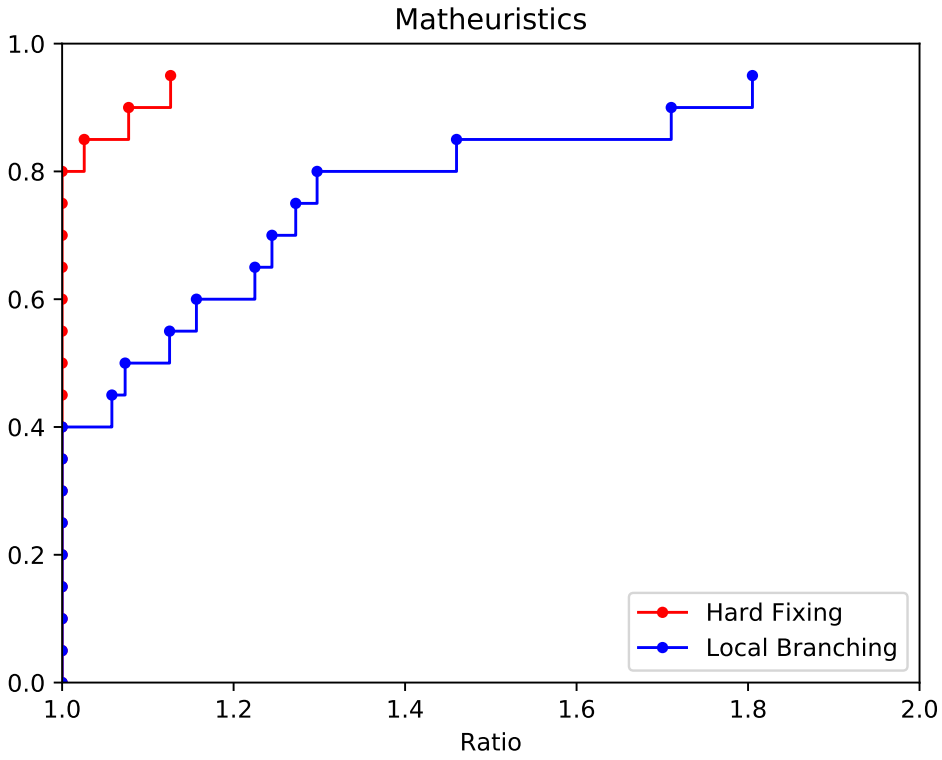


Figure 4.1: Performance profile of Matheuristics

Chapter 5

Stand-alone heuristics

On this chapter we will illustrate some stand-alone heuristics. These algorithms are so called because they provide an heuristic solution without relying on CPLEX or any other solver. The literature in this case is huge: there are plenty of different approaches and each of them can have multiple variations. On this report we will cover some of the major ideas.
For all the experiments to collect data for the performance profiles and the comparison between different algorithms we used a dataset of 20 instances, with a size between 400 and 3000 points. A detailed list of all the instances used and their dimension can be found on the tables of the Section 7.4.

5.1 Heuristic solution builder

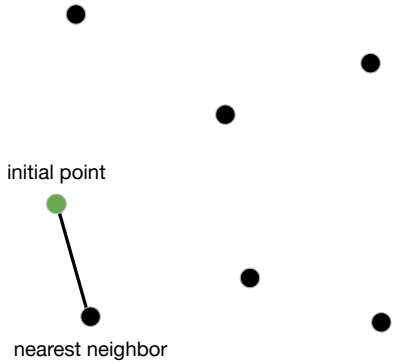
On this section we will illustrate some heuristics algorithm to generate in a rapid way a solution for the TSP problem. Of course the results provided by these algorithms are admissible solutions but not the optimum solution.

5.1.1 Nearest neighborhood

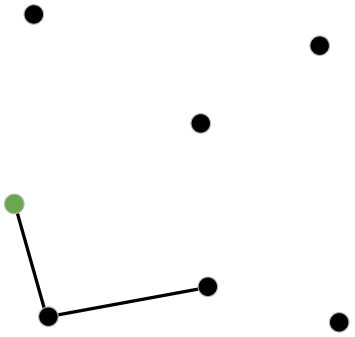
Nearest neighborhood is a greedy algorithm [1] that works in the following way: start from a random point that is considered the first visited node. Then pick as next node the nearest to the last visited (greedy choice) and iterate until all the nodes are visited. This procedure can be repeated until a time limit is reached, each time starting from a different random point, keeping in memory the best solution and eventually update it if a new one is found. The algorithm is pretty simple but provide an admissible solution (that of course is not the best one and in the very most of the cases is quite far from the optimum) in a short time.

On our implementation we decided to trade space for time. Since the operation to compute the nearest point is repeated several time, we decide to store for each point

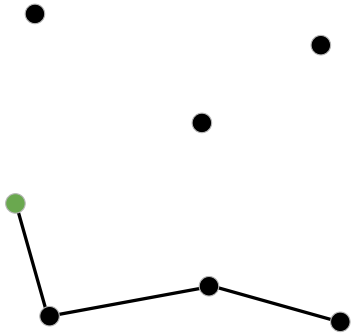
[1] algorithm that always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution [11].



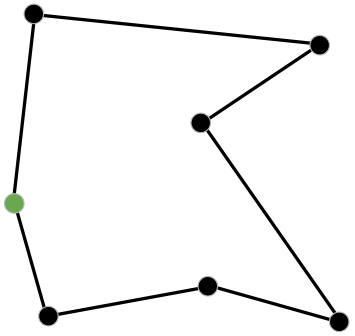
(a) Initial step



(b) Second step



(c) Third step



(d) Final result

Figure 5.1: Execution of nearest neighbor algorithm

the list of all others points sorted by their distances from that one. This require $O(n^2)$ additional space. Considering the time, instead, to sort the points we implemented a simple *insertion sort*, that is $O(n^2)$, so in total we need an initial $O(n^3)$ (This part could be further improved with a better sort algorithm like *merge sort* that is $O(n \cdot log(n))$, but since the initial time to order the nodes on our experiments was negligible, we decided to use insertion sort that was easier to implement). So, by paying this initial cost we were able to speed up each iteration of nearest neighborhood, in fact each time we needed to find the nearest point we just had to pick the first available one from the ordered list. Of course in the worst case this is $O(n)$, because it can happen that the first elements of the ordered list have already been visited, so I have to run across most of the list until I find an available point. However, what we saw in practice is that, except when there are left few nodes to visit, the first available element is always on the initial positions of the ordered list so the operation to find the nearest neighbor became $\Theta(1)$ in most of the cases (for

example, we tried on multiple instances between 100 and 500 points and the average number of access to the ordered list to find the nearest point was between 3 and 6). Of course this technique is efficient only when we have to generate lots of solutions (like in the genetic algorithm) so we implemented a variant without the initial list, to use when we need only few solutions.

The major problem of this technique is that visiting always the next closest point leads to the creation of lots of intersection between edges, that of course are inefficient. A possible solution to improve this initial result will be shown on the next section dedicated to the refining algorithms.
Another problem is that this algorithm is deterministic, so this mean that if it is run twice starting from the same initial point, it produce the same solution. The consequence of this is that the number of different circuit that it can explore is limited to the total number of nodes.

5.1.2 GRASP

GRASP is a variation of nearest neighborhood that introduce a random component. On each iteration, rather than picking the nearest point to the last visited, select the three closest points and choose one of them at random as next visited vertex. (Or alternatively assign a different probability to each of the three points, for example 0.6 to the closest point and 0.3 and 0.1 respectively to the second and the third).

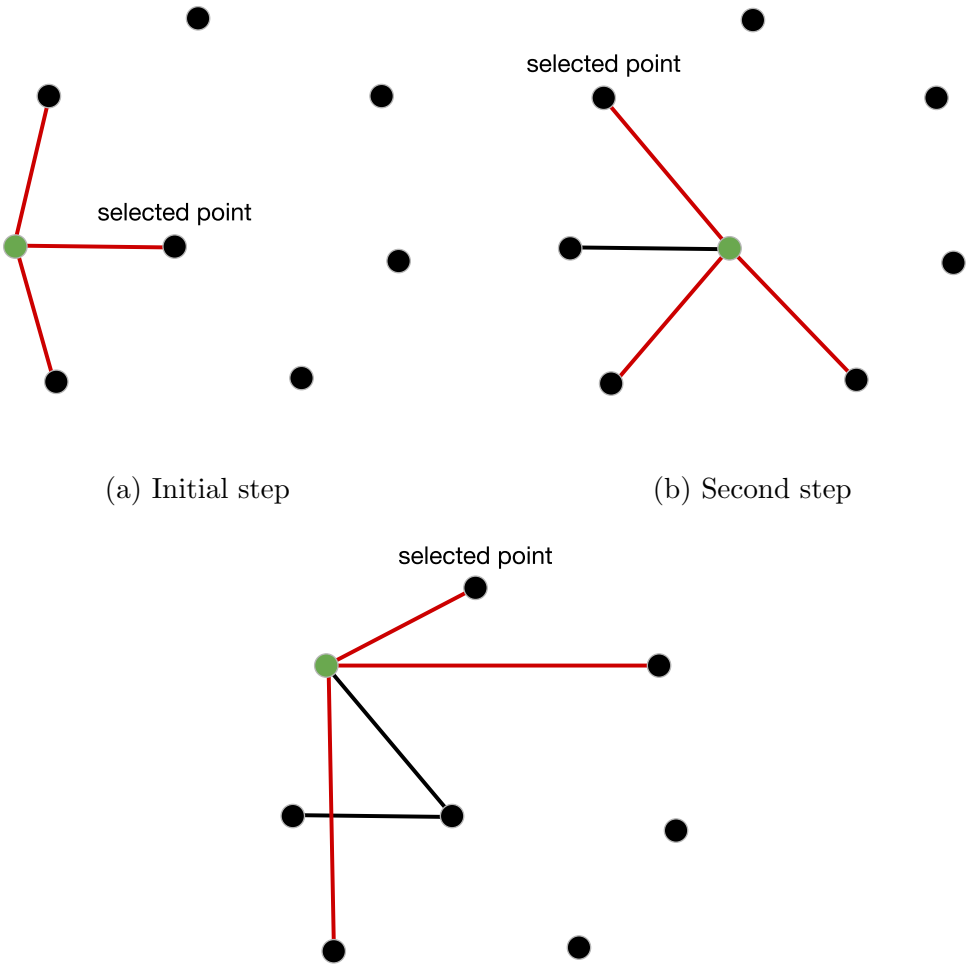
Algorithm 10 GRASP

Input: Instance I of the TSP
Output: Admissible solution for the instance

```
1:  cost(bestSolution) ← ∞
2:  while ! termination condition do
3:      S ← empty list
4:      notVisitedList ← all vertices
5:      currentVertex ← pick 1 random vertex from I
6:      notVisitedList ← remove(currentVertex)
7:      S ← add(currentVertex)
8:      for i = 1, i < N, i++ do
9:          v1, v2, v3 ← pick 3 closest and not visited vertex from currentVertex
10:         nextVertex ← choose at random one between v1, v2, v3
11:         notVisitedList ← remove(nextVertex)
12:         S ← add(nextVertex)
13:         currentVertex ← nextVertex
14:     if cost(S) < cost(bestSolution) then
15:         bestSolution ← S
16: return S
```

Our implementation was almost the same of nearest neighborhood, with the construction for each point of the initial list of all other points sorted by their distance

from that point, with the only difference that first three available points are picked from the list, rather than only one, and the next visited point is chosen at random from them. The introduction of the random component means that the probability to repeat the same sequence starting from the same point is very minimal, so GRASP can explore lots of different circuit resolving the problem of determinism of nearest neighborhood.



(a) Initial step

(b) Second step

(c) Third step

Figure 5.2: Execution of 3 steps of GRASP. Each time the 3 closest points are considered and one is picked at random.

5.1.3 Insertion heuristic

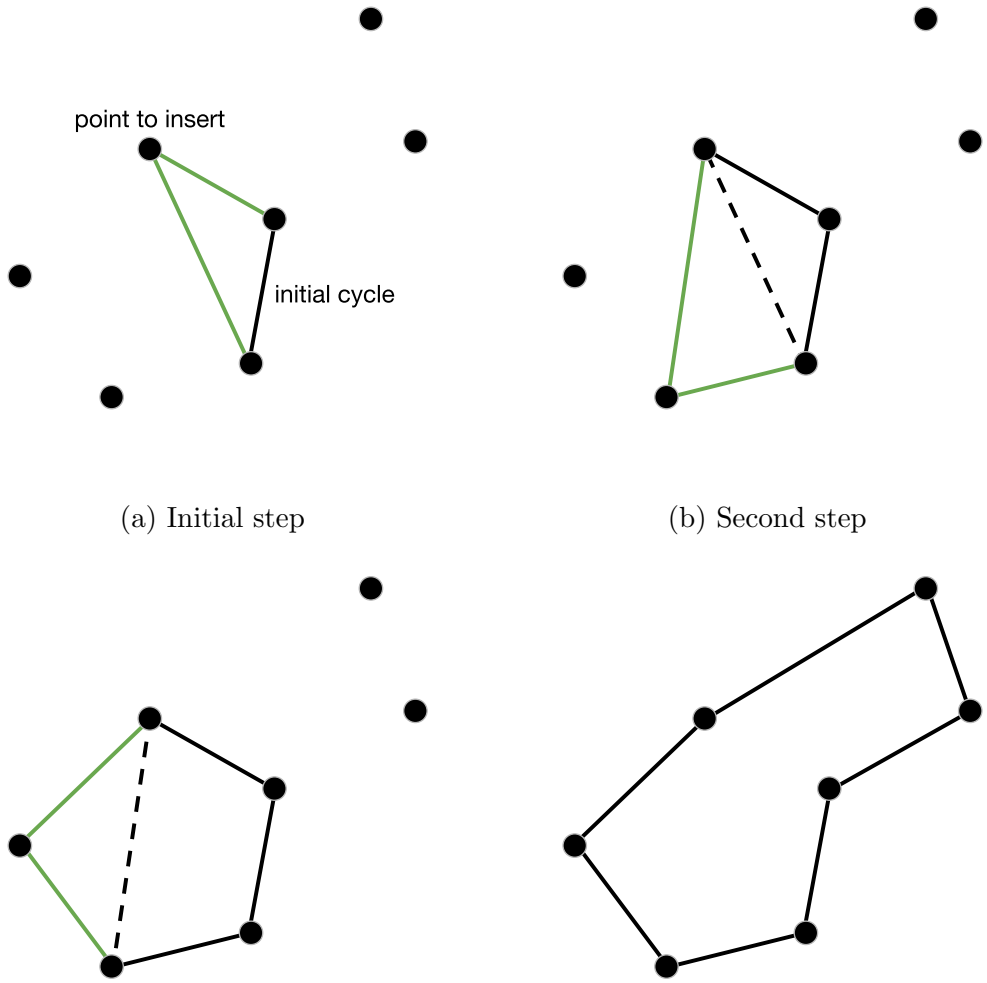
This algorithm starts from a simple circuit made with only two points. At each iteration, for each of the remaining vertices to insert, compute the minimal extra

mileage. The extra mileage is the cost to insert a point on the circuit. So, for example, if we want to insert the vertex z between vertex x and vertex y, the extra mileage is: $c_{xz} + c_{zy} - c_{xy}$, where c_{ij} is the cost of the edge (i, j). The fact that for each point we search the minimal extra mileage means that we are looking to the best position where insert that point. Insert vertex that has the best extra mileage into the partial circuit and repeat the procedure until all the points are inserted (Figure 5.3). The algorithm than can be executed several time to explore new solutions, starting every time from different initial points and updating the incumbent if necessary.
The cost to compute the best extra mileage is $O(n^2)$, so this mean that the entire algorithm is $O(n^3)$.
It is also possible to add the GRASP rule, so rather then selecting the point with the best extra mileage, find the three smallest extra mileages and choose at random between them, in this way the algorithm doesn't converge to the same solution starting from the same initial circuit.

Algorithm 11 Insertion heuristic

```
    Input: Instance I of the TSP
    Output: Admissible solution for the instance
 1: cost(bestSolution) ← ∞
 2: while ! termination condition do
 3:     S ← empty list
 4:     S ← add 2 initial vertices
 5:     while ! all point are inserted do
 6:         bestExtraMileage ← ∞
 7:         for each point p to insert do
 8:             minExtraMileage ← ∞
 9:             for each edge e ∈ S do
10:                 extraMileage ← computeExtraMileage(p, e)
11:                 if extraMileage < minExtraMileage then
12:                     minExtraMileage ← extraMileage
13:             if minExtraMileage < bestExtraMileage then
14:                 bestExtraMileage ← minExtraMileage
15:         S ← insert point with bestExtraMileage in the best poisiton
16:     if cost(S) < cost(bestSolution) then
17:         bestSolution ← S
18: return bestSolution
```

On our implementation to select the initial circuit we decided to pick at random one point and then to select the furthest point from that one. (Of course this is not the only solution: other possibility are, for example, to pick a vertex and its closest one, or to choose two points completely at random.) In addition, at the beginning of the algorithm, we built a matrix where the element i j contains the distance be-



(a) Initial step

(b) Second step

(c) Third step

(d) Final result

Figure 5.3: Execution of insertion heuristic algorithm

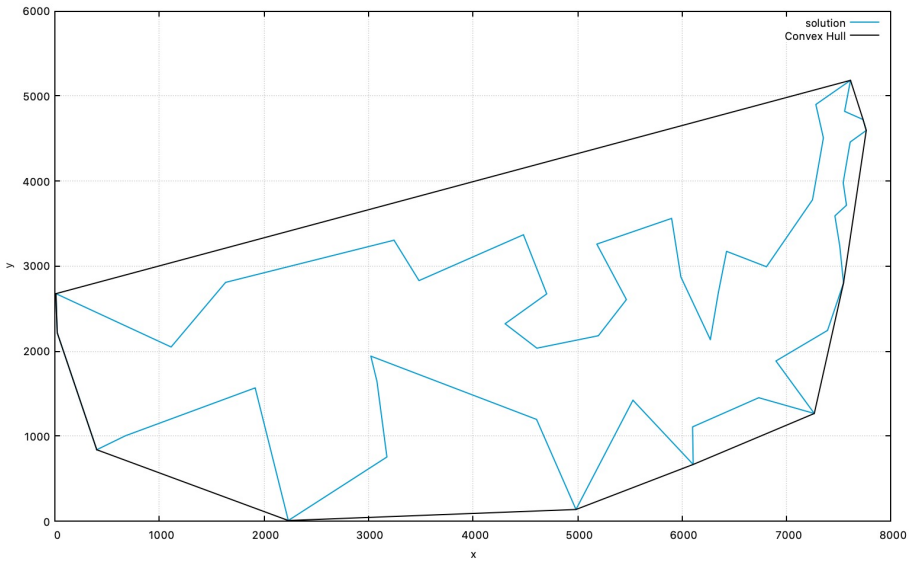
tween vertices i and j (So in total we payed an additional $O(n^2)$ in space). We made this choice because on this algorithm distances between points are computed several times and in addition the same distance can be calculated more than once.

5.1.4 Insertion with convex hull

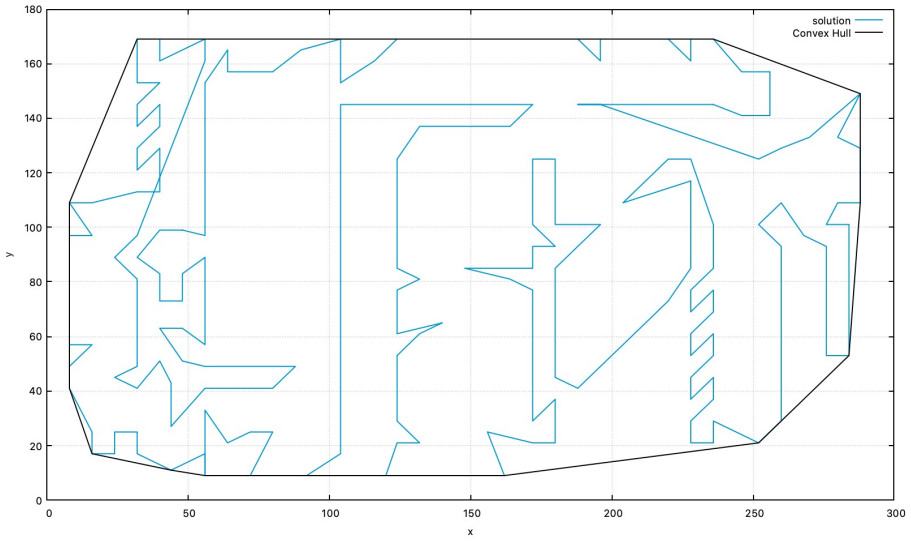
This is a variation of the insertion algorithm. It works in the same way but the initialization is different: rather than starting from an initial cycle made of two random points, compute the convex hull [2] of the instance. Then iterate and insert

[2]In geometry, the convex hull or convex envelope or convex closure of a shape is the smallest convex set that contains it, `https://en.wikipedia.org/wiki/Convex_hull`.

all the remaining points, that are all inside the convex hull, in the same way of Insertion heuristic (Figure 5.4).
On our implementation, to find the initial convex hull, we used the Graham Scan algorithm, freely provided by the website www.geeksforgeeks.org. This algorithm is $O(n \cdot log(n))$ and it is performed only once at the beginning so it doesn't impact the overall execution time for the insertion that is $O(n^3)$ as explained on the previous subsection. For more information about the Graham Scan consult `https://www.geeksforgeeks.org/convex-hull-set-2-graham-scan/`.



(a) problem att48



(b) problem a280

Figure 5.4: Execution of the insertion with convex hull on 2 different problems. It is possible to see the initial convex hull and the final result when all the points inside it are inserted

5.1.5 Final comparison between solution builder

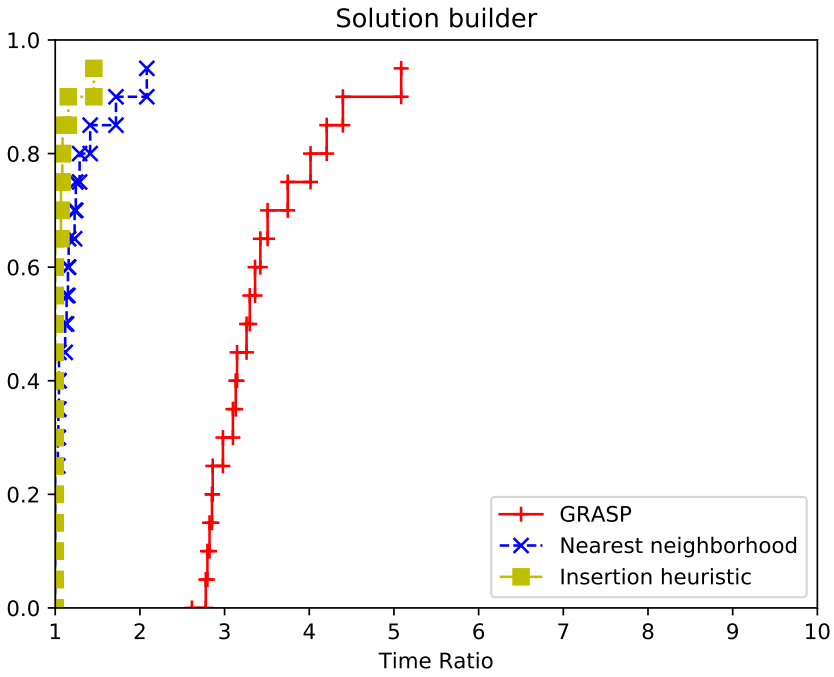


Figure 5.5: Performance profile of the solution builders

On Figure 5.5 the final confrontation between the heuristic solution builders. To obtain these result we test the algorithms on the dataset of 20 instances with a time limit of 5 minutes.

5.2 Refining algorithms

Refining algorithm are so called because they start from an admissible solution that could by provided by any other algorithm and they try to improve it with a series of iterations.

5.2.1 TWO-OPT

The first algorithm we present is TWO-OPT (or 2-OPT). To explain this technique, let's consider two edges, the first delimited by vertices a and b and with cost c_{ab}, the second delimited by c and d and with cost c_{cd} (Both edges belong to an admissible solution provided by any algorithm). Now let's remove these edges and reconnect the vertices in a different way: as result we obtain the edges (a, c) and (b, d) with cost respectively c_{ac} and c_{bd}. Now lets compute:

$$\Delta = c_{ab} + c_{cd} - (c_{ac} + c_{bd})$$

that is the cost of the initial removed edges minus the cost of the new edges. If $\Delta > 0$, it means that the new edges are better than the older and the swap is worth because it reduce (improve) the objective function. Of course the new solution produced is still admissible (Figure 5.6).

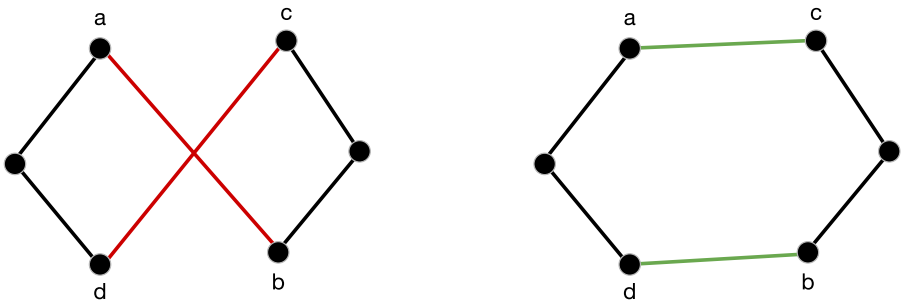


Figure 5.6: The 2-OPT move, edges in red are removed and replaced by green edges.

2-OPT works in this way: start from an admissible solution, consider all the possible couple of edges (that are in total $O(n^2)$) and for each of them compute the Δ (That can be calculated in $\Theta(1)$). Then pick the largest Δ, remove the relative edges and reconnect the vertices as previously explained. After this operation, one of the two parts of the cycle that has been divided by the swap, need to be retraced in order to invert the visit sequence of the vertices and reconnect it to the new edge (This is $O(n)$, but since it is done after a $O(n^2)$ doesn't change the overall cost of the move). Repeat the procedure until all the Δ are negative, or a fixed time limit is reached.

On our implementation we followed the procedure described above, with the addition of a matrix to store all the distances between points, to avoid to recompute

Algorithm 12 2-OPT

Input: Admissible solution S
Output: Best solution found

```
1: while ! termination condition do
2:     best∆ ← ∞
3:     for each couples of edge v_ab, v_cd ∈ S do
4:         ∆ ← c_ab + c_cd − (c_ac + c_bd)
5:         if ∆ < Best∆ then
6:             best∆ ← ∆
7:     if best∆ > 0 then
8:         S ← perform move with best∆
9:     else
10:        return S
11: return S
```

them more than once.
This algorithm is very good to remove edge intersections, like the ones produced by nearest neighborhood (or GRASP). Here in Table 5.1 some examples.

Instance	**GRASP** (5 sec)	**2-OPT** (5 sec)	**Improve**
att48	41806	35522	15%
eil101	879	691	21%
ch130	9801	6347	35%
ch150	10477	7278	30%
a280	4546	2886	36%
lin318	76940	46235	39%
att532	155010	97487	37%
pr1002	492311	285078	42%

Table 5.1: Values of the objective functions after running on each instance GRASP algorithm for 5 seconds and then 2-OPT for other 5 seconds.

These results were obtained after running GRASP for five seconds and then refining the solution with 2-OPT for other five seconds. Of course this dataset is very small but it gives the idea of the power of this technique, we have, in fact, a significant improve of the objective function in all the cases. To notice that the improve increase with large problem.
It is possible to visualize the action of 2-OPT by plotting the solution before and after the refining like in Figure 5.7.

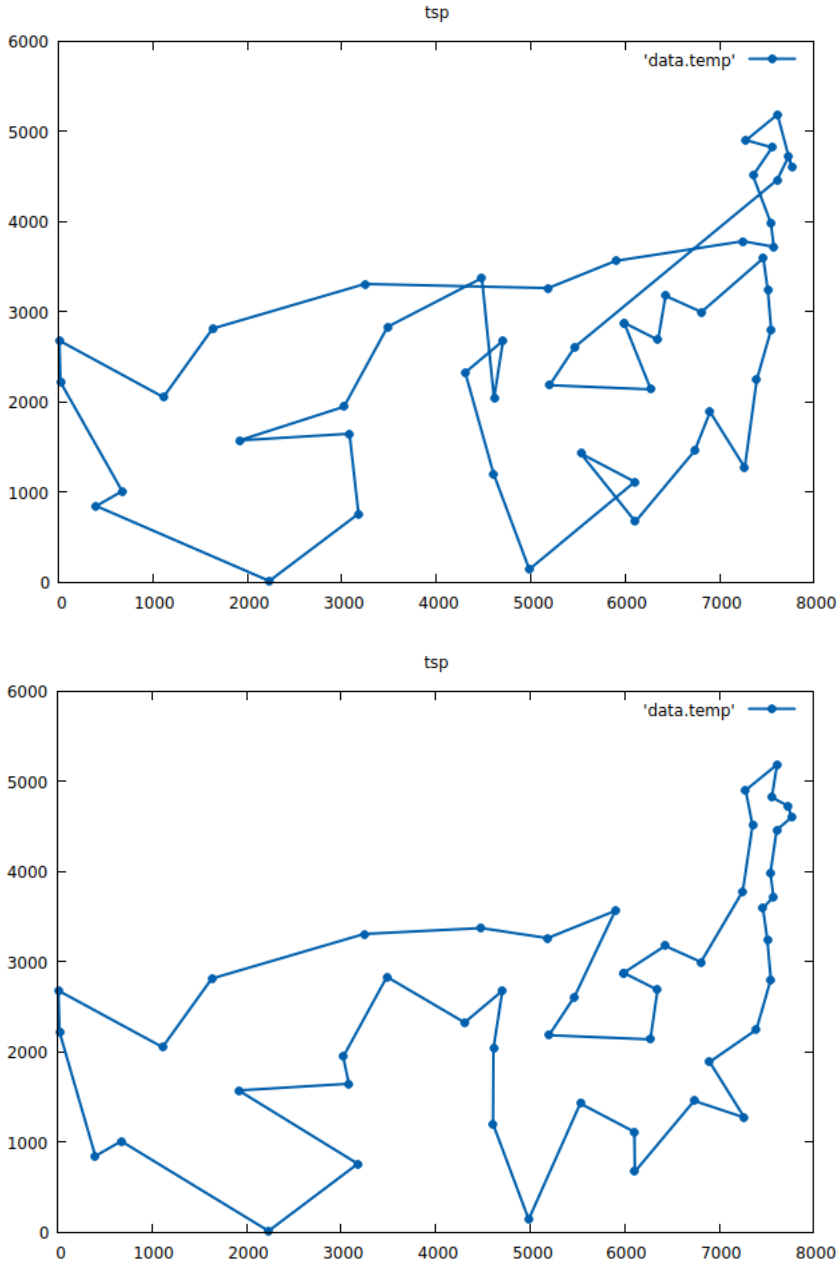


Figure 5.7: Solution of 5 sec of GRASP before (top image) and after (bottom image) a refining of 5 sec

5.2.2 THREE-OPT

3-OPT is a variation of 2-OPT that consider 3 edges for the swap rather than only two. This time there are multiple way to reconnect the vertices of the 3 removed edges, in particular for a triad there are 7 possible moves (Figure 5.8). Moreover, we have to consider all the possible triad to find the best one and this is $O(n^3)$. So in general 3-OPT is more complicated to implement and due to the fact that it is

cubic, it doesn't work well with big problems, for example with instances with more than 10000 points.

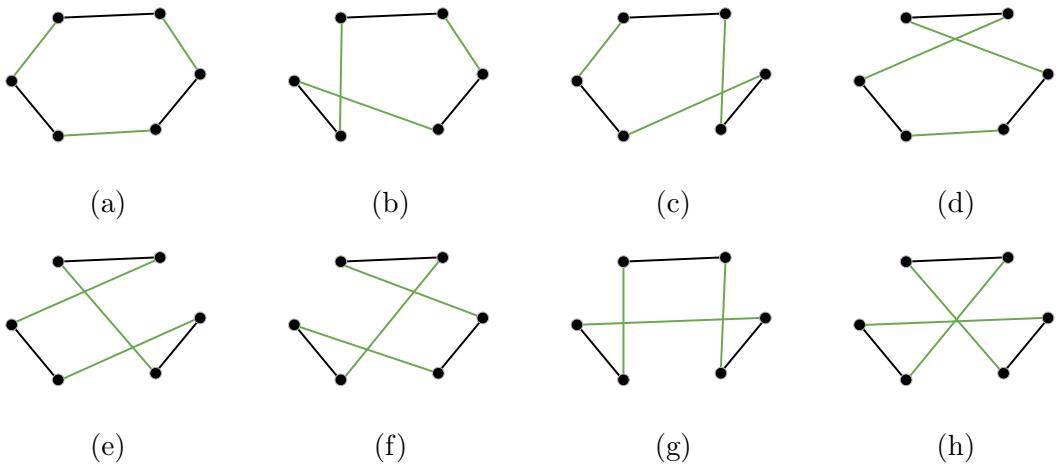


Figure 5.8: All the possible 3-OPT moves given a triad of edges.

We also repeated the same experiment we did for the 2-OPT, so we run on different problems GRASP for 5 seconds and then 3-OPT on the result for other 5 second. The results are shown on Table 5.2. Again the dataset is small because the only goal was to show the power of this refinement algorithm and not to compare it with other techniques. Observing the table we can notice that the improve for the smaller problems is almost the same, or sometimes better than the one with 2-OPT, but more importantly we can notice that the algorithm scales bad with big problems due to the fact that it's cubic. In fact 5 seconds is a relatively small amount of time and in this short period the number of 3-OPT moves that the algorithm is able to perform is small so the improve is poor.

Instance	**GRASP** (5 sec)	**3-OPT** (5 sec)	**Improve**
att48	42086	33588	20%
eil101	875	665	25%
ch130	9894	6373	35%
ch150	10503	6828	35%
a280	4588	3247	29%
lin318	75526	56496	25%
att532	153954	144295	6%
pr1002	487157	481381	1%

Table 5.2: Values of the objective functions after running on each instance GRASP algorithm for 5 seconds and then 3-OPT for other 5 seconds.

5.3 Metaheuristics

A metaheuristic is a higher-level procedure or heuristic designed to find, generate, or select a heuristic (partial search algorithm) that may provide a sufficiently good solution to an optimization problem, especially with incomplete or imperfect information or limited computation capacity [12].

5.3.1 Multi-start

After performing a refinement technique, like 2-OPT, if the time limit is not reached, the algorithm stop on a local minima of the objective function. Of course at this point we want to escape from this local minima and search for a better one that hopefully is also the optimum. A possible solution is the multi-start approach, so each time we restart from the beginning, providing another initial solution and refining it. On each iteration, if the algorithm that provide the initial solution is well randomized (like GRASP) we will stop on a different local minima. So we can keep track of the best solution and return it when the time limit for this procedure is reached. The main problem of this technique is that it is not very efficient because each time we restart all the information we obtained on the previous local minima is unused. More powerful technique will be shown on the following subsections.

5.3.2 Variable neighborhood search

Variable neighborhood search (VNS) provide a simple yet powerful way to escape the local minima. Let's assume that we have completed the refinement using 2-OPT. At this point we can perform a random move with a bigger research radius, like a 3-OPT or a 5-OPT for example. Since this move is completely random, it will make the solution worse but it will also provide the "kick" that we need to escape the local minima. After this we can restart with 2-OPT until a new minima is found (that could also be the optimum).
In synthesis, VNS, works with two phases that alternate continuously: an intensification phase (2-OPT) where the algorithm move toward a minima and a diversification phase where VNS perform a k-OPT random move to exit from the minima.
On our implementation we use the technique showed in Figure 5.10. We select the edges to remove (that in our example are three) and then we reconnect the vertices in a specific way, so that the reversal of some edges to reconstruct the cycle (the one described in 5.2.1) is never required. In particular, because we can assign to the cycle a direction of travel, for each edge that we want to remove we can distinguish an initial vertex (the ones in red) and a final vertex (the ones in green). The idea is to always reconnect an initial vertex to a final vertex so the direction of travel is preserved. So for example we can start from the initial vertex of the first edge to remove and connect it to the final vertex of the second edge to remove. Then we pick the initial vertex of the second edge and we connect it to the final vertex of the third and so on, until the cycle is reconstructed. Of course this is valid for k

Algorithm 13 VNS

Input: Admissible solution S
Output: Best solution found

```
1:  bestSolution ← S
2:  while ! termination condition do
3:      bestMove ← find best 2-OPT move of S
4:      if Δ(bestMove) > 0 then
5:          S ← perform bestMove
6:      else
7:          S ← perform random k-OPT move
8:      if cost(S) < cost(bestSolution) then
9:          bestSolution ← S
10: return bestSolution
```

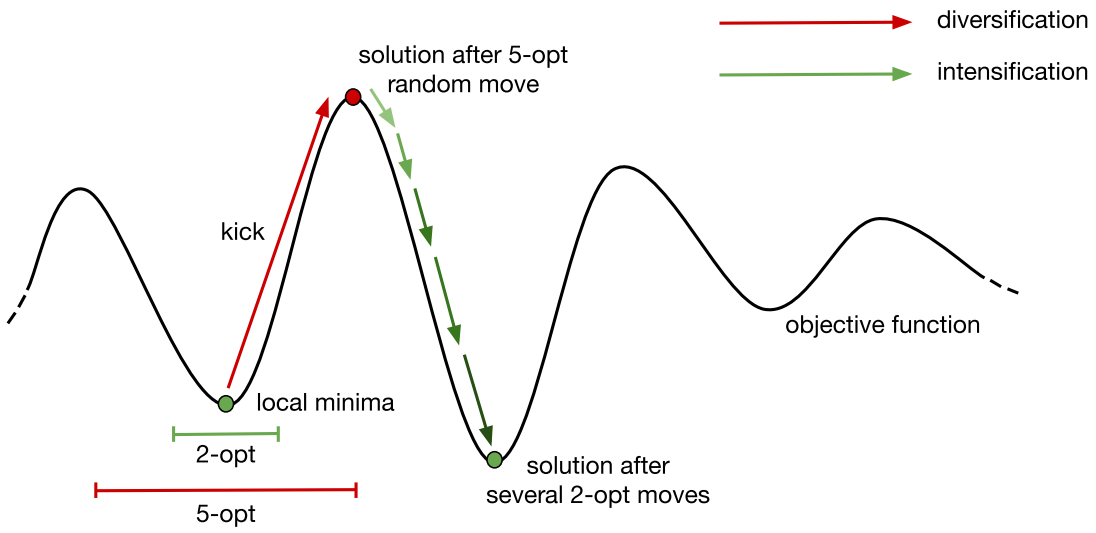


Figure 5.9: Representation of an objective function during VNS and the "kick" of a 5-OPT random move

edges, so on our implementation we can perform a k-opt random move, where k is establish at the beginning of the execution.

5.3.3 Tabu search

Created by Fred W. Glover during the 80s, Tabu search provide an alternative procedure to VNS to escape from local minima. Let's assume that we have performed a refinement and we have reached a local minima, let's call this solution x_k. At this point, every 2-OPT move will make the objective function worse. So we choose and perform the least pejorative move, obtaining the solution x_{k+1}. Now, if we make

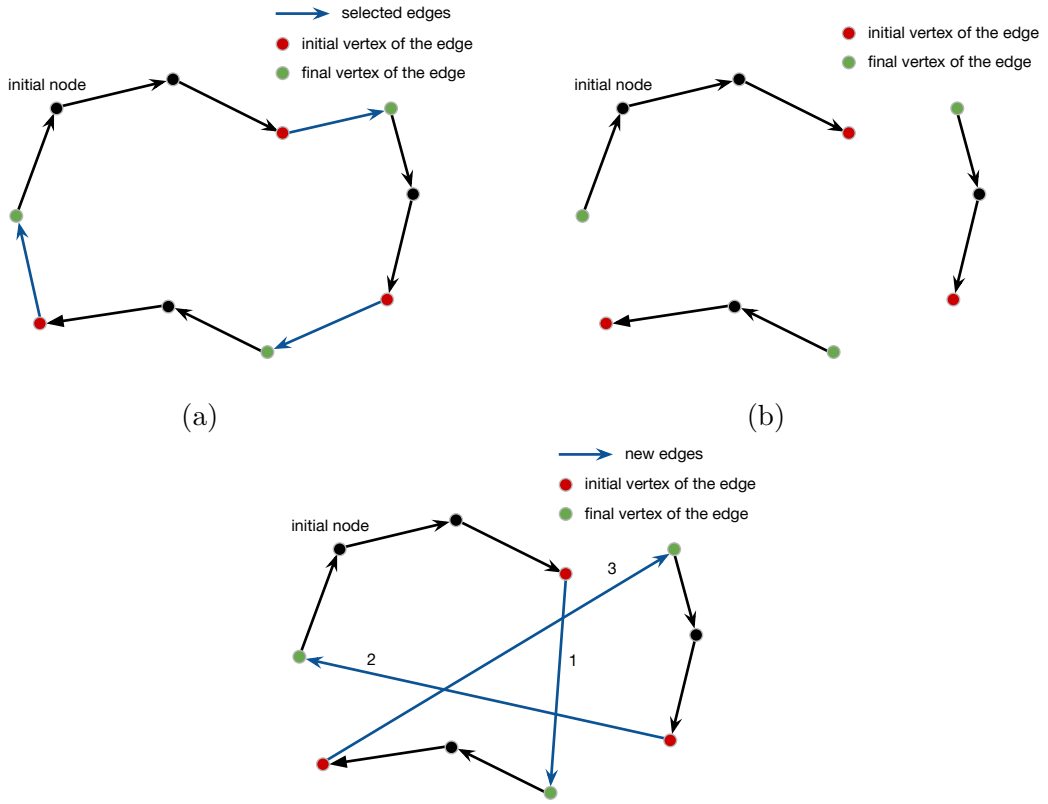


Figure 5.10: Passages for a k-OPT random move (In this case a 3-OPT move).

another move following the 2-OPT rule, the algorithm will return to the solution x_k by simply swapping again the edges used on the last step, cause this is the only move that improve the objective function. To avoid this, the idea is to insert these edges on a tabu list that is a list of edges that cannot be used for a 2-OPT move. This means that the move from x_k to x_{k+1} became forbidden. So we repeat this last step, performing the least pejorative move and each time inserting the edges used in the tabu list, until we reach a point where the objective function will return to improve. Of course we don't insert edges into the tabu list if the move is not pejorative.
To describe the algorithm we asserted that the tabu list is made of couples of edges, but this was only for a description purpose. In fact the general idea for this algorithm is that the tabu list can be made of any element that prevent to return back after a pejorative move. So for the TSP problem we can use, for example, couples of edges, a single edge or a single point.

The handle of the tabu list is very important and different strategies can have a big impact on the algorithm. Of course if we continue inserting element we will reach a point where there aren't possible moves. So we limit the size of the tabu list.

Algorithm 14 Tabu search

Input: Admissible solution S
Output: Best solution found

```
1: bestSolution ← S
2: while ! termination condition do
3:     if tabuList is full then
4:         tabuList ← empty list
5:     bestMove ← find best and not forbidden 2-OPT move of S
6:     if Δ(bestMove) > 0 then
7:         S ← perform bestMove
8:     else
9:         S ← perform bestMove
10:        tabuList ← add(bestMove)
11:    if cost(S) < cost(bestSolution) then
12:        bestSolution ← S
13: return bestSolution
```

The size is called tenure. When the list is full, the older element are removed. The tenure is another parameter to set. A possible way to do this is to set a minimal and a maximal value and let the tenure oscillate among them during the algorithm. So we will have diversification phases when the tenure is big and there are lot of forbidden move and intensification phases when the tenure is small and the algorithm has lot of freedom. This is called reactive tabu search. Another shrewdness is that if we encounter a move that is forbidden but that improves the incumbent, we will not perform that move but we will update the value of the incumbent in any case (Aspiration criterion).

On our implementation we use a fixed-size tabu list, with the tenure that is a parameter decided by the user. The list records only one vertex involved in the move: so for example if we perform a pejorative move and we substitute the edges v_{ab} and v_{cd} with v_{ac} and v_{bd}, only the vertex a is stored in the tabù list. When we find a good move, we check that all the 4 vertices involved are not in the tabu list, so even if we record only one point per move, this is sufficient to avoid to return back after a pejorative move. The check if a move is forbidden require a $O(k)$, where k is the tenure. So we perform this control only to moves that are better than the best move found until that moment (It's useless to check pejorative moves or moves that are not candidate to be selected as best move).
Finally we also implemented the aspiration criterion, so if a forbidden move leads to an improve of the incumbent, we keep that solution and we update the incumbent too.

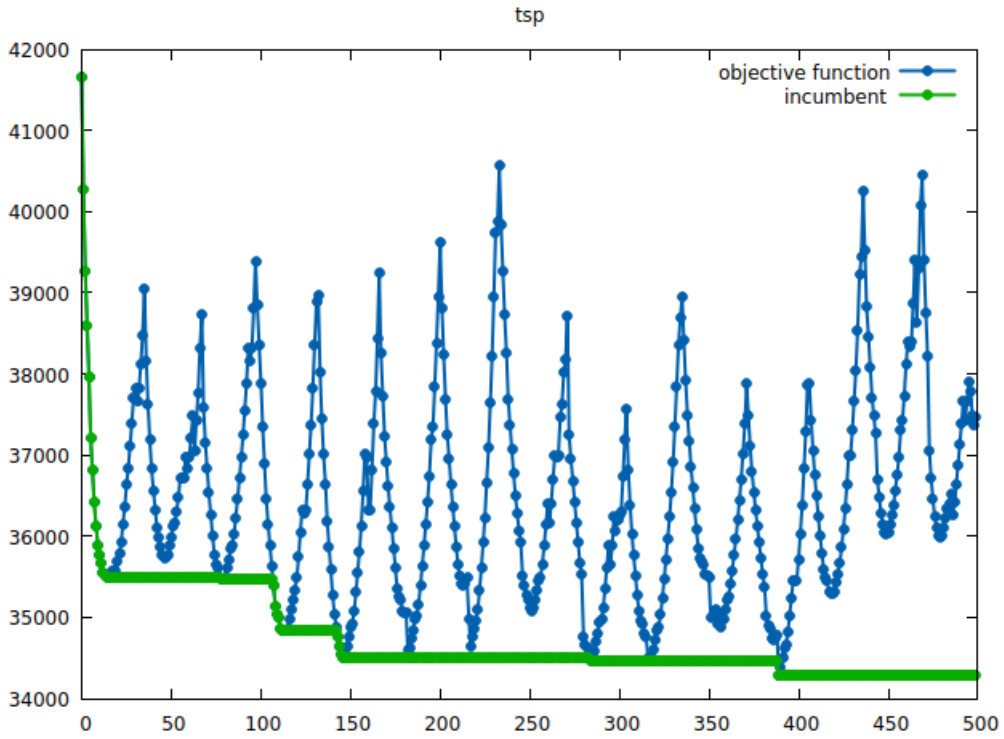


Figure 5.11: Objective function and incumbent during the execution of tabu search

5.3.4 Simulated Annealing

This metaheuristic algorithm takes inspiration from the annealing technique in metallurgy that consist on heating the material and then cooling it in a controlled way. When the temperature of the materials is high, the atoms inside it it vibrates and move in a chaotic way. As the temperature decreases, the atoms reduce their state of agitation until a stable point is reached.
The simulated annealing works in a similar way. Let's establish a parameter, the temperature T, that initially has an high value that gradually reduce during the iterations of the algorithm. The algorithm starts from an already provided solution. On each iteration we perform a random 2-OPT (or 3-OPT) move that of course can be also pejorative. The fundamental idea is that the probability to accept the move is related to the temperature: when T is high, the system is chaotic and we accept with high probability all the random move. As the system get cooler we increase the probability to reject a bad move (so we increase the probability to perform a good move). When the temperature is close to 0, we only make meliorative moves until we reach a minima.
Considering the objective function, we have that, when T is high, the algorithm is randomly exploring solutions. Only when T became low the algorithm moves toward a minima that hopefully is the optimum.
Of course the choose of the cooling schedule and the function that regulate the probability to accept a move with respect to the temperature are pivotal and can

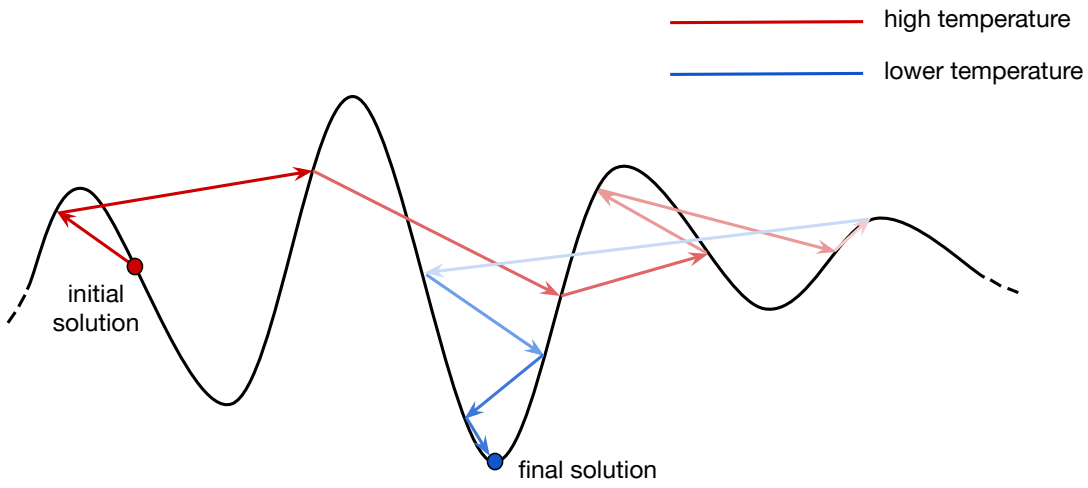


Figure 5.12: Example of the work of the simulated annealing. Note the initial random exploration with high temperature and the move toward the minima with a lower temperature.

have a big impact on the performance of the algorithm itself.
On our implementation we followed the procedure showed in the paper [13] that we are going to briefly explain. Let X be the initial solution and Y a solution produced after a random 2-OPT move on x, let $f(X)$, $f(Y)$ be the cost respectively of X and Y, let p_0 be the initial probability. On each iteration, until the temperature list is full, we generate a new Y. If $f(Y) > f(X)$ (pejorative move) we compute a temperature T with the following formula:

$$T = \frac{-|f(y) - f(x)|}{\ln(p_0)}$$

otherwise, if $f(Y) < f(X)$, we put X = Y and then we compute T in the same way. Now to describe the remaining part of our implementation let's consider N, that is a parameter decided by the user that indicate the maximum number of iteration where we use the same temperature. Let also i = 0, 1... N be a counter for those iterations. We start by extracting T_{max} from the temperature list, this will be the temperature of the system for the next following N iteration. On each iteration we generate a random 2-OPT move: if it is meliorative the solution is simply updated. If it is pejorative we compute a probability p_i defined as:

$$p_i = \exp\left(\frac{-(f(Y) - f(X))}{t_{max}}\right)$$

then we generate a random number $r_i \in [0, 1)$ and if $r_i < p_i$ the move is accepted. At this point, if we accept a bad move, we need to calculate another temperature T_i, defined as:

$$T_i = \frac{-d_i}{p_i}$$

Algorithm 15 Simulated Annealing

Input: Admissible solution S
Output: Best solution found

```
1: let p be the probability to accept a bad move
2: let T be the temperature
3: while ! termination condition do
4:     Δ ← random 2-OPT move
5:     if Δ > 0 then
6:         S ← perform the move
7:     else
8:         p ← f(T)
9:         r ← random number ∈ [0, 1)
10:        if r < p then
11:            S ← perform the move
12:    if temperature change condition then
13:        decrease T
14: return S
```

where d_i is the difference between $f(Y)$ and $f(X)$ on iteration i. We need also to keep in memory a counter c for each time we have performed a pejorative move. When we reach N iteration, we have to decrease the temperature: we compute the mean

$$T_{mean} = \frac{1}{c} \sum T_i$$

and we substitute T_{max} with T_{mean}. Since $T_i < T_{max}$ we have also that $T_{mean} < T_{max}$ so with the substitution we decrease the temperature. Now we pick another T_{max} and we repeat the entire procedure until the termination condition is reached.

5.3.5 Genetic Algorithm

The genetic algorithm is inspired by the evolution theory, formulated by Charles Darwin during the 18^{th} century. The algorithm starts with a population where each individual that belong to the population itself is a solution to the problem, so in our case each individual is a solution for the TSP. Each individual has also a fitness associated that is defined as the cost of the solution. On each iteration (also called epoch), the individuals that have the highest fitness are removed from the population: maintaining the parallelism with the evolution theory we can say that the individuals that didn't adapt to the environment (high fitness) died. On each iteration we have also that couples of individuals reproduce and the new elements substitute the ones that we have removed (so the population size remain constant). The children generated inherit characteristics from both their parents: a simple mechanism is just to pick part of the edges from one parent and the remaining from the other. Of course in this way the children is not a solution of the TSP problem cause some node may be visited more than once or may miss, so a repair function

is required. In addition, at this point, a mutation function can be added: this mean that a children with probability x (that in general is low) can suffer a modify that hasn't any relation with its parents, like a random exchange of two vertices in the visit sequence. We can introduce this shrewdness to emulate in the algorithm the mutation mechanism that happens also in real life.
After some iterations we have that the average fitness improve due to the fact that we remove bad solutions and we generate new solutions from the better ones. When the algorithm end, the champion of the population, that is the solution with the best fitness, is the final result.

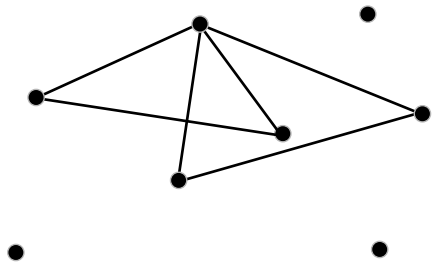
Algorithm 16 Genetic

Input: Population P_t of solutions S, N number of new element per epoch
Output: Best individual of the population

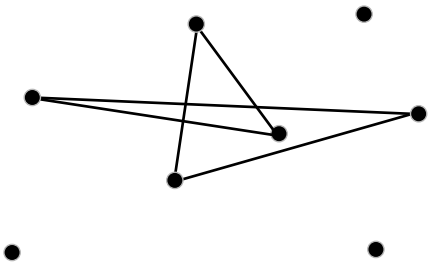
```
1:  t ← 0
2:  bestSolution
3:  fitnessList
4:  fitnessList, bestSolution ← computeFitness(P_t)
5:  while ! condition of termination do
6:      for i = 0, i < N, i++ do
7:          removeElement(P_t)
8:          update(fitnessList)
9:      for i = 0, i < N, i++ do
10:         S ← generateNewChildren(P_t)
11:         repair(S)
12:         fitnessList ← add(fitness(S))
13:         if fitness(S) < fitness(bestSolution) then
14:             bestSolution ← S
15:     t ← t+1
16: return bestSolution
```

Our first implementation of this algorithm was pretty simple. We generated a population of a size decided by the user using the GRASP algorithm. We created a probability mass function (PMF) normalizing the sum of the fitness, so on each iteration we randomly chose the elements to remove based on that mass function. This means that during the firsts iterations, when all the individual have almost the same fitness, the element are almost randomly remove, while when the algorithm progress and there is a more evident difference between good and bad individuals, with high probability only the worst solutions are removed. The number of element to remove on each iteration (that is equal to the number of children to generate) is a parameter decided by the user.
To reproduce two elements we used the following technique: we chose a random number x between 0.3 and 0.7. Then we picked the first x visited node from one of the parent and the remaining from the other. At this point we used a repair function

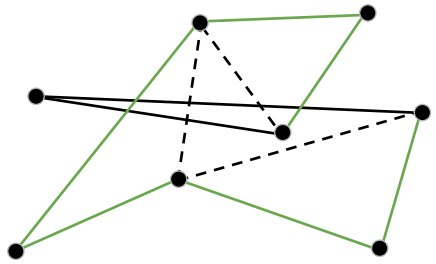
(Figure 5.13) that remove all the points visited twice and insert the missing ones using the insertion algorithm described in 5.1.3



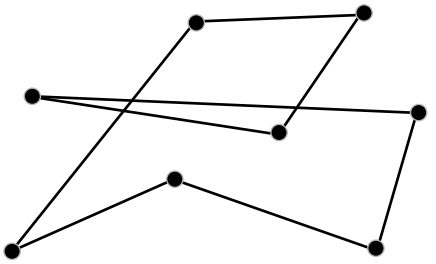
(a) Non repaired children



(b) Twice-visited vertices removal



(c) Insertion of remaining element



(d) Final result

Figure 5.13: Example of the steps of a repair function

The couples to reproduce are simply chosen at random from the population, we didn't implement any mechanism to control that same individual aren't choose more than once. We made this choice because on this algorithm populations ofetn have big size, in general more than 10000 individuals, so the probability to pick the exactly same individuals more than once is very low. After some run, plotting the average fitness and the best fitness, we could observe the expected behaviour (Figure 5.14): the average fitness continuously decrease until it get closer to the best fitness (that improves only sometimes).

The problem of our implementation became evident after testing it on different instances: the solutions had lot of intersection (like the ones generated by the GRASP algorithm) that of course don't belong to the optimum. An example of this can be seen on Figure 5.15. So we tried to introduce the 2-OPT someway inside the genetic. From this point, to distinguish the different versions of the genetic that we realized, we will call the implementation just described *genetic 1.*
A possibility that we first considered but we immediately discarded was to just ap-

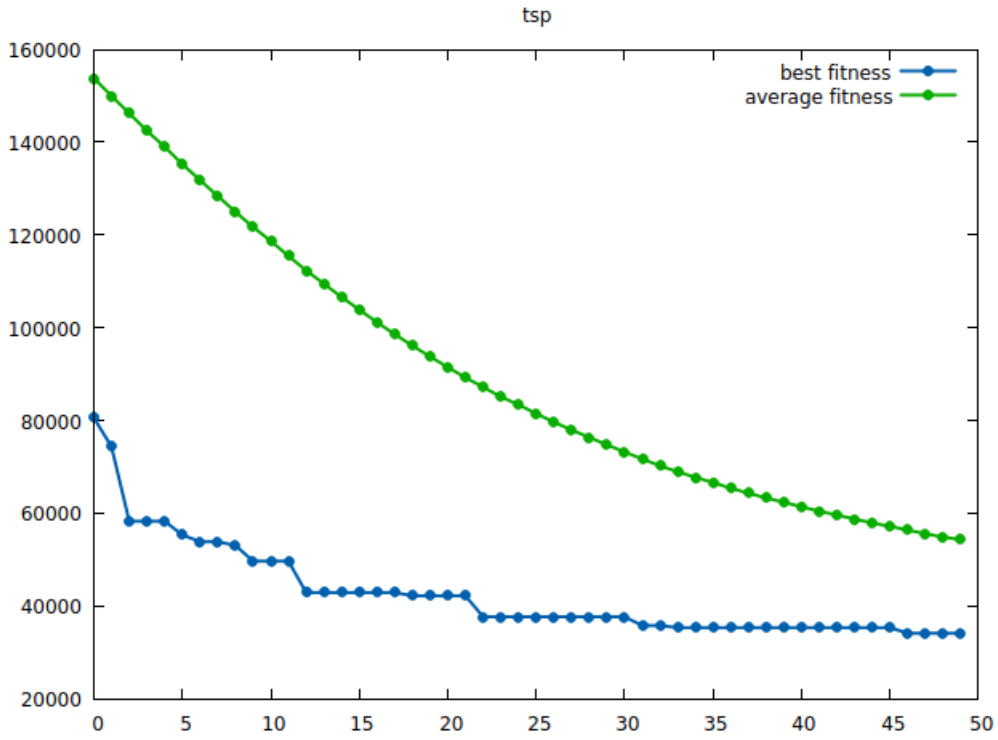


Figure 5.14: Plot of the average fitness and best fitness during the execution of a genetic algorithm.

ply the 2-OPT algorithm on the final solution provided by the genetic. This didn't make lot of sense because we did a lot of effort to produce a result with the genetic and then we discard a consistent part of it using the 2-OPT: a simple GRASP plus 2-OPT would be more efficient. Then we decided to add the 2-OPT to the repair function. So each children, after being repaired was also refined using the 2-OPT rule. Let's call this version *genetic 2*. We immediately noticed a large reduction of the number of epochs executed in the same amount of time with respect to genetic 1. This is because the populations used in the genetic algorithm are large and even if 2-OPT is quite fast it has to be executed a lot of time.
Finally we tried an hybrid approach between genetic 1 and genetic 2: after the repair function the refinement is executed with a probability x decided by the user. With this version each epoch is faster than genetic 2 because the 2-OPT is executed less time, however the price of this is the introduction of another parameter that is the probability x.

We tested these three versions on the dataset to find which ones is the best. We establish a time limit of 10 minutes and a population size of 2000 elements with 200 new elements each epoch. The 2-OPT, when applied inside the repair function, has a time limit of one second. Note that we were forced to use a population with a limited size, in fact after few tests we discovered that it's very difficult to handle a

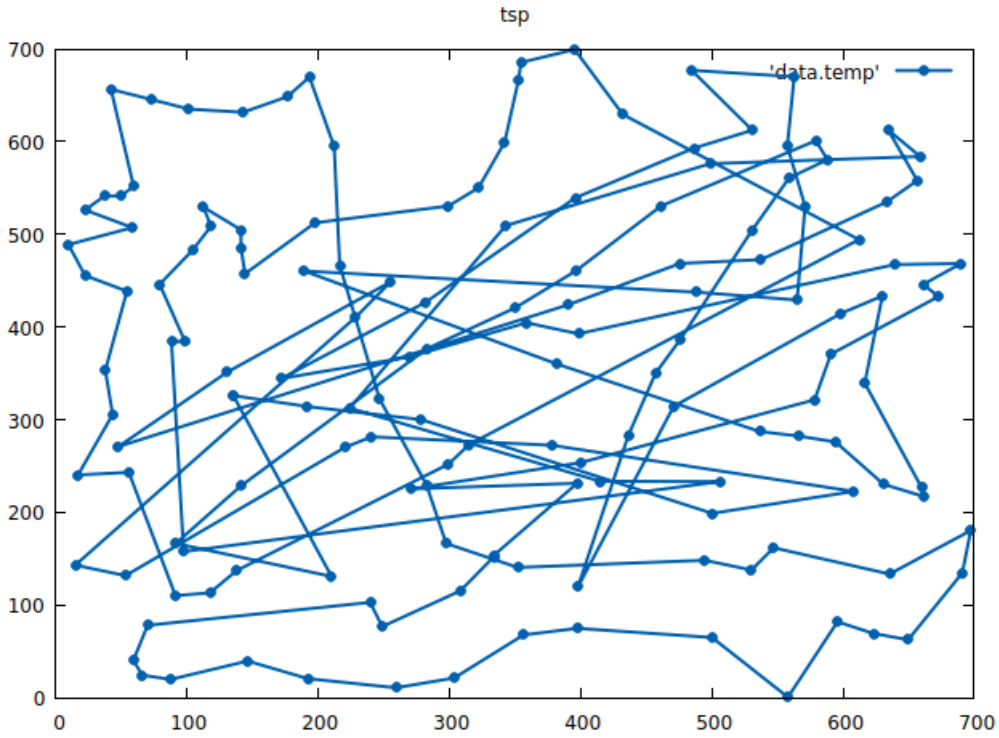


Figure 5.15: Solution of the instance ch150 provided by our first implementation of the genetic. Note the big number of edge intersections.

big population (like 10000 or more elements) on a single machine, cause the algorithm became incredibly slow. The Table 5.3 report the final values of the objective functions obtained by each algorithm and the number of epoch that it was able to execute within the time limit. The first thing that we can notice, as expected, is that the introduction of the 2-OPT involves a big decrease of the epochs performed. However in both genetic 2 and 3 results are better, so even if we pay an increase of execution time for each epoch, the addition of the refinement is worth. This became more evident if we consider the performance profile (Figure 5.16), where genetic 1 is far the worst algorithm.
Again, looking to the performance profile, genetic 2 seems to be the best, followed by genetic 3 that is slightly worse. However we chose genetic 3 as the winner of this first "eliminatory phase" for the following reasons: first of all the difference in the performance profile is tiny and since the dataset used is quite small we cannot say with absolute certainty that genetic 2 is the best. Moreover if we perform a 2-OPT on each new element, we move away from the idea of the genetic, in fact each children has to pass through a repair function and then a refinement, so with high probability it will maintain very few elements of its parents. Finally we execute very few epochs with respect to genetic 1 and 3 and this is against the principle of the evolution of the population. So this is why we selected the third version of the algorithm to compete with the others metaheuristics.

Instance	**Genetic 1**	**Epochs**	**Genetic 2**	**Epochs**	**Genetic 3**	**Epochs**
d657	58281	67	51628	6	51485	13
d1291	82270	9	60116	2	61857	3
fl417	13066	247	12040	3	12046	6
fl1400	29775	8	21836	2	21635	3
fl1577	37798	6	27996	1	30190	2
nrw1379	77437	7	65374	2	67865	3
p654	39289	67	34986	2	35216	5
pcb1173	83380	12	63693	2	63075	3
pcb3038	224091	1	217479	0	220982	0
pr1002	342579	19	278606	2	278935	4
pr2392	635460	1	566707	0	576564	0
rl1304	396680	9	316148	2	315956	3
rl1323	430922	8	325125	2	341412	3
rl1889	532907	3	486805	1	490274	1
u724	50800	48	44697	3	44857	7
u1060	309589	17	240649	2	241563	4
u2152	111343	1	98474	0	99573	0
u2319	316450	1	295777	0	297665	0
vm1084	326657	15	255282	2	258157	4
vm1748	502384	3	460075	1	480036	1

Table 5.3: Results of the eliminatory between the different versions of the genetic we implemented

To conclude we noticed that the genetic algorithm is well suited to a distributed approach. On the big data domain, when, for example, we need to perform a clustering algorithm, a possible solution is to split the set of points into smaller chunks, and locate them on different machines that execute the operations on them independently. We can think to a similar solution for the genetic: we can divide a big population into smaller subset, assign each of them to a different machine (or a different core if we have only one computer) and apply the algorithm. Considering the parallelism with the evolution theory, we can think to this as if we have different populations living in different habitats. At this point it is also possible to introduce "migrations" between different populations, this means that on each epoch we can randomly shuffle some elements between different subsets. So with this strategy we can run the genetic on a cluster, solving the problem of having to menage a big population and speeding up the execution of the algorithm.

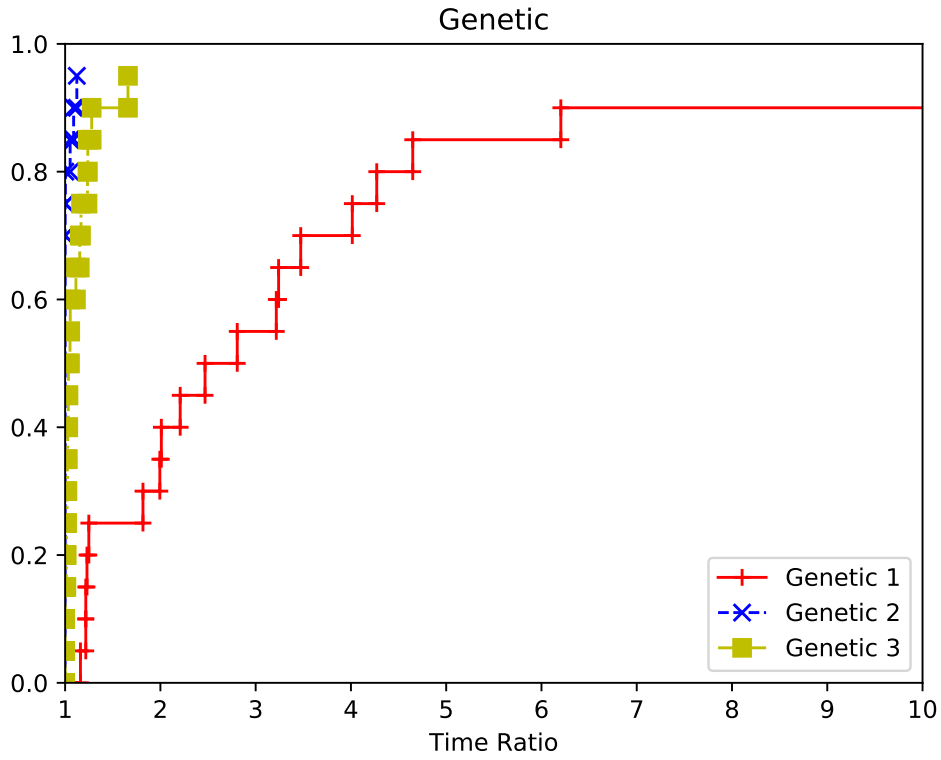


Figure 5.16: Performance profile of the three versions of the genetic algorithm we implemented

5.3.6 Final comparison between metaheuristics

On Figure 5.17 it is possible to see the performance profile of all the metaheuristics we analyzed in this section. We tested each algorithm on the usual dataset with a time limit of 30 minute per instance (all the other details about the experiments are report on Section 7.4). The parameter we used are the following:

- **Multi start**: no parameter except for time limit.
- **VNS**: 5-OPT random move for the kick.
- **Tabu search**: tenure equals to 40% of the instance size.
- **Simulated annealing**: temperature list of 5000 elements, 1000 iteration with the same temperature.
- **Genetic**: population size of 2000, 200 new elements per epoch, probability 0.5 to execute 2-OPT after a repair.

The first thing we can notice is that the VNS has the best performance. Of course our dataset is quite small, so we cannot say with absolute certainty that VNS is in

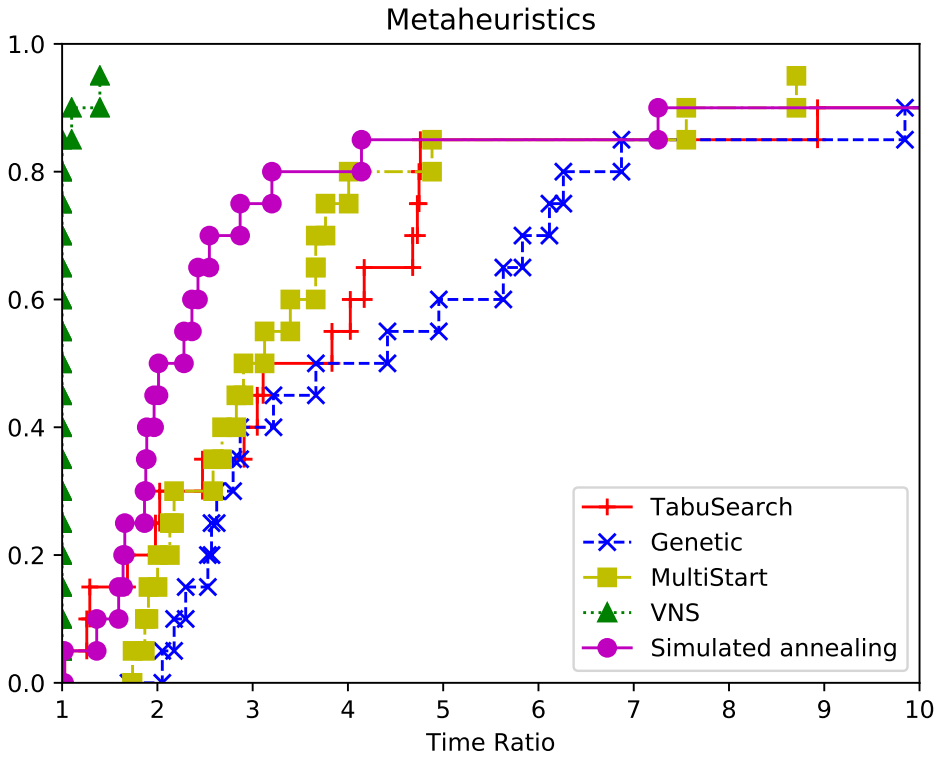


Figure 5.17: Performance profile of the metaheuristics

absolute the best algorithm. However our results are aligned with the ones we saw during the course and the ones obtained by other students, so we can conclude that VNS is a very good heuristic solution for the TSP.
Another thing we can notice are the poor performance of the genetic. This result was expected due to the fact we were forced to use small populations and to introduce the 2-OPT at the expense of the number of epoch for each run.
Finally, noteworthy is the simulates annealing that seems to have very good performance.

Chapter 6

Conclusions

In conclusion let's summarize the results we obtained. Considering the compact models, we have that the flow based formulations, in particular flow 1 and flow 2, perform better than the MTZ and the time based formulations. Looking to the Performance profile we can conclude that flow 1 is the best compact model. Moving to the exact algorithms, here we use the base TSP model, with different strategies to add the subtour eliminator constraints during (callbacks) or after (loop method) the execution of CPLEX. In general we have that these type of algorithms work better than a simple compact model. Again looking to the performance profile we can establish that the combination of the UserCut and Heuristic Callbacks based on the Generic Callback is the best strategy. In the matheuristic chapter we analyze some heuristic solutions that are based on CPLEX and we reach the conclusion that hard fixing is the best matheuristic. Finally in the stand-alone heuristics chapter we explore several strategies that don't provide an exact solution and are independent from CPLEX. We conclude that VNS is probably the best performing heuristic and simulated annealing and tabu search can be good solutions too.

So if the goal is to resolve a relatively small instance of the TSP or we have big computational power and lots of time, we suggest to use CPLEX with UserCut and Heuristic Callbacks based on the Generic Callback. Otherwise if we have to resolve a big instance or we are not interested to obtain the exact solution and we have a time limit that is very important to respect we suggest VNS.

Bibliography

[1] J. D. U. John E. Hopcroft, Rajeev Motwani, *Automi, linguaggi e calcolabilità*. Pearson, 3rd ed., 2018.

[2] M. Fischetti, *Lezioni di ricerca operativa*. Kindle Direct Publishing, 4th ed., 2018.

[3] E. D. Dolan and J. Moré, "Benchmarking optimization software with performance profiles," *Mathematical Programming*, vol. 91, no. 2, pp. 201–213, 2002.

[4] A. J. Orman and H. P. Williams, "A survey of different integer programming formulations of the travelling salesman problem," 2007.

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

[6] B. Gavish and S. C. Graves, "The Travelling Salesman Problem and Related Problems," July 1978.

[7] G. Reinelt, "TSPLIB—A Traveling Salesman Problem Library," *INFORMS Journal on Computing*, vol. 3, pp. 376–384, November 1991.

[8] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson, "On a linear-programming, combinatorial approach to the traveling-salesman problem," *Operations Research*, vol. 7, no. 1, pp. 58–66, 1959.

[9] J. F. Benders, "Partitioning procedures for solving mixed-variables programming problems," *Numerische Mathematik*, vol. 4, no. 1, pp. 238–252, 1962.

[10] M. Fischetti and A. Lodi, "Local branching," *Mathematical Programming*, vol. 98, no. 1, pp. 23–47, 2003.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd ed., 2009.

[12] R. Balamurugan, A. M. Natarajan, and K. Premalatha, "Stellar-mass black hole optimization for biclustering microarray gene expression data," *Applied Artificial Intelligence*, vol. 29, no. 4, pp. 353–381, 2015.

[13] Z.-j. Z. Shi-hua Zhan, Juan Lin and Y. wen Zhong, "List-based simulated annealing algorithm for traveling salesman problem," *Computational Intelligence and Neuroscience*, 2016.

Chapter 7

Results Tables

In this chapter we report the time measurements of all the algorithms that we have implemented and described in this report. For Callback methods we have also reported the number of nodes solved to compute the problem to optimum.

7.1 Compact Models

In this section we report the time measurements of the Compact Models with their relative seed and algorithm.

Table 7.1: **Compact Models**

Instance	Seed	MTZ	Flow 1	Flow 2	Flow 3
ulysses22	23	516.927	0.343	0.359	3.348
att48	23	3.855	1.327	1.791	186.627
ulysses16	23	3.938	0.452	0.142	0.734
st70	23	1147.487	137.717	192.168	8418.164
pr76	23	77.406	100.940	284.219	2667.292
eil76	23	20.430	183.920	130.450	18915.042
eil51	23	3.328	32.711	32.251	1659.084
berlin52	23	30.065	35.455	73.215	4405.665
burma14	23	0.218	0.110	0.114	0.193
ulysses22	2123	527.003	0.390	0.907	3.498
att48	2123	5.303	1.793	1.808	194.977
ulysses16	2123	3.114	0.106	0.286	0.740
st70	2123	1356.786	230.846	122.904	4243.212
pr76	2123	108.559	112.697	190.324	2616.969
eil76	2123	15.946	282.910	353.393	14471.835
eil51	2123	2.289	7.554	46.835	6745.074
berlin52	2123	92.319	12.675	83.720	4713.600
burma14	2123	0.151	0.131	0.110	0.216

Continued on next page

Table 7.1 – continued from previous page

Instance	Seed	MTZ	Flow 1	Flow 2	Flow 3
ulysses22	48399	768.739	0.253	0.496	3.308
att48	48399	4.348	1.547	1.453	197.239
ulysses16	48399	3.141	0.227	0.250	0.665
st70	48399	1356.786	257.183	265.412	4616.888
pr76	48399	71.636	76.051	234.136	2707.392
eil76	48399	8.915	230.493	421.678	24884.819
eil51	48399	0.888	15.096	60.214	1033.519
berlin52	48399	46.524	18.031	69.704	7270.184
burma14	48399	0.149	0.212	0.078	0.216

7.2 Exact Algorithms

In this section we report the time measurements (in seconds) of the Exact Algorithms with their relative seed and algorithm.

Table 7.2: **Loop Methods**

Instance	Seed	Simple Loop	Heuristic Loop
a280	201909284	29.781	32.546
att48	201909284	0.681	0.594
att532	201909284	663.910	719.934
berlin52	201909284	0.095	0.063
bier127	201909284	4.407	5.622
ch130	201909284	5.103	5.557
ch150	201909284	11.128	15.697
d198	201909284	72.351	69.021
gil262	201909284	55.548	49.800
kroA100	201909284	4.419	4.651
kroA150	201909284	23.498	16.290
kroA200	201909284	64.854	82.079
kroB100	201909284	7.698	7.828
kroB150	201909284	34.584	31.630
kroB200	201909284	22.810	19.158
kroC100	201909284	4.956	3.526
kroD100	201909284	4.964	4.677
kroE100	201909284	4.540	4.475
lin105	201909284	3.567	4.633
lin318	201909284	97.536	110.049
pr226	201909284	197.544	260.199
rat195	201909284	69.132	47.261
pr76	201909284	9.330	8.509
pr107	201909284	0.631	2.172
pr124	201909284	20.794	23.092
pr136	201909284	5.627	6.156
tsp225	201909284	32.324	29.078
u159	201909284	7.316	7.618
rat99	201909284	2.413	2.727
rd400	201909284	305.766	456.781
a280	19	29.235	24.520
att48	19	0.593	0.671
att532	19	734.706	756.783
berlin52	19	0.097	0.065
bier127	19	4.456	5.910

Continued on next page

Table 7.2 – continued from previous page

Instance	Seed	Simple Loop	Heuristic Loop
ch130	19	5.183	6.728
ch150	19	10.200	17.139
d198	19	65.949	70.533
gil262	19	54.885	42.880
kroA100	19	4.190	4.722
kroA150	19	25.094	17.489
kroA200	19	82.620	84.170
kroB100	19	8.183	7.517
kroB150	19	33.041	29.433
kroB200	19	28.127	12.029
kroC100	19	4.825	3.915
kroD100	19	5.098	4.810
kroE100	19	4.362	4.274
lin105	19	2.935	4.384
lin318	19	95.144	103.283
pr226	19	375.243	245.419
rat195	19	63.654	72.675
pr76	19	8.932	9.417
pr107	19	0.578	3.075
pr124	19	19.232	19.579
pr136	19	5.519	7.685
tsp225	19	32.647	27.622
u159	19	7.472	8.665
rat99	19	1.829	3.217
rd400	19	372.713	321.477
a280	190796	26.739	25.632
att48	190796	0.632	0.611
att532	190796	723.908	777.047
berlin52	190796	0.092	0.166
bier127	190796	4.320	5.399
ch130	190796	5.034	7.076
ch150	190796	12.824	18.325
d198	190796	63.516	52.336
gil262	190796	52.419	43.887
kroA100	190796	4.025	4.328
kroA150	190796	23.075	17.093
kroA200	190796	84.148	78.703
kroB100	190796	7.607	7.928
kroB150	190796	34.306	33.014
kroB200	190796	24.217	13.384
kroC100	190796	4.926	3.206

Continued on next page

Table 7.2 – continued from previous page

Instance	Seed	Simple Loop	Heuristic Loop
kroD100	190796	5.197	4.223
kroE100	190796	5.123	4.568
lin105	190796	3.519	2.837
lin318	190796	99.489	91.392
pr226	190796	181.923	555.863
rat195	190796	59.865	50.482
pr76	190796	10.316	8.644
pr107	190796	0.555	2.458
pr124	190796	21.904	23.429
pr136	190796	6.314	6.603
tsp225	190796	32.958	32.733
u159	190796	7.975	6.037
rat99	190796	2.170	2.127
rd400	190796	336.298	391.179
a280	19071996	31.560	22.689
att48	19071996	0.668	0.679
att532	19071996	656.097	696.500
berlin52	19071996	0.103	0.086
bier127	19071996	4.610	4.364
ch130	19071996	4.844	6.604
ch150	19071996	10.942	15.347
d198	19071996	71.141	67.801
gil262	19071996	61.853	45.738
kroA100	19071996	4.449	4.557
kroA150	19071996	27.511	15.670
kroA200	19071996	59.241	81.205
kroB100	19071996	8.467	7.117
kroB150	19071996	37.819	27.418
kroB200	19071996	23.213	18.620
kroC100	19071996	4.877	3.207
kroD100	19071996	5.093	4.278
kroE100	19071996	4.483	3.851
lin105	19071996	3.633	4.231
lin318	19071996	95.643	99.793
pr226	19071996	156.506	146.272
rat195	19071996	67.494	63.071
pr76	19071996	8.173	9.800
pr107	19071996	0.552	1.506
pr124	19071996	22.828	18.270
pr136	19071996	6.206	8.585
tsp225	19071996	34.639	32.599

Continued on next page

Table 7.2 – continued from previous page

Instance	Seed	Simple Loop	Heuristic Loop
u159	19071996	7.453	7.642
rat99	19071996	2.339	2.207
rd400	19071996	337.442	374.722

Table 7.3: **Legacy Callbacks**

Instance	Seed	Lazy	UserCut	Heuristic
a280	201909284	6.229	3.674	3.671
att48	201909284	0.095	0.069	0.057
att532	19071996	3532.022	3511.272	3504.440
berlin52	19071996	0.028	0.029	0.029
bier127	19071996	1.219	0.592	0.626
ch130	19071996	1.389	0.990	0.915
ch150	19071996	5.351	2.396	1.571
d198	19071996	40.486	4.199	4.336
gil262	19071996	163.289	21.162	23.463
kroA100	19071996	1.183	0.758	0.678
kroA150	19071996	6.082	3.761	4.292
kroA200	19071996	209.747	41.006	78.697
kroB100	19071996	1.614	0.799	0.647
kroB150	19071996	20.045	9.197	8.712
kroB200	19071996	22.930	4.129	3.325
kroC100	19071996	0.804	0.670	0.666
kroD100	19071996	1.064	0.794	0.632
kroE100	19071996	3.159	1.957	3.083
lin105	19071996	0.497	0.210	0.218
lin318	19071996	103.086	22.930	26.731
pr226	19071996	27.445	4.292	4.380
rat195	19071996	43.155	21.471	14.749
pr76	19071996	14.067	11.805	4.295
pr107	19071996	0.068	0.052	0.087
pr124	19071996	2.134	1.826	1.614
pr136	19071996	1.226	1.533	1.563
tsp225	19071996	40.637	4.016	2.634
u159	19071996	1.880	0.761	0.926
rat99	19071996	0.488	0.350	0.369
rd400	19071996	3551.366	461.552	564.476
a280	19	8.130	4.269	4.317
att48	19	0.089	0.071	0.056

Continued on next page

Table 7.3 – continued from previous page

Instance	Seed	Lazy	UserCut	Heuristic
att532	19	3535.718	1061.775	1233.420
berlin52	19	0.028	0.028	0.026
bier127	19	0.896	0.785	0.663
ch130	19	1.438	1.083	1.128
ch150	19	7.742	3.105	2.110
d198	19	23.188	6.341	4.156
gil262	19	116.108	44.069	34.565
kroA100	19	1.222	0.869	0.612
kroA150	19	8.917	3.338	3.783
kroA200	19	110.787	36.201	36.140
kroB100	19	2.114	0.711	0.464
kroB150	19	28.851	6.063	7.115
kroB200	19	32.032	4.254	6.101
kroC100	19	0.844	0.722	0.656
kroD100	19	0.983	0.770	0.789
kroE100	19	2.466	1.265	1.430
lin105	19	0.419	0.212	0.236
lin318	19	198.618	23.924	22.958
pr226	19	18.191	4.840	4.886
rat195	19	51.779	11.243	12.493
pr76	19	13.690	11.829	3.883
pr107	19	0.052	0.053	0.088
pr124	19	7.439	2.004	1.831
pr136	19	1.416	1.803	1.726
tsp225	19	29.004	4.494	3.006
u159	19	2.805	0.844	1.090
rat99	19	0.460	0.470	0.302
rd400	19	3565.286	601.292	704.836
a280	190796	10.591	4.896	5.036
att48	190796	0.091	0.070	0.059
att532	190796	3550.588	639.059	1448.989
berlin52	190796	0.031	0.031	0.028
bier127	190796	0.994	0.590	0.626
ch130	190796	2.110	1.267	0.924
ch150	190796	6.682	1.663	1.957
d198	190796	19.916	5.745	5.395
gil262	190796	231.023	13.520	22.168
kroA100	190796	1.269	0.751	0.739
kroA150	190796	10.130	3.806	2.339
kroA200	190796	243.669	96.150	62.074
kroB100	190796	1.104	0.930	0.947

Continued on next page

Table 7.3 – continued from previous page

Instance	Seed	Lazy	UserCut	Heuristic
kroB150	190796	23.558	6.720	6.849
kroB200	190796	79.954	5.045	4.684
kroC100	190796	0.998	0.645	0.657
kroD100	190796	1.026	0.727	0.731
kroE100	190796	2.600	1.667	1.776
lin105	190796	0.573	0.214	0.213
lin318	190796	211.935	48.083	17.926
pr226	190796	19.142	5.849	5.889
rat195	190796	43.599	15.382	15.497
pr76	190796	12.072	8.309	4.701
pr107	190796	0.052	0.052	0.088
pr124	190796	2.698	2.273	2.232
pr136	190796	2.104	1.412	1.259
tsp225	190796	13.332	5.804	2.825
u159	190796	2.792	0.754	0.896
rat99	190796	0.639	0.381	0.427
rd400	190796	3562.556	312.888	249.195
a280	19071996	15.257	5.946	5.931
att48	19071996	0.093	0.070	0.056
att532	19071996	3547.388	2244.815	3504.358
berlin52	19071996	0.031	0.030	0.028
bier127	19071996	0.896	0.610	0.627
ch130	19071996	1.651	1.051	0.949
ch150	19071996	6.329	2.499	2.583
d198	19071996	55.018	5.765	5.581
gil262	19071996	168.740	34.147	32.721
kroA100	19071996	1.515	0.756	0.679
kroA150	19071996	8.992	4.686	3.257
kroA200	19071996	453.677	98.514	112.663
kroB100	19071996	1.847	0.760	0.710
kroB150	19071996	18.980	7.197	10.153
kroB200	19071996	16.922	4.777	4.853
kroC100	19071996	0.853	0.658	0.682
kroD100	19071996	0.627	0.658	0.551
kroE100	19071996	2.643	1.991	2.032
lin105	19071996	0.532	0.256	0.219
lin318	19071996	93.532	33.001	21.742
pr226	19071996	16.314	4.834	4.878
rat195	19071996	29.843	13.269	12.237
pr76	19071996	19.724	7.595	4.206
pr107	19071996	0.052	0.049	0.088

Continued on next page

Table 7.3 – continued from previous page

Instance	Seed	Lazy	UserCut	Heuristic
pr124	19071996	3.873	1.828	1.899
pr136	19071996	1.953	1.352	1.729
tsp225	19071996	11.742	4.704	4.759
u159	19071996	2.977	0.769	0.808
rat99	19071996	0.496	0.352	0.353
rd400	19071996	3563.643	223.105	408.701

Table 7.4: **Generic Callbacks**

Instance	Seed	Lazy	UserCut	Heuristic
a280	201909284	12.946	12.593	14.801
att48	201909284	0.184	0.187	0.209
att532	201909284	2446.941	639.188	460.378
berlin52	201909284	0.077	0.096	0.080
bier127	201909284	3.442	1.739	1.149
ch130	201909284	4.504	4.221	1.673
ch150	201909284	9.392	6.332	6.024
d198	201909284	29.695	12.840	11.985
gil262	201909284	64.250	42.878	25.663
kroA100	201909284	2.967	2.888	0.898
kroA150	201909284	10.574	6.380	9.067
kroA200	201909284	69.231	25.079	28.455
kroB100	201909284	5.871	2.644	1.980
kroB150	201909284	15.901	6.044	6.368
kroB200	201909284	26.611	11.157	14.564
kroC100	201909284	1.204	1.121	0.981
kroD100	201909284	1.418	1.694	1.141
kroE100	201909284	4.858	3.181	1.501
lin105	201909284	1.444	0.801	0.648
lin318	201909284	67.349	25.349	29.021
pr226	201909284	29.891	18.124	15.023
rat195	201909284	32.835	33.280	12.177
pr76	201909284	6.359	6.835	2.957
pr107	201909284	0.126	0.086	0.136
pr124	201909284	4.793	2.574	2.315
pr136	201909284	2.673	1.776	1.540
tsp225	201909284	21.783	13.709	9.678
u159	201909284	4.659	1.953	2.395
rat99	201909284	0.579	0.585	0.586

Continued on next page

Table 7.4 – continued from previous page

Instance	Seed	Lazy	UserCut	Heuristic
rd400	201909284	197.624	131.576	113.149
a280	19	20.864	15.620	9.475
att48	19	0.174	0.181	0.145
att532	19	2744.552	534.519	281.131
berlin52	19	0.073	0.072	0.086
bier127	19	3.684	2.627	1.267
ch130	19	5.851	4.489	1.615
ch150	19	12.946	8.178	4.030
d198	19	31.508	13.839	10.980
gil262	19	64.308	34.720	31.280
kroA100	19	1.984	1.401	0.906
kroA150	19	12.968	9.416	5.820
kroA200	19	67.595	33.454	17.821
kroB100	19	3.784	2.542	1.878
kroB150	19	18.894	13.727	4.816
kroB200	19	16.304	12.599	8.692
kroC100	19	2.578	1.514	1.061
kroD100	19	1.531	1.163	1.328
kroE100	19	4.791	3.421	1.445
lin105	19	0.657	0.601	0.761
lin318	19	55.896	25.346	21.318
pr226	19	28.894	15.489	15.526
rat195	19	43.827	33.000	15.290
pr76	19	6.057	8.389	4.631
pr107	19	0.086	0.087	0.137
pr124	19	6.104	3.381	4.564
pr136	19	2.411	2.316	2.354
tsp225	19	14.538	12.006	9.069
u159	19	5.335	2.706	2.917
rat99	19	0.610	0.592	0.598
rd400	19	228.320	117.198	96.235
a280	190796	16.525	22.052	16.995
att48	190796	0.159	0.163	0.133
att532	190796	2196.201	583.651	404.120
berlin52	190796	0.067	0.068	0.069
bier127	190796	3.782	1.934	1.298
ch130	190796	4.024	3.708	1.990
ch150	190796	12.687	4.771	3.670
d198	190796	31.920	9.793	13.984
gil262	190796	47.178	56.418	32.368
kroA100	190796	2.773	2.263	1.023

Continued on next page

Table 7.4 – continued from previous page

Instance	Seed	Lazy	UserCut	Heuristic
kroA150	190796	10.338	11.541	6.809
kroA200	190796	21.538	52.614	22.347
kroB100	190796	2.759	2.189	1.679
kroB150	190796	12.962	14.925	4.931
kroB200	190796	12.705	13.328	11.764
kroC100	190796	2.144	1.681	0.946
kroD100	190796	1.527	1.420	1.133
kroE100	190796	3.471	4.130	1.416
lin105	190796	0.926	0.647	0.722
lin318	190796	176.928	30.363	33.881
pr226	190796	113.334	16.882	18.258
rat195	190796	30.761	19.633	16.996
pr76	190796	6.822	9.637	4.680
pr107	190796	0.086	0.088	0.138
pr124	190796	9.535	2.334	4.686
pr136	190796	2.576	3.237	2.173
tsp225	190796	15.866	13.499	7.800
u159	190796	5.308	2.526	1.916
rat99	190796	0.505	0.507	0.569
rd400	190796	186.584	231.009	81.896
a280	19071996	15.386	15.735	17.806
att48	19071996	0.160	0.164	0.134
att532	19071996	1077.461	1191.602	303.004
berlin52	19071996	0.073	0.073	0.085
bier127	19071996	4.199	1.947	1.435
ch130	19071996	2.771	2.104	1.762
ch150	19071996	8.672	6.927	6.316
d198	19071996	23.319	11.724	14.634
gil262	19071996	46.932	42.738	26.112
kroA100	19071996	1.599	1.499	1.007
kroA150	19071996	17.055	8.977	7.263
kroA200	19071996	111.190	36.090	30.106
kroB100	19071996	5.167	2.665	1.836
kroB150	19071996	16.517	14.767	5.580
kroB200	19071996	14.375	10.629	8.862
kroC100	19071996	2.695	2.154	0.923
kroD100	19071996	1.610	1.815	0.914
kroE100	19071996	4.294	3.357	1.509
lin105	19071996	0.854	0.640	0.609
lin318	19071996	87.572	30.366	38.257
pr226	19071996	25.345	17.512	13.138

Continued on next page

Table 7.4 – continued from previous page

Instance	Seed	Lazy	UserCut	Heuristic
rat195	19071996	34.013	24.517	13.680
pr76	19071996	4.712	7.046	6.066
pr107	19071996	0.087	0.092	0.144
pr124	19071996	5.513	3.154	5.490
pr136	19071996	2.491	2.855	2.662
tsp225	19071996	24.909	15.986	14.130
u159	19071996	4.774	2.629	3.573
rat99	19071996	0.539	0.546	0.580
rd400	19071996	300.483	176.906	71.727

Table 7.5: **All Callbacks (* Generic Callback)**

Instance	Seed	Lazy	UserCut	Heuristic	Lazy*	UserCut*	Heuristic*
a280	201909284	6.229	3.674	3.671	12.946	12.593	14.801
att48	201909284	0.095	0.069	0.057	0.184	0.187	0.209
att532	201909284	3532.022	3511.272	3504.440	2446.941	639.188	460.378
berlin52	201909284	0.028	0.029	0.029	0.077	0.096	0.080
bier127	201909284	1.219	0.592	0.626	3.442	1.739	1.149
ch130	201909284	1.389	0.990	0.915	4.504	4.221	1.673
ch150	201909284	5.351	2.396	1.571	9.392	6.332	6.024
d198	201909284	40.486	4.199	4.336	29.695	12.840	11.985
gil262	201909284	163.289	21.162	23.463	64.250	42.878	25.663
kroA100	201909284	1.183	0.758	0.678	2.967	2.888	0.898
kroA150	201909284	6.082	3.761	4.292	10.574	6.380	9.067
kroA200	201909284	209.747	41.006	78.697	69.231	25.079	28.455
kroB100	201909284	1.614	0.799	0.647	5.871	2.644	1.980
kroB150	201909284	20.045	9.197	8.712	15.901	6.044	6.368
kroB200	201909284	22.930	4.129	3.325	26.611	11.157	14.564
kroC100	201909284	0.804	0.670	0.666	1.204	1.121	0.981
kroD100	201909284	1.064	0.794	0.632	1.418	1.694	1.141
kroE100	201909284	3.159	1.957	3.083	4.858	3.181	1.501
lin105	201909284	0.497	0.210	0.218	1.444	0.801	0.648
lin318	201909284	103.086	22.930	26.731	67.349	25.349	29.021
pr226	201909284	27.445	4.292	4.380	29.891	18.124	15.023
rat195	201909284	43.155	21.471	14.749	32.835	33.280	12.177
pr76	201909284	14.067	11.805	4.295	6.359	6.835	2.957
pr107	201909284	0.068	0.052	0.087	0.126	0.086	0.136
pr124	201909284	2.134	1.826	1.614	4.793	2.574	2.315
pr136	201909284	1.226	1.533	1.563	2.673	1.776	1.540

Continued on next page

Table 7.5 – continued from previous page

Instance	Seed	Lazy	UserCut	Heuristic	Lazy*	UserCut*	Heuristic*
tsp225	201909284	40.637	4.016	2.634	21.783	13.709	9.678
u159	201909284	1.880	0.761	0.926	4.659	1.953	2.395
rat99	201909284	0.488	0.350	0.369	0.579	0.585	0.586
rd400	201909284	3551.366	461.552	564.476	197.624	131.576	113.149
a280	19	8.130	4.269	4.317	20.864	15.620	9.475
att48	19	0.089	0.071	0.056	0.174	0.181	0.145
att532	19	3535.718	1061.775	1233.420	2744.552	534.519	281.131
berlin52	19	0.028	0.028	0.026	0.073	0.072	0.086
bier127	19	0.896	0.785	0.663	3.684	2.627	1.267
ch130	19	1.438	1.083	1.128	5.851	4.489	1.615
ch150	19	7.742	3.105	2.110	12.946	8.178	4.030
d198	19	23.188	6.341	4.156	31.508	13.839	10.980
gil262	19	116.108	44.069	34.565	64.308	34.720	31.280
kroA100	19	1.222	0.869	0.612	1.984	1.401	0.906
kroA150	19	8.917	3.338	3.783	12.968	9.416	5.820
kroA200	19	110.787	36.201	36.140	67.595	33.454	17.821
kroB100	19	2.114	0.711	0.464	3.784	2.542	1.878
kroB150	19	28.851	6.063	7.115	18.894	13.727	4.816
kroB200	19	32.032	4.254	6.101	16.304	12.599	8.692
kroC100	19	0.844	0.722	0.656	2.578	1.514	1.061
kroD100	19	0.983	0.770	0.789	1.531	1.163	1.328
kroE100	19	2.466	1.265	1.430	4.791	3.421	1.445
lin105	19	0.419	0.212	0.236	0.657	0.601	0.761
lin318	19	198.618	23.924	22.958	55.896	25.346	21.318
pr226	19	18.191	4.840	4.886	28.894	15.489	15.526
rat195	19	51.779	11.243	12.493	43.827	33.000	15.290
pr76	19	13.690	11.829	3.883	6.057	8.389	4.631
pr107	19	0.052	0.053	0.088	0.086	0.087	0.137
pr124	19	7.439	2.004	1.831	6.104	3.381	4.564
pr136	19	1.416	1.803	1.726	2.411	2.316	2.354
tsp225	19	29.004	4.494	3.006	14.538	12.006	9.069
u159	19	2.805	0.844	1.090	5.335	2.706	2.917
rat99	19	0.460	0.470	0.302	0.610	0.592	0.598
rd400	19	3565.286	601.292	704.836	228.320	117.198	96.235
a280	190796	10.591	4.896	5.036	16.525	22.052	16.995
att48	190796	0.091	0.070	0.059	0.159	0.163	0.133
att532	190796	3550.588	639.059	1448.989	2196.201	583.651	404.120
berlin52	190796	0.031	0.031	0.028	0.067	0.068	0.069
bier127	190796	0.994	0.590	0.626	3.782	1.934	1.298
ch130	190796	2.110	1.267	0.924	4.024	3.708	1.990
ch150	190796	6.682	1.663	1.957	12.687	4.771	3.670

Continued on next page

Table 7.5 – continued from previous page

Instance	Seed	Lazy	UserCut	Heuristic	Lazy*	UserCut*	Heuristic*
d198	190796	19.916	5.745	5.395	31.920	9.793	13.984
gil262	190796	231.023	13.520	22.168	47.178	56.418	32.368
kroA100	190796	1.269	0.751	0.739	2.773	2.263	1.023
kroA150	190796	10.130	3.806	2.339	10.338	11.541	6.809
kroA200	190796	243.669	96.150	62.074	21.538	52.614	22.347
kroB100	190796	1.104	0.930	0.947	2.759	2.189	1.679
kroB150	190796	23.558	6.720	6.849	12.962	14.925	4.931
kroB200	190796	79.954	5.045	4.684	12.705	13.328	11.764
kroC100	190796	0.998	0.645	0.657	2.144	1.681	0.946
kroD100	190796	1.026	0.727	0.731	1.527	1.420	1.133
kroE100	190796	2.600	1.667	1.776	3.471	4.130	1.416
lin105	190796	0.573	0.214	0.213	0.926	0.647	0.722
lin318	190796	211.935	48.083	17.926	176.928	30.363	33.881
pr226	190796	19.142	5.849	5.889	113.334	16.882	18.258
rat195	190796	43.599	15.382	15.497	30.761	19.633	16.996
pr76	190796	12.072	8.309	4.701	6.822	9.637	4.680
pr107	190796	0.052	0.052	0.088	0.086	0.088	0.138
pr124	190796	2.698	2.273	2.232	9.535	2.334	4.686
pr136	190796	2.104	1.412	1.259	2.576	3.237	2.173
tsp225	190796	13.332	5.804	2.825	15.866	13.499	7.800
u159	190796	2.792	0.754	0.896	5.308	2.526	1.916
rat99	190796	0.639	0.381	0.427	0.505	0.507	0.569
rd400	190796	3562.556	312.888	249.195	186.584	231.009	81.896
a280	19071996	15.257	5.946	5.931	15.386	15.735	17.806
att48	19071996	0.093	0.070	0.056	0.160	0.164	0.134
att532	19071996	3547.388	2244.815	3504.358	1077.461	1191.602	303.004
berlin52	19071996	0.031	0.030	0.028	0.073	0.073	0.085
bier127	19071996	0.896	0.610	0.627	4.199	1.947	1.435
ch130	19071996	1.651	1.051	0.949	2.771	2.104	1.762
ch150	19071996	6.329	2.499	2.583	8.672	6.927	6.316
d198	19071996	55.018	5.765	5.581	23.319	11.724	14.634
gil262	19071996	168.740	34.147	32.721	46.932	42.738	26.112
kroA100	19071996	1.515	0.756	0.679	1.599	1.499	1.007
kroA150	19071996	8.992	4.686	3.257	17.055	8.977	7.263
kroA200	19071996	453.677	98.514	112.663	111.190	36.090	30.106
kroB100	19071996	1.847	0.760	0.710	5.167	2.665	1.836
kroB150	19071996	18.980	7.197	10.153	16.517	14.767	5.580
kroB200	19071996	16.922	4.777	4.853	14.375	10.629	8.862
kroC100	19071996	0.853	0.658	0.682	2.695	2.154	0.923
kroD100	19071996	0.627	0.658	0.551	1.610	1.815	0.914
kroE100	19071996	2.643	1.991	2.032	4.294	3.357	1.509

Continued on next page

Table 7.5 – continued from previous page

Instance	Seed	Lazy	UserCut	Heuristic	Lazy*	UserCut*	Heuristic*
lin105	19071996	0.532	0.256	0.219	0.854	0.640	0.609
lin318	19071996	93.532	33.001	21.742	87.572	30.366	38.257
pr226	19071996	16.314	4.834	4.878	25.345	17.512	13.138
rat195	19071996	29.843	13.269	12.237	34.013	24.517	13.680
pr76	19071996	19.724	7.595	4.206	4.712	7.046	6.066
pr107	19071996	0.052	0.049	0.088	0.087	0.092	0.144
pr124	19071996	3.873	1.828	1.899	5.513	3.154	5.490
pr136	19071996	1.953	1.352	1.729	2.491	2.855	2.662
tsp225	19071996	11.742	4.704	4.759	24.909	15.986	14.130
u159	19071996	2.977	0.769	0.808	4.774	2.629	3.573
rat99	19071996	0.496	0.352	0.353	0.539	0.546	0.580
rd400	19071996	3563.643	223.105	408.701	300.483	176.906	71.727

Table 7.6: **Exact Algorithms (* Generic Callback, ** Loop Methods)**

Instance	Lazy	UserCut	Heuristic	Lazy*	UserCut*	Heuristic*	Simple**	Heuristic**
Seed: 201909284								
a280	6.229	3.674	3.671	12.946	12.593	14.801	29.781	32.546
att48	0.095	0.069	0.057	0.184	0.187	0.209	0.681	0.594
att532	3532.022	3511.272	3504.440	2446.941	639.188	460.378	663.910	719.934
berlin52	0.028	0.029	0.029	0.077	0.096	0.080	0.095	0.063
bier127	1.219	0.592	0.626	3.442	1.739	1.149	4.407	5.622
ch130	1.389	0.990	0.915	4.504	4.221	1.673	5.103	5.557
ch150	5.351	2.396	1.571	9.392	6.332	6.024	11.128	15.697
d198	40.486	4.199	4.336	29.695	12.840	11.985	72.351	69.021
gil262	163.289	21.162	23.463	64.250	42.878	25.663	55.548	49.800
kroA100	1.183	0.758	0.678	2.967	2.888	0.898	4.419	4.651
kroA150	6.082	3.761	4.292	10.574	6.380	9.067	23.498	16.290
kroA200	209.747	41.006	78.697	69.231	25.079	28.455	64.854	82.079
kroB100	1.614	0.799	0.647	5.871	2.644	1.980	7.698	7.828
kroB150	20.045	9.197	8.712	15.901	6.044	6.368	34.584	31.630
kroB200	22.930	4.129	3.325	26.611	11.157	14.564	22.810	19.158
kroC100	0.804	0.670	0.666	1.204	1.121	0.981	4.956	3.526
kroD100	1.064	0.794	0.632	1.418	1.694	1.141	4.964	4.677
kroE100	3.159	1.957	3.083	4.858	3.181	1.501	4.540	4.475
lin105	0.497	0.210	0.218	1.444	0.801	0.648	3.567	4.633
lin318	103.086	22.930	26.731	67.349	25.349	29.021	97.536	110.049
pr226	27.445	4.292	4.380	29.891	18.124	15.023	197.544	260.199
rat195	43.155	21.471	14.749	32.835	33.280	12.177	69.132	47.261

Continued on next page

Table 7.6 – continued from previous page

Instance	Lazy	UserCut	Heuristic	Lazy*	UserCut*	Heuristic*	Simple**	Heuristic**
pr76	14.067	11.805	4.295	6.359	6.835	2.957	9.330	8.509
pr107	0.068	0.052	0.087	0.126	0.086	0.136	0.631	2.172
pr124	2.134	1.826	1.614	4.793	2.574	2.315	20.794	23.092
pr136	1.226	1.533	1.563	2.673	1.776	1.540	5.627	6.156
tsp225	40.637	4.016	2.634	21.783	13.709	9.678	32.324	29.078
u159	1.880	0.761	0.926	4.659	1.953	2.395	7.316	7.618
rat99	0.488	0.350	0.369	0.579	0.585	0.586	2.413	2.727
rd400	3551.366	461.552	564.476	197.624	131.576	113.149	305.766	456.781
Seed: 19								
a280	8.130	4.269	4.317	20.864	15.620	9.475	29.235	24.520
att48	0.089	0.071	0.056	0.174	0.181	0.145	0.593	0.671
att532	3535.718	1061.775	1233.420	2744.552	534.519	281.131	734.706	756.783
berlin52	0.028	0.028	0.026	0.073	0.072	0.086	0.097	0.065
bier127	0.896	0.785	0.663	3.684	2.627	1.267	4.456	5.910
ch130	1.438	1.083	1.128	5.851	4.489	1.615	5.183	6.728
ch150	7.742	3.105	2.110	12.946	8.178	4.030	10.200	17.139
d198	23.188	6.341	4.156	31.508	13.839	10.980	65.949	70.533
gil262	116.108	44.069	34.565	64.308	34.720	31.280	54.885	42.880
kroA100	1.222	0.869	0.612	1.984	1.401	0.906	4.190	4.722
kroA150	8.917	3.338	3.783	12.968	9.416	5.820	25.094	17.489
kroA200	110.787	36.201	36.140	67.595	33.454	17.821	82.620	84.170
kroB100	2.114	0.711	0.464	3.784	2.542	1.878	8.183	7.517
kroB150	28.851	6.063	7.115	18.894	13.727	4.816	33.041	29.433
kroB200	32.032	4.254	6.101	16.304	12.599	8.692	28.127	12.029
kroC100	0.844	0.722	0.656	2.578	1.514	1.061	4.825	3.915
kroD100	0.983	0.770	0.789	1.531	1.163	1.328	5.098	4.810
kroE100	2.466	1.265	1.430	4.791	3.421	1.445	4.362	4.274
lin105	0.419	0.212	0.236	0.657	0.601	0.761	2.935	4.384
lin318	198.618	23.924	22.958	55.896	25.346	21.318	95.144	103.283
pr226	18.191	4.840	4.886	28.894	15.489	15.526	375.243	245.419
rat195	51.779	11.243	12.493	43.827	33.000	15.290	63.654	72.675
pr76	13.690	11.829	3.883	6.057	8.389	4.631	8.932	9.417
pr107	0.052	0.053	0.088	0.086	0.087	0.137	0.578	3.075
pr124	7.439	2.004	1.831	6.104	3.381	4.564	19.232	19.579
pr136	1.416	1.803	1.726	2.411	2.316	2.354	5.519	7.685
tsp225	29.004	4.494	3.006	14.538	12.006	9.069	32.647	27.622
u159	2.805	0.844	1.090	5.335	2.706	2.917	7.472	8.665
rat99	0.460	0.470	0.302	0.610	0.592	0.598	1.829	3.217
rd400	3565.286	601.292	704.836	228.320	117.198	96.235	372.713	321.477
Seed: 190796								
a280	10.591	4.896	5.036	16.525	22.052	16.995	26.739	25.632

Continued on next page

Table 7.6 – continued from previous page

Instance	Lazy	UserCut	Heuristic	Lazy*	UserCut*	Heuristic*	Simple**	Heuristic**
att48	0.091	0.070	0.059	0.159	0.163	0.133	0.632	0.611
att532	3550.588	639.059	1448.989	2196.201	583.651	404.120	723.908	777.047
berlin52	0.031	0.031	0.028	0.067	0.068	0.069	0.092	0.166
bier127	0.994	0.590	0.626	3.782	1.934	1.298	4.320	5.399
ch130	2.110	1.267	0.924	4.024	3.708	1.990	5.034	7.076
ch150	6.682	1.663	1.957	12.687	4.771	3.670	12.824	18.325
d198	19.916	5.745	5.395	31.920	9.793	13.984	63.516	52.336
gil262	231.023	13.520	22.168	47.178	56.418	32.368	52.419	43.887
kroA100	1.269	0.751	0.739	2.773	2.263	1.023	4.025	4.328
kroA150	10.130	3.806	2.339	10.338	11.541	6.809	23.075	17.093
kroA200	243.669	96.150	62.074	21.538	52.614	22.347	84.148	78.703
kroB100	1.104	0.930	0.947	2.759	2.189	1.679	7.607	7.928
kroB150	23.558	6.720	6.849	12.962	14.925	4.931	34.306	33.014
kroB200	79.954	5.045	4.684	12.705	13.328	11.764	24.217	13.384
kroC100	0.998	0.645	0.657	2.144	1.681	0.946	4.926	3.206
kroD100	1.026	0.727	0.731	1.527	1.420	1.133	5.197	4.223
kroE100	2.600	1.667	1.776	3.471	4.130	1.416	5.123	4.568
lin105	0.573	0.214	0.213	0.926	0.647	0.722	3.519	2.837
lin318	211.935	48.083	17.926	176.928	30.363	33.881	99.489	91.392
pr226	19.142	5.849	5.889	113.334	16.882	18.258	181.923	555.863
rat195	43.599	15.382	15.497	30.761	19.633	16.996	59.865	50.482
pr76	12.072	8.309	4.701	6.822	9.637	4.680	10.316	8.644
pr107	0.052	0.052	0.088	0.086	0.088	0.138	0.555	2.458
pr124	2.698	2.273	2.232	9.535	2.334	4.686	21.904	23.429
pr136	2.104	1.412	1.259	2.576	3.237	2.173	6.314	6.603
tsp225	13.332	5.804	2.825	15.866	13.499	7.800	32.958	32.733
u159	2.792	0.754	0.896	5.308	2.526	1.916	7.975	6.037
rat99	0.639	0.381	0.427	0.505	0.507	0.569	2.170	2.127
rd400	3562.556	312.888	249.195	186.584	231.009	81.896	336.298	391.179
				Seed: 19071996				
a280	15.257	5.946	5.931	15.386	15.735	17.806	31.560	22.689
att48	0.093	0.070	0.056	0.160	0.164	0.134	0.668	0.679
att532	3547.388	2244.815	3504.358	1077.461	1191.602	303.004	656.097	696.500
berlin52	0.031	0.030	0.028	0.073	0.073	0.085	0.103	0.086
bier127	0.896	0.610	0.627	4.199	1.947	1.435	4.610	4.364
ch130	1.651	1.051	0.949	2.771	2.104	1.762	4.844	6.604
ch150	6.329	2.499	2.583	8.672	6.927	6.316	10.942	15.347
d198	55.018	5.765	5.581	23.319	11.724	14.634	71.141	67.801
gil262	168.740	34.147	32.721	46.932	42.738	26.112	61.853	45.738
kroA100	1.515	0.756	0.679	1.599	1.499	1.007	4.449	4.557
kroA150	8.992	4.686	3.257	17.055	8.977	7.263	27.511	15.670

Continued on next page

Table 7.6 – continued from previous page

Instance	Lazy	UserCut	Heuristic	Lazy*	UserCut*	Heuristic*	Simple**	Heuristic**
kroA200	453.677	98.514	112.663	111.190	36.090	30.106	59.241	81.205
kroB100	1.847	0.760	0.710	5.167	2.665	1.836	8.467	7.117
kroB150	18.980	7.197	10.153	16.517	14.767	5.580	37.819	27.418
kroB200	16.922	4.777	4.853	14.375	10.629	8.862	23.213	18.620
kroC100	0.853	0.658	0.682	2.695	2.154	0.923	4.877	3.207
kroD100	0.627	0.658	0.551	1.610	1.815	0.914	5.093	4.278
kroE100	2.643	1.991	2.032	4.294	3.357	1.509	4.483	3.851
lin105	0.532	0.256	0.219	0.854	0.640	0.609	3.633	4.231
lin318	93.532	33.001	21.742	87.572	30.366	38.257	95.643	99.793
pr226	16.314	4.834	4.878	25.345	17.512	13.138	156.506	146.272
rat195	29.843	13.269	12.237	34.013	24.517	13.680	67.494	63.071
pr76	19.724	7.595	4.206	4.712	7.046	6.066	8.173	9.800
pr107	0.052	0.049	0.088	0.087	0.092	0.144	0.552	1.506
pr124	3.873	1.828	1.899	5.513	3.154	5.490	22.828	18.270
pr136	1.953	1.352	1.729	2.491	2.855	2.662	6.206	8.585
tsp225	11.742	4.704	4.759	24.909	15.986	14.130	34.639	32.599
u159	2.977	0.769	0.808	4.774	2.629	3.573	7.453	7.642
rat99	0.496	0.352	0.353	0.539	0.546	0.580	2.339	2.207
rd400	3563.643	223.105	408.701	300.483	176.906	71.727	337.442	374.722

Below we report the number of nodes solved by the Callbacks. Algorithms with * are **Generic Callbacks**. A value of 0 indicates that the algorithm has solved the TSP problem in the root node.

Table 7.7: **Number of nodes solved by Callbacks**

Instance	Seed	Lazy	UserCut	Heuristic	Lazy*	UserCut*	Heuristic*
a280	201909284	130	0	0	145	0	0
att48	201909284	0	0	0	0	0	0
att532	201909284	64018	37015	42003	21651	1610	3825
berlin52	201909284	0	0	0	0	0	0
bier127	201909284	14	0	0	0	0	0
ch130	201909284	6	6	7	0	9	0
ch150	201909284	651	22	8	1625	34	18
d198	201909284	5722	25	23	4118	0	0
gil262	201909284	16411	797	837	1347	154	670
kroA100	201909284	71	0	0	0	0	0
kroA150	201909284	1180	136	164	2458	39	294
kroA200	201909284	44038	4811	9682	4914	1263	2419

Continued on next page

Table 7.7 – continued from previous page

Instance	Seed	Lazy	UserCut	Heuristic	Lazy*	UserCut*	Heuristic*
kroB100	201909284	380	5	0	2971	0	0
kroB150	201909284	4719	966	746	3394	225	270
kroB200	201909284	3982	45	4	1203	0	0
kroC100	201909284	4	0	0	0	0	0
kroD100	201909284	45	0	0	0	0	0
kroE100	201909284	1635	548	1092	613	98	151
lin105	201909284	3	0	0	0	0	0
lin318	201909284	4947	203	316	1214	113	166
pr226	201909284	1483	3	3	1519	0	15
rat195	201909284	6306	1027	603	3845	1793	452
pr76	201909284	17038	9313	3175	1880	3557	1793
pr107	201909284	0	0	0	0	0	0
pr124	201909284	542	15	8	375	0	0
pr136	201909284	71	5	5	0	0	0
tsp225	201909284	2090	3	0	1756	25	0
u159	201909284	64	14	0	904	0	0
rat99	201909284	0	5	5	0	0	0
rd400	201909284	168365	13418	16244	3991	1923	4720
a280	19	330	0	0	499	0	0
att48	19	0	0	0	0	0	0
att532	19	62351	15797	16684	28841	2842	2353
berlin52	19	0	0	0	0	0	0
bier127	19	0	0	0	0	0	0
ch130	19	6	7	8	0	0	0
ch150	19	1122	109	7	3555	0	0
d198	19	2955	77	32	3740	0	0
gil262	19	13659	2608	1916	1244	1699	1244
kroA100	19	45	0	0	0	0	0
kroA150	19	1710	124	151	2476	53	0
kroA200	19	16147	4123	4556	16191	2442	3655
kroB100	19	409	0	0	0	0	0
kroB150	19	8751	485	483	1281	5267	23
kroB200	19	6402	13	13	2412	0	0
kroC100	19	9	0	0	102	0	0
kroD100	19	95	0	0	0	0	0
kroE100	19	1432	133	269	1961	155	35
lin105	19	0	0	0	0	0	0
lin318	19	9035	247	243	1266	191	0
pr226	19	828	1	1	1531	0	38
rat195	19	8615	330	463	6041	1461	1324
pr76	19	18042	9269	2786	4370	5481	5299

Continued on next page

Table 7.7 – continued from previous page

Instance	Seed	Lazy	UserCut	Heuristic	Lazy*	UserCut*	Heuristic*
pr107	19	0	0	0	0	0	0
pr124	19	3331	18	15	1915	0	0
pr136	19	23	7	7	0	0	0
tsp225	19	3342	8	0	694	0	0
u159	19	409	14	4	278	0	0
rat99	19	1	3	0	0	0	0
rd400	19	155759	19627	20649	2653	2924	1722
a280	190796	483	39	39	344	329	0
att48	190796	0	0	0	0	0	0
att532	190796	43092	8935	18259	28768	1670	3046
berlin52	190796	0	0	0	0	0	0
bier127	190796	0	0	0	0	0	0
ch130	190796	11	6	7	0	58	0
ch150	190796	1325	11	7	2464	0	0
d198	190796	2144	28	47	4458	0	67
gil262	190796	24552	429	961	1378	1314	100
kroA100	190796	5	0	0	0	0	0
kroA150	190796	2160	105	46	1222	104	5
kroA200	190796	49216	14484	7812	1999	3611	2478
kroB100	190796	51	10	10	34	0	0
kroB150	190796	7683	366	510	2452	1227	23
kroB200	190796	17855	18	23	1245	379	14
kroC100	190796	7	0	0	0	0	0
kroD100	190796	47	0	0	0	0	0
kroE100	190796	1532	389	362	414	104	55
lin105	190796	14	0	0	0	0	0
lin318	190796	13835	789	105	1661	183	413
pr226	190796	1098	4	3	14797	0	0
rat195	190796	7774	554	819	4090	759	806
pr76	190796	16828	6879	3747	3166	5731	3824
pr107	190796	0	0	0	0	0	0
pr124	190796	569	10	24	2470	0	0
pr136	190796	53	5	5	0	0	0
tsp225	190796	982	5	0	855	0	0
u159	190796	299	14	0	0	0	0
rat99	190796	5	5	0	0	0	0
rd400	190796	167653	8429	5672	3839	5399	1247
a280	19071996	1080	39	42	225	58	59
att48	19071996	0	0	0	0	0	0
att532	19071996	32140	27825	44875	8934	6412	2669
berlin52	19071996	0	0	0	0	0	0

Continued on next page

Table 7.7 – continued from previous page

Instance	Seed	Lazy	UserCut	Heuristic	Lazy*	UserCut*	Heuristic*
bier127	19071996	0	0	0	0	0	0
ch130	19071996	13	11	11	0	0	0
ch150	19071996	1075	8	8	870	50	0
d198	19071996	8538	40	30	3749	69	27
gil262	19071996	18227	1867	1477	1604	862	488
kroA100	19071996	0	0	0	0	0	0
kroA150	19071996	1618	314	75	1233	0	133
kroA200	19071996	71703	13186	15121	11243	1246	2442
kroB100	19071996	602	4	0	1938	100	0
kroB150	19071996	5781	500	907	2456	1232	16
kroB200	19071996	2005	6	6	1251	159	0
kroC100	19071996	16	0	0	0	0	0
kroD100	19071996	33	0	0	162	0	0
kroE100	19071996	1239	497	441	470	62	59
lin105	19071996	3	0	0	0	0	0
lin318	19071996	4335	533	283	2748	305	471
pr226	19071996	816	3	3	1520	0	11
rat195	19071996	4607	758	658	3012	1269	194
pr76	19071996	24450	5499	3143	3874	2794	4531
pr107	19071996	0	0	0	0	0	0
pr124	19071996	1107	24	20	2096	0	0
pr136	19071996	94	3	3	61	0	0
tsp225	19071996	976	3	3	1936	0	3
u159	19071996	382	14	0	272	11	0
rat99	19071996	5	5	5	0	0	0
rd400	19071996	169473	6129	9471	6597	4514	2818

7.3 Matheuristics

In this section we report the objective function values of the Matheuristic algorithms.

Table 7.8: **Hard Fixing**

Instance	**Time Limit** (min)	**Time*** (sec)	**Objective function**	**Optimum**	**Gap** (%)
d657	30	1082.196	48907	48907	0.00
d1291	30	-	51382	50801	1.13
fl417	30	430.301	11861	11861	0.00
fl1400	30	-	20784	20127	3.16
fl1577	30	-	22600	22204	1.75
nrw1379	30	-	57410	56638	1.34
p654	30	29.874	34643	34643	0.00
pcb1173	30	-	57939	56892	1.80
pcb3038	30	-	145539	137694	5.39
pr1002	30	573.221	259045	259045	0.00
pr2392	30	-	394861	378032	4.26
rl1304	30	-	254441	252948	0.59
rl1323	30	-	278498	270199	2.98
rl1889	30	-	333773	316536	5.16
u724	30	526.134	41908	41908	0.00
u1060	30	1801.107	224094	224094	0.00
u2152	30	-	66062	64253	2.74
u2319	30	540.117	234256	234256	0.00
vm1084	30	-	243560	239297	1.75
vm1748	30	-	347571	336556	3.17

Time*: time taken to compute the optimal solution.

Table 7.9: **Local Branching**

Instance	**Time Limit** (min)	**Time*** (sec)	**Objective function**	**Optimum**	**Gap** (%)
d657	30	812.298	48907	48907	0.00
d1291	30	-	52728	50801	3.65
fl417	30	360.047	11861	11861	0.00
fl1400	30	-	21036	20127	4.32
fl1577	30	-	22707	22204	2.22
nrw1379	30	-	57944	56638	2.25
p654	30	32.424	34643	34643	0.00
pcb1173	30	-	58610	56892	2.93

Continued on next page

Table 7.9 – continued from previous page

Instance	**Time Limit** (min)	**Time*** (sec)	**Objective function**	**Optimum**	**Gap** (%)
pcb3038	30	-	144274	137694	4.56
pr1002	30	1801.185	259045	259045	0.00
pr2392	30	-	396768	378032	4.72
rl1304	30	-	259250	252948	2.43
rl1323	30	-	280767	270199	3.76
rl1889	30	-	333128	316536	4.98
u724	30	982.572	41908	41908	0.00
u1060	30	-	226185	224094	0.92
u2152	30	-	66867	64253	3.90
u2319	30	915.828	234256	234256	0.00
vm1084	30	-	242896	239297	1.48
vm1748	30	-	348641	336556	3.47

Time*: time taken to compute the optimal solution.

Table 7.10: **Comparison between objective functions**

Instance	**Optimum**	**Hard Fixing**		**Local Branching**	
		Obj. Function	Gap	Obj. Function	Gap
d657	48907	48907	0.00	48907	0.00
d1291	50801	51382	1.13	52728	3.65
fl417	11861	11861	0.00	11861	0.00
fl1400	20127	20784	3.16	21036	4.32
fl1577	22204	22600	1.75	22707	2.22
nrw1379	56638	57410	1.34	57944	2.25
p654	34643	34643	0.00	34643	0.00
pcb1173	56892	57939	1.80	58610	2.93
pcb3038	137694	145539	5.39	144274	4.56
pr1002	259045	259045	0.00	259045	0.00
pr2392	378032	394861	4.26	396768	4.72
rl1304	252948	254441	0.59	259250	2.43
rl1323	270199	278498	2.98	280767	3.76
rl1889	316536	333773	5.16	333128	4.98
u724	41908	41908	0.00	41908	0.00
u1060	224094	224094	0.00	226185	0.92
u2152	64253	66062	2.74	66867	3.90
u2319	234256	234256	0.00	234256	0.00
vm1084	239297	243560	1.75	242896	1.48
vm1748	336556	347571	3.17	348641	3.47

7.4 Stand-alone heuristics

In this section we report the tables with the results and their details we obtained on the tests of the different algorithms.

Table 7.11: **Multi-start**

Instance	**Time Limit** (min)	**Objective function**	**Optimum**	**Gap** (%)
d657	30	51608	48907	5.23
d1291	30	56577	50801	10.21
fl417	30	12176	11861	2.59
fl1400	30	21291	20127	5.47
fl1577	30	23953	22204	7.30
nrw1379	30	61766	56638	8.30
p654	30	36361	34643	4.70
pcb1173	30	62375	56892	8.79
pcb3038	30	154679	137694	10.98
pr1002	30	281778	259045	8.07
pr2392	30	422804	378032	10.59
rl1304	30	276760	252948	8.60
rl1323	30	298429	270199	9.46
rl1889	30	348391	316536	9.14
u724	30	45316	41908	7.52
u1060	30	240853	224094	6.96
u2152	30	73356	64253	12.40
u2319	30	249621	234256	6.16
vm1084	30	260289	239297	8.06
vm1748	30	368113	336556	8.57

Table 7.12: **VNS**

Instance	**Time Limit** (min)	**Objective function**	**Optimum**	**Gap** (%)
d657	30	49921	48907	2.03
d1291	30	52217	50801	2.71
fl417	30	11970	11861	0.91
fl1400	30	20657	20127	2.57
fl1577	30	22692	22204	2.15
nrw1379	30	59272	56638	4.44

Continued on next page

Table 7.12 – continued from previous page

Instance	**Time Limit** (min)	**Objective function**	**Optimum**	**Gap** (%)
p654	30	34832	34643	0.54
pcb1173	30	58668	56892	3.02
pcb3038	30	149738	137694	8.04
pr1002	30	264881	259045	2.20
pr2392	30	399147	378032	5.29
rl1304	30	255863	252948	1.14
rl1323	30	275537	270199	1.93
rl1889	30	324641	316536	2.50
u724	30	43054	41908	2.66
u1060	30	230070	224094	2.60
u2152	30	69208	64253	7.16
u2319	30	242850	234256	3.54
vm1084	30	244211	239297	2.01
vm1748	30	350364	336556	3.94

Table 7.13: **Tabu Search**

Instance	**Time Limit** (min)	**Objective function**	**Optimum**	**Gap** (%)
d657	30	50222	48907	2.62
d1291	30	58274	50801	12.82
fl417	30	12339	11861	3.87
fl1400	30	21751	20127	7.46
fl1577	30	24728	22204	10.20
nrw1379	30	59349	56638	4.57
p654	30	37672	34643	8.04
pcb1173	30	59146	56892	3.81
pcb3038	30	146111	137694	5.76
pr1002	30	285259	259045	9.19
pr2392	30	423364	378032	10.70
rl1304	30	281595	252948	10.17
rl1323	30	293050	270199	7.80
rl1889	30	350028	316536	9.57
u724	30	45604	41908	8.10
u1060	30	243788	224094	8.08
u2152	30	74865	64253	14.17
u2319	30	249119	234256	5.97
vm1084	30	264646	239297	9.58
vm1748	30	372908	336556	9.75

Table 7.14: **Simulated Annealing**

Instance	**Time Limit** (min)	**Objective function**	**Optimum**	**Gap** (%)
d657	30	50853	48907	3.83
d1291	30	55631	50801	8.68
fl417	30	11960	11861	0.83
fl1400	30	21011	20127	3.61
fl1577	30	23208	22204	4.33
nrw1379	30	60290	56638	6.06
p654	30	36663	34643	5.51
pcb1173	30	61274	56892	7.15
pcb3038	30	160064	137694	13.98
pr1002	30	270144	259045	4.11
pr2392	30	412826	378032	8.43
rl1304	30	275741	252948	8.27
rl1323	30	286098	270199	5.56
rl1889	30	353047	316536	10.34
u724	30	44111	41908	4.99
u1060	30	236150	224094	5.10
u2152	30	72906	64253	11.87
u2319	30	243034	234256	3.61
vm1084	30	252216	239297	5.12
vm1748	30	369762	336556	8.98

Table 7.15: **Genetic Algorithm**

Instance	**Time Limit** (min)	**Objective function**	**Optimum**	**Gap** (%)
d657	30	51590	48907	5.20
d1291	30	56405	50801	9.94
fl417	30	12030	11861	1.40
fl1400	30	21317	20127	5.58
fl1577	30	26054	22204	14.78
nrw1379	30	63077	56638	10.21
p654	30	35033	34643	1.11
pcb1173	30	62303	56892	8.69
pcb3038	30	215382	137694	36.07
pr1002	30	276044	259045	6.16

Continued on next page

Table 7.15 – continued from previous page

Instance	**Time Limit** (min)	**Objective function**	**Optimum**	**Gap** (%)
pr2392	30	546694	378032	30.85
rl1304	30	284911	252948	11.22
rl1323	30	306511	270199	11.85
rl1889	30	457862	316536	30.87
u724	30	44930	41908	6.72
u1060	30	240476	224094	6.81
u2152	30	93952	64253	31.61
u2319	30	284055	234256	17.53
vm1084	30	255868	239297	6.48
vm1748	30	432545	336556	22.19

Table 7.16: **Comparison between objective functions**

Instance	**Optimum**	**Multi-start**	**VNS**	**Tabu Search**	**Simulated Annealing**	**Genetic Algorithm**
d657	48907	51608	49921	50222	50853	51590
d1291	50801	56577	52217	58274	55631	56405
fl417	11861	12176	11970	12339	11960	12030
fl1400	20127	21291	20657	21751	21011	21317
fl1577	22204	23953	22692	24728	23208	26054
nrw1379	56638	61766	59272	59349	60290	63077
p654	34643	36361	34832	37672	36663	35033
pcb1173	56892	62375	58668	59146	61274	62303
pcb3038	137694	154679	149738	146111	160064	215382
pr1002	259045	281778	264881	285259	270144	276044
pr2392	378032	422804	399147	423364	412826	546694
rl1304	252948	276760	255863	281595	275741	284911
rl1323	270199	298429	275537	293050	286098	306511
rl1889	316536	348391	324641	350028	353047	457862
u724	41908	45316	43054	45604	44111	44930
u1060	224094	240853	230070	243788	236150	240476
u2152	64253	73356	69208	74865	72906	93952
u2319	234256	249621	242850	249119	243034	284055
vm1084	239297	260289	244211	264646	252216	255868
vm1748	336556	368113	350364	372908	369762	432545