UNIVERSITÀ DEGLI STUDI DI PADOVA

Operations Research 2 - 2021/2022

Traveling Salesman Problem: Solution Strategies

Supervisor:
Prof. Matteo Fischetti

Candidates:
Federico Michelotto 1218322
Mikele Milia 1218949

14th October 2021

Contents

Chapter 1

Introduction

The purpose of this paper is to present, analyze and compare different approaches to solve the *Traveling Salesman Problem* (TSP). The TSP is one of the most relevant problems in combinatorial optimization and consists in finding a Hamiltonian circuit of minimum cost on a given directed graph $G = (V, A)$. In some cases, the problem can be analogously defined on a undirected graph and this happens when the cost associated with an arc does not depend on its orientation.

The TSP arises naturally in an extremely wide number of practical applications, unfortunately it belongs to the class of $\mathcal{NP}$-hard problems, which means that no polynomial time algorithms are known to solve this problem. To find the optimal solution it is not necessary to calculate the cost of every possible solution. In fact, TSP can be formulated as a linear integer programming problem, and solved by using techniques such as branch-and-bound and branch-and-cut. Nevertheless, the time required to solve an arbitrary instance of the problem can, at worst, increase exponentially with the size of the instance, making an exact algorithm unpractical for many applications. In these situations we can still find approximate solutions through the so called Heuristics algorithms.

Our objective is to determine the better solution strategy for a given TSP instance. Specifically, the following algorithms will be presented and analyzed: Exact algorithms, which seek the optimal solution of a given instance; Matheuristic algorithms, that exploits the characteristics of the branch-and-cut tree to improve a non-optimal solution by exploring its neighborhood; Heuristic algorithms, that produce and improve an approximate solution; Metaheuristic algorithms, that combine different heuristic techniques to make the most of their features.

In our work, we used IBM ILOG CPLEX [1] as Integer Linear Programming (ILP) solver, Gnuplot [14], an open source command-line driven multi-platform plotting program, and Concorde [13], an optimized library used for the resolution of TSP-like problems. Experimental runs of our algorithms have been made with University of Padova Department of Information Engineering's computing cluster Blade.

As input data we used a set of instances taken from the TSPLIB [10], a famous library of sample instances for the TSP taken from various sources and of various types. Accordingly, the paper is organized as follows. In Chapter 2 after a brief background history of the TSP, we present its mathematical model, both in the symmetrical and asymmetrical scenarios. In Chapter 3, we analyze exact algorithms, introducing compact models and studying two state-of-the-art techniques, Benders and Lazy Constraints Callback methods, for finding optimal solution to TSP instances. In Chapter 4 we discuss about matheuristic algorithm also known as Hard and Soft Fixing. In chapter 5 we examine heuristic algorithms, in detail constructive and refining one. In the last but not least Chapter 6 we propose metaheuristic algorithm such as Tabu Search and Genetic. Finally in chapter 7 we provide our final considerations, regarding what has been discussed throughout this paper.

Chapter 2

Traveling Salesman Problem

In this chapter we briefly present the history of how the problem known today as *Traveling Salesman Problem* was born, then we define and analyze symmetric and asymmetric TSP variants of it.

First we give a formal definition of the two variants of this problem and then we formulate it as an integer linear programming problem. We decide to treat the asymmetric case as a more general version than the symmetric one. Is it useful to note that the STSP can also be formulated and solved as an ATSP after transforming the undirected graph $G = (V, E)$ into a directed graph $G = (V, A)$ with two arcs for every edge. In particular, for every edge $e = [i, j]$[1] $\in E$ incident in nodes $i, j \in V$, one can create two arcs (i, j) and (j, i) with cost $c_{ij} = c_{ji} = c_e$. Since we will assume that each edge has a positive cost, we can discard self-loops, i.e. edges that connect a node to itself.

Although in this chapter we present both cases, throughout this paper we focus on the STSP treating the ATSP only when we talk about compact models (Section 3.1).

2.1 Behind the Problem

As stated in [12] the origins of the traveling salesman problem are unclear. As evidence, an 1832 traveling salesmen's manual first mentions the problem, including some examples of travel through Germany and Switzerland, but contains no mathematical treatment.

The traveling salesman problem, was formulated mathematically in the late 1800s by W.R. Hamilton and Thomas Kirkman. Through the aid of the Icosaian game invented in 1857 by W.R. Hamilton himself, whose purpose is to find a Hamiltonian cycle along the edges of a dodecahedron such that each vertex is visited only once, and the endpoint is the same as the starting point. The problem was first studied and considered mathematically in the 1930s by Merrill M. Flood, Hassler Whitney, and especially Karl Menger, who defines it as follows:

> We denote by the messenger problem the task of finding, for a finite number of points whose pairwise distances are known, the shortest path connecting the points. Of course, this problem is solvable with a finite number of trials. No rules are known that push the number of trials below the number of permutations of the given points. The rule that one should go first from the starting point to the nearest point, then to the nearest point to this one, etc. generally does not give the shortest path.

The first publication to use the phrase "traveling salesman problem" was the 1949 RAND Corporation report by Julia Robinson [11]. In the 1950s and 1960s, the problem became increasingly popular in

[1]Notation used to indicate edges not oriented.

scientific circles in Europe and the United States after the RAND Corporation in Santa Monica offered prizes for steps in solving the problem.

Notable contributions were made by George Dantzig, Delbert Ray Fulkerson, and Selmer M. Johnson of the RAND Corporation, who expressed the problem as an integer linear program and developed the cutting plane method for its solution. They wrote what is considered the seminal paper on the subject, in which using these new methods they solved an instance with 49 cities optimally by constructing one round and showing that no other round could be shorter. Dantzig, Fulkerson, and Johnson, however, hypothesized that given a near-optimal solution, we might be able to find optimality or prove optimality by adding a small number of extra inequalities (cuts). While this paper did not provide an algorithmic approach to TSP problems, the ideas in it were indispensable for later creating exact solution methods for TSP, even though it would take 15 years to find an algorithmic approach in creating these cuts. In addition to cutting plane methods, Dantzig, Fulkerson, and Johnson used branch and bound algorithms for perhaps the first time.

Later in 1972 Richard M. Karp proved that the Hamiltonian cycle problem was $\mathcal{NP}$-complete, implying $\mathcal{NP}$-hardness of TSP. This provided a mathematical explanation for the apparent computational difficulty of finding optimal paths. Great progress were also made in the late 1970s and early 1980s, when Grötschel, Padberg, Rinaldi, and others were able to solve exact instances with up to 2,392 cities, using cut and branch and bound plans.

In the 1990s, Applegate, Bixby, Chvátal, and Cook developed the Concorde program that has been used in many recent solutions of record. Gerhard Reinelt published TSPLIB in 1991, a collection of reference instances of varying difficulty, which has been used by many research groups to compare results.

2.2 Symmetric Traveling Salesman Problem

The *Symmetric Traveling Salesman Problem* (STSP) is defined on a complete, weighted, not oriented graph $G = (V, E)$, where each edge $e \in E$ is associated with a cost c_e. Given a set of $|V| = n$ nodes and distances for each pair of them, the problem is to find the shortest (of minimal length) undirected tour which allows each node to be visited exactly once.

A basic formulation for the STSP problem, uses the following binary decision variables:

$$x_e = \begin{cases} 1 & \text{if edge } e \in E \text{ is chosen in the optimal circuit} \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

The model we can obtain, using such variables is defined as follows:

$$\min \sum_{e \in E} c_e x_e \tag{2.2}$$

$$\sum_{e \in \delta(v)} x_e = 2, \qquad \forall\, v \in V \tag{2.3}$$

$$x_e \in \{0, 1\} \text{ integer}, \qquad \forall\, e \in E \tag{2.4}$$

In the proposed basic model, equation (2.2) defines the objective function, (2.3) is the degree equation for each vertex, (2.4) defines the integrality constraints. Since the number of constraints is polynomial this model can be referred as compact, nevertheless it is not complete since at the moment the proposed optimal solution contains several disconnected tours (or islands) as shown in figure 2.1. To forbid solutions consisting of several disconnected tours, it is necessary to add some constraints for subtours[2] elimination.

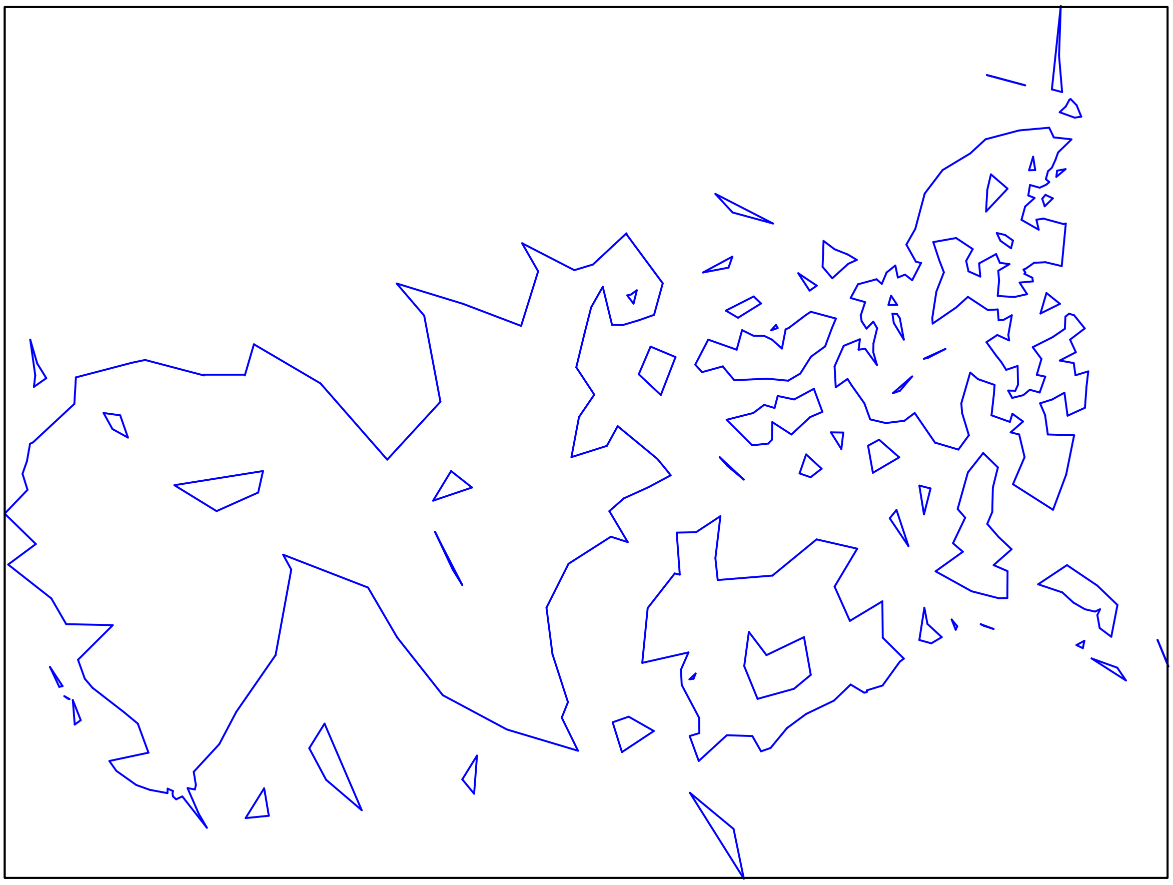


Figure 2.1: TSP solution of the att532 problem returned by the Symmetric TSP model solved with CPLEX. The model is defined only by constraints (2.2), (2.3) and (2.4). As you can see the solution contains many subtours.

The *Subtour Elimination Constraints* (a.k.a. SECs) can be formulated in different ways. One of the most efficient approach to exclude subtours, derives from the paper of Dantzig, Fulkerson and Johnson [2]. The SEC condition stated in that paper is formulated as:

$$\sum_{e \,\in\, E(S)^3} x_e \leq |S| - 1, \qquad \forall\, S \subset V : |S| \geq 3 \tag{2.5}$$

Such condition implies that the number of edges which can be packed in the clique defined by the set of nodes S cannot exceed $|S| - 1$. And given the degree-constraints of the basic formulation (2.3), this means that cannot be created any tour of nodes $S \subset V$. Since in this case we are dealing with undirected graphs, there cannot exist subtours with only 1 or 2 nodes, for this reason in (2.5) we are considering only the subsets S of V with at least 3 nodes.
The drawback is that this formulation introduces 2^{n-1} constraints and $n(n-1)$ binary variables. The exponential number of constraints makes it impractical to solve directly the model. Hence, the usual procedure is to apply the subtour elimination constraints (2.5) only when violated. This approach will be presented in the next chapter.

[2]**Subtour:** closed path that returns back to where it starts, but without visiting all the nodes.

[3]$E(S) := \{[i,j] \in E : i,j \in S\}$.

2.3 Asymmetric Traveling Salesman Problem

The *Asymmetric Traveling Salesman Problem* (ATSP) is defined on a complete, weighted, oriented graph $G = (V, A)$, where each arc $(i, j) \in A$ is associated with a cost c_{ij}. Given a set of $|V| = n$ nodes and distances for each pair of them, the problem is to find the shortest (of minimal length) undirected tour which allows each node to be visited exactly once. In this particular case, the distance from node i to node j and the distance from node j to node i may be different.

A basic formulation for the ATSP problem, uses the following binary decision variables:

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

The model we can obtain, using such variables is defined as follows:

$$\min \sum_{(i,j)\in A} c_{ij} x_{ij} \tag{2.6}$$

$$\sum_{i:(i,j)\in A} x_{ij} = 1, \qquad \forall\, j \in V \tag{2.7}$$

$$\sum_{j:(i,j)\in A} x_{ij} = 1, \qquad \forall\, i \in V \tag{2.8}$$

$$x_{ij} \in \{0,1\} \text{ integer}, \qquad \forall\, (i,j) \in A \tag{2.9}$$

As in the STSP case, this model is not complete. As a matter of fact, a SEC condition is needed to avoid the formation of subtours. Condition (2.5) can be used in the ATSP case as well, and can be rewritten as shown in equation (2.10).

$$\sum_{(i,j)\,\in\, A(S)^{4}} x_{ij} \leq |S| - 1, \qquad \forall\, S \subset V, |S| \geq 2 \tag{2.10}$$

In order to avoid the exponential number of constraints added by the Dantzig formulation, we will present alternative techniques in section 3.1 which belong to the compact formulations.

[4] $A(S) := \{(i,j) \in A : i, j \in S\}$.

Chapter 3

Exact algorithms

Exact algorithms are designed to find the optimal solution to the TSP, that is, the tour of minimal length. They are computationally expensive because they must (implicitly) consider all solutions in order to identify the optimum. These exact algorithms are typically derived from the integer linear programming formulation of the TSP.

The algorithms discussed in this section use CPLEX to find the optimal solution for the TSP. Keeping in mind that we are working with a $\mathcal{NP}$-hard problem, the algorithms have been tested with relatively “small” size instances. We first study compact models in section 3.1, which are able to solve STSP and ATSP instances using a polynomial number of variables and constraints. Then, we introduce more efficient techniques such as Benders’ method and Lazy Constraints Callback respectively in section 3.2 and 3.3.

3.1 Compact Formulations

In order to avoid the exponential number of constraints introduced by condition (2.5) or (2.10), we can instead use a so-called compact formulation of the TSP. A formulation of the Traveling Salesman Problem as an integer linear programming problem is compact if it introduces a polynomial number of both variables and constraints. In the following sections we will present two of these formulations, the Sequential (3.1.1) proposed by *Miller, Tucker and Zemlin* (MTZ) in the 1960, and the Flow Based (3.1.2) proposed by *Gavish and Graves* (GG) in the 1978. In addition we briefly present two useful constraints that can be added to the following compact models in order to speed-up their computation.

3.1.1 Sequential Formulations

MTZ

The idea behind the *Miller, Tucker and Zemlin* formulation is to associate each node to its position inside the tour. Let u_i be the position of node i in the tour, and let suppose that our tour must start from node 1 (first node), so that $u_1 = 0$. We want to impose that if the arc (i, j) is selected (with $i \neq j$) then $u_j = u_i + 1$. To implement this logical implication we apply the Big-M method, in order to activate it only when $x_{ij} = 1$, or in other terms, only if the arc (i, j) is selected. This translates to the following constraints:

$$u_j \geq u_i + 1 - M(1 - x_{ij}) \quad \forall\, i,j \in V - \{1\} \tag{3.1}$$

$$0 \leq u_i \leq n - 1 \quad \text{integer}, \quad \forall\, i \in V - \{1\} \tag{3.2}$$

$$u_1 = 0. \tag{3.3}$$

The definition of the M value plays a crucial role, in fact we don't want an M too large, otherwise we would have a loose formulation of the problem (and hence of the search space) and consequently a much higher computation to find the optimal solution. So we can ask ourselves:

> *Which is the smallest M such that the feasible tours for this problem are not cut off by the Big-M constraint?*

The smallest valid M we can be computed through the evaluation of the Big-M constraint on the *most critical* arc, which is the arc (i, j) with $x_{ij} = 0$ (i.e. not selected), $u_i = N - 1$ and $u_j = 1$, as shown in figure 3.1. This specific arc is the not selected one with the largest gap between u_i and u_j.

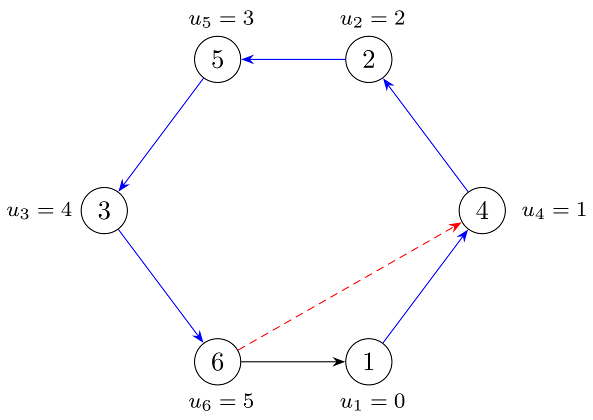


Figure 3.1: Visual example of Big-M constraint computation. Blue arcs represent selected arcs (i.e. the arc (i, j) with $x_{ij} = 1$). The *most critical* arc that occur in the tour is represented by the red dashed line, in this case is the arc $(6, 4)$. For simplicity the other arcs not selected are not shown.

If we analyze the Big-M constraint for this arc, we have the following system of equations for $i = 6$ and $j = 4$:

$$\begin{cases} u_4 \geq u_6 + 1 - M \\ u_4 = 1 \\ u_6 = N - 1 \end{cases} \tag{3.4}$$

whose solution is $M \geq N - 1$. As a counter example you can see that if we let M be equal to $N - 2$ we get

$$u_4 \geq u_6 + 1 - M = N - (N - 2) = 2$$

but this clearly contradicts our hypothesis of $u_4 = 1$. Note that the arc we are considering is the most critical one because if we consider for example a different arc (v, w) with $u_v - u_w \leq N - 2$, and let $M = N - 2$, the Big-M constraint on this arc would not cause any contradiction.

As a side note, since we are dealing with complete graphs, there will be always an arc (i, j) not selected with $u_i = N - 1$ and $u_j = 1$, and this means that if M is set less than $N - 1$ all feasible solutions would be cut off by this ill defined constraint.

3.1.2 Flow Based Formulations

Single Commodity Flow

The *Single Commodity Flow* formulation presented in 1978 by Gavish and Graves, is part of a more general family of formulations called flow based formulations. The idea behind the single commodity flow formulation is to have an initial number of commodities to spend in order to form a TSP solution. To visit a new node a commodity must be spent. In this way if we have $N-1$ commodities available at the starting node and we visit each node just once, we will end up with no commodities left when all nodes have been visited.

To do so, Gavish and Graves defined the flow variables y_{ij} one for each arc (i,j), where each y_{ij} represent the amount of commodities (flow) that pass through that specific arc. The formulation of the Gavish and Graves model (GG) is the following:

$$\sum_{i \in V} y_{ih} - \sum_{j \in V} y_{hj} = 1 \quad \forall\, h \in V - \{1\} \tag{3.5}$$

$$0 \le y_{ij} \le (N-2)x_{ij} \quad \text{integer}, \forall\, i,j \in V - \{1\} \tag{3.6}$$

$$y_{1j} = (N-1)x_{1j} \quad \text{integer}, \forall\, j \in V - \{1\} \tag{3.7}$$

$$y_{i1} = 0 \quad \text{integer}, \forall\, i \in V - \{1\} \tag{3.8}$$

The $N-1$ constraints (3.5) impose that the difference between the in-flow and the out-flow of a generic node $h \in V - \{1\}$ must be equal to 1. The $(N-1)^2$ constraints in (3.6) force the flow of the arcs (i,j) to zero in case $x_{ij} = 0$. That is, the arcs that are not selected have no flow passing through it, or alternatively, only the selected arcs may have a positive flow. These constraints are also called *linking constraints* since they link the x_{ij} variables to the y_{ij} flow variables. While the constraints (3.7) and (3.8) impose the incoming and outgoing flow relative to node 1.

As a side note, we would like to point out that this formulation (i.e. the one illustrated in class) is not exactly the one presented in [6], that is shown below:

$$\sum_{i \in V} y_{ih} - \sum_{j \in V} y_{hj} = 1 \quad \forall\, h \in V - \{1\} \tag{3.9}$$

$$0 \le y_{ij} \le (N-1)x_{ij} \quad \text{integer}, \forall\, i,j \in V \tag{3.10}$$

$$\sum_{j \in V - \{1\}} y_{1j} = N-1 \tag{3.11}$$

The difference with the previous model is how the flow variables corresponding to the incoming and outgoing arcs of node 1 are handled. In fact, the constraints (3.10), (3.11) in combination with the basic constraints of the TSP formulation (2.3) impose that if an arc x_{1j} is selected, then, the flow along that arc y_{1j} must be equal to $N-1$. That is equivalent to the constraint (3.7) of the previous model. Knowing this, it is easy to see that the in-flow/out-flow constraints (3.9), the original linking constraints (3.10), and the basic constraints of the TSP formulation (2.3) impose that the incoming flow to the initial node must be equal to 0, that is, constraints (3.8).

3.1.3 Additional Constraints

Lazy Constraints

Don't go crazy. Embrace being lazy.

Instead of applying all the subtour elimination constraints at once, it could be useful to use a lazy approach. When we talk about lazy approaches, we are referring to the fact that a specific constraint can be applied only when necessary and not before needed. In this way, a constraint defined lazy is not specified in the constraint matrix of the MIP problem, but in any case must be not violated in a feasible solution. Using lazy constraints makes sense when there are a large number of constraints that must be satisfied, but are unlikely to be violated if they are left out.

Degree Two Subtour Elimination Constraints

Sometimes it may be useful to add within the model subtour elimination constraints of degree two (3.12), which remove from the solution subtours created by two nodes i and j.

$$x_{ji} + x_{ij} \leq 1 \tag{3.12}$$

These types of constraints can be useful for speeding up the computation of a solution that is feasible within a directed graph configuration.

3.2 Benders' Method

As discussed above, CPLEX MIP solver must have the subtour elimination constraints (2.5) to find an exact solution for a TSP instance. Without using compact models, it is mandatory to find another way to avoid the exponential growth of such constraints.

A method worth mentioning is the Benders' one, which iteratively focuses on those subtours which are selected by CPLEX without any subtours elimination constraint and forbid it to choose them a second time. Specifically, as you can see in figure 3.2, after CPLEX returns a solution of the current model, Benders' method aim to find each subtours within it. Then for each subtour found, a constraint able to remove it is added to the current model. After all SECs have been added, the solver is restarted with the updated model, in order to produce an improved solution which does not allow the presence of subtours already found in previous iterations of the method. When the solution returned by CPLEX does not contains subtours, it means that the optimal solution is found.

Formally, whenever the solution returned by CPLEX contains m subtours S_k with $k \in \{1, ..., m\}$, Benders' method update the current model adding the following SECs:

$$\sum_{e \in E(S_k)} x_e \leq |S_k| - 1 \quad \forall\, k \in \{1, ..., m\} \tag{3.13}$$

3.2.1 Subtour Detection

A key aspects of Benders' method is an efficient way to detect connected components (subtours) within a solution. To achieve this goal, many ideas can be exploited such as the use of union finding data structures, Kruskal's method or the method presented by Fischetti. Kruskal's method and the one presented by Fischetti were implemented in this report.

Specifically, in order to use Kruskal's method, which is mostly used to find the minimum spanning tree of a connected graph, some adjustments are required to make it able to identify isolated connected components. Whereas, the method presented by Fischetti iteratively generates every single connected component through the following idea which is simple and efficient. Starting at node zero, all edges must be scanned in order to determine which ones are incident to the current node within the solution. Since the aim is to get an hamiltonian circuit, it is useful to exploit the fact that at most there will be two edges with $x_{ij} = 1$ in the solution returned by CPLEX. So, after selecting which will be the next, the current node is moved to the selected one and the operation described above is repeated iteratively. It is useful to notice that if a node has already been inserted within the connected component, it can be skipped and therefore implicitly assigned an orientation to the component. If some nodes remain disconnected after a component is found, the first available disconnected node becomes the starting node of the new component and the whole procedure is repeated from scratch, avoiding taking into account nodes already connected.

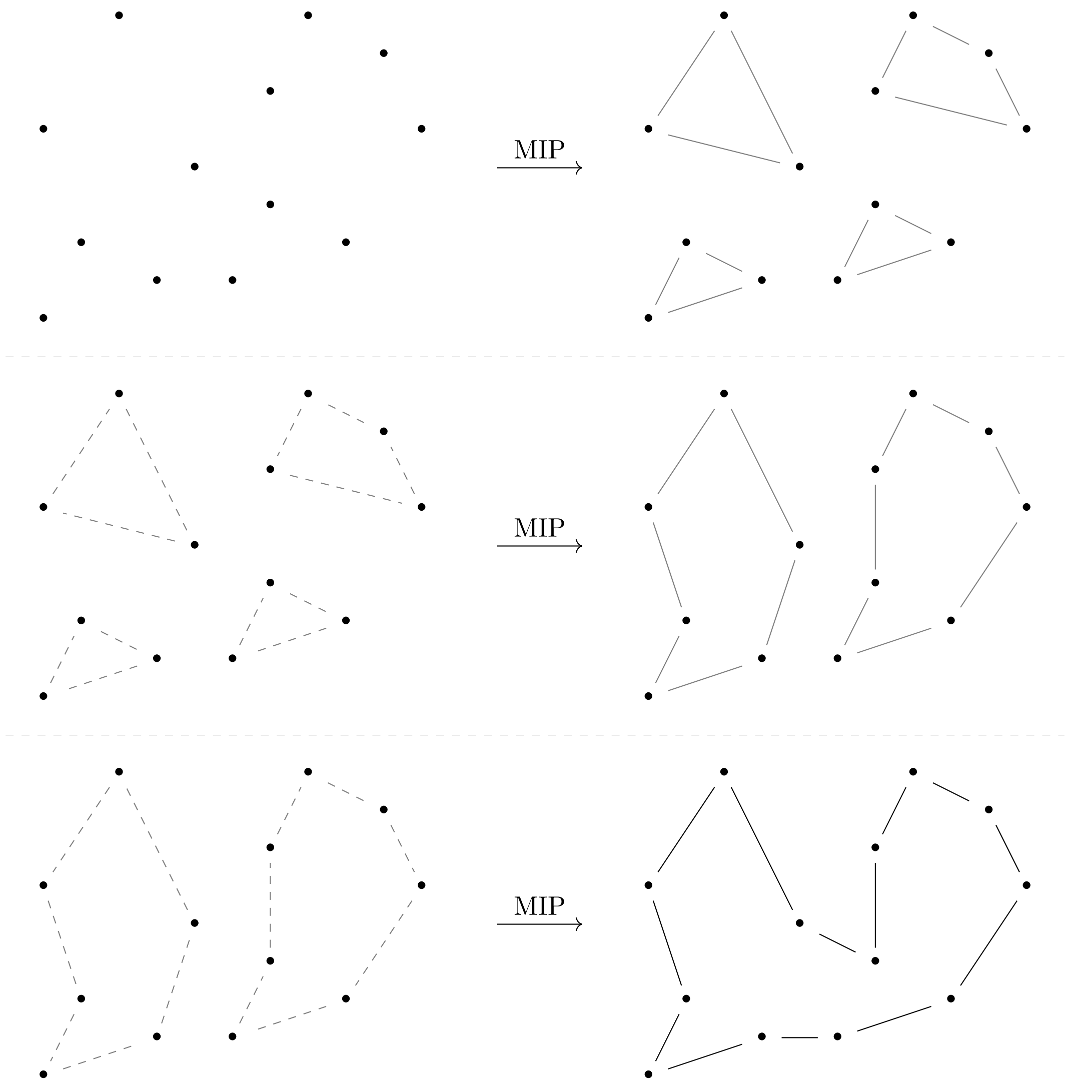


Figure 3.2: Visual explanation of Benders' method. After providing to CPLEX the base model without SEC, whenever a feasible solution is found, subtours detection is carried out and if subtours are present, specific constraints to remove them are added to the base model. Iteratively CPLEX uses the updated model to find an improved feasible solution. When no more subtours are present, Benders' method terminates and returns an hamiltonian cycle, which is the optimal solution for the tested instance.

3.3 Callback Method

An elegant way to solve exactly a TSP instance is to exploit the callback mechanism available in CPLEX to intercept the solutions found during the execution of the branch-and-cut algorithm. In this section we will describe two types of callbacks, the lazy constraint callbacks and the user cut callbacks. A lazy constraint callback is activated whenever an integer solution (also called candidate solution) is found, and it is used to reject the solutions that are infeasible applying one or more lazy constraints (already discussed in 3.1.3).

A user cut callback is instead activated whenever a relaxed LP fractional solution is found during the execution of the cut loop inside the branch-and-cut algorithm of CPLEX. Through this callback, we can apply the so called user cuts, that are cuts (like Gomory cuts) not strictly necessary for the correctness of the method but that can help tighten the LP model.
To inform CPLEX in which *contexts* we are interested to be "called back" we have to set an internal value of the *generic callback function* of CPLEX. To do this we have to call the method *CPXcallbacksetfunc()* passing as *contextmask* the bit-wise OR between the following bitmasks:

- *CPX_CALLBACKCONTEXT_CANDIDATE*
- *CPX_CALLBACKCONTEXT_RELAXATION*

3.3.1 Lazy Constraint Callbacks (Integer Solutions)

Every time CPLEX founds a new integer solution it triggers a callback function defined by the user. Once inside the callback function we can use the method *CPXcallbackgetcandidatepoint()* to retrieve the candidate solution and then we can add to the ILP model the SECs violated by the solution as we did in the Benders' method. Since the callback mechanism is handled by CPLEX during the execution of the branch-and-cut we say that these constraints are added "on-the-fly". In case the candidate solution has not subtours, we decided to devise an optional procedure, that if selected, optimizes the solution using the 2-OPT technique (discussed in 5.2.1) and then posts to CPLEX the optimized solution using the method *CPXcallbackpostheursoln()*.

3.3.2 User Cut Callbacks (Fractional Solutions)

In this section it will be presented an advanced use of the callback mechanism that intercept the fractional LP solutions returned by CPLEX in order to provide violated SECs for fractional solutions. For integer solutions, we have seen that we first have to find the connected components, and then we add one SEC for each cycle present in the solution. This approach cannot be used with fractional solutions, since we cannot interpret a fractional solution as a collection of cycles. However, the values x_e associated to the arcs of a fractional solution can be interpreted as the capacities of the arcs in a network. Thus, finding a violated SEC corresponds to find a cut in the network with a value less than 2.
We recall that in graph theory, a cut of a graph $G = (V, E)$ is a partition of the nodes V in two subsets $S', V - S'$, and that the value of a cut is equal to the sum of the weights of the edges that have one vertex in S' and the other vertex in $V - S'$. In figure 3.3 is shown an example of a fractional TSP solution with a minimum cut value equal to 1.

Note that, given a fractional solution with only one connected component, if it exists a cut with a value less than 2, then it must exists at least one (fractional) subtour in each one of the two subgraphs generated by such cut. This property is a direct consequence of constraint (2.3), that imposes that each node must be connected to a set of edges such that the sum of their costs is equal to 2, and of the relaxed version of constraint (2.4) that imposes that the cost of each edge must be between 0 and 1. For this reason we are only interested in cuts that have a value smaller than 2. If instead the solution has more than one connected component, there will be always a cut with a value equal to 0. In this case we can simply generate a SEC for each connected component as we did before.
To find the minimum cut in a network, we can exploit the duality between the maximum flow

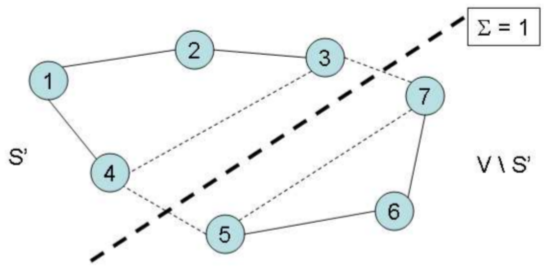


Figure 3.3: Example of a fractional TSP solution. Dashed lines represent edges selected with a value of 0.5. Solid lines represent edges with a value of 1. The bold dashed line that traverses the graph represents a cut, and the value of this cut is equal to $\Sigma = 1$.
Picture taken from *Exact methods for the Traveling Salesman Problem* by L. De Giovanni and M. Di Summa [5].

and the minimum cut, and therefore we can solve the max-flow problem, using for example the Ford-Fulkerson algorithm, to retrieve the minimum cut in the network. Unfortunately this procedure is very time consuming, and if implemented inefficiently it would make this approach impractical. Likely for us, Bill Cook devised an open-source program named *Concorde* [13] that implements a very efficient branch-and-cut algorithm specifically designed for the TSP. Concorde is distributed both as an executable program and as a static library in which are present many optimized functions for the resolution of the TSP. For our purposes we are interested in two functions present in the Concorde library: CCcut_connect_components() and CCcut_violated_cuts(). CCcut_connect_components() returns the connected components present in a graph. While CCcut_violated_cuts() is a very efficient algorithm that returns the minimum cut present in a graph and calls a callback function every time a cut with a value below a specified threshold is found. We can therefore devise a procedure that exploits these methods to produce violated SECs for fractional solutions:

1. Given a fractional solution, we use the method CCcut_connect_components() to find the connected components.

2. If there is only one connected component, we apply the method CCcut_violated_cuts() to find all the cuts with a value less than 2.

3. Through the callback function passed to CCcut_violated_cuts(), we generate for each cut found a SEC on one of the two subsets of nodes that define the cut.

3.3.3 Practical Considerations (User Cut Callbacks)

This procedure is very time consuming, if we apply it for every fractional solution found by CPLEX, the execution of the program would be slow down too much. For this reason we apply it only when the fractional solutions are produced by the root node.

Since implementing the user-cut callbacks using the Concorde library was quite tricky, here we discuss some problematics that we encountered and how we solved them. For us the main problem was related to the function *CCcut_violated_cuts*(). The prototype of this function is the following:

```
int CCcut_violated_cuts (int ncount, int ecount, int *elist, double *dlen, double
    cutoff, int (*doit_fn) (double, int, int *, void *), void *pass_param)
```

Regarding the callback mechanism of this function, the author provided the following description:

doit_fn (if not NULL) will be called for each cut having capacity less than or equal to the cutoff value; the arguments will be the value of the cut, the number of nodes in the cut, the array of the members of the cut, and pass_param.

Thus, $doit\_fn()$ is the callback function that we have to implement in order to generate the SECs. For us it was not clear to what the author refers to when he says "nodes in the cut" or "members of the cut". To untangle our doubts, we started searching in the (not very rich) documentation and we found some more information only in the documentation of the method $CCcut\_mincut\_st()$, in which it is explained that in this case the "nodes in the cut" are the nodes that belong to the "side" in which the node t (a parameter of this method) is present.

Therefore, we supposed that in the case of the method $CCcut\_violated\_cuts()$ the "nodes in the cut" are the nodes that belong to one of the two subsets of nodes that define such cut. To confirm this hypothesis, once inside the $doit\_fn()$ function, we manually computed the cut value coherently with our guess and we compared it with the cut value received as argument. Since they were equal, we concluded that the above supposition was correct.

3.4 Comparisons and Results

Results obtained from exact algorithms discussed within this section are analyzed below. In both subsections, performance profiles will be provided to make the results obtained easier to read. Experimental results can be found within tables located in appendix A.2.

3.4.1 Compact Models

Experiments on Compact Models, discussed in subsection 3.1, were carried out on small instances with no more than 110 nodes. For each model 100 results were obtained, specifically 20 instances were tested five times through a different seed, in order to let CPLEX initialize each specific run in different ways.

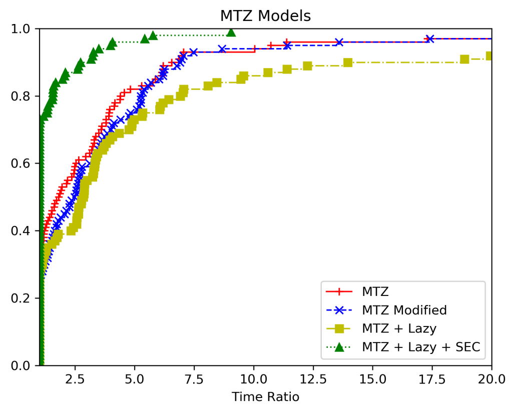
As can be seen from performance profiles shown in figures 3.4a and 3.4b, in order to understand how additional constraints affects models performances, some tests were performed by adding to the base model combinations of the constraints discussed in 3.1.3. It can be clearly seen that both models exhibit the same behavior with respect to the addition of specific constraints. On the one hand, when only lazy constraints are applied (+ Lazy, in the figure legend) to the base model, the time needed to find the exact solution increases considerably. On the other hand, whereas lazy constraints are combined with the degree two subtour elimination constraints (+ SEC, in the figure legend), their combination yields to the best performances.

Although experiments performed individually on the presented models lead to the same general behavior, in the performance profile shown in figure 3.4c it can be seen that a gap clearly exists between MTZ and GG. In fact, the model presented by Gavish and Graves turns out to outperform the Miller, Tucker and Zemlin one in the majority of the executions.

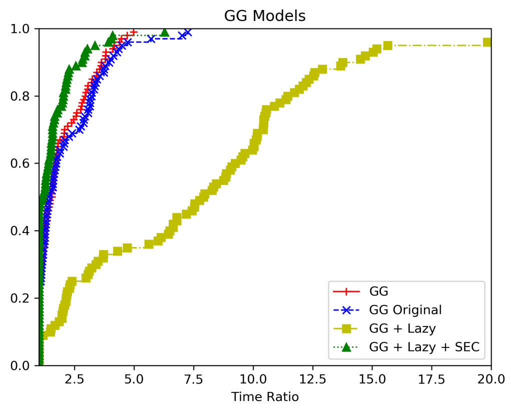
3.4.2 Benders' Method vs Callback

Experiments on Benders' and Callback methods, discussed in subsections 3.2 and 3.3, were carried out on instances with a number of nodes ranging from 200 to 700. For each model 100 results were obtained, specifically 25 instances were tested four times through a different seed, in order to let CPLEX initialize each specific run in different ways.

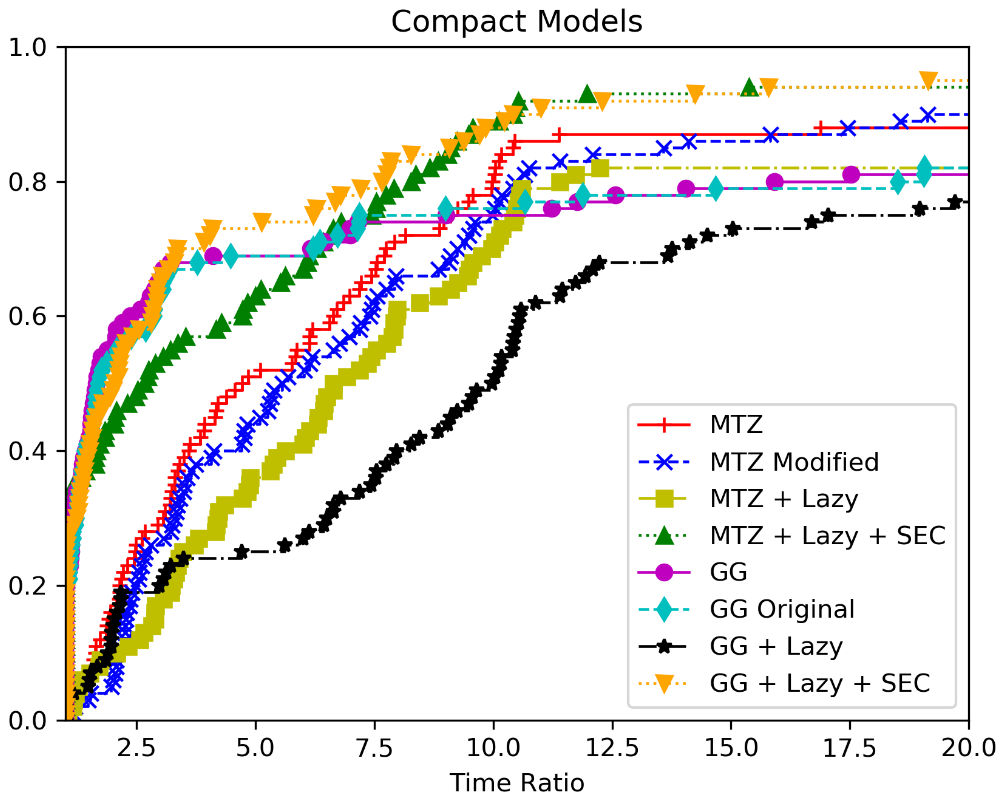
As mentioned in subsection 3.2.1, in order to detect subtours within a solution, methods based on Kruskal and Fischetti's suggestion were implemented. As can be seen from the performance profile



(a)



(b)



(c)

Figure 3.4: Performance profiles on compact models. In figure 3.4a a comparison is shown between the basic MTZ model, a slightly modified version of it, the addition of lazy constraints only (+ Lazy) and the combined addition of lazy constraints with degree two subtour elimination constraints (+ Lazy + SEC). In figure 3.4a a comparison is shown between the basic GG model, its original version, the addition of lazy constraints only (+ Lazy) and the combined addition of lazy constraints with degree two subtour elimination constraints (+ Lazy + SEC). Lastly, in figure 3.4c is shown the comparison between all the presented models in order to highlight which is the best.

shown in figure 3.5a, both methods are able to effectively find connected components within a solution and although there is not a huge gap between them, Fischetti's method turns out to outperform

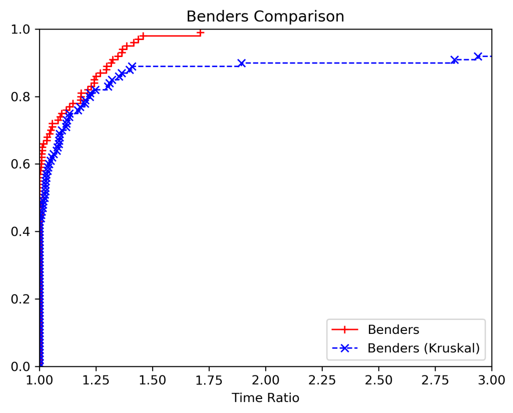
Kruskal's.

In addition to Benders' method, tests were performed on the callback method. To evaluate which configuration led to the best results. In figure 3.5b is shown the performance profile for these methods. The method named "Callback" is the method that implements both the lazy constraint callback, and the user cut callback (only at the root node). The next three methods are all variations of this method "Callback". The method "Callback + 2OPT" add a 2-OPT optimization for the candidate (integer) solutions that have only one connected component (in this case the lazy constraint callback does not generate any SEC). The method "Callback (only integer)" simply handles only candidate solution, that is, the user cut callback is never triggered in this method. The method "Callback (depth 5)" is a variant of the method "Callback", in which the user cut callback accepts not only fractional solutions from the root node, but also from all the nodes with a depth smaller than 5. The method "Callback (fractional + SEC)" implements a modified version of the the user cut callback such that, if the fractional solution has more than one connected components, it generates a SEC for each of these components.

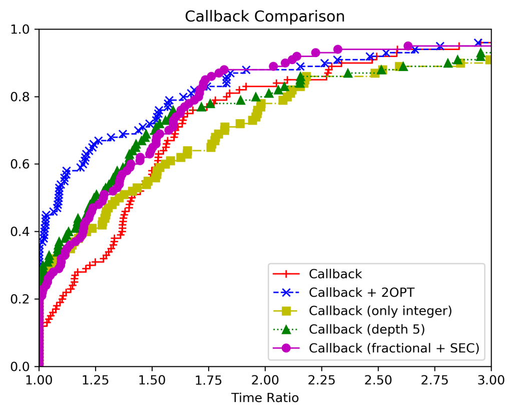
Comparing the three variants of the callback method that do not exploit the 2-OPT optimization, with the original one (red line), we can note that up to a time ratio of 1.5 they provide a sensible improvement, with the variant "Callback (depth 5)" that performs slightly better than the other two. Whereas from the time ratio 1.5 to 3.0 the performance profile shows a situation not so defined as before, in which it can be seen that the the best methods is the "Callback (fractional + SEC)" variant and that the original callback method is not the worst method as before.

To conclude, it is interesting to observe that although the basic configuration turns out to be less efficient than its variant just discussed, once the 2-OPT refinement heuristic is activated inside the lazy constraint callback, the performance of this method turns out to be the best in general. This proves how effective 2-OPT is.

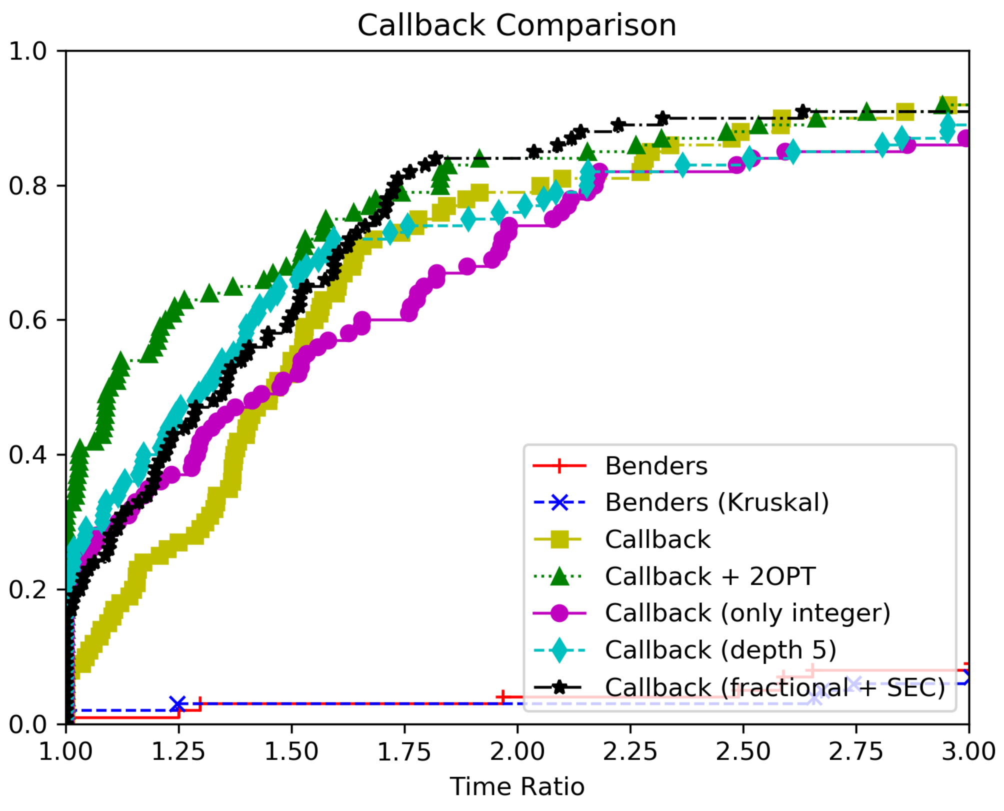
Placing now to comparison the methods previously discussed, it can be observed from the performance profile in figure 3.5c that even if the Benders' method seemed to be promising, with respect to the Callback method it turns out to be extremely inefficient. This result is not so surprising, indeed it is quite expected, since as previously discussed Benders' method needs to stop the execution of CPLEX in order to add the SECs and then restart a new run of CPLEX, generating overhead that is not present in the callback method. This behaviour of the Benders' method does not allow to add as much constraints as the callback method, since the callbacks are triggered every time a solution (integer or fractional) is found.



(a)



(b)



(c)

Figure 3.5: Performance profiles on Benders and Callback methods. In figure 3.5a is shown a comparison of Kruskal's and Fischetti's implementations for finding connected components within a solution. In figure 3.5b is shown a comparison between different configurations of the Callback method. Lastly, in figure 3.5c is shown the comparison between all the presented models in order to highlight which is the best.

Chapter 4

Matheuristic Algorithms

In Chapter 3 we have discussed about exact algorithms that use CPLEX to find the optimal solution. Since we are dealing with instances too big to be solved in a reasonable computing time exploiting exact methods, we want to find alternative approaches to tighten the search space of these bigger instances. Hereafter, we will focus on matheuristic algorithms, which are heuristic algorithms that use a mathematical model. Hence, we can say that this class of algorithms is a sort of hybridization between exact and heuristic algorithms. As a matter of fact, matheuristic algorithms often reach the optimal solutions but only when the instance is not too complex or its size is not too large.

The algorithms discussed in this section use CPLEX to explore the neighborhood of a given solution in the branching tree. Unlike algorithms described in Chapter 3, this class of algorithms will treat the graph of the instance as undirected.

4.1 Hard Fixing Heuristic

The idea behind the hard fixing heuristic is to explore the neighborhood of a reference solution by solving a new ILP problem in which some arcs of this solution are fixed. And then repeat this procedure in a loop fashion on the solution with the lowest incumbent encountered until that moment. Of course, selecting a priori some arcs, it modifies the initial formulation of the problem, but it also allows to reduce the complexity of the original ILP problem generating a sub-ILP problem of our original formulation.

Since we want to find a good solution for our original problem (ideally the optimal one), we must apply the neighborhood search procedure many times with the aim to explore a significant part of the solution space. An overview of this technique can be described in the following steps:

1. Find an initial solution of the original ILP problem and set it as the best solution.
2. Fix a fraction of the arcs that composes the best solution.
3. Find a solution of the correspondent sub-ILP problem (ideally the optimal one, since now the ILP problem is easier to solve).
4. If the incumbent of the new solution is lower than the incumbent of the best solution found so far, set the current solution as the best solution.
5. If the time limit is reached STOP, otherwise go back to step 2.

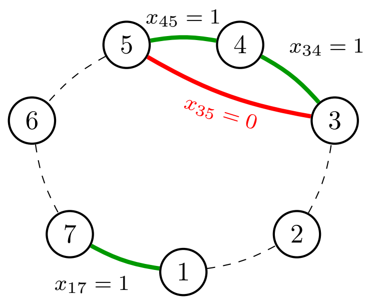


Figure 4.1: Example in which the best solution is shown, where in green are highlighted the edges that have been fixed for the next iteration. The arc (3,5) has been marked in red to point out that this arc can be set to 0 since we don't want subtours. While the black dashed edges are the edges of the best solution found so far that have not been fixed.

4.1.1 Practical Considerations

Being the hard fixing heuristic an iterative local search technique, the final solution returned is very dependent on the initial one. Thus, to build a good initial solution we use our best method that implement the exact formulation of the TSP, that in our case is the one the exploits the callbacks (both for integer and fractional solutions), for a very short time (in our case, overall time limit divided by 20) and only at the root node. In this way we let most of the available time to the hard fixing heuristic. The motivation to let CPLEX only works at the root node is related to the fact that we want to reduce as much as possible the overhead, since the time available for this stage is very small. At each iteration we create a new sub-ILP problem fixing each edge of the best solution with a probability equal to *fix_ratio* (e.g. 0.9), and for each iteration we impose the same time limit applied to the run related to the initial solution. Since we know that a feasible TSP solution cannot have subtours, after we have fixed the edges, we also set to 0 all the edges that, if selected, would cause a subtour. More precisely, these edges are the ones that connect the first and the last node of a sequence of at least two contiguous fixed edges (for example the edge [3,5] in figure 4.1). To implement what described above, we store the best solution as an array of successors, and set these edges to 0 during the fixing procedure. If for many iterations the method is not able to return an improved solution, one possibility is to enlarge the neighborhood window decreasing the *fix_ratio*. In our implementation, if the best solution does not get improved for 10 consecutive iterations, we decrease the *fix_ratio* by 0.05, until we reach the limit of 0.5. Note that lowering the *fix_ratio* implies that less variables are fixed, and therefore, the problem becomes harder to solve. Before starting a new run of CPLEX we have to initialize the incumbent of the new run as the incumbent of the previous solution, this initialization procedure is called *warm start*. In theory CPLEX does this operation automatically, but to make sure that CPLEX will start effectively with such incumbent we can use the method *CPXaddmipstarts()*.

4.2 Soft Fixing Heuristic

The soft fixing heuristic, also called Local Branching, is a technique devised by M.Fischetti and A.Lodi in 2003 [4]. The main idea, as in the hard fixing heuristic, is to iteratively explore the neighborhood of the best solution instead of looking into the whole solution space, with the aim to solve larger instances than the ones solvable by the exact methods.

The main difference between the soft fixing and the hard fixing heuristic is in the definition of the neighborhood. In a generic iteration of the hard fixing heuristic, k edges of the best solution encountered until that moment are fixed. Thus, in the hard fixing, the neighborhood is constituted by all the feasible solutions in which those k edges are present.

For the soft fixing, the following considerations are done. Recalling that a TSP solution in the CPLEX format is represented by a binary vector of length $|E|$, then we can measure the distance between two solutions under some metric function.

$$\begin{array}{c|c|c|c|c|c|c|}
 & 1 & 2 & 3 & 4 & \cdots & |E| \\
\cline{2-7}
\text{X} = & 0 & 0 & 1 & 0 & \cdots & 1 \\
\cline{2-7}
\end{array}$$

Figure 4.2: Example of a solution stored as a binary vector in the CPLEX format, in which each cell represent an edge.

The metric function adopted in this heuristic is the Hamming distance, that given two vectors measures the minimum number of substitutions to change one vector into the other. One property of this metric is that it is linear, and so it can be used in ILP. Since we are dealing with binary vectors, let's suppose that x and y are binary vectors of equal length, then the Hamming distance between x and y can be defined in the following way:

$$H(x,y) = \sum_{j:x_j=1} (1-y_j) + \sum_{j:x_j=0} y_j. \tag{4.1}$$

The first summation counts the number of bits that are equal to 1 in x but that are equal to 0 in y, while the second summation counts the number of bits that are equal to 0 in x but that are equal to 1 in y. The total summation is thus equal to the number of bits of x that must be flipped in order to obtain y (or vice versa).

We can then define the neighborhood of a reference solution x^H as the set of all feasible solutions x such that $H(x, x^H) \leq r$, where r is a parameter that represent the radius of the neighborhood space. Because of the nature of the TSP, all feasible solutions must have exactly n bits set to 1 ($\sum_{e:x_e=1} x_e = n$), this imply that in the computation of the Hamming distance between two feasible solutions, the two summations in (4.1) must be equal. In fact for each edge added into a solution (one bit flipped from 0 to 1), one edge must be removed (one bit flipped from 1 to 0). Thus, from the above consideration we can write the following inequalities:

$$H(x, x^H) \leq 2k \tag{4.2}$$

$$\sum_{e:x_e^H=1} (1-x_e) + \sum_{e:x_e^H=0} x_e \leq 2k \tag{4.3}$$

$$\sum_{e:x_e^H=1} (1-x_e) \leq k \tag{4.4}$$

$$\sum_{e:x_e^H=1} x_e \geq \sum_{e:x_e^H=1} 1 - k \tag{4.5}$$

$$\sum_{e:x_e^H=1} x_e \geq n - k. \tag{4.6}$$

The property that for the TSP $\sum_{e:x_e^H=1}(1-x_e) = \sum_{e:x_e^H=0} x_e$, it has been used to go from (4.3) to (4.4). Inequality (4.6) tells that all the feasible solutions that belong in the neighborhood with radius $2k$ of a reference solution x^H must preserve at least $n-k$ edges of x^H (or alternatively, at most k edges can be removed). For this reason inequality (4.6) is called *k-OPT neighborhood.*

From an algorithmic perspective, the soft fixing heuristic adopts the the same scheme described in (4.1) for the hard fixing heuristic with the exception of step 2. In fact, in step 2, instead of fixing some edges of the best solution as done for the hard fixing heuristic, now it is added to the original ILP model the k-OPT neighborhood constraint. Therefore, the main difference between the two matheuristics presented in this chapter is that in the hard fixing heuristic some edges are selected a priori, while in the soft fixing heuristic it is only constrained the minimum number of edges that must be preserved, without specifying which edges.

4.2.1 Practical Considerations

The initialization stage is the same of the hard fixing heuristic, and can be found in (4.1.1). Typically, k is initialized between 2 and 5. If after a soft fixing iteration the incumbent does not improve, we enlarge the neighborhood by increasing the value k by one, until we reach the maximum value of 15, since for higher values the neighborhood becomes too large making this approach impracticable. It is also the same the time limit that we applied for each iteration. When we add the additional k-OPT neighborhood constraint to the original ILP problem, we must make sure to remove the additional constraint that we added in the previous iteration. To do so we can use the CPLEX function *CPXdelrows()*. Alternatively, since all the additional k-OPT neighborhood constraints that we will add differ only in the constant term, we could only change the right hand side of the k-OPT neighborhood constraint already present in the ILP model.

4.3 Comparisons and Results

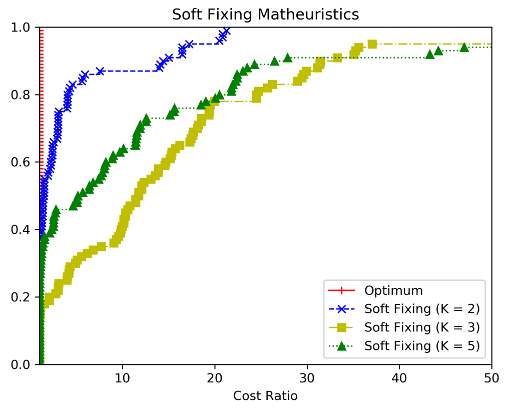
Results obtained from matheuristic algorithm discussed within this section are analyzed below. Performance profiles will be provided to make the results obtained easier to read. Experimental results can be found within tables located in appendix A.3.

For each model 100 results were obtained, specifically 25 instances were tested four times through a different seed, in order to let CPLEX initialize each specific run in different ways. Experiments on the matheuristic algorithms, discussed in subsections 4.1 and 4.2, were carried out on instances with a number of nodes ranging from 200 to 1100.

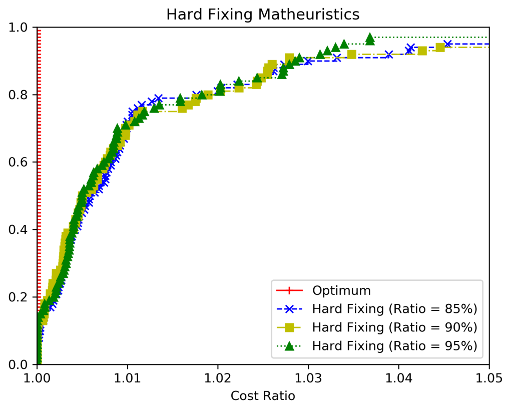
When we talk about heuristics, since it is not necessarily true that it is possible to achieve an optimal solution, it is more interesting to evaluate which is the best solution that can be found within a certain time limit. In the experiments carried out, for each matheuristic three different configurations were analyzed, specifically the soft fixing was tested with an initial k value of 2, 3 and 5 while the hard fixing with an initial fix-ratio of 85%, 90% and 95%.

As can be seen from the performance profiles shown in figures 4.3a, for what concerns the soft fixing matheuristic it seems to be more convenient to set a very small initial value of k allowing the algorithm to adjust itself at run time whenever remains stuck into a local optima, instead of starting from the beginning with a larger value of k which is equal to probing a wider research space through k-OPT moves. Different is what happens with the hard fixing heuristic, as a matter of fact in figure 4.3b we see how varying the initial fix-ratio in a neighborhood of 90% does not produce any clear alterations in the performance of the matheuristic.

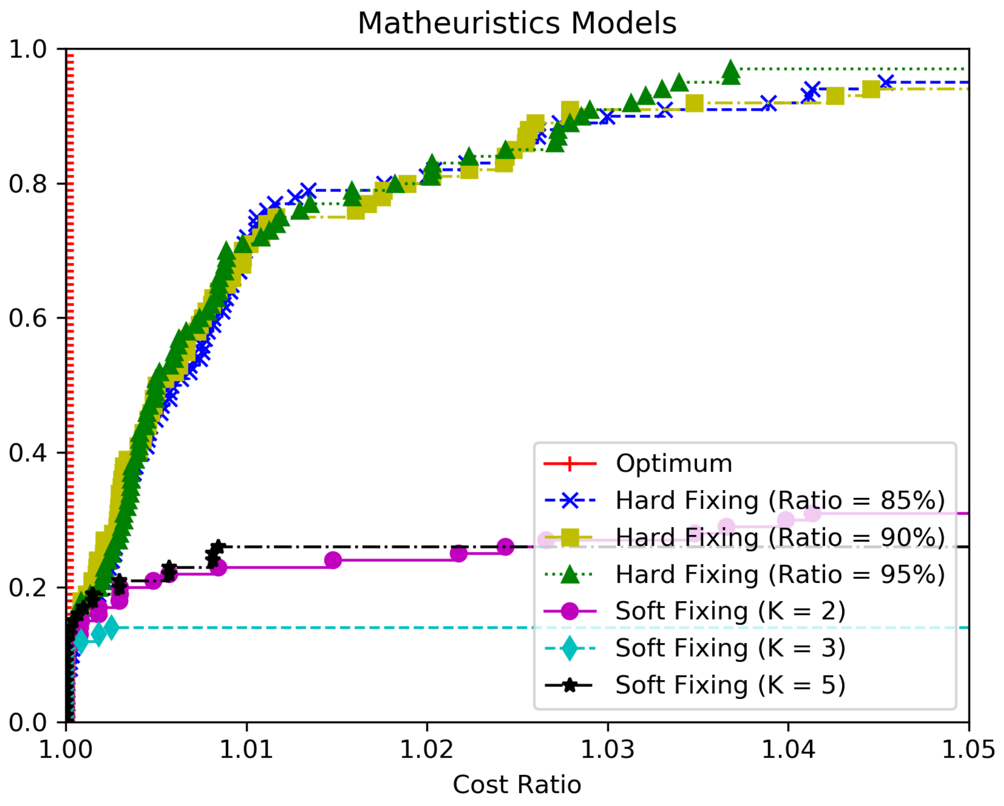
Regardless of the matheuristics local behaviors, one can see in figure 4.3c how hard fixing turns out to outperform soft fixing, and that for the instances we tested, the soft fixing heuristic is able to obtain with a probability grater than 0.9 a solution with a cost within the 5% with respect to the optimal solutions.



(a)



(b)



(c)

Figure 4.3: Performance profiles on Hard and Soft Fixing matheuristics. In figure 4.3a is shown a comparison of Soft Fixing matheuristics with different values of initial k. In figure 4.3b is shown a comparison of Hard Fixing matheuristics with different values of initial fix-ratio. Lastly, in figure 4.3c is shown the comparison between all the presented matheuristics in order to highlight which one produces the closest result to the optimum solution.

Chapter 5

Heuristic Algorithms

In Chapters 3 and 4, we analyzed relative small instances because, with the instance size growth, the execution of the previously discussed algorithms becomes more and more demanding of computational and time resources, up to the point where it becomes unsustainable for any given machine. In these conditions, we can rely on heuristic algorithms that seek good solutions at a reasonable computational cost, without being able to guarantee optimality or even in many cases to state how close to optimality a particular feasible solution is [9].

Inside this class of algorithms, there are many different suitable heuristics for TSP. For the sake of clarity, we classify the following heuristics in two separate subclasses, tour construction and refinement heuristics (as proposed in [7]).

5.1 Tour Construction

The tour construction heuristics play an important role in solving the TSP, since takes care of generating initial tours for the tour refinement heuristics. With the help of an effective tour construction heuristic, the performance of a heuristic can be improved. Taking the role of starting block, a constructive heuristic stops whenever a solution is found and never try to improve it. Best tour construction algorithms usually reach feasible solution within a range of $10 - 15\%$ from the optimal one. Within constructive heuristics, the key points are efficiently selecting the next node and identifying the best place to insert it. In the following sections we will introduce two construction heuristics.

5.1.1 Nearest Neighbors

Nearest Neighbors is a greedy, 2-approximation algorithm, that follows the problem-solving heuristic of making the locally optimal choice at each stage. This kind of algorithm quickly generates a solution for TSP by iteratively inserting in the tour the nearest node with respect to the last one added. The initialization takes a random node and the algorithm keeps iterating until all the nodes are inserted in the tour. The solution generated is defined by the sequence in which nodes are visited, figure 5.1 shows an example of execution.

The advantages of this algorithm are the easy implementation and the short running time: since Nearest Neighbors checks $n - 1$ incident edges for each of the n nodes, its complexity is $O(n^2)$. A disadvantage of Nearest Neighbors is that the current choice depend on all choices made so far, since it iteratively makes one greedy choice after another without considering any previous step, this may lead to “miss” some good nodes and when the tour is almost completed, the algorithm has to go back to visit the missed node and this may potentially result in a great increase of the tour cost.

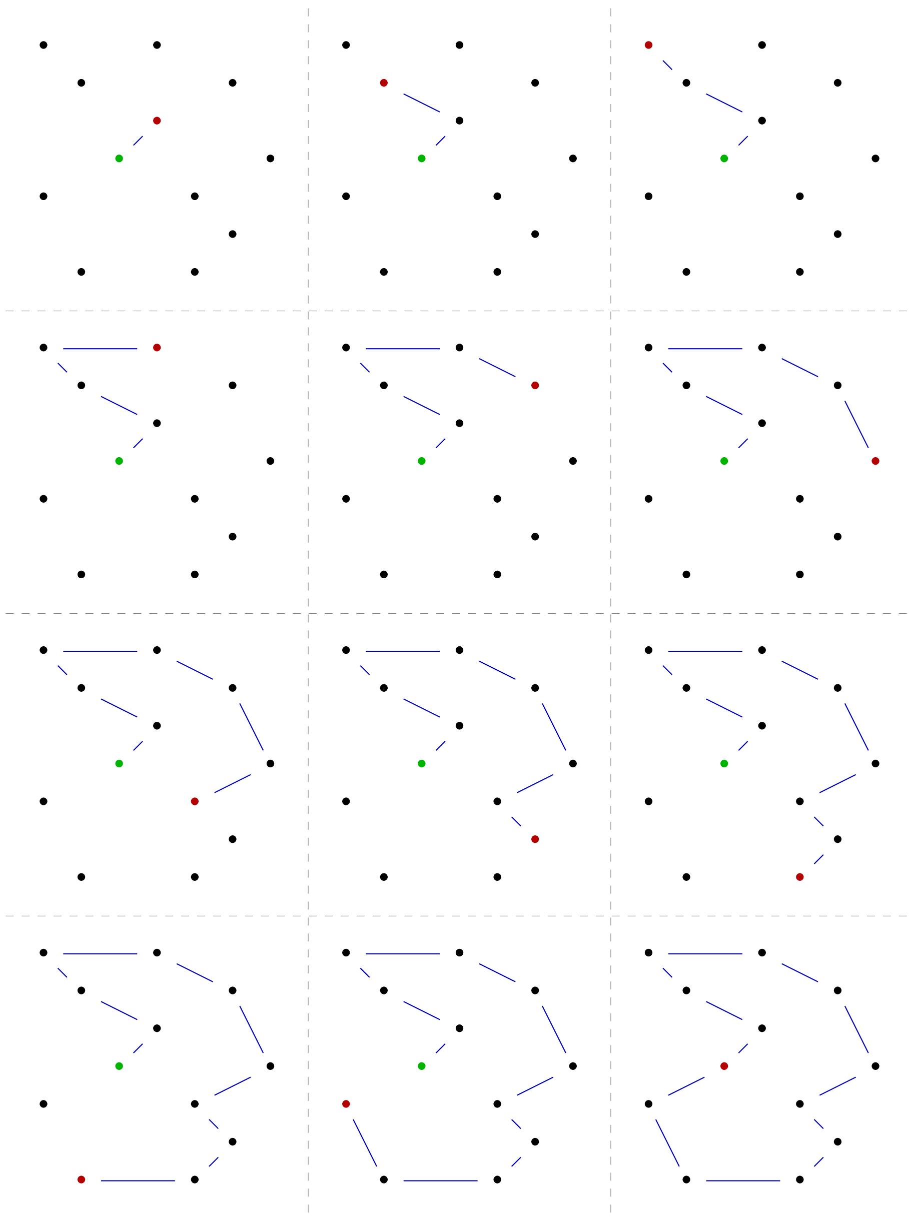


Figure 5.1: Visual explanation of nearest neighbor heuristic. After selecting a random starting node (green), nearest neighbor scan its neighborhood and select the not-yet-touched closest one, then move the current node to the selected one (red) and repeatedly perform the same operation. Once the current node touch the selected one, or in other words once it is not possible to select another not-yet-touched node, nearest neighbor stops its execution and return the solution found with respect to the provided starting node.

5.1.2 Extra Mileage

Extra mileage is a cycle-growth algorithm that uses a greedy approach to generate a solution for TSP. The algorithm initializes the tour with two nodes, one randomly selected between all the vertices and the other taken as the farthest from it. Then, at each iteration, the algorithm chooses the node k, among all the nodes not yet in the tour, whose distance with respect to the tour is minimal in order to increase the tour length by the lowest amount possible. This is done by finding the edge (i, j) in the current tour that minimizes the extra-mileage delta (see equation 5.1) that is literally the cost that the tour must spend in order to gain another node.

$$\Delta(i,j,k) = C_{ik} + C_{kj} - C_{ij} \tag{5.1}$$

This algorithm complexity is $O(n^2)$, it is useful to notice that within it the bottleneck lies inside the selection of the best pair node k - edge (i,j). Figure 5.2 shows an example of execution.

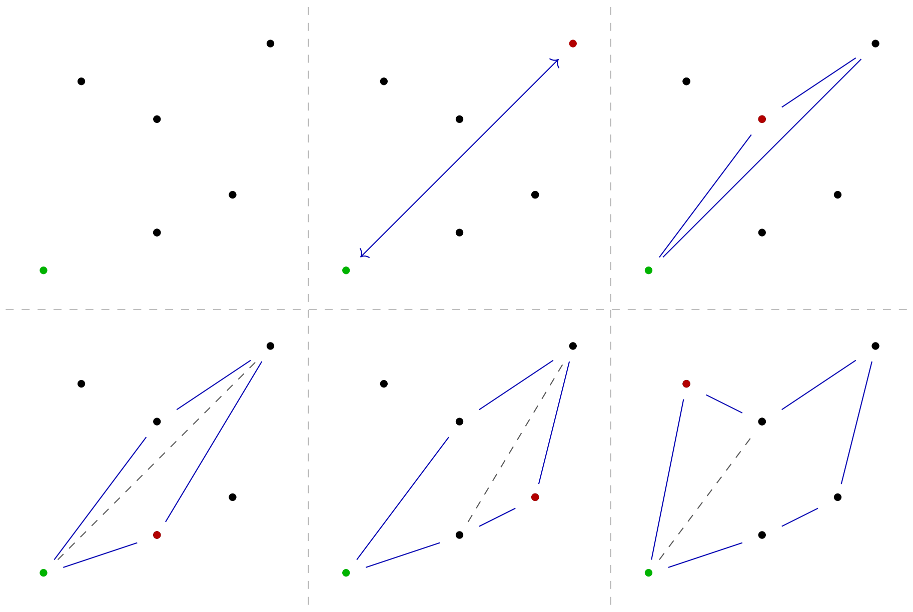


Figure 5.2: Visual explanation of extra mileage heuristic. After selecting a random starting node (green), extra mileage scan the not-yet-touched node in order to find the closest one to the current circuit, then with respect to the selected one (red) find which of the circuit edge is the most suitable to be replaced, and repeatedly perform the same operation. Once all the nodes are considered inside the circuit or in other words once it is not possible to select another not-yet-touched node, extra mileage stops its execution and return the solution found with respect to the provided starting node. In light grey are reported all the edges replaced by the algorithm. The only replaced edge not highlighted is the first one since it overlaps with the one used to close the minimal two node circuit.

As mentioned above, the extra mileage algorithm initializes the minimal tour taking the farthest node from the randomly selected one. Clearly nothing forbids any other kind of initialization, such as a convex hull or a tour made from the farthest absolute nodes of the instance, which has been implemented.

5.1.3 Construction Guidance

The proposed heuristic algorithms are in a way deterministic since once the starting node is fixed the choices made will always be the same. One way to solve this limitation lies in the application of the following techniques.

Multistart

Multistart methods, as the name suggests, deal with executing the proposed heuristics several times by varying one or more initialization parameters. In TSP case the only useful parameter to vary is the starting node from which the construction heuristics starts to build the solution.

Specifically, each node in the instance is selected to be the starting point of the heuristics and after each global iteration produces a solution (usually a local optima), the best overall is returned. Generally this kind of methods are used to avoid ending up in the same local optima by efficiently exploring the solution space. Commonly, after a first phase in which the solution is generated, there may be a second phase in which the generated solution can be typically (but not necessarily) improved.

GRASP

GRASP (Greedy Randomized Adaptive Search Procedure) was introduced by Feo and Resende in 1995. Introduces non-determinism to the resolution of an instance. In particular, while a deterministic algorithm always makes the choice that looks optimal at the moment, GRASP make use of randomization during the selection phase and build different solutions at different runs. Such randomization, is bounded to a maximum number of k possible nodes at any time when a choice must be made.

5.2 Tour Refinement

Starting from feasible solutions, tour refinement heuristics allow to narrow the feasible search space of an improved solution by making a very small number of changes. The performances of these algorithms are somewhat linked to the construction heuristic used. Among the tour refinement heuristics, the most widely used is the so called k-OPT which modifies the current solution by replacing k arcs in the tour with k new arcs in order to generate an improved tour. Typically, the exchange heuristics are applied iteratively until a local optimum is found, namely, a tour which cannot be improved further via the exchange heuristic under consideration. In the following, we will present and analyze the most efficient version of k-OPT, which is 2-OPT.

5.2.1 2-OPT

2-OPT also known as pairwise exchange was proposed by Croes in 1958 and still is one of the most basic and widely used tour refinement heuristic for the TSP. 2-OPT algorithm takes as input a tour initialized randomly or built by using some tour construction heuristic as shown in section 5.1, and then incrementally improves the tour exchanging two edges in the tour with two other edges.

In particular, in each improving step the 2-OPT algorithm selects two edges (a, b) and (c, d) from the tour such that a crossing occur between their four distinct node a, b, c, d as shown in figure 5.3. Then, the algorithm after replacing such crossing edges with (a, c) and (b, d), reverse the tour between nodes b and c in order to maintain consistency inside the solution. The simple but effective rationale behind this swap is to be sought in the triangular inequality, because if we split the arcs (a, b) and (c, d) in the crossing point k we can easily see that:

$$C_{ab} + C_{cd} = (C_{ak} + C_{kb}) + (C_{ck} + C_{kd}) = (C_{ak} + C_{ck}) + (C_{kb} + C_{kd}) \leq C_{ac} + C_{bd} \tag{5.2}$$

In other words, a single pairwise exchange, guarantees the decrease of the solution cost as well as its overall length, is often referred to as a 2-OPT move. The solution improvement after a single 2-OPT move can be expressed as:

$$\Delta(a, b, c, d) = C_{ac} + C_{bd} - (C_{ab} + C_{cd}) \leq 0 \tag{5.3}$$

It is useful to notice that, any 2-OPT improvement move should lead to a negative delta, otherwise it is a worsening move. Keeping that in mind, the algorithm continues to swap edges inside the tour until no more crossing can be solved, or in other words a worsening move is found, and terminates in a local optimum in which no further improving steps are possible. Such local optimum may be very different from the best solution, since it strongly depends on the initial tour considered.
A crucial point of 2-OPT lies in when to make a move. In the following are briefly presented two implemented policy.

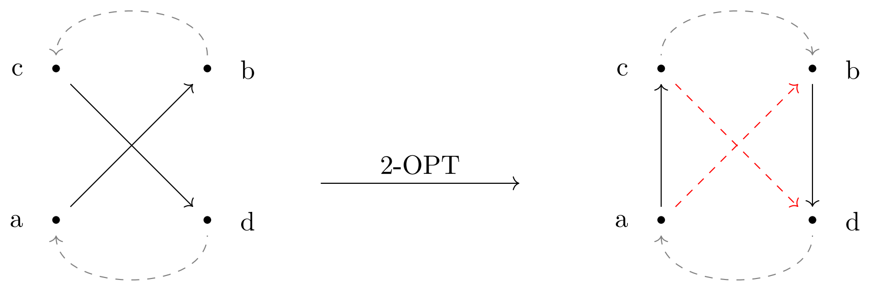


Figure 5.3: Visual example of a 2-OPT operation on a euclidean instance with negative delta. Whenever a crossing between four distinct nodes a, b, c, d is found within a feasible solution, the edges (a, b) and (c, d) are replaced respectively with edges (a, c) and (b, d) in order to remove the crossing. This replacement follows the triangular inequality and allows the total cost of the solution to be reduced.

Deep Policy

A traditional implementation of this heuristic suggests to make a move only when the greatest improvement among all is found within the instance. This policy then allows only the optimal move to be made at each iteration, allowing for a linear drop in the cost of the solution that is maximized in the first iterations.

Within this selection policy a single 2-OPT move has time complexity of $O(n^2)$. This because in the worst case we need to check all possible combination of one edge and other $(n - 1)$ edges in order to find if some of that crossing could improve the tour - thus the $O(n)$. We then need to check it for all the edges, therefore we get the equation $n \times O(n) = O(n^2)$.

Quick Policy

Another valid implementation of this heuristic suggests to make the first available move regardless of the improvement. This policy then allows any sub-optimal move to be made at each iteration, allowing for a non-linear drop in the cost of the solution.

Within this selection policy a single 2-OPT move has time complexity of $O(n^2)$. This because in the worst case we need to check all possible combination of one edge and other $(n - 1)$ edges in order to find if some of that crossing could improve the tour - thus the $O(n)$. We then do not need to check it for all the edges, since we are not interested in an optimal move.

5.3 Comparisons and Results

Results obtained from heuristic algorithm discussed within this chapter are analyzed below. Performance profiles will be provided to make the results obtained easier to read. Experimental results can be found within tables located in appendix A.4.

Experiments on the heuristic algorithms, discussed in sections 5.1 combined with refinement heuristics analyzed in 5.2, were carried out on instances with a number of nodes ranging from 200 to 2300. For each model 100 different instances were tested in order to obtain results. Since it was not possible to find 100 distinct instances within TSPLIB, 57 random instances were generated.

Starting with the Nearest Neighbors construction heuristic, as we can see from the performance profile in figure 5.4, a variety of tests were performed. In the first place, we evaluated the effectiveness of the multistart on the Nearest Neighbors heuristic. To do so we applied two different initialization strategy. In one we applied the Nearest Neighbors heuristics N times, starting each time from a different node and selected the best solution. While in the other case we simply applied the Nearest

Neighbors once, starting from a random node. As it is easy to observe, the random approach is in general slightly worse than the multistart one, and this result is expected since it is not certain that the random node produces the best possible solution.

Then, to introduce some randomization in the heuristic, GRASP was added allowing the choice between 3 nodes among the closest ones in each greedy step. As can be expected, even with a pool of 3 nodes, the introduction of GRASP significantly worsen the performance of the heuristics.

Finally, taking into account the results just discussed, the power of the 2-OPT refinement technique (detailed in 5.2.1) was evaluated. 2-OPT is highly dependent on the starting solution to be optimized, this can be clearly observed from figure 5.4, in which it can be seen that there is a clear difference between applying the refinement heuristics while GRASP is active or not. In figure 5.4, the first four methods listed in the legend use the multistart approach, while the others four methods pick a random node where to start the Nearest Neighbors procedure. From the performance profile we can then establish that the simplest method with the 2-OPT refinement procedure applied at the end, is the one that obtained the best result. Note in fact that when the Nearest Neighbors is followed by the 2-OPT refinement, if the multistart approach has been used or not is completely irrelevant. The attempt of introducing some randomization to the greedy approach of the Nearest Neighbors did not produce positive effects; even when followed by the 2-OPT procedure, the scores obtained are sensibly higher than the scores of the best models.

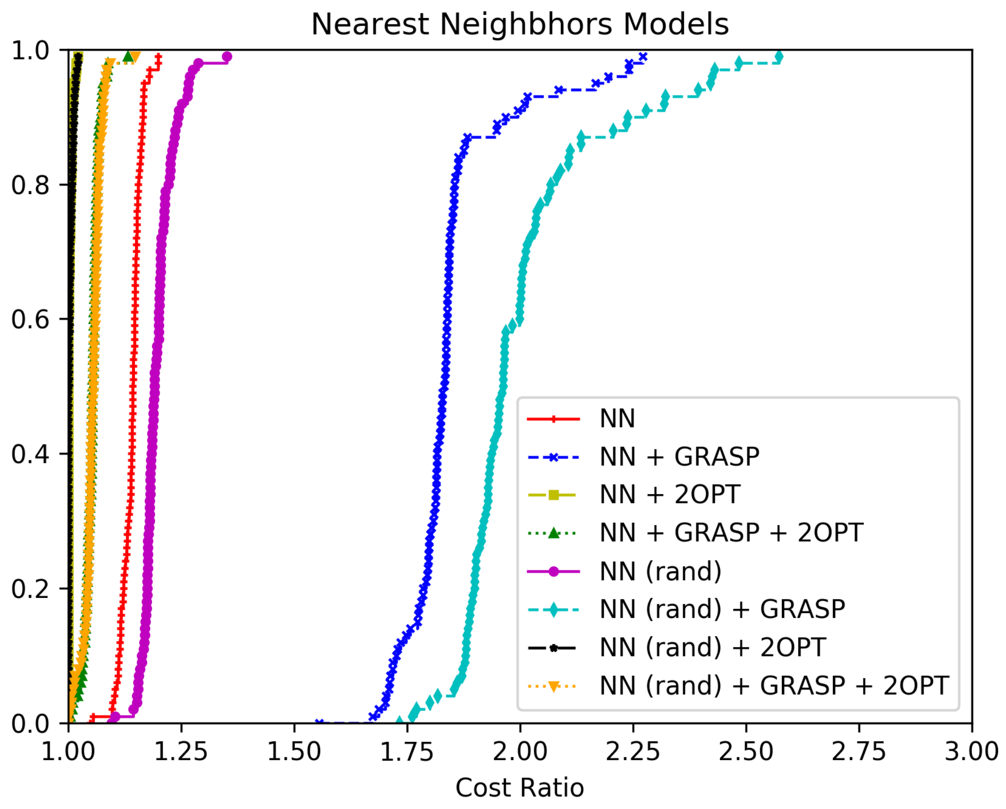


Figure 5.4: Performance profiles on Nearest Neighbors heuristic. A comparison between different approaches applied are shown. Two main approaches can be seen, the one that exploits multistart (NN) and the one that randomly chooses a starting node (NN RAND). Both approaches were combined with GRASP ($k = 3$) and 2-OPT refinement heuristic.

Speaking now of Extra Mileage, we tested the variant that considers as a starting path the one formed by the two farthest nodes within the instance. In figure 5.5 it is possible to observe how Extra Mileage performs with respect to Nearest Neighbors and it can be noticed that without the application of any refinement, it turns out to produce solutions close to Nearest Neighbors based on multistart. From the performance profile we can also notice that once 2-OPT is applied, the best results are obtained again with Nearest Neighbors, but very close results are obtained also with the Extra Mileage method. This outcome is due to the fact that Nearest Neighbors can introduce with high probability crossings within the solution, while Extra Mileage tends to avoid their introduction thanks to its construction policy. Therefore, this suggests how 2-OPT refinement heuristic express better its potential if the solutions provided contain several crossings and its not already near-optimal.

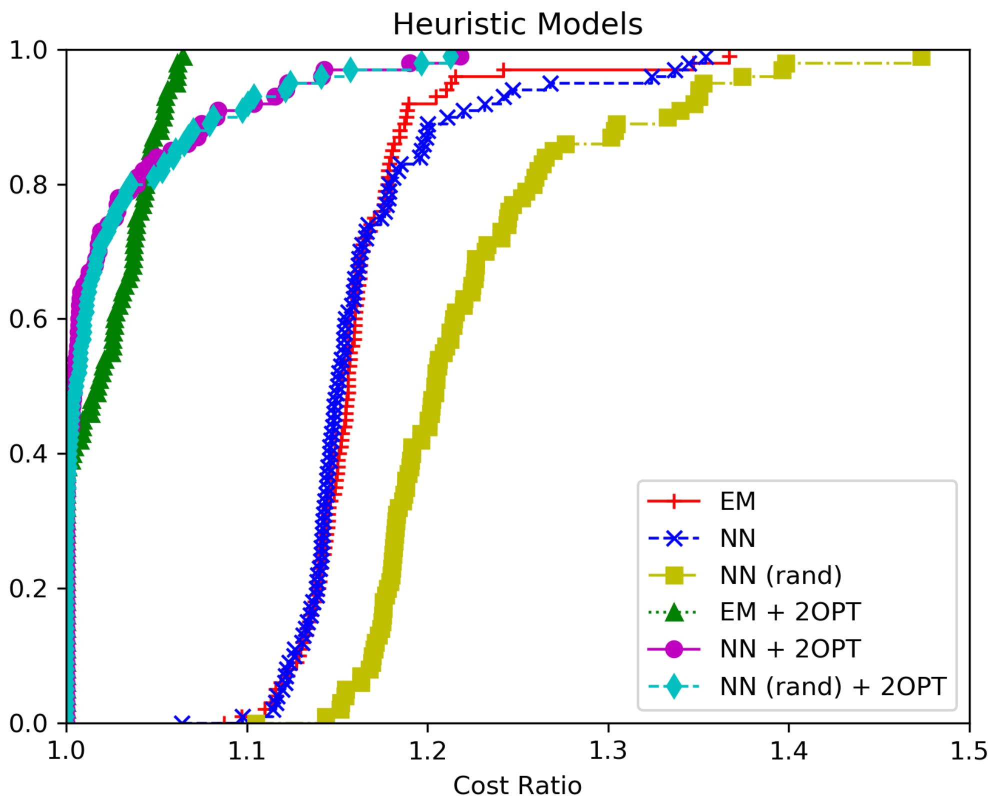


Figure 5.5: Performance profile of Nearest Neighbors and Extra Mileage heuristics. Extra Mileage is initialized with the furthest absolute nodes within each instance and it is compared with both Nearest Neighbors initialized randomly and with multistart. Each heuristic is then optimized through the 2-OPT refinement heuristic.

Chapter 6

Metaheuristic Algorithms

In Chapters 5, we analyzed some useful heuristics to achieve nearly optimal TSP solution. Starting from these heuristics, we now introduce metaheuristic algorithms which are based on both construction and refinement heuristics. The main idea behind the development of metaheuristic algorithms is that an heuristic does not necessarily have to be completely problem-dependent and rather than reinventing one from scratch, a metaheuristic algorithm can offers many problem-independent features, making this kind of algorithms capable of solving a wider range of tasks.

Many metaheuristic algorithms are derive from a single heuristic, such as Tabu Search, others are metaphor-based. One of the most successful metaphor was the natural evolution, which led to the development of the Genetic Algorithm. In this chapter, Tabu Search and Genetic Algorithm are presented and analyzed.

6.1 Tabu Search

Tabu Search (TS) is a metaheuristic algorithm invented by Fred Glover in 1986. This algorithm exploits all those optimization techniques that inside each iteration apply some type of "moves" that slightly modify the current solution with the guarantees to improve the cost function. These techniques, that can also be tour refinement heuristics as discussed in section 5.2.1, can improve the current solution until a predefined move would cause the cost function to increase due to the finding of a local optimum solution. For this reason, these techniques are also called local search methods.

Tabu Search aims to escape from these local optimum solutions once reached. The simple but effective idea of Fred Glover was to keep applying a move also when a local optimum solution is found, climbing in this way the cost function aware of worsening the current solution. In order to avoid the next move to undo the worsening one, leading in this way into a closed loop between these two solution, Fred Glover suggest to forbid the worsening one marking it as a **tabu** one.

6.1.1 Tabu List

A tabu move is collected inside a **tabu list** defined with a maximum capacity, called **tenure**. The maximum capacity of a tabu list, must be coherent since if it is too high with respect to the instance dimension the majority of the moves can be forbidden, leading to an unlike scenario in which the solution space could be so restricted that many solutions become unreachable, thus, not allowing an effective improvement of the current solution. Once the tenure is reached, and the tabu list full of forbidden moves, if a new tabu move has to be added a FIFO (First In First Out) policy can be adopted in order to remove the oldest forbidden move, allowing that move to be selected again.

Inside our implementation, the moves we are dealing with are 2-OPT moves. In order to forbid a 2-OPT move to be done repeatedly, the most straightforward implementation consists in storing inside the tabu list the two edges removed in the worsening move that we want to make tabu. A different, but still effective implementation, presented in class, focuses on the nodes used in the move. In detail, defining a **tabu node** as a node of the graph whose edges connected to it cannot be changed, when we want to insert a move tabu, we must mark one of the four nodes involved in that move as a tabu node. It is useful to notice that a tabu node entails a stronger constraint than the one associated to the two tabu edges. While this could seem unwanted, the idea is to force the algorithm to look in another part of the graph.

In our implementation, the tabu list is represented as an array of length equal to the number of nodes of the graph. Each element is associated to a node of the graph, and its value will be equal to the iteration in which the node has been declared tabu, or zero otherwise. The tenure associated to the tabu list is handled by checking if a node has been declared tabu no more than T iterations ago. In this way, since we can define at most one tabu node per iteration, there cannot be more than T tabu nodes active at any iteration. For clarity, let h be a selected node, i the current iteration and T the current tenure value In pseudo code this validity condition is:

$$TABU\_LIST[h] \geq max(\ 0,\ i - T\ )$$

6.1.2 Phases

When Tabu list tenure is set up efficiently, it is possible to convey the purpose of Tabu search in a way that allows it to enter an intensification or diversification phase. A simple way to implement this change, is to alternate these phases every K iterations, setting differently the tenure for each phase, high when it is necessary to intensify the search and possibly move in other region of the search space, or low when you want to diversify the local solution found searching for the first useful descent.

6.2 Genetic Algorithm

Genetic algorithm (GA) is a metaheuristic based on the principle of genetics and natural selection used to find optimal or near-optimal solution, due to the fact that has the ability to deliver a "good-enough" solution "fast-enough", it is really attractive for solving optimization problems.

Since this metaheuristic has taken inspiration by evolution theory, it is useful to provide some terminology before proceeding with further details. Attempting to synthesize the concept of evolution, we can derive this simple definition:

Evolution is the change in the heritable features of biological populations over successive generations. These features encode the expressions of genes that are passed on from parent to offspring during reproduction. Different traits tend to exist within a given population due to mutation, genetic recombination, and other sources of genetic variation.

Based on this definition, we can notice that some key ingredients to build a GA are the presence of a **population** of individuals that are **supposed to evolve** over an arbitrary number of generations **through variation of some genes** stored inside chromosomes. In order to match these key ingredients, a basic structure for a GA, that will be detailed in subsequent sections, is as follows:

- **Create** an initial population of N individuals and **evaluate** them

- **While** a termination criteria is not reached

  - **Create** M offsprings from the current population
    - **Select** through some policy two parents from the current population
    - **Generate** an offspring through chromosome merge and **evaluate** it
  - **Mutate** some individuals
  - **Select** which individuals should survive
- **Terminate** and **return** the epochs champion

6.2.1 Population

Inside a genetic algorithm a population can be referred as a subset of feasible solutions for the current problem, or in other words a set of individuals' chromosome. Usually is defined as a two dimensional array (figure 6.1) of size $N \times L$, where N is the number of individuals (feasible solution) taken in consideration and L denotes size of a chromosome, that in the TSP case is equal to the number of nodes inside the selected instance.

In order to avoid some premature convergence of the population, its diversity should be maintained through epochs, as well as its size which should not be kept very large as it can cause a global slow down nor too small as it might not be enough for a good mating pool. Inside our implementation, an individual can be generated both in a random way or through the Nearest Neighbor heuristic (5.1.1).

Representation

An important decision to take, in order to let a genetic algorithm to achieve good performances, is how a solution is represented inside it. Since a TSP solution has to take a tour of all the nodes of a given instance, with the constraints of visiting each node exactly once, the most suited is the permutation representation. With this representation, each individual in the population can easily be stored as a permutation of the nodes. It is useful to notice that in our implementation, each individual is stored as a list of successors and not as a list of nodes.

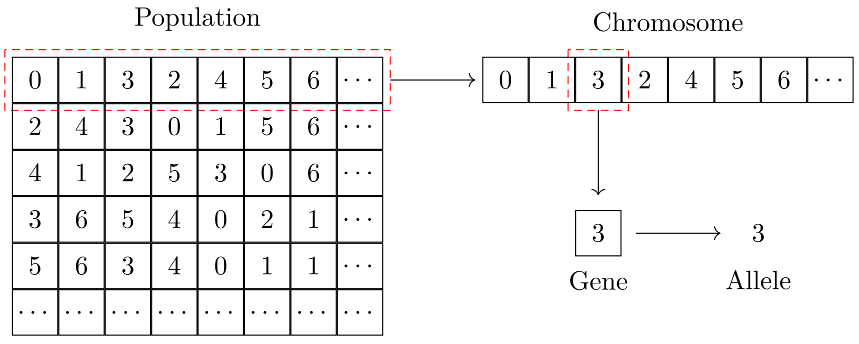


Figure 6.1: Visual example of how is represented a population when an individual is stored through a permutation representation inside a genetic algorithm.

Fitness Function

Another important decision to take, is how to compute the goodness of an individual with respect to the problem considered. Since this computation should be done frequently inside a genetic algorithm, it should be sufficiently fast to not slow down the algorithm. In TSP, the most useful fitness function that can be computed is the solution's cost, achieved through the sum of the length of selected edges. It is useful to notice that usually higher fitness refers to the best individuals, but since we are dealing with TSP this mindset must be reversed meaning that lowest fitness (smallest tour cost) refers to the best feasible solutions.

6.2.2 Parent Selection

Parent selection is the process of selecting parents which mate and recombine to create offsprings for the next generation. Parent selection is very crucial to the convergence rate of the genetic algorithm. It implements the survival of the fittest trying to maintain good diversity in the population since it is extremely crucial for the success of a genetic algorithm. It is useful to notice that selection alone cannot introduce any new individual into the population unlike crossover and mutation which are used to explore the solution space. In order to check which policy is the best suited for a TSP problem, we implemented the following ones. Note that the described processes are performed twice, to obtain a pair of individual.

Tournament Selection

Tournament Selection is also known as k-way tournament selection, this because k individuals are randomly selected in order to act as its participants. Then without performing of each possible match between the participants, the one with the lowest fitness (best tour) is selected to become one of the parent, as shown in figure 6.2, as suggested in [8].

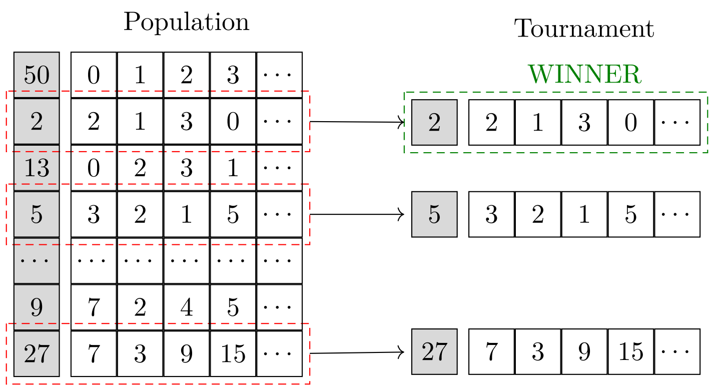


Figure 6.2: Visual example of how a 3-way tournament selection work. After a random selection of three individuals (red), only the one with the lowest fitness is the winner (green). Inside each gray node is reported an over-simplified fitness value used only for clarity purposes.

Roulette Wheel Selection

Roulette Wheel Selection is better known as Fitness Proportionate Selection (FPS) and it is one of the most popular parent selection policy. As suggested from its name, an individual can become a parent with a probability which is proportional to its fitness. Therefore, such strategy applies a selection pressure to the more fit individuals in the population, since they have higher chances of mating and propagating their features to the next generation.

In order to implement this policy we virtually build a wheel which is divided into N pies in which the $i-th$ pie is a proportional to the $i-th$ individual's fitness. For clarity, let f_i be fitness values of individual i, then its selection probability p_i is define as:

$$p_i = \frac{f_i}{\sum_{k=0}^{n} f_k} \tag{6.1}$$

Then a fixed point is chosen on the wheel circumference and the wheel is spun. The region of the wheel which comes in front of the fixed point is chosen as parent. The basic advantage of proportional roulette wheel selection is that it discards none of the individuals in the population and gives a chance to all of them to be selected.

Rank Selection

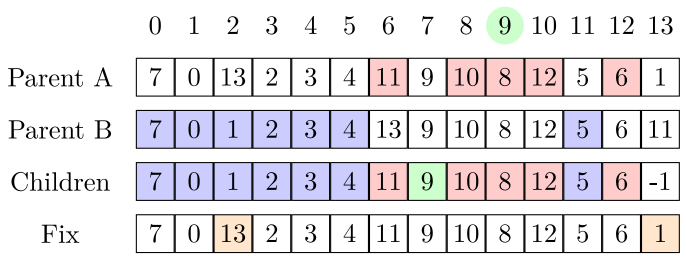
Rank selection can be seen as a variant of roulette wheel selection, since the probability to select an individual is this case is based on its rank rather than fitness. At first, each individual is sorted according to its fitness then thanks to a mapping function the individuals' indices are mapped to their selection probabilities. Such mapping function can be linear or non-linear, in our implementation a linear one is proposed. Hence rank-based selection can maintain a constant pressure in the evolutionary search where it introduces a uniform scaling across the population and is not influenced by super-individuals or the spreading of fitness values at all as in proportional selection.

6.2.3 Crossover

Crossover is a very controversial operator due to its disruptive nature (i.e., it can split important information). In fact, the usefulness of crossover is problem dependent. In TSP, since we need to include all locations exactly one time, we implement a one-point crossover which can be used both in a random or weighted way. Whenever the random policy is selected, the crossover point (c_p) is taken at random between index 0 and L otherwise its weighted in order to maintain more features from the fittest parent. Let f_A and f_B be the fitness of the two mating parent, where A is the fittest one, c_p is computed as follows:

$$c_p(f_A, f_B) = \frac{L}{f_A + f_B} \times f_A \tag{6.2}$$

Since we represent each individual as list of successors, the crossover point can be seen as the number of successors needed to be followed before jumping to the second parent. In order to improve the diversification inside a crossover operation, the first index to be copied is chosen at random.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Parent A	7	0	13	2	3	4	11	9	10	8	12	5	6	1
Parent B	7	0	1	2	3	4	13	9	10	8	12	5	6	11
Children	7	0	1	2	3	4	11	9	10	8	12	5	6	-1
Fix	7	0	13	2	3	4	11	9	10	8	12	5	6	1

Figure 6.3: Visual explanation of the crossover operation. Children's chromosome is fully initialized to -1. In this example the selected starting index is 9 (green circle) and the crossover point is set to 5, so starting from 9, 5 jumps are performed in order to copy the sequence 8 - 10 - 12 - 6 - 11 (red one). Once no more jumps can be performed inside parent A, the successors of the last entry is searched inside parent B, if it contains a valid node the copy starts, otherwise jumps are performed until a valid one is found. In this specific case at index 11 there's a valid node and the sequence 5 - 4 - 3 - 2 - 1 - 0 - 7 (blue one) is copied. Since the last jump lead to the starting node, 9 is inserted and the tour closed. Then a sanity check is performed and if the children's chromosome is broken (contains -1), extra-mileage is used to fix it.

6.2.4 Mutation

A mutation may be defined as a small random tweak in a chromosome, in order to obtain a new solution. It is used to maintain and introduce diversity in the genetic population. In our implementation the mutation operator, mimics the behavior of a non-optimized 2-OPT move, meaning that after two edges (a, b) and (c, d) are randomly selected, a swap occur between them (as described in subsection 5.2.1) without checking that it brings an improvement in the solution.

Since we won't apply a constant mutational rate throughout the population (i.e. each individual must mutate at least one time in each epoch), in our implementation each epoch's population is subject to a certain level of stress, which defines what percentage of individuals should experience a mutation.

6.2.5 Survival Selection

Survivor selection is a process useful to determine which individuals should be kept in the next generation and which should be kicked out. This process is crucial since it aims to ensure that the fitter individuals are not kicked out of the population, while at the same time diversity should be maintained in the population. In our implementation elitism is performed in order to propagate at each generation at least the champion which is the fittest individual.

Random Policy

Through this policy, each child has the opportunity to join the next generation. In order to make this, every possible child is compared to a random individual of the current population (except the champion) and if it turns out to have a better fitness, the individual is replaced by the child.

Battle Policy

Through this policy, after being sorted, the population is divided into three social classes: fit individuals, average fit individuals, and poorly fit individuals. The classes cover 15%, 70%, and the remaining 15% of the total population, respectively. Once divided into classes, the best individuals and the worst individuals do not participate in the struggle for survival as they are directly propagated in the next generation. On the other hand, individuals belonging to the middle class are required to participate in a battle in order to survive. As many individuals as children are randomly chosen to fight, and during a fight if the child is better than the chosen individual he has a 75% chance of defeating him, otherwise his chance drops to 35%.

6.3 Comparisons and Results

Results obtained from metaheuristic algorithm discussed within this chapter are analyzed below. Performance profiles will be provided to make the results obtained easier to read. Experimental results can be found within tables located in appendix A.5.

Experiments on the metaheuristic algorithms, discussed in sections 6.1 and 6.2, were carried out on instances with a number of nodes ranging from 400 to 5000. For each model 100 different instances were tested in order to obtain results. Since it was not possible to find 100 distinct instances within TSPLIB, 63 random instances were generated.

On the one hand, two different initializations based on the Nearest Neighbors heuristic were compared in order to test Tabu Search metaheuristic capabilities, namely random node selection approach with and without GRASP. This choice was made mainly to speed up the metaheuristic initialization time, since Nearest Neighbors with multistart approach and Extra Mileage are on average slower to build a solution than Nearest Neighbors with random approach. Clearly, this choice allows Tabu Search to be initialized with worse solutions, but it leaves much more time to sift through the search space in depth. As can be seen from the performance profile in figure 6.4, this choice allows Tabu Search to perform a great improvement on the initial solutions presented, and although there is an obvious gap between the starting solutions with and without GRASP, this metaheuristic effectively manages to reduce it of several times.

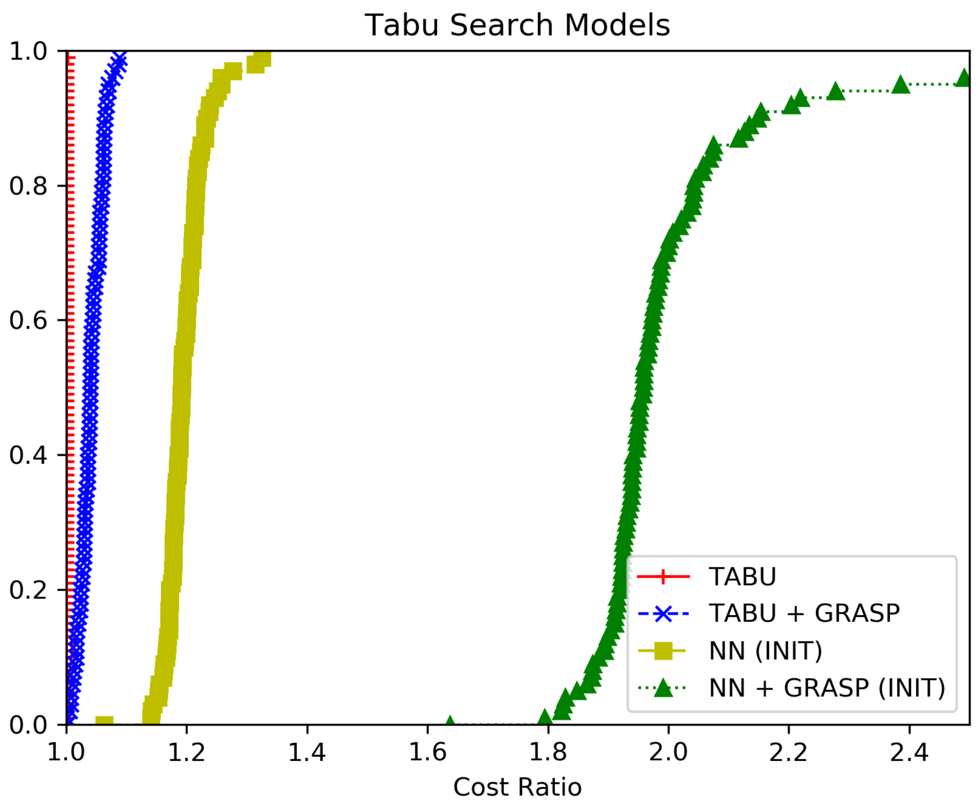
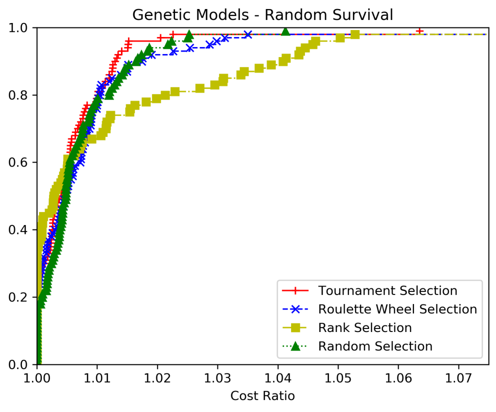


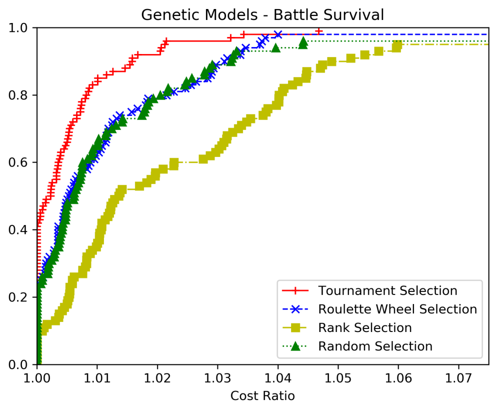
Figure 6.4: Performance profile of Tabu Search metaheuristics. As shown, initialization are done through Nearest Neighbors exploiting multistart approach with and without the activation of GRASP with $k = 3$.

On the other hand, in order to study which was the most suitable configuration for the resolution of the TSP through the genetic algorithm, all the possible combinations of the implemented parent and survival selection policies were analyzed. Each test was performed on an initial population of 1000 individuals, at each epoch 100 offspring were generated. The initial population consists for the 50% of purely random individuals while the remaining 50% are individuals generated through the Nearest Neighbors construction heuristic with random approach and GRASP. Among all the individuals generated via Nearest Neighbors, 15% were optimized via 2-OPT refinement heuristics. In the performance profiles presented in figures 6.5a and 6.5b, it is possible to observe how the results vary under the random survival and battle survival policies, respectively.

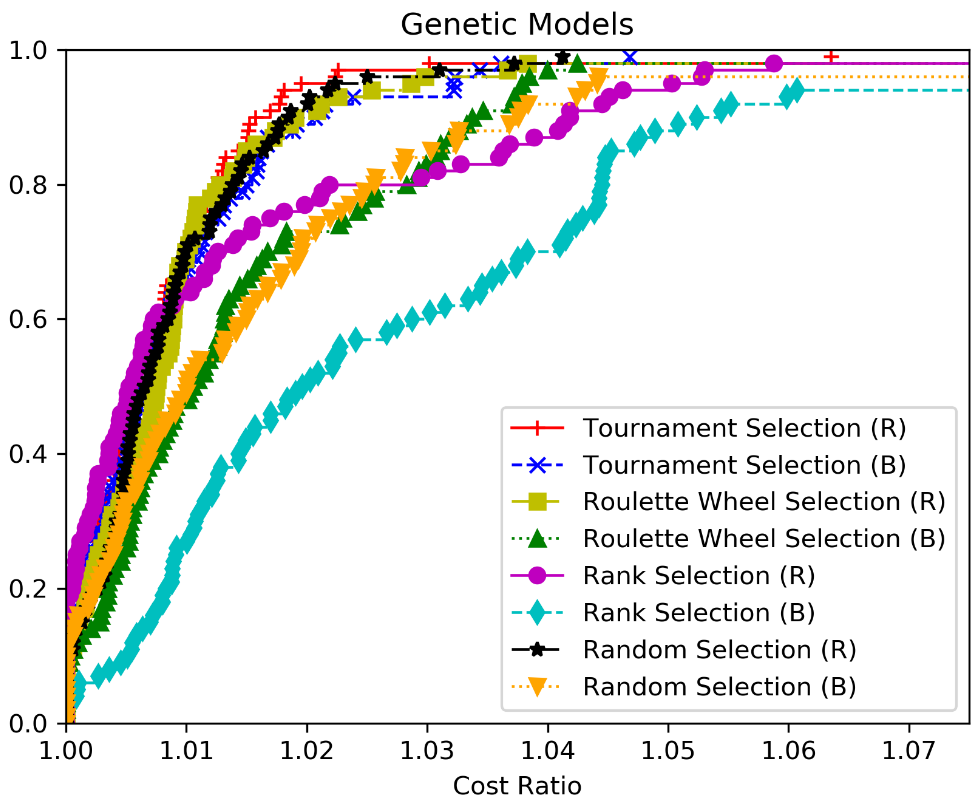
Among the various parent selection policies, it can be clearly seen how Rank Selection behave differently within the two survival policy. While the Tournament Selection seems to be quite stable. Interestingly, selecting a parent via the Roulette Wheel policy or choosing randomly does not result in such a significant difference in the results. Taking now survivor selection policies into account as well, it can be seen that applying a simple policy such as random turns out to be more efficient than a more constraining policy as battle. In summary, the best choice among the tested configurations for TSP resolution turns out to be tournament selection with a random survival policy at the end of each epoch.



(a)



(b)



(c)

Figure 6.5: Performance profiles on genetic algorithm different configurations. In figure 6.5a is shown a comparison of various configuration with the random survivor selection. In figure 6.5b is shown a comparison of various configuration with the battle survivor selection. Lastly, in figure 6.5c is shown the comparison between all the presented genetic configuration in order to highlight which one produces the best solutions.

When analyzing the genetic algorithm, it is useful to track population and champion fitness at the end of each epoch, in order to if the genetic algorithm tends to converge naturally to a sub-optimal solution as the epochs pass or not. This behavior can be observed in figure 6.6.

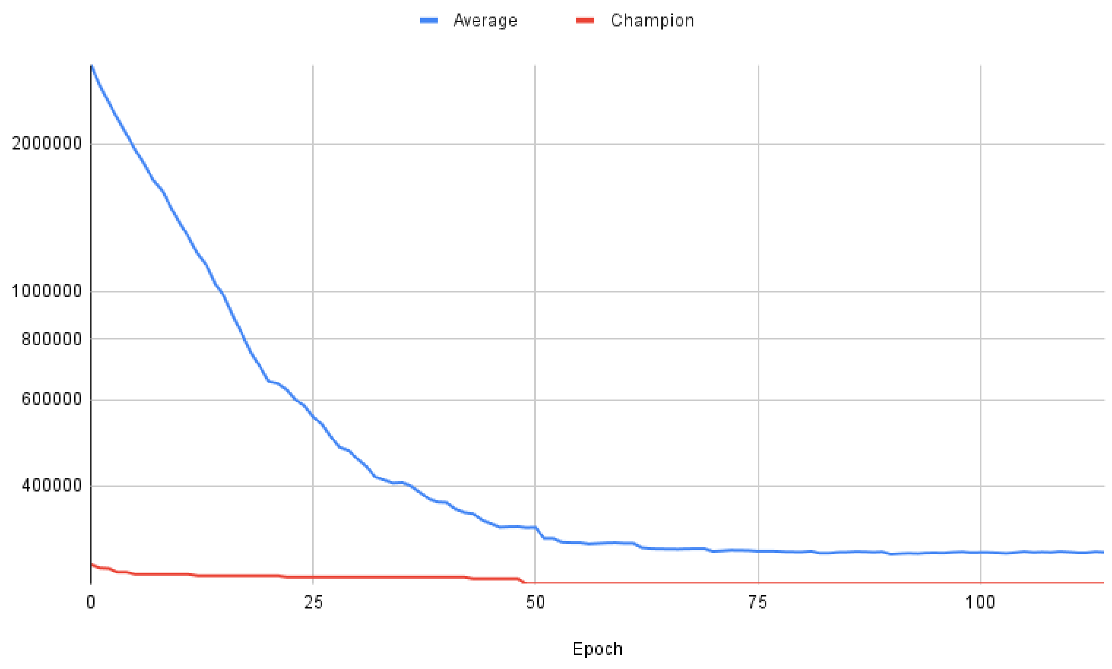


Figure 6.6: Instance pr1002 taken from TSPLIB. Visual example of how within an execution of the genetic algorithm the average value of the fitness of the whole population and the current sample tend to converge towards an optimal solution. In order to facilitate the visualization, and to evidence the convergent characteristic of the curves, the graph is in logarithmic scale.

Comparing now the generit algorithm with Tabu Search, it turns out to be clear how much the Tabu Search in its simplicity turns out to be more performing than the best configuration implemented for the genetic. As can be observed from the performance profile in Figure 6.7, we see that the best configuration of the genetic algorithm can compete with Tabu Search only when its initialized with GRASP. When compared with the version that does not use GRASP, it turns out to be significantly less performing, this is because genetic algorithm tends to have a very slow convergence towards an optimal solution while Tabu Search by construction tries to escape from local minima actually increasing its convergence capabilities.

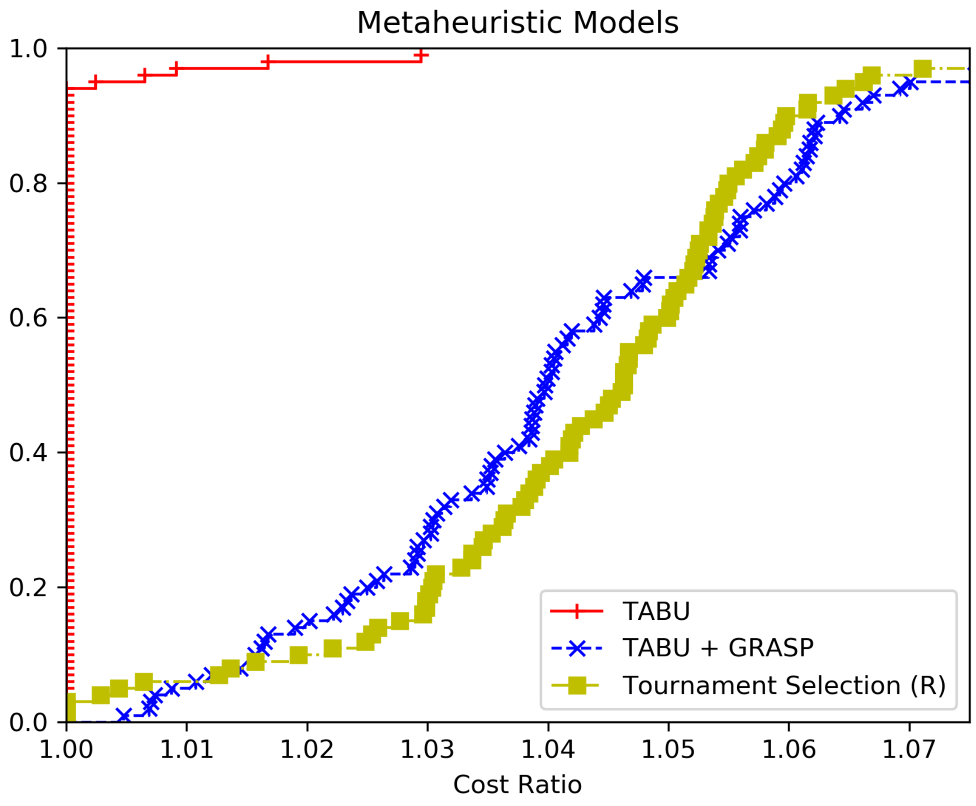


Figure 6.7: Performance profile of Tabu Search and Genetic metaheuristics. Both analyzed version of Tabu Search are compared with the best configuration of the genetic algorithm.

Chapter 7

Conclusion

Within this paper, a variety of models, methods and algorithms able to solve the Traveling Salesman Problem have been presented and analyzed. For each one, several variations and configurations were discussed in order to observe how certain variations affected the overall results.

Starting from the beginning, when the goal is to obtain the optimal solution, as discussed in Chapter 3, one might rely on the class of Exact algorithms that allow to obtain in a reasonably amount of time exact solutions of instances with no more than 700 nodes. Specifically, looking at the results presented in performance profiles 3.4c and 3.5c, and tables A.1, A.2 and A.3, we see how among all the Exact Method presented, Callbacks are able to outperform both Compact Models and Benders' iterative method. It is clear that as the size of the instances increases (over 700 nodes), the Exact Methods start to become less and less performing due to the increasing demands on time and space required for the search of an optimal solution within CPLEX's branching tree. For this reason, it is reasonable to relax the will to obtain exact solutions at all costs and settle for near-optimal ones.

To obtain them, it is possible to rely on heuristic algorithms which, as discussed in Chapters 4, 5 and 6, can be of various kinds. Beginning with Matheuristics, discussed in Chapter 4, it can be observed how CPLEX is able to cooperate with heuristics in order to increase the number of nodes to 1100, obtaining exact solutions only a fraction of times. It is useful to point out that the tricky part in this class of algorithms is to properly set the initial parameters in order to avoid the algorithms to be stuck into local minima. As it can be seen from performance profile 4.3c and table A.4, Hard Fixing is the most promising one.

Further increasing the number of nodes, CPLEX is left aside and as explained in Chapters 5 and 6, heuristic and metaheuristic algorithms have been analyzed to build near-optimal solutions without using any mathematical model. Due to the fact that no mathematical models are used to constrain the choices to be made, in general these classes of algorithms tend to perform slightly worse than matheuristics, but on the other hand they allow the number of nodes in the instances to be increased to 5000 or more. Both construction and refinement heuristics were analyzed. One can observe from the performance profile 5.5 and table A.5 how their combinations with different approaches are able to achieve remarkable results. Finally, with regard to Metaheuristics, Tabu Search and the genetic algorithm in its various configurations were compared, as can be observed from the performance profile 6.7 and table A.6, in our tests Tabu Search turns out to be the most effective.

To conclude this analysis, it makes sense to note that in general there is no optimal algorithm to solve the TSP. Each time, the most suited algorithm must be chosen since it strongly depends on several factors (i.e., size and complexity of the instance, optimal or near-optimal solution, time limit, etc.).

Appendix A

Experimental Results

All instances were executed within a computing cluster and since its resources are used competitively among all its nodes, a conversion from seconds to ticks was performed to avoid times inconsistency. Execution time limit was set to 1020000 ticks which are equivalent to 3600 seconds.

A.1 TSPLIB

TSPLIB [10] is a library containing sample instances for TSP and related problems, gathering together data taken from various sources and of various types. As discussed in chapter 2, we mostly focused on instances of the symmetric TSP, considering the asymmetric one only in the case of compact models 3. Moreover, the best known solution is provided for each instance and can be found here.

A.2 Exact Algorithms

			MTZ		**MTZ Modified**		**MTZ Lazy**		**MTZ Lazy Sec**	
	Run	**Solution**	**Ticks**	**Bound**	**Ticks**	**Bound**	**Ticks**	**Bound**	**Ticks**	**Bound**
att48	1	10628	36262	✓	75176	✓	479231	✓	14310	✓
berlin52	1	7542	996	✓	1768	✓	831	✓	252	✓
burma14	1	3323	54	✓	43	✓	18	✓	15	✓
eil101	1	629	27870	✓	21656	✓	112387	✓	4492	✓
eil51	1	426	1154	✓	2368	✓	1079	✓	464	✓
eil76	1	538	4773	✓	3565	✓	3016	✓	1045	✓
gr96	1	55209	168243	✓	191170	✓	-	✓	259180	✓
kroA100	1	21282	376235	✓	250871	✓	-	30308	-	✓
kroB100	1	22141	-	✓	-	✓	-	33474	-	22212
kroC100	1	20749	-	✓	-	✓	-	27201	912943	✓
kroD100	1	21294	-	21925	-	21870	-	22652	-	25910
kroE100	1	22068	388015	✓	408210	✓	-	✓	-	22240
lin105	1	14379	-	✓	-	✓	-	16103	89580	✓
pr107	1	44303	-	49924	-	53525	-	54525	-	75105
pr76	1	108159	163569	✓	207394	✓	129796	✓	418246	✓
rat99	1	1211	15514	✓	25171	✓	19759	✓	3724	✓
rd100	1	7910	93139	✓	111105	✓	160704	✓	123890	✓
st70	1	675	979800	✓	541125	✓	-	✓	17423	✓
ulysses16	1	6859	482	✓	481	✓	280	✓	438	✓
ulysses22	1	7013	62863	✓	52147	✓	13047	✓	17178	✓
att48	2	10628	85170	✓	62386	✓	157756	✓	22468	✓
berlin52	2	7542	1156	✓	1466	✓	1040	✓	391	✓
burma14	2	3323	32	✓	46	✓	19	✓	16	✓

Continued on next page

eil101	2	629	17387	✓	17662	✓	26192	✓	5373	✓
eil51	2	426	1662	✓	1781	✓	974	✓	286	✓
eil76	2	538	3837	✓	2710	✓	6836	✓	1061	✓
gr96	2	55209	349797	✓	256862	✓	710483	✓	-	✓
kroA100	2	21282	512327	✓	397027	✓	-	29715	460671	✓
kroB100	2	22141	-	✓	-	✓	-	27495	-	30499
kroC100	2	20749	-	✓	-	20769	-	25802	-	✓
kroD100	2	21294	-	21309	-	21835	-	23236	-	25211
kroE100	2	22068	296933	✓	530984	✓	-	22219	399870	✓
lin105	2	14379	-	✓	-	14485	-	15415	165014	✓
pr107	2	44303	-	48015	-	49373	-	52373	-	93529
pr76	2	108159	113185	✓	194089	✓	145097	✓	-	108313
rat99	2	1211	25498	✓	19191	✓	25276	✓	3654	✓
rd100	2	7910	115240	✓	147882	✓	-	✓	106719	✓
st70	2	675	-	✓	377707	✓	-	✓	43633	✓
ulysses16	2	6859	419	✓	609	✓	231	✓	388	✓
ulysses22	2	7013	82863	✓	7734	✓	21989	✓	26996	✓
att48	3	10628	80379	✓	70098	✓	255218	✓	18292	✓
berlin52	3	7542	779	✓	1503	✓	1050	✓	363	✓
burma14	3	3323	68	✓	42	✓	18	✓	17	✓
eil101	3	629	24822	✓	18446	✓	143169	✓	2472	✓
eil51	3	426	1397	✓	1810	✓	1210	✓	317	✓
eil76	3	538	4996	✓	3737	✓	4880	✓	1511	✓
gr96	3	55209	176989	✓	636890	✓	528955	✓	-	✓
kroA100	3	21282	-	✓	716758	✓	727125	✓	-	21456
kroB100	3	22141	-	✓	-	22146	-	26475	-	22331
kroC100	3	20749	-	✓	-	✓	-	28757	-	20901
kroD100	3	21294	-	21309	-	21493	-	23236	-	25910
kroE100	3	22068	258457	✓	319861	✓	-	22100	528227	✓
lin105	3	14379	-	✓	-	✓	-	17141	-	✓
pr107	3	44303	-	53783	-	53966	-	53966	-	67289
pr76	3	108159	132615	✓	205651	✓	631199	✓	718724	✓
rat99	3	1211	14193	✓	46515	✓	32372	✓	3424	✓
rd100	3	7910	98470	✓	176110	✓	314752	✓	86271	✓
st70	3	675	-	✓	367635	✓	722654	✓	58994	✓
ulysses16	3	6859	429	✓	688	✓	265	✓	554	✓
ulysses22	3	7013	64662	✓	55009	✓	10590	✓	27562	✓
att48	4	10628	95614	✓	55919	✓	623873	✓	18072	✓
berlin52	4	7542	848	✓	1142	✓	1135	✓	342	✓
burma14	4	3323	38	✓	36	✓	19	✓	14	✓
eil101	4	629	11886	✓	9398	✓	39362	✓	3714	✓
eil51	4	426	1341	✓	2311	✓	1284	✓	382	✓
eil76	4	538	3775	✓	6152	✓	9654	✓	1142	✓
gr96	4	55209	371235	✓	313583	✓	-	56703	-	✓
kroA100	4	21282	435659	✓	-	21305	-	34693	370741	✓
kroB100	4	22141	-	✓	-	22284	-	28687	-	22141
kroC100	4	20749	-	✓	-	✓	-	28757	-	20769
kroD100	4	21294	-	✓	-	21803	-	26568	-	27238
kroE100	4	22068	354879	✓	-	✓	664092	✓	232236	✓
lin105	4	14379	-	✓	-	✓	-	15108	144789	✓
pr107	4	44303	-	45289	-	44769	-	45799	-	74741
pr76	4	108159	138834	✓	182535	✓	314953	✓	119580	✓
rat99	4	1211	24759	✓	17164	✓	15694	✓	3613	✓
rd100	4	7910	126303	✓	179076	✓	-	✓	200492	✓
st70	4	675	195011	✓	892091	✓	-	✓	51280	✓
ulysses16	4	6859	422	✓	656	✓	393	✓	349	✓
ulysses22	4	7013	63722	✓	43437	✓	20494	✓	20785	✓
att48	5	10628	66413	10628	88322	✓	317099	✓	16819	✓
berlin52	5	7542	1045	7542	1424	✓	990	✓	266	✓

Continued on next page

burma14	5	3323	32	3323	38	✓	25	✓	16	✓
eil101	5	629	9922	629	18614	✓	-	942	4575	✓
eil51	5	426	2460	426	1364	✓	1085	✓	373	✓
eil76	5	538	2861	538	5130	✓	7180	✓	1470	✓
gr96	5	55209	507059	55209	272473	✓	-	55291	191470	✓
kroA100	5	21282	322887	21282	429685	✓	779061	✓	-	21338
kroB100	5	22141	-	✓	-	✓	-	36613	-	28096
kroC100	5	20749	-	✓	-	✓	-	25802	-	20882
kroD100	5	21294	-	✓	-	21577	-	25321	-	26016
kroE100	5	22068	354978	✓	-	✓	-	✓	313824	✓
lin105	5	14379	-	✓	-	✓	-	15108	622505	✓
pr107	5	44303	-	46576	-	51314	-	52514	-	79761
pr76	5	108159	112082	✓	149375	✓	556491	✓	173462	✓
rat99	5	1211	22338	✓	21147	✓	9721	✓	3806	✓
rd100	5	7910	31979	✓	113647	✓	193178	✓	87062	✓
st70	5	675	245064	✓	-	✓	-	676	39918	✓
ulysses16	5	6859	516	✓	594	✓	218	✓	429	✓
ulysses22	5	7013	8943	✓	61750	✓	15940	✓	13997	✓

Table A.1: Results of the Miller, Tucker and Zemlin (MTZ) compact models. Each computing time is expressed in ticks and represent the amount of time required by CPLEX to compute the exact solution for a given instance with a specific compact model. Inside the table '−' means that the model hit the time limit of 1020000 ticks and '✓' means that the exact solution was found.

			GG		GG Original		GG Lazy		GG Lazy Sec	
	Run	**Solution**	**Ticks**	**Bound**	**Ticks**	**Bound**	**Ticks**	**Bound**	**Ticks**	**Bound**
att48	1	10628	4859	✓	4945	✓	6758	✓	4048	✓
berlin52	1	7542	2991	✓	3731	✓	6055	✓	6045	✓
burma14	1	3323	101	✓	111	✓	32	✓	51	✓
eil101	1	629	78761	✓	107353	✓	789795	✓	123178	✓
eil51	1	426	10654	✓	8877	✓	8828	✓	2385	✓
eil76	1	538	36151	✓	39052	✓	107991	✓	9494	✓
gr96	1	55209	141925	✓	97422	✓	-	61023	155885	✓
kroA100	1	21282	220717	✓	170176	✓	-	26546	496120	✓
kroB100	1	22141	258113	✓	216676	✓	-	44012	305274	✓
kroC100	1	20749	190050	✓	234621	✓	-	21680	142115	✓
kroD100	1	21294	124790	✓	112183	✓	-	24864	142131	✓
kroE100	1	22068	181659	✓	232453	✓	-	23734	319008	✓
lin105	1	14379	97718	✓	125347	✓	-	14527	122426	✓
pr107	1	44303	-	✓	-	✓	-	53970	491873	✓
pr76	1	108159	111408	✓	100507	✓	-	108202	228074	✓
rat99	1	1211	97387	✓	71024	✓	477518	✓	38112	✓
rd100	1	7910	139851	✓	116202	✓	967941	✓	189714	✓
st70	1	675	20770	✓	28228	✓	184283	✓	41883	✓
ulysses16	1	6859	171	✓	298	✓	65	✓	88	✓
ulysses22	1	7013	421	✓	625	✓	711	✓	203	✓
att48	2	10628	7873	✓	6671	✓	7018	✓	3575	✓
berlin52	2	7542	4415	✓	4193	✓	5552	✓	2441	✓
burma14	2	3323	125	✓	111	✓	34	✓	51	✓
eil101	2	629	85519	✓	63816	✓	762334	✓	59104	✓
eil51	2	426	8010	✓	11538	✓	11630	✓	2704	✓
eil76	2	538	36489	✓	38037	✓	235045	✓	8781	✓
gr96	2	55209	152075	✓	112881	✓	-	56602	243996	✓
kroA100	2	21282	196997	✓	83452	✓	-	25810	212393	✓
kroB100	2	22141	128965	✓	115380	✓	-	28514	242829	✓
kroC100	2	20749	110921	✓	110395	✓	-	28298	144645	✓
kroD100	2	21294	135318	✓	158398	✓	-	21720	138698	✓
kroE100	2	22068	264654	✓	196265	✓	-	22848	158686	✓
lin105	2	14379	103330	✓	100311	✓	-	17809	335899	✓
pr107	2	44303	-	45557	-	44387	-	57955	318108	✓
pr76	2	108159	103580	✓	85504	✓	-	✓	349754	✓
rat99	2	1211	86980	✓	113623	✓	310787	✓	35508	✓
rd100	2	7910	102148	✓	134825	✓	-	9740	96725	✓
st70	2	675	26782	✓	37350	✓	368426	✓	34711	✓
ulysses16	2	6859	139	✓	219	✓	81	✓	96	✓
ulysses22	2	7013	650	✓	709	✓	441	✓	225	✓
att48	3	10628	5671	✓	4637	✓	6760	✓	3661	✓
berlin52	3	7542	5132	✓	10153	✓	6098	✓	2866	✓
burma14	3	3323	123	✓	121	✓	35	✓	53	✓
eil101	3	629	91665	✓	87988	✓	-	629	82707	✓
eil51	3	426	9277	✓	9797	✓	8294	✓	2444	✓
eil76	3	538	47889	✓	76408	✓	74019	✓	10910	✓
gr96	3	55209	66338	✓	158225	✓	997186	✓	132040	✓
kroA100	3	21282	151897	✓	135549	✓	-	27945	530381	✓
kroB100	3	22141	166583	✓	183686	✓	-	27625	259860	✓
kroC100	3	20749	153256	✓	219821	✓	-	30730	107677	✓
kroD100	3	21294	146074	✓	170585	✓	-	22684	129021	✓
kroE100	3	22068	157222	✓	248795	✓	-	24538	332147	✓
lin105	3	14379	98034	✓	111678	✓	-	14391	124344	✓
pr107	3	44303	-	✓	-	✓	-	76728	877247	✓
pr76	3	108159	131909	✓	85853	✓	982221	✓	191614	✓
rat99	3	1211	106294	✓	132573	✓	964570	✓	65592	✓
rd100	3	7910	74650	✓	131294	✓	-	8392	123086	✓

Continued on next page

st70	3	675	42370	✓	49389	✓	784232	✓	34499	✓
ulysses16	3	6859	177	✓	175	✓	94	✓	62	✓
ulysses22	3	7013	333	✓	556	✓	567	✓	190	✓
att48	4	10628	6085	✓	5403	✓	7044	✓	3532	✓
berlin52	4	7542	4321	✓	2468	✓	5861	✓	5428	✓
burma14	4	3323	112	✓	114	✓	33	✓	52	✓
eil101	4	629	90890	✓	95665	✓	1015271	✓	80725	✓
eil51	4	426	11232	✓	12548	✓	9225	✓	3997	✓
eil76	4	538	29330	✓	56244	✓	164208	✓	7760	✓
gr96	4	55209	96956	✓	157187	✓	-	55572	118737	✓
kroA100	4	21282	128284	✓	159641	✓	-	25969	358436	✓
kroB100	4	22141	240281	✓	153776	✓	-	28298	328304	✓
kroC100	4	20749	100838	✓	112971	✓	-	25122	195223	✓
kroD100	4	21294	157187	✓	164205	✓	-	21309	97699	✓
kroE100	4	22068	232249	✓	84448	✓	-	29038	529701	✓
lin105	4	14379	101763	✓	-	✓	-	20120	287949	✓
pr107	4	44303	-	✓	-	✓	-	56771	333739	✓
pr76	4	108159	100700	✓	65418	✓	711724	✓	264358	✓
rat99	4	1211	140453	✓	104996	✓	294332	✓	28224	✓
rd100	4	7910	118316	✓	94516	✓	-	10667	86979	✓
st70	4	675	41291	✓	31166	✓	171936	✓	25324	✓
ulysses16	4	6859	146	✓	272	✓	105	✓	71	✓
ulysses22	4	7013	512	✓	499	✓	376	✓	173	✓
att48	5	10628	6244	10628	8833	✓	8381	✓	3930	✓
berlin52	5	7542	2412	7542	2407	✓	5279	✓	3289	✓
burma14	5	3323	125	3323	125	✓	25	✓	51	✓
eil101	5	629	100179	629	84708	✓	-	756	65193	✓
eil51	5	426	9034	426	8565	✓	9273	✓	2500	✓
eil76	5	538	39034	538	65133	✓	173421	✓	11415	✓
gr96	5	55209	172665	55209	104809	✓	884776	✓	152632	✓
kroA100	5	21282	123493	21282	197038	✓	-	25621	172166	✓
kroB100	5	22141	137341	22141	150391	✓	-	27600	402097	✓
kroC100	5	20749	131726	20749	208186	✓	-	22160	137099	✓
kroD100	5	21294	106699	21294	142628	✓	-	22841	166958	✓
kroE100	5	22068	212650	22068	153860	✓	-	27361	274079	✓
lin105	5	14379	102395	14379	161478	✓	-	21604	153145	✓
pr107	5	44303	-	44339	-	✓	-	51663	-	✓
pr76	5	108159	70347	108159	78881	✓	-	✓	212878	✓
rat99	5	1211	135199	1211	147273	✓	742575	✓	37437	✓
rd100	5	7910	97676	7910	91820	✓	-	7917	104990	✓
st70	5	675	24995	675	24584	✓	237248	✓	37027	✓
ulysses16	5	6859	258	6859	181	✓	61	✓	68	✓
ulysses22	5	7013	458	7013	420	✓	759	✓	243	✓

Table A.2: Results of the Garvish and Graves (GG) compact models. Each computing time is expressed in ticks and represent the amount of time required by CPLEX to compute the exact solution for a given instance with a specific compact model. Inside the table '−' means that the model hit the time limit of 1020000 ticks and '✓' means that the exact solution was found.

			Callback		Callback (2OPT)		Callback (V1)		Callback (V2)		Callback (V3)		Benders		Benders (Kruskal)	
	Run	Solution	Ticks	Best	Ticks	Best	Ticks)	Best	Ticks	Best	Ticks	Best	Best	Ticks)	Best	Ticks
a280	1	2579	3488	✓	3155	✓	2772	✓	2782	✓	2667	✓	16232	✓	21421	✓
ali535	1	202310	51636	✓	46132	✓	100656	✓	61182	✓	78859	✓	169161	✓	236508	✓
att532	1	27686	133631	✓	167823	✓	197372	✓	100348	✓	110209	✓	441106	✓	478698	✓
d493	1	35002	50294	✓	54471	✓	48619	✓	52937	✓	58692	✓	971896	✓	-	35127
fl417	1	11861	84747	✓	79754	✓	59171	✓	64677	✓	55443	✓	231903	✓	169606	✓
gil262	1	2378	7315	✓	6317	✓	8244	✓	8175	✓	6629	✓	34829	✓	24583	✓
gr202	1	40160	2277	✓	2277	✓	1307	✓	1418	✓	2230	✓	15394	✓	15139	✓
gr229	1	134602	10661	✓	7084	✓	7709	✓	6632	✓	6516	✓	32712	✓	29781	✓
gr431	1	171414	36929	✓	56197	✓	66200	✓	39961	✓	38361	✓	212985	✓	228943	✓
gr666	1	294358	173834	✓	105620	✓	146083	✓	67314	✓	79444	✓	424854	✓	441052	✓
kroA200	1	29368	6074	✓	4437	✓	4052	✓	4740	✓	4608	✓	15175	✓	16005	✓
kroB200	1	29437	3415	✓	2647	✓	2631	✓	2590	✓	3096	✓	6697	✓	7108	✓
lin318	1	42029	13365	✓	11458	✓	9235	✓	14710	✓	11384	✓	30190	✓	31090	✓
p654	1	34643	71234	✓	197527	✓	138419	✓	109192	✓	304273	✓	-	35932	-	35932
pcb442	1	50778	10715	✓	7626	✓	7956	✓	8506	✓	10118	✓	-	50810	-	50810
pr226	1	80369	6833	✓	4478	✓	18221	✓	12773	✓	8145	✓	40504	✓	28181	✓
pr264	1	49135	2951	✓	2572	✓	5594	✓	6087	✓	3904	✓	69175	✓	455812	✓
pr299	1	48191	36015	✓	12605	✓	41659	✓	40919	✓	40581	✓	87978	✓	86920	✓
pr439	1	107217	98058	✓	83102	✓	99032	✓	79226	✓	76319	✓	381532	✓	378040	✓
random541	1	28313	64409	✓	77324	✓	61243	✓	62150	✓	99701	✓	356216	✓	-	✓
rat575	1	6773	171323	✓	111773	✓	114144	✓	75096	✓	89470	✓	395583	✓	557125	✓
rd400	1	15281	20993	✓	17356	✓	16911	✓	12676	✓	27132	✓	199296	✓	225643	✓
ts225	1	126643	-	126726	-	126726	-	126643	-	126758	-	126810	-	12668	-	12668
tsp225	1	3919	3120	✓	3000	✓	2698	✓	4093	✓	4249	✓	12702	✓	10158	✓
u574	1	36905	128558	✓	54990	✓	118450	✓	107657	✓	127630	✓	209852	✓	234419	✓
a280	2	2579	3629	✓	1729	✓	3396	✓	3726	✓	2959	✓	28128	✓	23764	✓
ali535	2	202310	69226	✓	39042	✓	116865	✓	54622	✓	67340	✓	233834	✓	318724	✓
att532	2	27686	197891	✓	80015	✓	256755	✓	92860	✓	177853	✓	451780	✓	470888	✓
d493	2	35002	51153	✓	44861	✓	78943	✓	50270	✓	68081	✓	-	35127	-	35127
fl417	2	11861	69237	✓	47765	✓	56038	✓	124677	✓	64738	✓	143524	✓	186928	✓
gil262	2	2378	8279	✓	6059	✓	5925	✓	5925	✓	9086	✓	26156	✓	31964	✓
gr202	2	40160	1247	✓	2494	✓	1802	✓	1157	✓	1582	✓	14995	✓	14936	✓
gr229	2	134602	6757	✓	6700	✓	7066	✓	13948	✓	6688	✓	31042	✓	31226	✓
gr431	2	171414	43736	✓	46471	✓	50068	✓	49466	✓	47926	✓	265538	✓	-	171430
gr666	2	294358	111569	✓	205956	✓	233896	✓	191649	✓	150142	✓	457806	✓	550291	✓

Continued on next page

kroA200	2	29368	4946	✓	2683	✓	4366	✓	4058	✓	4010	✓	17217	✓	17461	✓
kroB200	2	29437	2213	✓	1603	✓	2122	✓	2531	✓	3352	✓	6961	✓	7561	✓
lin318	2	42029	10780	✓	11960	✓	10134	✓	9595	✓	7829	✓	30152	✓	33720	✓
p654	2	34643	173128	✓	49893	✓	44538	✓	59654	✓	222649	✓	-	35920	-	35920
pcb442	2	50778	7542	✓	9319	✓	13423	✓	9176	✓	8283	✓	-	50798	-	50798
pr226	2	80369	3204	✓	5400	✓	9952	✓	9464	✓	5330	✓	40200	✓	27570	✓
pr264	2	49135	4798	✓	3428	✓	5256	✓	5051	✓	3645	✓	72439	✓	71548	✓
pr299	2	48191	49840	✓	41842	✓	40686	✓	42505	✓	41935	✓	80047	✓	-	✓
pr439	2	107217	214235	✓	157108	✓	85921	✓	106656	✓	85940	✓	345779	✓	347974	✓
random541	2	28313	105313	✓	59185	✓	90018	✓	222308	✓	276815	✓	363981	✓	396489	✓
rat575	2	6773	60370	✓	126974	✓	142007	✓	54776	✓	70096	✓	541410	✓	-	✓
rd400	2	15281	23512	✓	21899	✓	41346	✓	28673	✓	24392	✓	185399	✓	179295	✓
ts225	2	126643	-	126726	-	126726	-	126643	-	126643	-	126726	-	126657	-	126657
tsp225	2	3919	3817	✓	2924	✓	2684	✓	2899	✓	3016	✓	9010	✓	11222	✓
u574	2	36905	95023	✓	69448	✓	123410	✓	204956	✓	111435	✓	294559	✓	212577	✓
a280	3	2579	3317	✓	2072	✓	2970	✓	2928	✓	2663	✓	14852	✓	12550	✓
ali535	3	202310	100751	✓	80914	✓	78061	✓	55495	✓	44250	✓	256374	✓	335780	✓
att532	3	27686	148007	✓	50126	✓	156663	✓	107943	✓	131895	✓	510919	✓	488988	✓
d493	3	35002	44446	✓	35587	✓	121819	✓	67278	✓	62713	✓	-	35127	-	35127
fl417	3	11861	68385	✓	211197	✓	50022	✓	259172	✓	72441	✓	230162	✓	185331	✓
gil262	3	2378	8390	✓	6602	✓	5494	✓	8063	✓	6131	✓	31557	✓	25412	✓
gr202	3	40160	1896	✓	1772	✓	2300	✓	924	✓	936	✓	15294	✓	15285	✓
gr229	3	134602	7160	✓	7675	✓	5262	✓	6991	✓	9090	✓	33056	✓	33793	✓
gr431	3	171414	54403	✓	40961	✓	81130	✓	49957	✓	55719	✓	295637	✓	232919	✓
gr666	3	294358	151449	✓	66653	✓	105377	✓	117086	✓	84235	✓	415912	✓	396276	✓
kroA200	3	29368	4864	✓	4002	✓	3324	✓	4474	✓	3861	✓	19118	✓	15759	✓
kroB200	3	29437	3514	✓	2746	✓	2708	✓	3713	✓	3247	✓	7015	✓	7197	✓
lin318	3	42029	10290	✓	12313	✓	10344	✓	7521	✓	10543	✓	30095	✓	31162	✓
p654	3	34643	114135	✓	236031	✓	80686	✓	193635	✓	59595	✓	-	35920	-	35920
pcb442	3	50778	12254	✓	8330	✓	8235	✓	8244	✓	8097	✓	-	50798	-	50798
pr226	3	80369	10188	✓	8963	✓	6667	✓	9460	✓	3367	✓	37539	✓	35540	✓
pr264	3	49135	6248	✓	4339	✓	4263	✓	3995	✓	5417	✓	74879	✓	82212	✓
pr299	3	48191	44862	✓	24525	✓	29678	✓	34980	✓	39088	✓	80517	✓	-	✓
pr439	3	107217	140333	✓	63096	✓	61180	✓	123353	✓	103187	✓	344037	✓	374767	✓
random541	3	28313	80261	✓	72551	✓	150723	✓	105418	✓	86850	✓	360436	✓	-	✓
rat575	3	6773	84918	✓	213235	✓	207624	✓	72516	✓	108993	✓	424985	✓	422359	✓
rd400	3	15281	22505	✓	40425	✓	41346	✓	16415	✓	26154	✓	230649	✓	271762	✓

Continued on next page

ts225	3	126643	-	126810	-	126809	-	126643	-	126643	-	126643	-	126657	-	126657
tsp225	3	3919	2586	✓	3925	✓	4285	✓	4110	✓	4123	✓	10796	✓	8323	✓
u574	3	36905	95631	✓	91630	✓	193433	✓	365462	✓	132459	✓	219932	✓	237462	✓
a280	4	2579	2456	✓	3543	✓	1565	✓	3377	✓	2202	✓	19140	✓	14025	✓
ali535	4	202310	59308	✓	31440	✓	57262	✓	44931	✓	51062	✓	192587	✓	145411	✓
att532	4	27686	135563	✓	82703	✓	249870	✓	99891	✓	102377	✓	630231	✓	478477	✓
d493	4	35002	45702	✓	67642	✓	53146	✓	50491	✓	43074	✓	-	35127	-	35127
fl417	4	11861	83155	✓	177480	✓	54774	✓	53662	✓	54485	✓	340204	✓	198884	✓
gil262	4	2378	7436	✓	8226	✓	8720	✓	6805	✓	12225	✓	23846	✓	26970	✓
gr202	4	40160	1397	✓	2119	✓	1715	✓	1488	✓	1157	✓	15166	✓	-	✓
gr229	4	134602	6013	✓	6133	✓	6196	✓	6118	✓	6543	✓	32996	✓	31968	✓
gr431	4	171414	57561	✓	39930	✓	52960	✓	35924	✓	37266	✓	220652	✓	224323	✓
gr666	4	294358	141559	✓	116792	✓	401973	✓	110128	✓	88625	✓	429332	✓	393234	✓
kroA200	4	29368	5411	✓	3313	✓	4235	✓	5168	✓	5036	✓	17406	✓	15160	✓
kroB200	4	29437	3463	✓	2736	✓	2523	✓	3530	✓	3505	✓	7644	✓	6747	✓
lin318	4	42029	11384	✓	12026	✓	11213	✓	9849	✓	10038	✓	30678	✓	-	✓
p654	4	34643	123296	✓	80509	✓	62211	✓	335485	✓	31771	✓	-	35920	-	35920
pcb442	4	50778	10651	✓	9526	✓	7985	✓	8089	✓	9743	✓	-	50798	-	50798
pr226	4	80369	8235	✓	5626	✓	8327	✓	5494	✓	9032	✓	56936	✓	76886	✓
pr264	4	49135	3731	✓	4001	✓	4200	✓	2539	✓	5383	✓	75157	✓	90180	✓
pr299	4	48191	38069	✓	24966	✓	27521	✓	62767	✓	50857	✓	84679	✓	85784	✓
pr439	4	107217	98059	✓	92959	✓	110548	✓	60750	✓	105451	✓	351088	✓	324457	✓
random541	4	28313	104589	✓	70044	✓	80829	✓	70423	✓	85788	✓	328061	✓	325424	✓
rat575	4	6773	114671	✓	79480	✓	103737	✓	102085	✓	73459	✓	462673	✓	391770	✓
rd400	4	15281	27529	✓	20073	✓	41346	✓	19516	✓	25122	✓	182857	✓	223686	✓
ts225	4	126643	-	127056	-	126713	-	126643	-	126726	-	126726	-	126657	-	126657
tsp225	4	3919	3326	✓	1978	✓	2553	✓	2764	✓	2947	✓	11261	✓	11266	✓
u574	4	36905	199514	✓	83337	✓	86064	✓	158847	✓	77184	✓	220265	✓	-	✓

Table A.3: Results of the Callback method. In particular, Callback V1 refers to the variant that takes into account only integer solutions, Callback V2 to the ability to make user-cut go up to nodes with depth 5, and Callback V3 to the generation of SEC inside fractional solutions when there is more than one connected component. Each computing time is expressed in ticks and represent the amount of time required by CPLEX to compute the exact solution for a given instance with a specific compact model. Inside the table '–' means that the model hit the time limit of 1020000 ticks and '✓' means that the exact solution was found.

A.3 Matheuristic Algorithms

	Run	Solution	Hard 85%	Hard 90%	Hard 95%	Soft (2)	Soft (3)	Soft (5)
a280	1	2579	✓	2602	✓	3432	✓	2600
ali535	1	202310	204158	205864	205993	273029	2049261	1297401
att532	1	27686	28057	27765	27758	30848	135255	76593
d493	1	35002	35133	35160	35122	39533	399755	222367
d657	1	48912	49222	49282	49204	144712	740286	760486
fl417	1	11861	11878	11862	11861	11870	439337	265611
gil262	1	2378	2385	2380	✓	2380	2380	2385
gr202	1	40160	✓	✓	40175	✓	✓	✓
gr431	1	171414	171975	171516	171778	177382	1596841	970342
gr666	1	294358	297263	295532	296313	707837	5087203	4449720
lin318	1	42029	43051	43051	42694	42155	126465	377371
p654	1	34643	34899	36443	36443	82347	2102218	1499622
pcb442	1	50778	50897	50883	51206	127207	744923	602413
pr1002	1	259045	265821	265632	266556	3887291	6646166	6083165
pr264	1	49135	✓	✓	✓	✓	✓	✓
pr299	1	48191	48279	✓	48312	48279	50901	48597
pr439	1	107217	109357	108375	108375	115491	1563045	✓
random541	1	28313	28409	28441	28400	42278	157314	31159
rat575	1	6773	6824	6802	6815	7053	28212	138959
rat783	1	8806	8886	8892	8884	49865	171511	101482
rd400	1	15281	15352	15394	15335	46128	184512	15284
u1060	1	224094	234260	229645	231286	3854112	6634677	561038
u574	1	36905	37246	37107	37229	148428	667926	682796
u724	1	41910	42351	42295	42097	180474	776840	7719095
vm1084	1	239297	252484	264351	248103	5087020	8386005	1105695
a280	2	2579	2600	2600	✓	3541	3974	2600
ali535	2	202310	202375	202662	203191	270589	2015454	514896
att532	2	27686	28515	27773	27798	34658	188190	42711
d493	2	35002	35160	35167	35114	38162	485534	280216
d657	2	48912	49043	49016	49088	144524	767187	763988
fl417	2	11861	11870	11870	11877	11861	368800	313793
gil262	2	2378	2380	2380	2385	✓	2384	✓
gr202	2	40160	40190	✓	✓	✓	✓	✓
gr431	2	171414	171450	171997	172268	172246	1675132	1335713
gr666	2	294358	296964	297226	295349	846111	5118508	3355493
lin318	2	42029	42182	✓	43051	43051	129153	✓
p654	2	34643	34776	38617	34743	72886	2103875	1628794
pcb442	2	50778	50912	50927	51095	117132	104939	580566
pr1002	2	259045	265472	265642	266094	3718988	6343614	5806726
pr264	2	49135	✓	49431	49378	✓	✓	✓
pr299	2	48191	48352	48597	48597	48466	48279	48224
pr439	2	107217	107250	107752	109388	113491	1103248	127623
random541	2	28313	28403	28533	28379	42570	278004	32250
rat575	2	6773	6799	6818	6808	6953	26851	164424
rat783	2	8806	8899	8861	8875	49110	170381	101330
rd400	2	15281	15326	15355	15394	46059	184236	15285
u1060	2	224094	233306	229834	231101	3681894	6697766	564150
u574	2	36905	37270	37316	37211	149246	671607	701195
u724	2	41910	42764	43778	42080	175112	778033	8133118
vm1084	2	239297	248602	244635	245975	4892700	8513145	1206425
a280	3	2579	✓	✓	✓	3789	✓	2580
ali535	3	202310	205875	205843	206393	279207	2050347	1493989
att532	3	27686	27953	27726	27920	34658	139944	30138
d493	3	35002	35157	35092	35127	37957	484719	286739
d657	3	48912	49026	49016	49090	116847	860834	374197

Continued on next page

fl417	3	11861	11881	11878	11862	11870	372890	330768
gil262	3	2378	2384	2385	2395	✓	✓	2385
gr202	3	40160	✓	✓	✓	✓	✓	✓
gr431	3	171414	172128	172561	172395	177681	2008990	1165211
gr666	3	294358	295891	295270	294976	877897	3846127	2969646
lin318	3	42029	42315	46942	42694	42943	115946	42091
p654	3	34643	36425	36413	36413	77234	1800511	1875216
pcb442	3	50778	51160	51147	50961	96761	104939	492633
pr1002	3	259045	265807	264296	266094	3648512	6343614	5643490
pr264	3	49135	49378	50325	✓	✓	✓	✓
pr299	3	48191	48466	✓	48279	48279	52512	48466
pr439	3	107217	108574	108462	108462	111491	1637118	113351
random541	3	28313	28462	28403	28453	42760	175026	33341
rat575	3	6773	6825	6818	6833	7294	29176	150760
rat783	3	8806	8875	8892	8892	66220	230858	110462
rd400	3	15281	15284	15342	15342	46026	138078	15296
u1060	3	224094	233352	231890	231702	3687485	6590942	558512
u574	3	36905	37258	37316	37229	149246	596984	738862
u724	3	41910	42351	42295	42080	191463	812717	7934762
vm1084	3	239297	245832	245975	244635	4980215	7953251	1206900
a280	4	2579	2600	✓	✓	3974	31730	✓
ali535	4	202310	202375	205555	203239	276517	1935738	543756
att532	4	27686	29431	27733	27706	33702	116405	36558
d493	4	35002	35107	35109	35282	40404	377638	72892
d657	4	48912	49337	49306	49204	116847	748880	394155
fl417	4	11861	11882	11870	✓	✓	372191	273241
gil262	4	2378	2383	✓	2380	✓	✓	✓
gr202	4	40160	40187	40187	✓	40187	✓	✓
gr431	4	171414	172391	172138	171753	173949	1769943	261594
gr666	4	294358	297226	295836	298335	902076	3977095	2610904
lin318	4	42029	42315	✓	✓	42155	126465	42091
p654	4	34643	34844	34743	35631	77259	1801848	1531326
pcb442	4	50778	51286	51293	51380	97456	579322	577682
pr1002	4	259045	265631	263935	267590	3613910	6404767	5685162
pr264	4	49135	✓	49955	49378	✓	✓	✓
pr299	4	48191	✓	✓	✓	48597	50901	48466
pr439	4	107217	109590	107397	109388	115053	1258009	1275865
random541	4	28313	28409	28393	28403	42564	217823	33776
rat575	4	6773	6804	6793	6799	6793	27172	147564
rat783	4	8806	8882	8861	8920	51889	170897	110707
rd400	4	15281	15322	15355	15294	46128	161448	16284
u1060	4	224094	231515	229917	230158	3681894	6481845	517258
u574	4	36905	37316	36947	37321	147788	651516	49955
u724	4	41910	42394	43695	42097	165548	835919	8263644
vm1084	4	239297	251371	245975	248103	4980215	8432551	1231641

Table A.4: Results of the matheuristics. Each computation is meant to hit the time limit of 1020000 ticks. For each execution the best solution found is reported, a '✓' means that the exact solution was found.

A.4 Heuristic Algorithms

	NN	**NN**	**NN**	**NN**	**NN**	**NN**	**NN**	**NN**	**EM**	**EM**
		G	**O**	**G|O**	**R**	**R|G**	**R|O**	**R|G|O**		**O**
a280	2975	4824	2687	2803	3229	5383	2723	2910	2925	2481
ali535	240828	384969	219888	223286	250303	418658	216860	221696	249791	213842
att532	33387	51163	29116	30520	33734	54744	29328	29506	31727	27822
d1291	58680	114393	53140	57421	62095	118043	52806	56941	58658	49801
d1655	73357	127362	65335	70290	76184	139479	65533	70888	70859	62224
d2103	86501	170858	81971	92791	89845	180809	82789	94122	90749	81317
d493	40186	61474	37016	37249	46917	65593	36446	37900	37891	34144
d657	60175	93802	51042	54425	63098	101982	52110	54319	55819	49702
fl1400	25115	38489	21164	21772	26523	42577	20941	21171	22414	18970
fl1577	25534	52037	22904	23943	27981	58926	23005	24773	25648	22486
fl417	13887	23185	12366	12818	14507	25168	12433	12454	12903	10389
gil262	2823	4324	2558	2564	3112	4780	2541	2557	2757	2385
gr202	47060	65901	42365	43935	50016	73373	42954	44160	45027	37119
gr229	157394	236281	138713	139889	162760	260858	139288	147732	151804	129046
gr431	207189	304003	178489	188459	224693	323118	177792	184902	197457	166131
gr666	350243	527625	307263	323434	363543	567761	305182	327761	344747	310679
kroA200	34543	53778	30189	31463	40760	64437	30565	30626	36688	31479
kroB200	35389	54057	31796	31688	36501	59085	31934	31566	36547	31621
lin318	49201	78200	44109	46002	51811	86211	44292	46284	53015	43799
nrw1379	68531	102924	59642	63162	70717	105017	59871	62443	65170	59407
p654	43027	65869	35926	35949	43210	83288	35880	36054	36641	32002
pcb1173	70115	111999	60885	64252	72332	125461	60918	64494	68930	59509
pcb442	58950	95045	52873	55808	62174	101715	53698	56442	60241	50706
pr1002	313745	500761	273632	282938	336561	555388	268831	283536	299282	271242
pr226	92552	166545	83277	87146	100708	205880	82900	88056	92089	68368
pr264	54491	90255	52084	54341	57362	116349	52002	53500	53799	45577
pr299	58279	94403	50126	53393	61940	114203	50647	53127	58260	49187
pr439	127230	211244	112493	116736	133389	237132	114963	117503	122924	106329
random1000	199749	304763	169260	177850	215887	327290	169173	178545	105991	94426
random1007	105447	171335	94442	97530	108762	189100	95192	99118	265599	239837
random1046	276205	445047	245303	259610	285284	494219	245371	259691	148976	134007
random1135	77264	121767	67991	71872	81639	130788	67699	72245	150709	134647
random1158	148530	231997	129512	137678	151418	262955	129512	137617	348963	316404
random1198	44722	72397	39556	41893	46656	78524	39280	41216	46178	41666
random1201	290134	463834	252691	267386	306818	478706	253410	264018	247463	225641
random1260	135653	214557	119469	123685	140746	224968	119364	124596	38728	34523
random1269	106577	172170	94049	98625	112025	178896	93343	99513	215995	192658
random1320	146134	232804	128171	135785	151542	241880	128305	135795	29578	26157
random1335	255702	409474	223154	234926	260961	448891	225988	237503	108320	99164
random1349	214296	339472	185769	197941	223680	353276	185777	198501	90188	82117
random1381	38933	60833	33626	35763	40417	64983	33865	35485	161010	144909
random1440	285731	453087	249356	262557	294722	473731	252127	264913	329556	297345
random1527	189327	303326	164599	174103	196888	336410	165051	174477	141330	126415
random1538	208380	334847	183485	192572	213611	379121	182163	193479	194801	176090
random1592	150999	242605	131475	137705	156224	247997	131258	137775	219270	198154
random1606	190630	303996	168528	175330	199070	329294	168668	178046	86387	78048
random1640	126461	202881	110461	116529	130593	214458	111542	116289	152761	138472
random1643	287376	462346	251766	266161	303094	475459	252199	266928	121184	105396
random1697	157949	257949	139730	146982	162503	279340	140240	146900	209815	183295
random1705	105499	167558	91673	95716	108532	172188	92347	95868	199314	174243
random1711	85373	137043	74770	78488	87886	139879	74911	78939	274874	248142
random1723	243562	393288	213953	225780	250532	432434	215127	224184	283274	251398
random1728	350129	549335	305792	321425	359770	581359	302872	318686	208344	185864
random1773	330133	536103	289086	303783	339358	535548	289011	305215	193567	171496

Continued on next page

random1852	273220	442497	239075	253117	283695	464194	239009	250205	257962	231551
random1892	216200	345455	186136	198919	229298	354043	186757	198871	183866	162518
random1930	135830	216838	118173	125869	139551	227944	118258	125130	170982	155095
random1942	104765	166166	90962	97679	110052	182424	91047	95486	204256	179544
random2037	182511	291122	158098	167096	186599	310320	159585	169580	182544	164203
random2056	106763	170610	93423	98370	111275	180252	93481	98467	138753	125095
random2058	169666	272698	147081	155403	178320	294984	148529	156610	155637	137416
random2097	152671	244646	131552	138374	159529	257764	131376	139972	372718	339725
random2104	194667	312809	170826	181253	200711	329532	170112	179377	149520	132813
random2126	197435	319090	172209	181679	207068	329528	172734	182606	202345	181863
random2131	63553	99533	55027	57252	68160	104048	54800	57608	373543	338657
random2143	267596	428439	234942	245185	276614	449670	234895	245961	127322	114571
random2151	89809	143499	78160	82774	91718	153184	77951	82462	190697	170568
random2160	216518	345745	190445	197779	223000	359797	189785	198130	192519	175049
random2189	140012	225900	122587	129254	143245	234975	121868	129054	137201	123318
random2223	367899	595725	323023	342760	381279	634743	323186	341090	130363	119157
random2236	350682	556811	302677	317947	361486	587474	300793	319465	62655	56803
random2238	365238	579525	319183	337433	378599	626897	319685	334664	186408	164334
random2279	130792	209395	113965	118433	136255	223473	113915	119770	286793	259489
random522	183645	295032	161787	173766	195067	315207	163611	172250	104533	94175
random541	33699	53462	29952	30895	35933	58910	30213	31065	182988	163256
random722	188656	300409	171913	182275	200729	337720	172149	177787	109971	99059
random730	197171	312276	172059	181834	214192	344459	172613	181762	104151	92533
random743	215637	339128	188984	194448	223269	359019	189885	196815	293540	260514
random786	114154	182651	102996	107715	118123	197031	102300	106582	289539	263600
random820	158571	249298	137935	148140	165529	275474	137634	143837	218056	196300
random831	202289	330170	180740	188776	206172	376491	180282	189008	216220	193908
random834	148713	236337	130627	135070	157380	261264	128328	135070	76795	69431
random857	190998	297159	165873	172346	207306	319721	163870	174273	34463	29729
random873	179135	284487	159009	166603	185792	313604	158174	165909	148853	131754
random949	29486	47042	25883	27091	31658	50337	25809	26815	353893	316891
rat575	7993	12109	7077	7577	8424	13296	7127	7622	7842	6882
rat783	10540	16402	9246	9862	11212	17665	9287	9868	10324	9095
rd400	18431	27294	15898	16717	19718	31140	15892	16436	18515	15946
rl1304	306195	602774	270610	276332	323008	644130	269050	281510	307204	260373
rl1323	312845	629314	280799	301463	333932	679762	285128	303371	326022	282863
rl1889	374845	733686	334342	346973	384987	810843	335183	359564	397657	338893
ts225	140486	255202	127961	139007	152862	310956	128227	128002	155946	114090
tsp225	4503	6851	4001	4174	4831	8021	3989	4290	4367	3625
u1060	277095	437808	239502	242522	301605	491112	237621	244580	261154	231400
u1432	186094	271458	163286	170529	193510	287171	162175	172331	175331	161241
u1817	66187	120892	59964	65006	72012	126529	60957	65537	66043	59036
u574	45439	72466	38959	40177	47841	76549	38899	40687	43518	37870
u724	50801	78893	43651	46382	53508	83089	44461	47020	50240	43687
vm1084	290802	490435	257291	262193	302150	583679	251754	261809	276340	247664
vm1748	404406	693568	352357	368853	424432	742079	355138	365256	399217	355420

Table A.5: Results of the heuristics. Each computation is meant to hit the time limit of 3600 seconds. For each execution the best solution found is reported. For clarity the legend for the Nearest Neighbors heuristic is shown. If there is the character G it means that GRASP has been applied with $k = 3$, if there is the character O it means that in that execution the refinement heuristic 2-OPT has been applied, if there is the character R it means that as initialization policy the random one has been chosen instead of multistart.

A.5 Metaheuristic Algorithms

	TABU	TABU G	GEN B|T	GEN B|RW	GEN B|RK	GEN B|RA	GEN T	GEN RW	GEN RK	GEN RA
ali535	211204	224131	211501	216704	215923	212389	214519	214838	216128	215402
att532	28703	29175	29175	29561	29416	29281	29669	29676	29382	29760
d1291	51656	53597	54604	54477	56197	54983	54348	55404	53802	54572
d1655	64272	65281	67549	67001	68317	68193	66491	67317	67693	67369
d2103	81519	86493	86537	87535	91445	88308	85781	87143	89187	84717
d493	36416	36683	37339	37196	37065	37096	36911	36862	37123	37478
d657	50935	52138	52979	53035	52872	52216	52699	52472	52615	52732
fl1400	21178	21389	20689	20584	20753	20732	20572	20671	20821	20750
fl1577	22662	23979	23778	24211	23848	23702	23332	23542	23448	23853
fl3795	29210	31013	30359	31048	36903	34487	30951	30804	32143	30973
fl417	12196	12656	11985	12041	12050	12093	11995	12068	11964	12036
fnl4461	191506	202211	206656	225425	226070	227163	216287	227251	226170	203375
gr431	176446	183342	182955	183278	183508	183314	180922	185010	183725	181784
gr666	313535	323022	319265	318071	319519	319554	323078	324687	319855	320911
nrw1379	59101	60056	62336	62274	62347	62540	61952	62014	62078	62148
p654	35383	37249	35039	35338	35411	35215	35154	35335	35539	35305
pcb1173	59717	62134	61608	62025	62771	62067	61878	61145	61385	61875
pcb3038	144001	149070	152487	153151	172884	169221	151797	153140	152900	152575
pcb442	51898	52524	54053	53750	54380	54368	53473	54027	54611	54288
pr1002	272865	279121	277365	280336	281122	278675	281225	282315	278465	277054
pr2392	395578	411377	413831	429491	423248	421229	421152	422564	415305	421156
pr439	111246	111197	111503	111980	111442	112207	110238	113034	111932	111562
random1013	71344	73837	416609	428693	431092	420870	413921	412826	428871	413508
random1015	143576	149401	52028	51656	52189	52333	51971	51794	51841	51873
random1099	179410	185737	393270	400525	404114	402906	391063	393369	409140	391824
random1164	48052	49252	359245	362399	362454	355970	359263	358254	360456	359234
random1228	67502	69281	521231	537681	552304	532580	523587	522703	536759	520702
random1236	222419	232850	85554	85108	85558	85673	82589	83311	82573	83602
random1256	243788	253148	49555	49509	49717	49596	49381	48641	49200	49351
random1330	224336	234319	86125	89032	89045	89539	85839	86214	88372	85888
random1428	197169	200254	54312	54499	56488	54190	54200	54219	54701	54081
random1609	296591	305708	79542	79652	82703	80999	79748	80126	83334	79344
random1626	211759	220479	203241	203534	203580	204007	203052	202167	202332	204120
random1632	79568	83119	149647	149237	149212	148455	149855	149482	148818	149169
random1658	45689	47282	156945	158432	158831	158723	156380	157803	157173	157126
random1858	184134	191795	154256	154566	155769	153888	153477	154690	153555	154932
random1878	98068	101957	102085	101933	102702	101994	102481	102025	102047	102459
random2120	278314	286735	193218	200422	200969	194184	192266	194577	196072	193925
random2243	249766	256198	228473	236179	235041	234343	231896	228869	236686	228817
random2273	343902	351536	522692	543571	546097	539742	526329	542723	550420	530276
random2279	85273	86698	232023	233858	234393	231127	231761	230559	230013	230964
random2356	46081	47551	476817	472068	500309	492877	477065	476508	476954	475674
random2406	330809	344127	111068	111573	111464	111335	111385	111131	111455	111601
random2464	211541	219244	324185	325873	334983	322053	322421	320902	323567	320652
random2676	219054	228818	234494	234528	233707	234685	233085	233489	233704	234194
random2688	299647	308372	299364	308124	310589	309053	300843	310330	299845	301017
random2838	370202	386408	291659	291188	292917	290357	291274	291404	292084	292352
random2859	192728	198304	95402	96407	102618	95672	96626	96786	95363	99293
random2993	91332	95008	296346	290403	299823	293060	287811	287115	298841	287370
random3503	355626	365786	241189	248007	248772	247175	242150	241640	252064	241511
random3554	51783	53794	249874	249076	260874	256314	248260	249502	258519	249492
random3580	415873	434283	231144	231134	240638	230336	231499	232019	228054	231786
random3608	184528	194370	88367	89228	91108	88789	88067	88086	88131	88380
random3803	185252	195130	23321	23246	23529	23389	23430	23462	23640	23369

Continued on next page

random3805	218928	230603	69968	70087	69985	69637	69885	69062	68341	69827
random3851	455976	477823	348361	347079	349774	345870	348592	349582	346709	349226
random3856	370835	391184	311347	313244	314561	313354	311670	313194	310121	310289
random3862	306322	323802	372572	375166	389915	372411	374565	373276	372471	372176
random3984	387330	408983	387431	391820	392761	388969	387334	387385	386764	387755
random400	153044	159463	220354	219419	221419	219265	220597	220427	218774	219652
random4198	462206	484325	48096	47946	47780	48664	47756	48028	47730	48095
random4223	228110	241930	118153	117519	123639	120791	117487	117923	121814	118555
random4261	75129	79793	493065	492790	510240	508628	494669	492717	489836	498966
random4319	336097	356010	201487	200428	203829	205990	202677	201132	201316	201826
random4341	147645	155630	223073	225250	222199	223884	221419	220195	221194	221572
random4380	392984	416098	145212	143009	143082	142957	143631	143084	143035	143438
random4398	81453	86479	354785	364494	369610	354278	356163	356533	353922	355810
random4476	216507	233883	83017	82917	83235	82409	82626	83131	82879	82462
random4489	236477	251207	391138	400053	401345	389784	389431	390951	390485	388005
random4516	111846	119682	47404	47608	47897	47751	47414	47192	47533	47035
random4524	470503	500725	484577	498972	504312	485484	484688	487901	483562	487005
random4541	205742	218388	215356	222815	223601	219514	217027	218327	224367	218146
random4547	310609	330654	187503	187029	188446	188156	188503	186097	186917	187750
random455	23281	23391	73888	74247	73876	74099	74230	74335	73807	74138
random4568	494238	531532	192279	192533	194346	194217	193031	193988	192577	192912
random4571	371725	392521	451091	437014	459840	436113	438181	439873	445298	438560
random4615	498343	542056	330134	338556	343375	341059	330383	330435	343602	328955
random4645	160888	171519	235325	234573	231663	232323	233561	233681	231737	232163
random4723	79280	84590	417271	408493	421006	415592	405888	404676	404240	406981
random4784	275130	294175	250879	253998	252235	251771	253388	252910	250230	252348
random4848	285181	309607	159788	154812	161158	154945	156080	155668	160430	155611
random659	153003	162507	169661	172492	176107	174521	168174	169890	168762	172377
random758	140008	145421	261364	259806	264234	258380	260291	260770	260664	259838
random806	107224	110828	194191	201377	202857	202771	196185	195275	200551	200211
random935	50466	51640	309634	307800	308930	308976	307424	310792	306204	307326
rat575	7072	7174	7421	7412	7387	7347	7443	7399	7393	7421
rat783	9225	9506	9737	9649	9672	9616	9619	9720	9627	9749
rd400	15833	16449	16045	16121	16217	16144	16033	16058	16073	16137
rl1304	271487	270832	276398	272266	276961	264051	272004	268953	275103	273868
rl1323	282353	290724	292050	291387	293097	294321	290846	287274	290779	292253
rl1889	335696	347650	337901	344088	348095	342676	344021	339950	340019	342518
u1060	234844	241688	247386	242479	242249	242693	247664	242690	242689	245131
u1432	157205	158893	167686	168271	169064	169761	167713	169920	167591	168396
u1817	59828	60963	63224	63748	64465	64124	63788	63678	63632	63738
u2152	66566	69136	71506	71614	73894	73049	72175	71370	71670	71786
u2319	236897	238515	250663	249236	248547	250621	250055	251256	249854	251199
u574	38882	39155	39658	39317	39632	39598	39631	39611	39407	39611
u724	44019	44907	45924	45169	45749	45382	45695	45537	45208	45299
vm1084	251326	259201	260495	257848	257577	255237	259786	258495	258769	259729
vm1748	352422	366026	370753	369310	365128	367873	368736	367837	366079	366655

Table A.6: Results of the metaheuristics. Each computation is meant to hit the time limit of 3600 seconds. For each execution the best solution found is reported. For clarity, in Tabu Search if there is the character G it means that GRASP has been applied with $k = 3$, whereas in genetic algorithm whenever there is the character B it means that used, the battle survivor selection policy was applied, random otherwise. T, RW, RK, RA stands for the parent selection policy used which are respectively Tournament, Roulette Wheel, Rank and Random.

Appendix B

Project Structure

A possible way to structure your project folder is the following:

```
project-name
 ├─ build
 ├─ data
 │   ├─ instances
 │   └─ ...
 ├─ output
 │   ├─ logs
 │   ├─ plots
 │   └─ ...
 ├─ include
 │   ├─ .h files
 │   └─ ...
 ├─ src
 │   ├─ .c files
 │   └─ ...
 ├─ scripts
 │   ├─ .py files
 │   ├─ .sh files
 │   ├─ .job files
 │   └─ ...
 ├─ CMakeLists.txt
 └─ README.md
```

Figure B.1: Minimal C/C++ project structure

Appendix C

CMake

C.1 Introduction

CMake is a **cross-platform, open-source build system** which offers a family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of your choice. Originally, CMake was designed as a generator for various dialects of Makefile, today CMake generates modern buildsystems such as Ninja as well as project files for IDEs such as CLion, Visual Studio, XCode, Sublime and many others.

Nowadays, CMake is widely used for the C and C++ languages, but it may be used to build source code of other languages too.

C.2 Installation

C.2.1 Linux

There are several ways to install CMake, depending on your platform. If you're under Linux, you may probably want to install the latest version of CMake instead of the one installed by default, to do so you can follow multiple paths: by following these simple steps:

- Uninstall the default version provided by Ubuntu's package manager:

```
$ sudo apt remove --purge cmake
```

- Go to the official CMake download page, then download and extract the latest version. Or if you prefer a command line approach, update the version and build variables in the following command to get the desired version, you can check the correct filename inside the official CMake files page:

```
version = 3.X
build= Y          # could be alphanumeric (e.g. 3, 3-rc4, 3-rc5)
mkdir ~/temp
cd ~/temp
wget https://cmake.org/files/v$version/cmake-$version.$build.tar.gz
tar -xzvf cmake-$version.$build.tar.gz
cd cmake-$version.$build/
```

- Install the extracted source by running:

```
./bootstrap
make -j$(nproc)
sudo make install
```

- Test your new cmake version.

```
cmake --version
```

C.2.2 Windows

In order to install CMake on Windows, you can find pre-compiled binaries here as MSI packages and ZIP files. The Windows installer has an option to modify the system PATH environment variable.

Otherwise you can download and build CMake from source. In order to build CMake from a source tree on Windows, you must first install the latest binary version of CMake because it is used for building the source tree. Once the binary is installed, run it on CMake as you would any other project. Typically this means selecting CMake as the Source directory and then selecting a binary directory for the resulting executables.

C.2.3 macOS

In order to install CMake on macOS, you can find pre-compiled binaries here as disk images and tarballs. After copying CMake.app into /Applications (or a custom location), run it and follow the "How to Install For Command Line Use" menu item for instructions to make the command-line tools (e.g. cmake) available in the PATH.

Otherwise you can download and build CMake from source as in the section C.2.1.

C.3 CMake Scripts

CMake is a meta build system that uses scripts to generate build files for a specific environment. Those scripts are called CMakeLists, and the following section will help you throughout the definition of a each part of a complete scripts that will safely compile your project.

C.3.1 Setup The Project

In order to setup your project, CMake needs to know what is the minimum version you intend to support within it. Higher the threshold, fewer the devices you will be able to run your code on. A good practice is to set a minimum intermediate version. After identifying the minimum version, you need to set the project name (note that this will also be the name of the executable that will be generated at the end of the build). Once done, a minimal CMakeLists should look like this:

```
# Set the minimum version of CMake that can be used. To find the cmake version run
# $ cmake --version
cmake_minimum_required(VERSION 3.X)

# Setting the project name
project(project_name)
```

C.3.2 Adding Executables

Until this moment, the project has a name and a minimal version, the next step to do is to add executables, to do this you can use the following functions:

```
# Create a sources variable with a link to all .c/.cpp files to compile
set(SOURCE_FILES src/main.c ...)

# Add an executable with the above sources
add_executable(${PROJECT_NAME} ${SOURCE_FILES})
```

Within the `set(...)` function you can define variables to which you can assign values, through the function `add_executable(...)` instead you can load the executables in which we are interested.

It is useful to notice that inside the variable `PROJECT_NAME` is contained the value set through the function `project(...)` to the value of `project_name`.

C.3.3 Adding Directories

In case you want to add directories to your project, which perhaps contain other libraries or executables, just use `target_include_directories(...)` or `include_directories(...)`.

These functions do the same job, the only difference is that `include_directories(x/y/z)` affects directory scope. All targets in this CMakeLists, as well as those in all subdirectories added after the point of its call, will have the path `x/y/z` added to their include path.

Instead `target_include_directories(target PRIVATE|PUBLIC|INTERFACE x/y/z)` has target scope and adds `x/y/z` to the include path for the specific target. Here are reported, the two version explained above:

```
# Add a directory to the target (choose among PRIVATE, PUBLIC or INTERFACE)
target_include_directories(${PROJECT_NAME} PRIVATE|PUBLIC|INTERFACE <path>)

# Add a directory
include_directories(<path>)
```

C.3.4 Linking Libraries

In addition to including directories to the project, you can also link libraries in order to extend the functionality of the project using already implemented and optimized functions. To do this, it is useful to use the `target_link_libraries(...)` function. Its use is shown below:

```
# Add a specific library through it's real name
target_link_libraries(${PROJECT_NAME} -l<path>)

# Add a specific directory to search for (binary) libraries
target_link_libraries(${PROJECT_NAME} -L<path>)
```

C.4 Concorde Library

Whenever it is necessary to deal with fractional solutions Bill Cook devised an open-source library called Concorde that can come in handy since it implements a very efficient Branch&Cut algorithm for the TSP. Specifically, Concorde is distributed both as an executable program and as a static library in which are present many optimized functions for the resolution of the TSP. For our purposes, this guide will come in handy to install as a static library. Once installed, it can be added and linked within a makefile as an external library via the commands explained in C.3.3 and C.3.4.

C.5 A Full Fledged Example

A possible basic configuration for the CMakeLists file can be obtained as follows:

```
# Set the minimum version of CMake that can be used. To find the cmake version run
# $ cmake --version
cmake_minimum_required(VERSION 2.8)

# Setting the project name
project(project_name)

# Create a sources variable with a link to all cpp files to compile
set(SOURCE_FILES
        src/main.c
        src/foo.c
        src/...)

# Add one or more executable with the above sources variable
add_executable(${PROJECT_NAME} ${SOURCE_FILES})

# Check which operating system is detected by CMake
message(STATUS "You're using ${CMAKE_SYSTEM_NAME}")

if (WIN32)

  # put here some stuff related to WIN32

elseif (UNIX AND NOT APPLE)

  # To link libraries and directories stored inside your distro
  set(CPLEX_HOME /opt/ibm/ILOG/CPLEX_Studio201/cplex)
  set(CONCORDELIB /opt/concorde)

  # To link and include useful libraries and directories
  target_link_libraries(${PROJECT_NAME} -lcplex -lm -lpthread -ldl)
  target_link_libraries(${PROJECT_NAME} -L${CPLEX_HOME}/lib/x86-64_linux/static_pic)
  target_link_libraries(${PROJECT_NAME} ${CONCORDELIB}/concorde.a)
  target_include_directories(${PROJECT_NAME} PRIVATE ${CPLEX_HOME}/include/ilcplex)
  target_include_directories(${PROJECT_NAME} PRIVATE ${CONCORDELIB})

else (APPLE)

  # put here some stuff related to APPLE (MAC OS)

endif ()
```

Bibliography

[1] IBM ILOG Cplex. "V12. 1: User's Manual for CPLEX". In: *International Business Machines Corporation* 46.53 (2009), p. 157.

[2] G. Dantzig, R. Fulkerson, and S. Johnson. "Solution of a Large-Scale Traveling-Salesman Problem". In: *Journal of the Operations Research Society of America* 2.4 (1954), pp. 393–410. DOI: `10.1287/opre.2.4.393`. eprint: `https://doi.org/10.1287/opre.2.4.393`. URL: `https://doi.org/10.1287/opre.2.4.393`.

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

[4] Matteo Fischetti and Andrea Lodi. "Local branching". In: *Mathematical Programming* 98 (Sept. 2003), pp. 23–47. DOI: `10.1007/s10107-003-0395-5`.

[5] L. De Giovanni and M. Di Summa. *Exact methods for the Traveling Salesman Problem*. URL: `https://www.math.unipd.it/~luigi/courses/metmodoc/m08.01.TSPexact.en.pdf`. (accessed: 01.10.2021).

[6] A.J. Orman and H.P. Williams. "A Survey of Different Integer Programming Formulations of the Travelling Salesman Problem". In: *Optimisation, Econometric and Financial Analysis*. Ed. by Erricos John Kontoghiorghes and Cristian Gatu. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 91–104.

[7] J. Potvin. "Genetic algorithms for the traveling salesman problem". In: *Annals of Operations Research* 63 (1996), pp. 337–370.

[8] Noraini Mohd Razali, John Geraghty, et al. "Genetic algorithm performance with different selection strategies in solving TSP". In: *Proceedings of the world congress on engineering*. Vol. 2. 1. International Association of Engineers Hong Kong. 2011, pp. 1–6.

[9] Colin R. Reeves, ed. *Modern Heuristic Techniques for Combinatorial Problems*. USA: John Wiley & Sons, Inc., 1993. ISBN: 0470220791.

[10] Gerhard Reinelt. "TSPLIB—A Traveling Salesman Problem Library". In: *ORSA Journal on Computing* 3.4 (1991), pp. 376–384. DOI: `10.1287/ijoc.3.4.376`. eprint: `https://doi.org/10.1287/ijoc.3.4.376`. URL: `https://doi.org/10.1287/ijoc.3.4.376`.

[11] J. B. Robinson. *On the Hamiltonian game (a traveling-salesman problem)*. Santa Monica, CA: RAND Corporation, 1949.

[12] Wikipedia contributors. *Travelling salesman problem — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Travelling_salesman_problem&oldid=1014002267`. [Online; accessed 26-March-2021]. 2021.

[13] William Cook. *Concorde*. Version 03.12.19. Dec. 19, 2003. URL: `https://www.math.uwaterloo.ca/tsp/concorde/index.html`.

[14] Thomas Williams, Colin Kelley, and many others. *Gnuplot 4.6: an interactive plotting program.* http://gnuplot.sourceforge.net/. 2013.