UNIVERSITY OF PADUA

School of Engineering

Master Degree in Computer Engineering

Course Name: Operations Research 2

Academic Year 2018-2019

Operations Research 2 Project

Students:	*Student ID:*
Samuele Papa	1189911
Davide Storato	1153692

Samuele Papa

Davide Storato

July 17, 2019

Contents

List of Figures

List of Tables

List of Algorithms

Abbreviations

ATSP	Asymmetric Travelling Salesman Problem
F1	Single Commodity Flow model
F2	Two Commodity Flow model
F3	Multi-Commodity Flow model
GRASP	Greedy Randomized Adaptive Search Procedure
ILP	Integer Linear Programming
MIP	Mixed Integer Programming
MP	Mathematical Programming
MTZ	Miller-Tucker-Zemlin model
RCL	Restricted Candidate List
SA	Simulated Annealing

SEC	Subtour Elimination Constraints
SP	Separation Problem
T1	1st Timed Stage Dependent model
T2	2nd Timed Stage Dependent model
T3	3rd Timed Stage Dependent model
TSP	Travelling Salesman Problem
VNS	Variable Neighborhood Search

1 Introduction

The project detailed in this report focuses on the different approaches to solving the Travelling Salesman Problem (TSP) as a way to understand more deeply the various issues that arise when approaching a MIP problem and how to address them. This was developed during the Operations Research 2 (Ricerca Operativa 2) course by Matteo Fischetti. In this project the TSP, widely known and studied in the community, is first solved by modifying its formulation, then by using techniques that add constraints iteratively, and finally by adopting several different heuristical methods. The MIP solver used in this project is the Gurobi Optimizer (v. 8.1) [20] while JetBrains' IDE CLion is used for the C programming. The instances of the TSP are from the TSPLIB library [31].

1.1 History of the Travelling Salesman Problem

The origin of the TSP is not entirely clear. The first mathematical formulation of this problem was in the 1800s by the Irish mathematician William Rowan Hamilton and British mathematician Thomas Kirkman.
The general form of the TSP appears to have been studied in the 1920s by Karl Menger as "the messenger's problem" in Vienna and by mathematicians at Harvard. In the 1950s and 1960s, the problem became very popular in the scientific circles after the RAND Corporation in Santa Monica offered prizes for steps in solving the problem. Remarkable contributions were made by George Dantzig, Delbert Ray Fulkerson and Selmer M. Johnson, who formulated the problem as an integer linear-programming problem and developed the cutting planes method for its solution.
In 1972 Richard Manning Karp showed that the Hamiltonian cycle problem was NP-complete, which implies the NP-hardness of the TSP. This supplied a mathematical explanation for the apparent computational difficulty of finding optimal tours.
In the following years, many approaches were devised, for the symmetrical and the asymmetrical variants of the problem. The size of the instances solved is ever increasing. Concorde is known for being the "fastest TSP solver, for large instances, currently in existence" [1] (Concorde was used to solve an instance with 85900 cities).

1.2 First formulation of the Travelling Salesman Problem

The TSP consists in finding a Hamiltonian circuit of minimum cost in a complete directed graph $G = (V, A)$. A Hamiltonian circuit is a cycle that visits each vertex once and only once.
The best known Integer Linear Programming (ILP) formulation of the TSP is by Dantzig, Fulkerson and Johnson [11], defined in 1954. In this project we mostly use the version of this problem on an undirected graph $G = (V, E)$. Follows the DTZ formulation adapted to the undirected case.

Given the decision variables:

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

the problem is formulated as follows:

$$\min \quad \sum_{e \in E} c_e x_e \tag{1.1}$$

$$\sum_{e \in \delta(v)} x_e = 2 \quad \forall v \in V \tag{1.2}$$

$$\sum_{e \in E(S)} x_e \leq |S| - 1 \quad \forall S \subset V, \quad |S| \geq 2 \tag{1.3}$$

$$0 \leq x_e \leq 1 \; integer, \quad \forall e \in E. \tag{1.4}$$

Where (1.1) sums the costs of all the vertices in the path, often the cost is computed by using some measure of distance between the two vertices which define the edge. Constraints (1.2) force the degree of each vertex to be 2, setting up the cycle. Constraints (1.3) are called Subtour Elimination Constraints (SEC) and make sure that the model does not create subtours by allowing only one cycle, the one formed by all the vertices.
This formulation has $O(2^n)$ constraints introduced by (1.3) and $\frac{n(n-1)}{2}$ variables. Because of the exponential number of constraints it is difficult to solve with all the constraints added in the model before the optimization begins. To address this issue, when using branch-and-cut techniques it is possible to remove the SEC and later add only those that are violated by the current solution by solving a Separation Problem (SP). A SP is generally defined as finding those sets which violate some constraint, consequently, solving this problem means being able to then define constraints only on those specific sets.

2 Compact models

This section introduces the compact models studied and implemented during the course. Some of these models have formulations that are defined for the Asymmetric Travelling Salesman Problem (ATSP). To address this issue the undirected edges of the original graph are transformed into directed edges with weights equal to the corresponding undirected edges.

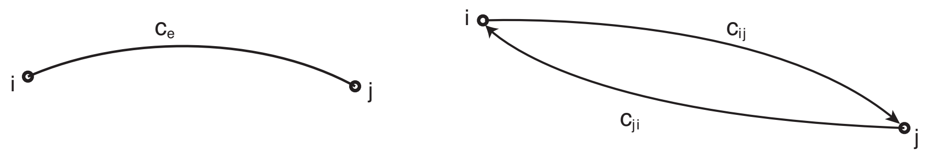


Figure 2.1: Conversion from undirected to directed graph. On the left, undirected edge e from vertex i to vertex j and weight c_e. On the right, conversion to directed graph, weights $c_{ij} = c_{ji} = c_e$.

2.1 Miller-Tucker-Zemlin

This model was designed by Clair E. Miller, Albert W. Tucker and Richard A. Zemlin [26] in 1960. The model is based on the Dantzig, Fulkerson and Johnson model, explained in section 1.2, but uses another strategy to implement the SEC (1.3). This approach adds to the Dantzig model the continuous variables u_i, $\forall i \in V$. These variables indicate the position of node i in the optimal tour. The model is defined as follows.

Given the variables

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

the problem is modeled as follows:

$$\min \quad \sum_i \sum_j c_{ij} x_{ij} \tag{2.1}$$

$$\sum_i x_{ih} = 1 \quad \forall h \in V \tag{2.2}$$

$$\sum_j x_{hj} = 1 \quad \forall h \in V \tag{2.3}$$

$$x_{ij} + x_{ji} \leq 1 \quad \forall i < j \tag{2.4}$$

$$u_j \geq u_i + 1 - M(1 - x_{ij}) \quad \forall i, j : j \neq 1 \tag{2.5}$$

$$u_1 = 1 \tag{2.6}$$

$$1 \leq u_i \leq n \text{ integer} \tag{2.7}$$

$$x_{ii} = 0 \tag{2.8}$$

$$0 \leq x_{ij} \leq 1 \text{ integer}, \quad \forall i \neq j \in V. \tag{2.9}$$

The constraints (2.4) remove the subtour created by two nodes. The constraints (2.5) force the model to create only one tour, the final one. This model uses the Big-M method. This method allows us to disable a constraint under certain conditions. When the arc x_{ij} is not selected the Big-M disables the constraint, otherwise it is enabled. The best value of M is $n-1$. Larger values can be used but are automatically reduced by the Mixed Integer Programming (MIP) solver with internal methods based on variable bounds. Without this automatic method a very large value of M is used. Large values of M lead to an increase on the resolution time of the MIP solver.

2.2 Exercise compact

Given the variables

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

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

the problem is modeled as follows:

$$\min \quad \sum_i \sum_j c_{ij} x_{ij}$$

$$\sum_i x_{ih} = 1 \quad \forall h \in V \tag{2.10}$$

$$\sum_j x_{hj} = 1 \quad \forall h \in V \tag{2.11}$$

$$\sum_{h=1}^{n} z_{vh} = 1 \quad \forall v \in V \tag{2.12}$$

$$\sum_{v \in V} z_{vh} = 1 \quad \forall h \in \{1, \dots, n\} \tag{2.13}$$

$$\sum_{t=1}^{h} z_{it} + x_{ij} + \sum_{t=h+2}^{n} z_{jt} \le 2 \quad \forall i \neq j \in V, \forall h \in \{2, \dots, n-2\} \tag{2.14}$$

$$\sum_{t=3}^{n-1} z_{it} + x_{i1} \le 1 \quad \forall i \in V \setminus \{1\} \tag{2.15}$$

$$z_{11} = 1 \tag{2.16}$$

$$0 \le z_{vh} \le 1 \text{ integer}, \quad \forall v \in V, 1 \le h \le n \tag{2.17}$$

$$x_{ii} = 0$$

$$0 \le x_{ij} \le 1 \text{ integer}, \quad \forall i \neq j \in V.$$

The model was proposed during the course as a further example on how to define a formulation capable of correctly describing the TSP and keeping the number of constraints polynomial in the number of variables. It was created by professor Matteo Fischetti taking inspiration from Miller-Tucker-Zemlin model (MTZ) but using binary variables to describe whether or not a vertex is in a specific position in the tour. Thanks to these variables it is then possible to define constraints to obtain the wanted behaviour. What follows is a description of the idea behind the constraints used.

Constraints (2.12) and (2.13) guarantee that a vertex is in one and only one position and that all positions are selected once and only once.
Constraint (2.15) guarantees that if the arc $(i, 1)$ is selected, then the position of vertex i can only be 2 or n.
Constraint (2.14) is what allows the creation of the tour. The desiderata is: let u and p be two adjacent vertices in the optimal tour and let t_u and t_p be their positions respectively, then, if u comes before p, $t_p = t_u + 1$. This is made possible by removing all scenarios where this does not happen: for all arcs that are selected, given a position h, if the vertex with lower index is in a position between 1 and h, then the vertex with higher index cannot be in a position with index between $h + 2$ and n and vice versa.
Forcing the position of node 1 to be 1, the tour can be defined as expected.

2.3 Single Commodity Flow

The model was designed by Bezalel Gavish and Stephen C. Graves [18] in 1978. This model is based on the MTZ model. Gavish and Graves modified the MTZ model as a flow model using the additional variables to sum the flow in the arcs of the graph. The model is defined as follows.

Given the variables

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

and the continuous variables

$$y_{ij} = \text{`Flow' in an arc } (i, j)\, i \neq j.$$

the problem is modeled as follows:

$$\min \quad \sum_i \sum_j c_{ij} x_{ij}$$

$$\sum_i x_{ih} = 1 \quad \forall h \in V$$

$$\sum_j x_{hj} = 1 \quad \forall h \in V$$

$$y_{ij} \leq (n-1)x_{ij} \quad \forall i, j \in N, i \neq j \tag{2.18}$$

$$\sum_{\substack{j \\ j \neq 1}} y_{1j} = n-1 \tag{2.19}$$

$$\sum_{\substack{i \\ i \neq j}} y_{ij} - \sum_{\substack{k \\ j \neq k}} y_{jk} = 1 \quad \forall j \in N \setminus \{1\} \tag{2.20}$$

$$x_{ii} = 0$$

$$0 \leq x_{ij} \leq 1 \text{ integer}, \quad \forall i \neq j \in V.$$

The Single Commodity Flow model (F1) changes from the MTZ model in the SEC part. Where the constraints (2.18) allow the flow through the selected arc. The constraints (2.19) limit to $n-1$ the flow into node 1. The constraints (2.20) bound to 1 the flow out of the other nodes. After each node the flow needs to go down by 1, starting from $n-1$, given all the constraints, this makes the only feasible solution the solution to the TSP.

2.4 Two Commodity Flow

The model was designed by Gerd Finke, Armin Claus and Eldon Gunn [13] in 1984. This model is a ATSP modelling that takes the form of a two-commodity network flow problem. The model is defined as follows.

Given the variables

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

and the continuous variables

$$y_{ij} = \text{‘Flow’ of commodity 1 in arc } (i,j)\, i \neq j.$$
$$z_{ij} = \text{‘Flow’ of commodity 2 in arc } (i,j)\, i \neq j.$$

the problem is modeled as follows:

$$\min \quad \sum_i \sum_j c_{ij} x_{ij}$$

$$\sum_i x_{ih} = 1 \quad \forall h \in V$$

$$\sum_j x_{hj} = 1 \quad \forall h \in V$$

$$\sum_{\substack{j \\ j \neq 1}} (y_{1j} - y_{j1}) = n - 1 \tag{2.21}$$

$$\sum_j (y_{ij} - y_{ji}) = -1 \quad \forall i \in N \setminus \{1\},\, i \neq j \tag{2.22}$$

$$\sum_{\substack{j \\ j \neq 1}} (z_{1j} - z_{j1}) = -(n - 1) \tag{2.23}$$

$$\sum_j (z_{ij} - z_{ji}) = 1 \quad \forall i \in N \setminus \{1\},\, i \neq j \tag{2.24}$$

$$\sum_j (y_{ij} + z_{ij}) = n - 1 \quad \forall i \in N \tag{2.25}$$

$$y_{ij} + z_{ij} = (n-1)x_{ij} \quad \forall i,\ j \in N \tag{2.26}$$

$$x_{ii} = 0$$

$$0 \leq x_{ij} \leq 1 \text{ integer}, \quad \forall i \neq j \in V.$$

Where the constraint (2.21) forces $n - 1$ units of commodity 1 to flow out of node 1. Constraints (2.22) force the flow of commodity 1 to be reduced by 1 unit after each node it traverses.
The constraint (2.23) forces $n-1$ units of commodity 2 to flow in node 1. Constraints (2.24) instead force the flow of commodity 2 to be increased by 1 unit every node it traverses.
The constraints (2.25) and (2.26) control the commodity flow in every single arc. The constraints (2.25) force $n - 1$ unit of commodity 1 and 2 in each arc. The constraints (2.26) allow the commodity 1 and 2 to flow only in the selected tour arcs.

2.5 Multi-Commodity Flow

The model was first developed by Richard T. Wong [40] in 1980. The model was formulated as multi-commodity flow model with the use of additional non-negative variables to describe the flow. In 1984 Armin Claus [7] proposed another formulation of the multi-commodity flow uses a fewer number of commodities than Wong model. The model is defined as follows.

Given the variables

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

and the continuous variables

$$y_{ij}^k = \text{`Flow' of commodity } k \text{ in arc } (i,j),\ \ k \in N \setminus \{1\}.$$

the problem is modeled as follows:

$$\begin{aligned}
\min \quad & \sum_i \sum_j c_{ij} x_{ij} \\
& \sum_i x_{ih} = 1 \quad \forall h \in V \\
& \sum_j x_{hj} = 1 \quad \forall h \in V \\
& y_{ij}^k \le x_{ij} \quad \forall i,j,k \in N,\ \ k \neq 1 && (2.27) \\
& \sum_j y_{1j}^k = 1 \quad \forall k \in N \setminus \{1\} && (2.28) \\
& \sum_i y_{i1}^k = 0 \quad \forall k \in N \setminus \{1\} && (2.29) \\
& \sum_i y_{ik}^k = 1 \quad \forall k \in N \setminus \{1\} && (2.30) \\
& \sum_j y_{kj}^k = 0 \quad \forall k \in N \setminus \{1\} && (2.31) \\
& \sum_i y_{ij}^k - \sum_i y_{ji}^k = 0 \quad \forall j,k \in N \setminus \{1\},\ j \neq k && (2.32) \\
& \quad x_{ii} = 0 \\
& 0 \le x_{ij} \le 1 \text{ integer}, \quad \forall i \neq j \in V.
\end{aligned}$$

Where the constraints (2.27) allow flow only in the selected tour arcs.
The constraints (2.28) force only one unit of each commodity to flow out of node 1. Constraints (2.29) prevent any commodity to flow in at node 1.
The constraints (2.30) force only one unit of commodity k to flow in at node k. Constraints (2.31) prevent any of commodity k to flow out of node k.
The constraints (2.32) force a balance for all commodities in each node, excepted the node 1 and node k. This allows for a decrease in the number of "materials" that are travelling through a node by removing a material each time an arc is selected (the node 1 has $N-1$ "materials" flowing out, while the last one, node k, only has "material" k flowing in and nothing flowing out).

2.6 1st Timed Stage Dependent

This model was designed by Kenneth R. Fox, Bezalel Gavish Stephen C. and Graves [16] in 1980. The model is a ATSP problem in which a set of variables is added to keep track of when the arc (i, j) is chosen. The model is defined as follows.

Given the variables

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

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

the problem is modeled as follows:

$$\begin{aligned}
\min \quad & \sum_i \sum_j c_{ij} x_{ij} \\
& \sum_i x_{ih} = 1 \quad \forall h \in V \\
& \sum_j x_{hj} = 1 \quad \forall h \in V \\
& \sum_{i,j,t} y_{ij}^t = n && (2.33) \\
& \sum_{\substack{j,t \\ t \geq 2}} t y_{ij}^t - \sum_{k,t} t y_{ki}^t = 1 \quad \forall i \in N \setminus \{1\} && (2.34) \\
& x_{ij} - \sum_t y_{ij}^t = 0 \quad \forall i, j \in N,\, i \neq j && (2.35) \\
& x_{ii} = 0 \\
& 0 \leq x_{ij} \leq 1 \text{ integer}, \quad \forall i \neq j \in V.
\end{aligned}$$

Where the constraint (2.33) force the creation of the Hamiltonian circuit.
The constraints (2.34) guarantee that if a node is met at time t it is left at time $t + 1$.
The constraints (2.35) guarantee that if an arc is in the circuit can be selected in a time t once.

2.7 2nd Timed Stage Dependent

This model was designed by Kenneth R. Fox, Bezalel Gavish Stephen C. and Graves [16] in 1980. This model is a disaggregated form of the first timed stage dependent model described in section 2.6. The model is defined as follows.

Given the variables

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

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

the problem is modeled as follows:

$$\min \quad \sum_i \sum_j c_{ij} x_{ij}$$

$$\sum_i x_{ih} = 1 \quad \forall h \in V$$

$$\sum_j x_{hj} = 1 \quad \forall h \in V$$

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

$$\sum_{\substack{i,t \\ i \neq j}} y_{ij}^t = 1 \quad \forall j \in N \tag{2.36}$$

$$\sum_{\substack{j,t \\ j \neq i}} y_{ij}^t = 1 \quad \forall i \in N \tag{2.37}$$

$$\sum_{\substack{i,j \\ j \neq i}} y_{ij}^t = 1 \quad \forall t \in N \tag{2.38}$$

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

$$x_{ii} = 0$$

$$0 \leq x_{ij} \leq 1 \text{ integer}, \quad \forall i \neq j \in V.$$

Where the constraints (2.36) and (2.37) force each node to have only one head end adjacent arc and only one tail end adjacent arc when arc is traversed at stage t respectively. The constraints (2.38) force to have only one arc traversed in each stage.

2.8 3rd Timed Stage Dependent

This model was designed by Steven Vajida [37] in 1961. The model is defined has follows.

Given the variables

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

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

the problem is modeled as follows:

$$\min \quad \sum_i \sum_j c_{ij} x_{ij}$$

$$\sum_i x_{ih} = 1 \quad \forall h \in V$$

$$\sum_j x_{hj} = 1 \quad \forall h \in V$$

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

$$\sum_j y_{1j}^1 = 1 \tag{2.39}$$

$$\sum_i y_{i1}^n = 1 \tag{2.40}$$

$$\sum_j y_{ij}^t - \sum_k y_{ki}^{t-1} = 0 \quad \forall i, t \in N \setminus \{1\} \tag{2.41}$$

$$x_{ii} = 0$$

$$0 \leq x_{ij} \leq 1 \text{ integer}, \quad \forall i \neq j \in V.$$

Where the constraints (2.39) and (2.40) force the node 1 to be left at stage 1 and to be entered in stage n respectively.
The constraints (2.41) force the node entered in stage $t-1$ to be leaved in stage t.

3 Exact methods

This section describes the algorithms that solve to optimality a MIP problem using the MIP solver in different setups. One approach iteratively adds constraints and then solves the problem until the correct optimal solution is found, while the other approaches make use of callbacks within the solver. A callback is a user function that is called periodically by the MIP optimizer in order to allow the user to query or modify the state of the optimization.

3.1 Loop method

The loop method follows the idea described by P. Miliotis [29]. The method uses a MIP solver to find an optimal solution to the TSP model. This step is iteratively repeated until an optimal integer solution that does not violate any of the constraints is found. In more detail, the algorithm starts with the definition of the model for the solver stripped of all SECs. The solver then performs optimization until a time limit is reached or the optimal solution is found. Then, the connected components of the solution are found. If the number of connected components is greater than one the SECs for each one of the connected components are added to the model and the optimization is started with the updated model repeating the hole process. The algorithm continues until the optimal solution with only one connected component is found or an overall time limit is reached.

The use of the time limit during the iteration is used to make the process quicker. Let us say that the optimal solution that does not violate any of the constraints of the DTZ formulation has value f^*, then if the solver at some point finds an incumbent solution with value $\overline{f} < f^*$, then we can already say that some of the constraints are being violated, which means that subtours have been formed. Letting the solver continue running would only find a solution which has a value less than or equal to the one found up to now which would also mean that subtours are present. By heuristically choosing a time limit for each iteration we are hoping to stop the solver soon after an incumbent solution better than f^* is found, thus allowing the algorithm to insert new SECs. This is not to say that all these constraints are strictly necessary to find the correct solution with only one tour, however they are useful to eliminate incorrect solutions.

Implementation details of the loop method are described in section 6.2.1.

3.2 Lazy callback

In this method a callback function of the MIP solver is used to add the SEC (1.3) to the model. This function allows the user to monitor the progress of the optimization and to modify the behavior of the MIP optimizer. This particular callback function is called when a new incumbent solution is found by the solver. When the callback is called, the connected components in the solution are found using a union-find algorithm. For more details about the union-find algorithm see appendix A.2. After all the connected components in the solution have been found, the SECs are created and added to the model. The model is updated and the optimization continues. After adding the constraints the incumbent solution found will not be valid anymore. The lazy callback algorithm stops when the optimal solution is found that does not have more than one connected component or when a time limit is reached.

Implementation details of the lazy callback algorithm are described in section 6.2.2.

3.3 User-cut callback

In the user-cut callback method, once a node has been explored, the callback takes the current solution with respect to the sub-problem at that node, and looks for violations of the SECs. When the current solution is a new incumbent one, it means that it is discrete and the callback behaves exactly like in the lazy callback method, the connected components are identified and the constraints added. Instead, when a fractional solution is found, a different formulation of the SECs is used to address the separation problem. Given the usual graph $G = (V, E)$:

$$\sum_{e \in \delta(S)} x_e \geq 2 \quad \forall\, S \subset V, S \neq \emptyset. \tag{3.1}$$

These constraints are mathematically equivalent to those introduced in the DTZ formulation but refer to edges between vertices belonging to a section S and its complement $V \setminus S$. In order to find whether the solution violates these constraints, a max-flow problem is set up, interpreting the cost of each edge as its capacity. Thus the section having minimal capacity S^* is found. If S^* has capacity $k(S) \geq 2$, then the constraint is never violated, otherwise we have found a set $S = S^*$ which violates the constraint. The reason why this different formulation is used for the SP is because of the ease with which the associated max-flow problem can be solved in a fractional solution.

The max-flow associated problem is only solved in a fraction p of nodes that are visited, where p is chosen a priori. This is an hyper-parameter which regulates the computation time dedicated to finding cuts at each node, which can be quite expensive and not worth the increased number of cuts.

Implementation details of the user-cut callback algorithm are described in section 6.2.3.

4 Metaheuristic models

Heuristics are computationally cost-efficient techniques capable of finding solutions which are close to the optimal one. They do so without any formal guarantee of optimality and many do not provide any means of knowing how close to optimality a feasible solution is [30].

The TSP is known to be a NP-hard problem. In some cases the optimal solution cannot be found within a useful amount of time, so it is preferable to obtain an approximate one or one that is close to the optimal one and that requires far less time to be found. The benefits of using heuristics becomes clear in those applications where deadlines are very strict but optimality is not necessary. Here, using a heuristic can succeed in finding a feasible solution even within the deadline, while exact approaches might fail to even obtain it.

Metaheuristics are defined in [22] as: «[...] a generic technique or approach that is used to guide or control an underlying problem-specific heuristic method in order to improve its performance or robustness».
Thus, metaheuristic are used to define a framework which looks through the solution space in order to find a feasible solution which is also close to the optimal one. It does so by intelligently defining a criterion to define a new candidate solution which is then quickly evaluated to find out whether it is better than the one found up to now.

Metaheuristics thus define a method of looking through the solution space and finding those candidate solutions which perform better. The space can be explored using two different approaches. The *constructive approach*, according to which a complete solution is found by iteratively extending partial candidate solutions. How the expansion is performed can be seen as a search problem. The *perturbative approach*, where known candidate solutions are modified to obtain new ones. This approach can be seen as a search of the neighborhood of a solution that has already been found.

The process of finding better solutions is often rendered stronger by adding a stochastic element to the algorithms. The choices that are made when generating or selecting a candidate solution have a certain degree of randomness. Using this approach has been shown to drastically improve the efficiency and robustness of the algorithms (more details in [22]). However, the excessive use of randomness can lead to a blind and inefficient search of the solution space.

Given a generic problem, there is no systematic correlation between a local minimum and the global one. Once an algorithm is capable of finding a "good" solution, the random choices it makes might not be random enough to lead him to even consider the global minimum as a candidate solution. On the other hand, taking large steps can lead to a frantic search without ever being able to even land on a local minimum. This is where *Intensification* and *Diversification* come into play. During *Intensification* the algorithm greedily improves the solution using the objective function as a guide. Instead, during

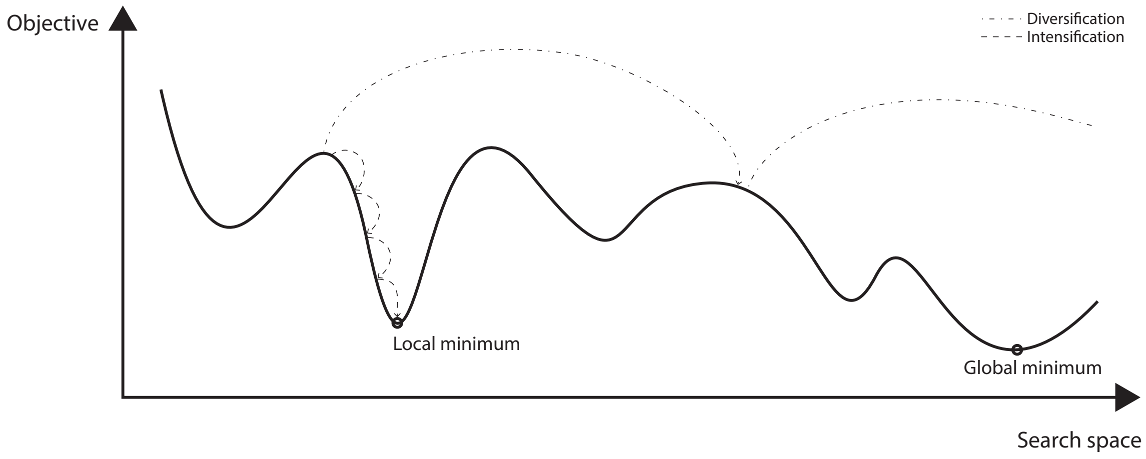


Figure 4.1: The result of the intensification and diversification phases.

Diversification the algorithm tries to prevent stagnation by taking large steps in the search space. Alternating between the two phases makes it possible to quickly find more than one local minimum, increasing the probability of finding the global one. Balancing the two is not always straightforward and often intuition and experience is more effective than theoretically derived principles.

This section introduces some of the metaheuristic algorithms implemented during the course.

4.1 Greedy Randomized Adaptive Search Procedure (GRASP)

The Greedy Randomized Adaptive Search Procedure (GRASP) is the hybridization of a semi-greedy algorithm with a local search method embedded in a multi-start framework. The method consists of multiple applications of local search, each starting from a solution generated with a semi-greedy construction procedure. The best solution found (which can be a local optimum) over all GRASP iterations, is returned.
Algorithm 1 illustrates a basic GRASP heuristic for minimization. After initializing the value of the incumbent the following steps are repeated until the stopping criterion is reached:

1. Generate a solution with the solution generator procedure.
2. If the generated solution is not feasible, a repair procedure is used.
3. Apply local search from the feasible solution provided by the solution generator procedure or, if necessary, by the repair procedure.
4. If the value of the objective function of the local minimum is less than the value of the best objective function found so far, than update the best objective function value and the incumbent solution.

Algorithm 1: Greedy Randomized Adaptive Search Procedure (GRASP)

```
Input: Instance of the problem.
Output: The best solution found during the cycle.
begin
    $f^* \longleftarrow +\infty$;
    repeat
        $x \longleftarrow$ solution generator procedure;
        if $x$ is not feasible then
            Repair($x$);
        $x \longleftarrow$ local_search($x$);
        if $f(x) < f^*$ then
            $f^* \longleftarrow f(x)$;
            $x^* \longleftarrow x$;
    until Stopping criterion;
    return $x^*$;
```

Solution generator procedure The procedure has the task of creating an initial solution, which can be not optimal, for the local search method. The generation of an initial solution can be done with a greedy or a greedy randomized algorithm.
The *greedy algorithm* at each iteration adds to the partial solution being constructed the edge with the minimum cost. This cost is computed based on the algorithm being used (i.e. nearest neighbour or extra-mileage described later). The addition of nodes continues until a complete solution is obtained.
The *greedy randomized algorithm* is based on the same principle guiding pure greedy algorithms, but makes use of randomization to build different solutions at different runs. Randomization can also be used to break ties, enabling different trajectories to be followed from the same initial solution in multi-start methods, or sampling different parts of large neighborhoods.
Algorithm 2 illustrates a greedy randomized algorithm for minimization [32]. At each iteration, a list of candidate is built with all the elements that can be added to the partial solution under construction without destroying feasibility. The selection of the next element is determined by evaluating all the candidates according to a greedy evaluation function. These evaluations lead to the construction of the Restricted Candidate List (RCL). This list contains the best elements. The dimension of the RCL can be limited either by the number of elements (cardinality based) or by their quality (value based). In the first case, the p elements with the best incremental costs are added to the RCL. In the second case, only the elements with incremental cost less or equal than a given threshold are added. A threshold can be:

$$c(e) \in [c^{min}, c^{min} + \alpha \cdot (c^{max} - c^{min})] \tag{4.1}$$

with $\alpha \in [0.0, 1.0]$ and c^{min} and c^{max} the minimum and the maximum incremental cost that a node can given to the solution. The case $\alpha = 0.0$ corresponds to a pure greedy algorithm, while $\alpha = 1.0$ is equivalent to a random construction. After the creation of the RCL an element is chosen randomly and added to the partial solution. Once the element is added to the partial solution the candidate list is updated and the incremental costs are reevaluated.

Algorithm 2: Greedy randomized generator

```
Input: The instance ($V$,$E$), with edge costs, and a seed.
Output: A solution of the problem.
begin
    $x \longleftarrow \emptyset$;
    Initialize the candidate set: $C \longleftarrow E$;
    Evaluate the incremental cost $c(e)\,\forall e \in C$;
    while $C \neq \emptyset$ do
        Build a list with the candidate elements having the smallest incremental
          costs;
        Select an element $s$ from the Restricted Candidate List at random;
        Incorporate $s$ into the solution: $x \longleftarrow x \cup \{s\}$;
        Update the candidate set $C$;
        Reevaluate the incremental cost $c(e)\,\forall e \in C$;
    return $x$;
```

Local search Since the solution generator procedure does not guarantee that the solution found is optimal, a local search procedure is used after having obtained the solution. The local search procedure tries to find a better solution in a neighborhood of the current solution. It terminates when no better solution is found. Algorithm 3 illustrates a basic local search algorithm for minimization [32].
The effectiveness of a local search procedure depends on several aspects, such as the neighborhood structure, the neighborhood search technique, the speed of evaluation of the cost function, and the starting solution [32].

Implementation details of the GRASP algorithm are described section 6.3.1.

Algorithm 3: Local search

```
Input: The solution found by the solution generator procedure.
Output: The best solution in the neighborhood of the input solution.
begin
    while $x$ is not locally optimal do
        Find $x' \in N(x)$ with $f(x') < f(x)$;
        $x \longleftarrow x'$;
    return $x$;
```

4.1.1 Extra mileage

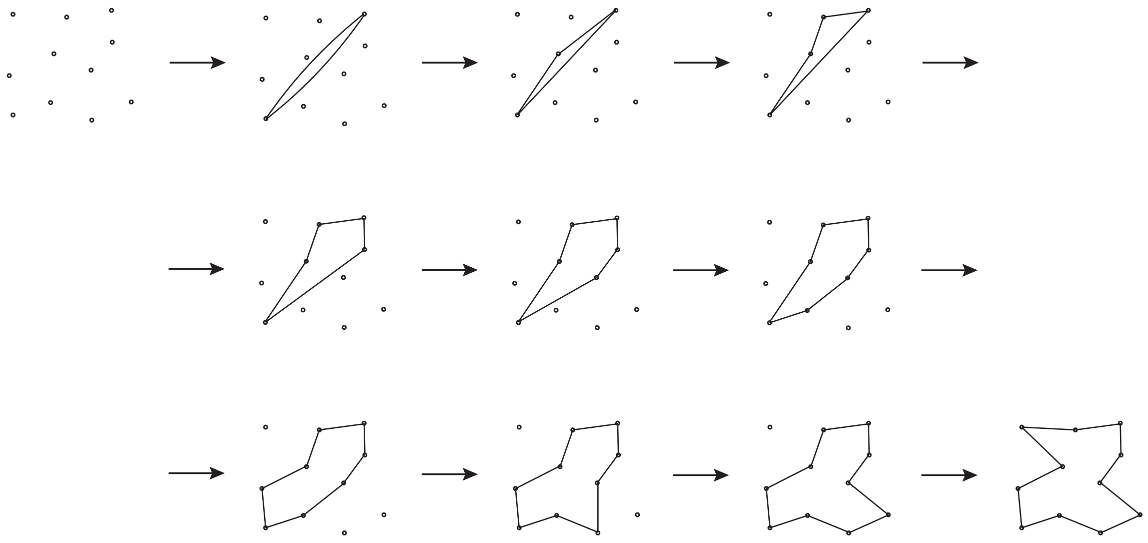


Figure 4.2: Extra mileage algorithm in action.

The extra mileage algorithm is a constructive heuristic. This type of algorithm allows the insertion of any point in the solution sequence of the problem. Since in the case of the TSP one must get back to the starting point, it would be even better to expand a closed route, rather than an open sequence that is then closed at the last step of the procedure. This idea leads to the algorithm illustrated in algorithm 4. Where it starts by selecting the two nodes with maximum or minimum distance, given the set of nodes V. These two nodes should be visited in the final solution. With these nodes, a cycle x is created. After that, all of the nodes not yet visited are considered as potential candidates to being added to the cycle. Figure 4.2 illustrates an example of the phases of the extra mileage algorithm. Whether a node is included in the cycle at the next iteration is decided based on the *extra mileage* criterion. Where at the ith iteration, given a cycle x and a number of nodes not visited in V, for each node $k \in V$ the more convenient insertion point is computed in the following manner. Every edge in x is considered and for each edge $(i,j) \in x$ the extra mileage is calculated:

$$c_{ik} + c_{kj} - c_{ij} = \Delta_k. \tag{4.2}$$

For each edge $(i,j) \in x$ the $\min_{(i,j)\in x} \Delta_k$ is taken. The computation of the extra mileage is repeated for all the nodes in V and the minimum of all the extra mileage is taken. The node with that extra mileage value is added to the cycle replacing the edge (i,j) with the edges (i,k) and (k,j). The algorithm continues until a complete tour is obtained.

Algorithm 4: Extra mileage algorithm

```
Input: The instance of the problem.
Output: The cycle constructed.
begin
    Select the two nodes $(i,j)$ with the maximum (or minimum) distance;
    Add $i$ and $j$ to $x$;
    Mark $i$ and $j$ as visited;
    repeat
        foreach node $k \in V$ not visited do
            foreach edge $(i,j) \in x$ do
                Compute the extra mileage and store the minimum;
        Select $k$ with minimum extra mileage;
        Replace edge $(i,j)$ with the edges $(i,k)$ and $(k,j)$ in $x$;
        Mark $k$ as visited;
    until all nodes in $V$ are visited;
    return $x$;
```

In this version of the algorithm a *first improvement* criterion can be used instead of the *best improvement* one which has been used. Since all addition to the cycle will make the cost worst, one could think to stop the search through the nodes when the extra mileage cost is worse than the previous choice. Although this change might seem unreasonable, it becomes necessary when this algorithm is applied to large instances. When any constructive or perturbative heuristic is used within a metaheuristic schema, often finding a new candidate solution is more important than trying to get the best possible one, because the local minima is then found using other techniques and exploring the space quickly takes priority.

Extra mileage randomized algorithm. A randomized version of this algorithm can be designed by constructing the Restricted Candidate List (RCL) using the *extra mileage* cost. Algorithm 5 shows the steps taken.

Algorithm 5: Extra mileage randomized algorithm

```
Input: The instance of the problem.
Output: The cycle constructed.
begin
    Select the two nodes $(i, j)$ with the maximum (or minimum) distance;
    Add $i$ and $j$ to $x$;
    Mark $i$ and $j$ as visited;
    repeat
        foreach node $k \in V$ not visited do
            foreach edge $(i, j) \in x$ do
                Compute the extra mileage and build candidate list;
        Use value-based or cardinality-based criterion to build RCL;
        Select $k$ randomly from RCL and mark $k$ as visited;
        Replace edge $(i, j)$ with the edges $(i, k)$ and $(k, j)$ in $x$;
    until all nodes in $V$ are visited;
    return $x$;
```

Implementation details of the extra mileage based algorithm are described in section 6.3.3.

4.1.2 Nearest neighbour

The nearest neighbour algorithm is a constructive heuristic. It is based on adding to the route the node which has the smallest cost. The algorithm starts from selecting one of the nodes, in the deterministic version of this algorithm a node with a certain index can be always chosen. This introduces some unexpected randomness, however with no prior knowledge on the instance of the problem, no clever choice can be made. At a given iteration i, let C be the set of all nodes already added to the cycle x, where index i identifies the last node added. For all nodes $k \in V \setminus C$, the edges (i, k) are considered and the one with smallest cost (i, k^*) is picked. Node k^* is added to the cycle after i and the algorithm repeats the steps until all nodes have been added to the cycle. Figure 4.3 shows an example of cycle construction.

As for the *extra mileage* algorithm, a *first improvement* version can be defined, where the first edge which does not have a cost lower that the previous one considered is added.

Nearest neighbour randomized algorithm. A randomized version of this algorithm can be designed just like the *extra mileage* one. A list of the best candidates is kept and the edge is chosen randomly from the RCL built from the list. Algorithm 7 shows the structure.

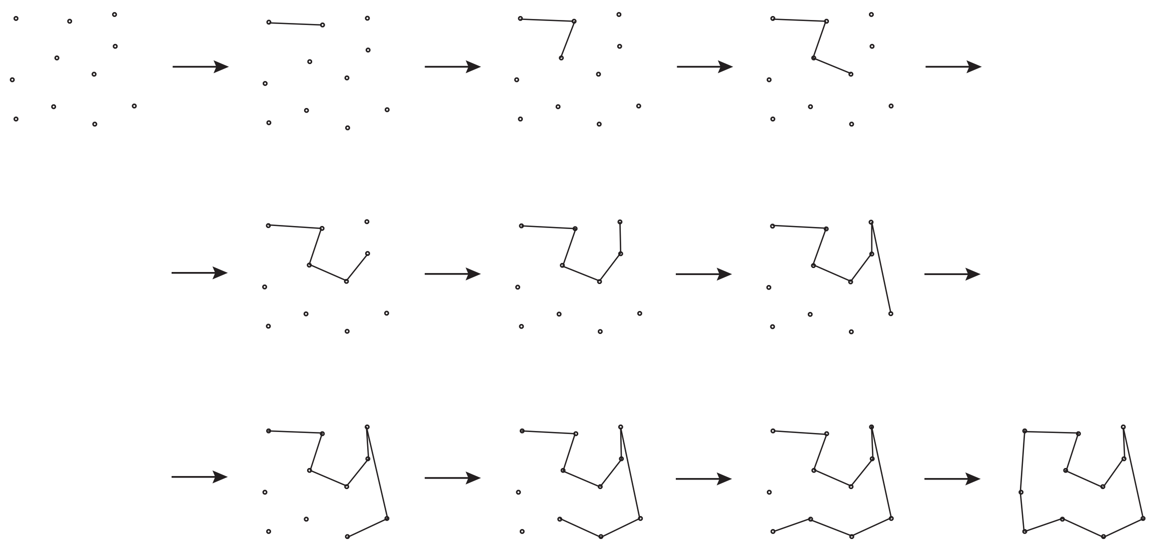


Figure 4.3: Nearest neighbour algorithm in action.

Algorithm 6: Nearest neighbour algorithm

```
Input: The instance of the problem on the graph $G = (V, E)$ with costs
       $c_e \; \forall \; e \in E$.
Output: The cycle constructed.
begin
    Select an arbitrary node $i$ ;
    Add $i$ to $C$;
    repeat
        foreach node $k \in V \setminus C$ do
            Store edge $(i, k^*)$ having smallest cost
        Insert edge with least cost, $(i, k^*)$ in $x$;
        Add node $k^*$ to $C$ ;
        $i \longleftarrow k^*$;
    until $V \setminus C = \emptyset$;
    return $x$;
```

Algorithm 7: Nearest neighbour randomized algorithm

```
Input: The instance of the problem on the graph $G = (V, E)$ with costs
       $c_e \; \forall \; e \in E$.
Output: The cycle constructed.
begin
    Select an arbitrary node $i$ ;
    Add $i$ to $C$;
    repeat
        foreach node $k \in V \setminus C$ do
            Store costs of edges $(i, k)$ in a list
        Use value-based or cardinality-based criterion to build RCL;
        Select $k$ randomly from RCL;
        Insert edge with least cost, $(i, k)$ in $x$;
        Add node $k$ to $C$ ;
        $i \longleftarrow k$;
    until $V \setminus C = \emptyset$;
    return $x$;
```

4.2 Variable Neighborhood Search (VNS)

The Variable Neighborhood Search (VNS) is a heuristic algorithm proposed by N. Mladenović, P. Hansen [27]. The algorithm is based on local search but differs from the others in that it does not follows a trajectory but explores increasingly distant neighborhoods of the current incumbent solution, and jumps from a solution to another if and only if an improvement has been made.

The authors mention three facts to support VNS:
Fact 1 *A local minimum w.r.t. one neighborhood structure is not necessarily so for another;*
Fact 2 *A global minimum is a local minimum w.r.t. all possible neighborhood structures;*
Fact 3 *For many problems, local minima w.r.t. one or several neighborhoods are relatively close to each other.*
These facts are useful to understand how the VNS metaheuristic can be successful. Considering different neighborhoods makes it possible to explore different local minima. The more we explore, the more likely it is that one of the local minima is actually the global one. The last fact comes from empirical evidence, which is useful to determine the structure of the neighborhoods. The idea is first applied to a method to find local minima, Variable Neighborhood Descent (VND).
Variable Neighborhood Descent. VND is a local search heuristic which tries to avoid

falling into local minima by looking at differently-sized neighborhoods. The change of neighborhoods is performed deterministically.
The basic VND scheme is described in algorithm 8.

Algorithm 8: Variable Neighborhood Descent (VND)

```
Input: The instance of the problem.
Output: The local minima.
begin
    Select the set of neighborhood structures $N_k$, for $k = 1, \dots, k_{max}$ that will be
      used in the search;
    Find an initial solution $x$ ;
    repeat
        $k \longleftarrow 1$;
        repeat
            Find the best neighbour $x'$ of $x$ $(x' \in N_k(x))$ ;
            if $f(x') \leq f(x)$ then
                $x \longleftarrow x'$;
                $k \longleftarrow 1$;
            else
                $k \longleftarrow k + 1$;
        until $k = k_{max}$;
    until no improvement is obtained;
    return $x$;
```

Basic VNS. The Basic Variable Neighborhood Search (Basic VNS) algorithm starts with a finite set of preselected neighborhood structures N_k with $k = 1, \dots, k_{max}$, an initial solution generated by a given algorithm and a stopping criterion. The neighborhoods are usually incrementally larger as k increases. The stopping criterion can be a maximum CPU time allowed, a maximum number of total iterations or a maximum number of iteration between two improvements. Then, a new solution x' is randomly chosen from the kth neighborhood of x $(x' \in N_k(x))$, with $N_k(x)$ the kth neighborhood of x. This is done by performing a random *perturbation* of x, this is why this step is called *shaking*. After having obtained x', a local search procedure is used to find the locally optimal solution x''. After x'' is found, if the local solution value is better than the incumbent solution value, then the local solution is set as the new incumbent solution and the algorithm starts from the first neighborhood by setting $k = 1$. Otherwise, k is incremented to change the neighborhood. The steps are repeated until the stopping criterion is reached. Algorithm 9 illustrates the Basic VNS algorithm [21]

Algorithm 9: Variable Neighborhood Search (VNS)

```
Input: The instance of the problem.
Output: The best solution found.
begin
    Select the set of neighborhood structures $N_k$, for $k = 1, \dots, k_{max}$ that will be used in the search;
    Find an initial solution $x$;
    Choose a stopping condition;
    repeat
        $k \longleftarrow 1$;
        while $k \leq k_{max}$ do
            $x' \longleftarrow shaking(x)$;
            $x'' \longleftarrow local\_search(x')$;
            if $f(x'') \leq f(x)$ then
                $x \longleftarrow x''$;
                $k \longleftarrow 1$ ;
            else
                $k \longleftarrow k + 1$;
    until stopping criterion;
    return $x$;
```

It is important to note that the *local_search* is any general approach which looks in the given neighborhood and picks a candidate solution which improves the objective and does so iteratively until no further improvements can be made.

Reduced VNS
This algorithm has the same steps as the Basic VNS but does not perform local search.

General VNS
This is the more general version of VNS. The initial solution is improved with Reduced VNS and the local search is performed by using Variable Neighborhood Descent.

In this project we decided to implement only Basic VNS but the other schemas and VND have been mentioned for completeness. The details on the implementation are described in section 6.3.2.

4.3 Simulated annealing

The Simulated Annealing (SA) is a technique for solving hard combinatorial problems. This technique takes inspiration from the annealing process used in metallurgy. The annealing technique involves the heating and subsequent controlled cooling of a material to increase the quality of the crystal lattices, with the reduction of their defects. When applied to combinatorial optimization, simulated annealing aims to find an optimal configuration (or state with minimum "energy") of a complex problem.

SA was originally proposed by N. Metropolis in the early 1950s as a model of the crystallization process. It was only in the 1980s, however, that independent research done by S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi [24] and V. Cerny [6] noted the similarities between the physical process of annealing and some combinatorial optimization problems. They noted a correspondence between the different physical states and the solution space of an optimization problem. They even observed that the objective function of an optimization problem corresponds with the free energy of the material. A locally optimal solution is associated with a crystal with still some defects, whereas a perfect crystal corresponds to the optimal solution. The analogy is not complete, however, because the temperature in the SA used as an optimization technique is simply a control parameter that has to be properly tuned in order to achieve the desired result. Instead in the annealing process the temperature is a physical variable, which when properly control leads to a perfect crystal.

The SA algorithm starts with an initial configuration (x_0), an initial temperature ($T = T_0$) and an initial number of configurations to generate ($N = N_0$). At a generic iteration, a candidate is chosen if its cost is less than that of the current incumbent solution. Otherwise, it can still be accepted with a certain probability even if it is worse than the current incumbent solution. This ability to perform uphill moves allows simulated annealing to escape from locally optimal configurations. After having generated the N candidates, the algorithm decreases the temperature, the new number of candidates to generate at the temperature level is determined and the process is then repeated. The entire process is controlled by the cooling schedule that determines how the temperature is decreased during

the optimization.
The SA algorithm is summarized in algorithm 10. Where T_k and N_k are, respectively, the control parameter (temperature in the physical annealing) and the number of alternatives generated at the kth temperature level. Initially, when T is large, more deterioration in the cost function is allowed; as the temperature decreases, the simulated annealing algorithm becomes greedier, and only smaller deteriorations are accepted; finally, when T goes to zero, no deteriorations are accepted.

Algorithm 10: Simulated annealing

```
Input: Instance of the problem.
Output: The solution found until stopping criterion.
begin
    Initialize ($T_0$, $N_0$);
    $k \longleftarrow 0$;
    Generate initial configuration $x_h$;
    repeat
        for $1 \leq L \leq N_k$ do
            Generate $x_j$ from $x_h$;
            if $f(x_j) \leq f(x_h)$ then
                $x_h \longleftarrow x_j$;
            else
                if $P_T\,[Accept\, x_j] > random\,[0,1]$ then
                    $x_h \longleftarrow x_j$;
        $k \longleftarrow k+1$;
        Calculation of the length ($N_k$);
        Determine control parameter ($T_k$);
    until Stopping criterion;
    return $x_h$
```

From the current incumbent solution x_h a neighbour solution x_j is generated by the transition mechanism, with costs $f(x_h)$ and $f(x_j)$ respectively. The following probability used in the acceptance test is calculated:

$$P_{T_k}[Accept\, S_j] = \begin{cases} 1, & f(x_j) \leq f(x_h) \\ e^{\dfrac{f(x_h)-f(x_j)}{T_k}}, & f(x_j) > f(x_h). \end{cases}; \tag{4.3}$$

Figure 4.4 illustrates the behaviour of the probability of acceptance given a certain search space and objective function. At a given temperature, the higher the increase of the objective function, the more significant is the probability of accepting a worst move. The lower the increase of the objective function, the less significant is the probability of accepting a worst move. A move which improves the objective is always accepted.

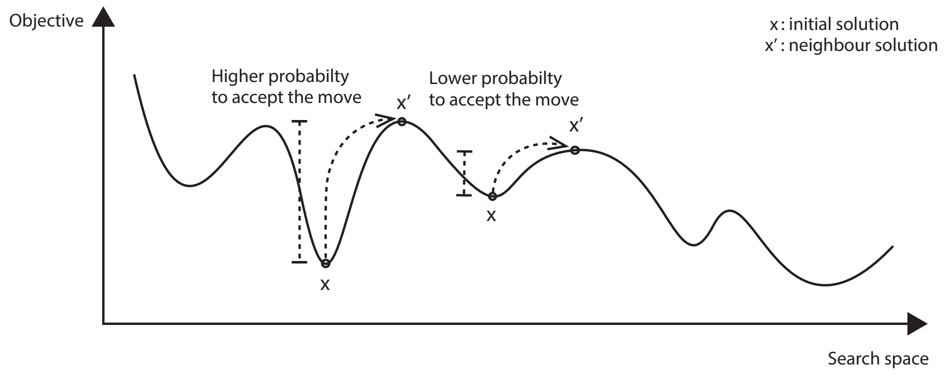

Objective
x: initial solution
x′: neighbour solution
Higher probabilty to accept the move
Lower probabilty to accept the move
x′
x′
x
x
Search space

Figure 4.4: Behaviour of the probability of acceptance and its relation with escaping from local minima in simulated annealing.

The cooling schedule is the control strategy used in the SA algorithm. As the algorithm proceed, the probability of acceptance decreases. The cooling schedule is characterized by four parameters: *initial temperature, number of transitions at a temperature level, the temperature rate of change and the final temperature.*

Initial temperature There are several ways of determining the initial temperature T_0 for the SA algorithm. One consists in the use of a constructive experimental process that simulates the first temperature level of the SA algorithm. With the methodology described in [2] T_0 can be computed as follows:

$$T_0 = \frac{\overline{\Delta f^+}}{\ln\left(\frac{m_2}{m_2\chi - m_1(1-\chi)}\right)}, \tag{4.4}$$

where $\overline{\Delta f^+}$ is the average value of differences in the objective function considering only the raising values within m_0 tries $(m_0 = m_1 + m_2)$. m_1 corresponds with the number of moves with decreasing costs, m_2 is the number of moves with increasing costs and χ corresponds with the acceptance ratio of new configurations.

Another way of determining the initial temperature was proposed in [30]:

$$T_0 = \frac{\mu}{-\ln\phi} f(x^0), \tag{4.5}$$

where it is assumed that $\phi\%$ of the uphill moves, which are $\mu\%$ worse than the initial solution $f(x^0)$, are accepted at the initial temperature level T_0.

Number of transitions at a temperature level The number of transitions N_k is closely related with the temperature reduction rate and should be such that the condition of thermal near-equilibrium is guaranteed. Most algorithms use a value of N_k that depends on the number of decision variables of the problem. There are two ways to compute the number of transitions at each temperature level: *static* and *adaptive*.
The *static strategy* determines the number of transitions before the search starts. For instance, a given proportion y of the neighborhood $N(s)$ is explored. Two static methods for determining the number of transitions N_k at temperature level k are:

$$N_{k+1} = N_0 \tag{4.6}$$

and

$$N_{k+1} = \rho \cdot N_k, \tag{4.7}$$

where $\rho \geq 1$ is a supplied parameter and N_0 is the number of transition at the initial temperature level [33].

In the *adaptive approach* the number of generated neighbors will depend on the characteristics of the search. For instance, it is not necessary to reach the equilibrium state at each temperature. Non-equilibrium simulated annealing algorithms may be used: the cooling schedule may be enforced as soon as an improving neighbour solution is generated. This may result in the reduction of the computational time without compromising the quality of the obtained solutions [5].
Another adaptive approach uses both the worst and the best solutions found in the neighborhood search step of the algorithm. Let f_l (resp. f_h) denote the smallest (resp. largest) objective function value in the explored neighborhood. The next number of transitions N is defined as follows:

$$N = N_0 + \lfloor N_0 \cdot F_- \rfloor, \tag{4.8}$$

where $F_- = 1 - e^{-\frac{f_h - f_l}{f_h}}$, and N_0 is the initial value of the number of transitions [4].

Temperature rate All methods for determining the temperature reduction are based on the fact that thermal equilibrium should be reached before the temperature goes to zero. The cooling rate can be computed with a *constant* or *variable* method.

The *constant cooling rate* is calculated as follows:

- Geometric schedule

$$T_{k+1} = \alpha \cdot T_k, \tag{4.9}$$

  where $\alpha \in (0.0, 1.0)$. It is the most popular cooling function. Experience has shown that α should be between 0.5 and 0.99.

- Linear schedule

$$T_{k+1} = T_0 - k \cdot \beta, \tag{4.10}$$

  where β is a specified constant value.

- Logarithmic schedule

$$T_{k+1} = \frac{T_0}{\log(k)}, \tag{4.11}$$

  this schedule is too slow to be applied in practice but has the property of the convergence proof to a global optimum [19].

The *variable cooling rate* is calculated with one of the following formulas:

$$T_{k+1} = \frac{3 \cdot \sigma(T_k)}{3 \cdot \sigma(T_k) + \ln(1+\delta) T_k} T_k, \tag{4.12}$$

where $\delta \in [0.01, 0.20]$;

$$T_{k+1} = e^{-\frac{\lambda \cdot T_k}{\sigma(T_k)}} T_k, \tag{4.13}$$

where $\lambda \leq 1$ and $\sigma(T_k)$ is the standard deviation of the costs of the configurations generated at the previous temperature level T_k.

Stopping criterion For the stopping condition, theory suggests a final temperature equal to zero. In practice the following stopping criteria may be used:

- When the probability of accepting a move is negligible, one can stop the search.
- Reaching a determined final temperature T_f that must be small.
- Achieving a predetermined number of iterations without improvement of the best found solution until now [34].

- Achieving a predetermined number of times that a percentage of neighbors at each temperature is accepted; that is, a counter increases by 1 each time a temperature iteration is completed with a percentage of accepted moves less than a predetermined limit and is reset to 0 when a new best solution is found. If the counter reaches a predetermined limit R, the SA algorithm is stopped [23].

Implementation details of the SA algorithm are described in section 6.3.5.

5 Matheuristics

Matheuristics are heuristic algorithms created by the combination of meta-heuristics and Mathematical Programming (MP) techniques. An essential feature is the exploitation in some part of the algorithms of features derived from the mathematical model of the problems of interest.

5.1 Hard-fixing

The hard-fixing scheme starts from a correct heuristic solution x^{heu} for the problem and, with a certain probability p, randomly chooses whether an edge is fixed in the solution or not. Fixing the edges is done by changing the lower bound of the MIP variables or using fixing constraints. Once a new solution x_s is found using the solver, its value is compared to the one of the best solution found so far x^{heu} and if its better, then the best heuristic solution is updated $x^{\text{heu}} = x_s$ and the fixing steps are repeated. The algorithm continues until a stopping criterion is reached. The stopping criterion can be a time limit or a maximum number of iterations without improvement.
The parameter p is an hyper-parameter and indicates the fixing probability of the hard-fixing. A constant value of p can be used for all the iterations, e.g. $p = 0.5$. Otherwise, one can starts with a large value of p, e.g. 0.9, and after a number of iterations without improvements of the solution values the value of p is reduced.

Another method is using a *soft-fixing constraint* (5.1) to be added in the MIP model after a new solution is found by the solver. The soft-fixing constraint is defined as follows:

$$\sum_{e:x_e^{\text{heu}}=1} x_e \geq p \cdot |V|. \tag{5.1}$$

where p can be seen as akin to the fixing probability. The value of p increases or decreases the neighborhood radius of the solution x, as illustrated in fig. 5.1.

Implementation details of the hard-fixing are described in section 6.4.1.

5.2 Local branching

The local branching scheme, devised by Fischetti and Lodi [15], uses the MIP solver in a "black-box" manner. This approach adds to the MIP model the *local branching constraint*:

$$\sum_{e:x_e^{\text{heu}}=0} x_e + \sum_{e:x_e^{heu}=1} (1 - x_e) \leq k \tag{5.2}$$

where x^{heu} is the heuristic solution found so far and k is the radius of a small neighborhood. (solutions with Hamming distance less or equal to k.) The value of k must be at most 20; typical values are 5, 10 and 20.

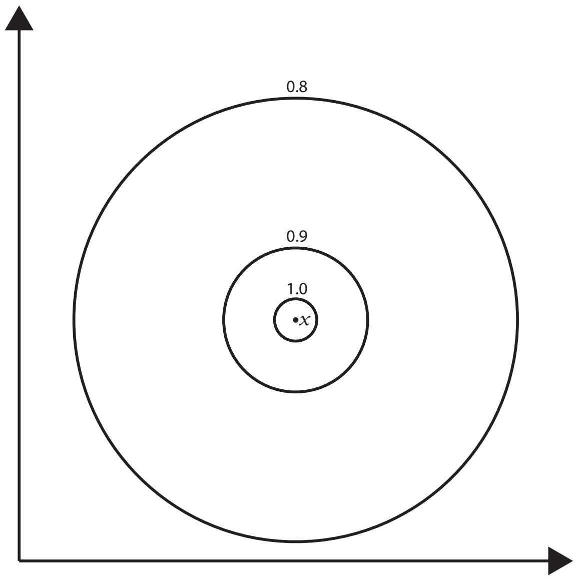


Figure 5.1: Behaviour of the radius of the soft-fixing neighborhood for different values of p.

In the TSP the first addend in (5.2) is useless for the fixing. Therefore the asymmetric form of the local branching constraint (5.3) can be used.

$$\sum_{e:x_e^{\text{heu}}=1} (1 - x_e) \leq k. \tag{5.3}$$

With a succession of algebraic steps one can obtain:

$$\sum_{e:x_e^{\text{heu}}=1} x_e \geq |V| - k. \tag{5.4}$$

Notice how now the constant k refers to half the maximum Hamming distance because we allow k edges to go from 1 to 0, which in the TSP implies that the same number of edges must go from 0 to 1, effectively doubling the the amount of variables that have been changed (and thus the Hamming distance).

Implementation details of the local branching are described in section 6.4.2.

6 Implementation

The section describes the implementation of some of the models and methods explained during the course. Firstly the compact models are described, followed by exact models, metaheuristics and matheuristic algorithms.

All the models start with the initialization of the instance of the problem. This is a structure containing the input data file, the specific parameters for the different models and the pointer to the Gurobi environment and model, where the MIP solver is used.

6.1 Compact model

The compact model require the use of the Gurobi MIP solver. So before the optimization starts one must to create the environment, the model and add the variables and the constraints.

The Gurobi environment can be initialized with the method:

```
int GRBloadenv(GRBenv **envP, const char *logfilename)
```

where `envP` is the location in which the pointer to the newly created environment should be placed and `logfilename` is the name of the log file for this environment. May be null (or an empty string), in which case no log file is created [20].

A new model can be created with the method:

```
int GRBnewmodel(GRBenv *env, GRBmodel **modelP,
       const char *Pname, int numvars, double *obj,
       double *lb, double *ub, char *vtype,
       const char **varnames)
```

where `env` is the environment in which the new model should be created. `modelP` is the location in which the pointer to the new model should be placed, `Pname` is the name of the model, `numvars` is the number of variables in the model, `obj` is the objective coefficients for the new variables, `lb` is the lower bounds for the new variables, `ub` is the upper bounds for the new variable, `vtype` is the types for the variables and `varnames` is the names for the new variables [20].
The `obj`, `lb`, `ub`, `vtype`, `varnames` can be null, for more details consult the Gurobi documentation [20].

After created the Gurobi environment and the model the problem variables are added to the model with their upper and lower bounds, variables types and names.

To add the variables to the model the following method can be used:

```
int GRBaddvars(GRBmodel *model, int   numvars, int numnz,
       int *vbeg, int *vind, double *vval, double *obj,
       double *lb, double *ub, char *vtype,
       const char **varnames)
```

where `model` is the model to which the new variables should be added, `numvars` is the number of new variables to add, `numnz` is the total number of non-zero coefficients in the new columns, `vbeg` is the constraint matrix non-zero values are passed into this routine in Compressed Sparse Column (CSC) format, `vind` is the constraint indices associated with non-zero values, `vval` is the numerical values associated with constraint matrix non-zeros, `obj` is the objective coefficients for the new variables, `lb` is the lower bounds for the new variables, `ub` is the upper bounds for the new variables, `vtype` is the types for the variables and `varnames` is the names for the new variables [20].
The `obj`, `lb`, `ub`, `vtype`, `varnames` can be null, for more details consult the Gurobi documentation [20].

To add a new constraint to the model the following method is used:

```
int GRBaddconstr(GRBmodel *model, int numnz, int *cind,
       double *cval, char sense, double rhs,
       const char *constrname)
```

where `model` is the model to which the new constraint should be added, `numnz` is the number of non-zero coefficients in the new constraint, `cind` is the variable indices for non-zero values in the new constraint, `cval` is the numerical values for non-zero values in the new constraint, `sense` is the sense for the new constraint, `rhs` is the right-hand-side value for the new constraint and `constrname` is the name for the new constraint [20].
Only `constrname` can be null, in which case the constraint is given a default name [20].

One can add all the constraints to the model using the following method:

```
int GRBaddconstrs(GRBmodel *model, int numconstrs,
       int numnz, int *cbeg, int *cind, double *cval,
       char *sense, double *rhs, const char **constrnames)
```

where

`model` is the model to which the new constraints should be added, `numconstrs` is the number of new constraints to add, `numnz` is the total number of non-zero coefficients in the new constraints, `cbeg` is the constraint matrix non-zero values are passed into this routine in Compressed Sparse Row (CSR) format by this routine, `cind` is the variable indices associated with non-zero values, `cval` is the numerical values associated with constraint matrix non-zeros, `sense` is the sense for the new constraints, `rhs` is the right-hand-side values for the new constraints and `constrnames` is the names for the new constraints [20].
Only `constrname` can be null, in which case the constraint is given a default name [20].

The variables and constraints added to the model, due to the lazy update approach of Gurobi, will not be added until one update the model (using `GRBupdatemodel`), optimize the model (using `GRBoptimize`) or write the model to disk (using `GRBwrite`).

6.1.1 Miller-Tucker-Zemlin

In the MTZ model an ATSP is used. After added all the variables, their bounds and the indegree and outdegree constraints likes in eq. (2.9), (2.2) and (2.3), the supplementary variables u are added to the model using the `GRBaddvars` and `GRBaddconstr` methods. Subsequently, the lazy constraints (2.4) and (2.5) are added to the model with the `GRBaddconstr` following by the setting of the attribute lazy to the new added constraints. For setting an attribute value to an element in the model one can use the following method:

```
int GRBsetintattrelement(GRBmodel *model,
      const char *attrname, int element, int newvalue)
```

where `model` is a loaded optimization model, `attrname` is the name of an integer-valued array attribute, `element` is the index of the array element to be changed and `newvalue` is the value to which the attribute element should be set [20]. For more details consult the Gurobi documentation [20].

For declared an element lazy one must set `attrname` to `"Lazy"` and select one of the three lazy levels provided by Gurobi. The lazy levels are the following [20]:

- With a value of **1**, the constraint can be used to cut off a feasible solution, but it won't necessarily be pulled in if another lazy constraint also cuts off the solution.
- With a value of **2**, all lazy constraints that are violated by a feasible solution will be pulled into the model.
- With a value of **3**, lazy constraints that cut off the relaxation solution at the root node are also pulled in.

After added all the lazy constraints the optimization started with a `GRBoptimize` call.

6.1.2 Exercise compact

In the exercise compact model a TSP is used. After added all the problem variables with their lower and upper bounds and the degree constraints, the supplementary variables z are added to the model with their bounds defined in eq. (2.16) and (2.17). Subsequently the constraints (2.12), (2.13) and (2.14) are added to the model. Added all the variables and constraints the MIP optimizer is started.

6.1.3 Single Commodity Flow

In the single commodity flow an ATSP instance of the problem is used. First the variables x and y are added to the model with their lower and upper bounds. For the y variables an upper bound equal to infinity is chosen. Added the problem variables the indegree and outdegree constraints are added to the model. Subsequently the commodity flow constraints (2.18), (2.19) and (2.20) are added. After added all the constraints the MIP optimizer is started.

6.1.4 Two Commodity Flow

In the two commodity flow an ATSP instance of the problem is used. First the variables x, y and z are added to the model with their lower and upper bounds. For the y and z variables an upper bound equal to infinity is chosen. Added the problem variables to the model the indegree and outdegree constraints are added to it. Subsequently the commodity flow constraints (2.21), (2.22), (2.23), (2.24), (2.25) and (2.26) are added. After added all the constraints the MIP optimizer is started.

6.1.5 Multi-Commodity Flow

In the multi-commodity flow an ATSP instance of the problem is used. First the variables x and y are added to the model with their lower and upper bounds. Added the problem variables to the model the indegree and outdegree constraints are added to it. Subsequently the commodity flow constraints (2.27), (2.28), (2.29), (2.30), (2.31) and (2.32) are added. After added all the constraints the MIP optimizer is started.

6.1.6 1st Timed Stage Dependent

In the 1st timed stage dependent model an ATSP instance of the problem is used. First the variables x and y are added to the model with their lower and upper bounds. Added the problem variables to the model the indegree and outdegree constraints are added to it. Subsequently the commodity flow constraints (2.33), (2.34) and (2.35) are added. After added all the constraints the MIP optimizer is started.

6.1.7 2nd Timed Stage Dependent

In the 2nd timed stage dependent model an ATSP instance of the problem is used. First the variables x and y are added to the model with their lower and upper bounds. Added the problem variables to the model the indegree and outdegree constraints are added to it. Subsequently the commodity flow constraints (2.36), (2.37) and (2.38) are added. After added all the constraints the MIP optimizer is started.

6.1.8 3rd Timed Stage Dependent

In the 3rd timed stage dependent model an ATSP instance of the problem is used. First the variables x and y are added to the model with their lower and upper bounds. Added the problem variables to the model the indegree and outdegree constraints are added to it. Subsequently the commodity flow constraints (2.39), (2.40) and (2.41) are added. After added all the constraints the MIP optimizer is started.

6.2 Exact model

Exact models, like the compact models, require the use of the Gurobi MIP solver. Therefore, a procedure similar to that described in section 6.1 is used for the initialization of the Gurobi MIP solver and the model.

6.2.1 Loop method

The loop algorithm starts with the initialization of the Gurobi environment and model. Afterwards, the variables and the constraints of the TSP are added to the Gurobi model. A variable time limit is set heuristically to try and get to a point where an incumbent solution has been found that has more than one connected component. When the solver reaches the time limit the solution is retrieved and the connected components are searched using the iterative mode (appendix A.3) or the union-find (appendix A.2). The SECs are added to the model and a new optimization is started. After the Gurobi solver reaches the time limit, it is incremented until a threshold is reached. When this threshold is reached the time limit is removed from the model and the optimization continues until the optimal solution is found.

6.2.2 Lazy callback

The implementation of the lazy callback model starts with the standard TSP model defined in 1.2 without the SECs (1.3). The implementation starts with the definition of the Gurobi environment and model. The environment is setup so that pre-processing does not influence adding cuts later one. It is done using `PreCrush` which allows the presolve to translate constraints on the original model to equivalent constraints on the presolved model. Also, another parameter is changed to tell Gurobi that lazy constraints will be added during the Branch and Cut. Subsequently, the variables and the constraints of the model are added to the Gurobi model and the optimization is launched. When the solver finds an integer (not optimal) solution, a callback is called. The callback looks for the connected components with the union-find algorithm (appendix A.2). The union-find algorithm uses a path halving method and a union by size method. The union by size was chosen because for each connected component, its representative has the total number of nodes in the component and this value is used for the right-hand side of (1.3) when the SEC is generated and added to the model. To use a callback in Gurobi is useful create a structure to pass the data to the callback function. Afterwards, define the steps to be performed in the callback with

```
int __stdcall mycallback(GRBmodel *model, void *cbdata,
      int  where, void *usrdata)
```

function and set the callback to the model with

```
int GRBsetcallbackfunc(GRBmodel *model,
      int (*cb)(GRBmodel *model, void *cbdata, int where,
         void *usrdata), void *usrdata)
```

before start the optimization.

6.2.3 User-cut callback

The user-cut callback implementation starts with the initialization of the Gurobi environment and model. Subsequently, the variables and the constraints of the TSP are added to the Gurobi model, without the SEC. The callback function is added to the Gurobi model and the optimization is started. When the Gurobi solver finds a integer but not optimal solution the callback are called. The callback function search for connected components as done in the lazy callback model, section 6.2.2. When the Gurobi solver explores a MIP node if the number of connected components is larger than 1, the algorithm adds the SEC to the model as done in the lazy callback algorithm. Instead, if the number of connected components is equal to one the callback uses the Concorde method:

```
int CCcut_mincut(int ncount, int ecount, int *elist,
      double *dlen, double *cutval, int **cut,
      int *cutcount)
```

where `ncount` is the number of nodes in the graph, `ecount` is the number of edges in the graph, `elist` is the list of edges in end0 end1 format, `dlen` is a list of the edge capacities, `cutval` returns the capacity of the mincut (it can be null), `cut` will return the indices of the nodes in the minimum cut, and `cutcount` will return the number of nodes in the minimum cut if cut is not null (if cut is null, then `cutcount` can be null). `cut` can be passed in as null, otherwise it will be an allocated to an array of the appropriate length [8]. The `CCcut_mincut` computes the global minimum cut in the solution found by the Gurobi solver when the callback is called. With the returned data the cut is added to the model with the Gurobi method:

```
int GRBcbcut(void *cbdata, int cutlen, const int *cutind,
    const double *cutval, char  cutsense, double cutrhs)
```

where `cbdata` is the cbdata argument that was passed into the user callback by the Gurobi optimizer, `cutlen` is the number of non-zero coefficients in the new cutting plane, `cutind` is the variable indices for non-zero values in the new cutting plane, `cutval` is the numerical values for non-zero values in the new cutting plane, `cutsense` is the sense for the new cutting plane and `cutrhs` is the right-hand-side value for the new cutting plane [20].

6.3 Metaheuristics

6.3.1 Greedy randomized adaptive search procedure (GRASP)

The implementation of the GRASP algorithm follows the structure described in algorithm 1. The algorithm starts with the population of a structure contained the edge costs of the problem and the two extreme points of the edge of the graph. After the initialization of the structure the algorithm starts the computation of a feasible solution with the greedy randomized algorithm, described in algorithm 2.
The nearest neighbour randomized algorithm starts choosing randomly the first node of the tour. From this node the minimum and maximum edge cost values are found and the

threshold (4.1) is computed by selecting α randomly. With this threshold, a new RCL is generated for the current node. Randomly, a node in the RCL is selected and the edge is added to the solution. The node connected by the edge becomes the new node. Before building a new RCL the edges connected to the previous node are made non-selectable by the RCL generator. Afterwards, a new RCL is generated with the remaining edges. The procedure is repeated until a feasible solution is generated.
After having generated the feasible solution with the nearest neighbour randomized algorithm, the local search starts. For the local search a 2-opt algorithm is used (details in appendix A.4). When the algorithm finds the best candidate in the 2-exchange neighborhood, the value of the current solution is compared with that of the best solution found so far. If the value of the current solution is better, the current solution becomes the new best solution. Otherwise, the best solution remains the same. The GRASP procedure is repeated until a time limit is reached.

6.3.2 Variable neighborhood search (VNS)

In this project we decided to implement the Basic VNS, seen in section 4.2. The initial solution is obtained using a heuristic method chosen by the user, the preferred method is nearest neighbour. At each iteration a `kick` is performed in order to *shake* the current incumbent solution and explore the neighborhood. Afterwards, 2-opt is applied and the local minima are compared to the incumbent solution, which is updated whenever a better candidate solution is found. The kick is always performed starting from the incumbent solution. The number of times a kick is performed at each iteration increases whenever the neighborhood to explore needs to be larger. This can be justified by looking at 2-opt moves. When three consecutive random 2-opt moves are performed, the number of different cycles which are obtainable is higher than what can be obtained with only one single random 2-opt move. That is we can create a solution which was not possible to obtain with only a random 2-opt move. Furthermore, all cycles that can be obtained with a single 2-opt random move are also obtainable with three consecutive 2-opt random moves (i.e. swapping in and out any pair of edges, then performing the 2-opt move). This is a very non-rigorous way to show that consecutive 2-opt random moves explore neighborhoods that are increasingly larger. In our case we perform 3-opt moves and it is important to notice that the neighborhoods are not nested. The number of possible candidate solutions increases as the number of consecutive random 3-opt moves increases, but not necessarily the neighborhood explored with one move is nested within the one explored with two moves and so on (this is true also for 2-opt moves). Thus, the use of multiple moves is only an approximation of the increasingly large neighborhood structures required by the VNS.

When running VNS, we realized that a certain neighborhood would be worth exploring for more than one iteration before going to the next one. This is just an empirical observation. In order to do so in a more systematic way, we applied a *drag*, which would increase the number of unsuccessful iterations necessary before increasing the number of kicks.

6.3.3 Extra-mileage

The extra-mileage algorithm was implemented following the pseudocode shown in algorithm 4. One observation is in the randomized version (algorithm 5). Here the RCL is implemented by picked the K best candidates, where K is a constant hyper-parameter chosen a priori. With a probability p the best one is chosen (usually 0.5), this leaves a probability of $\dfrac{1-p}{N-1}$ that one of the other candidates is chosen. This was done to reduce the randomness in the choice of candidate solution. Keeping the RCL ordered was done using insertion sort, but it does not influence in the overall complexity of the algorithm, as the number of candidates in the RCL is a constant.

6.3.4 Nearest neighbour

The nearest neighbour algorithm was implemented following the pseudocode shown in algorithm 6 and algorithm 7. Here the RCL is managed similarly to the extra-mileage algorithm.

6.3.5 Simulated annealing

The implementation of the simulated annealing algorithm starts with the computation of the initial temperature T_0 with the eq. (4.5) and the initial number of neighbors N_0 to be generated at the initial temperature is proportional to the number of edges in the TSP instance.
The algorithm, after the initialization phase, starts the annealing procedure, as illustrated in algorithm 10. The annealing phase starts with the generation of a neighbour with a random 2-opt move. Afterwards, the value of the new candidate solution and the value of the incumbent solution are compared and if the new value is better then the candidate solution becomes the new incumbent solution. Otherwise, a probability of acceptance is computed, as per eq. (4.3), which is used to decide whether the candidate solution should be set as the new incumbent solution despite having a value which is worse. This is repeated until N_k neighbors are generated and compared.
After this, k is incremented by 1, the temperature and the new number of neighbors to generate for the updated temperature are calculated. For the number of neighbors, the eq. (4.8) is used and for the new temperature value the eq. (4.12) is used. The data required for the computation of these values are stored in variables and arrays during the previous phase (generation of the N_k neighbors). For the computation of the standard deviation the following formula is used:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(c_i - \overline{c})^2}{N-1}}, \tag{6.1}$$

where c is the vector of costs of the neighbors generated at the previous temperature, N is the number of neighbors generated and $\overline{c}$ is the mean cost of the candidate solutions.

The annealing procedure continues until the time limit is reached, this stopping condition was chosen mainly so that it would be easier to compare with other metaheuristics.

6.4 Matheuristics

6.4.1 Hard-fixing

The hard-fixing algorithm starts with the generation of a starting feasible solution of the TSP and the initialization of one of the exact model introduced in section 3. After the initialization, the selected exact model is run to finding a solution after a short time. If no better solution was found by the solver then, the initial solution is used. Subsequently, the solution founds by the solver is stored and the percentage of improvement of the solution is calculated. If this percentage is less than a threshold and the number of times the solution not improved is bigger than a threshold the neighborhood radius is increased (the probability is reduced). The solution is updated if it is better than the incumbent solution. Subsequently, the algorithm fixes a percentage of edges of the solution by setting the lower bounds of the corresponding variables to 1. The model is updated and the solver is started. The algorithm terminates when the time limit is reached or the optimal solution is found.

6.4.2 Local branching

The local branching algorithm starts with the generation of an initial feasible solution of the TSP and the initialization of one of the exact models introduced in section 3. After the initialization, the solution is provided to the Gurobi solver and a short time limit is set in the hopes of slightly improving the initial solution. After obtaining a feasible solution, the local branching constraints phase starts. In that phase the percentage of improvement of the solution is calculated. If this percentage is less than a threshold and the number of times the solution does not improve is bigger than a threshold, the variable k in eq. (5.4) is increased. Subsequently, the latest local branching constraint added to the model is removed and the new local branching constraint is added to the model according to the eq. (5.4). After added the constraint the model is updated and the solver is started. The algorithm terminates when the time limit is reached or the optimal solution is found.

6.5 Combining metaheuristics and exact methods

When running metaheuristics, it becomes apparent that after a while the improvements made are negligible. After this stage has been reached, one can think to apply a more conventional method of looking for an optimal solution, starting from what we already know is a good candidate solution in the space. In our implementation we decided to use the lazy callback method, as it was the one which performed best amongst all exact methods tested.
Combining the two methods is quite straightforward, after the metaheuristic has been run, the solution is stored in an array and is then fed to the Gurobi solver using the `Start` attribute. What is instead not obvious is the choice of the time to dedicate to the metaheuristic. This is an hyperparameter which needs to be tuned so that the metaheuristic

reaches a plateau in the improvement. Since we wanted to avoid hypertuning the parameters, we settled on a subdivision a priori without changing it after getting the results: $\frac{T}{4}$ to the metaheuristic and $\frac{3T}{4}$ to the exact method.

7 Experiments and evaluation

In order to effectively benchmark the performance of the methods implemented we make use of *performance profiling*, introduced by Dolan and Moré [12].

Follows a brief introduction of *performance profiling* applied to a general scenario, and then some considerations on using it in our specific case.

7.1 Performance profiling

A set of methods M is used to solve a set of problems P. Let $|M| = n_m$ and $|P| = n_p$. The performance measure used is the computing time $t_{m,p}$, which is the time taken by method m to solve problem p. Comparing the performance of a specific method on a certain instance is done by calculating the *performance ratio*:

$$r_{m,p} = \frac{t_{m,p}}{\min\{t_{m,p} : p \in P\}}$$

Here it is assumed that all ratios are within a maximum value $r_M \geq r_{m,p} \; \forall \; m \in M, p \in P$ which is chosen such that $r_{m,p} = r_M$ if and only if the method did not find a solution within the time limit.

Using this ratio we can gauge the performance of a method on a given problem, by understanding how much slower it was with respect to the best one. However, this is not enough to asses the overall performance of the method used. To achieve this, we define the *performance profile*:

$$\rho_m(\tau) = \frac{1}{n_p} |\{p \in P : r_{m,p} \leq \tau\}|$$

where $\tau \in \mathbb{R}$. It follows from the definition that the *performance profile* is the empirical cumulative distribution function for the performance ratio, the probability that method m has a performance ratio within a factor τ of the best possible ratio.

Some observation on the meaning of the values of the *performance profile* can be made. From the definition of *performance ratio* we can say that $\rho_m(\tau) = 0 \; \forall \; \tau < 1, m \in M$. In particular, $\rho_m(1)$ will be the probability that method m solves a problem before any other method. Moreover, because of how we constrained the value of the *performance ratio*, it will be that $\rho_m(r_M) = 1 \; \forall \; m \in M$ and by approaching r_M from the left, we get the probability that a method is capable of solving any one problem. This last observation is due to the fact that $r_{m,p} = r_M$ iff the method was not capable of finding a solution for that problem within the time limit, which means that every problem that was solved within the time limit had a *performance ration* $r_{m,p} < r_M$. Consequently, we can define the probability that method m solves a problem:

$$\rho_m^* = \lim_{\tau \to r_M^-} \rho_m(\tau)$$

7.2 Description of the performance profiling setup

Now, a brief summary of the different classes of solvers studied in this report and compared in this analysis. The first ones each use a different formulation of the TSP problem which has a number of constraints polynomial in the number of nodes in the graph (compact models section 2). The second ones are the exact-methods, they use the solver and gradually add the constraints when it is deemed necessary. These methods are callback-based or loop-based solvers: with callback-based solvers the optimal solution is reached by adding Subtour Elimination Constraints (SECs) in the sub-problems of the branch-and-cut tree, while with loop-based solvers, SECs are added after finding the optimal solution to a problem with potential subtours. The third ones make use of heuristics to try to get near the optimal solution. To this class belong matheuristics, metaheuristics and a combination of metaheuristics with an exact method.

Because of the diverse nature of the solvers, they are compared separately. The testbed used for the metaheuristics is made of larger instances.

For both classes of problems the main metric will be the *performance profile*, the first two classes of problems use as performance measure the computing time necessary to reach the optimum of the instance (with a time limit of 3600 seconds). The third class, since we are comparing heuristics, the performance measure is the value of best solution found within the time limit.

A summary of what was recorded is also provided in the appendix.

7.3 Performance variability

Performance variability can be briefly defined as a change in the performance of the solver because of a difference in the environment which should not influence the performance. It is an unwanted change in the speed with which one reaches the solution. This could manifest in a change in the number of nodes visited during the branch&cut or a change in the number of iterations needed to run the simplex algorithm. The reasons behind this variability have been extensively studied in recent years (a short overview is available at [10]).

There are several factors which contribute to this issue [25]. The first cause are imperfect tie-breaking. When a choosing between different candidates, the one with the highest score is picked. However, there needs to be a secondary criterion to choose amongst candidates with the same score. If this criterion is imperfect, the selection might be made based on the order the candidates appear or may be influenced by rounding errors made by the machine which is running the solver. Another factor is floating-point arithmetic, which can be subject to the order in which the computation is performed, and this order is decided by the compiler at optimization. Variability is thus depended on the nature of the model itself and the MIP solver used. Furthermore, several heuristics are applied during the optimization, and these are based on the use of a random seed, which can influence the performance irrespective of the instance being solved.

We decided to deal with this problem by running our tests using different seeds. Each instance is run using five different seeds, and each is seen as a new instance. This effectively reduces the effects of the performance variability: we change the environment and we consider this change as a new instance for the solver. The idea is that a certain method can be lucky only on a small percentage of the seeds and the performance profile then lets compete the solvers on the same ground. Excluding some of the measurements, for example by selecting the best one amongst different seeds, could introduce some bias because of the relatively small set of seeds and instances used.

7.4 Results

The methods are compared using *performance profiling* and are split into different groups as to better understand the performance without over-crowding the graph. The time limit for compact models and exact methods is 3600 seconds, while for the various heuristics it is 1800 seconds.

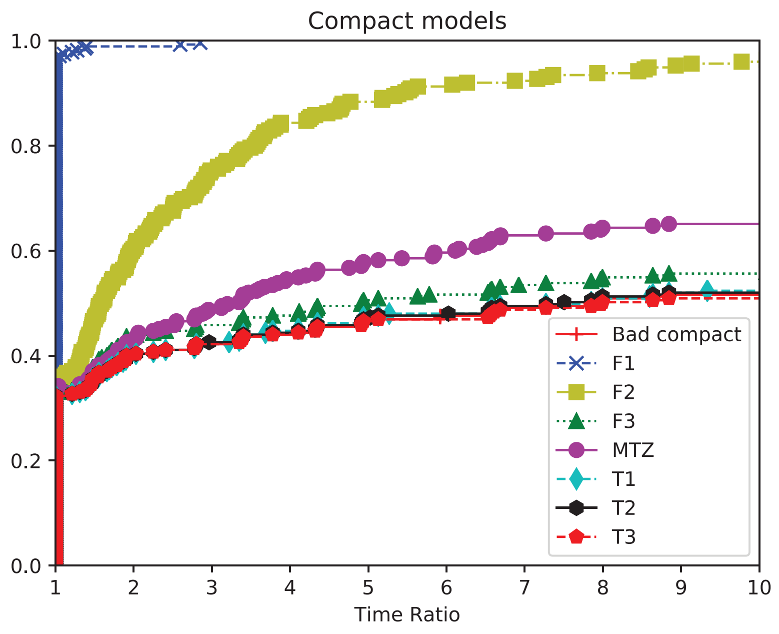
Compact models Amongst the compact models, *Flow 1* (details at section 2.3) is starkly the best performing one. Amongst the other models, it is important to note that most of them failed to even reach optimum before the time limit because the memory needed to store all the variables was not enough. Furthermore, all methods failed to reach optimality within the time limit in a number of instances that were too big. This is not an issue since *performance profiling* allows for a clear understanding of what happened during the runs and it was necessary to compare these methods to more the more powerful exact methods.

In fig. 7.1 we can also see a detail of the performance profiling graph. This detailed view clearly shows how F1 is capable of reaching the optimum before any other method in almost all the instances.

Exact methods When comparing the exact methods, we decided to include F1, which outperformed every other compact method, to have a point of comparison.

Here the methods have a more similar performance. The lazy callback method outperforms the others in most of the instances, however it does not do so consistently. In a small percentage of instances it takes a really long time to finish and is outperformed by the loop method. This means that if we were to look at the percentage of instances solved quickly, lazy callback is the best method, but if we were to ask which method is capable of solving most instances within 12 times the time it takes the fastest method to find the optimum, the loop method would be the choice. Which means that loop is slightly more consistent than lazy callback in being fairly quick.

An important observation on the user-cut callback method. The bad performance is most likely due to the hyper-parameter which regulates the probability of computing the max-flow on a given node. This was set a priori without knowledge on the influence it would have on the performance. It is now clear that it results in a slow-down compared to the



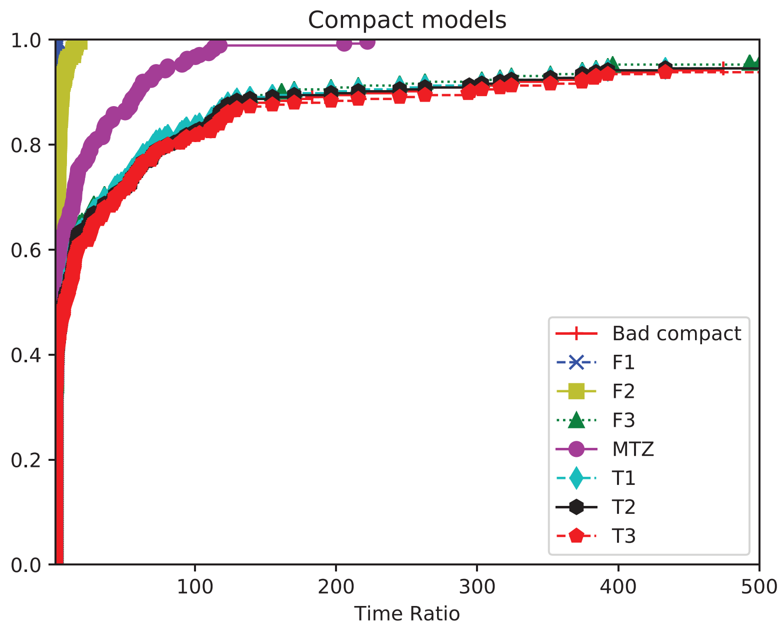


Figure 7.1: On top: detailed view of performance profiling of compact models. On the bottom: full view of the performance profiling of compact models.

lazy callback method. Which means that the cuts that are added are not enough to balance the cost of performing the max-flow on the nodes.

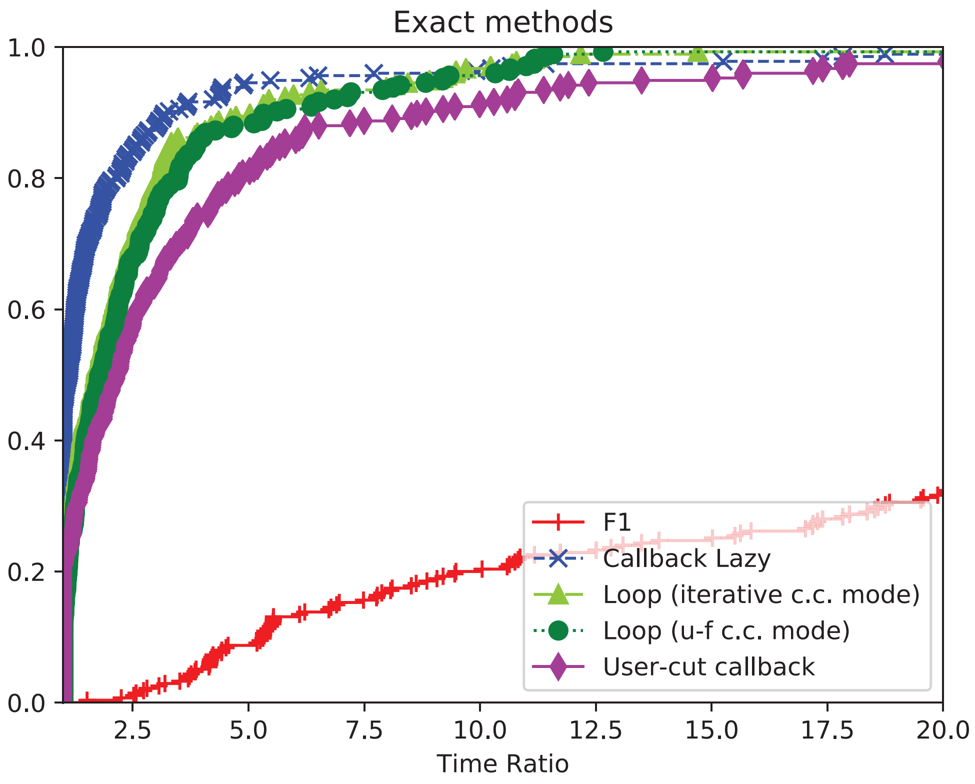


Figure 7.2: Performance profiling of exact methods.

Metaheuristics Among the metaheuristics, VNS consistently outperforms all the other. One important thing to note is that the failure of Simulated Annealing is most likely due to the fact the GRASP uses a local search while Simulated Annealing explores the space with random moves.

Matheuristics Between hard-fixing and local branching, the one which consistently performs best is local branching.

Metaheuristics and metaheuristics mixed with exact methods In Figure 7.6 we can see a comparison between the different mixed metaheuristics with lazy callback. The choice of lazy callback was driven by the results obtained in the comparison between the exact methods. An important thing to note is that here, within the same time limit, the methods often reach optimality. The best performing one is VNS.

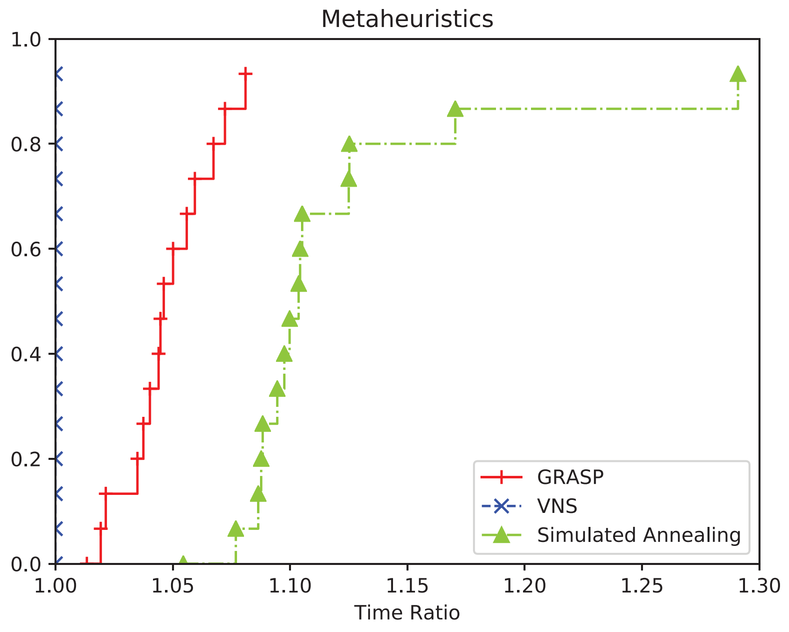


Figure 7.3: Performance profiling of metaheuristics methods.

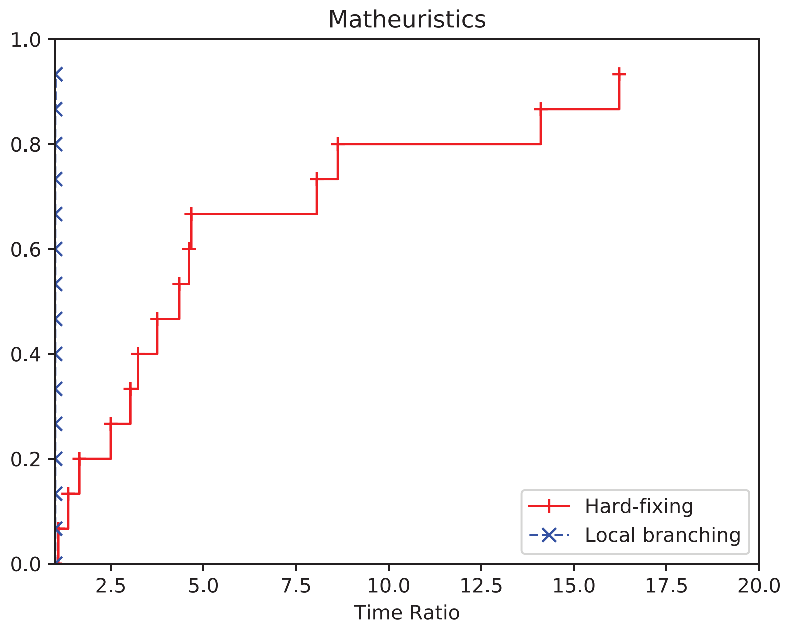
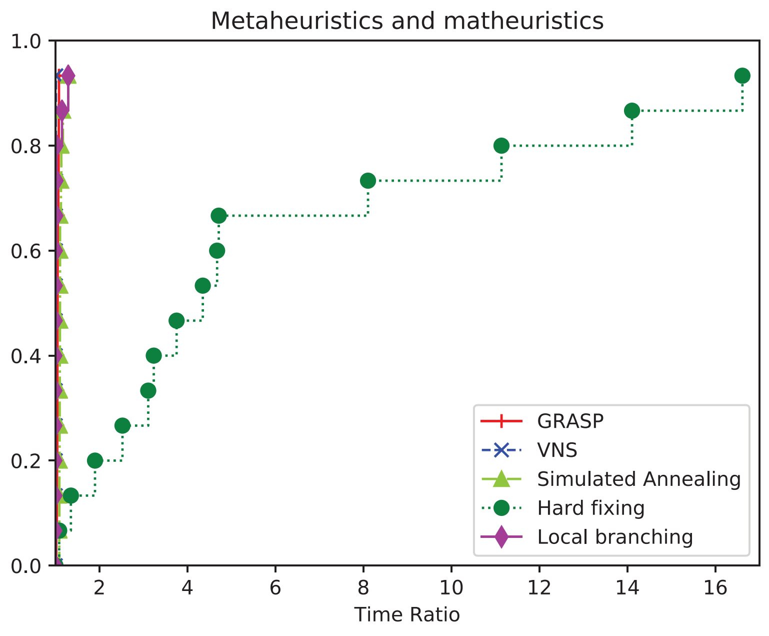


Figure 7.4: Performance profiling of matheuristics.



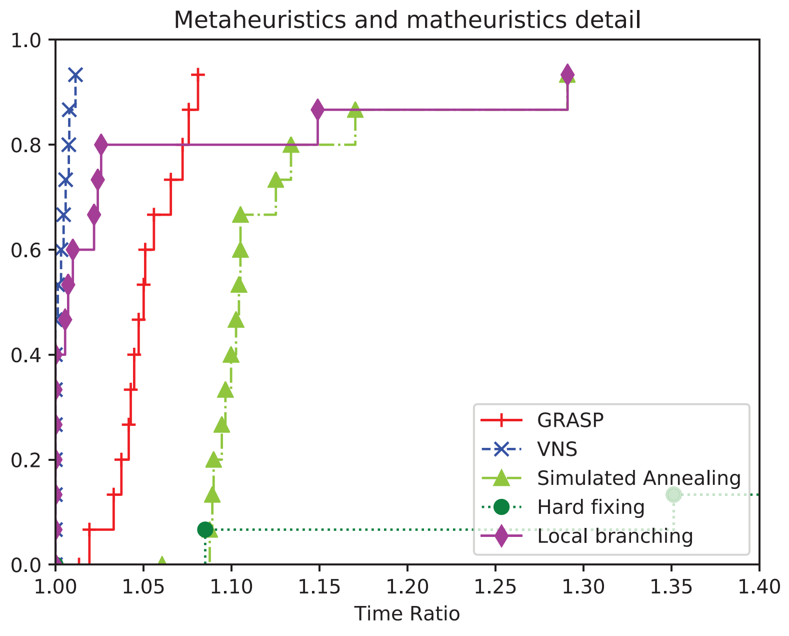
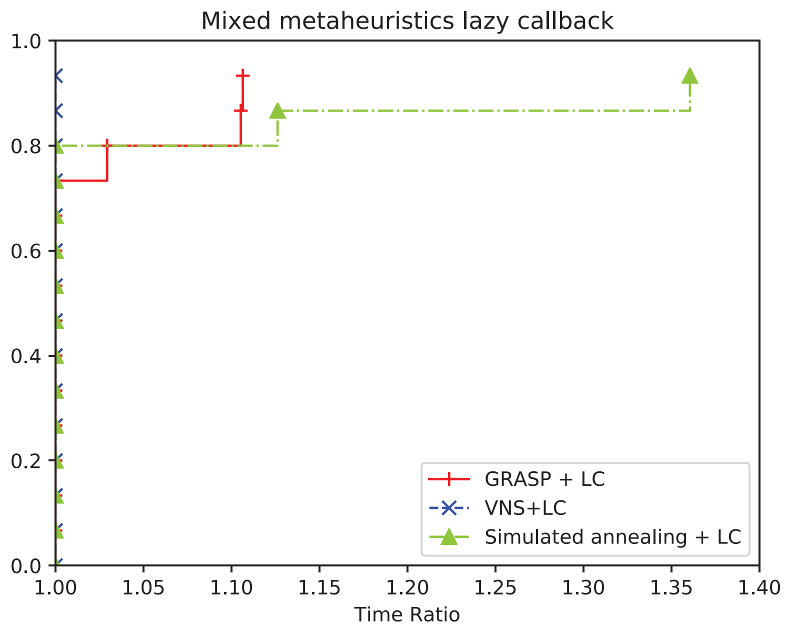


Figure 7.5: Performance profiling of matheuristics and metaheuristics compared. On the bottom: detailed view.

When looking at fig. 7.6, we can see that, although the use of an exact method after a short run of the metaheuristic often leads to optimality, in those cases where this fail, the result is far from the optimum. This is indicative of the fact that in order to have consistently good results in all instances, it is best to use a well built metaheuristic instead of trying to achieve the optimum with an application of both. This effect is particularly pronounced where the metaheuristic is not good.



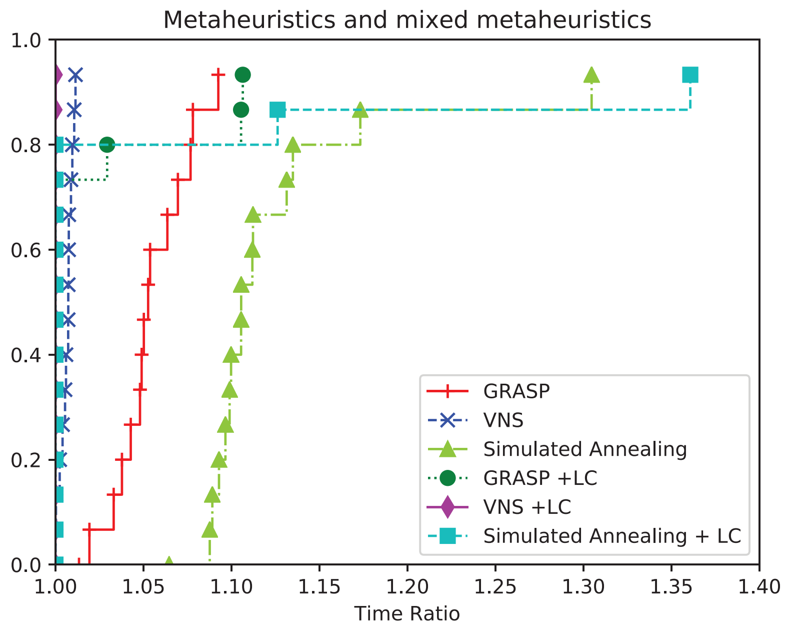


Figure 7.6: On top: performance profiling of metaheuristics mixed with exact methods. On the bottom: performance profiling of metaheuristics mixed with exact methods and metaheuristics.

A Appendix

Notes. We used Ubuntu 14.04.2 LTS and Ubuntu 18.04.02 LTS. All commands and software used refers to commands and software that runs on these distributions of Linux.

A.1 Running Gurobi on the Blade Computing Cluster

When experimenting with models for solving the TSP, we made use of the Blade Computing Cluster at the Department of Information Engineering. It uses the Sun Grid Engine to manage the queue. This is a queueing system that allows one to run a job according to the requirements specified (it reads the requirements and then queues it based on the priority given by the requirements themselves).

Basic commands and file transfer In order to connect to the computing cluster one needs to use SSH. This is something that is provided with every major Linux distribution:

```
$ ssh username@login.dei.unipd.it
```

Running this command will then prompt the input of the password and allow us to start an SSH communication with the server. Here it is possible to access the space on the server's machine and issue commands like the ones required to queue jobs or compile the code.

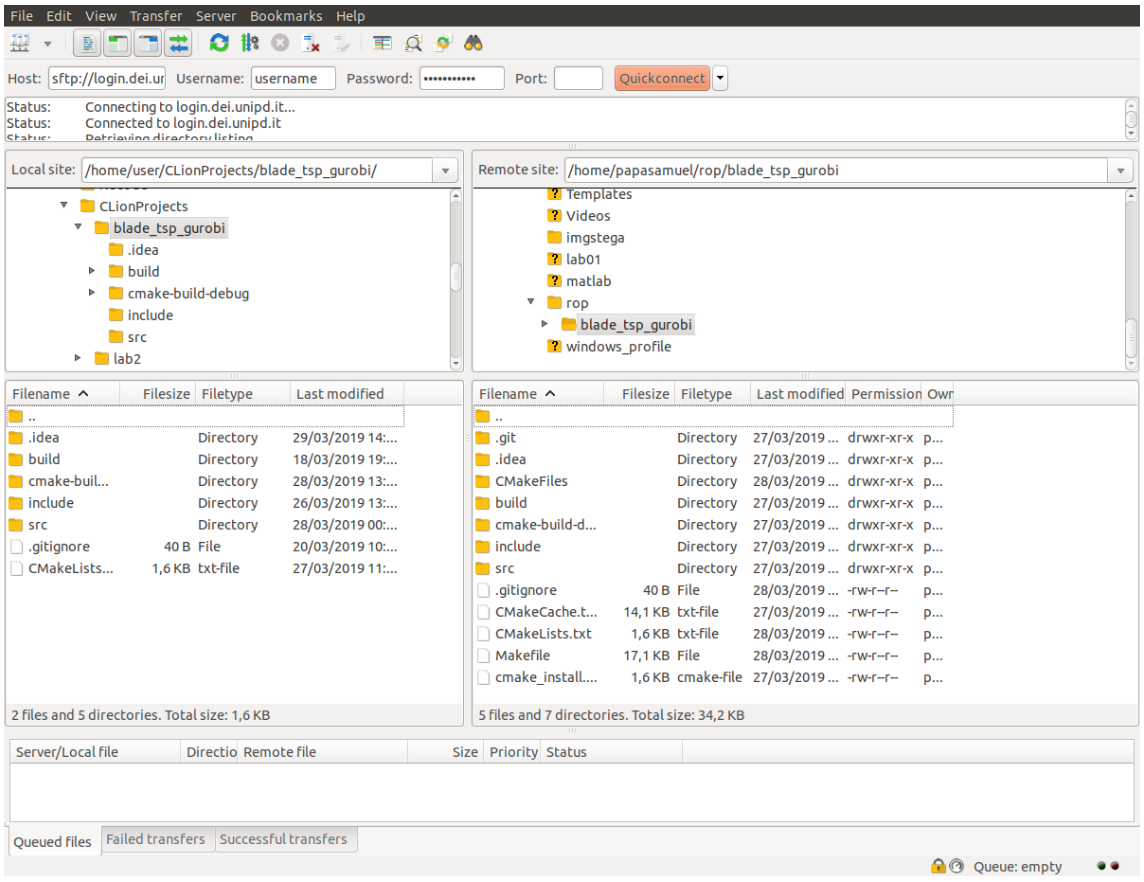
To transfer files from the local machine to the cluster there are a number of alternatives but we used mainly two of them. The first one is the `scp` command, which can be used to transfer files between any two hosts via SSH, but we use it to upload files from the local machine. To achieve this, the basic syntax of the command is:

```
$ scp [options] source_dir/source_filename username@login.dei
    .unipd.it_host:directory/filename
```

An alternative is using FileZilla which provides an easy-to-use GUI, as shown in fig. A.1. Installation is straightforward by using:

```
$ sudo apt-get install filezilla
```

The interface is self explanatory, there are four fields at the top which are filled with the same information used for the SSH command, the only thing to note is that in the `Host` field it should be specified that we are trying to establish an SFTP connection with the server, by writing: `sftp://login.dei.unipd.it`.



Host: sftp://login.dei.ur Username: username Password: •••••••••• Port:

```
Status:      Connecting to login.dei.unipd.it...
Status:      Connected to login.dei.unipd.it
Status:      Retrieving directory listing...
```

Local site: /home/user/CLionProjects/blade_tsp_gurobi/

- CLionProjects
  - blade_tsp_gurobi
    - .idea
    - build
    - cmake-build-debug
    - include
    - src
  - lab2

Remote site: /home/papasamuel/rop/blade_tsp_gurobi

- Templates
- Videos
- imgstega
- lab01
- matlab
- rop
  - blade_tsp_gurobi
- windows_profile

Filename	Filesize	Filetype	Last modified
..			
.idea		Directory	29/03/2019 14:...
build		Directory	18/03/2019 19:...
cmake-buil...		Directory	28/03/2019 13:...
include		Directory	26/03/2019 13:...
src		Directory	28/03/2019 00:...
.gitignore	40 B	File	20/03/2019 10:...
CMakeLists...	1,6 KB	txt-file	27/03/2019 11:...

2 files and 5 directories. Total size: 1,6 KB

Filename	Filesize	Filetype	Last modified	Permission	Owr
..					
.git		Directory	27/03/2019 ...	drwxr-xr-x	p...
.idea		Directory	27/03/2019 ...	drwxr-xr-x	p...
CMakeFiles		Directory	28/03/2019 ...	drwxr-xr-x	p...
build		Directory	27/03/2019 ...	drwxr-xr-x	p...
cmake-build-d...		Directory	27/03/2019 ...	drwxr-xr-x	p...
include		Directory	27/03/2019 ...	drwxr-xr-x	p...
src		Directory	27/03/2019 ...	drwxr-xr-x	p...
.gitignore	40 B	File	28/03/2019 ...	-rw-r--r--	p...
CMakeCache.t...	14,1 KB	txt-file	27/03/2019 ...	-rw-r--r--	p...
CMakeLists.txt	1,6 KB	txt-file	28/03/2019 ...	-rw-r--r--	p...
Makefile	17,1 KB	File	28/03/2019 ...	-rw-r--r--	p...
cmake_install....	1,6 KB	cmake-file	27/03/2019 ...	-rw-r--r--	p...

5 files and 7 directories. Total size: 34,2 KB

Server/Local file	Directio	Remote file	Size	Priority	Status

Figure A.1: Interface of FileZilla.

Gurobi's license The license is available on the cluster. However, Gurobi cannot access it while running, because the environment variable is not set in the machine where the code is run. The Sun Grid Engine used by the Blade Computing Cluster allows to specify environment variables to be used. The following is a typical command used to run an batch job in parallel and setting the environment variable:

```
$ qsub -v GRB_LICENSE_FILE=/location/of/gurobi/license.lic -b
    y -cwd -pe parallel N ./executable [arguments for
   executable]
```

A brief description of the command:

- `-v ENVVAR=value` : sets the environment variable
- `-b y` : specifies that it can run a binary file
- `-cwd` : tells the system to use the current working directory
- `-pe parallel N` : tells the cluster to use N processors/threads

- `./executable ...` : specifies the executable file to run and its arguments.

There also exist a workaround if one wants to avoid writing the environment variable in the command. The environment variable can be set directly using the `setenv()` function in C. The syntax is the following:

```
int setenv(const char *name, const char *value, int overwrite
    );
```

Where `name` is the name of the environment variable, which is `GRB_LICENSE_FILE` in the case of the variable used to tell Gurobi where the license file is, `value` is the actual value of the variable, which in this case will be the path to the license file and finally `overwrite` is a boolean that specifies whether or not the environment variable should be overwritten.

A.2 Union-Find algorithm

Some applications involve grouping n distinct elements into a collection of disjoint sets. This problem arises in the computation of the TSP with lazy callback, loop and user-cut callback methods, described in section 3.2, section 3.1 and section 3.3 respectively. In these methods it is required to find the connected components generated during the computation of the optimal solution of the problem.
The union-find algorithm is based on a disjoint-set data structure (also called union-find data structure or merge-find set) that maintains a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets. Where each set is identified by a representative, which is some member of the set [9].
The algorithm has the following operations:

1. **Make Set** create a new set with only one member.
2. **Find(x)** find the representative of the set containing x.
3. **Union(x, y)** unite the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets.

At the end of this algorithm a forest of disjoint sets (connected components in the TSP) is created.

Make Set This method creates a new set with only one element. This element has unique id, has rank or size (depends on the union operation) equal to 0 or 1 respectively and has as parent itself. The last condition indicates that the node is the representative member of its own set. Algorithm 11 shows the Make Set operation.

Algorithm 11: Make Set

```
Input: Element $x$.
begin
    if $x$ not present then
        Add $x$ with:
        $x.parent \longleftarrow x$;
        $x.rank \longleftarrow 0$;
        $x.size \longleftarrow 1$;
```

Find This operation follows the chain of parents from a node x up to the tree until the root (representative) node is found. The Find method can be implemented in three ways: via path compression, path halving and path splitting.

The *path compression* makes each node on the *find* path point directly to the root. This is possible, since each element visited on the way to a root is part of the same set. This results in a flatter tree, which speeds up future operations not only on these elements, but also on those referencing them. Algorithm 12 shows the recursive version of the find method with path compression. In fig. A.2 there is an example of the tree before and after path compression.

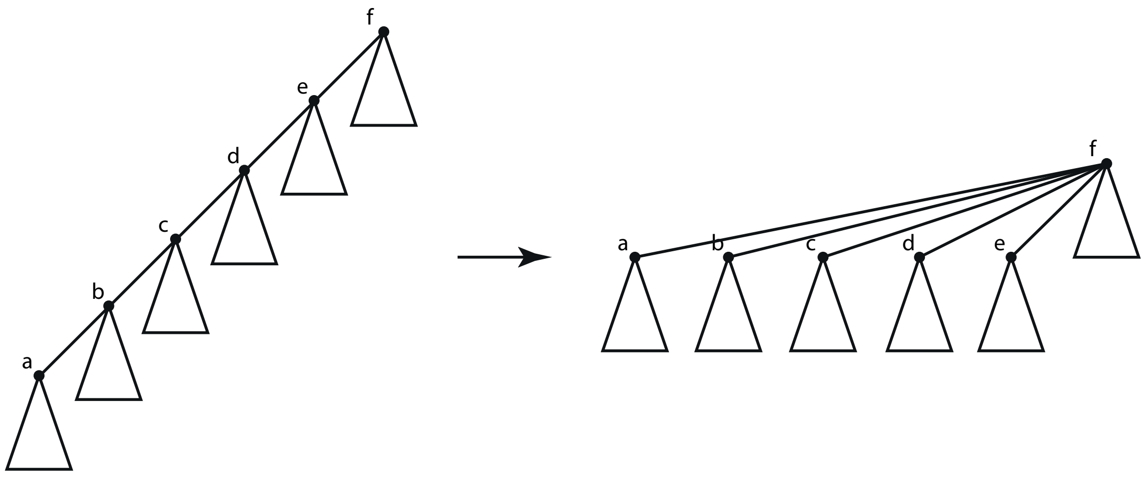


Figure A.2: Path compression.

Algorithm 12: Find with path compression

```
Input: Element $x$.
Output: The representative of the set containing $x$.
begin
    if $x \neq x.parent$ then
        $x.parent \longleftarrow find(x.parent)$;
    return $x.parent$;
```

The *path halving*, designed by R. van der Weide and J. Van Leeuwen [38], is a one-pass algorithm. This variant makes every other node along the *find* path (except the last and the next-to-last) point the node which is two nodes after itself. Halving keeps the nodes on the *find* path together while it halves the length of the find, so that later finds will produce more compression. Algorithm 13 shows the find method with path halving. In fig. A.3 there is an example of the tree before and after path halving.

The *path splitting*, designed by R. van der Weide and J. Van Leeuwen [38], is a one-pass algorithm like path halving. This variant makes every other node along the find path (except the last and the next-to-last) point the node two nodes after itself. Splitting breaks a find path into two paths, each about half as long as the original. Algorithm 14 shows the find method with path splitting. In fig. A.4 there is an example of the tree before and after path splitting.

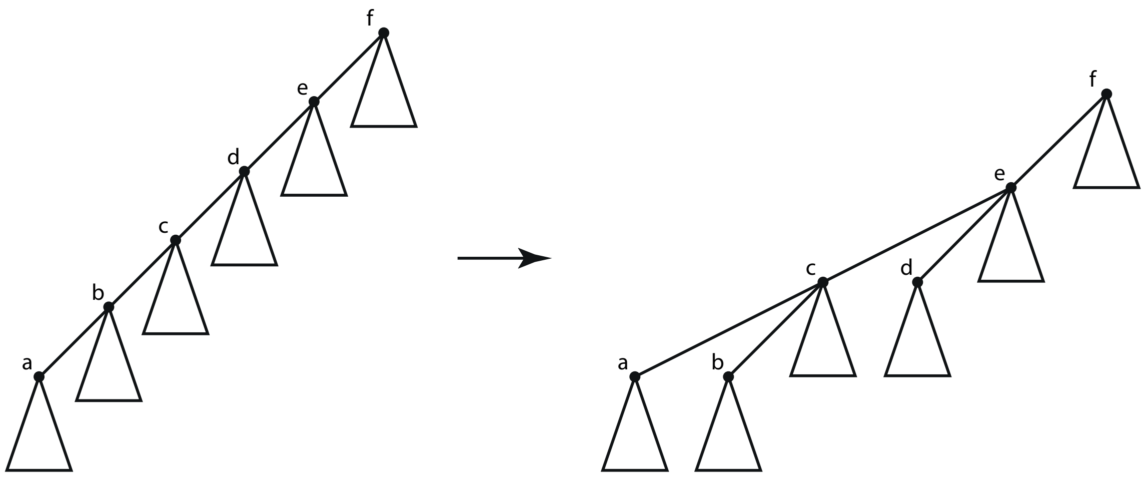


Figure A.3: Path halving.

Algorithm 13: Find with path halving

```
Input: Element $x$.
Output: The representative of the set containing $x$.
begin
    while $x \neq x.parent$ do
        $x.parent \longleftarrow (x.parent).parent$;
        $x \longleftarrow x.parent$;
```

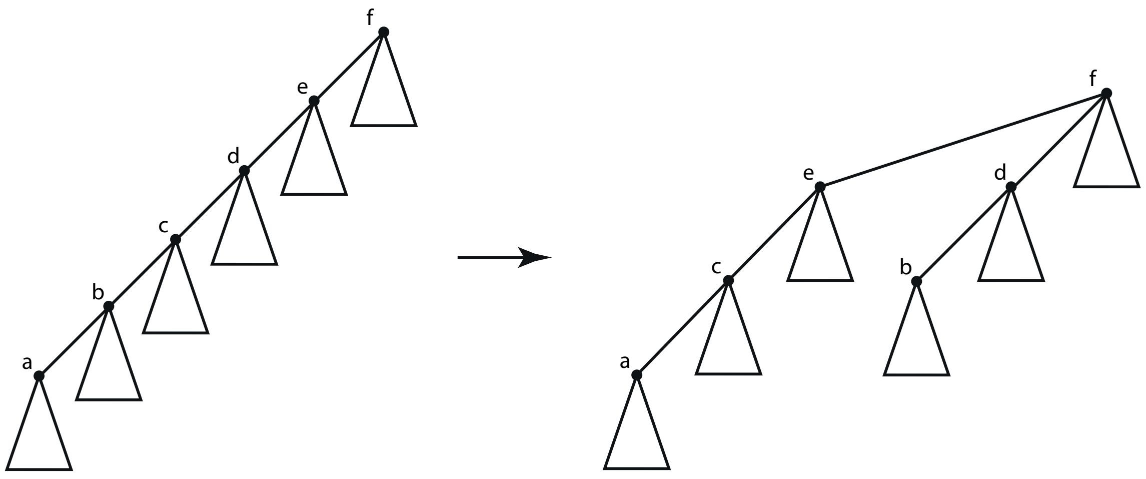


Figure A.4: Path splitting.

Algorithm 14: Find with path splitting

```
Input: Element $x$.
Output: The representative of the set containing $x$.
begin
    while $x \neq x.parent$ do
        $next \longleftarrow x.parent$;
        $x.parent \longleftarrow next.parent$;
        $x \longleftarrow next$;
    return $x$;
```

Union This operation merges two trees connecting the two roots and choosing one of them as the root of the new tree. The union procedure requires halving the roots of the trees where the nodes x and y belong. If the two roots are distinct then one root of the tree becomes the child of the other. This can be done naively by always adding the root of y to x but the height of the tree can grow as $O(n)$, with n the number of leaf nodes. To prevent this, two methods are designed: the *union by size* and *union by rank*.

Union by size was proposed by B. A. Galler and M.J. Fischer [17]. In the union by size, when a union operation occurs the root of the smaller tree points to the root of the larger tree. In the event of a tie, the root of the second tree becomes the child of the root of the first tree. Algorithm 15 shows the union by size method. With the union by size, no *find* path has length exceeding $\log_2 n$ [36].

Algorithm 15: Union by size

```
Input: The representative of the set containing $x$ ($x_{root}$) and the representative of
       the set containing $y$ ($y_{root}$).
begin
    if $x_{root} = x_{root}$ then
        return;
    if $x_{root}.size \geq y_{root}.size$ then
        $y_{root}.parent \longleftarrow x_{root}$;
        $x_{root}.size \longleftarrow x_{root}.size + y_{root}.size$;
    else
        $x_{root}.parent \longleftarrow y_{root}$;
        $y_{root}.size \longleftarrow y_{root}.size + x_{root}.size$;
```

In the *union by rank* when a union operation occurs the root of the lower ranked tree points to the root of the larger one. In the event of a tie, the root of the second tree become the child of the root of the first tree and only in this case the rank of the first tree is incremented by one. Algorithm 16 shows the union by rank method. This method has the same effect in term of path length as the union by size.

Algorithm 16: Union by rank

```
Input: The representative of the set containing $x$ ($x_{root}$) and the representative of
       the set containing $y$ ($y_{root}$).
begin
    if $x_{root} = x_{root}$ then
        return;
    if $x_{root}.rank > y_{root}.rank$ then
        $y_{root}.parent \longleftarrow x_{root}$;
    if $x_{root}.rank < y_{root}.rank$ then
        $x_{root}.parent \longleftarrow y_{root}$;
    if $x_{root}.rank = y_{root}.rank$ then
        $x_{root}.rank \longleftarrow x_{root}.rank + 1$;
```

The computational time of the union-find algorithm changes with the method used. In fact, the union-find algorithm, without the two union variants (rank or size), has complexity $O(n)$ [37] due to the find operation and naive union method. Using path compression alone gives a running time of $\Theta(n + f \cdot (1 + \log_{2+\frac{f}{n}} n))$ [9], for a sequence of n elements and f find operations. The use of union by rank or union by size gives a running time of $O(m \log n)$ [9] for m operations with initially n sets having one element each. Instead, using one of the find methods and one of the union methods explained above gives an amortized time $O(m\alpha(m, n))$ [36] (where α is the inverse Ackermann function [39]).

A.3 Iterative method for finding the connected components

The iterative method, based on Kruskal's algorithm, searches for the connected components by adding iteratively subset of nodes having edges connected with the nodes in that component. Algorithm 17 illustrates the iterative algorithm for finding the connected components. The algorithm first creates connected components with only one element within it. Afterwards, two connected components are joined when there is an edge that connects two nodes, one in each component. The steps are repeated until the edges are all visited.

Algorithm 17: Iterative finder for connected components

```
Input: The instance $(V, E)$ of the problem.
Output: The connected components.
begin
    /* One element for each connected component                          */
    foreach $v \in V$ do
        $comp[v] \longleftarrow v$;
    /* Join the nodes with edges in the same connected component         */
    foreach $[i, j] \in E$ do
        if $comp[i] \neq comp[j]$ then
            $C_1 \longleftarrow comp[i]$;
            $C_2 \longleftarrow comp[j]$;
            /* Update connected component                                */
            foreach $v \in V$ do
                if $comp[v] = C_2$ then
                    $comp[v] \longleftarrow C_1$;
```

The computation time of the iterative algorithm is $O(n^2)$.

A.4 *k*-exchange neighborhood

The k-exchange neighborhood, or k-opt, is one of the most widely used types of neighborhood relations. In the k-exchange neighborhood two instance are neighbours if and only if they differ for at most k components. Algorithm 18 illustrates the k-opt procedure and fig. A.5 illustrates a possible swap moves for the 2-opt, 3-opt and 4-opt.

Algorithm 18: k-exchange neighborhood

```
Input: Feasible solution of a problem $S^0$.
Output: The best neighbour.
begin
    $S^t \longleftarrow S^0$;
    repeat
        $S^t_i \longleftarrow$ remove $k$ edges from current tour $S^t$, making it incomplete;
        Build all feasible solutions (complete tours) from $S^t_i$;
        $S^* \longleftarrow$ select the best tour among these tours;
        if $f(S^*) < f(S^t)$ then
            $S^t \longleftarrow S^*$;
        else
            failure;
    until failure;
```

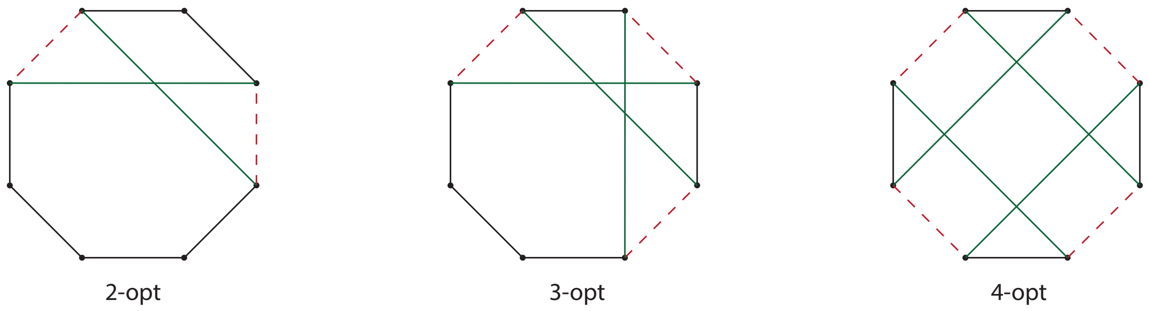


Figure A.5: Possible swap moves for the 2-opt, 3-opt and 4-opt.

2-opt The *2-opt* algorithm has been thoroughly studied in the literature [3]. Some simple considerations can be made about the implementation.
Particular care needs to be put in the data structure used to represent the cycle of nodes. When an array is used, it becomes clear that each move which involves edges that are far away, requires the middle edges to be swapped. This means changing the order of $O(n)$

nodes in the cycle (where n is the number of nodes). Moreover, the weights for all the edges should be computed beforehand and available in $O(1)$.

When selecting which pair of edges to exchange, once a pair has been found, the rest of the tour should be examined, without starting from the beginning, as it is more likely that some pair of edges that has yet to be considered is capable of improving the cycle. Starting from the beginning of the cycle, all pairs have already been checked, so only neighbours of those that have been swapped can potentially improve the cycle.

If we allow the loss of local optimality we can make the algorithm much faster. A technique that we explored was using ***don't look bits***. Here the observation is that, if we previously failed to find an improving exchange for a given vertex, it is unlikely that we will find it in a later iteration and is marked as *non-improving*. When the vertex that was previously marked as *non-improving* is involved in an exchange, then it is eligible to being checked again. However, in our array structure implementation, the bottleneck became the structure itself and the trade-off between speed and non-local-optimality resulted in no change in performance.

A.5 Result tables

In the section is reported all the test results obtained from the models and the heuristic algorithm implemented. The results are the geometric mean of the time, in seconds, for found the optimal solution for the compact and exact models.

The geometric mean is defined as the nth root of the product of n numbers, i.e., for a set of number $x_1, x_2, \dots, x_n$ the geometric mean is defined as

$$\left(\prod_{i=1}^{n} x_i\right)^{\frac{1}{n}} = \sqrt[n]{x_1 \cdot x_2 \cdot \dots \cdot x_n}. \tag{A.1}$$

Table 1: Geometric mean times of the better models.

Begin of Table							
Instance	Models						
	F1	F2	Lazy callback	Loop 1	Loop 2	MTZ	User-cut callback
a280	2240.34847	3242.297904	10.01346256	15.57731186	14.80062779	seg fault	27.45682413
ali535	seg fault	seg fault	1754.608237	275.8987086	424.0095553	seg fault	2210.375421
att48	4.648554229	17.31516279	0.636681025	0.808215606	0.930517592	152.5533266	0.643625431
att532	seg fault	seg fault	2243.986461	884.6079888	1076.282083	seg fault	1457.074721
berlin52	1.777768059	2.604254688	0.259444221	0.252857364	0.42268288	5.450201618	0.525033505
bier127	104.1305501	427.5969602	1.336982289	1.840950736	1.190828815	timelimit	1.654626363
burma14	0.453766053	0.718520573	0.680615654	0.376162487	0.385759383	0.754765881	0.251555456
ch130	128.4991324	449.4162098	1.364873085	2.168086661	2.40183	timelimit	3.249006313
ch150	245.8429041	431.0316139	7.584660523	5.373910962	6.03962399	timelimit	6.773158699
d198	2295.848577	timelimit	23.73298709	52.68781815	57.21235282	seg fault	44.87076587
d493	seg fault	seg fault	1104.284989	1012.256147	1068.098841	seg fault	2933.063456
d657	seg fault	seg fault	2667.686988	1110.454638	1281.974439	seg fault	seg fault
eil51	6.004959094	7.176574271	0.286824528	0.747381843	0.452358646	9.992139659	0.479564997
eil76	9.594415307	12.4022787	0.393977335	0.621842294	0.802673449	27.44775958	0.555309693
eil101	32.61086361	40.32751264	0.971530533	1.570389298	1.302333705	103.9388826	0.950030723
fl417	seg fault	seg fault	423.9462122	428.7328067	581.292426	seg fault	1327.169747
gil262	timelimit	timelimit	39.66981962	29.70119173	30.0732627	seg fault	120.2983251
gr202	769.4560334	1402.539498	5.726215907	14.26020273	15.75461753	seg fault	4.793856258
gr229	1874.485372	timelimit	35.10489403	51.36820435	56.62988882	seg fault	106.8209372
gr431	seg fault	seg fault	657.5797855	738.5740225	711.1687037	seg fault	1145.455644
kroA100	36.4556841	118.0797965	1.274936133	2.519702828	2.44572	timelimit	5.555894565
kroA150	267.1898383	470.2291587	5.152611174	10.6608224	11.1286808	seg fault	6.283681784
kroA200	752.1361025	2187.466356	41.8882522	37.72972204	41.21644989	seg fault	56.88168774
kroB100	48.15195525	245.3384166	1.867025609	4.614008113	5.1136572	timelimit	6.107100357
kroB150	396.6236548	670.1441225	9.376343694	19.31350625	20.41214747	seg fault	20.47579783
kroB200	515.2461494	1205.740559	9.880876961	9.826770783	10.17260036	seg fault	14.20110982

Continuation of Table 1

Instance	Models						
	F1	F2	Lazy callback	Loop 1	Loop 2	MTZ	User-cut callback
kroC100	55.94599418	134.798036	1.182507964	2.112949288	2.464415473	timelimit	2.816127609
kroD100	43.02541767	234.0945749	0.955617228	2.275057148	2.454568466	timelimit	2.121331203
kroE100	65.69249291	170.1187199	1.952557343	2.734317348	2.1908995	3195.205213	4.37676286
lin105	44.16246972	229.3560374	0.433719059	1.322214801	1.498630767	3285.060155	0.804281941
lin318	timelimit	timelimit	94.44380461	52.60880539	55.34291439	seg fault	112.5314318
p654	seg fault	seg fault	2858.577113	timelimit	timelimit	seg fault	443.4724555
pcb442	seg fault	seg fault	115.8755174	442.1893882	488.587694	seg fault	478.230303
pr76	91.19857186	177.5091699	5.0312516	3.699024648	4.36899225	479.6469545	8.404009411
pr107	428.0564528	timelimit	0.551821161	0.992356724	0.849057658	timelimit	0.657454076
pr124	224.76967	976.3106485	4.595385446	14.37983094	14.98038133	timelimit	3.455188058
pr136	195.8365459	1492.863217	1.100702188	4.20258341	4.538720351	seg fault	3.162109056
pr144	256.8329244	3494.166965	5.768208076	12.26388602	13.24389064	seg fault	5.766920032
pr152	1379.28652	timelimit	5.940186096	5.2410348	5.065054318	seg fault	9.13744657
pr226	2464.842023	timelimit	32.61191086	102.7275904	101.5100796	timelimit	12.71353159
pr264	timelimit	timelimit	32.08630508	123.6418717	136.0425595	seg fault	13.05766777
pr299	timelimit	timelimit	93.76632052	135.305022	165.2194118	seg fault	376.9583098
pr439	seg fault	seg fault	491.9920165	480.2905198	581.2405169	seg fault	2287.989568
rat99	24.57863254	38.03406546	0.628051712	1.644796748	1.643809883	100.0929528	0.830295392
rat195	1294.549863	3269.325653	13.99368616	39.87517491	42.34459109	seg fault	48.65498014
rat575	seg fault	seg fault	1233.609898	508.0970698	494.2525137	seg fault	seg fault
rat783	seg fault	seg fault	2537.990873	190.5182229	206.2199417	seg fault	seg fault
rd100	31.0616388	162.3308484	1.107428075	1.674184867	1.83143859	745.1177245	1.221396091
rd400	timelimit	seg fault	159.5560942	202.3225984	216.1870601	seg fault	1298.347657
st70	13.35297037	41.96559655	0.494087392	0.913681267	1.034291533	312.8076312	0.687194251
u159	72.18898716	182.7259748	1.195385023	3.21695882	3.437814111	timelimit	1.818635484
u574	seg fault	seg fault	499.7291937	184.9823661	261.6905626	timelimit	seg fault
u724	seg fault	seg fault	2139.514829	1017.746193	811.9906905	seg fault	seg fault
ulysses16	0.826004263	1.191384027	0.119107779	0.296830188	0.65856234	2.027680727	0.926623774

Continuation of Table 1							
Instance	Models						
	F1	F2	Lazy callback	Loop 1	Loop 2	MTZ	User-cut callback
ulysses22	0.818401745	1.842499235	0.396852789	0.865919877	0.525383941	98.35268169	0.156214691
End of Table							

Table 2: Exercise compact model times.

Begin of Table					
Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
a280	seg fault	seg fault	seg fault	seg fault	seg fault
att48	timelimit	timelimit	timelimit	timelimit	timelimit
att532	seg fault	seg fault	seg fault	seg fault	seg fault
berlin52	timelimit	timelimit	timelimit	timelimit	timelimit
bier127	seg fault	seg fault	seg fault	seg fault	seg fault
burma14	4.18936	3.39765	4.70768	3.92699	3.68589
ch130	seg fault	seg fault	seg fault	seg fault	seg fault
ch150	seg fault	seg fault	seg fault	seg fault	seg fault
d198	seg fault	seg fault	seg fault	seg fault	seg fault
d493	seg fault	seg fault	seg fault	seg fault	seg fault
d657	seg fault	seg fault	seg fault	seg fault	seg fault
eil51	timelimit	timelimit	timelimit	timelimit	timelimit
eil76	timelimit	timelimit	timelimit	timelimit	timelimit
eil101	seg fault	seg fault	seg fault	seg fault	seg fault
fl417	seg fault	seg fault	seg fault	seg fault	seg fault
gil262	seg fault	seg fault	seg fault	seg fault	seg fault
gr202	seg fault	seg fault	seg fault	seg fault	seg fault
gr229	seg fault	seg fault	seg fault	seg fault	seg fault
gr431	seg fault	seg fault	seg fault	seg fault	seg fault
kroA100	seg fault	seg fault	seg fault	seg fault	seg fault
kroA150	seg fault	seg fault	seg fault	seg fault	seg fault
kroA200	seg fault	seg fault	seg fault	seg fault	seg fault
kroB100	seg fault	seg fault	seg fault	seg fault	seg fault
kroB150	seg fault	seg fault	seg fault	seg fault	seg fault
kroB200	seg fault	seg fault	seg fault	seg fault	seg fault
kroC100	seg fault	seg fault	seg fault	seg fault	seg fault
kroD100	seg fault	seg fault	seg fault	seg fault	seg fault
kroE100	seg fault	seg fault	seg fault	seg fault	seg fault
lin105	seg fault	seg fault	seg fault	seg fault	seg fault
lin318	seg fault	seg fault	seg fault	seg fault	seg fault
p654	seg fault	seg fault	seg fault	seg fault	seg fault
pcb442	seg fault	seg fault	seg fault	seg fault	seg fault
pr76	timelimit	timelimit	timelimit	timelimit	timelimit
pr107	seg fault	seg fault	seg fault	seg fault	seg fault
pr124	seg fault	seg fault	seg fault	seg fault	seg fault
pr136	seg fault	seg fault	seg fault	seg fault	seg fault
pr144	seg fault	seg fault	seg fault	seg fault	seg fault
pr152	seg fault	seg fault	seg fault	seg fault	seg fault

Continuation of Table 2					
Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
pr226	seg fault	seg fault	seg fault	seg fault	seg fault
pr264	seg fault	seg fault	seg fault	seg fault	seg fault
pr299	seg fault	seg fault	seg fault	seg fault	seg fault
pr439	seg fault	seg fault	seg fault	seg fault	seg fault
rat99	seg fault	seg fault	seg fault	seg fault	seg fault
rat195	seg fault	seg fault	seg fault	seg fault	seg fault
rat575	seg fault	seg fault	seg fault	seg fault	seg fault
rat783	seg fault	seg fault	seg fault	seg fault	seg fault
rd100	seg fault	seg fault	seg fault	seg fault	seg fault
rd400	seg fault	seg fault	seg fault	seg fault	seg fault
st70	timelimit	timelimit	timelimit	timelimit	killed
u159	seg fault	seg fault	seg fault	seg fault	seg fault
u574	seg fault	seg fault	seg fault	seg fault	seg fault
u724	seg fault	seg fault	seg fault	seg fault	seg fault
ulysses16	22.2032	17.2032	22.2796	18.7381	15.5786
ulysses22	169.405	174.842	171.773	174.05	175.084
End of Table					

Table 3: Single commodity flow model times.

Begin of Table					
Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
a280	2091.96	2883.88	2133.73	2171.58	2018.98
ali535	seg fault	seg fault	seg fault	seg fault	seg fault
att48	3.85461	8.30687	4.48028	4.38025	3.45436
att532	seg fault	seg fault	seg fault	seg fault	seg fault
berlin52	1.64586	2.00231	1.61998	1.28046	2.59762
bier127	100.673	134.628	102.367	103.756	85.049
burma14	0.17336	0.573806	0.795364	0.685723	0.354594
ch130	137.794	129.727	139.363	139.241	101.001
ch150	210.851	354.443	213.524	216.762	259.619
d198	2324.51	2556.23	2314.44	2318.58	2000.4
d493	seg fault	seg fault	seg fault	seg fault	seg fault
d657	seg fault	seg fault	seg fault	seg fault	seg fault
eil51	5.35768	6.24675	5.39762	5.15391	8.38649
eil76	9.1702	9.61426	9.37023	9.61918	10.2308
eil101	29.2131	40.3216	30.5824	30.2522	33.8429

Continuation of Table 3

Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
fl417	seg fault	seg fault	seg fault	seg fault	seg fault
gil262	timelimit	timelimit	timelimit	timelimit	timelimit
gr202	827.588	495.121	833.118	829.059	953.019
gr229	1885.88	1773.75	1892.29	1894.02	1930.33
gr431	seg fault	seg fault	seg fault	seg fault	seg fault
kroA100	30.6772	44.6289	31.7371	32.431	45.6948
kroA150	327.811	225.88	318.498	317.728	181.734
kroA200	731.465	702.117	728.857	733.25	876.97
kroB100	57.1104	34.9183	58.9472	61.2811	35.9345
kroB150	458.601	252.496	450.337	451.614	416.772
kroB200	548.967	407.052	551.406	547.374	538.421
kroC100	51.1371	63.3897	52.1436	52.3	61.9994
kroD100	39.6911	38.3512	50.2005	49.7215	38.8062
kroE100	57.372	65.0043	66.7549	65.0375	75.5596
lin105	48.0739	32.4151	60.2902	58.2109	30.7156
lin318	timelimit	timelimit	timelimit	timelimit	timelimit
p654	seg fault	seg fault	seg fault	seg fault	seg fault
pcb442	seg fault	seg fault	seg fault	seg fault	seg fault
pr76	79.9585	89.7885	83.0661	83.0253	127.415
pr107	232.487	267.505	239.223	241.497	seg fault
pr124	278.803	252.912	278.656	277.743	105.126
pr136	146.31	270.612	149.466	148.453	327.881
pr144	253.918	356.738	247.779	249.595	199.485
pr152	1060.54	2745.66	1075.13	1066.98	1494.45
pr226	2435.76	2525.69	2444.89	2450.18	2468.73
pr264	timelimit	timelimit	timelimit	timelimit	timelimit
pr299	timelimit	timelimit	timelimit	timelimit	timelimit
pr439	seg fault	seg fault	seg fault	seg fault	seg fault
rat99	18.334	29.2292	27.7707	25.9168	23.2566
rat195	1285.31	1058.02	1296.72	1290.01	1598.28
rat575	seg fault	seg fault	seg fault	seg fault	seg fault
rat783	seg fault	seg fault	seg fault	seg fault	seg fault
rd100	21.1451	51.7266	28.0003	29.2795	32.2457
rd400	timelimit	timelimit	timelimit	timelimit	timelimit
st70	11.3862	13.6907	16.6928	14.6933	11.1029
u159	66.7998	86.3782	71.2673	71.4314	66.7415
u574	seg fault	seg fault	seg fault	seg fault	seg fault
u724	seg fault	seg fault	seg fault	seg fault	seg fault
ulysses16	0.744125	1.45247	0.698304	0.573928	0.88768
ulysses22	0.357064	0.656254	1.24182	0.951336	1.32624

Continuation of Table 3					
Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
End of Table					

Table 4: Two commodity flow model times.

Begin of Table					
Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
a280	3602.36	2600.75	3602.14	3603.04	2946.79
ali535	seg fault	seg fault	seg fault	seg fault	seg fault
att48	12.8295	23.1013	16.2138	14.9735	21.6311
att532	seg fault	seg fault	seg fault	seg fault	seg fault
berlin52	2.25836	3.96074	2.44291	2.69694	2.03268
bier127	357.347	512.957	368.075	362.436	584.567
burma14	1.05281	0.608634	1.0573	0.838881	0.33697
ch130	411.371	475.149	409.49	414.26	552.922
ch150	496.778	356.812	505.583	493.37	336.495
d198	timelimit	timelimit	timelimit	timelimit	timelimit
d493	seg fault	seg fault	seg fault	seg fault	seg fault
d657	seg fault	seg fault	seg fault	seg fault	seg fault
eil51	7.67745	6.48138	8.21232	7.74566	6.01419
eil76	12.1025	15.6413	12.2253	12.0783	10.4977
eil101	38.9794	64.1094	38.523	32.2293	34.3779
fl417	seg fault	seg fault	seg fault	seg fault	seg fault
gil262	timelimit	timelimit	timelimit	timelimit	timelimit
gr202	1621.4	937.804	1604.44	1604.51	1386.46
gr229	timelimit	timelimit	timelimit	timelimit	timelimit
gr431	seg fault	seg fault	seg fault	seg fault	seg fault
kroA100	146.392	76.8609	144.923	145.17	96.9708
kroA150	363.095	570.16	366.331	359.957	842.183
kroA200	1837.29	3279.33	1828.1	1835.31	2477.62
kroB100	213.191	341.256	211.21	224.832	257.28
kroB150	635.616	632.08	632.843	627.953	846.547
kroB200	892.249	2106.97	899.464	890.456	1692.5
kroC100	115.662	245.915	112.163	112.651	123.839
kroD100	211.645	350.316	211.237	214.827	208.944
kroE100	176.651	207.472	174.54	176.846	125.949
lin105	223.632	238.684	221.75	220.216	243.49
lin318	timelimit	timelimit	timelimit	timelimit	timelimit

Continuation of Table 4					
Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
p654	seg fault	seg fault	seg fault	seg fault	seg fault
pcb442	seg fault	seg fault	seg fault	seg fault	seg fault
pr76	145.301	276.256	149.349	149.803	196.246
pr107	timelimit	timelimit	timelimit	timelimit	timelimit
pr124	817.531	1401.67	797.864	806.343	1203.21
pr136	1255.92	2416.54	1263.16	1266.87	1526.7
pr144	3416.31	timelimit	3433.88	3425.87	timelimit
pr152	timelimit	timelimit	timelimit	timelimit	timelimit
pr226	timelimit	timelimit	timelimit	timelimit	timelimit
pr264	timelimit	timelimit	timelimit	timelimit	timelimit
pr299	timelimit	timelimit	timelimit	timelimit	timelimit
pr439	seg fault	seg fault	seg fault	seg fault	seg fault
rat99	37.4364	40.7433	37.0452	38.3595	36.7205
rat195	timelimit	3104.7	timelimit	timelimit	2578.48
rat575	seg fault	seg fault	seg fault	seg fault	seg fault
rat783	seg fault	seg fault	seg fault	seg fault	seg fault
rd100	153.908	171.866	154.61	151.71	181.678
rd400	seg fault	seg fault	seg fault	seg fault	seg fault
st70	48.5504	43.5747	47.8541	52.4301	24.5211
u159	158.65	196.571	158.903	158.608	259.17
u574	seg fault	seg fault	seg fault	seg fault	seg fault
u724	seg fault	seg fault	seg fault	seg fault	seg fault
ulysses16	1.5056	1.09267	1.53542	0.811089	1.17156
ulysses22	1.51483	1.72774	1.92201	1.83526	2.30007
End of Table					

Table 5: Multi-commodity flow model times.

Begin of Table					
Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
a280	seg fault	seg fault	seg fault	seg fault	seg fault
ali535	seg fault	seg fault	seg fault	seg fault	seg fault
att48	2167.91	664.783	2209.36	2195.57	557.342
att532	seg fault	seg fault	seg fault	seg fault	seg fault
berlin52	killed	killed	killed	killed	804.975
bier127	seg fault	seg fault	seg fault	seg fault	seg fault
burma14	0.276143	0.715869	1.16236	0.937974	0.495703

Continuation of Table 5

Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
ch130	seg fault	seg fault	seg fault	seg fault	seg fault
ch150	seg fault	seg fault	seg fault	seg fault	seg fault
d198	seg fault	seg fault	seg fault	seg fault	seg fault
d493	seg fault	seg fault	seg fault	seg fault	seg fault
d657	seg fault	seg fault	seg fault	seg fault	seg fault
eil51	seg fault	seg fault	seg fault	seg fault	seg fault
eil76	seg fault	seg fault	seg fault	seg fault	seg fault
eil101	seg fault	seg fault	seg fault	seg fault	seg fault
fl417	seg fault	seg fault	seg fault	seg fault	seg fault
gil262	seg fault	seg fault	seg fault	seg fault	seg fault
gr202	seg fault	seg fault	seg fault	seg fault	seg fault
gr229	seg fault	seg fault	seg fault	seg fault	seg fault
gr431	seg fault	seg fault	seg fault	seg fault	seg fault
kroA100	seg fault	seg fault	seg fault	seg fault	seg fault
kroA150	seg fault	seg fault	seg fault	seg fault	seg fault
kroA200	seg fault	seg fault	seg fault	seg fault	seg fault
kroB100	seg fault	seg fault	seg fault	seg fault	seg fault
kroB150	seg fault	seg fault	seg fault	seg fault	seg fault
kroB200	seg fault	seg fault	seg fault	seg fault	seg fault
kroC100	seg fault	seg fault	seg fault	seg fault	seg fault
kroD100	seg fault	seg fault	seg fault	seg fault	seg fault
kroE100	seg fault	seg fault	seg fault	seg fault	seg fault
lin105	seg fault	seg fault	seg fault	seg fault	seg fault
lin318	seg fault	seg fault	seg fault	seg fault	seg fault
p654	seg fault	seg fault	seg fault	seg fault	seg fault
pcb442	seg fault	seg fault	seg fault	seg fault	seg fault
pr76	seg fault	seg fault	seg fault	seg fault	seg fault
pr107	seg fault	seg fault	seg fault	seg fault	seg fault
pr124	seg fault	seg fault	seg fault	seg fault	seg fault
pr136	seg fault	seg fault	seg fault	seg fault	seg fault
pr144	seg fault	seg fault	seg fault	seg fault	seg fault
pr152	seg fault	seg fault	seg fault	seg fault	seg fault
pr226	seg fault	seg fault	seg fault	seg fault	seg fault
pr264	seg fault	seg fault	seg fault	seg fault	seg fault
pr299	seg fault	seg fault	seg fault	seg fault	seg fault
pr439	seg fault	seg fault	seg fault	seg fault	seg fault
rat99	seg fault	seg fault	seg fault	seg fault	seg fault
rat195	seg fault	seg fault	seg fault	seg fault	seg fault
rat575	seg fault	seg fault	seg fault	seg fault	seg fault
rat783	seg fault	seg fault	seg fault	seg fault	seg fault

Continuation of Table 5					
Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
rd100	seg fault	seg fault	seg fault	seg fault	seg fault
rd400	seg fault	seg fault	seg fault	seg fault	seg fault
st70	seg fault	seg fault	seg fault	seg fault	seg fault
u159	seg fault	seg fault	seg fault	seg fault	seg fault
u574	seg fault	seg fault	seg fault	seg fault	seg fault
u724	seg fault	seg fault	seg fault	seg fault	seg fault
ulysses16	0.848088	0.559691	1.19386	0.876533	1.20909
ulysses22	5.30116	4.54353	5.1116	5.35655	7.6642
End of Table					

Table 6: Lazy callback model times.

Begin of Table					
Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
a280	13.7797	4.92874	16.5952	16.2016	5.51322
ali535	3456.79	1168.08	3463.51	3468.13	342.881
att48	0.496338	0.653503	0.787403	0.719939	0.568973
att532	timelimit	1261.42	timelimit	timelimit	966.794
berlin52	0.128325	0.185834	0.234685	0.209799	1.00114
bier127	2.07139	1.34064	1.57816	1.44272	0.675649
burma14	0.638126	0.681334	0.725387	0.703109	0.658644
ch130	1.92229	1.17437	1.10912	1.11615	1.69487
ch150	9.35956	4.15025	11.0228	9.15801	6.40115
d198	33.0561	28.2911	36.5051	35.8846	6.14609
d493	1182.93	1680.32	1193.6	1194.91	579.243
d657	2747.78	1934.96	2751.03	2749.07	3360
eil51	0.989454	0.203604	0.348119	0.274844	0.100713
eil76	1.07448	0.746417	0.326434	0.269873	0.134345
eil101	0.84396	1.35927	0.946222	0.836012	0.953782
fl417	332.628	1256.61	340.076	334.752	287.803
gil262	39.9444	35.6305	43.3193	42.5668	37.4344
gr202	5.54292	5.40719	6.65065	5.87932	5.25336
gr229	27.6838	35.5821	36.6882	37.3073	39.5422
gr431	781.599	481.064	780.137	784.265	534.468
kroA100	1.49205	1.03422	1.84771	0.934958	1.26363
kroA150	5.21295	4.16466	6.35066	6.11781	4.30585
kroA200	53.7576	34.6324	54.58	56.8445	22.3263

Continuation of Table 6

Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
kroB100	1.71568	1.85322	2.02962	1.94503	1.80737
kroB150	8.28329	10.3699	7.57855	12.0224	9.26004
kroB200	7.94601	10.3761	8.72876	9.06073	14.4437
kroC100	1.70113	1.02535	1.38677	0.705571	1.35477
kroD100	1.518	0.609253	0.701322	1.1549	1.06387
kroE100	1.92988	1.3143	2.50846	1.89945	2.34833
lin105	0.219738	0.471612	0.725201	0.598187	0.341395
lin318	95.197	94	99.9163	99.6397	84.3427
p654	2957.48	timelimit	2953.65	2981.87	2035.53
pcb442	138.169	179.314	145.137	147.477	39.3941
pr76	5.43789	3.70211	6.30414	6.3136	4.02344
pr107	0.836861	0.946091	0.0715489	1.01606	0.888962
pr124	3.70145	6.19464	4.64098	4.55486	4.22802
pr136	0.891231	1.48059	1.1141	1.31262	0.837261
pr144	5.55396	4.63489	6.85719	6.46244	5.59783
pr152	5.53754	6.59783	5.21229	5.9859	6.4882
pr226	24.1915	26.3643	32.4482	34.2372	52.0611
pr264	27.022	29.4888	36.9951	38.6674	29.8356
pr299	84.8491	145.326	87.2958	86.0956	78.2113
pr439	524.64	266.874	524.879	535.63	732.316
rat99	0.32278	0.771741	1.10493	0.66189	0.536387
rat195	14.521	19.975	20.1957	22.0833	4.14815
rat575	1250.57	1575.31	1235.38	1226.58	957.015
rat783	timelimit	1139.33	timelimit	timelimit	1981.04
rd100	1.15168	1.3493	0.869035	1.39074	0.886858
rd400	170.561	158.116	177.788	178.336	120.94
st70	0.919083	0.190453	0.466994	0.32977	1.09233
u159	0.876216	1.33629	1.44942	0.894153	1.6085
u574	283.502	1487.39	289.818	286.788	889.214
u724	1531.37	3397.3	1550.52	1543.77	timelimit
ulysses16	0.0740852	0.123793	0.174618	0.149275	0.100276
ulysses22	0.336434	0.398495	0.461809	0.431361	0.368571

End of Table

Table 7: Loop model times. (with iterative mode for connected components.)

Begin of Table					
Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
a280	18.3526	15.935	17.6547	18.8052	9.44658
ali535	228.392	379.788	227.126	338.565	239.672
att48	1.16448	0.863666	0.568716	1.21489	0.496279
att532	814.42	861.25	972.915	657.862	1206.61
berlin52	1.02357	0.159287	0.299164	0.225892	0.0938145
bier127	2.11443	1.65032	2.22091	1.94518	1.40268
burma14	0.292199	0.380126	0.47472	0.427247	0.334314
ch130	2.38369	1.81682	2.10115	2.44079	2.15692
ch150	5.16373	5.18267	6.0066	5.298	5.26253
d198	52.3517	54.4075	48.7617	51.1829	57.116
d493	1023.53	1399.3	664.117	1076.12	1038.33
d657	1173.55	863.659	1282.04	1153.69	1126.34
eil51	0.569266	0.755292	0.958819	0.855304	0.661341
eil76	0.306324	0.634259	1.15579	0.886464	0.467104
eil101	1.81457	1.4624	1.74095	1.58668	1.30293
fl417	549.955	396.781	462.018	474.996	302.487
gil262	30.7975	31.2584	27.9137	29.199	29.4579
gr202	13.3644	13.8297	13.6548	16.0592	14.5498
gr229	51.4808	52.4117	50.3516	51.2258	51.3921
gr431	1017.82	715.214	650.248	649.796	714.508
kroA100	3.18648	2.48804	2.1294	2.42092	2.48508
kroA150	9.74136	10.7676	9.68899	11.1955	12.103
kroA200	39.3447	34.5609	42.1561	39.5901	33.6899
kroB100	4.10057	5.37837	3.91793	4.09188	5.91449
kroB150	19.1791	19.8999	18.6907	19.0751	19.7485
kroB200	9.64277	9.38866	9.85389	10.2387	10.0322
kroC100	2.35186	2.20631	1.79211	2.01992	2.24217
kroD100	2.68885	2.34369	1.87517	2.07243	2.48871
kroE100	2.5913	2.70217	2.86451	2.75236	2.76859
lin105	1.15619	0.955106	1.87416	1.90303	1.02607
lin318	53.4143	50.9378	53.4363	53.9192	51.4061
p654	timelimit	timelimit	timelimit	timelimit	timelimit
pcb442	417.569	435.648	456.245	459.236	443.551
pr76	3.0822	3.69382	3.96754	4.49226	3.41282
pr107	1.38578	1.2257	1.05177	0.661712	0.814087
pr124	13.7919	15.1797	13.6263	13.352	16.142
pr136	4.21594	4.6875	3.49946	4.16438	4.55192
pr144	11.1543	13.2393	13.6566	10.9494	12.5632

Continuation of Table 7

Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
pr152	5.14096	5.64345	4.60753	6.01305	4.91964
pr226	115.004	93.4574	95.087	120.585	92.8313
pr264	105.473	139.202	106.333	135.106	136.993
pr299	132.445	116.546	139.108	144.418	146.239
pr439	462.621	587.67	446.147	530.871	396.912
rat99	1.35348	1.69409	1.91839	1.77686	1.54022
rat195	37.1768	40.7227	37.6355	39.8812	44.3648
rat575	380.732	661.623	391.195	639.203	537.615
rat783	206.916	175.236	150.567	264.794	173.631
rd100	1.43315	2.04621	1.79683	1.41681	1.7618
rd400	192.215	214.957	203.331	196.088	205.792
st70	0.810212	0.579729	1.25876	0.914181	1.17808
u159	2.92119	3.11856	3.61281	2.94776	3.55121
u574	204.647	149.993	142.158	209.431	237.008
u724	1703.13	930.469	789.509	938.381	930.061
ulysses16	0.161541	0.314793	0.478659	0.390235	0.242594
ulysses22	0.667902	0.877043	1.09345	0.985231	0.771468

End of Table

Table 8: Loop model times. (with union-find mode for connected components.)

Begin of Table

Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
a280	13.5849	17.2664	18.9313	18.2752	8.75185
ali535	184.452	468.487	868.885	380.645	479.529
att48	1.36042	1.17409	1.00823	0.585954	0.739308
att532	1191.05	1262.78	1079.68	833.554	1066.95
berlin52	0.916092	0.144402	0.380557	0.259724	1.03188
bier127	1.24199	1.17537	1.20092	1.15437	1.1833
burma14	0.297239	0.393685	0.480362	0.439254	0.345974
ch130	2.40183	2.40183	2.40183	2.40183	2.40183
ch150	6.4657	4.4846	6.6957	6.55829	6.31137
d198	55.0094	61.6037	53.4967	53.2568	63.4897
d493	1069.06	1570.44	1060.21	672.676	1161.01
d657	1444.2	1191.52	1370.98	864.736	1697.28
eil51	0.20155	0.505856	0.806	0.658771	0.349893
eil76	1.16437	0.615338	1.25844	0.939751	0.393228

Continuation of Table 8

Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
eil101	1.37582	2.09907	1.21016	1.13558	0.943979
fl417	434.676	488.732	499.215	557.017	1123.52
gil262	27.4513	31.4909	28.9299	31.9031	30.8301
gr202	17.3155	13.7547	14.8111	16.9476	16.2352
gr229	58.4295	58.1011	55.0705	54.1061	57.5767
gr431	650.677	778.302	785.495	830.564	550.596
kroA100	2.55554	1.9342	2.81642	2.81701	2.23133
kroA150	11.163	11.6912	11.3714	11.5041	9.99797
kroA200	43.3806	40.2418	46.8787	40.4964	35.8912
kroB100	6.14849	5.61324	4.49494	4.67342	4.823
kroB150	19.8667	19.5383	18.2282	22.2093	22.5503
kroB200	10.5393	8.42315	9.99408	11.3301	10.8367
kroC100	1.81411	2.5524	3.01457	2.49073	2.6146
kroD100	2.69332	2.24494	2.12691	2.26664	3.05669
kroE100	2.00406	2.27088	2.2227	2.27697	2.19164
lin105	1.2781	1.45059	1.92584	1.58445	1.33618
lin318	56.6795	54.636	57.8024	55.2051	52.5388
p654	timelimit	timelimit	timelimit	timelimit	timelimit
pcb442	555.037	470.312	506.523	461.635	456.149
pr76	4.45937	4.63479	3.89766	4.24208	4.65822
pr107	1.4805	0.60014	0.455146	1.04607	1.04307
pr124	13.8461	15.2386	14.7622	13.3301	18.1701
pr136	5.69115	3.91492	4.4839	4.35587	4.42602
pr144	13.8821	12.3036	13.3806	12.6685	14.0731
pr152	5.68139	5.27863	5.0444	4.58282	4.80842
pr226	100.473	112.2	96.5273	103.408	95.7854
pr264	108.945	152.186	148.285	117.881	160.787
pr299	189.442	145.587	183.96	135.502	179.076
pr439	492.905	471.299	871.34	579.166	565.886
rat99	1.65178	2.11299	1.42178	1.24599	1.94116
rat195	41.5437	43.8174	39.3012	41.1039	46.2966
rat575	425.338	651.846	566.443	444.27	422.729
rat783	207.777	232.973	211.745	201.89	180.228
rd100	2.51756	1.48775	1.35495	1.9918	2.03837
rd400	224.166	220.595	215.944	203.693	217.103
st70	1.40745	1.30488	1.087	0.69895	0.848273
u159	2.88977	3.14335	4.33818	3.46008	3.52178
u574	179.404	197.476	368.669	469.182	200.27
u724	685.573	949.706	799.601	797.455	850.223
ulysses16	0.789214	0.977823	0.168406	1.07683	0.885166

Continuation of Table 8					
Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
ulysses22	0.29526	0.55738	0.82243	0.694285	0.425982
End of Table					

Table 9: Miller-Tucker-Zemlin model times.

Begin of Table					
Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
a280	killed	killed	killed	killed	killed
ali535	seg fault	seg fault	seg fault	seg fault	seg fault
att48	162.547	135.993	162.335	162.162	141.988
att532	seg fault	seg fault	seg fault	seg fault	seg fault
berlin52	6.32115	5.09906	5.18067	6.08895	4.72986
bier127	timelimit	timelimit	timelimit	timelimit	timelimit
burma14	0.601485	1.17896	0.810547	0.498793	0.854356
ch130	timelimit	timelimit	timelimit	timelimit	timelimit
ch150	timelimit	timelimit	timelimit	timelimit	timelimit
d198	seg fault	seg fault	seg fault	seg fault	seg fault
d493	seg fault	seg fault	seg fault	seg fault	seg fault
d657	seg fault	seg fault	seg fault	seg fault	seg fault
eil51	11.0391	10.3278	10.7725	12.9816	6.2475
eil76	33.6582	17.2382	37.0599	34.7641	20.8409
eil101	195.466	136.993	197.545	193.166	11.872
fl417	seg fault	seg fault	seg fault	seg fault	seg fault
gil262	killed	killed	killed	killed	killed
gr202	killed	killed	killed	killed	killed
gr229	killed	killed	killed	killed	killed
gr431	seg fault	seg fault	seg fault	seg fault	seg fault
kroA100	timelimit	timelimit	timelimit	timelimit	timelimit
kroA150	killed	timelimit	killed	killed	killed
kroA200	killed	killed	killed	killed	killed
kroB100	timelimit	timelimit	timelimit	timelimit	timelimit
kroB150	killed	killed	killed	killed	killed
kroB200	killed	killed	killed	killed	killed
kroC100	timelimit	timelimit	timelimit	timelimit	timelimit
kroD100	timelimit	timelimit	timelimit	timelimit	timelimit
kroE100	seg fault	timelimit	timelimit	timelimit	1784.54
lin105	3024.89	3061.64	seg fault	3066.79	3367.51

Continuation of Table 9

Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
lin318	killed	killed	killed	killed	killed
p654	seg fault	seg fault	seg fault	seg fault	seg fault
pcb442	seg fault	seg fault	seg fault	seg fault	seg fault
pr76	492.573	548.738	485.548	485.19	398.684
pr107	killed	timelimit	killed	killed	timelimit
pr124	timelimit	timelimit	timelimit	timelimit	killed
pr136	killed	killed	killed	killed	killed
pr144	killed	killed	killed	killed	killed
pr152	killed	killed	killed	killed	killed
pr226	timelimit	timelimit	timelimit	timelimit	timelimit
pr264	killed	killed	killed	killed	killed
pr299	killed	killed	killed	killed	killed
pr439	killed	killed	killed	killed	killed
rat99	77.0294	158.684	80.0209	75.8498	135.416
rat195	killed	killed	killed	killed	killed
rat575	seg fault	seg fault	seg fault	seg fault	seg fault
rat783	seg fault	seg fault	seg fault	seg fault	seg fault
rd100	822.817	836.243	824.855	646.366	626.084
rd400	killed	killed	killed	killed	killed
st70	250.639	415.963	249.583	249.345	461.602
u159	killed	2969.15	killed	killed	killed
u574	killed	2969.15	killed	killed	killed
u724	seg fault	seg fault	seg fault	seg fault	seg fault
ulysses16	2.1996	1.99831	2.74625	1.79203	1.58455
ulysses22	73.4459	145.798	75.0388	76.3917	149.927

End of Table

Table 10: 1st Timed Stage Dependent model times.

Begin of Table

Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
a280	seg fault	seg fault	seg fault	seg fault	seg fault
ali535	seg fault	seg fault	seg fault	seg fault	seg fault
att48	timelimit	timelimit	timelimit	timelimit	timelimit
att532	seg fault	seg fault	seg fault	seg fault	seg fault
berlin52	timelimit	timelimit	timelimit	timelimit	timelimit
bier127	seg fault	seg fault	seg fault	seg fault	seg fault

Continuation of Table 10

Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
burma14	1.8532	3.02265	2.56129	1.83788	2.89694
ch130	seg fault	seg fault	seg fault	seg fault	seg fault
ch150	seg fault	seg fault	seg fault	seg fault	seg fault
d198	seg fault	seg fault	seg fault	seg fault	seg fault
d493	seg fault	seg fault	seg fault	seg fault	seg fault
d657	seg fault	seg fault	seg fault	seg fault	seg fault
eil51	timelimit	timelimit	timelimit	timelimit	timelimit
eil76	seg fault	seg fault	seg fault	seg fault	seg fault
eil101	seg fault	seg fault	seg fault	seg fault	seg fault
fl417	seg fault	seg fault	seg fault	seg fault	seg fault
gil262	killed	seg fault	seg fault	seg fault	seg fault
gr202	seg fault	seg fault	seg fault	seg fault	seg fault
gr229	seg fault	seg fault	seg fault	seg fault	seg fault
gr431	seg fault	seg fault	seg fault	seg fault	seg fault
kroA100	seg fault	seg fault	seg fault	seg fault	seg fault
kroA150	seg fault	seg fault	seg fault	seg fault	seg fault
kroA200	seg fault	seg fault	seg fault	seg fault	seg fault
kroB100	seg fault	seg fault	seg fault	seg fault	seg fault
kroB150	seg fault	seg fault	seg fault	seg fault	seg fault
kroB200	seg fault	seg fault	seg fault	seg fault	seg fault
kroC100	seg fault	seg fault	seg fault	seg fault	seg fault
kroD100	seg fault	seg fault	seg fault	seg fault	seg fault
kroE100	seg fault	seg fault	seg fault	seg fault	seg fault
lin105	seg fault	seg fault	seg fault	seg fault	seg fault
lin318	seg fault	seg fault	seg fault	seg fault	seg fault
p654	seg fault	seg fault	seg fault	seg fault	seg fault
pcb442	seg fault	seg fault	seg fault	seg fault	seg fault
pr76	seg fault	seg fault	seg fault	seg fault	seg fault
pr107	seg fault	seg fault	seg fault	seg fault	seg fault
pr124	seg fault	seg fault	seg fault	seg fault	seg fault
pr136	seg fault	seg fault	seg fault	seg fault	seg fault
pr144	seg fault	seg fault	seg fault	seg fault	seg fault
pr152	seg fault	seg fault	seg fault	seg fault	seg fault
pr226	seg fault	seg fault	seg fault	seg fault	seg fault
pr264	seg fault	seg fault	seg fault	seg fault	seg fault
pr299	seg fault	seg fault	seg fault	seg fault	seg fault
pr439	seg fault	seg fault	seg fault	seg fault	seg fault
rat99	seg fault	seg fault	seg fault	seg fault	seg fault
rat195	seg fault	seg fault	seg fault	seg fault	seg fault
rat575	seg fault	seg fault	seg fault	seg fault	seg fault

Continuation of Table 10

Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
rat783	seg fault	seg fault	seg fault	seg fault	seg fault
rd100	seg fault	seg fault	seg fault	seg fault	seg fault
rd400	seg fault	seg fault	seg fault	seg fault	seg fault
st70	seg fault	seg fault	seg fault	seg fault	seg fault
u159	seg fault	seg fault	seg fault	seg fault	seg fault
u574	seg fault	seg fault	seg fault	seg fault	seg fault
u724	seg fault	seg fault	seg fault	seg fault	seg fault
ulysses16	6.94603	5.80376	7.12612	7.43082	9.51197
ulysses22	23.5565	28.3075	21.9579	22.6011	40.5673

End of Table

Table 11: 2nd Timed Stage Dependent model times.

Begin of Table

Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
a280	seg fault	seg fault	seg fault	seg fault	seg fault
ali535	seg fault	seg fault	seg fault	seg fault	seg fault
att48	timelimit	timelimit	timelimit	timelimit	timelimit
att532	seg fault	seg fault	seg fault	seg fault	seg fault
berlin52	timelimit	timelimit	timelimit	timelimit	timelimit
bier127	seg fault	seg fault	seg fault	seg fault	seg fault
burma14	2.85735	2.88435	2.35744	3.00495	2.52864
ch130	seg fault	seg fault	seg fault	seg fault	seg fault
ch150	seg fault	seg fault	seg fault	seg fault	seg fault
d198	seg fault	seg fault	seg fault	seg fault	seg fault
d493	seg fault	seg fault	seg fault	seg fault	seg fault
d657	seg fault	seg fault	seg fault	seg fault	seg fault
eil51	timelimit	timelimit	timelimit	timelimit	timelimit
eil76	seg fault	seg fault	seg fault	seg fault	seg fault
eil101	seg fault	seg fault	seg fault	seg fault	seg fault
fl417	seg fault	seg fault	seg fault	seg fault	seg fault
gil262	seg fault	seg fault	seg fault	seg fault	seg fault
gr202	seg fault	seg fault	seg fault	seg fault	seg fault
gr229	seg fault	seg fault	seg fault	seg fault	seg fault
gr431	seg fault	seg fault	seg fault	seg fault	seg fault
kroA100	seg fault	seg fault	seg fault	seg fault	seg fault
kroA150	seg fault	seg fault	seg fault	seg fault	seg fault

Continuation of Table 11

Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
kroA200	seg fault	seg fault	seg fault	seg fault	seg fault
kroB100	seg fault	seg fault	seg fault	seg fault	seg fault
kroB150	seg fault	seg fault	seg fault	seg fault	seg fault
kroB200	seg fault	seg fault	seg fault	seg fault	seg fault
kroC100	seg fault	seg fault	seg fault	seg fault	seg fault
kroD100	seg fault	seg fault	seg fault	seg fault	seg fault
kroE100	seg fault	seg fault	seg fault	seg fault	seg fault
lin105	seg fault	seg fault	seg fault	seg fault	seg fault
lin318	seg fault	seg fault	seg fault	seg fault	seg fault
p654	seg fault	seg fault	seg fault	seg fault	seg fault
pcb442	seg fault	seg fault	seg fault	seg fault	seg fault
pr76	seg fault	seg fault	seg fault	seg fault	seg fault
pr107	seg fault	seg fault	seg fault	seg fault	seg fault
pr124	seg fault	seg fault	seg fault	seg fault	seg fault
pr136	seg fault	seg fault	seg fault	seg fault	seg fault
pr144	seg fault	seg fault	seg fault	seg fault	seg fault
pr152	seg fault	seg fault	seg fault	seg fault	seg fault
pr226	seg fault	seg fault	seg fault	seg fault	seg fault
pr264	seg fault	seg fault	seg fault	seg fault	seg fault
pr299	seg fault	seg fault	seg fault	seg fault	seg fault
pr439	seg fault	seg fault	seg fault	seg fault	seg fault
rat99	seg fault	seg fault	seg fault	seg fault	seg fault
rat195	seg fault	seg fault	seg fault	seg fault	seg fault
rat575	seg fault	seg fault	seg fault	seg fault	seg fault
rat783	seg fault	seg fault	seg fault	seg fault	seg fault
rd100	seg fault	seg fault	seg fault	seg fault	seg fault
rd400	seg fault	seg fault	seg fault	seg fault	seg fault
st70	seg fault	seg fault	seg fault	seg fault	seg fault
u159	seg fault	seg fault	seg fault	seg fault	seg fault
u574	seg fault	seg fault	seg fault	seg fault	seg fault
u724	seg fault	seg fault	seg fault	seg fault	seg fault
ulysses16	8.51911	8.91792	8.39897	9.12781	13.7112
ulysses22	105.108	76.2662	105.693	102.994	130

End of Table

Table 12: 3rd Timed Stage Dependent model times.

Begin of Table

Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
a280	seg fault	seg fault	seg fault	seg fault	seg fault
ali535	seg fault	seg fault	seg fault	seg fault	seg fault
att48	timelimit	timelimit	timelimit	timelimit	timelimit
att532	seg fault	seg fault	seg fault	seg fault	seg fault
berlin52	timelimit	timelimit	timelimit	timelimit	timelimit
bier127	seg fault	seg fault	seg fault	seg fault	seg fault
burma14	7.50358	5.97829	5.25801	6.1992	6.0097
ch130	seg fault	seg fault	seg fault	seg fault	seg fault
ch150	seg fault	seg fault	seg fault	seg fault	seg fault
d198	seg fault	seg fault	seg fault	seg fault	seg fault
d493	seg fault	seg fault	seg fault	seg fault	seg fault
d657	seg fault	seg fault	seg fault	seg fault	seg fault
eil51	timelimit	timelimit	timelimit	timelimit	timelimit
eil76	seg fault	seg fault	seg fault	seg fault	seg fault
eil101	seg fault	seg fault	seg fault	seg fault	seg fault
fl417	seg fault	seg fault	seg fault	seg fault	seg fault
gil262	seg fault	seg fault	seg fault	seg fault	seg fault
gr202	seg fault	seg fault	seg fault	seg fault	seg fault
gr229	seg fault	seg fault	seg fault	seg fault	seg fault
gr431	seg fault	seg fault	seg fault	seg fault	seg fault
kroA100	seg fault	seg fault	seg fault	seg fault	seg fault
kroA150	seg fault	seg fault	seg fault	seg fault	seg fault
kroA200	seg fault	seg fault	seg fault	seg fault	seg fault
kroB100	seg fault	seg fault	seg fault	seg fault	seg fault
kroB150	seg fault	seg fault	seg fault	seg fault	seg fault
kroB200	seg fault	seg fault	seg fault	seg fault	seg fault
kroC100	seg fault	seg fault	seg fault	seg fault	seg fault
kroD100	seg fault	seg fault	seg fault	seg fault	seg fault
kroE100	seg fault	seg fault	seg fault	seg fault	seg fault
lin105	seg fault	seg fault	seg fault	seg fault	seg fault
lin318	seg fault	seg fault	seg fault	seg fault	seg fault
p654	seg fault	seg fault	seg fault	seg fault	seg fault
pcb442	seg fault	seg fault	seg fault	seg fault	seg fault
pr76	seg fault	seg fault	seg fault	seg fault	seg fault
pr107	seg fault	seg fault	seg fault	seg fault	seg fault
pr124	seg fault	seg fault	seg fault	seg fault	seg fault
pr136	seg fault	seg fault	seg fault	seg fault	seg fault
pr144	seg fault	seg fault	seg fault	seg fault	seg fault

Continuation of Table 12					
Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
pr152	seg fault	seg fault	seg fault	seg fault	seg fault
pr226	seg fault	seg fault	seg fault	seg fault	seg fault
pr264	seg fault	seg fault	seg fault	seg fault	seg fault
pr299	seg fault	seg fault	seg fault	seg fault	seg fault
pr439	seg fault	seg fault	seg fault	seg fault	seg fault
rat99	seg fault	seg fault	seg fault	seg fault	seg fault
rat195	seg fault	seg fault	seg fault	seg fault	seg fault
rat575	seg fault	seg fault	seg fault	seg fault	seg fault
rat783	seg fault	seg fault	seg fault	seg fault	seg fault
rd100	seg fault	seg fault	seg fault	seg fault	seg fault
rd400	seg fault	seg fault	seg fault	seg fault	seg fault
st70	seg fault	seg fault	seg fault	seg fault	seg fault
u159	seg fault	seg fault	seg fault	seg fault	seg fault
u574	seg fault	seg fault	seg fault	seg fault	seg fault
u724	seg fault	seg fault	seg fault	seg fault	seg fault
ulysses16	20.4292	26.413	22.2698	16.632	28.8868
ulysses22	362.311	525.824	366.432	364.059	389.364
End of Table					

Table 13: User-cut callback model times.

Begin of Table					
Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
a280	14.3204	77.2792	19.1065	18.6189	39.6368
ali535	1986.24	1727.89	1961.84	1959.11	killed
att48	0.888081	0.325614	0.686269	0.508093	1.0954
att532	1275	1623	1461	1492	1456
berlin52	0.441846	0.542943	0.604049	0.573434	0.480125
bier127	1.91902	2.31748	1.27952	1.3222	1.64839
burma14	0.213541	0.254701	0.290839	0.271948	0.234163
ch130	2.97707	3.48076	3.46354	2.90078	3.4774
ch150	8.0919	9.58887	5.96125	6.44847	4.77908
d198	62.1176	27.0516	59.4338	61.8227	29.4597
d493	2861.62	2325.82	2849.29	2861.69	killed
d657	killed	killed	killed	killed	killed
eil51	0.892554	0.180936	0.469007	0.32287	1.03722
eil76	0.597625	0.874719	0.136934	1.00462	0.734287

Continuation of Table 13

Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
eil101	0.795013	1.05046	1.14108	0.59798	1.3581
fl417	killed	163.052	killed	killed	394.572
gil262	142.557	96.4863	140.24	140.262	93.1176
gr202	4.48275	4.4219	5.271	4.81959	5.02769
gr229	152.505	24.9749	154.052	153.824	154.1
gr431	1520.53	849.01	1508.77	1509.65	670.632
kroA100	6.48131	5.39353	5.86041	5.40752	4.77868
kroA150	6.68655	6.49084	5.75119	5.88243	6.67196
kroA200	41.0263	56.726	44.5068	49.4873	116.171
kroB100	6.32173	6.50903	5.13153	5.99876	6.70679
kroB150	16.8119	26.3928	16.9391	18.8935	25.3454
kroB200	21.416	11.866	19.6429	20.2015	5.72769
kroC100	3.04197	3.06996	2.98862	2.34029	2.71165
kroD100	1.88807	2.19462	2.32629	1.75014	2.54641
kroE100	4.06621	4.4915	4.73584	4.42387	4.19744
lin105	1.07078	0.679565	1.23293	0.955592	0.392554
lin318	103.475	127.895	101.984	101.301	131.988
p654	676.18	171.913	677.466	676.071	322.168
pcb442	456.743	429.196	457.273	451.77	617.682
pr76	5.27755	14.0336	4.71522	5.67827	21.1404
pr107	0.812273	0.983359	0.160006	1.07285	0.895855
pr124	2.98815	3.73824	3.15595	5.01038	2.78797
pr136	3.03499	4.47984	2.95743	3.23589	2.42972
pr144	4.63701	4.79982	6.16458	4.8412	9.60285
pr152	11.0702	7.16887	11.2237	11.4951	6.22113
pr226	9.44849	15.649	10.3364	12.7394	17.0594
pr264	11.8372	13.6099	13.0191	14.2489	12.7017
pr299	522.861	239.498	531.948	534.096	213.939
pr439	2499.76	killed	2530.5	2496.77	992.492
rat99	1.14814	1.14705	0.882015	0.514732	0.659977
rat195	55.1469	39.2084	55.3565	54.9675	41.444
rat575	killed	killed	killed	killed	killed
rat783	killed	killed	killed	killed	killed
rd100	1.38172	1.13886	1.68983	0.934176	1.09426
rd400	2106.25	755.921	2117.12	2091.99	523.195
st70	0.668607	0.310787	1.04919	0.680965	1.03225
u159	1.1772	2.87581	1.17117	2.02938	2.47249
u574	killed	killed	killed	killed	killed
u724	killed	killed	killed	killed	killed
ulysses16	0.842717	0.927846	1.01391	0.972936	0.885678

Continuation of Table 13					
Instance	Seed				
	88325680	202644352	733894336	921394368	952644352
ulysses22	0.0688889	0.174377	0.280687	0.226007	0.122075
End of Table					

Table 14: Best solution found for each instance using metaheuristic methods.

Begin of Table			
Instance	Metaheuristic		
	GRASP	VNS	Simulated Annealing
ali535	212997	202829	237391
att532	29042	27798	30426
d493	36760	35212	38255
d657	52024	49267	54404
fl417	12090	11861	12901
gr431	179650	172697	190595
lin318	43420	42508	45776
p654	35114	34650	38108
pcb442	53509	51148	56137
pr439	111269	107250	118528
rat575	7302	6810	7663
rat783	9621	8900	11489
rd400	15935	15397	16757
u574	39473	37259	39289
u724	45128	42280	47565
End of Table			

Table 15: Best solution found for each instance using matheuristic methods.

Begin of Table		
Instance	Matheuristic	
	Hard-fixing	Local branching
ali535	3.37E+06	207708
att532	309636	35888
d493	113549	35100
d657	232159	50348
fl417	55445	11861

Continuation of Table 15		
Instance	Metaheuristic	
	Hard-fixing	Local branching
gr431	233064	172467
lin318	42029	42333
p654	107737	35550
pcb442	221440	50912
pr439	270646	108311
rat575	12934	7825
rat783	72134	8949
rd400	215558	15281
u574	40197	37044
u724	157485	41950
End of Table		

Table 16: Best solution found for each instance using metaheuristics and lazy callback.

Begin of Table			
Instance	Method		
	GRASP + LC	VNS+LC	Simulated annealing + LC
ali535	202339	202339	202339
att532	27686	27686	27686
d493	35002	35002	35002
d657	48913	48913	48913
fl417	11861	11861	11861
gr431	171414	171414	171414
lin318	42029	42029	42029
p654	38305	34655	47150
pcb442	50778	50778	50778
pr439	107217	107217	107217
rat575	6972	6773	6773
rat783	8806	8806	8806
rd400	15281	15281	15281
u574	36905	36905	36905
u724	46372	41910	47201
End of Table			

References

[1] Samuel A. Mulder and Donald C. Wunsch. Million city traveling salesman problem solution by divide and conquer clustering with adaptive resonance neural networks. *Neural Networks*, 16(5):827 – 832, 2003. Advances in Neural Networks Research: IJCNN '03.

[2] Emile Aarts and Jan Korst. *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing.* John Wiley & Sons, Inc., New York, NY, USA, 1989.

[3] Emile Aarts and Jan K. Lenstra, editors. *Local Search in Combinatorial Optimization.* John Wiley & Sons, Inc., New York, NY, USA, 1997.

[4] M Montaz Ali, A. Törn, and Sami Viitanen. A direct search variant of the simulated annealing algorithm for optimization involving continuous variables. *Computers & Operations Research*, 29(1):87–102, 2002.

[5] Margarida F. Cardoso and Sebastiao F. Salcedo, Romualdo L.and de Azevedo. Nonequilibrium simulated annealing: a faster approach to combinatorial minimization. *Industrial & engineering chemistry research*, 33(8):1908–1918, 1994.

[6] Vladimír Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985.

[7] Armin Claus. A new formulation for the travelling salesman problem. *SIAM Journal on Algebraic Discrete Methods*, 5(1):21–25, 1984.

[8] William Cook. Concorde tsp solver, 2019.

[9] Thomas H. Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms.* MIT press, 2009.

[10] Danna, E. Performance variability in mixed integer programming, 2018. `http://coral.ie.lehigh.edu/mip-2008/talks/danna.pdf`.

[11] George Dantzig, Ray Fulkerson, and Selmer Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410, 1954.

[12] E. D. Dolan and J.J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91(2):201–213, 2002.

[13] Gerd Finke, Armin Claus, and Eldon Gunn. A two-commodity network flow approach to the traveling salesman problem. *Congress Num*, 41, 01 1984.

[14] Matteo Fischetti. *Lezioni di ricerca operativa.* Libreria Progetto, 2014.

[15] Matteo Fischetti and Andrea Lodi. Local branching. *Mathematical Programming*, 98:23–47, 09 2003.

[16] Kenneth R. Fox, Bezalel Gavish, and Stephen C. Graves. An n-constraint formulation of the (time-dependent) traveling salesman problem. *Operations Research*, 28(4):1018–1021, 1980.

[17] Bernard A. Galler and Michael J. Fisher. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, 1964.

[18] Bezalel Gavish and Stephen C. Graves. The travelling salesman problem and related problems, 1978.

[19] Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. In *Readings in computer vision*, pages 564–584. Elsevier, 1987.

[20] LLC Gurobi Optimization. Gurobi optimizer reference manual, 2019.

[21] Pierre Hansen and Nenad Mladenović. Variable neighborhood search: Principles and applications. *European journal of operational research*, 130(3):449–467, 2001.

[22] H. Hoos Holger and Stützle Thomas. *Stochastic Local Search: Foundations and Applications*. Elsevier, 2005.

[23] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; part i, graph partitioning. *Operations research*, 37(6):865–892, 1989.

[24] Vecchi Mario P. Kirkpatrick Scott, Gelatt C. Daniel. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[25] Thorsten Koch, Tobias Achterberg, Erling Andersen, Oliver Bastert, Timo Berthold, Robert E. Bixby, Emilie Danna, Gerald Gamrath, Ambros M. Gleixner, Stefan Heinz, Andrea Lodi, Hans Mittelmann, Ted Ralphs, Domenico Salvagnin, Daniel E. Steffy, and Kati Wolter. Miplib 2010. *Mathematical Programming Computation*, 3(2):103, Jun 2011.

[26] Zemlin Richard A. Miller Clair E., Tucker Albert W. Integer programming formulation of traveling salesman problems. *Journal of the ACM (JACM)*, 7(4):326–329, 1960.

[27] Nenad Mladenović and Pierre Hansen. Variable neighborhood search. *Computers & operations research*, 24(11):1097–1100, 1997.

[28] A.J. Orman and H. Paul Williams. A survey of different integer programming formulations of the travelling salesman problem. *Optimisation, Econometric and Financial Analysis*, 9:93–108, 2006.

[29] Miliotis P. Integer programming approaches to the travelling salesman problem. *Mathematical Programming*, 10(1):367–378, 1976.

[30] Colin R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems.* John Wiley & Sons, Inc., New York, NY, USA, 1993.

[31] Gerhard Reinelt. TSPLIB—a traveling salesman problem library. *ORSA Journal on Computing*, 3(4):376–384, November 1991.

[32] Mauricio G.C. Resende and Celso C. Ribeiro. Grasp: Greedy randomized adaptive search procedures. In *Search methodologies*, pages 287–312. Springer, 2014.

[33] R. Romero, R.A. Gallego, and A. Monticelli. Transmission system expansion planning by simulated annealing. In *Proceedings of Power Industry Computer Applications Conference*, pages 278–283. IEEE, 1995.

[34] Youssef G. Saab and Vasant B. Rao. Combinatorial optimization by stochastic evolution. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 10(4):525–535, 1991.

[35] T. Sawik. A note on the miller-tucker-zemlin model for the asymmetric traveling salesman problem. *Bulletin of the Polish Academy of Sciences Technical Sciences*, 64(3):517–520, 2016.

[36] Robert E. Tarjan and Jan Van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM (JACM)*, 31(2):245–281, 1984.

[37] Steven Vajda. *Mathematical Programming.* Addison-Wesley, 1961.

[38] Jan van Leeuwen and R. van der Weide. *Alternative path compression techniques*, volume 77. Unknown Publisher, 1977.

[39] Wikipedia contributors. Ackermann function — Wikipedia, the free encyclopedia, 2019. [Online; accessed 10-May-2019].

[40] Richard T. Wong. Integer programming formulations of the traveling salesman problem. In *Proceedings of the IEEE international conference of circuits and computers*, pages 149–152. IEEE Press Piscataway, NJ, 1980.