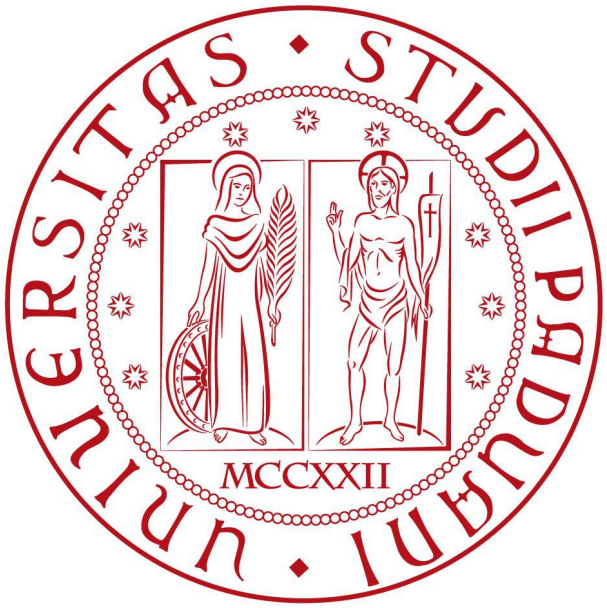
Università degli studi di Padova
Anno accademico 2014-2015
Dipartimento di Ingegneria dell'Informazione
Tesi di laurea magistrale in Ingegneria Informatica



Modern branch-and-cut solvers for Mixed-Integer Linear Programming: a computational comparison

Relatore: Ch.mo Prof. Matteo Fischetti
Laureando: Alessandro Beltramin

Alla mia famiglia, Leandro e Roberta,
e alle tre mie care Laura, Laura e Laura.

Abstract

This thesis describes the design and the implementation of an interface for a modern Mixed Integer Programming (MIP) solver and the development of an advanced example for the Uncapacited Facility Location (UFL) problem. In the field of Operational Research, UFL is a well-known NP-complete problem. That means, there is no algorithm known by now which can solve this problem in polynomial time. In general, modern solvers have to solve different types of MIP problems, so the internal cuts and the internal heuristics they use cannot fully exploit the special structure of such a specific problem. Thus, we used advanced features, such as callback procedures, to provide some prior knowledge and improve the performance of the existing branch-and-cut method for the UFL problem. Also, we developed two additional algorithms, based on the well-known Hard Fixing and Local Branching matheuristics. We tested our algorithms on several hard instances using the commercial solver IBM ILOG CPLEX. Then, to have a comparison also with another commercial solver, we developed an interface for Gurobi which is able to reuse the same software already working for CPLEX. Our interface, without modifying the original code, captures every call to CPLEX C library and redirects it to Gurobi C library, preserving its functionalities. Using both the UFL code and our interface, we finally made a computational comparison between CPLEX and Gurobi.

Contents

1 Mixed Integer Programming **6**
- 1.1 MIP and Branch-and-Bound . . . 6
- 1.2 Branch-and-Cut . . . 8
- 1.3 Branch-and-cut with callbacks . . . 9

2 A specific MIP problem: UFL **10**
- 2.1 Introduction . . . 10
- 2.2 Uncapacitated Facility Location . . . 10
  - 2.2.1 Traditional problem . . . 10
  - 2.2.2 Advanced resolution strategies . . . 11
- 2.3 Benders' Decomposition . . . 16
  - 2.3.1 Master problem . . . 17
  - 2.3.2 Benders cuts . . . 17
- 2.4 Matheuristics . . . 20
  - 2.4.1 Variable fixing (diving) . . . 20
  - 2.4.2 Local branching . . . 22
- 2.5 Computational results . . . 23
  - 2.5.1 Instances . . . 23
  - 2.5.2 Software interface . . . 24
  - 2.5.3 Solving to proven optimality . . . 26
  - 2.5.4 Heuristic solutions . . . 28

3 An interface for Gurobi **35**
- 3.1 Some available solutions . . . 35
- 3.2 The idea: a direct interface Cplex-to-Gurobi . . . 37
  - 3.2.1 Interface development . . . 38
  - 3.2.2 How to match functions . . . 38
  - 3.2.3 How to implement functions . . . 43
- 3.3 Functions implemented . . . 51
  - 3.3.1 Environment and problem creation . . . 52
  - 3.3.2 Problem modification . . . 55
  - 3.3.3 Optimize . . . 62
  - 3.3.4 Access problem data . . . 64
  - 3.3.5 File Input/Output . . . 72
  - 3.3.6 Parameters setting . . . 76
  - 3.3.7 Callbacks . . . 79
  - 3.3.8 Other functions . . . 90
- 3.4 Main differences . . . 93
  - 3.4.1 Callback functions . . . 95

4 Test: Cplex vs Gurobi **100**
4.1 Test description 100
4.2 Testbed 101
4.3 Computational Results 102
4.3.1 Solve to proven optimality 102
4.3.2 Local Branching 111

5 Conclusions **115**

6 References **117**

Introduction

This thesis describes the design and the implementation of an interface for a modern Mixed Integer Programming (MIP) solver and the development of an advanced example for the Uncapacited Facility Location (UFL) problem.

In the field of Operational Research, UFL is a well-known NP-complete problem. That means, there is no algorithm known by now which can solve this problem in polynomial time. As a consequence, it is necessary to use combinatorial optimization methods and, among modern MIP solvers, branch-and-cut algorithm is the most used [FLS10]. In general, modern solvers have to solve different types of MIP problems, so the internal cuts and the internal heuristics they use cannot fully exploit the special structure of such a specific problem. Thus, we use advanced features, such as callback procedures, to provide some prior knowledge and improve the performance of the existing branch-and-cut method. Also, we developed two additional algorithms, based on the well-known Hard Fixing and Local Branching matheuristics [FL03]. To validate their usefulness, we tested both the developed branch-and-cut algorithm and the two matheuristics on several hard instances, using the software libraries of the commercial solver IBM ILOG CPLEX.

Then, we developed an interface for executing the algorithms we coded for the UFL problem, with another modern MIP solver, Gurobi. Our interface, without modifying the original code, captures every call to CPLEX C library and redirects it to Gurobi C library, preserving its functionalities. In general, our work can be very useful because the software we developed it is not limited only to our UFL example, but it is ready to be applied to most of the code already written for CPLEX. Existing software, then, can run directly Gurobi solver without having to migrate the algorithms.

Finally, using both the UFL code and our interface, we make a computational comparison between CPLEX and Gurobi. On fair conditions and with all the callbacks enabled, our UFL code interfaced with Gurobi had a small loss of speed. At the same time, after switching the callbacks off, no substantial differences emerged between CPLEX and Gurobi, and rather Gurobi satisfied the positive expectations on much many occasions.

We believe that such a result is due to a different type of management for some of the advanced features the two MIP solvers provide, in particular for the user-cut and the heuristic callbacks. Nevertheless, all the results we obtained from both the commercial solvers are very satisfactory.

Our thesis is organized as follows:

- In the Chapter 1, we briefly recall some basic concepts, such as the definition of a MIP problem, how branch-and-bound algorithm works and in what branch-and-cut differs, presenting also its variant based on callback procedures.
- Then we introduce, in Chapter 2, an advanced example of code development for the UFL problem, based on Benders' cuts and callbacks.
- Chapter 3 describes the idea of a direct interface for comparing CPLEX and Gurobi, in particular, all the functions we implemented, as well as the limits of our interface and the main differences in the solvers' APIs.
- In Chapter 4, we present several performance comparisons between CPLEX and Gurobi, using both the UFL code and our Gurobi interface.
- Finally, in Chapter 5 there are some final considerations about the work.

1 Mixed Integer Programming

We present here some basic concepts required to understand the rest of our thesis. We recall the definition of a MIP problem and how to solve it with the branch-and-bound algorithm. Finally we outline the differences with the branch-and-cut algorithm and its variant with callbacks.

1.1 MIP and Branch-and-Bound

Description A problem of mixed integer programming (MIP) consists of the minimization of a linear function, subject to a finite number of linear constraints and to the additional constraint that some or all the variables are restricted to integer values. When the integer restriction is applied only to a subset of the variables, the problem is defined as Mixed Integer Linear Programming (MILP).

Mathematical formulation A general problem is defined as:

$$\begin{cases} \min & c_C^T x_C + c_I^T x_I \\ \text{subject to:} & A_C x_C + A_I x_I = b \\ & x_C, x_I \geq 0,\ x_I \text{ integer} \end{cases}$$

where x_I is the vector of non-negative integer variables, x_C is the vector of non-negative continuous variables, c_I and c_C are the respective objective function coefficients, A_I and A_C are the respective left-hand-side coefficient matrices of the constraint set and b is the right-hand-side column vector.

Branch-and-bound algorithm Branch-and-bound (B&B) is a generic optimization algorithm based on intelligent enumeration to arrive at an optimal solution for any problem of combinatorial optimization.

B&B uses the divide-and-conquer strategy to partition the search space of the original problem into sub-problems. This technique involves construction of a search tree. Each node is another MIP problem, but restricted: it consists of the original constraints along with some additional constraints on the bounds of the integer variables. At each node, the algorithm solves its linear programming relaxation, i.e. it solves the restricted sub-problem with all its variables relaxed to be continuous.

B&B begins by solving the root node at the top of the tree, with all its variables relaxed to assume continuous values.

- If the linear programming (LP) relaxation of the root node is infeasible, then the original problem is also infeasible. The LP relaxation is in fact more general than the more restricted, original problem, and the algorithm terminates with no feasible solutions.
- At the same time, if the linear programming (LP) relaxation of the root node produces an optimal solution where integer restricted variables have already integer values, then the solution is also optimal for the original MIP problem.
- Finally, if the optimal solution for the relaxation has at least one of integer-restricted variables assuming fractional values, then branching occurs: one of these variables is chosen, e.g. $x_i = f$, and two sub-problems are created. The first problem has the additional constraint $x_i \leq \lfloor f \rfloor$, while the second has $x_i \geq \lceil f \rceil$. These sub-problems are then solved recursively, with the new bounds remaining in effect both for them and for any of their descendants.

Obviously, an explicit enumeration of all the nodes is unacceptable, because of the exponential growth in the size of the search tree. Then the effectiveness of the branch-and-bound algorithm relies on its ability to prune nodes, i.e. skip the unnecessary sub-problems. When the algorithm is able to demonstrate that a region of the solution space does not contain better solutions than the already known one, called incumbent, the pruning (by optimality) occurs. In a minimization problem, for example, non-integral solutions to LP relaxations serve as lower bounds and integral solutions serve as upper bounds. Then, a node can be pruned if its lower bound is greater than or equal to an existing upper bound.

When the gap, i.e. the difference between the best upperbound (incumbent) and the minimum of the optimal LP objectives (global lower bound), reaches the zero, then the proof of optimality is completed and the algorithm returns the optimal solution.

1.2 Branch-and-Cut

Branch-and-Cut (B&C) is an improved version of the Branch-and-Bound algorithm. It is an hybrid algorithm which combines the B&B decision tree with *cutting planes*: at each node of the decision tree, the formulation of the related LP relaxation can be dynamically enforced through the generation of linear inequalities able to *separate* the optimum from the convex hull of the true feasible set. The aim is twofold: to obtain an integer-feasible solution from the linear relaxation and to obtain a better lower bound, for a more effective pruning.

The branch-and-cut procedure, thus, consists of performing branches and applying cuts at the nodes of the tree:

- A *branch* is the creation of two new nodes from a parent node, as in the B&B. After the branching constraints are added to the model, the two resulting nodes have then distinct solution domains.

- A *cut* is a constraint added to the model. The purpose of adding any cut is to limit the size of the solution domain for the continuous LP problems represented at the nodes, while not eliminating legal integer solutions. The addition of such cuts can yield significant performance improvements, because it reduces the number of branches required to solve the MIP problem.

When solving the LP relaxations, the additional cutting planes generated may be either *global cuts*, i.e., valid for all feasible integer solutions, or *local cuts*, meaning that they are satisfied only by the solution from the currently considered subtree, i.e. the current node and its potential descendents in the B&B search tree.

Global cuts are preferable because they can be stored in a unique global *pool* as they are generated, by a separation procedure, along the search tree. For each new node having a fractional solution, the global cut pool is scanned and the violated cuts are added to the current formulation. The new formulation is then solved and the pool is scanned again, until all the cuts in the pool are satisfied. In case the solution is still fractional, the separator is activated: new violated cuts are generated and inserted into the global pool.

Modern solvers use branch-and-cut search when solving MIP models [FLS10], even though most of their strategies differ completely from each other. When combining the branch-and-cut algorithm with advanced features or proprietary algorithms, as commercial MIP solvers do, the whole optimization process speeds up, even to orders of magnitude.

1.3 Branch-and-cut with callbacks

We present now a variant of the branch-and-cut algorithm based on callbacks.

In general, modern solvers have to solve different types of MIP problems, so the cuts they use are typically cutting planes which can be generated independently of the problem. To give an example, a solver can make use of Chvátal-Gomory cuts, clique cuts, flow cover cuts, flow path cuts and several other cuts. Unfortunately, these cuts are so general that they often cannot exploit the special structure of the problem to be solved. To obviate this limit and exploit all the user prior knowledge, it is necessary to manually find some additional cuts for the specific problem, in a process called *polyhedral analysis*, and apply them to the model in order to improve the performance of the existing branch-and-cut method. Obviously, as the number of cuts can be huge and they cannot be included all into the model, we need to build a *separator* which is a procedure able to generate, when requested, some of the problem-specific cuts, strictly necessary to boost the model optimization.

They only way modern solvers allow to dynamically interact with the optimization process is through *callback procedures*. In fact, *callbacks* allow user code to be executed regularly during an optimization, in order to monitor the process closely or to guide its behavior. According to the area where they are applied, there are several kinds of callbacks. The commercial solver CPLEX, for example, has the *user-cut callback*, which is the one where to put the separator we described above, but there are also the *lazy-constraint callback*, the *heuristic callback*, the *informational callback*, and so on.

The *lazy-constraint callback* is used to add *lazy constraints*, that is, constraints that are not used unless they are violated. As for the cutting planes described above, also the regular constraints can be too many in the model formulation, to be efficiently managed by the solver. For this reason, we do not add all the constraints to the model, but we provide them to the solver only when they are violated.

The *heuristic callback* is used to implement a user custom heuristic that tries to generate a new *incumbent*, i.e. a better integer-feasible solution, from the optimal solution of the LP relaxation at each node of the MIP search tree.

Finally, the *informational callback* is used to retrieve information about the current MIP optimization, e.g. the objective value of the current best integer solution, and can also force the MIP search to terminate.

In general, only few modern MIP solvers support callbacks as they are advanced features and they often demand a low-level understanding of the algorithms used by the solver. Nevertheless, to use callbacks it is sufficient to write the callback function, with the appropriate careful, and then pass it to the solver.

2 A specific MIP problem: UFL

We developed, here, an advanced example for a well-known problem: the Uncapacited Facility Location (UFL) problem. As one can show, the facility location problem is NP-complete. That means, by now, no algorithm known can solve this problem in polynomial time. Therefore, we will present different methods to approach this problem.

2.1 Introduction

Facility location, or k-center problem, is one of the most studied problems in the Operations Research literature.

In our work we describe the traditional formulation and try some solution strategies to proven optimality. Then we use the more efficient Benders formulation and, finally, we implement two well-known matheuristics, Hard Fixing and Local Branching. With the help of a state-of-the-art MIP solver, we perform a computational analysis.

2.2 Uncapacitated Facility Location

The Uncapacitated Facility Location (UFL) problem can be stated as follows. Given a set I of potential facility locations, and a set J of customers, the goal is to find a subset of facility locations to open, and to allocate each customer to a single open facility, so that the resulting fixed cost for the opened facilities plus the resulting cost for customer allocation are minimized.

2.2.1 Traditional problem

In its classical version, the allocation cost for each customer is assumed to be a linear function of the demand served by open facilities. The problem can be formulated as a Mixed-Integer Linear Program (MILP).

Let I be the index set of facility locations ($|I| = n$), let $f_i \geq 0$ be opening costs for each facility $i \in I$, and let J be the index set of customers ($|J| = m$) with allocation costs $c_{ij} \geq 0$ defined for each pair $(i, j) \in I \times J$. We will assume without loss of generality that each customer can be linked to every facility. In this formulation, $n+m \cdot n$ variables are used to model the problem: for each $i \in I$, the binary variable y_i is set to one if facility i is open, otherwise to zero. For each $i \in I$ and for each $j \in J$, the allocation variable x_{ij} is set to one, if customer j is linked to facility i, and to zero otherwise.

$$
\begin{aligned}
\min \quad & \textstyle\sum_{i \in I} f_i y_i + \sum_{i \in I} \sum_{j \in J} c_{ij} x_{ij} && && (1)\\
s.t. \quad & \textstyle\sum_{i \in I} x_{ij} = 1 && \forall j \in J && (2)\\
& x_{ij} \leq y_i && \forall i \in I,\, j \in J && (3)\\
& x_{ij} \geq 0 && \forall i \in I,\, j \in J && (4)\\
& y_i \in \{0, 1\} && \forall i \in I
\end{aligned}
$$

The objective (1) is to minimize the sum of facility opening costs, plus the customer allocation costs. This function is subject to some conditions:

- the constraint (2) make sure that each customer is assigned to exactly one facility,
- the capacity constraints (3) make sure that allocation to a facility i is only possible if this facility is open.

The integrality conditions on variables x_{ij} are redundant in this case: for any integer y^*, each customer j will set $x^*_{ij} = 1$ for the closest facility i with $y^*_i = 1$. For this reason, only non-negative constraints (4) are strictly necessary.

2.2.2 Advanced resolution strategies

Even with state-of-the-art hardware and software, MIP problems can require hours, or even days, of run time and are not guaranteed to yield an optimal solution [KMN13]. Branch-and-bound uses intelligent enumeration to arrive at an optimal solution for a mixed integer program, which involves a construction of a search tree. At each node of the branch-and-bound tree, the algorithm solves a linear programming relaxation of the restricted problem, i.e., the Mixed Integer Problem with all its variables relaxed to be continuous. Due to the exponential growth in the size of such a tree, exhaustive enumeration would quickly become hopelessly computationally expensive for MIPs with even few tens of variables.

The advanced strategies we are going to try for the model resolution are:

- Improve the effectiveness of the branch-and-bound algorithm
- Reduce the input dimension (number of variables and constraints)

The effectiveness of the branch-and-bound algorithm depends on its ability to prune nodes. As the algorithm proceeds, it maintains the incumbent integer feasible solution with the best objective function, which provides an upper

bound on the optimal objective value. In fact, having a better incumbent increases the number of nodes that can be pruned in the search tree. At the same time, as the algorithm proceeds the lower bound is also updated, by using the property that each child node has an objective value no better than that of its parent. Performance is further improved with the cuts added at the nodes, i.e. constraints involving linear expressions of one or more variables, general or specific for some classes of problems.

Another effective strategy is, of course, reducing the model dimension, i.e. the input for the algorithm. We obtain a performance boost, for example, by:

- using less variables in the model formulation
- avoiding to add some constraints to the model (and, as we need to warrant the correctness of the model, we add them as *Lazy constraints*).

All that enhances the throughput of the linear programming relaxations.

Lazy constraints Lazy constraints are constraints not specified in the constraint matrix of the MIP problem, but that must be not be violated in a solution. Using lazy constraints makes sense when there are a large number of constraints that must be satisfied at a solution, but are unlikely to be violated if they are left out.

In our case, we notice that the number of capacity constraints (3) written in the model is huge ($\sim m \cdot n$). For example, with instances of 1000 clients and 1000 facilities, we have a model with more than one million constraints and as much variables. So the trick is not to write them directly to the model, but add them only when needed.

There are two main choices:

- Add, once, all the capacity constraints as lazy constraints and let the solver manage them.
- Write a lazy callback procedure which generates dynamically the violated constraints, but only when needed, i.e. when an integer solution candidate is available and needs to be tested for complete feasibility.

In our implementation, we set a threshold of one million constraints as upper limit for using the first method, i.e. the static one. For instances of higher dimensions, the dynamic method is preferred. It is important to keep in mind that lazy constraints represent a portion of the constraint set: if they are absent for some reasons, the model is incomplete and delivers incorrect answers.

Below, we provide the pseudo-code for the lazy procedure responsible for the recognition and generation of violated constraints, which are then passed to the solver and added to the model.

Algorithm 1 Lazy callback procedure

Input: integer-feasible candidate solution $(\mathbf{x}^*, \mathbf{y}^*)$

Output: set of at most 1000 violated constraints

```
1: count ← 0
2: for all i ∈ I do
3:   for all j ∈ J do
4:     if (x*_ij > y_j + ε) then
5:       /* violated constraint */
6:       "add x*_ij ≤ y_j to the model"
7:       count ← count + 1
8:     end if
9:     if (count = 1000) then
10:      return
11:    end if
12:  end for
13: end for
```

Note the presence of the epsilon term, which is necessary whenever we compare values of type *double* due to precision issues. In this procedure the tolerance threshold must be very small ($\epsilon \approx 10^{-5}$) because we are expecting integer solutions from input.

Unfortunately, using a custom lazy procedure has some side effects. The inner mechanisms of the solver, such as the heuristics responsible for providing a good initial gap, are in a certain sense misled because they cannot see the lazy constraints and whenever they think of having found a good solution, the lazy callback is called and, most likely, rejects it. To compensate the lack of an incumbent solution, we provided a custom primal heuristic procedure, derived from a specific knowledge of the model.

Heuristics Many state-of-the-art optimizers have built-in heuristics to determine initial and improved integer solutions. However, it is always recommended to supply the algorithm with initial solutions, no matter how obvious they may appear to a human. Such a solution may provide a better starting point than what the algorithm can derive on its own, and algorithmic heuristics may perform better in the presence of an initial solution, regardless of

the quality of its objective function value. In addition, the faster progress in the cutoff value associated with the best integer solution may enable the optimizer features such as probing to fix additional variables, further improving performance.

Then, we intend to provide the solver obvious solutions based on specific knowledge of the model. We implemented the following primal heuristic callback procedure: at each branching node a custom rounding heuristic is applied. Given the current LP solution y^*, we order the variables by decreasing values and we start with the opening of the first facility in the ordered list. Then, following that order, we open one more facility at each iteration and evaluate the cost of the integer solution found. This approach requires at most $O(n \cdot \log n + n \cdot m)$ time, where n is the number of facilities and m the number of clients in the model. We provide also a flag which can be enabled in case the number of non-zero values is very high. As we skip all the facilities with a zero value, we can reach a computational complexity of $O(h \cdot \log h + h \cdot m)$, where h is the number of facilities with a non-zero value in the corresponding solution variable of the current LP relaxation.

The pseudocode for the heuristic callback we implemented is written below. The behavior of the algorithm can be changed with the following flags:

$$\text{NONZERO\_FLAG} \leftarrow \begin{cases} 1 & \text{if the number of zero values in the solution is likely to be high} \\ 0 & \text{otherwise} \end{cases}$$

$$\text{ONESOL\_FLAG} \leftarrow \begin{cases} 1 & \text{to interrupt at the first new comer solution available} \\ 0 & \text{otherwise} \end{cases}$$

Algorithm 2 Heuristic callback procedure

Input zopt← cost of the incumbent solution z_{OPT}
ystar ← solution array of the current LP relaxation $\mathbf{y}^*$

Algorithm

```
1: h ← 0
2: for k ← 0 to n − 1 do
3:   if (NONZERO_FLAG = 1) and (List[h].lpValue < ϵ) then
4:     /* skipping zero values */
5:   else
6:     List[h].index = k
7:     List[h].lpValue = ystar[k]
8:     h ← h + 1
9:   end if
10: end for
11: y_ord ← list of facilities with lpValue> 0
12: sort the list by decreasing values of LP relaxation, to obtain:
    List[0].lpValue ≥ List[1].lpValue ≥ ··· ≥ List[h-1].lpValue
13: min ← zopt
14: fsum ← 0
15: for k ← 0 to h − 1 do
16:   /* open the facility in position k of the list */
17:   fsum ← fsum + fixedCosts[List[k].index]
18:   csum ← compute the resulting best costs for the clients
19:   valObj ← fsum + csum
20:   if (valObj + ϵ < min) then
21:     min ← valObj
22:     if (ONESOL_FLAG = 1) then
23:       /* we want only the first better solution */
24:       break
25:     end if
26:   end if
27: end for
28: if (min + ϵ < zopt) then
29:   "new comer solution found!"
30: end if
```

During our experiments, the time spent in the heuristic callback by the solver results negligible (under 1%), compared with the execution of the other procedures. Also, due to this fact, we noticed that it pays more to let the

algorithm look for better solutions and not to interrupt the callback as soon as the first new comer is available. This provides a good upper bound, which is more useful for the node pruning, instead to be satisfied with a quick but minimal improvement.

User cuts While lazy constraints are constraints that the user knows are unlikely to be violated and are applied only when necessary, user cuts may not be strictly necessary to the problem, but they tighten the model. User cuts may be more liberally added to a model because the solver is not obligated to use any of them and can apply its own rules to govern their efficient use. In any case, the formulation of the model remains correct whether or not the cuts are included to the model.

In our case, as the hypothesis of infrequent violation is not actually true, we decided to add the violated constraints also during the resolution of the linear relaxation, with a user cut callback procedure. In this way, we act before the integer constraints are inserted into the model and we can strengthen the linear formulation.

For the classical model, the separation procedure we provided is the same we used for the lazy callback, with the difference that the input solution is fractional. For this reason, the value for the tolerance threshold must be set higher ($\epsilon \approx 10^{-3}$) in order to avoid the risk of numerical instability, which can cause an infinite loop between the LP solver and the callback procedure.

User cuts are useful, for improving the lower bound, only when we are in the higher part of the search tree, but the more we go in depth in the search tree, the more useless they become, or worse, the more they risk to overload the nodes. For this reason, we used a depth criterion to decide whether to generate more cuts or not.

2.3 Benders' Decomposition

Because of the huge number of allocation variables, UFL imposes a challenge also for modern general-purpose MILP solvers. Fortunately, thanks to Benders it is possible to have a very significant performance boost, by removing unnecessary variables and constraints. In Benders formulation, the model is thinned-out and the huge number of allocation variables is replaced by a linear number of continuous variables that model the customer allocation cost directly. For UFL with linear costs, the resulting model involves only $|I| + |J|$ variables and $|J| \cdot (|I| + 1)$ constraints.

2.3.1 Master problem

As x_{ij} variables are a bottleneck for MIP solvers, we just remove them from the model, and introduce in the objective function a new set of continuous variables w_i representing the allocation-cost for all $i \in I$. The resulting master problem is then given by

$$\begin{aligned} \min \quad & \textstyle\sum_{j\in J} f_j y_j + \sum_{i\in I} w_i && && (5)\\ s.t. \quad & \textstyle\sum_{j\in J} y_j \geq 1 && && (6)\\ & w_i \geq \Phi_i(y) && \forall i \in I && (7)\\ & y_j \in \{0,1\} && \forall j \in J \end{aligned}$$

where the convex function $\Phi_i(y)$ appearing in (7) gives the minimum allocation cost for customer i for any given (possibly non-integer) point $y \in [0,1]^J$ where there is at least one facility opened.

2.3.2 Benders cuts

For the linear case, the literature for Benders decomposition (see ref. [FLS15]) shows that the family of Benders cuts is of polynomial size. There are only n distinct cuts, for a given customer i, of the form:

$$w_i \geq c_{ik} - \sum_{j=1}^{k-1} (c_{ik} - c_{ij}) y_j, \ \forall k \in \{1, \ldots, n\}$$

where the locations have been permuted so as to have increasing costs $c_1 \leq \cdots \leq c_n$.

Below, we provide the related pseudo-code for the separator, which can be used in a user cut callback.

We used quicksort for the initial list ordering, which requires $O(|\text{List}| \cdot \log(|\text{List}|))$ time. Then, the overall separation procedure takes at most $O(m \cdot |\text{List}| \cdot \log(|\text{List}|))$ and it is very efficient, in particular when the number of opened facilities is low.

This separator can be used in the MILP branch-and-cut framework, not only by primal heuristics, but also when a candidate integer solution is available. As the solution found always needs to be checked for validity, before updating the incumbent, a lazy cut callback is required in order to certificate its validity or to return one or more violated cuts.

Although we could obtain these cuts by applying the same separation procedure, we prefer to write a more efficient procedure for the so-called lazy callback. In this case the candidate solution is integer and every facility is

Algorithm 3 Benders separation procedure

Input node relaxation continuous solution $(\mathbf{x}^*, \mathbf{y}^*)$

List $\leftarrow$ list of non-zero y variables $\{ j : y_j^* > \epsilon \}$
for all $i \in I$ **do**
  "sort List by increasing link costs, so that $c_{i,1} \leq c_{i,2} \cdots \leq c_{i,|List|}$"
  j $\leftarrow 1$
  ysum $\leftarrow 0.0$
  for $h \leftarrow 1$ to |List| **do**
    ysum $\leftarrow$ ysum + ystar[List[h]]
    if ysum $\geq 1.0 - \epsilon$ **then**
      break
    end if
  end for
  k $\leftarrow$ List[h] /* facility where the most violated constraint is */
  viol $\leftarrow c_{ik} - w_i^*$
  for all $j \in$ List **do**
    if $c_{ij} < c_{ik}$ **then**
      viol $\leftarrow$ viol - $(c_{ik} - c_{ij})y_j^*$
    end if
  end for
  if viol $> \epsilon$ **then**
    /* violated constraint */
    "add $w_i + \sum_{j:c_{ij}<c_{ik}}(c_{ik} - c_{ij})y_j \geq c_{ik}$ to the model"
  end if
end for

either open or closed, so there is no need for the initial list sorting. In fact, it is easy to find the facility where the most violated constraint is, for each client, by simply looking for the minimum link cost between the selected facilities. Then, for each client i, if the value of w_i is lower than the actual minimum cost, a violated constraint is generated and added to the model.

2.4 Matheuristics

In case we are not interested in a proven optimality solution, but we are looking only for a good solution, it is sufficient to run some heuristics. Unlike the other search methods, such as Tabu Search or Greedy, Matheuristics are model-based meta-heuristics which exploit the existing mathematical programming model. In general, the solution (math-)heuristics provide could be optimal or not, but in all cases this fact is not certified by any mathematical proof. Also, this kind of algorithms are usually faster in finding good feasible solutions but, in the long term, they perform worse than the MIP solver and, possibly, they could never reach the optimality.

2.4.1 Variable fixing (diving)

Hard variable fixing, or diving, is a matheuristic which performs local search around a given integer-feasible solution. Assuming to have an exact black-box solver for our problem, we fix some variables to their values and apply it iteratively on the resulting restricted problem. Fixing the binary variables strongly simplifies the problem, thus each iteration is very fast. Anyway, we imposed to each sub-problem a time-limit, e.g. 30 seconds, and a node-limit, e.g. 10 nodes, in order not to lose too much time on a wrong inference.

As the algorithm requires a starting solution for the diving, before calling our matheuristic, we decided to run the solver and interrupt it as soon as an integer-feasible solution is found. Another possibility is to keep the solver running, until some target gap from upper and lower bound is reached (e.g. less than 20%).

The pseudo-code for the procedure we implemented is written below. The behaviour of the algorithm depends on the following parameters:

N_ITER the number of hard-fixing iterations

HARDFIX_THRESHOLD the probability used to fix each variable

USE_BEST_SOL $\begin{cases} 1 & \text{to keep updated the reference with the best solution found} \\ 0 & \text{to use always the same solution, found at the root node} \end{cases}$

FIX_ONLY_ZEROS $\begin{cases} 1 & \text{to avoid the fixing of one-valued variables} \\ 0 & \text{otherwise} \end{cases}$

Note that as we unfix the variables from the model at each iteration, the solver will not use the same search tree. Also, when passing to the solver a solution previously found, pay attention that it can result infeasible, as the

Algorithm 4 Hard fixing procedure

$\mathbf{y}^* \leftarrow$ "get reference solution"
$z_{opt} \leftarrow$ "get reference obj" /* best current obj */
$\mathbf{y}_{opt} \leftarrow \mathbf{y}^*$ /* best current solution */
for k $\leftarrow$ 1 to N_ITER **do**
  for all $y_j \in J$ **do**
    if (FIX_ONLY_ZEROS $= 1 \wedge y_j^* > \epsilon$) **then**
      /* skip variables with value equal to one */
    else
      if (random($[0, 1]$) $<$ HARDFIX_THRESHOLD) **then**
        "fix y_j to y_j^*, in the model"
      end if
    end if
  end for
  $(z', \mathbf{y}') \leftarrow$"optimize the model"
  if $(z' < z_{opt})$ **then**
    $z_{opt} \leftarrow z'$
    $\mathbf{y}_{opt} \leftarrow \mathbf{y}'$
    if (USE_BEST_SOL $= 1$) **then**
      /* start from the best solution found */
      $\mathbf{y}^* \leftarrow \mathbf{y}'$
    end if
  end if
  "unfix all variables, from the model"
end for

solution space can be different for every restricted sub-problem. Anyway, it is always possible to set the best incumbent as a cutoff parameter, to let the solver know the results reached in previous runs.

2.4.2 Local branching

Local branching is another type of matheuristic, described in [FL03], which is based on local search starting from a given integer-feasible solution. Differently from Hard Fixing, Local branching has a definition of neighborhood based on the so-called Hamming distance. In fact, the problem is simplified by restricting the maximum number of binary variables flipping, k, from a starting reference solution, $\widetilde{\mathbf{x}}$, around which the local search is performed. In case the search is not successful, the neighborhood is widened by incrementing the parameter k, and process is re-iterated until a better integer-feasible solution is found, or a range-limit for k is reached.

The local branching constraint has the following form:

$$\Delta(\mathbf{x}, \tilde{\mathbf{x}}) \triangleq \underbrace{\sum_{j:\tilde{x}_j=0} x_j}_{\widetilde{x}_j=0\to 1} + \underbrace{\sum_{j:\tilde{x}_j=1} (1 - x_j)}_{\widetilde{x}_j=1\to 0} \leq k$$

Note that we are adding to the model a dense constraint, i.e. every variable has a nonzero coefficient. The procedure we implemented is written below, while the available parameters are:

kMIN minimum value for k

kMAX maximum value for k, before aborting the search

kDELTA increment for k, in case no better solution is found

Note that this method highly depends on the reference solution and, also, from the choice of k. In fact, the value for k must be sufficiently small to be optimized within short computing time, but still large enough to likely contain a better solution than the reference one. Experimental results shows that it is better to stay within a range of [5,20], to better exploit the effectiveness of this method.

Algorithm 5 Local branching procedure

Input $\mathbf{x}_{opt}$ best incumbent solution
z_{opt} objective value of best incumbent solution

```
1: k ← kMIN
2: x̃ ← x_opt
3: while ( k ≤ kMAX ) do
4:    "add local branching constraint, using (x̃, k)"
5:    "optimize the model" /* black-box solver */
6:    z* ←get optimal obj value
7:    if ( z* < z_opt ) then
8:       z_opt ← z*
9:       x̃ ← get optimal solution
10:      /* reset the parameter k */
11:      k ← kMIN
12:   else
13:      k ← k+ kDELTA
14:   end if
15:   "remove local branching constraint"
16: end while
```

2.5 Computational results

All codes were executed on a 2.8 GHz Intel Core 2 Duo CPU equipped with IBM ILOG Cplex 12.6.1 as a MIP solver. All codes use callback functions, thus deactivating IBM ILOG Cplex 's proprietary dynamic search. Experiments were performed in multi-thread mode, with 4 concurrent threads running on the same PC, and with deterministic mode enabled. Computing times can be expressed in CPU clocks (seconds), memory accesses (ticks) or wall-clock real-time (seconds).

The execution has been inspected by a time profiler tool. Most of the resources are used by Cplex routines, instead only a small amount of CPU time slices are occupied by the callback routines we provided. In particular, for the Benders model formulation, the heuristic callback function took about less than 1% of the total execution time, while the separator procedure (user cut callback) and the lazy callback took less than 5%.

2.5.1 Instances

Our algorithms has been computationally tested on a large set of MIP instances from the UflLib library by Max-Planck-Institut Informatik.

For the benchmark, we randomly selected 18 instances, 6 for each package (K-median, Koerkel-Ghosh, M). In particular,

- from K-median: 1000-10, 1500-10, 2000-10, 2500-10, 3000-10, 500-10;
- from M: MO5, MP4, MQ3, MR2, MS1, MT1;
- from Koerkel-Ghosh asymmetric: ga250b-2, ga500a-2, ga750c-4;
- from Koerkel-Ghosh symmetric: gs250a-1, gs500b-1, gs750c-3.

2.5.2 Software interface

When building an optimization model in a modeling language, it is typical to separate the optimization model itself from the input data. This allows to run different instances of the same model, without having to re-compile all the software again and again. As the benchmark libraries are provided within data files, we wrote a simple parser. According to the format, the input file is parsed in order to populate the structure we defined.

Data file format The instances are stored in two basic formats: ORLIB-cap and Simple format.

For ORLIB-cap, the first line of a file consists n and m: [n] [m], assuming that n is the number of facilities and m the number of clients. Then the next n lines consist of the capacity and the opening cost of the corresponding facility. Finally, there are the demand for each client and the cost of the connections to all facilities.

```
[n][m]
// for each facility j: (j = 1 ,..., n)
[capacity] [opening cost]
// for each client i (i = 1, ... ,m)
[demand of i] [cost of conneting client i to facility j] (j =
    1,...,n)
```

Simple format, instead, is only suitable for instances of the uncapacitated facility location problem. The first line consists of the text 'FILE: ' and the name of the file. In the second line n, m and 0 are denoted. The next n lines consist of the number of the facility, the opening cost and the connection cost to the clients.

```
[n][m]0
// for each facility j (j = 1, ... , n)
[j] [opening cost] [cost of conneting client i to facility j] (i
    = 1, ... ,m)
```

Data structure We defined a struct in C-language as:

```
struct Instance {
        long n_facilities;
        long n_clients;
        double *fixed_costs;
        double *costs; // c[i,j] = inst->costs[i*inst->
            n_facilities+j]
};
```

2.5.3 Solving to proven optimality

On Table 1, we provide the results of our computations using the Benders formulation. We set a maximum execution time limit of 30 minutes and run the solver to our randomly selected instances.

For each instance we report in Table 1 the name of the instance (name), the size in terms of facilities (n) and of clients (m), the best solution found (opt), the total wall-clock time (in seconds), the final gap between upper- and lower bound (g), the time spent to solve the root node (t_{root}), the gap after solving root node (g_r), the lower bound after solving root node (rootbound), the total number of Branch&Bound nodes solved (nodes), the total number of memory accesses in ticks (total det time) and, finally, the optimization status at the end of the computation (status).

We highlighted in boldface the instances for which we had found an optimal solution, or a better upper bound, not yet available in the UflLib library's website. Actually, literature showed that most of these instances had been solved to optimality afterwards.

name	n, m	opt	total time	$g[\%]$	$t_{root}[s]$	$g_r[\%]$	rootbound	nodes	total det time	status
1000-10	1000	1434154.0000	65 s	<0.01	19.73	0.85	1429271.5266	632	13521.9950	OPTIMAL
1500-10	1500	2000801.0000	296 s	<0.01	24.35	2.32	1982195.9143	1157	63777.7977	OPTIMAL
2000-10	2000	2558125.0000	1174 s	<0.01	35.48	4.09	2516533.8569	2973	212263.5308	OPTIMAL
2500-10	2500	**3101567.0000**	>1800 s	0.15	336.38	0.72	3096372.7736	3728	368759.4982	TIME_LIMIT
3000-10	3000	3584477.0000	>1800 s	3.88	53.77	8.37	3395705.3774	1919	340050.5026	TIME_LIMIT
500-10	500	798577.0000	6.19 s	<0.01	2.93	2.22	790462.5972	151	2110.3827	OPTIMAL
MO5	100	1408.7664	0.56 s	<0.01	0.21	3.71	1356.4685	72	85.6388	OPTIMAL
MP4	200	2938.7500	2.76 s	<0.01	0.87	4.92	2794.2860	320	1162.7881	OPTIMAL
MQ3	300	4275.4317	3.40 s	0.00	1.79	3.71	4116.7052	137	1702.5024	OPTIMAL
MR2	500	**2654.7347**	20.58 s	0.00	6.68	5.59	2506.3659	298	7928.9240	OPTIMAL
MS1	1000	**5283.7574**	274.29 s	0.00	32.87	6.83	4922.8181	1448	73487.3358	OPTIMAL
MT1	2000	**10069.8028**	>1800 s	7.17	93.17	9.50	9130.7228	4202	364631.3486	TIME_LIMIT
ga250b-2	250	**275141.0000**	434.08 s	<0.01	2.75	1.32	272351.3236	45635	129750.9048	OPTIMAL
ga500a-2	500	511671.0000	>1800 s	0.24	3.29	27.70	373067.8219	43376	424387.8592	TIME_LIMIT
ga750c-4	750	901634.0000	>1800 s	2.49	49.84	2.90	875448.7759	12313	362380.4758	TIME_LIMIT
gs250a-1	250	257964.0000	>1800 s	0.02	0.62	28.32	186991.8398	246348	513937.3713	TIME_LIMIT
gs500b-1	500	538439.0000	>1800 s	0.82	26.20	1.20	532861.4677	34185	431669.6462	TIME_LIMIT
gs750c-3	750	902240.0000	>1800 s	2.65	43.21	3.06	874607.0401	11670	360320.1308	TIME_LIMIT

Table 1: Solving selected instances to proven optimality with Benders

2.5.4 Heuristic solutions

We present, in Figure 1, some runs of Hard Fixing matheuristic using 3 different random seeds. The execution is limited to 10 iterations, where the first iteration starts when the MIP solver finds a integer-feasible solution with a gap of about 20% (from the lower bound) or the root node is solved. The random fixings are made only for zero-values with a probability of 85%, using the best incumbent solution. The gap in the plots is related to the best known solution, while on x-axis there is the number of hardfix iteration. We note an high-sensitivity to initial conditions, in particular at the first iterations, while in the last iterations the diverse runs tend to converge to a common result.

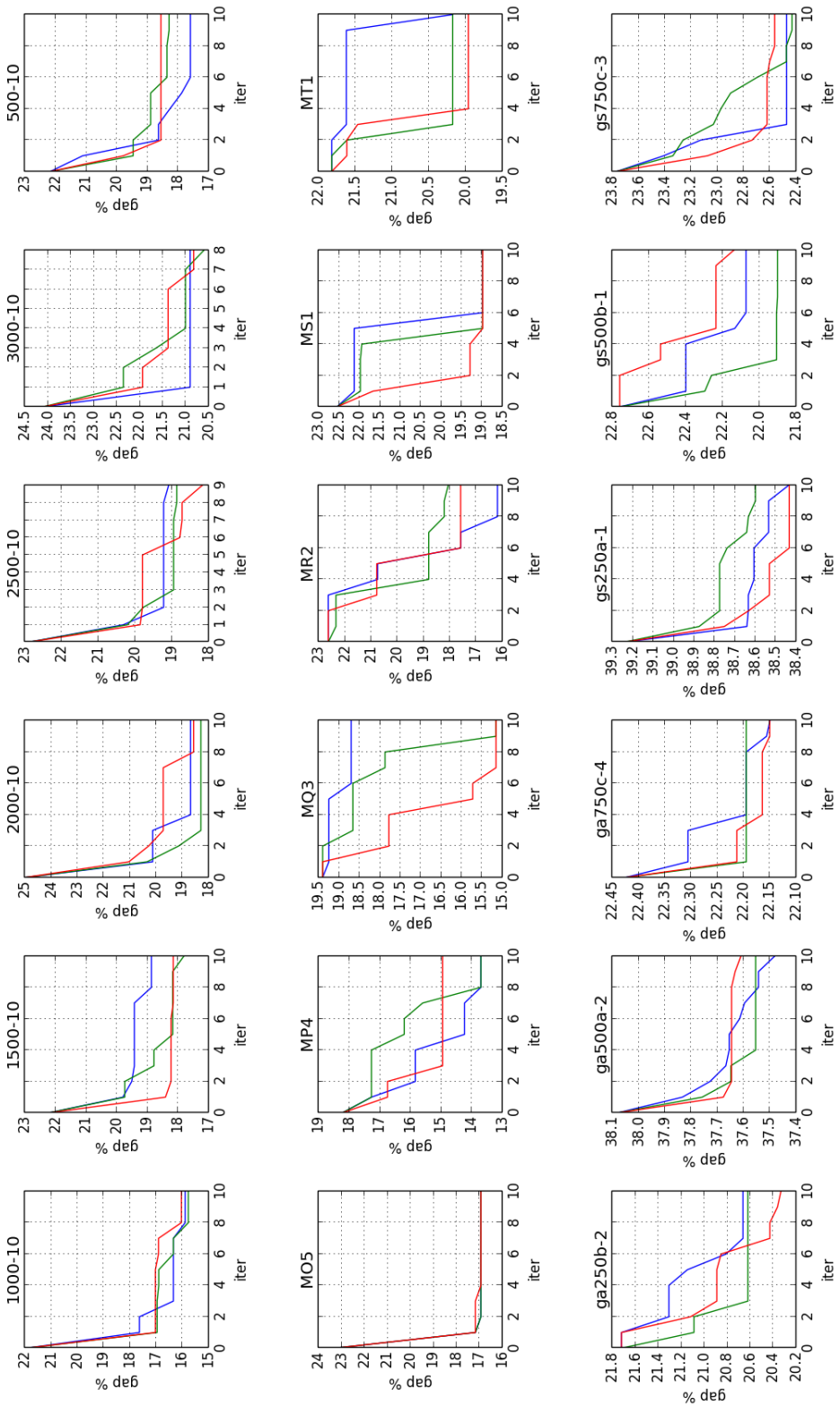


Figure 1: Hard Fixing runs with 3 different random seeds

Here, instead, we present the performance of Local Branching matheuristic at each iteration. The total time limit is set to 15 minutes and the first iteration starts when the MIP solver finds a integer-feasible solution with a gap of about 20% from the lower bound or the root node is solved. In the plots of Figure 2, we note a less chaotic behaviour than diving and a very good performance since the early iterations. As the gap plotted on y-axis is related to the best known upper bound, a negative value means that a better solution has been found by our matheuristic.

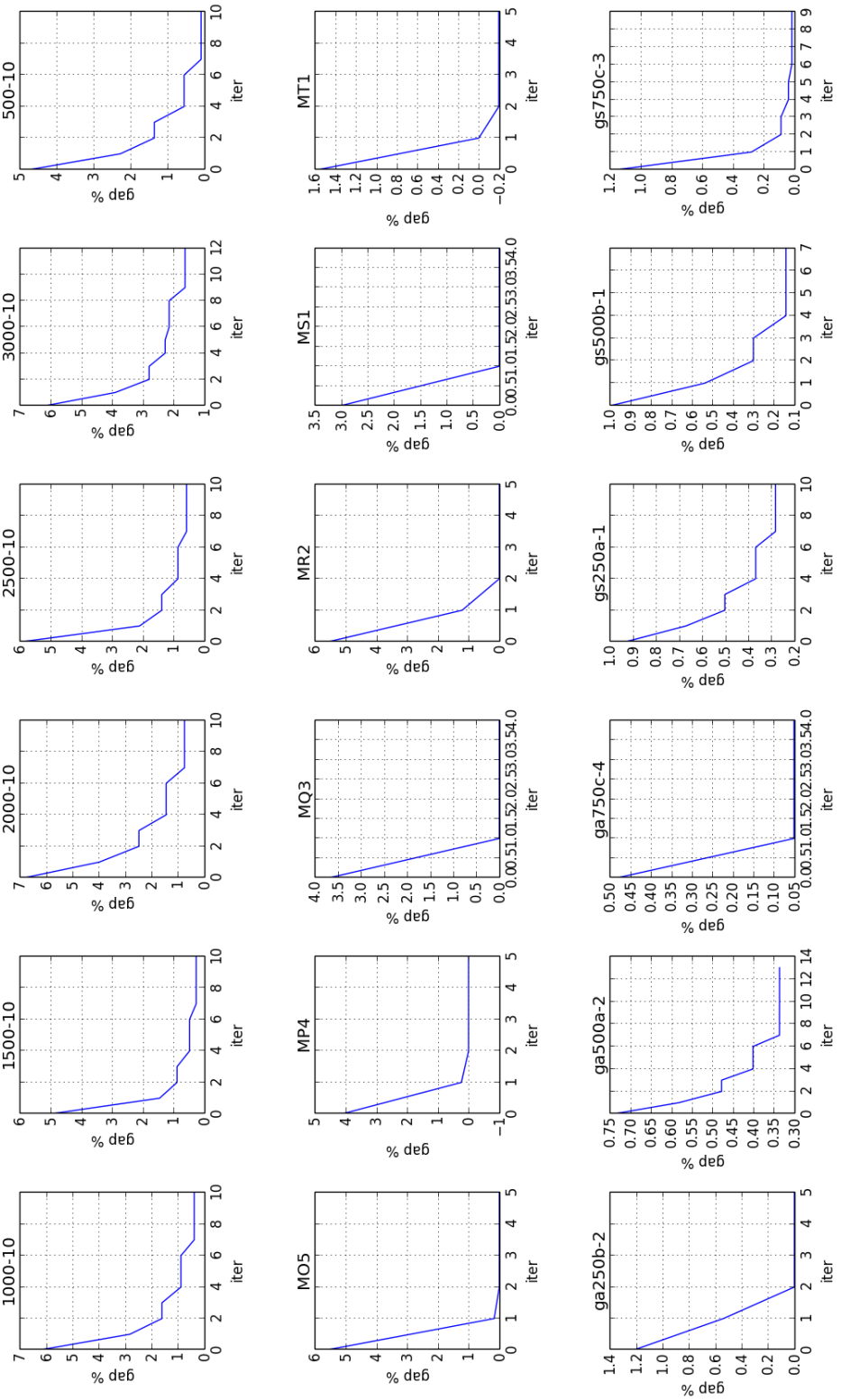


Figure 2: Local Branching runs

For completeness, we provide in Figure 3 a comparison between Hard Fixing and Local Branching in terms of resources used. They both start from the first solution found at the root node, having a gap (from the lower bound) lower than 20%. The plots show how much CPU time has been used in order to find better solutions. In the first phases, it seems that Local Branching is more efficient, while Hard Fixing seems wasting a lot of resources, maybe because of a bad inference taken. Instead, in the long term, the probability for the diving of finding a better solution is higher and grows more and more, thanks to randomization. Anyway, due to the erratic behavior of MIP solvers (see ref. [FM14]) we remark the fact that we cannot draw further conclusions from these plots.

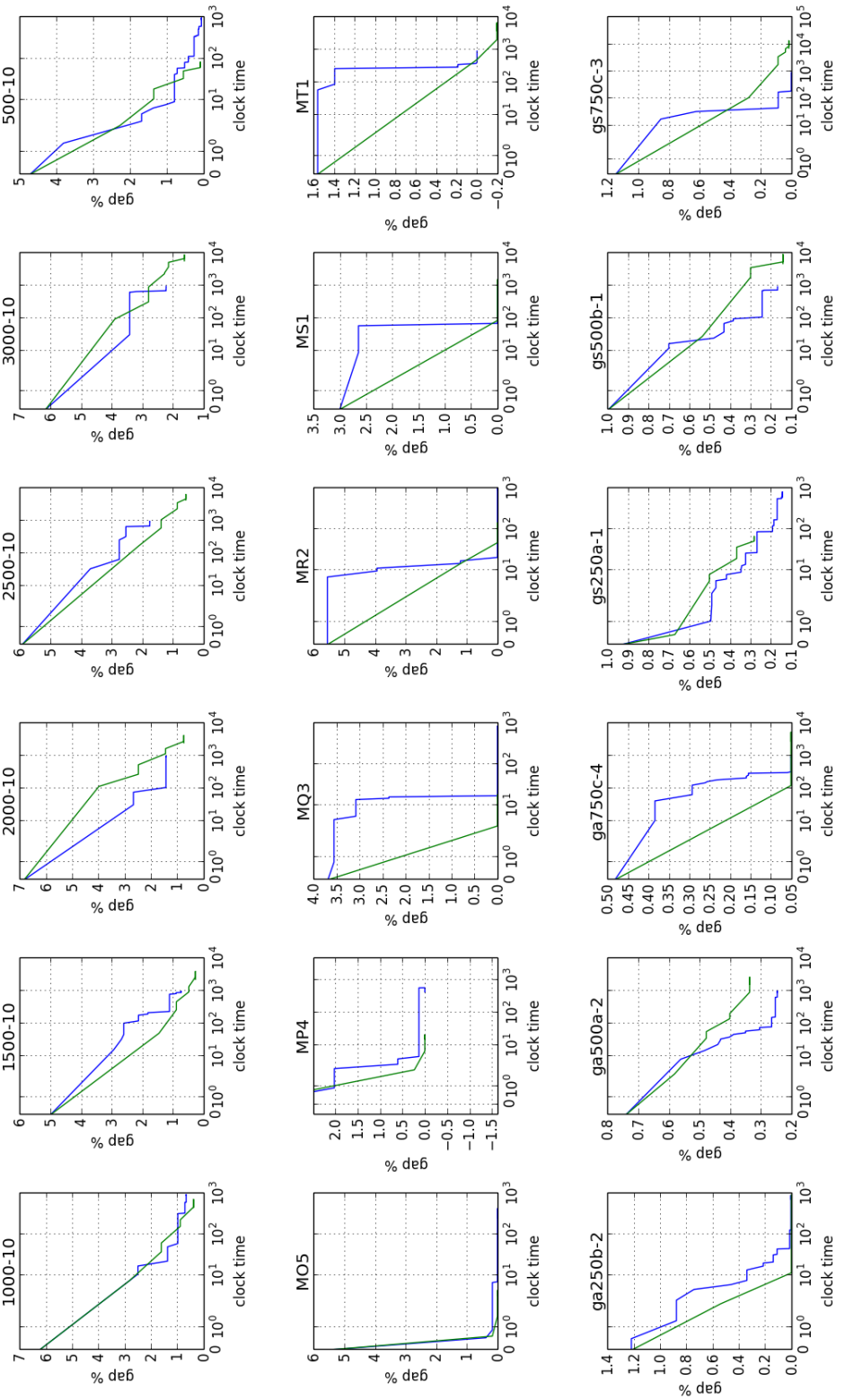


Figure 3: Matheuristics comparison: Local Branch (green) and Hard Fixing (blue)

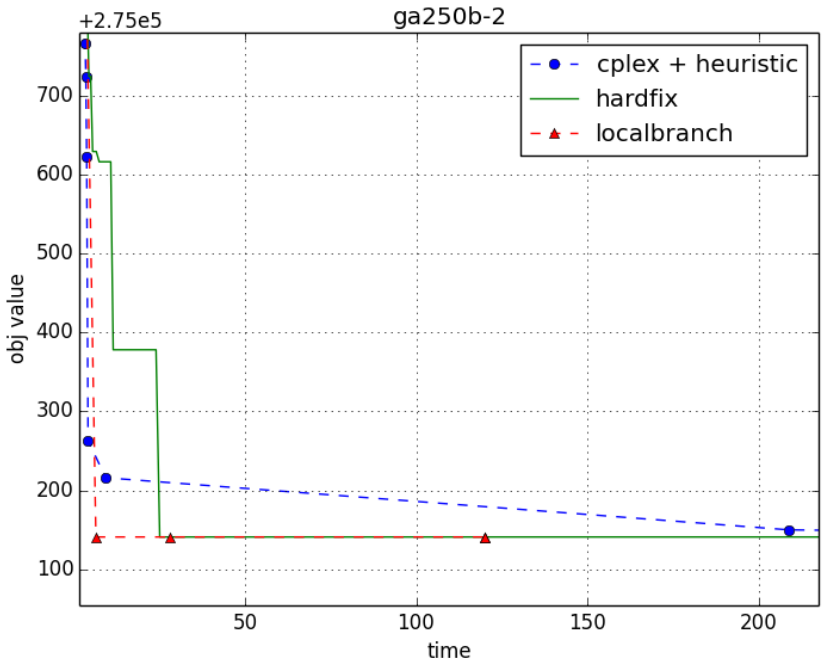


Figure 4: Proven optimality vs matheuristics for instance "ga250b-2"

Finally in Figure 4, in we compare our black-box MIP solver with the two matheuristics, Local Branch and Hard Fixing. We chose only one instance to show the common behaviour of matheuristics compared to the proven optimality runs of the previous Section. To have a fair comparison, they all start from the same solution, after the root node has been solved.

3 An interface for Gurobi

In the previous section, we developed an advanced example of a MIP problem and tested it on an modern MIP solver. Unfortunately, not all available solvers can be used to implement our example, because we used some advanced features that only some commercial solvers supports (e.g. callbacks). Our initial purpose was to make also a comparison with another commercial solver, for example Gurobi, which has almost equivalent performance.

In general, we did not want to rewrite the whole software we had already developed once, every time we intended to migrate into another product. Specifically, we wanted to test our example also with Gurobi solver, without having to maintain two different versions of the UFL, for example reusing the same code working with the CPLEX APIs.

To reach our goal, we first present some existing solutions dealing with questions of portability, and we outline their limits, too. Then, we present a completely different solution we considered the most suitable for our necessities, describing accurately how we designed and implemented it. Finally, we sum up in few points the main differences between CPLEX and Gurobi API interfaces we encountered.

3.1 Some available solutions

The state of the art of numerical computing is characterized by two main classes of languages: the highly efficient low-level languages (e.g. C, C++, Fortran) and the highly expressive high-level languages (such as Python, R, Matlab, etc.). While the interpreted languages are usually platform and solver independent, C and C++ are still preferable for many reasons:

- they are compiled offline, producing extremely efficient machine code;
- they have strict variable typing, which allows advanced optimization of the code.

Between these two different classes, some researchers developed some hybrid frameworks, in order to capture the main advantages from both of them. Their aim is to have the expressiveness, the ease of use and the solver independence of modern scripting languages together with the performance of lower-level languages.

Open Solver Interface (OSI) Open Solver Interface (Osi) provides an abstract base class to a generic linear programming (LP) solver, along with derived classes for specific solvers. Osi is written in C++ and is released

as open source code as part of the COIN-OR initiative [COIN]. Thanks to the derived classes, Osi interacts with different solvers (Cbc, Clp, CPLEX, DyLP, GLPK, Gurobi, MOSEK, SoPlex, SYMPHONY, Vol, XPRESS). At the current version (0.107.5), the OSI supports linear programming solvers, but unfortunately it has rudimentary support for integer programming. In particular, OSI does not support callback procedures, neither other advanced features.

Julia language (JuMP) JuMP (Julia for Mathematical Programming) combines the speed of commercial products with the benefits of remaining within a fully-functional high-level modern language [JUL]. Extensive cross-language benchmarks suggest that Julia is capable of obtaining state-of-the-art performance [LD15]. The language uses a just-in-time (JIT) compiler which generates efficient code. Thanks to its macros, evaluated only at compile time, there is no runtime overhead.

- JuMP supports advanced features such as efficient LP hotstarts and branch & bound callbacks.
- The language provides wrappers around high-performance low-level C libraries of different solvers.

Anyway, we understood that the aim of JuliaMP is a little different, which is to build the final model as fast as possible. Instead, our main purpose is to neglect the performance in the model building phase and to obtain the best solver performance for the optimization.

For these reasons, our choice remains constrained to a low-level language we already know and which allows advanced solver features, such as callback procedures.

3.2 The idea: a direct interface Cplex-to-Gurobi

We chose, then, to combine our UFL code, written for IBM ILOG Cplex 12.6.1, with the solver and the functionalities of Gurobi 6.0.3. Our intention was not to modify any line of the original code, but to write an interface able to capture each function call to CPLEX and redirect it to Gurobi.

We have re-implemented all the most popular Cplex functions from the C API, including the advanced ones used by our UFL example, using the C functions available from Gurobi low-level library. For some functions the work was pretty easy (because Gurobi has a style very similar to CPLEX), but in many occasions the lack of an appropriate mapping between the functions required an accurate study of solver-specific behaviours and a complex combination of its functionalities.

Anyway, we obtained a great simulation of CPLEX and we characterized all the limits of our interface. The result is very good because all the UFL code works and the performance respects the powerful of the solver. We can say that our interface is "transparent", because each function does the mapping with a small overhead.

Our interface is made by two files: *cpx2grb.h* , where the status codes, the function prototypes of CPLEX and the types of object are redefined; *cpx2grb.cpp* where all the CPLEX functions are implemented. The language used is C++, the same as the UFL code, but we call Gurobi solver only through its C API library, because it is the lowest layer available. In our UFL code, we substituted only the inclusion of CPLEX interface "cplex.h" with "cpx2grb.h" and the *CMakeLists.txt* file, for Cmake, to compile also our files. In general, the changes required to the code are minimal and, in this case, we did not need to change any other thing.

3.2.1 Interface development

The development of our interface required a working environment, equipped with CPLEX 12.6.1 and Gurobi 6.0.3. We chose Mac OS 10.10 as operating system because we used Xcode, a freeware integrated development environment (IDE) provided by Apple Inc., which includes a lot of tools for debugging and profiling the code. For managing the build process of our software we used a compiler-independent method, thanks to CMake 3.2.0-rc2 [CMK].

We analyzed, then, all the documentation, manuals and guides provided by each solver [CPX, GRB], for a short feasibility study. We were aware that Gurobi and CPLEX used different algorithms and strategies, so we did not expect to find a perfect and complete matching between the two solvers. Nevertheless, we realized that Gurobi and CPLEX shared a lot of features and an interface development was possible.

The implementation started only after a quick design of the common features we were able to map. We opted for a "step-by-step agile development": for each function, we first wrote all the possible tests coding significant examples and a lot of *asserts*, that is a test for an expected result of an expression. During this phase, it is important to disable all the internal heuristics, cuts and presolver of Gurobi. Then, we implemented the code and executed it, step-by-step. In this way, the whole coding required less time, because the purpose of each routine was immediately clear when writing the tests and only the strictly necessary code was inserted. Also, no additional tests were required as we had, obviously, already written them.

In the following chapters, we will accurately describe the method of preliminary design we adopted and, also, the main choices we took, before and during our interface implementation.

3.2.2 How to match functions

In order to find the most appropriate matching between CPLEX and Gurobi routines, we first compared the prototypes and the description of each function, from the provided documentation of each solver. Then we listed all the features of each routine and we tried to find the most promising correspondences. For each couple of functions found, we rated their degree of equivalence in a scale of "Good", "Limited" and "Not Available". The Table below shows the above-mentioned results of our search. For some CPLEX features, the Gurobi equivalent is an "*Attribute*", which means it is an internal property of a Gurobi model and can be accessed only through the appropriate attribute-management routines.

CPLEX Function	Gurobi equivalent	Mapping
CPXaddlazyconstraints	*Attribute:* Lazy	Good
CPXaddrows	GRBaddvars	Good
CPXbranchcallbackbranchbds	N/A	N/A
CPXchgbds	*Attribute:* LB, UB	Good
CPXchgcoef	GRBchgcoeffs	Good (*)
CPXchgobj	*Attribute:* Obj	Good
CPXchgqpcoef	GRBaddqconstr, GRBaddqpterms	N/A
CPXchgrhs	*Attribute:* RHS	Good
CPXcloseCPLEX	GRBfreeenv	Good
CPXcopymipstart	*Attribute:* Start	Limited
CPXcopyorder	*Attribute:* BranchPriority	Limited
CPXcreateprob	GRBnewmodel	Good
CPXcutcallbackadd	GRBcbcut, GRBcblazy	Good
CPXdelmipstarts	*Attribute:* Start	Limited
CPXdelrows	GRBdelconstrs	Good
CPXfreeprob	GRBfreemodel	Good
CPXgetcallbackinfo	GRBcbget	Limited
CPXgetcallbacknodex	GRBcbget	Good
CPXgetctype	*Attribute:* VType	Good
CPXgetdblparam	GRBgetdblparam	Good (*)
CPXgetintparam	GRBgetintparam	Good (*)
CPXgetlb	*Attribute:* LB	Good
CPXgetnumcols	*Attribute:* NumVars	Good
CPXgetnumcores	(not exists)	Good (*)
CPXgetnummipstarts	*Attribute:* Start	Limited
CPXgetnumrows	*Attribute:* NumConstrs	Good
CPXgetobj	*Attribute:* OBJ	Good
CPXgetobjval	*Attribute:* ObjVal	Good
CPXgetstat	*Attribute:* Status	Limited

CPLEX Function	Gurobi equivalent	Mapping
CPXgetub	*Attribute:* UB	Good
CPXgetx	*Attribute:* X, Xn	Good
CPXmipopt	GRBoptimize	Good
CPXnewcols	GRBaddvars	Good
CPXnewrows	(see Table matrix)	Good
CPXopenCPLEX	GRBloadenv	Good
CPXordwrite	GRBwrite	Good
CPXreadcopyorder	GRBread	Good
CPXsetbranchcallbackfunc	N/A	N/A
CPXsetdblparam	GRBsetdblparam	Good (*)
CPXsetintparam	GRBsetintparam	Good (*)
CPXsetlazyconstraintcallbackfunc	GRBsetcallbackfunc	Good (*)
CPXsetusercutcallbackfunc	GRBsetcallbackfunc	Good (*)
CPXsolwrite	GRBwrite	Good
CPXwritemipstarts	GRBwrite	Limited
CPXwriteparam	GRBwrite	Limited
CPXwriteprob	GRBwrite	Good
CPXdelcols	GRBdelvars	Good
CPXlpopt	GRBoptimize	Good
CPXgetbestobjval	*Attribute:* ObjBound	Good
CPXgettime	*Attribute:* Runtime	Good (*)
CPXgetnodecnt	*Attribute:* NodeCount	Good
CPXaddmipstarts	*Attribute:* Start	Limited
CPXgetheuristiccallbackfunc	GRBgetcallbackfunc	Good (*)
CPXgetinfocallbackfunc	GRBgetcallbackfunc	Good (*)
CPXgetusercutcallbackfunc	GRBgetcallbackfunc	Good (*)
CPXsetheuristiccallback	GRBsetcallbackfunc	Good (*)
CPXsetinfocallbackfunc	GRBsetcallbackfunc	Good (*)
CPXsetlazyconstraintcallback	GRBsetcallbackfunc	Good (*)
		(*) using a workaround.

In case of multiple candidates for the matching, we selected the Gurobi functions able to cover all the features of the Cplex routine to implement. As an example, we reported on a table the different operations each function is able to perform. The functions for Gurobi (GRB) and Cplex (CPX) solvers are along rows, while the associated operations are along columns. The Batch column indicates whether the routine can perform multiple insertions (∞) or only a single insertion (1), with the same user call.

In this case, the implementation of *CPXnewrows* will require more than a Gurobi routine: *GRBaddconstrs* and *GRBaddrangeconstrs*. But also *GRBaddconstr* and *GRBaddrangeconstr* are good, because they differ only for the number of insertions, single or multiple, the user can do, with one call, to the routine.

In general, when two Gurobi routines can be reduced one to the other, we should prefer the most general one (the multiple-insertion version, in the previous example). This is not only to decrease the number of calls to the solver, but to limit the inconsistent states in case the solver raises an error while executing our routine. In fact, we do not want our interface to be interrupted in the middle of a loop of single calls to the solver, because of an error. This situation would be unrecoverable and has to be managed accordingly, for example with transactions.

API Function	Batch	Set RHS	Set row coeff	Not ranged	Ranged	Set rngval	Set var coeff	Add vars
CPXnewrows	∞	X		X	X	X		
CPXaddrows	∞	X	X	X	X			X
GRBaddconstr	1	X	X	X				
GRBaddconstrs	∞	X	X	X				
GRBaddrangeconstr	1		X		X	X		
GRBaddrangeconstrs	∞		X		X	X		
GRBaddvar	1						X	X
GRBaddvars	∞						X	X

Table 2: Comparison matrix for variables and constraints insertion

3.2.3 How to implement functions

After having grouped all the necessary elements to build each CPLEX function, the implementation of our mapping functions is ready to begin.

The overarching goal of our interface is to map from Cplex to Gurobi all the input the user provides, and to map from Gurobi to Cplex all the output the solver produces. Writing an appropriate mapping is sometimes not hard work, especially when involved objects, or data structures passed through function arguments, are isolated enough across the functions. Yet, in many occasions, the context in which these data elements operate may be more complex and requires an accurate design of the interface scope.

Parameter functions As an example, we consider both the following get/set parameter functions:

> int CPXsetintparam (CPXENVptr env, int whichparam, CPXINT newvalue)
>
> int CPXgetintparam (CPXCENVptr env, int whichparam, CPXINT *value_p)

In this case, all the elements in *input* are the environments, the names of the parameters and the values to store. At the same time, all the elements in *output* are the values retrieved and the return status of the routines. We note that the values of CPLEX parameters are considered both in *input* and in *output*, which means that a one-to-one mapping for them is required to preserve the full compatibility.

Data elements	Input	Output	Mapping sense
Environment	X		CPX→GRB
Parameter names	X		CPX→GRB
Parameter values	X	X	CPX↔GRB
Return status		X	CPX←GRB

Table 3: Type of mapping required for data objects

The design of our example of solution continues below and it is organized as follows:

- As the way CPLEX manages the environment objects is different from how Gurobi does, we will outline in the paragraph *Environment consistency* how to handle this circumstance.
- Even the strategies and algorithms Gurobi uses often differ from the ones of CPLEX. We do not expect to obtain a complete mapping of the parameters, but there are some common parameters the two solvers share and that we can try to match, in the paragraph *Parameter mapping.*
- Obviously, a formal bijection of all the values is not always possible, for example because of the *int-double* type conversions. Thus, we will delineate these kind of limits in paragraph *Parameter value mapping.*
- Finally, in the last paragraph we will manage the reverse mapping of the return status from the procedures.

Environment consistency The environment is a data structure where to create model objects and in which are stored parameter settings and other solver variables. An instance of the environment is always passed as an argument in almost all CPLEX routines.

In CPLEX, all models use the same parameter values of their *parent* environment. Gurobi, instead, gives each model its own copy of a Gurobi environment and allows each model to have its own parameter settings. It means that when the user, for example, changes a parameter in the *parent* environment through the setter function *CPXsetintparam*, the new value must be seen from all the problems associated with that environment, and not stored only on a single copy of it.

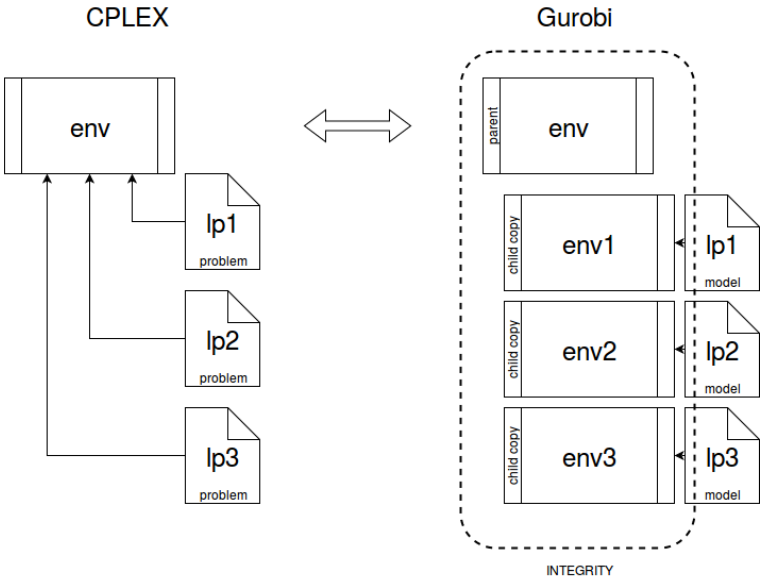


Figure 5: CPLEX and Gurobi environments

To maintain the compatibility with Cplex, our interface must warrant the consistency between the different models belonging to the same environment. To achieve this goal, our interface needs to preserve the following invariant for the user:

> "*In every moment, all the models created from the same parent environment have the same parameter values and the same callback functions*.".

We also defined an internal data structure where storing, for each CPLEX environment, an entry containing: a pointer to the Gurobi parent environment, a list of pointers of the Gurobi models associated to the environment,

the function pointers to the callbacks and the pointers to the callback user-data. A new entry will be created together with the opening of a new parent environment.

To preserve the invariant stated above, our interface must perform some steps before returning the control to the caller routine. In particular, when the creation of a new model is requested, it is necessary to:

1. Retrieve the entry associated with the specified parent environment from our global list.
2. Initialize a new model environment with the same parameters of the parent environment.
3. Store the pointer of the new model in the selected entry.

To maintain the invariant also when setting any parameter value on a model, the steps to be performed are:

1. Retrieve the entry associated with the specified parent environment from our global list.
2. Do an appropriate parameter mapping and conversion of the value.
3. Get the list of the models, associated with the parent environment, from the selected entry.
4. For each model, get its own environment and set the new parameter value.
5. Finally, set the new parameter value also in the parent environment.

When running the optimizer on a model, it is necessary to:

1. Retrieve the entry associated with the specified parent environment from our global list.
2. Set the callback wrapper function for this model, if there is at least a callback pointer associated with the parent environment.
3. Optimize this model.
4. Disable the callback wrapper function for this model, i.e. set a *NULL* pointer as callback.

The following consideration is optional and has only debug purposes. To check even more that the invariant has not been compromised, before any retrieval of a parameter value our interface performs the following steps:

1. Retrieve the entry associated with the specified parent environment from our global list.

2. Get the parameter value from the parent environment.

3. Get the list of the models, associated with the parent environment, from the selected entry.

4. For each model, get its own environment and check that it has the same parameter value of the parent environment.

5. Do the appropriate parameter mapping and value conversion.

Obviously, when releasing a model it is necessary to remove its pointer also from the entry associated with their parent environment. Only when there are no models left, the parent environment can be closed and the entry is removed from the global list.

Parameter mapping As Gurobi and CPLEX use different strategies and algorithms, we do not expect to find a matching Gurobi parameter for every CPLEX parameter. A tuning strategy that works very well with CPLEX, often it may not be necessary at all with Gurobi. Rather, Gurobi recommend to start with default settings and only change parameters when we observe a specific behavior that we would like to modify [GRB].

Despite the differences, our purpose is to produce a code which achieves a similar result and we list, in the table below, the Gurobi equivalent for the most common CPLEX parameters. Most of the correspondences we wrote are given also from the CPLEX migration guide of Gurobi, except for the last six matchings we added for the UFL problem.

CPLEX Parameter	Gurobi equivalent
CPX_PARAM_BARALG	BarHomogeneous
CPX_PARAM_BARCROSSALG	Crossover
CPX_PARAM_BAREPCOMP	BarConvTol
CPX_PARAM_BARQCPEPCOMP	BarQCPConvTol
CPX_PARAM_BRDIR	BranchDir
CPX_PARAM_CLIQUES	CliqueCuts
CPX_PARAM_COVERS	CoverCuts
CPX_PARAM_CUTPASS	CutPasses
CPX_PARAM_EPGAP	MIPGap
CPX_PARAM_EPAGAP	MIPGapAbs
CPX_PARAM_EPINT	IntFeasTol
CPX_PARAM_EPOPT	OptimalityTol
CPX_PARAM_FLOWCOVERS	FlowCoverCuts
CPX_PARAM_FPHEUR	PumpPasses
CPX_PARAM_FRACPASS	GomoryPasses
CPX_PARAM_GUBCOVERS	GUBCoverCuts
CPX_PARAM_HEURFREQ	Heuristics
CPX_PARAM_INTSOLLIM	SolutionLimit
CPX_PARAM_LPMETHOD	Method
CPX_PARAM_MIPEMPHASIS	MIPFocus
CPX_PARAM_MIRCUTS	MIRCuts
CPX_PARAM_NODEFILEIND	NodeFileStart
CPX_PARAM_POLISHAFTEREPGAP	ImproveStartGap
CPX_PARAM_POLISHAFTERTIME	ImproveStartTime
CPX_PARAM_PREDUAL	PreDual
CPX_PARAM_PREIND	Presolve
CPX_PARAM_RINSHEUR	RINS
CPX_PARAM_STARTALG	Method
CPX_PARAM_SUBALG	NodeMethod
CPX_PARAM_THREADS	Threads
CPX_PARAM_TIMELIMIT	TimeLimit
CPX_PARAM_VARSEL	VarBranch
CPX_PARAM_ZEROHALFCUTS	ZeroHalfCuts
Other parameters:	
CPX_PARAM_MIPCBREDLP	PreCrush
CPX_PARAM_PRELINEAR	LazyConstraints
CPX_PARAM_SCRIND	OutputFlag
CPX_PARAM_NODELIM	NodeLimit
CPX_PARAM_CUTLO	Cutoff
CPX_PARAM_CUTUP	Cutoff

Parameter value mapping The routines for access parameters are both getters and setters, i.e. they can either read or write the parameter values. To maintain the full compatibility with Cplex, the mapping of parameter values should be bi-directional, that is representable by a bijective function.

Unfortunately, there are some limitations on this approach that we had to take into consideration. In general, a one-to-one correspondence is not always possible because:

- when performing type conversions, a lose of precision happens. For example, with the parameter *CPX_PARAM_NODELIM,* we had to force a cast from *double* to *int* value type.

- two different values can be mapped into the same value. This is again the case of the parameter *CPX_PARAM_NODELIM*, where we mapped 0 to 1 and 1 to 1, too. In fact, while CPLEX let the optimization terminate after the processing at the root (setting the value zero) or after the branching from the root (setting the value one), Gurobi does not make any distinction and counts the root as a simple node (value one).

In most cases, parameter values are kept as they are, except for two parameters, *CPX_PARAM_MIPCBREDLP* and *CPX_PARAM_PRELINEAR,* we had both to map from 0 (zero) to 1 (one) and from 1 (one) to 0 (zero).

Return value mapping Because the return value is an *output* to the user, a reverse map from Gurobi to CPLEX is expected.

Gurobi C functions always return a status that represents an error code in case of a nonzero value, or a success when zero. In the same way, CPLEX routines have a return status and the user waits for a value to realize whether the requested operation succeeded or not. Have a complete map also for the error codes is not so significant, because such an error usually implies a termination of the whole execution. As the programmer needs to know exactly where is the problem located, a forced mapping of the error code would hide where the real problem was.

Also, in case our interface has been called to perform multiple model modifications and, suddenly, the solver returns an error, only the last modification is aborted. The other modifications, on the contrary, persist and we may enter into an inconsistent state, because we are unable to restore the previous status of the objects. To solve this problem it is necessary to use some kind of transactions, but this would slow down the whole interface execution and can result very complex.

For these reasons, we prefer to raise immediately an exception in case something goes wrong and to return a zero value only when the requested operation succeeded. The exception can be caught only to release the allocated variables, before the whole program aborts.

3.3 Functions implemented

We list, here, all the CPLEX functions we implemented in our interface for Gurobi. We grouped the routines by topic, to present the related design choices in a compact way. Then, for each routine, we give a brief description and we also explain the limits of our implementation.

- Environment and problem creation
  - CPXopenCPLEX, CPXcloseCPLEX, CPXcreateprob, CPXfreeprob
- Problem modification
  - CPXaddrows, CPXnewcols, CPXnewrows, CPXdelrows, CPXdelcols, CPXaddlazyconstraints, CPXchgbds, CPXchgobj, CPXchgrhs, CPXgetctype, CPXchgcoef
- Optimize
  - CPXmipopt, CPXlpopt
- Access problem data
  - CPXgetlb, CPXgetnumcols, CPXgetnummipstarts, CPXgetnumrows, CPXgetobj, CPXgetobjval, CPXgetstat, CPXgetub, CPXgetx, CPXgetbestobjval, CPXgetnodecnt
- File Input/Output
  - CPXordwrite, CPXreadcopyorder, CPXsolwrite, CPXwritemipstarts, CPXwriteparam, CPXwriteprob
- Parameters setting
  - CPXgetdblparam, CPXgetintparam, CPXsetdblparam, CPXsetintparam
- Callbacks
  - CPXcutcallbackadd, CPXgetcallbackinfo, CPXgetcallbacknodex, CPXsetlazyconstraintcallbackfunc, CPXsetusercutcallbackfunc, CPXgetheuristiccallbackfunc, CPXgetinfocallbackfunc, CPXgetusercutcallbackfunc, CPXsetheuristiccallback, CPXsetinfocallbackfunc, CPXgetlazyconstraintcallbackfunc
- Others
  - CPXcopymipstart, CPXcopyorder, CPXdelmipstarts, CPXgetnumcores, CPXgettime, CPXaddmipstarts

3.3.1 Environment and problem creation

CPXopenCPLEX

Initializes the environment.

Prototype CPXENVptr CPXopenCPLEX (int *statusp)

Description The routine creates and initializes a new environment. The pointer returned will be required by every non-advanced CPLEX routine. This must be the first CPLEX routine called. In this case, the return status is stored into an *int* type variable, whose pointer is passed through the *status_p* argument. Its value is set to 0 (zero) only if the environment initialization is successful.

Gurobi reduction: The mapping uses the following Gurobi function:

- GRBloadenv

In addition, as already mentioned, our interface keeps track of each environment it has opened. This is required to maintain the compatibility with the global parameters of Cplex.

CPXcloseCPLEX

Frees the environment.

Prototype int CPXcloseCPLEX (CPXENVptr *envp)

Description This routine closes the environment and releases all of the associated data structures. It should only be called once all models built using the specified environment have been freed.

Gurobi reduction: The mapping uses the following Gurobi function:

- GRBfreeenv

To maintain the compatibility with the global environment of Cplex, we check that each child model has been correctly closed. The pointer in input has to be referred to the parent environment and not a derived copy of it. Then, before closing the Gurobi environment, we perform these additional steps:

1: Retrieve the entry associated with the environment, from our global list
2: Check all derived (child) models have been correctly closed
3: Remove the environment entry from our global list

CPXcreateprob

Creates a problem object.

Prototype CPXLPptr CPXcreateprob (CPXCENVptr env, int *statusp, char const *probnamestr)

Description This routine creates an empty problem object in the specified environment.

Gurobi reduction: The mapping uses the following Gurobi functions:

- GRBnewmodel
- GRBsetcallbackfunc

To maintain the compatibility with Cplex, after the model creation, our interface performs the following steps:

1: Retrieve the parent environment from our global list
2: Create a new model entry associated with the parent environment
3: Store the model entry in our global list
4: {Set up our callback wrapper function for the new model}

The last step is necessary as Gurobi callbacks are always associated with the single model, not with the environment. Cplex instead associates the callbacks with the environment. To maintain the compatibility, a new callback set up is required for each new model. Also, the callback wrapper can be set just before the optimize call.

CPXfreeprob

Frees a problem object.

Prototype int CPXfreeprob (CPXCENVptr env, CPXLPptr *lp_p)

Description This routine removes the specified problem object from the environment and releases the associated memory.

Gurobi reduction: The mapping uses the following Gurobi function:

- GRBfreemodel

To maintain the compatibility with Cplex, our interface requires in input only the parent environment, not a copy of it, and the model to close. Only models regularly opened are allowed to be released, because our interface has to find and remove the associated entry from our global list of models.

3.3.2 Problem modification

CPXaddrows

Add constraints.

Prototype int CPXaddrows (CPXCENVptr env, CPXLPptr lp, int ccnt, int rcnt, int nzcnt, double const *rhs, char const *sense, int const *rmatbeg, int const *rmatind, double const *rmatval, char **colname, char **rowname)

Description This routine add constraints to a problem object.

Gurobi reduction: The Gurobi functions used for the mapping are:

- GRBaddvar
- GRBaddrangeconstr
- GRBaddconstr
- GRBupdatemodel

The mapping is not very hard, although some prudence is required when working with ranged constraints. In fact, when Gurobi adds a range constraint to the model, it adds both a new constraint and a new variable. This is because, unlike Cplex, Gurobi stores range constraints internally as equality constraints and an extra variable is needed to capture the range information. If one is keeping a count of the variables in the model, he must remember to add one for each range constraint. A possible pseudocode can be:

1: **for all** $i \in [0, \text{ccnt}]$ **do**
2: Add empty variable x_i to the model
3: **end for**
4: Update model
5: **for all** $j \in [0, \text{rcnt}]$ **do**
6: **if** $\text{sense}[j] = R$ **then**
7: Add ranged constraint with an additional variable
8: **else**
9: Add constraint c_i to the model

10: **end if**
11: **end for**
12: Update model

CPXnewcols

Adds empty variables.

Prototype int CPXnewcols (CPXCENVptr env, CPXLPptr lp, int ccnt, double const *obj, double const *lb, double const *ub, char const *xctype, char **colname)

Description This routine adds empty columns to the specified problem object.

Gurobi reduction: The map is direct and the equivalence between the different types of variables (i.e. continuous, binary, general integer, semi-continuous, semi-integer) is granted. The Gurobi functions used are:

- GRBaddvars
- GRBupdatemodel

CPXnewrows

Adds empty constraints.

Prototype int CPXnewrows (CPXCENVptr env, CPXLPptr lp, int rcnt, double const *rhs, char const *sense, double const *rngval, char **rowname)

Description This routine adds empty constraints to the specified problem object.

Gurobi reduction: The mapping is simpler than *CPXaddrows*, but the same prudence is required when working with ranged constraints (see the notes for *CPXaddrows*). The Gurobi functions used are:

- GRBaddrangeconstr
- GRBaddconstr
- GRBupdatemodel

CPXdelrows

Removes a range of constraints.

Prototype int CPXdelrows (CPXCENVptr env, CPXLPptr lp, int begin, int end)

Description This routine deletes a range of constraints from the specified model object.

Gurobi reduction: The map is easy to implement and requires the use of an additional array for the indices. The Gurobi functions used are:

- GRBdelconstrs
- GRBupdatemodel

CPXdelcols

Removes a range of variables.

Prototype int CPXdelcols (CPXCENVptr env, CPXLPptr lp, int begin, int end)

Description This routine deletes a range of variables from the specified model object.

Gurobi reduction: The map is easy to implement and requires the use of an additional array for the indices. The Gurobi functions used are:

- GRBdelvars
- GRBupdatemodel

CPXaddlazyconstraints

Adds lazy constraints.

Prototype int CPXaddlazyconstraints (CPXCENVptr env, CPXLPptr lp, int rcnt, int nzcnt, double const *rhs, char const *sense, int const *rmatbeg, int const *rmatind, double const *rmatval, char **rowname)

Description This routine adds constraints to the list of constraints that should be added to the LP sub-problem of a MIP optimization if they are violated. This routine is necessary because the user may have a prior knowledge of the model and may know a large set of cuts. In this case, it is better to add the cuts as Lazy constraints: rather than adding them to the original problem one by one, they are added only when they are violated.

Gurobi reduction: While CPLEX has a dedicated routine for the lazy constraint insertion, Gurobi need two steps to perform the same operation. In fact, our procedure calls the previously implemented *CPXaddrows* routine in order to add the appropriate number of new constraints, then sets their *Lazy* attribute to 1 (one) to mark them as lazy. Gurobi, in general, provides three different level of aggressiveness for each lazy constraint: it can be used only to cut off a feasible solution, or can be pulled into the model or, even more, used to cut off also the relaxation solution. Unfortunately, we had to choose only one value for the mapping and the CPLEX user cannot control this option. Also, unlike CPLEX, our implementation do not allow ranged rows as lazy constraints.

The functions used are:

- CPXaddrows (previously implemented)
- GRBsetintattrelement
- GRBupdatemodel

CPXchgbds

Changes the lower or upper bounds of variables.

Prototype int CPXchgbds (CPXCENVptr env, CPXLPptr lp, int cnt, int const *indices, char const *lu, double const *bd)

Description The routine changes the lower or upper bounds of a list of variables of the specified problem. By setting the upper and lower bounds to the same value, the value of a variable can be fixed at one value.

Gurobi reduction: The mapping is a simple loop that changes the LB and UB attributes of the specified variables of a Gurobi model.

Functions used:

- GRBsetdblattrelement
- GRBupdatemodel

CPXchgobj

Changes the objective coefficients.

Prototype int CPXchgobj (CPXCENVptr env, CPXLPptr lp, int cnt, int const *indices, double const *values)

Description This routine changes the linear objective coefficients of a set of variables in the specified problem object.

Gurobi reduction: Unlike CPLEX, Gurobi does not provide any dedicated function for changing the coefficients of the objective function. Our mapping is a simple loop that changes the *Obj* attributes of the specified variables of a Gurobi model, using the following functions:

- GRBsetdblattrelement
- GRBupdatemodel

CPXchgrhs

Changes a list of RHS values.

Prototype int CPXchgrhs (CPXCENVptr env, CPXLPptr lp, int cnt, int const *indices, double const *values)

Description This routine changes the right-hand side coefficients of a set of linear constraints in the specified problem object.

Gurobi reduction: The mapping is a simple loop that changes the *RHS* attributes of the specified constraints of a Gurobi model, using the functions:

- GRBsetdblattrelement
- GRBupdatemodel

CPXgetctype

Gets the variable type.

Prototype int CPXgetctype (CPXCENVptr env, CPXCLPptr lp, char *xctype, int begin, int end)

Description This routine gets the types for a list of variables in a problem object. The range of variables to select is specified with the parameters *begin* and *end*.

Gurobi reduction: The mapping is simple as Gurobi stores the type of each variable in the attribute array *VType* of the specified Gurobi model. Our interface uses the Gurobi function:

- GRBgetcharattrarray

CPXchgcoef

Changes a coefficient.

Prototype int CPXchgcoef (CPXCENVptr env, CPXLPptr lp, int i, int j, double newvalue)

Description This routine changes a single coefficient in the constraint matrix, linear objective coefficients, right-hand side or ranges of the specified problem object.

Gurobi reduction: Unlike CPLEX, multiple Gurobi routines are involved according to the type of coefficient to be changed. The linear objective row is referenced with the argument i = -1, while the RHS column is referenced with j = -1. In these special cases the Gurobi attributes involved are *OBJ* and *RHS*. Unfortunately, our interface does not support any change in the range value of a constraint.

The Gurobi functions used are:

- GRBsetdblattrelement
- GRBchgcoeffs
- GRBupdatemodel

3.3.3 Optimize

CPXmipopt

Solves a MIP problem.

Prototype int CPXmipopt (CPXCENVptr env, CPXLPptr lp)

Description This routine tries to find a solution to the specified MIP problem. The value returned is 0 (zero) in case no errors occur, which does not necessarily mean that a solution exists. A successful termination can also be exceeding a user-specified limit or finding that the problem was infeasible or unbounded.

The solution status must be obtained with the routine CPXgetstat.

Gurobi reduction: The mapping is direct, but the Gurobi routine is more general than the Cplex one, because GRBoptimize solves every model type. Then, before solving the specified model our interface also checks that the input model is really a MIP.

Furthermore, if callbacks are set, we check here that the parameter configuration is correct, as required by Gurobi:

- LazyConstraints to 1, for the lazy-constraint callback
- PreCrush to 1, for the user-cut callback

Functions used:

- GRBoptimize

CPXlpopt

Solves a LP problem.

Prototype int CPXlpopt (CPXCENVptr env, CPXLPptr lp);

Description This routine is similar to CPXmipopt, except for the type of the problem to be solved. Linear optimizers are used.

Gurobi reduction: The mapping is direct, but the Gurobi routine is more general than the Cplex one, because GRBoptimize solves every model type. Then, before solving the specified model our interface also checks that the input model is really a LP. See the routine above.

Functions used:

- GRBoptimize

3.3.4 Access problem data

We outline the design choices for mapping for the status codes related to the *CPXgetstat* function.

The optimization status code is a value returned to the user after an optimize call. The user should always check the status of the optimization once it completes. Because we consider the optimization status code as an output to the user, our interface needs to do a reverse mapping, from Gurobi to Cplex. In general, Gurobi uses status codes in the following situations:

- after an optimize call, the optimization status is stored in the model attribute *Status;*
- in a callback, when the current MIP node has terminated the optimization.

CPLEX has a lot of status codes and includes some additional information in the status, such as if the model has an integer solution or still not. Gurobi instead has fewer optimization statuses and let the user find out them manually from the attributes *isMIP* and *SolCount*. For this reason, our mapping depends also on the values of the these two model attributes and, in the Table below, we reported their possible combinations along the columns (LP, when not MIP; MIP feasible, when there is at least one solution; or MIP infeasible, when no solutions have been found yet).

Notes for Table 4:

- Brackets mean that an appropriate matching was not available and we had to choose the one we consider the most suitable. We remark that this is only a proposal, in order to have a unique definition and, if necessary, it can be changed.
- The asterisk (*) means that to obtain a more definitive conclusion, the *DualReductions* parameter must be set to 0 and the model re-optimized.
- We do not consider the "In progress" status, as it is used only for asynchronous optimization calls.

Gurobi status	LP	MIP (feasible)	MIP (infeasible)
LOADED	0 (zero)		
OPTIMAL	CPX_STAT_OPTIMAL	CPXMIP_OPTIMAL	//
INFEASIBLE	CPX_STAT_INFEASIBLE	//	CPXMIP_INFEASIBLE
INF_OR_UNBD	CPX_STAT_INForUNBD*	//	CPXMIP_INForUNBD (*)
UNBOUNDED	CPX_STAT_UNBOUNDED	CPXMIP_UNBOUNDED	
CUTOFF	(CPX_STAT_ABORT_USER)	(CPXMIP_ABORT_FEAS)	(CPXMIP_ABORT_INFEAS)
ITERATION_LIMIT	CPX_STAT_ABORT_IT_LIM	(CPXMIP_ABORT_FEAS)	(CPXMIP_ABORT_INFEAS)
NODE_LIMIT	//	CPXMIP_NODE_LIM_FEAS	CPXMIP_NODE_LIM_INFEAS
TIME_LIMIT	CPX_STAT_ABORT_TIME_LIM	CPXMIP_TIME_LIM_FEAS	CPXMIP_TIME_LIM_INFEAS
SOLUTION_LIMIT	//	CPXMIP_SOL_LIM	
INTERRUPTED	CPX_STAT_ABORT_USER	CPXMIP_ABORT_FEAS	CPXMIP_ABORT_INFEAS
NUMERIC	(CPX_STAT_NUM_BEST)	(CPXMIP_FAIL_FEAS)	(CPXMIP_FAIL_INFEAS)
SUBOPTIMAL	(CPX_STAT_NUM_BEST)	(CPXMIP_FAIL_FEAS)	//
INPROGRESS	//		

Table 4: Mapping for Gurobi optimization status

CPXgetlb

Gets variable lower bounds.

Prototype int CPXgetlb (CPXCENVptr env, CPXCLPptr lp, double *lb, int begin, int end)

Description This routine gets a range of lower bounds on the variables of the specified problem object. In case a variable has no lower bound, then the value returned is less than or equal to -CPX_INFBOUND.

Gurobi reduction: The mapping is immediate, as it is sufficient to access the *LB* attribute in the specified Gurobi model, through the function:

- GRBgetdblattrarray

CPXgetnumcols

Gets the number of variables.

Prototype int CPXgetnumcols (CPXCENVptr env, CPXCLPptr lp)

Description This routines returns the number of columns in the constraint matrix, i.e. the number of variables in the specified problem object.

Gurobi reduction: The mapping is immediate, as it is sufficient to access the *NumVars* attribute in the specified Gurobi model.
Function used:

- GRBgetintattr

CPXgetnummipstarts

Gets the number of loaded MIP starts.

Prototype int CPXgetnummipstarts (CPXCENVptr env, CPXCLPptr lp)

Description This routine gets the number of MIP starts in the problem object.

Gurobi reduction: Unfortunately, Gurobi supports only one solution vector as MIP start. Thus, the value returned is only 1 (one) or 0 (zero). To decide if a MIP start has been set, our interface checks the attribute vector *Start*. If at least one element has a value different from GRB_UNDEFINED, then a MIP Start is already present.
Function used:

- GRBgetdblattrelement

CPXgetnumrows

Gets the number of constraints.

Prototype int CPXgetnumrows (CPXCENVptr env, CPXCLPptr lp)

Description This routine accesses the number of rows in the constraint matrix, not including the objective function nor the bounds constraints on the variables.

Gurobi reduction: The mapping is immediate, as it is sufficient to access the *NumConstrs* attribute in the specified Gurobi model through the function:

- GRBgetintattr

CPXgetobj

Gets the obj function coefficients.

Prototype int CPXgetobj (CPXCENVptr env, CPXCLPptr lp, double *obj, int begin, int end)

Description This routine gets a range of objective function coefficients of the specified problem object. The beginning and end of the range are required.

Gurobi reduction: The mapping is immediate, as it is sufficient to access the *OBJ* attribute in the specified Gurobi model through the function:

- GRBgetintattr

CPXgetobjval

Gets the solution objective value.

Prototype int CPXgetobjval (CPXCENVptr env, CPXCLPptr lp, double *objvalp)

Description This routine gets the solution objective value, returning a non-zero value if no solution exists.

Gurobi reduction: The mapping is immediate, as it is sufficient to access the *ObjVal* attribute in the specified Gurobi model, through the function:

- GRBgetdblattr

Note that the model must have been solved to optimality

CPXgetstat

Gets the solution status.

Prototype int CPXgetstat (CPXCENVptr env, CPXCLPptr lp)

Description This routine gets the solution status of the specified problem after an LP or MIP optimization.

Gurobi reduction: This is a reverse mapping, that is from *Gurobi* to *CPLEX*, because the solution status returned by the solver must be converted accordingly in order to be recognized by the CPLEX user. Choosing how to convert the solution is quite complicate, because involved attributes are *Status, isMip* and *SolCount* and a combination of them is necessary to preserve the compatibility with CPLEX (see the Table 4 and the design choices at the beginning of this Chapter).

The Gurobi functions involved are only multiple calls to:

- GRBgetintattr

CPXgetub

Gets variable upperbounds.

Prototype int CPXgetub (CPXCENVptr env, CPXCLPptr lp, double *ub, int begin, int end)

Description This routine gets a range of upper bounds on the variables of the specified problem object. In case a variable has no upper bound, then the value returned is less than or equal to +CPX_INFBOUND.

Gurobi reduction: The mapping is immediate, as it is sufficient to access the *UB* attribute in the specified Gurobi model, through the function:

- GRBgetdblattrarray

CPXgetx

Gets the solution values.

Prototype int CPXgetx (CPXCENVptr env, CPXCLPptr lp, double *x, int begin, int end)

Description This routine gets the solution values for a range of problem variables. The beginning and end of the range must be specified.

Gurobi reduction: The mapping is immediate, as it is sufficient to access the X attribute in the specified Gurobi model, through the functions:

- GRBgetdblattrarray

CPXgetbestobjval

Gets the best bound.

Prototype int CPXgetbestobjval (CPXCENVptr env, CPXCLPptr lp, double *objvalp)

Description This routine gets the currently best known bound of all the remaining open nodes in a branch-and-cut tree. If no solution information is available, then for a MIP this routine returns *-infinity* for minimization problems and *+infinity* for maximization problems.

Gurobi reduction: The mapping is immediate, as it is sufficient to access the *ObjBound* attribute in the specified Gurobi model, through the functions:

- GRBgetdblattr

Note: to use this routine, the model must have been solved to optimality.

CPXgetnodecnt

Gets the number of nodes used.

Prototype int CPXgetnodecnt (CPXCENVptr env, CPXCLPptr lp)

Description This routine gets the number of nodes used to solve a mixed integer problem.

Gurobi reduction: The mapping is immediate, as it is sufficient to access the *NodeCount* attribute in the specified Gurobi model, through the functions:

- GRBgetdblattr

Note: to use this routine, the model must have been solved to optimality.

3.3.5 File Input/Output

CPXordwrite

Writes a priority order file.

Prototype int CPXordwrite (CPXCENVptr env, CPXCLPptr lp, char const *filenamestr)

Description This routine creates a *.ord* file and stores the priority order information associated to the specified problem object.

Gurobi reduction: The mapping is direct, but the Gurobi routine is more general than the Cplex one. In fact, GRBwrite is also used to write optimization models, solutions vectors, basis vectors, start vectors and parameter settings in the specified file. As Gurobi determines the type of data to store according to the file suffix, our interface will check that the specified file extension is correct, i.e. ".ord".

Functions used:

- GRBwrite

CPXreadcopyorder

Retrieves the priority order information from file.

Prototype int CPXreadcopyorder (CPXCENVptr env, CPXLPptr lp, char const *filenamestr)

Description This routine reads the specified priority order file and copies the priority order information into the specified problem object.

Gurobi reduction: The mapping is direct, but the Gurobi routine is more general than the Cplex one. In fact, GRBread is also used to read start vectors for MIP models, basis files for LP models or parameter settings from the specified file. As Gurobi determines the type of data to read according to the file suffix, our interface will check that the specified file extension is correct, i.e. ".ord".

Functions used:

- GRBread
- GRBupdatemodel

CPXsolwrite

Writes a solution file.

Prototype int CPXsolwrite (CPXCENVptr env, CPXCLPptr lp, char const *filenamestr)

Description This routine creates a *.sol* file and stores the solution for the specified problem object.

Gurobi reduction: The mapping is direct, but the Gurobi routine is more general than the Cplex one. In fact, GRBwrite is also used to write optimization models, priority order information, basis vectors, start vectors and parameter settings in the specified file. As Gurobi determines the type of data to store according to the file suffix, our interface will check that the specified file extension is correct, i.e. ".sol".

Functions used:

- GRBwrite

CPXwritemipstarts

Writes a MIP start file.

Prototype int CPXwritemipstarts (CPXCENVptr env, CPXCLPptr lp, char const *filenamestr, int begin, int end)

Description This routine creates a *.mst* file and stores the solution for the specified problem object.

Gurobi reduction: A direct mapping is not possible as Gurobi supports only one MIP Start at a time. Then, our interface allows only a value of 0 (zero) for the *begin* and *end* arguments.

Also, the Gurobi routine is more general than the Cplex one. In fact, GRBwrite is also used to write optimization models, priority order information, basis vectors, start vectors and parameter settings in the specified file. As Gurobi determines the type of data to store according to the file suffix, our interface will check that the specified file extension is correct, i.e. ".mst".

Functions used:

- GRBwrite

CPXwriteparam

Writes a parameter settings file.

Prototype int CPXwriteparam (CPXCENVptr env, char const *filenamestr)

Description This routine creates a *.prm* file and stores the name and current value of the parameters that are not at their default setting in the environment specified.

Gurobi reduction: The mapping is simple for the routine itself, but for the file data content the work is very hard. As a direct mapping for each parameter of Cplex is not possible, we chose to store only Gurobi parameters to the specified file.

Anyway, the Gurobi routine is more general than the Cplex one. In fact, GRBwrite is also used to write optimization models, priority order information, basis vectors and start vectors in the specified file. As Gurobi determines the type of data to store according to the file suffix, our interface will check that the specified file extension is correct, i.e. ".prm".

Functions used:

- GRBwrite

CPXwriteprob

Stores an optimization model to file.

Prototype int CPXwriteprob (CPXCENVptr env, CPXCLPptr lp, char const *filenamestr, char const *filetypestr)

Description This routine writes a problem object to a file in the specified format.

Gurobi reduction: The mapping is direct, but the Gurobi routine is more general than the Cplex one. In fact, GRBwrite is also used to write priority order information, basis vectors, start vectors and parameter settings in the specified file. As Gurobi determines the type of data to store according to the file suffix, valid file extensions are ".mps", ".rew", ".lp" and ".rlp".

Functions used:

- GRBwrite

3.3.6 Parameters setting

CPXgetdblparam

Gets a parameter value of type *double*.

Prototype int CPXgetdblparam (CPXCENVptr env, int whichparam, double *valuep)

Description This routine obtains the current value of a parameter of type double.

Gurobi reduction: As already seen, this mapping is rather complex. First, a matching for each parameter of Cplex is not always possible (for a complete list, see the Parameter chapter). Second, there is a compatibility problem with the environments, because Cplex uses the same environment for all the associated problems. Gurobi, instead, creates a separate copy of the environment for each model object associated with the parent environment. To maintain the compatibility with Cplex, we have to manually warrant the parameter consistency of each copy of the parent environment. We expect that the environment specified in input is a "parent" environment and not a "child", i.e. a derived copy of it. Thus, before returning the requested parameter, our interface performs the following steps:

1: Convert the parameter name from CPLEX to Gurobi
2: Retrieve the associated environment entry from our global list
3: Get the list of models associated with this parent environment
4: Get the parameter value from the parent environment
5: Check all the child environments have the same parameter value

Functions used:

- GRBgetdblparam

CPXgetintparam

Gets a parameter value of type *int*.

Prototype int CPXgetintparam (CPXCENVptr env, int whichparam, CPXINT *valuep)

Description This routine gets the current value of the specified parameter of type CPXINT, or *int*.

Gurobi reduction: The implementation is similar to *CPXgetdblparam* routine. Functions used:

- GRBgetintparam

CPXsetdblparam

Sets a parameter value of type *double*.

Prototype int CPXsetdblparam (CPXENVptr env, int whichparam, double newvalue)

Description This routine sets the value of the specified parameter of type double.

Gurobi reduction: The implementation is similar to *CPXgetdblparam* routine. Functions used:

- GRBsetdblparam

CPXsetintparam

Sets a parameter value of type *int*.

Prototype int CPXsetintparam (CPXENVptr env, int whichparam, CPXINT newvalue)

Description This routine sets the value of the specified parameter of type *int*.

Gurobi reduction: The implementation is similar to *CPXgetdblparam* routine. Functions used:

- GRBsetintparam

3.3.7 Callbacks

While CPLEX provides several kinds of callbacks, Gurobi models can only have a single callback function. The *where* argument in the Gurobi callback function is the only way to distinguish the context, in the optimization process, where the callback was invoked. To map the multiple CPLEX callbacks into a single Gurobi callback, we implemented a dispatcher routine able to wrap the user custom procedures and call them at the right point. Currently, our interface properly supports the following callbacks: Lazy, Usercut, Heuristic and Info callback.

Data structures We defined the following data structures to store the pointers to the custom procedures the user can provide and their data inside callbacks.

```
/* Define struct to point user procedures */
struct callback_pointers {
        int (*lazy_cb)(CALLBACK_CUT_ARGS);
        int (*usercut_cb)(CALLBACK_CUT_ARGS);
        int (*heur_cb)(CALLBACK_HEURISTIC_ARGS);
        int (*info_cb)(CPXCENVptr, void *, int, void *);
};

/* Define struct to point user data in callbacks */
struct callback_data {
        void *lazyData;
        void *usercutData;
        void *heurData;
        void *infoData;
};
```

Dispatcher for the callbacks Here, we describe the general behaviour of the algorithm we used to dispatch the user callbacks. Our implementation is completely transparent to the user, because all the provided data input and output are properly wrapped.

About the Gurobi *where* attribute:

- GRB_CB_MIPSOL is obtained when all binary/integer variables are integer or a heuristic finds a new incumbent (i.e. a new integer-feasible solution)

Algorithm 6 Callback dispatcher algorithm

Input: *where* status from the solver

Output: *return* status to the solver

```
if (where = GRB_CB_MIP) then
  wherefrom← CPX_CALLBACK_MIP
  "Point to INFO callback"
else if (where = GRB_CB_MIPSOL) then
  wherefrom ← CPX_CALLBACK_MIP_CUT_FEAS
  "Point to LAZY callback"
else if (where = GRB_CB_MIPNODE) then
  status ← "Get optimization status of LP relaxation"
  if (status = GRB_UNBOUNDED) then
    wherefrom ← CPX_CALLBACK_MIP_CUT_UNBD
    "Point to LAZY callback"
  else if (status = GRB_OPTIMAL) then
    wherefrom ← CPX_CALLBACK_MIP_HEURISTIC
    "Point to HEURISTIC callback"
    "Add returned solution to Gurobi"
    wherefrom ← CPX_CALLBACK_MIP_CUT_LAST
    "Point to USERCUT callback"
  else
    {skip callback if node is reported to be cut off or infeasible}
  end if
end if
```

- GRB_CB_MIPNODE (optimal) when the LP relaxation at the node is proven optimal (solution can be fractional)
- GRB_CB_MIPNODE (unbounded) in case of an unbounded ray is found (solution can be fractional)

Callback infos mapping The table below provides the mapping for the most common information codes requested with the *CPXgetcallbackinfo* function. Along rows there are the information requested by the CPLEX user through the callbacks, along columns the Gurobi context (*where* argument) in which the request was generated. To obtain the Gurobi mapping, it is necessary to match the information with the context.

For some information codes, several workarounds were necessary because Gurobi does not provide that information natively. For example, to obtain the *MIP relative gap* we computed the following expression:

$$\frac{|\text{objbst} - \text{objbnd}|}{e^{-10} + |\text{objbst}|}$$

For the *cutoff* information, we used both the *best bound* information and the *cutoff* parameter value.

The asterisk (*) means that the information is an obvious consequence, e.g. from the fact that Gurobi callbacks are always executed in single-thread mode, while the *MIP feasibility* depends on the number of solutions found.

CPLEX requested information	GRB_CB_MIP	GRB_CB_MIPSOL	GRB_CB_MIPNODE
CPX_CALLBACK_INFO_BEST_INTEGER	MIP_OBJBST	MIPSOL_OBJBST	MIPNODE_OBJBST
CPX_CALLBACK_INFO_BEST_REMAINING	MIP_OBJBND	MIPSOL_OBJBND	MIPNODE_OBJBND
CPX_CALLBACK_INFO_NODE_COUNT	MIP_NODCNT	MIPSOL_NODCNT	MIPNODE_NODCNT
CPX_CALLBACK_INFO_NODES_LEFT	MIP_NODLFT	N/A	N/A
CPX_CALLBACK_INFO_MIP_ITERATIONS	MIP_ITRCNT	N/A	N/A
CPX_CALLBACK_INFO_CUTOFF	(with workaround)	(with workaround)	(with workaround)
CPX_CALLBACK_INFO_MIP_FEAS	MIP_SOLCNT (*)	MIPSOL_SOLCNT (*)	MIPNODE_SOLCNT (*)
CPX_CALLBACK_INFO_MY_THREAD_NUM	0 (*)	0 (*)	0 (*)
CPX_CALLBACK_INFO_USER_THREADS	1 (*)	1 (*)	1 (*)
CPX_CALLBACK_INFO_MIP_REL_GAP	(with workaround)	(with workaround)	(with workaround)

Table 5: Callback information codes mapping

CPXcutcallbackadd

Adds user-cuts and lazy-constraints through callbacks.

Prototype int CPXcutcallbackadd (CPXCENVptr env, void *cbdata, int wherefrom, int nzcnt, double rhs, int sense, int const *cutind, double const *cutval, int purgeable)

Description This routine adds lazy constraints and globally valid cuts to the current node LP subproblem during MIP branch and cut. It can be called only from a lazy-constraint or user-cut callback.

Gurobi reduction: The mapping is a bit complex, as our interface has to discriminate the calls the user performs (to insert lazy constraint or user-cuts), based on the wherefrom parameter.

Unfortunately, Gurobi does not support any purgeable parameter in user cuts, neither is possible to force the user-cut insertion to the model. User-cuts are subjected to Gurobi filter and can be removed in any moment. Also, range constraints are not supported. For using this routine, is necessary to set parameters CPX_PARAM_MIPCBREDLP and CPX_PARAM_PRELINEAR to zero. According to the context, the Gurobi functions used are:

- GRBcblazy
- GRBcbcut

CPXgetcallbackinfo

Gets information through callbacks.

Prototype int CPXgetcallbackinfo (CPXCENVptr env, void *cbdata, int wherefrom, int whichinfo, void *resultp)

Description This routine accesses information about the current optimization process from within a user-written callback function.

Gurobi reduction: This mapping required hard work, because of the different kinds of information we can retrieve though this routine. The implementation is based on the Table previously presented. As we can see, the mapped information is different, according to the context (*wherefrom* parameter). Also, some values needed type or value conversions before being presented to the CPLEX user.

Depending on the information requested, the Gurobi functions used are:

- GRBcbget
- GRBgetintattr
- GRBgetdblparam

CPXgetcallbacknodex

Gets the LP solution through a callback.

Prototype int CPXgetcallbacknodex (CPXCENVptr env, void *cbdata, int wherefrom, double *x, int begin, int end)

Description This routine retrieves the primal variable values for the subproblem at the current node during MIP optimization from within a user-written callback.

Gurobi reduction: The implementation is a bit tricky, because Gurobi need as argument a solution vector of size equal to the number of variables in the model. To maintain the compatibility with CPLEX and let the user retrieve a partial solution vector, our interface has to allocate an intermediate buffer vector where to store the LP solution values and return only the ones requested.

Also, Gurobi let the retrieval of the LP relaxation solution only when the where argument is GRB_CB_MIPSOL or GRB_CB_MIPNODE with status OPTIMAL. Unfortunately, in case the node LP is unbounded we cannot retrieve the solution vector to separate lazy constraints.

Finally, the values are related to the original problem only if the parameter CPX_PARAM_MIPCBREDLP is set to zero (0).

The Gurobi functions used are:

- GRBgetintattr
- GRBcbget

CPXsetlazyconstraintcallbackfunc

Sets the lazy-constraint callback.

Prototype int CPXsetlazyconstraintcallbackfunc (CPXENVptr env, int(*lazyconcallback) (CALLBACKCUTARGS), void *cbhandle)

Description This routine sets and modifies the user-written callback for adding lazy constraints. A NULL pointer disables the callback.

Gurobi reduction: The pointers to the user procedure and callback data are stored on a global list and associated to the specified environment.

As described before, our interface uses two internal data structures, in order to wrap both the pointer to the callback and the user callback data. In fact, Gurobi uses only a single multi-purpose callback and the user function provided is called by our internal dispatcher. No Gurobi functions are directly called from this routine.

CPXsetusercutcallbackfunc

Sets the user-cut callback.

Prototype int CPXsetusercutcallbackfunc (CPXENVptr env, int(*cutcallback) (CALLBACKCUTARGS), void *cbhandle)

Description This routine sets and modifies the user-written callback for adding cuts. A NULL pointer disables the callback.

Gurobi reduction: The pointers to the user procedure and callback data are stored on a global list and associated to the specified environment.

As described before, our interface uses two internal data structures, in order to wrap both the pointer to the callback and the user callback data. In fact, Gurobi uses only a single multi-purpose callback and the user function provided is called by our internal dispatcher. No Gurobi functions are directly called from this routine.

CPXgetheuristiccallbackfunc

Gets the heuristic callback.

Prototype int CPXgetheuristiccallbackfunc CPXCENVptr env, int(**heuristiccallbackp) (CALLBACKHEURISTICARGS), void ** cbhandlep)

Description This routine accesses the user-written callback to be called during MIP optimization after the LP relaxation has been solved to optimality.

Gurobi reduction: The pointers to the user procedure and callback data are retrieved from a global list, according to the specified environment.

As described before, our interface uses two internal data structures, in order to wrap both the pointer to the callback and the user callback data. In fact, Gurobi uses only a single multi-purpose callback and the user function provided is called by our internal dispatcher. No Gurobi functions are directly called from this routine.

CPXgetinfocallbackfunc

Gets the informational callback.

Prototype int CPXgetinfocallbackfunc (CPXCENVptr env, int(**callbackp) (CPXCENVptr, void *, int, void *), void **cbhandlep)

Description This routine accesses the user-written callback routine to be called regularly during the optimization of a mixed integer program (MIP).

Gurobi reduction: The pointers to the user procedure and callback data are retrieved from a global list, according to the specified environment.

As described before, our interface uses two internal data structures, in order to wrap both the pointer to the callback and the user callback data. In fact, Gurobi uses only a single multi-purpose callback and the user function provided is called by our internal dispatcher. No Gurobi functions are directly called from this routine.

CPXgetusercutcallbackfunc

Gets the user-cut callback.

Prototype int CPXgetusercutcallbackfunc (CPXCENVptr env, int(**cutcallbackp) (CALLBACKCUTARGS), void **cbhandlep)

Description This routine accesses the user-written callback for adding cuts.

Gurobi reduction: The pointers to the user procedure and callback data are retrieved from a global list, according to the specified environment.

As described before, our interface uses two internal data structures, in order to wrap both the pointer to the callback and the user callback data. In fact, Gurobi uses only a single multi-purpose callback and the user function provided is called by our internal dispatcher. No Gurobi functions are directly called from this routine.

CPXsetheuristiccallbackfunc

Sets the heuristic callback.

Prototype int CPXsetheuristiccallbackfunc (CPXENVptr env, int(*heuristiccallback) (CALLBACKHEURISTICARGS), void *cbhandle)

Description This routine sets or modifies the user-written callback to be called during MIP optimization after the LP relaxation has been solved to optimality. The heuristic procedure the user supplied is not called if the LP relaxation is infeasible or cut off.

Gurobi reduction: The pointers to the user procedure and callback data are stored on a global list and associated to the specified environment.

As described before, our interface uses two internal data structures, in order to wrap both the pointer to the callback and the user callback data. In fact, Gurobi uses only a single multi-purpose callback and the user function provided is called by our internal dispatcher. No Gurobi functions are directly called from this routine.

CPXsetinfocallbackfunc

Sets the informational callback.

Prototype int CPXsetinfocallbackfunc (CPXENVptr env, int(*callback) (CPXCENVptr, void *, int, void *), void *cbhandle)

Description This routine sets the user-written callback. The informational routine is called regularly during the optimization of a mixed integer program and during certain cut generation routines.

Gurobi reduction: The pointers to the user procedure and callback data are stored on a global list and associated to the specified environment.

As described before, our interface uses two internal data structures, in order to wrap both the pointer to the callback and the user callback data. In fact, Gurobi uses only a single multi-purpose callback and the user function provided is called by our internal dispatcher. No Gurobi functions are directly called from this routine.

CPXgetlazyconstraintcallbackfunc

Gets the lazy-constraint callback.

Prototype int CPXgetlazyconstraintcallbackfunc (CPXCENVptr env, int(**cutcallbackp) (CALLBACKCUTARGS), void **cbhandlep)

Description This routine accesses the user-written callback for adding lazy constraints.

Gurobi reduction: The pointers to the user procedure and callback data are retrieved from a global list, according to the specified environment.

As described before, our interface uses two internal data structures, in order to wrap both the pointer to the callback and the user callback data. In fact, Gurobi uses only a single multi-purpose callback and the user function provided is called by our internal dispatcher. No Gurobi functions are directly called from this routine.

3.3.8 Other functions

CPXcopymipstart

Sets MIP start values.

Prototype int CPXcopymipstart (CPXCENVptr env, CPXLPptr lp, int cnt, int const *indices, double const *values)

Description This routine copies MIP start values to the specified problem object.

Gurobi reduction: Gurobi allows only one MIP start solution at a time. For this reason, our interface allows only a value of 1 (one) for the *cnt* argument. Our implementation is a reduction to *CPXaddmipstarts*, described below.

CPXcopyorder

Sets a priority order for branching.

Prototype int CPXcopyorder (CPXCENVptr env, CPXLPptr lp, int cnt, int const *indices, int const *priority, int const *direction)

Description This routine copies a priority order for each variable to the specified problem object. During branching, integer variables with higher priorities are given preference over integer variables with lower priorities.

Gurobi reduction: Gurobi allows only to specify one priority value for each variable, to decide the branching order. No settings for the branching direction are provided. To set the *BranchPriority* attribute in the specified Gurobi model, the following functions are used:

- GRBsetintattrlist
- GRBupdatemodel

CPXdelmipstarts

Deletes a range MIP starts.

Prototype int CPXdelmipstarts (CPXCENVptr env, CPXLPptr lp, int begin, int end)

Description This routine deletes a range MIP starts. The range is specified using a beginning and ending index that represent the first and last MIP start to delete.

Gurobi reduction: Unfortunately, only one solution can be inserted as *MIP start* to the solver. The pseudocode of the operations performed by our interface is the following:

1: $n \leftarrow$get the number of variables
2: **for all** model variable $x_i, i \in (0, n-1)$ **do**
3: $x_i \leftarrow$ GRB_UNDEFINED
4: **end for**

The Gurobi functions used are:

- GRBsetdblattrelement
- GRBupdatemodel

CPXgetnumcores

Gets the number of cores.

Prototype int CPXgetnumcores (CPXCENVptr env, int *numcoresp)

Description This routine gets the number of logical cores of the machine where the code is being run.

Gurobi reduction: No Gurobi function is available for this purpose. Thus, our interface retrieves the number of cores directly from the low-level system libraries: "*windows.h*" for WIN32, *"sys/param.h*" and *"sys/sysctl.h*" for MacOS,"*unistd.h*" for Unix-derived systems.

CPXgettime

Gets the current timestamp.

Prototype int CPXgettime (CPXCENVptr env, double *timestampp)

Description This routine returns an absolute value from the wall-clock real time.

Gurobi reduction: Our interface supports only the wall-clock time, while for CPU time it is sufficient to call the function *clock()* of the C library "time.h. No Gurobi functions are called, but only the *time()* utility function of the previously mentioned C library.

CPXaddmipstarts

Adds multiple MIP starts.

Prototype int CPXaddmipstarts (CPXCENVptr env, CPXLPptr lp, int mcnt, int nzcnt, int const *beg, int const *varindices, double const *values, int const *effortlevel, char **mipstartname)

Description This routine adds multiple MIP starts to the specified problem object

Gurobi reduction: As Gurobi allows only one MIP start solution at a time, our interface allows only a value of 1 (one) for the *mcnt* argument. The provided solution can be complete or partial, in which case Gurobi will attempt to fill in values for missing start values. As *effortlevel* argument we allow only the CPX_MIPSTART_AUTO value, to remind that Gurobi do not support this setting. To set the *Start* attribute for the variables in the specified Gurobi model, the following functions are used:

- GRBsetdblattrlist
- GRBupdatemodel

3.4 Main differences

Below, we sum up in few points the main differences between CPLEX and Gurobi API interfaces we encountered during our thesis.

- First of all, and as already mentioned, Gurobi gives each model its own copy of a Gurobi *environment* and allows each model to have its own parameter settings. In CPLEX, all models belonging to the same environment use the same parameter values. This peculiarity has been completely managed by our interface.

- Unlike CPLEX, Gurobi use a "*lazy update*" approach. It means that after making any change to a model, we need to call *GRBupdatemodel* in order for those changes to be applied and visible. This approach makes it more efficient to build or modify a model, because it reduces the overhead due to repetitive calls for modifying the model. As CPLEX applies immediately any change to the model, our interface has the responsibility of calling the Gurobi update routine, before the control returns to the user.

- *Attributes* are an important concept in Gurobi. While CPLEX provides a lot of different routines for accessing and modifying the various attributes of a model, Gurobi handle them through an elegant interface of attribute management routines. There are several kind of attributes: the ones associated with variables (e.g. lower bounds), with constraints (e.g. the right-hand side) or with the overall model (e.g. the objective value for the current solution).

- *Parameters* are instead a common concept. Both solvers shares some parameters for certain algorithms although they uses different names, while in other cases the available parameters and their values are completely different. Find a matching Gurobi parameter for every CPLEX parameter is impossible, as well as pointless, because Gurobi and CPLEX use different strategies. In general, keeping the default solver settings is the most clever thing, but sometimes a similar result can be achieved.

- As far as *MIP Start solutions*, they are poorly supported by Gurobi. While CPLEX supports multiple hint solutions at a time with different effort levels, Gurobi permits only one start solution in the pool and the effort level option is not available. For this reason, our interface allows only the CPX_MIPSTART_AUTO option, to mean that Gurobi will decide whether to consider the suggested solution or not.

- As far as *range constraints*, the way each solver stores them internally differs. In Gurobi, every range constraint adds both a new constraint and a new variable, because they are stored internally as equality constraints, while the extra variable captures the range information. In CPLEX, no additional variable is added, but only the new range constraint.

- As far as *static lazy constraints*, while CPLEX has a dedicated routine for their insertion, Gurobi provides an attribute able to convert every inserted constraint to a lazy constraint, with up to three different levels of aggressiveness, as already described.

- Unlike CPLEX, Gurobi lacks of a *deterministic time counter*. The related CPLEX routine counts, in ticks unit, the memory accesses the solver required. This metric is very informative and useful, because RAM access is usually the main bottleneck for the optimization process.

- About the *callback procedures* and their utility routines, a lot of differences emerged from our study. As they deserved special attention, we outline their different characteristics in the following dedicated paragraph.

Finally, also our interface has some known limits, for example we do not allow the *long int* versions of CPLEX variables (e.g. CPX_CALLBACK_INFO_BEST_INTEGER_LONG) or the CPXX versions of the routines (e.g. *CPXXaddcols*). This was a design choice, but with few changes is possible to adapt the code to make it work.

3.4.1 Callback functions

As already seen, the most relevant difference between the callback functions of the two solvers lies in their number. While CPLEX makes available to the user several kinds of callbacks, even very advanced, Gurobi has only a single multi-purpose callback. For this reason our interface required a dispatcher, able to call the different callbacks the CPLEX user had provided, within a single Gurobi callback.

Also the number of threads involved in callbacks execution differs. When Gurobi solves a model using multiple threads, the user callback is only ever called from a single thread [GRB]. CPLEX, instead, allows multi-threading also for the callbacks. By default, only one thread is used inside CPLEX callbacks, but the user can force more threads if he takes the responsibility to manage the thread-safety of the involved data structures.

When we use callbacks to add our custom cuts or lazy constraints, usually two parameters need to be set. The first parameter, *MIPCBREDLP* for CPLEX and *PreCrush* for Gurobi, controls whether the callback accesses node information of the original model or node information of the reduced, presolved model. The second parameter, *PRELINEAR* for CPLEX and *LazyConstraints* for Gurobi, controls the type of reductions, linear or full, that occurs during preprocessing. Only linear reductions always guarantee that users can add their own custom cuts or lazy constraints to the presolved model, because each variable in the original model can be expressed as a linear form of variables in the presolved model.

Lazy-constraint callback CPLEX calls lazy-constraint callback at integer nodes or when an integer-feasible solution is available, for example found by a heuristic.

Lazy-constraint callbacks are also called when the LP relaxation is unbounded, to give us the opportunity to cut off the unbounded ray. In fact, the possible values for the *wherefrom* argument are CPX_CALLBACK_MIP_CUT_FEAS or CPX_CALLBACK_MIP_CUT_UNBD. On the contrary, Gurobi does not well support unbounded models with the lazy-constraint callback. In fact, when Gurobi finds an unbounded ray and the lazy callback is called, with a *where* argument of *MIPNODE* and an *UNBOUNDED* status, the relaxation solution for the current node is not available, because Gurobi allows to retrieve it only when its optimization status is *OPTIMAL*. For this reason, it is not possible to cut off the unbounded ray within the Gurobi lazy callback. This force the user to build only model formulations representable by a bounded polytope, as we did in our UFL example.

In general, Gurobi allows lazy-callbacks calls when the *where* argument is either equal to GRB_CB_MIPSOL or GRB_CB_MIPNODE, and the most relevant operations a Gurobi user can perform inside a lazy-constraint callback are: add a constraint that cuts off the solution, retrieve additional information or terminate the optimization.

User-cut callback We report as Algorithm 7 an example of *cut loop*, adopted on a previous version of CPLEX. In general, a cut loop is a strategy very useful to reduce the lower bound, because gives many opportunities to exploit the user prior knowledge about the model. As a possible drawback, we can also have an excess of generated user cuts, which can heavily slow down the model formulation, or a delay in the solver execution. For these reasons, each solver manages this phase in different ways:

- At each node, CPLEX is involved on a loop where, after an optimal solution for the LP relaxation is available, the user-cut callback is called to separate some cuts, that is, generate some violated cuts. If performed steps turn out to be effective, the whole separation is reiterated several times until the user signals to terminate or the progress becomes poor. In these cases, CPLEX stops its internal cut separation and gives the user a last opportunity to separate other cuts.

- Instead, we noticed that Gurobi performs only a single call to the user-cut callback (through *where* = MIPNODE), once at each node. Except for the root node, our custom separator has a single opportunity to generate violated cuts for each node, as soon as an optimal solution for the LP relaxation has been found.

Algorithm 7 How cut loop works in CPLEX 12.5.1 (ref. [AIBM13])

```
1: while ("Root node is not solved") do
2:   "Solve LP relaxation"
3:   if ("Enough progress has been done, since last iteration") then
4:     "Call HEURISTIC callback"
5:     "Call CPLEX internal heuristics"
6:     "Apply probing"
7:     wherefrom ← CPX_CALLBACK_MIP_CUT_LOOP
8:     "Call USER-CUT callback"
9:     if ("User did not signal to terminate loop") then
10:      "Separate CPLEX internal cuts and the user cut pool"
11:      "Filter cuts" {according to user's cut purging flag}
12:      continue {jump to next iteration}
13:    end if
14:  end if
15:  wherefrom ← CPX_CALLBACK_MIP_CUT_LAST
16:  "Call USER-CUT callback"
17:  if ("Cuts have been found") then
18:    "Filter cuts"
19:  else
20:    break {exit the while}
21:  end if
22: end while
23: while ("MIP node is not solved") do
24:  "Solve LP relaxation"
25:  if ("Enough progress has been done, since last iteration") then
26:    wherefrom ← CPX_CALLBACK_MIP_CUT_LOOP
27:    if ("User did not signal to terminate loop") then
28:      if ("No cuts have been found at the first iteration") then
29:        "Separate CPLEX internal cuts and the user cut pool"
30:      end if
31:      "Filter cuts"
32:      continue {jump to next iteration}
33:    end if
34:  end if
35:  wherefrom ← CPX_CALLBACK_MIP_CUT_LAST
36:  "Call the USER-CUT callback"
37:  if ("Cuts have been found") then
38:    "Filter cuts"
39:  else
40:    break {exit the while}
41:  end if
42: end while
```

As a consequence of the Gurobi behaviour described above, our interface always performs calls the user-cut callback only with the value CPX_CALLBACK_MIP_CUT_LAST in the wherefrom argument, and never with the CPX_CALLBACK_MIP_CUT_LOOP value. Furthermore, we cannot set any degree for the purgeable value, i.e. to mark when cuts can be eliminated, but we have to trust Gurobi to manage them conveniently.

Heuristic callback As already described in the algorithm for the user cut loop, CPLEX usually calls the heuristic callback after an optimal solution to the subproblem has been obtained, and Gurobi does the same (through the argument *where* = MIPNODE). The CPLEX user can provide an integer solution replacing the incumbent in case it has a better objective value, while for Gurobi this is not necessarily always true. In general, both solvers support only one solution, at a time, and also the user can provide, to both of them, an incomplete solution to let the internal heuristics complete it. But, while CPLEX always let the user load the objective value for a solution he found, even without forcing a check for its feasibility, Gurobi only accepts the whole solution vector. That means Gurobi has no other ways to know its objective value until the solution vector is checked for feasibility. Unfortunately, Gurobi behaviour can result tricky for some strategies of our heuristics, because the lack of the objective information can delay the solver awareness about a new incumbent. Also, new solutions can be added only from here (*where* = MIPNODE) and not from other positions, such as from a lazy callback (having argument *where* = MIPSOL).

Another thing we remark is that, before using the heuristic callback, it is often recommended to set the same parameters already described for the lazy-constraint and user-cut callback (*MIPCBREDLP* and *PRELINEAR*). This is because our custom heuristics usually have to be applied to the original problem, and not to a presolved version.

Finally, a limit of our interface is that the heuristic callback is called whenever the user-cut callback is called. This is because for both of them Gurobi uses the same *where* argument, i.e. *MIPNODE*. To module independently the calls to the heuristic and user-cut procedures, a tradeoff parameter based on priority or frequency may result useful, for example:

```
if (rand() > HeurFreqParameter * RAND_MAX) { .. }
```

Informational callback Both CPLEX and Gurobi call regularly the informational callback during the MIP optimization and also during certain cut generation routines. There are only few differences, all about the information we are able to retrieve inside the informative callback.

CPLEX, for example, allows the retrieval of the solution vector of the current incumbent (through *CPXgetcallbackincumbent*), while Gurobi can only get its objective value. Another situation is when the user needs to know the number of MIP iterations performed: a CPLEX user can access the information *CPX_CALLBACK_INFO_MIP_ITERATIONS* from several callback positions, while the Gurobi equivalent *MIP_ITRCNT* is readable only from the informative callback. This fact can preclude the control of some user-cut procedures or heuristic algorithms based on the number of MIP iterations.

4 Test: Cplex vs Gurobi

We present here a computational comparison between CPLEX and Gurobi, using both the UFL code (Section 2) and the Gurobi interface (Section 3).

4.1 Test description

To perform the tests, we compiled the code we wrote previously for the UFL problem, linking the CPLEX APIs; then we combined the same code with our interface and compiled it, linking the Gurobi APIs. Now, for each compiled executable, our challenge consists of solving the testbed instances to proven optimality, within a time limit of 30 minutes, and with the same hardware we described in Section 2.5. As the original code had been developed and calibrated with CPLEX, we have also to consider that Gurobi may result penalized.

For a fair comparison, we wanted to make sure that both CPLEX and Gurobi solvers operated in the same conditions, to avoid that one solver were disadvantaged by some solver-specific features or different behaviours of the other. As CPLEX and Gurobi can also exploit the capability of parallel computing, they could make use of all available cores to explore several nodes simultaneously. Unfortunately, as already reported in [ASJ15], there are some issues when working with multiple threads and using the user-generated cuts on Gurobi. In fact, the user-cut callback is only ever executed in single-thread mode (see [GRB]) and, as a consequence, the additional cuts generated by our separator are added only periodically to the model formulation. This may lead to situations where Gurobi explores child nodes without taking into account the violated cuts in the parent node. A solution to this problem can be the setting of the thread count to one and this ensures that the cuts are applied immediately. At the same time, turning off parallel computation can lead to a non-realistic performance.

For all these reasons, we will first make some "fair" comparisons between CPLEX and Gurobi, in single-thread mode only. On this context, we will try to enable and disable the user-cut and the heuristic callbacks, in turn. The lazy-constraint callback is always kept enabled, as it is strictly required for the correctness of the model. Then we will compare Gurobi with itself, trying to switch the number of threads from one to two, in order to realize how much the parallelism can influence the optimization performance. In this case, the user-cut and the heuristic callbacks will be kept disabled.

Finally, we will briefly outline a computational comparison also for the Local Branching matheuristic (described in Section 3). We chose, instead, not to report also the runs with Hardfix matheuristic, as it involves some

solver-specific features and the comparison would not have been fair.

4.2 Testbed

The testbed we used is composed by the same 18 instances, randomly selected, we described in Chapter 2.5 (ref. to Section 2).

4.3 Computational Results

4.3.1 Solve to proven optimality

The first comparison we report is between CPLEX and Gurobi in single-thread mode, with all the callbacks enabled (lazy, user-cut and heuristic). The runs are shown in Figure 6, where we reported the respective upper- and lower- bounds over time. As the time is on a logarithmic scale, the very first time slice is not very significant. We chose to represent only the real wall-clock time, and not other units, because we believed this metric it more suitable to compare the overall performance of two different solvers, although it is less accurate from a strictly algorithmic point of view.

On this challenge, CPLEX reaches the proven optimality all the times Gurobi does, plus two more instances. Gurobi reaches at the root node a lower bound which, in many cases, is not very satisfactory, probably because of the smaller amount of time spent at the root node. This probably due to the fact that Gurobi inserts less cuts to the model formulation and, as the resulting LP relaxation is thinner, the root node can be solved quickly. Obviously, the drawback emerges on the lower bound value. In fact, CPLEX seems to prefer to spend many more cut-loop iterations at the root node, in order to fully exploit the deeper cuts we provide. In general, root node gap improvement is a good indicator of the quality of cuts, but in the UFL case we know for sure our cuts were rather efficient in reducing computational time and the number of enumeration nodes. Indeed, as Table 6 reports, our separator procedure for the UFL problem proved to be extremely effective with CPLEX, while Gurobi seems not exploiting it properly, in particular when solving B&B nodes.

For this reason, we made another comparison between the two solvers and this time we disabled our cut separator, that is, for both of the solvers we turned the user-cut callback off. Surprisingly, Figure 7 shows that, on a fair comparison, Gurobi demonstrates better abilities to reduce the gap and, in particular, to increase the best lowerbound, even at the nodes. Most probably, because Gurobi internal separator can generate deeper cuts than CPLEX for this problem.

To have a confirmation of what observed above, we made a test also with Gurobi itself, with the user-cut callback enabled and disabled. In Figure 8 we notice no substantial differences between the two runs and we can say that Gurobi does not exploit effectively the additional cuts we provided through our separator procedure inside the user-cut callback.

The reasons for such a result are several, but we identified two main contributions:

Instance	n, m	z_{opt}		gap_{final}		gap_{root}		LB_{root}		t_{root} [s]		B&B nodes		t_{final} [s]		Optimization Status	
		CPX	GRB	CPX	GRB	CPX	GRB	CPX	GRB	CPX	GRB	CPX	GRB	CPX	GRB	CPX	GRB
1000-10	1000	1434164.0000	1434154.0000	0.0001	0.0001	0.0085	0.5804	1429271.5266	812831.2353	19.7	14.0	114	1060	58.5	928.0	OPT.	OPT.
1500-10	1500	2000837.0000	2004074.0000	0.0001	0.0040	0.0232	0.7340	1982195.9143	777349.6106	24.3	22.0	396	675	408.5	1800.0	**OPT.**	T.LIM.
2000-10	2000	2558118.0000	2598627.0000	0.0001	0.0230	0.0409	0.7987	2516533.8569	760894.4430	34.7	29.0	95	664	401.7	1808.0	**OPT.**	T.LIM.
2500-10	2500	3102074.0000	3158120.0000	0.0015	0.0325	0.0072	0.9336	3096372.7736	541966.1755	335.3	21.0	1350	884	1800.0	1801.0	T.LIM.	T.LIM.
3000-10	3000	3573788.0000	3638619.0000	0.0045	0.2318	0.0837	0.7181	3395705.3774	1436940.6659	53.7	112.0	882	994	1800.1	1800.0	T.LIM.	T.LIM.
500-10	500	798577.0000	798582.0000	0.0001	0.0001	0.0222	0.7104	790462.5972	317160.0624	2.6	2.0	48	567	4.8	167.0	OPT.	OPT.
MO5	100	1408.7664	1408.7664	0.0000	0.0000	0.0371	0.1949	1356.4685	1237.1107	0.0	0.0	56	345	0.2	1.0	T.LIM.	OPT.
MP4	200	2938.7500	2938.7500	0.0000	0.0000	0.0492	0.1806	2794.2860	2523.4156	0.8	0.0	191	813	4.7	13.0	OPT.	OPT.
MQ3	300	4275.4317	4275.4317	0.0000	0.0000	0.0371	0.1816	4116.7052	3776.8440	1.7	1.0	60	624	4.4	22.0	OPT.	OPT.
MR2	500	2654.7347	2654.7347	0.0000	0.0000	0.0559	0.3234	2506.3659	2069.6700	6.5	2.0	211	940	40.4	92.0	OPT.	OPT.
MS1	1000	5283.7574	5283.7574	0.0001	0.0001	0.0683	0.6840	4922.8181	2875.6647	31.9	6.0	2157	5214	486.3	1036.0	OPT.	OPT.
MT1	2000	10069.8028	10246.9406	0.0641	0.4775	0.0950	0.6989	9130.7228	5260.9162	95.4	19.0	3730	891	1800.1	1801.0	T.LIM.	T.LIM.
ga250b-2	250	275141.0000	275141.0000	0.0001	0.0001	0.0132	0.0610	272351.3236	265401.0983	2.6	0.0	36818	39606	572.7	935.0	OPT.	OPT.
ga500a-2	500	511740.0000	511278.0000	0.0024	0.0015	0.2770	0.0270	373067.8219	505447.1467	3.0	6.0	24900	7925	1800.0	1800.0	T.LIM.	T.LIM.
ga750c-4	750	901634.0000	902329.0000	0.0238	0.0257	0.0290	0.0875	875448.7759	852616.9394	48.3	7.0	11042	2672	1800.0	1800.0	T.LIM.	T.LIM.
gs250a-1	250	257964.0000	257964.0000	0.0001	0.0001	0.2832	0.0169	186991.8398	255688.6190	0.5	1.0	134984	72841	1566.4	1535.0	OPT.	OPT.
gs500b-1	500	539133.0000	538501.0000	0.0097	0.0091	0.0120	0.0507	532861.4677	522136.8684	25.1	2.0	21620	8891	1800.0	1800.0	T.LIM.	T.LIM.
gs750c-3	750	902240.0000	903093.0000	0.0264	0.0277	0.0306	0.0854	874607.0401	846231.4747	41.5	5.0	10754	2529	1800.0	1800.0	T.LIM.	T.LIM.

Table 6: CPLEX vs Gurobi proven optimality - Single-thread - Usercut, Heuristic and Lazy callbacks ON

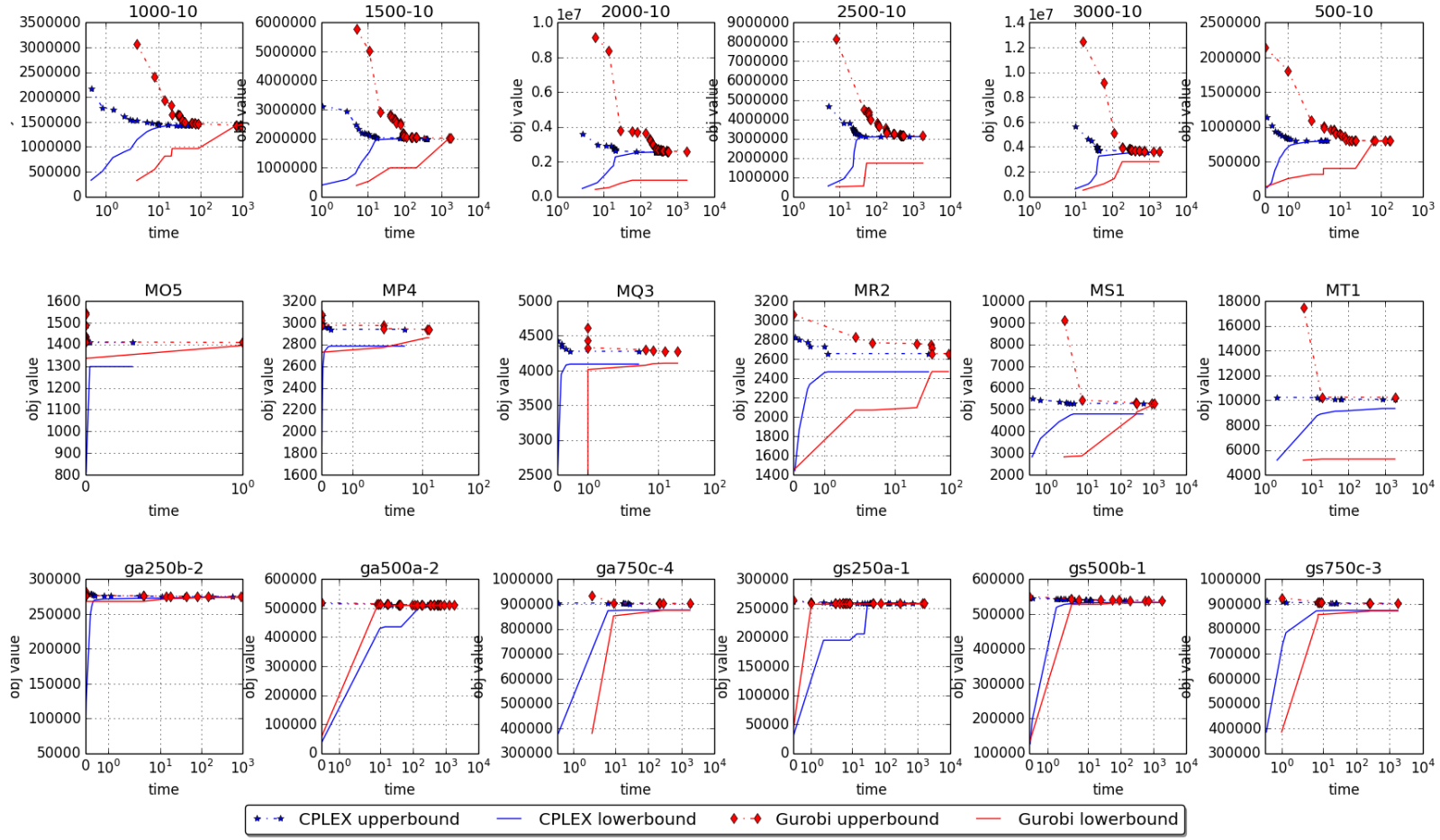


Figure 6: CPLEX vs Gurobi, single-thread comparison - Usercut cb ON - Heuristic cb ON - Lazy cb ON

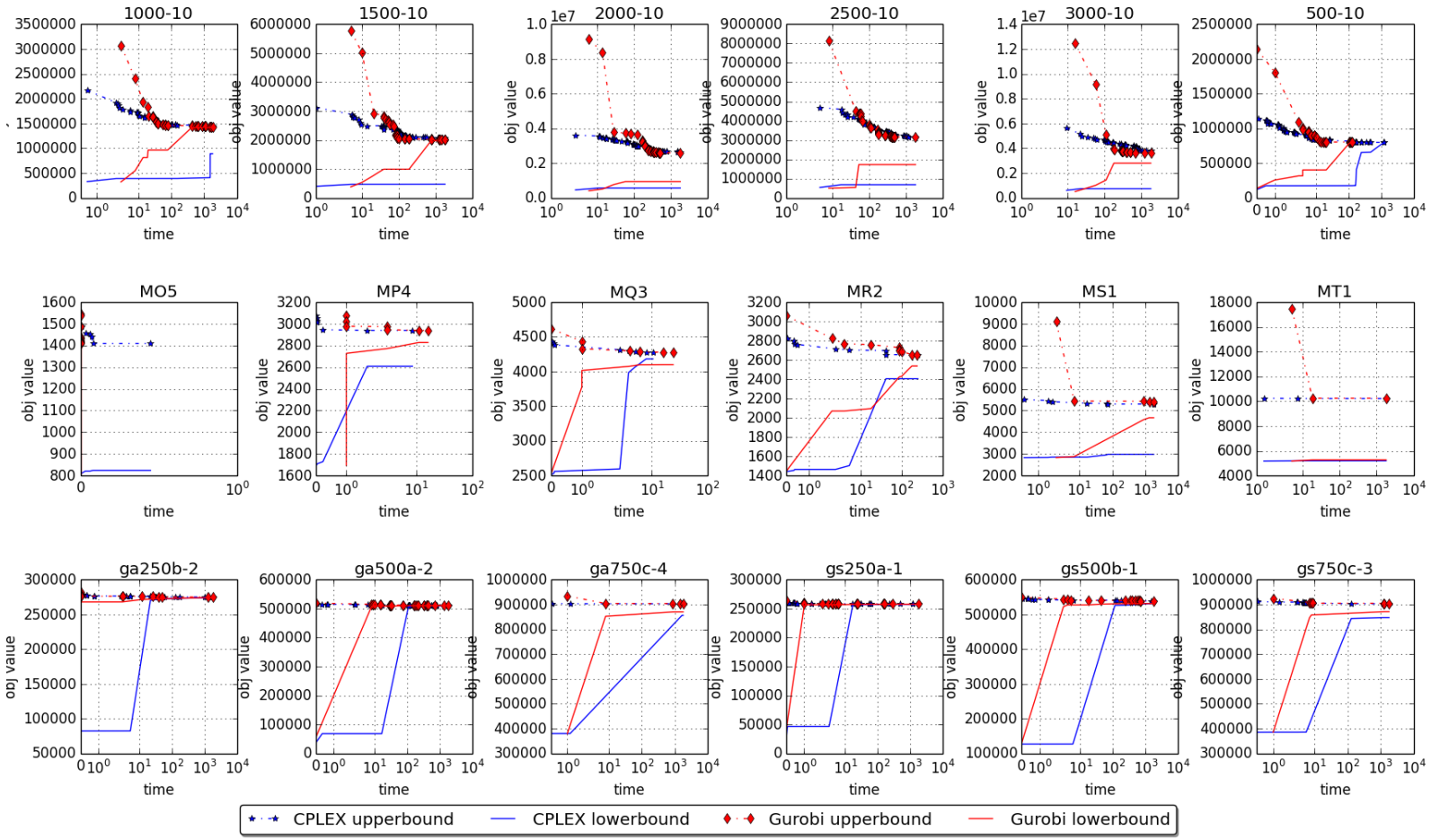


Figure 7: CPLEX vs Gurobi, single-thread comparison - Usercut cb OFF - Heuristic cb ON - Lazy cb ON

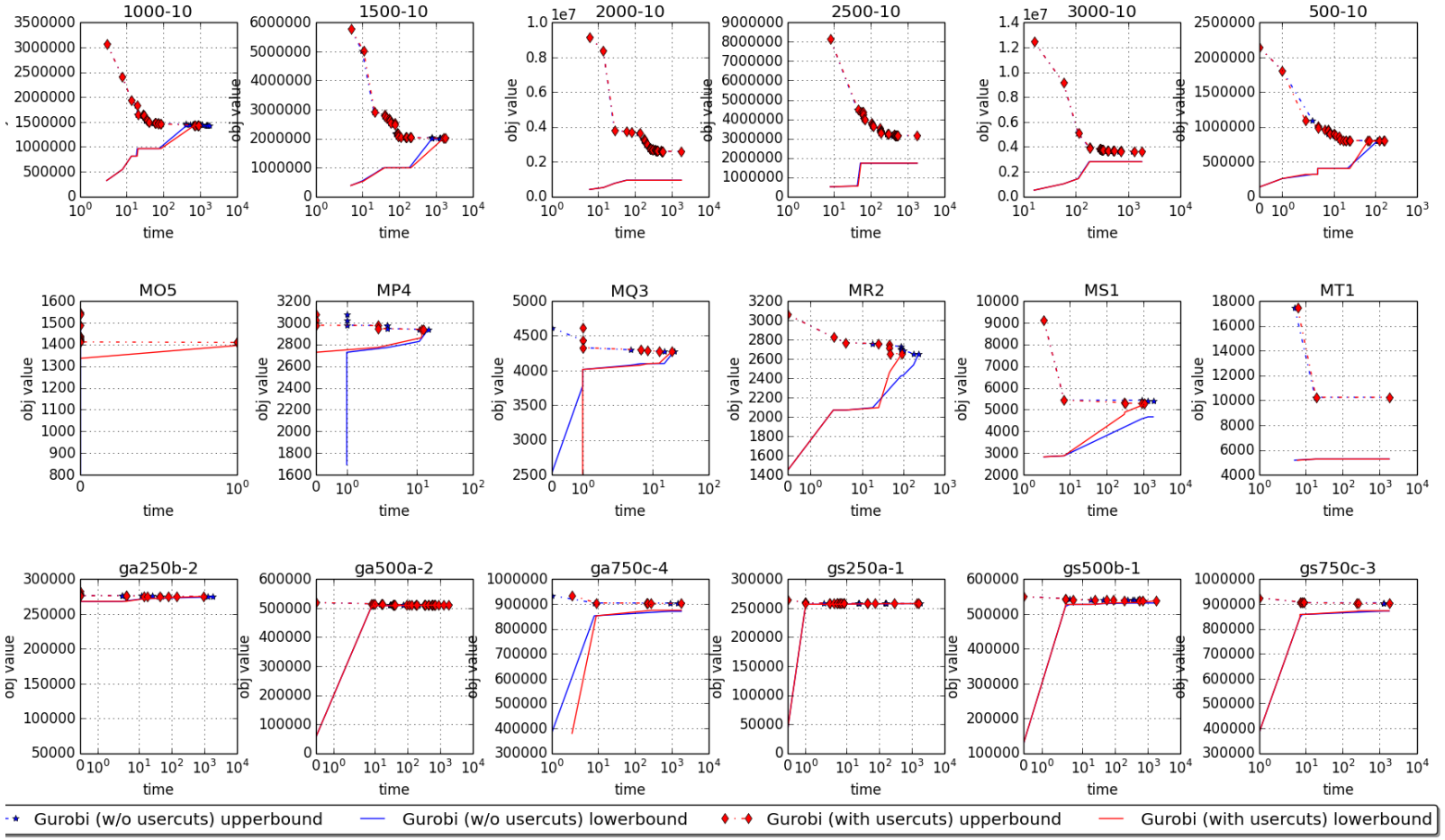


Figure 8: Gurobi single-thread comparison, with VS without Usercut cb - Heuristic cb ON - Lazy cb ON

1. After each new node has been solved, Gurobi calls our cut separator only once (with callback argument *where* = MIPNODE) and does not iterate the user-cut callback on the same node, as CPLEX instead does. The lack of a cut-loop may justify the lower number of user-cuts inserted on each B&B node.

2. At the same time, a lot of user cuts are purged too early, because of the Gurobi cut filter. In order not to hurt the performance, Gurobi developers claimed, that at root node as soon as cuts become inactive, they will be removed. At nodes, if there are more than one cuts, Gurobi will pick more promising cuts based on some measures, like violation [ZGRB10]. Cuts that are "relatively similar" to cuts already in the cut-pool are filtered out [DFI13] (e.g. cutting planes for which the angle between their normal vectors is smaller than a certain amount). As a result, only a small fraction of the user-cuts we generated are actually used and there is no way to force their insertion.

At this point, we wondered also how much the heuristic callback we provided was actually exploited by the solvers. On a fair comparison, where our algorithm is switched off and both solvers have only their internal heuristics, Gurobi is much more competitive (see Figure 9) than when our algorithm is turned on. We will come back on this in the next Chapter.

Until now, we had been performing all the runs on single-thread mode. Turning off parallel computation often can lead to a non-realistic performance, because on a realistic environment we want to exploit all the power of the solver. To see how much the performance can change when working in single-thread rather than with 2 threads, we compared Gurobi with itself, without the usercut and heuristic callbacks. Figure 10 shows that Gurobi scales very well and it cleverly exploits all cores available.

Finally, for completeness, we made a comparison also between the two solvers, both with 2 threads, both without the usercut and heuristic callbacks. As Figure 11 shows, they are both very competitive and its performance are almost comparable.

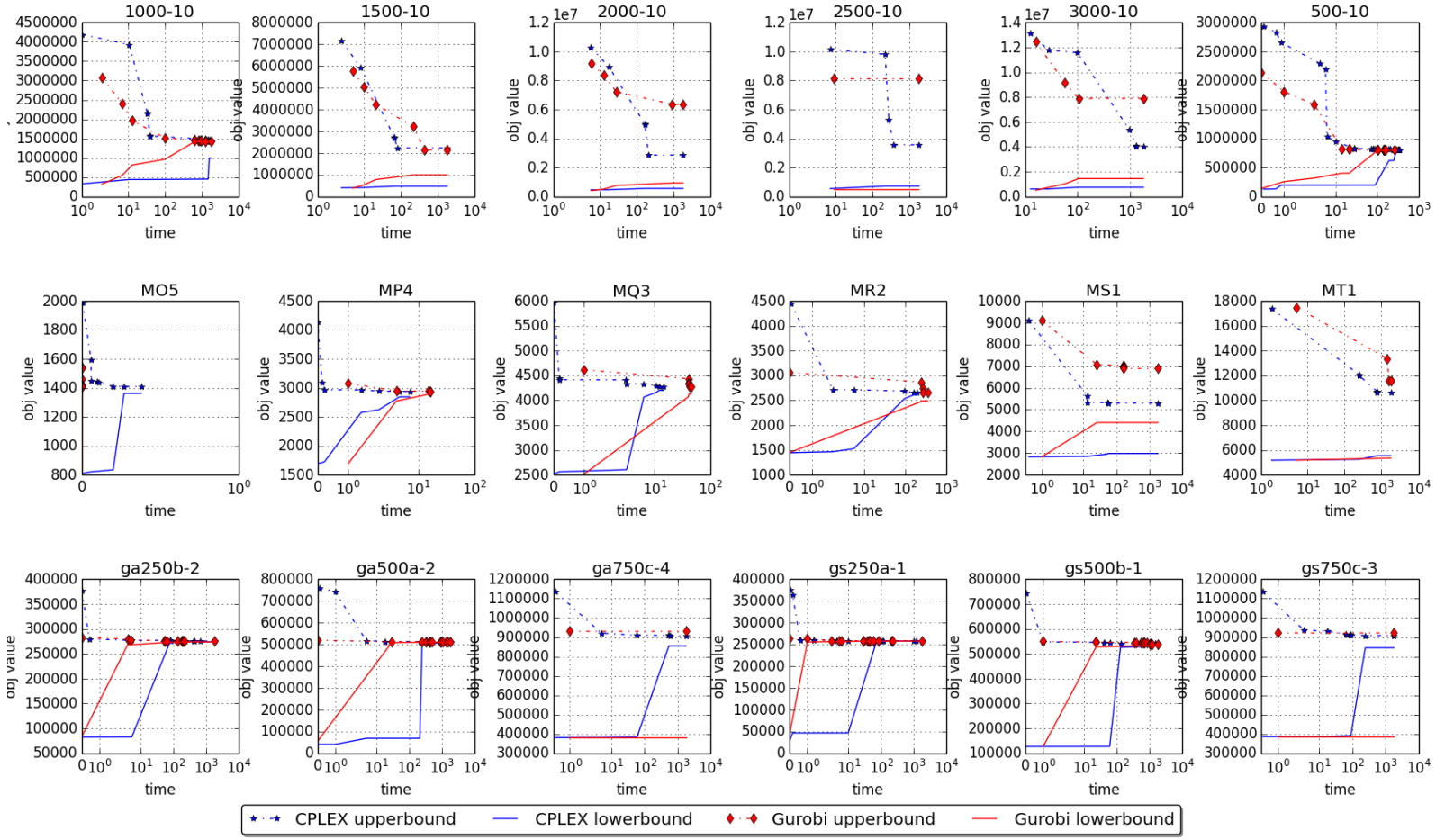


Figure 9: CPLEX vs Gurobi, single-thread comparison - Usercut cb OFF - Heuristic cb OFF - Lazy cb ON

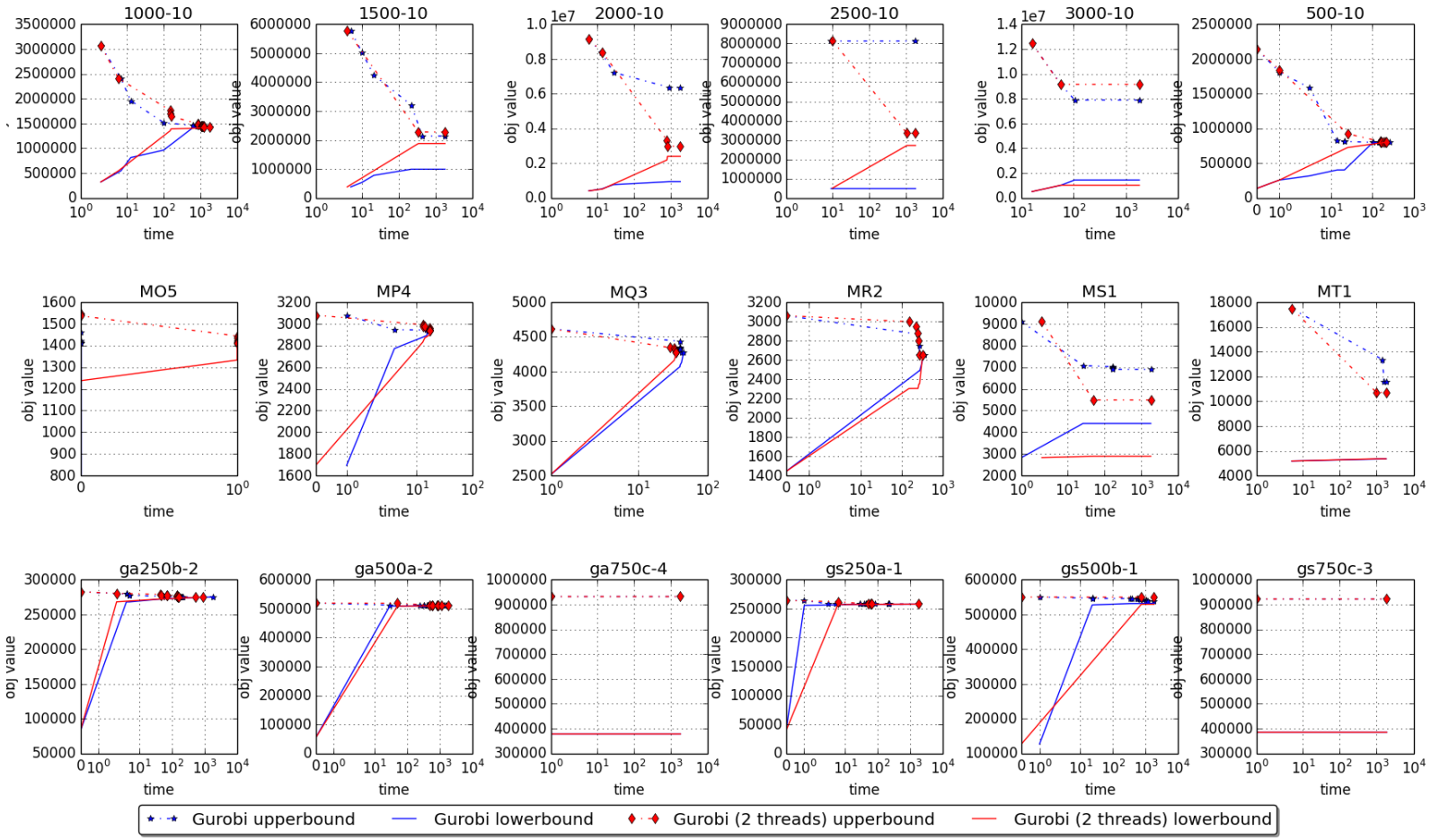


Figure 10: Gurobi performance with 1 or 2 threads - Usercut and heuristic cb OFF - Lazy cb ON

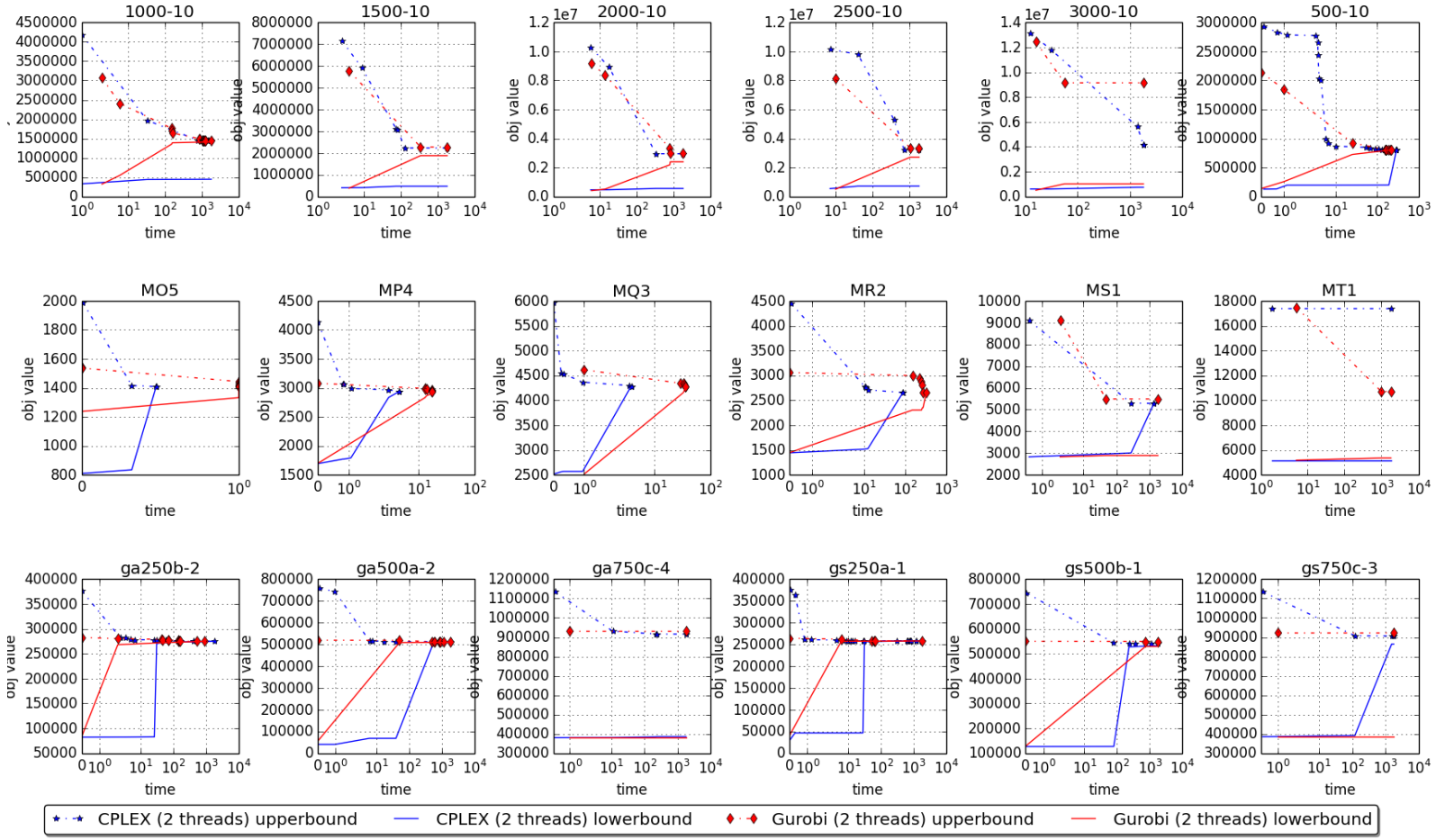


Figure 11: CPLEX vs Gurobi with 2 threads - Usercut and heuristic cb OFF - Lazy cb ON

4.3.2 Local Branching

We present, here, a computational comparison for one of the two matheuristics we developed in Section 2. Thanks to our interface, we reused the code of Local Branching and ran it both with CPLEX and Gurobi solvers.

For each instance, the execution had a time limit of 15 minutes and the root node computation was forced to end after a gap limit of 20% (from the lower bound) had been reached. In this case we are not interested in a proven optimal solution, but we are looking only for a good heuristic solution. The runs are shown in Figure 12, where we plotted for each instance the gap of the incumbent solution over time, from the optimal or best solution we knew. As the time is on a logaritmic scale, the very first time slice is not very significant. We chose to represent only the real wall-clock time, and not other units, because we judged this metric it more suitable to compare two different solvers, altrough it is less accurate.

Differently from Cplex, in many cases Gurobi seems terminating the root node processing much later and, often, before reaching the gap of 20% we had set. In fact, observing the results we reported in Table 7, we have also a numerical confirmation of this fact. Actually, we notice that for Gurobi the gap at the root node (g_{root}) is often higher than CPLEX, but sometimes Gurobi solves the root node with a better lower bound (LB_{root}). This means that the issue resides not only in the lower bound at the root node, but also in the upperbound, that is, Gurobi lacks of a good incumbent solution on this phase.

Observing the output log of the solver, we realized that our heuristic callback seems less effective for Gurobi. In certain cases, it happens that our custom heuristic finds a better integer-feasible solution, but when passing it to the solver, the incumbent solution remains the same and our hint is ignored. We suppose that this behaviour is due to the differences, that we already described, between CPLEX and Gurobi about the heuristic callback management. In fact, while CPLEX receives both the solution vector and the objective value from the heuristic callback, Gurobi allows only the transfer of the solution vector, taking the responsibility to compute its objective value. As Gurobi seems involved only periodically in such a computation, the objective value of the heuristic solution stays unknown to the branch-and-cut algorithm until the Gurobi feasibility-check ends. When the number of heuristic solutions found grows rapidly, having an objective value immediately available is very important, for example to give a priority to solutions and to discard the worst ones. Moreover, CPLEX allows us to insert a heuristic solution even without the need of a feasibility-check and this is why CPLEX seems performing better on this sense. Also there are some issues

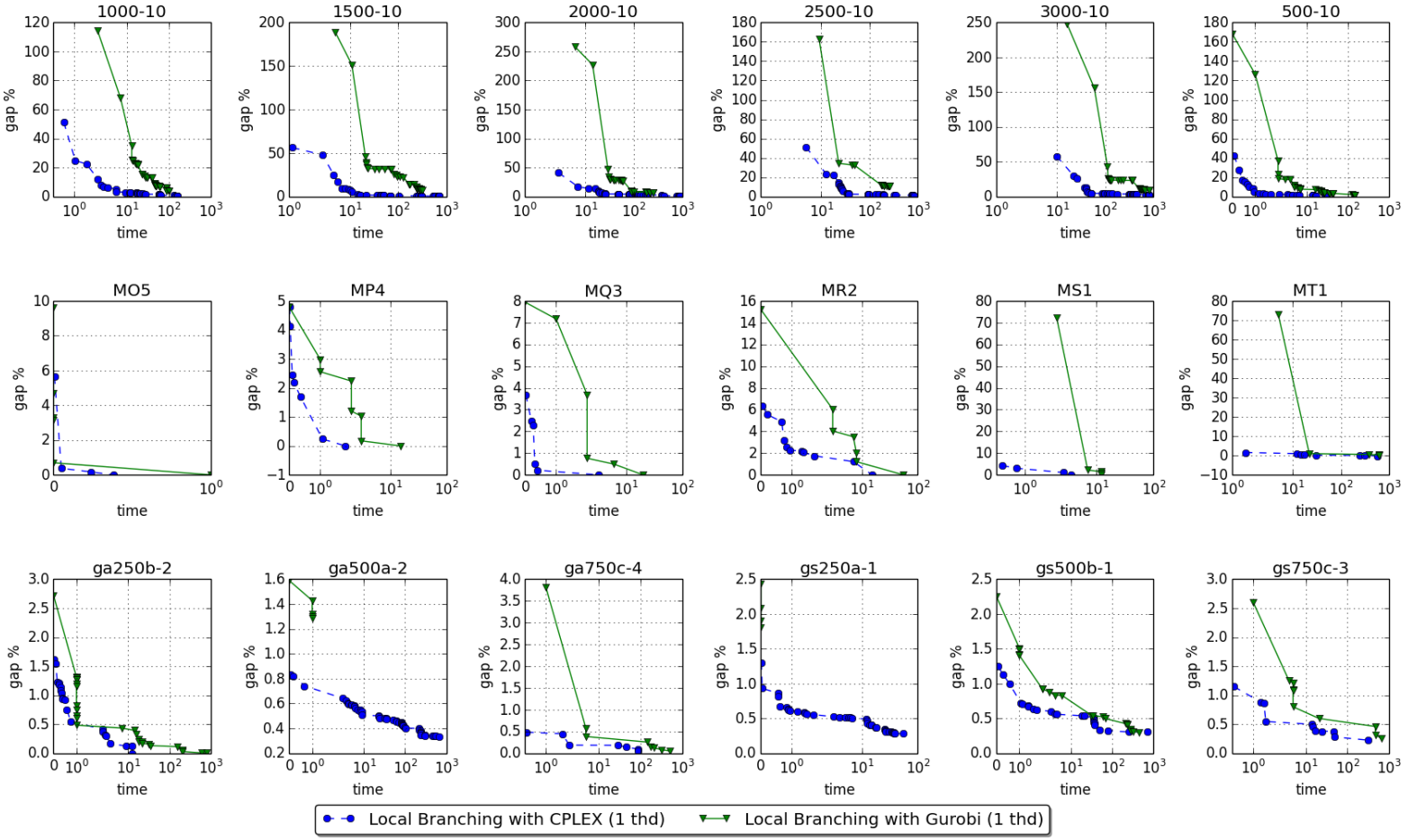


Figure 12: Gap from best solution known over time - Usercut, Heuristic and Lazy callbacks ON

Instance	n, m	z_{opt}		t_f [s]		t_{root} [s]		g_{root}		LB_{root}	
		CPX	GRB	CPX	GRB	CPX	GRB	CPX	GRB	CPX	GRB
1000-10	1000	**1439640.0000**	1485436.0000	232.8	206.2	**4.6**	13.7	**0.1798**	0.5804	**1249883.4586**	812863.1470
1500-10	1500	**2006022.0000**	2152761.0000	898.1	386.2	**13.2**	21.1	**0.1817**	0.7340	**1719316.5079**	777378.0653
2000-10	2000	**2577568.0000**	2733338.0000	897.5	564.0	**22.1**	28.6	**0.1999**	0.7987	**2186732.4772**	760940.5990
2500-10	2500	**3129081.0000**	3449333.0000	899.1	300.6	30.8	**22.3**	**0.1862**	0.8800	**2673620.7134**	541968.9256
3000-10	3000	**3648700.0000**	3914146.0000	898.9	899.4	**45.3**	111.2	**0.1941**	0.7181	**3054665.8218**	1436967.2225
500-10	500	**799263.0000**	807536.0000	39.4	235.8	0.9	2.5	**0.1813**	0.7104	**684597.7223**	317166.7865
MO5	100	1408.7664	1408.7664	1.9	2.8	0.1	0.1	**0.1888**	0.1988	1207.0835	**1237.1107**
MP4	200	2938.7500	2938.7500	21.0	52.8	0.1	0.1	**0.1540**	0.1806	**2588.5998**	2523.4156
MQ3	300	4275.4317	4275.4317	17.5	127.0	0.1	0.4	**0.1624**	0.1816	3713.3536	**3776.8440**
MR2	500	2654.7347	2654.7346	152.8	539.5	0.4	1.7	**0.1844**	0.2670	**2285.4275**	2069.8675
MS1	1000	5283.7574	5283.7574	899.5	899.4	**1.7**	6.0	**0.1839**	0.4709	**4441.4889**	2875.6778
MT1	2000	**10069.8028**	10089.4599	890.0	900.4	**9.0**	18.4	**0.1791**	0.4859	**8411.9819**	5267.4684
ga250b-2	250	**275141.0000**	275150.0000	885.2	899.3	0.2	0.2	0.1784	**0.0610**	228809.5708	**265401.0983**
ga500a-2	500	**513051.0000**	517898.0000	889.6	1.3	2.7	1.0	0.2758	**0.0270**	373067.8219	**505447.1467**
ga750c-4	750	900502.0000	900502.0000	894.3	899.4	1.0	3.7	0.1832	**0.0875**	738729.5652	**852616.9394**
gs250a-1	250	**258686.0000**	262617.0000	69.8	0.2	0.5	0.1	0.2818	**0.0382**	186991.8398	**254149.8239**
gs500b-1	500	539562.0000	**539529.0000**	899.0	899.0	0.9	1.0	0.1854	**0.0507**	442574.1053	**522136.8684**
gs750c-3	750	**903732.0000**	904007.0000	878.1	899.1	1.1	3.1	0.1920	**0.0854**	736923.7408	**846231.4747**

Table 7: Local Branching run, CPLEX vs Gurobi comparison - Usercut, Heuristic and Lazy callbacks ON

when passing a previously found solution through the *MIP Start*, because this kind of suggestion is very important between each Local Branching iteration and the others. Any inefficiency on this phase impacts negatively not only on the branch-and-cut algorithm but also on internal heuristics (e.g. RINS).

Finally, for the reasons written above and in the presence of some solver-specific features, as far as our Hardfix matheuristic we chose not to report its runs, as any comparison would not have been fair on these circumstances.

5 Conclusions

In our thesis, we developed an advanced example of a MIP problem and tested it on an modern MIP solver, CPLEX. Our purpose was to have a comparison also with another commercial solver, able to compete at an almost equivalent performance. Then, we developed an interface for Gurobi, in order to reuse the same code already working for the CPLEX APIs. Using both the UFL code and our interface, we finally made a computational comparison between CPLEX and Gurobi.

As the original code was developed and calibrated with CPLEX, Gurobi resulted penalized. On fair conditions and with the callbacks enabled, our UFL code interfaced with Gurobi had a significant loss of speed. At the same time, after switching the callbacks off, no substantial differences emerged between CPLEX and Gurobi, and rather Gurobi satisfied the positive expectations on many more cases.

We believe that such a result is due to a different type of management for some of the advanced features the two MIP solvers provide, in particular for the user-cut and the heuristic callbacks. In fact, as seen in Section 3, Gurobi user-cut callbacks appear less advanced, because:

- they work exclusively on single-thread;
- they do not offer any option to force addition of cuts;
- they offer only one point where to add cuts (*MIPNODE* context);
- they share the same "*where*" argument with the heuristic callback;
- the priority among heuristic and the user-cut procedures has to be managed manually by the programmer, as well as there is no option for tuning their respective frequency of execution.

Also, as outlined in Section 4, Gurobi heuristic callback does not provide any argument for passing the objective value of a new solution found to the solver, neither permits to bypass the feasibility check. User's heuristic solutions are considered only a hint for Gurobi and sometimes can be totally ignored, at Gurobi discretion (e.g. in case the solver is busy in other tasks or too many solutions have been proposed and there is no time to check them all).

As far as the lazy-constraint callback, its functionalities are fully preserved, as verified on Section 4. The same we can say for the informative callback.

Nevertheless, all the results we obtained from both the commercial solvers are very satisfying. Also the software interface we implemented resulted very

useful because it is not limited only to our UFL example, but it is ready to be applied on most codes already written for CPLEX. Existing software, then, can run directly with Gurobi solver preserving its functionalities and without the need of any migration process.

6 References

[FLS10] M. Fischetti, A. Lodi, and D. Salvagnin. “Just MIP it!”. In: Matheuristics. Springer US (2010): p. 39-70

[KMN13] E. Klotz, A.M. Newman. "Practical Guidelines for Solving Difficult Mixed Integer Linear.". Surveys in Operations Research and Management Science 18.1 (2013): p. 18-32.

[FLS15] M. Fischetti, I. Ljubic, M. Sinnl. "Thinning out facilities: a Benders decomposition approach for the uncapacitated facility location problem with separable convex costs." (2015). Submitted

[FL03] M. Fischetti, A. Lodi. "Local branching.", Mathematical programming 98.1-3 (2003): p. 23-47

[FM14] M. Fischetti, M. Monaci. "Exploiting erraticism in search.", Operations Research 62.1 (2014): p. 114-122

[COIN] COIN-OR website, *http://www.coin-or.org/*

[JUL] JuliaOpt website, *http://www.juliaopt.org/*

[LD15] M. Lubin, I. Dunning. "Computing in operations research using Julia.", INFORMS Journal on Computing 27.2 (2015): p. 238-248.

[B04] D. Baracco. “Interfacing a MIP heuristic based on ILOG CPLEX with different LP solvers”, master’s degree thesis (2004).

[CMK] CMake website, *http://cmake.org/*

[CPX] IBM ILOG CPLEX Documentation, Version 12.6.1, © Copyright IBM Corp. 1987, 2014

[GRB] Gurobi Optimizer Reference Manual, Version 6.0, Copyright © 2014, Gurobi Optimization, Inc. (website: http://www.gurobi.com/documentation/6.0/refman/)

[AIBM13] T. Achterberg, IBM archived material (2013).

[ZGRB10] Zonghao Gu, Gurobi forum (2010).

[ASJ15] S.J. Albinski. "A branch-and-cut method for the Vehicle Relocation Problem in the One-Way Car-Sharing.", master’s degree thesis (2015).

[DFI13] I. R. de Farias Jr, E. Kozyreff, R. Gupta, M. Zhao. "Branch-and-cut for separable piecewise linear optimization and intersection with semi-continuous constraints.", Mathematical Programming Computation 5.1 (2013): p. 75-112.

Appendix

Useful tips

We briefly collect here some useful tips we discovered during our work.

ParallelMode Setting the parallel mode switch to Deterministic (instead of Opportunistic) adds some synchronization points for the threads, in order to have a deterministic behaviour in the parallel search. Even though we renounce a faster performance, we have the warranty that the solver will always produce the same output, which is very useful for live demos or debugging purposes.

Callbacks When using Cplex callbacks, it is necessary to set CPX_PARAM_MIPCBREDLP parameter to CPX_OFF and CPX_PARAM_PRELINEAR to zero, in order to keep the original mapping we defined for the variables of the model.

Ticks The deterministic ticks that CPLEX measures is basically the number of memory accesses that CPLEX performs. Two runs using the same CPLEX binary with the same settings and the same data will produce the same deterministic time.

Mipstarts Remember that every change to the model discards the search tree, so there's no way to keep the cuts for the next start. Cut separation is typically cheap, though, so there's not a lot of scope for improving performance by reusing them.

Multithread When using more than one thread, be careful to the thread safety, in particular in the callbacks: when editing data, mutual exclusion techniques or, better, private structures for each CPU must be implemented.

Loop In case we dynamically try to add a non-violated constraint to the model, e.g. through a deterministic lazy callback or separator, the solver goes in loop, because the solver does not see any violated constraint and calls the user callback another time, with the same input. As the results are the same, the solver enters in an infinite loop.

Epsilon Due to numerical precision, we used an absolute TOLERANCE of $1 \cdot 10^{-5}$ whenever we compare values of type double. In particular, in case we are in a lazy callback and we are deciding

whether a constraint is violated or not, using an epsilon of tolerance is imperative to avoid an infinite loop. In fact, the solver may not recognize as violated a constraint we generated, because of different results during comparison.

Frozen Due to numerical instability, a thread can freeze during the runs, without raising any warning: it is useful to implement a counter to react, e.g. skip the node, when a thread is frozen.

Unix Scripts

- To solve multiple instances in batch:

```
#!/bin/sh
if [ -z "$1" ]; then # input argument is empty
    echo "Usage: ./list.sh data/instances"
else
    for file in $(ls $1 | grep -Ev ".opt|.bub"$)
    do
        if [ -f "$1/$file" ];then
            echo "Solving instance: $file"
          ./uflSolver $1/$file 2>&1 | tee -a data/$file.txt
        fi
    done
fi
```

- To extract the best known upperbound from solution files:

```
cat */*.bub | grep -o "[0-9.]*"$
```

- To read the stats from execution:

```
grep -E "STAT" output.txt
```

Python Scripts

- To plot the results we used the PyLab library for Python. An example:

```
#!/usr/bin/python

import sys, getopt, csv, re
from pylab import *

def main(argv):
  inputfile = []
  try:
    opts, args = getopt.getopt(argv,"hi:",["ifile="])
  except getopt.GetoptError:
    print 'test.py -i <inputfile>'
    sys.exit(2)
  for opt, arg in opts:
    if opt == '-h':
      print 'test.py -i <inputfile1.csv> -i <inputfile2.
                                csv> ..'
      sys.exit()
    elif opt in ("-i", "--ifile"):
      inputfile.append(arg)

  if inputfile == []:
    print 'No input files provided. Type -h for help'
    sys.exit(2)

  print 'Input file is "', inputfile

  for dataFile in inputfile:
    convertCsvToPlot(dataFile)

  show()


def convertCsvToPlot(inputfile):
  with open(inputfile, 'rb') as csvfile:
    lines = csv.reader(csvfile, delimiter=';')
    x = []
    y = []
    m = 3 # number of vertical sub-plots
    n = 6 # number of horizontal sub-plots
    p = 1
    for col in lines:
      if col[0] == 'HARDFIX':
        x.append(float(col[2])) #walltime
        y.append(float(col[3])) #obj value
```

```
        print '[', col[3] , ',' , col[2], ']'
        print ' '
      elif col[0] == 'LOCALBRANCH':
        x.append(float(col[1])) #iter
        y.append(float(col[3])) #obj value
        print '[', col[3] , ',' , col[1], ']'
        print ' '
      elif col[0] == 'STAT':
        try:
          found = re.search('[^/]*$', col[1]).group(0)
        except AttributeError:
          found = col[1]
          sys.exit(2)
        print 'was file: ', found
        print '\n'

        #divide by .bub or .opt
        opt = getStateOfArt(found)
        for k in range(len(y)):
          if (opt < float(y[k])):
            y[k] = abs(opt-float(y[k]))/(1e-10+abs(opt))
                                * 100
          else:
            y[k] = -abs(opt-float(y[k]))/(1e-10+abs(opt))
                                 * 100

        subplot(m,n,p)
        p += 1
        plot(x, y)
        xlabel('iter')
        #xscale('symlog')
        ylabel('gap %')
        title(found)
        grid(True)
        #reset variable arrays
        x=[]
        y=[]

    subplots_adjust(left=0.05, bottom=0.05, right=0.95,
                                top=0.95, wspace=0.5,
                                hspace=0.6)


def getStateOfArt(string):
  val = { '1000-10' : 1434154 ,
    '1500-10' : 2000801 ,
    '2000-10' : 2558118 ,
    '2500-10' : 3101800 ,
    '3000-10' : 3570766 ,
```

```
    '500-10' : 798577 ,
    'MO5' : 1408.76638 ,
    'MP4' : 2938.75002 ,
    'MQ3' : 4275.43167 ,
    'MR2' : 2654.734663 ,
    'MS1' : 5283.757394 ,
    'MT1' : 10089.459863 ,
    'ga250b-2' : 275141 ,
    'ga500a-2' : 511333 ,
    'ga750c-4' : 900044 ,
    'gs250a-1' : 257964 ,
    'gs500b-1' : 537931 ,
    'gs750c-3' : 901714
  }
  return val[string]

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

Acknowledgements

Ritengo innanzitutto doveroso porgere i miei più sentiti ringraziamenti al mio relatore, prof. **Matteo Fischetti**, per la sua disponibilità ed i suoi preziosi consigli, ma soprattutto per la passione che ha saputo trasmettere a noi studenti per questa materia.

Ringrazio infinitamente la mia famiglia, che mi è sempre stata accanto in ogni momento e mi ha dato tutti i mezzi per poter raggiungere questo traguardo.

Grazie ai miei genitori, **Leandro** e **Roberta**, per tutto.

Grazie a **Laura**, mia sorella, che mi ha sopportato sin dalla nascita e che ha intrapreso anche lei un arduo percorso (tieni duro!), nonostante i miei tentativi iniziali per farla desistere.

Grazie a **Laura**, mia morosa, che è entrata nella mia vita ed è la mia felicità.

Grazie a **Laura**, mia nonna, che aspetta sin da quando ero piccolo di esserci oggi, solo per poter vedere la laurea del suo primo nipote.

Ringrazio tutti i miei parenti, zii e zie, cugini e cugine, nonni e nonne, che hanno sempre fatto il tifo per me.

Ringrazio tutti i miei amici e tutte le persone che non hanno mai smesso di credere in me.

Un grazie a **Pancio**, che mi ha sempre accolto come un fratello.

Un grazie a **Edo**, che sin dalle elementari non mi ha mai abbandonato.

Un grazie a **Com** e **Pippo**, per l'altruismo dimostrato a Londra, e a tutti gli altri inseparabili amici delle superiori che mi sono vicini nonostante le mie assenze.

Ringrazio infine tutti i miei compagni di progetto, in particolare **Luca**, **Nicola**, **Mauro** e **Matteo** che hanno scelto di lavorare con me in più di una occasione. Non ci siamo mai accontentati del massimo, ma siamo sempre andati oltre!!

Padova, 12 ottobre 2015

Ale