Math. Prog. Comp. (2016) 8:113–132
DOI 10.1007/s12532-015-0096-0

FULL LENGTH PAPER

Improving branch-and-cut performance by random sampling

Matteo Fischetti[1] · **Andrea Lodi**[2] · **Michele Monaci**[1] · **Domenico Salvagnin**[1] · **Andrea Tramontani**[3]

Received: 30 September 2013 / Accepted: 2 October 2015 / Published online: 7 November 2015

Abstract We discuss the variability in the performance of multiple runs of branch-and-cut mixed integer linear programming solvers, and we concentrate on the one deriving from the use of different optimal bases of the linear programming relaxations. We propose a new algorithm exploiting more than one of those bases and we show that different versions of the algorithm can be used to stabilize and improve the performance of the solver.

Keywords Integer programming · Performance variability

Mathematics Subject Classification 90C10 Integer programming · 90C11 Mixed Integer programming · 90-08 Computational methods

✉ Andrea Lodi
andrea.lodi@unibo.it

Matteo Fischetti
matteo.fischetti@unipd.it

Michele Monaci
michele.monaci@unipd.it

Domenico Salvagnin
domenico.salvagnin@unipd.it

Andrea Tramontani
andrea.tramontani@it.ibm.com

1 DEI, University of Padova, Padua, Italy

2 DEI, University of Bologna, Bologna, Italy

3 CPLEX Optimization, IBM, Bologna, Italy

1 Introduction

We consider a general mixed integer linear program (MIP) in the form

$$\min\{c^T x : Ax \geq b, x \geq 0,\ x_j \in \mathbb{Z}\ \forall j \in I\} \tag{1}$$

where the knowledge of the structure of matrix A (if any) is not directly exploited and $I \neq \emptyset$. Thus, the algorithmic approach relies on the solution, through general-purpose techniques, of the linear programming (LP) relaxation

$$\min\{c^T x : Ax \geq b, x \geq 0\}, \tag{2}$$

i.e., the same as problem (1) above but with the integrality requirement on the x variables in the set I dropped.

State-of-the-art MIP solvers use LP computation as a tool by integrating the *branch-and-bound* and the *cutting plane* algorithms [20] within a general *branch-and-cut* scheme (see, e.g., [15,16] for an overview of the computational framework and the MIP software, respectively).

Performance variability In a highly influential talk in 2008, Danna [7] analyzed for the first time in an explicit manner some variability in performance of MIP solvers apparently unrelated to algorithmic reasons, and gave this phenomenon the name of *performance variability*. Danna's running example was the solution of a classical MIP instance called `10teams`: by using exactly the same release of IBM ILOG CPLEX (namely, version 11) the instance was solved in 0 branch-and-bound nodes and 2731 Simplex iterations on a Linux platform, while it needed 1426 nodes and 122,948 iterations on an AIX one.

That severe and unexpected variability in performance was later explained in [14] essentially as *imperfect tie-breaking*. Many crucial decisions within the branch-and-cut framework implemented by all MIP solvers are guided by the computation of scores for several candidates and the selection among those candidates is based on the score. Of course, such a score is generally far from "perfect", whatever perfect could mean, and variability is often observed, when, in case of ties in the score, secondary criteria are used to break it. In these cases, the selection can be made arbitrarily (although deterministically), e.g., based on the order in which the candidates are considered, or can be influenced by (tiny) rounding errors that are different in different computing environments. Note that the above discussion highlights the fact that performance variability is not restricted to running the same code on different machines/computing platforms but might appear on the same machine if, for example, seemingly neutral changes of the mentioned order of the candidates (or variables) are performed. (The reader is referred to [18] for a recent tutorial on the mechanisms and effects of performance variability in MIP.)

Optimal simplex bases and variability Among all possible sources of variability, a somehow "obvious" one is associated with the degeneracy of the optimal basis of the (initial) LP relaxation. It is well known that many LPs are indeed highly dual degenerate, i.e., many equivalent optimal bases can be enumerated within the optimal face by randomly pivoting on variables with zero reduced cost. To the best of our knowledge, all current implementations of the simplex algorithm (of any type) as well as the crossover phase of interior point algorithms for LP, return an arbitrary basis among those that are optimal. That means, for example, that a random permutation of the variable order used to express a MIP instance might then very likely determine a different optimal basis to be returned by the LP solver in case degeneracy exists. Although all theoretically equivalent, these alternative bases have instead a huge and rather unpredictable impact on the solution of the MIP because they do affect immediately three of the main ingredients in MIP computation, namely cutting plane generation, primal heuristics and, of course, branching. Recall indeed that cutting plane generation highly depends on the basis of the simplex tableau at hand (e.g., Mixed-Integer Gomory cuts, see, e.g., [6]), and most of the cheap and extensively used primal heuristics perform rounding operations on the LP fractional solution(s) (see, e.g., [1]). As a matter of fact, the selection of the optimal basis, even the first one, i.e., that within the optimal face of the very first LP relaxation, appears as a crucial decision for the evolution of the whole MIP enumeration. In addition, because it is encountered at the beginning of the whole branch-and-cut algorithm (after preprocessing only), that source of variability seems to be the first one that needs to be understood.

Our contribution Of course, one can optimistically think of performance variability as an opportunity. More precisely, the execution can be opportunely randomized so as to exploit variability, especially within a parallel algorithm. This is essentially what has been tried within the Constraint Programming and Satisfiability communities, see, e.g., [11], in which restart strategies are proposed for the solution of satisfiability instances. The basic idea is to execute the algorithm for a short time limit, possibly restarting it from scratch with some changes (and possibly with an increased time limit), until a feasible solution is found. In a similar spirit, within MIP, a heuristic way to take advantage of erraticism has recently been suggested in [10], where the proposed algorithm executes for a short time limit a number of different sample runs, and some criteria are then applied to select a single run that will be executed for a long(er) computing time. The existence of alternative optimal LP solutions—which is a main source of variability—is also used to improve cut filtering in [2]: given the current optimal LP solution x^* along with a collection of cuts violated by x^*, an alternative optimal solution x' is first constructed through "pump-reduce" (a light version of the feasibility pump heuristic [9]), and then used to discard the cuts that happen not to be violated by x'. Finally, in [5] multiple threads are used to exploit variability by running a different parameters' configuration in each of them and allowing various degrees of communication. In one of the proposed implementations (named *CutsComm*) the pool of cutting planes generated at the root node by each thread is shared with the other threads—in the concluding remarks of the paper, however, it is observed that the communication of cutting planes seems not to be very helpful to improve the overall computing time, at least in the proposed setting. Similar ideas were already introduced in [19], and have recently been used for the racing ramp-up phase of ParaSCIP (see [21] for details).

In this paper we concentrate on the variability associated with the selection of the optimal basis. We propose an algorithm that essentially samples the optimal face of the initial LP relaxation(s), and for each of the samples, executes the solver's default cutting plane loop and applies the default primal heuristics. At the end of this process, for each sample (i.e., every different initial optimal basis) cutting planes and feasible solutions are collected and used as input for a final run.

This sampling scheme can be implemented in parallel by feeding K threads with K alternative optimal bases. By changing the value of K we can push in two somehow opposite directions.

- On the one hand, we are interested in *reducing variability* by stabilizing the run of a MIP solver. By using a relatively large value of K we are able to empirically show the strong correspondence between the high variability in performance of multiple runs of a MIP solver and the degree of degeneracy of the optimal bases on the MIP instances from the *benchmark* and *primal* sets of the MIPLIB 2010 [14] testbed. The algorithm has a much more stable computational behavior and is able to:
  1. strongly reduce the variability of the root node of MIP executions both in terms of primal and dual percentage gap closed (value and standard deviation), and
  2. significantly reduce the deterministic time and the number of branch-and-bound nodes of the same MIP executions.

  The stability is also testified by the fact that the algorithm solves to optimality within the time limit more instances than the reference MIP solver. Finally, as a byproduct, the algorithm is able to (optimally) solve at the root node, i.e., without enumeration, a significant number of difficult instances in the *primal* set of the MIPLIB 2010 testbed, thus showing a quite interesting behavior as a heuristic for instances in which the challenge is to find the primal solution while the dual bound is already tight.
- On the other hand, the above "stabilized" algorithm might not be so effective in terms of pure running times because of an unavoidable overhead due to synchronization and to the fact that the threads used by our algorithm are subtracted from other (crucial) operations performed by a MIP solver. In other words, the computing time reduction obtained during the enumeration phase may be outweighed by the root node overhead. A compromise can be obtained by a small value of K, thus achieving the aim of *exploiting variability* and getting an algorithm whose performance in terms of running time compares favorably with the state-of-the-art of MIP solvers. Indeed, while we started our study on performance variability with IBM ILOG CPLEX (CPLEX for short) version 12.5.0, the preliminary results were so encouraging that a commercial implementation of this second option was made and is now included in the default of the latest IBM ILOG CPLEX version 12.5.1.

Organization of the paper The next section is devoted to a detailed analysis of the variability showing that such a variability is independent of the MIP solver considered. Section 3 describes our algorithm and the computational setting we use to assess the results. Section 4 reports the computational experiments aimed at showing how the scheme can be used to substantially improve the stability of the solver. In Sect. 5 we describe the implementation of the algorithm within the commercial MIP solver CPLEX and the improved performance that is achieved. Finally, in Sect. 6 we draw some conclusions and outline open questions and future research directions.

2 Cross-solver performance variability

A good variability generator can be obtained by permuting columns and rows in the original model. This affects all types of problems and all components of a typical MIP solver. For each instance in the *benchmark* and *primal* sets of the MIPLIB 2010 testbed, we generated ten different row/columns permutations and tested the 3 major commercial solvers, namely IBM ILOG CPLEX 12.5.0 [13], GUROBI 5.1 [12] and XPRESS 23 [8]. Indeed, as already mentioned, changing the order of the variables and constraints of a MIP instance has a quite direct effect on the optimal basis of the first LP relaxation in case of dual degeneracy (which is usually the case). For each instance, we computed the variability score [14] of computing time and branch-and-cut nodes, which is defined as the ratio between the standard deviation and the arithmetic mean of the n performance measures obtained with n different permutations of the same instance.

Scatter plots of the variability scores for all pairwise comparisons between solvers can be found in Fig. 1, whereas detailed results are shown in Table 8 in the “Appendix”.

According to the plots, performance variability is clearly not specific to only a given solver. In particular, all commercial solvers exhibit a comparable performance variability on our testbed, both in terms of running times and enumeration nodes. As far as a possible correlation is concerned, the result is less clear: while there are quite a few instances that are either unstable with all solvers or with none, many of them behave quite differently with different solvers. This is not surprising, as performance variability is affected by many factors including:

– Problem difficulty: if a problem is consistently easy (or too hard) for a given solver, then no performance variability can be measured on that solver. However, the same instance can be easy for a solver and difficult for another.
– Preprocessing: different solvers apply different preprocessing reductions, which greatly affect the actual model to be solved. As such, any piece of information that we can collect on the original formulation (the only one that is common to all solvers) can be quite uninformative of the formulation that is actually solved.

3 The algorithm

A natural and mathematically sophisticated question to be asked about LP degeneracy within the branch-and-cut algorithm concerns the existence of a “best” basis among all optimal ones. Although this seems a reasonably well-posed question, there are a number of reasons why we are far away from being able to answer it directly. First, the selection of a basis impacts multiple ingredients of the enumerative algorithm: cutting plane generation and primal heuristics somehow immediately, as already mentioned, but branching is also involved later on. Hence, it is conceivable, and actually likely, that a “good” basis for cutting planes might be not so “good” for primal heuristics and vice versa. Moreover, the heuristic nature of many components of the algorithmic framework (see, e.g., [17] for a extensive discussion on the topic), row aggregation in cutting plane generation just to mention one, makes the characterization of the *best*

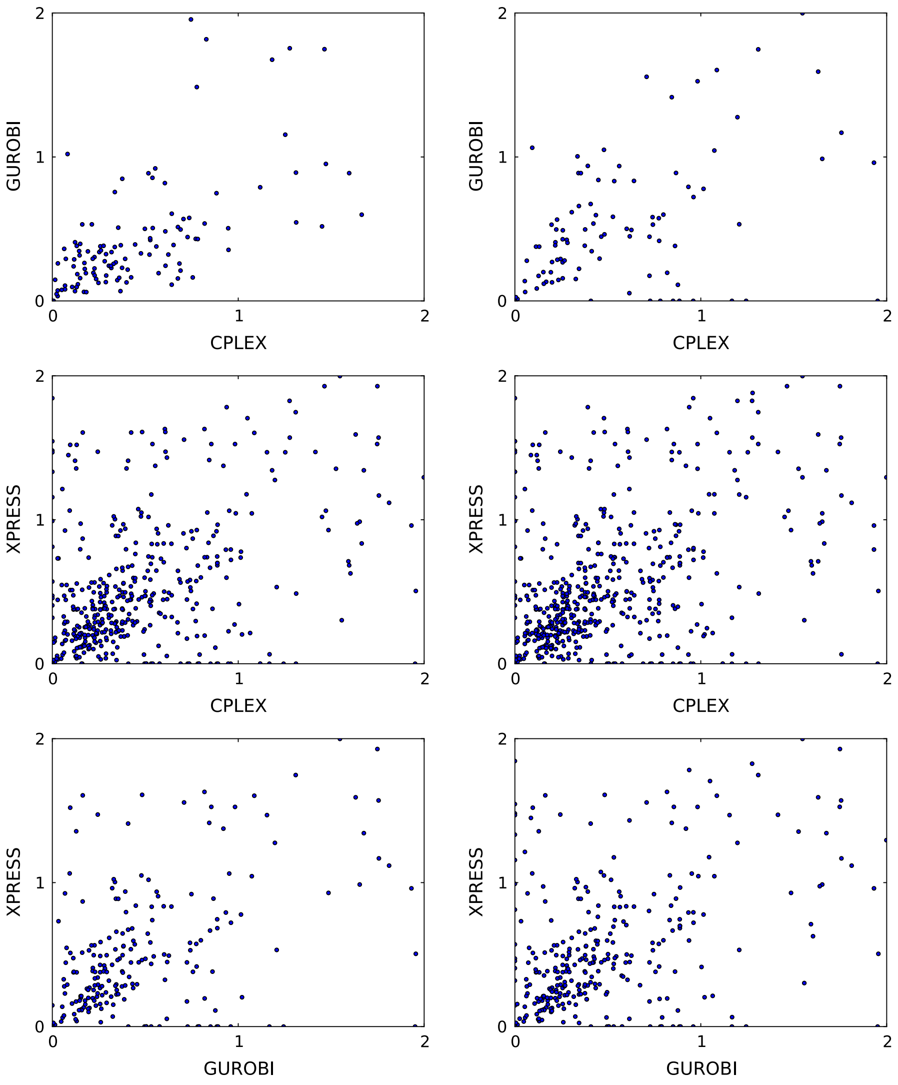

Fig. 1 Variability Comparison across solvers. Each point represents the variability scores of a given instance in the testbed w.r.t. two solvers and a given performance measure (time on the *left* and nodes on the *right*). For readability, values have been clipped to the range [0, 2]

basis impossible in the practical setting we are interested in, i.e., within a real MIP solver.

Thus, the basic idea guiding our algorithm is that, in presence of LP degeneracy and in the practical impossibility of characterizing the *best* basis, a good strategy is to *sample* the optimal face of the (initial) LP relaxation and collect both cutting planes and feasible solutions while executing the cut loop for each of the sampled bases. (The actual sampling procedure we use will be described in Sect. 4.1). All collected cuts and solutions are then put together in a (hopefully) stabilized and enriched root node from which the enumeration is started in a subsequent run. Formally, the above scheme is turned into Algorithm 1.

Algorithm 1: `ksample`

```
   Input: a MIP instance
   Output: an optimal solution
 1 preprocess the MIP instance and store it;
   // root-node sampling, with default cut separators and primal heuristics
 2 for $i = 1, \dots, K - 1$ do
 3     sample an optimal basis $B_i$ of the (initial) LP relaxation;
 4     while executing the default root-node cut loop starting from $B_i$ do
 5         collect cutting planes in a local cut pool $P_i$;
 6         collect feasible solutions in a local solution pool $S_i$;
   // final run
 7 solve the stored MIP instance (without any further preprocessing) by using the aggregated pools
   $P := \cup_{i=1}^{K-1} P_i$ and $S := \cup_{i=1}^{K-1} S_i$;
```

For multiple reasons discussed in the introduction, there is a tradeoff between stabilization and (improved) performance. We use a unique parameter to push for different goals, namely the number K of root nodes processed ($K - 1$ in sampling, and one in the final run). Roughly speaking, a higher value of K guarantees a more stable behavior, while a small one allows one to exploit variability by getting improved performance. The computational setting in which the above simple scheme is used to obtain those somehow competing goals is described in the next section with special emphasis to stability, whereas in Sect. 5 the performance emphasis is given.

4 Implementation

We implemented our codes in C++, using IBM ILOG CPLEX 12.5.0 [13] as black-box MIP solver through the CPLEX callable library APIs. All tests in this section have been performed on a cluster, where each node is equipped with an Intel Xeon E3-1220 V2 CPU running at 3.10 GHz and 16 GB of RAM.

We tested our codes on the problem instances from the *benchmark* and *primal* sets in the MIPLIB 2010 testbed, for a total of 121 instances.

4.1 Sampling procedure

Having assessed that performance variability is common to all commercial solvers, we will now try to exploit it to improve the stability and performance of the solution process. From now on, we will only use CPLEX as our testing framework. Because controlling the load of the nodes of our computational cluster is not trivial (the cluster being shared), we are not reporting wall clock running times but rather deterministic times which are an abstract unit, based on counting memory accesses [3], used by

CPLEX to report a deterministic measure of the computational effort that is independent of the current load of the machine. Each process was given a deterministic time limit of 10^7, which corresponds approximately to a real wall clock time limit of 3 h on our machines. As far as performance measures are concerned, we always report shifted geometric means, with a shift of 10^3 for deterministic times and of ten for node numbers. The shifted geometric mean of values $t_1, \ldots, t_n$ with shift s is defined as $\sqrt[n]{\prod(t_i + s)} - s$, see, e.g., [1]. Further, using a *geometric* mean prevents hard instances at or close to the time limit from having a huge impact on the measures, while the shift is used to reduce the effect of very easy instances in the mean values. Thus, the shifted geometric mean has the advantage that it reduces the influence of outliers in both directions.

A possible way to obtain K different root nodes would be to exploit LP degeneracy and to load into CPLEX K different optimal bases of the first LP relaxation, obtained by applying some random pivots on the optimal face of the LP relaxation; see [10] for details. However, for the sake of simplicity, we decided to mimic this random perturbation by using the "random seed" CPLEX parameter (`CPX_PARAM_RANDOMSEED` in the CPLEX callable library), which is publicly available since version 12.5.0. The random seed parameter controls the initialization of the random number generator that CPLEX uses in some of its internal operations with the purpose of breaking ties in candidate selection and speeding up the computation. The random number generator affects several components of the solver (e.g., heuristics), and in particular it is used to deal with LP degeneracy and accelerate the converge of the LP solvers. As discussed in Danna [7], changing the random seed is almost a perfect way to enforce random variability, because it introduces a seemingly neutral change to the solution process that in reality can have a major and unpredictable impact on the path taken by the solver.

By using the above random seed parameter, our sampling procedure was implemented as follows. All instances were preprocessed once at the very beginning, independently of the random seed used, in order to have feasible solutions and cutting planes expressed in the same space. Then, we ran CPLEX with $K - 1$ different random seeds, stopping each run at the end of the root node, and collecting all the cutting planes generated in these $K - 1$ root nodes and the primal feasible solutions (if any, usually found by CPLEX primal heuristics).

Finally, we used these pieces of information for a final branch-and-cut run. Both the sampling phase and the final enumeration were done with traditional branch and cut (no dynamic search), no preprocessing, and by using 1 thread only. In order to simulate a real-world implementation, in which the sampling root nodes are executed in parallel and sampling information is available only at the end of the root node processing, we implemented the final run in the following way:

– all warm starting information is discarded and the final run is started from scratch;
– at the end of the root node, we use CPLEX callbacks to provide all the pieces of information collected by sampling, i.e., the best feasible solution and cutting planes;
– the run continues without any further action from callbacks.

We denote with `ksample` the above procedure, with $K > 1$. It is easy to see that we can "simulate" a traditional CPLEX run just by setting $K = 1$ (in the following, we will call this method `cpxdef`). Note that preprocessing the instances once at the very beginning and disabling presolve in the final run is only an approximation of a standard default CPLEX run: one side effect, for example, is that probing is disabled in this way. The effect is usually minor, but a few pathological instances that are easy with CPLEX default turn out to be very time consuming with our codes (both `cpxdef` and `ksample`): for these reasons, we removed instances `bley_xl1`, `ex9` and `ex10` from our testbed.

4.2 Root node stability

We ran `cpxdef` and `ksample`, with $K = 10$, on all the instances in our testbed, except for instance `glass4` because of numerical issues (slightly different optimum objectives were reported with different random seeds), and the 3 infeasible instances `ash608gpia-3col`, `ns1766074` and `enlight14` as they have no integrality gap.

In order to measure the performance variability at the end of the root node, each instance was solved by each method with 10 different random seeds. To be more specific, we first randomly generated 10 nonnegative integers $S_0, \ldots, S_9$, to be used as random seed for each final run, for both `cpxdef` and `ksample`. In addition, `ksample` also uses the current seed associated with the final run to generate nine additional random seeds, which are used to process the additional nine root nodes during the sampling phase. For each execution, we recorded the final primal and dual bounds at the end of the root node of the final run, indicated with $\overline{z}$ and $\underline{z}$, respectively. Given those values, we compute the integrality gap `igap` as

$$\text{igap}(\overline{z}, \underline{z}) = \begin{cases} 0 & \text{if } \ \overline{z} = \underline{z} = 0 \\ 1 & \text{if } \ \overline{z} \cdot \underline{z} < 0 \\ \frac{|\overline{z} - \underline{z}|}{\max(|\overline{z}|, |\underline{z}|)} & \text{otherwise.} \end{cases}$$

A similar formula was used in [4] for measuring the impact of primal heuristics in MIP solvers. Finally, since we know the optimal value z^* of all the instances we can compute in a similar fashion also the primal gap `pgap` and the dual gap `dgap`. For each measure, we computed the average over the ten runs and the standard deviation. As to standard deviation, we first computed the standard deviation of each instance across the ten different seeds, and then took the average.

Average results for the remaining testbed of 117 instances are reported in Table 1; split results for the *benchmark* (79 instances) and *primal* (38 instances) sets are given as well.

According to Table 1, the average integrality gap is significantly reduced, in both testbeds. As far as stability is concerned, `ksample` turns out to be consistently more stable than `cpxdef`, with the only exception of `pgap` in the *benchmark* set, where it is roughly equivalent. Interestingly, the behavior of the method is not much different between the *primal* and the more general *benchmark* set (of course, in the primal case

Table 1 Root node comparison on the whole testbed

Testbed	Instance	igap		pgap		dgap	
		Avg (%)	St.dev (%)	Avg (%)	St.dev (%)	Avg (%)	St.dev (%)
Benchmark	`cpxdef`	53.80	5.66	45.20	6.35	22.32	0.89
	`ksample`	37.74	4.63	25.88	6.49	21.61	0.84
Primal	`cpxdef`	63.37	10.05	63.37	10.05	0.00	0.00
	`ksample`	31.79	3.28	31.79	3.28	0.00	0.00
All	`cpxdef`	56.91	7.08	51.10	7.55	15.07	0.60
	`ksample`	35.81	4.19	27.80	5.45	14.59	0.57

`dgap` is always 0 by definition). For this reason, in the following we will not split the results between *primal* and *benchmark*.

4.3 Branch-and-cut stability

According to the previous section, `ksample` proves to be effective in reducing the integrality gap and improving the stability at the root node. In the current section, we evaluate if such improvements carry over to the overall solution process. Note that the result is not obvious, since the sampling phase can do nothing about the variability induced by branching.

We report three performance measures, namely: number of instances solved and shifted geometric means of deterministic times and nodes. Note that some care must be taken in order to report deterministic times for `ksample`: ideally, since the overall idea is to mimic a possible implementation where the sampling phase is performed in a parallel fashion by solving the K root nodes concurrently in a multi-thread environment, we would not report the deterministic time spent in the sampling phase, but the one related to the final branch and cut only. However, this would give an unfair advantage to `ksample`, as the method would be accounted for 0 ticks whenever it can solve an instance *during* sampling. At the same time, removing those instances from the testbed would be unfair as well, as on those `ksample` is by design superior to `cpxdef`. For the above reasons, the deterministic time reported for `ksample` refers to the final branch-and-cut run when the instance is *not* solved during sampling, and to the sampling root node that solved the instance if the instance is solved during sampling.[1]

Table 2 reports these aggregated results on the whole testbed for five different random seeds. Again, we discarded instance `glass4` for numerical reasons. Thus, we are left with 120 instances. The table is split into two sections: `all` refers to the

[1] Note that even a parallel implementation of the sampling phase would introduce some unavoidable overhead in the method. Such overhead is disregarded in the results reported in this section, but is instead taken into account in Sect. 5, where a real parallel implementation of the sampling phase is evaluated.

Table 2 Branch-and-cut performance comparison on the whole testbed

Seed	Method	All		Opt		
		Solved	Det.time	Solved	Det.time	Nodes
1	`cpxdef`	107	172,584	104	109,569	2175
	`ksample`	108	151,004	104	96,370	1765
2	`cpxdef`	105	172,022	102	102,103	1871
	`ksample`	104	153,985	102	85,571	1296
3	`cpxdef`	107	170,467	104	104,152	2171
	`ksample`	106	142,778	104	84,352	1581
4	`cpxdef`	106	182,415	106	118,001	2314
	`ksample`	109	154,826	106	104,730	1816
5	`cpxdef`	105	199,470	104	119,466	2415
	`ksample`	111	148,777	104	100,566	1780

whole testbed, while `opt` reports the results only for the subset of instances solved to optimality by both methods. Note that nodes are reported only for the latter case.

According to Table 2, starting from a "better" root node can indeed improve the performance of the subsequent branch-and-cut run. Indeed, `ksample` consistently improves upon `cpxdef`, with a significant reduction in deterministic time (approximately 15 %) and nodes (approximately 25 %), consistent across seeds. As for stability, results are mixed: if we consider `all`, the standard deviation of the deterministic times is indeed reduced considerably, by more than a factor of two. However, this does not carry over to the `opt` subset. This confirms that reducing variability *during* enumeration is still an open issue.

5 Tuning for performance: CPLEX implementation

The `ksample` algorithm illustrated in the previous sections has been successfully implemented within the default of the latest IBM ILOG CPLEX 12.5.1 version [13] for the case $K = 2$. Precisely, after solving the initial LP relaxation, two different cut loops are concurrently applied in a parallel fashion (if enough threads are available), possibly rooted at two different LP bases (both optimal for the initial LP relaxation). Each cut loop is performed by enforcing some random diversification in the solution process, in order to explore different regions of the solution space, to collect more cutting planes and (possibly) better primal solutions. Along the process, the two cut loops are synchronized on a deterministic basis by sharing feasible solutions and cutting planes. At the end of the root node, a final synchronization step is applied to collect and filter the solutions and the cuts generated by the two cut loops, and then the subsequent branch and cut is started.

The performance impact of the implementation highlighted above is reported in Sect. 5.1. It is worth noting that an attempt to implement the `ksample` idea with $K > 2$ concurrent cut loops has been made. However, the computational experiments conducted for the cases $K > 2$ have shown a performance degradation over the case $K = 2$. On the one hand, the addition of any concurrent cut loop introduces some unavoidable overhead in the whole solution process, and the overhead clearly increases with K. On the other side, the key advantage of `ksample` is to enforce random diversification, with the aim of producing more information (i.e., feasible solutions and cutting planes). However, such a positive effect quickly saturates as K increases. As a matter of fact, for the specific case of CPLEX and for the specific way we implemented the `ksample` idea on it, the benefit obtained by applying more than two concurrent cut loops was not worth the additional overhead introduced in the whole root node.

5.1 Computational results

The testbed used in this section consists of 3221 problem instances coming from a mix of publicly available and commercial sources. A time limit of 10,000 s was used for all the tests. Additionally, we employed a tree memory limit of 6 GB. If this tree memory limit was hit, we treated the model as if a time limit was hit, by setting the solve time to 10,000 s and scaling the number of processed nodes accordingly. All tests in this section were conducted by running IBM ILOG CPLEX 12.5.1 on a cluster of identical 12 core Intel Xeon CPU E5430 machines running at 2.66 GHz and being equipped with 24 GB of memory.

Tables 3, 4, 5, 6 and 7 report the performance impact of our `ksample` algorithm embedded within IBM ILOG CPLEX 12.5.1 for $K = 2$. The tables compare case $K = 2$ where two different cut loops are concurrently run in a parallel fashion (this method is denoted `2-sample` in the following) against case $K = 1$ where only the standard cut loop is run (this method is named `no-sample` in the following).

Table 3 Performance impact of parallel cut loop in IBM ILOG CPLEX 12.5.1 (seed 1)

Class	All models				Affected		
	#models	#tilim	Time	Nodes	#models	Time	Nodes
All	3157	84/82	0.99	0.96	1312	0.98	0.92
[0,10k]	3086	13/11	0.99	0.96	1312	0.98	0.92
[1,10k]	1870	13/11	0.98	0.97	1090	0.97	0.94
[10,10k]	1092	13/11	0.97	0.96	678	0.96	0.94
[100,10k]	559	13/11	0.95	0.92	367	0.92	0.88
[1k,10k]	219	13/11	0.90	0.86	154	0.86	0.81
[0,10k)	3062	0/0	1.00	0.97	1288	0.99	0.93
[1,10k)	1846	0/0	0.99	0.98	1066	0.99	0.96
[10,10k)	1068	0/0	0.99	0.98	654	0.98	0.97
[100,10k)	535	0/0	0.98	0.95	343	0.97	0.93
[1k,10k)	195	0/0	0.97	0.94	130	0.96	0.91

Table 4 Performance impact of parallel cut loop in IBM ILOG CPLEX 12.5.1 (seed 2)

Class	All models				Affected		
	#models	#tilim	Time	Nodes	#models	Time	Nodes
All	3164	86/82	0.99	0.97	1345	0.97	0.94
[0,10k]	3094	16/12	0.99	0.97	1345	0.97	0.94
[1,10k]	1875	16/12	0.98	0.96	1106	0.96	0.93
[10,10k]	1099	16/12	0.96	0.94	696	0.94	0.90
[100,10k]	575	16/12	0.94	0.92	377	0.91	0.88
[1k,10k]	223	16/12	0.93	0.90	154	0.91	0.85
[0,10k)	3066	0/0	0.99	0.98	1317	0.97	0.94
[1,10k)	1847	0/0	0.98	0.96	1078	0.97	0.94
[10,10k)	1071	0/0	0.97	0.95	668	0.95	0.92
[100,10k)	547	0/0	0.95	0.94	349	0.93	0.90
[1k,10k)	195	0/0	0.96	0.94	126	0.94	0.92

Table 5 Performance impact of parallel cut loop in IBM ILOG CPLEX 12.5.1 (seed 3)

Class	All models				Affected		
	#models	#tilim	Time	Nodes	#models	Time	Nodes
All	3159	81/78	0.97	0.93	1340	0.92	0.85
[0,10k]	3091	13/10	0.97	0.93	1340	0.92	0.85
[1,10k]	1864	13/10	0.95	0.90	1098	0.91	0.84
[10,10k]	1089	13/10	0.92	0.87	689	0.88	0.80
[100,10k]	565	13/10	0.86	0.81	370	0.80	0.73
[1k,10k]	215	13/10	0.85	0.80	146	0.79	0.72
[0,10k)	3068	0/0	0.97	0.94	1317	0.94	0.87
[1,10k)	1841	0/0	0.96	0.92	1075	0.93	0.86
[10,10k)	1066	0/0	0.94	0.89	666	0.90	0.83
[100,10k)	542	0/0	0.89	0.85	347	0.84	0.77
[1k,10k)	192	0/0	0.93	0.90	123	0.90	0.85

All the tables have the same structure, each giving aggregated results over the whole testbed obtained with a different random seed. For each seed, the instances are divided in different subsets, based on the *hardness of the models*. To avoid any bias in the analysis, the level of difficulty is defined by taking into account the two solvers that are compared, namely, `no-sample` and `2-sample`. First, the set "all" is defined by keeping all the models but the ones for which one of the solvers encountered a failure of some sort or where numerical difficulties led to different optimal objective values for the two solvers (both values being correct due to feasibility tolerances). Then, the set "all" is divided in subclasses "[n, 10k]" ($n = 1, 10, 100, 1\text{k}$), containing the subset of "all" models for which at least one of the solvers took at least n seconds

Table 6 Performance impact of parallel cut loop in IBM ILOG CPLEX 12.5.1 (seed 4)

Class	all models				Affected		
	#models	#tilim	Time	Nodes	#models	Time	Nodes
All	3162	89/95	0.99	0.97	1323	0.98	0.93
[0,10k]	3084	11/17	0.99	0.97	1323	0.98	0.93
[1,10k]	1856	11/17	0.99	0.96	1094	0.97	0.93
[10,10k]	1089	11/17	0.98	0.95	686	0.97	0.92
[100,10k]	557	11/17	0.97	0.93	369	0.95	0.89
[1k,10k]	217	11/17	0.96	0.87	146	0.94	0.82
[0,10k)	3056	0/0	0.99	0.97	1295	0.98	0.93
[1,10k)	1828	0/0	0.99	0.96	1066	0.98	0.94
[10,10k)	1061	0/0	0.98	0.96	658	0.97	0.93
[100,10k)	529	0/0	0.97	0.95	341	0.96	0.92
[1k,10k)	189	0/0	0.97	0.92	118	0.95	0.88

Table 7 Performance impact of parallel cut loop in IBM ILOG CPLEX 12.5.1 (seed 5)

Class	All models				Affected		
	#models	#tilim	Time	Nodes	#models	Time	Nodes
all	3161	94/97	0.99	0.97	1339	0.97	0.94
[0,10k]	3077	10/13	0.99	0.97	1339	0.97	0.94
[1,10k]	1856	10/13	0.98	0.96	1112	0.97	0.93
[10,10k]	1077	10/13	0.98	0.95	681	0.96	0.93
[100,10k]	569	10/13	0.98	0.95	377	0.96	0.92
[1k,10k]	203	10/13	0.95	0.92	137	0.93	0.89
[0,10k)	3054	0/0	0.99	0.97	1316	0.98	0.94
[1,10k)	1833	0/0	0.99	0.96	1089	0.98	0.94
[10,10k)	1054	0/0	0.98	0.96	658	0.97	0.94
[100,10k)	546	0/0	0.99	0.96	354	0.98	0.95
[1k,10k)	180	0/0	0.99	0.98	114	0.98	0.96

to solve and that were solved to optimality within the time limit by at least one of the solvers. Finally, the subclasses "[n, 10k)" ($n = 1, 10, 100, 1\text{k}$) contain all models in "[n, 10k]" but considering only the models that were solved to optimality by both the compared solvers (i.e., models for which any of the solvers hit a time or memory limit are disregarded).

Table 3 has the following structure: the first column, "class", identifies the group of models. The first group of four columns (i.e., columns from 2 to 5) under the heading "all models" reports for each class of models, the results on all models contained in the class. Column "#models" reports the number of problem instances in each class. Note that only 3157 out of the 3221 problem instances are listed for the class "all" due to the exclusion rules explained above. Column “#tilim” gives the number of models for which a time (or memory) limit was hit by the two solvers `no-sample` and `2-sample`, respectively. For instance, the values 84/82 for the class “all” indicate that the number of time limit hits was overall 84 and 82 for the two solvers `no-sample` and `2-sample`, respectively, while the values 13/11 for all classes “[n, 10k]” indicate that `no-sample` hit the time limit for 13 models that were solved to optimality by `2-sample`, while `2-sample` hit the time limit for 11 models that were solved to optimality by `no-sample`. Observe that it is not accidental that there are no time limits in the last 5 rows, because they are disregarded on purpose for the classes “[n, 10k)”. Column “time” displays the shifted geometric mean of the ratios of solution times between `2-sample` and `no-sample` with a shift of $s = 1$ s. A value $t < 1$ in the table indicates that `2-sample` is by a factor of $1/t$ faster (in shifted geometric mean) than `no-sample`. Note that time limit hits are accounted with a value of 10,000 s for classes “[n, 10k]”, which it is likely to introduce an unavoidable bias against the solver with fewer timeouts. Column “nodes” is similar to the previous column but shows the shifted geometric mean of the ratios of the number of branch-and-cut nodes needed for the problems by each solver, using a shift of $s = 10$ nodes. When a time limit is hit, we use the number of nodes at that point, which again introduces a bias. Finally, the second group of three columns (i.e., columns from 6 to 9) under the heading “affected”, repeat the same information for the subset of models in each class where the two compared solvers took a different *solution path*. For the sake of simplicity, we assume that the solution path is identical if both the number of nodes and the number of simplex iterations are identical for the two solvers. Models that are solved to optimality by only one of the compared solvers are always included in the set of “affected” models, while models for which both solvers hit the time (or memory) limit are never included in this set, regardless the number of nodes and simplex iterations. For this reason, we did not report the column “#tilim” for the affected models, because the numbers of time limit hits are the same as for the set containing all models of each class.

As already stated, Tables 4, 5, 6 and 7 have the same structure as Table 3, but it is worth noting that the size of the subclass of models, as well as the models in each class, are different for each table, because the level of difficulty of a given model may change with the random seed.

The results reported in Tables 3, 4, 5, 6 and 7 clearly show that the addition of a parallel cut loop at the root node of the branch-and-cut algorithm yields a performance improvement that is consistent across the 5 random seeds considered, both in terms of computing time and number of branch-and-bound nodes. Moreover, the results on the different classes of problems indicate that the harder the models, the larger the performance improvement achieved, and even this information is consistent across the 5 seeds. Finally, a closer look at the results of classes “[100,10k]” and “[1k,10k]” also shows the impact of performance variability. Indeed, the improvement on those subclasses, although consistent, may vary quite substantially by changing the random seed. Precisely, the improvement on class “[100,10k]” varies from $1/0.98 = 1.02\times$ with seed 5 to $1/0.86 = 1.16\times$ with seed 3, while the improvement on class “[1k,10k]” varies from $1/0.96 = 1.04\times$ with seed 4 to $1/0.85 = 1.18\times$ with seed 3. Those quite large differences are not surprising, because (*i*) the number of models in those subclasses is pretty small, if compared with the number of models in the class "all", and thus these classes of models are less robust to outliers, and (*ii*) the hardest models are typically those exhibiting the largest performance variability.

6 Conclusions

High sensitivity to initial conditions is a characteristic of MIP solvers, that leads to a possibly very large performance variability of multiple runs performed when starting from slightly-changed initial conditions. A primary source of variability comes from dual degeneracy affecting the root node LPs, producing a large number of alternative choices for the initial (optimal) LP basis to be used to feed the cutting plane/primal heuristic/branching loops.

In this paper we have studied the above source of variability and proposed a new sampling scheme that solves—through parallel independent runs—the root node with diversified initial conditions (random seeds) and uses all the collected primal and dual information to feed the final complete run to the same MIP solver. By simply adjusting the number of simultaneous samples, we can either emphasize stabilization, i.e., devise an algorithm that stabilizes the MIP solver by reducing its sensitivity to initial conditions, or exploit variability to gain on performance, namely significantly reducing computing times.

More precisely, computational results on the *primal* and *benchmark* sets of the MIPLIB 2010 testbed clearly show the stabilization effect of our algorithm when a large number of samples is taken. Namely, the algorithm

1. produces significantly improved primal solutions at the root node;
2. reduces the root-node variability in terms of integrality gap;
3. consistently reduces the average computational effort spent in the final run.

In addition, on the internal CPLEX testbed composed of 3221 problem instances, a version of our algorithm implemented within the commercial solver IBM ILOG CPLEX 12.5.1 which uses only two samples obtains a consistent reduction in both computing times and number of nodes that appears to be significant on non trivial models requiring at least 10 s to be solved. These impressive results led to the inclusion of the algorithm as default in the release of the commercial solver.

Many questions concerning the stabilization of the entire framework remain open, the main one being the stabilization of the branching tree.

Acknowledgments This research was partially supported by MiUR, Italy (PRIN project "Mixed-Integer Nonlinear Optimization: Approaches and Applications") and by the University of Padova (Progetto di Ateneo "Exploiting randomness in Mixed-Integer Linear Programming"). Thanks are due to two anonymous referees for their helpful comments; we also thank Hans D. Mittelmann for the use of the cluster at Arizona State University.

Appendix: detailed results

See Table 8.

Table 8 Cross-solver performance variability

instance	vsTime			vsNodes		
	CPLEX	GUROBI	XPRESS	CPLEX	GUROBI	XPRESS
30_70_45_095_100	0.11	0.10	0.51	0.00	0.00	1.55
30n20b8	0.71	0.57	0.91	0.48	0.46	1.07
acc-tight4	0.39	0.29	0.50	0.72	0.45	0.59
acc-tight5	0.88	0.75	0.92	0.87	0.89	0.97
acc-tight6	0.82	0.54	0.74	0.96	0.72	0.80
aflow40b	0.23	0.30	0.32	0.26	0.27	0.31
air04	0.57	0.19	0.12	0.62	0.45	0.38
app1-2	0.68	0.51	0.65	0.34	0.89	0.70
ash608gpia-3col	0.77	0.43	0.30	3.16	1.19	2.11
beasleyC3	0.12	0.41	1.41	0.31	0.62	1.43
biella1	0.25	0.34	0.54	0.25	0.29	0.52
bienst2	0.07	0.11	0.16	0.05	0.14	0.19
binkar10_1	0.34	0.76	0.38	0.34	1.00	0.41
bley_xl1	0.17	0.06	0.08	0.00	0.00	0.00
core2536-691	0.35	0.51	2.89	0.34	0.66	3.11
cov1075	0.76	0.16	0.87	0.82	0.20	0.74
csched010	0.48	0.33	1.02	0.86	0.38	0.97
danoint	0.17	0.26	0.12	0.19	0.20	0.11
dfn-gwin-UUM	0.15	0.16	0.11	0.17	0.13	0.11
eil33-2	0.14	0.35	0.25	0.22	0.50	0.24
eilB101	0.34	0.27	0.22	0.26	0.49	0.22
ex10	0.03	0.07	0.44	0.00	0.00	1.33
ex9	0.03	0.03	0.73	0.00	0.00	0.00
gmu-35-40	3.16	3.16	3.16	3.16	3.16	3.16
iis-100-0-cov	0.29	0.13	0.04	0.23	0.15	0.05
iis-bupa-cov	0.25	0.13	0.05	0.20	0.13	0.05
iis-pima-cov	0.53	0.42	0.45	0.60	0.50	0.50
lectsched-2	0.13	0.38	0.56	0.41	0.00	2.07
lectsched-3	1.25	1.15	1.47	0.98	1.53	1.35
lectsched-4-obj	0.42	0.16	1.61	0.48	1.05	1.71
map18	0.12	0.07	0.23	0.16	0.12	0.20
map20	0.17	0.22	0.25	0.26	0.16	0.35
mcsched	0.64	0.11	0.48	0.72	0.17	0.39
mik-250-1-100-1	0.08	1.02	0.20	0.09	1.06	0.21
mine-166-5	0.22	0.20	0.10	0.27	0.28	0.22
mine-90-10	0.50	0.50	0.47	0.44	0.60	0.44
mspp16	0.37	0.07	0.93	2.05	0.00	1.48
mzzv11	0.30	0.24	0.34	0.56	0.94	0.60
n4-3	0.34	0.37	0.28	0.41	0.67	0.29

Table 8 continued

instance	vsTime			vsNodes		
	CPLEX	GUROBI	XPRESS	CPLEX	GUROBI	XPRESS
neos-1109824	0.45	0.39	0.29	0.42	0.54	0.48
neos-1171692	0.35	0.14	0.11	3.16	0.00	0.47
neos-1224597	0.32	0.23	0.33	0.00	0.00	0.00
neos-1396125	0.55	0.92	1.37	0.64	0.83	1.05
neos-1440225	0.61	0.82	1.63	0.39	0.94	1.78
neos-1601936	0.83	1.82	3.01	1.09	2.48	2.95
neos-476283	0.24	0.15	0.21	0.38	0.38	0.46
neos-506422	0.56	0.38	0.65	0.80	0.60	0.71
neos-686190	0.19	0.34	0.29	0.22	0.41	0.31
neos-738098	0.33	0.26	0.38	1.95	0.00	2.35
neos-777800	0.32	0.34	0.28	0.00	0.00	0.00
neos-824661	0.29	0.18	0.18	0.00	0.00	0.00
neos-824695	0.69	0.21	0.56	2.42	0.00	0.00
neos-826694	0.53	0.43	0.27	0.00	0.00	0.00
neos-826812	0.14	0.12	0.38	0.00	0.00	0.00
neos-849702	1.28	1.75	1.57	1.31	1.75	1.53
neos-885086	0.65	0.39	2.10	2.04	0.00	2.10
neos-885524	0.75	1.96	0.51	2.11	2.65	0.86
neos-932816	0.18	0.06	0.33	0.00	0.00	0.00
neos-933638	0.68	0.26	0.59	1.24	0.00	1.16
neos-933966	0.07	0.29	0.16	0.00	0.00	0.46
neos-934278	0.28	0.28	0.38	0.73	0.00	0.57
neos-935627	0.61	0.48	1.61	1.20	1.28	1.83
neos-935769	0.26	0.38	0.38	0.78	0.00	0.81
neos-937511	0.23	0.18	0.15	0.85	0.00	0.41
neos-941313	0.36	0.16	0.51	0.96	0.00	1.84
neos-957389	0.03	0.26	0.03	0.00	0.00	0.00
neos13	0.29	0.32	0.07	0.46	0.45	0.76
neos18	0.28	0.38	0.56	0.28	0.42	0.49
neos6	0.61	0.24	1.47	0.89	0.00	1.47
neos808444	0.41	0.22	0.20	1.17	0.00	0.32
net12	1.31	0.55	0.49	0.78	0.42	0.44
netdiversion	0.78	1.49	0.93	1.07	1.04	1.18
noswot	1.45	0.52	1.02	1.21	0.53	1.18
ns1116954	1.46	1.75	1.93	1.28	2.61	1.88
ns1688347	0.52	0.32	0.43	0.88	0.11	0.40
ns1758913	0.95	0.35	0.23	0.00	0.00	0.00
ns1766074	0.02	0.05	0.04	0.01	0.03	0.05
ns1830653	0.15	0.40	0.80	0.23	0.56	0.83

Table 8 continued

instance	vsTime			vsNodes		
	CPLEX	GUROBI	XPRESS	CPLEX	GUROBI	XPRESS
ns1952667	1.18	1.68	1.34	1.55	2.00	1.29
opm2-z7-s2	0.15	0.32	0.21	0.15	0.20	0.24
pg5_34	0.13	0.10	1.52	0.12	0.09	1.45
pigeon-10	0.13	0.19	0.26	0.06	0.06	0.21
qiu	0.38	0.23	0.38	0.46	0.29	0.54
rail507	0.62	0.32	0.96	0.53	0.58	0.73
ran16x16	0.11	0.24	0.21	0.11	0.38	0.23
reblock67	0.12	0.29	0.24	0.13	0.38	0.29
rmatr100-p10	0.06	0.36	0.47	0.06	0.28	0.47
rmatr100-p5	0.22	0.29	0.38	0.23	0.29	0.40
rmine6	0.37	0.39	0.19	0.22	0.39	0.19
rocII-4-11	0.73	0.44	0.57	0.38	0.50	0.60
rococoC10-001000	0.21	0.53	0.44	0.20	0.53	0.40
roll3000	0.54	0.86	1.53	0.84	1.42	1.47
satellites1-25	0.40	0.13	1.36	0.62	0.05	1.21
sp98ic	0.26	0.35	0.46	0.26	0.43	0.47
sp98ir	0.18	0.19	0.26	0.20	0.27	0.29
tanglegram1	0.05	0.08	0.55	0.28	0.40	0.44
tanglegram2	0.07	0.08	0.29	0.45	0.84	0.84
timtab1	0.38	0.85	0.67	0.35	0.89	0.70
triptim1	0.68	0.16	0.21	0.00	0.00	0.99
unitcal_7	0.23	0.23	0.18	0.33	0.15	0.97
zib54-UUE	0.31	0.24	0.23	0.35	0.22	0.24
Average	0.44	0.44	0.63	0.60	0.49	0.76

References

1. Achterberg, T.: Constraint integer programming. Ph.D. thesis, ZIB (2007)
2. Achterberg, T.: LP basis selection and cutting planes (2010). MIP 2010 workshop in Atlanta
3. Achterberg, T., Wunderling, R.: Mixed integer programming: Analyzing 12 years of progress. In: Facets of Combinatorial Optimization, pp. 449–481 (2013)
4. Berthold, T.: Measuring the impact of primal heuristics. Oper. Res. Lett. **41**(6), 611–614 (2013)
5. Carvajal, R., Ahmed, S., Nemhauser, G., Furman, K., Goel, V., Shao, Y.: Using diversification, communication and parallelism to solve mixed-integer linear programs. Oper. Res. Lett. **42**(2), 186–189 (2014)
6. Cornuéjols, G.: Valid inequalities for mixed integer linear programs. Math. Progr. **112**, 3–44 (2008)
7. Danna, E.: Performance variability in mixed integer programming (2008). MIP 2008 workshop in New Work. http://coral.ie.lehigh.edu/~jeff/mip-2008/talks/danna.pdf
8. FICO: FICO XPRESS Optimization Suite (2013). http://www.fico.com
9. Fischetti, M., Glover, F., Lodi, A.: The feasibility pump. Math. Progr. **104**(1), 91–104 (2005)
10. Fischetti, M., Monaci, M.: Exploiting erraticism in search. Oper. Res. **62**(1), 114–122 (2014)

11. Gomes, C.P., Sellmann, B., Kautz, H.: Boosting combinatorial search through randomization. In: Proceedings of the National Conference on Artificial Intelligence, pp. 431–437. AAAI Press (1998)
12. GUROBI: GUROBI Optimizer (2013). http://www.gurobi.com
13. IBM: IBM ILOG CPLEX Optimization Studio (2013). http://www.cplex.com
14. Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R.E., Danna, E., Gamrath, G., Gleixner, A.M., Heinz, S., Lodi, A., Mittelmann, H., Ralphs, T., Salvagnin, D., Steffy, D.E., Wolter, K.: MIPLIB 2010—Mixed Integer Programming Library version 5. Math. Progr. Comput. **3**, 103–163 (2011)
15. Linderoth, J.T., Lodi, A.: MILP software. In: Cochran, J.J. (ed.) Wiley Encyclopedia of Operations Research and Management Science, vol. 5, pp. 3239–3248. Wiley, Chichester (2011)
16. Lodi, A.: Mixed integer programming computation. In: Jünger, M., Liebling, T.M., Naddef, D., Nemhauser, G.L., Pulleyblank, W.R., Reinelt, G., Rinaldi, G., Wolsey, L.A. (eds.) 50 Years of Integer Programming 1958–2008, pp. 619–645. Springer, Heidelberg (2009)
17. Lodi, A.: The heuristic (dark) side of MIP solvers. In: Talbi, E.G. (ed.) Hybrid Metaheuristics, pp. 273–284. Springer, Heidelberg (2012)
18. Lodi, A., Tramontani, A.: Performance variability in mixed-integer programming. In: Topaloglu, H. (ed.) TutORials in Operations Research: Theory Driven by Influential Applications, pp. 1–12. INFORMS, Catonsville, MD (2013)
19. Mitra, G., Hai, I., Hajian, M.T.: A distributed processing algorithm for solving integer programs using a cluster of workstations. Parallel Comput. **23**(6), 733–753 (1997)
20. Nemhauser, G.L., Wolsey, L.A.: Integer and Combinatorial Optimization. Wiley, New York (1988)
21. Shinano, Y., Achterberg, T., Berthold, T., Heinz, S., Koch, T., Winkler, M.: Solving hard MIPLIB2003 problems with ParaSCIP on supercomputers: An update. Tech. Rep. 13-66, ZIB (2013)