Noname manuscript No.
(will be inserted by the editor)

Tree Search Stabilization by Random Sampling

Matteo Fischetti · Andrea Lodi · Michele Monaci · Domenico Salvagnin · Andrea Tramontani

Submitted: September 2013

Abstract We discuss the variability in the performance of multiple runs of Mixed Integer Linear 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

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

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

Matteo Fischetti
DEI, University of Padova, Italy, E-mail: matteo.fischetti@unipd.it

Andrea Lodi
DEI, University of Bologna, Italy, E-mail: andrea.lodi@unibo.it

Michele Monaci
DEI, University of Padova, Italy, E-mail: michele.monaci@unipd.it

Domenico Salvagnin
DEI, University of Padova, Italy, E-mail: domenico.salvagnin@unipd.it

Andrea Tramontani
IBM S.p.A., Italy, E-mail: andrea.tramontani@it.ibm.com

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 within a general *branch-and-cut* scheme (see, e.g., [13] and [12] for an overview of the computational framework and the MIP software, respectively).

Performance variability. In a highly influential talk in 2008, Danna [5] 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 2,731 Simplex iterations on a Linux platform, while it needed 1,426 nodes and 122,948 iterations on an AIX one.

That severe and unexpected variability in performance was later explained in [11] 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 like the mentioned order of the candidates (or variables) are performed. (The reader is referred to [15] 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., [4]), 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 at 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., [8], 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 [7], 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. Finally, in [3] multiple threads are used to exploit variability by running a different parameters' configuration in each of them and allowing various degrees of communication.

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 side, 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 [11] 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 to other (crucial) operations performed by a MIP solver. 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 on 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 Section 5 we describe the implementation of the algorithm within the commercial MIP solver CPLEX and the improved performance that are achieved. Finally, in Section 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 10 different row/columns permutations and tested the 3 major commercial solvers, namely IBM ILOG CPLEX 12.5.0 [10], GUROBI 5.1 [9] and XPRESS 23 [6]. 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 [11] of computing time and branch-and-cut nodes, which is defined as the ratio between the standard deviation and the arithmetic of the

n performance measures obtained with n different permutations of the same instance.

Scatter plots of the variability scores for all pairwise comparison between solvers can be found in Figures 1 and 2, whereas detailed results are shown in Table 9 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., [14] for a extensive discussion on the topic), row aggregation in cutting plane generation just to mention one, makes the characterization of the *best* 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. All collected cuts and solutions are then put together in a (hopefully) stabilized and enriched root node from which

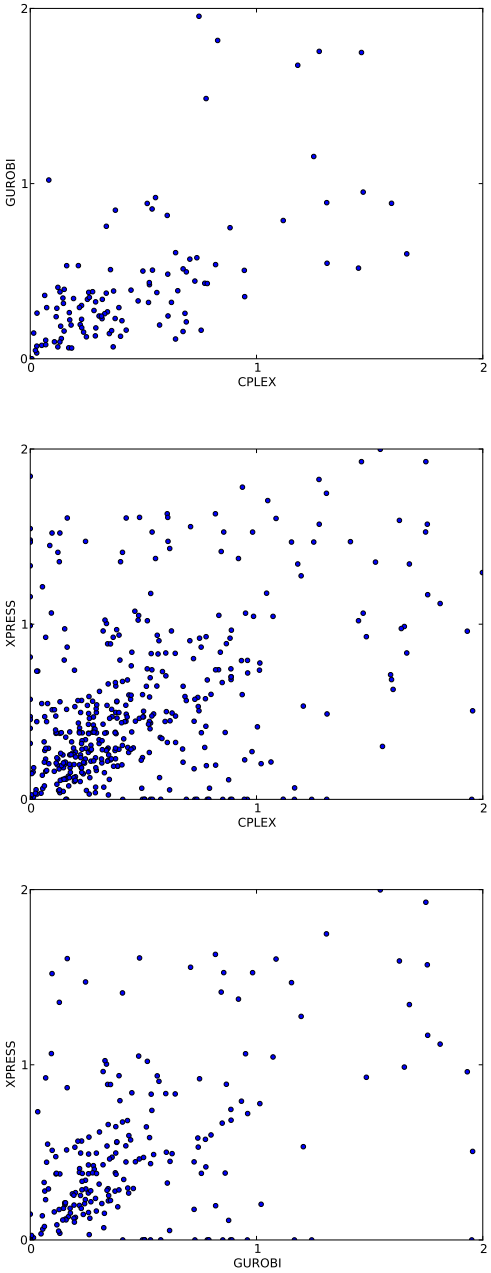

Fig. 1 Time Variability Comparison

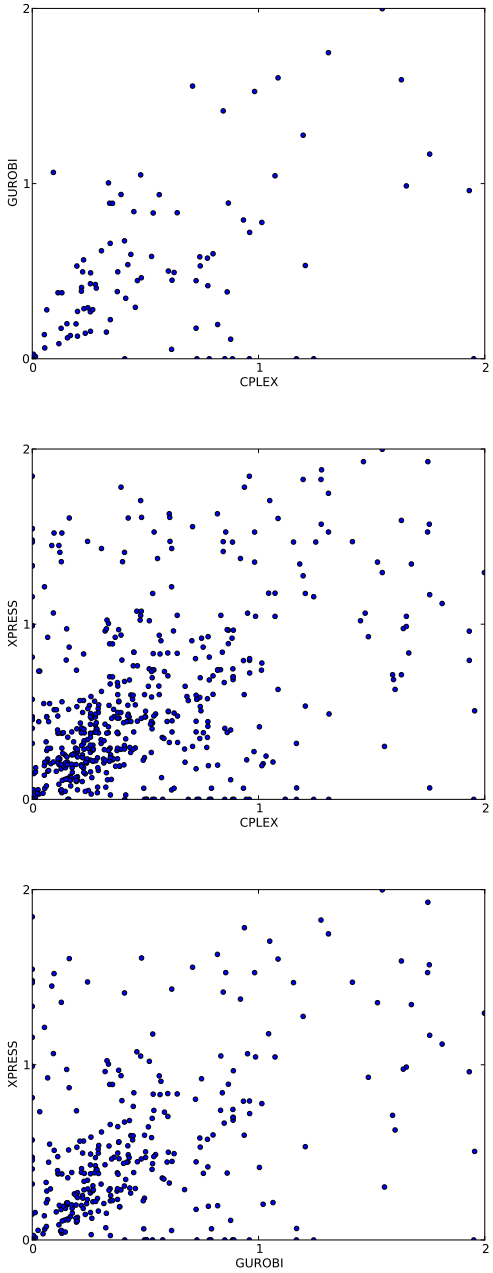

Fig. 2 Node Variability Comparison

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

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 of sampled bases K. 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 Section 5 the performance emphasis is given.

4 Tuning for Computational Stability

We implemented our codes in C++, using IBM ILOG CPLEX 12.5.0 [10] 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 a Intel i7-2600 CPU running at 3.40GHz and 16GB 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 running times but rather *deterministic ticks*, which are an abstract unit 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 tick limit of 10^7, which corresponds approximately to a real time limit of 3 hours on our machines. As far as performance measures are concerned, we always report shifted geometric means (see, e.g., [1]), with a shift of 10^4 for deterministic ticks and of 100 for node numbers.

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 [7] 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, and that CPLEX uses in some of its internal operations to deal with LP degeneracy. Changing the random seed can be seen as a way to enforce a random diversification in the root node and hence in the whole branch-and-cut search.

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 different random seeds, stopping each run at the end of the root node, and collecting all the cutting planes generated in these K root nodes and the primal feasible solutions (if any).

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 copy 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 $K > 0$. It is easy to see that we can "simulate" a traditional CPLEX run just by setting $K = 0$ (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

In order to measure the performance variability at the end of the root node, we ran `cpxdef` and `ksample` with 10 different random seeds on all the instances in our testbed and recorded the final primal and dual bounds, indicated with $\overline{z}$ and $\underline{z}$, respectively. Given those values, we compute the integrality gap `igap` as

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

This is the integrality gap as reported by many commercial solvers, and has the advantage of always being a number between 0 and 1. In addition, a similar formula was used in [2] 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 10 runs and the corresponding standard deviation.

In order to provide a fair comparison between the methods, we removed from the testbed the instances that were solved by `ksample` during the sampling phase (23 instances), instance `glass4` for numerical issues (different optimum objectives were reported with different random seeds), and the 3 infeasible instances `ash608gpia-3col`, `ns1766074` and `enlight14` because they have no integrality gap. Average results on the remaining testbed of 91 instances are reported in Table 1.

Table 1 Root node comparison on the whole testbed

	igap		pgap		dgap	
instance	avg	st.dev	avg	st.dev	avg	st.dev
`cpxdef`	58.8%	3.9%	52.4%	4.5%	17.8%	0.2%
`ksample`	43.5%	3.1%	34.4%	3.9%	17.3%	0.1%

Based on the results in Table 1, not only the average integrality gap is significantly reduced in `ksample`, but also stability is improved.

4.3 Branch-and-cut Stability

According to the previous section, `ksample` proves to be effective in improving performance and stability at the root node. In the current section, we evaluate if such improvements carry over to the overall solution process. Note that the results 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 ticks and nodes. Note that the deterministic ticks reported for `ksample` are related to the final branch and cut only. As already mentioned, 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 do not report the deterministic ticks spent in the sampling phase. Obviously, however, even a parallel implementation of the sampling phase would introduce some overhead in the method. Table 2 reports these aggregated results on the whole testbed for six different random seeds (the default one plus additional five). To guarantee a fair comparison, we discarded the 23 instances that `ksample` could solve during sampling, and instance `glass4`, again for numerical reasons. Thus, we are left with 94 instances.

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

seed	method	solved	ticks	nodes
0	`cpxdef`	81	379,186	17,967
	`ksample`	83	327,901	15,538
1	`cpxdef`	83	364,170	16,996
	`ksample`	82	286,434	13,735
2	`cpxdef`	81	356,798	17,316
	`ksample`	82	275,634	12,668
3	`cpxdef`	84	325,351	15,663
	`ksample`	83	326,020	16,015
4	`cpxdef`	81	377,064	18,126
	`ksample`	83	290,286	14,077
5	`cpxdef`	82	417,147	21,018
	`ksample`	84	268,603	13,139

According to Table 2, starting from a "better" root node can indeed improve the performance of the subsequent branch-and-cut run, at least on average. Indeed, `ksample` consistently improves upon `cpxdef`, with a significant reduction in ticks and nodes in 5 out of 6 cases, while performing the same in the other case (seed 3).

For the sake of completeness, Table 3 reports average results (again, for 6 different random seeds) for `cpxdef` on the 23 instances solved by `ksample` during the sampling phase. The stabilization by sampling has the positive side effect of acting as an effective primal heuristic, especially for instances where the challenge is finding a feasible solution.

Table 3 Performance of `cpxdef` on the instances solved by `ksample` during sampling

instance	ticks	nodes
30_70_45_095_100	3,894	1
neos-1171692	1,428	1
neos-1224597	131	1
neos-738098	4,725	2
neos-777800	1,796	1
neos-824661	3,720	1
neos-824695	2,164	1
neos-826694	3,373	1
neos-826812	1,060	1
neos-885086	19,188	19
neos-885524	2,688	15
neos-932816	892	1
neos-933638	33,458	6
neos-933966	5,605	1
neos-935769	92,432	21
neos-937511	92,720	24
neos-957389	693	1
neos6	20,170	341
neos808444	7,399	1
netdiversion	22,167	1
ns1116954	169,268	7
ns1758913	112,866	2
triptim1	60,893	1

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 [10] 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 Section 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 worthy the additional overhead introduced in the whole root node.

5.1 Computational Results

The testbed used in this section consists of 3,221 problem instances coming from a mix of publicly available and commercial sources. A time limit of 10,000 seconds 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 seconds 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 4–8 report the performance impact of our `ksample` algorithm embedded within IBM ILOG CPLEX 12.5.1 for $K = 0$ and $K = 2$. The tables compare case $K = 0$ where only the standard cut loop is run (this method is named `no-sample` in the tables) against case $K = 2$ where two different cut loops are concurrently run in a parallel fashion (`2-sample` in the tables).

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, `2-sample` and `no-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 to solve and that were solved to optimality within the time limit by at least one of the solvers.

Table 4 has the following structure: the first column, "class", identifies the group of models. Column "#models" reports the number of problem instances in each class. Note that only 3,157 out of the 3,221 problem instances are listed for the class "all" due to the exclusion rules explained above. Columns "#tilim" give the number of models in each class for which a time (or memory) limit was hit by the two solvers. Observe that it is not accidental that the values in the last 5 rows match, because these 13 time-limit models are always included in the corresponding subsets. Column "time" displays the shifted geometric mean of the ratios of solution times (see, e.g., [1]) with a shift of

$s = 1$ second. 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 seconds, which introduces 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 last three columns, under the heading "affected", repeat some of the 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.

As already stated, Tables 5–8 have the same structure as Table 4, 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.

Table 4 Performance impact of parallel cut loop in IBM ILOG CPLEX 12.5.1 (seed 1)

		`no-sample`	`2-sample`			affected		
class	#models	#tilim	#tilim	time	nodes	#models	time	nodes
all	3,157	84	82	0.99	0.96	1,312	0.98	0.92
[0,10k]	3,086	13	11	0.99	0.96	1,312	0.98	0.92
[1,10k]	1,870	13	11	0.98	0.97	1,090	0.97	0.94
[10,10k]	1,092	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

Table 5 Performance impact of parallel cut loop in IBM ILOG CPLEX 12.5.1 (seed 2)

		`no-sample`	`2-sample`			affected		
class	#models	#tilim	#tilim	time	nodes	#models	time	nodes
all	3,164	86	82	0.99	0.97	1,345	0.97	0.94
[0,10k]	3,094	16	12	0.99	0.97	1,345	0.97	0.94
[1,10k]	1,875	16	12	0.98	0.96	1,106	0.96	0.93
[10,10k]	1,099	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

The results reported in Tables 4–8 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

Table 6 Performance impact of parallel cut loop in IBM ILOG CPLEX 12.5.1 (seed 3)

		no-sample	2-sample			affected		
class	#models	#tilim	#tilim	time	nodes	#models	time	nodes
all	3,159	81	78	0.97	0.93	1,340	0.92	0.85
[0,10k]	3,091	13	10	0.97	0.93	1,340	0.92	0.85
[1,10k]	1,864	13	10	0.95	0.90	1,098	0.91	0.84
[10,10k]	1,089	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

Table 7 Performance impact of parallel cut loop in IBM ILOG CPLEX 12.5.1 (seed 4)

		no-sample	2-sample			affected		
class	#models	#tilim	#tilim	time	nodes	#models	time	nodes
all	3,162	89	95	0.99	0.97	1,323	0.98	0.93
[0,10k]	3,084	11	17	0.99	0.97	1,323	0.98	0.93
[1,10k]	1,856	11	17	0.99	0.96	1,094	0.97	0.93
[10,10k]	1,089	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

Table 8 Performance impact of parallel cut loop in IBM ILOG CPLEX 12.5.1 (seed 5)

		no-sample	2-sample			affected		
class	#models	#tilim	#tilim	time	nodes	#models	time	nodes
all	3,161	94	97	0.99	0.97	1,339	0.97	0.94
[0,10k]	3,077	10	13	0.99	0.97	1,339	0.97	0.94
[1,10k]	1,856	10	13	0.98	0.96	1,112	0.97	0.93
[10,10k]	1,077	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

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 on 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., devising 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. strongly reduces the root-node variability both in terms of primal and dual gap;
3. for some instances, significantly reduces the computational effort spent in the final run;
4. solves to optimality within the time limit more instances than the MIP solver alone.

In addition, on the internal CPLEX testbed composed by 3,221 problem instances, a version of our algorithm implemented within the commercial solver IBM ILOG CPLEX 12.5.1 by using only two samples obtains a significant reduction in both computing times and number of nodes. 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.

Acknowledgements We thank Hans D. Mittelmann for the use of the cluster at Arizona State University.

References

1. Achterberg, T.: Constraint integer programming. Ph.D. thesis, ZIB (2007)

2. Berthold, T.: Measuring the impact of primal heuristics. Tech. rep., ZIB (2013)
3. Carvajal, R., Ahmed, S., Nemhauser, G., Furman, K., Goel, V., Shao, Y.: Using diversification, communication and parallelism to solve mixed-integer linear programs. Tech. rep., Optimization Online (2013). URL http://www.optimization-online.org/DB_HTML/2013/05/3900.html
4. Cornuéjols, G.: Valid inequalities for mixed integer linear programs. Mathematical Programming **112**, 3–44 (2008)
5. Danna, E.: Performance variability in mixed integer programming. Presentation at Workshop on Mixed Integer Programming (2008)
6. FICO: FICO XPRESS Optimization Suite (2013). http://www.fico.com
7. Fischetti, M., Monaci, M.: Exploiting erraticism in search. Tech. rep., University of Padova (2012)
8. 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)
9. GUROBI: GUROBI Optimizer (2013). http://www.gurobi.com
10. IBM: IBM ILOG CPLEX Optimization Studio (2013). http://www.cplex.com
11. 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. Mathematical Programming Computation **3**, 103–163 (2011)
12. Linderoth, J.T., Lodi, A.: MILP software. In: J.J. Cochran (ed.) Wiley Encyclopedia of Operations Research and Management Science, vol. 5, pp. 3239–3248. Wiley (2011)
13. Lodi, A.: Mixed integer programming computation. In: M. Jünger, T.M. Liebling, D. Naddef, G.L. Nemhauser, W.R. Pulleyblank, G. Reinelt, G. Rinaldi, L.A. Wolsey (eds.) 50 Years of Integer Programming 1958-2008, pp. 619–645. Springer (2009)
14. Lodi, A.: The heuristic (dark) side of MIP solvers. In: E.G. Talbi (ed.) Hybrid Metaheuristics, pp. 273–284. Springer (2012)
15. Lodi, A., Tramontani, A.: Performance variability in mixed-integer programming. In: H. Topaloglu (ed.) TutORials in Operations Research: Theory Driven by Influential Applications, pp. 1–12. INFORMS, Catonsville, MD (2013)

Appendix: detailed results

Table 9: Cross-solver performance variability.

	vsTime			vsNodes		
instance	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

Table 9 continued

	vsTime			vsNodes		
instances	CPLEX	GUROBI	XPRESS	CPLEX	GUROBI	XPRESS
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
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

Table 9 continued

	vsTime			vsNodes		
instances	CPLEX	GUROBI	XPRESS	CPLEX	GUROBI	XPRESS
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
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

Table 9 continued

	vsTime			vsNodes		
instances	CPLEX	GUROBI	XPRESS	CPLEX	GUROBI	XPRESS
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