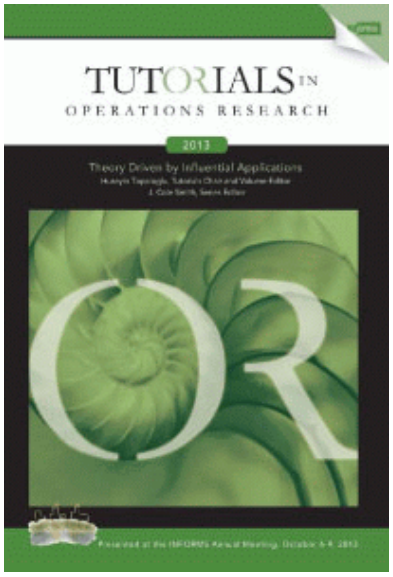

INFORMS TutORials in Operations Research

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

Performance Variability in Mixed-Integer Programming

Andrea Lodi, Andrea Tramontani

To cite this entry: Andrea Lodi, Andrea Tramontani. Performance Variability in Mixed-Integer Programming. *In* INFORMS TutORials in Operations Research. Published online: 14 Oct 2014; 1-12.
https://doi.org/10.1287/educ.2013.0112

TUTORIALS IN
OPERATIONS RESEARCH
INFORMS 2013

 | ISBN 978-0-9843378-4-2
http://dx.doi.org/10.1287/educ.2013.0112

Performance Variability in Mixed-Integer Programming

Andrea Lodi
Department of Electrial, Electronic and Information Engineering, University of Bologna, Bologna, Italy, andrea.lodi@unibo.it

Andrea Tramontani
CPLEX Optimization, IBM, Bologna, Italy, andrea.tramontani@it.ibm.com

Abstract The performance of mixed-integer programming solvers is subject to some unexpected variability that appears, for example, when changing from one computing platform to another, when permuting rows and/or columns of a model, when adding seemingly neutral changes to the solution process, etc. This phenomenon has been observed for decades, but only recently has it started to be methodologically analyzed with the two possible aims of either reducing or exploiting it, ideally both. In this tutorial we discuss the roots of performance variability, we provide useful tips to recognize it, and we point out some severe misinterpretations that might be generated by not performing/analyzing benchmark results carefully. Finally, we report on the most recent attempts to gain from variability.

Keywords mixed-integer programming; computation; branch and bound; benchmarking; erraticism; floating-point precision

1. Introduction

A general *mixed-integer programming* (*MIP*) problem can be given in the form

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

where no special structure for matrix A is assumed. Problem (1) is thus approached by iteratively solving, through general-purpose techniques, its *linear programming* (*LP*) relaxation

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

i.e., the same as (1), but with the integrality requirement dropped on the x variables in the set I. Of course, the reason for forgetting the integrality constraints is that the MIP problem is $\mathcal{NP}$-hard, whereas LP is polynomially solvable with general-purpose algorithms that are also efficient in practice. Loosely speaking, using the LP computation as a tool, MIP solvers integrate the *branch-and-bound* and the *cutting plane* algorithms through variations of the general *branch-and-cut* scheme proposed by Padberg and Rinaldi [23] in the context of the famous traveling salesman problem.

Although we try to keep the tutorial rather untechnical, throughout it we assume the reader to be familiar with the basic components of MIP machinery and MIP software for which we refer to Achterberg [1], Lodi [20], Linderoth and Lodi [18], and Achterberg and Wunderling [2].

The performance of MIP solvers is subject to some unexpected variability that appears, for example, when changing from one computing platform to another, when permuting rows

and/or columns of a model, etc. A first, instructive example reported in the MIP literature by Danna [6] concerns the well-known sport-scheduling instance `10teams` (Achterberg et al. [3]):

- The instance is solved by the MIP solver IBM ILOG [16] CPLEX (sometimes referred to as CPLEX for short) version 11 on a Linux platform at the root node (conventionally, 0 branch-and-bound nodes) and 2,731 simplex iterations, whereas
- the same version of the software running on an AIX platform (precisely the same code, compiled and run on a different computer architecture) requires 1,426 nodes and 122,948 simplex iterations.

This phenomenon has certainly been observed for decades and has been the topic of an extensive literature in the artificial intelligence and SATisfiability communities; see, e.g., Gomes et al. [12, 13, 14]. However, to the best of our knowledge, in the MIP context it has only been reported explicitly rather recently (indeed, Danna [6]), and it is now referred to as *performance variability*. Since then, some work has been devoted to gaining a deeper understanding of performance variability and to point out its implications in the ways the benchmarking of MIP solvers and, in general, of optimization codes is performed.

Indeed, the most dangerous effect of performance variability is the misinterpretation of the computational results obtained by testing a scientific idea or even a change in the code that might appear harmless. Precisely in the attempt of showing potential mistakes associated with performance variability, yet another very instructive example is discussed in the provocative talk by Fischetti and Monaci [8]:

(1) A set of 38 "difficult" instances is selected by running IBM ILOG CPLEX version 12.2 on two standard benchmark libraries (Achterberg et al. [3], COR@L [5]). The set contains the instances requiring more than 10,000 nodes and 100 central processing unit (CPU) seconds.

(2) A single trivially valid and redundant linear inequality $\sum_{j=1}^{n} x_{\pi^k(j)} \geq -1$ is added to the model associated with a random permutation π^k of the nonnegative variables $x_1, \ldots, x_n$, thus producing nine virtually identical copies $k = 1, \ldots, 9$ of each initial instance.

(3) IBM ILOG CPLEX is then executed on the instances obtaining a geometric time ratio with respect to the performance on the initial version of the instance of

$$[0.75, 0.79, 0.88, 0.92, 1.09, 0.80, 0.86, 0.84, 0.79].$$

In other words, in eight out of nine cases the addition of a redundant constraint determines a nontrivial speed up in the computation.

As discussed in Fischetti and Monaci [8], the behavior above has two technical explanations. On the one side, the addition of a redundant constraint to the instance changes the order in which the variables are loaded by the MIP solver, thus determining implicitly a permutation of its columns.[1] This is a classical situation in which performance variability appears because the order in which the variables are loaded and then processed makes the solution process change (see the next section). On the other hand, one should still expect the performance of the solver on the perturbed model to be sometimes better and sometimes worse than that on the original model. In fact, because of the way the instances have been selected (see point (1) above), the test bed has been heavily biased to favor problems in which the solver does not perform well; thus virtually any change in the solution process (due to the column permutation) is a winner.

Of course, the addition of a redundant constraint to an MIP model does not represent a new scientific idea, but it simply adds noise (variability) to the solution process. Nevertheless, the example shows that the computational testing must be very carefully conducted so as

[1] Note that this permutation effect is obtained on purpose by (1) writing/reading the model in "LP" format, (2) adding the redundant constraint as the first constraint in the file, and (3) replacing the objective function by a single variable, which is, in turn, defined at the end of the file.

to distinguish between the scientific impact of the idea and the noise it creates. Because in most of the cases a neat cut between them is likely to be impossible, the awareness of the phenomenon is crucial to avoid misinterpretations of the observed results.

The remainder of this paper is organized as follows. In the next section the roots of variability are discussed, and the common measures used in the literature are introduced. In §3 we discuss the impact of variability in the benchmarking of MIP solvers, whereas in §4 we survey some methods that use the variability to gain performance. Finally, some conclusions are drawn in §5.

2. The Roots of Performance Variability

Again to the best of our knowledge, the first paper that formally discusses performance variability in the MIP context is the one introducing the latest release of the Mixed Integer Problem Library (MIPLIB 2010; Koch et al. [17]). There, *one* source of performance variability is rooted in the so-called *imperfect tie breaking*. Most of the decisions taken by an MIP solver are based on ordering candidates according to scores and selecting the candidate with the *best* score. This is true for cut separation and cut filtering as well as for one of the most crucial decisions of the MIP solution process, namely, the variable to branch on at each node. It is virtually impossible to find a score suitable to fully distinguish among the candidates, and although the scores are designed for taking into account multiple characteristics that should allow to break ties, the process is structurally imperfect.

This explanation is certainly satisfying for the example in the previous section in which the variables of the same model are permuted so as to create copies of it: because the variables are processed in a different order, the easiest and most common tiebreaker that selects the *first* candidate among those with the same score will most probably return different choices, e.g., in branching, on the permuted copies of the model, thus leading to the algorithm evolving differently.

Indeed, and needless to be stressed, as soon as the *path changes* within the search tree, it is fair to expect that the whole evolution can be completely different with rather severe consequences.

However, imperfect tie breaking is not the end of the story, or at least the reasons and consequences of it have to be analyzed carefully. Note that, in principle, the same explanation would not seem fully satisfactory on the first example of the previous section in which the same model is solved by the same code on different computing platforms. In fact, some additional ties or, conversely, tiebreaks can be induced by floating-point operations (see, e.g., Goldberg [11]), which are intrinsically different in different computing environments, and by compiler optimizations (see Koch et al. [17] for details). Then, the behavior of IBM ILOG CPLEX on instance `10teams` becomes much more understandable.

Diving toward the root of the "candidate/score" game, and even deeper at the level of the entire MIP machinery, one needs to observe that the single components of the MIP solvers are essentially taking decisions heuristically. The heuristic nature of the decisions of MIP solvers is discussed in detail in Lodi [21], and two main issues can be distinguished.

• On the one side, the *perfect* score that uniquely determines the *best* decision might not exist because of the lack of theoretical knowledge. The most noticeable example of that is the "best" basis within the optimal face of the initial[2] LP relaxation. We are not even close to define theoretically what "best" means in this case, and it is likely we will never be, in the sense that it surely depends on what the basis is used for, namely, cutting plane separation, primal heuristics, branching, etc. The consequence is that the simplex algorithm just converges to a (random) basis depending on the floating-point arithmetics

[2] Of course, the same applies to the optimal face of *any* LP relaxation, but, because of what we have already discussed, the earlier the choice diverges, the worse in terms of variability.

of the computing environment or on the order in which the variables have been loaded. The same applies if the LP is solved by an interior point algorithm that, once reached the (unique) center of the optimal face, performs a (random) crossover to one of its vertices.

• On the other hand, even if such a perfect score would be known, it might be too expensive to compute. In a certain sense, it is always known but only a posteriori, i.e., after full enumeration of all choices, which is, in turn, the essence of MIP $\mathcal{NP}$-hardness. But most of the scores, even those that are much less ambitious and much more local, are actually too expensive, too. This is the case of the so-called *strong branching* (see, e.g., Linderoth and Savelsbergh [19]) that essentially simulates one level of branching for each fractional variable to select the best variable to branch on at each node. The only versions of strong branching affordable for MIP solvers are way less computationally heavy by (i) restricting on a subset of fractional variables, (ii) not solving the LPs to optimality, and (iii) acting only on a subset of nodes.

Of course, one could argue that to avoid variability it is not necessary to know the perfect score, but simply to pick one without tie breaks. However, a similar score might be too expensive or too complex to compute as well because it would probably require random tie-breaking in a machine-independent manner. Indeed, because of the wide (and unpredictable) range of difficulty of MIP instances, no general-purpose MIP solver can afford correcting variability (if possible) at the price of an overall decrease in performance. In fact, variability, or at least the awareness of it, can be "used" either for a more informed benchmarking (see the next section) or to gain some performance (see §4).

The above discussion leads to consider performance variability intrinsic to MIP and largely independent of the solvers, although it is conceivable that some solvers pay more attention to performance robustness than others. This conclusion is confirmed by the performance variability reported in Koch et al. [17] for the noncommercial MIP solver SCIP [24], and that of the two commercial MIP solvers GUROBI [15] and IBM ILOG CPLEX [16] as discussed in Mittlemann's [22] benchmark page. Interestingly, the analysis conducted in Koch et al. [17] and the computational results reported in Mittlemann [22] are different not only in the way in which performance variability is simulated but also in the way it is measured.

• In Koch et al. [17], performance variability is measured as a single number, denoted as a *variability score* (VS), associated with a given MIP model and computed by using a fixed solver and a fixed platform. Precisely, given

  ◦ a random variable X representing the performance on a given instance I, measured as the computing time to solve I to proven optimality, and

  ◦ N different observations of X on I,

the performance variability of I is simply defined/computed as the relative standard deviation of X:

$$\mathrm{VS}(I) := \sqrt{\frac{\sum_{j=1}^{N}(X_j - \langle X\rangle)^2}{N\langle X\rangle^2}}, \tag{3}$$

where

$$\langle X\rangle = \frac{\sum_{j=1}^{N} X_j}{N}$$

is the arithmetic mean of the N observations of X.

To produce different observations of X, each model was solved $N = 100$ times after applying a random permutation of columns and rows. The computational results, obtained with the MIP solver SCIP [24] on a subset of 67 models from the MIPLIB 2010 library, indicated that all the considered models are somehow "affected" by performance variability, as there was no instance for which all N permutations showed the same behavior. Just looking at the ratio of the maximal and the minimal solution times among the N runs, the minimum (resp., maximum) ratio observed was 1.29 (resp., 915.0).

• In Mittlemann [22], the same picture is shown in a different way, by running GUROBI [15] and IBM ILOG CPLEX [16], again on the MIPLIB 2010 test bed. GUROBI 5.1.0 and CPLEX 12.5.0 now offer a *random seed* parameter that can be used to change the initialization of the random number generator that is used in some internal operations, precisely with the purpose of breaking ties and speeding up the computation. As discussed in Danna [6], 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 impact on the path taken by the solver, and thus it can be seen as a natural way to mimic the imperfect tie breaking induced by floating-point operations that can be observed by running the same code on different computing platforms. The natural question of how much a given MIP solver S is affected by performance variability is answered in Mittlemann [22] in the following way. Each model is run with a default random seed, and then with other $N = 5$ different random seeds (i.e., with seeds 1001–1005). The solution time to optimality obtained with the default seed, namely, $S(D)$, is then compared against the best computing time among the N nondefault random seeds, namely, $S(N)$.[3] Note that there is no relation between $S(D)$ and $S(N)$, because all the N seeds used to obtain $S(N)$ are different than the default one, so one could have $S(D) < S(N)$. The following table, taken from Mittlemann [22] on March 15, 2013, reports the aggregated results on the 87 instances of the "benchmark" test bed of the MIPLIB 2010 library (Koch et al. [17]), for the two commercial MIP sovers GUROBI 5.1.0 (GUROBI [15]) and CPLEX 12.5.0 (IBM ILOG [16]):

GUROBI(D)	CPLEX(D)	GUROBI(N)	CPLEX(N)
43.7	50.62	30.14	29.89

The numbers reported in the table are geometric means of the computing times obtained on an Intel Xeon X5680 machine (32 GB, Linux, 64 bits, 2×6 cores), where each solver was run with 12 threads. In Mittlemann [22], several noncommercial and commercial solvers are compared. However, only GUROBI and CPLEX are tested with multiple random seeds. In any case, it is worth mentioning that the results reported in this paper are not intended to induce any performance comparison, but just to assess that performance variability should be definitely considered as an issue intrinsic to MIP and largely independent of the solvers.

Far from being able to produce any meaningful comparison between the two solvers, the above picture leads to the following observations:

—The two solvers exhibit roughly the same performance variability. This leads again to the conclusion that performance variability is intrinsic to MIP, or at least it is intrinsic to the branch-and-cut paradigm used to tackle a general-purpose MIP model, whereas it appears to be largely independent of the specific solvers.

—The improvement of $S(N)$ with respect to $S(D)$ is definitely not negligible. Thus, a lucky predictor that is always able to identify the best one among the different N runs would certainly allow to obtain a nontrivial performance improvement on the default run. Possible ways to (try to) exploit variability to gain some performance are discussed in §4.

3. Benchmarking and Variability

When conducting a performance comparison among different MIP solvers, or when evaluating the computational impact of adding a new feature on top of an existing solver, a first fundamental step is to define a correct benchmarking methodology. As highlighted in

[3] It is worth mentioning that the multiple-run experiments from Mittlemann [22] follow a very similar experiment conducted by Fischetti and Monaci [8, 9]. In Fischetti and Monaci [9], only CPLEX 12.3.0 is used and performance variability is simulated by running the MIP solver with up to 100 different settings of the parameters. The results reported in Fischetti and Monaci [9] provide the same indications as the ones reported in Mittlemann [22].

the previous section, the severe performance variability that can be observed on individual models makes MIP performance analysis a challenging task. Indeed, any change applied to an existing code introduces a random noise in the solver that can affect the solution process in a rather unpredictable way. Therefore, the computational testing must be very carefully conducted so as to distinguish between the real impact of the new idea that needs to be evaluated and the noise it creates. This section provides some suggestions to define a correct benchmarking methodology and to avoid some common mistakes that can introduce a bias in the performance evaluation. A deeper analysis of this topic is conducted by Achterberg and Wunderling [2].

A first important step is the selection and the size of the test bed. If the purpose of the analysis is to evaluate the impact of a scientific idea on a specific class of models (e.g., a class of MIP models originated from a combinatorial optimization problem that share a given known structure exploited by the idea itself), then it is obvious that the experiments will be conducted on an homogeneous set of instances. However, a computational evaluation of ideas for general-purpose MIP solvers should be conducted on a heterogenous test bed: virtually all the types of models that the solver will be used for should be equally represented, and an overrepresentation of one or few types of models should be avoided. The size of the test bed is important as well, to limit the side effects of performance variability: the smaller the performance difference that should be analyzed, the larger the test bed needed. Furthermore, when comparing aggregated results, geometric means should be used instead of arithmetic means to limit the impact of outliers on the evaluation.

The test bed used in this section consists of problem instances coming from a mix of publicly available and commercial sources. We excluded all problem instances from our collection that we have never been able to solve to optimality with any version between CPLEX 6.0 and CPLEX 12.5.0. This left a total of 3,189 instances. To keep the computation time under control, 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. All tests were conducted by running CPLEX 12.5.0 (with a default or nondefault parameter setting) on a cluster of identical 12 core Intel Xeon CPU E5430 machines running at 2.66 GHz and equipped with 24 GB of memory.

3.1. A Common Bias

A common approach that is used in MIP performance analysis is to divide the test bed in different subsets, based on the *hardness of the models*. This is, in principle, very useful to gain a better understanding of the computational results. However, it is of crucial importance to pay attention to the way in which the *hardness* of a given model is defined. Precisely, the level of difficulty must be defined by taking into account all the solvers that are to be compared. Instead, a typical mistake that is often made is to define the level of difficulty only based on the reference solver (i.e., the competitor that we want to use to evaluate the impact of a new feature).

We illustrate the effect of this mistake in the following example, based on a similar experiment that was conducted in Achterberg and Wunderling [2]. Suppose that we want to measure the impact of disabling one of the features of CPLEX 12.5.0, namely, the zero-half cuts separator. Then, our reference solver is CPLEX default, whereas the solver to test against the reference is CPLEX with the zero-half cuts separator disabled.

On the one hand, a correct way to divide the test bed into subclasses could be as follows. First, we identify a set of "all" models by excluding only the models 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 (which could actually be, due to the definition of feasibility tolerances, both correct). Then, we divide this set "all" in subclasses "[n, 10k]" ($n = 1, 10, 100, 1$k), containing the subset of "all" models for which at least one of the solvers

being compared took at least n seconds to solve and that were solved to optimality within the time limit by at least one of the solvers.

On the other side, a wrong way to divide the test bed into subclasses (that is often used in practice) is to divide the models exactly in the same classes "[n, 10k]," but looking only at the solution time of the reference solver (i.e., CPLEX default) to identify which class the models belong to.

The results obtained with the correct and the biased model grouping are reported in Tables 1 and 2, respectively. The tables have both the following structure: The first column, "Class," identifies the group of the models. The second column, "#models," reports the number of problem instances in each class. Note that only 3,136 out of the 3,189 problem instances are listed for the class "all" due to the exclusion rules explained above. The third column, "Default #tilim," gives the number of models in each subset for which a time (or memory) limit was hit by the reference solver (i.e., CPLEX default). It is by design that the numbers in the last five rows match, since these 17 time-limit models are always included in the corresponding subsets. Similarly, the fourth column gives the number of models in each subset for which the solver to evaluate (i.e., no zero-half cuts) hits a time limit. The fifth column, "#wins," and sixth column, "#losses," show the number of problem instances in each subset that the "no zero-half cuts" setting solved faster and slower than the default setting, respectively. A model is considered a win (resp., a loss) if CPLEX without zero-half cuts was at least 10% faster (resp., slower) than the reference solver CPLEX default. The seventh column, "Time," displays the shifted geometric mean of the ratios of solution times (see Achterberg [1]), with a shift of $s = 1$ second. A value $t > 1$ in the table indicates that CPLEX without zero-half cuts is by a factor of t slower (in shifted geometric mean) than CPLEX default. 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. Thus, the performance ratios need to be considered in conjunction with the number of timeouts of each solver. Finally, the last two columns, under the heading "Affected," repeat some of the information for the subset of models in each class where the two compared solvers took different solution paths. For the sake of simplicity, we assume that the solution paths are

Table 1. Impact of disabling zero-half cuts in CPLEX 12.5 (correct model grouping).

		Default	No zero-half cuts				Affected	
Class	#models	#tilim	#tilim	#wins	#losses	Time	#models	Time
All	3,136	88	106	395	370	1.03	1,406	1.07
[0, 10k]	3,065	17	35	395	370	1.03	1,406	1.07
[1, 10k]	1,875	17	35	376	359	1.05	1,096	1.09
[10, 10k]	1,103	17	35	272	274	1.08	715	1.13
[100, 10k]	598	17	35	168	184	1.13	422	1.19
[1k, 10k]	243	17	35	69	85	1.23	188	1.31

Table 2. Impact of disabling zero-half cuts in CPLEX 12.5 (biased model grouping).

		Default	No zero-half cuts				Affected	
Class	#models	#tilim	#tilim	#wins	#losses	Time	#models	Time
All	3,136	88	106	395	370	1.03	1,406	1.07
[0, 10k]	3,065	17	35	395	370	1.03	1,406	1.07
[1, 10k]	1,846	17	34	376	333	1.03	1,068	1.06
[10, 10k]	1,072	17	34	272	243	1.03	684	1.05
[100, 10k]	549	17	33	168	135	0.97	373	0.95
[1k, 10k]	207	17	25	69	50	0.82	153	0.76

identical if both the number of nodes and the number of simplex iterations are identical for the two solvers.

Table 1 (the correct one) clearly shows that disabling zero-half cuts introduce a performance degradation that goes from 3% on the class "all" up to 23% on the class "[1k, 10k]": the harder the models, the larger the degradation. This is confirmed also by the larger number of time limit hits. However, it is worth noting that disabling zero-half cuts produces overall more wins than losses (i.e., 395 wins versus 370 losses on the class "all"), and that the number of wins is not negligible even on the class "[1k, 10k]" (69 wins versus 85 losses). This could be precisely due to performance variability, as any change on the solver can introduce unpredictable side effects that turn some hard models, where the default solver was just "unlucky," to easy ones. The large number of models with a different solution path between the two solvers (i.e., 1,406 path changes out of 3,136 models) could be a symptom of performance variability as well.

On the contrary, Table 2 (the biased one) could lead to a misinterpretation of the results. Indeed, disregarding the different number of time limit hits, one could argue that disabling zero-half cuts introduces a 3% performance degradation overall but improves on the hard models (3% improvement on "[100, 10k]" and 18% improvement on "[1k, 10k]"), even if we know from the unbiased analysis that the opposite is actually the case. This misinterpretation of the results is again related to variability: the biased subset selection puts problem instances for which the reference solver was unlucky into the "harder" subsets, giving the other solver a good chance of doing better on those.

3.2. Dealing with Performance Variability

As highlighted in the previous section, variability can somehow hurt performance analysis even when evaluating a feature like the zero-half cut separator, which has a strong (beneficial) impact on the solver. The issue becomes much harder to deal with when one has instead to analyze small performance differences, that could be difficult to distinguish from the noise created by the code change that needs to be evaluated.

Table 3 reports a comparison of CPLEX 12.5 using two different random seeds. The results show once more that performance variability matters. A seemingly neutral change to the code produces a huge number of path changes in the solution process (namely, 2,264 models with different paths out of 3,121 in the subset "all," and 235 out of 236 in "[1k, 10k]"), and also a large number of wins/losses. Disregarding the number of time limit hits by the two seeds (and then applying a bias against Seed 1) and looking at the computing time, Seed 2 produces a 1% performance degradation on nontrivial models in the subset "[1, 10k]" and a 8% degradation on the hardest models in the subset "[1k, 10k]." In other words, performance differences in CPLEX of about 1% cannot be distinguished using our test set of 3,000 problem instances, and the issue gets worse as the difficulty of the models increases. Indeed, the performance change of 8% in "[1k, 10k]" indicates that any new feature added to the solver that produces a change of about 10% in that class of models could be just

Table 3. Comparison of CPLEX 12.5 using two different random seeds (correct model grouping).

		Seed 1	Seed 2				Affected	
Class	#models	#tilim	#tilim	#wins	#losses	Time	#models	Time
All	3,121	87	105	581	588	1.00	2,264	1.01
[0, 10k]	3,051	17	35	581	588	1.00	2,264	1.01
[1, 10k]	1,864	17	35	558	558	1.01	1,693	1.01
[10, 10k]	1,110	17	35	398	396	1.01	1,056	1.01
[100, 10k]	596	17	35	238	237	1.02	585	1.02
[1k, 10k]	236	17	35	101	100	1.08	235	1.08

random noise. This noticeable difference between "[1k, 10k]" and the other subsets of models has two possible explanations: (i) class "[1k, 10k]" contains only 236 models, and thus it is less robust to outliers, and (ii) harder models are typically the ones exhibiting the larger variability, so that one solver can solve them easily by luck, whereas the other solver hits the time limit.

Given the picture above, when one needs to analyze small performance differences of the same order of what can be obtained with a random perturbation, several other tools should be used. First, it is useful to increase the number of observations by running the two solvers under comparison with multiple random seeds. Second, descriptive statistics such as the average or the geometric mean give limited insights, whereas robust indicators such as truncated averages and rank statistics are instead more resistant to outliers. Finally, inferential statistics, such as statistical tests and confidence intervals, give the most insights and allow us to answer questions such as, which is the probability that the observed performance change is just created by variability rather than by genuine algorithmic changes? (see, e.g., Conover [4]).

4. Exploiting Variability

The ability to simulate variability, especially by random seeds, discussed in the previous sections helps in designing computational experiments in which the intrinsic noise is taken into account, thus leading to more robust decisions while testing new ideas. To some extent, then, the awareness about variability led to an improvement in the best practices in development cycles of MIP solvers.

Nevertheless, as shown by the numbers in Mittlemann's [22] benchmark page, the luckiest solver over five random seeds would guarantee an improvement of 40% on average with respect to the default one both for IBM ILOG CPLEX and GUROBI. So, the question about the possibility of exploiting variability not only at the benchmarking level but also in terms of new algorithmic features becomes quite natural.

In this concern, it is easy to note that an "all-win" strategy is to remove sources of variability by designing better candidate/score games. More precisely, each time in the algorithm a decision is taken with a weak policy, it is very likely that tie breaking will also be weak because too much influenced by factors like the computing platform or the order of the variables. In other words, a "better" algorithm generally leads to better performance and to a smaller variability. An example in this context is given by Danna and Lodi [7] that, starting from version 12, improved both the performance and the variability of IBM ILOG CPLEX through a better *pseudocost* formula, essentially the score that is responsible for the selection of the branching variable (see, e.g., Achterberg [1]).

- For each variable x_j, a score, indeed called the *pseudocost*, is computed by taking into account the (past) effect in the bound of branching on that particular variable.
- The improvement obtained by Danna and Lodi [7] comes from the introduction in the pseudocost formula of a term taking into account the infeasible nodes encountered during search by branching on x_j.
- That term is set to have a high relative importance with respect to that of the feasible nodes before the first feasible solution has been found, whereas it goes to zero after.

This modification had a slight improving effect (2%) on an internal IBM ILOG CPLEX test bed (more than 2,000 MIP instances) and performed especially well for (i) satisfiability models (without objective function), with a 3× speedup and a 36% reduction in the variability computed according to (3), and (ii) infeasible models or models that are hard for finding a first feasible solution, with a 30% speedup and roughly the same variability reduction factor. Moreover, the idea improved, in general, on the time to the first feasible solution, being roughly 1.5 times faster.

Although the above work was not motivated by an attempt to exploit variability, it contributes to reducing it because it improves the largely imperfect branching score on the variables, thus making, as a by-product, the tie breaking less random. Two very recent methods work instead directly with the idea of exploiting the already discussed presence of multiple optimal bases within the optimal face[4] of the initial LP relaxation of an MIP. Both methods are based on the recognition of the potentially high variability associated with the selection of such an initial basis, and, in the light of (or maybe we should say, in the "darkness" derived by) the lack of knowledge of which basis is better, they take two different paths.

• With the algorithm called *bet-and-run*, Fischetti and Monaci [9] apply a look-ahead strategy. Precisely, after the first LP relaxation is solved, some alternative (equivalent) optimal solutions are obtained by random pivoting on the optimal face. This is obtained by (1) fixing all nonbasic variables with nonzero reduced cost (including slack variables) to their current value, (2) installing a random objective function, and (3) running the simplex algorithm with a limit on the number of pivots. Because of what we have discussed so far, each of those optimal solutions corresponds to a different version of the original instance, at least in terms of the evolution of the branch-and-bound run that originated from it. Then, the MIP solver is applied for at most N branch-and-bound nodes[5] to each of them, and, unless one of these runs ends by proving optimality, in which case the algorithm stops, the most promising of the alternatives is selected and continued till the end. In other words, because of the lack of understanding of which is the best basis, Fischetti and Monaci [9] decide to try some of them for a limited time and a posteriori evaluate which is the best alternative. Of course, the *bet-and-run* algorithm just postpones the decision, but the rationale is that more pieces of information can be collected within the limited-time runs (eight different indicators are used in Fischetti and Monaci [9]), and the decision is then more informed. The results of a proof-of-concept implementation are very promising.

• In the algorithm called *ksample*, Fischetti et al. [10] and Tramontani et al. [25] again look at alternative optimal bases but with the idea of collecting and using the information associated with some of them, namely, K. Although one could obtain different bases as in Fischetti and Monaci [9], the *ksample* algorithm makes use of the random seed parameter of IBM ILOG CPLEX (see the previous sections) to randomize the process and get different starting points. To each of them, the algorithm applies the full machinery of the root node of an MIP solver, which includes cutting plane generation and primal heuristics. Valid cuts and feasible solutions of the K root node runs[6] are collected and put in pools, which are then used by a final run that starts from scratch with yet another (different) random seed. The rationale of the method is that sampling the optimal face, one can collect more global information (in the form of cuts and solutions) with respect to simply picking (randomly) one basis. Indeed, the root node of the final run amended and enhanced by those pieces of information is more "stable." Precisely, using IBM ILOG CPLEX 12.4.0, 10 runs with different random seeds of CPLEX are compared with 10 runs of *ksample*, that is, CPLEX, but with the cutting planes and the feasible solutions collected in the sampling phase discussed above stored in the CPLEX user cut and solution pools. The results on the MIPLIB 2010 instances (Koch et al. [17]) show that the performance variability of *ksample* is significantly smaller than the one of IBM ILOG CPLEX over the 10 runs. In addition, *ksample* produces root nodes with less variability in the percentage gap and some performance gain on the branch-and-cut is achieved.

[4] Of course, it is conceivable that the optimal vertex is unique and not even primal degenerate (a unique basis would then exist), but in practice this happens very infrequently.

[5] Rather surprisingly, Fischetti and Monaci [9] set for the experiments $N = 5$ and computationally show that such a very small number of nodes is generally enough to appropriately guess the quality of a run.

[6] Parameter K is set to 10 in the computational experiments conducted in Fischetti et al. [10].

Both methods above suffer from a computational overhead associated with the initial need of collecting information: N truncated tentative runs for *bet-and-run* and K, potentially parallel, root nodes for *ksample*. The situation is more severe for *ksample*, and in the experiments in Fischetti et al. [10], this overhead is not included while evaluating the performance gain because the emphasis is put on the stabilization effect of the algorithm. Versions of both *bet-and-run* and *ksample* more strictly integrated within an MIP solver should overcome some of this overhead, and, indeed, a simpler version of *ksample* will be included in a forthcoming release of IBM ILOG CPLEX [16].

5. Conclusions

We have shown that the variability in performance that is observed while running mixed-integer programming solvers is intrinsic to the MIP technology and largely independent of the solvers. The examples presented show that the variability introduces noise in the computational evaluation of new ideas and algorithms, making their assessment quite difficult. We have discussed some sources of variability and suggested methods to simulate it artificially. In this way, one can be prepared to its impact so as to mitigate its effect. Finally, we have surveyed some recent methods that, motivated by the impressive improvement one could get from lucky decisions (as pointed out by the variability experiments with different random seeds on noncommercial and commercial solvers), try to exploit variability to gain performance.

We believe that there is a lot of work to be done in the area: insights to gain for a better understanding and for limiting variability, new ideas for exploiting it, and more accurate testing mechanisms and best practices for a full awareness of the MIP computation.

Acknowledgments

The authors are indebted with Tobias Achterberg, Emilie Danna, Matteo Fischetti, Ed Klotz, Michele Monaci, and Domenico Salvagnin for endless and stimulating discussions about this topic. The first author acknowledges the support by MUIR Italy [Grant PRIN2009].

References

[1] T. Achterberg. Constraint integer programming. Ph.D. thesis, Technische Universität Berlin, Berlin, 2007.

[2] T. Achterberg and R. Wunderling. Mixed integer programming: Analyzing 12 years of progress. M. Jünger, G. Reinelt, eds. *Facets of Combinatorial Optimization*. Springer, Berlin, 449–481, 2013.

[3] T. Achterberg, T. Koch, and A. Martin. MIPLIB 2003. *Operations Research Letters* 34:361–372, 2006.

[4] W. J. Conover. *Practical Nonparametric Statistics*. John Wiley & Sons, New York, 1999.

[5] COR@L. Computational Optimization Research at Lehigh. Accessed July 21, 2013, http://coral.ie.lehigh.edu/data-sets/mixed-integer-instances/.

[6] E. Danna. Performance variability in mixed integer programming. Presentation, Workshop on Mixed Interger Programming (MIP 2008), Columbia University, New York. Accessed July 21, 2013, http://coral.ie.lehigh.edu/~jeff/mip-2008/talks/danna.pdf, 2008.

[7] E. Danna and A. Lodi. Using infeasible nodes to select branching variables. Presentation, International Symposium on Mathematical Programming (ISMP 2009), Chicago. Accessed July 21, 2013, http://www.or.deis.unibo.it/andrea/pscost-ISMP2009.pdf, 2009.

[8] M. Fischetti and M. Monaci. On the role of randomness in exact tree search methods. Presentation, Matheuristics 2012, Angra dos Reis, Rio de Janeiro, Brazil. Accessed July 21, 2013, http://mat.tepper.cmu.edu/blog/wp-content/uploads/2012/09/2012-Matheuristic-Fischetti-erraticism.pdf, 2012.

[9] M. Fischetti and M. Monaci. Exploiting erraticism in search. Working paper, University of Padova, Padova, Italy.

[10] M. Fischetti, A. Lodi, M. Monaci, D. Salvagnin, and A. Tramontani. Tree search stabilization by random sampling. Technical Report OR/13/5, DEI, University of Bologna, Bologna, Italy, 2013.

[11] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys* 23:5–48, 1991.

[12] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. *Proceedings of the National Conference on Artificial Intelligence*, Vol. 48. AAAI Press, Menlo Park, CA, 431–437, 1998.

[13] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. Satisfiability solvers. F. van Harmelen, V. Lifschitz, B. Porter, eds. *Handbook of Knowledge Representation*, Foundations of Artificial Intelligence, Vol. 3. Elsevier, Amsterdam, 89–134, 2008.

[14] C. P. Gomes, B. Selman, N. Crato, and H. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning* 24:67–100, 2000.

[15] GUROBI. GUROBI optimization. Accessed July 21, 2013, http://www.gurobi.com/.

[16] IBM ILOG. CPLEX Optimization Studio. Accesses July 21, 2013, http://www.cplex.com.

[17] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, A. Lodi, H. Mittelmann, T. Ralphs, D. Salvagnin, D. E. Steffy, and K. Wolter. MIPLIB 2010. *Mathematical Programming Computation* 3:103–163, 2011.

[18] J. T. Linderoth and A. Lodi. MILP software. J. J. Cochran, ed. *Wiley Encyclopedia of Operations Research and Management Science*. John Wiley & Sons, Hoboken, NJ, 3239–3248, 2011.

[19] J. T. Linderoth and M. W. P. Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing* 11(2):173–187, 1999.

[20] A. Lodi. MIP computation. 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*. Springer-Verlag, Berlin, Heidelberg, 619–645, 2009.

[21] A. Lodi. The heuristic (dark) side of MIP solvers. E.-G. Talbi, ed. *Hybrid Metaheuristics*. Springer, Berlin, 273–284, 2012.

[22] H. Mittlemann. Mixed integer linear programming benchmark (MIPLIB 2010). Accessed March 15, 2013, http://plato.asu.edu/ftp/milpc.html, 2010.

[23] M. W. Padberg and G. Rinaldi. A branch and cut algorithm for the resolution of large-scale symmetric traveling salesmen problems. *SIAM Review* 33:60–100, 1991.

[24] SCIP. Solving Constraint Integer Programs. http://scip.zib.de/.

[25] A. Tramontani, M. Fischetti, A. Lodi, M. Monaci, and D. Salvagnin. Root cut loop parallelization for solving mixed integer programs. IP.com Prior Art Database Disclosure IPCOM000225942D, IP.com, Fairport, NY, 2013.