

Testing Cut Generators for Mixed-Integer Linear Programming

François Margot *

Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA 15213
fmargot@andrew.cmu.edu

September 2007; revised May 2009

Abstract

In this paper, a methodology for testing the accuracy and strength of cut generators for mixed-integer linear programming is presented. The procedure amounts to random diving towards a feasible solution, recording several kinds of failures. This allows for a ranking of the accuracy of the generators. Then, for generators deemed to have similar accuracy, statistical tests are performed to compare their relative strength. An application on eight Gomory cut generators and six Reduce-and-Split cut generators is given. The problem of constructing benchmark instances for which feasible solutions can be obtained is also addressed.

1 Introduction

Empirical testing of cut generators for Mixed-Integer Linear Programming (MILP) is a problem appearing in various settings. In its simplest form, it occurs when studying a new class of cutting planes, or when a new implementation of a cut generator needs to be compared with an existing one. A related situation is the problem of testing different cutting strategies, where decisions about which cut generators to apply and how to set their respective parameters need to be made. Note also that generators need to be compared on speed (Does the generator help solving a problem faster?) and accuracy (How often does the generator produce invalid cuts?). The latter point, in particular, seems to have been completely ignored in the literature. Developers usually add various safeguards to avoid as much as possible the generation of invalid cuts due to limited numerical accuracy, but the effectiveness and pertinence of these safeguards is rarely discussed and almost never supported by data.

Testing accuracy of cut generators is part of the larger problem of testing the accuracy of MILP solvers. As pointed out by Neumaier et al. [34], very little work has been done in this area. There has been interest in testing the accuracy of LP solvers and developing exact LP solvers [5, 12, 20], but virtually nothing for testing the accuracy of MILP solvers. Neumaier et al. [34] discuss post-processing of the LP solution to ensure a valid lower bound and variants of several cutting planes algorithms to ensure the validity of the cuts even when computations are done in finite precision arithmetic. Note also that Applegate et al. [5] have implemented an exact MILP solver that can be used to solve modest-sized instances, but

*Supported by ONR grant N00014-09-1-0033.

report computing times that are usually two or three orders of magnitude larger than those of commercial finite precision MILP solvers.

In this paper, we are interested in getting an empirical measure of the accuracy and strength of a given implementation of a cut generation algorithm with computation performed in finite precision arithmetic. Most empirical papers benchmarking cut generators run experiments on a collection of MILP instances using one of the two following approaches. The first one is to apply a cut generator repeatedly on the linear relaxation of the instance and to report the change in the LP bound. The second one embeds the cut generator in a branch-and-cut code and reports the solution times and number of nodes to solve the sample instances. In either cases, the evaluation of the contribution of the cut generator is based on an average result on the sample instances.

The second approach dominates in published empirical papers, as it benchmarks the “true” objective: solving instances as fast as possible. But this is more about benchmarking branch-and-cut codes than testing cut generators. As pointed out by Hooker in [16, 17], this type of empirical results yield very little insight on how to improve the tested algorithms. McGeoch develop this idea in several papers [26, 27, 28, 29].

For example, given two algorithms for solving a particular class of instances, one might want to list properties of instances on which the first algorithm is usually better than the second one. By carefully devising random instances with and without a particular property, one can test how the property impacts the performances of the two algorithms and test if one is significantly better than the other when the property is present in the instance. Studies of this type can be found in the literature, e.g., for the Bin Packing Problem [26], for the Generalized Assignment Problem [3], for the Computer Network Design Problem [32], for computing asymptotic estimates of algorithm complexity [30], for analysis of Integer Linear Programming algorithms [21], for Network Reoptimization Techniques [2], and for an analysis of the effect of the choice in starting points for Nonlinear Optimization Algorithms [15].

The goal of this paper is to describe a methodology for testing cut generators in the spirit of the empirical analysis of algorithms of Hooker and avoiding pitfalls of the two approaches mentioned above. We now discuss weaknesses of these two approaches.

For the first one, the strength of the cuts generated at the root node is not always a good indicator of the usefulness of a family of cuts, and it is not clear how many rounds of cutting should be used to get results relevant for the typical use of the generator. Moreover, this approach is unlikely to detect that a cut generator generates more invalid cuts than another. It is also difficult to obtain convincing evidence of the effect of small changes in the parameter settings of similar cut generators. As an illustration, consider Table 1. It gives the percent gap closed by ten cutting rounds for six different cuts generators on sample instances (the generators and instances are described in more detail in sections 3 and 4). The percent gap closed by ten cutting rounds on instance I is defined as

$$100 \cdot \frac{F - LB}{OPT - LB}$$

where OPT is the value of an optimal solution of I , LB is the value of the initial LP relaxation of I , and F is the value of the LP relaxation obtained after generating ten rounds of cuts. The last two lines in the table give the average gap closed on the thirty two instances and the number of instances where the generator has the best performance among the six generators.

Table 1: Percent gap closed at the root by ten rounds of cutting for six cut generators on MIPLIB3_C instances. Best performance on each instance is in bold face.

name	G	GN	G2P4	G2P5	RSP4	RSP5
bell3a_c	63.08	63.30	63.08	63.08	62.41	62.41
bell4_c	93.20	92.77	92.94	93.20	48.50	48.79
bell5_c	87.58	87.59	87.58	87.58	64.03	64.03
blend2_c	31.86	31.85	31.86	31.86	31.85	31.85
dcmulti_c	68.19	75.32	67.96	68.15	51.31	50.00
dsbmip_c	0.00	0.00	0.00	0.00	0.00	0.00
egout_c	56.59	56.20	56.59	56.59	22.70	22.70
enigma_c	100.00	100.00	100.00	100.00	100.00	100.00
fixnet3_c	6.62	6.62	6.62	6.62	3.19	3.19
fixnet4_c	12.81	13.08	12.81	12.81	9.03	9.03
flugpl_c	14.83	14.74	14.83	14.83	16.27	16.27
gen_c	60.46	63.35	60.66	60.66	60.79	60.26
gesa3_c	64.13	61.78	64.13	64.13	75.54	75.54
gesa3_o_c	64.65	65.83	64.65	64.65	77.05	75.39
gt2_c	80.93	80.95	80.94	80.93	33.97	30.62
khb05250_c	96.20	96.47	96.17	96.20	63.37	63.37
l152lav_c	29.62	31.01	26.80	29.62	18.04	20.91
lseu_c	64.92	77.79	64.92	64.92	72.13	72.13
misc03_c	14.81	18.48	14.16	14.81	18.97	18.97
misc06_c	69.16	69.28	69.16	69.16	41.31	41.31
mitre_c	96.15	99.76	99.76	96.15	94.57	94.57
mod008_c	30.13	30.13	30.13	30.13	50.98	50.98
p0033_c	12.60	12.60	12.60	12.60	12.68	12.68
p0201_c	57.41	53.33	58.00	56.77	58.10	61.64
p0282_c	15.78	12.11	13.79	15.78	6.53	8.47
p0548_c	4.78	4.96	4.78	4.78	4.52	4.56
qnet1_c	29.20	29.26	35.22	29.54	35.63	41.04
qnet1_o_c	54.36	51.53	53.38	54.36	64.62	63.79
rgn_c	42.53	33.41	27.10	34.89	94.22	94.22
stein27_c	0.00	0.00	0.00	0.00	0.00	0.00
stein45_c	0.00	0.00	0.00	0.00	0.00	0.00
vpm1_c	62.87	59.65	62.20	62.97	75.84	89.24
Average	46.42	46.66	46.03	46.18	42.75	43.37
#Best	9	17	8	9	12	13

The best-to-worse ranking based on the average is GN, G, G2P5, G2P4, RSP5, RSP4. However, it is clear that the first four algorithms are fairly close to each other and it is difficult to confidently claim that GN is better than G2P4 based on these results. Indeed, most of the

difference in average gap closed between these two algorithms comes from a single instance (`lseu.c`), `G2P4` closes at least 1% more gap than `GN` on 6 instances while the converse happens 7 times.

Comparing performances of algorithms by statistical tests is well covered in the literature. We refer the reader to [8] for an excellent introduction to the topic. The basic test commonly used when comparing performances of two algorithms is a t -test¹. Such a test comparing `GN` and `G2P4` rates their difference in performances as non significant with a 95% confidence level. This is not very surprising, as the difference is small and the number of experiments is low. A little bit more surprising, perhaps, is that even the difference between `GN` and `RSP4` is not significant with a 95% confidence level. This illustrates the difficulty of obtaining convincing evidence for the superiority of a cut generator with this approach when differences in performances and number of sample instances are small.

The second approach, embedding the cut generator in a branch-and-cut code, is also problematic. First, due to the complexity of the branch-and-cut algorithm, the results of the tests are affected by a multitude of side-effects introduced by small changes in cut generation. For example, the choice of branching variable might change completely when cut generation is modified if the choice is based on the solution of the LP relaxation. As a result, small changes in cut generation might yield completely different enumeration trees, with large variance in performances. This approach, of course, gives important insight on how to use various cut generators within one particular branch-and-cut code, but it is questionable if studies with one code are relevant for another code. The variation in performances of these experiments is larger than for the first approach, but the number of experiments remains relatively low. It is thus still difficult to obtain statistically significant results. Moreover, these experiments do not provide a clean feedback on the effect of modifications of a cut generator if more than one cut generator is used. It is well-known for example that Mixed-Integer Gomory cuts [14], Mixed-Integer Rounding cuts [33], and Disjunctive cuts [6, 19] are equivalent [22]. It is then difficult to assess the contribution of one cut generator when several of them are used together.

The second approach is marginally better than the first one for testing if cut generators generate invalid cuts; when the optimal solution value is known, one can compare it to the solution value found. It is generally assumed implicitly that all computations were accurate when the code finds an optimal solution of the instance. However, it might be the case that an optimal solution is found early in the search by a heuristic algorithm. What happens afterwards with the cut generators is then moot. To push things to the extreme, having a cut generator that generates invalid cuts can even appear positive, as the solution time might go down significantly.

To avoid these problems, one can run the branch-and-cut code using only one cut generator and turning off the heuristic algorithm, with the drawback that an instance that could be solved in minutes requires a much longer solution time. Assuming that it is possible to solve the instance in this way, the size of the enumeration tree would be a reasonable measure for the strength of the generated cuts. Indeed, cutting planes are used to reduce the size of the enumeration tree and strong cuts should yield small enumeration trees. However, the results would still be affected by other parts of the branch-and-cut code (branching variable

¹A short description of this test is given in Section 4.2.

selection, in particular) and they would not provide much information on the accuracy of the cut generator.

Table 2: Average number of variables set while diving towards a given feasible solution (20 trials for each solution). Ten rounds of cutting are applied after each setting of a variable. Number of solutions used is listed in column #Sol. Best performance on each instance is in bold face.

name	#Sol.	G	GN	G2P4	G2P5	RSP4	RSP5
bell3a_c	12	13.00	13.00	12.15	12.30	12.35	12.35
bell4_c	11	30.75	30.75	30.90	31.10	31.45	32.35
bell5_c	11	19.70	19.70	20.45	19.95	21.75	21.85
blend2_c	10	41.80	41.80	39.70	36.90	45.00	42.70
dcmulti_c	12	27.28	26.81	25.60	25.06	25.05	26.38
dsbmip_c	1	42.20	42.20	43.55	42.60	47.45	43.60
egout_c	1	24.10	24.10	23.95	23.20	37.60	37.20
enigma_c	1	15.10	11.75	13.90	14.10	13.40	16.70
fixnet3_c	11	119.35	119.35	119.15	119.15	122.05	122.05
fixnet4_c	1	136.44	137.41	136.30	138.55	138.65	139.20
flugpl_c	10	4.20	4.20	4.15	4.15	4.10	4.10
gen_c	13	16.75	18.45	19.90	19.65	20.85	20.15
gesa3_c	1	18.65	18.65	18.40	18.80	19.85	20.80
gesa3_o_c	1	19.15	19.15	20.35	19.80	22.90	22.70
gt2_c	19	31.30	34.80	36.95	37.58	36.20	35.65
khb05250_c	12	13.90	13.90	14.30	14.30	15.75	15.85
l152lav_c	14	34.40	34.60	31.05	31.05	32.40	32.40
lseu_c	13	13.25	13.80	14.30	15.65	15.80	16.55
misc03_c	16	12.05	10.70	10.55	12.30	11.05	12.25
misc06_c	1	6.75	6.75	5.95	5.95	10.00	10.00
mitre_c	11	42.05	51.40	79.35	86.15	83.25	76.70
mod008_c	25	11.15	11.15	13.50	13.80	15.25	15.40
p0033_c	11	9.85	9.90	11.05	9.90	11.65	12.05
p0201_c	15	11.85	10.80	12.25	13.95	12.75	11.85
p0282_c	1	40.50	40.74	43.45	45.70	51.30	48.25
p0548_c	11	95.20	96.00	113.90	121.95	132.85	131.60
qnet1_c	1	44.15	48.20	46.15	46.45	45.65	46.15
qnet1_o_c	1	53.75	52.45	46.20	52.95	50.00	50.10
rgn_c	1	11.55	11.55	13.95	11.75	7.50	7.50
stein27_c	12	12.90	11.65	12.35	12.80	11.05	11.90
stein45_c	14	22.00	20.75	20.60	20.10	21.05	20.95
vpm1_c	1	24.95	24.95	24.50	27.10	19.45	22.00
Average	--	31.2	31.34	32.23	32.92	34.28	34.24
#Best	--	14	8	8	6	4	2

This paper proposes to address these issues using an algorithm we call *random diving towards a 0-feasible solution*. A detailed description of this algorithm and the definition of a 0-feasible solution are given in Section 2 and the generation of 0-feasible solutions is addressed in Section 3. A sketch of the algorithm is the following: For an instance I with optimal solution x^* , simulate the path of the enumeration tree of a branch-and-cut code that will end with an LP relaxation containing x^* . More precisely, repeat the following operations until a feasible solution \bar{x} is obtained: apply the cut generator, get an optimal solution \bar{x} of the current LP relaxation, pick uniformly at random an integer variable x_i with \bar{x}_i fractional and set its value to x_i^* . The fact that x^* is feasible implies that it should not violate any of the generated cuts. As the choice of the variable to be set is random, it is possible to repeat this diving a number of times (we use 20 trials in this paper) and record the average number of variables set before reaching an integer solution and the number of times x^* violates one or more generated cuts.

This algorithm gives also information on the strength of the generated cuts, as the number of variables set during a dive is the length of a possible path of the enumeration tree of a branch-and-cut algorithm. The randomization makes the estimation independent of the particular branching variable choice used in a branch-and-cut code, focusing more on the strength of the generated cuts themselves. It should be clear that, in this paper, the meaning of the term “strength” for a cut generator relates to the size of the enumeration tree of a branch-and-cut algorithm using the generator. There are other pertinent measures of strength of families of cuts, each with its own purpose.

Note that the diving algorithm can be applied using any feasible solution in place of x^* . As an illustration of strength comparison, consider Table 2 with the average results for this diving algorithm for the same generators and instances as in Table 1.

One can note that the ranking of the algorithms based on their average performances in tables 1 and 2 are quite similar. For Table 2, the best-to-worse ranking is **G**, **GN**, **G2P4**, **G2P5**, **RSP5**, **RSP4**. This ranking is obtained from the ranking of Table 1 by swapping **G** with **GN** and **G2P4** with **G2P5**. However, a major difference between the two tables is that statistical tests (see Section 4.2 for details) based on the results of the diving experiments show that, with a 95% confidence level, **G** and **GN** are better than **G2P4** which is better than **G2P5** which is better than **RSP4** and **RSP5**. Only the ranking between **G** and **GN** and between **RSP4** and **RSP5** can not be determined with that confidence level.

The paper is organized as follows. Section 2 describes in detail the proposed method for testing a cut generator. This method, random diving towards a feasible solution, requires the knowledge of a feasible solution of the instance, where the feasibility requirement is much stricter than the commonly used “not violating too much any constraint”. This requirement is quite difficult to meet for many usual benchmark MILP instances. Section 3 describes how benchmark instances were altered to obtain the instances and 0-feasible solutions used in the application example of Section 4. This application example tests eight Gomory cut generators and six Reduce-and-Split generators. First, these fourteen generators are compared for accuracy in Section 4.1 and six of them are deemed having similar accuracy. These six generators are then compared for strength in Section 4.2. Conclusions are given in Section 5.

An extended abstract of this paper appeared in [24].

2 Random diving towards 0-feasible solutions

Consider the instance I of an MILP instance:

$$\begin{aligned}
 \min \quad & c \cdot x + d \cdot y \\
 \text{s.t.} \quad & A \cdot x + B \cdot y \geq b \\
 & x \in \mathbb{R}^{n_1} \\
 & y \in \mathbb{Z}^{n_2},
 \end{aligned} \tag{1}$$

where $c \in \mathbb{Q}^{n_1}$, $d \in \mathbb{Q}^{n_2}$, $b \in \mathbb{Q}^m$, A is an $m \times n_1$ rational matrix, and B is an $m \times n_2$ rational matrix. Let (x^I, y^I) be a solution of I and let $\epsilon \geq 0$. The solution is ϵ -integer if each entry in y^I is within ϵ of an integer value. The solution is ϵ -feasible if it is ϵ -integer and the absolute violation of any of the constraints $A \cdot x^I + B \cdot y^I \geq b$ is at most ϵ .

It is quite difficult to find how often a cut generator for MILP generates invalid cuts. We suggest to estimate this by generating a set S of feasible solutions and testing how often one of them is cut. In order to have a valid test, it is necessary that the solutions in S are 0-feasible solutions, as any solution that is not ϵ -feasible for some $\epsilon > 0$ might correctly be cut by a generated cut. This is more than a minor problem, as MILP solvers return slightly infeasible solutions on most instances. Even worse, it is usually impossible to set the required precision on the solution returned by the solver. Most of them have options for setting feasibility tolerance and integer tolerance of the solution, but due to numerical inaccuracies in the LP solver or in the cut generation, pruning, fixing of variables or other, getting 0-feasible solutions is virtually impossible. The way we generate instances with 0-feasible solutions is described in the next section.

In this section, just assume that we have a 0-feasible solution (x^I, y^I) of an instance I of (1). Assume that we want to test the accuracy of a cut generator. We can dive towards the solution, while using the cut generator, and record if the solution is still ϵ -feasible or not for some value of ϵ (we use $\epsilon = 10^{-6}$ in the tests). More precisely:

Algorithm 1: Diving towards a 0-feasible solution.

1. Start with the LP relaxation of I ; flag := 0.
2. Repeat
 - 2.1 Repeat k times
 - 2.1.1. Generate and apply cuts.
 - 2.1.2. Resolve the LP.
 - 2.1.3. If the LP is infeasible then flag := 2 and stop.
 - 2.1.4. Otherwise, let (\bar{x}, \bar{y}) be the optimal LP solution.
 - 2.2. If (x^I, y^I) is not ϵ -feasible then flag := 1 and continue.
 - 2.3. If (\bar{x}, \bar{y}) is ϵ -integer then stop.
 - 2.4. Otherwise select randomly an index j with \bar{y}_j fractional.
 - 2.5. Set $y_j := y_j^I$ in the LP.
 - 2.6. If a time limit is reached then flag := 3 and stop.

The algorithm either terminates with `flag = 0`, meaning that the LP relaxation has an ϵ -integer feasible optimal solution and (x^I, y^I) is still ϵ -feasible, or it raises one of three types of failures indicated by the value of `flag`:

- `flag = 1`: (x^I, y^I) is no longer ϵ -feasible, but another ϵ -integer feasible solution is reached.
- `flag = 2`: The LP relaxation is infeasible.
- `flag = 3`: The time limit is reached.

Terminating with `flag = 1` or `flag = 2` is annoying, but reaching `flag = 1` is less severe than reaching `flag = 2`, as a slight alteration of the values of the continuous variables x^I might restore ϵ -feasibility. Note also that reaching `flag = 2` implies that (x^I, y^I) is infeasible, i.e., that `flag = 1` is implicitly raised too. Terminating with `flag = 3` is not an indication that invalid cuts have been generated, but this is an indication that the time spent for solving the LP or for generating the cuts is unusually high, or that the random branching choices have been unlucky. The former case indicates some problems with the generator, although no precision problem. A study of the average time per iteration of the algorithm could easily distinguish between the two cases. Alternatively, one could consider reaching `flag = 3` a serious failure if the time limit is large enough.

Notice that it is possible that the algorithm terminates in step 2.3 with a solution (\bar{x}, \bar{y}) that is ϵ -integer but not ϵ -feasible. It could also happen that the algorithm stops in step 2.1.3, reporting incorrectly that the LP is infeasible due to a similar lack of precision in the LP solver. However, both cases seems quite unlikely to happen, in particular if some control on the numerical stability of the instance and of the generated cuts are used (this is addressed in the next section). Note that these problems could be avoided by using an exact LP solver such as `QSopt.ex` [5], `Lpex` [12], or `perPlex` [20]. However, if the cut generator is intended to be used with a non-exact LP solver, an investigation of the joint reliability of the cut generator and LP solver is more relevant than an investigation of the reliability of a cut generator coupled with an exact LP solver.

It is debatable what should be reported in the case where the time limit is reached in step 2.6 with `flag = 1` already set. Maybe reporting that both flag values 1 and 3 were raised would be better. This has virtually no impact for the results reported in this paper, as the case occurred only a couple of time for five of the generators, and all these occurred on a single instance creating numerous numerical failures for these generators. This instance (`bienst1.c`) and related numerical issues are discussed in Section 4.1.

The above scheme has several interesting features: First, the randomization in step 2.4 allows for statistical testing. From one instance with a few hundred of integer variables, one can generate many observations. It is also possible to make statistics on the number of variables set to integer values in step 2.4 before reaching an integer solution (this allows for an estimation of the strength of the generated cuts), cut generation time, LP resolve time, evolution of the lower bound, and, of course, the failure types. Observe also that, in opposition to the experiments tracking only the gap closed at the root, some information is derived from instances that either have a zero gap (such as `enigma`) or for which the cut generator fails to reduce the gap at the root (such as `stein27` and `stein45`).

On the other hand, we do not get information on the effect of the cuts on any deterministic branching choice, or on the size of the enumeration tree obtained by a branch-and-cut code. This test is devised to test the accuracy of the cuts, not to predict the power of a cut generation strategy in a branch-and-cut algorithm. In a branch-and-cut algorithm, many parameters interfere with each other, such as cutting strategy and branching strategy. It is then difficult to assign praise or blame on either cutting or branching strategy independently of the other. This test is focused on cut generators alone and its goal is to obtain information about how to rank and improve performances of various cut generators. Note, however, that it is conceivable that some information about the effect of a cut generator on the size of the enumeration tree could be obtained by modifying Algorithm 1 in order to explore a restricted enumeration tree “centered” around the feasible solution, in a way similar to the restricted enumeration performed in the MILP heuristic *local branching* [13] and variants [11].

The underlying contention advanced in this paper is that a fair comparison of strength of cut generators has to take into account the respective accuracy of the generators. One can try to describe Pareto-optimal cut generators with respect to strength and accuracy, or, as pursued in this paper, try to set the parameters of the generators in order to get a similar accuracy and then compare these generators with respect to strength.

Note also that a test similar to Algorithm 1 could be devised for other parts of a branch-and-cut algorithm, such as preprocessing and variable fixing, but this seems to require that the 0-feasible solution then used is the unique optimal solution of the MILP. This requirement can be lifted only if the modifications made to the instance are valid for all feasible solutions (or, at least the solution at hand). For example, fixing integer variables based on reduced cost fixing or based on the result of strong branching computations might correctly exclude some feasible solutions. Valid column aggregation or variable fixing can even exclude some optimal solutions.

Note that Algorithm 1 could also be used to test the accuracy of an LP solver, putting more stress on it than in tests only on the instance I as done in [5, 12, 20]. As mentioned in [34], ill-conditioning is likely to arise in the LP relaxation of subproblems created by branch-and-cut.

3 Generation of 0-feasible solutions

As mentioned in the previous section, Algorithm 1 requires diving towards a 0-feasible solution. However, generating such a solution is quite challenging for many MILP benchmark instances. This section explains what was done to construct the benchmark instances and 0-feasible solutions used in this paper. A reader not interested in these details can skip this section. The only information relevant for the remainder of the paper is that instances are constructed from usual benchmark instances by reducing the number of significant digits in coefficients and right-hand side entries and by possibly enlarging slightly the feasible region. The instance names and number of 0-feasible solutions obtained are listed in Figure 1.

Let us first explain the goals pursued. The first objective is to limit accuracy problems created by the initial formulation. The second one is getting instances for which generating 0-feasible solutions is not too difficult. Finally, we would like to have instances whose structure is similar to classic benchmark instances. We thus chose to start with instances from MIPLIB3

[7] and from a collection called MITT put together by Mittelmann [31], selecting (and altering slightly) some of the instances, as described below.

Define the *dynamism* of an instance as the largest ratio between the smallest and largest absolute values of coefficients in a constraint. Some instances from MIPLIB3 such as `arki001` has dynamism as high as $1.3 \cdot 10^{10}$. Basic numerical analysis [37] can easily convince the reader that the likelihood of inaccurate rounding errors with severe consequences when solving this instance is quite high. This is not to say that this instance is useless. It might be quite interesting to test the accuracy of an LP solver, or to test the robustness of a branch-and-cut code. It is just inappropriate for testing the accuracy of a cut generator, in our opinion.

The MIPLIB instances selected for inclusion in the benchmark were the thirty six instances from MIPLIB3 that are not in MIPLIB2003 [1], except three: `air03`, `mod010`, and `swath`. The first one is excluded since it is solved at the root by most cutting plane algorithms, the second one since very few cuts are generated (in most runs not a single cut is produced) and the third one since the three instances `swath1`, `swath2`, and `swath3` from MITT are relaxations of `swath`. In total, we thus have thirty three MIPLIB3 instances and 10 MITT instances.

Note that this selection of instances is quite arbitrary. As the results of this paper show, the conclusions on the strength and precision of cut generators depend on the test instances. If an actual test of generators is performed, it is thus important to select appropriately the test instances to reflect the types of instances the code will face in the future. Since the results in this paper are only for illustration purposes, not much effort was made in this selection. Note that rounded instances and 0-feasible solutions for all the MIPLIB3 instances are available from [25], as well as code for producing them for any other instance.

These forty three instances are then modified in order to obtain instances for which 0-feasible solutions can be constructed. The modifications are of two types. First, all coefficients and right-hand sides are replaced by their respective *d-digits rounding*, i.e., the closest number represented with a mantissa with up to d digits and multiplied by an integer power of ten. In addition, a *d-digits rounding* of a value v with $0 < v < 10^{-d}$ (resp. $-10^{-d} < v < 0$) can only be 0 or 10^{-d} (resp. -10^{-d} or 0). All entries of the 0-feasible solution sought should also be *d-digits roundings*. These rounding operations might give an infeasible instance, in particular for instances with several equality constraints. To restore feasibility and to be able to generate 0-feasible solutions (with d significant digits), the right-hand sides are modified, enlarging slightly the feasible region. We could use an exact-arithmetic MILP solver such as `QSopt_ex` [5] to solve the modified instances. However, it is not guaranteed that a *d-digits rounding* of the solution obtained by `QSopt_ex` would be 0-feasible. Also, large MILP instances that are part of usual benchmark instances can not be solved in a reasonable amount of time in exact arithmetic with current codes.

Let a *d-digits rounding* of a solution (x, y) of (1) be obtained by a *d-digits rounding* of each x_j , and by rounding each y_j to the closest integer having at most d significant digits.

While it is debatable if rounding the coefficients in the instance truly makes the instance more stable numerically or not, the fact that we also use 0-feasible solutions that are rounded to d digits is likely to force the solution slightly in the interior of the feasible region, making it less likely to be cut due to numerical errors. Of course, this makes the generation of these solutions more difficult.

Note that it is almost impossible to modify these instances in order to get no rounding errors affecting the results. The most commonly used representation of a `double` number

uses 8 bytes and has roughly 16 digits of precision and thus a rounding unit $\epsilon_M \approx 10^{-16}$. The relative error on computing a sum $z_1 + \dots + z_n$ is less than $\kappa \cdot \epsilon_M$ where κ is the condition number of the sum, defined as:

$$\kappa = \frac{|z_1| + \dots + |z_n|}{|z_1 + \dots + z_n|}.$$

This gives, of course, an upper bound on the error, and the actual error is usually smaller. Nevertheless, it is no surprise that making sums with large positive and negative numbers can lead to major loss of precision in the computation. Note also that most instances have variables with no upper bounds, implying that, potentially, having very large absolute values in a sum is possible. In addition, some cut generators (such as Gomory cut generators, see Section 4 for details) generate cuts with right-hand side between 0 and 1, but with coefficients on the left-hand side that might be large. Computing the violation of a solution that violates the cut only slightly is then exactly the type of computation that might suffer from severe loss of precision. Nevertheless, limiting the number of significant digits in the instances and 0-feasible solutions might help to reduce the likelihood of damaging rounding errors.

Indeed, observe that the product of any two numbers having mantissas with 6 decimal digits has a mantissa of at most 12 decimal digits and can thus be computed with no error assuming that no overflow for the exponent occurs, something we can safely assume for the instances at hand. To compute exactly the left-hand side of an inequality $a \cdot x \leq b$ where both the coefficient vector a and the solution x have 6 significant digits, the number of digits required can be bounded by

$$\bar{d} = \left\lceil \log_{10} \left(\frac{\max \left\{ \sum_{i=1}^n (a_i \cdot x_i)^+, -\sum_{i=1}^n (a_i \cdot x_i)^- \right\}}{10^{-6} \cdot t} \right) \right\rceil, \quad (2)$$

where $(a_i \cdot x_i)^+ = \max(0, a_i \cdot x_i)$, $(a_i \cdot x_i)^- = \min(0, a_i \cdot x_i)$, and $t = \min\{|a_i \cdot x_i| : |a_i \cdot x_i| > 0, i = 1, \dots, n\}$. Indeed, the numerator (resp. denominator) of the fraction is an upper bound (resp. lower bound) on the absolute value of the largest number (resp. smallest nonzero number) that can occur during the computation of the sum. It turns out that for the instances and solutions used in the experiments, we have $\bar{d} \leq 16$. This guarantees that feasibility of the solutions produced for the modified instances can be checked accurately without having to rely on exact arithmetic packages. Note that, however, when checking ϵ -feasibility of (x^I, y^I) with respect to the generated cuts in Algorithm 1, it could happen that numerical errors in the computations induce a wrong answer. As equation (2) indicates, the probability of this occurring can be reduced by lowering the dynamism of the generated cuts.

The algorithm used to adjust the coefficients of an instance I and generate a 0-feasible solution with a value close to the original optimal value is Algorithm 2.

If the algorithm reaches step 6, then (\bar{x}, \bar{y}) is a 0-feasible solution for \bar{I} . However, we do not claim that (\bar{x}, \bar{y}) is an optimal solution for \bar{I} .

Note that in step 5.5, when computing a new upper (resp. lower) bound on a constraint, the d -digits rounding is obtained by rounding up (resp. down). When dealing with an equality in step 5.5, the equality is first transformed to a ranged constraint and then either the lower or upper bound (as appropriate) is modified. On the other hand, when initially rounding the

right-hand side of an equality in step 4, the equality is not modified to a ranged constraint. As a result, the initial rounded instance might be infeasible or unbounded and modifications in step 5.5 always enlarge the feasible set of the instance.

Algorithm 2: Modifying an MILP formulation and solution.

1. Let d be the maximum number of significant digits to use.
2. Solve I with an MILP solver, setting parameters on the precision of the solution to the highest possible level. Let (x', y') be the solution.
3. Let (\bar{x}, \bar{y}) be a d -digits rounding of (x', y') .
4. Let \bar{I} be the instance obtained by replacing every entry in I by a d -digit rounding of the entry.
5. Repeat up to k times
 - 5.1. Let \bar{I}_F be the linear program obtained by fixing $y = \bar{y}$ in \bar{I} .
 - 5.2. If \bar{I}_F is infeasible, go to step 5.5. If it is unbounded, stop with a failure.
 - 5.3. Let (\bar{x}, \bar{y}) be a d -digit rounding of the optimal solution of \bar{I}_F .
 - 5.4. If (\bar{x}, \bar{y}) is 0-feasible for \bar{I} then go to step 6.
 - 5.5. Replace the right-hand side value for all violated constraints in \bar{I} by a d -digits rounding of the left-hand side value obtained for (\bar{x}, \bar{y}) .
 - 5.6. Let δ be the maximum absolute change of any right-hand side entry in step 5.5 and let Δ the relative change in objective function value for the last two solutions (\bar{x}, \bar{y}) . If $\delta < 10^{-d}$ and $\Delta < 10^{-d}$ then go to step 6.
6. Write \bar{I} , (\bar{x}, \bar{y}) and stop.

The stopping criterion in step 5.6 is useful on a couple of instances of MIPLIB3 namely `qnet1` and `qnet1_o`, and on several MITT.C instances (`bc1`, `bienst1`, `seymour`, `swath1`, `swath2`, and `swath3`). Without this condition, the algorithm never stops before hitting the maximum iteration number.

If the algorithm performs k iterations of step 5 without finding a 0-feasible solution, the current solution and instance are written out. This happened a few times on the selected instances (`dsbmip`, `gesa3`, `gesa3_o`, and `misc06`, `neos2`, `neos3`). The algorithm had a single failure, on instance `rentacar`. In this case, the initial LP becomes infeasible when coefficients are rounded to 6 significant digits². Note that this instance has a dynamism larger than 10^{10} and should probably have been discarded on that basis alone. Another MIPLIB3 instance, `dsbmip`, creates difficulties as it contains several hundred of free rows and several variables appearing only in these rows. Once these rows and variables are removed, the algorithm runs successfully.

²The failure was traced to the fact that this instance has a number of rows that have no nonzero left-hand side coefficients, a case that was not handled properly with the original implementation.

To generate more than one 0-feasible solution for an instance \bar{I} , we try to obtain a diversified set of 0-feasible solutions that are within 10% of the value obtained by the 0-feasible solution (\bar{x}, \bar{y}) of Algorithm 2. This is achieved using a simple branch-and-bound algorithm on instance \bar{I} with a customized selection rule for the next node to be processed and an upper bound 10% higher than the optimal value of the solution (\bar{x}, \bar{y}) . (This upper bound is not changed during the course of the algorithm, as any solution within 10% of the value of (\bar{x}, \bar{y}) is potentially interesting.) At node a , let $\ell^a(y_i)$ and $u^a(y_i)$ be the lower and upper bounds on variable y_i for $i = 1, \dots, n_2$. The *distance of a to (\bar{x}, \bar{y})* is defined as

$$\sum_{i=1}^{n_2} \{\max(0, \ell^a(y_i) - \bar{y}_i) + \max(0, \bar{y}_i - u^a(y_i))\}$$

This value is positive only if (\bar{x}, \bar{y}) is not feasible at node a . The branch-and-bound algorithm always dives from the current node, picking the son with largest distance to (\bar{x}, \bar{y}) . A time limit of 10 minutes on this diving algorithm is set and we hope to generate 10 alternate solutions. All feasible solutions found are recorded. Then, each of them is rounded as in steps 5.1 and 5.3 of Algorithm 2. If the resulting solution is 0-feasible for \bar{I} , it is saved and discarded otherwise.

The instances and solutions used in this paper are obtained using Algorithm 2 with $d = 6$ and $k = 10$. They are available from [25]. The names of the modified instances are the original names with a “_c” appended. Note that for instances `bienst1_c` and `swath3_c`, we are unable to generate additional 0-feasible solutions using $d = 6$. The additional solutions generated have $d = 8$ for `bienst1_c` and $d = 7$ for `swath3_c`. The initial solutions used as input to Algorithm 2 are obtained using `Cplex 10.1` [18], but `Clp 1.3` (available from `COIN-OR` [9]) is the LP solver used in Algorithm 2. The diving algorithm is coded on top of the generic branch-and-cut-and-price software `BCP` (version `stable/1.0`) [9, 23] with `Clp` as LP solver.

The number of solutions generated and the instances considered are listed in Figure 1. This gives 275 solutions for the thirty two `MIPLIB3_C` instances and 81 solutions for ten `MITT_C` instances.

4 Application example

As an example of application of Algorithm 1, tests on variants of four cut generators are reported. Note that the purpose of this section is to illustrate the type of analysis that can be done using the proposed method. The tested generators were hand picked without extensive study and it is almost certain that none of the variants tested in this section is the ideal generator. The four generators are:

1. `CglGomory`: The default Gomory cut generator of the Cut Generation Library (`Cgl`) of `COIN-OR` [9]. This generator is denoted by G in the remainder.
2. `CglGomory_nogcd`: `CglGomory` removes some of the generated cuts based on the result of a greatest common divisor computation for coefficients of the integer variables. `CglGomory_nogcd` skips this step. This generator is denoted by GN in the remainder.

Figure 1: MIPLIB3_C instances (left) and MITT_C instances with number of 0-feasible solutions.

name	#sol.	name	#sol.
bell3a_c	12	l152lav_c	14
bell4_c	11	lseu_c	13
bell5_c	11	misc03_c	16
blend2_c	10	misc06_c	1
dcmulti_c	12	mitre_c	11
dsbmip_c	1	mod008_c	25
egout_c	1	p0033_c	11
enigma_c	1	p0201_c	15
fixnet3_c	11	p0282_c	1
fixnet4_c	1	p0548_c	11
flugpl_c	10	qnet1_c	1
gen_c	13	qnet1_o_c	1
gesa3_c	1	rgn_c	1
gesa3_o_c	1	stein27_c	12
gt2_c	19	stein45_c	14
khb05250_c	12	vpm1_c	1

name	#sol.
bc1_c	1
bienst1_c	13
ip_c	6
mas284_c	26
neos2_c	4
neos3_c	1
prod1_c	11
swath1_c	9
swath2_c	4
swath3_c	6

3. **Cg1GomoryTwo**: A Gomory cut generator written by the author that has many parameters. This generator uses optimal tableau information provided by the LP solver whereas **G** and **GN** recompute the optimal tableau from the optimal basis information. This generator is denoted by *G2* in the remainder.
4. **Cg1RedSplit**: The default Reduce-and-Split [4] cut generator of **Cg1**. This generator uses optimal tableau information provided by the LP solver, similarly to **G2**. This generator is denoted by *RS* in the remainder.

All these generators are based on the mixed-integer Gomory cut (*MIG*) formula, **RS** having a heuristic way to generate linear combinations of the rows of the optimal Simplex tableau before using the *MIG* formula. This formula is usually applied to one row of the optimal tableau whose basic variable is integer. For example, assume that $\{y_j | j \in M\}$ are integer variables and that $\{x_j | j \in N\}$ are continuous nonnegative variables, a row of the optimal tableau having y_i as basic variable might be:

$$y_i + \sum_{j \in M \setminus i} \bar{a}_{ij} y_j + \sum_{j \in N} \bar{a}_{ij} x_j = \bar{a}_{i0} . \quad (3)$$

Define

$$f_j = \bar{a}_{ij} - \lfloor \bar{a}_{ij} \rfloor \quad \text{for all } j \in M \cup N \cup \{0\} . \quad (4)$$

Then if $f_0 > 0$, the *MIG* cut is:

$$\sum_{f_j \leq f_0} f_j y_j + \sum_{f_j > f_0} \frac{f_0(1-f_j)}{1-f_0} y_j + \sum_{\bar{a}_{ij} > 0} \bar{a}_{ij} x_j - \sum_{\bar{a}_{ij} < 0} \frac{f_0}{1-f_0} \bar{a}_{ij} x_j \geq f_0 . \quad (5)$$

As mentioned in the previous section, applying this formula blindly to generate cuts is likely to generate some invalid cuts. This is why all the generators listed above have ways to prevent the generation of invalid cuts as well as for discarding small coefficients in a cut. (Ways to generate Gomory cuts in finite precision arithmetic that are valid if validity is tested with infinite precision arithmetic are given in [34] when all variables are bounded, and in [10] in general; however, it is not clear how “safe” these cuts are when used in finite precision arithmetic.) Generators **G2** and **RS** are fully parametrized, making it easy to use them with different values of the parameters. Generators **G** and **GN** on the other hand have hard-coded constants requiring modification of the code to experiment with different settings. This is the reason behind the use of the **G2** generator for varying values of the parameters instead of the **G** generator for the experiments.

The parameters that are modified in the experiments and their default values (in **G2** and **RS**) are:

- **LUB** = 10^4 : If the absolute value of the upper bound on a variable is larger than that, it is considered large.
- **EPS_COEFF** = 10^{-5} : Any cut coefficient smaller than that for a variable that does not have a large upper bound is replaced by zero.
- **EPS_COEFF_LUB** = 10^{-13} : Similar to **EPS_COEFF** for variables having a large upper bound.
- **MAXDYN** = 10^8 : A cut is discarded if none of the variables with nonzero coefficient have a large upper bound and its dynamism is larger than this value.
- **MAXDYN_LUB** = 10^{13} : Similar to **MAXDYN**, but for cuts where some of the variables with nonzero coefficients have a large upper bound.
- **AWAY** = 0.05: Lower bound on the absolute value of $\min\{f_0, 1 - f_0\}$ (refer to (4) for the definition of f_0).
- **MINVIOL** = 10^{-7} : If the violation of the cut by the current optimal solution of the LP relaxation is lower than this number, the cut is discarded.

In addition to the default settings above, five variants of **G2** and **RS** are tested. All variants are more restrictive than the default generator and should generate fewer and “safer” cuts. Variants are labeled **G2P1** through **G2P5** and **RSP1** through **RSP5** with parameters set as in the default setting except the following: **P1** has **MINVIOL** = 10^{-4} , **P2** has **MINVIOL** = 10^{-2} , **P3** has **AWAY** = 0.08, **P4** has **MAXDYN** = 10^4 and **MAXDYN_LUB** = 10^8 , and **P5** has **MAXDYN** = 10^6 and **MAXDYN_LUB** = 10^{10} .

All generators are set so that there is no limit on the number of cuts they generate and a limit of 1,000 for the number of nonzero entries in a cut. The latter is obtained by setting the `limit` parameter of the generators to $\min\{n + 1, 1001\}$ where n is the number of variables in the instance. All results are obtained using $k = 10$ in Algorithm 1.

Before discussing the results, let us make it clear that these choices of ten rounds of cutting and up to 1,000 non zero entries in a cut are intended to put stress on the generators and LP solver. Using 10 rounds of cutting after each fixing of a variable is also probably not

the optimal setting for using these generators in a branch-and-cut code. Nevertheless, the comparison across the fourteen variants considered is a fair one.

The machine used for the test is a 64 bits Monarch Empro 4-Way Tower Server with four AMD Opteron 852 processors, each with eight DDR-400 SDRAM of 2 GB and running Linux Fedora 7. The compiler is g++ version 4.1.2 20070502 (Red Hat 4.1.2-12).

The LP solver used is Clp (stable version 1.3) without lapack and blas libraries. The generators G, GN and RS are from Cgl (stable version 0.5). Both Clp and Cgl are available from COIN-OR [9].

4.1 Comparing accuracy

The goal of this section is to compare the accuracy of the fourteen generators. We first compare the eight Gomory cut generators on the MIPLIB3_C instances. We use Algorithm 1 with $k = 10$, a time limit of 10 minutes for each dive, and 20 trials for each 0-feasible solution as listed in Figure 1.

In total, each cut generator is tested by 5,500 trials on the MIPLIB3_C instances and 1,620 trials on the MITT_C instances.

Table 3: Gomory cut generators comparison on MIPLIB3_C instances.

name	Flag			
	0	1	2	3
G	5,332	1	6	161
GN	5,319	3	5	173
G2	5,315	0	39	146
G2P1	5,359	0	38	103
G2P2	5,390	0	26	84
G2P3	5,440	0	23	37
G2P4	5,473	0	11	16
G2P5	5,456	0	10	34

Table 3 reports the value of `flag` at the end of Algorithm 1 for the eight variants of Gomory cut generators. In term of success (i.e. `flag = 0`), the winner is G2P4. For failures of types 1 or 2 (the most critical ones) both G and GN perform best, but the number of trials they are unable to complete within the time limit (10 minutes cpu) is much larger. There is an obvious difference in the failure patterns for these algorithms. A partition of them into the four pairs {G, GN}, {G2, G2P1}, {G2P2, G2P3}, and {G2P4, G2P5} seems a fair grouping. In term of failure 1 and 2, pairs {G, GN}, and {G2P4, G2P5} are similar, but the latter solve many more instances within the time limit.

The detailed breakdown of trials ending with `flag > 0` for the eight generators is given in Table 4.

From Table 4, results regarding failures of types 1 and 2 on three instances are worth pointing out: First, `dcmulti_c` creates problems for four out of six variants of G2. This instance has 548 variables, 75 of them binary, 290 constraints with a maximum absolute

Table 4: Distribution of failures for the Gomory cut generators on the MIPLIB3_C instances.

name	trials	G			GN			G2		G2P1		G2P2		G2P3		G2P4		G2P5	
		1	2	3	1	2	3	2	3	2	3	2	3	2	3	2	3	2	3
bell4_c	220	-	1	-	-	1	-	7	-	7	-	4	-	2	-	-	-	-	-
bell5_c	220	-	-	-	-	-	-	12	-	12	-	7	-	13	-	-	-	1	-
dcmulti_c	240	-	-	14	-	-	26	7	66	6	58	9	66	5	37	-	16	-	25
dsbmip_c	20	-	-	-	-	-	-	-	-	-	-	1	1	-	-	-	-	-	-
fixnet4_c	20	-	-	4	-	-	3	-	2	-	1	-	1	-	-	-	-	-	-
gen_c	260	-	-	-	-	-	-	1	-	1	-	1	-	-	-	-	-	-	-
gt2_c	380	-	2	-	-	4	-	9	-	9	-	3	-	2	-	8	-	9	-
misc03_c	320	-	-	-	-	-	-	-	-	-	-	-	-	1	-	1	-	-	-
mitre_c	220	-	3	-	-	-	-	1	-	1	-	1	-	-	-	-	-	-	-
p0033_c	220	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-
p0201_c	300	-	-	2	-	-	-	2	-	2	-	-	-	-	-	2	-	-	-
p0282_c	20	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-
p0548_c	220	1	-	141	1	-	143	-	78	-	44	-	16	-	-	-	-	-	9
stein27_c	240	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-

value of 265 for right-hand side, dynamism of 600, and a maximum absolute value of 500 for the entries of the 0-feasible solutions used. None of this is particularly alarming. It seems thus likely that the collection of cuts added by the generators is the culprit for the failures. An even more surprising situation is obtained with the instance `gt2_c`. This instance has 188 variables, 24 of them binary, 29 constraints with a maximum absolute value for right-hand side of about 6,600, all variables bounds at most 15 in absolute value, dynamism of 152, and a maximum absolute value of 9 for the entries of the 0-feasible solutions used. Yet, all generators fail at least twice on this instance. The instance `mitre_c` creates difficulties for some of the generators. It is a little bit larger (more than 10,000 variables), but all variables are binary and its dynamism, 1.63, is quite low. Finally, let us point out two extremely surprising failures for GN on `p0033_c` (33 variables, all binary) and `stein27_c` (27 variables, all binary, and with a binary constraint matrix).

Table 5: Reduce-and-Split cut generators comparison on MIPLIB3_C instances.

name	Flag			
	0	1	2	3
RS	5,359	1	40	100
RSP1	5,362	1	32	105
RSP2	5,372	1	27	100
RSP3	5,430	3	20	47
RSP4	5,478	0	4	18
RSP5	5,428	1	4	67

Let us now turn to the six variants of the RS generator. Table 5 shows that the variants `RSP4` and `RSP5` are clearly safer than the others. However, most of the failures of types 1 or 2 occur on a single instance, `dcmulti_c`. Overall the results are similar to the results obtained by the six corresponding variants of the G2 generator discussed above, except for instance `dcmulti_c` and instance `gt2_c`. The first one is handled better by the variants of the G2 generator and the second one by the variants of the RS generator. Note also here the failure

Table 6: Distribution of failures for the Reduce-and-Split cut generators on the MIPLIB3_C instances.

name	trials	RS			RSP1			RSP2			RSP3			RSP4		RSP5		
		1	2	3	1	2	3	1	2	3	1	2	3	2	3	1	2	3
bell4_c	220	-	12	-	-	12	-	-	6	-	-	4	-	-	-	-	1	-
bell5_c	220	-	1	-	-	1	-	-	1	-	-	1	-	-	-	-	-	-
dcmulti_c	240	-	23	100	-	15	105	-	17	100	-	8	47	4	18	-	-	67
dsbmip_c	20	-	1	-	-	1	-	-	-	-	-	-	-	2	-	-	-	-
gt2_c	380	1	-	-	1	-	-	1	-	-	-	-	-	-	-	1	-	-
mitre_c	220	-	3	-	-	3	-	-	3	-	1	1	-	-	-	-	2	-
misc03_c	320	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	1	-
p0201_c	300	-	-	-	-	-	-	-	-	-	1	3	-	-	-	-	-	-
stein27_c	240	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-

of RSP3 on `stein27_c`.

For the MITT_C instances, the global comparison for the Gomory cut generators is given in Table 7. The raw numbers indicate a superior performance for the G and GN generators, with only G2P4 coming close. However, a closer look at the repartition of the failures of types 1 and 2 listed in Table 8 shows that, for the variants of the G2 generator, most of the failures occur on `bienst1_c`. Removing this instance makes G2P4 and G2P5 comparable to G and GN. What happens on `bienst1_c` for most of the failures is that after adding valid cuts (keeping the 0-feasible solution under consideration ϵ -feasible for the LP) and resolving, one of the rows of the optimal tableau returned by the solver is not satisfied within $\epsilon = 10^{-6}$. The G and GN generators do not suffer this type of failures, as they recompute the optimal tableau from the basis information. In other words, a lack of precision in the solver optimal tableau causes the failures. Solving this instance with the variants of the G2 generator on a different machine (32-bits, with `lapack` and `blas` available to the solver) results in no failures of type 2. The `bienst1_c` instance has 505 variables, 28 of them binary, 576 constraints with a maximum absolute value of 15 for right-hand side, dynamism of 81, and a maximum absolute value of 51 for the entries of the 0-feasible solutions used.

Let us also mention that this lack of precision in the optimal tableau information the LP solver provides is not a particular weakness of `Clp`. In preliminary tests, we also tested a commercial solver and this type of failure was much more common than with `Clp`, at least when the LP solver is interfaced using the Open Solver Interface (`Osi`) of `COIN-OR`.

A second instance, `swath1_c`, creates some problems for all solvers. The source of the problem here is not apparent: This instance has 6,805 variables, 2,306 of them binary, 884 constraints with a maximum absolute value of about 19 for right-hand side, dynamism of 1,100, and a maximum absolute value of 404 for the entries of the 0-feasible solutions used. Note also that the six failures occurring for each algorithm never occur for the same 0-feasible solution. (This observation holds for all instances and all failures of types 1 or 2: when many failures occur, they are spread out over several solutions.) All algorithms have trouble with the same solutions, G and GN failing once on solution 2 and four times on solution 4, G2P3 failing twice on solutions 1 and 2 and four times on solution 4, all the other generators failing twice on solutions 1, 2, and 4.

Let us now turn to the six variants of the RS generator. Table 9 shows that the variants RSP4 and RSP5 are clearly safer than the others. However, most of the failures of type 2 occur on `bienst1_c` for reasons explained above. Overall the results are similar to the results

Table 7: Gomory cut generators comparison on MITT_C instances.

name	Flag			
	0	1	2	3
G	1,135	5	0	480
GN	1,077	5	1	537
G2	1,058	11	185	366
G2P1	1,060	9	158	393
G2P2	1,062	11	183	364
G2P3	1,052	9	117	442
G2P4	1,260	11	72	277
G2P5	1,089	6	25	500

Table 8: Distribution of failures for the Gomory cut generators on the MITT_C instances.

name	trials	G			GN			G2			G2P1		
		1	2	3	1	2	3	1	2	3	1	2	3
bienst1_c	260	-	-	238	-	-	256	5	181	74	3	155	102
ip_c	120	-	-	-	-	-	-	-	2	-	-	2	-
neos2_c	80	-	-	55	-	1	76	-	2	70	-	1	75
neos3_c	20	-	-	20	-	-	20	-	-	20	-	-	19
prod1_c	220	-	-	167	-	-	185	-	-	202	-	-	197
swath1_c	180	5	-	-	5	-	-	6	-	-	6	-	-

name	trials	G2P2			G2P3			G2P4			G2P5		
		1	2	3	1	2	3	1	2	3	1	2	3
bienst1_c	260	5	181	74	1	116	143	5	72	130	-	24	221
ip_c	120	-	2	-	-	1	-	-	-	-	-	1	-
neos2_c	80	-	-	70	-	-	74	-	-	79	-	-	70
neos3_c	20	-	-	16	-	-	19	-	-	20	-	-	19
prod1_c	220	-	-	204	-	-	206	-	-	48	-	-	190
swath1_c	180	6	-	-	8	-	-	6	-	-	6	-	-

obtained by the six corresponding variants of the G2 generator.

Let us denote by MITT_C_NOB the collection of MITT_C instances minus the instance `bienst1_c`. Table 11 aggregates the results of the G, GN, G2P4, G2P5, RSP4 and RSP5 generators on the union of the MIPLIB3_C and MITT_C_NOB instances. It support the claim that the six listed generators have more or less the same accuracy after excluding the instance `bienst1_c`. The other generators fail significantly more frequently. Note that statistical tests (such as the non-parametric Wilcoxon signed-rank test [8, 36]) could also be applied to confirm that the six selected generators have similar failure rates, but this seems unnecessary if one consider only failures with `flag = 1` and `flag = 2`. The case of experiments ending with `flag = 3`

Table 9: Reduce-and-Split cut generators comparison on MITT_C instances.

name	Flag			
	0	1	2	3
RS	1,135	8	67	410
RSP1	1,130	8	56	426
RSP2	1,123	8	64	425
RSP3	1,174	9	79	358
RSP4	1,360	9	4	247
RSP5	1,135	8	12	465

Table 10: Distribution of failures for the Reduce-and-Split cut generators on the MITT_C instances.

name	trials	RS			RSP1			RSP2			RSP3			RSP4			RSP5		
		1	2	3	1	2	3	1	2	3	1	2	3	1	2	3	1	2	3
bienst1_c	260	-	64	195	-	51	209	-	58	202	-	77	183	1	4	237	-	11	241
ip_c	120	-	2	-	-	2	-	-	2	-	-	-	-	-	-	-	-	-	-
neos2_c	80	-	1	6	-	2	6	-	3	4	-	1	2	-	-	7	-	1	13
neos3_c	20	-	-	2	-	1	5	-	1	6	-	1	2	-	-	3	-	-	2
prod1_c	220	-	-	207	-	-	206	-	-	213	-	-	171	-	-	-	-	-	209
swath1_c	180	7	-	-	7	-	-	7	-	-	8	-	-	7	-	-	7	-	-
swath3_c	120	1	-	-	1	-	-	1	-	-	1	-	-	1	-	-	1	-	-

is more difficult to deal with, since they do not correspond to precision failures, but could potentially lead to failures of type 1 or 2 if left running longer. In this paper, we simply ignore them and treat the corresponding experiments as if they were missing. If a real test of generators is done using the method, setting the time limit to a high enough value so that very few experiments end with `flag = 3` occur might be wise.

Table 11: Failures for six cut generators on the union of the MIPLIB3_C and MITT_C_NOB instances.

name	Flag			
	0	1	2	3
G	6,445	6	6	403
GN	6,392	8	6	454
G2P4	6,680	6	11	163
G2P5	6,530	6	11	313
RSP4	6,820	8	4	28
RSP5	6,555	9	5	291

4.2 Comparing strength

While Algorithm 1 is designed to test the accuracy of a generator, it is possible to get information about the strength of the generated cuts by performing statistical tests on the number of variables set to integer values in step 2.4 in each trial. The statistical test commonly used when comparing performances of two algorithms is a t -test. This is a test to decide if the means of two normally distributed populations are equal. It provides an estimate of the probability to obtain a result at least as extreme than the one observed, assuming that the two populations have the same mean. The normality assumption can be relaxed when the number of observations is large enough (typically, more than 20 observations) or non-parametric tests (Wilcoxon test, for example) can be used. We refer the reader to [8, 36] for a detailed discussion of alternative tests.

However, when comparing more than two algorithms, pairwise comparisons using t -tests can lead to inaccurate and inconsistent results. To address this problem, other tests looking at the whole set of algorithms at once are to be preferred. In this paper, we use Tukey’s Honest Significant Differences test (*THSD test*). Both the t -test and THSD test are based on Analysis of Variance (*ANOVA*). The purpose of ANOVA is to attribute the variation observed in the response variable (in our case, the number of integer variables set to integer values in step 2.4) to the different factors of the experiment (in our case, the cut generator, the instance, and the solution used). We hope to see that a significant portion of the variation is attributed to the algorithm, meaning that it is unlikely that all tested algorithms have similar average strength. The output of a THSD test for k algorithms is a list of $\frac{k(k-1)}{2}$ confidence intervals for the difference of the performances of each pair of algorithms and the assessed probability to obtain the observed results if the algorithms have the same mean performance.

The statistical design used for our application is a two-way factorial design with three factors: “algorithm” (denoted by **f1** in the tables below, “instance” (**f2**), and “solution” (**f3**). The factor “solution” is embedded in the factor “instance”, and the factor “algorithm” is crossed with “instance/solution”. For each value of “algorithm”, “instance”, and “solution”, we have 20 observations for the number of variables fixed to integer values. The observations for runs that fail are removed. Hence, if none of the observations results in a failure, we have a balanced design. Otherwise, assuming that only a low percentage of runs end with a failure, the design is slightly unbalanced. This is supported by the tables listed in Section 4.1. Both ANOVA and THSD might give misleading results with unbalanced designs, but can handle slightly unbalanced designs. Moreover, we are mostly interested in the effect associated with the factor **f1** (i.e., the effect associated with using different algorithms), and the ANOVA computations can be trusted for the main factor, even with unbalanced designs.

The results below are obtained using the statistical package R [35] version 2.7.2 (2008-08-25). Based on the results of the previous section, the six algorithms **G**, **GN**, **G2P4**, **G2P5**, **RSP4** and **RSP5** are compared. Table 12 gives the ANOVA results. The row “**f1**” gives information about the effect associated with the factor “algorithm”: the number of degree of freedom of the F -statistics (“Df”), a measure of the variance in the results associated with **f1** (“Sum Sq”) and its average value (“Mean Sq”), the value of the F -statistics (“ F value”), and the probability to observe a value larger than the F value assuming that all algorithms have the same average performance (“Pr(> F)”). A code appears next to that last value, with “***” meaning that the value is lower than 0.001, allowing an easy identification of significant

factors. The other codes are listed below the table. The effect associated with a factor is significant at the α level if the value in the last column is smaller than $1 - \alpha$. One can see that the effect associated with the factor **f1** (“algorithm”) is significant with more than 95% confidence. The row **f2** can be interpreted similarly for the factor “instance”, the row **f2:f3** (resp. **f1:f2**, **f1:f2:f3**) is for the effect associated with a pair “instance/solution” (resp. a pair “algorithm/instance”, a triple “algorithm/instance/solution”). Finally, the row **Residuals** gives the variance left unexplained.

Table 12: ANOVA results for generators **G**, **GN**, **G2P4**, **G2P5**, **RSP4** and **RSP5** on the **MIPLIB3_C** instances.

	Df	Sum Sq	Mean Sq	<i>F</i> value	Pr(> <i>F</i>)	
f1	5	202175	40435	1423.9016	$< 2.2 \cdot 10^{-16}$	***
f2	31	26711291	861655	30342.8362	$< 2.2 \cdot 10^{-16}$	***
f2:f3	243	317521	1307	46.0139	$< 2.2 \cdot 10^{-16}$	***
f1:f2	155	494726	3192	112.3973	$< 2.2 \cdot 10^{-16}$	***
f1:f2:f3	1215	40600	33	1.1767	$2.589 \cdot 10^{-5}$	***
Residuals	30836	875659	28			

Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Table 13: THSD results for generators **G**, **GN**, **G2P4**, **G2P5**, **RSP4** and **RSP5** on the **MIPLIB3_C** instances.

	G	GN	G2P4	G2P5	RSP4	RSP5
G	·	·	—	—	—	—
GN	·	·	—	—	—	—
G2P4	+	+	·	—	—	—
G2P5	+	+	+	·	—	—
RSP4	+	+	+	+	·	·
RSP5	+	+	+	+	·	·

The results of the THSD test are given in Table 13. A “+” (resp. “−”) entry in row *A* and column *B* means that algorithm *A* required more (resp. less) variables to be fixed than algorithm *B* with a significance threshold of 95%. A (“.”) entry means that no conclusion can be drawn from the results.

A total order can almost be derived from Table 13: **G** and **GN** have similar strengths, but both are superior to **G2P4** which is superior to **G2P5** which is superior to **RSP4** and **RSP5**, the latter two having similar strengths.

The discussion in Section 4.2 shows that the algorithms have similar reliability on the **MITT_C_NOB** instances. The comparison of strength below is thus made on these instances. Tables 14 and 15 give these results.

Table 14: ANOVA results for generators **G**, **GN**, **G2P4**, **G2P5**, **RSP4**, and **RSP5** on the **MITT_C_NOB** instances.

	Df	Sum Sq	Mean Sq	<i>F</i> value	Pr(> <i>F</i>)	
f1	5	98988	19798	432.5540	$< 2 \cdot 10^{-16}$	***
f2	8	1431133	178892	3908.5782	$< 2 \cdot 10^{-16}$	***
f2:f3	59	13183	223	4.8817	$< 2 \cdot 10^{-16}$	***
f1:f2	37	30777	832	18.1739	$< 2 \cdot 10^{-16}$	***
f1:f2:f3	280	14211	51	1.1089	0.1070	
Residuals	6546	299604	46			

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Table 15: THSD results for generators **G**, **GN**, **G2P4**, **G2P5**, **RSP4**, and **RSP5** on the **MITT_C_NOB** instances.

	G	GN	G2P4	G2P5	RSP4	RSP5
G	.	+	−	+	−	−
GN	−	.	−	.	−	−
G2P4	+	+	.	+	−	.
G2P5	−	.	−	.	−	−
RSP4	+	+	+	+	.	+
RSP5	+	+	.	+	−	.

Table 15 indicates that **GN** and **G2P5** have similar strengths and are superior to **G** which is superior to **G2P4** and **RSP5**, these two having similar strengths and being superior to **RSP4**.

To illustrate the difference that the sample instances make, let us now look at the results for the analysis on the union of the **MIPLIB3_C** instances and the **MITT_C_NOB** instances. These are given in Tables 16 and 17. One can see that overall **G** and **GN** are comparable and superior to **G2P5**, **G2P4**, **RSP5**, **RSP4** in that order.

The conclusions reached on the strength of the generators depend on the sample instances, as illustrated by the slightly different rankings obtained on the **MIPLIB3_C**, **MITT_C_NOB** instances and their union. Note that while **G** is a tiny bit safer than **GN** and both have identical strengths on the **MIPLIB3_C** instances, **GN** turns out stronger on the **MITT_C_NOB** instances, but this is not apparent on the union of the two instance sets. However, since **G** is overall significantly slower than **GN**, **GN** seems to be a better generator than **G** on these instances.

As a side remark, note that although ANOVA finds a significant effect for the “instance”/“solution” pair, the ranking of the algorithms by the THSD test seems to be quite stable across solutions. For example, repeating the above analysis using a single 0-feasible solution (the one found by Algorithm 2) for each instance yields rankings identical to those listed above.

Table 16: ANOVA results for generators G, GN, G2P4, G2P5, RSP4, and RSP5 on the union of MIPLIB3_C and MITT_C_NOB instances.

	Df	Sum Sq	Mean Sq	<i>F</i> value	Pr(> <i>F</i>)	
f1	5	246112	49222	466.7201	$< 2 \cdot 10^{-16}$	***
f2	31	24574728	792733	7516.5840	$< 2 \cdot 10^{-16}$	***
f2:f3	274	1060311	3870	36.6924	$< 2 \cdot 10^{-16}$	***
f1:f2	155	702060	4529	42.9473	$< 2 \cdot 10^{-16}$	***
f1:f2:f3	1367	131360	96	0.9111	0.9904	
Residuals	37589	3964307	105			

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Table 17: THSD results for generators G, GN, G2P4, G2P5, RSP4, and RSP5 on the union of MIPLIB3_C and MITT_C_NOB instances.

	G	GN	G2P4	G2P5	RSP4	RSP5
G	.	.	—	—	—	—
GN	.	.	—	—	—	—
G2P4	+	+	.	.	—	—
G2P5	+	+	.	.	—	—
RSP4	+	+	+	+	.	+
RSP5	+	+	+	+	—	.

5 Conclusions

Comparing cut generators for MILP is not an easy matter. One might want to compare speed of generation, speed of reoptimization after adding a round of cuts, or strength of the generated cuts. Testing for strength, in particular, is difficult. The contention of this paper is that comparing strength of cut generators without a sense of how accurate the generators are is not very informative. While one could try to devise a method to test simultaneously accuracy and strength, we propose here to first assess the accuracy of a cut generator and then compare the strength of cut generators that have similar accuracy.

The proposed method, random diving towards a feasible solution, has the attractive feature that its results depend only on the cut generator and the precision of the LP solver. While the latter dependency introduces a source of numerical error independent from the generator, it makes sense to test cut generators and LP solver precision together as numerical error in the LP solver are usually caused by the generated cuts. Possibly weakening the strength of generated cuts in order to avoid numerical difficulties in the LP solver computations should be part of fine tuning a cut generator. The dependency of the proposed method on algorithmic parts outside the cut generator is far smaller than in any other test that we are aware of. As is usual when testing numerical precision of algorithms, the results might

also depend on the machine and compiler used in the tests. The contribution of this paper is thus more the testing method than the ranking of the generators obtained in Section 4. The dependency of the ranking obtained on the choice of the sample instances unfortunately prevents to draw conclusions on the relative strength of families of cuts in general. This weakness of the proposed method does not seem easy to remove.

Another interesting feature of the method is that analyzing the results raises many interesting questions directly related to improving the performance of a cut generator. For example, studying why failures occur on an apparently innocuous instance such as `gt2_c` might suggest new ways to prevent the generation of invalid cuts. Investigating how aggressive one can be with the parameter setting, yet keeping a low probability of generating invalid cuts, is a question with important practical implications, in particular if this can be linked to properties of the instance. The method is well-suited to explore such questions.

The proposed method can be seen as a starting point to develop better testing procedures. Many modifications of Algorithm 1 could lead to faster testing or more accurate results. For example, instead of fixing variables as in step 2.5, one could only change the lower or upper bound on y_j to $\lceil \bar{y}_j \rceil$ or $\lfloor \bar{y}_j \rfloor$ respectively. When the results of the tests are used to tune cut generators for their use in a specific branch-and-cut code, it is also possible to replace the random choice in step 2.4 by a choice closer to the choice that the code will make or exploring a subtree “centered” around the solution at hand. Adding cut management in Algorithm 1 similar to what is done in the branch-and-cut code is another important potential improvement, as it could speed up the testing at the cost of having results dependant on the particular choice of cut management.

Similarly, potential improvements for the generation of the benchmark instances are possible. For example, the 0-feasible solutions produced by Algorithm 2 are likely not the optimal 6-digits rounded solution of the corresponding instance. This is not crucial for the experiments in this paper, but might be an issue under different circumstances. Developing an algorithm for producing benchmark instances with all coefficients rounded to d digits and corresponding optimal d -digits rounded solutions could be useful.

Acknowledgements

We thank the referees for many valuable comments and suggestions.

References

- [1] Achterberg T., Koch T., Martin A., “MIPLIB 2003”, *Operations Research Letters* 34 (2006), 361–372.
- [2] Amini M.M., Barr R.S, “Network Reoptimization Algorithms: A Statistical Design Comparison”, *ORSA Journal on Computing* 5 (1993), 395–408.
- [3] Amini M.M., Racer M., “A Rigorous Computational Comparison of Alternative Solution Methods for the Generalized Assignment Problem”, *Management Science* 40 (1994), 868–890.
- [4] Anderson K., Cornuéjols G., Li Y., “Reduce-and-Split Cuts: Improving the Performance of Mixed Integer Gomory Cuts”, *Management Science* 51 (2005), 1720–1732.

- [5] Applegate D.L., Cook W., Dash S., Espinoza D.G., “Exact Solutions to Linear Programming Problems”, *Operations Research Letters* 35 (2007), 693–699.
- [6] Balas E., “Disjunctive Programming: Cutting Planes from Logical Conditions”, in: Mangasarian O.L. et al., eds., *Nonlinear Programming*, Vol. 2, Academic Press, New York (1975) 279–312.
- [7] Bixby R.E., Ceria S., McZeal C.M., Savelsbergh M.W.P, MIPLIB 3.0, <http://www.caam.rice.edu/~bixby/miplib/miplib.html>.
- [8] Cohen P.R., *Empirical Methods for Artificial Intelligence*, MIT Press (1995).
- [9] COIN-OR, <http://www.coin-or.org>.
- [10] Cook W., Dash S., Fukasawa R., Goycoolea M., “Numerically Safe Gomory Mixed-Integer Cuts”, Working Paper (2008).
- [11] Danna E., Rothberg E., Le Pape C., “Exploring Relaxation Induced Neighborhoods to Improve MIP Solutions”, *Mathematical Programming* 102 (2005), 71–90.
- [12] Dhiflaoui M., Funke S., Kwappik C., Mehlhorn K., Seel M., Schömer E., Schulte R., Weber D., “Certifying and Repairing Solutions to Large LPs. How Good are LP-solvers?”, Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms (Baltimore, MD, 2003), 255–256, ACM, New York, (2003).
- [13] Fischetti M., Lodi A., “Local branching”, *Mathematical Programming* 98 (2003), no. 1-3, Ser. B, 23–47.
- [14] Gomory R., “An Algorithm for the Mixed Integer Problem”, Technical Report RM-2597, The RAND Corporation (1960).
- [15] Hoaglin D.C., Klema V.C., Peters S.C., “Exploratory Data Analysis in a Study on the Performance of Nonlinear Optimization Routines”, *ACM Transactions on Mathematical Software* 8 (1982), 145–162.
- [16] Hooker J.N., “Needed: An Empirical Science of Algorithms”, *Operations Research* 42 (1994), 201–212.
- [17] Hooker J.N., “Testing Heuristics: We Have It All Wrong”, *Journal of Heuristics* 1 (1995), 33–42.
- [18] *ILOG CPLEX 10.1 User’s Manual*, (2006).
- [19] Jeroslow R., “Cutting Plane Theory: Disjunctive Methods”, *Annals of Discrete Mathematics* 1 (1972) 293–330.
- [20] Koch T., “The Final Netlib Results”, *Operations Research Letters* 32 (2004), 138–142.
- [21] Lin B.W., Rardin R.L., “Controlled Experimental Design for Statistical Comparison of Integer Programming Algorithms”, *Management Science* 25 (1979), 1258–1271.

- [22] Marchand H., Martin A., Weismantel R., Wolsey L., “Cutting Planes in Integer and Mixed Integer Programming”, Workshop on Discrete Optimization, DO’99 (Piscataway, NJ), *Discrete Appl. Math.* 123 (2002), 397–446.
- [23] Margot F., “BAC : A BCP Based Branch-and-cut Example”, IBM Research Report RC22799 (W0305-064) (2003), revised August 2008.
- [24] Margot F., “Testing Cut Generators for MILP”, *Optima* 77 (2008), 6–9.
- [25] <http://wpweb2.tepper.cmu.edu/fmargot/>.
- [26] McGeoch C.C., “Toward an Experimental Method for Algorithm Simulation”, *INFORMS Journal on Computing* 8 (1996), 1–15.
- [27] McGeoch C.C., “Experimental Analysis of Algorithms”, *Notices of the American Mathematical Association* 48 (2001), 304–311.
- [28] McGeoch C.C., “Experimental Analysis of Optimization Algorithms”, *Handbook of Applied Optimization*, Oxford University Press (2002), 1044–1052.
- [29] McGeoch C.C., “Experimental Analysis of Algorithms”, *Handbook of Global Optimization*, Vol. 2, Kluwer (2002), 489–513.
- [30] McGeoch C.C., Sanders P., Fleischer R., Cohen P.R., Precup D., “Using Finite Experiments to Study Asymptotic Performances”, in *Experimental Algorithmics: From Algorithm Design to Robust and Efficient Software*, Fleischer et al., eds., *Lecture Notes in Computer Science* 2547 (2002), 93–126.
- [31] Mittelman H., <http://plato.asu.edu/topics/testcases.html>, no longer available.
- [32] Nance R.E., Moose R.L., Foutz R.V., “A Statistical Technique for Comparing Heuristics: An Example from Capacity Assignment Strategies in Computer Network Design”, *Communications of the ACM* 30 (1987), 430–442.
- [33] Nemhauser G.L., Wolsey L.A., “A Recursive Procedure to Generate all Cuts for 0-1 Mixed Integer Programs”, *Mathematical Programming* 46 (1990) 379–390.
- [34] Neumaier A., Shscherbina O., “Safe Bounds in Linear and Mixed-Integer Linear Programming”, *Mathematical Programming* 99 (2004), 283–296.
- [35] R statistical software, <http://www.r-project.org/>.
- [36] Sheskin D.J., *Parametric and Nonparametric Statistical Procedures*, 2nd Ed., Chapman & Hall/CRC (2000).
- [37] Stuart G.W., *Matrix Algorithms, Vol. I: Basic Decompositions*, SIAM (1998).