Carnegie Mellon University
Research Showcase @ CMU

Tepper School of Business

5-2012

On the safety of Gomory cut generators

Gérard Cornuéjols
Carnegie Mellon University, gc0v@andrew.cmu.edu

François Margot
Carnegie Mellon University, fmargot@andrew.cmu.edu

Giacomo Nannicini
Singapore University of Technology and Design

Follow this and additional works at: http://repository.cmu.edu/tepper

Recommended Citation
.

This Working Paper is brought to you for free and open access by Research Showcase @ CMU. It has been accepted for inclusion in Tepper School of Business by an authorized administrator of Research Showcase @ CMU. For more information, please contact research-showcase@andrew.cmu.edu.

On the safety of Gomory cut generators

Gérard Cornuéjols[1], François Margot[1], Giacomo Nannicini[2]

[1] *Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA*
Email: {gc0v,fmargot}@andrew.cmu.edu

[2] *Singapore University of Technology and Design, Singapore, and*
Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA
Email: nannicini@sutd.edu.sg

May 16, 2012

Abstract

Gomory mixed-integer cuts are one of the key components in Branch-and-Cut solvers for mixed-integer linear programs. The textbook formula for generating these cuts is not used directly in open-source and commercial software due to the limited numerical precision of the computations: Additional steps are performed to avoid the generation of invalid cuts. This paper studies the impact of some of these steps on the safety of Gomory mixed-integer cut generators. As the generation of invalid cuts is a relatively rare event, the experimental design for this study is particularly important. We propose an experimental setup that allows statistically significant comparisons of generators. We also propose a parameter optimization algorithm and use it to find a Gomory mixed-integer cut generator that is as safe as a benchmark cut generator from a commercial solver even though it generates many more cuts.

1 Introduction

Gomory Mixed-Integer (GMI) cuts [15] are one of the key components in Branch-and-Cut solvers for Mixed-Integer Linear Programs (MILPs) [5, 6]. The textbook formula for generating these cuts is simple enough, but due to the limited numerical precision of the computations, all open-source and commercial software use additional steps to avoid the generation of invalid cuts. This paper studies the usefulness and practical impact of these post-processing steps.

We perform statistical tests of several hypotheses related to these post-processing steps, in the context of a reasonable use of a GMI cut generator over a large enough and relevant set of instances. The use of the cut generator should be reasonable because we want to state properties that hold true in a practical Branch-and-Cut setting. The set of instances should be large enough so that we can draw statistically meaningful conclusions, and it should be relevant in the sense that it should contain the kind of instances that are routinely solved in real-world applications. The hypotheses that we want to test relate to the effectiveness of the post-processing steps that are typically applied by existing cut generators. In particular, we would like to identify which steps are beneficial, irrelevant or harmful towards generating safe cuts.

In Section 2 we describe the cut post-processing steps that we investigate. These steps are selected based on inspection of the open-source codes of COIN-OR [10] and `SCIP` [1], as well as discussion with developers. The steps are based on the fractionality of the basic integer variable used to generate the cut, "dynamism" of the generated cut, violation of the cut by the

current LP solution, support of the cut, zeroing-out of small coefficients, and relaxation of the right-hand side of the cut.

Numerical failures related to cut generation come in two flavors: generation of invalid cuts and difficulties in the LP reoptimization process. In this paper, we focus on the former. In Section 3 we propose a framework called Dive-and-Cut for the statistical analysis of cutting plane generators. Its basic idea is to generate a number of feasible solutions $S_{\mathcal{I}}$ for each test instance $\mathcal{I}$, and perform the following experiment several times: randomly fix a number of integer variables in instance $\mathcal{I}$ to get an instance $\mathcal{I}_F$ such that solutions in a nonempty subset S_F of $S_{\mathcal{I}}$ are feasible for $\mathcal{I}_F$, generate several rounds of cuts for $\mathcal{I}_F$ and report whether a solution in S_F violates at least one of the generated cuts. We show that Dive-and-Cut has significant advantages over the similar framework (called here RandomDives) of [22].

Our investigation focuses on two measures of performance of the generators, the failure rate and the rejection rate. Failure rate is measured using Dive-and-Cut, and rejection rate is based on the number of cuts that the post-processing steps removed. We argue that a good generator should have low failure and rejection rates. Having a low failure rate is an obvious goal, while having a low rejection rate allows to decouple cut generation from the cut selection process in an efficient Branch-and-Cut solver.

Section 4 presents the instances used in the tests and studies the impact of some parameters of Dive-and-Cut (number of rounds of cutting, number of experiments). We show that by increasing these two parameters we can increase the power of the statistical tests used to compare cut generators.

Section 5 reports on the variation of cut validity and cut rejection rate when modifying a single cut generation parameter. We find that steps using the fractionality of the basic integer variable, the violation of the cut by the LP solution, or zeroing-out small coefficients have a direct impact on the safety of the generated cuts.

We also find that the relaxation of the right-hand side should be done carefully. We observe a spike in invalid cuts when the right-hand side is relaxed by a constant close to the tolerance used to test if a value is integer.

This sets up the stage for Section 6, where we seek to "optimize" over all the parameters used in the post-processing step. Our goal is to obtain a GMI cut generator with the following characteristics: its failure rate should be the same as or lower than that of the GMI cut generator of a commercial solver (we chose Cplex as our benchmark) and its rejection rate should be lowest possible, subject to this constraint. We describe a black-box optimization algorithm to achieve this goal. Note that the strength of the cuts is not addressed here. Our philosophy is that solvers should pick good cuts in a second stage, among cuts that are deemed safe in a first stage. In this paper, we focus on the first stage, which is to reject unsafe cuts. The cut generator we consider has twelve parameters and optimizing over all of them simultaneously would require an excessive amount of CPU time. We thus first use regression techniques to identify a set of six most influential parameters over which the optimization is performed. The remaining parameters are considered afterwards. We are able to find GMI cut generators that are as safe as the Cplex cut generator and that have a rejection rate around 40%.

Section 7 validates the results of Section 6. We use a different set of test instances to compare five GMI cut generators obtained by our optimization algorithm to six cut generators from commercial or open-source solvers. The conclusions are that our generators are consistently safer than the cut generators (commercial or open-source) that have a similar rejection rate, and they reject much fewer cuts than the only generator (commercial) that has a similar safety.

1.1 Preliminaries

Consider a MILP in canonical form

$$\left.\begin{array}{rrcl} \min & c^\top x & & \\ & Ax & \geq & b \\ & x & \in & \mathbb{R}^n_+ \\ & x_j & \in & \mathbb{Z} \quad \text{for all } j \in N_I, \end{array}\right\} \tag{MILP}$$

where $c \in \mathbb{Q}^n$, $b \in \mathbb{Q}^m$, $A \in \mathbb{Q}^{m\times n}$ and $N_I \subset \{1,\dots,n\}$. Lower (resp. upper) bounds on x are denoted by x^L (resp. x^U) and are included in $Ax \geq b$. Rows of A are denoted by $a^i, i = 1,\dots,m$. For a positive integer k, we denote by $[k]$ the set $\{1,\dots,k\}$ and by $\mathbf{0}_k$ the all-zero k-vector. The nearest integer to $z \in \mathbb{R}$ is denoted by $\lfloor z \rceil$. (MILP) can be expressed in standard form by defining $\hat{A} = (A,\ -I)$, $\hat{c}^\top = (c^\top,\ \mathbf{0}_m{}^\top)$ and appending m variables to the vector x. We assume that the first n components of x are the original variables. Variables numbered $n+1$ to $n+m$ are called *surplus* variables. We thus obtain

$$\left.\begin{array}{rrcl} \min & \hat{c}^\top x & & \\ & \hat{A}x & = & b \\ & x & \in & \mathbb{R}^{n+m}_+ \\ & x_j & \in & \mathbb{Z} \quad \text{for all } j \in N_I. \end{array}\right\} \tag{MILP$_s$}$$

The LP relaxation of (MILP$_s$) is the linear program obtained by dropping the integrality constraints, and is denoted by (LP). Let $B \subset [n+m]$ be an optimal basis of (LP), and let $J = [n+m] \setminus B$ be the set of nonbasic columns. Let B_I, J_I and J_C be the sets of integer basic variables, integer nonbasic variables, and continuous nonbasic variables respectively. The Simplex tableau associated with B is given by

$$x_i = \bar{x}_i - \sum_{j\in J} \bar{a}_{ij} x_j \quad \forall i \in B. \tag{1}$$

Choose $i \in B_I$ such that $x_i \notin \mathbb{Z}$. Define $f_0 := \bar{x}_i - \lfloor \bar{x}_i \rfloor$ and $f_j := \bar{a}_{ij} - \lfloor \bar{a}_{ij} \rfloor$ for all $j \in J_I$. The GMI cut obtained from the row where x_i is basic is

$$\sum_{j\in J_I: f_j \leq f_0} f_j x_j + \sum_{j\in J_I: f_j > f_0} \frac{f_0(1-f_j)}{1-f_0} x_j + \sum_{j\in J_C: \bar{a}_{ij}\geq 0} \bar{a}_{ij} x_j - \sum_{j\in J_C: \bar{a}_{ij}<0} \frac{f_0 \bar{a}_{ij}}{1-f_0} x_j \geq f_0. \tag{2}$$

A cut of the form (2) is generated from the problem in standard form (MILP$_s$), but virtually all Branch-and-Cut codes require the cut to be expressed in the space of the original variables before adding it to (LP). Therefore, surplus variables with nonzero cut coefficients must be substituted by their expression in terms of the original variables. In the following, for simplicity the cutting plane will be written as

$$\sum_{j\in[n]} \alpha_j x_j \geq \alpha_0, \tag{3}$$

or, if we need its expression in the $n+m$ space, as

$$\sum_{j \in [n+m]} \hat{\alpha}_j x_j \geq \hat{\alpha}_0. \tag{4}$$

Software using finite precision arithmetic works with tolerances for constraint violation. For MILPs, a natural choice is to use a tolerance for considering a number to be integer (ϵ_{int}), a tolerance for absolute violation of a constraint (ϵ_{abs}), and a tolerance for relative violation of a constraint (ϵ_{rel}). Available commercial solvers typically use $\epsilon_{\text{rel}} = \infty$. Given nonnegative values for ϵ_{int}, ϵ_{abs}, and ϵ_{rel}, we say that a point x^* is *($\epsilon_{abs}, \epsilon_{rel}, \epsilon_{int}$)-feasible* for (MILP) if

(i) $\forall i \in N_I$, $x_i^* - \lfloor x_i^* \rceil \leq \epsilon_{\text{int}}$,

(ii) $\forall i \in [m]$, $b_i - a^i x^* \leq \epsilon_{\text{abs}}$,

(iii) $\forall i \in [m]$, $(b_i - a^i x^*)/\|a^i\|_2 \leq \epsilon_{\text{rel}}$, and $\exists x' : Ax' \geq b, x' \geq 0, \|x^* - x'\| \leq \epsilon_{\text{rel}}$.

The last part of point (iii) above requires that there exists a feasible point that is close to x^*. Its purpose is to allow slight constraint violations while guaranteeing that x^* is not too far away from the feasible region. The motivation for this condition is given in Appendix B.

A *cut generator* is an algorithm whose input is an optimal Simplex tableau for the LP relaxation of an MILP and whose output is a set of cuts. Post-processing steps are part of the cut generator. A cut generator can produce *failures* of the following types.

Type 1: A cutting plane that cuts off a known integral feasible solution is generated.

Type 2: The linear relaxation becomes infeasible after the addition of the generated cutting planes, but an integral feasible solution is known.

Type 3: A time limit for cut generation and LP resolve is hit.

Failures of Type 1 and 2 depend on the precision of the machine and of the computations. A Type 3 failure depends on the time limit that is set, and can be seen as less severe than failures of Type 1 and 2. However, if the time limit is sufficiently large, a Type 3 failure is still a major defect of the cut generator.

Ideally, no failure should occur when the cut generator is used in a Branch-and-Cut algorithm. However, practitioners in integer programming know that numerical problems are not uncommon, and if the choice of the cut generation parameters is not careful, failures of the three types listed above can happen [22]. To decrease the occurrence of failures, several cut post-processing steps have been devised empirically. Some of these steps modify the cut, while others are numerical checks that result in the acceptance or rejection of the cut. Rejected cuts are simply discarded.

1.2 Related work

A Branch-and-Bound solver that relies on rational arithmetic and thus is not affected by the numerical problems mentioned above, is described in [14]. A combination of the rational LP solver [4] with the Branch-and-Cut code `SCIP` [1] is in progress. The idea is to use finite precision arithmetic for the majority of the computations, and switch to (slower) rational arithmetic only for those operations that would invalidate optimality of the result if carried out in a non-exact

fashion (such as pruning based on dual bounds). More details can be found in [13]. Note however that the implementations described in [13, 14] do not include cutting planes.

The generation of numerically safe GMI cuts has been investigated in [23] when all the variables are bounded, and in [12] in general. The latter paper assumes that the validity of cutting planes is tested in rational arithmetic. The proposed cut generation procedure offers no guarantee that the generated cut will not cut off an integer solution if computations are performed with finite precision.

A first step in the direction of testing safety of cutting plane generators is represented by [22]. The paper proposes a methodology for comparing the strength of cut generators. Safety is a fundamental issue in this context, since only cut generators with a similar safety can be compared on equal ground with respect to strength (otherwise, the comparison would favor "unsafe" but more aggressive cut generators). [22] provides experimental evidence that existing cut generators in COIN-OR `Cgl` [10] may run into numerical problems on seemingly innocuous instances. The methodology proposed in [22] will be discussed in more detail in Section 3.

2 Cut generation and post-processing parameters

Generating GMI cuts using finite precision computations involves three basic nonegative parameters.

(i) `ZERO`: Any number $z \in \mathbb{R}$ such that $|z| \leq \texttt{ZERO}$ is replaced by zero;

(ii) `EPS`, `EPS_REL`: Any two numbers $z, w \in \mathbb{R}$ such that $|z - w| \leq \max\{\texttt{EPS}, \texttt{EPS_REL} \cdot \max\{|z|, |w|\}\}$ are considered to be equal.

The choice $\texttt{EPS_REL} = 0$ is common in practice.

Two broad classes of cut post-processing procedures are *cut modifications* (modifying the coefficients or the right-hand side of the cut) and *numerical checks* (performing checks on the cut in order to either accept or reject it). We now describe the post-processing procedures that we consider in this paper.

2.1 Cut modification

In a typical GMI cut generator, each cut computed by the Gomory formula (2) is modified by up to three procedures before being added to (LP).

(i) Coefficient Removal: First, small cut coefficients for surplus variables are removed. Then the cut is expressed in the original space. Finally, small cut coefficients are removed, possibly adjusting the right-hand side of the cut to ensure validity or to strengthen the cut.

(ii) Right-hand Side Relaxation: The right-hand side of the cutting plane is relaxed to generate a safer cut.

(iii) Scaling: The coefficients and right-hand side of the cut are scaled by a positive number.

Coefficient Removal is applied by all open-source cut generators in `Cgl` [10] and `SCIP` [1]. Right-hand Side Relaxation and Scaling procedures are not always employed. Scaling can be performed in various ways. For example, one can scale to obtain the largest cut coefficient

equal to 1 or scale to obtain integral cut coefficients. Note that SCALING affects the absolute violation of the cut at a point $\bar{x}$, i.e. the value of $\alpha_0 - \alpha\bar{x}$. Since the LP solver typically rescales cuts added to (LP), SCALING will not be part of our investigation. The first two of the above modification procedures require the following parameters.

(i) `EPS_ELIM`: For $j \in \{n+1, \ldots, n+m\}$, cut coefficients $\hat{\alpha}_j$ such that $|\hat{\alpha}_j| \leq$ `EPS_ELIM` are set to zero, without substituting the corresponding surplus variable with its expression in terms of the original variables;

(ii) `LUB`: For $j \in [n]$, a variable x_j with $x_j^L = \beta$ or $x_j^U = \beta$ for some β such that `LUB` $\leq |\beta| < \infty$ is considered having a large bound. Define L as the set of such variables;

(iii) `EPS_COEFF`: For $j \in [n] \setminus L$, cut coefficients α_j such that $|\alpha_j| \leq$ `EPS_COEFF`, are set to zero, adjusting the right-hand side of the cut as follows. If $\alpha_j > 0$ (resp. $\alpha_j < 0$), the right-hand side α_0 becomes $\alpha_0 - \alpha_j x_j^U$ (resp. $\alpha_0 - \alpha_j x_j^L$) unless $x_j^U = \infty$ (resp. $x_j^L = -\infty$), in which case the cut is discarded;

(iv) `EPS_COEFF_LUB`: For $j \in [n] \cap L$, cut coefficients α_j such that $|\alpha_j| \leq$ `EPS_COEFF_LUB` are set to zero and no adjustment of the right-hand side occurs; typically `EPS_COEFF_LUB` is much smaller than `EPS_COEFF`;

(v) `EPS_RELAX_ABS`: The cut right-hand side α_0 is relaxed to $\alpha_0 -$ `RELAX_RHS_ABS`;

(vi) `EPS_RELAX_REL`: The cut right-hand side α_0 is relaxed to $\alpha_0 - |\alpha_0| \cdot$ `RELAX_RHS_REL`.

2.2 Numerical checks

All generated cutting planes undergo a sequence of checks aimed at deciding whether or not they should be added to (LP). These checks test the numerical properties of the cuts, as well as their effectiveness. The *support* of a cut is the set of all variables whose coefficient is nonzero. In a typical cut generator, the following checks are performed.

(i) FRACTIONALITY CHECK: A cut is discarded (rather, not generated), if the value of the corresponding integer basic variable is too close to an integer value;

(ii) VIOLATION CHECK: A cut is discarded if it does not cut off the optimal solution to (LP) by at least a given amount;

(iii) SUPPORT CHECK: A cut is discarded if the cardinality of its support is too large;

(iv) DYNAMISM CHECK: A cut is discarded if the ratio between the largest and smallest nonzero absolute values of the coefficients is too large;

(v) SCALING CHECK: A cut is discarded if it is badly scaled.

The SCALING CHECK is not regulated by a single parameter. For example, a cut might be discarded if its ℓ_2-norm does not fall within a given range. For the reason given in Section 2.1, SCALING CHECK will not be part of our investigation. The other four checks require the following parameters. Let $\bar{x}$ denote the current basic solution as defined in (1).

(i) `AWAY`: The cut generated from the tableau row where x_i is basic is discarded if $|\bar{x}_i - \lfloor \bar{x}_i \rceil| <$ `AWAY`;

(ii) MIN_VIOL: The cut is discarded if $\alpha_0 - \sum_{j\in[n]} \alpha_j \bar{x}_j < \max\{1, |\alpha_0|\} \cdot$ MIN_VIOL;

(iii) MAX_SUPP_ABS, MAX_SUPP_REL: The cut is discarded if the support of α is larger than MAX_SUPP_ABS $+ n \cdot$ MAX_SUPP_REL;

(iv) MAX_DYN: The cut is discarded if $\max\{|\alpha_j| : j \in [n]\} >$ MAX_DYN$\cdot\min\{|\alpha_j| : |\alpha_j| > 0, j \in [n]\}$ and $L \cap \{j : |\alpha_j| > 0\} = \emptyset$;

(v) MAX_DYN_LUB: The cut is discarded if $\max\{|\alpha_j| : j \in [n]\} >$ MAX_DYN_LUB $\cdot \min\{|\alpha_j| : |\alpha_j| > 0, j \in [n]\}$ and $L \cap \{j : |\alpha_j| > 0\} \neq \emptyset$.

Observe that any of the cut modification and numerical check procedures can be disabled by setting the corresponding parameter to an appropriate value. For example, using MAX_SUPP_ABS $= n$ implies that no Support Check is performed.

3 Dive-and-Cut

In this section we propose a method for testing the safety of cut generators. It assumes that a set of problem instances is available and it involves preprocessing the instances. The goal of this preprocessing step is the generation of many feasible (or almost feasible) solutions.

Once the instances have been preprocessed, the testing phase for an instance amounts to diving randomly by fixing a number of integer variables, and then generating rounds of cuts.

This scheme is similar to the method RandomDives proposed in [22], in the sense that the work horse is a large number of random dives to be able to perform meaningful statistical tests. However, Dive-and-cut improves over RandomDives on the three criteria mentioned in Section 1 (reasonable use of the generator, large and relevant set of instances). In addition, Dive-and-Cut is usually faster than RandomDives. We discuss this further at the end of this section, after the presentation of Dive-and-Cut.

3.1 Instance preprocessing phase

Testing the generation of invalid cuts requires knowledge of feasible solutions. Branch-and-Cut solvers typically accept $(\epsilon_{\text{abs}}, \infty, \epsilon_{\text{int}})$-feasible solutions with positive and finite values for ϵ_{abs} and ϵ_{int}. Suppose that for problem (MILP) we have an $(\epsilon_{\text{abs}}, \infty, \epsilon_{\text{int}})$-feasible solution $\tilde{x}$, and there exists at least one row a^i such that $b_i - a^i\tilde{x} > 0$. Thus, we can find $\lambda \in \mathbb{R}^m_+$ such that $\lambda^\top(b - A\tilde{x}) > \epsilon'$ for arbitrary $\epsilon' > 0$. In other words, we can find a valid inequality $\alpha x \geq \alpha_0$ for the system $Ax \geq b$, with $\alpha = \lambda^\top A$ and $\alpha_0 = \lambda^\top b$, that is violated by $\tilde{x}$ by an arbitrary amount. It follows that we should be careful when choosing the solutions that are used for testing the validity of the cuts. On the one hand, using slightly infeasible solutions may lead to mislabeling cuts as invalid. On the other hand, commercial MILP solvers typically return slightly infeasible solutions that are often acceptable for practical purposes, therefore cutting off such a solution can reasonably be considered a failure of the cut generator.

We use algorithm GenerateSolutions given in Appendix B to generate $(\epsilon_{\text{abs}}, \epsilon_{\text{rel}}, 0)$-feasible solutions for an instance, with positive and finite values for ϵ_{abs} and ϵ_{rel}. It applies a Branch-and-Cut solver and acts whenever the solver discovers an integer solution. First, integer variables are set to integer values and this updated solution is checked for $(\epsilon_{\text{abs}}, \epsilon_{\text{int}}, 0)$-feasibility. If it satisfies both the absolute violation tolerance ϵ_{abs} and the first condition required for the relative violation tolerance ϵ_{rel}, a rational solver is used to find a feasible solution close to the

updated solution, and this solution is used to check whether the second condition needed for relative violation is also satisfied. Details are in Appendix B.

3.2 Testing phase

In this section we assume that for each instance in our test set, a collection of feasible solutions is available. These will be the solutions used to detect invalid cuts. A formal description of the method that we propose is given in Algorithm 1.

Algorithm 1 DIVE-AND-CUT.

INPUT: Problem $P = (A, b, c)$, set of solutions S, maximum gap threshold T, upper bound U, number of rounds ρ, tolerances $\epsilon_{\text{abs}}, \epsilon_{\text{rel}} \geq 0$, time limit for a dive
OUTPUT: Flag failure
Let $F \leftarrow \emptyset$
Randomly choose $x^* \in S$
Randomly choose $t \in [0, T]$
Compute $\bar{x} = \arg\min\{c^\top x | Ax \geq b, x \geq 0\}$
Initialize $\bar{x}' \leftarrow \bar{x}$
while $(c^\top(\bar{x}' - \bar{x}) < t \cdot (U - c^\top \bar{x}))$ **do**
  Randomly choose $j \in \{i \in N_I \setminus F\}$
  Append the constraint $x_j = x_j^*$ to (A, b)
  Let $F \leftarrow \{j\}$
  Compute $\bar{x}' = \arg\min\{c^\top x | Ax \geq b, x \geq 0\}$
Compute $S(x^*, F) = \{x \in S | x_i = x_i^* \ \forall i \in F\}$
for $1, \ldots, \rho$ **do**
  Generate cuts $\alpha^i x \geq \alpha_0^i, i = 1 \ldots, h$ and append to (A, b)
  Resolve $\min\{c^\top x | Ax \geq b, x \geq 0\}$
  if (time limit is hit) **then**
    return failure $\leftarrow 3$
  else if (LP is infeasible) **then**
    return failure $\leftarrow 2$
  else if $(\exists \tilde{x} \in S(x^*, F) : ((\max_{i \in [h]}\{\alpha_0^i - \alpha^i \tilde{x}\} > \epsilon_{\text{abs}}) \vee (\max_{i \in [h]}\{(\alpha_0^i - \alpha^i \tilde{x})/\|\alpha^i\|\} > \epsilon_{\text{rel}})))$
  then
    return failure $\leftarrow 1$
  Perform cut management
return failure $\leftarrow 0$

We call this algorithm DIVE-AND-CUT. It starts by diving towards a feasible solution x^* chosen at random among the available solutions. This is achieved by selecting uniformly at random a value t between 0 and T (we use $T = 80\%$ in our experiments) and fixing randomly chosen integer variables to their value in x^* until a fraction t of the initial gap is closed. The gap is computed with respect to a given upper bound U. In this paper we set U to the value of the best solution returned by GENERATESOLUTIONS (in the vast majority of cases, this is the same as the value of the best known solution for the instance, see Appendix B.2). This simulates the generation of a node of a hypothetical Branch-and-Cut tree. Once at this node, DIVE-AND-CUT generates ρ rounds of cuts and determines the outcome of the experiment. If we hit the time limit during cut generation, or the LP is infeasible, or we cut off any feasible

solution, the algorithm returns a failure. Otherwise, it returns that no failure occurred. Note that we only test against the feasible solutions in the set S that have the same value as x^* on the variables that have been fixed[1].

This method is designed to represent a reasonable use of a cut generator. In the majority of Branch-and-Cut solvers, cutting planes are mostly generated at the root node and cut management procedures are used. In Dive-and-Cut the node obtained after the dive mimics a root node. Dive-and-Cut involves several random decisions, therefore the algorithm can be applied as many times as required to obtain statistically significant results.

3.3 Comparison between Dive-and-Cut and RandomDives

RandomDives [22] is similar to Dive-and-Cut, but important differences exist. First, the preprocessing phases are quite different. For RandomDives, instances are modified by relaxing the right-hand side of some of the constraints, so that $(0, 0, 0)$-feasible solutions can be found and checked with finite precision computations. This changes the difficulty of the instances sometimes significantly. The preprocessing of Dive-and-Cut on the other hand does not modify the instance. This clearly helps Dive-and-Cut score higher on the relevancy criterion.

A second difference is that RandomDives dives towards a feasible solution, generating ρ rounds of cuts after each single variable fixing. This way of using the cut generator is very different from standard usage in efficient Branch-and-Cut solvers. The rationale advanced in [22] is that it puts a lot of stress on the generator, and thus safe generators according to RandomDives are likely to be safe when used in a Branch-and-Cut. However, it is likely that this experiment puts too much stress on the generator and LP solver.

Another problem with RandomDives is the rare occurrence of Type 1 failures. RandomDives works with a single feasible solution x^*. Cutting off that solution is a severe failure of the cut generator but it happens very rarely. It is clear from the results of [22] that RandomDives typically leads to failures of Type 2 and 3 much more often than to failures of Type 1. This does not match empirical observations, where cutting off the optimal solution is the most common failure and is relatively frequent on MIPLIB instances when an aggressive cutting strategy is used.

Appendix C contains a detailed comparison between the two procedures. Conclusions are that the safety rankings of generators obtained by the two procedures is similar, but the distribution of failures is quite different. RandomDives generates significantly more failures than Dive-and-Cut. The ratio of Type 1 to Type 2 failures is much larger for Dive-and-Cut while the overall number of Type 3 failures is smaller for Dive-and-Cut. This suggests that RandomDives puts a lot of stress on the LP solver, increasing the difficulty of resolving the LPs until the LP solver runs into numerical problems or times out. Finally, Dive-and-Cut is on average much faster than RandomDives.

4 Empirical testing: Preliminaries

In this section we describe the framework for the empirical testing of GMI cut generators conducted using Dive-and-Cut. We first analyze the effect of changing the number of rounds of cut generation or the number of dives.

[1]This is not necessary if the cuts being tested can be generated independently of the bounds on the variables, or are lifted to be globally valid.

4.1 Parameters and implementation

We list here the parameters and implementation features used throughout the computational experiments. `ZERO` is set to 10^{-20}, `EPS` and `EPS_REL` are set to 10^{-12}. A number α is considered integer valued if $|\alpha - \lfloor\alpha\rceil| \leq \max\{10^{-9}, 10^{-15} \cdot |\alpha|\}$. The absolute feasibility tolerance ϵ_{abs} is set to 10^{-9}, the relative feasibility tolerance ϵ_{rel} is set to 10^{-9}. The number of rounds of cut generation ρ is set to 30, unless otherwise stated (see discussion in Section 4.4).

Throughout our code except in the cut generator itself, the computation of all sums of a sequence of numbers (e.g., dot product, norm) is carried out with the compensated summation algorithm [19] to compute the left-hand side of inequalities. (Compensated summation ensures that the numerical error is independent of the number of additions.) In the GMI cut generator, compensated summation is not used, as it is not standard practice in commercial and open-source Branch-and-Cut solvers. The GMI cut generator recomputes the Simplex tableau from scratch, using the basis information, instead of obtaining it directly from the LP solver.

The algorithms discussed in this paper are implemented in C++ within the COIN-OR framework. We use several functions available in COIN-OR `Cbc` 2.7 [8]. The GMI cut generator is implemented as a `CglCutGenerator`, following the guidelines of `Cgl` [10]. The LP solver of choice is IBM ILOG `Cplex` 12.2 [18]. Dive-and-Cut's implementation can use COIN-OR `Clp` [9] instead of `Cplex`, exploiting the common interface `OsiSolverInterface`.

Due to the large amount of processing time required, the experiments were run in parallel on the Condor [21] grid at the University of Wisconsin-Madison, unless otherwise stated. All machines running Linux in the Condor pool were candidates for executing the experiments. For this reason, our code is compiled to run on a generic x86 architecture. Since we use different machines, some variation on the results of the computations across machines should be expected. A preliminary computational evaluation revealed that this is not a major problem, as the differences recorded by running the same experiment several times were not statistically significant.

4.2 Instance selection and cut management

In order to have a large and diverse set of instances to test the cut generators, we built an initial test set containing all instances from MIPLIB3 [7], MIPLIB2003 [2], and the Benchmark set of MIPLIB2010 [20] beta (downloaded March 2011) for a total of 169 instances.

For each instance in the set, we applied `Cplex`'s Branch-and-Cut and GenerateSolutions in order to generate the set of feasible solutions (see Appendix B). As GenerateSolutions fails to generate any feasible solution for ten of the instances, we are left with 159 instances.

Since running experiments on instances that do not generate failures is useless in the present context, we keep only instances for which a crude GMI cut generator called CgBase generates some failures. (Its parametrization can be found in Table 16 in Appendix C.) The test runs consist in applying Dive-and-Cut 200 times per instance with a time limit of 300 seconds per dive, and the tolerances described in Section 4.1. In addition, four cut management procedures are tested. We say that a generated cut is *inactive* in an optimal solution of (LP) if its dual variable has a value smaller than 10^{-5}. The four cut management procedures that we considered are to remove all cuts that are inactive for k consecutive rounds for $k = 1, 2, 3$ and $k = \infty$.

We also allow for an early stopping criterion: if more than 30 Type 3 failures are detected on an instance with a given cut management procedure, the execution of Dive-and-Cut is

stopped. The experiments are run in parallel on the Condor pool as discussed in Section 4.1. All instances on which no failures of Type 1 or 2 occurred are removed.

The test runs show that the number of failures decreases if we remove inactive cuts more aggressively. At the same time, CPU time decreases, which is expected. Experiments where cuts are never removed from the LP turn out to be very time-consuming, with a significant number of Type 3 failures. The total number of recorded failures for the four cut management procedures are given in Table 1.

	Failures			
k	T. 1	T. 2	T. 3	Tot.
1	270	80	187	537
2	403	360	308	1071
3	379	329	447	1155
∞	254	198	740	1192

Table 1: Number of failures on the full test set, minus instances on which no failures of Type 1 or 2 were recorded. This test set comprises 74 instances. The value of k is the number of consecutive rounds of inactivity after which cuts are removed from the LP.

It can be seen from Table 1 that there is a significant increase in the number of failures if we do not remove all inactive cuts immediately. At the same time, there is little difference between $k = 2, 3$ and ∞. Indeed, the total number of failure is relatively stable for these three values of k. The fact that the number of Type 1 and 2 failures decreases for $k = 3$ and ∞ can be explained by observing that some of the Type 3 failures might generate failures of Type 1 or 2, if given more time. Furthermore, more instances time out, hence we perform fewer dives overall because of the early stopping criterion. Since we are interested in producing a large number of failures as quickly as possible, we always use the cut management procedure with $k = 2$ in the remainder of the paper.

The final modification to the test set consists in removing the instances taking too much time with $k = 2$. We remove instances where more than 10 failures of Type 3 are recorded or such that 200 dives take more than 6 hours. We are left with a set of 51 instances that we call Failure Set, see Appendix B.2.

4.3 Statistical Tests

A brief survey of the statistical tests used in this paper is given in Appendix A. In the remainder of the paper, we will make statements of the form "we apply a Friedman test on the failure rate". This means that we use a Friedman test where "treatments" are the cut generators, each "block" is an instance, and the performance of the cut generator on an instance is the average failure rate over all dives on that instance. The null hypothesis is that the cut generators are indistinguishable in terms of their failure rate. All statistical tests will be performed with a significance level of 95%. We will often report the p-value. If $p < 0.05$, the null hypothesis is rejected.

Similarly, the statement "we apply a Friedman test on the rejection rate" means that we use a Friedman test where "treatments" are cut generators, each "block" is an instance, and the performance of the cut generator on an instance is the average cut rejection rate over all

dives on that instance.

4.4 Number of rounds and number of dives

The number ρ of cut generation rounds is one of the key decisions in cutting plane algorithms. In this section we show that this choice does not affect the conclusions that can be drawn by applying Dive-and-Cut, in the sense that increasing ρ simply increases the power of the statistical tests that we use but does not change the safety rankings of cut generators. The same is true for the number of dives; this will be discussed at the end of this section.

We start by providing some data on the distribution of failures for different values of ρ. The data is obtained by applying $h = 300$ dives of Dive-and-Cut on the 51 instances of the Failure Set with the cut generator CgBase (see Table 16 in Appendix C for its parameter specifications). For each type of failure, we record the number of occurrences in each round and, for each round r, we plot the point (r, f) on the graph in Figure 1 where f is the number of failures that occurred up to round r. Similar graphs are obtained with other cut generators, therefore we only report results for CgBase.

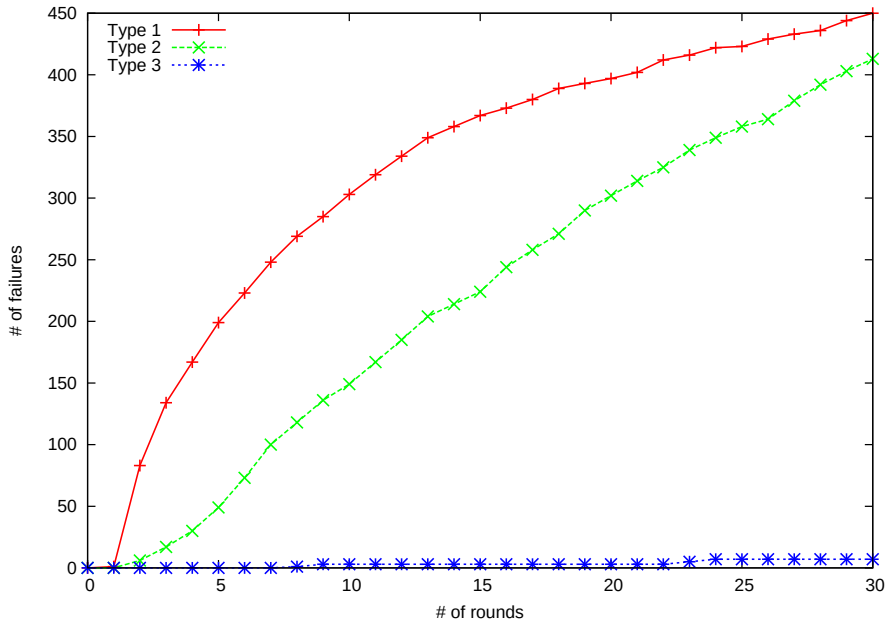

Figure 1: Number of failures depending on the value of ρ (x axis).

An interesting fact that can be observed in Figure 1 is that the number of Type 1 failures is a concave function of ρ and the number of failures of Type 2 and 3 increases almost linearly with ρ. This is surprising, as we expected very few failures in the first few rounds and a super-linear increase for larger values of ρ. The importance of this finding lies in the fact that we can increase the number of rounds to increase the number of failures. This helps in detecting differences between otherwise indistinguishable cut generators, without severely affecting the ranking of the generators.

We verified this claim by comparing the safety of a set of cut generator for $\rho = 5, 10, 20$, and 30 using a Friedman test on the failure rate. Some differences among the cut generators that are

detected for $\rho \geq 20$ are not detected for $\rho = 5, 10$, but the results of the pairwise comparisons for different values of ρ never give opposite results. For the sake of brevity we do not include detailed results.

A similar effect is observed by varying the number of dives h. The total number of failures recorded by the cut generator is essentially linear in h (see Figure 2). We verified that changing the number of dives does not change the ranking of a set of cut generators in terms of safety.

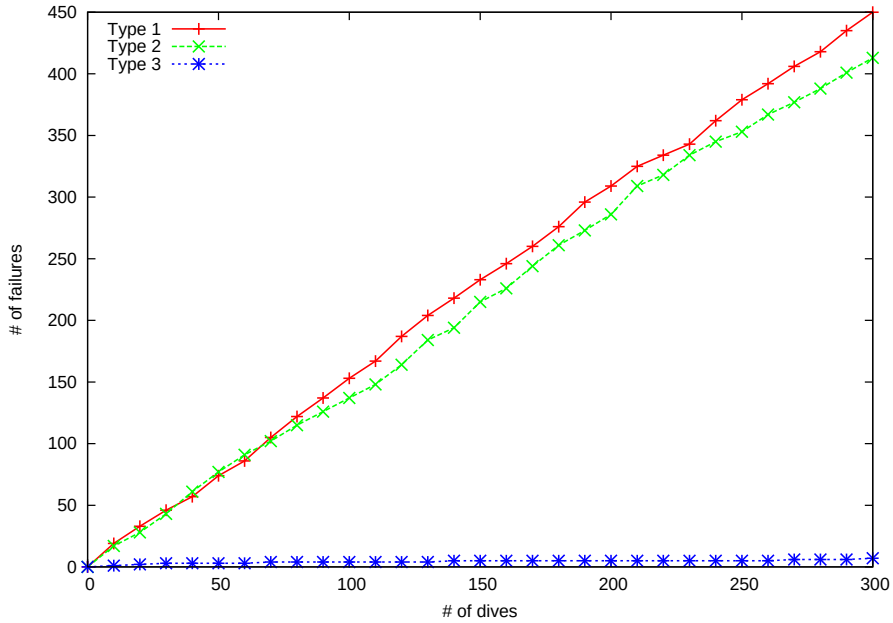

Figure 2: Number of failures depending on the number of dives (x axis).

To summarize, the number of rounds of cut generation and the number of dives have a direct influence on the detection power of our tests. By increasing those two parameters, we can magnify the differences between cut generators, at the expense of requiring more computing time.

5 Empirical testing: Parameter ranges

In this section we discuss one-at-a-time changes in the cut generation parameters. The base cut generator in this section is CgBase, described in Table 16 in Appendix C.

For each parameter listed in Sections 2.1 and 2.2, we perform several tests over the range of possible values, recording the number of failures and the average cut rejection rate. This allows us to observe how the occurrence of failures and cut rejection evolves when changing parameter values. This information is used in Section 6.3 to determine the initial parameter ranges for the optimization.

Through this section, we always perform 150 dives on each instance of Failure Set. The first dive out of 150 does not fix any variable (i.e., we generate cuts at the root node without diving), so that we can test against the full set of feasible solutions. In our experiments, the

first dive did not generate significantly more or significantly less failures than the remaining dives.

For the sake of brevity, instead of reporting extensive statistical tests for each parameter value, we simply graph the number of failures and cut rejection rate for a range of possible values of the parameter.

5.1 Variables with large bounds

The `LUB` parameter theoretically takes value in $[0, \infty]$. There are only 5 instances in Failure Set for which `LUB` $\geq 10^4$ yields a different set L of variables with a large bound compared to `LUB` $= 10^3$: `bell4a` ($|L|$ decreases from 92 to 62), `blend2` ($|L|$ decreases from 90 to 88), `maxgasflow` ($|L|$ decreases from 4920 to 4912) `noswot` ($|L|$ decreases from 53 to 28), `roll3000` ($|L|$ decreases from 244 to 6 for `LUB` $= 10^4$, 2 for `LUB` $> 10^4$). For the remaining instances, any value of `LUB` $\geq 10^3$ yields the same set of variables with a large bound.

The value of the `LUB` parameter affects both `MAX_DYN_LUB` and `EPS_COEFF_LUB` and these parameters are hard to decouple. Therefore we do not analyze them in this section where the focus is on one-at-a-time changes.

5.2 Numerical check parameters

We now analyze the effect varying separately each numerical check parameter given in Section 2.2. In each case, we plot the number of failures and the rejection rate.

5.2.1 Fractionality of the right-hand side

The `AWAY` parameter takes its value in $[0, 0.5]$. Since the integrality tolerance is 10^{-9}, we use a lower bound of 10^{-9} for `AWAY`. We tested the values `AWAY` $= 10^k$ for $k = -9, \ldots, -1$. Smaller values of `AWAY` lead to generating more cuts and more failures.

Figure 3 graphs the number of failures and percentage of rejected cuts depending on the value of `AWAY`.

Figure 3 shows that generating cuts with small `AWAY` is extremely risky and leads to many failures. Since a small value of `AWAY` allows for the generation of cuts with large dynamism, the generated unsafe cuts could possibly be discarded through Dynamism Check, but such interactions will only be considered later through the optimization algorithm of Section 6.2. By increasing `AWAY`, a much safer cut generator can be obtained, while still rejecting few cuts. The rejection rate starts increasing to non-negligible levels only for `AWAY` $> 10^{-5}$.

5.2.2 Dynamism

The `MAX_DYN` parameter takes value in $[0, \infty]$. We tested the values `MAX_DYN` $= 10^k$ for $k = 2, 4, \ldots, 30$.

Figure 4 shows that the cut rejection rate decreases at a low rate while k increase from 2 to 16, decreases sharply when k increases from 16 to 26, and is almost 0 when k is larger than 26. This implies that most of the generated cuts when `AWAY` $= 10^{-9}$ have very poor numerical properties. We suspect that this happens especially in later rounds. By relying on Dynamism Check only, halving the number of Type 1 and 2 failures comes at the cost of rejecting more than half of the generated cuts. If we accept only cuts with `MAX_DYN` $= 10^2$, only 10 failures are

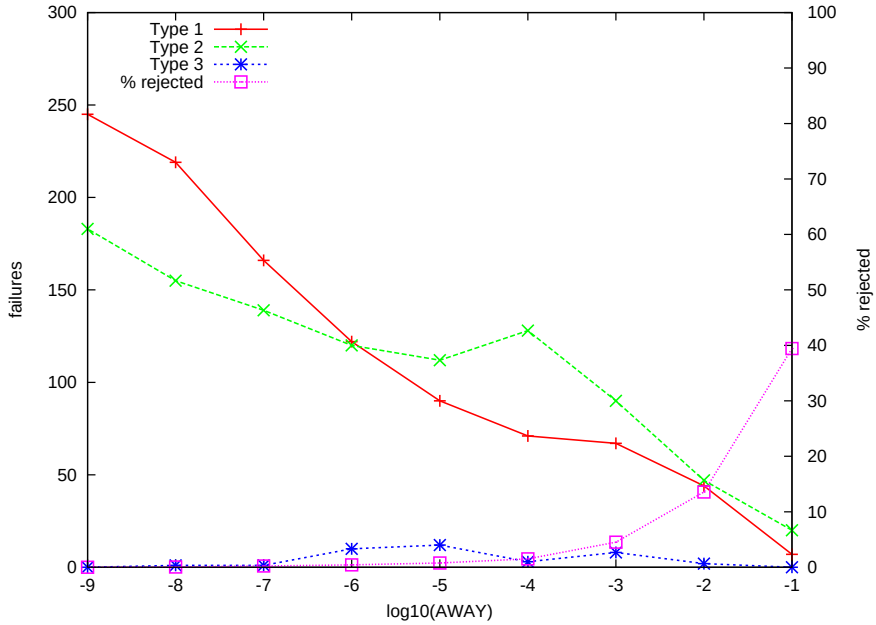

Figure 3: Number of failures and cut rejection rate for cut generators with different values of `AWAY`. `AWAY` is set to 10^k, where k is the value on the x axis.

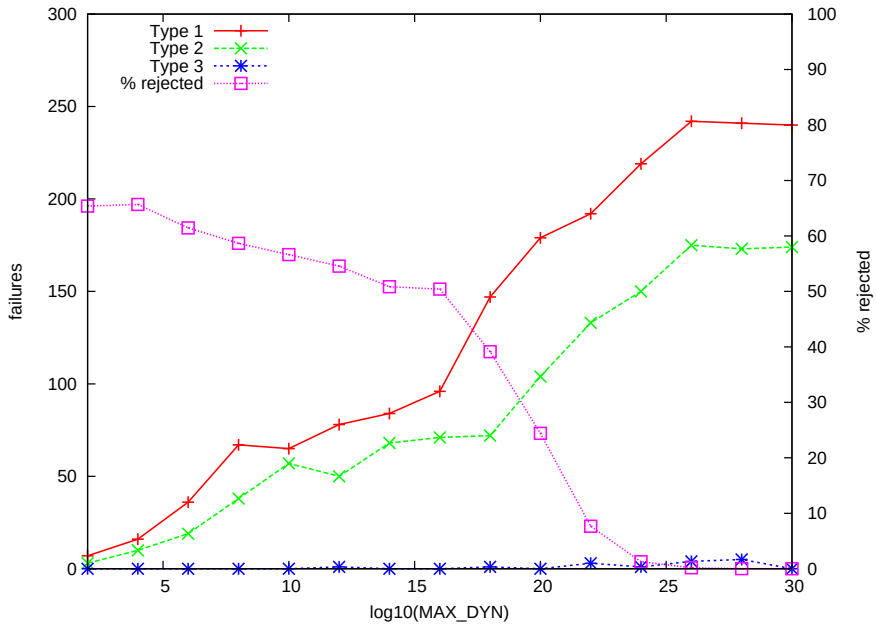

Figure 4: Number of failures and cut rejection rate for cut generators with different values of `MAX_DYN`. `MAX_DYN` is set to 10^k, where k is the value on the x axis.

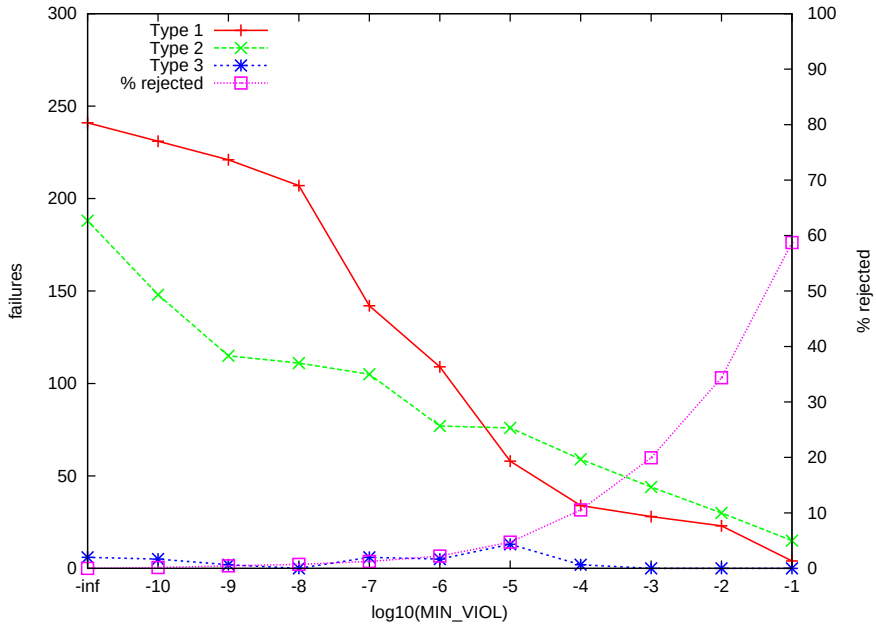

Figure 5: Number of failures and cut rejection rate for cut generators with different values of `MIN_VIOL`. `MIN_VIOL` is set to 10^k, where k is the value on the x axis.

recorded overall, but almost 2 cuts out of 3 are rejected. Interestingly, for $k < 22$ essentially no Type 3 failures are recorded. This suggests that adding cuts with large dynamism makes the LP solution process significantly more time-consuming.

5.2.3 Violation

The `MIN_VIOL` parameter theoretically takes its value in $[0, 1]$, but due to errors in the finite precision computations, applying Violation Check even with `MIN_VIOL` $= 0$ could reject some cutting planes. We tested the values `MIN_VIOL` $= 10^k, k = -10, \ldots, -1$, and `MIN_VIOL` $= 0$ (which is reported as $k = -$`inf` in the figure for sake of simplicity).

Figure 5 shows that even small values of `MIN_VIOL` are surprisingly effective in reducing the number of failures. The number of Type 2 failures can be reduced by 50% by rejecting less than 1% of the generated cuts. Reducing by 50% the number of Type 1 failures requires a further increase of `MIN_VIOL`, but can still be achieved by rejecting less than 2% of the cuts. For $k > -6$, the fraction of rejected cuts begins to rise sharply. An interesting observation is that contrary to Dynamism Check, Violation Check does not seem to have much effect on the number of Type 3 failures. A much larger rejection rate than for Dynamism Check is required to bring the number of Type 3 failures close to 0.

5.2.4 Maximum support

The maximum allowed support for the cuts is kept under control by two parameters: `MAX_SUPP_ABS` that takes value in $[0, \infty]$, and `MAX_SUPP_REL` that takes value in $[0, 1]$. The largest instance in Failure Set has 10,724 columns, therefore we can consider `MAX_SUPP_ABS` to take value in

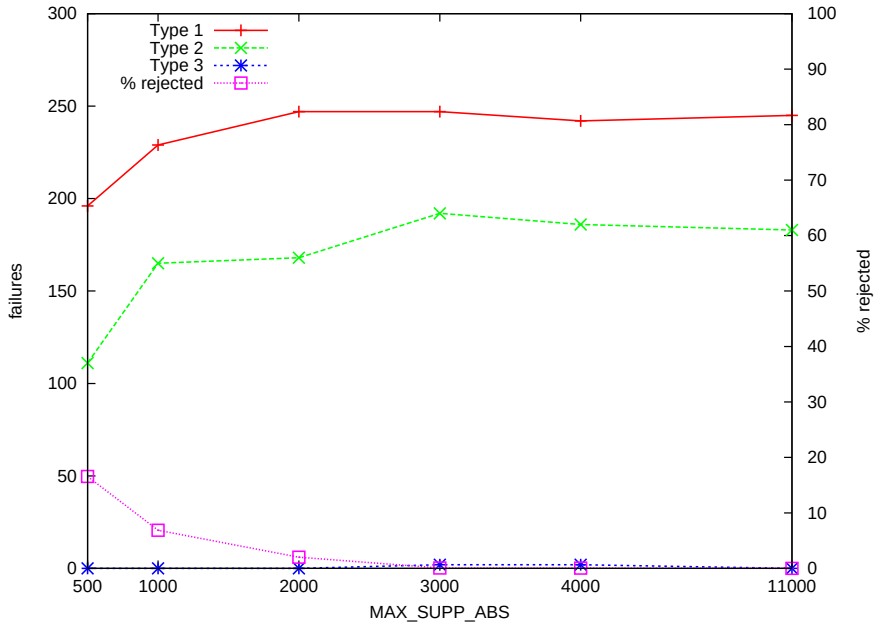

Figure 6: Number of failures and cut rejection rate for cut generators with different values of `MAX_SUPP_ABS`.

$[0, 10724]$. We report results for the values `MAX_SUPP_ABS` $= 500, 1000, 2000, 3000, 4000, 11000$. For `MAX_SUPP_REL`, we report results for the values $0.1, 0.2, 0.5, 0.8, 0.9, 1.0$. Note that several cut generators in COIN-OR `Cgl` and `SCIP` have a nonzero value for both parameters, but for simplicity here we test the parameters one at a time.

Figure 6 graphs the number of failures and percentage of rejected cut depending on the value of `MAX_SUPP_ABS`, whereas Figure 7 reports the same information depending on the value of `MAX_SUPP_REL`.

The graphs show that limiting the maximum cut support has effect mostly on Type 2 failures, but little effect on Type 1 failures (there are too few Type 3 failures to detect any difference). A Friedman test to compare cut generators with `MAX_SUPP_ABS` ≥ 1000 does not reject the null hypothesis that they have the same number of failures of Type 1, with a p-value of 0.6007. If we compare the number of dives that end with a failure of Type 2 instead, the null hypothesis is rejected with a p-value of 0.0008. Similarly, for `MAX_SUPP_REL` ≥ 0.5 no difference in the number of failures of Type 1 is detected, but it is detected for Type 2 failures. If we limit the support even more, then the number of Type 1 failures decreases as well. This suggests that limiting the maximum cut support does not affect much the generation of invalid cuts unless we use a low threshold, however it can help in making the LPs easier to solve.

5.3 Cut modification parameters

We now turn our attention to the cut modification procedures by varying the corresponding parameters described in Section 2.1. As `MIN_VIOL` is set to 0, cuts can only be rejected if, after modification, they are no longer violated by the current LP solution.

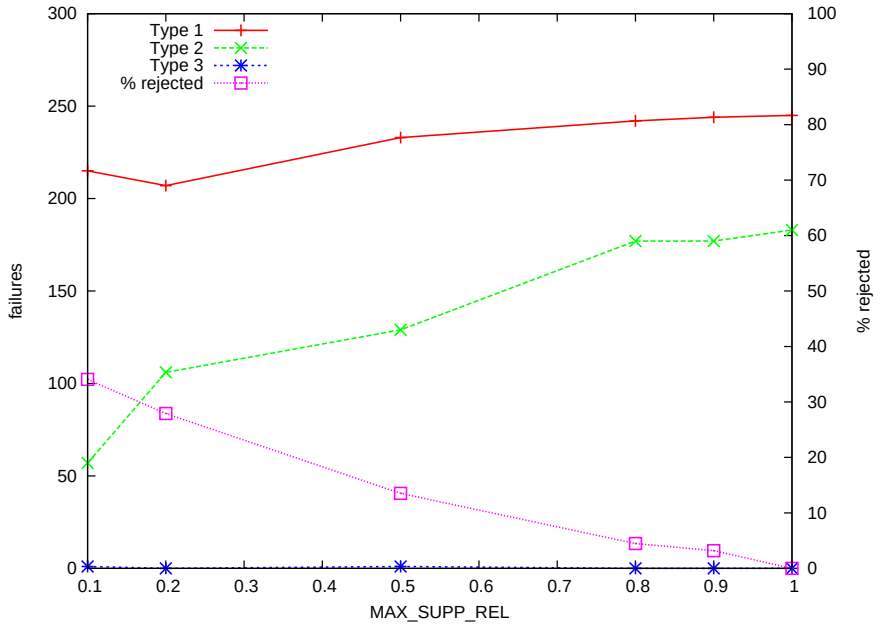

Figure 7: Number of failures and cut rejection rate for cut generators with different values of `MAX_SUPP_REL`.

5.3.1 Elimination of coefficients on surplus variables

The `EPS_ELIM` parameter takes value in $(0, \infty]$, but it is reasonable to assume that it should have a relatively small positive value. Indeed, eliminating large cut coefficients on surplus variables before their substitution in terms of original variables is likely to yield invalid cuts. We test the values `EPS_ELIM` $= 10^k$ for $k = -20, -18, \ldots, -2$ to get a sense of the impact of the parameter. The number of recorded failures and the cut rejection rate are reported in Figure 8.

In our experiments, the number of failures of Type 1 grows exponentially for `EPS_ELIM` $> 10^{-12}$. By eliminating large coefficients on surplus variables before their substitution, we generate invalid cuts. As expected, only small values for `EPS_ELIM` make sense in practice. For all values of `EPS_ELIM` $< 10^{-12}$, we observe very similar performance in terms of number of failures and rejection rate.

5.3.2 Elimination of small cut coefficients

`EPS_COEFF` takes value in $(0, \infty]$. When cut coefficients smaller than `EPS_COEFF` are set to zero, the right-hand side of the cut is adjusted accordingly to preserve validity. We tested the values `EPS_COEFF` $= 10^k$ for $k = -20, -18, \ldots, -2$.

We can see from Figure 9 that `EPS_COEFF` seems to have an impact on the number of failures of all three types, especially Type 1 failures. The rejection rate increases quickly for `EPS_COEFF` $\geq 10^{-6}$ as the cut is no longer violated by the current LP solution.

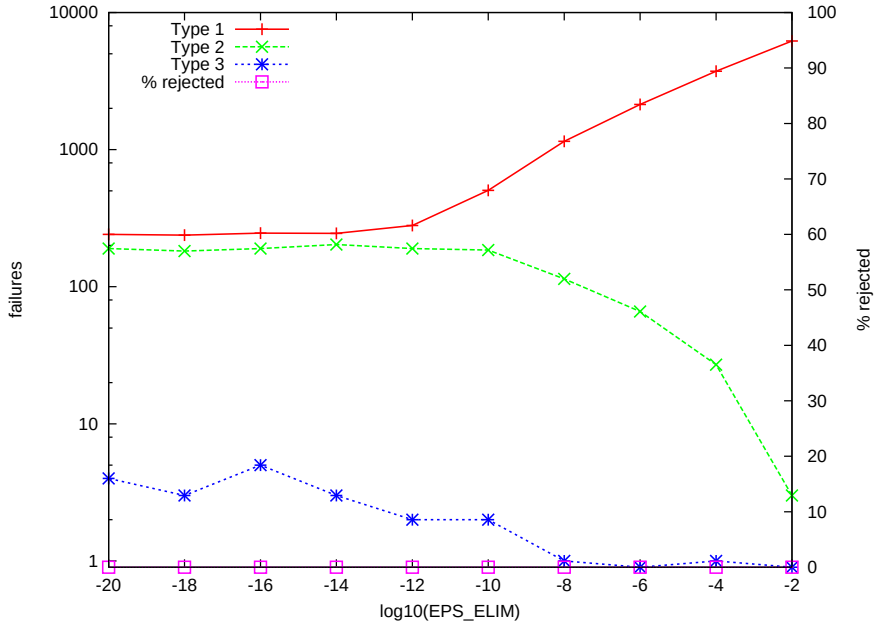

Figure 8: Number of failures and cut rejection rate for cut generators with different values of `EPS_ELIM`. `EPS_ELIM` is set to 10^k, where k is the value on the x axis. The left y-axis has a logarithmic scale.

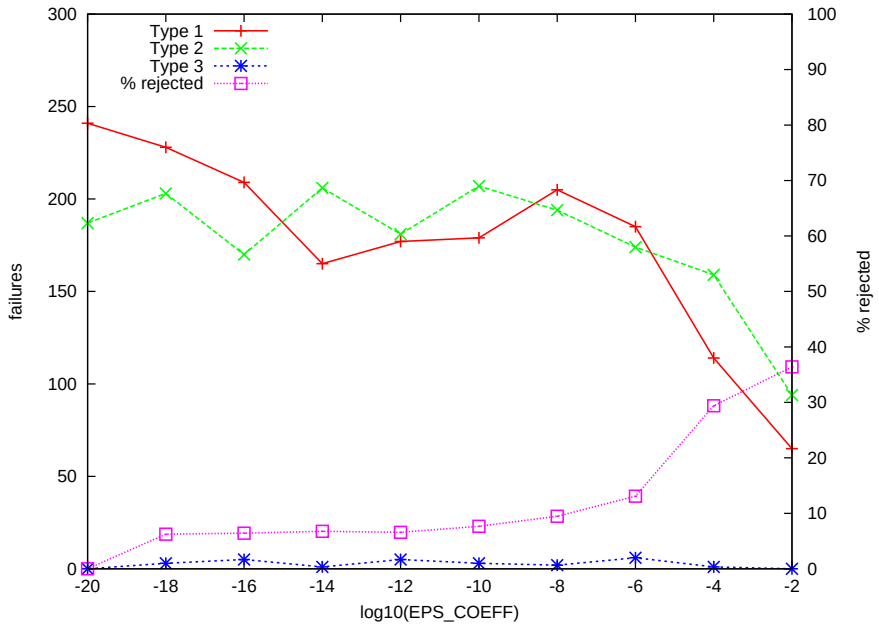

Figure 9: Number of failures and cut rejection rate for cut generators with different values of `EPS_COEFF`. `EPS_COEFF` is set to 10^k, where k is the value on the x axis.

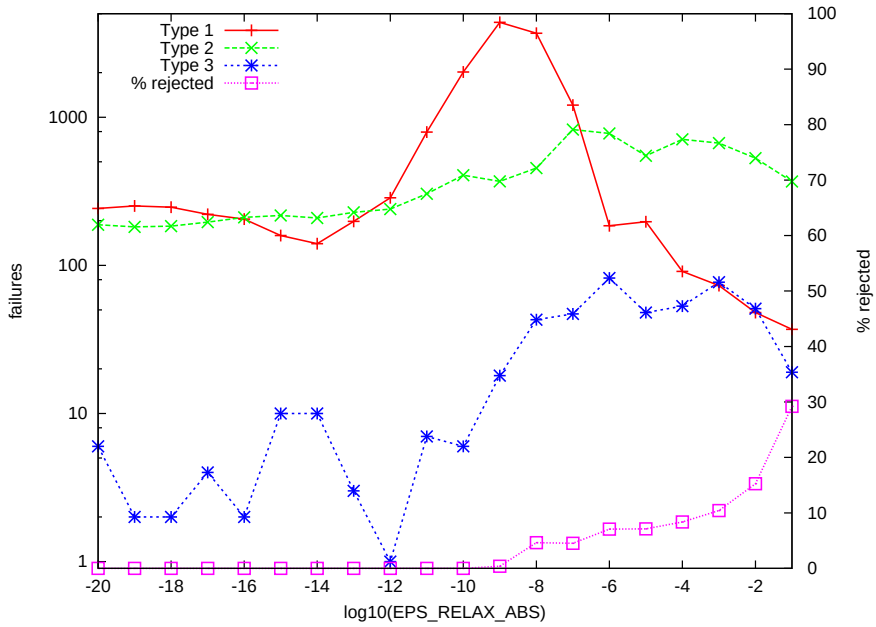

Figure 10: Number of failures and cut rejection rate for cut generators with different values of EPS_RELAX_ABS. EPS_RELAX_ABS is set to 10^k, where k is the value on the x axis. The left y-axis has a logarithmic scale.

5.3.3 Relaxation of the right-hand side value

Relaxation of the cut right-hand side value is controlled by two parameters: An absolute relaxation EPS_RELAX_ABS and a relative relaxation EPS_RELAX_REL. They both take value in $[0, \infty]$, but large values are likely to lead to an inequality not violated by the LP solution. For both parameters, we test the values 10^k for $k = -20, -19, \ldots, -1$.

Figure 10 plots the results for EPS_RELAX_ABS and Figure 11 those for EPS_RELAX_REL.

For both parameters, values larger than 10^{-6} increase the rejection rate while decreasing the number of failures. This is not surprising as the cut relaxation is significant. For values smaller than 10^{-12}, the cut rejection rate is small and the number of failures is fairly stable. However for values of the parameters in the range $[10^{-12}, 10^{-6}]$, the number of Type 1 failures increases significantly (up to a factor of 25 in the case of EPS_RELAX_ABS). We investigated this behavior and found that the amount by which the rhs of the cut is relaxed directly affects the fractionality of the basic integer variables at later rounds. We provide data to support this claim.

For each value of EPS_RELAX_ABS tested above and for each dive, we record the fractionality of the basic integer variables in all rounds, regardless of whether or not a GMI cut is derived from the corresponding row. For each instance in Failure Set, we compute over all dives and over all rounds the percentage p_k of basic integer variables whose fractionality falls in each of the ranges $[10^k, 10^{k+1}), k = -9, \ldots, -1$. Then, we compute the average $E[p_k]$ over all the instances, for all values of k. The heat map in Figure 12 shows that $E[p_k]$ is maximum when 10^k is close to the value of EPS_RELAX_ABS. Experiments with different cut generators yield the same conclusions.

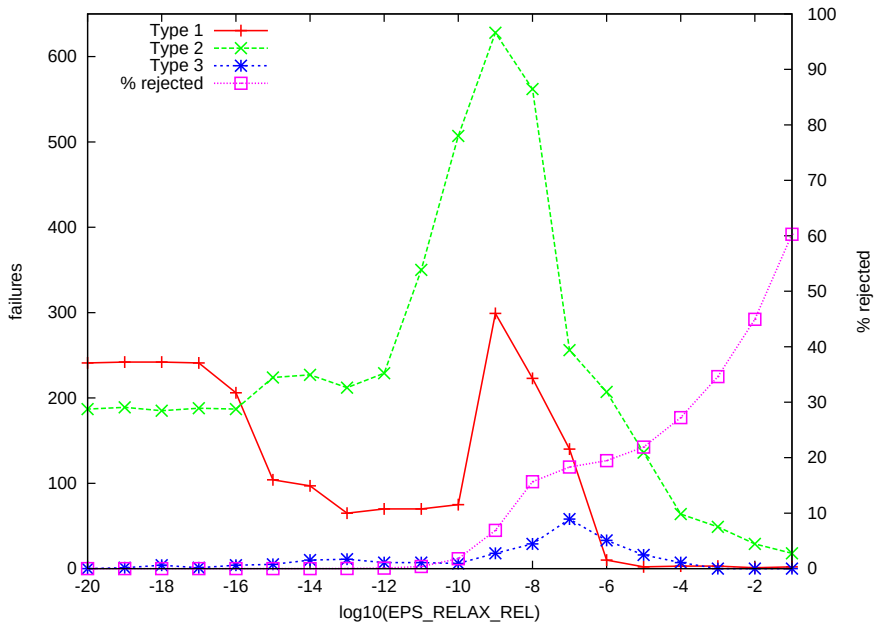

Figure 11: Number of failures and cut rejection rate for cut generators with different values of `EPS_RELAX_REL`. `EPS_RELAX_REL` is set to 10^k, where k is the value on the x axis.

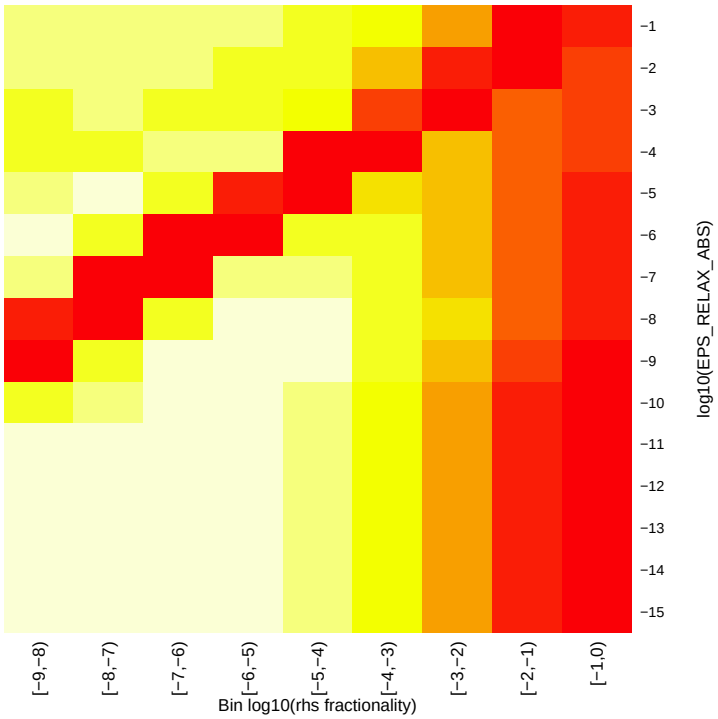

Figure 12: Heat map of the matrix relating the fractionality of the basic integer variables and the value of `EPS_RELAX_ABS`. Darker colors correspond to larger values in the matrix. To enhance the picture, each column is rescaled to have its maximum equal to 1, i.e. the darkest color.

In light of these results, we can explain why relaxing the rhs by values in the range $[10^{-12}, 10^{-6}]$ yields an increase in the number of failures: the number of basic integer variables with small fractionality ($\leq 10^{-6}$) increases, which leads to potentially dangerous cuts, as shown in Section 5.2.1.

We also note that 10^{-9} is the primal feasibility tolerance. Therefore, cut relaxations of that order could increase the degeneracy of the LP bases and potentially lead to numerical troubles. However, we were unable to confirm whether or not primal degeneracy plays any role in the observed behavior of the cut generators.

6 Empirical testing: Parameter optimization

To investigate the question of finding the optimal values for the cut generation parameters, we must first specify what we mean by "optimal".

In this paper, we are concerned with the safety of the cuts, as opposed to their strength. Our assumption is that cut generators should be compared in terms of strength only when they are comparable in terms of safety. We think of the cut modification and numerical check procedures as a filter. The cuts are modified and then checked, and they can be either accepted or rejected depending on whether or not they are judged safe. Strength does not play a role here. We would like the filter to be as loose as possible, while maintaining a given level of safety. In other words, we want to minimize the rejection rate of the cut generator, while achieving at least a given level of safety. We assume that choosing good cuts among the cuts that are not rejected will come at a later stage. This approach has the advantage of decoupling two complicated issues, cut strength and cut safety.

It is important to note that in our preliminary computational experiments we observed that, for a given level of safety, minimizing the cut rejection rate also led to an advantage in terms of gap closed, which is a measure of strength. A comparison of several cut generators with a similar performance according to a Friedman test on the failure rate showed that cut generators that reject fewer cuts close more integrality gap at the end of the cut generation loop. A more detailed presentation of these experiments is given in Section 7.

6.1 Choice of the safety level

The optimization algorithm treats safety of the cut generator as a constraint. This implies that we should first define our measure of safety, and decide what is an acceptable level.

We measure safety of a cut generator on a set of instances by computing its *failure rate* with the Dive-and-Cut procedure. The failure rate is defined as the fraction of dives that result in a failure of Type 1, 2 or 3. In our experiments we have 51 instances and we typically perform 300 dives on each for a total of 15,300 dives. Note that each dive may itself involve the generation of hundreds or even thousands of cuts. The validity of each cut is tested against a set of known feasible solutions.

To compute what is an acceptable failure rate over Failure Set, we test the commercial MILP solver IBM ILOG `Cplex` 12.2. We use `Cplex` as a black-box GMI cut generator within the Dive-and-Cut framework. To obtain the generator we call CpxGMI, we do the following.

(i) Disable all preprocessing routines and all cutting planes except GMI cuts;

(ii) Set `CPX_PARAM_PRELINEAR` = 0 and `CPX_PARAM_MIPCBREDLP` = 0 to obtain information on the original, unmodified LP;

(iii) Set the node limit to 1;

(iv) Set the `branchcallback()` function to a user-defined function;

(v) Store the number of rows m of the original LP;

(vi) Launch Branch-and-Bound (the `mipopt()` function);

(vii) Within the user-defined `branchcallback()` function, read and return the rows with index larger than m. These are the cutting planes added by `Cplex`.

Note that we let `Cplex` generate GMI cuts with its default settings (number of candidate variables, number of rounds). As a result, we do not know precisely how many rounds are applied. Furthermore, we do not know if some of the generated cutting planes are discarded in subsequent rounds. Since we cannot control the cut generation loop in `Cplex`, the only possibility is to read from the LP formulation the cuts that are still in the LP when `Cplex` decides to branch at the root.

We apply Dive-and-Cut with the cutting planes generated by `Cplex` as described above, performing 300 dives on each instance of Failure Set with a time limit of 600 seconds per dive. The experiments in this section were run on a machine equipped with an `AMD Opteron 4176 HE` processor clocked at 2.4GHz and 48GB RAM, running Linux. In Table 2 we report the number of failures on each instance in Failure Set. The total failure rate is of 0.03%.

	# of failures		
Instance	T. 1	T. 2	T. 3
`arki001`	1	0	0
`gt2`	1	0	0
`opt1217`	1	0	0
`p0033`	1	0	0
`p2756`	1	0	0

Table 2: Number of failures recorded by the GMI cuts generated by `Cplex` with default parameters. We report only the instances in Failure set on which at least one failure occurred. For each instance, we performed 300 dives with Dive-and-Cut.

By default, `Cplex` generates several families of cuts. Using the CpxGMI generator described in Section 6.1 yields an average of 22.70 cuts per dive. To compute the number of generated cuts per dive we proceed as follows: We initialize nrows $\leftarrow m$, numcuts $\leftarrow 0$ and employ a user-defined `cutcallback()` function that performs the following steps each time it is called.

(i) Obtain the current LP relaxation with m' rows

(ii) Set numcuts $\leftarrow$ numcuts $+ \max\{m' - nrows, 0\}$

(iii) Set nrows $\leftarrow m'$.

The final value of numcuts is a lower bound to the number of generated cuts. Despite the small number of GMI cuts generated by `Cplex`, some failures occur. It is interesting to note that failures of Type 1 are recorded on seemingly innocuous instances such as `p0033`. The small number of cuts explains the absence of Type 2 or Type 3 failures.

Even though we plan to generate significantly more cuts than CpxGMI, we want to achieve a similar or better level of safety. To this end, we only consider cut generators that satisfy a constraint on the maximum failure rate, and that are found to perform equally or better than the GMI cut generator of `Cplex` according to a Friedman test on the failure rate. The next section explains this in more detail.

6.2 The optimization algorithm

Optimizing the cut generation parameters is a black-box optimization problem. The objective function (cut rejection rate) and the constraints (failure rate) are unknown functions of the decision variables. Moreover, evaluating these unknown functions is computationally expensive, since this is done by running Dive-and-Cut on all instances of the test set.

Several methods for optimizing expensive black-box functions can be found in the literature. One possible approach is to use a response surface method (see e.g. [16, 25]), where the constraint violations can be embedded into the objective function as penalty terms. Black-box optimization methods are typically tailored for continuous problems, avoiding the difficulty of dealing with discrete variables. More recently, some attempts at solving problems with integer variables have been made [17].

Instead of using an existing method from the literature, we decided to develop an ad hoc optimization algorithm for three reasons. First, we want to use a multidimensional objective function. That is, instead of considering the average cut rejection rate over all the instances and compare generators based on this single value, we consider the cut rejection rate on each instance. Second, for assessing the safety of the generator, the failure rate must be below the threshold and, in addition, a Friedman test on the failure rate must show that the generator is comparable to or better than a reference generator. Third, we have the possibility of evaluating several points in the parameter space in parallel, using the Condor grid. Traditional response surface methods evaluate only one point at a time in a sequential fashion.

By using a statistical test for comparing points, we avoid the pitfall of aggregating results on several instances into a single measure of quality.

An important observation from the single-parameter experiments in Section 5 is that the cut rejection rate is monotone along each axis of the parameter space, and has a convex or almost-convex shape. Thus, an optimization algorithm that performs some kind of local search can reasonably expected to find a good solution. Note however that since we use a vector-valued objective function and use a Friedman test on failure rates for comparing points, even convexity of the objective function would not guarantee convergence.

We discretize the set of possible values for each cut generation parameter. We use evenly spaced points in a reference interval, sometimes using a logarithmic scale for the parameter, using our best judgment for each parameter.

For the optimization algorithm, we assume that there are h parameters to optimize, and the i-th parameter can take values in the ordered set $P_i = \{p_{i,1}, p_{i,2}, \ldots, p_{i,\mathrm{len}(i)}\}$ where $\mathrm{len}(i) \in \mathbb{N}$, and $p_{i,j} < p_{i,j+1}$ for $j = 1, \ldots, \mathrm{len}(i) - 1$. A cut generator g is completely characterized by a point in $P_1 \times P_2 \times \cdots \times P_h$, and we denote by $g(i)$ the value of the i-th parameter that defines

g. For all $i, j \in \mathbb{Z}$, we define the function:

$$\text{midpoint}(i,j) = \begin{cases} i & \text{if } |i-j| \leq 1, \\ \lceil (i+j)/2 \rceil & \text{otherwise.} \end{cases}$$

For all $S_i \subseteq \{1, \ldots, \text{len}(i)\}$, we use the notation $P_i(S_i) = \{p_{i,j} : j \in S_i\}$. Algorithm 2 describes the main loop of the optimization algorithm. We label this algorithm OptimizeParameters. The algorithm is a simple grid refinement algorithm. It repeatedly selects up to three values for each parameter, evaluates all generators with parameters on the grid defined by these values, selects the best generators (using the subroutine select_best(G) whose description is given in Algorithm 3), and computes the smallest box containing all generators in that set. OptimizeParameters employs simple mechanisms to ensure that the search does not collapse too quickly towards a single point of the grid.

Algorithm 2 OptimizeParameters

.

```
for i = 1, ..., h do
    ℓ_i ← 1
    u_i ← len(i)
    S_i ← {ℓ_i, midpoint(ℓ_i, u_i), u_i}
G' ← P_1(S_1) × P_2(S_2) × ··· × P_h(S_h)
repeat
    G ← G'
    Evaluate cut generators at grid points g ∈ G
    B ← select_best(G)
    for i = 1, ..., h do
        ℓ'_i ← arg min_j {p_{i,j} : (∃g ∈ B : g(i) = p_{i,j})}
        u'_i ← arg max_j {p_{i,j} : (∃g ∈ B : g(i) = p_{i,j})}
        if ℓ'_i = u'_i then
            center ← midpoint(ℓ_i, u_i)
            if ℓ'_i = ℓ_i then
                ℓ'_i ← 2ℓ_i − midpoint(ℓ_i, center)
                u'_i ← midpoint(ℓ_i, center)
            else if u'_i = u_i then
                ℓ'_i ← midpoint(u_i, center)
                u'_i ← 2u_i − midpoint(u_i, center)
            else
                ℓ'_i ← midpoint(ℓ_i, center)
                u'_i ← midpoint(u_i, center)
        ℓ_i ← max(1, ℓ'_i)
        u_i ← min(len(i), u'_i)
        S_i ← {ℓ_i, midpoint(ℓ_i, u_i), u_i}
    G' ← P_1(S_1) × P_2(S_2) × ··· × P_h(S_h)
until G = G'
```

Given two cut generators $g, g' \in G$, we write $g <_R g'$ if a Friedman test on the cut rejection rate prefers g over g'. Similarly, we write $g <_F g'$ if a Friedman test on the failure rate yields

that g is better than g'. Note that the $<_R$ and $<_F$ relations depend on the set of cut generators included in the statistical test. In Algorithm 3, $<_R$ or $<_F$ always refer to the test just performed.

For a set G of generators and a reference generator $\tilde{g}$, Algorithm 3 first selects in G' all generators in G that satisfy the upper bound on the failure rate and are not dominated by $\tilde{g}$ according to a Friedman test on the failure rate. In this paper, we use the CpxGMI cut generator discussed in Section 6.1 as the reference cut generator. It then applies a Friedman test on the cut rejection rate on generators in G' and selects in B all generators that are not rated as worse than any other generator in G'. As mentioned in Appendix A, pairwise comparisons based on a Friedman test are not transitive, implying that it is possible to have three generators g_1, g_2 and g_3 with pairwise comparisons $g_1 <_F g_2$, $g_2 <_F g_3$, and $g_3 <_F g_1$. When this happens, none of the three generators are included in B, even if as a group they dominate all other generators in G'. To mitigate this unfortunate situation, the algorithm has a last loop that can increase the set B. That loop computes the set C of all generators in $G' \setminus B$ such that adding any one of the generators $c \in C$ results in c dominating a generator in B according to a Friedman test on $B \cup c$. One of the generators in C is then selected and added to B and this is repeated while the computed set C is nonempty. Note that more sophisticated selection of dominant subsets from inconsistent pairwise comparisons can be found in the literature [3, 27]. For our purposes, the simple approach above seems to work well enough.

The selection of the generator in C requires a distance function, which we define next. Observe that in OptimizeParameters a grid G has up to 3 possible values for each parameter. Given two generators $g, g' \in G$ and parameter i, the distance between g and g' along parameter i is 0 if $g(i) = g'(i)$, it is 2 if $|S_i| = 3$ and one of $g(i)$ or $g'(i)$ is the minimum value in S_i and the other is the maximum value, and it is 1 in all other cases. The distance $d(g, g')$ is then defined as the sum over all parameters i of the distance between g and g' along parameter i.

Algorithm 3 select_best()

Input: Set of cut generators G, maximum failure rate γ, reference generator $\tilde{g}$
Output: Set of best cut generators B
Apply a Friedman test on $G \cup \{\tilde{g}\}$ on the failure rate
$G' \leftarrow \{g \in G : (\text{failure_rate}(g) \leq \gamma) \wedge (g <_F \tilde{g})\}$
Apply on G' a Friedman test on the cut rejection rate
$B \leftarrow \{g \in G' : (\nexists g' \in G : g' <_R g)\}$
repeat
  $C \leftarrow \emptyset$
  for all $g \in G' \setminus B$ **do**
    Apply on $B \cup \{g\}$ a Friedman test on the cut rejection rate
    if $\exists g' \in B : g <_R g'$ **then**
      $C \leftarrow C \cup \{g\}$
  if $C \neq \emptyset$ **then**
    Select $c \in C$ such that $\min_{g \in B}\{d(c, g)\}$ is minimum; break ties selecting c such that $\text{conv}(B \cup \{c\})$ contains the largest number of elements in C; break further ties arbitrarily
    $B \leftarrow B \cup \{c\}$
until $C = \emptyset$

Observe that OptimizeParameters terminates if we were not able to refine the grid during the previous iteration. Refining the grid depends on the detection power of the statistical test

performed in the subroutine select_best() (Algorithm 3). When the grid cannot be refined, we could follow several strategies, such as increasing the number of dives or rounds to increase the detection power of the statistical tests, branching on the parameter space, or focusing only on one area of the parameter space. However, in our experiments, we were always able to refine the grid until it was sufficiently small, and therefore we did not need to resort to such strategies.

OptimizeParameters is fully determined once the discretized values for each parameter are given. At the end of algorithm, we obtain a set of cut generators that yield lower cut rejection rates than the remaining generators tested during the course of the optimization, and such that a Friedman test on the rejection rate does not detect differences within the set.

6.3 Most influential parameters and initial grid

OptimizeParameters evaluates 3^h points at each iteration. Since we have 12 parameters, this would require evaluating an overwhelming $3^{12} = 531,441$ generators at each iteration. Therefore, we first identify the most useful parameters, optimize over this smaller set and then find good values for the remaining ones.

To select the most useful parameters, we sample 500 points uniformly at random in the discretized parameter space. We run Dive-and-Cut on these 500 points and fit a quadratic model for the total number of failures and for the rejection rate. We then use classical regression techniques to identify the most relevant parameters. Details are presented in the remainder of the section.

To choose the initial parameter ranges, we start with the ranges considered in Section 5, and reduce them based on the results of the experiments reported in that section. In particular, let P_i be the ordered set of values tested in Section 5 for the i-th parameter. We set the lower (resp. upper) bound to the smallest (resp. largest) value such that the failure rate is at most 6% and the rejection rate is at most 66%.

The 6% value for maximum failure rate allowed is chosen to exclude cut generators that yield more failures than CgBase. The value 66% for the maximum rejection rate allowed is chosen after testing a small number of "good" cut generators with typical parameter values. The smallest cut rejection rate recorded was 65.78% and the corresponding failure rate was 0.04%. Since we already know a cut generator with a rejection rate of 65.78% and low failure rate, we exclude parameter ranges that are not likely to contain a better generator. Unfortunately this range reduction technique was not very effective: we could only reduce the range for `MAX_DYN` (lower bound increased to 10^6) and for `EPS_ELIM` (upper bound decreased to 10^{-12}). The resulting discretized parameter space P, from which the parameter values are randomly sampled is reported in Table 3. We sample 500 points $p_1, \ldots, p_{500}$ from P uniformly at random.

We evaluate the performance of cut generators parametrized with $p_1, \ldots, p_{500}$ on a subset of 25 instances of Failure Set, chosen randomly. Let $f : P \to \mathbb{R}$ and $r : P \to \mathbb{R}$ be the function returning respectively the failure rate and rejection rate.

We use the 12 parameters and their 66 first-order interaction terms and compute the best (smallest ℓ_2-norm of the vector of residuals) linear model fitting the points $(p_i, f(p_i))$ for $i = 1, 2, \ldots, 500$. We do this with the additional restriction that the linear model must use exactly s of the terms, for $s = 1, 2, \ldots, 12$. Results are reported in Table 4. For brevity, we use l() to indicate log, and only the initials of each parameter (e.g., `MDL` instead of `MAX_DYN_LUB`). Some parameters can assume the value 0; we substitute $\log(0) = -50$ for regression. Computations are performed with the open-source software `R` [24], using the packages `biglm` and `leaps`.

AWAY	10^i	$i = -9, \ldots, -1$
EPS_COEFF	10^i	$i = -\infty, -20, \ldots, -1$
EPS_RELAX_ABS	10^i	$i = -\infty, -20, \ldots, -1$
EPS_RELAX_REL	10^i	$i = -\infty, -20, \ldots, -1$
MAX_DYN	10^i	$i = 6, \ldots, 30$
MIN_VIOL	10^i	$i = -\infty, -20, \ldots, -1$
MAX_SUPP_ABS	$250i$	$i = 1, \ldots, 16$
MAX_SUPP_REL	$i/10$	$i = 1, \ldots, 10$
EPS_ELIM	10^i	$i = -\infty, -20, \ldots, -12$
LUB	10^i	$i = 2, 3, 4, 50$
MAX_DYN_LUB	10^i	$i = 6, \ldots, 30$
EPS_COEFF_LUB	10^i	$i = -\infty, -20, \ldots, -1$

Table 3: Discretized parameter space. By convention, $10^{-\infty}$ means 0.

size	l(A)	l(ERR)	l(MV)	l(A) l(ERA)	l(A) l(ERR)	l(A) l(MV)	l(EC) MSA	l(EC) l(MV)	l(EC) l(MDL)	l(ERA) l(MD)	l(ERR) l(MV)	MSA l(ECL)	l(MDL) l(ECL)	BIC
1	*													-55.99319
2					*	*								-75.88324
3		*			*						*			-87.27452
4		*	*		*						*			-89.88623
5		*	*		*	*					*			-98.61016
6		*	*	*	*	*					*			-99.13274
7		*	*	*	*	*				*	*			-96.73726
8		*	*		*	*	*	*			*	*		-100.72865
9		*	*		*	*	*		*		*	*	*	-105.34726
10		*	*	*	*	*	*		*		*	*	*	-105.75085
11		*	*	*	*	*	*		*	*	*	*	*	-105.89380
12		*	*	*	*	*	*	*	*	*	*	*	*	-105.57976

Table 4: Independent variables defining the best subset of parameters for fitting a linear model to the failure rate function f. Column "s" specifies the size of the subset. Terms in a subset are identified with a "*". If a column label contains two parameters, it indicates an interaction term. The last column reports the Bayesian Information Criterion (BIC) value for each model.

We repeat the same process for the cut rejection rate, using the points $(p_i, r(p_i))$ for $i = 1, 2, \ldots, 500$. Results are reported in Table 5.

We consider that the optimization can be performed in reasonable time on up to six parameters. From Tables 4 and 5, we select the parameters AWAY, EPS_COEFF, EPS_RELAX_ABS, EPS_RELAX_REL, MAX_DYN, and MIN_VIOL. These six parameters (and their interaction terms) are sufficient to form the best subsets of parameters of size up to 7 for the model of the failure rate f, and up to 5 for the model of the cut rejection rate r. The Bayesian Information Criterion (BIC) values for the model using 7 terms for f and for the model using 5 terms for r are within 10% of the minimum reported values, suggesting that these models are not over-fitting the data and predict the dependent variable well compared to the other subsets.

The grid over which the parameters are optimized is therefore the one reported in Table 3, limited to the six chosen parameters.

size	l(EC)	l(ERR)	l(MDL)	l(A) l(ERA)	l(A) l(ERR)	l(A) l(MD)	l(A) l(MDL)	l(EC) l(MD)	l(ERA) l(ERR)	l(ERR) l(MD)	l(ERR) l(MV)	l(MD) l(L)	l(MV) l(MDL)	l(L) l(MDL)	l(L) l(ECL)	BIC
1										*						-136.9948
2						*				*						-227.1880
3	*					*				*						-254.1869
4	*			*				*		*						-280.0600
5	*					*		*	*		*					-309.9264
6	*					*		*	*	*			*			-317.0011
7	*					*		*	*	*			*	*		-317.0788
8	*					*		*	*	*		*	*	*		-319.7481
9	*					*	*	*	*	*		*	*	*		-318.6268
10	*	*			*		*	*	*	*	*	*		*		-318.9703
11	*	*	*		*	*		*	*	*	*	*		*		-317.5127
12	*	*	*		*	*		*	*	*	*	*		*	*	-315.7235

Table 5: Independent variables defining the best subset of parameters for fitting a linear model to the cut rejection rate function r. Column "s" specifies the size of the subset. Terms in a subset are identified with a "*". If a column label contains two parameters, it indicates an interaction term. The last column reports the Bayesian Information Criterion (BIC) value for each model.

6.4 Results of the optimization algorithm

We ran OptimizeParameters as described in Section 6.2 for 5 iterations. For each tested cut generator we perform 150 dives per instance. Our target failure rate is $\gamma = 0.05\%$, close to `Cplex`'s 0.03%. However, we start with $\gamma = 0.2\%$ at the first iteration, and lower this value by 0.05% at each iteration until we reach the desired level. This prevents the failure rate constraint to eliminate a large portion of the parameter space in the first iterations, while the parameter grid is still very coarse. Later, the average and the standard deviation of the failure rate of the tested cut generator decrease and we can be more strict with the maximum failure rate constraint. In the end, the cut generators must be at least as safe as a reference generator (CpxGMI here) according to a Friedman test on the failure rate.

In Table 6 we provide a summary of the first 5 iterations of OptimizeParameters. We report the bounds of the parameter ranges at each iteration, the maximum allowed failure rate γ, the fraction of tested cut generators that satisfy the constraint on the failure rate, the average and standard deviation of the failure rate of the tested cut generators, and the average and standard deviation of the cut rejection rate.

We note that there is typically a trade-off between cut rejection rate and failure rate, hence minimizers of the rejection rate have a failure rate close to the allowed maximum. This explains why the fraction of feasible cut generators does not increase between successive iterations. The best points in terms of rejection rate are close to the boundary of the feasible region, and OptimizeParameters focuses on areas that are close to the boundary. Since we lower the maximum failure rate γ in the first four iterations, the fraction of feasible cut generator drops. This can be seen for instance in Iteration 4, where only a few of the tested cut generators are feasible. Note that in the following iteration a larger fraction of cut generators is feasible, as γ is not changed. The average failure rate and cut rejection rate clearly show that OptimizeParameters is successful in identifying promising areas of the parameter space. By Iteration 5, both failure rate and rejection rate are very low and very stable across the tested generators (small standard deviation).

We report the parameters of the best cut generators found at Iteration 5 in Table 7. For

	Iteration				
	1	2	3	4	5
AWAY	$[10^{-9}, 10^{-1}]$	$[10^{-5}, 10^{-1}]$	$[10^{-2}, 10^{-1}]$	$[10^{-2}, 10^{-2}]$	$[10^{-2}, 10^{-2}]$
EPS_COEFF	$[0, 10^{-1}]$	$[0, 10^{-11}]$	$[10^{-16}, 10^{-11}]$	$[10^{-12}, 10^{-10}]$	$[10^{-11}, 10^{-11}]$
EPS_RELAX_ABS	$[0, 10^{-1}]$	$[0, 10^{-1}]$	$[0, 10^{-11}]$	$[10^{-13}, 10^{-9}]$	$[10^{-12}, 10^{-10}]$
EPS_RELAX_REL	$[0, 10^{-1}]$	$[0, 10^{-1}]$	$[0, 10^{-11}]$	$[10^{-18}, 10^{-13}]$	$[10^{-14}, 10^{-12}]$
MAX_DYN	$[10^{6}, 10^{30}]$	$[10^{6}, 10^{30}]$	$[10^{6}, 10^{18}]$	$[10^{6}, 10^{9}]$	$[10^{6}, 10^{6}]$
MIN_VIOL	$[0, 10^{-1}]$	$[0, 10^{-11}]$	$[0, 10^{-11}]$	$[0, 10^{-11}]$	$[0, 10^{-11}]$
γ	0.20%	0.15%	0.10%	0.05%	0.05%
% feasible	61.72%	35.80%	16.26%	1.23%	11.11%
Avg fail rate	0.74%	0.55%	0.31%	0.21%	0.12%
Std dev fail rate	1.65%	0.66%	0.19%	0.13%	0.05%
Avg rej rate	71.72%	57.12%	47.96%	40.03%	41.88%
Std dev rej rate	21.40%	22.73%	12.52%	5.53%	1.27%

Table 6: Summary of the results of the OptimizeParameters. "γ" indicates the maximum failure rate allowed at each iteration. "% feasible" indicates the fraction of tested cut generators that satisfy the constraints (maximum failure rate and, from Iteration 4 on, at least as safe as reference generator). We then report, for each iteration, the average and standard deviation of the failure rate of the cut generators, and the average and standard deviation of the average cut rejection rate per instance.

comparison, in Figure 13 we report a histogram of the average cut rejection rate of all cut generators analyzed by OptimizeParameters that satisfy the maximum failure rate constraint (0.05%). This is a total of 309 cut generators. The cut generators in Table 7 are among the best 1% generators encountered by OptimizeParameters in terms of average cut rejection rate. It is interesting to note that no cut generator falls in the 50%-60% bin for the average rejection rate. In hindsight, we can explain this gap. OptimizeParameters explores areas of the parameter space with small cut rejection rate but failure rate slightly above the allowed threshold 0.05%, converging towards the only remaining feasible cut generators in the area that was identified as having the lowest rejection rate.

6.5 Parameter sensitivity

We now proceed to analyze the sensitivity of the failure and cut rejection rates with respect to the cut generation parameters in the neighborhood of one of the best generators found in the previous section. Our reference cut generator g^* is #1 from Table 7. Note that since the three generators are very similar and differ in one parameter value only, it seems likely that the results in this section are valid for the other two generators also. Details of our methodology and results are given in Appendix D.

Results in Appendix D suggest that for some of the parameters (e.g. `AWAY, EPS_RELAX_ABS, EPS_RELAX_REL, MAX_DYN`) even small changes have a visible effect. For other parameters, there is more freedom in choosing the parameter value. In our experiments, the parameters controlling the maximum support of the cutting planes have almost no effect on the number of failures. This is probably due to the data set and the large time limit before a Type 3 failure is reported

	Cut generator		
	1	2	3
AWAY	10^{-2}	10^{-2}	10^{-2}
EPS_COEFF	10^{-11}	10^{-11}	10^{-11}
EPS_RELAX_ABS	10^{-11}	10^{-11}	10^{-11}
EPS_RELAX_REL	10^{-13}	10^{-13}	10^{-13}
MAX_DYN	10^{6}	10^{6}	10^{6}
MIN_VIOL	0	10^{-16}	10^{-11}
Failure rate	0.04%	0.04%	0.04%
Rejection rate	41.27%	41.27%	41.27%

Table 7: Best cut generators returned by OptimizeParameters.

(10 minutes). In practice, it may be desirable to set some limit for the cut support to speed up LP resolves, but in this paper we focus on safety and we did not find evidence to support the claim that dense cuts are less safe than sparse cuts, provided that the most important cut generation parameters are well chosen. Setting to zero small coefficients on surplus variables does not show any positive effect in our experiments: any nonzero value for EPS_ELIM yields a small (but statistically significant) increase in the number of failures, and for larger values many invalid cuts are generated (see Section 5.3.1). Using a positive value for EPS_ELIM may yield some CPU time savings, but in terms of safety it does not seem advantageous. The experiments with EPS_RELAX_ABS, EPS_RELAX_REL showed the behavior already observed in Section 5.3.3: a value of the parameter approximately in the range $[10^{-9}, 10^{-6}]$ yields an increase in the number of failures, although here the increase is not as large as in Section 5.3.3 because the remaining cut generation parameters mitigate the effect. It does not seem a good idea to choose EPS_RELAX_ABS or EPS_RELAX_REL close to 10^{-9}. The ranges for MAX_DYN_LUB are very similar to those of MAX_DYN, and the differences could be explained by the fact that MAX_DYN_LUB is less influential than MAX_DYN as it acts on fewer variables. In light of these results, there is not much evidence to support using for MAX_DYN_LUB a value different than the one used for MAX_DYN. In our experiments, using any value for EPS_COEFF_LUB other than the starting value 10^{-13} yielded a small but statistically significant increase in the number of failures. We do not have an explanation for this behavior.

Finally, we briefly comment on the sensitivity of the results of OptimizeParameters with respect to the number of cut generation rounds ρ. Instead of running OptimizeParameters from scratch with a different value of ρ, we performed a local reoptimization as follows. The initial grid for the reoptimization is obtained from the grid at Iteration 5 of OptimizeParameters by extending each parameter range towards the direction of decreasing cut rejection rate. If parameter i has n_i discretized values in the grid at Iteration 5, the new range is extended to have $\max\{2n_i - 1, 2\}$ discretized values. Thus, if $n_i > 1$ one of the endpoints of the interval at Iteration 5 becomes the new midpoint. The range of MIN_VIOL at Iteration 5 cannot be extended in the direction of decreasing cut rejection rate, hence we keep the same range. We then set $\rho = 15$ and apply OptimizeParameters starting with this new grid. We remark that the algorithm allows the parameter ranges to extend past the initial bounds if necessary. The optimal generators are given in Table 8. The discussion in Section 4.4 explains why the optimal values of the cut generation parameters do not change significantly. In particular, the optimal

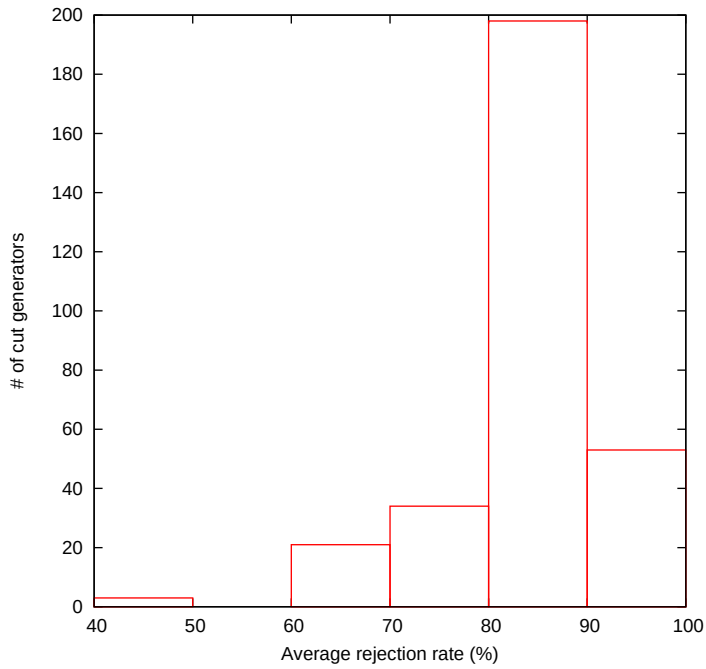

Figure 13: Histogram of the average rejection rate for the cut generators that satisfy maximum failure rate of 0.05%.

values `AWAY` $= 10^{-2}$ and `MAX_DYN` $= 10^{6}$ stay the same, because in our experiments deviating from these values yields cut generators that violate the safety constraints or reject too many cuts, even for $\rho = 15$.

	Cut generator		
	1	2	3
`AWAY`	10^{-2}	10^{-2}	10^{-2}
`EPS_COEFF`	10^{-12}	10^{-12}	10^{-12}
`EPS_RELAX_ABS`	10^{-10}	10^{-10}	10^{-10}
`EPS_RELAX_REL`	10^{-16}	10^{-16}	10^{-16}
`MAX_DYN`	10^{6}	10^{6}	10^{6}
`MIN_VIOL`	0	10^{-16}	10^{-11}
Failure rate	0.03%	0.03%	0.03%
Rejection rate	41.11%	41.11%	41.11%

Table 8: Best cut generators obtained by reoptimizing with $\rho = 15$.

7 Validation of the results

The goal of this section is to show that the conclusions drawn in the preceding sections have useful practical implications. It is important to note that the experiments of Sections 5 and 6 were carried out on the Condor grid, which is a heterogeneous computational environment. For

this reason, all code was compiled for a generic i386 architecture (gcc compiler option `-m32`) using 32 bit `Cplex`. All experiments in this section are performed on an AMD Opteron 4176 HE processor clocked at 2.4GHz and 48GB RAM. This is a x86_64 architecture, and we used 64 bit `Cplex` and `Cbc` [8] compiled with the `-mtune=native` option. Therefore, we can verify if our conclusions carry over to different architectures.

There are several points that we want to investigate. First, we want to check that the optimal generators obtained at the end of OptimizeParameters are safe and reject fewer cuts than other generators. We would also like to confirm that a small cut rejection rate translates into a cutting plane generator producing stronger cuts as a whole. A thorough analysis of the strength of cut generators is beyond the scope of this paper. However, here, we would like to investigate whether, if two cut generators have a comparable level of safety, the generator rejecting fewer cuts is at least as strong as the other one. We use the percent of integrality gap closed as a measure of strength. This not a very accurate measure of strength, but is a widely accepted approximation.

To address these points, we compare 11 cut generators over 300 dives of Dive-and-Cut on Failure Set. We use a different random seed than in previous experiments, and we fix variables randomly in all dives, including the first dive. Thus, we are not testing on the same instances that were used for OptimizeParameters, i.e. the integer variables are fixed in a different way. The cut generators that we test are the following.

- BestGen: generator #1 from Table 7.
- BestGenAway: generator #1 from Table 7 with `AWAY` set to $5 \cdot 10^{-3}$ as may be suggested by the results in Table 21.
- Iter1: the cut generator with lowest failure rate in the set of best cut generators explored at Iteration 1 of OptimizeParameters.
- Iter2: the cut generator with lowest failure rate in the set of best cut generators explored at Iteration 2 of OptimizeParameters.
- Iter3: the cut generator with lowest failure rate in the set of best cut generators explored at Iteration 3 of OptimizeParameters.
- CglGomory: the Gomory cut generator from `Cgl`.
- CglGomoryMod: our GMI cut generator parametrized in a similar way to CglGomory.
- CglLandP: the Lift&Project cut generator from `Cgl`, parametrized with `pivotLimit = 0` so that it generates GMI cuts.
- CglLandPMod: our GMI cut generator parametrized in a similar way to CglLandP.
- CpxGMI: `Cplex`'s GMI cut generator with default parameters (i.e. `Cplex` decides the number of cuts and the number of rounds of cutting planes).
- Cpx: `Cplex`'s cut generators with default parameters (i.e. `Cplex` decides which cutting plane families should be applied, the number of cuts and the number of rounds).

	Fail. rate	BestGen	BestGenAway	Iter1	Iter2	Iter3	CglGomory	CglGomoryMod	CglLandP	CglLandPMod	CpxGMI	Cpx
BestGen	0.065		=	=	=	=	-	=	=	-	=	-
BestGenAway	0.072	=		=	=	=	-	=	=	-	=	-
Iter1	0.084	=	=		=	=	-	=	=	-	=	-
Iter2	0.098	=	=	=		=	-	=	=	-	=	-
Iter3	0.065	=	=	=	=		-	=	=	-	=	-
CglGomory	3.555	+	+	+	+	+		+	+	+	+	=
CglGomoryMod	0.229	=	=	=	=	=	-		=	=	=	-
CglLandP	1.680	=	=	=	=	=	-	=		-	=	-
CglLandPMod	1.379	+	+	+	+	+	-	=	+		+	-
CpxGMI	0.039	=	=	=	=	=	-	=	=	-		-
Cpx	1.634	+	+	+	+	+	=	+	+	+	+	

Table 9: Comparison of the failure rate per instance. Column "Fail. rate" gives the average failure rate (%).

We note that the implementation of CglGomory uses many more tolerances than our GMI cut generator, therefore CglGomoryMod will yield different results. On the other hand, our implementation of CglLandPMod is very similar to CglLandP, but important differences remain. In particular, CglLandP is tied to COIN-OR `Clp` [9] as the LP solver, whereas we use `Cplex`. Furthermore, CglLandP generates at most 50 cuts per round and uses the optimal Simplex tableau returned by the LP solver, while our GMI cut generator has no limit on the number of generated cuts and internally recomputes the optimal Simplex tableau from scratch. For these reasons, comparisons with CglLandP are difficult to interpret and should be taken with a grain of salt.

We consider CglGomoryMod and CglLandPMod as "reasonable" parametrizations of the GMI cut generator that are interesting to compare to our reference BestGen. Comparing BestGen with Iter1, Iter2 and Iter3 allows us to verify that Algorithm 2 made progress and found better cut generators in later iterations. Note that Iter3 differs from BestGen only in the value of `EPS_RELAX_REL` (10^{-16} instead of 10^{-13}), hence they should have very similar results. We do not report results with Iter4 as it is identical to BestGen. BestGenAway is parametrized similarly to BestGen, but uses a smaller value of the `AWAY` parameter. This should yield a smaller cut rejection rate with a comparable safety level, according to the experiments in Appendix D.

We first compare the safety of the cut generators. We apply a Friedman test on the failure rate. The null hypothesis that all generators have the same failure rate is rejected with a p-value of 0.0000. We perform post-hoc analysis to identify which generators are safer by determining, for each pair of cut generators, if the difference in the failure rates is significant. Results are reported in Table 9.

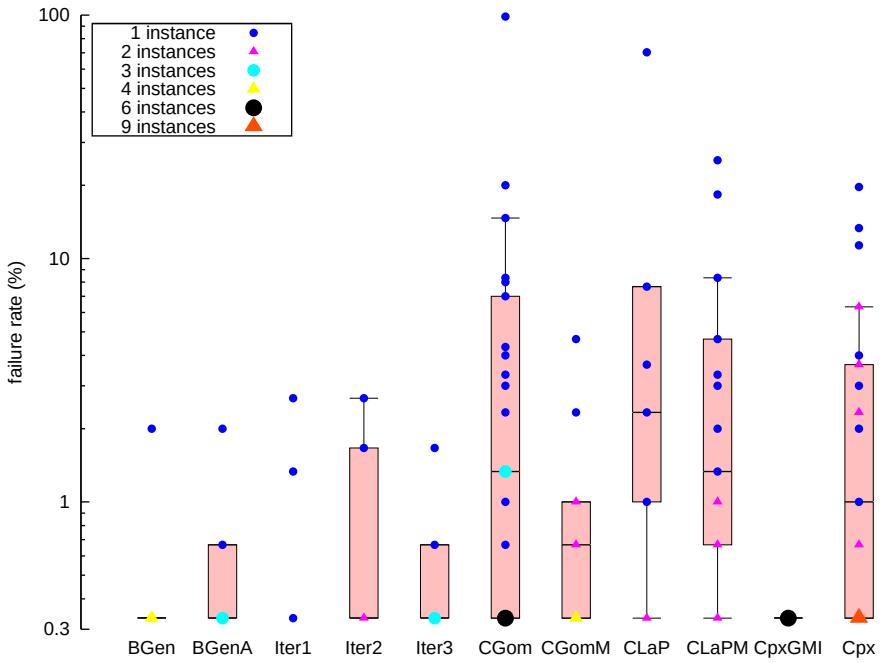

Figure 14: Average failure rate for all instances with nonzero failure rate. When two or more instances have the same failure rate for a cut generator, we group them together and increase the point size as indicated by the legend. For each cut generator, we additionally plot a box-and-whisker graph of the average failure rate. The box extends from the 25% to the 75% quantile, the middle bar indicates the median. The whiskers extend from the end of the box to the most distant point whose value lies within 1.5 times the difference between the 75% and the 25% quantile.

The results show that the total failure rate has increased slightly with respect to the maximum allowed failure rate during the optimization run. This is explained by the fact that we are now running dives on different instances and on a different machine, therefore it is not surprising that the computations give different results. However, according to the statistical tests reported in Table 9, BestGen, BestGenAway, Iter1, Iter2, and Iter3 are still as safe as CpxGMI, which is our target. They are safer than CglGomory, CglLandPMod and Cpx. Notice that CglLandP has a much larger failure rate than CpxGMI, but no difference is detected by the Friedman test, for two reasons. First, the test does not take into account the magnitude of the differences. Second, the large average is mostly due to a single instance with very large failure rate (`arki001` with 70.33% failure rate). In Figure 14 we report the average failure rate per instance. Figure 14 shows that among the cut generators deemed as safe as CpxGMI by a Friedman test, CglLandP is the least consistent, exhibiting a large failure rate on a few instances. CpxGMI, BestGen and BestGenAway are the most consistent, with very low failure rates on every instance.

As we cannot compute the cut rejection rate for cut generators that are not based on our implementation (i.e. `Cplex` and the two `Cgl` generators), we use the total number of cuts instead of the cut rejection rate. This is of course different than comparing the cut rejection rate, as a cut generator may have a larger cut rejection rate while generating more cuts in the 30 rounds of

	Gen. cuts	BestGen	BestGenAway	Iter1	Iter2	Iter3	CglGomory	CglGomoryMod	CglLandP	CglLandPMod	CpxGMI	Cpx
BestGen	1222.47		-	+	+	=	=	+	-	-	+	+
BestGenAway	1246.43	+		+	+	+	+	+	=	-	+	+
Iter1	991.82	-	-		=	-	-	-	-	-	+	+
Iter2	992.28	-	-	=		-	-	-	-	-	+	+
Iter3	1227.72	=	-	+	+		=	+	=	-	+	+
CglGomory	822.48	=	-	+	+	=		+	=	-	+	+
CglGomoryMod	1106.29	-	-	+	+	-	-		-	-	+	+
CglLandP	1034.41	+	=	+	+	=	=	+		-	+	+
CglLandPMod	1599.87	+	+	+	+	+	+	+	+		+	+
CpxGMI	22.53	-	-	-	-	-	-	-	-	-		-
Cpx	126.95	-	-	-	-	-	-	-	-	-	+	

Table 10: Comparison of the total number of cuts. Column "Gen. cuts" gives the average number of generated cuts per dive.

cut generation. The comparisons are based on a Friedman test, where the performance measure is the total number of generated cuts. Results are reported in Table 10.

Additionally, we compare the cut rejection rate using a Friedman test for the six cut generators where this number can be computed. Results for this experiment can be found in Table 11.

In Table 10, we see that BestGenAway generates more cuts than all remaining cut generators except CglLandP and CglLandPMod. Interestingly, CglLandP has a limit of 50 cuts per pass, but on average over 30 passes it generates only 35% fewer cuts than the cut generator that generates the most. BestGenAway generates more cuts than BestGen, which in turn generates more than CpxGMI and Cpx. Thus, while still being as safe as CpxGMI, our generators generate more cuts. Table 10 shows that BestGenAway generates roughly 50 times as many cuts as CpxGMI while still yielding a comparable number of failures. However, we have no way of computing exactly the rejection rate of `Cplex`, as explained in Section 6.1. BestGenAway has a smaller cut rejection rate than BestGen according to Table 11, which confirms the results of Appendix D.

Finally, we compare the integrality gap closed by the cut generators. We use a Friedman test where the performance measure is the gap closed after applying cutting planes after each dive. The integrality gap for each dive is computed using the objective value of the best solution against which validity of the cuts is tested. This is an upper bound on the optimum value of the instance obtained after fixing random variables in Dive-and-Cut. The null hypothesis that all cut generators close the same amount of gap is rejected with a p-value of 0.0000. Pairwise comparisons are reported in Table 12. Note that with this test we do not have any quantitative information in the improvement on the gap closed: The Friedman test only tells us whether some algorithms consistently rank better than others. The performance difference could be

	Rej. rate	BestGen	BestGenAway	Iter1	Iter2	Iter3	CglGomoryMod	CglLandPMod
BestGen	37.21		+	-	-	=	-	=
BestGenAway	35.11	-		-	-	-	-	-
Iter1	57.40	+	+		=	+	=	+
Iter2	57.37	+	+	=		+	-	+
Iter3	37.26	=	+	-	-		-	=
CglGomoryMod	63.30	+	+	=	+	+		+
CglLandPMod	36.31	=	+	-	-	=	-	

Table 11: Comparison of the rejection rate. Column "Rej. rate" gives the average rejection rate (%).

negligible and could depend on other factors than the strength of the cuts. To eliminate this undesired effect, we perform a related experiment where we require a minimum difference of 1% integrality gap for a cut generator to be considered better than another. A Friedman test for this experiment yields exactly the same pairwise comparisons as in Table 12.

We observe that BestGenAway closes a similar amount of gap as BestGen and Iter3, and all three close more gap than all other generators except CglGomory, CglLandP and CglLandPMod. Recall, however, that the failure rate of CglGomory and CglLandPMod is significantly larger than those of BestGen, BestGenAway and Iter3. BestGenAway is comparable to CglLandPMod, thus it could be argued that it is stronger than BestGen and Iter3 (recall that the pairwise comparisons are not transitive).

Quantitative information on the gap closed is given in Table 13, listing the average integrality gap closed by each cut generator on all the dives. The dives are partitioned into 10 groups according to the largest gap closed by any of the generators, and we report the average for each group.

Table 13 shows that the strongest generators are CglLandP and CglLandPMod. In general the best 5 generators (CglLandP, CglLandPMod, BestGen, BestGenAway, Iter3) are very close, while the rest is clearly worse. BestGenAway and BestGen have similar performance, with only a slight advantage for BestGenAway, which explains why the Friedman test does not detect any significant difference. There is also a considerable improvement when going from Iter1 to Iter3. Our generators appear to be much stronger than CpxGMI and even Cpx. It is clear from these results that we can generate many more cuts than `Cplex`, closing more integrality gap while achieving a similar safety level. Of course, other computational considerations matter when deciding whether the generated cuts should be kept in the LP, such as LP resolve and root processing times.

Our final experiment consists in testing the cut generators BestGen, BestGenAway, CglLandP, CpxGMI, and Cpx on the set of instances labeled Extended Set, see Appendix B.2. These 69 instances were not used for OptimizeParameters as they yield fewer failures than

	BestGen	BestGenAway	Iter1	Iter2	Iter3	CglGomory	CglGomoryMod	CglLandP	CglLandPMod	CpxGMI	Cpx
BestGen		=	+	+	=	=	+	-	-	+	+
BestGenAway	=		+	+	=	=	+	-	=	+	+
Iter1	-	-		=	-	-	=	-	-	+	=
Iter2	-	-	=		-	-	=	-	-	+	=
Iter3	=	=	+	+		=	+	-	-	+	+
CglGomory	=	=	+	+	=		+	-	-	+	+
CglGomoryMod	-	-	=	=	-	-		-	-	+	=
CglLandP	+	+	+	+	+	+	+		=	+	+
CglLandPMod	+	=	+	+	+	+	+	=		+	+
CpxGMI	-	-	-	-	-	-	-	-	-		-
Cpx	-	-	=	=	-	-	=	-	-	+	

Table 12: Comparison of the gap closed.

Gap bin	BestGen	BestGenAway	Iter1	Iter2	Iter3	CglGomory	CglGomoryMod	CglLandP	CglLandPMod	CpxGMI	Cpx
0%-10%	2.50	2.59	2.01	2.01	2.52	2.72	2.01	2.88	2.86	1.46	2.20
10%-20%	10.21	10.47	7.95	7.95	10.19	10.41	7.79	11.21	11.30	4.74	7.43
20%-30%	18.25	18.64	15.10	15.12	18.43	18.13	14.26	20.35	20.00	7.71	12.02
30%-40%	26.67	27.16	21.96	21.95	26.74	26.65	21.80	29.04	28.10	10.88	18.85
40%-50%	34.31	34.67	28.46	28.48	34.36	35.19	30.14	37.23	36.79	14.00	26.02
50%-60%	41.58	42.08	34.59	34.58	41.76	41.41	37.09	45.51	44.02	15.94	30.30
60%-70%	51.26	52.07	39.82	39.81	51.27	47.91	42.88	55.73	54.10	19.43	34.06
70%-80%	60.92	61.11	42.25	42.27	61.09	52.82	49.31	64.79	65.12	23.12	42.53
80%-90%	71.14	72.15	48.68	48.69	71.36	56.23	59.30	76.23	75.98	28.90	53.56
90%-100%	87.32	89.06	62.60	62.59	87.33	75.71	74.98	93.39	93.40	31.07	34.44

Table 13: Average gap closed by the cut generator on the dives where the maximum gap closed falls in the bin reported in the first column (label "Gap bin").

	Failures				# distinct instances			
	T. 1	T. 2	T. 3	Tot.	T. 1	T. 2	T. 3	Tot.
BestGen	0	33	17	50	0	4	5	9
BestGenAway	0	24	24	48	0	5	5	10
CglLandP	0	49	170	219	0	10	7	12
CpxGMI	35	2	0	37	6	2	0	8
Cpx	400	2	0	402	21	2	0	23

Table 14: Number of failures recorded on Extended Set and number of distinct instances with at least one failure.

instances in Failure Set. We perform 300 dives per instance. The number of recorded failures is reported in Table 14.

BestGen, BestGenAway and CglLandP do not generate Type 1 failures, performing better than CpxGMI and Cpx in this regard. However, they generate more Type 2 and Type 3 failures. We remark that 29 of the 33 failures of Type 2 for BestGen and 19 of the 24 failures of Type 2 for BestGenAway are recorded on the same instance (`rococoC10-001000`). CglLandP is safe on most instances but shows again poor performance on a few problems. 41 out of the 49 Type 2 failures occur on three instances only (`lectsched-4-obj`, `momentum1`, `momentum2`). On four instances (`30n20b8`, `momentum2`, `net12`, `rocII-4-11`), the 300 dives are not carried out to completion because of the early stopping criterion (more than 30 Type 3 failures). It is clear that our BestGen generators are very effective in rejecting potentially invalid cuts, whereas CpxGMI and Cpx are more effective in rejecting cuts that may slow down or create troubles in the LP solution process. This is not surprising as these aspects are not the primary focus of our investigation, while this is obviously an important practical consideration.

We draw some conclusions. The results of OptimizeParameters depend on the data set and on the machine. Our optimization algorithm was run on the Condor grid. To validate the results that it produced, we used a machine with a different architecture (64 bit instead of 32 bit, different compiler options) and different instances. We are able to confirm that our results are still meaningful. The best cut generation parameters obtained in Section 6.4 yield a cut generator with low failure and rejection rates. The results of Appendix D also seem to carry over, and we were able to modify one parameter of our cut generator to decrease rejection rate while keeping the same safety level. It seems that the generators that generate more cuts close more gap. This suggests that either increasing the number of cuts is a good way to increase the gap closed, or that reducing the rejection rate allows stronger cuts to pass the numerical checks for rejection. Selecting the best cutting planes among those that are deemed to be safe is an important topic for future research.

References

[1] T. Achterberg. SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[2] T. Achterberg, T. Koch, and A. Martin. MIPLIB 2003. *Operations Research Letters*, 34(4):361–372, 2006.

[3] N. Ailon. Aggregation of partial rankings, p-ratings and top-m lists, *Algorithmica*, 57:284–300, 2010.

[4] D. L. Applegate, W. Cook, S. Dash, and D. G. Espinoza. Exact solutions to linear programming problems. *Operations Research Letters*, 35(6):693–699, 2007.

[5] E. Balas, S. Ceria, G. Cornuéjols, and N. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19(1):1–9, 1996.

[6] R. Bixby and E. Rothberg. Progress in computational mixed integer programming – a look back from the other side of the tipping point. *Annals of Operations Research*, 149(1):37–41, 2007.

[7] R. E. Bixby, S. Ceria, C. M. McZeal, and M. W. P. Savelsbergh. An updated mixed integer programming library: MIPLIB 3.0. *Optima*, 58:12–15, 1998.

[8] COIN-OR Branch-and-Cut 2.7 stable r1676. `https://projects.coin-or.org/Cbc/`

[9] COIN-OR Linear Programming 1.14 stable r1753. `https://projects.coin-or.org/Clp/`

[10] COIN-OR Cut Generation Library 0.57 stable r1033. `https://projects.coin-or.org/Cgl/`

[11] W. J. Conover. *Practical Nonparametric Statistics, 3rd edition.* Wiley, 1999.

[12] W. Cook, S. Dash, R. Fukasawa, and M. Goycoolea. Numerically safe Gomory mixed-integer cuts. *INFORMS Journal of Computing*, 21(4):641–649, 2009.

[13] W. Cook, T. Koch, D. E. Steffy, and K. Wolter. An exact rational mixed-integer programming solver. In O. Günlük and G. J. Woeginger, editors, *Proceedings of IPCO 2011*, volume 6655 of *Lecture Notes in Computer Science*, 104–116, Berlin Heidelberg, 2011. Springer-Verlag.

[14] D. G. Espinoza. *On Linear Programming, Integer Programming and Cutting Planes.* PhD thesis, School of Industrial and Systems Engineering, Georgia Institute of Technology, March 2006.

[15] R. E. Gomory. An algorithm for the mixed-integer problem. Technical Report RM-2597, RAND Corporation, 1960.

[16] H.-M. Gutmann. A radial basis function method for global optimization. *Journal of Global Optimization*, 19:201–227, 2001. 10.1023/A:1011255519438.

[17] T. Hemker. Derivative free surrogate optimization for mixed-integer nonlinear black-box problems in engineering. Master's thesis, Technischen Universität Darmstadt, December 2008.

[18] *IBM ILOG CPLEX 12.2 User's Manual.* IBM ILOG, Gentilly, France, 2011.

[19] W. Kahan. Pracniques: further remarks on reducing truncation errors. *Communications of the ACM*, 8(1), 1965.

[20] 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(2):103–163, 2011.

[21] M. Litzkow, M. Livny, and M. Mutka. Condor – a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing System*, pages 104–111, 1998.

[22] F. Margot. Testing cut generators for mixed-integer linear programming. *Mathematical Programming Computation*, 1(1):69–95, 2009.

[23] A. Neumaier and O. Shscherbina. Safe bounds in linear and mixed-integer linear programming. *Mathematial Programming A*, 99:283–296, 2004.

[24] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[25] R. Regis and C. Shoemaker. Improved strategies for radial basis function methods for global optimization. *Journal of Global Optimization*, 37:113–135, 2007. 10.1007/s10898-006-9040-1.

[26] D. J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures.* Chapman & Hall/CRC, 2007.

[27] A. van Zuylen and D.P. Williamson. *Deterministic algorithms for rank aggregation and other ranking and clustering problems.* Mathematics of Operations Research, 34:594–620, 2009.

A Statistical tests

In this section, we briefly cover the application of statistical tests to the analysis of results of $c \geq 2$ algorithms on r instances. In the usual presentation of these tests in statistics textbooks [11, 26], algorithms are referred to as "treatments" and instances are referred to as "blocks" or "subjects". We keep this tradition and mention treatments and blocks when applying the test. For a complete presentation of these tests, see [11, 26] or any reference statistics book.

The Friedman and Paired Sign statistical tests used in the analysis of our results are non-parametric tests, i.e., tests that do not assume any form for the distribution of the population data. Each test assumes that a null hypothesis is true and gives the probability (the p-value) of obtaining a test statistic at least as extreme as the one observed. The p-value is then compared to a given α value (we use $\alpha = 0.05$ in this paper) for a test with significance $(1 - \alpha)$. If the

p-value is smaller than α, the null hypothesis is rejected, as the observed results have a low probability of occurring if the null hypothesis were true.

Friedman test:

(i) Application: 1) each block receives all c treatments; 2) the outcome of using a treatment on a block is a real value called the performance of the treatment on that block.

(ii) Null hypothesis: The c treatments have similar performances; Alternative hypothesis: There is a difference in performance between some of the treatments.

(iii) Assumptions: 1) The set of r blocks is a random sample of the population; 2) The r c-variate random variables are mutually independent; 3) The outcome is a continuous random variable.

Assumption 2) requires that the results within one block do not influence the results within the other blocks. Assumption 3) is often not adhered to when applying the test in practice [26]. We use the Iman-Davenport's variant of the Friedman test, known to be more accurate than the original version of the test [11]. Note that the Friedman test is based only on the ranking of the performance of the treatments on each block. It does not take into account the magnitude of the differences in performance.

When the Friedman test rejects the null hypothesis, an additional statistic can be used to test if treatment A has a better performance than treatment B, for any A, B among the tested treatments. The result of all pairwise comparisons is not always a total order, as transitivity is not guaranteed.

We use a 2-dimensional table for displaying the result of pairwise comparisons. Rows and columns of the table corresponds to the tested treatments. Entry in cell in row i and column j is a "+" sign (resp. "−" sign) if the treatment corresponding to row i has larger (resp. smaller) performance value than the one corresponding to column j at the given significance level. A "=" means that no difference could be detected at the given significance level. All comparisons in this paper are carried out at a significance level of 95%.

When we say that "we apply a Friedman test on the failure rate", it means that treatments are algorithms, each block is an instance, and the performance of the algorithm on the instance is the average failure rate over all dives on that instance. A similar interpretation holds for the statement "we apply a Friedman test on measure M", where M is averaged on all dives associated with a particular instance.

In Section C.4, we use a paired sign test.

Paired sign test:

(i) Application: 1) Each block receives two treatments; 2) the outcome of using a treatment on a block is a real value called the performance of the treatment on that block.

(ii) Null hypothesis: The two treatments have similar performances; Alternative hypothesis: There is a difference in performance between the two treatments.

(iii) Assumptions: 1) The set of r blocks is a random sample of the population; 2) The r bivariate random variables are mutually independent; 3) The measurement scale is at least ordinal within each pair.

Assumption 3) means that, for each block, we can determine if the outcome of the first treatment is larger, smaller or equal to the outcome of the second treatment.

We note that, for all tests, the assumption of having a random sample of the population of all possible blocks cannot be verified in our applications of these tests. We can only hope that blocks (i.e., instances in our applications) used are varied enough to be representative of the set of all instances creating difficulties for GMI cut generators. Even if this assumption is not verified, the statistical analysis based on these tests is more convincing than a simple eyeballing of the results or an estimation based on a simple averaging of the results.

B Dive-And-Cut preprocessing phase

In this section, we describe the algorithm GenerateSolutions (see Algorithm 4) used to generate a set S of $(\epsilon_{\text{abs}}, \epsilon_{\text{rel}}, 0)$-feasible solutions for a problem instance P. GenerateSolutions applies a Branch-and-Cut solver $\mathcal{B}$ to the instance at hand P and acts whenever the solver discovers an integer solution. In `Cplex`, this can be achieved by a call to the function `incumbentcallback()`.

Whenever a candidate solution $\tilde{x}$ is discovered, GenerateSolutions rounds each integer variable to the nearest integer, and checks the feasibility of this rounded solution. If it fails to satisfy the constraints within an absolute feasibility tolerance ϵ_{abs} and a relative feasibility tolerance ϵ_{rel}, it is rejected.

If $\tilde{x}$ is not rejected, we use a rational LP solver to generate provably feasible solutions to the problem P that have value $\tilde{x}_i$ for all integer variables x_i. First, we seek a provably feasible solution x^* that minimizes the original objective function. If we are able to compute x^* , it is added to the set S of feasible solutions. If its Euclidean distance to $\tilde{x}$ is less than ϵ_{rel}, we are done and $\tilde{x}$ is accepted. Otherwise, we seek the point x' in the feasible region of the LP that is closest to $\tilde{x}$ in ℓ_1-norm. If $\|\tilde{x} - x'\|_2 \leq \texttt{ZERO}$, these two points are essentially the same and x' is added to S. If $\|\tilde{x} - x'\|_2 \leq \epsilon_{\text{rel}}$, $\tilde{x}$ is "feasible enough" according to our criteria and is added to S. In both cases, $\tilde{x}$ is accepted. Otherwise, it is rejected.

Requiring the existence of a feasible point x' that is not too far from $\tilde{x}$ is motivated by the following simple example. Suppose $\tilde{x}$ satisfies the relative feasibility tolerance and there are two rows of (MILP), say rows 1 and 2, such that $(b_i - a^i\tilde{x})/\|a^i\| = \epsilon_{\text{rel}}$ for $i = 1, 2$. Now consider the constraint $(a^1 + a^2)x \geq b_1 + b_2$. It is redundant for (LP), but it is not satisfied by $\tilde{x}$ according to the relative feasibility tolerance unless a^1 and a^2 are parallel. Indeed,

$$\frac{(b_1 + b_2) - (a^1 + a^2)\tilde{x}}{\|a^1 + a^2\|_2} = \frac{\epsilon_{\text{rel}}(\|a^1\|_2 + \|a^2\|_2)}{\|a^1 + a^2\|_2} \geq \epsilon_{\text{rel}}$$

and the inequality is strict unless a^1 and a^2 are parallel. In other words, even though the Euclidean distance between $\tilde{x}$ and each violated constraint is at most ϵ_{rel}, the distance between $\tilde{x}$ and the closest point in the feasible set of the LP could be much larger than ϵ_{rel}. In this situation, a cutting plane that cuts off $\tilde{x}$ should not be marked as invalid. Therefore we do not want $\tilde{x}$ in our set of feasible solutions.

Note that GenerateSolutions uses the ℓ_1-norm for the problem of finding x' close to $\tilde{x}$ since this can be formulated as an LP using the usual reformulation: The objective function $\min_{x \in \mathbb{R}^n} \|\tilde{x} - x\|_1$ can be expressed in linear form by adding n extra variables and $2n$ constraints: $\min_{w,x \in \mathbb{R}^n} \{\mathbf{1}_n w \mid \tilde{x} - x \leq w, x - \tilde{x} \leq w\}$. Since the ℓ_1-norm overestimates the ℓ_2-norm this

Algorithm 4 GenerateSolutions. The solution of an LP is the symbol NIL if the LP is infeasible.

Input: problem $P = (A, b, c)$, Branch-and-Cut solver $\mathcal{B}$, rational LP solver $\mathcal{R}$, tolerances $\epsilon_{\text{abs}}, \epsilon_{\text{rel}}$
Output: set S of $(\epsilon_{\text{abs}}, \epsilon_{\text{rel}}, 0)$-feasible solutions
Apply $\mathcal{B}$ to P
for (every candidate solution $\tilde{x}$ discovered) **do**
  Set feasible $\leftarrow$ `false`
  Set $\tilde{x}_i \leftarrow \lfloor \tilde{x}_i \rceil \; \forall i \in N_I$
  if $(\max_{i \in [m]}\{b_i - a^i \tilde{x}\} \leq \epsilon_{\text{abs}})$ **and** $(\max_{i \in [m]}\{(b_i - a^i \tilde{x})/\|a^i\|_2\} \leq \epsilon_{\text{rel}})$ **then**
    Compute $x^* = \arg\min\{c^\top x \mid Ax \geq b, x \geq 0, x_i = \tilde{x}_i \; \forall i \in N_I\}$ with $\mathcal{R}$
    if $(x^* \neq \text{NIL})$ **then**
      Set $S \leftarrow S \cup \{x^*\}$
      if $(\|x^* - \tilde{x}\|_2 \leq \epsilon_{\text{rel}})$ **then**
        Set feasible $\leftarrow$ `true`
      else
        Compute $x' = \arg\min\{\|\tilde{x} - x\|_1 \mid Ax \geq b, x \geq 0, x_i = \tilde{x}_i \; \forall i \in N_I\}$ with $\mathcal{R}$
        if $(\|\tilde{x} - x'\|_2 \leq$ `ZERO`$)$ **then**
          Set $S \leftarrow S \cup \{x'\}$
          Set feasible $\leftarrow$ `true`
        else if $(\|\tilde{x} - x'\|_2 \leq \epsilon_{\text{rel}})$ **then**
          Set $S \leftarrow S \cup \{\tilde{x}\}$
          Set feasible $\leftarrow$ `true`
  if (feasible = `true`) **then**
    Report to $\mathcal{B}$ that $\tilde{x}$ is accepted
  else
    Report to $\mathcal{B}$ that $\tilde{x}$ is rejected

guarantees that x' is also at most ϵ_{rel} away from $\tilde{x}$ in ℓ_2-norm. As a consequence, if such an x' exists, the relative violation of any nonnegative linear combinations of rows of $Ax \geq b$ by $\tilde{x}$ is at most ϵ_{rel}.

GenerateSolutions accomplishes several desirable goals. First, it does not modify the original instances. Second, given enough time, it finds an optimal solution (feasible according to our criteria) and adds it to the set S. Third, it potentially returns a large and diverse set of feasible solutions, with different values for the integer variables. This is valuable for our purposes, as this allows testing the validity of cutting planes with respect to solutions in different parts of the feasible region.

However, GenerateSolutions has also some drawbacks. First, the generated solutions are the nearest floating point representation (component-wise) of the feasible solutions computed in rational numbers. Therefore, they could be infeasible when the left-hand side of the constraints is computed in finite precision. Second, the generated solutions might be cut off in terms of absolute violation by linear combinations of the problem constraints.

We stress that, in our computational tests, the first drawback occurred only once, in a specific case that is discussed in Appendix C.1. That particular instance was discarded. Using only a relative violation tolerance would address the second issue, as explained above. However,

using also an absolute feasibility tolerance makes sense, because all available Branch-and-Cut codes have an absolute feasibility tolerance $\epsilon_{\text{abs}} > 0$.

B.1 Implementation

GenerateSolutions's implementation is tied to `Cplex` 12.2 because it uses advanced Branch-and-Cut functions. In particular, we use the `incumbentcallback()` function to intercept the discovery of feasible solutions and run our implementation of Algorithm 4. The feasibility and integrality tolerances for `Cplex` were set to 10^{-9}, the smallest feasibility tolerance allowed by `Cplex`[2]. The rational LP solver used by GenerateSolutions is `QSopt_ex` [4].

We parametrize the Branch-and-Cut algorithm of `Cplex` as follows. The integrality gap allowed for optimality is set to 0 (both relative and absolute). The time limit is set to 6 hours, the number of parallel threads to 2, and `SolutionPoolIntensity` is set to 2. We disable rescaling of the LP matrix, in order to avoid discovery of solutions that are infeasible for the unscaled problem (`Cplex` error code: `OptimalInfeas`). If the instance is solved to optimality before the 6 hours time limit, we call the `CPXpopulate()` procedure to generate more solutions until the time limit is hit or an additional 100 solutions are found. The strategy for replacing solutions in the pool when it is full is set to `Diversify`[3].

The experiments in this section are run on a machine equipped with an `AMD Opteron 4176 HE` processor clocked at 2.4GHz and 48GB RAM, running Linux.

B.2 Computational experiments on MIPLIB instances

Our initial test set contains all instances from MIPLIB3 [7], MIPLIB2003 [2], and the Benchmark set of MIPLIB2010 [20] beta (downloaded March 2011) for a total of 169 instances. Ten instances are eliminated, as GenerateSolutions does not find any feasible solution for them. We are left with the 159 instances listed in Table 15. On the `satellites1-25` instance, more than 10^6 solutions are found during the initial Branch-and-Cut; we keep the best solution found and an additional 1499 solutions selected at random.

The vast majority of the solutions found by GenerateSolutions are feasible with very small tolerances. In Figure 15 we report a histogram of the maximum absolute violation for the generated solutions, computed as $\max_{i\in[m]}\{b_i - a^i x^*\}$. The maximum violation for 62.99% of the solution is 0, i.e., they are feasible even when checked using finite precision and compensated summation [19]. We note that the use of compensated summation here has a small effect, increasing the fraction of solutions with a 0 violation by 0.27%.

For 98.8% of the solutions the maximum violation is smaller than 10^{-11}. We eliminate the solutions for which the maximum violation is 10^{-9} or more.

For 152 instances out of 159 the optimal solution is known, therefore we can analyze the objective value of the solutions found by GenerateSolutions on these instances. We compute the relative distance from the known optimum f^* of the best solution value $\bar{f}$ among the solutions accepted by GenerateSolutions, using the formula: $(\bar{f} - f^*)/|f^*|$ if $f^* \neq 0$, or $(\bar{f} - f^*)$ otherwise. In 138 cases out of 152, this value is less than 0.5%, hence we found a solution that satisfies our criteria and can be considered optimal. For the remanining 14 cases,

[2]The smallest integrality tolerance is 0.0, but the user manual warns that such a small tolerance may not always be attainable.

[3]This setting should have no effect on the outcome since we intercept each solution discovered and store it in a separate pool; we mention it only for completeness.

10teams	103	*flugpl*	105	*momentum1*	27	*pp08aCUTS*	120
30n20b8	17	**gen**	44	*momentum2*	34	*pp08a*	117
a1c1s1	65	**gesa2**	97	momentum3	1	protfold	6
acc-tight5	52	**gesa2_o**	101	msc98-ip	11	*pw-myciel4*	52
aflow30a	107	**gesa3**	112	mspp16	56	**qiu**	125
aflow40b	113	**gesa3_o**	101	mzzv11	104	*qnet1*	110
air03	103	**glass4**	179	mzzv42z	104	*qnet1_o*	104
air04	109	**gmu-35-40**	71	n3div36	75	rail507	58
air05	105	**gt2**	2	n3seq24	7	*ran16x16*	69
app1-2	8	**harp2**	129	*n4-3*	95	*rd-rplusc-21*	2
arki001	58	*iis-100-0-cov*	53	*neos-1109824*	58	*reblock67*	27
atlanta-ip	32	*iis-bupa-cov*	56	neos-1337307	43	**rgn**	102
bab5	67	*iis-pima-cov*	56	*neos-1396125*	83	*rmatr100-p10*	56
beasleyC3	69	*khb05250*	105	**neos13**	67	*rmatr100-p5*	60
bell3a	102	*l152lav*	106	neos-1601936	67	*rmine6*	72
bell5	102	*lectsched-4-obj*	59	**neos18**	57	*rocII-4-11*	19
biella1	65	*liu*	315	neos-476283	33	*rococoC10-001000*	74
bienst2	86	*lseu*	105	*neos-686190*	60	**roll3000**	156
binkar10_1	55	*m100n500k4r1*	5	neos-849702	304	**rout**	125
blend2	115	**macrophage**	66	*neos-916792*	73	satellites1-25	1500
bley_xl1	56	*manna81*	101	neos-934278	8	**set1ch**	127
cap6000	111	map18	128	*net12*	106	*seymour*	113
core2536-691	11	map20	127	netdiversion	54	sp97ar	122
cov1075	57	*markshare1*	117	**newdano**	73	*sp98ic*	68
csched010	98	*markshare2*	117	**noswot**	104	*sp98ir*	64
dano3mip	10	*mas74*	159	**ns1688347**	6	**stein27**	102
danoint	2	**mas76**	137	ns1758913	65	*stein45*	103
dcmulti	117	**maxgasflow**	1236	*ns1830653*	189	*swath*	124
dfn-gwin-UUM	72	mcsched	70	*nsrand-ipx*	114	t1717	16
dln	52	**mik.250-1-100.1**	54	nw04	105	tanglegram1	11
ds	23	*mine-166-5*	70	opm2-z7-s2	65	*tanglegram2*	56
egout	3	*mine-90-10*	71	**opt1217**	2	**timtab1**	132
eil33.2	66	*misc03*	102	**p0033**	104	*timtab2*	170
eilB101	95	**misc06**	107	**p0201**	103	**tr12-30**	217
enigma	3	*misc07*	105	*p0282*	106	unitcal_7	133
enlight13	22	**mitre**	82	**p0548**	2	**vpm1**	106
ex9	23	**mkc**	123	**p2756**	109	**vpm2**	113
fast0507	108	*mod008*	102	*pg5_34*	62	vpphard	19
fiber	106	*mod010*	4	**pigeon-10**	2	*zib54-UUE*	70
fixnet6	111	**modglob**	203	*pk1*	108		

Table 15: Number of feasible solutions found by GenerateSolutions on each instance. Instances in bold face are in the Failure Set. Instances in italics are in the Extended Set.

the average relative distance is 231.02%. This large value is due to a small number of instances where GenerateSolutions returns solutions of poor quality: `markshare1` (200% away from the optimum), `markshare2` (2100% away from the optimum), `satellites1-25` (700% away from the optimum).

The 51 instances in bold face in Table 15 form the Failure Set. The selection is discussed in Section 4.2. The 69 instances in italics in Table 15 form the Extended Set used to validate our results on the Failure Set. The Extended Set contains all instances of Table 15 that are not part of Failure Set and for which 300 dives require less than 24 hours of CPU time and generate less than 30 Type 3 failures.

The instances, solutions and code used in this paper are available on the website of the authors. (Note for reviewers: website under construction at the time of submission.)

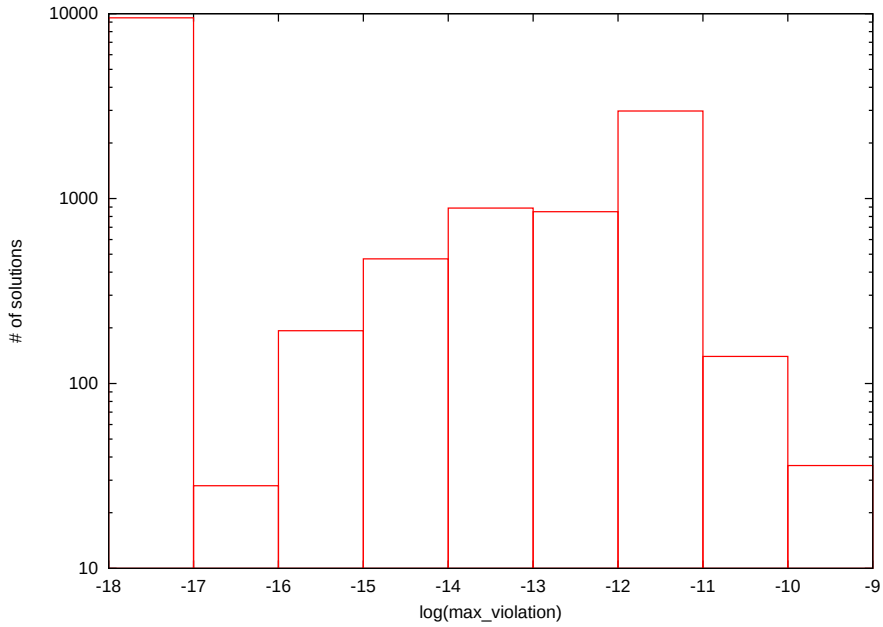

Figure 15: Maximum absolute violation of the solutions found by GenerateSolutions.

C Detailed comparison between Dive-and-Cut and RandomDives

This section is a summary of empirical tests comparing the results obtained by Dive-and-Cut and RandomDives. We are interested in investigating if both procedures yield similar conclusions or not, if failures are distributed in similar fashion over all instances, and to compare the CPU time required.

C.1 Instance selection and testing configuration

For this comparison, our test set is the full MIPLIB3 except the instance `rentacar` that was discarded for numerical reasons [22]. For Dive-and-Cut we use the original MIPLIB3 instances. For RandomDives we use the modified instances obtained through its preprocessing phase, using the best solution in the sets of feasible solutions generated by GenerateSolutions (see Appendix B) as starting feasible solution. It is worth noting that for 44 instances out of 56, this initial solution used is accepted as feasible without performing any modification to the instance. For the remaining instances, the preprocessing phase of RandomDives typically requires few iterations, and only for six instances it reaches the maximum number of iterations without finding a solution and instance pair that satisfies the criteria for feasibility. We keep these 6 instances in the test set nonetheless as the maximum constraint violation of the reported solution is smaller than 10^{-14}.

We use the code from [22] for our experiments. Feasibility is only tested through the absolute violation of the cuts. We also tested a variant of the code using compensated summation [19] to compute the left-hand side of inequalities. (Compensated summation ensures that the numerical

error is independent of the number of additions.) Since the results of this variant and the original code differ only very slightly, we only report on the original code.

We decided to use the same cut management policy in both algorithms. After applying a round of cuts, we remove all cutting planes that have been inactive for two consecutive rounds. This differs from the original implementation of RandomDives.

The experiments are run on an `AMD Opteron 4176 HE` @2.4Ghz with 48GB of RAM. Note that we do not use the Condor pool in this section, so that we can compare running times of the algorithms. For both algorithms, the time limit for each dive was set to 300 seconds and we perform 200 dives per instance. We test three cut generators, that we call CgBase, CgDyn and CgStd. Their parameters are given in Table 16. CgBase is the most basic cut generators that can be designed given our parameters, as it accepts all cuts generated from rows whose basic variable has a fractionality exceeding the integrality tolerance. CgDyn applies Dynamism Check on top of CgBase and CgStd is a more sophisticated cut generator that performs several post-processing steps using typical parameter values found in COIN-OR `Cgl` cut generators.

Parameter	CgBase	CgDyn	CgStd
`AWAY`	10^{-9}	10^{-9}	10^{-4}
`MIN_VIOL`	$-\infty$	$-\infty$	10^{-7}
`MAX_SUPP_ABS`	∞	∞	1000
`MAX_SUPP_REL`	∞	∞	0.2
`MAX_DYN`	∞	10^{+8}	10^{+8}
`MAX_DYN_LUB`	∞	∞	∞
`EPS_ELIM`	0	0	10^{-20}
`LUB`	∞	∞	∞
`EPS_COEFF`	0	0	10^{-12}
`EPS_COEFF_LUB`	0	0	0
`EPS_RELAX_ABS`	0	0	10^{-7}
`EPS_RELAX_REL`	0	0	0

Table 16: Parameters defining the cut generators CgBase, CgDyn, CgStd.

Some instances are removed from the test set. The `arki001` instance is removed because its optimal solution, chosen as target for the dive for RandomDives, turns out to be infeasible when compensated summation is not used to compute the left-hand side of the constraints. Instance `swath` is also removed because of numerical problems encountered by `Cplex` during the solution of some of the LPs[4]. Instances `dano3mip` and `nw04` were removed because the total computing time for 200 dives with at least one algorithm was more than 24 hours. The final test set thus contained 52 problems. The total number of failures recorded by each method is reported in Table 17.

[4] `Cplex` reported the error "Singular basis" and stopped the execution.

	CgBase			CgDyn			CgStd		
Method	T. 1	T. 2	T. 3	T. 1	T. 2	T. 3	T. 1	T. 2	T. 3
RandomDives	711	4108	1320	447	3090	919	5	1414	278
Dive-and-Cut	94	41	434	34	6	221	0	14	57

Table 17: Number of failures recorded by RandomDives and Dive-and-Cut on MIPLIB3 with three different cut generators.

C.2 Failure distribution

In the next experiment, we fix the cut generator to either CgBase, CgDyn, or CgStd and compare the number of failures recorded by the two testing procedures using a Friedman test. In each case, the blocks of the experiment are the number of failures for each instance and the treatments are the two testing procedures (RandomDives and Dive-and-Cut). The null hypothesis is always rejected with a p-value of 0.0000. Pairwise comparisons yield that RandomDives generates significantly more failures than Dive-and-Cut.

We now look into the failure distribution over all instances. RandomDives generates significantly more failures than Dive-and-Cut. The ratio of Type 1 to Type 2 failures is much larger for Dive-and-Cut than for RandomDives (see Table 17). The overall number of Type 3 failures is smaller for Dive-and-Cut. This suggests that RandomDives puts a lot of stress on the LP solver, increasing the difficulty of resolving the LPs until the LP solver runs into numerical problems or times out.

C.3 Comparison of safety rankings

We now investigate if the rankings obtained from the two testing procedures are similar or not. We compare the number of failures recorded by each procedure by the three cut generators using a Friedman test (see Appendix A).

The blocks of the experiment are the number of failures on each instance and the treatments are the three cut generators. The null hypothesis is rejected with a p-value 0.0000 for both procedures. We then perform a pairwise comparison of the cut generators.

Results are reported in Tables 18 and 19.

	CgBase	CgDyn	CgStd
CgBase		+	+
CgDyn	-		+
CgStd	-	-	

Table 18: Comparison of CgBase, CgDyn and CgStd using RandomDives.

According to RandomDives, CgBase generates significantly more failures than CgDyn which in turn generates significantly more failures than CgStd. For Dive-and-Cut, CgBase generates significantly more failures than CgDyn, but no difference is detected between CgDyn and CgStd. We stress that the Friedman test does not take into account differences in the magnitude of the performance measure. This explains why CgDyn is considered as safe as CgStd according to Dive-and-Cut. Indeed, 200 of the 221 Type 3 failures recorded by

	CgBase	CgDyn	CgStd
CgBase		+	+
CgDyn	-		=
CgStd	-	=	

Table 19: Comparison of CgBase, CgDyn and CgStd using Dive-and-Cut.

	RandomDives	Dive-and-Cut
CgBase	1281.14	304.03
CgDyn	926.92	124.90
CgStd	754.92	229.30

Table 20: Geometric mean of the CPU time for each cut generator and testing algorithm.

CgDyn occur on the same instance, and the number of instances on which CgDyn generates more failures than CgStd is small. We conclude that the two testing procedures induce similar ranking in the cut generators.

C.4 CPU times

Finally, we look at CPU times. We consider the total CPU time for each instance, which is dominated by the time required to perform 200 dives. The geometric mean of these CPU times taken by each of the testing procedures on all instances is reported in Table 20. For a given cut generator, we use a paired sign test of the null hypothesis "the difference in the paired samples (CPU time taken by RandomDives minus CPU time taken by Dive-and-Cut) has median no greater than zero". For all cut generators, the null hypothesis is rejected with a p-value of 0.0000. This shows that Dive-and-Cut is consistently faster than RandomDives. Table 20 show that Dive-and-Cut is much faster on average.

D Parameter Sensitivity

This section contains a sensitivity analysis of the failure and rejection rates when small changes are applied to the parameters of the cut generator #1 from Table 7, denoted by g^* in this section.

We start with the parametrization p^* of g^* given in Table 7 and we investigate changes in a single parameter at a time. For each parameter i, we want to find a range R_i such that changing the value in p^* of p_i^* to any value in R_i yields a generator such that both a Friedman test on the rejection rate and a Friedman test on the failure rate deem the generator comparable to g^*. We then determine the variation of the failure rate and cut rejection rate for values slightly outside of R_i.

The algorithm we use is a simple sampling algorithm. It maintains an interval $\underline{R_i}$ containing the convex hull of all tested values of p_i for which the modified generator is deemed similar to g^*. The initial interval $\underline{R}_i$ is the single point p^*. The algorithm also maintains the smallest interval $\overline{R}_i$ containing $\underline{R}_i$ and one tested point on each side of $\underline{R}_i$ for which the modified generator is deemed different from g^*. If no such point has been tested yet, the initial value of the

corresponding endpoint of $\overline{R}_i$ is kept. The initial interval $\overline{R}_i$ is chosen as the initial range of parameter i used in Section 6.4. However, if we already know from previous experiments one value p_i smaller (resp. larger) than p_i^* such that the corresponding cut generator is not equivalent to g^*, we remove all values smaller (resp. larger) than p_i from the initial interval $\overline{R}_i$.

At each iteration, we select k parameter values $p_1, \ldots, p_k \in \overline{R}_i \setminus \underline{R}_i$, compare the corresponding cut generators with g^* and update both $\underline{R}_i$ and $\overline{R}_i$. The algorithm continues until we are satisfied with the gap between $\underline{R}_i$ and $\overline{R}_i$. We use $k = 10$, except when the gap between $\underline{R}_i$ and $\overline{R}_i$ is very small. In that case, we use a smaller value for k. The tested values $p_1, \ldots, p_k$ are equally spaced in $\overline{R}_i \setminus \underline{R}_i$, with $k/2$ points on each side of $\underline{R}_i$. We use a logarithmic scale for all parameters that have a logarithmic scale in Table 3.

A similar algorithm is used to compute intervals $\underline{F}_i$ and $\overline{F}_i$ such that all generators in $\underline{F}_i$ have a safety level similar to g^* according a Friedman test on the failure rate, without regard to the rejection rate. The interval $\overline{F}_i$ contains $\underline{F}_i$ and one tested point on each side of $\underline{F}_i$ for which the modified generator is deemed different to g^* in term of safety.

Results are reported in Table 21. The columns are as follows. "p^*" contains the optimal value of the parameter in cut generator #1 of Table 7. For the parameters that are not reported in Table 7, we use the value corresponding to disabling the cut modification or numerical check that employs the parameter. For the `_LUB` parameters, we set the value to the corresponding non-`LUB` parameter. In this table, `LUB` was set to 10^3 (see Section 5.1). Columns "$\underline{R}$", "$\overline{R}$", "$\underline{F}$", and "$\overline{F}$" contain the four intervals of interest for each parameter. "Increase" indicates the direction along which the cut rejection rate increases; the column contains "—" if our experiments did not provide enough information for computing the direction.

Parameter	p^*	$\underline{R}$	$\overline{R}$	Increase	$\underline{F}$	$\overline{F}$
AWAY	10^{-2}	$[9.9 \cdot 10^{-3}, 1.1 \cdot 10^{-2}]$	$[9.8 \cdot 10^{-3}, 1.2 \cdot 10^{-2}]$	$\Longrightarrow$	$[5.0 \cdot 10^{-3}, 10^{-1}]$	$[2.0 \cdot 10^{-3}, 1]$
EPS_COEFF	10^{-11}	$[10^{-14}, 10^{-8}]$	$[10^{-15}, 10^{-7}]$	$\Longleftarrow$	$[10^{-18}, 10^{-6}]$	$[0, \infty]$
EPS_RELAX_ABS	10^{-11}	$[10^{-12}, 2.0 \cdot 10^{-11}]$	$[10^{-13}, 4.0 \cdot 10^{-11}]$	$\Longrightarrow$	$[10^{-14}, 10^{-9}]$	$[10^{-15}, 10^{-8}]$
EPS_RELAX_REL	10^{-13}	$[10^{-14}, 10^{-12}]$	$[10^{-15}, 10^{-11}]$	$\Longrightarrow$	$[10^{-18}, 10^{-10}]$	$[0, 10^{-9}]$
MAX_DYN	10^{6}	$[9.2 \cdot 10^{5}, 1.1 \cdot 10^{6}]$	$[8.4 \cdot 10^{5}, 1.2 \cdot 10^{6}]$	$\Longleftarrow$	$[10^{5}, 10^{7}]$	$[0, 5.0 \cdot 10^{7}]$
MIN_VIOL	0	$[0, 10^{-9}]$	$[0, 10^{-7}]$	$\Longrightarrow$	$[0, 10^{-3}]$	$[0, \infty]$
MAX_SUPP_ABS	∞	$[2.0 \cdot 10^{3}, 5.0 \cdot 10^{3}]$	$[1.5 \cdot 10^{3}, \infty]$	$\Longleftarrow$	$[1.0 \cdot 10^{3}, 5.0 \cdot 10^{3}]$	$[0, \infty]$
MAX_SUPP_REL	1	$[9.2 \cdot 10^{-1}, 1.0]$	$[8.8 \cdot 10^{-1}, 1.0]$	$\Longleftarrow$	$[0, 1.0]$	$[0, 1.0]$
EPS_ELIM	0	$[0, 0]$	$[0, 10^{-20}]$	—	$[0, 0]$	$[0, 10^{-20}]$
MAX_DYN_LUB	10^{6}	$[5.0 \cdot 10^{5}, 10^{7}]$	$[10^{5}, 5.0 \cdot 10^{7}]$	$\Longleftarrow$	$[5.0 \cdot 10^{3}, 5.0 \cdot 10^{8}]$	$[0, \infty]$
EPS_COEFF_LUB	10^{-11}	$[10^{-11}, 10^{-11}]$	$[10^{-12}, 10^{-10}]$	—	$[10^{-11}, 10^{-11}]$	$[10^{-12}, 10^{-10}]$

Table 21: Sensitivity analysis of the parameters of the reference cut generator.