Math. Prog. Comp. (2011) 3:281–318
DOI 10.1007/s12532-011-0028-6

FULL LENGTH PAPER

Practical strategies for generating rank-1 split cuts in mixed-integer linear programming

Gerard Cornuéjols · Giacomo Nannicini

Received: 15 January 2011 / Accepted: 13 July 2011 / Published online: 31 July 2011

Abstract In this paper we propose practical strategies for generating split cuts, by considering integer linear combinations of the rows of the optimal simplex tableau, and deriving the corresponding Gomory mixed-integer cuts; potentially, we can generate a huge number of cuts. A key idea is to select subsets of variables, and cut deeply in the space of these variables. We show that variables with small reduced cost are good candidates for this purpose, yielding cuts that close a larger integrality gap. An extensive computational evaluation of these cuts points to the following two conclusions. The first is that our rank-1 cuts improve significantly on existing split cut generators (Gomory cuts from single tableau rows, MIR, Reduce-and-Split, Lift-and-Project, Flow and Knapsack cover): on MIPLIB instances, these generators close 24% of the integrality gap on average; adding our cuts yields an additional 5%. The second conclusion is that, when incorporated in a Branch-and-Cut framework, these new cuts can improve computing time on difficult instances.

Mathematics Subject Classification (2000) 90C11 · 90C57

1 Introduction

Although solving mixed-integer linear programs (MILPs) is NP-hard [22], there exist both commercial and free software packages that are able to efficiently solve many MILPs arising from real-life applications. A dramatic improvement in the performance of this software came from the introduction of general cutting planes, such as

G. Cornuéjols · G. Nannicini (✉)
Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA, USA
e-mail: nannicini@sutd.edu.sg

G. Cornuéjols
e-mail: gc0v@andrew.cmu.edu

Gomory mixed-integer (GMI) cuts [18], and Mixed-integer rounding (MIR) cuts [21]. Indeed, computational experiments reported in [9] show that, on a set of benchmark instances, disabling cutting plane generation in the commercial software Cplex yields a slow down of the solution process by more than a factor of 50. The most effective cuts, according to [9], fall into the category of *split cuts* [15], that is, cutting planes that can be derived from the LP relaxation together with a disjunction of the form $\pi^\top x \leq \pi_0 \vee \pi^\top x \geq \pi_0 + 1$ with (π, π_0) integer. Any such cut derived from the original LP relaxation is said to be *rank-1*.

In this paper, we investigate efficient algorithms for the generation of rank-1 split cuts from the optimal simplex tableau. The reason for focusing on these cuts is that several computational studies [8,11,17] show that the split closure (i.e. the intersection of all rank-1 split cuts) often provides a tight approximation of the convex hull of feasible solutions; yet available cut generators are still far from achieving the full potential of split cuts. An objective of this paper is to see how much we can add to the existing families of split cuts, in reasonable computational time. Existing split cut generators in the Branch-and-Cut code COIN-OR Cbc [12] (GMI, MIR, Reduce-and-Split, Lift & Project, Knapsack cover, Flow cover) close 24% of the integrality gap[1] on average on MIPLIB instances, of which 11% is contributed by GMI cuts alone. Adding our new cut generation strategies, we can close 29% of the gap on average. A characteristic of our cuts is that they are valid for the corner polyhedron associated with the optimal LP basis.

The cut generation algorithm that we propose computes integral linear combinations of the rows of the optimal simplex tableau, with the aim of reducing the cut coefficients on the continuous nonbasic variables, in the spirit of the Reduce-and-Split algorithm [3,16]. In this paper, we build upon this idea, by proposing and evaluating several strategies to choose the simplex tableau rows involved in the linear combinations, and the columns affected by the reduction algorithm. These strategies aim to generate cuts with the following desirable properties: they are sparse, they have small coefficients, and they are mutually orthogonal. Although some of these ideas have already been proposed in the literature, their effectiveness in practice has never been thoroughly studied. Computational experiments show that our strategies outperform the implementation of Reduce-and-Split [3] currently available in COIN-OR Cgl [13]. The methods that we propose can potentially generate a very large number of split cuts, therefore we study policies to generate only a manageable number of strong cuts. Our computational experiments show that, using our cuts, we can improve the computing time (or gap closed in a fixed amount of time) in a Branch-and-Cut algorithm on difficult MIPLIB instances.

The rest of this paper is organized as follows. First, we provide the necessary theoretical background (Sect. 2). Then, we discuss the coefficient reduction algorithm with all its variants (Sect. 3), which is the main contribution of this paper. An extensive experimental evaluation of the proposed cut generation strategies is provided in Sect. 4. Sect. 5 concludes the paper.

[1] For a minimization problem, the integrality gap closed is defined as: (objective_after_cuts − objective_LP)/(objective_integer_optimum − objective_LP).

2 Theoretical backgound

Consider the following Mixed Integer Linear Program in standard form:

$$\left.\begin{array}{ll} \min & c^\top x \\ & Ax = b \\ & x \geq 0 \\ \forall j \in N_I & x_j \in \mathbb{Z}, \end{array}\right\} \qquad (\mathcal{P})$$

where $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $A \in \mathbb{R}^{m \times n}$ and $N_I \subset N = \{1, \ldots, n\}$. The LP relaxation of $\mathcal{P}$ is the linear program obtained by dropping the integrality constraints, and is denoted by $\bar{\mathcal{P}}$. Let $B \subset N$ be an optimal basis of $\bar{\mathcal{P}}$, and $J = N \setminus B$ the set of nonbasic variables. The corresponding simplex tableau is given by:

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

Consider an arbitrary equality $\sum_{j \in N} g_j x_j = d$ satisfied by all feasible solutions to $\mathcal{P}$. Define $f_0 := d - \lfloor d \rfloor$ and $f_j := g_j - \lfloor g_j \rfloor$ for all $j \in N_I$. Suppose $f_0 > 0$; the GMI cut associated with this equation is:

$$\begin{aligned} &\sum_{j \in N_I : f_j \leq f_0} \frac{f_j}{f_0} x_j + \sum_{j \in N_I : f_j > f_0} \frac{1 - f_j}{1 - f_0} x_j \\ &+ \sum_{j \in N \setminus N_I : g_j \geq 0} \frac{g_j}{f_0} x_j - \sum_{j \in N \setminus N_I : g_j < 0} \frac{g_j}{1 - f_0} x_j \geq 1. \end{aligned} \qquad (2)$$

It can be shown that (2) is valid for $\mathcal{P}$ [18]. Let $B_I = B \cap N_I$, $J_I = J \cap N_I$, $J_C = J \setminus N_I$ be the sets of integer basic variables, integer nonbasic variables and continuous nonbasic variables respectively. Now consider a linear combination with integer coefficients λ_i of those rows of (1) where $i \in B_I$:

$$\sum_{i \in B_I} \lambda_i x_i = \tilde{x} - \sum_{j \in J} \tilde{a}_j x_j, \qquad (3)$$

where

$$\begin{aligned} \tilde{x} &= \sum_{i \in B_I} \lambda_i \bar{x}_i \\ \tilde{a}_j &= \sum_{i \in B_I} \lambda_i \bar{a}_{ij} \quad \text{for } j \in J. \end{aligned} \qquad (4)$$

Equation (3) is satisfied by all feasible solutions to $\mathcal{P}$, which yields $f_0 = \tilde{x} - \lfloor \tilde{x} \rfloor$, $f_j = \tilde{a}_j - \lfloor \tilde{a}_j \rfloor$ for all $j \in J$ according to the definition above. Note that in order to generate

a GMI cut we need $\tilde{x} \notin \mathbb{Z}$ and therefore $\lambda \neq 0$. Applying (2) to (3) we obtain:

$$\sum_{j \in J_I : f_j \leq f_0} \frac{f_j}{f_0} x_j + \sum_{j \in J_I : f_j > f_0} \frac{1 - f_j}{1 - f_0} x_j + \sum_{j \in J_C : \tilde{a}_j \geq 0} \frac{\tilde{a}_j}{f_0} x_j - \sum_{j \in J_C : \tilde{a}_j < 0} \frac{\tilde{a}_j}{1 - f_0} x_j \geq 1. \tag{5}$$

This is the GMI cut associated with the row obtained through the row multipliers λ. Clearly, (5) cuts off the optimal basic solution $\bar{x}$. In order to generate a deep cut, we want to find integer multipliers λ_i for all $i \in B_I$ such that the resulting coefficients in (5) are as small as possible.

In this paper, we work with an optimal basis B of $\bar{\mathcal{P}}$, and our cuts are valid for the corresponding corner polyhedron. In principle, cutting planes of the form (5) can be generated from other bases as well, but this faces the problem of selecting a promising basis from which the current fractional point $\bar{x}$ can be cut off. This topic is not investigated here.

3 The reduction algorithm

In order to obtain cutting planes with small coefficients (and hopefully zeroes) on the continuous nonbasic columns, we select $J_W \subseteq J_C$, and we attempt to minimize $\tilde{a}_j$ for $j \in J_W$ over vectors $\lambda \in \mathbb{Z}^{|B_I|}$. More specifically, we would like to minimize $\|\tilde{d}\|$, where

$$\tilde{d} = (\tilde{a}_j)_{j \in J_W}. \tag{6}$$

Here, the choice of the L_2 norm of $\tilde{d}$ is motivated by the fact that it yields simpler norm minimization problems. As can be seen from (5), small $\|(\tilde{a}_j)_{j \in J_W}\|$ is likely to translate into better cut coefficients on the continuous variables in J_W. On integer variables, this is not true because of the modular arithmetic applied, so that a smaller row coefficient $\tilde{a}_j$ on an integer variable offers no prospect of a better corresponding cut coefficient. This explains why we focus on continuous nonbasic variables only. This idea has been pursued also in [3,16] in the special case where $J_W \equiv J_C$. Our algorithm to compute λ, which we call *coefficient reduction algorithm*, is described next.

Choose $J_W \subseteq J_C$; apply a permutation to the simplex tableau in order to obtain $B_I = \{1, \ldots, |B_I|\}$, $J_W = \{1, \ldots, |J_W|\}$, and define the matrix $D \in \mathbb{R}^{|B_I| \times |J_W|}$, $d_{ij} = \bar{a}_{ij}$. Thus, we can rewrite the problem of minimizing $\|\tilde{d}\|$ as

$$\min_{\lambda \in \mathbb{Z}^{|B_I|} \setminus \{0\}} \left\| \sum_{i \in B_I} \lambda_i d_i \right\|. \tag{7}$$

This is a shortest vector problem in the additive group generated by the rows of D. Assuming linear independency between the rows, the group defines a lattice, and (7) becomes the shortest vector problem in a lattice, which is NP-hard under randomized reductions [2].

The approach of Andersen, Cornuéjols and Li [3] consists in applying to D (with $J_W \equiv J_C$) an iterative algorithm which is related to the basis reduction algorithm of Lenstra, Lenstra and Lovász [19]. Given two rows d_i and d_j, the optimal integer coefficient δ_{ij} such that $\|d_i + \delta_{ij} d_j\|$ is minimum can be computed in closed form. Therefore, for each row d_k they choose, among the remaining rows, the row d_j such that $\|d_k + \delta_{kj} d_j\|$ is minimum, and replace d_k with $d_k + \delta_{kj} d_j$ in D. The process is iterated until no rows can be combined together to reduce the norm of the first one by at least a given factor $1 - \sigma$ with $0 \leq \sigma < 1$. This yields a new matrix D whose rows form an angle between 60° and 120° with each other. By recording the row operations performed, i.e. the values of δ_{kj} used in the combinations, it is easy to recover the corresponding row multipliers λ that can be applied to the original simplex tableau to derive GMI cuts (5).

In [16], for each row d_k of D, a subset $R_k \subset B_I$ of the rows of the simplex tableau with $d_k \in R_k$ is chosen; then, integral multipliers are calculated, in order to reduce $\|d_k\|$ as much as possible with a linear combination of the rows d_i for all $i \in R_k \setminus \{k\}$. Relaxing the integrality requirements on λ, for each row d_k that we want to reduce we have the following convex minimization problem:

$$\min_{\lambda^k \in \mathbb{R}^{|R_k|}, \lambda^k_k = 1} \left\| \sum_{i \in R_k} \lambda^k_i d_i \right\|. \tag{8}$$

An integer (not necessarily optimal) solution can be found by rounding each coefficient λ^k_i of the continuous solution to the nearest integer $\lfloor \lambda^k_i \rceil$. The optimal continuous multipliers λ that solve (8) can be obtained by solving a $|R_k| \times |R_k|$ linear system. Note that, in order to avoid the trivial solution $\lambda^k = 0$, we impose a normalization constraint $\lambda^k_k = 1$. This way, the initial row d_k has unitary coefficient in the linear combination, and we obtain different optimization problems for each row $k = 1, \ldots, |B_I|$.

Another reason to impose a normalization is that we are interested in vectors of row multipliers λ with small norm. This was already observed in [16], and will be further discussed in Sect. 4. Computational experiments suggest that the most interesting split cuts are those generated from disjunctions with a small absolute sum of the components. In particular, Balas and Saxena [8] observe that, on a standard set of benchmark instances (MIPLIB 3.0 [10]), in order to separate over the first split closure, only disjunctions with support ≤ 20 are needed in practice, with very few exceptions. Furthermore, the average integer coefficient appearing in the split disjunctions is typically small. There is a direct relationship between our vector of row multipliers λ and the split disjunction π associated with the corresponding split cut: indeed, the GMI cut derived from a linear combination of rows with multipliers λ is the same as a split cut derived from the disjunction $\pi^\top x \leq \lfloor \pi^\top \bar{x} \rfloor \vee \pi^\top x \geq \lfloor \pi^\top \bar{x} \rfloor + 1$, where the basic part of the disjunction is given by $\pi_i = \lambda_j$ if x_i is basic in the jth row, and the nonbasic part can be computed by strengthening the disjunction exploiting integrality

of the nonbasic integer variables [5,7]. Hence, we are only interested in generating integer vectors λ with small absolute sum of the coefficients. Transforming (8) into a constrained optimization problem adding some condition on $\|\lambda\|$ would increase the difficulty of solving the problem. Therefore, we opted for simply penalizing large values of $\|\lambda\|$ in the objective function; that is, we replace (8) with:

$$\min_{\lambda^k \in \mathbb{R}^{|R_k|}, \lambda^k_k = 1} \left\| \sum_{i \in R_k} \lambda^k_i d_i \right\|^2 + \gamma' \|\lambda\|^2, \tag{9}$$

where γ' is a given parameter that weights the normalization. In practice, our approach is to first solve (9) with $\gamma' = 0$, and if the solution λ does not satisfy a criterion $\sum_i |\lambda_i| \leq \Lambda$, we resolve (9) with $\gamma' > 0$. We chose to penalize the L_2 norm of λ instead of the L_1 norm in (9) because it gives rise to a quadratic convex minimization problem, which is easy to solve.

We solve (9) by vanishing the derivatives of $\| \sum_{i \in R_k} \lambda^k_i d_i \|^2 + \gamma' \|\lambda\|^2$ with respect to λ^k_i, $i \in R_k$. This amounts to solving the following linear system for $k \in \{1, \ldots, |B_I|\}$:

$$A^k \lambda^k = b^k, \tag{10}$$

where $A^k \in \mathbb{R}^{|R_k| \times |R_k|}$ and $b^k \in \mathbb{R}^{|R_k|}$ are defined as follows:

$$A^k_{ij} = \begin{cases} 1 & \text{if } i = j = k \\ 0 & \text{if } i = k \text{ or } j = k \text{ but not both} \\ \sum_{h=1}^{|J^C|} d_{ih} d_{jh} + \gamma' & \text{if } i = j \neq k \\ \sum_{h=1}^{|J^C|} d_{ih} d_{jh} & \text{otherwise,} \end{cases}$$
$$b^k_i = \begin{cases} 1 & \text{if } i = k \\ -\sum_{h=1}^{|J^C|} d_{ih} d_{kh} & \text{otherwise.} \end{cases} \tag{11}$$

Once the linear systems are solved and the optimal continuous coefficients $\lambda^k \in \mathbb{R}^{|R_k|}$ for all $k \in \{1, \ldots, |B_I|\}$ are available, they are rounded to the nearest integer. Then, consider the norm of $\sum_{i \in R_k} \lfloor \lambda^k_i \rceil d_i$. If $\| \sum_{i \in R_k} \lfloor \lambda^k_i \rceil d_i \| < (1 - \sigma) \|d_k\|$, where $0 \leq \sigma < 1$ is a given parameter, there is an improvement with respect to the original row of the simplex tableau, and we expect the associated GMI cut to be stronger, at least on the working set of columns J_W; in this case, the equation

$$\sum_{i \in R_k} \lambda^k_i x_i = \sum_{i \in R_k} \lambda^k_i \bar{x}_i - \sum_{j \in J} \sum_{i \in R_k} \lambda^k_i \bar{a}_{ij} x_j, \tag{12}$$

is used in order to compute a GMI cut according to (5).

We now propose several strategies to choose the set of working variables J_W, and a corresponding set of rows R_k for the coefficient reduction algorithm.

3.1 Selection of the working set of nonbasic continuous variables

In order to generate a large number of split cuts, we work on different sets of continuous nonbasic variables, aiming each time to reduce the coefficients for the variables in the working set J_W. This serves two purposes: one is to generate more cuts, and the second is to obtain more orthogonal cutting planes. We generate more cuts because, given a reduction algorithm to compute the row multipliers λ (such as the one described above) and a set of rows R_k, we potentially obtain one new cut for each different J_W. We obtain more orthogonal cutting planes because we generate cuts with small coefficients, potentially zeroes, on the columns in the set J_W; by iterating the cut generation algorithm choosing disjoint sets J_W at each iteration, we obtain a collection of cutting planes which should be more orthogonal with each other than the GMI cuts obtained from the original simplex tableau rows.

We take into account the reduced costs of the variables with index set J_C for choosing J_W. The motivation is that there is a direct relationship between the improvement in the objective function given by a cut, its coefficients, and the reduced costs of the variables. Consider the current solution to the LP relaxation $\bar{x}$ and the extreme rays r^j, $j \in J$ of the cone $C = \{Ax = 0, x_j \geq 0 \ \forall j \in J\}$ associated with the corresponding basic solution. Note that $c^\top r^j$ is the reduced cost associated with x_j; we denote it by $\bar{c}_j$ in the following. Given a GMI cut (5), rewrite it as an intersection cut $\sum_{j \in J} \frac{x_j}{\alpha_j} \geq 1$, so that the cutting plane supports the points $x^j = \bar{x} + \alpha_j r^j$ (see [4] for details). Now the objective function value of $\mathcal{P}$ after the addition of the cut is at least $\min_{j \in J} c^\top x^j = \min_{j \in J} (c^\top \bar{x} + \alpha_j c^\top r^j)$. In order to maximize this quantity, it is clear that for the variables with small reduced cost we should aim for a large α_j, i.e. a small cut coefficient. We give another reason for taking into account reduced costs. Variables with large reduced costs are likely to stay nonbasic even after the addition of cutting planes, whereas variables with small reduced costs have a larger probability of entering the basis; therefore, cutting planes that cut deeply on variables with small reduced costs are more likely to have a large effect on the solution to the LP relaxation, i.e. they "move" $\bar{x}$ by a larger amount.

We choose the working sets J_W by partitioning the set of continuous nonbasic columns into k disjoint sets; each of these partitions gives rise to k disjoint J_W's, hence a collection of potentially orthogonal cuts. However, we do not always partition the whole set of continuous nonbasic columns J_C: in some cases, we only consider the variables with smallest reduced costs. When computing the partitions, we always sort the variables in J_C by increasing reduced costs; we denote by S the ordered set obtained by sorting J_C by increasing reduced costs $\bar{c}_j$. We distinguish between contiguous k-partitions and k-partitions with interleaving pattern. By *contiguous k-partition* we denote a partition $S_1, S_2, \ldots, S_k$ such that for $i < j, \forall p \in S_i, q \in S_j$ we have $\bar{c}_p \leq \bar{c}_q$, and the cardinality of S_i differs by at most one from the cardinality of any other set S_j. In other words, each set of the partition contains variables which are contiguous in S. By *k-partition with interleaving pattern* we denote a partition $S_1, S_2, \ldots, S_k$ of the first $r|J_C|$ variables of S where $0 < r \leq 1$, the cardinality of S_i differs by at most one from the cardinality of any other set S_j, and from each set of $2k$ contiguous variables, we assign 2 variables to each of the k sets S_i. We denote a contiguous k-partition by C-kP, and a k-partition with interleaving pattern by I-kP-r or

simply by I-kP if $r = 1$. We consider the following 7 partitions: C-3P, C-5P, I-2P-1/2, I-2P-2/3, I-2P-4/5, I-3P, I-4P.

We believe that the particular pattern used to generate the interleaving partitions does not affect their usefulness. Indeed, preliminary computational tests conducted by comparing partitions with randomly generated interleaving patterns to partitions with fixed interleaving patterns showed very similar results. We chose to eliminate the computational overhead of generating random interleaving partitions by hard-coding different variable selection patterns. We reach a different conclusion for contiguous partitions: the first set of the partition should give stronger cutting planes, whereas the remaining sets of the partition should yield progressively weaker cuts. This allows to verify our claim that reduced costs play an important role in determining the quality of a cut; a computational analysis of this conjecture is given in Sect. 4.

3.2 Selection of the rows for the reduction algorithm

In order to reduce the norm of row d_k, the reduction algorithm of Sect. 3 needs a set of candidate rows R_k for the linear combination. Here we propose several different criteria to choose R_k. There are two natural objectives for an effective row selection strategy: on the one hand, we want to select rows which allow for a large reduction of the coefficients on the continuous nonbasic variables; on the other hand, we do not want to deteriorate the cut on the integer nonbasic variables, as those are not taken into consideration when solving the optimization problem (8). For each row selection strategy we have a parameter μ, which represents the maximum number of rows that should be selected; that is, $|R_k| \leq \mu$. We have three basic strategies, which we describe below, that are applied to different sets of columns to yield a total of eight row selection strategies. We now give details on the three basic strategies, relative to a generic set of nonbasic columns $S \subseteq J$.

1. The first basic row selection strategy (BRS1) derives from the original one proposed in [16], and it consists in ranking the rows with index set $B_I \setminus \{k\}$ by increasing number of nonzeroes on the nonbasic columns S where d_k is zero (in this section we always use the row number as a tie breaker), and picking the first μ such rows.
2. The second basic row selection strategy (BRS2) is a variant of the first one: we employ a greedy algorithm to select, at each iteration, the row which introduces the smallest number of nonzeroes on the columns with index set S where the original row d_k is zero and we did not introduce a nonzero at previous iterations. The greedy algorithm is iterated until μ rows are selected. This strategy is computationally more expensive than the first one, but should result in a smaller number of nonzeroes in the linear combinations of the selected rows.
3. The third basic row selection strategy (BRS3) chooses the first μ rows with index in $B_I \setminus \{k\}$ such that their angle with respect to d_k in the space of columns with index set S is the smallest.

Each of the three basic strategies described above is applied to different choices of S. In order to choose S, we make the following observations. Our coefficient reduction algorithm only takes into account the continuous nonbasic columns in the set

J_W for the optimization problem (9); therefore, we hopefully generate cuts which are strong on the continuous nonbasic variables. However, at the same time we would like the coefficients on the integer nonbasic variables not to deteriorate as an effect of the linear combination (even though these coefficients are bounded). Clearly, when the coefficient reduction algorithm is applied to a row d_k, there is no interest in choosing rows which have all zero coefficients in the columns with index set J_W where d_k has a nonzero: in all basic strategies BRS1, BRS2, BRS3, we only consider rows which have at least one nonzero coefficient on the columns with index set J_W where d_k is nonzero.

There are three natural choices for $S : S = J_I$ (we try not to deteriorate the cut coefficients on the integer nonbasic variables, by introducing few nonzeroes), $S = J_W$ (we focus on reducing the coefficients on the columns ($j \in J_W$), while introducing few nonzeroes) and $S = J_I \cup J_W$ (we combine both goals). Both BRS1 and BRS2 are applied with all these possibilities. For BRS3, our intuition is that using $S = J_I$ would not produce good results: as the coefficients of the cut on the integer nonbasic variables after combining several rows together are difficult to control (because of the modular arithmetic involved), choosing rows which have similar coefficients on the columns ($j \in J_I$) does not seem a good idea. Therefore, for BSR3 we only pick $S = J_W$ and $S = J_I \cup J_W$. This leaves us with a total of eight row selection strategies, which we label as follows:

1. RS1: BRS1 with $S = J_I$
2. RS2: BRS1 with $S = J_W$
3. RS3: BRS1 with $S = J_I \cup J_W$
4. RS4: BRS2 with $S = J_I$
5. RS5: BRS2 with $S = J_W$
6. RS6: BRS2 with $S = J_I \cup J_W$
7. RS7: BRS3 with $S = J_W$
8. RS8: BRS3 with $S = J_I \cup J_W$

4 Computational experiments

In this section we provide a computational evaluation of the ideas discussed in Sect. 3. It should be noted that, as with most codes involving floating point computations, our computational results depend on the platform on which they are run (architecture, compiler, libraries). However, despite differences in the numbers when executed on different platforms, we found the conclusions that could be drawn to be consistent. The experiments reported in this paper were perfomed on a 32-bit machine equipped with an Intel Xeon X3220 clocked at 2.40 Ghz and 8 GB RAM, running Linux. We employed COIN-OR Cbc 2.3.0 [12] as Branch-and-Cut software, with COIN-OR Clp 1.10.0 [14] as underlying LP solver. Our cut generator was implemented within the COIN-OR Cgl [13] framework, and is available in Cgl. We remark that the combination of our code and the COIN-OR framework can prove numerically unstable, and we experienced some failures of the Branch-and-Cut code during our tests. We found Cplex to be more stable in this respect, but we decided to use the COIN-OR framework because of greater flexibility and the possibility of finely tuning the cut generators.

We employ several cut generators available in COIN-OR Cgl 0.55.0, and in particular: MIR (CglMixedIntegerRounding1[2]), Two-step MIR (CglTwomir), the original Reduce-and-Split (CglRedSplit), Lift and Project (CglLandP), Knapsack cover (CglKnapsackCover) and Flow cover (CglFlowCover). We also use our own implementation of a GMI cut generator (CglGMI). The union of all these cut generators, which represents our baseline and is a superset of the generators employed in Cbc by default,[3] will be called CGLALLCUTS in the following.

The reason for using our implementation of a GMI cut generator is that we have more flexibility in determining the cut acceptance/rejection criteria than with the existing implementation (CglGomory), and slightly faster computation times.

In almost all experiments we will test our cut generator on top of CGLALLCUTS. This is because our heuristics are not designed to be faster than existing cut generators: they are designed to find split cuts which cannot be obtained through existing methods. For difficult MILPs, investing a few more seconds in cut generation at the root can bring a large reduction in the enumeration tree. For easy problems, this approach may not be the most effective one, as simple enumeration is often faster than Branch-and-Cut.

In this paper, all average values are reported as geometric averages. Since the set of values that we want to average frequently contains zero, we add 1 to each value before computing the average, and then subtract 1 from the final result. Instances for which all tested method close zero integrality gap are excluded from the averages reported at the end of the corresponding tables.

4.1 Test instances

We tested our cut generation algorithm on mixed-integer benchmark problems taken from MIPLIB 3.0 [10], MIPLIB 2003 [1], and difficult problems from the University of Bologna as available at: http://plato.asu.edu/sub/testcases.html. All of these problems were processed to find a feasible solution guaranteed to be within 1% of the optimum, using the procedure described in [20]. The value of these optima was employed to compute integrality gaps and provided as cutoff to the Branch-and-Cut solver, if not otherwise stated. For one instance (`t1717`), where the optimal solution is not known and we could not find a solution within 1% of the lower bound, we took the best solution found by Cplex 12.1 after 6 h of computation using 2 cores of an Intel Xeon clocked at 3.20 Ghz running Linux (note that this is not the same machine where the remaining experiments are run).

Our instance selection criterion is as follows: we excluded instances with more than 500,000 nonzeroes in the constraint matrix, and those instances such that solving the LP relaxation at the root took more than 30 s, or Clp encountered numerical difficulties.

We distinguish between instances with continuous structural variables, and instances where the only continuous variables are artificial (i.e. slacks introduced

[2] We employed CglMixedIntegerRounding1 instead of its variant CglMixedIntegerRounding2 (which generates the same cuts) because we found it to be slightly faster.

[3] Excluding CglClique, whose implementation is not sufficiently general.

to deal with inequalities). We first focus on instances with continuous structural variables. These instances have more continuous variables to which our coefficient reduction algorithm can be applied; furthermore, continuous variables play a more important role, because they appear in the original formulation and typically also in the objective function. Therefore, for testing purposes, these instances are more meaningful for this paper, in particular for rank-1 cuts. On pure integer instances, continuous artificial variables may or may not be present in the first round of cutting planes, depending on the presence of inequalities in the original problem, but the introduction of cutting planes ensures the presence of slacks in the following rounds.

We label the first set of test instances `MILP_C`; the list of instances is given in Table 1. `MILP_C` contains mixed-integer instances with more than 2 continuous structural variables. The second set of test instances is labeled `ILP_C`; it contains

Table 1 Instances in the test set `MILP_C`

Name	# Variables			# Constr.
	Cont.	Int.	Slacks	
10teams_c	225	1800	95	230
a1c1s1_c	3456	192	2072	3312
aflow30a_c	421	421	421	479
aflow40b_c	164	1364	1364	1442
arki001_c	877	511	1048	1048
b1c1s1_c	3584	288	2624	3904
b2c1s1_c	3584	288	2624	3904
bell3a_c	62	71	123	123
bell4_c	53	64	105	105
bell5_c	46	58	91	91
bg512142_c	552	240	1071	1307
blend2_c	97	256	185	274
danoint_c	465	56	577	664
dcmulti_c	473	75	212	290
dg012142_c	1440	640	5670	6310
dsbmip_mod_c	1717	160	854	1182
egout_c	86	55	56	98
fiber_c	44	1254	0	363
fixnet3_c	500	378	378	478
fixnet4_c	500	378	378	478
fixnet6_c	500	378	378	478
flugpl_c	7	11	13	18
gen_c	720	150	630	780
gesa2_c	816	408	1345	1392
gesa2_o_c	504	720	1196	1248
gesa3_c	768	384	1323	1368
gesa3_o_c	480	672	1174	1224

Table 1 continued

Name	# Variables			# Constr.
	Cont.	Int.	Slacks	
glass4_mod_c	19	302	361	396
khb05250_c	1326	24	24	101
misc06_c	1696	112	601	820
mod011_c	10862	96	248	4480
modglob_c	324	98	226	291
momentum_c	2825	2349	42421	42680
net12_c	12512	1603	13469	14021
noswot_c	28	100	180	182
pk1_c	31	55	30	45
pp08a_c	176	64	72	136
pp08aCUTS_c	176	64	182	246
qiu_c	792	48	1064	1192
qnet1_c	124	1417	176	503
qnet1_o_c	124	1417	129	456
rgn_c	80	100	4	24
roll3000_c	428	738	2120	2294
rout_c	241	315	261	291
set1ch_c	472	240	252	492
swath_c	81	6724	381	884
timtab1_c	239	158	4	171
timtab2_c	398	277	15	294
tr12-30_c	720	360	390	750
vpm1_c	210	168	195	234
vpm2_c	210	168	201	234

mixed-integer instances with at most 2 continuous structural variables, and pure integer instances. The list of instances in ILP_C is given in Table 2. For each instance we give the number of continuous and integer structural variables, the number of artificial variables for inequalities or ranged constraints introduced when the problem is reformulated in standard form (which can be exploited by our coefficient reduction algorithm), and the number of rows.

4.2 Implementation

Our cut generator is implemented within the COIN-OR Cgl framework as a CglCutGenerator called CglRedSplit2; thus, it can be used within any project that supports Cgl. The linear systems involved for the solution of (9) are solved via LU decomposition and backward substitution, using our own implementation based on [23]. The option to use LAPACK for this task is available; however, after a brief computational

Table 2 Instances in the test set ILP_C

Name	# Variables			# Constr.
	Cont.	Int.	Slacks	
air04_c	0	8904	0	823
air05_c	0	7195	0	426
cap6000_c	0	6000	2053	2176
disctom_c	0	10000	0	399
ds_c	0	67732	0	656
enigma_c	0	100	0	21
fast0507_c	0	63009	507	507
gt2_c	0	188	29	29
harp2_c	0	2993	39	112
l152lav_c	0	1989	1	97
lseu_c	0	89	28	28
manna81_c	0	3321	6480	6480
mas74_c	1	150	13	13
mas76_c	1	150	12	12
misc03_c	1	159	69	96
misc07_c	1	259	177	212
mitre_c	0	10724	1671	2054
mkc_c	2	5323	3410	3411
mod008_c	0	319	6	6
nsrand-ipx_c	1	6620	735	735
nw04_c	0	87482	0	36
opt1217_c	1	768	16	64
p0033_c	0	33	15	15
p0201_c	0	201	133	133
p0282_c	0	282	241	241
p0548_c	0	548	176	176
p2756_c	0	2756	755	755
protfold_c	0	1835	2075	2112
seymour_c	0	1372	4944	4944
sp97ar_c	0	14101	1761	1761
stein27_c	0	27	118	118
stein45_c	0	45	331	331
t1717_c	0	73885	0	551

evaluation we decided to employ our code. The reason for this is that LAPACK typically provides better numerical accuracy at the cost of some speed, but in our case, there is no need for high accuracy. Indeed, the solution of each linear system is rounded to the nearest integer, therefore we decided to favour speed.

Our cut generator has several parameters required by the coefficient reduction algorithm (Sect. 3): the maximum number of rows μ that can be combined in a linear

combination ($|R_k| \leq \mu$, in the notation of Sect. 3.2), the column selection strategy (Sect. 3.1), and the row selection strategy (Sect. 3.2). Two additional parameters play a role in the cut acceptance/rejection criteria: the minimum norm reduction factor σ to accept a vector of row multipliers, and the maximum accepted value Λ of $\|\lambda\|_1$. Finally, the last parameter is the weight that penalizes $\|\lambda\|$ whenever (9) is resolved because the first solution does not satisfy $\|\lambda\|_1 \leq \Lambda$. In particular, we set $\gamma' = \gamma \|d_k\|$ in (9), where γ is a parameter.

In our experiments, we tested six possible values for μ : 3, 5, 10, 15, 20, 50. As μ determines the size of the linear system (11), and therefore the computational effort required to generate each cut, we did not try values above 50. Additionally, there are 21 possible column selection strategies, as described in Sect. 3.1, and 8 row selection strategies which are discussed in Sect. 3.2. This gives a total of 1008 combinations, meaning that for each row of the simplex tableau where the basic integer variable takes on a fractional value, we can potentially generate 1008 different vectors of row multipliers λ, hence 1008 different cuts. Several combinations of these will typically give rise to the same vector of row multipliers.

Cuts are generated using the standard Gomory formula for GMI cuts (see e.g. [24]) only when the fractionality of the right hand side of the equality [i.e. f_0 in (5)] exceeds a given value, which was set to 0.01 for these experiments. This parameter is typically called `Away` in the integer programming community. Additionally, cuts are rejected if they are violated by the current solution of the LP relaxation by $<10^{-7}$, or if their support exceeds $1000 + n'/5$, where n' is the number of *structural* variables. Note that we set the same value of `Away`, minimum violation and maximum support for *all* cut generators employed in this paper, in order to have a fair comparison.

4.3 Comparison with the split closure

The aim of this section is to evaluate the quality of the cuts computed by existing cut generators as compared to the split closure, and assess the contribution provided by our cut generation heuristics. It is known that the first split closure gives a tight approximation of the integer hull [8,11,17]; but what is the integrality gap closed by existing cut generators? Do the heuristics presented in this paper help in obtaining a better approximation of the split closure? These are the questions that we attempt to answer.

In particular, we compare with the results reported by Balas and Saxena [8], who optimize over the split closure, and [17], who consider the (equivalent) MIR closure, but with different techniques. Since our cut generation algorithm requires the presence of nonbasic continuous variables, we only use mixed-integer instances. Our test set, for Sect. 4.3 only, consists in the mixed-integer instances of the original MIPLIB 3.0, without preprocessing. This coincides with the instances reported in Table 1 of [8], and Table 2 of [17]. We excluded five instances from the set: `dsbmip` and `noswot`, which have zero integrality gap; `markshare1`, `markshare2`, and `pk1`, for which neither paper reports a positive amount of gap closed.

The setup for this experiment is as follows. We test GMI cuts alone, and three different sets of cut generators; we generate one round of cutting planes from each

generator from the initial LP relaxation of the test instances. The first set of cut generators is EXISTING SPLIT CUTS, which includes all rank-1 split cuts generators in Cgl (i.e. CGLALLCUTS except CglTwomir). The second set consists in EXISTING SPLIT CUTS and our CglRedSplit2. The third set consists again in EXISTING SPLIT CUTS and CglRedSplit2, but with a different configuration that we call "CglRedSplit2 light". Since in this experiment we are not interested in computing time, in the second set of cut generators we parameterized CglRedSplit2 in such a way that it generates all cuts that can be obtained by combining the heuristics discussed in this paper. That is, we generate cuts using all combinations of the values listed in Sect. 4.2 for the three main parameters: μ, the column selection strategy, and the row selection strategy. Furthermore, σ was set to 0.001, so as to accept almost all generated cuts, while Λ was set to 20, as suggested in [8]. Recall that we first solve (9) with $\gamma = 0$; whenever the solution λ is such that $\|\lambda\|_1 \geq 20$, we resolve (9) using $\gamma = 0.0001\|d_k\|^2$. This parameterization of our cut generator is not meant to be used for practical purposes, since it generates an unnecessarily large number of cuts. Thus, we also consider a "light" version of CglRedSplit2 in the third set of cut generators. The light version uses $\mu = 3, 5$ and row strategies RS7 and RS8 only (all column selection strategies); this choice of the parameters is dictated by our experiments in Sect. 4.4.1, and should produce a smaller number of cuts. Finally, for the experiments in this section only, instead of using default parameters for CglLandP we employ a more aggressive parameterization: we set the option `generateExtraCuts` to "all violated GMI cuts" (i.e. we generate all violated GMI cuts from the Lift-and-Project tableaux obtained at the end of a pivoting sequence), and employ two different pivoting strategies: "most negative reduced cost" and "best pivot" (see [6] for details). In conjunction with the "best pivot" strategy, we apply the iterative modularization heuristic proposed in the same paper. This should produce a more diverse set of cuts.

Results are reported in Table 3. For each instance, we report the percentage of integrality gap closed by the split closure (i.e. the best bound given in [8,17]), and for GMI cuts and each of the three sets of cut generators which are tested we report the percentage of integrality gap closed and the number of generated cuts.

We can see that on average, the split closure closes 63.77% of the integrality gap, whereas existing cut generators do not even exploit half of this potential, closing 24.39%, of which 11.95% is contributed by GMI cuts alone. Adding the Reduce-and-Split heuristics proposed in this paper increases this value to 29.53%, which represents almost half of the potential of split cuts. Notice that we generated a large number of cuts to achieve this results, but as shown by CglRedSplit2 "light", we can generate considerably fewer cuts, while losing very little in terms of integrality gap closed: only 1.2%. On the instances `mas76` and `rentacar`, all split cut generators perform very poorly. If we compute the average closed gap without taking these two instances into account, the split closure yields 67.35%, GMI cuts alone yield 14.05%, existing cut generators yield 29.71%, and adding CglRedSplit2 achieves 36.33%; hence, on this reduced test set we exploit more than half of the potential of split cuts, and our cut generator contributes more than 6.5% on top of the generators in Cgl. CglRedSplit2 "light" contributes 5% on top of existing split cut generators, but instead of adding $\approx$550 cuts on average, it generates only $\approx$100.

Table 3 Comparison between the amount of integrality gap closed by the split closure and by existing cut generators, on the mixed-integer instances of MIPLIB 3.0

Instance	Split Closure	GMI		Existing Split Cuts		Existing Split Cuts + CglRedSplit2		Existing Split Cuts + CglRedSplit2 light	
	Gap (%)	Gap (%)	# Cuts	Gap (%)	# Cuts	Gap (%)	# Cuts	Gap (%)	# Cuts
10teams	100.00	57.14	2	100.00	32	100.00	286	100.00	49
arki001	83.05	32.97	68	33.18	210	41.77	2691	39.78	752
bell3a	99.60	32.36	18	52.50	48	63.92	176	63.92	93
bell5	92.95	5.61	14	85.37	47	85.79	192	85.79	132
blend2	46.52	15.98	6	16.04	32	19.07	245	16.04	54
dano3mip	0.22	0.01	2	0.02	9	0.10	671	0.04	28
danoint	8.20	0.26	31	0.68	117	1.36	2249	1.29	543
dcmulti	100.00	27.68	31	44.51	110	49.18	930	48.22	262
egout	100.00	35.95	27	61.83	131	68.70	446	67.89	194
fiber	99.68	64.27	46	79.57	171	80.68	888	79.59	368
fixnet6	99.75	10.74	33	71.22	135	72.72	1244	72.23	236
flugpl	100.00	10.93	6	11.17	19	93.02	300	92.69	143
gen	100.00	60.69	43	74.24	164	79.66	1843	74.65	465
gesa2	99.70	10.95	12	69.36	273	69.42	550	69.41	443
gesa2_o	99.97	29.95	71	33.22	288	65.44	725	55.97	454
gesa3	95.81	5.42	11	72.99	233	86.21	793	76.50	387
gesa3_o	95.20	49.07	96	77.43	278	88.97	1222	84.42	556
khb05250	100.00	74.91	19	74.91	51	82.55	109	81.86	59
mas74	14.02	6.67	12	6.83	26	9.03	552	8.34	250
mas76	26.52	0.00	11	0.00	29	0.00	251	0.00	144
misc06	100.00	25.80	5	34.62	41	34.62	109	34.62	54
mkc	36.16	7.59	80	10.16	188	10.16	1800	10.16	610
mod011	72.44	9.72	15	35.64	65	35.64	88	35.64	76
modglob	92.18	16.33	23	17.35	69	28.46	328	24.53	122
pp08aCUTS	95.81	33.31	46	41.61	199	50.88	2784	50.77	517
pp08a	97.03	54.69	53	62.59	214	84.64	676	84.00	318
qiu	77.51	0.00	2	4.20	42	4.47	1451	4.31	169
qnet1	100.00	15.63	47	32.13	312	36.89	3736	36.74	495
qnet1_o	100.00	43.55	11	56.08	54	59.94	607	57.09	242
rentacar	23.40	0.00	2	0.00	7	0.00	33	0.00	21
rgn	100.00	3.15	17	37.56	82	40.28	437	38.00	158
rout	70.70	0.67	32	4.21	187	5.19	3430	4.21	548
set1ch	89.74	39.09	135	39.30	528	57.94	2041	57.94	726
swath	33.93	2.54	7	26.02	119	26.02	1193	26.02	334
vpm1	100.00	26.91	18	83.64	107	94.55	198	83.64	141
vpm2	81.05	15.94	33	46.49	175	47.52	512	46.98	329
Average	63.77	11.95	18.93	24.39	89.51	29.53	592.24	28.36	206.89

On some instances, our Reduce-and-Split cuts are very strong: examples are arki001, flugpl, gesa2_o, gesa3, gesa3_o, pp08a, set1ch. On the other hand, there are problems for which they do not seem to help: mas76, rentacar, swath. Note that mas74, mas76 only have one continuous structural variable in the initial formulation. In conclusion, there is still a large potential for split cuts. It would be interesting to know how much of the remaining integrality gap can be closed with cutting planes that are valid for the corner polyhedron $P_C(B^*)$ associated with the optimal basis B^* of the original LP relaxation. Most of the existing cut generators, including ours, derive valid inequalities for $P_C(B^*)$, whereas [8,17] do not have this restriction. Therefore, an interesting question for future research is to investigate how much can we still hope to achieve by exploiting $P_C(B^*)$.

4.4 Further experiments with rank-1 cuts

In this section we still consider rank-1 cuts only; that is, we generate a single round of cutting planes from the initial LP relaxation. We measure the contribution of our cut generation heuristics, both when used in conjunction, and individually. We also investigate the effect of the parameters σ, Λ and γ. All experiments are performed on the set of test instances MILP_C.

We concentrate on rank-1 cuts for two main reasons. The first reason is that the computational experiments can be carried out in reasonable time. Since we combine all our heuristics together, for a total of 1000 different combinations, generating all the cuts for several rounds takes a significant amount of time, also because the size of the LP increases after each round. The second reason is that more factors come into play when generating several rounds of cuts (for instance, how many rounds do we generate? Do we discard inactive cuts after each pass, or keep them in the LP for some rounds?) Thus, here we only compare results for the first round of cutting planes. Iterated cut generation will be discussed in the remaining sections.

4.4.1 Analysis of cut generation parameters and strategies

In this section we analyze the effect of the parameters and algorithmic options available in our cut generator. For the sake of brevity, here we report just a summary of our findings. The interested reader can find detailed results and comments in the Appendix, Sect. A.

First, we investigated the role of the parameters σ and Λ. As expected [8,16], we found that using small valued for Λ (between 10 and 20) is the best choice. Indeed, cuts which are accepted only if Λ is large close a marginal amount of integrality gap on average. In our experiments, almost all useful cutting planes stem from disjunctions whose 1-norm is not larger than 10, and in the majority of cases we only need $\Lambda = 5$. We also observed that large values of σ can significantly reduce the number of generated cuts, at the expense of the amount of closed gap; however, for $\sigma < 0.1$ we lose very little (compared to $\sigma = 0.001$, which is a very loose acceptance threshold). Values in the interval $[0.01, 0.1]$ seem to achieve the best tradeoffs.

We also analyzed the effect of a nonzero γ. Recall that, whenever the solution to (9) does not satisfy $\|\lambda\|_1 \leq \Lambda$ (after rounding to the nearest integer), we resolve the optimization problem, penalizing $\|\lambda\|_2$. In particular we solve the following problem:

$$\min_{\lambda^k \in \mathbb{R}^{|R_k|}, \lambda_k^k = 1} \left\| \sum_{i \in R_k} \lambda_i^k d_i \right\|^2 + \gamma \|d_k\| \|\lambda\|^2.$$

We found that penalizing the norm of λ has an almost negligible, although positive, effect: the amount of integrality gap closed increases, on average, by a very small fraction. We expect this positive effect to be more significant when generating cuts using fewer values for μ, column selection strategy and row selection strategy. The reason is that in the current experiment there is much redundancy; that is, it is very likely that each cut could be obtained through different combinations of the parameters. Therefore, there are fewer additional good cuts that we can discover by resolving (9) with $\gamma > 0$ when we fail the test on $\|\lambda\|_1$, simply because these cuts can also be obtained with another combination of the parameters. In a practical Branch-and-Cut setting, we will aim for little redundancy to improve computational times, hence there will be more useful cuts that can potentially be generated by our generator, but which are "missed" when using $\gamma = 0$. In our experiments we saw that good cuts are associated with λ's with small norm; we hope to recover some of the "missed" cuts by using $\gamma > 0$.

Then, we tried to assess the effectiveness of our various cut generation heuristics; in particular, the effect of the parameter μ (maximum number of rows that can are involved in each linear combination), of the column selection strategy, and of the row selection strategy. Our results clearly show that small values of μ ($\mu = 3, 5$), which are associated with split disjunction with small support, yield the best results. This is not a surprising finding. Concerning row selection strategies, we found that RS7 and RS8 are yield stronger cuts than the remaining ones. RS1 and RS4, which consider the nonzeroes on the integer nonbasic variables only, are weaker than the rest. In general, the best strategies seem to be those that take into account the coefficient on both the integer and the continuous nonbasic variables, so that at the same time we try to reduce the cut coefficients on the continuous columns, and keep the coefficients on the integer columns under control. There is no clear winner among the column selection strategies: almost all seem to be roughly equally effective, and almost all provide at least a marginal contibution that cannot be obtained through other means. On the other hand, there are two column selection strategies that are clearly less effective: those that select columns corresponding to variables with the largest reduced costs.

To confirm the effect of reduced costs, we performed an additional experiment: we applied 10 rounds of cutting planes on the set of instances `MILP_C`, where at each round we apply GMI cuts and our Reduce-and-Split cuts generated with $\mu = 5, 10, 15, 20, 50$, RS1 through RS8, and one among the three column selection strategies of C-3P. The average amount of integrality gap closed after 10 rounds by GMI cuts only is 34.71%; adding Reduce-and-Split cuts where the coefficient

reduction algorithm is applied to the first set of columns of C-3P (variables with smallest reduced costs) yields 48.68%, to the second set of C-3P yields 49.26%, to the third set of C-3P (variables with largest reduced costs) yields 43.99%. These experiments confirm that cuts which cut deeply on variables with large reduced costs are less effective than those that focus on small reduced costs *when employed alone*: since cuts are typically generated in rounds with other cuts, and geometrical intuition suggests that sets of orthogonal cuts perform well, these "weaker" cuts may prove useful in practice when used in combination with other cuts if they are orthogonal to them.

4.4.2 Parameters for efficient computation

After discussing the effect of each parameter for our cut generation algorithm, we turn our attention to a practical setting, where we ideally want to generate good split cuts in a short CPU time, and possibly in small numbers. Thus, we are interested in finding a parameter combination aimed at these objectives. Based on the experiments analyzed in Sect. A.1 and Sect. A.2, we decided to employ our CglRedSplit2 cut generator with the following parameters: $\sigma = 0.01$, $\Lambda = 10$, $\gamma = 0.0001$, and to generate cuts using $\mu = 5$, row selection strategies RS7 and RS8, and all column selection strategies. We put a time limit of 30 seconds for our cut generation algorithm. We will label CGLCUTS the set containing all cut generators in CGLALLCUTS except the original Reduce-and-Split (CglRedSplit). In Table 4 we compare CGLALLCUTS, CGLCUTS + CglRedSplit2 and CGLALLCUTS + CglRedSplit2.

Observe that the average CPU time required by CglRedSplit2 to generate one round is 2 s, which corresponds to slightly more than double the time employed by all cut generators in CGLALLCUTS. For hard problems with a solution time in the order of minutes or hours, this extra computational effort at the root represents a very small fraction of the total CPU time. More importantly, we are able to close, on average, an additional 5% of integrality gap, while generating ≈70% more cuts. Comparing with Table 10, we see that we have lost 2% of integrality gap that we could achieve by applying all our heuristics; however, at the same time we generate significantly fewer cuts (a reduction by a factor of 2.5, on average), which is a great advantage from a practical point of view.

On several instances, the improvement in the lower bound that we can obtain with rank-1 cuts only generated by our heuristics is large; to name a few of the difficult ones: `a1c1s1_c`, `arki001_c`, `b2c1s1_c`, `roll3000_c`, `timtab2_c`.

We also see that CglRedSplit2 is more effective than CglRedSplit, at least for the first round of cuts: indeed, employing CglRedSplit2 instead of CglRedSplit yields an additional 4% of integrality gap closed (32.5% instead of 28.7%), at the cost of an increase in computing time.

Summarizing, these results show that on mixed-integer instances, our cut generation heuristic have the potential to generate strong split cuts and to be useful from a practical standpoint: we are able to close an additional 5% of integrality gap compared to existing split cut generators (an improvement of 15% in relative terms), while requiring a very short CPU time.

Table 4 Results obtained on MILP_C using our cut generator with $\sigma = 0.01$, $\Lambda = 10$, $\gamma = 0.0001$, $\mu = 5$, RS7-RS8 and all column selection strategies

Instance	CglAllCuts			CglCuts + CglRedSplit2			CglAllCuts + CglRedSplit2		
	Gap (%)	Time	# Cuts	Gap (%)	Time	# Cuts	Gap (%)	Time	# Cuts
10teams_c	100.00	2.34	11	100.00	18.61	25	100.00	20.32	28
a1c1s1_c	20.29	5.60	382	26.88	18.33	500	26.88	19.45	507
aflow30a_c	22.83	0.05	132	22.87	0.25	140	22.92	0.26	161
aflow40b_c	24.56	0.25	157	24.68	1.55	203	24.71	1.57	229
arki001_c	34.47	0.44	264	40.89	1.72	610	40.89	2.66	646
b1c1s1_c	23.15	10.13	559	28.34	43.53	639	28.34	42.44	641
b2c1s1_c	15.00	10.37	556	21.95	39.46	736	21.81	40.33	726
bell3a_c	51.58	0.01	78	63.76	0.03	82	63.76	0.02	90
bell4_c	27.52	0.02	150	27.70	0.04	169	27.70	0.04	197
bell5_c	85.37	0.01	73	85.79	0.02	85	85.79	0.02	96
bg512142_c	2.38	2.15	412	2.43	10.15	1458	2.44	10.10	1559
blend2_c	16.04	0.01	39	15.99	0.03	43	16.04	0.03	48
danoint_c	1.74	0.43	139	1.74	0.81	352	1.74	0.75	381
dcmulti_c	48.09	0.07	168	49.09	0.27	261	49.09	0.29	293
dg012142_c	0.45	17.57	189	0.45	51.18	301	0.45	51.37	318
dsbmip_mod_c	−0.00	0.95	161	−0.00	2.16	116	0.00	2.18	174
egout_c	69.87	0.01	168	72.35	0.03	216	72.35	0.04	220
fiber_c	79.57	0.10	226	79.57	5.33	355	79.57	5.34	387
fixnet3_c	88.60	0.04	215	88.60	0.27	269	88.60	0.27	274
fixnet4_c	69.08	0.04	206	69.67	0.24	325	69.73	0.26	331
fixnet6_c	71.32	0.03	181	71.89	0.20	257	72.01	0.21	262
flugpl_c	11.74	0.00	28	92.69	0.00	100	92.69	0.00	105
gen_c	72.59	0.17	165	73.48	0.34	365	73.48	0.36	375
gesa2_c	70.40	0.41	333	70.88	1.17	419	70.88	1.17	471
gesa2_o_c	33.22	0.21	264	53.86	1.49	357	53.86	1.55	417
gesa3_c	67.01	0.72	278	73.79	2.40	338	75.43	2.42	382
gesa3_o_c	74.55	0.40	261	74.84	2.68	415	75.18	2.63	469
glass4_mod_c	0.00	0.03	292	0.00	0.05	277	0.00	0.05	315
khb05250_c	74.91	0.02	69	81.86	0.06	64	81.86	0.05	76
misc06_c	11.89	0.04	24	10.90	0.10	27	11.89	0.11	33
mod011_c	23.15	1.15	35	18.06	5.51	39	23.15	6.23	45
modglob_c	17.34	0.02	95	23.69	0.08	132	23.69	0.09	148
momentum1_c	47.68	321.62	266	47.68	343.37	286	47.68	345.58	308
net12_c	8.25	48.45	2266	8.25	89.97	2330	8.25	88.30	2351
noswot_c	0.00	0.01	53	0.00	0.02	98	0.00	0.02	104
pk1_c	0.00	0.01	60	0.00	0.03	164	0.00	0.03	179
pp08a_c	62.59	0.02	261	83.00	0.09	312	83.00	0.08	359

Table 4 continued

Instance	CglAllCuts			CglCuts + CglRedSplit2			CglAllCuts + CglRedSplit2		
	Gap (%)	Time	# Cuts	Gap (%)	Time	# Cuts	Gap (%)	Time	# Cuts
pp08aCUTS_c	41.61	0.11	189	50.76	0.23	383	50.76	0.25	425
qiu_c	1.00	0.86	51	1.00	0.98	119	1.00	1.18	126
qnet1_c	36.28	0.65	152	36.15	30.60	235	36.34	31.13	258
qnet1_o_c	64.09	0.05	57	57.94	22.83	152	65.00	15.87	163
rgn_c	47.73	0.01	88	33.15	0.03	149	47.73	0.03	161
roll3000_c	13.95	2.57	663	22.09	32.58	1837	22.09	32.72	1922
rout_c	3.13	0.14	160	3.13	0.40	398	3.13	0.43	423
set1ch_c	39.30	0.07	580	57.94	1.01	657	57.94	1.06	772
swath_c	27.83	0.64	100	27.83	9.49	258	27.83	9.20	285
timtab1_c	34.17	0.05	579	44.15	0.75	1203	44.34	0.67	1337
timtab2_c	24.93	0.09	781	33.78	2.97	1977	34.01	13.65	2097
tr12-30_c	60.27	0.11	1146	84.50	14.44	1558	84.50	11.59	1610
vpm1_c	83.64	0.01	81	83.64	0.05	107	83.64	0.06	116
vpm2_c	49.81	0.02	193	51.16	0.10	278	51.16	0.09	303
Avg.	28.72	0.85	172.92	32.54	2.72	270.13	33.14	2.83	293.81

CPU time is in seconds

4.5 Comparison with the original Reduce-and-Split

In this section we compare CglRedSplit2, the cut generator based on the strategies proposed in this paper, with CglRedSplit, the Reduce-and-Split generator based on [3]. The two algorithms are described in Sect. 3. Let $\bar{x}$ be the optimal LP solution, and let $F = \{i \in B_I : \bar{x}_i \notin \mathbb{Z}\}$. We remark that CglRedSplit2 implements a large number of cut generation strategies that can be combined together, with the result that more than $1000|F|$ cuts can potentially be generated at each round. On the other hand, the algorithm described in [3] and implemented in CglRedSplit produces at most $|F|$ cuts. There is another important difference between the two generators: whenever CglRedSplit fails to improve a row k with associated basic variable $i \in F$ (i.e. it cannot find another row that can be combined with k to reduce the row coefficients), it generates a simple GMI cut from k. On the other hand, in our current implementation CglRedSplit2 only generates those cuts that are *not* simple GMI cuts; in other words, it only generates cuts obtained from linear combinations of at least two rows. In order to compare CglRedSplit and CglRedSplit2, we must take into account these differences. Through all this section, CglRedSplit2 uses parameters $\sigma = 0.01$, $\Lambda = 10$, $\gamma = 0.0001$, and a time limit of 30 seconds for cut generation.

The comparison is performed on the set MILP_C, by measuring the integrality gap closed by the tested cut generators after 10 rounds of cutting planes. After each round of cut generation, we reoptimize the LP and remove inactive cuts; these inactive cuts are stored into a pool, and at subsequent rounds we add back into the LP all cuts that

are found to be violated. To generate a similar number of cuts from both generators, we proceed as follows. For each instance, we first apply 10 rounds of cutting planes with CglRedSplit, yielding p cuts. Then, we generate cuts with CglRedSplit2, with a limit of p cuts in total. This way, the number of cuts generated by CglRedSplit2 does not exceed that of CglRedSplit.

We test two versions of CglRedSplit2. The first one mimicks the cut generation algorithm of CglRedSplit: for each row whose associated basic variable is in F, we perform only one attempt to compute a combination of rows that yields improved cut coefficients; if this process fails, we generate the GMI cut from the unmodified row. The single combination is computed using $\mu = 5$, the first column selection strategy of C-3P (smallest reduced costs), and row selection strategy RS8. We label this version SINGLE CUT. SINGLE CUT is designed to allow a fair comparison with CglRedSplit, as it generates at most $|F|$ cuts per round. In order to measure how many "good" cutting planes we are missing by generating $|F|$ cuts per round only, we also test a DEFAULT version of CglRedSplit2, which uses all column selection strategies, row selection strategies RS7-RS8, and $\mu = 3, 5$. This is the a similar configuration to the one used in our Branch-and-Cut experiments (Sect. 4.6), but here we allow CglRedSplit2 to generate a GMI cut from a single row whenever its coefficients cannot be reduced with our algorithm.

Furthermore, we design an experiment to assess the importance of the coefficient reduction algorithm. The purpose of this experiment is to verify whether the cutting planes obtained through our method are effective thanks to the row selection procedures only, or if the coefficient reduction algorithm also plays an important role. Therefore, we generate split cuts obtained using our usual row selection strategies, but instead of computing the multipliers through (9), we generate random integer row multipliers for the linear combinations. To do this, we need a procedure to randomly sample an integral vector λ of dimension $k = \mu - 1$ such that $\|\lambda\|_1 \leq \Lambda - 1$; this is because one component of the vector of row multipliers (the one corresponding to the initial row being modified) has to be equal to 1. We proceed as follows. First we uniformly sample a direction $\tilde{\lambda} \in \mathbb{R}^k$ at random in the unit L_∞ ball; then we sample a length ℓ from a uniform distribution over the interval $[0, (\Lambda - 1)/\|\tilde{\lambda}\|_1)]$. Finally, we obtain an integral vector by rounding each component of $\ell\tilde{\lambda}$ to the nearest integer. This yields a procedure to obtain integral row multipliers. We use it instead of our usual algorithm in SINGLE CUT, i.e. for each row whose basic integer variable is fractional, we select $\mu = 5$ rows with the RS8 strategy, and generate a random integer combination. Hence, we generate at most $|F|$ cuts per round, and we stop after generating as many cuts as CglRedSplit over 10 round. We call this generator RANDOM.

Results are reported in Table 5; for each cut generation algorithm, we report the gap closed at the root node after 10 rounds, the required CPU time, and the number of generated cuts. Geometric averages can be found in the last row.

Table 5 shows that our Reduce-and-Split algorithm is more effective than the one described in [3], when compared on equal ground. Indeed, CglRedSplit2 SINGLE CUT closes more gap than CglRedSplit on average while generating approximately the same number of cuts: the improvement is of $\approx$10% in relative terms. At the same time, RANDOM is weaker than both CglRedSplit and CglRedSplit2 SINGLE CUT on average, showing that the coefficient reduction algorithm is an important component

Table 5 Comparison of CglRedSplit and CglRedSplit2 after 10 rounds of cut generation at the root node

Instance	CglRedSplit			CglRedSplit2 SINGLE CUT			CglRedSplit2 RANDOM			CglRedSplit2 DEFAULT		
	Gap (%)	Time	# cuts	Gap (%)	Time	# cuts	Gap (%)	Time	# cuts	Gap (%)	Time	# cuts
10teams_c	100.00	2.78	4	100.00	0.47	4	85.71	1.17	4	100.00	302.49	19
a1c1s1_c	6.75	1.94	198	27.69	2.14	198	27.69	2.08	198	64.22	306.36	1180
aflow30a_c	29.77	0.27	305	29.87	0.30	305	26.04	0.24	305	33.03	31.10	3428
aflow40b_c	23.53	0.43	81	17.61	0.27	81	16.72	0.23	81	25.41	83.57	345
arki001_c	39.97	0.48	187	46.43	0.37	187	42.99	0.37	187	61.43	98.81	4903
b1c1s1_c	29.66	6.89	449	30.62	6.07	449	30.43	6.08	449	46.01	322.50	1657
b2c1s1_c	25.50	8.65	428	27.15	5.31	428	25.28	5.08	428	42.86	326.97	1730
bell3a_c	53.82	0.00	17	25.84	0.00	17	21.34	0.01	17	72.34	0.99	335
bell4_c	57.76	0.02	199	76.00	0.02	199	80.67	0.01	199	94.55	2.00	2742
bell5_c	40.29	0.01	76	24.88	0.01	76	23.12	0.01	76	27.70	0.94	1161
bg512142_c	3.64	4.00	1621	2.21	3.25	1621	1.77	2.78	1621	1.91	284.77	19914
blend2_c	23.98	0.03	86	33.22	0.05	86	25.64	0.03	86	31.55	5.84	1398
danoint_c	0.45	0.16	280	0.90	0.19	280	0.57	0.17	280	0.95	12.25	5047
dcmulti_c	65.45	0.24	462	70.95	0.19	462	68.56	0.16	462	71.45	16.86	5705
dg012142_c	0.28	32.12	1315	0.51	39.20	1315	0.45	31.08	1315	0.60	377.29	4539
dsbmip_mod_c	0.00	0.35	412	0.00	1.35	152	0.00	1.37	137	0.00	105.66	487
egout_c	28.91	0.01	97	73.09	0.02	97	65.65	0.01	97	83.65	2.59	2325
fiber_c	89.68	0.20	130	80.89	1.59	130	70.68	1.71	130	91.89	304.15	1641
fixnet3_c	7.02	0.05	58	67.82	0.05	58	49.69	0.03	58	85.66	36.11	3091
fixnet4_c	10.33	0.08	93	42.34	0.10	93	30.32	0.06	93	73.58	87.78	5503
fixnet6_c	11.96	0.09	128	60.48	0.18	128	33.67	0.09	128	66.15	54.95	3929
flugpl_c	14.23	0.00	66	67.77	0.00	66	13.84	0.00	66	99.82	0.09	726
gen_c	75.51	0.10	124	70.29	0.06	124	69.01	0.06	124	79.71	15.50	3575
gesa2_c	93.48	0.21	337	94.07	0.58	337	77.02	0.54	337	95.39	87.69	5938
gesa2_o_c	92.49	0.39	408	77.91	1.09	408	76.14	0.80	408	97.11	166.03	5996
gesa3_c	79.31	0.32	211	55.06	0.39	211	55.26	0.27	211	82.44	99.36	1820
gesa3_o_c	76.46	0.39	199	57.75	0.40	199	62.94	0.30	199	83.44	161.96	2717
glass4_mod_c	0.00	0.05	631	0.00	0.05	631	0.00	0.05	631	0.00	4.69	878
khb05250_c	74.98	0.06	93	94.53	0.07	56	84.53	0.06	93	96.44	3.61	308
misc06_c	17.60	0.02	16	39.63	0.02	16	9.63	0.02	16	74.01	3.44	601
mod011_c	35.14	0.96	136	3.42	3.26	129	15.41	4.97	136	15.76	298.92	568
modglob_c	21.13	0.06	176	61.51	0.10	176	42.96	0.07	176	74.47	9.95	2736
momentum1_c	47.68	2.86	68	22.51	271.38	68	11.00	207.85	68	47.51	5842.28	715
net12_c	14.85	576.61	3675	13.28	1021.29	3675	12.14	879.24	3675	13.63	2029.76	7068
noswot_c	0.00	0.01	114	0.00	0.02	114	0.00	0.01	114	0.00	1.77	2458
pk1_c	0.00	0.01	143	0.00	0.01	142	0.00	0.01	143	0.00	0.58	1614
pp08a_c	91.21	0.07	367	94.55	0.07	331	82.27	0.06	367	95.85	5.16	2937
pp08aCUTS_c	70.84	0.09	359	69.29	0.10	359	55.26	0.07	359	83.07	6.12	4617
qiu_c	12.41	0.30	193	13.99	0.18	193	12.95	0.17	193	12.16	19.77	1805

Table 5 Continued

Instance	CglRedSplit			CglRedSplit2 SINGLE CUT			CglRedSplit2 RANDOM			CglRedSplit2 DEFAULT		
	Gap (%)	Time	# cuts	Gap (%)	Time	# cuts	Gap (%)	Time	# cuts	Gap (%)	Time	# cuts
qnet1_c	32.19	0.63	333	37.84	11.93	333	25.95	6.82	333	37.99	313.42	1780
qnet1_o_c	66.11	0.54	333	61.03	17.27	333	53.66	13.50	333	64.04	310.51	1133
rgn_c	98.80	0.01	53	52.95	0.00	53	5.65	0.01	53	77.52	0.78	1654
roll3000_c	57.04	4.13	1297	3.79	16.56	1297	4.37	13.68	1297	38.06	320.07	4803
rout_c	9.54	0.17	269	12.74	0.13	269	7.83	0.12	269	9.70	12.75	3551
set1ch_c	83.66	0.59	929	88.10	0.95	911	74.55	0.73	929	86.06	79.49	13836
swath_c	28.93	0.90	93	22.67	2.89	93	22.67	3.04	93	29.84	309.02	781
timtab1_c	42.44	0.29	1029	43.99	0.23	1029	42.53	0.20	1029	58.05	25.59	11263
timtab2_c	32.14	1.86	1878	32.66	1.15	1878	31.64	1.23	1878	49.76	160.66	21396
tr12-30_c	82.22	2.20	1265	98.22	14.07	1265	95.21	13.86	1265	97.63	309.15	2644
vpm1_c	76.36	0.01	97	85.45	0.02	97	49.13	0.02	97	100.00	2.67	624
vpm2_c	63.27	0.06	253	42.73	0.06	253	40.32	0.04	253	59.47	7.88	3738
Avg.	31.54	0.90	206.41	34.10	1.53	203.45	27.42	1.44	206.41	45.20	45.71	2164.86

of our procedure. Finally, CglRedSplit2 DEFAULT generates roughly twice as many cuts as SINGLE CUT, but closes significantly more gap over 10 rounds. This suggests that using a larger family of cuts than those generated by SINGLE CUT (which mimicks the original Reduce-and-Split algorithm implemented in CglRedSplit) has some advantages: at the cost of increased CPU time and a larger number of cuts, we are able to close more gap. On some of the larger instances (e.g. momentum1), our cut generator required too much time, partly because of the LP solver taking longer than usual to provide the optimal tableau, and the time limit was exceeded; in a practical framework, this behavior should be detected in order to stop using the generator.

4.6 Branch-and-Cut: mixed-integer instances

We now test the cut generation strategies proposed in this paper on the test set MILP_C within a Branch-and-Cut framework. We employ a Branch-and-Cut code based on COIN-OR Cbc. The setup is as follows: we apply 10 rounds of cutting planes at the root node, then we branch until optimality is proven or a time limit of two hours (including cut generation) is hit, with node selection criterion set to *best bound*. The value of the optimal solution minus a small tolerance is given as a cutoff; the tolerance on integrality gap to prove optimality of a solution is set to zero. The reason for using a cutoff value slightly below the optimal solution value is that we want to make sure that the time of discovery of an integer solution does not have an influence on the size of the enumeration tree. Independent testing was conducted on our cut generator to ensure the validity of the cutting planes. For the experiments in this section, since we generate a large number of cuts, we used a safer set of parameters for *all* cut generators in order

to avoid numerical troubles: cuts were discarded if they were violated by $<10^{-4}$ or if the ratio between the largest and the smallest coefficients exceeded 10^6. Note that by default, Cbc does not provide advanced cut management techniques: whenever a cut becomes inactive, it is removed from the LP. Thus, we implemented a simple cut management mechanism: all generated cuts (up to 20000) are stored into a cut pool, and at selected nodes in the enumeration tree, we add back into the LP any violated cut that is found. We call this a *cut pool iteration*. If a cut is found to be inactive for a total of 10 cut pool iterations, then it is permanently removed from the cut pool. Cut pool iterations are performed at the root node after each round of cut generation, and in the enumeration tree at all nodes whose depth is multiple of 4.

Within this framework, we compare three sets of cut generators: CGLALLCUTS, which represents our baseline, CGLCUTS + CglRedSplit2 (i.e. we employ CglRedSplit2 instead of CglRedSplit), and CGLALLCUTS + CglRedSplit2 (i.e. CglRedSplit2 is used in conjunction with all split cut generators in Cgl, including CglRedSplit). The parameters for the CglRedSplit2 generator in these experiments are as follows: $\Lambda = 10, \sigma = 0.01, \gamma = 0.0001, \mu = 5$, and we employ the row selection strategies RS7-RS8, as well as all column selection strategies discussed in Sect. 3.1. We set a time limit of 30 s for each round of cut generation with CglRedSplit2. For easy instances, we do not expect this setup to perform well, since too much time is spent cutting; however, our aim is to prove that our Reduce-and-Split cuts are advantageous for the remaining instances, where investing more time in cut generation should prove useful. In particular, we want to show that for difficult instances, our cut generation algorithm is able to find useful split cuts in a reasonably small computational time, so that we can solve those instances more quickly in a Branch-and-Cut framework. Results are reported in Table 6. For each instance and for each set of cut generators, we report, in the first three columns, the percentage of integrality gap closed at the root by cutting (after 10 rounds), the time spent in cut generation, and the total number of generated cuts; in columns 4 to 6 we report the gap closed at the end of the enumeration (i.e. after a total of 2 h, if optimality is not proven first), the total CPU time in seconds (including cut generation), and the number of enumerated nodes. Note that in these tables, the fraction of integrality gap closed refers to the integrality gap remaining after preprocessing. Instance `noswot_c` is not reported here because none of the tested methods was able to close any integrality gap.

In Table 7 we report the geometric averages for the numbers in Table 6, where we group the instances into three different sets: Easy instances (those that can be solved in <30 s employing the baseline CGLALLCUTS), Medium instances (those that can be solved in <2 h employing the baseline CGLALLCUTS and are not Easy), Hard instances (unsolved after 2 h). The Easy set contains 21 instances, the Medium set contains 13 instances, the Hard set contains 16 instances. Additionally, we consider a subset of the Medium instances, where we exclude outliers, i.e. instances for which one the methods performs significantly worse than the remaining methods. These instances are: `flugpl_c`, on which CGLALLCUTS has very poor performance, `modglob_c`, `qiu_c`, and `set1ch_c` on which at least one of the two sets of cut generators that include CglRedSplit2 does not manage to solve the instance within the time limit. These outliers are likely due to numerical problems (in the case of `flugpl_c`) and bad branching choices, possibly due to the combination of cutting planes. By providing averages for a

Table 6 Results obtained in a Branch-and-Cut framework over the set of test instances MILP_C

Instance	CglAllCuts						CglCuts + CglRedSplit2						CglAllCuts + CglRedSplit2					
	Cuts at root			Total			Cuts at root			Total			Cuts at root			Total		
	Gap (%)	Time	# cuts	Gap (%)	Time	# nodes	Gap (%)	Time	# cuts	Gap (%)	Time	# nodes	Gap (%)	Time	# cuts	Gap (%)	Time	# nodes
10teams_c	100.00	2.31	12	100.00	2.36	0	100.00	30.71	26	100.00	30.81	0	100.00	31.58	29	100.00	31.68	0
a1c1s1_c	53.56	131.25	1239	72.95	7202.00	71826	55.24	382.11	1630	73.72	7201.21	42877	57.09	369.26	1626	74.68	7201.61	55890
aflow30a_c	54.19	8.97	1692	100.00	453.98	35430	52.96	17.82	4786	100.00	359.95	31038	54.05	18.65	4862	100.00	430.59	34464
aflow40b_c	44.30	29.49	617	82.67	7204.36	149196	44.57	95.93	773	77.70	7204.66	129700	45.33	101.59	781	79.18	7205.04	139194
arki001_c	60.92	12.20	934	83.49	7204.25	419779	74.88	33.44	4139	94.13	7201.99	237575	74.43	29.93	4284	95.87	7202.82	296738
b1c1s1_c	41.37	220.27	1626	65.86	7201.92	62689	39.76	466.91	1890	62.14	7202.67	74328	39.22	471.03	1864	61.90	7202.44	75087
b2c1s1_c	32.28	182.98	1710	59.72	7202.43	51149	36.56	508.61	1856	55.97	7201.92	31410	36.94	522.03	2000	58.11	7202.18	37686
bell3a_c	70.74	0.04	132	100.00	20.47	29004	75.44	0.30	367	100.00	17.45	24942	75.35	0.33	378	100.00	16.40	23552
bell4_c	95.88	0.41	682	100.00	46.65	53232	95.22	1.01	1475	100.00	44.34	36754	95.19	1.14	1852	100.00	49.41	47562
bell5_c	94.58	0.14	355	100.00	69.12	149708	96.22	0.48	1197	100.00	3.19	5474	95.53	0.48	1399	100.00	3.57	5618
bg512142_c	3.96	64.10	1994	27.70	7202.40	163180	5.76	197.22	11737	28.81	7201.07	86664	5.06	147.40	9857	37.53	7201.50	93719
blend2_c	40.08	0.51	603	100.00	3003.58	2044	36.85	1.70	1308	100.00	5895.07	2290	36.77	1.71	1479	100.00	1375.89	1276
danoint_c	2.03	7.97	954	43.82	7202.95	207174	1.74	12.61	3831	45.22	7203.76	209074	1.63	12.91	4204	48.07	7202.90	213601
dcmulti_c	90.23	4.64	1265	100.00	7.89	202	93.91	7.18	3281	100.00	10.60	22	90.25	9.20	4218	100.00	13.70	22
dg012142_c	0.76	435.30	3450	25.05	7202.45	47996	0.82	720.66	4462	21.31	7202.42	42416	0.85	722.37	4470	19.78	7202.06	38655
dsbmip_mod_c	0.00	10.03	326	100.00	36.48	127	0.00	39.05	311	100.00	84.00	453	0.00	28.64	399	100.00	56.70	266
egout_c	100.00	0.04	288	100.00	0.05	0	100.00	0.14	483	100.00	0.16	0	100.00	0.15	505	100.00	0.16	0
fiber_c	93.43	4.37	1208	100.00	6.03	288	95.85	123.77	2372	100.00	126.29	78	95.57	126.47	2609	100.00	131.28	270
fixnet3_c	100.00	0.23	378	100.00	0.27	0	100.00	2.62	679	100.00	2.68	0	100.00	2.70	749	100.00	2.76	0
fixnet4_c	90.55	3.56	942	100.00	12.96	596	91.34	13.22	2147	100.00	27.59	1072	89.88	13.32	2257	100.00	28.98	1064
fixnet6_c	83.58	3.80	738	100.00	13.01	564	87.16	12.08	2705	100.00	20.61	886	83.82	14.44	3024	100.00	714.47	112574

Table 6 continued

Instance	CGLALLCUTS						CGLCUTS + CglRedSplit2						CGLALLCUTS + CglRedSplit2					
	Cuts at root			Total			Cuts at root			Total			Cuts at root			Total		
	Gap (%)	Time	# cuts	Gap (%)	Time	# nodes	Gap (%)	Time	# cuts	Gap (%)	Time	# nodes	Gap (%)	Time	# cuts	Gap (%)	Time	# nodes
flugpl_c	76.26	0.02	253	100.00	1124.84	486	99.53	0.05	607	100.00	0.07	16	100.00	0.03	378	100.00	0.03	0
gen_c	100.00	0.16	220	100.00	0.18	0	100.00	0.72	559	100.00	0.77	0	100.00	0.94	549	100.00	1.01	0
gesa2_c	98.20	4.71	867	100.00	14.24	1356	99.29	19.90	2382	100.00	22.71	114	99.14	22.72	2855	100.00	25.36	118
gesa2_o_c	98.65	9.22	1398	100.00	15.57	328	99.29	39.85	4566	100.00	45.73	214	100.00	33.01	2836	100.00	33.42	0
gesa3_c	86.95	12.40	960	100.00	16.51	56	91.73	30.66	1514	100.00	32.43	12	88.89	32.14	1422	100.00	33.68	26
gesa3_o_c	95.20	7.22	924	100.00	10.17	20	95.71	39.62	1766	100.00	43.05	26	95.67	37.01	1928	100.00	42.51	66
glass4_mod_c	0.00	3.92	3511	65.00	7216.29	2048985	0.00	11.42	10694	56.25	7210.59	2031187	0.00	11.14	11104	50.00	7215.34	1886240
khb05250_c	99.32	0.82	259	100.00	1.33	20	99.40	1.48	324	100.00	1.91	14	99.32	1.43	346	100.00	1.94	14
misc06_c	95.94	0.57	134	100.00	0.90	4	98.90	1.53	382	100.00	1.93	2	99.59	1.58	441	100.00	1.92	0
mod011_c	35.01	178.48	227	100.00	5144.85	22824	31.33	373.14	580	100.00	2676.06	6014	41.35	313.62	730	100.00	2861.35	5036
modglob_c	78.82	1.62	586	100.00	79.95	23856	71.23	6.21	1454	96.94	7225.39	2069437	69.74	5.50	1470	100.00	781.65	201912
momentum1_c	47.68	423.10	356	47.68	7244.67	40	47.68	694.35	379	47.68	7223.66	29	47.68	823.58	385	47.68	7222.20	39
net12_c	25.03	1518.89	23069	25.03	7210.83	0	26.45	2160.60	22027	26.45	7318.06	0	23.44	2485.44	23321	23.44	7444.05	0
noswot_c	0.00	1.76	837	0.00	7272.93	4628492	0.00	4.90	4476	0.00	7279.19	5164617	0.00	4.71	4972	0.00	7259.09	4957567
pk1_c	0.00	0.52	764	100.00	248.84	309336	0.00	0.91	3620	100.00	235.93	294036	0.00	1.02	2996	100.00	221.98	269874
pp08a_c	93.27	1.53	1236	100.00	12.28	3246	96.62	2.78	1965	100.00	9.30	1756	96.66	3.53	3132	100.00	11.49	1794
pp08aCUTS_c	88.25	2.67	1131	100.00	20.56	4200	83.31	4.38	3402	100.00	20.14	3612	84.28	4.86	4203	100.00	22.66	4248
qiu_c	18.87	14.51	604	100.00	4686.73	111816	19.89	17.87	1471	100.00	3732.19	122658	17.71	20.28	1413	93.93	7200.83	158034
qnet1_c	99.48	18.60	702	100.00	20.89	2	100.00	309.63	2877	100.00	312.37	0	100.00	313.31	3465	100.00	315.76	0
qnet1_o_c	100.00	9.71	1291	100.00	11.38	0	100.00	300.53	3092	100.00	302.80	0	100.00	294.01	3866	100.00	296.37	0

Table 6 continued

Instance	CglAllCuts						CglCuts + CglRedSplit2						CglAllCuts + CglRedSplit2					
	Cuts at root			Total			Cuts at root			Total			Cuts at root			Total		
	Gap (%)	Time	# cuts	Gap (%)	Time	# nodes	Gap (%)	Time	# cuts	Gap (%)	Time	# nodes	Gap (%)	Time	# cuts	Gap (%)	Time	# nodes
rgn_c	96.88	0.13	446	100.00	0.77	276	75.72	0.69	1808	100.00	3.52	2278	98.80	0.37	1409	100.00	0.55	26
roll3000_c	76.14	71.01	4452	87.88	7201.10	46539	74.84	346.20	12084	91.22	7201.42	44375	79.18	377.64	9648	89.74	7202.38	59226
rout_c	33.47	4.82	1495	100.00	1015.86	132920	40.62	11.59	5232	100.00	2426.48	252694	37.67	9.74	5265	100.00	1391.91	166928
set1ch_c	95.55	10.91	2752	100.00	120.75	12054	92.40	27.57	8973	98.88	7212.58	680921	91.35	29.64	9722	97.68	7213.31	543346
swath_c	29.91	26.82	492	37.15	7240.37	177197	30.37	288.59	1995	38.23	7234.21	168198	29.87	305.33	1595	53.23	7225.77	183080
timtab1_c	63.62	6.41	4036	85.35	7220.74	1506843	56.85	18.60	12240	76.56	7227.45	1522520	60.85	20.42	13474	75.66	7221.10	1102357
timtab2_c	43.54	28.75	6030	50.71	7207.17	461949	50.24	99.29	21448	54.99	7211.52	463484	48.14	120.17	22293	52.07	7206.00	273056
tr12-30_c	96.00	74.83	3916	97.17	7200.95	26852	95.04	350.33	7068	95.85	7200.49	19055	95.08	330.41	7149	95.93	7200.60	20453
vpm1_c	89.65	0.28	392	100.00	1.28	174	100.00	0.25	275	100.00	0.27	0	100.00	0.29	360	100.00	0.33	0
vpm2_c	76.82	1.89	1231	100.00	46.05	20176	96.47	5.22	2920	100.00	6.94	22	95.43	5.29	3325	100.00	7.54	70
Avg.	40.55	8.23	841.83	75.59	181.35	2614.26	41.51	22.56	1957.24	75.22	249.77	1761.62	41.57	22.81	2046.49	75.67	247.34	1507.97

CPU time is in seconds

Table 7 Average values for Table 6, grouped by the difficulty of the instances

	Cuts at root			Total		
	Gap (%)	Time	# cuts	Gap (%)	Time	# nodes
Easy instances						
CglAllCuts	93.53	2.43	493.49	100.00	5.43	62.13
CglCuts + CglRedSplit2	94.13	9.21	1108.94	100.00	14.21	33.85
CglAllCuts + CglRedSplit2	94.84	9.51	1208.95	100.00	16.58	29.16
Medium instances						
CglAllCuts	56.79	3.86	716.51	100.00	411.28	21891.21
CglCuts + CglRedSplit2	58.88	7.56	1896.76	99.62	324.49	12766.84
CglAllCuts + CglRedSplit2	59.20	7.45	1978.87	99.22	244.50	8256.71
Medium instances (without outliers)						
CglAllCuts	56.30	3.95	723.21	100.00	378.06	32128.91
CglCuts + CglRedSplit2	58.06	7.90	1925.26	100.00	213.45	6517.84
CglAllCuts + CglRedSplit2	59.74	7.64	2185.68	100.00	172.12	6841.40
Hard instances						
CglAllCuts	26.89	73.43	1950.15	54.18	7209.89	35047.60
CglCuts + CglRedSplit2	28.10	207.68	4066.61	53.95	7215.71	28634.76
CglAllCuts + CglRedSplit2	27.82	212.71	4010.94	55.64	7222.60	29877.28

set of instances without these extreme cases, we obtain a better comparison. Instances where none of the methods in the comparison managed to close a nonzero amount of integrality gap by cutting are not taken into account when computing the averages in Table 7.

Table 7 suggests that the cut generation methods proposed in this paper are useful for difficult instances. In particular, on average we consistently close more gap at the root node on Easy, Medium and Hard instances, which results in a smaller enumeration tree (or in a larger closed gap when the time limit is hit). On Easy instances we reduce the number of nodes by a factor of two; however, this does not yield shorter computing time, because of the time required for cut generation, which almost quadruples. Similarly, on Medium instances we obtain a large reduction in the number of nodes, but now this is also reflected by a reduction of total computing time: CglAllCuts + CglRedSplit2 is 40% faster than CglAllCuts on average. The improvement in the average number of nodes and computing time is visible on the Medium (without outliers) instances as well; this confirms our results. Observe that on Medium instances, CglRedSplit2 generates more than 1000 cuts on average, but the extra computing time is offset by the reduction of the enumeration tree. However, 3 out of the 4 instances that constitute the set of outliers are in general not favorable to CglRedSplit2; this suggests that in some cases, generating too many split cuts could lead to bad branching choices or numerical problems. In a practical setting, care should be taken to detect this behaviour.

On Hard instances, we consistently close more gap per node: even though by using CglRedSplit we enumerate on average fewer nodes in two hours, we close a very similar amount of gap. This indicates that the full enumeration tree, should the instances be solved to optimality, is likely to be smaller. Note that to achieve this result, we invest two extra minutes of cut generation at the root; however, this seems to pay off. The drawback of generating 2000 extra cutting planes at the root (on average) is that the we enumerate fewer nodes in the allotted time frame: because processing the root node takes longer, and because cut pool iterations are more expensive, especially at the beginning of Branch-and-Bound where all generated cuts are still in the pool.

4.7 Branch-and-Cut: integer instances

We now test the cut generation methods proposed in this paper on the instances of the test set `ILP_C` in a Branch-and-Cut framework. We use *exactly* the same setup and parameters described in Sect. 4.6; therefore, we apply 10 rounds of cutting planes at the root node, and then branch until optimality is proven or a time limit of 2 h is hit.

We apply our Branch-and-Cut algorithm on the instances of the test set `ILP_C`. On several instances, our cut generator does not generate any cut in 10 rounds at the root node. We do not report results for these instances, which are: `air04_c`, `air05_c`, `cap6000_c`, `disctom_c`, `fast0507_c`, `manna81_c`, `nw04_c`, `t1717_c`. On most of them, very few cuts (if any) are generated by all cut generators, because they are too dense or have bad numerics; hence, the cuts are discarded. Our generator displays the same behaviour. We are left with 25 instances.

Full results are reported in Table 8; column labels are the same as in Table 6, and we test the same set of cut generators; that is, CGLALLCUTS versus CGLCUTS + CglRedSplit2 and CGLALLCUTS + CglRedSplit2. In Table 9 we report the average values, grouped by instance difficulty, as we did for Table 7. There are 13 Easy instances, 5 Medium instances, and 7 Hard instances.

On this set of instances, our cuts are not very effective. Good results are obtained on Medium instances, on average, but overall, there is no clear winner and it is hard to conclude anything. The extra computation time required to generate Reduce-and-Split cuts does not seem to pay off. Summarizing, our cut generator is very effective on instances with many continuous (structural) variables, i.e. mixed-integer instances, but does not perform as well on pure integer instances.

5 Conclusion

In this paper, we presented a cut generation algorithm based on the idea of reducing cut coefficients on the continuous nonbasic variables, by computing integral linear combinations of rows of the optimal simplex tableau. The coefficient reduction algorithm solves a norm minimization problem through the solution of a linear system of equations. We discussed several heuristic procedures to select the rows involved in the combination, and the set of columns on which the coefficient reduction algorithm should focus. We provided a detailed computational testing of the proposed ideas,

Table 8 Results obtained in a Branch-and-Cut framework over the set of test instances `ILP_C`

Instance	CglAllCuts						CglCuts + CglRedSplit2						CglAllCuts + CglRedSplit2					
	Cuts at root			Total			Cuts at root			Total			Cuts at root			Total		
	Gap (%)	Time	# cuts	Gap (%)	Time	# nodes	Gap (%)	Time	# cuts	Gap (%)	Time	# nodes	Gap (%)	Time	# cuts	Gap (%)	Time	# nodes
ds_c	0.15	929.06	24	0.18	7221.66	267	0.05	942.92	2	0.12	7204.29	1195	0.15	998.00	24	0.18	7201.56	293
enigma_c	0.00	0.41	778	100.00	0.51	0	0.00	0.73	2905	100.00	0.81	0	0.00	0.98	3443	100.00	1.10	0
gt2_c	100.00	0.05	392	100.00	0.06	0	100.00	0.14	586	100.00	0.15	0	100.00	0.28	876	100.00	0.30	0
harp2_c	70.03	16.97	807	100.00	2816.85	461098	70.42	148.42	1105	100.00	2042.31	287514	70.76	155.00	1262	100.00	1575.47	232154
l152lav_c	24.74	4.27	113	100.00	18.44	1288	21.10	23.58	149	100.00	38.53	1584	24.42	37.62	163	100.00	48.22	1128
lseu_c	96.50	0.10	563	100.00	0.20	6	79.97	0.50	2497	100.00	1.17	610	93.72	0.37	2125	100.00	0.50	10
mas74_c	8.56	0.37	392	100.00	3749.19	5560154	9.39	0.90	882	100.00	3255.79	5009822	9.39	1.00	883	100.00	3341.88	5009822
mas76_c	9.20	0.33	397	100.00	196.89	324358	11.70	0.57	1435	100.00	219.11	380334	11.70	0.58	1437	100.00	222.69	380334
misc03_c	28.07	1.12	1189	100.00	6.09	846	26.82	2.18	3629	100.00	5.98	870	25.82	2.46	3964	100.00	5.99	560
misc07_c	6.52	1.91	1090	100.00	104.91	22310	6.21	3.44	2988	100.00	107.60	26724	7.97	3.60	3287	100.00	99.91	24650
mitre_c	100.00	15.73	3049	100.00	16.79	0	100.00	584.08	3214	100.00	585.16	0	100.00	458.91	3288	100.00	460.25	0
mkc_c	68.39	64.74	2762	94.13	7200.57	32831	58.22	483.40	2946	91.52	7202.91	17725	39.78	455.37	2843	97.53	7201.18	66019
mod008_c	88.99	0.21	450	100.00	0.48	24	65.23	0.74	1487	100.00	2.02	532	71.70	0.82	1873	100.00	1.91	320
nsrand-ipx_c	78.48	116.51	1076	83.11	7202.25	41276	80.31	405.94	1487	84.47	7202.06	34201	79.97	430.79	1421	84.25	7202.05	27762
opt1217_c	57.78	3.10	1174	57.78	7228.73	1171310	58.11	9.51	2973	58.26	7241.98	1208171	62.56	9.35	3185	63.59	7239.57	1258949
p0033_c	100.00	0.03	333	100.00	0.03	0	100.00	0.15	1356	100.00	0.17	0	100.00	0.08	885	100.00	0.10	0
p0201_c	82.47	2.06	1845	100.00	6.35	330	84.42	5.88	5078	100.00	10.86	218	87.77	5.65	4814	100.00	9.37	6
p0282_c	98.44	1.24	1481	100.00	1.63	10	98.55	2.87	4959	100.00	3.96	196	98.51	3.40	5080	100.00	5.70	788
p0548_c	99.99	0.41	1021	100.00	0.47	0	100.00	6.18	1551	100.00	6.28	0	100.00	8.34	1865	100.00	8.50	0
p2756_c	99.43	1.77	1119	100.00	2.43	20	99.66	63.03	1553	100.00	63.44	4	99.58	81.53	1851	100.00	82.01	4

Table 8 continued

Instance	CglAllCuts						CglCuts + CglRedSplit2						CglAllCuts + CglRedSplit2					
	Cuts at root			Total			Cuts at root			Total			Cuts at root			Total		
	Gap (%)	Time	# cuts	Gap (%)	Time	# nodes	Gap (%)	Time	# cuts	Gap (%)	Time	# nodes	Gap (%)	Time	# cuts	Gap (%)	Time	# nodes
protfold_c	26.00	28.31	1173	26.00	7207.28	145	27.17	434.56	1369	27.17	7237.58	10	27.28	410.59	1363	27.28	7216.42	161
seymour_c	24.54	365.43	6282	38.57	7200.54	5295	21.07	423.99	9324	34.88	7200.84	4697	23.14	726.38	8871	38.61	7201.16	4510
sp97ar_c	26.60	429.67	290	42.54	7201.99	18743	27.54	937.37	262	42.33	7202.07	10664	25.65	826.68	286	40.15	7202.20	13469
stein27_c	−0.00	0.42	820	100.00	3.17	8016	−0.00	0.67	2809	100.00	3.58	8200	−0.00	0.66	2854	100.00	3.59	7880
stein45_c	0.00	1.85	1313	100.00	71.41	73052	0.00	2.82	4322	100.00	84.00	80888	0.00	2.80	4349	100.00	89.29	90888
Avg.	24.88	5.75	755.72	71.37	72.63	756.93	24.18	16.62	1416.38	71.04	115.43	967.97	24.61	17.77	1606.01	71.76	116.55	793.92

CPU time is in seconds

Table 9 Average values for Table 8, grouped by the difficulty of the instances

	Cuts at root			Total		
	Gap (%)	Time	# cuts	Gap (%)	Time	# nodes
Easy instances						
CglAllCuts	76.07	1.24	750.64	100.00	2.27	14.73
CglCuts + CglRedSplit2	71.56	5.98	1685.97	100.00	8.19	34.01
CglAllCuts + CglRedSplit2	74.19	6.58	1776.54	100.00	8.46	16.93
Medium instances						
CglAllCuts	14.11	2.12	608.37	100.00	685.01	369067.11
CglCuts + CglRedSplit2	15.15	5.67	1429.87	100.00	630.54	347848.77
CglAllCuts + CglRedSplit2	16.07	5.90	1514.78	100.00	586.28	323146.14
Hard instances						
CglAllCuts	24.54	103.52	786.38	29.72	7209.00	9331.47
CglCuts + CglRedSplit2	23.50	318.48	747.36	29.21	7213.08	6484.89
CglAllCuts + CglRedSplit2	22.85	337.09	1015.16	30.31	7209.15	9438.53

showing that we can generate a large number of useful split cuts in reasonable CPU time.

Some conclusions can be drawn. Even by restricting our attention to the corner polyhedron associated with an optimal basis, we were able to contribute cutting planes closing an additional 5% of integrality gap, on top of all the other existing rank-1 split cut generators. Using reduced costs information to generate stronger cutting planes seems to be a promising idea in practice. Our experiments suggest that generating a large number of cuts at the root and managing them in an effective way could yield big improvements in the performance of a Branch-and-Cut algorithm on difficult instances.

Appendix A: Further experiments with rank-1 cuts: cut generation parameters and strategies

Here we provide more details and comments on the experiments discussed in Sect. 4.4.1. The setup for these experiments is as follows. We *always* test our cut generation algorithm on top of CglAllCuts, so that we can measure how much can be added with respect to existing cut generators. Therefore, the baseline is represented by the integrality gap closed by CglAllCuts. On `MILP_C`, on average CglAllCuts closes 28.72% of the integrality gap, generating 172 cuts (rounded to the nearest integer).

Appendix A.1: Row multipliers rejection parameters

In this section we investigate the role of the parameters σ, Λ and γ. We generate cutting planes from all possible combinations of μ, column selection strategy and row

Table 10 Effect of Λ, σ and γ on the performance of the cut generation algorithm: average values for the amount of integrality gap closed after 1 round, and number of cutting planes

Parameters	Gap (%)	# cuts
$\Lambda = 1000,\ \sigma = 0.001,\ \gamma = 0$	34.70	1018
$\Lambda = 20,\ \sigma = 0.001,\ \gamma = 0$	34.67	836
$\Lambda = 10,\ \sigma = 0.001,\ \gamma = 0$	34.61	721
$\Lambda = 5,\ \sigma = 0.001,\ \gamma = 0$	34.53	560
$\sigma = 0.001,\ \Lambda = 10,\ \gamma = 0$	34.61	721
$\sigma = 0.01,\ \Lambda = 10,\ \gamma = 0$	34.61	712
$\sigma = 0.05,\ \Lambda = 10,\ \gamma = 0$	34.53	683
$\sigma = 0.1,\ \Lambda = 10,\ \gamma = 0$	34.49	651
$\sigma = 0.5,\ \Lambda = 10,\ \gamma = 0$	34.02	497
$\gamma = 0,\ \sigma = 0.01,\ \Lambda = 10$	34.61	712
$\gamma = 0.01,\ \sigma = 0.01,\ \Lambda = 10$	34.68	802
$\gamma = 0.001,\ \sigma = 0.01,\ \Lambda = 10$	34.64	802
$\gamma = 0.0001,\ \sigma = 0.01,\ \Lambda = 10$	34.68	783

selection strategy as listed in Sect. 4.2, and evaluate the effect of using different values for σ, Λ and γ.

We start by setting $\sigma = 0.001$ and $\gamma = 0$; that is, we accept all vectors of row multipliers that yield even a slight reduction of $\|d_k\|$, and we do not penalize the norm of λ in the objective function of (9). Then we examine the effect of the parameter Λ. Results are reported in Table 10. First, we note that the integrality gap closed by setting $\Lambda = 1000$ and $\Lambda = 20$ is almost the same: that is, cuts associated with split disjunctions with very large norm close a very small amount of integrality gap in our experiments. This result is not surprising (cf. [8,16]), but is additional evidence to suggest that such cuts are the first that should be discarded. Next, we can see that decreasing Λ from 20 to 10 yields again a very small reduction in the amount of integrality gap closed, whereas the drop is larger when going from 10 to 5. However, it is clear from these numbers that almost all useful cutting planes arise from disjunctions with 1-norm not larger than 10, and in the majority of cases we only need $\Lambda = 5$. Out of the instances for which the drop in the integrality gap closed when going from $\Lambda = 20$ to $\Lambda = 5$ is not negligible, we report: `arki001_c` which drops from 43.90 to 40.89%, `flugpl_c` from 92.69 to 88.84%, `gen_c` from 79.29 to 76.05%. On the remaining instances, the difference is smaller than 2%.

Next, we fix $\Lambda = 10$, $\gamma = 0$ and measure the effect of σ on the cut generator. Recall that a vector of row multipliers λ is accepted only if $\| \sum_{i \in R_k} \lfloor \lambda_i^k \rceil d_i \| < (1 - \sigma) \|d_k\|$; thus, for larger σ we accept fewer cuts. From Table 10 we see that there is an effect by setting $\sigma > 0.01$; in particular, changing σ from 0.01 to 0.5 we lose almost 1% of integrality gap on average. On the other hand, we generate 200 fewer cuts. A setting of $\sigma = 0.01$ yields the same gap as $\sigma = 0.001$.

Finally, we analyze the effect of a nonzero γ, i.e. the penalization factor in (9) which is applied whenever the solution with $\gamma = 0$ does not satisfy $\|\lambda\|_1 \leq \Lambda$ (after rounding to the nearest integer). Table 10 shows that penalizing the norm of λ has

a very small (almost negligible) positive effect: the amount of integrality gap closed increases, on average, by a very small fraction. For the reasons discussed in Sect. 4.4.1, we expect this positive effect to be more significant when generating cuts using fewer values for μ, column selection strategy and row selection strategy.

Appendix A.2: Cut generation strategies

In this section we want to assess the effectiveness of our various cut generation heuristics. In particular, we are interested in evaluating the effect of the parameter μ (maximum number of rows that can are involved in each linear combination), of the column selection strategy, and of the row selection strategy. Our aim is to select only a few combinations, that hopefully generate most useful cuts from our cut generation methods, but in a much shorter computational time than using all possible combinations. The values of the parameters which are tested are listed in Sect. 4.2; whenever we write "all possible combinations", we refer to all combinations of the parameter values discussed in that section. For these experiments, we set $\Lambda = 10$, $\sigma = 0.01$, $\gamma = 0.0001$; Table 10 shows that we can close at most 34.68% of the integrality gap, when generating cuts from all possible combinations of parameters. We observe that adding our CglRedSplit2 cut generator (with given parameters) on top of CGLALLCUTS closes, on average, an additional 6% of the integrality gap. This corresponds to an improvement by 21% in relative terms.

We design two complementary experiments. In the first experiment we test CGLALLCUTS plus the cuts computed by all possible combinations of parameters for our cut generator, minus the combinations where a given parameter has a specified value. For instance, we generate cuts from all combinations *except* those involving $\mu = 50$. This way, we evaluate one aspect of the individual contribution of the cuts generated with $\mu = 50$, namely: what is the effect of the cuts that can only be obtained by setting $\mu = 50$? Results are reported in Table 11. It can be observed that $\mu = 3$ seems to generate more effective cutting planes that cannot be obtained with all the other values of μ combined. Among row selection strategy, RS1 through RS6 are obviously redundant: removing any one of those does not decrease the average closed gap. This was expected, as they are very similar. RS7 and RS8 were designed in a different way, and they seem to generate different cuts also in practice. Finally, from the table reporting results on column selection strategy, we observe that one possible set J_W can be removed while still attaining 34.68% of integrality gap closed: the set of columns that include the variables with large reduced cost in C-3P (cf. Sect. 3.1). Similarly, the set of variables with large reduced cost in C-5P can be excluded while losing very little (0.01% of integrality gap). This suggests that cutting planes that cut deeply on variables with large reduced costs are not effective in practice.

In the second experiment we test CGLALLCUTS plus the cuts computed only from those combinations of parameters where a given parameter has a specified value. For instance, we generate cuts from all parameter combinations where $\mu = 50$, and only those. This way, we evaluate another aspect of the individual contibution of the cuts generated with $\mu = 50$, namely: what is the effect of the cuts that can be obtained by setting $\mu = 50$, if they are used alone (on top of CGLALLCUTS)? Results are reported

Table 11 Results obtained by *excluding* all cut generators with a given parameter value; we report, for each excluded parameter, the integrality gap closed by the remaining cut generators and the number of instances where the gap closed decreases with respect to employing all generators

Number of rows μ												
$\mu = 3$:	34.36%	7										
$\mu = 5$:	34.64%	7										
$\mu = 10$:	34.66%	8										
$\mu = 15$:	34.61%	4										
$\mu = 20$:	34.66%	7										
$\mu = 50$:	34.62%	3										
Row selection strategy												
RS1:	34.67%	0										
RS2:	34.68%	0										
RS3:	34.68%	0										
RS4:	34.68%	0										
RS5:	34.68%	0										
RS6:	34.68%	0										
RS7:	34.65%	6										
RS8:	34.61%	5										
	Column selection strategy											
	Single set										Full part.	
C-3P:	34.67%	2	34.65%	4	34.68%	1					34.64%	6
C-5P:	34.59%	5	34.66%	4	34.65%	4	34.67%	2	34.67%	1	34.54%	11
I-2P-1/2:	34.60%	4	34.63%	5							34.56%	7
I-2P-2/3:	34.65%	7	34.63%	6							34.60%	9
I-2P-4/5:	34.67%	3	34.67%	4							34.66%	6
I-3P:	34.65%	3	34.65%	2	34.65%	6					34.59%	6
I-4P:	34.65%	4	34.65%	6	34.65%	6	34.65%	5			34.55%	11

For the column selection strategy, we report on each row the working sets of a different partition: for the contiguous partition, the sets are ordered from lowest reduced cost (left) to largest reduced cost (right). The final column reports results obtained excluding all working sets of the partition

in Table 12. We see that $\mu = 5$ seems to be a good tradeoff between closed gap and CPU time. Moreover, we have further evidence that RS7 and RS8 generate stronger cuts than the remaining row selection strategies. RS1 and RS4, which consider the nonzeroes on the integer nonbasic variables only, are weaker than the rest. The best strategies seem to be those that take into account the coefficient on both the integer and the continuous nonbasic variables, so that at the same time we try to reduce the cut coefficients on the continuous columns, and keep the coefficients on the integer columns under control. The greedy strategies RS4, RS5 and RS6 are only marginally slower than their non-greedy counterparts RS1, RS2 and RS3, but they are not more effective. Finally, this experiment confirms that cutting deeply on variables with large reduced costs is less effective than cutting deeply on variables with small reduced

Table 12 Results obtained by employing only the cut generators with a given parameter value; we report, for each parameter value, the integrality gap closed by all generators with that parameter value, and the corresponding CPU time in seconds

Number of rows μ		
$\mu = 3$:	33.82%	7.41
$\mu = 5$:	33.95%	7.92
$\mu = 10$:	33.95%	8.19
$\mu = 15$:	33.91%	8.44
$\mu = 20$:	33.27%	8.86
$\mu = 50$:	33.49%	11.02

Row selection strategy		
RS1:	34.06%	6.82
RS2:	34.07%	6.91
RS3:	34.06%	7.36
RS4:	34.04%	6.90
RS5:	34.06%	6.97
RS6:	34.07%	7.52
RS7:	34.31%	7.56
RS8:	34.32%	6.68

	Column selection strategy											
	Single set										Full part.	
C-3P:	31.60%	2.61	31.44%	3.39	29.56%	3.41					33.00%	7.41
C-5P:	30.82%	2.14	30.53%	2.29	31.52%	2.91	30.56%	2.92	29.44%	2.99	33.40%	9.15
I-2P-1/2:	31.79%	3.58	31.90%	3.61							32.91%	5.87
I-2P-2/3:	31.86%	3.98	31.84%	3.98							33.87%	6.52
I-2P-4/5:	31.70%	2.94	31.16%	3.00							33.54%	4.87
I-3P:	31.51%	4.24	30.83%	4.18	30.81%	4.10					33.17%	9.13
I-4P:	30.09%	3.90	30.64%	3.81	30.76%	3.82	31.45%	3.91			33.20%	10.34

For the column selection strategy, we report on each row the working sets of a different partition: for the contiguous partition, the sets are ordered from lowest reduced cost (left) to largest reduced cost (right). The final column reports results obtained by generating cuts employing all working sets of the partition

cost: by applying the reduction algorithm to a working set of columns J_W containing variables with very large reduced cost we lose 2% closed gap, with respect to choosing variables with small reduced cost in J_W (observe the decrease in the closed gap when moving from left to right on the rows corresponding to C-3P and C-5P in Table 12).

References

1. Achterberg, T., Koch, T., Martin, A.: MIPLIB 2003. Oper. Res. Lett. **34**(4), 361–372 (2006)
2. Ajtai, M.: The shortest vector problem in l_2 is NP-hard for randomized reductions. In: Proceedings of the 30th Annual ACM Symposium on Theory of Computing. Dallas, TX (1998)
3. Andersen, K., Cornuéjols, G., Li, Y.: Reduce-and-split cuts: improving the performance of mixed integer gomory cuts. Manag. Sci. **51**(11), 1720–1732 (2005)

4. Balas, E.: Intersection cuts—a new type of cutting planes for integer programming. Oper. Res. **19**(1), 19–39 (1971)
5. Balas, E.: Disjunctive programming. Annal. Discret. Math. **5**, 3–51 (1979)
6. Balas, E., Bonami, P.: Generating Lift-and-Project cuts from the LP simplex tableau: open source implementation and testing of new variants. Math. Program. Comput. **1**, 165–199 (2009)
7. Balas, E., Jeroslow, R.G.: Strengthening cuts for mixed integer programs. Eur. J. Oper. Res. **4**, 224–234 (1980)
8. Balas, E., Saxena, A.: Optimizing over the split closure. Math. Program. **113**(2), 219–240 (2008)
9. Bixby, R., Rothberg, E.: Progress in computational mixed integer programming—a look back from the other side of the tipping point. Annal. Oper. Res. **149**(1), 37–41 (2007)
10. Bixby, R.E., Ceria, S., McZeal, C.M., Savelsbergh, M.W.P.: An updated mixed integer programming library: MIPLIB 3.0. Optima **58**, 12–15 (1998)
11. Bonami, P., Cornuéjols, G., Dash, S., Fischetti, M., Lodi, A.: Projected Chvátal-Gomory cuts for mixed integer linear programs. Math. Program. **113**, 241–257 (2008)
12. COIN-OR Branch-and-Cut. https://projects.coin-or.org/Cbc. Accessed Oct 2010
13. COIN-OR Cut Generation Library. https://projects.coin-or.org/Cgl. Accessed Oct 2010
14. COIN-OR Linear Programming. https://projects.coin-or.org/Clp. Accessed Oct 2010
15. Cook, W., Kannan, R., Schrijver, A.: Chvátal closures for mixed integer programming problems. Math. Program. **47**, 155–174 (1990)
16. Cornuéjols, G., Liberti, L., Nannicini, G.: Improved strategies for branching on general disjunctions. Math. Program. A (2009). Published online
17. Dash, S., Günlük, O., Lodi, A.: MIR closures of polyhedral sets. Math. Program. **121**(1), 33–60 (2010)
18. Gomory, R.E.: An algorithm for the mixed-integer problem. Tech. Rep. RM-2597, RAND Corporation, (1960)
19. Lenstra, A.K., Lenstra, H.W. Jr., Lovász, L.: Factoring polynomials with rational coefficients. Mathematische Annalen **4**(261), 515–534 (1982)
20. Margot, F.: Testing cut generators for mixed-integer linear programming. Math. Program. Comput. **1**(1), 69–95 (2009)
21. Nemhauser, G.L., Wolsey, L.: A recursive procedure for generating all cuts for 0-1 mixed integer programs. Math. Program. **46**, 379–390 (1990)
22. Papadimitriou, C., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity. Dover, New York (1990)
23. Press, W., Teukolsky, S., Vetterling, W., Flannery, B.: Numerical Recipes in C, Second Edition. Cambridge University Press, Cambridge (1992, reprinted 1997)
24. Wolsey, L.: Integer Programming. Wiley, New York (1998)