Embedding Cuts in a Branch&Cut Framework: a Computational Study with $\{0, \frac{1}{2}\}$-Cuts

Giuseppe Andreello*, Alberto Caprara+, and Matteo Fischetti*

* *DEI, University of Padova, Italy*
+ *DEIS, University of Bologna, Italy*

Preliminary Draft, October 28, 2003

Abstract

Embedding cuts into a Branch&Cut framework is a delicate task, the main so when the implemented separation procedures are very successful and do produce a large set of violated cuts. In this case, it is of crucial importance to balance between the benefits deriving from a tighter (but larger) LP relaxation, and the overhead introduced for its solution.

In this paper we describe a separation heuristic for $\{0, \frac{1}{2}\}$-cuts, special cases of Chvátal-Gomory cuts which play an important role in combinatorial problems formulated as Integer Linear Programming (ILP) problems [4]. Our separation procedure is embedded within ILOG-Cplex 8.1, a widely-used commercial MIP solver.

Computational results on a large testbed of ILP instances including satisfiability, max-satisfiability, and linear ordering problems, are reported. On these problems, our first attempt of incorporating $\{0, \frac{1}{2}\}$-cuts within the ILOG-Cplex framework produced a code which was not significantly faster than the standard version, due to the excessive number of $\{0, \frac{1}{2}\}$-cuts generated—though these cuts appear to be of better quality with respect to those found by other general-purpose methods, and sometimes turn out to be facet defining for the underlying integer polytope. However, a more sophisticated cut-selection strategy produced a considerable speedup on our testbed. This is particularly interesting in that our separation procedure was used as black-box—all the improvements came from a more clever way to exploit the violated cuts. In other words, our cut selection policy does not need a tight interaction with the separation procedure. Therefore, we expect that a similar approach can be useful to deal with other families of cuts, in particular those for which finding a violated member is easy, but finding a violated member with specific properties (maximum violation, maximum depth, etc.) can be difficult.

1 Introduction

Embedding cuts into a Branch&Cut framework is a delicate task. Paradoxically, a difficult situation arises when the implemented separation procedures are very successful and do produce a large set of violated cuts. In this case, it is of crucial importance to balance between the benefits deriving from a tighter (but larger) LP relaxation, and the overhead introduced for its solution. As a matter of fact, it may well be the case that embedding the new cuts in a naive way does reduce the number of branching nodes, but the overall computing time required increases in a significant way. This (often frustrating) situation is well known to the designers of Branch&Cut methods, who often avoid at all the generation of the new cuts, or allow for cut generation only at the root node of the branching tree according to the so-called Cut&Branch strategy.

In this paper we describe a separation heuristic for the family of $\{0, \frac{1}{2}\}$-cuts, a subset of Chvátal-Gomory cuts playing an important role in combinatorial problems formulated as Integer Linear Programming (ILP) problems [4]. Our separation procedure is embedded within ILOG-Cplex 8.1, a widely-used and very effective commercial MIP solver. Computational results on a large testbed of ILP instances including satisfiability, max-satisfiability, and linear ordering problems, are reported. On these problems, our first attempt of incorporating $\{0, \frac{1}{2}\}$-cuts within the ILOG-Cplex framework produced a code which was not significantly faster than the standard version. This was somehow unexpected, in that $\{0, \frac{1}{2}\}$-cuts appear to be of better quality than those generated with other general-purpose methods, and sometimes turn out to be even facet defining for the underlying integer polytope. However, a more sophisticated cut-selection strategy produced a considerable speedup on our testbed. This is particularly interesting in that our separation procedure was used as black-box—all the improvements came from a more clever way to exploit the violated cuts. In other words, our cut selection policy does not need a tight interaction with the separation procedure. Therefore, we expect that a similar approach can be useful to deal with other classes of cuts, in particular those for which finding a violated member is (relatively) easy, but finding a violated member with specific properties (maximum violation, maximum depth, etc.) can be difficult—as, e.g., for the classical Gomory cuts, or for the cuts based on the theory of corner polyhedra and T-spaces that have been recently proposed by [10].

The paper is organized as follows. In Section 1 we review the theory of $\{0, \frac{1}{2}\}$-cuts, whereas in Section 2 we present our heuristic separation procedure. Section 3 describes the ILOG-Cplex 8.1 Branch&Cut framework where the separation procedure is embedded, and the main call-back functions we designed for handling our own implementation of the cut pool. In the same section we introduce the cut quality measures we used for cut selection. Section 4 reports our computational experience and compares various cut-selection strategies on a large testbed containing ILP instances with a strong combinatorial structure (taken from satisfiability, max-satisfiability, and linear ordering problems). Finally, some conclusions are drawn in Section 5.

2 The theory of $\{0, \frac{1}{2}\}$-cuts

In this section we briefly illustrate the theory of $\{0, \frac{1}{2}\}$-cuts, which is needed to describe the separation method used in our code. We follow the presentation in [4], to which the reader is referred for further details.

For any given $z \in Z$ and $q \in Z_+$, let $z \bmod q := z - \lfloor z/q \rfloor q$. As customary, notation $a \equiv b \pmod{q}$ stands for $a \bmod q = b \bmod q$. For any integer matrix $Q = (q_{ij})$, let $\overline{Q} = (\overline{q}_{ij}) := Q \bmod 2$ denote the *binary support* of Q, i.e., $\overline{q}_{ij} = 1$ if q_{ij} is odd; $\overline{q}_{ij} = 0$ otherwise. Moreover, given an undirected multigraph $G = (V, E)$, let $\delta(j)$ denote the set of edges incident with a node $j \in V$.

Given an $m \times n$ integer matrix $A = (a_{ij})$ and an m-dimensional integer vector b, let $P := \{x \in R^n : Ax \leq b\}$, $P_I := \text{conv}\{x \in Z^n : Ax \leq b\}$, and assume $P_I \neq P$. We assume without loss of generality that each row of (A, b) contains relatively prime entries.

A *Chvátal-Gomory* (*CG*) *cut* [9, 6] is a valid inequality for P_I of the form $\lambda^T Ax \leq \lfloor \lambda^T b \rfloor$, where $\lambda \in R^m_+$ is such that $\lambda^T A \in Z^n$, and $\lfloor \cdot \rfloor$ denotes lower integer part. It is known that undominated CG cuts only arise for rational $\lambda \in [0, 1)^m$.

CG cuts can equivalently be obtained in the following way [5]. Let $\mu \in Z^m_+$ and $q \in Z_+$ be such that $\mu^T A \equiv 0 \pmod{q}$ and $\mu^T b = kq + r$ with $k \in Z$ and $r \in \{1, \ldots, q-1\}$. Then the *mod-q cut* $\mu^T Ax \leq kq$ is a valid inequality for P_I. This inequality can be written as $\mu^T (b - Ax) \geq r$, hence a given $x^* \in P$ violates $\mu^T Ax \leq kq$ if and only if $\mu^T (b - Ax^*) < r$.

In [4], the important family of $\{0, \frac{1}{2}\}$*-cuts* is investigated, corresponding to a CG cut with $\lambda \in \{0, \frac{1}{2}\}^m$ (or, equivalently, to mod-2 cuts with $\mu := 2\lambda \in \{0, 1\}^m$).

The separation problem for $\{0, \frac{1}{2}\}$-cuts, in its optimization version, can be stated as follows.

$\{0, \frac{1}{2}\}$**-SEP:** *Given* $x^* \in P$, *compute* $s^* := b - Ax^* \geq 0$ *and solve*

$$z^*_{SEP} := \min\{s^{*T} \mu : \mu \in \mathcal{F}(\overline{A}, \overline{b})\} \text{ , } \textit{where}$$

$$\mathcal{F}(\overline{A}, \overline{b}) := \{\mu \in \{0, 1\}^m : \overline{b}^T \mu \equiv 1 \pmod{2}, \overline{A}^T \mu \equiv 0 \pmod{2}\}.$$

By construction, there exists a $\{0, \frac{1}{2}\}$-cut violated by the given point x^* if and only if $z^*_{SEP} < 1$. In this case, setting $\lambda := \mu/2$ produces a $\{0, \frac{1}{2}\}$-cut with violation $(1 - z^*_{SEP})/2$.

$\{0, \frac{1}{2}\}$-SEP is NP-complete in the general case [4], but there are important special cases for which a polynomial separation algorithm can be derived. Among these cases, one that appears to have a main impact on several classes of combinatorial optimization problems arises when A has at most 2 odd entries for each row. Though this is almost never the case in practice, in [4] it is shown how to relax the original constraints so as to achieve this property. The resulting procedure is a general separation heuristic for $\{0, \frac{1}{2}\}$-cuts, that we outline next.

Let $M := \{1, \ldots, m\}$ and $N := \{1, \ldots, n\}$ be the indices of the rows and columns of A, respectively, and let

$$O_i := \{j \in N : \overline{a}_{ij} = 1\}, \quad \text{for all } i \in M.$$

The following result, proved in [4], is the key to our separation method, hence we also give a short proof of constructive type.

Theorem 1 *$\{0, \frac{1}{2}\}$-SEP can be solved in polynomial time if $|O_i| \leq 2$ for all $i \in I$.*

Proof. Let $G = (V, E)$ be an undirected multigraph having a node j for each column j of $\overline{A}$, plus a special node q. Graph G is weighted and labelled on its edges. There is an edge e_i for each row $i \in M$, with *weight* $w(e_i) := s_i^*$ and labelled *odd* if $\overline{b}_i = 1$, *even* otherwise; this edge connects the two nodes h and k such that $O_i = \{h, k\}$ (if $O_i = \{h\}$, then let the edge connect node h to the special node q). A given $E^* \subseteq E$ is a *Eulerian cycle* if each component of the subgraph induced by E^* is Eulerian, i.e., $|E^* \cap \delta(j)|$ is even for all j. E^* is an *odd Eulerian cycle* if, in addition, it contains an odd number of odd edges. It is easy to see that there is a one-to-one correspondence between the 0-1 vectors $\mu \in \mathcal{F}(\overline{A}, \overline{b})$ and the odd Eulerian cycles E^* of G, given by $\mu_i = 1$ if and only if $e_i \in E^*$. This correspondence guarantees that $w(E^*) := \sum_{e \in E^*} w(e)$ equals $s^{*T}\mu$.

The above discussion shows that the optimization version of $\{0, \frac{1}{2}\}$-SEP is equivalent to finding a minimum-weight odd Eulerian cycle of G, say E^*. Because of the well-known theorem of Euler, E^* can be partitioned into a collection of simple cycles. Since E^* is odd, at least one such cycle, say C^*, must be odd, where $w(C^*) \leq w(E^*)$ holds as $w(e) \geq 0$ for all $e \in E$.

A polynomial algorithm for the minimum-weight odd cycle problem is known from folklore (see, e.g., [11]). This algorithm is based on shortest path computations on an auxiliary graph, and runs in $O(n^3)$ time. $\square$

As already mentioned, the separation algorithm outlined in the proof of the previous theorem can be applied to an arbitrary ILP, provided that the original system $Ax \leq b$ is relaxed into $A'x \leq b'$ (say), with the property that A' has at most two odd entries per row—and therefore satisfies the conditions in Theorem 1. For example, in [4] the case where bound constraints of the type $l \leq x \leq d$ are part of the system $Ax \leq b$ is considered (possibly $l_j = -\infty$ and/or $d_j = +\infty$ for some j). Then a suitable relaxation $A'x \leq b'$ can readily be obtained from $Ax \leq b$ by replacing each inequality $\sum_j a_{ij}x_j \leq b_i$ with $|O_i| \geq 3$, by its so-called *LU-weakenings*

$$a_{ih}x_h + a_{ik}x_k + \sum_{j \notin O_i} a_{ij}x_j + \sum_{j \in L}(a_{ij} - 1)x_j + \sum_{j \in U}(a_{ij} + 1)x_j \leq b_i + \sum_{j \in U} d_j - \sum_{j \in L} l_j \tag{1}$$

for all $h, k \in O_i$, $h < k$, and for all partitions (L, U) of $O_i \setminus \{h, k\}$. These inequalities are obviously valid for P, in that they are obtained by adding together $\sum_j a_{ij}x_j \leq b_i$ and the bound constraints $-x_j \leq -l_j$ ($j \in L$) and $x_j \leq d_j$ ($j \in U$).

Although $A'x \leq b'$ has, in general, an exponential number of rows, still the corresponding $\{0, \frac{1}{2}\}$-SEP can be solved in polynomial time. Indeed, for each triple (i, h, k) only two LU-weakenings are worth considering for the given point x^* to be separated, namely those with even and odd right-hand side having minimum slack. These two weakenings can be computed, in $O(|O_i|)$ time, through a simple dynamic programming scheme (analogous to the one outlined in the next section) that considers, for each $j \in O_i \setminus \{h, k\}$, the two possibilities $j \in L$ and $j \in U$.

3 The $\{0, \frac{1}{2}\}$–SEP heuristic

In this section we describe our implementation of the $\{0, \frac{1}{2}\}$–SEP heuristic outlined in the previous section, whose running time is $O(\sum_{i \in M} |O_i|^3 + n^3) = O(mn^3)$.

Let $x^* \in P$ denote the fractional point to be separated, and let $s^* := b - Ax^* \geq 0$ be the corresponding slack vector. If successful, the separation heuristic produces a subset I^* of M whose characteristic vector μ produces a violated $\{0, \frac{1}{2}\}$-cut with respect to x^*.

First of all, following [4] we apply the following preprocessing steps in the attempt to reduce the size of the separation problem.

- R1. We scale the coefficients of all rows of (A, b) to relatively prime integers.
- R2. Every row i of $Ax \leq b$ with $s^*_i \geq 1$ is removed, in that the choice $\mu_i = 1$ implies $s^{*T}\mu \geq 1$, hence it cannot lead to a violated $\{0, \frac{1}{2}\}$-cut.
- R3. For $j = 1, \ldots, n$, we consider the (possibly empty) set

$$S_j := \{i \in M : \overline{a}_{ik} = 1 \text{ iff } k = j\} \tag{2}$$

These rows include variable lower/upper bound constraints of the form $-x_j \leq l_j$ or $x_j \leq d_j$, if present. If there is a row $i \in S_j$ with $s^*_i = 0$ (i.e., if the corresponding constraint is tight for x^*), we can easily get rid off of variable x_j. Indeed, assuming for notational convenience $i = m$ and $j = n$, the input $(\overline{A}, \overline{b}, s^*)$ for $\{0, \frac{1}{2}\}$-SEP has the form:

$$\overline{A} = \left[\begin{array}{c|c} M & d \\ 0 \cdots 0 & 1 \end{array}\right], \overline{b} = \begin{bmatrix} \beta \\ \overline{b}_m \end{bmatrix}, \text{ and } s^* = \begin{bmatrix} \sigma^* \\ 0 \end{bmatrix}.$$

Observe that any feasible solution $\mu \in \{0,1\}^m$ of $\{0, \frac{1}{2}\}$-SEP has $\mu_m \equiv \sum_{i=1}^{m-1} \mu_i d_i \pmod 2$. We then define a reduced instance of $\{0, \frac{1}{2}\}$-SEP, whose input is given by (M, f, σ^*), where $f := \beta$ if $\overline{b}_m = 0$, $f := (\beta + d) \bmod 2$ otherwise. It is easy to verify that there is a one-to-one correspondence between the feasible solutions $\mu \in \{0,1\}^m$ and $\nu \in \{0,1\}^{m-1}$ to the original and reduced $\{0, \frac{1}{2}\}$-SEP, respectively, where $\mu_k = \nu_k$ for $k = 1, \ldots, m-1$, and $\mu_m \equiv d^T\nu \pmod 2$.

In particular, the above reduction applies to the case where $x^*_j = 0$ and $x_j \geq 0$ is part of the inequality system (in which case x_j is simply removed along with the corresponding column of $\overline{A}$), or $x^*_j = 1$ and $x_j \leq 1$ (in which case x_j is first complemented and then removed along with the corresponding column of $\overline{A}$). This typically allows for a quite substantial size reduction of the separation instance.

Reduction R3 is iterated until no further variable can be removed. Then we construct the separation multigraph $G = (V, E)$ as in the proof of Theorem 1. Recall that V contains a node j for each variable x_j, plus a special node q used to deal with the constraints with just one odd coefficient. Clearly, for each pair of nodes h, k in G only the odd and even edges with minimum weight have to be stored. Let $odd(h,k)$ and $even(h,k)$ denote these minimum weights. For each odd and even edge between (h,k) of G, we also store the index i of the constraint in the original system $Ax \leq b$ that produced weight $odd(h,k)$ and $even(h,k)$, respectively, so as to be able to reconstruct the output set I^*.

Initially, we set $odd(h,k) := +\infty$ and $even(h,k) := +\infty$ for $h, k \in V \setminus \{q\}$. For each $j \in V$ we then compute

$$\delta^p_j := \min\{s^*_i : i \in S_j, \overline{b}_i = p\} \text{ for } p = 0, 1$$

($\delta^p_j = +\infty$ if no suitable i exists), and we set $odd(j,q) := \delta^1_j$ and $even(j,q) := \delta^0_j$.

Afterwards, for each $i \in M$ with $|O_i| = 2$, say $O_i = \{h,k\}$, and $\overline{b}_i = 1$ (resp., $\overline{b}_i = 0$), we set $odd(h,k) := \min\{odd(h,k), s^*_i\}$ (resp., $even(h,k) := \min\{even(h,k), s^*_i\}$).

Finally, we address the "complicated" constraints i with $|O_i| \geq 3$, and consider each of the $O(|O^2_i|)$ pairs $h, k \in O_i$. For each triple (i, h, k), we update $odd(h,k) := min\{odd(h,k), best^1\}$ and $even(h,k) := min\{even(h,k), best^0\}$, where values $best^1$ and $best^0$ take into account the minimum-slack odd and even (respectively) combination between the i-th row and the rows in S_j for all $j \in O_i \setminus \{h,k\}$. They are computed, by dynamic programming, by initializing $best^0 := best^1 := s^*_i$ and then by computing, for each $j \in O_i \setminus \{h,k\}$ (in any sequence)

$$old\_best^p := best^p \qquad \text{for } p = 0, 1$$

$$best^p := \min\{old\_best^p + \delta_j^0,\ old\_best^{1-p} + \delta_j^1\} \qquad \text{for } p = 0, 1$$

(of course, this computation can be be aborted as soon as $\min\{best^0, best^1\} \geq 1$).

After having defined the weights of the edges in G, we find a minimum-weight odd cycle of G in a standard way (see, e.g., [11]). To be specific, we define the auxiliary graph $H = (V_1 \cup V_2, F)$ that contains two vertices $h_1 \in V_1, h_2 \in V_2$ for each node $h \in V$. For each even edge in G joining nodes h, k, having weight $even(h, k)$, H contains edges h_1, k_1 and h_2, k_2 of weight $even(h, k)$. Moreover, for each odd edge in G joining nodes h, k, having weight $odd(h, k)$, H contains edges h_1, k_2 and h_2, k_1 of weight $odd(h, k)$. The shortest odd cycle visiting each node h in G corresponds to the shortest path joining h_1 to h_2 in H.

In our implementation, for each node $h \in V$, we compute by Dijkstra's algorithm the shortest path arborescence rooted in h_1 for graph H (representing the shortest paths from h_1 to each other node). By symmetry, this also provides the shortest path anti-arborescence rooted in h_2 (representing the shortest paths from each other node to h_2). With this information, we consider each node $k \in V \setminus \{h\}$ along with the shortest paths from h_1 to k_1 and from k_1 to h_2 in H. As mentioned above, concatenation of these two paths yields a shortest path in H from h_1 to h_2 visiting k_1, that is easily checked to correspond to the shortest odd cycle in G visiting nodes h, k.

If this shortest odd cycle C^* has weight $z(C^*) < 1$, then the $\{0, \frac{1}{2}\}$-cut obtained by the combination of the *weakened* inequalities corresponding to the edges in the cycle, is certainly violated by x^*. Even in case $z(C^*)$ is (slightly) larger than 1, however, there is the hope that a $\{0, \frac{1}{2}\}$-cut obtained by combining the inequalities of the *original* system be violated. Indeed, let I^* denote the inequalities in the original system corresponding to the edges of the minimum-weight odd cycle, and observe that $\sum_{i \in I^*} s_i^* \leq z(C^*)$. However, in order to get a valid $\{0, \frac{1}{2}\}$-cut we still need to perform a cut post-processing. Indeed, the determination of I^* considered a weakened version of the original inequalities, hence summing together only the inequalities in I^* (without the inequalities in S_j used in the edge-weight definition) would produce an inequality $\beta x \leq \beta_0$ (say) for which some of the left-hand side coefficients may be odd. In other words, setting $\mu_i := 1$ only for $i \in I^*$ does not necessarily yield $\mu \in \mathcal{F}(\overline{A}, \overline{b})$. In order to have a valid (and hopefully violated) $\{0, \frac{1}{2}\}$-cut, we therefore apply the same dynamic programming method outlined above, modified so as to find the minimum-slack weakening of $\beta x \leq \beta_0$ with *all* even left-hand side coefficients, and odd right-hand side. It is easy to see that the violation of the resulting inequality is at least as large as the one corresponding to the weakened system, i.e., not smaller than $(1 - z(C^*))/2$.

For problems with a strong combinatorial structure such as those in our testbed (see Section 5), the cuts found by this heuristic tend to be of "good" quality, in the sense that they are generally sparse and with relatively small coefficients, but still are violated by a relatively large amount. In some cases, we could verify that the cuts produced were facet defining for the underlying integer polyhedron P_I.

4 The overall Branch&Cut algorithm

Our work is aimed at analyzing the benefits deriving from the use of $\{0, \frac{1}{2}\}$-cuts within an existing Branch&Cut algorithm, without modifying other important features of the algorithm such as primal heuristics, branching variable selection, next node selection, etc. ILOG-Cplex 8.1 [7] was selected as the external framework to work with. ILOG-Cplex is a widely-used Branch&Cut MIP software; since its release 7.0, this code allows one to customize the main features of the algorithm, including cut generation. To this end, a *cut callback* shell was implemented so as to embed our separator for $\{0, \frac{1}{2}\}$-cuts within the existing Branch&Cut algorithm. As explained below, the shell also handles an internal cut pool, whose interaction with the current LP is governed by a clever cut selection procedure—as shown in the computational section, cut handling is of crucial importance.

All procedures we used in our study have been written in C++ (except the $\{0, \frac{1}{2}\}$-cut separator, which is written in C) and are available, on request, from the authors.

4.1 Cut quality measures

On problems with a strong combinatorial structure, our $\{0, \frac{1}{2}\}$-separator may add to the cut pool a huge list of violated cuts. Adding all these cuts to the original formulation is therefore likely to increase the LP size in an excessive way, leading to higher and higher node solution times. So, it is essential to select in a very careful way the pool cuts to be added to the formulation. Our selection rules are based on the following cut quality measures, analogous to those proposed in [1].

Given a cut $\alpha x \leq \alpha_0$ and a point x^*, the Euclidean distance between x^* and the hyperplane induced by $\alpha x = \alpha_0$ can be computed as

$$\text{dist}(x^*, \alpha, \alpha_0) := \frac{|\alpha^T x^* - \alpha_0|}{\|\alpha\|} \tag{3}$$

Following geometrical intuition, it is quite natural to consider this distance as a first-order measure of the expected efficacy of a cut $\alpha x \leq \alpha_0$ violated by the fractional point x^*, the larger the distance the better the cut [1].

One could argue that this measure has some drawbacks. In particular, assume that we have $\alpha_j > 0$ for a certain variable with $x_j^* = 0$. Clearly, the numerator of (3) is not affected by component α_j, hence the presence of the norm of α at the denominator implies that setting $\alpha_j = 0$ would produce a violated (and valid) cut with increased depth, whereas this inequality is mathematically dominated by the original one. We implemented and tested some variants of (3), but we could not find a version producing consistently better results: in spite of its drawbacks, the Euclidean distance seems to be an easily-computable and quite reliable measure of the cut efficacy. Therefore, our first quality measure is the *cut efficacy* computed as:

$$\text{eff}(x^*, \alpha, \alpha_0) := \frac{\alpha^T x^* - \alpha_0}{\|\alpha\|} \tag{4}$$

Notice that, with this definition, efficacy is positive for violated cuts. As a heuristic rule, only the pool cuts with

$$\text{eff}(x^*, \alpha, \alpha_0) \geq min\_eff \tag{5}$$

qualify as candidates to be inserted into the LP formulation, where min_eff is a threshold defined in an adaptive way (as explained in the sequel).

Once the pool has been ordered by non-increasing efficacy, we need to face the issue of preventing "similar" cuts from being added together to the LP. We consider two cuts $\alpha x \leq \alpha_0$ and $\beta x \leq \beta_0$ to be similar if they have (almost) the same efficacy and their defining hyperplanes are (almost) parallel, the maximum dissimilarity arising when the two hyperplanes are orthogonal, i.e., when $\alpha^T \beta = 0$. Accordingly, we choose a minimum threshold $max\_par \in [0, 1)$ and require that all cuts to be added to the LP formulation verify (pairwise) the *maximum parallelism condition*:

$$\text{par}(\alpha, \beta) := \frac{|\alpha^T \beta|}{\|\alpha\| \, \|\beta\|} \leq max\_par \tag{6}$$

In particular, the choice $max\_par = 0$ forces the cuts to be orthogonal, i.e. with disjoint support. Lift-and-project cuts in [1] are selected in the same way, but using $max\_par = 0.999$ in condition (6) so as to just avoid adding duplicate cuts; no computational results for other values of max_par are reported.

4.2 The ILOG-Cplex 8.1 framework

Implanting a piece of software within a sophisticated code such as ILOG-Cplex requires some knowledge of the main strategies therein used, so as to avoid "rejection" of the host code with respect to the guest one.

Looking at ILOG-Cplex 8.1 documentation (and trying to reconstruct its strategies from the log files), we found that this code has been designed to be somehow "conservative" with respect to the addition of new cuts.

First of all, ILOG-Cplex does not implement a cut pool: once a (globally valid) cut has been generated at run time, it is added statically to the LP formulation and never removed. An important consequence is that only a few, strong cuts can be added during the whole run. An important exception (not exploited in our code) arises for locally-valid cuts, which are automatically removed from the formulation the first time they are no longer guaranteed to be valid (because of a backtracking step); removed cuts are not saved.

Second, cuts are not generated at each branching node. We inferred from the ILOG-Cplex log files that an intense attempt of generating new cuts is only made at the root node, and then a moderate separation effort is spent after each backtracking step. This choice is motivated by the overall ILOG-Cplex tree exploration strategy, a sort of "cut and dive" scheme where diving continues until both the current node and its brother are fathomed, and new cuts are generated only after a backtracking step.

This strategy was designed and tuned by ILOG-Cplex 's designers, and proved successful on a vast collection of hard MIP instances. Therefore we had no reason to try to alter it; rather, we followed ILOG-Cplex 's vision and implemented the cut callback procedure described in Algorithm 1, where LP denotes the set of constraints of the current LP relaxation.

Some comments follow.

- At the root node the separation procedure is called only a limited number of times (5, in our implementation). In addition, as in [1] we found that better overall results are typically obtained if separation is only invoked at each k-th backtracking (we choose $k = 4$ in our implementation).

- As already mentioned, ILOG-Cplex does not implement an internal cut pool. (Actually, ILOG-Cplex does allow some constraints of the initial formulation to be stored in a cut pool rather than in the LP, but this pool cannot be extended at run time, as required in our application.) Hence, we implemented our own POOL data-structure to store (in a C-like manner) the $\{0, \frac{1}{2}\}$-cuts.

- A possible issue in handling the interaction between the current LP and our internal pool is the impossibility of removing a (globally valid) cut from the current LP. This makes it impossible to purge the current LP by removing some of its constraints. A possible work-around is to define the $\{0, \frac{1}{2}\}$-cuts as local (as opposed to global) cuts, so that they are automatically removed from time to time from the LP. Our C++ classes do support this feature, that works reasonably well in practice. However, in the present computational testing we decided not to activate any LP purging procedure, and followed the ILOG-Cplex strategy of adding cuts statically to the LP. Of course, this implies the use of conservative cut-selection criteria so as not to overload the LP.

- If the pool contains more than *max_pool* cuts, after it has been ordered by efficacy, the least efficacious cuts are removed. In our experience, sorting the whole pool typically requires a negligible fraction of the separation time; in case the pool contained a huge number of cuts, a more efficient priority queue could be preferable.

- The maximum parallelism constraint (6), controlled by parameter *max_par*, is applied only with respect to cuts added in the current node, and not to the whole set of cuts generated during the optimization. This ensures that new and efficacious cuts can be added even if they are parallel to old cuts.

- The minimum efficacy in (5) is not fixed *a priori*. At the first successful cut generation, this value is set to the minimum between ub_min_eff, which is a parameter of our algorithm, and 70% of the efficacy of the best cut generated. During the run, we count the number of times the pool does not contain efficacious cuts, and every 20 failures the bound is reduced. This compensates for the fact that the tighter the LP relaxation, the harder to build deep cuts.

- The parameter cut_factor imposes an upper bound on the number of cuts that we add to the LP during the whole optimization. In particular, the number of cuts added to the LP can never exceed cut_factor times the number of constraints in the initial LP formulation.

- Our $\{0, \frac{1}{2}\}$-cut separation heuristic works on the current LP formulation, i.e., on the initial LP amended by the cuts added in the previous iterations. We found that this approach produces typically better results than the one of deriving only rank-1 cuts based on the initial LP formulation. For testing purposes, however, we introduced a flag, called *recomb*, to instruct the separator to generate $\{0, \frac{1}{2}\}$-cuts by combining constraints of the initial formulation ($recomb = off$) or of the current one ($recomb = on$).

Algorithm 1 Separation strategy

```
1: if this is the first attempt to generate cuts then
2:     /* define some global counters */
3:     $added\_cuts := 0$
4:     $max\_added\_cuts := cut\_factor *$ (number of LP rows)
5: end if
6: if (the number of backtrackings is not a multiple of 4) or ($added\_cuts \geq max\_added\_cuts$) then
7:     return
8: end if
9: add to POOL all the $\{0, \frac{1}{2}\}$-cuts violated by $x^*$ found by the heuristic separation procedure
10: sort POOL by decreasing efficacy, POOL[0] containing the cut with largest efficacy
11: if $|POOL| > max\_pool$ then
12:     remove the last $|POOL| - max\_pool$ cuts from $POOL$
13: end if
14: if this is the first attempt to generate cuts then
15:     /* define the global variable $min\_eff$ */
16:     $min\_eff := \min\{ub\_min\_eff,\ 0.7 * \text{eff}(POOL[0])\}$
17: end if
18: $miss := 0$
19: if $\text{eff}(POOL[0]) < min\_eff$ then
20:     $miss := miss + 1$
21:     if miss = 20 then
22:         $miss := 0$
23:         $min\_eff := min\_eff - 0.03$
24:     end if
25:     return
26: end if
27: $S := \emptyset$
28: $i := 0$
29: while ($i \leq |POOL|$) and ($\text{eff}(POOL[i]) \geq min\_eff$) do
30:     if $\text{par}(c, POOL[i]) \leq max\_par$ for all cuts $c \in S$ then
31:         add $POOL[i]$ to the current LP
32:         $S := S \cup \{POOL[i]\}$
33:         $added\_cuts := added\_cuts + 1$
34:         if $added\_cuts \geq max\_added\_cuts$ then
35:             return
36:         end if
37:     end if
38:     $i := i + 1$
39: end while
40: return
```

5 Computational experiments

Like for many other classes of widely-used MIP cuts (notably, cover and clique inequalities), there is no guarantee that the family of $\{0, \frac{1}{2}\}$-cuts contains a member that cuts any given fractional extreme point of any ILP. In addition, due to its heuristic nature, our separation algorithm can fail to generate a violated $\{0, \frac{1}{2}\}$-cut even if one exists. So, we cannot claim $\{0, \frac{1}{2}\}$-cuts are useful for the solution of *any* ILP, but we have to content ourselves to find classes of ILP's which can benefit from the use of our $\{0, \frac{1}{2}\}$-cut separator. As a matter of fact, as our computational experiments show, there are important classes of combinatorial ILP's for which the usual cover and clique separation procedures are completely ineffective, whereas $\{0, \frac{1}{2}\}$-cuts are quite useful and allow for a considerable speedup (if dealt with in a proper way).

5.1 The testbed

We considered the following classes of combinatorial problems, which produce notoriously hard-to-solve ILP's. All instances are available, on request, from the authors.

5.1.1 Satisfiability Problems (SAT)

A boolean variable x is a variable whose feasible values are *true* and *false* (usually represented with numbers 1 and 0, respectively). A boolean formula is a combination of boolean variables using the logical connectives *not* ($\bar{x}$), *or* ($\vee$), *and* ($\wedge$). A *literal* is a variable or a negation of a variable, and a disjunction of literals is a *clause*. Given a set of clauses $C_1, C_2, \ldots, C_m$ on the boolean variables $x_1, x_2, \ldots, x_n$, the SAT problem is to find an assignment of values to the variables that satisfies the formula $C_1 \wedge C_2 \wedge \ldots \wedge C_m$.

A SAT instance is easily formulated as an ILP with a binary variable for each of the boolean variables of the SAT, while the clauses C_k are converted to linear constraints of the form $\sum_i x_i + \sum_j (1 - x_j) \geq 1$. The objective function is not specified, and a constant value is usually taken. However, we found useful to generate a random objective function given by the sum of a random subset of the left hand sides of the constraints (each constraint is in the subset with probability 0.5). Of course, the optimization is stopped as soon as an integer solution is found.

The SAT instances we considered were downloaded from the web page of the second DIMACS Challenge [8]; see [12] for more information.

5.1.2 Maximum Satisfiability Problems (MAXSAT)

The MAXSAT problem is only slightly different from SAT. Given a set of clauses, MAXSAT asks calls for an assignment of values to the variables that maximizes the number of satisfied clauses.

An ILP formulation of MAXSAT is simply obtained from the SAT one: for each clause C_k, we introduce a slack binary variable z_k and replace the right-hand side value 1 by z_k in the corresponding constraint. The objective function is then $\max \sum_k z_k$.

We solved the MAXSAT version of every SAT instance considered.

5.1.3 Linear Ordering Problems (LINORD)

The linear ordering problem is defined as follows. We are given a set of n items $\{1, \ldots, n\}$ and a cost matrix $c_{ij}, i, j \in \{1, \ldots, n\}$. Each entry c_{ij} represents the cost of placing item i before item j in a permutation of the n items, the goal being to find a minimum-cost such permutation.

The variables of a standard ILP formulation are x_{ij} for $1 \leq i < j \leq n$, where $x_{ij} = 0$ if i precedes j in the permutation, $x_{ij} = 0$ otherwise. The objective function then reads

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (c_{ij} x_{ij} + c_{ji}(1 - x_{ij}))$$

whereas the consistency of a binary solution x is enforced by the following *triangle inequalities* (all of which are put explicitly in the model):

$$\begin{cases} x_{ij} + x_{jk} - x_{ik} \leq 1 \\ -x_{ij} - x_{jk} + x_{ik} \leq 0 \end{cases} \quad 1 \leq i < j < k \leq n$$

We found that the instances from the public LOLIB library tend to have a negligible integrality gap at the root node, hence they can be solved quite easily. For other known classes of large-size instances, instead, the main difficulty is the solution of the initial LP, hence our cuts cannot help their solution. The instances we considered in our testbed were therefore generated according to the following random procedure suggested by G. Reinelt [13], which is intended to produce medium-size instances with a significant integrality gap at the root node of the initial formulation. We considered case $n = 30$ and defined each entry c_{ij} to be either 0 (with probability *prob*), or a random integer number in range $[0, C_{max}]$ (with probability $1 - prob$). Our instances where then obtained by generating about 10 instances for each of the following combinations of parameters $(prob, C_{max})$: $(0.5, 4)$, $(0.5, 10)$, $(0.2, 10)$, $(0.3, 10)$, $(0.1, 10)$, $(0.05, 10)$, $(0.5, 1e6)$. As already mentioned, for benchmarking purposes our LINORD instances are available, on request, from the authors.

5.2 Speedup evaluation

Our algorithm was tested and compared to a reference ILOG-Cplex configuration, proceeding in the same way as ILOG does in order to validate its own new releases, as described in [3].

In each table presented in the sequel, we compare two different codes. Comparison is made either on all instances in the testbed or on a subset of them. Depending on the results of the two codes on the instances considered, these are subdivided in the following disjoint sets:

1. *easy* instances, solved by both codes in less than 5 CPU seconds on a AMD Athlon XP 2,400+;

2. *regular* instances, solved by both codes in less than 1,800 CPU seconds on the same PC (excluding those in the previous set);

3. *hard* instances, solved by one code in less than 1,800 CPU seconds, but not by the other;

4. *impossible* instances, solved by neither code in less than 1,800 CPU seconds.

In the tables we report the total number of instances ("count"), the number of instances of each class, and the number of (hard) instances solved by one code and not by the other.

As in [3], the relative code effectiveness is evaluated, on the regular instances, as the *geometric* mean of the speedups, where the speedup ρ^i_{AB} of code A versus code B for instance i is the inverse of the ratio of their running times, and the geometric mean over instances $1, \dots, k$ is computed as $\left(\prod_{i=1}^k \rho^i_{AB}\right)^{\frac{1}{k}}$. For the hard instances the speedup is calculated in a similar way, but we favor the slower code by assuming that it terminates right after the time limit (1,800 CPU seconds).

A final comment is in order. If some instances are discarded because they are too easy or too hard, they should be too easy or too hard for *both* codes, as it may be unfair to mark as "too easy" the instances solved in less than a few seconds by just one of the two codes, in that this will put the slower code in an advantageous position.

5.3 Parameter tuning

In this section, we compare various versions of our algorithm to a reference ILOG-Cplex version, with the aim of setting its main parameters in an appropriate way.

The reference code is default ILOG-Cplex 8.1 version, called Cplex in the sequel, with a dummy cut callback that does nothing. This choice is motivated by that fact that the presence of a cut callback forces ILOG-Cplex not to apply irreversible transformations to the model, so it seems to be fair to deactivate these irreversible transformations for all codes under comparisons. (Acually, we found that the dummy cut callbak improves slightly the default ILOG-Cplex performance in most of instances in our testbed.) Internal cut generation is enabled as in the default ILOG-Cplex setting.

Our algorithm, called 012 in the sequel, is obtained from Cplex (the reference ILOG-Cplex version) by just replacing the dummy cut callback by the $\{0, \frac{1}{2}\}$-cut separation procedure illustrated in the previous sections.

For the purpose of tuning, we decided to work with a reduced subset instances, and disregarded some instances that appeared extremely easy or extremely hard to solve.

Table 1 compares Cplex with a naive version of 012, where we apply no clever cut selection policy. The table shows that this version of 012 is only slightly better than Cplex: the number of instances solved by the two codes is the same, and 012 is only marginally faster.

A first attempt to improve the 012 performance is to simply limit the number of cuts that are added at each branching node, by setting parameter *cut_factor* to a smaller value. After appropriate tuning of this parameter, we obtained the improvement reported in Table 2: 7 more instances were solved, and the average speedup on the hard instances raised to 20.

Table 1: Naive cut selection policy. Speedup w.r.t. Cplex. Parameter values: $ub\_min\_eff = +\infty$, $max\_par = 1$, $cut\_factor = 10$, $recomb = off$.

	LINORD	SAT	Global
count	69	74	143
too easy	11	1	12
regular	58	41	99
hard	0	18	18
impossible	0	14	14
solved by both	69	42	111
not solved by 012	0	23	23
not solved by Cplex	0	23	23
solved only by 012	0	9	9
solved only by Cplex	0	9	9
speedup on regular	1.03	1.37	1.16
speedup on hard	–	1.71	1.71
speedup on the whole	1.03	1.36	1.19
CPU hours 012	1.11	14.48	15.59
CPU hours Cplex	0.86	15.35	16.21

Table 2: Limiting the number of cuts. Speedup w.r.t. Cplex. Parameter values: $ub\_min\_eff = +\infty$, $max\_par = 1$, $cut\_factor = 0.3$, $recomb = off$.

	LINORD	SAT	Global
count	69	74	143
too easy	11	2	13
regular	58	48	106
hard	0	9	9
impossible	0	15	15
solved by both	69	50	119
not solved by 012	0	16	16
not solved by Cplex	0	23	23
solved only by 012	0	8	8
solved only by Cplex	0	1	1
speedup on regular	1.01	1.67	1.27
speedup on hard	–	19.84	19.84
speedup on the whole	1.00	2.00	1.43
CPU hours 012	1.14	11.56	12.70
CPU hours Cplex	0.86	15.35	16.21

An approximate tuning of the remaining parameters leads to the results in Table 3: 012 can now solve 9 more instances, and has a speedup of 3 over the whole testbed.

Strangely enough, the speedup over the hard instances in Table 3 is lower than the one reported in Table 1. This is explained by the fact that the number of hard instances in the two tables is different, in that several "impossible" instances became "hard" as they were solved by 012 (but not by Cplex) in less than 1,800 CPU seconds. Therefore a straight comparison of the speedups may be misleading.

At this point, we investigated the deterioration in the performance of 012+ (i.e., code 012 with the "good tuning" parameter setting of Table 3), when one of its parameters is disabled by setting it to a dummy value.

Table 3: Good-tuned 012 version (namely, 012+). Speedup w.r.t. Cplex. Parameter values: $ub\_min\_eff = 0.02$, $max\_par = 0.1$, $cut\_factor = 0.3$, $recomb = on$

	LINORD	SAT	Global
count	69	74	143
too easy	12	1	13
regular	57	44	101
hard	0	28	28
impossible	0	1	1
solved by both	69	45	114
not solved by 012+	0	7	7
not solved by Cplex	0	23	23
solved only by 012+	0	22	22
solved only by Cplex	0	6	6
speedup on regular	1.42	4.89	2.44
speedup on hard	–	12.10	12.10
speedup on the whole	1.34	7.00	3.06
CPU hours 012+	0.67	6.66	7.33
CPU hours Cplex	0.86	15.35	16.21

Tables 4, 5, and 6 illustrate the results obtained by disabling, respectively, the minimum efficacy requirement, the maximum parallelism requirement, and the possibility of considering previously-generated $\{0, \frac{1}{2}\}$-cuts as inequalities to be combined in the separation procedure. In all cases, the results are notably worse for what concerns both the number of instances solved and the running times. This shows the important role played by these three parameters.

Table 4: Efficacy disabled. Speedup w.r.t. 012+. Parameter values: $ub\_min\_eff = +\infty$, $max\_par = 0.1$, $cut\_factor = 0.3$, $recomb = on$.

	LINORD	SAT	Global
count	69	74	143
too easy	12	15	27
regular	57	41	98
hard	0	14	14
impossible	0	4	4
solved by both	69	56	125
not solved by 012	0	15	15
not solved by 012+	0	7	7
solved only by 012	0	3	3
solved only by 012+	0	11	11
speedup on regular	0.80	0.81	0.81
speedup on hard	–	0.11	0.11
speedup on the whole	0.86	0.58	0.69
CPU hours 012	0.83	10.30	11.13
CPU hours 012+	0.67	6.66	7.33

Finally, Table 7 illustrates the behavior of the variant of 012+ in which the number of cuts added to the model is essentially unbounded. Quite surprisingly, the set of instances solved is unchanged, but running times are significantly smaller. This shows that, as long as only "good" cuts (according to the efficacy and

Table 5: Parallelism bound disabled. Speedup w.r.t. 012+. Parameter values: $ub\_min\_eff = 0.02$, $max\_par = 1$, $cut\_factor = 0.3$, $recomb = on$.

	LINORD	SAT	Global
count	69	74	143
too easy	11	13	24
regular	58	33	91
hard	0	27	27
impossible	0	1	1
solved by both	69	46	115
not solved by 012	0	22	22
not solved by 012+	0	7	7
solved only by 012	0	6	6
solved only by 012+	0	21	21
speedup on regular	0.88	0.50	0.71
speedup on hard	–	0.12	0.12
speedup on the whole	0.89	0.34	0.54
CPU hours 012	0.89	14.74	15.63
CPU hours 012+	0.67	6.66	7.33

Table 6: Only rank-1 cuts (cut recombination disabled). Speedup w.r.t. 012+. Parameter values: $ub\_min\_eff = 0.02$, $max\_par = 0.1$, $cut\_factor = 0.3$, $recomb = off$

	LINORD	SAT	Global
count	69	74	143
too easy	13	13	26
regular	56	41	97
hard	0	17	17
impossible	0	3	3
solved by both	69	54	123
not solved by 012	0	16	16
not solved by 012+	0	7	7
solved only by 012	0	4	4
solved only by 012+	0	13	13
speedup on regular	1.01	0.59	0.80
speedup on hard	–	0.09	0.09
speedup on the whole	1.00	0.43	0.65
CPU hours 012	0.67	11.39	12.06
CPU hours 012+	0.67	6.66	7.33

parallelism requirement) are selected, it is not essential to limit the number of cuts to be added.

As outcome of the above tests, we decided that the version of 012 with the parameter setting as in Table 7 represents the best-tuned version of our code. This version will be called 012* in the sequel.

5.4 Results on the whole testbed

Table 8 presents a comparison of Cplex (the reference versions of ILOG-Cplex) and 012* (our code based on $\{0, \frac{1}{2}\}$-cuts) on the whole testbed. The table shows that 012* was able to solve a significantly larger number

Table 7: Best-tuned 012 version (namely, 012*). Speedup w.r.t. 012+. Best parameter values: $ub\_min\_eff = 0.02$, $max\_par = 0.1$, $cut\_factor = 10$, $recomb = on$.

	LINORD	SAT	Global
count	69	74	143
too easy	13	25	38
regular	56	42	98
hard	0	0	0
impossible	0	7	7
solved by both	69	67	136
not solved by 012*	0	7	7
not solved by 012+	0	7	7
solved only by 012*	0	0	0
solved only by 012+	0	0	0
speedup on regular	1.00	1.81	1.29
speedup on hard	–	–	–
speedup on the whole	1.00	1.40	1.19
CPU hours 012*	0.67	5.49	6.16
CPU hours 012+	0.67	6.66	7.33

of instances that Cplex—several instances that Cplex could not solve in 1,800 CPU seconds, are solved by 012* in just a few minutes. LINORD instances turn out to be relatively easy to solve by both codes. On the other hand, for SAT and MAXSAT instances the performance of 012* is much better than that of Cplex, with a speedup of about 20 on the hard ones. This confirms the important role played by $\{0, \frac{1}{2}\}$-cuts for this kind of problems, as well as the effectiveness of our cut-selection policy.

Table 8: Final computational results: 012* vs Cplex. Speedup w.r.t. Cplex.

	LINORD	MAXSAT	SAT	Global
count	69	222	222	513
too easy	12	60	120	192
regular	57	87	45	189
hard	0	37	34	71
impossible	0	38	23	61
solved by both	69	147	165	381
not solved by 012*	0	45	30	75
not solved by Cplex	0	68	50	118
solved only by 012*	0	30	27	57
solved only by Cplex	0	7	7	14
speedup on regular	1.43	1.77	4.54	2.07
speedup on hard	–	15.53	20.20	17.61
speedup on the whole	1.34	1.97	2.15	1.95
CPU hours 012*	0.67	28.94	17.64	47.25
CPU hours Cplex	0.86	41.24	29.03	71.13

6 Conclusions

Branch&Cut designers often have to face an Hamletic question—to cut or not to cut? In this paper we have discussed our own experience on how to exploit as effectively as possible a specific class of cuts. We addressed the family of $\{0, \frac{1}{2}\}$-cuts introduced in [4], and implemented a quite successful heuristic separator.

This was however just the beginning of our work. We learned that cut-selection criteria are at least as important as cut separation methods. As a matter of fact, as shown in our computational study, different cut-selection criteria used on top of a same cut separator can lead to quite different speedups on a same testbed. Moreover, the cut-selection criteria cannot be independent of the Branch&Cut implementation they are embedded into. In particular, the widely-used ILOG-Cplex framework seems to suggest conservative cut-selection policies.

A future direction of research should address other families of cuts which can be generated in a large number and with little effort—though imposing specific properties such as maximum violation or maximum depth, is difficult. A familiar example is made by the classical Gomory cuts: they can be generated in copious way from the fractional rows of the LP tableau, but finding a most-violated Gomory cut is a hard problem. Potential candidates are also the cuts based on the theory of corner polyhedra and T-spaces that have been recently proposed by [10].

We conjecture that the strategy of having very large cut pools feeded up in a generous way by fast separation heuristics (if coupled with a rigorous cut-selection policy) can compare favorably with the opposite cut handling policy—spend a significant portion of the overall computing time in separating just a few deep cuts. The working hypothesis here is that the larger freedom in choosing cuts from a very large pool can compensate from the poorer average quality of the pool cuts. If confirmed, this hypothesis can give an additional explanation for the empirical evidence reported, e.g., in [3], according to which the most effective cuts in practice seem to be those that can be derived easily from the LP tableau—notably, mixed-integer Gomory cuts: besides their inherent strength, these cuts have the advantage of being generated easily and in large bunches at each separation call.

Another direction of research is the study of alternative separation procedures for $\{0, \frac{1}{2}\}$-cuts or, more generally, for mod-k cuts. A very preliminary computational experience showed that alternative separation procedures based on local search or on the solution of a suitable system of linear equations over $GF(k)$ [5], tend to produce a large number of violated cuts as well, that however appear to be less effective than those found by the heuristic illustrated in the present paper. It would be interesting to see if this drawback can be overcome either by improving these separation procedures, or by an appropriate post-processing of the cuts obtained.

Acknowledgements

Work partially supported by MIUR and CNR, Italy.

References

[1] E. Balas, S. Ceria, G. Cornuéiols, "Mixed 0-1 Programming by Lift-and-Project in a Branch-and-Cut Framework", Management Science, Vol. 42, p. 1229–1246, 1996.

[2] E. Balas, S. Ceria, G. Cornuéjols and N. Natraj, "Gomory Cuts Revisited", Operations Research Letters, vol. 19, p. 1–9, 1996.

[3] R.E. Bixby, M. Fenelon, Z. Gu, E. Rothberg, R. Wunderling, "MIP: Theory and practice—closing the gap", in M.J.D. Powell, S. Scholtes (eds.) *System Modelling and Optimization: Methods, Theory, and Applications.* Kluwer Academic Publishers:New York, p. 19–49, 2000.

[4] A. Caprara, M. Fischetti, "$\{0, \frac{1}{2}\}$-Chvátal-Gomory Cuts", Mathematical Programming, vol. 74, p. 221-235, 1996.

[5] A. Caprara, M. Fischetti, A.N. Letchford, "On the Separation of Maximally Violated mod-k Cuts", Mathematical Programming, vol. 87, p. 37–56, 2000.

[6] V. Chvátal, "Edmonds polytopes and a hierarchy of combinatorial problems", Discrete Mathematics, vol. 4, p. 305-337, 1973.

[7] ILOG Cplex 8.1: User's Manual and Reference Manual, ILOG, S.A., `http://www.ilog.com/`, 2003.

[8] DIMACS Challenge Web Page: `http://mat.gsia.cmu.edu/challenge.html`.

[9] R.E. Gomory, "An algorithm for integer solutions to linear programs". In R.L. Graves, P. Wolfe (eds.) *Recent Advances in Mathematical Programming.* McGraw-Hill:New York, 1963.

[10] R.E. Gomory, E.L. Johnson, L. Evans, "Corner Polyhedra and their connection with cutting planes", Mathematical Programming, vol. 96, p. 321–339, 2003.

[11] M. Grötschel, L. Lovász, A.J. Schrijver, *Geometric Algorithms and Combinatorial Optimization.* Wiley: New York, 1988.

[12] D.S. Johnson, M.A. Trick (eds.), *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science 26. AMS Press:Providence, 1996.

[13] G. Reinelt, personal communication, 2003.