INFORMS Journal on Computing
Vol. 19, No. 2, Spring 2007, pp. 229–238
ISSN 1091-9856 | EISSN 1526-5528 | 07 | 1902 | 0229

DOI 10.1287/ijoc.1050.0162

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

Giuseppe Andreello
DEI, Università di Padova, Via Gradenigo, 6/A, I-35131 Padova, Italy, a.beppe@libero.it

Alberto Caprara
DEIS, Università di Bologna, Viale Risorgimento, 2, I-40136 Bologna, Italy, acaprara@deis.unibo.it

Matteo Fischetti
DEI, Università di Padova, Via Gradenigo, 6/A, I-35131 Padova, Italy, fisch@dei.unipd.it

Embedding cuts into a branch-and-cut framework is a delicate task, especially when a large set of cuts is available. In this paper we describe a separation heuristic for $\{0, \frac{1}{2}\}$-cuts, a special case of Chvátal-Gomory cuts, that tends to produce many violated inequalities within relatively short time. We report computational results on a large testbed of *integer linear programming* (ILP) instances of combinatorial problems including satisfiability, max-satisfiability, and linear ordering problems, showing that a careful cut-selection strategy produces a considerable speedup with respect to the cases in which either the separation heuristic is not used at all, or all of the cuts it produces are added to the LP relaxation.

Key words: programming, integer, algorithms, cutting plane; programming, integer, applications
History: Accepted by William J. Cook, Area Editor for Design and Analysis of Algorithms; received May 2004; revised January 2005; accepted July 2005.

1. Introduction

Embedding cuts into a branch-and-cut framework is a delicate task. Paradoxically, a difficult situation arises when the implemented separation procedures are very successful and produce many cuts quickly. In this case, it is of crucial importance to balance between the benefits deriving from a tighter (but larger) LP relaxation, and the time required for its solution. Indeed, embedding the new cuts in a naive way could well reduce the number of branching nodes, but the overall computing time required increases significantly. This (often frustrating) situation is well known to the designers of exact algorithms based on the LP relaxation, who often avoid at all the generation of the new cuts, or allow cut generation only at the root node of the branching tree according to the so-called cut-and-branch strategy.

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 ILPs (Caprara and Fischetti 1996). Our separation procedure is embedded within ILOG-Cplex 8.1, a widely-used and very effective commercial ILP solver. Computational results on a large testbed of ILP instances of combinatorial problems, including satisfiability, max-satisfiability, and linear ordering problems, are reported. On these problems, our first attempt at incorporating $\{0, \frac{1}{2}\}$-cuts within the ILOG-Cplex 8.1 framework produced a code that 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 even turn out to be facet-defining for the underlying integer polytope. However, a more sophisticated cut-selection strategy produced considerable speedup on our testbed. Interestingly, our separation procedure was used as a black box—all the improvements came from a better way to exploit the cuts. In other words, our cut-selection policy does not need tight interaction with the separation procedure.

In Section 2 we review the theory of $\{0, \frac{1}{2}\}$-cuts, and in Section 3 we present our heuristic separation procedure. Section 4 describes the ILOG-Cplex 8.1 branch-and-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 measure we used for cut selection. Section 5 reports our computational experience and compares various cut-selection strategies on a large testbed containing ILP instances with a strong combinatorial structure. Finally, some conclusions are drawn in Section 6.

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 Caprara and Fischetti (1996).

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

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

A *Chvátal-Gomory* (*CG*) *cut* (Gomory 1963, Chvátal 1973) is a valid inequality for P_I of the form $\lambda^T Ax \le \lfloor \lambda^T b \rfloor$, where $\lambda \in R^m_+$ is such that $\lambda^T A \in \mathbb{Z}^n$, and $\lfloor \cdot \rfloor$ denotes lower integer part. Undominated CG cuts arise only for rational $\lambda \in [0, 1)^m$ (see, e.g., Caprara et al. 2000).

CG cuts can equivalently be obtained in the following way (Caprara et al. 2000). Let $\mu \in \mathbb{Z}^m_+$ and $q \in \mathbb{Z}_+$ be such that $\mu^T A \equiv 0 \pmod q$ and $\mu^T b = kq + r$ with $k \in \mathbb{Z}$ and $r \in \{1, \dots, q-1\}$. Then the *mod-q cut* $\mu^T Ax \le kq$ is a valid inequality for P_I. This inequality can be written as $\mu^T(b - Ax) \ge r$, so a given $x^* \in P$ violates $\mu^T Ax \le kq$ if and only if $\mu^T(b - Ax^*) < r$.

In Caprara and Fischetti (1996), 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

$$s^* := b - Ax^* \ge 0 \text{ and solve}$$

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

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

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

$\{0, \frac{1}{2}\}$-SEP is NP-complete in the general case (Caprara and Fischetti 1996), but there are important special cases for which a polynomial separation algorithm can be derived. Among these cases, one that appears to have an important 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, Caprara and Fischetti (1996) show how to relax the original constraints to achieve this property. The resulting procedure is a general separation heuristic for $\{0, \frac{1}{2}\}$-cuts, which we outline next.

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

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

The following result is a corollary of a general result in Caprara and Fischetti (1996) involving binary clutters. As it is the key to our separation method, here we give a short constructive proof.

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

PROOF. Let $G = (V, E)$ be an undirected multigraph having a vertex j for each column j of $\bar{A}$, plus a special vertex q. Graph G is weighted and labeled on its edges. There is an edge e_i for each row $i \in M$, with *weight* $w(e_i) := s^*_i$ and labeled *odd* if $\bar{b}_i = 1$, *even* otherwise; this edge connects the two vertices h and k such that $O_i = \{h, k\}$ (if $O_i = \{h\}$, then let the edge connect vertex h to the special vertex q). A given $E^* \subseteq E$ is an *Eulerian cycle* if each component of the subgraph induced by E^* is Eulerian, i.e., $|E^* \cap \delta(j)|$ is even for $j \in V$. 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 $\mu \in \mathcal{F}(\bar{A}, \bar{b})$ 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 $w(C^*) \le w(E^*)$ holds as $w(e) \ge 0$ for all $e \in E$.

A polynomial algorithm for the minimum-weight odd cycle problem is known from folklore (see, e.g., Grötschel et al. 1988). This algorithm is based on shortest-path computations on an auxiliary graph, and runs in $O(n^3)$ time. □

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 $Ax \le b$ is relaxed into $A'x \le b'$ (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 Caprara and Fischetti (1996) the case where bound constraints of the type $l \le x \le d$ are part of the system $Ax \le b$ is considered (possibly $l_j = -\infty$ or $d_j = +\infty$ for some j). Then a suitable relaxation $A'x \le b'$ can readily be obtained from $Ax \le b$ by

replacing each inequality $\sum_j a_{ij}x_j \le b_i$ with $|O_i| \ge 3$, by its so-called *LU-weakenings*

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

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

Although $A'x \le b'$ has, in general, an exponential number of rows, the corresponding $\{0, \frac{1}{2}\}$-SEP can still 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 $j \in O_i \backslash \{h, k\}$, 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 $O(\sum_{i \in M} |O_i|^3 + n^3)$ and then $O(mn^3)$, as $|O_i| = O(n)$ for $i \in M$.

Let $x^* \in P$ denote the fractional point to be separated, and let $s^* := b - Ax^* \ge 0$ 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^*. In fact, as explained in the sequel the heuristic may produce more than one cut.

3.1. Reduction

First of all, following Caprara and Fischetti (1996), we apply the following preprocessing steps in an 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 $Ax \le b$ with $s_i^* \ge 1$ is removed.
- R3. For $j \in N$, we consider the (possibly empty) set $S_j := \{i \in M: \bar{a}_{ik} = 1 \text{ if and only if } k = j\}$. If there is a row $i \in S_j$ with $s_i^* = 0$, we can get rid of variable x_j and row i. This reduction is iterated until no further variable can be removed.

In particular, Reduction R3 applies to the case in which some components of x^* are equal to the lower/upper bounds of the corresponding variables, typically allowing for a substantial size reduction of the separation instance.

3.2. Construction of the Separation Multigraph

We construct the separation multigraph $G = (V, E)$ as in the proof of Theorem 1. Recall that V contains a vertex j for each variable x_j, plus a special vertex q used to deal with the constraints with just one odd coefficient. Clearly, for each pair of vertices h, k in G only the odd and even edges with minimum weight have to be stored. Let $\text{odd}(h, k)$ and $\text{even}(h, k)$ denote these minimum weights. For each odd and even edge of G between vertices h, k, we also store the index i of the constraint in the original system $Ax \le b$ that produced weight $\text{odd}(h, k)$ and $\text{even}(h, k)$, respectively, to be able to reconstruct the output set I^*.

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

$$\delta_j^p := \min\{s_i^*: i \in S_j, \bar{b}_i = p\} \quad \text{for } p = 0, 1$$

($\delta_j^p = +\infty$ if no suitable i exists), and we set $\text{odd}(j, q) := \delta_j^1$ and $\text{even}(j, q) := \delta_j^0$.

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

Finally, we address the "complicated" constraints i with $|O_i| \ge 3$, and consider each of the $O(|O_i^2|)$ pairs $h, k \in O_i$. For each triple (i, h, k), we update $\text{odd}(h, k) := \min\{\text{odd}(h, k), best^1\}$ and $\text{even}(h, k) := \min\{\text{even}(h, k), best^0\}$, 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 $j \in O_i \backslash \{h, k\}$. They are computed, by dynamic programming, by initializing $best^0 := best^1 := s_i^*$ and then by computing, for each $j \in O_i \backslash \{h, k\}$ (in any sequence)

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

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

(This computation can be aborted as soon as $\min\{best^0, best^1\} \ge 1$.)

3.3. Determination of a Minimum-Weight Odd Cycle

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., Grötschel et al. 1988). 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 vertex $h \in V$. For each even edge in G joining vertices h, k, having weight $\text{even}(h, k)$, H contains edges h_1,k_1 and h_2,k_2 of weight $\text{even}(h, k)$. Moreover, for each odd edge in G joining vertices h,k, having weight $\text{odd}(h, k)$, H contains edges h_1,k_2 and h_2,k_1 of weight $\text{odd}(h, k)$. The shortest odd cycle visiting vertex h in

G corresponds to the shortest path joining h_1 to h_2 in H.

In order to find a larger number of cuts, rather than simply finding the shortest odd cycle visiting each vertex $h \in V$, with a comparable computational effort we find the shortest odd cycle visiting each pair of vertices $h, k \in V$. To this aim, for each vertex $h \in V$, 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 vertex). By symmetry, this also provides the shortest-path anti-arborescence rooted in h_2 (representing the shortest paths from each other vertex to h_2). With this information, we consider each vertex $k \in V \setminus \{h\}$ 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 vertices h, k.

3.4. Determination of Violated Cuts

For each odd cycle C^* with weight $z(C^*) < 1$, 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 $\sum_{i \in I^*} s_i^* \le z(C^*)$. 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, so summing together only the inequalities in I^* (without the inequalities in S_j used in the edge-weight definition) would produce an inequality $\beta x \le \beta_0$ (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 $\mu \in \mathcal{F}(\bar{A}, \bar{b})$. 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 $\beta x \le \beta_0$ 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.

4. The Overall Branch-and-Cut Algorithm

Our work is aimed at analyzing the benefits deriving from the use of $\{0, \frac{1}{2}\}$-cuts within an existing branch-and-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 was selected as the external framework. ILOG-Cplex 8.1 is widely-used branch-and-cut 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 to embed our separator for $\{0, \frac{1}{2}\}$-cuts within the existing branch-and-cut algorithm. As explained below, the shell also handles an internal cut pool, whose interaction with the current LP is governed by a nontrivial cut-selection procedure—as shown in the computational section, cut handling is of crucial importance.

All procedures used in our study were 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 Measure

On problems with a strong combinatorial structure, our $\{0, \frac{1}{2}\}$-separator may add to the cut pool a huge list of cuts. Adding all these cuts to the original formulation is therefore likely to increase the LP size in an excessive way, leading to higher node solution times. So, it is essential to select carefully the cuts to be added to the formulation. Our selection rules are based on the following cut quality measure, analogous to those proposed in Balas et al. (1996).

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

$$\text{dist}(x^*, \alpha, \alpha_0) := \frac{|\alpha^T x^* - \alpha_0|}{\|\alpha\|}. \tag{2}$$

Following geometric intuition, it is quite natural to consider this distance as a first-order measure of the expected efficacy of a cut $\alpha x \le \alpha_0$ violated by the fractional point x^*, the larger the distance the better the cut (Balas et al. 1996).

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 (2) is not affected by component α_j, so the presence of the norm of α in 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 (2), 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

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

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

$$\text{eff}(x^*, \alpha, \alpha_0) \geq min\_eff \tag{4}$$

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 nonincreasing efficacy, we need to face the issue of preventing "similar" cuts from being added together to the LP. 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*:

$$\text{par}(\alpha, \beta) := \frac{|\alpha^T \beta|}{\|\alpha\| \, \|\beta\|} \leq max\_par. \tag{5}$$

In particular, the choice $max\_par = 0$ would force the cuts to be orthogonal, e.g., with disjoint support. Lift-and-project cuts in Balas et al. (1996) are selected in the same way, but using $max\_par = 0.999$ in condition (5) just to avoid adding duplicate cuts; no computational results for other values of *max_par* are reported in Balas et al. (1996). The actual value of *max_par* used in our implementation is discussed below.

4.2. The ILOG-Cplex 8.1 Framework

Implanting a piece of software within a sophisticated code such as ILOG-Cplex 8.1 requires some knowledge of the main strategies used therein.

First, ILOG-Cplex 8.1 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. We note that a cut pool is implemented in the latest ILOG-Cplex version 9.0, which allows for a simplified implementation of the approach illustrated in the sequel but does not change the essence of the results we present. In particular, adding the cuts to the LP in a static way (as opposed to implementing a suitable cut-purging policy within the ILOG-Cplex 9.0 framework) appears to be beneficial for the combinatorial problems addressed in our computational section.

Second, cuts are not generated at each branching node. An intense attempt to generate new cuts is made only at the root node, and then a moderate separation effort is spent after each backtracking step.

We implemented our own cut pool and used as separation procedure the cut-callback procedure described in Algorithm 1, where *LP* denotes the set of constraints of the current LP relaxation. Some comments follow.

ALGORITHM 1. Separation strategy

if this is the first attempt to generate cuts **then**
  /* define some global counters */
  $added\_cuts := 0$
  $max\_added\_cuts := cut\_factor * (\#\text{ LP rows})$
  $max\_pool := \max\{8{,}000, 4 * (\#\text{ LP rows})\}$
end if
if (the number of backtrackings is not a multiple of 4) **or** ($added\_cuts \geq max\_added\_cuts$) **then**
  return
end if
add to *POOL* all the $\{0, \frac{1}{2}\}$-cuts violated by x^* found by the heuristic separation procedure
sort *POOL* by decreasing efficacy, *POOL*[0] containing the cut with largest efficacy
if $|POOL| > max\_pool$ **then**
  remove the last $|POOL| - max\_pool$ cuts from *POOL*
end if
if this is the first attempt to generate cuts **then**
  /* define the global variable *min_eff* */
  $min\_eff := \min\{ub\_min\_eff, 0.7 * \text{eff}(POOL[0])\}$
end if
$miss := 0$
if $\text{eff}(POOL[0]) < min\_eff$ **then**
  $miss := miss + 1$
  if $miss = 20$ **then**
    $miss := 0$
    $min\_eff := min\_eff - 0.03$
  end if
  return
end if
$S := \varnothing$
$i := 0$
while ($i \leq |POOL|$) **and** ($\text{eff}(POOL[i]) \geq min\_eff$) **do**
  if $\text{par}(c, POOL[i]) \leq max\_par$ for all cuts $c \in S$ **then**
    add $POOL[i]$ to the current LP
    $S := S \cup \{POOL[i]\}$
    $added\_cuts := added\_cuts + 1$
    if $added\_cuts \geq max\_added\_cuts$ **then**
      return
    end if
  end if
  $i := i + 1$
end while
return.

• In the description, the values of the parameters used in our implementation are reported explicitly, with the exception of *ub_min_eff*, *cut_factor*, *max_par*, and *recomb* (the last one not mentioned in the description but illustrated below). The reason is that the performances of the branch-and-cut algorithm are critically dependent on these last four parameters, whose tuning is described in the next section, whereas the other parameters do not appear to be crucial.

• At the root node the separation procedure is called only a limited number of times (equal to 5 in our implementation). In addition, as in Balas et al. (1996) we found that better overall results are typically obtained if separation is invoked only at each k-th backtracking (we choose $k = 4$ in our implementation).

• If the pool contains more than *max_pool* cuts, after it has been ordered by efficacy, the least efficacious cuts are removed.

• The maximum parallelism constraint (5), 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. (Our experiments showed that applying the condition to the whole cut pool rejects too many cuts when small values of *max_par* are used.) This choice ensures that new and efficacious cuts can be added even if they are parallel to old cuts.

• The minimum efficacy in (4) 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 it is to build deep cuts.

• The number of cuts added to the LP can never exceed *cut_factor* times the number of constraints in the initial LP formulation.

• Depending on the value of a flag, called *recomb*, the separator generates $\{0, \frac{1}{2}\}$-cuts by combining constraints of the initial formulation (*recomb* = *off*) or of the current one, i.e., the initial LP amended by the cuts added in the previous iterations (*recomb* = *on*).

5. Computational Experiments

Generally speaking, we cannot claim $\{0, \frac{1}{2}\}$-cuts are useful for the solution of *any* ILP, and we have to content ourselves to find classes of ILPs that can benefit from use of our $\{0, \frac{1}{2}\}$-cut separator. The computational experiments presented in this section show that there are important classes of combinatorial ILPs for which the usual cover- and clique-separation procedures are completely ineffective, whereas $\{0, \frac{1}{2}\}$-cuts are quite useful and allow for considerable speedup (if dealt with properly).

5.1. The Testbed

We considered the following classes of combinatorial problems, which produce notoriously hard-to-solve ILPs. 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 *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 $C_1 \wedge C_2 \wedge \cdots \wedge C_m$.

SAT is easily formulated as an ILP with a binary variable for each of the boolean variables, 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 it 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). 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 (Johnson and Trick 1996).

5.1.2. Maximum Satisfiability Problems (MAXSAT). The MAXSAT problem is only slightly different from SAT. Given a set of clauses, MAXSAT 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 replace the right-hand side value 1 by a binary variable z_k in the corresponding constraint. The objective function is then $\max \sum_k z_k$.

We solved a MAXSAT version of every SAT instance considered.

5.1.3. Linear Ordering Problems (LOP). 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 $x_{ij} = 1$ if i precedes j in the permutation, and $x_{ij} = 0$ 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):

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

We found that the instances from the public LOLIB library tend to have a negligible integrality gap at the root node, so they can be solved quite easily. For other known classes of large instances, the main difficulty is instead the solution of the initial LP, so our cuts cannot help in their solution. The instances we considered in our testbed were therefore generated according to the following random procedure suggested by Reinelt (2003), which is intended to produce medium-size instances with a significant integrality gap at the root node of the initial formulation. We considered the 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 ten instances for each of the following combinations of parameters $(prob, C_{\max})$: $(0.5, 4)$, $(0.5, 10)$, $(0.2, 10)$, $(0.3, 10)$, $(0.1, 10)$, $(0.05, 10)$, $(0.5, 10^6)$. As already mentioned, for benchmarking purposes our LOP instances are available, on request, from the authors.

5.2. Speedup Evaluation

Our algorithm was tested and compared to a reference ILOG-Cplex 8.1 configuration, proceeding in the same way as ILOG does in order to validate its own new releases, as described in Bixby et al. (2000).

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 2400+;
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. *very hard* 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 Bixby et al. (2000), 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 $1, \ldots, k$ is computed as $(\prod_{i=1}^{k} \rho^i_{AB})^{1/k}$. For the hard instances the speedup is calculated similarly, but we favor the slower code by assuming that it terminates right after the time limit (1,800 CPU seconds).

Instances are discarded only if they are too easy or too hard for *both* codes, as it is 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 puts the slower code in an advantageous position.

5.3. Parameter Tuning

With the aim of setting the four parameters mentioned in Section 4.2 in an appropriate way, in this section we compare various versions of our algorithm to a reference 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 Cplex not to apply irreversible transformations to the model. (Actually, we found that the dummy cut callback improves slightly the Cplex performance in most of the instances in our testbed.) Internal cut generation is enabled.

Our algorithm, called A012 in the sequel, is obtained from Cplex 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 of instances, and disregarded some instances that appeared extremely easy or extremely hard to solve.

Table 1 compares Cplex with a naive version of A012, where we apply a straightforward cut-selection policy. The table shows that this version of A012 is only slightly better than Cplex: the number of instances solved by the two codes is the same, and A012 is only marginally faster.

A first attempt to improve the A012 performance is simply to limit the number of cuts that are added, by setting parameter *cut_factor* to a smaller value. After some tuning of this parameter, we obtained the improvement reported in Table 2. Seven more instances were solved, and the average speedup on the hard instances increased to 20.

An approximate tuning of the remaining parameters leads to the results in Table 3. A012 can now solve nine more instances, and has a speedup of 3 over the whole testbed. Note that 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 "very hard" instances became "hard" as they were solved by A012 (but not by Cplex) in less than 1,800 CPU seconds. Therefore a straight comparison of the speedups is misleading.

Table 1 Naive Cut-Selection Policy

	LOP	SAT	Global
Count	69	74	143
Too easy	11	1	12
Regular	58	41	99
Hard	0	18	18
Very hard	0	14	14
Solved by both	69	42	111
Not solved by A012	0	23	23
Not solved by Cplex	0	23	23
Solved only by A012	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 A012	1.11	14.48	15.59
CPU hours Cplex	0.86	15.35	16.21

Note. Speedup w.r.t. Cplex. Parameter values: *ub_min_eff* $= +\infty$, *max_par* $= 1$, *cut_factor* $= 10$, *recomb* $=$ *off*.

Table 3 Well-Tuned A012 Version (Namely, A012+)

	LOP	SAT	Global
Count	69	74	143
Too easy	12	1	13
Regular	57	44	101
Hard	0	28	28
Very hard	0	1	1
Solved by both	69	45	114
Not solved by A012+	0	7	7
Not solved by Cplex	0	23	23
Solved only by A012+	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 A012+	0.67	6.66	7.33
CPU hours Cplex	0.86	15.35	16.21

Note. Speedup w.r.t. Cplex. Parameter values: *ub_min_eff* $= 0.02$, *max_par* $= 0.1$, *cut_factor* $= 0.3$, *recomb* $=$ *on*.

We also investigated the deterioration in the performance of A012+ (i.e., code A012 with the parameter setting of Table 3), when one of its parameters is disabled by setting it to a dummy value, and found that the results are notably worse in terms of both the number of instances solved and the running times. This shows the important role played by these three parameters.

Finally, Table 4 illustrates the behavior of the variant of A012+ in which the number of cuts added to the model is essentially unbounded. In this case, the set of instances solved is unchanged, but running times are significantly smaller. (For SAT instances, the total number of cuts generated for the whole testbed increased from 15,300 to 28,166; LOP instances are not affected by this change.) This shows that, as long as only "good" cuts (according to the efficacy and parallelism requirement) are selected, it is not essential to limit the number of cuts to be added.

As a result of the above tests, we decided to use the version of A012 with the parameter settings as in Table 4. This version is called A012* in the sequel.

5.4. Results on the Whole Testbed

Table 5 presents a comparison of Cplex and A012* on the whole testbed. The table shows that A012* was able to solve a significantly larger number of instances than was Cplex—several instances that Cplex could not solve in 1,800 CPU seconds were solved by A012* in just a few minutes. LOP instances turn out to be relatively easy to solve by both codes. On the other hand, for SAT and MAXSAT instances the performance of A012* 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

Table 2 Limiting the Number of Cuts

	LOP	SAT	Global
Count	69	74	143
Too easy	11	2	13
Regular	58	48	106
Hard	0	9	9
Very hard	0	15	15
Solved by both	69	50	119
Not solved by A012	0	16	16
Not solved by Cplex	0	23	23
Solved only by A012	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 A012	1.14	11.56	12.70
CPU hours Cplex	0.86	15.35	16.21

Note. Speedup w.r.t. Cplex. Parameter values: *ub_min_eff* $= +\infty$, *max_par* $= 1$, *cut_factor* $= 0.3$, *recomb* $=$ *off*.

Table 4 Best-Tuned A012 Version (Namely, A012*)

	LOP	SAT	Global
Count	69	74	143
Too easy	13	25	38
Regular	56	42	98
Hard	0	0	0
Very hard	0	7	7
Solved by both	69	67	136
Not solved by A012*	0	7	7
Not solved by A012+	0	7	7
Solved only by A012*	0	0	0
Solved only by A012+	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 A012*	0.67	5.49	6.16
CPU hours A012+	0.67	6.66	7.33

Note. Speedup w.r.t. A012+. Best parameter values: *ub_min_eff* $= 0.02$, *max_par* $= 0.1$, *cut_factor* $= 10$, *recomb* $=$ *on*.

Table 5 Final Computational Results: A012* vs. Cplex

	LOP	MAXSAT	SAT	Global
Count	69	222	222	513
Too easy	12	60	120	192
Regular	57	87	45	189
Hard	0	37	34	71
Very hard	0	38	23	61
Solved by both	69	147	165	381
Not solved by A012*	0	45	30	75
Not solved by Cplex	0	68	50	118
Solved only by A012*	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 A012*	0.67	28.94	17.64	47.25
CPU hours Cplex	0.86	41.24	29.03	71.13

Note. Speedup w.r.t. Cplex.

this kind of problems, as well as the effectiveness of our cut-selection policy.

6. Conclusions

Branch-and-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 a specific class of cuts effectively. We addressed the family of $\{0, \frac{1}{2}\}$-cuts introduced in Caprara and Fischetti (1996), and implemented a heuristic separator that proved quite successful.

We learned that, besides cut-separation methods, cut-selection criteria are also very important, since different criteria used on top of the same separator can lead to quite different speedups on the same testbed, as long as the separator tends to produce a fairly large set of cuts relatively quickly. Moreover, the cut-selection criteria cannot be independent of the branch-and-cut implementation into which they are embedded.

A future direction of research should address other families of cuts that can be generated in large numbers 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 copiously 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 Gomory et al. (2003).

We conjecture that the strategy of having very large cut pools fed 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—to 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 for 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 Bixby et al. (2000), 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.

Other directions of research are the use of $\{0, \frac{1}{2}\}$-cuts for branch-and-cut codes for specific combinatorial problems such as LOP, and the study of alternative separation procedures for $\{0, \frac{1}{2}\}$-cuts or, more generally, for mod-k cuts. Reinelt and Wenger (2006) use the latter in a branch-and-cut algorithm for the capacitated vehicle-routing problem, and discuss criteria to choose among the huge number of cuts available from the separation procedure in Caprara et al. (2000). From our side, our very preliminary computational experience on combinatorial ILPs showed that the cuts produced by the method in Caprara et al. (2000) or by alternative separation procedures based on local search 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.

Acknowledgments

The authors are grateful to Gerd Reinelt for suggesting a way to generate LOP instances with large integrality gaps, and to three anonymous referees for their helpful comments that led to the improvement of the presentation.

References

Balas, E., S. Ceria, G. Cornuéjols. 1996. Mixed 0-1 programming by lift-and-project in a branch-and-cut framework. *Management Sci.* **42** 1229–1246.

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

Caprara, A., M. Fischetti. 1996. $\{0, \frac{1}{2}\}$-Chvátal-Gomory cuts. *Math. Programming* **74** 221–235.

Caprara, A., M. Fischetti, A. N. Letchford. 2000. On the separation of maximally violated mod-k cuts. *Math. Programming* **87** 37–56.

Chvátal, V. 1973. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Math.* **4** 305–337.

Gomory, R. E. 1963. An algorithm for integer solutions to linear programs. R. L. Graves, P. Wolfe, eds. *Recent Advances in Mathematical Programming*. McGraw-Hill, New York.

Gomory, R. E., E. L. Johnson, L. Evans. 2003. Corner polyhedra and their connection with cutting planes. *Math. Programming* **96** 321–339.

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

Johnson, D. S., M. A. Trick, eds. 1996. *Cliques, Coloring and Satisfiability: Second DIMACS Implementation Challenge, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 26. American Mathematical Society, Providence, RI.

Reinelt, G. 2003. Personal communication.

Reinelt, G., K. M. Wenger. 2006. Maximally violated mod-*p* cuts for the capacitated vehicle-routing problem. *INFORMS J. Comput.* **18** 466–479.