Zuse Institute Berlin

Takustr. 7
14195 Berlin
Germany

MARK TURNER[1], TIMO BERTHOLD [2], MATHIEU BESANÇON[3],
THORSTEN KOCH [4]

Branching via Cutting Plane Selection: Improving Hybrid Branching

[1] 0000-0001-7270-1496
[2] 0000-0002-6320-8154
[3] 0000-0002-6284-3033
[4] 0000-0002-1967-0077

ZIB Report 23-17 (June 2023)

Zuse Institute Berlin
Takustr. 7
14195 Berlin
Germany

Telephone: +49 30 84185-0
Telefax: +49 30 84185-125

E-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

BRANCHING VIA CUTTING PLANE SELECTION: IMPROVING HYBRID BRANCHING

A PREPRINT

Mark Turner[*] [†] turner@zib.de
Timo Berthold[*] [‡] timoberthold@fico.com
Mathieu Besançon[†] besancon@zib.de
Thorsten Koch[*] [†] koch@zib.de

June 13, 2023

ABSTRACT

Cutting planes and branching are two of the most important algorithms for solving mixed-integer linear programs. For both algorithms, disjunctions play an important role, being used both as branching candidates and as the foundation for some cutting planes. We relate branching decisions and cutting planes to each other through the underlying disjunctions that they are based on, with a focus on Gomory mixed-integer cuts and their corresponding split disjunctions. We show that selecting branching decisions based on quality measures of Gomory mixed-integer cuts leads to relatively small branch-and-bound trees, and that the result improves when using cuts that more accurately represent the branching decisions. Finally, we show how the history of previously computed Gomory mixed-integer cuts can be used to improve the performance of the state-of-the-art hybrid branching rule of SCIP. Our results show a 4% decrease in solve time, and an 8% decrease in number of nodes over affected instances of MIPLIB 2017.

1 Introduction

A Mixed-Integer Linear Program (MILP) is an optimisation problem that is classically defined as:

$$\underset{\mathbf{x}}{\operatorname{argmin}}\{\mathbf{c}^\intercal\mathbf{x} \mid \mathbf{A}\mathbf{x} \leq \mathbf{b},\ \mathbf{l} \leq \mathbf{x} \leq \mathbf{u},\ \mathbf{x} \in \mathbb{Z}^{|\mathcal{J}|} \times \mathbb{R}^{n-|\mathcal{J}|}\} \tag{1}$$

Here, $\mathbf{c} \in \mathbb{R}^n$ is the objective coefficient vector, $\mathbf{A} \in \mathbb{R}^{m\times n}$ is the constraint matrix, $\mathbf{b} \in \mathbb{R}^m$ is the right hand side constraint vector, $\mathbf{l}, \mathbf{u} \in \mathbb{R}^n \cup \{-\infty, \infty\}^n$ are the lower and upper variable bound vectors, and $\mathcal{J} \subseteq \{1, \ldots, n\}$ is the set of indices of integer variables. We denote the set of feasible solutions to (1) as $\mathcal{X}$, and the feasible region of the linear programming (LP) relaxation as $\mathcal{P}$, where the LP is derived by relaxing the integrality requirements of (1). An optimal solution to (1) is denoted $\mathbf{x}^*$, a feasible solution denoted $\hat{\mathbf{x}}$, and an LP optimal solution is denoted as $\mathbf{x}^{LP}$.

The core algorithm for solving MILPs is *branch-and-cut*, see [1] for a thorough introduction of MILP solving. Branch-and-cut consists of two main components, *branch-and-bound* and *cutting planes*. The branch-and-bound algorithm recursively divides the MILP into smaller subproblems, where splitting a problem into smaller subproblems is called *branching*. This recursion creates a *tree*, where each node is a subproblem. Traditionally branching is performed on an integer variable, x_i with fractional LP value, x_i^{LP}, creating the two LP subproblems with feasible regions $\mathcal{P} \cap \{x_i^{LP} \leq \lfloor x_i^{LP} \rfloor\}$ and $\mathcal{P} \cap \{x_i^{LP} \leq \lceil x_i^{LP} \rceil\}$. An example branching procedure is visualised in Figure 1. The algorithm bounds the optimal objective through upper bounds from feasible solutions of (1) obtained at leaf nodes of the tree, and lower bounds from LP relaxations that bound all subtrees of a node. The procedure of branch-and-bound that we are interested in is *variable selection*, which is concerned with determining which variable to branch on at a given node from the given candidates.

[*]Institute of Mathematics, Technische Universität Berlin, Straße des 17. Juni 135, 10623 Berlin, Germany
[†]Zuse Institute Berlin, Department of Mathematical Optimization, Takustr. 7, 14195 Berlin
[‡]Fair Isaac Germany GmbH, Takustr. 7, 14195 Berlin, Germany

Figure 1: (Left) An example branching decision. The red point is the LP optimal solution, the larger polytope the original LP feasible region, and the two blue polytopes the feasible LP regions of the subproblems. (Right) An example cutting plane.

A cutting plane, or *cut*, parameterised by $(\boldsymbol{\alpha}, \beta) \in \mathbb{R}^{n+1}$, is an inequality $\boldsymbol{\alpha}^\intercal \mathbf{x} \leq \beta$ that is violated by at least one solution of the LP relaxation but that does not increase the optimal value of the problem when added to the problem, i.e., it is valid for (1). This definition is more general than the classical one, which requires that cut not remove any integer-feasible solution to (1), and separate some LP feasible fractional solution. Our definition however captures additional families of cuts such as symmetry-breaking cuts [2, 3]. The cutting plane algorithm iteratively generates cuts, applies them, and re-solves the LP relaxation. In the classical case this algorithm is repeated until an integral LP relaxation solution is achieved. In branch-and-cut, this algorithm is repeated at the root node until some termination criteria is met, with additional cuts applied throughout the branch-and-bound tree. The procedure of cutting planes that we are interested in is *cut selection*, which is concerned with deciding which subset of the computed cuts to actually add to the LP relaxation.

In this paper we propose a technique for the variable selection problem based on cut selection. We note that the opposite direction – using variable selection techniques for cut selection – also presents a valid avenue of research, however this lies outside the scope of this work. For each branching candidate we generate a cut, greedily select the best cut according to standard cut scoring measures, and then branch on the corresponding candidate. Specifically, we will generate Gomory Mixed-Integer (GMI) cuts, for which we provide a detailed introduction in Section 3. This approach to variable selection is objective-free, i.e. not based on the objective vector, and is thus complementary to standard pseudo-cost based approaches [4]. We show the effectiveness of multiple variants of this branching rule using different levels of strengthened cuts, and compare them to standard branching rules from the literature. Finally, we show how this information can be incorporated into the hybrid branching rule of SCIP [5, 6], resulting in an improvement of general solver performance.

2 Related Work

Branching in MILP has been thoroughly studied, both theoretically and computationally, see [7, 4, 8]. The current state-of-the-art variable selection method is hybrid branching [5], which is reliability pseudo-cost branching [4] with integrated constraint satisfaction and satisfiability problem techniques. An array of other selection rules exist, such as nonchimerical fractionality branching [9], cloud branching [10], and general disjunction branching [11]. The above-stated methods, unlike our cut selection approach to branching, often depend on the objective function for variable selection. We note that variable selection has also served as the playground for introducing machine learning to MILP solvers, see [12, 13] for early examples, and [14] for an overview.

Cutting plane selection has been less studied than variable selection, however it is currently experiencing a recent refocus through machine learning driven research, see [15] for an overview. Early computational studies, see [1, 16], show that a diverse set of measures is necessary for good performance when scoring cuts. Both studies, as well as the computational study [17], use parallelism-based cut filtering algorithms, and show that the inclusion of filtering methods are critical for performance. These studies suggest that it is preferable for performance to select from a large set of weaker cuts than from a small set of stronger cuts. More recent work on cut selection, for which [18] provides ample motivation, is machine learning based. Specifically, research on theoretical guarantees [19, 20, 21], new scoring measures [22], the amount

of cuts to select [23], and learning to score cuts with supervised [24, 25], imitation [26], and reinforcement learning [27, 23].

Our work is not the first to use cutting plane selection to dictate branching decisions. Moreover, it is not the first to use GMI cuts specifically, see [28, 29]. In both papers the split disjunctions, which define the GMI cuts of tableau rows, are used as branching candidates. The efficacy of the GMI cuts are used to filter the set of branching candidates, where ultimately strong branching is used as the final selection criteria. In [29] additional experiments are presented that compare disjunctions derived from reduce-and-split cuts, see [30]. Our research differs from [28, 29] in that we branch on elementary splits, i.e., single variable disjunctions, we perform additional experiments using non-strengthened GMI cuts, and we integrate our approach with existing state-of-the-art history-based methods.

3 Gomory Mixed-Integer Cuts

This section is structured to give a thorough introduction to Gomory Mixed-Integer (GMI) cuts. Following the history and general introduction of Subsection 3.1, we introduce disjunctive, split, and intersection cuts in Subsection 3.2. We then step through the derivation of GMI inequalities in Subsection 3.3, ending with how GMI cuts are used and derived in practice 3.4. The geometric interpretation of the GMI cuts will be provided, and related to the overview of Subsection 3.2. For alternate overviews of GMI cuts see [31, 32, 30, 29].

3.1 GMI Introduction and History

First introduced in 1960 [33], GMI inequalities are general purpose inequalities valid for arbitrary bounded MILPs. They can be used to iteratively tighten a LP relaxation of an MILP, and when they are generated to separate a specific solution, are referred to as cutting planes, or *cuts*. In practice they are generated to separate the current LP solution using the simplex tableau, see Subsection 3.4.

Following the landmark paper [34], GMI cuts were empirically shown to be a computational success. This success was in spite of a commonly held belief that only cuts derived from MILP instance structure were computationally useful. Some examples of structured inequalities or cuts are knapsack cover and flow cover inequalities [35]. The summarised reasons for the success of [34] was their intelligent lifting procedure to globally valid cuts, their selection algorithm, their use of branch-and-cut as opposed to pure cutting plane approaches, and the recent robustness improvements of LP solvers. For a more complete history behind the resurgence of GMI inequalities, see [36]. Advances on GMI cuts have continued, where we name reduce-and-split cuts [30] and LaGromory cuts [37] as examples. To stress the importance of these cuts in the current day, we note that GMI cuts are continually noted as computationally necessary [38, 39], and are used in every state-of-the-art MILP solver, see Xpress [40], Gurobi [41], CPLEX [42], HiGHS [43], and SCIP [6].

3.2 Disjunctive, Split, and Intersection Cuts

It is common in the literature to find compact introductions of GMI cuts that mention they are either disjunctive cuts, intersection cuts, or split cuts. All these statements are true, and moreover, the families of cuts have a clear hierarchy [44, 45]:

$$\text{GMI cuts from basic feasible solutions} \subset \text{Split cuts} \subset \text{Intersection cuts} \subset \text{Disjunctive cuts}$$

We highlight that while our work on branching leverages GMI cuts, it can also leverage any family of cuts that fit into this hierarchy and can be derived from disjunctions.

3.2.1 Disjunctions and Disjunctive Cuts

A linear disjunction is a set of linear inequalities joined by *and*, *or*, and *negation* operators (see [44] for a thorough introduction). The solution set of a disjunction is a *disjunctive set*. For MILPs, every integer-feasible solution to (1) is an element of a disjunctive set. A *disjunctive cut* is any cut derived from such a disjunctive set, i.e., all elements in the disjunctive set remain feasible and some fractional solution outside the disjunctive set is separated.

A linear disjunctive set represents a union of polyhedra. It is defined as:

$$\mathcal{D} := \bigcup_{i=1}^{|\mathcal{D}|} \mathcal{D}_i, \quad \text{where;} \mathcal{D}_i \subseteq \mathcal{P} \quad \forall i \in \{1, \cdots |\mathcal{D}|\}$$

An example disjunction is visualised in Figure 2, with the Figure also showing a valid disjunctive cut.

Figure 2: (Left) An example disjunction $((x_1 \leq \lfloor x_1^{LP} \rfloor) \vee (x_1 \geq \lceil x_1^{LP} \rceil)) \wedge ((x_2 \leq \lfloor x_2^{LP} \rfloor) \vee (x_2 \geq \lceil x_2^{LP} \rceil))$. The disjunctive set is the union of blue polytopes. Here $\mathcal{D}_1$ is the empty set. (Right) An example disjunctive cut for the disjunction.

3.2.2 Splits and Split Cuts

A *split disjunction*, or *split*, is defined by an an integer $\pi_0 \in \mathbb{Z}$ and an integral vector $\boldsymbol{\pi} \in \mathbb{Z}^{|\mathcal{J}|} \times \mathbf{0}^{n-|\mathcal{J}|}$, which has zero entries for coefficients of continuous variables. We denote the split disjunction as $\mathcal{D}(\boldsymbol{\pi}, \pi_0)$, where $(\boldsymbol{\pi}, \pi_0)$ define the two hyperplanes:

$$\begin{aligned} \boldsymbol{\pi}^\intercal \mathbf{x} &\leq \pi_0 \\ \boldsymbol{\pi}^\intercal \mathbf{x} &\geq \pi_0 + 1 \end{aligned} \tag{2}$$

The disjunctive set $\mathcal{D} = \bigcup_{i \in \{1,2\}} \mathcal{D}_i$ formed by the hyperplanes is:

$$\begin{aligned} \mathcal{D}_1 &:= \mathcal{P} \cap \{\mathbf{x} \in \mathbb{R}^n | \boldsymbol{\pi}^\intercal \mathbf{x} \leq \pi_0\} \\ \mathcal{D}_2 &:= \mathcal{P} \cap \{\mathbf{x} \in \mathbb{R}^n | \boldsymbol{\pi}^\intercal \mathbf{x} \geq \pi_0 + 1\} \end{aligned}$$

The disjunction is valid as $\boldsymbol{\pi}^\intercal \mathbf{x}$ must always take an integer value in a feasible solution to (1) due to the design of $\boldsymbol{\pi}$. We observe that the disjunctive set $\mathcal{D}$ can be written as the complement of a set $\mathcal{S}$ intersected with $\mathcal{P}$, where $\mathcal{S}$ is defined as:

$$\mathcal{S} := \{\mathbf{x} \in \mathbb{R}^n \,|\, \pi_0 < \boldsymbol{\pi}^\intercal \mathbf{x} < \pi_0 + 1\} \tag{3}$$

Note that notation is often abused where the split can reference either the set $\mathcal{S}$ from (3) or the boundary of the set $\mathcal{S}$, i.e. the two hyperplanes from (2). From a split disjunction we can derive a *split cut*. A split cut, $(\boldsymbol{\alpha}, \beta)$, is a valid inequality for both $\mathcal{D}_1$ and $\mathcal{D}_2$, and separates some points from $\mathcal{S} \cap \mathcal{P}$. In the mixed-integer case, unlike the pure integer case [46], a finite amount of split cuts is not always sufficient for defining the integer hull and proving optimality, see [47]. A split is called *simple* or *elementary* if it only acts on a single variable, i.e. $\boldsymbol{\pi} = \mathbf{e}_i$ for some $i \in \mathcal{J}$. An example (simple) split disjunction alongside a valid split cut is visualised in Figure 3.

Figure 3: (Left) An example (simple) split. (Right) An example (simple) split cut.

3.2.3 Intersection Cuts

Some cuts reason on the standard form of a MILP, which is defined using equality constraints instead of inequalities. In particular, intersection cuts and GMI cuts are derived using this standard form. Given our

definition of a MILP in (1), we can transform it to a standard form MILP in higher dimension by adding non-negative slack variables to each constraint. We can additionally substitute and introduce variables to shift variable bounds while keeping an equivalent formulation. We do this procedure to obtain the following MILP, where for ease of notation we will continue to use $\mathbf{c}$ and $\mathbf{A}$.

$$\underset{\mathbf{x}}{\text{argmin}}\{\mathbf{c}^\intercal \mathbf{x} \;\mid\; \mathbf{A}\mathbf{x} = \mathbf{b},\;\; \mathbf{x} \geq \mathbf{0},\;\; \mathbf{x} \in \mathbb{Z}^{|\mathcal{J}|} \times \mathbb{R}^{n+m-|\mathcal{J}|}\} \tag{4}$$

The simplex method typically used to solve LP relaxations of (4) returns a *basis*, $\mathcal{B} \subseteq \{1, ..., n+m\}$, where $|\mathcal{B}| = m$. The basis is an index set of variables and relates to an extreme point of the LP relaxation, $\bar{\mathbf{x}} \in \mathbb{R}^{n+m}$, which is a basic solution. In practice the simplex method returns the optimal basic solution $\mathbf{x}^{LP}$. Associated with every basic solution $\bar{\mathbf{x}}$ is the LP cone, or corner polyehdron, $\mathcal{C}(\bar{\mathbf{x}}) \subseteq \mathbb{R}^{n+m}$, whose apex is $\bar{\mathbf{x}}$ and whose rays are defined by the n-hyperplanes that form the basis. These rays are the columns of the simplex tableau relating to the non-basic variables. Note that in the case of primal degeneracy, multiple bases may result in the same extreme point but in different LP cones, and as such $\mathcal{C}(\mathcal{B})$ is the more appropriate notation. Two example LP cones are visualised in Figure 4.

Figure 4: The shaded area is the feasible region of $\mathcal{C}(\bar{\mathbf{x}})$. The red dot is the apex of the simplicial conic relaxation of the feasible region of (1), and $\mathbf{r}_1, \mathbf{r}_2$ are the rays of the cone. (Left) The red dot is both $\mathbf{x}^{LP}$ and $\bar{\mathbf{x}}$. (Right) The red dot is a primal infeasible $\bar{\mathbf{x}}$.

An intersection cut, similar to a split cut, is defined w.r.t. a set $\mathcal{S} \subseteq \mathbb{R}^{n+m}$, which lies in the same dimension as $\mathbf{x}$ in the new space. Unlike split cuts however, $\mathcal{S}$ is not necessarily defined by two hyperplanes. Rather, it needs to be convex, to contain in its interior a current LP-feasible fractional solution we want to separate, and to not contain any integer-feasible solution in its interior. In the context of MILP, the set $\mathcal{S}$ is a lattice-free set [48]. In addition to the set $\mathcal{S}$, an intersection cut is also defined w.r.t. a simplicial conic relaxation of the feasible region of (1), see Figure 4 for example simplicial conic relaxations derived from bases. We note that while any simplicial conic relaxation can be exploited, an LP cone derived from a basis is used in practice. The idea behind intersection cuts is to collect the intersection points of each ray with the boundary of the closure of $\mathcal{S}$, and form a valid inequality as the hyperplane that contains all the intersection points. When using the LP cone $\mathcal{C}(\mathbf{x}^{LP})$ and a set $\mathcal{S}$ containing $\mathbf{x}^{LP}$, the generated inequality will be a cut. Two examples of intersection cuts are visualised in Figure 5. For a deeper look into intersection cuts, we refer readers to [48, 44].

Figure 5: (Left) Example intersection cut. (Right) Example intersection cut that is also a split cut.

3.3 GMI Inequality Derivation

We will now derive the GMI inequality, which we note again is general purpose and requires no additional problem structure.

Definition 1 (GMI inequality)**.** *Given a valid equality for the LP relaxation of* (4)*,* $\mathbf{a}^\intercal \mathbf{x} = b$*, we distinguish the variables into those with integer requirements and those that are continuous, i.e.,* $\sum_{i\in\mathcal{J}} a_i x_i + \sum_{i\in[n]\setminus\mathcal{J}} a_i x_i = b$*. Let* $[n] = \{1, ..., n\}$*,* $b = \lfloor b \rfloor + f_0$*, where* $0 < f_0 < 1$*, and* $a_i = \lfloor a_i \rfloor + f_i$*, where* $0 \le f_i < 1$ *and* $i \in [n]$*. The GMI inequality is:*

$$\sum_{i\in\mathcal{J}, f_i \le f_0} \frac{f_i}{f_0} x_i + \sum_{i\in\mathcal{J}, f_i > f_0} \frac{1-f_i}{1-f_0} x_i + \sum_{i\in[n]\setminus\mathcal{J}, a_i \ge 0} \frac{a_i}{f_0} x_i - \sum_{i\in[n]\setminus\mathcal{J}, a_i < 0} \frac{a_i}{1-f_0} x_i \ge 1 \tag{5}$$

Derivation. The logic of the GMI inequality is that if $f_0 > 0$, then fractional multiples of integer variables and multiples of continuous variables must account for f_0. Specifically, they must sum to f_0 and a potential integer. That is:

$$\sum_{i\in\mathcal{J}, f_i \le f_0} f_i x_i + \sum_{i\in\mathcal{J}, f_i > f_0} (f_i - 1) x_i + \sum_{i\in[n]\setminus\mathcal{J}} a_i x_i = k + f_0, \quad k \in \mathbb{Z} \tag{6}$$

This partition of f_i values around f_0 is possible due to the observation that a_i, where $f_i > 0$, can be equivalently written as $a_i = \lfloor a_i \rfloor + f_i$ or as $a_i = \lceil a_i \rceil + (f_i - 1)$. For example, $3.6 = 3 + 0.6$ or equivalently, $3.6 = 4 - 0.4$. This partition is done as it results in a strictly stronger cut than otherwise [32, 30, 29]. Specifically, it results in smaller coefficients for like terms as $\frac{1-f_i}{1-f_0} < \frac{f_i}{f_0}$ when $f_i > f_0$.

Let us create a disjunction for two cases for inequality (6), where $k \le -1$ or $k \ge 0$. In the case $k \le -1$ we have that:

$$\begin{aligned} &\sum_{i\in\mathcal{J}, f_i \le f_0} f_i x_i + \sum_{i\in\mathcal{J}, f_i > f_0} (f_i - 1) x_i + \sum_{i\in[n]\setminus\mathcal{J}} a_i x_i \le -(1 - f_0) \\ \Rightarrow &- \sum_{i\in\mathcal{J}, f_i \le f_0} \frac{f_i}{1-f_0} x_i + \sum_{i\in\mathcal{J}, f_i > f_0} \frac{1-f_i}{1-f_0} x_i - \sum_{i\in[n]\setminus\mathcal{J}} \frac{a_i}{1-f_0} x_i \ge 1 \end{aligned} \tag{7}$$

In the second case, $k \ge 0$ we have that:

$$\begin{aligned} &\sum_{i\in\mathcal{J}, f_i \le f_0} f_i x_i + \sum_{i\in\mathcal{J}, f_i > f_0} (f_i - 1) x_i + \sum_{i\in[n]\setminus\mathcal{J}} a_i x_i \ge f_0 \\ \Rightarrow &\sum_{i\in\mathcal{J}, f_i \le f_0} \frac{f_i}{f_0} x_i - \sum_{i\in\mathcal{J}, f_i > f_0} \frac{1-f_i}{f_0} x_i + \sum_{i\in[n]\setminus\mathcal{J}} \frac{a_i}{f_0} x_i \ge 1 \end{aligned} \tag{8}$$

As $\mathbf{x} \ge 0$, we can derive a globally valid inequality for the disjunctive set from the inequalities (7) - (8) by taking the maximum coefficient of each term over the two inequalities. That is the inequalities $\mathbf{a}^\intercal \mathbf{x} \ge 1$ and $\mathbf{a}'^\intercal \mathbf{x} \ge 1$ implies $\sum_{i=1}^{n} \max(a_i, a'_i) x_i \ge 1$. We have grouped the terms in their derivation above s.t. at most one is positive. The result of this derivation is exactly the GMI inequality (5). □

3.4 GMI Cuts in Practice

In general, it is $\mathcal{NP}$-hard to find a GMI cut that separates a given LP-*feasible* solution, or to determine if such a cut exists, see [49, 32]. It is not $\mathcal{NP}$-hard, however, to separate a given basic solution of $\mathcal{P}$, e.g., an LP-*optimal* solution found by a simplex algorithm.

Consider a row of the simplex tableau for variable x_j of basis $\mathcal{B}$. The row is an aggregated equality constraint, created from a linear combination of original constraints, where basic variable x_j is described purely in terms of the non-basic variables. That is:

$$x_j = \bar{x}_j - \sum_{i\notin\mathcal{B}} \bar{a}_{ji} x_i \tag{9}$$

Here $\bar{x}_j$ is the right hand side value of the tableau row and $\bar{a}_{ji}$ is the tableau entry for the row of basic variable x_j and column of variable x_i. In the considered case of $\mathbf{x} \ge \mathbf{0}$ the $\bar{x}_j$ is the value of variable x_j at the basic solution. Note that the constraint (9) is tight for the current basic solution.

A GMI cut is derived from applying the GMI inequality procedure from Subsection 3.3 to the aggregated equality constraint (9). This procedure is only applied to rows of the simplex tableau that correspond to integer variables with fractional LP solutions. This is because these rows have a fractional right hand side, and the resulting GMI inequality guarantees separation of the current LP solution. An inequality produced by this method is called a GMI cut.

Geometrically, a GMI cut is a split cut, and therefore it is also both an intersection cut and disjunctive cut. Specifically it is an intersection cut for the split $\mathcal{D}(\boldsymbol{\pi}^G, \lfloor \bar{x}_j \rfloor)$, where $\boldsymbol{\pi}^G$ is defined as follows:

$$\boldsymbol{\pi}^G \in \mathbb{Z}^n, \quad \text{where } \boldsymbol{\pi}^G_i := \begin{cases} \lfloor \bar{a}_{ji} \rfloor, & \text{if } (f_i \leq f_0) \ \wedge \ i \notin \mathcal{B} \\ \lceil \bar{a}_{ji} \rceil, & \text{if } (f_i > f_0) \ \wedge \ i \notin \mathcal{B} \\ 1, & \text{if } i = j \\ 0, & \text{if } (i \neq j) \wedge i \in \mathcal{B} \end{cases} \quad \forall i \in \{1, ..., n\} \tag{10}$$

The GMI cut of the tableau row (9) is the strengthened version of the intersection cut obtained from the elementary split $\mathcal{D}(\mathbf{e}_j, \lfloor \bar{x}_j \rfloor)$, see [30, 44]. Deriving a cut using the elementary split for the simplex tableau row (9) of variable x_j without the strengthening procedure results in the intersection cut (11). This cut is obtained by treating integer variables the same as continuous for the GMI derivation.

$$\sum_{i \notin \mathcal{B}, \bar{a}_{ji} \geq 0} \frac{\bar{a}_{ji}}{f_0} x_i - \sum_{i \notin \mathcal{B}, \bar{a}_{ji} < 0} \frac{\bar{a}_{ji}}{1 - f_0} x_i \geq 1 \tag{11}$$

We denote this inequality as *weak-GMI*, and note that the GMI cut will always dominate the associated weak-GMI cut. We also note that the strengthening procedure is performed by using fractional coefficient values f_i instead of $\bar{a}_{ij}$ for integer variables and the partitioning of those fractional coefficients f_i around f_0.

4 Cutting Plane Selection for Variable Selection

The core idea of our work is to use measures of cuts to evaluate and decide on corresponding branching candidates. Specifically, we will generate the GMI cut from the corresponding tableau row of each branching candidate, and use cut selection techniques to dictate branching decisions. We will additionally augment the default SCIP hybrid branching rule with history-based scores of already-computed GMI cuts from previous separation rounds.

Currently, history-based approaches, see [4, 5], are the backbone behind branching rules used in MILP solvers [6, 43]. *Pseudo-costs* [50], the most prolific case of history-based approaches, estimate scores for a branching candidate based on the historical objective value improvement of child nodes spawned from branching on the candidate. One can consider pseudo-costs as an approximation of *strong-branching* scores, see e.g. [4], which are derived from directly solving the upper and lower LP relaxations of all branching candidates. In our approach, branching scores are derived from cut quality measures of cuts generated from each branching candidate. It is complementary to pseudo-costs in that it provides an objective-free measure. These cut-based scores can be integrated into SCIP's default scoring rule, using a history of cut quality measures from previously generated cuts. This is similar to other history-based scores, such as those based on bound inferences, conflict information, and subproblem infeasibility [5].

The classical cut scoring measure is *efficacy*[4], which denotes the Euclidean distance between the LP optimal solution and the cut hyperplane. Given a cut $(\boldsymbol{\alpha}, \beta) \in \mathbb{R}^{n+1}$ and the LP optimal solution $\mathbf{x}^{LP}$, efficacy is defined as:

$$\texttt{eff}(\boldsymbol{\alpha}, \beta, \mathbf{x}^{LP}) := \frac{\boldsymbol{\alpha}^\intercal \mathbf{x}^{LP} - \beta}{\|\boldsymbol{\alpha}\|} \tag{12}$$

When scoring cuts for the purpose of branching, we will rely on efficacy as our cut measure. We note that there exists many more potential cut scoring measures [16, 22], however preliminary results of their inclusion led to negligible improvements.

GMI cuts are not the only cuts associated with split disjunctions, or even the elementary split, i.e. branching decisions. For example, lift-and-project cuts [51, 52] are intersection cuts of elementary splits. The elementary splits from which these cuts are derived, however, are not necessarily related to the current LP basis, nor even necessarily related to a primal-feasible LP basis. Nevertheless, scoring measures for this family of cuts are also a potentially potent indicator of good branching decisions. We however restricted our study to GMI cuts which are readily computed and available for all variables in all MILP solvers.

[4]Main selection criteria for most MILP solvers, e.g., FICO Xpress 9.0 and SCIP 8.0

5 Experiments

We conduct three experiments: First, we analyse the effectiveness of our initial approach compared to standard branching rules (Subsection 5.1). Then, we refine our approach to a history-based one, and determine the best parameter value for including our approach in the state-of-the-art branching rule *hybrid branching* (Subsection 5.2). Finally, we compare our integrated branching rule to default SCIP with experiments run in exclusive mode (Subsection 5.3). We perform experiments on the MIPLIB 2017 benchmark set[5] [53], which we will now simply refer to as MIPLIB. For all these experiments we present two variants: Firstly, we use default SCIP on the original instances to analyse the impact on the the out-of-the-box behaviour of a MIP solver. Secondly, we use SCIP with heuristics disabled and the optimal solution provided, which reduces random noise and emphasises the effect of branching rules.

We define a run as an instance random-seed pair for which we use a given branching rule. All results are obtained by averaging results over the SCIP random seeds $\{1, 2, 3, 4, 5\}$. For all experiments, SCIP 8.0.3 [6] is used, with PySCIPOpt [54] as the API, and Xpress 9.0.2 [40] as the LP solver. For Subsections 5.1 and 5.2, experiments are run in non-exclusive mode on a cluster equipped with Intel Xeon Gold 6342 CPUs running at 2.80GHz, where each run is restricted to 2GB memory, and the LP solver is restricted to a single thread. For Subsection 5.3 experiments are run in exclusive mode on a cluster equipped with Intel Xeon Gold 5122 CPUs running at 3.60GHz, where each run is restricted to 48GB memory, and the LP solver is restricted to a single thread. The code used for all experiments is available and open-source[6], and will be integrated in the next release of SCIP.

For the entirety of our experiments, we filter out any instance that for any random seed was solved to optimality without branching, hit a memory limit, or encountered LP errors. Note that the instance is only filtered in a comparison of branching rules when one of the criteria is met for a run on one of the compared branching rules. When comparing results from branching rules, we use individual instance-seed pairs as data points as opposed to the aggregate performance over the random seeds. Additionally, when shifted geometric means are referenced, we use a shift of 100, 10s, and 1s for number of nodes, solving time, and branching time respectively. We finally note that certain instances were excluded from the MIPLIB data set with an optimal solution due to the solutions being unavailable online.

5.1 Gomory Cut-Based Branching Rules

To rank the effectiveness of our GMI cut based branching rules, we compare them against standard branching rules from the literature, with Table 1 containing a complete list.

Branching Rule	Description
GMI	Generate GMI cuts from Tableau. Select candidate from cut with largest efficacy.
weak-GMI	Generate weak-GMI cuts from Tableau. Select candidate from cut with largest efficacy.
fullstrong	Solve LP relaxations of children nodes for all candidates, see [4].
hybrid	Reliability psuedo-cost / Hybrid. (Default SCIP scoring rule, see [4, 5])
random	Select random candidate.

Table 1: Branching rules used in Experiment 5.1

The shifted geometric means over three performance metrics on our data sets are presented in Table 2. We observe expected performance from the standard branching rules. *Fullstrong* requires the least nodes to prove optimality over all data sets, while *hybrid* is the branching rule that most quickly proves optimality. Our newly introduced branching rule *GMI*, is regrettably inferior to default SCIP over all metrics and data sets, however we observe that it clearly has a positive signal due to it requiring substantially less nodes than *random* to prove optimality over all data sets. Most interesting is the relative performance of *weak-GMI* to *GMI*, where *weak-GMI* wins over all metrics and data sets. This suggests that the strengthened cut, while strictly better than the weaker version in a cutting plane context, has lost some level of the representation of the branching decision that the weaker cut is derived from.

For running time, we must also address the overhead of our branching rule. While ultimately faster per node than strong branching, we still need to generate a GMI cut for every branching candidate at every node. This overhead is significant, and is the reason why *random* is on average faster to solve over MIPLIB both with and without a provided solution. This is despite requiring over twice as many nodes. This can be verified

[5] MIPLIB 2017 – The Mixed Integer Programming Library https://miplib.zib.de/.

[6] https://github.com/Opt-Mucca/branching-via-cut-selection

Metric	Pairs	*GMI*	*weak-GMI*	*fullstrong*	*hybrid*	*random*
MIPLIB						
Nodes	237	3877	3109	**512**	1562	8589
Time (s)	237	222	191	289	**101**	214
Time w/o branch time (s)	237	124	111	**64**	81	214
Branch time (s)	237	47	39	107	11	**0**
MIPLIB with optimal solution provided and heuristics disabled						
Nodes	227	3767	2812	**386**	1287	8286
Time (s)	227	179	138	126	**73**	168
Time w/o branch time (s)	227	97	82	**44**	59	168
Branch time (s)	227	46	32	39	8	**0**

(a) Instance-seed pairs where all branching rules solved to optimality.

Metric	Pairs	*GMI*	*weak-GMI*	*fullstrong*	*hybrid*	*random*
MIPLIB						
Time (s)	464	977	874	1200	**310**	858
Time w/o branch time (s)	464	362	337	**113**	260	858
Branch time (s)	464	340	298	689	26	**1**
MIPLIB with optimal solution provided and heuristics disabled						
Time (s)	440	1016	832	610	**269**	943
Time w/o branch time (s)	440	338	294	**85**	225	942
Branch time (s)	440	401	311	306	24	**1**

(b) Instance-seed pairs where at-least one branching rule solved to optimality.

Table 2: Shifted geometric mean results. Best branching rule per metric in **bold**.

by seeing that *GMI* and *weak-GMI* both are much faster than *random* when removing branching time from consideration.

5.2 History-Based GMI Branching

Our approaches *GMI* and *weak-GMI* were shown to make substantially better branching decisions than *random*, but were ultimately too slow, and were not as good as LP relaxation based branching rules. The default SCIP branching rule, while dominated by pseudo-costs, is a hybrid method, with scores from a weighted sum of metrics. Most of these metrics are history-based, meaning that they use information from different parts of the solving process, and are quick to evaluate. Given that GMI cuts are already generated by SCIP throughout the solve process, we can store for each variable, the average normalised efficacy of a GMI cut generated from a tableau row when the variable is basic and fractional. This normalised average can then be used to augment the branching candidate's score of default SCIP. We normalise efficacy by the maximum GMI cut's efficacy from the given separation round. We stress here that this approach requires no additional overhead, as the cuts themselves as well as their efficacies are already computed in the separation process.

We denote our new branching rule *gmi*-10^{-x}, where 10^{-x} denotes the coefficient used in the weighted sum scoring rule for average efficacy. The shifted geometric mean of performance metrics for various coefficient values are presented in Table 3. We observe that too high of a coefficient, as in *gmi*-10^{-2}, results in worse performance than default SCIP over all metrics and all data sets. By decreasing the coefficient value, we see an improvement in performance, with *gmi*-10^{-5} being the best performing rule w.r.t. both nodes and solve time over all data sets. We also observe that the branching rules on either side of *gmi*-10^{-5}, i.e. *gmi*-10^{-4} and *gmi*-10^{-6}, always outperform default SCIP, indicating that 10^{-5} is a sweet-spot. We therefore conclude that 10^{-5} is a good and robust coefficient choice for improving hybrid branching, i.e. default SCIP, once again noting that it requires no additional overhead since branching time is functionally identical.

We also performed preliminary experiments using the normalised efficacy of the most recently generated GMI cut, but quickly found that the approach was always outperformed by the historical average. In addition, we

Metric	Pairs	*hybrid*	*gmi*-10^{-2}	*gmi*-10^{-3}	*gmi*-10^{-4}	*gmi*-10^{-5}	*gmi*-10^{-6}
MIPLIB							
Nodes	472	4345	5025	4590	*4214*	***4101***	*4267*
Time (s)	472	253	276	253	*235*	***232***	*239*
Time w/o branch time (s)	472	208	228	209	*194*	***191***	*197*
Branch time (s)	472	24	26	24	***22***	*23*	*23*
MIPLIB with optimal solution provided and heuristics disabled							
Nodes	426	3870	4426	4023	*3822*	***3671***	*3820*
Time (s)	426	204	226	213	*203*	***195***	*199*
Time w/o branch time (s)	426	165	185	174	165	***158***	*161*
Branch time (s)	426	22	23	22	***21***	***21***	22

(a) Instance-seed pairs where all branching rules solved to optimality.

Metric	Pairs	*hybrid*	*gmi*-10^{-2}	*gmi*-10^{-3}	*gmi*-10^{-4}	*gmi*-10^{-5}	*gmi*-10^{-6}
MIPLIB							
Time (s)	548	370	402	378	*350*	***342***	*348*
Time w/o branch time (s)	548	308	336	315	*293*	***284***	*290*
Branch time (s)	548	33	34	*32*	***31***	***31***	***31***
MIPLIB with optimal solution provided and heuristics disabled							
Time (s)	507	310	357	333	314	***293***	*301*
Time w/o branch time (s)	507	254	295	275	259	***240***	*247*
Branch time (s)	507	31	34	32	***30***	***30***	31

(b) Instance-seed pairs where at-least one branching rule solved to optimality.

Table 3: Shifted geometric mean results. Branching rules better than default in *italics*, best in **bold**.

performed experiments using a new homogeneous MILP instance set, SNDlib-MIPs [55], which was inspired by SNDLib [56], but that the results while better than default SCIP, were only a marginal improvement. Up to this point, our runs were affected by our experimental setup, where memory limits reduced the size of instances that we considered, and the non-exclusive mode introduced additional noise w.r.t. solve time. We therefore perform a more in-depth comparison of *hybrid* and *gmi*-10^{-5} over MIPLIB in the following subsection.

5.3 Improving Default SCIP

Our in-depth comparison presented in Table 4 shows that our branching rule clearly outperforms default SCIP's hybrid branching. Over MIPLIB both with and without a solution provided, our augmented branching method results in faster solve times, and less nodes. We stress here that due to the size of the coefficient value for the average normalised GMI cut efficacy, our approach will often act more as a tie breaker rather than as a dominant decision maker.

The performance improvement of *gmi*-10^{-5} becomes even more apparent when we look only at affected instances, see Table 5. Over MIPLIB we achieve a 4% speedup on affected instances, and require 8% fewer nodes. This improvement becomes even more apparent when an optimal solution is provided. Our approach however is outperformed by default on unsolved instances. We do stress that upon further investigation, we found no evidence that *gmi*-10^{-5} is outperformed on larger or more complex instances that solved within the time limit. The large amount of affected instances indicates that the MILP solver ends up in situations where the scores of branching candidates, especially the pseudo-costs, are very similar.

Figure 6 shows the distribution of performance improvement per instance-seed pair. We observe a surprisingly diverse distribution, with the majority of instance-seed pairs either performing 10% better or 10% worse in both number of nodes and solve time. While many instances exhibit worse performance on *gmi*-10^{-5}, there are consistently more instance-seed pairs that perform correspondingly better than default SCIP over all levels of improvement. This is evident both for number of nodes and for solve time.

Metric	Pairs	*hybrid*	*gmi*-10^{-5}
MIPLIB			
Nodes	541	4385	**4150**
Time (s)	541	349	**339**
MIPLIB with optimal solution provided and heuristics disabled			
Nodes	491	4702	**4334**
Time (s)	491	325	**304**

(a) Instance-seed pairs where all branching rules solved to optimality.

Metric	Pairs	*hybrid*	*gmi*-10^{-5}
MIPLIB			
Time (s)	578	411	**398**
MIPLIB with optimal solution provided and heuristics disabled			
Time (s)	528	381	**361**

(b) Instance-seed pairs where at-least one branching rule solved to optimality.

Table 4: Shifted geometric mean results. Best branching rule per metric in **bold**.

MIPLIB 67.1% instance-seed pairs affected				
Time (s)	Nodes	$\Delta-$Solved	$\Delta-$Dual	$\Delta-$Primal
0.96	**0.92**	-1(/578)	-11(/472)	-4(/472)
MIPLIB with optimal solution provided and heuristics disabled 69.5% instance-seed pairs affected				
0.91	**0.89**	-1(/528)	-6(/307)	-

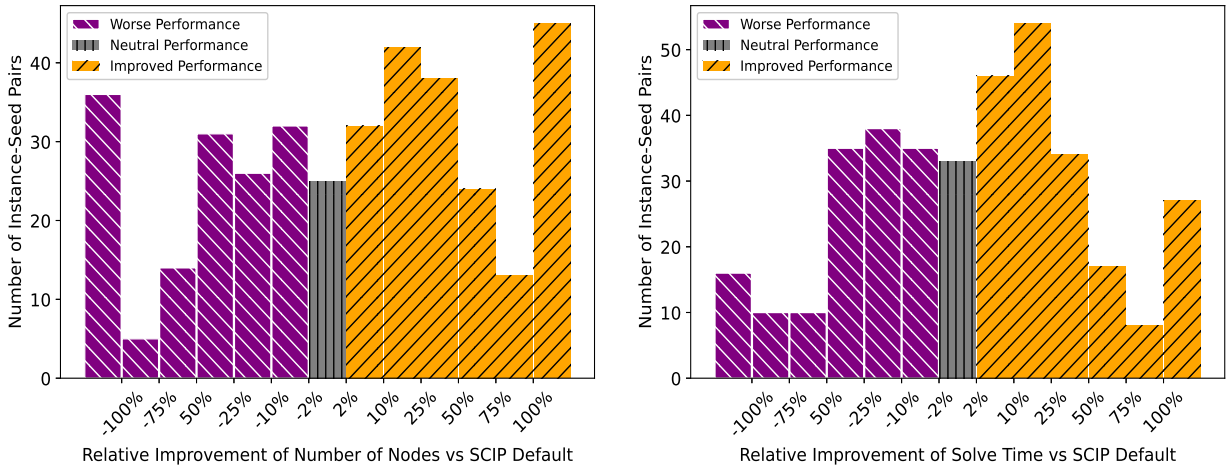
Table 5: Ratio of shifted geometric means for *gmi*-10^{-5} vs *hybrid* over affected instances (solved to optimality for both branching rules). Δ is the difference in wins over the instance-seed pairs for amount solved, and the respective bounds for unsolved instances. Entries are in bold when our approach is better than SCIP default.

6 Conclusion

In this paper, we developed a new branching rule based on the correspondence between Gomory mixed-integer cuts and split disjunctions, leveraging the efficacy of cutting planes as a measure for the relevance of a variable for branching. We used the branching rule with both unstrengthened and strengthened cuts, showing that the unstrengthened versions, which are directly derived from potential branching decisions, provide a better measure to reduce the number of nodes, and solve time. Our branching rule results in low numbers of nodes while being less expensive than strong branching. When integrated in the state-of-the-art hybrid branching algorithm of SCIP, the score provided by our branching rule reduces significantly both solve time and number of nodes over MIPLIB 2017. Future work includes extending our idea beyond Gomory mixed-integer cuts, to any cut that is linked to a split disjunction, e.g., lift-and-project cuts.

Acknowledgements

The work for this article has been conducted in the Research Campus MODAL funded by the German Federal Ministry of Education and Research (BMBF) (fund numbers 05M14ZAM, 05M20ZBM). The described research activities are funded by the Federal Ministry for Economic Affairs and Energy within the project UNSEEN (ID: 03EI1004-C).



(a) MIPLIB.

(b) MIPLIB with an optimal solution provided and heuristics disabled.

Figure 6: Bar plots of relative improvement of *gmi*-10^{-5} compared to default SCIP over affected instance-seed pairs. (Left) Number of nodes. (Right) Solve time.

References

[1] Tobias Achterberg. *Constraint integer programming*. PhD thesis, TU Berlin, 2007.

[2] Marc E Pfetsch and Thomas Rehn. A computational comparison of symmetry handling methods for mixed integer programs. *Mathematical Programming Computation*, 11:37–93, 2019.

[3] Christopher Hojny and Marc E Pfetsch. Polytopes associated with symmetry handling. *Mathematical Programming*, 175(1-2):197–240, 2019.

[4] Tobias Achterberg, Thorsten Koch, and Alexander Martin. Branching rules revisited. *Operations Research Letters*, 33(1):42–54, 2005.

[5] Tobias Achterberg and Timo Berthold. Hybrid branching. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 6th International Conference, CPAIOR 2009 Pittsburgh, PA, USA, May 27-31, 2009 Proceedings 6*, pages 309–311. Springer, 2009.

[6] Ksenia Bestuzheva, Mathieu Besançon, Wei-Kun Chen, Antonia Chmiela, Tim Donkiewicz, Jasper van Doornmalen, Leon Eifler, Oliver Gaul, Gerald Gamrath, Ambros Gleixner, Leona Gottwald, Christoph Graczyk, Katrin Halbig, Alexander Hoen, Christopher Hojny, Rolf van der Hulst, Thorsten Koch, Marco Lübbecke, Stephen J. Maher, Frederic Matter, Erik Mühmer, Benjamin Müller, Marc E. Pfetsch, Daniel

Rehfeldt, Steffan Schlein, Franziska Schlösser, Felipe Serrano, Yuji Shinano, Boro Sofranac, Mark Turner, Stefan Vigerske, Fabian Wegscheider, Philipp Wellner, Dieter Weninger, and Jakob Witzig. Enabling Research through the SCIP Optimization Suite 8.0. *ACM Transactions on Mathematical Software*, 2023.

[7] Jeff T Linderoth and Martin WP Savelsbergh. A computational study of search strategies for mixed integer programming. *INFORMS Journal on Computing*, 11(2):173–187, 1999.

[8] Santanu S Dey, Yatharth Dubey, Marco Molinaro, and Prachi Shah. A theoretical and computational analysis of full strong-branching. *arXiv preprint arXiv:2110.10754*, 2021.

[9] Matteo Fischetti and Michele Monaci. Branching on nonchimerical fractionalities. *Operations Research Letters*, 40(3):159–164, 2012.

[10] Timo Berthold and Domenico Salvagnin. Cloud branching. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems: 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings 10*, pages 28–43. Springer, 2013.

[11] Jonathan H Owen and Sanjay Mehrotra. Experimental results on using general disjunctions in branch-and-bound for general-integer linear programs. *Computational optimization and applications*, 20(2):159–170, 2001.

[12] Elias Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[13] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *arXiv preprint arXiv:1906.01629*, 2019.

[14] Lingying Huang, Xiaomeng Chen, Wei Huo, Jiazheng Wang, Fan Zhang, Bo Bai, and Ling Shi. Branch and bound in mixed integer linear programming problems: A survey of techniques and trends. *arXiv preprint arXiv:2111.06257*, 2021.

[15] Arnaud Deza and Elias B Khalil. Machine learning for cutting planes in integer programming: A survey. *arXiv preprint arXiv:2302.09166*, 2023.

[16] Franz Wesselmann and U Stuhl. Implementing cutting plane management and selection techniques. Technical report, Technical report, University of Paderborn, 2012.

[17] Giuseppe Andreello, Alberto Caprara, and Matteo Fischetti. Embedding $\{0, 1/2\}$-cuts in a branch-and-cut framework: A computational study. *INFORMS Journal on Computing*, 19(2):229–238, 2007.

[18] Santanu S Dey and Marco Molinaro. Theoretical challenges towards cutting-plane selection. *Mathematical Programming*, 170:237–266, 2018.

[19] Maria-Florina F Balcan, Siddharth Prasad, Tuomas Sandholm, and Ellen Vitercik. Sample complexity of tree search configuration: Cutting planes and beyond. *Advances in Neural Information Processing Systems*, 34, 2021.

[20] Maria-Florina Balcan, Siddharth Prasad, Tuomas Sandholm, and Ellen Vitercik. Improved sample complexity bounds for branch-and-cut. In *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, 2022.

[21] Mark Turner, Thorsten Koch, Felipe Serrano, and Michael Winkler. Adaptive Cut Selection in Mixed-Integer Linear Programming. *arXiv preprint arXiv:2202.10962*, 2022.

[22] Mark Turner, Timo Berthold, Mathieu Besançon, and Thorsten Koch. Cutting plane selection with analytic centers and multiregression. *arXiv preprint arXiv:2212.07231*, 2022.

[23] Zhihai Wang, Xijun Li, Jie Wang, Yufei Kuang, Mingxuan Yuan, Jia Zeng, Yongdong Zhang, and Feng Wu. Learning cut selection for mixed-integer linear programming via hierarchical sequence model. *arXiv preprint arXiv:2302.00244*, 2023.

[24] Radu Baltean-Lugojan, Pierre Bonami, Ruth Misener, and Andrea Tramontani. Scoring positive semidefinite cutting planes for quadratic optimization via trained neural networks. *optimization-online preprint 2018/11/6943*, 2019.

[25] Zeren Huang, Kerong Wang, Furui Liu, Hui-ling Zhen, Weinan Zhang, Mingxuan Yuan, Jianye Hao, Yong Yu, and Jun Wang. Learning to select cuts for efficient mixed-integer programming. *arXiv preprint arXiv:2105.13645*, 2021.

[26] Max B Paulus, Giulia Zarpellon, Andreas Krause, Laurent Charlin, and Chris Maddison. Learning to cut by looking ahead: Cutting plane selection via imitation learning. In *International Conference on Machine Learning*, pages 17584–17600. PMLR, 2022.

[27] Yunhao Tang, Shipra Agrawal, and Yuri Faenza. Reinforcement learning for integer programming: Learning to cut. In *International Conference on Machine Learning*, pages 9367–9376. PMLR, 2020.

[28] Miroslav Karamanov and Gérard Cornuéjols. Branching on general disjunctions. *Mathematical Programming*, 128(1-2):403–436, 2011.

[29] Gerard Cornuéjols, Leo Liberti, and Giacomo Nannicini. Improved strategies for branching on general disjunctions. *Mathematical Programming*, 130(2):225–247, 2011.

[30] Kent Andersen, Gérard Cornuéjols, and Yanjun Li. Reduce-and-split cuts: Improving the performance of mixed-integer gomory cuts. *Management Science*, 51(11):1720–1732, 2005.

[31] Markus Jörg. *k-disjunctive cuts and cutting plane algorithms for general mixed integer linear programs*. PhD thesis, Technische Universität München, 2008.

[32] Gérard Cornuéjols. Valid inequalities for mixed integer linear programs. *Mathematical programming*, 112(1):3–44, 2008.

[33] Ralph Gomory. An algorithm for the mixed integer problem. Technical report, RAND CORP SANTA MONICA CA, 1960.

[34] Egon Balas, Sebastian Ceria, Gérard Cornuéjols, and N Natraj. Gomory cuts revisited. *Operations Research Letters*, 19(1):1–9, 1996.

[35] Michele Conforti, Gérard Cornuéjols, Giacomo Zambelli, et al. *Integer programming*, volume 271. Springer, 2014.

[36] Gérard Cornuéjols et al. Revival of the gomory cuts in the 1990's. *Annals of Operations Research*, 149(1):63–66, 2007.

[37] Matteo Fischetti and Domenico Salvagnin. A relax-and-cut framework for gomory mixed-integer cuts. *Mathematical Programming Computation*, 3(2):79–102, 2011.

[38] Franz Wesselmann. Strengthening gomory mixed-integer cuts: a computational study. *University of Paderborn*, 2009.

[39] Tobias Achterberg and Roland Wunderling. Mixed integer programming: Analyzing 12 years of progress. *Facets of Combinatorial Optimization: Festschrift for Martin Grötschel*, pages 449–481, 2013.

[40] FICO Xpress Optimization. https://www.fico.com/en/products/fico-xpress-optimization. Accessed: 31-01-2023.

[41] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2023.

[42] IBM ILOG Cplex. V12. 1: User's manual for CPLEX. *International Business Machines Corporation*, 46(53):157, 2009.

[43] Qi Huangfu and JA Julian Hall. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*, 10(1):119–142, 2018.

[44] Egon Balas. *Disjunctive programming*. Springer, 2018.

[45] Kent Andersen, Gérard Cornuéjols, and Yanjun Li. Split closure and intersection cuts. *Mathematical programming*, 102(3):457–493, 2005.

[46] Alexander Schrijver et al. On cutting planes. *Combinatorics*, 79:291–296, 1980.

[47] William Cook, Ravindran Kannan, and Alexander Schrijver. Chvátal closures for mixed integer programming problems. *Mathematical Programming*, 47(1-3):155–174, 1990.

[48] Michele Conforti, Gérard Cornuéjols, and Giacomo Zambelli. Corner polyhedron and intersection cuts. *Surveys in operations research and management science*, 16(2):105–120, 2011.

[49] Alberto Caprara and Adam N Letchford. On the separation of split cuts and related inequalities. *Mathematical Programming*, 94:279–294, 2003.

[50] Michel Bénichou, Jean-Michel Gauthier, Paul Girodet, Gerard Hentges, Gerard Ribière, and Olivier Vincent. Experiments in mixed-integer linear programming. *Mathematical Programming*, 1:76–94, 1971.

[51] Egon Balas. Disjunctive programming. *Annals of discrete mathematics*, 5:3–51, 1979.

[52] Michael Perregaard. Generating disjunctive cuts for mixed integer programs. *Mellon University*, 2003.

[53] Ambros Gleixner, Gregor Hendel, Gerald Gamrath, Tobias Achterberg, Michael Bastubbe, Timo Berthold, Philipp Christophel, Kati Jarck, Thorsten Koch, Jeff Linderoth, et al. MIPLIB 2017: data-driven compilation of the 6th mixed-integer programming library. *Mathematical Programming Computation*, pages 1–48, 2021.

[54] Stephen Maher, Matthias Miltenberger, Joao Pedro Pedroso, Daniel Rehfeldt, Robert Schwarz, and Felipe Serrano. PySCIPOpt: Mathematical programming in python with the SCIP optimization suite. In *International Congress on Mathematical Software*, pages 301–307. Springer, 2016.

[55] Mark Turner, Timo Berthold, Mathieu Besançon, and Thorsten Koch. SNDlib-MIPs: A new set of homogeneous MILP instances. https://doi.org/10.5281/zenodo.8021237, June 2023.

[56] Sebastian Orlowski, Roland Wessäly, Michal Pióro, and Artur Tomaszewski. Sndlib 1.0—survivable network design library. *Networks: An International Journal*, 55(3):276–286, 2010.