

Two-row and two-column mixed-integer presolve using hashing-based pairing methods

Patrick Gemander^{*1} , Wei-Kun Chen² , Dieter Weninger¹ ,
Leona Gottwald³ , Ambros Gleixner³ , Alexander Martin¹ 

July 1st, 2020

Abstract In state-of-the-art mixed-integer programming solvers, a large array of reduction techniques are applied to simplify the problem and strengthen the model formulation before starting the actual branch-and-cut phase. Despite their mathematical simplicity, these methods can have significant impact on the solvability of a given problem. However, a crucial property for employing presolve techniques successfully is their speed. Hence, most methods inspect constraints or variables individually in order to guarantee linear complexity. In this paper, we present new hashing-based pairing mechanisms that help to overcome known performance limitations of more powerful presolve techniques that consider pairs of rows or columns. Additionally, we develop an enhancement to one of these presolve techniques by exploiting the presence of set-packing structures on binary variables in order to strengthen the resulting reductions without increasing runtime. We analyze the impact of these methods on the MIPLIB 2017 benchmark set based on an implementation in the MIP solver SCIP.

Keywords Linear programming · mixed-integer linear programming · optimization solver · presolve

Mathematics Subject Classification 90C05 · 90C10 · 90C11 · 65Y05

1 Introduction

Presolve for mixed-integer programming (MIP) is a set of routines that remove redundant information and strengthen the model formulation with the aim of accelerating a subsequent main solution process, which is usually a branch-and-cut approach. Further, presolve is an excellent complement to branch-and-cut, because it focuses on model reductions and reformulations that are commonly not in the working space of branch-and-cut, e.g., greatest common divisor reductions or redundancy detection. Presolve can be very effective and it frequently makes the difference between a problem being intractable and solvable [2].

Considering that presolve is undisputedly an important building block in solving MIP problems, there exist comparatively few articles in the MIP literature. Only in the last few years there has been a moderate increase in the number of publications on

¹*Friedrich-Alexander Universität Erlangen-Nürnberg, Department Mathematik, Cauerstr. 11, 91058 Erlangen, Germany, {patrick.gemander, alexander.martin, dieter.weninger}@fau.de*

²*School of Mathematics and Statistics, Beijing Institute of Technology, Beijing 100081, China, chenweikun@bit.edu.cn*

³*Zuse Institute Berlin, Department of Mathematical Optimization, Takustr. 7, 14195 Berlin, Germany, {gleixner, gottwald}@zib.de*

this subject. One of the earliest contributions was by Brearly et al. [13]. Later Johnson and Suhl [26], Guignard and Spielberg [24], Crowder et al. [15], and Hoffman and Padberg [25] investigated presolve techniques for zero-one inequalities. Savelbergh [35] published preprocessing and probing techniques for MIP problems. Many of the methods published in it are now standard in almost every MIP solver. Presolve also plays an important role in linear programming [5] and especially for interior point algorithms [23]. In recent years, presolve as a research topic has become increasingly important [19, 3]. Finally, the very recent publication by Achterberg et al. [4] deserves special attention. Not only does it present many new presolve methods, but also highlights theoretically interesting details of presolve reductions and relations to number theory.

How the presolve algorithms are implemented frequently makes the difference between a beneficial and a harmful method. Often it is reasonable to weight the strength of the reductions against the runtime behavior in order to rank the presolve methods appropriately. Details on implementation of various presolve reductions are discussed in Suhl and Szymanski [40], Martin [32], Atamtürk and Savelsbergh [6], and Achterberg [1]. Investigations concerning the performance impact of different features of a MIP solver were published in Bixby et al. [12], Bixby and Rothberg [11], and Achterberg and Wunderling [2]. These publications confirmed that presolve and cutting plane techniques are by far the most important components of modern MIP solvers.

In the most simple and usually fastest case, individual rows or columns are considered for presolve reductions. Some interactions between reductions may take place through e.g. tightened variables bounds, but strictly speaking this approach is limited to local information and thus often yields weak reductions. On the other hand one could try to find stronger reductions by building global data structures such as for example the clique table, which collects mutual incompatibilities between binary variables, or by considering larger parts of the problem or even the whole problem, which generally leads to higher runtimes.

It is therefore important to strike a good balance between effectiveness and computational overhead. One way to achieve this would be to examine two rows or two columns simultaneously. Bixby and Wagner [10] published an efficient algorithm to find parallel rows or columns of the constraint matrix. Multi-row presolve reductions were also the subject of research in Achterberg et al. [3], where they derive improvements to the model formulation by simultaneously examining multiple problem constraints. Finally, considering more than one row or column at a time allows more techniques to be used. For instance, matrix sparsification techniques as described in Chang and McCormick [14], Gondzio [23] and, more recently, Achterberg et al. [4], cannot be applied when considering only one row or one column at a time.

In many of the publications mentioned above, amazing results have been achieved with presolve reductions for solving MIPs. However, these results often are closely related to the exact details of the underlying implementation. Especially two-row or two-column techniques immediately raise the question of how to find promising pairs of rows or columns as simple enumeration of all combinations is usually too computationally expensive in a presolve context. In most publications there is no answer to this question. Therefore, in this article we develop two efficient methods for determining suitable pairs for presolve reductions via hashing methods [27]. In addition, we describe new presolve approaches or extensions of existing techniques that, to the best of our knowledge, have not been published yet.

The paper is organized as follows. Section 2 introduces the notation and some basic concepts used in the remainder of the paper. Sections 3-5 constitute the main part of the paper. Five new presolve approaches or extensions of already published methods are described in Section 3. The first two methods deal with sparsification of the constraint matrix, two more methods focus on bound tightening, and the method presented last is a dual approach resulting in variable and constraint fixings. Next, in Section 4, we describe

the hashing methods we use to address the question of finding promising pairs of two rows or columns of the problem for the previously presented presolve methods. Subsequently, Section 5 provides computational results to assess their performance impact. Finally, we summarize our conclusions in Section 6.

2 Notation and Background

This document builds on the following notation. Let $\mathcal{N} = \{1, \dots, n\}$ and $\mathcal{M} = \{1, \dots, m\}$ be index sets. Given a matrix $A \in \mathbb{R}^{m \times n}$, vectors $c \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, $\ell \in (\mathbb{R} \cup \{-\infty\})^n$, $u \in (\mathbb{R} \cup \{\infty\})^n$, variables $x \in \mathbb{R}^n$ with $x_j \in \mathbb{Z}$ for $j \in \mathcal{I} \subseteq \mathcal{N}$, and relations $\circ_i \in \{=, \leq, \geq\}$ for all $i \in \mathcal{M}$ of A , a *mixed-integer program* (MIP) can be formulated in the following form:

$$\begin{aligned} \min \quad & c^\top x \\ \text{s.t.} \quad & Ax \circ b, \\ & \ell_j \leq x_j \leq u_j \quad \text{for all } j \in \mathcal{N}, \\ & x_j \in \mathbb{Z} \quad \text{for all } j \in \mathcal{I}. \end{aligned} \tag{1}$$

The set of feasible solutions of (1) is defined by

$$P_{\text{MIP}} := \{x \in \mathbb{R}^n \mid x_j \in \mathbb{Z} \forall j \in \mathcal{I} \wedge Ax \circ b \wedge \ell_j \leq x_j \leq u_j \forall j \in \mathcal{N}\}.$$

In addition, we define the set of feasible solutions of the linear programming relaxation of (1) by

$$P_{\text{LP}} := \{x \in \mathbb{R}^n \mid Ax \circ b \wedge \ell_j \leq x_j \leq u_j \forall j \in \mathcal{N}\}.$$

The single coefficients of A are denoted by a_{ij} with $i \in \mathcal{M}$ and $j \in \mathcal{N}$. We use the notation A_i to identify the row vector given by the i -th row of A . Similarly, A_j is the column vector given by the j -th column of A . Let $R \subseteq \mathcal{M}$ and $V \subseteq \mathcal{N}$, then A_{RV} denotes a matrix consisting of the coefficients of A restricted to the row indices $i \in R$ and column indices $j \in V$. Similarly, x_V denotes the vector x limited to the index set V .

In some cases, we consult the *support function* which determines the indices of the nonzero entries of a vector. With $\text{supp}(A_i) = \{j \in \mathcal{N} \mid a_{ij} \neq 0\}$ we denote the support of row A_i . Correspondingly, $\text{supp}(A_j) = \{i \in \mathcal{M} \mid a_{ij} \neq 0\}$ designates the support of column A_j .

Depending on the coefficients of the matrix A and the bounds of the variables x , the *minimal activity* and *maximal activity* for each row $i \in \mathcal{M}$ in (1) are given by

$$\inf\{A_i x \mid \ell_j \leq x_j \leq u_j \forall j \in \mathcal{N}\} = \sum_{\substack{j \in \mathcal{N} \\ a_{ij} > 0}} a_{ij} \ell_j + \sum_{\substack{j \in \mathcal{N} \\ a_{ij} < 0}} a_{ij} u_j \tag{2}$$

and

$$\sup\{A_i x \mid \ell_j \leq x_j \leq u_j \forall j \in \mathcal{N}\} = \sum_{\substack{j \in \mathcal{N} \\ a_{ij} > 0}} a_{ij} u_j + \sum_{\substack{j \in \mathcal{N} \\ a_{ij} < 0}} a_{ij} \ell_j, \tag{3}$$

respectively. As infinite lower or upper bounds on the variables x are allowed, infinite minimal or maximal activities may occur. To simplify notation we write

$$\begin{aligned} \inf(A_i x_V) &:= \inf\{A_i x_V \mid \ell_j \leq x_j \leq u_j \forall j \in V\} \\ \sup(A_i x_V) &:= \sup\{A_i x_V \mid \ell_j \leq x_j \leq u_j \forall j \in V\} \end{aligned}$$

for row i and a subset $V \subseteq \mathcal{N}$ of variable indices, in order to refer to partial minimal and maximal activities. This coincides with (2) and (3) for $V = \mathcal{N}$.

Consider an inequality constraint

$$A_{iV}x_V + a_{ij}x_j \geq b_i$$

where $i \in \mathcal{M}$, $j \in \mathcal{N}$, $V = \text{supp}(A_i) \setminus \{j\}$, and $a_{ij} \neq 0$. Before deriving bounds for x_j an important question is how to determine bounds $\ell_i^V, u_i^V \in \mathbb{R} \cup \{-\infty, \infty\}$ with reasonable computational effort such that

$$\ell_i^V \leq A_{iV}x_V \leq u_i^V$$

holds for all $x \in P_{\text{MIP}}$. There are various possibilities in order to achieve this and the most simple approach for single-row presolve is to optimize in linear time over the bounding box, i.e.

$$\ell_i^V := \inf(A_{iV}x_V), \quad u_i^V := \sup(A_{iV}x_V). \quad (4)$$

If u_i^V is finite, we can determine bounds on variable x_j . Depending on the sign of a_{ij} we distinguish two cases. For $a_{ij} > 0$ a valid lower bound on x_j is given by

$$x_j \geq (b_i - u_i^V)/a_{ij} \quad (5)$$

and hence, we can replace ℓ_j by

$$\max\{\ell_j, (b_i - u_i^V)/a_{ij}\}. \quad (6)$$

Analogously, for $a_{ij} < 0$, a valid upper bound on x_j is given by

$$x_j \leq (b_i - u_i^V)/a_{ij} \quad (7)$$

and hence, we can replace u_j by

$$\min\{u_j, (b_i - u_i^V)/a_{ij}\}. \quad (8)$$

If x_j is an integer variable we replace (6) by

$$\lceil \max\{\ell_j, (b_i - u_i^V)/a_{ij}\} \rceil \quad (9)$$

and analogously (8) by

$$\lfloor \min\{u_j, (b_i - u_i^V)/a_{ij}\} \rfloor. \quad (10)$$

The observations above together with (4) can be used to realize an approach called *feasibility-based bound tightening* (FBBT), which is an iterative range-reduction technique [18]. It involves tightening the variable ranges using all of the constraint restrictions. This procedure is iterated as long as the variable ranges keep changing. FBBT might fail to converge finitely (see Achterberg et al. [4]). Therefore, in practice one stops iterating when the improvement after one or several iterations is too small, an alternate approach to deal with this special case is described in [9].

Instead of trivially optimizing over the bounding box one could calculate

$$\begin{aligned} \ell_i^V &= \min\{A_{iV}x_V \mid x \in P_{\text{MIP}}\}, \\ u_i^V &= \max\{A_{iV}x_V \mid x \in P_{\text{MIP}}\}. \end{aligned}$$

However, this approach would usually be very time consuming as it amounts to solving problem (1) with a different objective function. A lightweight alternative to solving the complete MIPs is to only consider the linear programming relaxation

$$\begin{aligned} \ell_i^V &= \min\{A_{iV}x_V \mid x \in P_{\text{LP}}\}, \\ u_i^V &= \max\{A_{iV}x_V \mid x \in P_{\text{LP}}\}. \end{aligned}$$

Still, this procedure is often too time-consuming.

A more reasonable compromise between computational complexity and tightness of the bounds is to use only a small number of constraints of the system instead of all constraints over which we maximize or minimize. This can again be done via solving a MIP or an LP. If we just use a single constraint, then solving the corresponding LP is of particular interest, since such problems can be solved in linear time in the number of variables, see Dantzig [17] and Balas and Zemel [7].

A closely related procedure is the so-called *optimization based bound tightening* (OBBT), which first has been applied in the context of global optimization by Shechtman and Sahinidis [38] and Quesada and Grossmann [34]. Here, the block we want to minimize and maximize consists of just one variable:

$$\min\{x_j \mid x \in P_{\text{MIP}}\} \leq x_j \leq \max\{x_j \mid x \in P_{\text{MIP}}\}.$$

Again, instead of solving the full MIP one can solve any relaxation of the problem to obtain valid bounds for x_j .

Presolve methods can be classified into two groups: *primal* and *dual* presolve methods. Primal presolve methods derive reductions based on the feasibility argument and thus are valid independently of the objective function. In contrast, dual presolve methods derive reductions by utilizing the information from the objective function while ensuring at least one optimal solution is retained, as long as the problem was feasible.

3 Two-Row and Two-Column Presolve Methods

In this section we will outline five presolve methods for which we developed the pairing mechanisms following in Section 4. First, Section 3.1 presents a primal presolve method to increase sparsity of the constraint matrix by cancelling nonzero coefficients using pairs of rows and the method in Section 3.2 extends this idea onto using columns for nonzero cancellation. The next two primal presolve methods, outlined in Section 3.3 and Section 3.4, improve variable bounds via FBBT on pairs of rows. Although these two methods work in a similar way they have different advantages. In short, the method in Section 3.3 is easier to implement and can be applied to single variables but cannot tighten bounds on all variables present in the involved constraints. Finally, Section 3.5 presents a way to reuse the method presented in Section 3.4 in order to improve a dual presolve method.

3.1 Two-Row Nonzero Cancellation

This presolve method tries to find pairs consisting of an equality and a second constraint such that adding the appropriately scaled equality to the other constraint sparsifies the latter by cancelling nonzero coefficients. This approach to reduce the number of nonzeros in the constraint matrix A has already been used by Chang and McCormick [14], Gondzio [23] and, more recently, Achterberg et. al. [4]. More precisely, assume two constraints

$$\begin{array}{rcccc} A_{iU}x_U & + & A_{iV}x_V & + & A_{iW}x_W & & = & b_i, \\ A_{rU}x_U & + & A_{rV}x_V & + & & A_{rY}x_Y & \circ_r & b_r, \end{array}$$

with $i, r \in \mathcal{M}$ and disjoint subsets of the column indices $U, V, W, Y \subseteq \mathcal{N}$. Further, assume there exists a scalar $\lambda \in \mathbb{R}$ such that $A_{rU} - \lambda A_{iU} = 0$ and $A_{rk} - \lambda A_{ik} \neq 0$ for all $k \in V$. Subtracting λ times the equality i from constraint r yields

$$\begin{array}{rcccc} A_{iU}x_U & + & & & & & = & b_i, \\ (A_{rV} - \lambda A_{iV})x_V & - & \lambda A_{iW}x_W & + & A_{rY}x_Y & \circ_r & b_r - \lambda b_i. \end{array} \quad (11)$$

The difference in the number of nonzeros of A is $|U| - |W|$. Since the case $|U| - |W| \leq 0$ does not seem to offer any advantage, the reduction is applied only if the number of nonzeros actually decreases. In all cases, this procedure takes at most $O(|\mathcal{N}|)$ time.

For mixed-integer programming, reducing the number of nonzeros of A has two main advantages. First, many subroutines in a MIP solver depend on this number. Especially the LP solver benefits from sparse basis matrices. Secondly, nonzero cancellation may open up possibilities for other presolve techniques to perform useful reductions or improvements on the formulation. One special case occurs if $W = \emptyset$, that is, the support of the equality constraint is a subset of the support of the other constraint. This case is of particular interest because decompositions may take place.

3.2 Two-Column Nonzero Cancellation

This presolve method extends the idea of the previous presolve method onto columns, i.e. we now aim for nonzero cancellations based on two columns of the constraint matrix A . To be more precise, consider the sum of two columns of problem (1)

$$\begin{bmatrix} A_{Uj} \\ A_{Vj} \\ A_{Wj} \end{bmatrix} x_j + \begin{bmatrix} A_{Uk} \\ A_{Vk} \\ A_{Yk} \end{bmatrix} x_k, \quad (12)$$

where $j, k \in \mathcal{N}$ and $U, V, W, Y \subseteq \mathcal{M}$ are disjoint subsets of the row indices. We first discuss the case where x_j is a continuous variable. Suppose there exists a scalar $\lambda \in \mathbb{R}$ such that $\lambda A_{Uj} - A_{Uk} = 0$, $\lambda A_{Vj} - A_{Vk} \neq 0$. By rewriting (12) as

$$\begin{bmatrix} A_{Uj} \\ A_{Vj} \\ A_{Wj} \end{bmatrix} (x_j + \lambda x_k) + \begin{bmatrix} A_{Vk} - \lambda A_{Vj} \\ -\lambda A_{Wj} \\ A_{Yk} \end{bmatrix} x_k,$$

and introducing a new variable $z := x_j + \lambda x_k$, we obtain

$$\begin{bmatrix} A_{Uj} \\ A_{Vj} \\ A_{Wj} \end{bmatrix} z + \begin{bmatrix} A_{Vk} - \lambda A_{Vj} \\ -\lambda A_{Wj} \\ A_{Yk} \end{bmatrix} x_k. \quad (13)$$

From the definition of z , it follows that the lower and upper bounds of z are

$$\ell_z = \begin{cases} \ell_j + \lambda \ell_k, & \text{for } \lambda > 0, \\ \ell_j + \lambda u_k, & \text{for } \lambda < 0, \end{cases} \quad \text{and} \quad u_z = \begin{cases} u_j + \lambda u_k, & \text{for } \lambda > 0, \\ u_j + \lambda \ell_k, & \text{for } \lambda < 0, \end{cases}$$

respectively. However, in order to keep the bounds of $\ell_j \leq x_j \leq u_j$, the constraint

$$\ell_j \leq z - \lambda x_k \leq u_j \quad (14)$$

needs to explicitly be added to problem (1). As is the case for the sparsification method from Section 3.1, the new representation consisting of (13) is of interest when $|U| - |W| > 0$ holds. One difference to the row-based version is that the reformulation contains the additional overhead of adding the constraint (14) to the problem (1). However, if x_j is a *free variable*, i.e. $\ell_j = -\infty$ and $u_j = \infty$ hold, the constraint (14) is redundant and does not need to be added to the problem (1). When the bounds of x_j are implied by the constraints and the bounds of other variables, x_j can also be treated as a free variable. In this case one speaks of an *implicit free variable*.

We now consider the case where x_j is an integer variable. To maintain integrality, we require that x_k is also an integer variable and λ is an integral scalar. This guarantees

that, when reversing the transformation during post-processing, we can obtain an integral value of x_k from the values of x_j and the new integer variable z .

Note that applying this presolve method to the problem (1) can be seen as applying the row-based version to the associated dual problem. Therefore it takes time linear in the number of constraints, i.e. its time complexity is $O(|\mathcal{M}|)$. Due to the additional overhead resulting from constraint (14) the difference in the number of nonzeros $|U| - |W|$ is generally required to be larger than in the row-based method.

3.3 LP-based Bound Tightening on Two Constraints

The idea of using two rows of the constraint matrix A to determine tighter variable bounds or redundant constraints has already been described in [4]. We will first present the presolve method and then discuss an important observation to omit unnecessary calculations.

Consider two arbitrary constraints of (1)

$$\begin{array}{rcl} A_{rU}x_U + A_{rV}x_V & & \geq b_r, \\ A_{sU}x_U + & A_{sW}x_W & \geq b_s, \end{array}$$

where $r, s \in \mathcal{M}$ and $U, V, W \subseteq \mathcal{N}$. Note that for any $i \in \mathcal{M}$ with $\circ_i \in \{\leq, =\}$ the corresponding constraint can be normalized to be of the form above. Together with the variable bounds $\ell \leq x \leq u$ we exploit the special form of the two constraints to set up the following two LPs

$$\begin{array}{ll} y_{max} & = \max A_{rU}x_U \\ & \text{s.t. } A_{sU}x_U + A_{sW}x_W \geq b_s \\ & \quad \ell \leq x \leq u, \\ y_{min} & = \min A_{rU}x_U \\ & \text{s.t. } A_{sU}x_U + A_{sW}x_W \geq b_s \\ & \quad \ell \leq x \leq u. \end{array} \tag{15}$$

As shown in [17] and [7], each of these problems can be solved within linear time. Provided we have finite values for y_{max} and y_{min} , i.e. the above problems have been feasible and bounded, the results can be used to, again in linear time, derive potentially stronger variable bounds and detect redundancy of constraint r . Bounds on variables $x_j, j \in V$, can be calculated via (16) where $V' = V \setminus \{j\}$ and constraint r is redundant if condition (17) holds. Note the similarity of (16) to (5) and (7), where $\sup(A_{rU}x_U)$ is replaced by y_{max} as well as the fact that y_{max} is at least as strong, and potentially stronger, as $\sup(A_{rU}x_U)$.

$$\begin{array}{l} x_j \geq \frac{b_r - \sup(A_{rV'}x_{V'}) - y_{max}}{a_{rj}}, \text{ for } a_{rj} > 0 \\ x_j \leq \frac{b_r - \sup(A_{rV'}x_{V'}) - y_{max}}{a_{rj}}, \text{ for } a_{rj} < 0 \end{array} \tag{16}$$

$$\inf(A_{rV}x_V) \geq b_r - y_{min} \tag{17}$$

Note that the roles of the rows r and s are interchangeable. As the following lemma shows, bound tightening on the constraints r and s may only improve the bounds on variables $x_j, j \in V$ if there exists at least one variable $x_i, i \in U$ with $a_{ri} < 0 < a_{si}$ or $a_{si} < 0 < a_{ri}$.

Lemma 3.1. *If for all variables $x_i, i \in U$, the corresponding coefficients a_{ri} and a_{si} are either both non-negative or both non-positive, then inequality (16) cannot improve the bounds on the variables $x_j, j \in V$, over regular FBBT on constraint r .*

Proof. Let $x_i, i \in U$ be a variable such that its coefficients a_{ri} and a_{si} are both non-negative. In this case, setting the variable x_i to its upper bound maximizes the value of y_{max} and also contributes most to satisfying the constraint $A_{sU}x_U + A_{sW}x_W \geq b_s$. Analogously, a variable with non-positive coefficients a_{ri} and a_{si} should be set to its lower bound. Hence, the value y_{max} of the optimal solution coincides with $\sup(A_{rU}x_U)$ such that (16) yields the same bounds as (5) and (7) on constraint r . \square

The lemma above shows that in order to improve the bounds over what can be achieved using FBBT on a single constraint, it is of crucial importance to find pairs of inequalities that contain variables such that their respective coefficients have opposite sign.

Remark 3.2. *By rewriting the objective function for y_{min} as $\max -A_{rU}x_U$ we can, by the same arguments as in Lemma 3.1, reason that equal signs are advantageous for detecting redundant constraints. Since we focus on bound tightening we did not continue this line of thought in this work, but it certainly is an interesting experiment for future research.*

Next we give an example to further illustrate the basic idea as well as the statements made in Lemma 3.1 and Remark 3.2.

Example 3.3. *Consider the following two constraints*

$$\begin{aligned} x_1 + 2x_2 + 3x_3 + 2x_4 &\geq 4, \\ x_2 + x_3 + x_4 &\geq 2, \end{aligned}$$

with $x_1, \dots, x_4 \in [0, 1]$, $U = \{2, 3, 4\}$, $V = \{1\}$, and $W = \emptyset$. Then we get $y_{max} = 7$ and $y_{min} = 4$ and applying (16) and (17) yields

$$\begin{aligned} x_1 &\geq 4 - 7 = -3 \\ 0 &\geq 4 - 4 = 0. \end{aligned}$$

Therefore, the lower bound of x_1 could not be tightened over simple FBBT on the first constraint, but we have detected that the first constraint can be removed from the problem as it is redundant with the second constraint.

If we keep the variable bounds and change the problem to

$$\begin{aligned} x_1 + 2x_2 + 3x_3 + 2x_4 &\geq 4, \\ -x_2 - x_3 - x_4 &\geq -1, \end{aligned}$$

we obtain $y_{max} = 3$ and $y_{min} = 0$. Again, we apply (16) as well as (17) and get

$$\begin{aligned} x_1 &\geq 4 - 3 = 1 \\ 0 &\geq 4 - 0 = 4. \end{aligned}$$

In this case, due to opposite signs for coefficients appearing in both constraints, we were able to tighten the lower bound on x_1 , but the first constraint is no longer redundant with the second one. Note that FBBT on the first constraint would still yield -3 as lower bound on x_1 .

In certain cases, presolve methods interact with each other. This is one reason why different presolve methods are often grouped by their runtime behavior and executed in repeated rounds [20]. Of particular interest are cases where one method generates a structure that helps another method to perform further useful reductions. As an example, we will look at an interaction between the method of Section 3.1 and the method described here.

Example 3.4. Consider the following problem:

$$\begin{array}{ll}
\min & x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 \\
\text{s.t.} & x_1 + x_2 + x_3 + x_4 + x_5 \geq 10 \\
& -x_1 - x_2 + x_6 - x_7 \geq 6 \\
& - x_4 - x_5 = -4 \\
& x_1, x_2, x_6, x_7 \in [0, 10] \\
& x_3, x_4, x_5 \geq 0.
\end{array}$$

With $U = \{1, 2\}$, $V = \{3, 4, 5\}$ and $W = \{6, 7\}$ we perform LP-based bound tightening for the first and second constraint as follows

$$\begin{array}{ll}
y_{max} = \max & x_1 + x_2 \\
\text{s.t.} & -x_1 - x_2 + x_6 - x_7 \geq 6 \\
& x_1, x_2, x_6, x_7 \in [0, 10]
\end{array}$$

to obtain $y_{max} = 4$. Now we would like to determine a tighter lower bound on x_3 , but this is not possible as $\sup(x_3 + x_4) = \infty$. However, if we first use the presolve method of Section 3.1 and add the equality to the first constraint, then the variables x_4 and x_5 disappear. As a result, we obtain $x_3 \geq (6 - y_{max})/1 = 2$.

3.4 Bound Tightening on Convex Combinations of Two Constraints

Belotti [8] presented an algorithm to efficiently derive bounds for all variables from the LP relaxation of the problem

$$\begin{array}{l}
A_r \cdot x \geq b_r \\
A_s \cdot x \geq b_s \\
\ell \leq x \leq u \\
x_j \in \mathbb{Z} \forall j \in \mathcal{I} \subseteq \mathcal{N},
\end{array} \tag{18}$$

where $r, s \in \mathcal{M}$. Note again that for any constraint i with $\circ_i \in \{\leq, =\}$ the corresponding constraint can be normalized to be of the same form as constraints r and s above. We will first summarize the original method and then discuss an extension onto two rows together with disjoint set packing constraints.

3.4.1 Basic Bound Tightening on Convex Combinations of Two Constraints

The basic idea is to reformulate the problem using convex combinations of the two constraints in (18). Let $\lambda \in [0, 1]$, for the convex combination of the coefficient vectors we write $\bar{a}^{rs}(\lambda) := \lambda A_r + (1 - \lambda)A_s$ and $\bar{b}^{rs}(\lambda) := \lambda b_r + (1 - \lambda)b_s$ for the convex combination of the right hand side. For single indices $j \in \mathcal{N}$ or index subsets $V \subseteq \mathcal{N}$ we write $\bar{a}_j^{rs}(\lambda)$ and $\bar{a}_V^{rs}(\lambda)$, respectively, to address the corresponding entries of $\bar{a}^{rs}(\lambda)$. One may now perform FBBT on the combined constraint $\bar{a}^{rs}(\lambda)x \geq \bar{b}^{rs}(\lambda)$. Note that for $\lambda \in \{0, 1\}$ this cannot improve the bounds over FBBT on the original constraints, and we therefore restrict ourselves to $\lambda \in (0, 1)$. In short, possibly stronger variable bounds $[\bar{\ell}_j, \bar{u}_j]$ on a variable x_j can be derived by solving the following one-dimensional optimization problems:

$$\bar{\ell}_j = \max \left\{ \frac{L_j(\lambda)}{\bar{a}_j^{rs}(\lambda)} \mid \lambda \in (0, 1) \wedge \bar{a}_j^{rs}(\lambda) > 0 \right\} \tag{19}$$

$$\bar{u}_j = \min \left\{ \frac{L_j(\lambda)}{\bar{a}_j^{rs}(\lambda)} \mid \lambda \in (0, 1) \wedge \bar{a}_j^{rs}(\lambda) < 0 \right\}, \tag{20}$$

where

$$L_j(\lambda) := \bar{b}^{rs}(\lambda) - \sup(\bar{a}^{rs}(\lambda)_{\mathcal{N} \setminus \{j\}} x_{\mathcal{N} \setminus \{j\}}).$$

Belotti [8] has then shown that these problems can be solved by evaluating the piecewise linear $L_j(\lambda)$ for a small finite set of values for λ , namely the set of all *breakpoints* of $L_j(\lambda)$. The next proposition summarizes the result.

Proposition 3.5. *The set*

$$\Lambda := \{\lambda \in (0, 1) \mid (\exists j \in \mathcal{N} : \bar{a}_j^{rs}(\lambda) = 0 \wedge \bar{a}_j^{rs}(\lambda^*) \neq 0 \forall \lambda^* \neq \lambda, \lambda^* \in (0, 1))\}$$

contains all optimal solutions to the problems (19) and (20).

Note that problems (19) and (20) may not have optimal solutions or are unbounded. However, these edge cases, which are discussed in more detail in the original paper [8], do not affect the overall viability of this approach.

Computing $L_j(\lambda)$ for all variables and each $\lambda \in \Lambda$ individually would result in an algorithm with running time in $O(|\mathcal{N}|^3)$. However, it is possible to simultaneously solve the optimization problems for all variables in time in $O(|\mathcal{N}|^2)$ by using constant-time update operations for each $\lambda \in \Lambda$ instead of recomputing each $L_j(\lambda)$ individually. We will outline the basic idea as well as the core operations; for full details of the algorithm, see [8].

Denote $\Lambda = \{\lambda_1, \dots, \lambda_{|\Lambda|}\}$ such that $0 < \lambda_1 < \dots < \lambda_{|\Lambda|} < 1$ and let $\tilde{\lambda} \in (0, \lambda_1)$. We define

$$\begin{aligned} L^r &:= b_r - \sum_{\bar{a}_j^{rs}(\tilde{\lambda}) > 0} a_{rj} u_j - \sum_{\bar{a}_j^{rs}(\tilde{\lambda}) < 0} a_{rj} \ell_j \\ L^s &:= b_s - \sum_{\bar{a}_j^{rs}(\tilde{\lambda}) > 0} a_{sj} u_j - \sum_{\bar{a}_j^{rs}(\tilde{\lambda}) < 0} a_{sj} \ell_j, \end{aligned}$$

which, not using $\tilde{\lambda}$, can be rewritten as

$$L^r = b_r - \sum_{\substack{a_{sj} > 0 \vee (a_{sj} = 0 \wedge a_{rj} > 0) \\ j \in \text{supp}(A_{r.})}} a_{rj} u_j - \sum_{\substack{a_{sj} < 0 \vee (a_{sj} = 0 \wedge a_{rj} < 0) \\ j \in \text{supp}(A_{r.})}} a_{rj} \ell_j \quad (21)$$

$$L^s = b_s - \sum_{\substack{a_{sj} > 0 \\ j \in \text{supp}(A_{s.})}} a_{sj} u_j - \sum_{\substack{a_{sj} < 0 \\ j \in \text{supp}(A_{s.})}} a_{sj} \ell_j. \quad (22)$$

Since the signs of $\bar{a}_j^{rs}(\lambda)$, $j \in \mathcal{N}$ do not change between 0 and λ_1 we can compute $L_j(\lambda)$ for all $j \in \mathcal{N}$, $\lambda \in [0, \lambda_1]$ via

$$\begin{aligned} L_j(\lambda) &= \lambda L^r + (1 - \lambda) L^s + \bar{a}_j^{rs}(\lambda) d_j \\ d_j &= \begin{cases} u_j, & \text{if } \bar{a}_j^{rs}(\lambda) > 0 \\ \ell_j, & \text{if } \bar{a}_j^{rs}(\lambda) < 0. \end{cases} \end{aligned}$$

In other words, if L^r and L^s are given and $\lambda \in [0, \lambda_1]$, we can compute bounds for a variable x_j in constant time using (23) if $\bar{a}_j^{rs}(\lambda) > 0$ and (24) if $\bar{a}_j^{rs}(\lambda) < 0$.

$$x_j \geq \frac{\lambda L^r + (1 - \lambda) L^s + \bar{a}_j^{rs}(\lambda) u_j}{\bar{a}_j^{rs}(\lambda)} \quad (23)$$

$$x_j \leq \frac{\lambda L^r + (1 - \lambda) L^s + \bar{a}_j^{rs}(\lambda) \ell_j}{\bar{a}_j^{rs}(\lambda)}. \quad (24)$$

In order to derive bounds for $\lambda \in [\lambda_1, \lambda_2]$ we need to first update L^r and L^s . More precisely, for each variable $x_j, j \in \mathcal{N}$ with $\bar{a}_j^{rs}(\lambda_1) = 0$, i.e. each variable whose combined coefficient switches sign in λ_1 , we need update L^r and L^s via (25) and (26) respectively. If a combined coefficient changes sign from positive to negative, we need to set the corresponding variable from its upper bound to its lower bound. Analogously, a variable needs to be set from its lower bound to its upper bound if its combined coefficient changes sign from negative to positive.

$$L^r \leftarrow L^r + a_{rj}\delta \quad (25)$$

$$L^s \leftarrow L^s + a_{sj}\delta \quad (26)$$

$$\delta = \begin{cases} u_j - \ell_j, & \text{if } a_{rj} < 0 < a_{sj}, \\ \ell_j - u_j, & \text{if } a_{sj} < 0 < a_{rj} \end{cases}$$

Iteratively repeating the bound calculations and updates for all $\lambda \in \Lambda$ solves the problems (19) and (20). To summarize, we state the resulting procedure in Algorithm 1. For more details, especially on implementation, see Algorithm 1 in [8].

Algorithm 1: CombineIneqs($A_{r\cdot}, A_{s\cdot}, b_r, b_s, \ell, u$)

```

1  $\Lambda \leftarrow \{\lambda \in (0, 1) \mid (\exists j \in \mathcal{N}: \bar{a}_j^{rs}(\lambda) = 0 \wedge \bar{a}_j^{rs}(\lambda^*) \neq 0 \forall \lambda^* \neq \lambda, \lambda^* \in (0, 1))\}$ 
2 denote  $\Lambda = \{\lambda_1, \dots, \lambda_{|\Lambda|}\}$  s.t.  $0 < \lambda_1 < \dots < \lambda_{|\Lambda|} < 1$ 
3 initialize  $L^r$  and  $L^s$  via (21) and (22)
4 foreach  $j \in \mathcal{N}$  do
5    $\bar{\ell}_j \leftarrow -\infty$ 
6    $\bar{u}_j \leftarrow \infty$ 
7 foreach  $k \in \{1, \dots, |\Lambda|\}$  do
8   foreach  $j \in \mathcal{N}$  do
9     if  $\bar{a}_j^{rs}(\lambda_k) > 0$  then
10       $\bar{\ell}_j \leftarrow \max\{\bar{\ell}_j, \frac{\lambda L^r + (1-\lambda)L^s + \bar{a}_j^{rs}(\lambda)u_j}{\bar{a}_j^{rs}(\lambda)}\}$ 
11     if  $\bar{a}_j^{rs}(\lambda_k) < 0$  then
12       $\bar{u}_j \leftarrow \min\{\bar{u}_j, \frac{\lambda L^r + (1-\lambda)L^s + \bar{a}_j^{rs}(\lambda)\ell_j}{\bar{a}_j^{rs}(\lambda)}\}$ 
13   update  $L^r, L^s$  via (25) and (26)
14 return  $[\bar{\ell}, \bar{u}]$ 

```

3.4.2 Further Improvement via Disjoint Set Packing Constraints

In this section, we show that Belotti's approach can be extended to derive bounds from Problem (18) together with an arbitrary number of disjoint set packing constraints of the form

$$\begin{aligned} \sum_{j \in S_p} x_j &\leq 1 & \forall S_p \in \mathcal{S} \\ x_j &\in \{0, 1\} & \forall S_p \in \mathcal{S} \forall j \in S_p, \end{aligned} \quad (27)$$

where $\mathcal{S} = \{S_1, \dots, S_{|\mathcal{S}|}\}$ is a partitioning of a subset of the integer variable index set \mathcal{I} . More precisely, for $\lambda \in (0, 1)$ we want to derive bounds from the problem

$$\begin{aligned} \bar{a}^{rs}(\lambda)x &\geq \bar{b}^{rs}(\lambda) \\ \ell &\leq x \leq u \\ (27) \\ x_j &\in \mathbb{Z} \forall j \in \mathcal{I} \subseteq \mathcal{N}. \end{aligned} \quad (28)$$

To further simplify notation, we write $S := \bigcup_{S_p \in \mathcal{S}} S_p$ and $T := \mathcal{N} \setminus S$.

In the following two examples, we illustrate the effect of the addition of set packing constraints and then proceed to analyze how Belotti's algorithm needs to be adjusted in order to achieve these results with minimal additional computational effort.

Example 3.6. Consider the following mixed-integer problem

$$\begin{aligned} 2x_1 - x_2 + x_3 &\geq 1 \\ -x_1 + 2x_2 + x_3 &\geq 1 \\ x_1 + x_2 &\leq 1 \\ x_1, x_2 &\in \{0, 1\}, x_3 \in [-2, 3] \end{aligned}$$

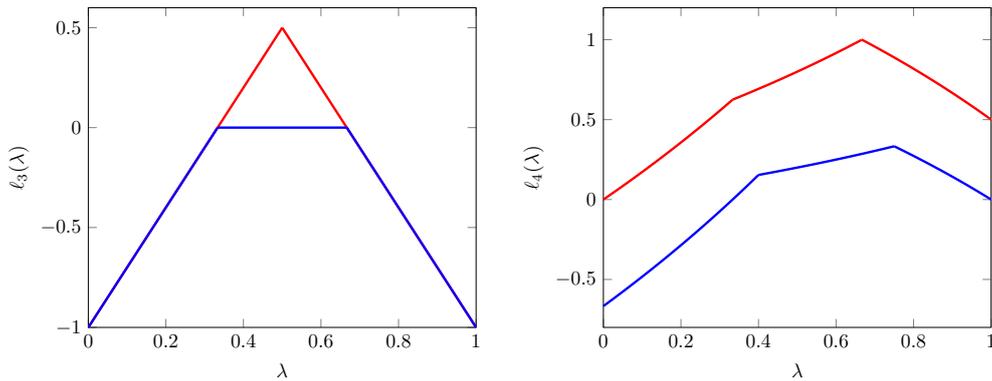
with one set packing constraint. Figure 1a shows a comparison between

$$\frac{L_3(\lambda)}{\bar{a}_3(\lambda)} = L_3(\lambda) = \begin{cases} -1 + 3\lambda, & \text{if } \lambda \in (0, \frac{1}{3}] \\ 0, & \text{if } \lambda \in (\frac{1}{3}, \frac{2}{3}] \\ 2 - 3\lambda, & \text{if } \lambda \in (\frac{2}{3}, 1), \end{cases} \quad (29)$$

which results from applying the original method presented in 3.4.1, and solving the following optimization problem for $\lambda \in (0, 1)$.

$$\begin{aligned} \min x_3 \\ (3\lambda - 1)x_1 + (2 - 3\lambda)x_2 + x_3 &\geq 1 \\ x_1 + x_2 &\leq 1 \\ x_1, x_2 &\in \{0, 1\}, x_3 \in [-2, 3] \end{aligned} \quad (30)$$

Whereas the original method cannot tighten the bound at all, when considering the set packing constraint we get a positive lower bound on x_3 for $\lambda \in (\frac{1}{3}, \frac{2}{3})$. First, this illustrates one case for which adding a set packing constraint indeed yields a better bound on x_3 . Second, the example shows that the set Λ needs to be extended as it no longer contains all optimal solutions to (30). In this example, there is the new breakpoint $\lambda = \frac{1}{2}$ where we have $\bar{a}_1(\frac{1}{2}) = \frac{1}{2} = \bar{a}_2(\frac{1}{2})$. This new breakpoint is clearly visible in Figure 1a. We will see later, that at all new breakpoints λ^* the index $\operatorname{argmax}_{j \in S_p} \{\bar{a}_j(\lambda)\}$ changes between $\lambda < \lambda^*$ and $\lambda > \lambda^*$.



(a) Lower bound on x_3 depending on λ computed via (29) in blue and (30) in red (b) Lower bound on x_4 depending on λ computed via (31) in blue and (32) in red

Figure 1: Bounds on variables x_3 in Example 3.6 (left) and x_4 in Example 3.7 (right)

Example 3.7. Consider the following mixed-integer program

$$\begin{aligned} x_1 + 3x_2 - x_3 + 2x_4 &\geq 4, \\ 2x_1 - 2x_2 + 3x_3 + 3x_4 &\geq 3, \\ x_1 + x_2 + x_3 &\leq 1, \\ x_1, x_2, x_3 &\in \{0, 1\}, x_4 \in [0, 4]. \end{aligned}$$

Again, Figure 1b shows for $\lambda \in (0, 1)$ the comparison between the results obtained by the method presented in 3.4.1, i.e. ,

$$\frac{L_4(\lambda)}{\bar{a}_4(\lambda)} = \begin{cases} \frac{6\lambda-2}{3-\lambda}, & \text{if } \lambda \in (0, \frac{2}{5}] \\ \frac{\lambda}{3-\lambda}, & \text{if } \lambda \in (\frac{2}{5}, \frac{3}{4}) \\ \frac{3-3\lambda}{3-\lambda}, & \text{if } \lambda \in (\frac{3}{4}, 1), \end{cases} \quad (31)$$

and solving

$$\begin{aligned} \min x_4 \\ (2-\lambda)x_1 + (5\lambda-2)x_2 + (3-4\lambda)x_3 + (3-\lambda)x_4 &\geq 3+\lambda \\ x_1 + x_2 + x_3 &\leq 1 \\ x_1, x_2, x_3 &\in \{0, 1\}, x_4 \in [0, 4]. \end{aligned} \quad (32)$$

Similar to the previous example, the set packing constraint allows us to derive stronger bounds. Whereas the original method yields a lower bound of $\frac{1}{3}$ for $\lambda = \frac{3}{4}$ the set packing constraint allows us to tighten that bound to 1 for $\lambda = \frac{2}{3}$ and we get new breakpoints again, namely $\lambda_1 = \frac{1}{3}$ and $\lambda_2 = \frac{2}{3}$. Note that for these breakpoints we also have $\bar{a}_1(\frac{1}{3}) = \frac{5}{3} = \bar{a}_3(\frac{1}{3})$ and $\bar{a}_1(\frac{2}{3}) = \frac{4}{3} = \bar{a}_2(\frac{2}{3})$ respectively. As a final note, while $L_j(\lambda)$ and $\bar{a}_j^{rs}(\lambda)$ are always (piecewise) linear, this does not necessarily hold for the quotient $\frac{L_j(\lambda)}{\bar{a}_j^{rs}(\lambda)}$.

Note that in both examples we have strengthened the bounds of a variable that did not appear in any set packing constraint. In the following, we assume $j \in T$, i.e. the variable x_j is not present in any set packing constraint. For a variable $x_j, j \in S_p, S_p \in \mathcal{S}$, we can still derive bounds by simply relaxing the corresponding set packing constraint to $\sum_{k \in S_p \setminus \{j\}} x_k \leq 1$. At the end of the section we present derivations that are stronger than what is achievable by relaxing the set packing constraint.

Let $j \in T$. Since for each subset $S_p \in \mathcal{S}$ we can choose at most one variable $x_k \in S_p$ to be set to one, we obtain, for any given $\lambda \in (0, 1)$ with $\bar{a}_j^{rs}(\lambda) > 0$, valid bounds by

$$x_j \geq \frac{1}{\bar{a}_j^{rs}(\lambda)} (\bar{b}^{rs}(\lambda) - \sup(\bar{a}_{T \setminus \{j\}}^{rs}(\lambda) x_{T \setminus \{j\}}) - \sum_{S_p \in \mathcal{S}} \max\{0, \max_{k \in S_p} \{\bar{a}_k^{rs}(\lambda)\}\}), \quad (33)$$

and analogously for $\lambda \in (0, 1)$ with $\bar{a}_j^{rs}(\lambda) < 0$ we have

$$x_j \leq \frac{1}{\bar{a}_j^{rs}(\lambda)} (\bar{b}^{rs}(\lambda) - \sup(\bar{a}_{T \setminus \{j\}}^{rs}(\lambda) x_{T \setminus \{j\}}) - \sum_{S_p \in \mathcal{S}} \max\{0, \max_{k \in S_p} \{\bar{a}_k^{rs}(\lambda)\}\}). \quad (34)$$

Note that (33) and (34) solve, in their respective cases, the problems $\min x_j$, s.t. (28), as well as $\max x_j$, s.t. (28) to optimality. This also shows that for any given $\lambda \in (0, 1)$ the set packing constraint corresponding to a subset $S_p \in \mathcal{S}$ is redundant if and only if $\bar{a}_j^{rs}(\lambda) > 0$ holds for at most one index $j \in S_p$. To simplify notation, define

$$\begin{aligned} \Phi_p(\lambda) &:= \max\{0, \max_{k \in S_p} \{\bar{a}_k^{rs}(\lambda)\}\}, \forall S_p \in \mathcal{S}, \\ \Gamma_j(\lambda) &:= \bar{b}^{rs}(\lambda) - \sup(\bar{a}_{T \setminus \{j\}}^{rs}(\lambda) x_{T \setminus \{j\}}) - \sum_{S_p \in \mathcal{S}} \Phi_p(\lambda). \end{aligned}$$

Transferring bounds (33) and (34) to the problems (19) and (20) then yields

$$\bar{\ell}_j = \max \left\{ \frac{\Gamma_j(\lambda)}{\bar{a}_j^{rs}(\lambda)} \mid \lambda \in (0, 1) \wedge \bar{a}_j^{rs}(\lambda) > 0 \right\}, \quad (35)$$

$$\bar{u}_j = \min \left\{ \frac{\Gamma_j(\lambda)}{\bar{a}_j^{rs}(\lambda)} \mid \lambda \in (0, 1) \wedge \bar{a}_j^{rs}(\lambda) < 0 \right\}. \quad (36)$$

Similar to problems (19) and (20), the problems (35) and (36) may not have optimal solutions or are unbounded and again, these edge cases do not affect the overall viability of this approach, and hence we restrict ourself to cases where these problems are bounded and have an optimal solution.

In order to show Belotti's algorithm can be extended to the problems above we prove that for solving these problems a linear number of evaluations of $\Gamma_j(\lambda)$ suffices. As a first step to prove the analogue of Proposition 3.5 for the problems (35) and (36), we show that $\Gamma_j(\lambda)$ is, just like $L_j(\lambda)$ in the original case, a piecewise linear function and determine all its breakpoints.

Lemma 3.8. *For any $j \in T$, the function $\Gamma_j(\lambda)$ is piecewise linear on its domain $(0, 1)$ and the set*

$$\tilde{\Lambda} := \Lambda \cup \{ \lambda \in (0, 1) \mid \exists S_p \in \mathcal{S} : \exists k, l \in S_p, k \neq l : (\bar{a}_k^{rs}(\lambda) = \bar{a}_l^{rs}(\lambda) = \Phi_p(\lambda) \wedge \bar{a}_k^{rs}(\lambda') \neq \bar{a}_l^{rs}(\lambda') \forall \lambda' \neq \lambda, \lambda' \in (0, 1)) \}$$

contains all its breakpoints.

Proof. Each $\Phi_p(\lambda), S_p \in \mathcal{S}$, is defined as the maximum over the constant zero function and the set of the linear terms $\bar{a}_k^{rs}(\lambda), k \in S_p$, and is hence piecewise linear. Consequently, we have that $\Gamma_j(\lambda)$ is piecewise linear as the sum of linear and piecewise linear functions.

By the same arguments as in the original case the set Λ contains all breakpoints of $\bar{b}^{rs}(\lambda) - \sup(\bar{a}_{T \setminus \{j\}}^{rs}(\lambda)x_{T \setminus \{j\}})$. Let $\lambda^* \in (0, 1)$ be a breakpoint of $\Phi_p(\lambda)$. We distinguish two cases:

- i. $\Phi_p(\lambda^*) = 0$:
In this case some $\bar{a}_k^{rs}(\lambda)$ intersects the constant zero function in λ^* . More precisely, there exists some $k \in S_p$ with $\bar{a}_k^{rs}(\lambda^*) = 0 \wedge \bar{a}_k^{rs}(\lambda) \neq 0 \forall \lambda \neq \lambda^*, \lambda \in (0, 1)$, and hence, λ^* is, by definition of Λ already contained in $\tilde{\Lambda}$.
- ii. $\Phi_p(\lambda^*) > 0$: Here we have an intersection between two $\bar{a}_k^{rs}(\lambda)$, i.e. there exist distinct indices $k, l \in S_p$ with $\bar{a}_k^{rs}(\lambda^*) = \bar{a}_l^{rs}(\lambda^*) = \Phi_p(\lambda^*)$ and $\bar{a}_k^{rs}(\lambda) \neq \bar{a}_l^{rs}(\lambda) \forall \lambda \neq \lambda^*, \lambda \in (0, 1)$. Therefore the breakpoint λ^* is also contained in $\tilde{\Lambda}$ which completes the proof. \square

Note that $|\tilde{\Lambda}|$ is linear in the number of variables since the set packing constraints are disjoint. Further, the set $\tilde{\Lambda}$ can be computed in time in $O(|\mathcal{N}| \log |\mathcal{N}|)$. A simple well known algorithm for this would be the following. For each $S_p \in \mathcal{S}$ we first sort the linear functions $\bar{a}_k^{rs}(\lambda), k \in S_p$ by their gradients $a_{rk} - a_{sk}$. For the $\bar{a}_k^{rs}(\lambda)$ with the smallest gradient we then have $\bar{a}_k^{rs}(\lambda) = \Phi_p(\lambda)$ for small enough values of λ . From there, go through the remaining $\bar{a}_k^{rs}(\lambda)$ in ascending order of their gradients and iteratively compute the next breakpoint. Since each $\bar{a}_k^{rs}(\lambda)$ needs to be considered at most once, the total running time is in $O(|\mathcal{N}| \log |\mathcal{N}|)$.

Theorem 3.9. *If an optimal solution to Problem (35) exists, then at least one optimal solution is also contained in $\tilde{\Lambda}$. The same statement holds for Problem (36).*

Proof. By Lemma 3.8 the set $\tilde{\Lambda}$ contains all breakpoints of the piecewise linear $\Gamma_j(\lambda)$. Together with $\bar{a}^{rs}(\lambda)$ being linear, we established the same conditions as in the proof of Proposition 3.5. The remaining proof of this theorem is then identical and can be found in full detail as proof of Proposition 1 in [8]. \square

To extend the original algorithm for computing bounds one now proceeds as follows. First, the extended algorithm must compute L^r and L^s in a slightly different way. For easier notation, let $\kappa_p \in S_p$ be the index of a variable x_{κ_p} with $a_{s\kappa_p} = \max_{j \in S_p} \{a_{sj}\}$ and $a_{r\kappa_p} - a_{s\kappa_p} \geq \max\{(a_{rj} - a_{sj}) \mid j \in S_p, a_{sj} = a_{s\kappa_p}\}$. In other words, of the variables $x_j, j \in S_p$, with maximum coefficient $\bar{a}_j^{rs}(\lambda)$ we choose the one with maximal gradient of $\bar{a}_j^{rs}(\lambda)$.

$$\begin{aligned}\tilde{L}^r &= b_r - \sum_{\substack{a_{sj} > 0 \vee (a_{sj} = 0 \wedge a_{rj} > 0) \\ j \in T}} a_{rj} u_j - \sum_{\substack{a_{sj} < 0 \vee (a_{sj} = 0 \wedge a_{rj} < 0) \\ j \in T}} a_{rj} \ell_j - \sum_{S_p \in \mathcal{S}} \Phi_p^r \\ \tilde{L}^s &= b_s - \sum_{\substack{a_{sj} > 0 \\ j \in T}} a_{sj} u_j - \sum_{\substack{a_{sj} < 0 \\ j \in T}} a_{sj} \ell_j - \sum_{S_p \in \mathcal{S}} \Phi_p^s\end{aligned}$$

where

$$\begin{aligned}\Phi_p^r &= \begin{cases} 0, & \text{if } a_{s\kappa_p} < 0 \vee (a_{s\kappa_p} = 0 \wedge a_{r\kappa_p} \leq 0) \\ a_{r\kappa_p}, & \text{else} \end{cases}, \\ \Phi_p^s &= \begin{cases} 0, & \text{if } a_{s\kappa_p} < 0 \vee (a_{s\kappa_p} = 0 \wedge a_{r\kappa_p} \leq 0) \\ a_{s\kappa_p}, & \text{else} \end{cases}\end{aligned}$$

Second, the extended algorithm must compute the new set of breakpoints $\tilde{\Lambda}$ which, as mentioned above, can be done in time in $O(|\mathcal{N}| \log |\mathcal{N}|)$ and therefore does not increase the total time complexity. Note that in the extension it is required to track which variable coefficient currently determines the value of $\Phi_p(\lambda)$ for each $S_p \in \mathcal{S}$. However, this only leads to linear computational overhead if the breakpoints and variables are labelled accordingly while computing $\tilde{\Lambda}$. The bound calculations corresponding to (23) and (24) as well as the updates corresponding to (25) and (26) essentially work the same but consist of larger case distinctions due to the set packing constraints.

Before closing this section we will discuss what can be done for variables appearing in the set packing constraints. Let $j \in S_p$ for some fixed $S_p \in \mathcal{S}$ and $\lambda \in \tilde{\Lambda}$. We distinguish four cases.

- i. $\bar{a}_j^{rs}(\lambda) > 0 \wedge \bar{a}_j^{rs}(\lambda) < \Phi_p(\lambda)$: In this case we directly apply (33) since the right hand side does not involve x_j . Due to checking for a lower bound and $x_j \in \{0, 1\}$ we assume $x_j = 0$ and with $\bar{a}_j^{rs}(\lambda) > 0$ the expression reduces to

$$0 \geq \Gamma_j(\lambda). \quad (37)$$

Together with the set packing constraint corresponding to S_p the original problem is infeasible if above condition does not hold.

- ii. $\bar{a}_j^{rs}(\lambda) > 0 \wedge \bar{a}_j^{rs}(\lambda) = \Phi_p(\lambda)$: Here we use a relaxation of (33) to detect infeasibility. More precisely, we assume there exists some $k \in S_p, k \neq j$ with $\bar{a}_j^{rs}(\lambda) = \bar{a}_k^{rs}(\lambda)$. Then, as in the previous case, we check for infeasibility via (37).

A proper lower bound for the given λ could be derived by strengthening the summand $\Phi_p(\lambda)$ of $\Gamma_j(\lambda)$ to $\max_{k \in S_p \setminus \{j\}} \{0, \max\{\bar{a}_k^{rs}(\lambda)\}\}$ and considering the additional breakpoints of this strengthening that may not already be contained in $\tilde{\Lambda}$. However, we are aware of no efficient way to incorporate these calculations without creating additional computational overhead.

- iii. $\bar{a}_j^{rs}(\lambda) < 0 \wedge \Phi_p(\lambda) = 0$: Now the set packing constraint corresponding to S_p has no effect and we get an upper bound on x_j via (34).
- iv. $\bar{a}_j^{rs}(\lambda) < 0 \wedge \Phi_p(\lambda) > 0$: Any solution with $x_j = 1$ immediately forces $x_k = 0, k \in S_p, k \neq j$. We can therefore adjust (34) to this case and set $x_j = 0$ if

$$\frac{\Gamma_j(\lambda) + \Phi_p(\lambda)}{\bar{a}_j^{rs}(\lambda)} < 1$$

holds. Note that a negative left hand side does not necessarily imply infeasibility.

Example 3.10. *This example illustrates the second case of above distinction. Consider the following mixed-integer problem.*

$$\begin{aligned} 3x_1 + x_2 + 4x_3 &\geq 3 \\ 3x_1 + 4x_2 + x_3 &\geq 3 \\ x_1 + x_2 + x_3 &\leq 1 \\ x_1, x_2, x_3 &\in \{0, 1\} \end{aligned}$$

By Lemma 3.8 we have $\tilde{\Lambda} = \{\frac{1}{3}, \frac{2}{3}\}$ and for $\lambda = \frac{1}{3}$ we get

$$\begin{aligned} 3x_1 + 3x_2 + 2x_3 &\geq 3 \\ x_1 + x_2 + x_3 &\leq 1 \\ x_1, x_2, x_3 &\in \{0, 1\}. \end{aligned}$$

Due to $\bar{a}_1(\frac{1}{3}) = 3 = \Phi(\frac{1}{3})$ we can, by the second case of above distinction, check for feasibility via

$$0 \geq \Gamma_1(\frac{1}{3}) = 0. \quad (38)$$

Because of $\bar{a}_1(\frac{1}{3}) = \bar{a}_2(\frac{1}{3}) = 3$ it is not even possible to strengthen $\Phi_1(\frac{1}{3})$ and similar considerations hold for $\lambda = \frac{2}{3}$. However, $\max\{0, \max_{k \in \{2,3\}} \bar{a}_k(\lambda)\}$ has an additional breakpoint at $\lambda = \frac{1}{2}$ for which it would be possible to tighten the lower bound of x_1 to

$$\ell_1 \geq (3 - \frac{5}{2})/3 = \frac{1}{6}. \quad (39)$$

Remark 3.11. *This presolve method is very similar to the one presented in the previous section, so we want to highlight some of the differences between them. Most importantly, the method in Section 3.3 can be implemented in a straightforward manner and it is possible to punctually apply the method to single variables. However, there are two major points that speak in favor of the method of this section.*

First, the previous method is restricted to tightening bounds of variables that do not have nonzero coefficients in both constraints whereas this section's method is able to tighten bounds of all variables involved. Second, it does not seem the case that the set packing extension translates to the presolve method from Section 3.3 in a canonical way. Consider the case that all variables are binary and present in one of the disjoint set packing constraints. Adding all the set packing constraints to the optimization problems shown in (16) results in solving so-called Multiple-Choice Knapsack Problems, see [39], which would exceed the scope of this paper.

3.5 Exploiting Complementary Slackness on Two Columns of Continuous Variables

By propagating the dual problem of the linear programming relaxation of problem (1) appropriately, we are able to derive bounds on the dual variables. These bounds can be

With $S = \{3, 4\}$ we obtain from (42) the following polyhedron $P(S)$:

$$\begin{aligned} -2y_1 + 2y_2 + y_4 &\leq 8 \\ y_1 - y_2 + 2y_3 - 0.1y_4 &\leq -1.8 \\ y_1, y_2, y_3, y_4 &\geq 0. \end{aligned} \tag{43}$$

The sign pattern of the first two columns in (43) prevents us from getting tighter bounds while considering only single rows. Such sign patterns or jammed substructures can only be resolved by a more global view or specifically in this case by a two-row approach. Now using a two-row approach as in Section 3.4 gives us improved bounds of $y_1 \in [0, \infty]$, $y_2 \in [1.25, \infty]$, $y_3 \in [0, 1.1]$, and $y_4 \in [0, 5.5]$. The lower bound $\bar{\ell}_2 = 1.25 > 0$ implies that constraint $x_2 + 2x_3 - x_4 \geq 1$ becomes an equality. Finally, $c_1 - \sup((A_1)^\top y) = 2 - 1.1 > 0$ implies $x_1 = 0$.

It should be noted that it would also be possible to use the presolve method described in Section 3.3 to determine tighter bounds for the dual variables. However, this was not done, because the presolve method shown in Section 3.4 always delivers bounds that are at least as strong as the bounds determined by LP-based bound tightening on two constraints.

3.5.3 Implementation Details

In order to further improve runtime behavior for this method it is advantageous to consider only a subset of the continuous variables to tighten the bounds of the dual variables y . Consider a continuous variable $x_j, j \in \mathcal{N} \setminus \mathcal{I}$, with an explicit upper bound. We can imagine this as an additional constraint, i.e., $-x_j \geq -u_j > -\infty$. The corresponding dual variable, say y_i , is a singleton column in the dual formulation or in other words $|\text{supp}((A^\top)_{\cdot i})| = 1$. That is the bounds of y_i can only be improved in the dual constraint $(A_{\cdot j})^\top y \leq c_j$. Unfortunately, y_i has an upper bound of ∞ and thus the minimal activity (2) needed for propagating bounds for the dual variables $y_k, k \in \text{supp}(A_{\cdot j}) \setminus \{i\}$ is always $-\infty$ as y_i has a coefficient of -1 . Consequently, no new bounds can be identified from this row for the dual variables $y_k, k \in \text{supp}(A_{\cdot j}) \setminus \{i\}$. In addition, the coefficient of -1 only allows the determination of a new lower bound for y_i .

At the beginning of our implementation, it is verified which continuous variable has bounds that are implied by constraints and other variable bounds. Careful attention must be paid to interdependencies between bounds. That is if the redundancy of two bounds depends on each other only one bound is redundant. Redundant variable bounds are subsequently removed, which usually gives us more implied free variables or variables with only one bound. As free variables are assigned to dual equalities and variables with only a lower bound to dual inequalities this results mostly in a stronger dual formulation and therefore to tighter bounds on the dual variables. As described above, variables with a finite upper bound are excluded from further consideration.

Let \bar{S} be the set of continuous variables without unimplied upper bounds. Our current implementation always performs FBBT (see Section 2) on $P(\bar{S})$, for determining bounds on the dual variables y , which is essentially the approach published in [4]. In addition, the current implementation in SCIP uses the previously described two-row approach directly before FBBT. In order to exclude unfavorable situations as shown in (43) the pairing mechanism explicitly searches for jammed substructures to find matching pairs of rows. This makes it possible that the bounds calculated in the two-row approach can be exploited and further tightened in a subsequent FBBT realization.

4 Pairing Methods

A straightforward exhaustive search for suitable pairs of rows or columns to which the presolve methods presented in the previous section can be applied may be expected to be impractical due to the quadratic complexity in the number of constraints or variables. Preliminary tests during development of the methods have confirmed that running the presolvers on all possible pairs is prohibitively expensive in terms of runtime and memory requirement. As a result, many problems would run into time or memory limits during presolving. Therefore it is important to avoid excessive usage of computational resources by setting working limits and to restrict the search to pairs of rows or columns that are worthwhile to consider, see also Section 5.6. In this section, we want to outline the hashing-based pairing mechanisms that we have developed in order to filter out pairs that are likely or even guaranteed to not lead to reductions for the respective presolvers.

A detailed introduction to hashing can be found in [27]. The basic idea of our pairing mechanisms is to scan through the problem and punctually remember promising constraints or variables in a hashlist or hashtable. We then go through the problem for a second time and check whether the hashlist or hashtable contains a fitting counterpart for the constraint or variable we are currently looking at.

4.1 Pair-Search Based on a Hashtable

The presolve methods presented in Section 3.1 and Section 3.2 require a high number of matching coefficients. To achieve a favorable trade-off between search time and effectiveness, we use a pairing mechanism on variable pairs and constraint pairs respectively. In the following we explain the mechanism as used for the presolve method from Section 3.1 and highlight the differences to the pairing mechanism required for the other presolve method at the end of this section.

For each pair of variable indices j, k in each equality i , the triple $(j, k, \frac{a_{ij}}{a_{ik}})$, consisting of the variable indices and their coefficients' ratio, is used as a key to store the quintuple $(i, a_{ij}, j, a_{ik}, k)$. Efficiently storing and querying these tuples is very important for the overall runtime behavior of the algorithm. Therefore, we use the hashtable and collision resolution scheme described in [31, Sec. 2.1.8], which allows for constant time access under practical assumptions. If the key $(j, k, \frac{a_{ij}}{a_{ik}})$ is already contained in the hashtable, then the entry corresponding to the sparser row is kept. Afterwards, for each pair of variable indices j, k in each inequality r , the hashtable is queried for $(j, k, \frac{a_{rj}}{a_{rk}})$. If the conditions $a_{ij} = a_{rj}, a_{ik} = a_{rk}, W = \emptyset$ hold for the corresponding entry $(i, a_{ij}, j, a_{ik}, k)$, then reformulation (11) is applied. To further decrease search time, we use the following set of limits to heuristically prevent unrewarding investigations.

First, we consider at most 4900 variable pairs per row to be hashed and added to the hashtable. The same limit is used when searching through the inequality rows after the hashtable is built. Further, we keep track of the number of failed hashtable queries of the current row. If no reduction is found on the current row, we add the number of failed hashtable queries to a global counter. If a reduction is successfully applied to the current row we instead decrease the global counter by the number of failed hashtable queries of this row. The presolve method stops when the global counter exceeds $100 \cdot |\mathcal{M}|$. The rationale is that we want to keep going if, after many useless calls that almost exceeded the budget, we finally reach a useful section of rows. However, for the case that the useful section is all at the beginning and we quickly want to quit afterwards, we do not allow a negative build-up, i.e. the global counter remains at zero if it were to become negative.

In addition, note that a target matrix with as few nonzeros as possible does not necessarily give the best results. In fact, it seems important to preserve special structures by not applying cancellation if it would destroy one of the following properties in the row r above: integrality of the coefficients; more specifically coefficients $+1$ and -1 ; set

packing, set covering, set partitioning, or logicor constraint types; variables with no or only one *up-/downlock* [1].

Also, it should be mentioned that adding scaled equations to other constraints needs to be done with care. In particular, too large or too small scaling factors λ can lead to numerical problems. Currently, as in [4], a limit of $|\lambda| \leq 1000$ is used.

For the presolve method from Section 3.2, we consider pairs of constraint indices and their coefficients. The pairing mechanism itself is then identical with the role of rows and columns being reversed. However, we only apply the reduction if we have $|U| - |W| \geq 100$ as the reduction induces a larger computational overhead.

Finally, it is important to note that this hashing mechanism can miss possible reductions in two cases. First, a pair might not be evaluated as one of the imposed limits is hit before it was processed. Second, the hashing mechanism disregards pairs with an overlap of size one. This follows the intuition that the methods presented in Subsections 3.1 and becomes more powerful with increasing overlap.

4.2 Pair-Search Based on Multiple Hashlists

The following pairing mechanism is used for the presolve methods presented in Section 3.3 and Section 3.4. It is also used indirectly by the presolve method from Section 3.5 as it uses the method from Section 3.4 without its extension in order to derive stronger bounds from the dual problem than could be achieved by using regular FBBT. In all three cases, as highlighted by Lemma 3.1, it is important to find pairs of constraints such that variables shared by the constraints have coefficients with opposite sign. In order to do this, we devised a pairing mechanism that uses an open surjective hash function $H(j, k)$ for variable indices $j, k \in \mathcal{N}$ and works as follows.

– Step 1:

Create the four sets

$$\begin{aligned} L_{++} &:= \{(H(j, k), i) \mid (i, j, k) \in \mathcal{M} \times \mathcal{N} \times \mathcal{N} : a_{ij} > 0 \wedge a_{ik} > 0\} \\ L_{--} &:= \{(H(j, k), i) \mid (i, j, k) \in \mathcal{M} \times \mathcal{N} \times \mathcal{N} : a_{ij} < 0 \wedge a_{ik} < 0\} \\ L_{+-} &:= \{(H(j, k), i) \mid (i, j, k) \in \mathcal{M} \times \mathcal{N} \times \mathcal{N} : a_{ij} > 0 \wedge a_{ik} < 0\} \\ L_{-+} &:= \{(H(j, k), i) \mid (i, j, k) \in \mathcal{M} \times \mathcal{N} \times \mathcal{N} : a_{ij} < 0 \wedge a_{ik} > 0\}. \end{aligned}$$

– Step 2:

Separate L_{++} , and analogously the other three sets L_{--}, L_{+-}, L_{-+} , into subsets

$$L_{++}^h := \{i \mid (h, i) \in L_{++}\}$$

for each key value h appearing in L_{++} . In other words, L_{++}^h contains all indices i of rows such that some variables x_j, x_k had positive coefficients in row i and $h = H(j, k)$.

– Step 3:

For each key value h for which L_{++}^h and L_{--}^h have both been created, perform the two-row bound tightening on all row pairs $(r, s) \in L_{++}^h \times L_{--}^h, r \neq s$, as this guarantees that for this row pair there exist at least two variables x_j, x_k whose coefficients a_{rj}, a_{sj} and a_{rk}, a_{sk} have opposite sign in the constraints r and s . This is done analogously for key values h for which L_{+-}^h and L_{-+}^h have both been created. While doing so, add each processed pair's hash to a hashtable. To prevent unnecessarily processing the same pairs multiple times we query the hashtable before applying the two-row bound tightening.

In order to make this approach more efficient, several working limits are used in our implementation. First, creating the four sets $L_{++}, L_{--}, L_{+-}, L_{-+}$ for all rows and variable pairs is too computationally expensive. Therefore, for each row we consider

at most 10000 variable pairs and also limit the total number of variable pairs to be considered over all rows by $10 \cdot |\mathcal{M}|$. Second, for each processed pair we check whether new bounds have been found and stop searching if 1000 consecutively processed pairs have not yielded tighter bounds. Third, before applying two-row bound tightening on a row pair we check if its hash is already contained in the hashtable tracking the already processed pairs. If 1000 consecutive pairs have been found to already be contained in the hashtable, the method also stops. Finally, the method stops after successfully processing at most $|\mathcal{M}|$ row pairs. Even if the method finds many reductions it is important to prevent the method from taking up too much time. To do this, we chose a limit which is linear in the problem size because the number of constraints in real-world problems as well as our testset can easily range from a few hundred to multiple millions and future instance sizes will increase even further.

Like the hashing mechanism presented in Section 4.1, this hashing mechanism can miss possible reductions if one of the limits is hit or a pair has overlap of size one.

5 Computational Results

In this section we will present our computational results. First, we discuss a comparison of each individual presolve method with a baseline run where all five methods are disabled (see Sections 5.1 to 5.5). Second, we provide an evaluation of the combined impact of all five presolve methods and confirm the importance of working limits (see Section 5.6).

The methods have been implemented in SCIP 6.0.2 with SoPlex 4.0.2 as LP solver and the computations were performed on machines running on a Xeon E3-1240 v5 CPU (“Skylake”, 4 cores, HT disabled, 3.5 GHz base frequency) and 32 GB RAM. Aside from en- or disabling the five presolve methods and setting a time limit of 7200 seconds we ran SCIP on its default settings. The run with all five presolve methods being disabled is referred to as *Baseline*. (Out of the five methods described in this paper, Two-Row Nonzero Cancellation is the only one that was already included in SCIP 6.0.2.)

Our testset is the second version of the MIPLIB2017 benchmark set [22, 33] released in June 2019. It consists of 240 instances from varying optimization backgrounds. To further increase the accuracy of our results we run each instance and each setting on four different random seed initializations as well as seed zero, which is used in the release version of SCIP. SCIP uses the random seed initializations to affect numerical perturbations in the LP solver, for tie breakings as well as heuristic decision rules that necessarily occur in many parts of the solving process. Each combination of instance and seed is considered its own data point to tackle the problem of *performance variability* [16, 30] where small changes which are neutral from a purely mathematical point of view have a large impact on the performance of a MIP solver. Finally, we discard all combinations of instance and seed where computations ran into numerical difficulties or, as happened in rare cases, the new code revealed bugs in the existing framework.

The result tables in this section each present the comparison between two different runs and are structured as follows. Each row presents the aggregate results on a set of problem instances. The row *all* contains the entire testset. The subsets corresponding to the brackets $[t, T]$ contain instances that were solved by at least one run and for which the maximum solving time (among both runs) lies between t seconds and T seconds. If a configuration did not solve the problem within the time limit it is recorded as T seconds. In this work, we set T to the time limit of 7200 seconds and with increasing t , this provides a hierarchy of subsets by difficulty. In particular, the bracket $[0, 7200]$ contains all instances that have been solved within the time limit and deserves special attention as it is the largest subset where computation times can be compared in a meaningful way. We then compare the number of instances that had been solved, as well as the shifted geometric mean of solving times and branch-and-bound nodes in the columns

solved, time and nodes respectively. The *shifted geometric mean* of values x_1, \dots, x_n is

$$\left(\prod_{i=1}^n (x_i + s) \right)^{\frac{1}{n}} - s,$$

where the shift s is set to 1 second and 100 nodes respectively.

5.1 Two-Row Nonzero Cancellation

For a detailed description of this method we refer to Section 3.1. The method found reductions on 390 instances. Despite the method being applied to a large number of instances its performance impact, as presented in Table 1, is slightly negative in terms of runtime and also causes three fewer instances to be solved within the time limit. Although the impact is still only neutral on the subset [100,7200] the results show a tendency towards a positive impact on difficult instances.

Table 1: Performance comparison for Two-Row Nonzero Cancellation

Subset	instances	Baseline			TwoRowNonzeroCancel			relative	
		solved	time	nodes	solved	time	nodes	time	nodes
all	1196	562	1368.7	7023	559	1385.1	7023	1.01	1.00
[0,7200]	570	562	220.3	3325	559	225.9	3313	1.03	1.00
[1,7200]	556	548	249.7	3601	545	256.2	3586	1.03	1.00
[10,7200]	516	508	345.0	4667	505	353.8	4645	1.03	1.00
[100,7200]	379	371	844.9	10543	368	857.6	10522	1.02	1.00
[1000,7200]	195	187	2219.9	36162	184	2213.7	36162	1.00	1.00

If a presolve method does not perform well, this is usually due to one or both of the following two reasons. First, the method itself takes too much time and therefore causes a degradation on models where it does not find enough reductions. Second, the reductions create a new substructure that is unfavorable for subsequent solves. In our case both reasons appear to play a role. From all of the presented presolve techniques, this method creates the largest runtime overhead. However, this does not suffice to fully explain the degradation observed in Table 1.

5.2 Two-Column Nonzero Cancellation

This presolve method was described in detail in Section 3.2. Since the rules for applying a reduction in this method are more restrictive than in the row version we observe reductions on only 115 instances. The performance results for this presolve method are presented in Table 2. In total, its impact can be considered neutral with two more instances being solved at the cost of a slightly prolonged runtime for instances in the subset [1000,7200].

Table 2: Performance comparison for Two-Column Nonzero Cancellation

Subset	instances	Baseline			TwoColNonzeroCancel			relative	
		solved	time	nodes	solved	time	nodes	time	nodes
all	1186	562	1354.6	7202	564	1353.2	7244	1.00	1.01
[0,7200]	567	562	218.0	3343	564	217.5	3346	1.00	1.00
[1,7200]	552	547	249.3	3647	549	248.8	3650	1.00	1.00
[10,7200]	507	502	358.6	4945	503	357.7	4947	1.00	1.00
[100,7200]	375	370	844.0	11159	372	846.8	11200	1.00	1.00
[1000,7200]	183	178	2304.5	41801	180	2316.7	40902	1.01	0.98

In contrast to the neutral performance impact on the MIPLIB 2017 benchmark set, it has a tremendous impact on the testset studied in [29]. These instances are constructed by [29] based on a deterministic equivalent mixed-integer programming formulation of the two-sided chance constrained program (TSCCP). In total, 30 instances are publicly available at [28]. Additional tests were performed on these instances using the same settings as above. Similar to other tests, each instance was solved five times with different random seed initializations, resulting in a testset consisting of 150 MIPs.

The computational results presented in Table 3 show that the proposed method found reductions on all instances. Overall, the proposed method improves performance by 74% on the TSCCP testset. In addition, with this method, the solver was able to solve 147 instances within the time limit of 7200 seconds. In comparison, without this method the solver was able to solve only 98 instances within the same time limit.

Table 3: Performance comparison for Two-Column Nonzero Cancellation on the TSCCP testset

Subset	instances	Baseline			TwoColNonzeroCancel			relative	
		solved	time	nodes	solved	time	nodes	time	nodes
all	150	98	1205.1	75480	147	318.7	2956	0.26	0.04
[0,7200]	147	98	1161.9	72720	147	299.0	2646	0.26	0.04
[1,7200]	147	98	1161.9	72720	147	299.0	2646	0.26	0.04
[10,7200]	147	98	1161.9	72720	147	299.0	2646	0.26	0.04
[100,7200]	133	84	1577.7	110550	133	399.5	3317	0.25	0.03
[1000,7200]	84	35	4619.3	357378	84	448.4	5030	0.10	0.01

5.3 LP-based Bound Tightening on Two Constraints

For a detailed description of this presolve method we refer to Section 3.3. The method applied reductions to 119 instances and the performance results are presented in Table 4. On the entire testset, the method has neutral impact. However, with increasing difficulty we can see a positive impact of up to 4% for the instances in the subset [1000,7200].

Table 4: Performance comparison for LP-based Bound Tightening on Two Constraints

Subset	instances	Baseline			LPBound			relative	
		solved	time	nodes	solved	time	nodes	time	nodes
all	1192	563	1361.0	7154	563	1356.8	7123	1.00	1.00
[0,7200]	567	563	216.3	3410	563	214.8	3393	0.99	1.00
[1,7200]	552	548	247.4	3722	548	245.7	3702	0.99	1.00
[10,7200]	504	500	363.3	5142	500	360.6	5111	0.99	0.99
[100,7200]	367	363	882.1	12430	363	872.9	12323	0.99	0.99
[1000,7200]	181	177	2345.0	48251	177	2245.4	45365	0.96	0.94

5.4 Bound Tightening on Convex Combinations of Two Constraints

The extension of this presolve method presented in Section 3.4 is incorporated using SCIPs so-called *clique table* which represents the solvers current knowledge of binary variables that may not be set to one at the same time. The clique table is generated from set packing constraints explicitly stated in the model as well as information extracted during earlier presolve and is also used for cutting plane separation, linear constraint propagation, and primal heuristics (see [21]). From this clique table we generate the set packing constraints used in the extension. It is important to keep in mind that, although this allows the method to find better bounds than the method presented in Section 3.3,

it also increases the cost in terms of running time. In our implementations, this amounts to an average of 0.11 seconds being spent per instance for the method from Section 3.3, which increases to an average of 0.31 seconds being spent per instance for this method. Note, however that this is in part due to this method running longer as it finds more reductions.

As mentioned in Remark 3.11 this method works on a superset of the variables for which LP-based bound tightening on two constraints from Section 3.3 can tighten bounds on. As a result this method applies reductions in 142 instances, i.e. 23 more instances than for LP-based bound tightening. The results presented in Table 5 show that the additional bound tightenings found by this method indeed yield a positive impact on solving times. As for LP-based bound tightening the impact of this presolve method scales with instance difficulty. As a result its impact changes from neutral on the overall testset towards a considerable 6% improvement on the subset [1000,7200].

Table 5: Performance comparison for ConvComb

Subset	instances	Baseline			ConvComb			relative	
		solved	time	nodes	solved	time	nodes	time	nodes
all	1196	563	1368.6	7026	563	1359.7	7026	0.99	1.00
[0,7200]	566	563	214.9	3382	563	212.0	3385	0.99	1.00
[1,7200]	551	548	245.9	3691	548	242.4	3694	0.99	1.00
[10,7200]	503	500	361.1	5099	500	355.3	5101	0.98	1.00
[100,7200]	368	365	866.1	12074	365	845.3	12035	0.98	1.00
[1000,7200]	180	177	2326.3	47019	177	2194.8	44973	0.94	0.96

The results presented in Table 6 illustrate the benefit of the clique extension over the standard procedure developed by Belotti [8]. It can be seen that it has an overall positive impact in terms of runtime. However, we have observed that the overall number of nodes increases slightly when using the extension. In particular, there are ten instances where the node count increases by more than half and six instances where the node count more than doubles. On those ten instances we also observed that significantly more time is spent on branching and the branching tree depth increases as well. Finally, we want to note that when deactivating the clique extension the number of instances where reductions are found drops from 142 to 119.

Table 6: Performance comparison for ConvComb without clique extension

Subset	instances	ConvCombNoClique			ConvComb			relative	
		solved	time	nodes	solved	time	nodes	time	nodes
all	1199	563	1375.2	6966	563	1365.3	7030	0.99	1.01
[0,7200]	566	563	215.2	3390	563	212.0	3421	0.99	1.01
[1,7200]	551	548	246.2	3700	548	242.4	3735	0.98	1.01
[10,7200]	500	497	369.1	5235	497	362.9	5288	0.98	1.01
[100,7200]	368	365	863.4	12021	365	845.4	12186	0.98	1.01
[1000,7200]	177	174	2318.6	43270	174	2258.5	44599	0.97	1.03

5.5 Exploiting Complementary Slackness on Two Columns of Continuous Variables

For a detailed description of this presolve method we refer to Section 3.5. As can be seen in Table 7, the impact of the method is almost neutral on the overall testset as well as all subsets. The method performs reductions on only 11 instances. However, the neutral impact is not necessarily due to the inefficiency of the reductions themselves, but rather a consequence of the very low number of instances where this method can be applied because of missing continuous variables.

Table 7: Performance comparison for CompSlack

Subset	instances	Baseline			CompSlack			relative	
		solved	time	nodes	solved	time	nodes	time	nodes
all	1197	563	1370.5	7025	563	1366.4	7023	1.00	1.00
[0,7200]	563	563	210.9	3250	563	209.6	3250	0.99	1.00
[1,7200]	549	549	239.2	3521	549	237.6	3521	0.99	1.00
[10,7200]	503	503	347.4	4774	503	345.0	4775	0.99	1.00
[100,7200]	362	362	867.2	11835	362	858.6	11836	0.99	1.00
[1000,7200]	176	176	2299.7	44241	176	2270.6	44294	0.99	1.00

On a more specialized testset of real-world supply chain instances, however, we have observed this presolve method to consistently produce more reductions. In [36], a testset of 40 real-world supply chain instances was studied. The instances contain 330108 variables and 145450 constraints in arithmetic mean. The proportion of discrete variables is about 3.9%. We performed additional tests on these instances with the same settings as above. These instances are extremely challenging and cannot be solved to optimality with or without this presolve method, but the applicability of the method and the computational efficiency of the hashing-based pairing mechanism becomes clear.

The method found reductions on 35 of the 40 instances, i.e., on 87.5% of the testset. On 6 instances more than 1% of the variables were fixed and in one instance we were able to fix 14% of all variables. Additionally, on 3 instances more than 2% percent of the inequality constraints were changed to equalities. The runtime of this presolve method was below one second for 29 instances, between one and six seconds on nine instances and close to 12 seconds on two instances. As already mentioned, these supply chain instances contain a very high percentage of continuous variables, which is one of the reasons why this presolve method is able to find a reasonable amount of reductions.

5.6 All Presolve Methods

In order to evaluate whether and how the methods interact, we ran all presolvers presented in this work in the following order: LPBound, CompSlack, ConvComb, TwoRowNonzeroCancel, TwoColNonzeroCancel. As previously illustrated in Example 3.4, different presolve methods can positively interact with each other, even if each method itself is neutral or even slightly negative. However, the results presented in Table 8 show that enabling all five presolve methods presented in this work has a slightly negative impact on solving times.

Table 8: Performance comparison using all presolve methods

Subset	instances	Baseline			All			relative	
		solved	time	nodes	solved	time	nodes	time	nodes
all	1181	563	1340.0	7309	568	1349.7	7293	1.01	1.00
[0,7200]	576	563	228.5	3549	563	231.9	3547	1.02	1.00
[1,7200]	561	548	261.1	3872	548	265.2	3867	1.02	1.00
[10,7200]	523	510	354.6	4986	510	359.3	4978	1.01	1.00
[100,7200]	390	377	842.9	10969	377	847.7	11024	1.01	1.01
[1000,7200]	204	191	2219.5	37948	191	2158.0	36013	0.97	0.95

Table 9 shows the number of instances each of the presolvers applied reductions to when run individually (left) and with all five methods enabled (right). We can make two noteworthy observations. First, we have an additional confirmation that the presolve methods presented in this work do not enable each other to find additional reductions, i.e. each presolve method finds more reductions when run with the other four being disabled. Second, bound tightening based on convex combinations applies reductions

on 99 instances even when run after LP-based bound tightening. This shows that it is indeed the more powerful method.

Table 9: Number of instances with reductions of each presolve method

Presolve method	single	all
TwoRowNonzeroCancel	390	384
TwoColNonzeroCancel	115	113
LPBound	119	117
ConvComb	142	99
CompSlack	11	10

Finally, we would like to demonstrate empirically how important working limits are for controlling the computational effort. While a comparison with exhaustive search on all pairs of variables and constraints is computationally out-of-scope, we conducted a small experiment where we removed only the working limits described in Section 4. Due to limited computational resources we only ran this test on the original 240 instances of the MIPLIB 2017 benchmark set. However, these results already demonstrate the performance deterioration quite clearly.

The removal of working limits changed the number of variables or constraints after presolve only on 36 out of 240 instances; most changes were negligible, only on 14 instances the presolved models became at least 1% smaller when disabling working limits. By contrast, the average presolving time increased by a factor of five, where on 41 instances, the presolving time increased by a factor of ten and on the worst instances we observed a factor of more than 100. Further, without working limits an additional 31 instances ran into memory issues.

Before closing our presentation of computational results, we want to add that for practical development of a fast and stable MIP solver, there are three reasons to include presolve methods that show only neutral or even slightly negative impact on the MIPLIB 2017 benchmark set. First, there may be positive interactions between new and existing presolve methods. However, this was not the case for among the five methods presented in this work. Second, presolve methods with neutral or slightly negative performance impact in the benchmark set may have significant impact on other instances, as we have seen on the two-sided chance-constrained program instances for the technique described in Section 3.2 and for real-world supply chain instances for the technique described in Section 3.5. Finally, it is also important to actively involve new presolve methods in the development process of a solver to avoid overtuning of established methods. It could be, for example, that a new presolve method, which is not performing well at the moment, will reveal its potential in combination with a newly developed heuristic in the future. If this method had not been included in the development process, this would not have been apparent.

6 Conclusions

In this paper, we considered five two-row and two-column methods for mixed-integer presolve that are generally effective, but suffer from the limitation that simply testing all methods on all pairs of rows or columns is computationally not appropriate. We resolved this bottleneck by introducing two hashing-based pairing mechanisms that identify promising pairs of rows and columns quickly. During discussion of the five two-row and two-column presolve methods we focused on the required properties of promising pairs of rows or columns. Additionally we discussed an extension based on set packing constraints to further strengthen the results of the bound tightening method applied to convex combinations of two rows. We then introduced two hashing-based

pairing mechanisms, implemented the ideas in the mixed-integer programming solver SCIP and conducted a performance evaluation on the MIPLIB 2017 benchmark set [33].

The first pairing mechanism, based on a hashtable, aimed at pairs of rows (or columns) such that there exist at least two coefficients that can be cancelled simultaneously by linearly combining the rows (or columns). This is required by the two nonzero cancellation methods, which turned out to have neutral or slightly negative effect on the performance.

The second pairing mechanism uses four hashlists to find pairs of rows such that there exist at least two variables with opposite sign. This increases the chance of finding stronger variable bounds when applying the two-row bound strengthening methods presented in Sections 3.3 and 3.4. Using bound tightening based on convex combinations, we were able to improve solving times by 1% on the entire testset and by up to a considerable 6% for instances with longer solving times.

Finally, we experimented with applying the method presented in Section 3.4 to improve the bound strengthening required during the dual presolve method presented in Section 3.5, but observed that this leads to only few reductions and no performance impact. This is largely due to the fact that many of the MIPLIB 2017 instances do not contain any continuous variables at all.

To conclude, the results of this paper demonstrate that two-row and two-column presolve methods hold potential for practically solving challenging mixed-integer programming problems when they are combined with a sophisticated selection mechanism. For the methods discussed in this paper, more extensive parameter tuning or extensions of the methods themselves may even lead to larger performance gains.

Other compelling directions for further investigation are finding new combinations of pairing mechanisms and two-row or two-column presolve methods as well as moving on to reductions based on three or even more rows or columns.

References

- [1] T. Achterberg. *Constraint Integer Programming*. PhD thesis, Technische Universität Berlin, 2007.
- [2] T. Achterberg and R. Wunderling. *Mixed Integer Programming: Analyzing 12 Years of Progress*, pages 449–481. Springer, 01 2013. ISBN 978-3-642-38188-1. doi: 10.1007/978-3-642-38189-8_18.
- [3] T. Achterberg, R. E. Bixby, Z. Gu, E. Rothberg, and D. Weninger. Multi-Row Presolve Reductions in Mixed Integer Programming. In T. Hosei University, editor, *Proceedings of the Twenty-Sixth RAMP Symposium*, pages 181–196, 2014. URL <http://www.orsj.or.jp/ramp/2014/paper/4-4.pdf>.
- [4] T. Achterberg, R. E. Bixby, Z. Gu, E. Rothberg, and D. Weninger. Presolve reductions in mixed integer programming. *INFORMS Journal on Computing*, 2019. doi: 10.1287/ijoc.2018.0857.
- [5] E. D. Andersen and K. D. Andersen. Presolving in linear programming. *Mathematical Programming*, 71:221–245, 1995. ISSN 0025-5610. doi: 10.1007/BF01586000. URL <http://dx.doi.org/10.1007/BF01586000>.
- [6] A. Atamtürk and M. W. P. Savelsbergh. Integer-programming software systems. *Annals of Operations Research*, 140:67–124, 2005. ISSN 0254-5330. doi: 10.1007/s10479-005-3968-2. URL <http://dx.doi.org/10.1007/s10479-005-3968-2>.
- [7] E. Balas and E. Zemel. An algorithm for large zero-one knapsack problems. *operations Research*, 28(5):1130–1154, 1980.
- [8] P. Belotti. Bound reduction using pairs of linear inequalities. *Journal of Global Optimization*, 56(3):787–819, 2013.

- [9] P. Belotti, S. Cafieri, J. Lee, and L. Liberti. Feasibility-based bounds tightening via fixed points. In *International Conference on Combinatorial Optimization and Applications*, pages 65–76. Springer, 2010.
- [10] R. Bixby and D. Wagner. A note on detecting simple redundancies in linear systems. *Operations Research Letters*, 6(1):15 – 17, 1987. ISSN 0167-6377. doi: [https://doi.org/10.1016/0167-6377\(87\)90004-6](https://doi.org/10.1016/0167-6377(87)90004-6). URL <http://www.sciencedirect.com/science/article/pii/0167637787900046>.
- [11] R. E. Bixby and E. Rothberg. Progress in computational mixed integer programming—a look back from the other side of the tipping point. *Annals of Operations Research*, 149: 37–41, 2007. ISSN 0254-5330. doi: 10.1007/s10479-006-0091-y. URL <http://dx.doi.org/10.1007/s10479-006-0091-y>.
- [12] R. E. Bixby, M. Fenelon, Z. Gu, E. Rothberg, and R. Wunderling. Mixed-integer programming: A progress report. In M. Grötschel, editor, *The Sharpest Cut: The Impact of Manfred Padberg and His Work*, MPS-SIAM Series on Optimization, chapter 18, pages 309–325. SIAM, 2004.
- [13] A. L. Brearley, G. Mitra, and H. P. Williams. Analysis of mathematical programming problems prior to applying the simplex algorithm. *Mathematical Programming*, 8:54–83, 1975. ISSN 0025-5610. doi: 10.1007/BF01580428. URL <http://dx.doi.org/10.1007/BF01580428>.
- [14] S. F. Chang and S. T. McCormick. Implementation and computational results for the hierarchical algorithm for making sparse matrices sparser. *ACM Transactions on Mathematical Software*, 19(3):419–441, 1993. doi: 10.1145/155743.152620.
- [15] H. Crowder, E. L. Johnson, and M. Padberg. Solving Large-Scale Zero-One Linear Programming Problems. *Operations Research*, 31(5):803–834, 1983. ISSN 0030364X. doi: 10.2307/170888. URL <http://dx.doi.org/10.2307/170888>.
- [16] E. Danna. Performance variability in mixed integer programming, 2008. Presentation at Workshop on Mixed Integer Programming.
- [17] G. B. Dantzig. Discrete-variable extremum problems. *Operations research*, 5(2):266–288, 1957.
- [18] E. Davis. Constraint propagation with interval labels. *Artificial intelligence*, 32(3):281–331, 1987.
- [19] G. Gamrath, T. Koch, A. Martin, M. Miltenberger, and D. Weninger. Progress in presolving for mixed integer programming. *Mathematical Programming Computation*, 7:367–398, 2015. doi: 10.1007/s12532-015-0083-5.
- [20] G. Gamrath, T. Fischer, T. Gally, A. Gleixner, G. Hendel, T. Koch, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, S. Schenker, R. Schwarz, F. Serrano, Y. Shinano, S. Vigerske, D. Weninger, M. Winkler, J. T. Witt, and J. Witzig. The scip optimization suite 3.2. Technical Report 15-60, ZIB, Takustr. 7, 14195 Berlin, 2016.
- [21] A. Gleixner, L. Eifler, T. Gally, G. Gamrath, P. Gemander, R. L. Gottwald, G. Hendel, C. Hojny, T. Koch, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, F. Schlösser, F. Serrano, Y. Shinano, J. M. Viernickel, S. Vigerske, D. Weninger, J. T. Witt, and J. Witzig. The SCIP Optimization Suite 5.0. Technical report, Optimization Online, December 2017. http://www.optimization-online.org/DB_HTML/2017/12/6385.html.
- [22] A. Gleixner, G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P. M. Christophel, K. Jarck, T. Koch, J. Linderoth, M. Lübbecke, H. D. Mittelmann, D. Ozyurt, T. K. Ralphs, D. Salvagnin, and Y. Shinano. MIPLIB 2017: Data-Driven Compilation of the 6th Mixed-Integer Programming Library. Technical report, Optimization Online, July 2019. http://www.optimization-online.org/DB_HTML/2019/07/7285.html.

- [23] J. Gondzio. Presolve analysis of linear programs prior to applying an interior point method. *INFORMS Journal on Computing*, 9(1):73–91, 1997. doi: 10.1287/ijoc.9.1.73. URL <https://doi.org/10.1287/ijoc.9.1.73>.
- [24] M. Guignard and K. Spielberg. Logical reduction methods in zero-one programming: Minimal preferred variables. *Operations Research*, 29(1):49–74, 1981.
- [25] K. L. Hoffman and M. Padberg. Improving LP-Representations of Zero-One Linear Programs for Branch-and-Cut. *ORSA Journal on Computing*, 3(2):121–134, 1991. URL <http://dx.doi.org/10.1287/ijoc.3.2.121>.
- [26] E. L. Johnson and U. H. Suhl. Experiments in integer programming. *Discrete Applied Mathematics*, 2(1):39–55, 1980. ISSN 0166-218X. doi: [http://dx.doi.org/10.1016/0166-218X\(80\)90053-0](http://dx.doi.org/10.1016/0166-218X(80)90053-0). URL <http://www.sciencedirect.com/science/article/pii/0166218X80900530>.
- [27] D. E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998. ISBN 0-201-89685-0.
- [28] S. Küçükyavuz. Tscpp testset. <http://faculty.washington.edu/simge/IntMixOS.zip>.
- [29] X. Liu, F. Kılınç-Karzan, and S. Küçükyavuz. On intersection of two mixing sets with applications to joint chance-constrained programs. *Mathematical Programming*, 175(1): 29–68, 2019.
- [30] A. Lodi and A. Tramontani. Performance variability in mixed-integer programming. *Tutorials in Operations Research*, pages 1–12, 2013. doi: 10.1287/educ.2013.0112.
- [31] S. J. Maher, T. Fischer, T. Gally, G. Gamrath, A. Gleixner, R. L. Gottwald, G. Hendel, T. Koch, M. E. Lübbecke, M. Miltenberger, et al. The scip optimization suite 4.0. Technical report, ZIB, Takustr. 7, 14195 Berlin, 2017.
- [32] A. Martin. General mixed integer programming: Computational issues for branch-and-cut algorithms. In D. Naddef and M. Jünger, editors, *Computational combinatorial optimization*, volume 2241, pages 1–25, Berlin, 2001. Springer. doi: 10.1007/3-540-45586-8_1.
- [33] miplib2017. MIPLIB 2017, 2018. <http://miplib.zib.de>.
- [34] I. Quesada and I. E. Grossmann. Global optimization of bilinear process networks with multicomponent flows. *Computers & Chemical Engineering*, 19(12):1219–1242, 1995.
- [35] M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994.
- [36] L. Schewe, M. Schmidt, and D. Weninger. A Decomposition Heuristic for Mixed-Integer Supply Chain Problems. *Operations Research Letters*, 2020. doi: 10.1016/j.orl.2020.02.006.
- [37] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986. ISBN 0-471-90854-1.
- [38] J. P. Shectman and N. V. Sahinidis. A finite algorithm for global minimization of separable concave programs. *Journal of Global Optimization*, 12:1–36, 1998.
- [39] P. Sinha and A. A. Zoltners. The multiple-choice knapsack problem. *Operations Research*, 27(3):503–515, 1979.
- [40] U. Suhl and R. Szymanski. Supernode processing of mixed-integer models. *Computational Optimization and Applications*, 3(4):317–331, 1994. ISSN 0926-6003. doi: 10.1007/BF01299207. URL <http://dx.doi.org/10.1007/BF01299207>.