Proceedings of the Twenty-Sixth RAMP Symposium
Hosei University, Tokyo, October 16-17, 2014

Multi-Row Presolve Reductions in Mixed Integer Programming

Tobias Achterberg[1*], Robert E. Bixby[1†], Zonghao Gu[1‡], Edward Rothberg[1§], and Dieter Weninger[2¶]

Abstract Mixed integer programming has become a very powerful tool for modeling and solving real-world planning and scheduling problems, with the breadth of applications appearing to be almost unlimited. A critical component in the solution of these mixed-integer programs is a set of routines commonly referred to as presolve. Presolve can be viewed as a collection of preprocessing techniques that reduce the size of and, more importantly, improve the "strength" of the given model formulation, that is, the degree to which the constraints of the formulation accurately describe the underlying polyhedron of integer-feasible solutions. In the Gurobi commercial mixed-integer solver, the presolve functionality has been steadily enhanced over time, now including a number of so-called *multi-row* reductions. These are reductions that simultaneously consider multiple constraints to derive improvements in the model formulation. In this paper we give an overview of such multi-row techniques and present computational results to assess their impact on overall solver performance.

Keywords mixed integer programming, presolving, Gurobi

1. Introduction

Presolve for mixed integer programming (MIP) is a set of routines that remove redundant information and strengthen a given model formulation with the aim of accelerating the subsequent solution process. Presolve can be very effective; indeed, in some cases it is the difference between a problem being intractable and solvable [2].

In this paper we differentiate between two distinct stages or types of presolve. *Root presolve* refers to the part of presolve that is typically applied before the solution of the first linear programming relaxation, while *node presolve* refers to the additional presolve reductions that are applied at the nodes of the branch-and-bound tree.

Several papers on the subject of presolve have appeared over the years, though the total number of such publications is surprisingly small given the importance of this subject. One of the earliest and most important contributions was the paper by Brearly et al. [9].

1 Gurobi Optimization, 3733-1 Westheimer Rd. 1001, Houston, Texas 77027 USA
2 University Erlangen-Nürnberg, Department Mathematik, Lehrstuhl für Wirtschaftsmathematik, Cauerstraße 11, 91058 Erlangen
∗ E-mail address: achterberg@gurobi.com
† E-mail address: bixby@gurobi.com
‡ E-mail address: gu@gurobi.com
§ E-mail address: rothberg@gurobi.com
¶ E-mail address: dieter.weninger@math.uni-erlangen.de

This paper describes techniques for removing redundant rows, variable fixings, generalized upper bounds, and more. Presolve techniques for zero-one inequalities were investigated by Guignard and Spielberg [13], Johnson and Suhl [16], Crowder et al. [11], and Hoffman and Padberg [14]. Andersen and Andersen [3] published results in the context of linear programming and Savelsbergh [18] investigated preprocessing and probing techniques for general MIP problems. Details on the implementation of various presolve reductions are discussed in Suhl and Szymanski [20], Atamtürk and Savelsbergh [5], and Achterberg [1].

Investigations on the performance impact of different features of the CPLEX MIP solver were published in [7] and [8], and most recently in [2]. One important conclusion was that presolve together with cutting plane techniques are by far the most important individual tools contributing to the power of modern MIP solvers.

This paper focuses on one particular subset of the root presolve algorithms that are employed in the Gurobi commercial MIP solver. We designate these reductions as *multi-row* presolve methods. They derive improvements to the model formulation by simultaneously examining multiple problem constraints. They are usually more complex than "single-row" reductions. In addition, they often present significant trade-offs between effectiveness and computational overhead. However, despite the associated overhead, they provide on average a significant performance improvement for solving MIPs.

The paper is organized as follows. Section 2 introduces the primary notation used in the remainder of the paper. Sections 3 to 7 constitute the main part of the paper; they describe the individual components of the Gurobi multi-row presolve engine. Finally, Section 8 provides computational results to assess the performance impact of the algorithms covered in this paper. We summarize our conclusions in Section 9.

2. Notation

Definition 1. *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* $x \in \mathbb{Z}^I \times \mathbb{R}^{N\setminus I}$ *with* $I \subseteq N = \{1,\ldots,n\}$*, and relations* $\circ_i \in \{=,\leq,\geq\}$ *for every row* $i \in M = \{1,\ldots,m\}$ *of* A*, then the optimization problem* $MIP = (M, N, I, A, b, c, \circ, \ell, u)$ *defined as*

$$\begin{array}{ll} \min & c^T x \\ s.t. & Ax \circ b \\ & \ell \leq x \leq u \end{array} \tag{1}$$

is called a mixed integer program (MIP).

Note that in the above definition, we have employed the convention that, unless otherwise specified, vectors such as c and x, when viewed as matrices, are considered *column vectors*, that is, as matrices with a single column. Similarly, a *row vector* is a matrix with a single row.

The elements of the matrix A are denoted by a_{ij}, $i \in M$, $j \in N$. We use the notation $A_{i\cdot}$ to identify the row matrix given by the i-th row of A. Similarly, $A_{\cdot j}$ is the column matrix given by the j-th column of A. $\operatorname{supp}(A_{i\cdot}) = \{j \in N : a_{ij} \neq 0\}$ denotes the *support* of $A_{i\cdot}$,

and $\text{supp}(A_{\cdot j}) = \{i \in M : a_{ij} \neq 0\}$ denotes the *support* of $A_{\cdot j}$. For a subset $S \subseteq N$ we define A_{iS} to be the vector $A_{i\cdot}$ restricted to the indices in S. Similarly, x_S denotes the vector x restricted to S.

Depending on the bounds ℓ and u of the variables and the coefficients, we can calculate a *minimal activity* and a *maximal activity* for every row i of problem (1). Because lower bounds $\ell_j = -\infty$ and upper bounds $u_j = \infty$ are allowed, infinite minimal and maximal row activities may occur. To ease notation we thus introduce the following conventions:

$$\begin{aligned} \infty + \infty &:= \infty & s \cdot \infty &:= \infty \quad \text{for } s > 0 \\ -\infty - \infty &:= -\infty & s \cdot \infty &:= -\infty \text{ for } s < 0 \end{aligned}$$

We can then define the minimal activity of row i by

$$\inf\{A_{i\cdot}x\} := \sum_{\substack{j \in N \\ a_{ij} > 0}} a_{ij}\ell_j + \sum_{\substack{j \in N \\ a_{ij} < 0}} a_{ij}u_j \tag{2}$$

and the maximal activity by

$$\sup\{A_{i\cdot}x\} := \sum_{\substack{j \in N \\ a_{ij} > 0}} a_{ij}u_j + \sum_{\substack{j \in N \\ a_{ij} < 0}} a_{ij}\ell_j \tag{3}$$

In an analogous fashion we define the minimal and maximal activity $\inf\{A_{iS}x_S\}$ and $\sup\{A_{iS}x_S\}$ of row i on a subset $S \subseteq N$ of the variables.

3. Redundancy detection

A constraint $r \in M$ of a MIP is called *redundant* if the solution set of the MIP stays identical when the constraint is removed from the problem:

Definition 2. *Given a* $MIP = (M, N, I, A, b, c, \circ, \ell, u)$ *of form* (1)*, let*

$$P_{MIP} = \{x \in \mathbb{Z}^I \times \mathbb{R}^{N \setminus I} : Ax \circ b, \ell \leq x \leq u\}$$

be its set of feasible solutions. For $r \in M$ *let* MIP_r *be the sub-MIP obtained by deleting constraint* r*, i.e.,*

$$MIP_r = (M \setminus \{r\}, N, I, A_{S\cdot}, b_S, c, \circ_S, \ell, u)$$

with $S = M \setminus \{r\}$*. Then, constraint* r *is called* redundant *if* $P_{MIP} = P_{MIP_r}$*.*

Note that deciding whether a given constraint r is non-redundant is $\mathcal{NP}$-complete, because deciding feasibility of a MIP is $\mathcal{NP}$-complete: given a MIP we add an infeasible row $0 \leq -1$. This infeasible row is non-redundant in the extended MIP if and only if the original MIP is feasible.

Moreover, removing MIP-redundant rows may actually hurt the performance of the MIP solver, because this can weaken the LP relaxation. For example, cutting planes are computationally very useful constraints, see for example [7] and [2], but they are redundant for the MIP: they do not alter the set of integer feasible solutions; they only cut off parts of the solution space of the LP relaxation. Consequently, redundancy detection is mostly concerned

with identifying constraints that are even redundant for the LP relaxation of the MIP.

Identifying whether a constraint is redundant for the LP relaxation can be done in polynomial time. Namely, for a given inequality $A_{r\cdot}x \leq b_r$ it amounts to solving the LP

$$b_r^\star := \max\{A_{r\cdot}x : A_{S\cdot}x \circ_S b_S, \ell \leq x \leq u\}$$

with $S = M\backslash\{r\}$, i.e., $A_{S\cdot}x \circ_S b_S$ representing the sub-system obtained by removing constraint r. The constraint is redundant for the LP relaxation if and only if $b_r^\star \leq b_r$.

Even though it is polynomial, solving an LP for each constraint in the problem is usually still too expensive to be useful in practice. Therefore, Gurobi solves full LPs for redundancy detection only in special situations when it seems to be effective for the problem instance at hand. Typically, we employ simplified variants of the LP approach by considering only a subset of the constraints.

The most important special case is the one where only the bounds of the variables are considered. This is the well-known "single-row" redundancy detection based on the supremum of the row activity as defined by equation (3): an inequality $A_{r\cdot}x \leq b_r$ is redundant if $\sup\{A_{r\cdot}x\} \leq b_r$. The next, but already much more involved, case is to use one additional constraint to prove redundancy of a given inequality. Here, the most important version is the case of parallel rows, which will be discussed in Section 4. Finally, one may also use structures like cliques or implied cliques to detect redundant constraints in a combinatorial way.

Another case of multi-row redundancy detection is to find linear dependent equations within the set of equality constraints. This can be done by using a so-called *rank revealing LU factorization* as it is done by Miranian and Gu [17]. If there is a row $0 = 0$ in the factorized system, the corresponding row in the original system is linear dependent and can be removed. If the factorized system contains a row $0 = b_r$ with $b_r \neq 0$, then the problem is infeasible.

4. Parallel and nearly parallel rows

A special case of redundant constraints are two rows that are identical up to a positive scalar. Such redundancies are identified as part of the so-called parallel row detection. Two rows $q, r \in M$ are called *parallel* if $A_{q\cdot} = sA_{r\cdot}$ for some $s \in \mathbb{R}$, $s \neq 0$. If q and r are parallel, then the following holds:

1. If both constraints are equations

$$A_{q\cdot}x = b_q$$
$$A_{r\cdot}x = b_r$$

   then r can be discarded if $b_q = sb_r$. The problem is infeasible if $b_q \neq sb_r$.
2. If exactly one constraint is an equation

$$A_{q\cdot}x = b_q$$
$$A_{r\cdot}x \leq b_r$$

then r can be discarded if $b_q \leq sb_r$ and $s > 0$, or if $b_q \geq sb_r$ and $s < 0$. The problem is infeasible if $b_q > sb_r$ and $s > 0$, or if $b_q < sb_r$ and $s < 0$.

3. If both constraints are inequalities

$$A_{q\cdot}x \leq b_q$$
$$A_{r\cdot}x \leq b_r$$

then r can be discarded if $b_q \leq sb_r$ and $s > 0$. On the other hand, q can be discarded if $b_q \geq sb_r$ and $s > 0$. For $s < 0$, the two constraints can be merged into a *ranged row* $sb_r \leq A_{q\cdot}x \leq b_q$ if $b_q > sb_r$ and into an equation $A_{q\cdot}x = b_q$ if $b_q = sb_r$. The problem is infeasible if $b_q < sb_r$ and $s < 0$.

Similar to what is described by Andersen and Andersen [3], the detection of parallel rows in Gurobi is done by a two level hashing algorithm. The first hash function considers the support of the row, i.e., the indices of the columns with non-zero coefficients. The second hash function considers the coefficients, normalized to have a maximum norm of 1 and to have a positive coefficient for the variable with smallest index. Still it can happen that many rows end up in the same hash bin, so that a pairwise comparison between the rows in the same bin is too expensive. In this case, we sort the rows of the bin lexicographically and compare only direct neighbors in this sorted bin.

A small generalization of parallel row detection can be done by considering singleton variables in a special way. A singleton variable x_j is a variable which has only one non-zero coefficient in the matrix, i.e., $|\mathrm{supp}\,(A_{\cdot j})| = 1$. Let x_1 and x_2 be two different singleton variables, $C = \{3, \ldots, n\}$, and $q, r \in M$ be two different row indices such that $A_{qC} = sA_{rC}$ with $s \in \mathbb{R}$. Then the following holds:

1. If both constraints are equations

$$\begin{aligned} a_{q1}x_1 \qquad\quad + A_{qC}x_C &= b_q \\ a_{r2}x_2 + A_{rC}x_C &= b_r \end{aligned}$$

with $a_{r2} \neq 0$, then we can substitute $x_2 := tx_1 + d$ with $t = a_{q1}/(sa_{r2})$ and $d = (b_r - b_q/s)/a_{r2}$, provided that we tighten the bounds of x_1 to

$$\begin{aligned} &\ell_1 := \max\{\ell_1, (\ell_2 - d)/t\} \;\text{ and }\; u_1 := \min\{u_1, (u_2 - d)/t\} \;\text{ for } t > 0, \\ &\ell_1 := \max\{\ell_1, (u_2 - d)/t\} \;\text{ and }\; u_1 := \min\{u_1, (\ell_2 - d)/t\} \;\text{ for } t < 0. \end{aligned}$$

Furthermore, after substitution the two rows are parallel, and constraint r can be discarded.

2. If exactly one constraint is an equation and only this equation contains an additional singleton variable:

$$\begin{aligned} a_{q1}x_1 + A_{qC}x_C &= b_q \\ A_{rC}x_C &\leq b_r \end{aligned}$$

with $a_{q1} \neq 0$, we can tighten the bounds of x_1 by

$$\ell_1 := \max\{\ell_1, (b_q - sb_r)/a_{q1}\} \text{ for } sa_{q1} > 0,$$
$$u_1 := \min\{u_1, (b_q - sb_r)/a_{q1}\} \text{ for } sa_{q1} < 0.$$

Furthermore, after the bound strengthening of x_1 the inequality r becomes redundant and can be discarded.

3. Suppose both constraints are inequalities

$$\begin{aligned} a_{q1}x_1 \qquad + A_{qC}x_C &\leq b_q \\ a_{r2}x_2 + A_{rC}x_C &\leq b_r \end{aligned}$$

with $a_{q1} \neq 0$, $a_{r2} \neq 0$, $s > 0$, $b_q = sb_r$, $c_1c_2 \geq 0$, $a_{q1}\ell_1 = sa_{r2}\ell_2$, and $a_{q1}u_1 = sa_{r2}u_2$. If x_1 and x_2 are continuous variables, $\{1, 2\} \subseteq N \setminus I$, we can aggregate $x_2 := a_{q1}/(sa_{r2})x_1$ and discard constraint r. We can do the same if both variables are integer, $\{1, 2\} \subseteq I$, and $a_{q1} = sa_{r2}$.

In Gurobi, the detection of "nearly parallel" rows is done together with the regular parallel row detection, see again [3]. To do so, we temporarily remove singleton variables from the constraint matrix and mark the constraints that contain these variables. Now, if the parallel row detection finds a pair of parallel row vectors and any of these rows is marked, we are in the "nearly parallel" row case. Otherwise, we are in the regular parallel row case.

5. Non-zero cancellation

Adding equations to other constraints leads to an equivalent model, potentially with a different non-zero structure. This can be used to decrease the number of non-zeros in the coefficient matrix A, see for example Chang and McCormick [10]. More precisely, assume we have two rows

$$\begin{aligned} A_{qS}x_S + A_{qT}x_T + A_{qU}x_U \qquad\qquad &= b_q \\ A_{rS}x_S + A_{rT}x_T \qquad\qquad + A_{rV}x_V &\leq b_r \end{aligned}$$

with $q, r \in M$ and

$$S \cup T \cup U \cup V = \operatorname{supp}(A_{q\cdot}) \cup \operatorname{supp}(A_{r\cdot})$$

being a partition of the joint support $\operatorname{supp}(A_{q\cdot}) \cup \operatorname{supp}(A_{r\cdot})$ of the rows. Further assume that there exists a scaler $s \in \mathbb{R}$ such that $A_{qS} = sA_{rS}$ and $A_{qj} \neq sA_{rj}$ for all $j \in T$. Then, subtracting s times row q from row r yields the modified system:

$$\begin{aligned} A_{qS}x_S + A_{qT}x_T \qquad\qquad + A_{qU}x_U \qquad\qquad &= b_q \\ + (A_{rT} - sA_{qT})x_T - sA_{qU}x_U + A_{rV}x_V &\leq b_r - sb_q. \end{aligned}$$

The number of non-zero coefficients in the matrix is reduced by $|S| - |U|$, so the transformation should be applied if $|S| > |U|$.

For mixed integer programming, reducing the number of non-zeros in the coefficient matrix A can be particularly useful because the run-time of many sub-routines in a MIP solve

depends on this number. Furthermore, non-zero cancellation may generate singleton columns or rows, which opens up additional presolving opportunities.

For this reason, a number of methods applied in Gurobi presolve try to add equations to other rows in order to reduce the total number of non-zeros in the matrix, each method having a different trade-off regarding the complexity of the algorithm and its effectiveness and applicability. A relatively general but expensive method picks equations from the system and checks for each other row with a large common support $S \cup T$ whether there is an opportunity for canceling non-zeros using the equation at hand. Other methods look for special structures (e.g., cliques) in the matrix and focus on eliminating the non-zeros associated with those structures. This is often much faster than the more general algorithm and can be effective for certain problem classes. But even though the structure based algorithms are empirically faster than the more general versions, their worst-case complexity is still quadratic in the maximal number of non-zeros per column or row. Thus, we use a work limit to terminate the algorithm prematurely if it turns out to be too expensive for a given problem instance.

In any case, adding equations to other constraints needs to be done with care. Namely, if the scalar factor s used in this aggregation is too large, this operation can easily lead to numerical issues in the subsequent equation system solves and thus in the overall MIP solving process. For this reason, Gurobi does not use aggregation weights s with $|s| > 1000$ to cancel non-zeros.

6. Bound and coefficient strengthening

Savelsbergh [18] presented the fundamental ideas of bound and coefficient strengthening for MIP that are still the backbone of the presolving process in modern MIP solvers like Gurobi. Moreover, he already described these reductions in the context of multi-row presolving. Nevertheless, due to computational complexity, implementations of these methods are often only considering the single-row case.

In its most general form, bound and coefficient strengthening can be described as follows. Given an inequality

$$A_{rS} x_S + a_{rj} x_j \leq b_r \tag{4}$$

with $S = N \setminus \{j\}$ we calculate bounds $\ell_{rS}, u_{rS} \in \mathbb{R} \cup \{-\infty, \infty\}$ such that

$$\ell_{rS} \leq A_{rS} x_S \leq u_{rS}$$

for all integer feasible solutions $x \in P_{\text{MIP}}$. Using ℓ_{rS} we can potentially tighten one of the bounds of x_j:

$$x_j \leq (b_r - \ell_{rS})/a_{rj} \text{ if } a_{rj} > 0$$
$$x_j \geq (b_r - \ell_{rS})/a_{rj} \text{ if } a_{rj} < 0$$

Bound strengthening can also be applied to equations by processing them as two separate inequalities.

If x_j is an integer variable, i.e., $j \in I$, we can use u_{rS} to strengthen the coefficient of x_j

in inequality (4). Namely, if $a_{rj} > 0$, $u_j < \infty$, and

$$a_{rj} \geq d := b_r - u_{rS} - a_{rj}(u_j - 1) > 0$$

then

$$A_{rS}x_S + (a_{rj} - d)x_j \leq b_r - du_j \tag{5}$$

is a valid constraint that dominates the original one in the sub-space of $x_j \in \{u_j - 1, u_j\}$, which is the only one that is relevant for this constraint under the above assumptions. The modified constraint (5) is equivalent to the old one (4) because for $x_j = u_j$ they are identical and for $x_j \leq u_j - 1$ the modified constraint is still redundant:

$$A_{rS}x_S + (a_{rj} - d)(u_j - 1) \leq u_{rS} + (a_{rj} - d)(u_j - 1) = b_r - du_j.$$

Constraint (5) dominates (4) for $x_j = u_j - 1$ because for this value the constraints read $A_{rS}x_S \leq u_{rS} + d$ and $A_{rS}x_S \leq u_{rS}$, respectively.

Analogously, for $a_{rj} < 0$, $l_j > -\infty$, and

$$-a_{rj} \geq d' := b_r - u_{rS} - a_{rj}(l_j + 1) > 0$$

we can replace the constraint by

$$A_{rS}x_S + (a_{rj} + d')x_j \leq b_r + d'l_j$$

to obtain an equivalent model with a tighter LP relaxation.

Now, the important question is how to calculate the bounds

$$\ell_{rS} \leq A_{rS}x_S \leq u_{rS}$$

with reasonable computational effort, so that they are valid for all $x \in P_{\text{MIP}}$ and as tight as possible. For single-row preprocessing we just use $\ell_{rS} = \inf\{A_{rS}x_S\}$ and $u_{rS} = \sup\{A_{rS}x_S\}$. This is very cheap to calculate, but it may not be very tight.

The other extreme would be to actually calculate

$$\ell_{rS} = \min\{A_{rS}x_S : x \in P_{\text{MIP}}\} \text{ and}$$
$$u_{rS} = \max\{A_{rS}x_S : x \in P_{\text{MIP}}\},$$

but this would usually be very expensive as it amounts to two MIP solves per inequality that is considered, each with the original MIP's constraint system. A light-weight alternative to MIP solves is to only use the LP bound to calculate ℓ_{rS} and u_{rS}:

$$\ell_{rS} = \min\{A_{rS}x_S : x \in P_{\text{LP}}\} \text{ and}$$
$$u_{rS} = \max\{A_{rS}x_S : x \in P_{\text{LP}}\}.$$

Still, even using just the LP bounds is usually too expensive to be practical.

A more reasonable compromise between computational complexity and tightness of the bounds is to use a small number of constraints of the system over which we maximize and minimize the linear form at hand. Again, this can be done using a MIP or an LP solve. If

we just use a single constraint, then the LP solve is particularly interesting since such LPs can be solved in $\mathcal{O}(n)$ with n being the number of variables in the problem, see Dantzig [12] and Balas and Zemel [6].

For structured problems it is often possible to calculate tight bounds for parts of the constraint. We partition the support of the constraint into blocks

$$A_{rS_1}x_{S_1} + \ldots + A_{rS_d}x_{S_d} + a_{rj}x_j \leq b_r$$

with $S_1 \cup \ldots \cup S_d = N \setminus \{j\}$ and $S_{k_1} \cap S_{k_2} = \emptyset$ for $k_1 \neq k_2$. Then we calculate individual bounds

$$\ell_{rS_k} \leq A_{rS_k}x_{S_k} \leq u_{rS_k}$$

for the blocks $k = 1, \ldots, d$ and use

$$\ell_{rS} := \ell_{rS_1} + \ldots + \ell_{rS_d} \text{ and}$$
$$u_{rS} := u_{rS_1} + \ldots + u_{rS_d}$$

to get bounds on A_{rS} for the bound and coefficient strengthening on variable x_j. As before, for calculating the individual bounds there is the trade-off between complexity and tightness. Often, we use the single-row approach

$$\inf\{A_{rS_k}x_{S_k}\} \leq A_{rS_k}x_{S_k} \leq \sup\{A_{rS_k}x_{S_k}\}$$

for most of the blocks and more complex algorithms for one or few of the blocks for which the problem structure is particularly interesting.

A related well-known procedure is the so-called *optimization based bound tightening* (OBBT), which has been first applied in the context of global optimization by Shectman and Sahinidis [19]. Here, the "block" that we want to minimize and maximize consists of just a single variable:

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

The result directly yields stronger bounds for x_j. Again, instead of solving the full MIP one can solve any relaxation of the problem, for example the LP relaxation or a problem that consists of only a subset of the constraints. Since strong bounds for variables are highly important in non-linear programming to get tighter relaxations, OBBT is usually applied in MINLP and global optimization solvers very aggressively. In mixed integer programming one needs to be more conservative to avoid presolving being too expensive. Thus, Gurobi employs OBBT only for selected variables and only when the additional effort seems to be worthwhile.

7. Clique merging

Cliques are particularly interesting and important sub-structures in mixed integer programs. They often appear in MIPs to model "one out of n" decisions. The name "clique" refers to a stable set relaxation of the MIP, which is defined on the so-called *conflict graph*, see Atamtürk, Nemhauser and Savelsbergh [4]. This graph has a node for each binary variable

and its complement and an edge between two nodes if the two corresponding (possibly complemented) binary variables cannot both take the value of 1 at the same time. Any feasible MIP solution must correspond to a stable set in this conflict graph. Thus, any valid inequality for the stable set polytope on the conflict graph is also valid for the MIP.

Many edges of the conflict graph can be directly read from the MIP formulation. In particular, every set packing (set partitioning) constraint

$$\sum_{j \in S} x_j + \sum_{j \in T} (1 - x_j) \leq 1 \ (= 1)$$

with $S, T \subseteq I$, $S \cap T = \emptyset$, $\ell_j = 0$ and $u_j = 1$ for all $j \in S \cup T$, gives rise to $|S \cup T| \cdot (|S \cup T| - 1)/2$ edges. Obviously, the corresponding nodes (i.e., variables) form a clique in the conflict graph. Now, clique merging is the task of combining several set packing constraints into a single inequality.

Example 1. *Given the three set packing constraints*

$$\begin{aligned} x_1 + x_2 &\leq 1 \\ x_1 + x_3 &\leq 1 \\ x_2 + x_3 &\leq 1 \end{aligned}$$

with binary variables x_1, x_2 and x_3, we can merge them into

$$x_1 + x_2 + x_3 \leq 1.$$

The clique merging process consists of two steps. First, we extend a given set packing constraint by additional variables using the conflict graph. This means to search for a larger clique in the graph that subsumes the clique formed by the set packing constraint, a procedure that is also used to find clique cuts, see Johnson and Padberg [15] and Savelsbergh [18]. Subsequently, we discard constraints that are now dominated by the extended set packing constraint. In the example above, we could extend $x_1 + x_2 \leq 1$ to $x_1 + x_2 + x_3 \leq 1$, exploiting the fact that neither $(x_1, x_3) = (1, 1)$ nor $(x_2, x_3) = (1, 1)$ can lead to a feasible MIP solution. Then we would recognize that the two other constraints $x_1 + x_3 \leq 1$ and $x_2 + x_3 \leq 1$ are dominated by the extended constraint.

Both the clique extension and the domination checks can be time consuming in practice, in particular for set packing models with a large number of variables. For this reason, Gurobi uses a work limit for the clique merging algorithm and aborts if it becomes too expensive.

8. Computational results

In this section we assess the performance impact of the various multi-row presolve reductions that have been discussed in this paper. Our benchmark runs have been conducted on a 4-core Intel i7-3770K CPU running at 3.5 GHz with 32 GB RAM. As a reference we use Gurobi 5.6.3 in default settings with a time limit of 10000 seconds. For each test the reference solver is compared to a version in which certain presolve reductions have been disabled (or non-default ones have been enabled), either through parameter settings or by modifying the

Table 1 Impact of disabling presolve

		default	no presolving					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	2930	91	369	294	1166	1.58	1.94	2444	1.74
[0,10k]	2850	11	289	294	1166	1.60	2.00	2364	1.77
[1,10k]	1426	11	289	229	1071	2.55	2.41	1418	2.57
[10,10k]	1067	11	289	157	839	3.10	2.59	1061	3.12
[100,10k]	754	11	289	88	627	3.88	2.98	749	3.91
[1000,10k]	486	11	289	41	427	4.89	3.79	483	4.92

source code. The test set consists of 2974 problem instances from public and commercial sources. It represents the subset of models of our mixed integer programming model library that we have ever been able to solve within 10000 seconds using any of the Gurobi releases.

We present our results in a similar form as in [2]. Consider Table 1 as an example, which shows the effect of turning off root presolve completely by setting the PRESOLVE parameter to 0. In the table we group the test set into solve time brackets. The "all" bracket contains all models of the test set, except those that have been excluded because the two solver versions at hand produced inconsistent answers regarding the optimal objective value. This can happen due to the use of floating point arithmetics and resulting numerical issues in the solving process. The "[n,10k]" brackets contain all models that were solved by at least one of the two solvers within the time limit and for which the slower of the two used at least n seconds. Thus, the more difficult brackets are subsets of the easier ones. In the discussion below, we will usually use the [10,10k] bracket, because this excludes the relatively uninteresting easy models but is still large enough to draw meaningful conclusions.

Column "models" lists the number of models in each bracket of the test set. The "default tilim" column shows the number of models for which Gurobi 5.6.3 in default settings hits the time limit. The second "tilim" column contains the same information for the modified code. As can be seen, a very large number of problem instances become unsolvable when presolve is disabled: 80 models of the full test set cannot be solved by either of the two versions, 11 models can only be solved with presolving disabled, but enabling presolve is essential to solve 289 of the models within the time limit.

The columns "faster" and "slower" list the number of models that get at least 10% faster or slower, respectively, when the modified version of the code is used. Column "time" shows the shifted geometric mean of solve time ratios, using a shift of 1 second, see [1]. Values larger than 1.0 mean that the modified version is slower than the reference solver. Similarly, column "nodes" lists the shifted geometric mean ratio of the number of branch-and-bound nodes required to solve the models, again using a shift of 1. Finally, the two "affected" columns repeat the "models" and "time" statistics for the subset of models for which the solving process was affected by the code change. As an approximate check for a model being affected we compare the total number of simplex iterations used to solve a model and call

Table 2 Impact of disabling all of multi-row presolving

		default	disable multi-row presolving					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	2954	91	128	384	606	1.10	1.05	1798	1.16
[0,10k]	2875	12	49	384	606	1.10	1.06	1719	1.17
[1,10k]	1334	12	49	367	573	1.22	1.12	1268	1.23
[10,10k]	901	12	49	254	433	1.31	1.12	867	1.33
[100,10k]	522	12	49	143	280	1.45	1.20	507	1.47
[1000,10k]	249	12	49	61	146	1.65	1.37	242	1.68

Table 3 Impact of enabling dependent row checking

		default	enable dependent row checking					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	2964	91	97	227	261	1.02	1.02	1019	1.05
[0,10k]	2885	12	18	227	261	1.02	1.02	943	1.05
[1,10k]	1305	12	18	223	259	1.04	1.04	812	1.06
[10,10k]	874	12	18	184	210	1.05	1.06	601	1.08
[100,10k]	470	12	18	116	137	1.08	1.07	357	1.10
[1000,10k]	202	12	18	62	64	1.08	1.07	173	1.09

the model "affected" if this number differs for the two versions.

As can be seen in Table 1, presolving is certainly an essential component of MIP solvers. The number of time-outs increases by 278, and the average solve time in the [10,10k] bracket is more than tripled when presolving is turned off. Note that Achterberg and Wunderling [2] measure an even higher degradation factor, but this is because they also disabled node presolve in their tests.

Table 2 shows the overall impact of the multi-row presolving methods that are described in this paper. Even though the performance gain from multi-row presolve is only a small fraction of the total speed-up obtained from presolving as a whole, it still contributes with a significant improvement. In the [10,10k] bracket the average time to solve the problem instances to optimality increases by 31% and the number of unsolved models increases by 37 if the multi-row presolving features are turned off.

Tables 3 to 7 provide an analysis of the performance impact of the individual presolving components that we discussed in this paper. Checking for redundancy using multiple rows as described in Section 3 does not seem to be worth the effort for MIP solving. There are a number of heuristic methods included in Gurobi to perform this task, but none of them is enabled in default settings. A more systematic approach for finding redundant equations is to use a rank revealing LU factorization [17]. For linear programs we use this algorithm to discover and discard linear dependent equations. But even this method is disabled by default for MIP as it hurts performance. Table 3 shows the impact when the rank revealing LU

Table 4 Impact of disabling parallel row detection

		default	no parallel row detection					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	2960	91	95	303	363	1.01	0.99	1384	1.02
[0,10k]	2881	12	16	303	363	1.01	0.99	1305	1.02
[1,10k]	1307	12	16	296	352	1.02	0.99	1055	1.02
[10,10k]	871	12	16	237	287	1.02	0.98	744	1.02
[100,10k]	475	12	16	154	166	1.02	0.97	419	1.02
[1000,10k]	204	12	16	80	72	0.97	0.94	190	0.97

Table 5 Impact of disabling non-zero cancellation methods

		default	no non-zero cancellation					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	2961	91	102	240	293	1.02	1.02	1189	1.04
[0,10k]	2882	12	23	240	293	1.02	1.02	1112	1.05
[1,10k]	1302	12	23	236	280	1.04	1.04	937	1.05
[10,10k]	872	12	23	197	229	1.05	1.03	680	1.06
[100,10k]	474	12	23	135	143	1.05	1.02	400	1.05
[1000,10k]	212	12	23	73	68	0.99	0.97	194	0.98

factorization is activated for our MIP test set. One can see a surprisingly large degradation of 5% in the [10,10k] bracket, and this degradation does not come from the LU factorization being such a large overhead. Instead, removing the dependent equations leads to larger search trees. We conjecture that this degradation is caused by the cutting planes being less effective when redundant equations are removed from the system, as the aggregation heuristics within the cutting plane separation procedures may fail to reconstruct such an equation from the remaining set of constraints.

Detecting parallel rows as described in Section 4 does have a positive effect on the MIP solver performance, but this is pretty modest, see Table 4. Even though a number of models get slower when the parallel row detection is disabled, the difference in the number of time-outs is marginal. Moreover, the speed-up due to parallel row detection is only 2%. This is in line with what we have observed for the multi-row redundancy checks of Section 3.

In contrast, the non-zero cancellation of Section 5 is improving Gurobi's performance significantly, as can be seen in Table 5. The speed-up in the [10,10k] bracket is 5%, and we can solve an additional 11 models.

Table 6 illustrates the performance impact of the clique merging algorithm of Section 7. It turns out that this method is very successful in reducing solve times and node counts. On average in the [10,10k] bracket, the time to solve a model increases by 13% and the number of nodes increases by 9% when clique merging is disabled.

An even more important multi-row presolve component is represented by the bound and

Table 6 Impact of disabling clique merging

		default	disable clique merging					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	2965	91	108	263	347	1.04	1.04	1195	1.10
[0,10k]	2881	7	24	263	347	1.04	1.04	1114	1.11
[1,10k]	1304	7	24	253	336	1.09	1.08	971	1.12
[10,10k]	879	7	24	210	282	1.13	1.09	699	1.16
[100,10k]	477	7	24	135	175	1.16	1.13	405	1.19
[1000,10k]	213	7	24	66	91	1.26	1.24	197	1.28

Table 7 Impact of disabling bound and coefficient strengthening

		default	disable domain and coefficient strengthening					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	2957	91	107	404	494	1.05	1.03	1661	1.09
[0,10k]	2877	11	27	404	494	1.05	1.03	1581	1.10
[1,10k]	1315	11	27	389	463	1.11	1.06	1211	1.13
[10,10k]	882	11	27	277	359	1.17	1.06	836	1.18
[100,10k]	500	11	27	164	222	1.23	1.12	480	1.24
[1000,10k]	226	11	27	84	102	1.17	1.06	219	1.17

coefficient strengthening algorithms of Section 6. Disabling these methods yields a performance degradation of 17%, see Table 7. Moreover, the number of unsolved problem instances increases by 16.

9. Conclusion

In this paper, we reported on a subset of the preprocessing techniques included in the commercial mixed-integer solver Gurobi, namely the so-called *multi-row* reductions, which consider multiple rows at a time to find improvements to the model formulation.

Extensive computational tests over a test-set of about three thousand models show that these multi-row presolving methods are successful in improving the performance of mixed integer programming solvers. Most notably, clique merging and multi-row bound and coefficient strengthening contribute a significant speed-up.

Nevertheless, multi-row presolving is just one particular aspect within the arsenal of MIP presolving techniques as implemented in Gurobi. In total, multi-row presolve provides a speed-up of 31%, which is certainly non-negligible. But on the other hand, disabling all of presolve significantly increases the number of models that cannot be solved within the time limit and degrades overall solver performance by more than a factor of 3.

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. In M. Jünger and G. Reinelt, editors, *Facets of Combinatorial Optimization*, pages 449–481. Springer Berlin Heidelberg, 2013.

[3] E. D. Andersen and K. D. Andersen. Presolving in linear programming. *Mathematical Programming*, 71:221–245, 1995.

[4] A. Atamtürk, G. L. Nemhauser, and M. W. P. Savelsbergh. Conflict graphs in solving integer programming problems. *European Journal of Operational Research*, 121(1):40–55, 2000.

[5] A. Atamtürk and M. W. P. Savelsbergh. Integer-programming software systems. *Annals of Operations Research*, 140:67–124, 2005.

[6] E. Balas and E. Zemel. An algorithm for large zero-one knapsack problems. *Operations Research*, 28(5):1130–1154, 1980.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] H. Crowder, E. L. Johnson, and M. Padberg. Solving Large-Scale Zero-One Linear Programming Problems. *Operations Research*, 31(5):803–834, 1983.

[12] G. B. Dantzig. Discrete-Variable Extremum Problems. *Operations Research*, 5(2):266–277, 1957.

[13] M. Guignard and K. Spielberg. Logical reduction methods in zero-one programming: Minimal preferred variables. *Operations Research*, 29(1):49–74, 1981.

[14] 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.

[15] E. L. Johnson and M. W. Padberg. Degree-two inequalities, clique facets, and biperfect graphs. *Annals of Discrete Mathematics*, 16:169–187, 1982.

[16] E. L. Johnson and U. H. Suhl. Experiments in integer programming. *Discrete Applied Mathematics*, 2(1):39–55, 1980.

[17] L. Miranian and M. Gu. Strong rank revealing LU factorizations. *Linear Algebra and its Applications*, 367(0):1 – 16, 2003.

[18] M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994.

[19] 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.

[20] U. Suhl and R. Szymanski. Supernode processing of mixed-integer models. *Computational Optimization and Applications*, 3(4):317–331, 1994.