Zuse Institute Berlin

Takustr. 7
14195 Berlin
Germany

TOBIAS ACHTERBERG, ROBERT E. BIXBY, ZONGHAO GU,
EDWARD ROTHBERG, AND DIETER WENINGER

Presolve Reductions in Mixed Integer Programming

This work has been supported by the Research Campus MODAL *Mathematical Optimization and Data Analysis Laboratories* funded by the Federal Ministry of Education and Research (BMBF Grant 05M14ZAM). All responsibility for the content of this publication is assumed by the authors.

ZIB Report 16-44 (September 2016)

Zuse Institute Berlin
Takustr. 7
D-14195 Berlin

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

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

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

Presolve Reductions in Mixed Integer Programming

Tobias Achterberg* Robert E. Bixby* Zonghao Gu*
Edward Rothberg* Dieter Weninger†

December 9, 2016

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. As our computational results will show, presolve is a key factor in the speed with which we can solve mixed-integer programs, and is often the difference between a model being intractable and solvable, in some cases easily solvable. In this paper we describe the presolve functionality in the Gurobi commercial mixed-integer programming code. This includes an overview, or taxonomy of the different methods that are employed, as well as more-detailed descriptions of several of the techniques, with some of them appearing, to our knowledge, for the first time in the literature.

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, see Achterberg and Wunderling [5].

*Gurobi Optimization, {achterberg,bixby,gu,rothberg}@gurobi.com
†University Erlangen-Nürnberg, dieter.weninger@math.uni-erlangen.de

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 search 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. [13]. This paper describes techniques for removing redundant rows, fixing variables, identifying generalized upper bounds, and more. Presolve techniques for zero-one inequalities were investigated by Guignard and Spielberg [23], Johnson and Suhl [26], Crowder et al. [16], and Hoffman and Padberg [24]. Andersen and Andersen [6] published results in the context of linear programming and Savelsbergh [34] investigated preprocessing and probing techniques for general MIP problems. Details on the implementation of various presolve reductions are discussed in Suhl and Szymanski [36], Atamtürk and Savelsbergh [8], and Achterberg [2].

Investigations of the performance impact of different features of the CPLEX MIP solver were published in Bixby et al. [11] and Bixby and Rothberg [12], and most recently in Achterberg and Wunderling [5]. 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 the presolve algorithms that are applied in the Gurobi commercial MIP solver. Our main contribution is to provide a complete overview of all the presolve steps in Gurobi version 6.5 and to give sufficient implementation details and computational results to illustrate the ideas and effectiveness of the presolve reductions.

The paper is organized as follows. Section 2 introduces the primary notation used in the remainder of the paper. Sections 3 to 8 constitute the main part of the paper; they describe the individual components of the Gurobi presolve engine and provide computational results to assess their performance impact. We summarize our conclusions in Section 9.

1.1 Benchmarking

Our computational experiments have been conducted on machines with a single 4-core Intel i7-3770K CPU running at 3.5 GHz and with 32 GB RAM. As a reference we use Gurobi 6.5 in default settings with a time limit of 3600 seconds. For each test the reference solver is compared to a version in which

Table 1: Impact of disabling presolve

		default	disable presolve					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3047	547	1035	255	1755	3.36	1.27	—	—
$\geq$ 0 sec	2511	16	504	255	1755	4.52	1.91	2411	4.80
$\geq$ 1 sec	1944	16	504	210	1634	6.60	2.12	1929	6.73
$\geq$ 10 sec	1575	16	504	141	1380	9.05	2.29	1564	9.23
$\geq$ 100 sec	1099	16	504	86	983	12.36	2.43	1095	12.50
$\geq$ 1000 sec	692	16	504	34	643	19.48	2.17	691	19.57

certain presolve reductions have been disabled (or non-default ones have been enabled) in the master MIP solve by modifying the source code. Note that we always keep the reductions enabled in sub-MIP solves, to avoid an implicit modification of primal heuristic behavior, where the automatic decisions whether to actually conduct a sub-MIP solve depend on the success of presolving in the sub-MIP.

The test set consists of 3182 problem instances from public and commercial sources. It represents the subset of models of our mixed integer programming model library that we have been able to solve within 10000 seconds using some one of the Gurobi releases.

We present our results in a similar form to that used in Achterberg and Wunderling [5]. Consider Table 1 as an example, which shows the effect of turning off presolve (including root and node presolve) completely. 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 arithmetic and resulting numerical issues in the solving process. The "$\geq n$ sec" 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 "$\geq$ 10 sec" 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 6.5 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: 531 models of the full test set cannot be solved by either of

the two versions, 16 models can only be solved with presolving disabled, but enabling presolve is essential to solve 504 of the models within the time limit. Note that all solve time brackets have identical numbers in the "tilim" columns because, by definition, a model that hits the time limit for one of the two solvers will be part of every time bracket subset.

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 ratio of the shifted geometric means of solve times, using a shift of 1 second, see Achterberg [2]. Values larger than 1.0 mean that the modified version is slower than the reference solver. Similarly, column "nodes" lists the ratio of shifted geometric means 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 the model "affected" if this number differs for the two versions. We are not providing "affected" statistics for the "all" model set, because for models that hit the time limit in both versions it cannot be inferred from the iteration and node counts whether the solving path was different.

As can be seen in Table 1, presolving is certainly an essential component of MIP solvers. The number of time-outs increases by 488, and the average solve time in the "$\geq$ 10 sec" bracket is scaled up by a factor of nine when presolving is turned off. Note that our results are in line with the findings of Achterberg and Wunderling [5], who measured a degradation factor of 11.4 in the "$\geq$ 10 sec" bracket for disabling presolve in CPLEX, using a time limit of 10000 seconds.

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 $x \in \mathbb{R}^N$ with $x_j \in \mathbb{Z}$ for $j \in I \subseteq N = \{1, \ldots, n\}$, 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*

$$\begin{array}{ll} \min & c^T x \\ s.t. & Ax \circ b \\ & \ell \leq x \leq u \\ & x_j \in \mathbb{Z} \text{ for all } j \in I \end{array} \tag{2.1}$$

is called a mixed integer program (MIP). We denote by

$$P_{MIP} = \{x \in \mathbb{R}^N : Ax \circ b, \ell \leq x \leq u, x_j \in \mathbb{Z} \text{ for all } j \in I\}$$

the set of feasible solutions for the MIP and by

$$P_{LP} = \{x \in \mathbb{R}^N : Ax \circ b, \ell \leq x \leq u\}$$

the set of feasible solutions for the LP relaxation of the 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 vector given by the i-th row of A. Similarly, $A_{\cdot j}$ is the column vector given by the j-th column of A. With $\operatorname{supp}(A_{i\cdot}) = \{j \in N : a_{ij} \neq 0\}$ we denote the *support* of $A_{i\cdot}$, and $\operatorname{supp}(A_{\cdot j}) = \{i \in M : a_{ij} \neq 0\}$ 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 in A, we can calculate a *minimal activity* and a *maximal activity* for every row i of problem (2.1). 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 introduce the following conventions:

$$\begin{array}{llll} \infty + \infty := \infty & s \cdot \infty := \infty & s \cdot (-\infty) := -\infty & \text{for } s > 0 \\ -\infty - \infty := -\infty & s \cdot \infty := -\infty & s \cdot (-\infty) := \infty & \text{for } s < 0 \end{array}$$

We can then define the minimal activity of row i by

$$\inf\{A_{i\cdot}x\} := \sum_{\substack{j \in N \\ a_{ij} > 0}} a_{ij}\ell_j + \sum_{\substack{j \in N \\ a_{ij} < 0}} a_{ij}u_j \tag{2.2}$$

and the maximal activity by

$$\sup\{A_{i\cdot}x\} := \sum_{\substack{j \in N \\ a_{ij} > 0}} a_{ij}u_j + \sum_{\substack{j \in N \\ a_{ij} < 0}} a_{ij}\ell_j. \tag{2.3}$$

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 $S \subseteq N$.

Table 2: Impact of disabling all single-row reductions

		default	disable single-row reductions					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3047	552	565	489	957	1.13	1.02	—	—
$\geq$ 0 sec	2536	46	59	489	957	1.16	1.06	2281	1.17
$\geq$ 1 sec	1755	46	59	467	872	1.22	1.08	1700	1.22
$\geq$ 10 sec	1251	46	59	361	651	1.26	1.06	1223	1.26
$\geq$ 100 sec	720	46	59	224	379	1.30	1.02	706	1.29
$\geq$ 1000 sec	272	46	59	95	141	1.32	0.94	269	1.33

Obviously, a constraint with $\circ_i$ = "$\geq$" can easily be converted into a constraint with $\circ_i$ = "$\leq$" by multiplying the row vector and right hand side by -1. For this reason, we will usually only consider one of the two inequality directions in the description of the presolve reductions.

3 Reductions for individual constraints

This section covers the single-row presolve reductions as implemented in Gurobi. These are problem transformations that consider only one of the constraints of the problem at a time, plus the bounds and integrality of the variables. Thus, the reductions in this section are also valid for the relaxation of (2.1)

$$\begin{array}{ll} \min & c^T x \\ \text{s.t.} & A_{i\cdot} x \circ_i b_i \\ & \ell \leq x \leq u \\ & x_j \in \mathbb{Z} \text{ for all } j \in I \end{array} \tag{3.1}$$

that includcdes only a single row $i \in M$.

Table 2 shows the total performance impact of disabling all of the single-row reductions. As expected, almost all models are affected by these reductions, i.e., for almost every model at least one of the single-row reductions is triggered. The number of unsolvable models increases by 13 when the reductions are disabled, and the performance degradation in the "$\geq$ 10 sec" bracket is 26%, which is surprisingly low given that the set of single-row presolve components includes bound and coefficient strengthening. One explanation could be that bound and coefficient strengthening are also covered by multi-row, full-problem and node presolve algorithms, in particular the multi-row bound and coefficient strengthening, probing and node bound strengthening; see Sections 5.4, 7.2 and 8.1.

3.1 Model cleanup and removal of redundant constraints

Let $\varepsilon, \psi \in \mathbb{R}$ be two given constants. The former denotes the so-called *feasibility tolerance*, while the latter may be interpreted as "infinity". In Gurobi, both values can be adjusted by the user. Default values are $\varepsilon := 10^{-6}$ and $\psi := 10^{30}$.

Let a row

$$A_{i\cdot}x \circ_i b_i \tag{3.2}$$

with $i \in M$ of problem (2.1) be given. If $\circ_i =$ "$\leq$" in (3.2) and $b_i \geq \psi$ or $\sup\{A_{i\cdot}x\} \leq b_i + \varepsilon$, we discard this row. On the other hand, if $\inf\{A_{i\cdot}x\} > b_i + \varepsilon$, the problem is infeasible. If $\circ_i =$ "$=$", $\inf\{A_{i\cdot}x\} \geq b_i - \varepsilon$, and $\sup\{A_{i\cdot}x\} \leq b_i + \varepsilon$, then the row can be discarded. An equation with $\inf\{A_{i\cdot}x\} > b_i + \varepsilon$ or $\sup\{A_{i\cdot}x\} < b_i - \varepsilon$ implies the infeasibility of the problem.

Redundant constraints can result from other presolve reductions, for example bound strengthening (Section 3.2). But they also occur very often as part of the original problem formulation: 2525 out of the 3182 models in our test set contain constraints that can be classified as redundant using only the original bounds on the variables.

In addition to detecting infeasibility or removing redundant rows, we also perform reductions on small coefficients. Let (3.2) with $k \in \operatorname{supp}(A_{i\cdot})$ be given. If $|a_{ik}| < 10^{-3}$ and $|a_{ik}| \cdot (u_k - \ell_k) \cdot |\operatorname{supp}(A_{i\cdot})| < 10^{-2} \cdot \varepsilon$, we update $b_i := b_i - a_{ik} \cdot \ell_k$ and then set $a_{ik} := 0$. Moreover, we scan the row starting from the first non-zero coefficient and set coefficients a_{ik} to zero as long as the total sum of modifications $|a_{ik}| \cdot (u_k - \ell_k)$ stays below $10^{-1} \cdot \varepsilon$. Finally, we set coefficients with $|a_{ik}| < 10^{-10}$ to zero.

As can be seen in Table 3, model cleanup does not lead to big improvements in the solvability of models: the time limit hits only increase by 9, while the node count does not change much when the cleanup is disabled. Nevertheless, it has a performance impact of 10% in the "$\geq$ 10 sec" bracket, which can be attributed to faster algorithms, in particular faster LP solves, due to the smaller matrix sizes.

3.2 Bound strengthening

This preprocessing approach tries to strengthen the bounds on variables, using an iterative process called *domain propagation*, see, e.g., Savelsbergh [34], Fügenschuh and Martin [21], or Achterberg [2].

In the following we assume that bounds on integer variables are integral; otherwise, they can always be rounded to shrink the domain of the variable,

Table 3: Impact of disabling model cleanup

		default	disable model cleanup					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3102	559	568	470	650	1.05	0.97	—	—
$\geq$ 0 sec	2568	30	39	470	650	1.06	0.99	2120	1.07
$\geq$ 1 sec	1741	30	39	455	615	1.08	1.00	1601	1.08
$\geq$ 10 sec	1213	30	39	355	479	1.10	1.01	1140	1.11
$\geq$ 100 sec	672	30	39	201	286	1.14	1.10	639	1.15
$\geq$ 1000 sec	243	30	39	81	113	1.21	1.16	239	1.21

see also Section 4.2. Consider an inequality constraint

$$A_{iS}x_S + a_{ik}x_k \leq b_i \tag{3.3}$$

where $i \in M$, $k \in N$, $S = \operatorname{supp}(A_{i\cdot}) \setminus \{k\}$, and $a_{ik} \neq 0$. Bound strengthening can also be applied to equations by processing them as two separate inequalities. We calculate a lower bound $\ell_{iS} \in \mathbb{R} \cup \{-\infty\}$ such that

$$A_{iS}x_S \geq \ell_{iS}$$

for all integer feasible solutions $x \in P_{\text{MIP}}$. In the single-row case considered here, we just use $\ell_{iS} = \inf\{A_{iS}x_S\}$. Compare Section 5.4 for the multi-row case.

If ℓ_{iS} is finite we can potentially tighten the bounds of x_k. Depending on the sign of a_{ik} we distinguish two cases. For $a_{ik} > 0$ we can derive an upper bound on x_k using

$$x_k \leq (b_i - A_{iS}x_S)/a_{ik} \leq (b_i - \ell_{iS})/a_{ik}$$

and hence replace u_k by

$$u_k := \min\{u_k, (b_i - \ell_{iS})/a_{ik}\}. \tag{3.4}$$

For the case where $a_{ik} < 0$, we can derive a lower bound on x_k using

$$x_k \geq (b_i - A_{iS}x_S)/a_{ik} \geq (b_i - \ell_{iS})/a_{ik}$$

and hence replace ℓ_k by

$$\ell_k := \max\{\ell_k, (b_i - \ell_{iS})/a_{ik}\}. \tag{3.5}$$

If x_k is an integer variable we can replace (3.4) by

$$u_k := \min\{u_k, \lfloor (b_i - \ell_{iS})/a_{ik} \rfloor\}$$

Table 4: Impact of disabling bound strengthening

		default	disable bound strengthening					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3134	562	578	411	591	1.05	1.02	—	—
$\geq$ 0 sec	2586	19	35	411	591	1.06	1.03	1645	1.08
$\geq$ 1 sec	1758	19	35	400	538	1.07	1.02	1256	1.10
$\geq$ 10 sec	1201	19	35	302	400	1.09	1.03	911	1.11
$\geq$ 100 sec	654	19	35	187	218	1.10	1.05	501	1.12
$\geq$ 1000 sec	229	19	35	70	102	1.28	1.22	192	1.34

and (3.5) by

$$\ell_k := \max\{\ell_k, \lceil (b_i - \ell_{iS})/a_{ik} \rceil\}.$$

Clearly, the above strengthening procedures can be applied iteratively by appropriately updating the infimum (2.2) of the rows as bounds change. However, the number of such applications must in some way be limited in order to guarantee finiteness of the procedure, as the following example illustrates.

Consider the constraints

$$\begin{aligned} x_1 &- ax_2 &&= 0 \\ ax_1 &- x_2 &&= 0 \end{aligned}$$

with $0 \leq x_1, x_2 \leq 1$ and $0 < a < 1$. Strengthening the bounds of x_1 using the first constraint yields $x_1 = ax_2 \leq a$. Exploiting this new upper bound on x_1, the second constraint implies $x_2 = ax_1 \leq a^2$. If we iterate and process each constraint t times, we reach $x_1 \leq a^{2t-1}$ and $x_2 \leq a^{2t}$. Thus, we will find an infinite sequence of bound reductions.

To avoid such long chains of tiny reductions, Gurobi only tightens bounds of continuous variables in domain propagation if the change in the bounds is at least $10^3 \cdot \varepsilon$ (ε being the feasibility tolerance, see Section 3.1) and the absolute value of the new bound is smaller than 10^8. Moreover, in each presolving pass we only perform one round of domain propagation on each constraint, so that the work limits on the overall presolve algorithm automatically limit the number of domain propagation passes.

Table 4 shows the performance impact of bound strengthening, which is surprisingly low with its 9% in the "$\geq$ 10 sec" bracket, given that bound strengthening is arguably the most important technique for constraint programming solvers. As already mentioned, one reason for the small impact measured by this experiment is that bound strengthening is also covered by other algorithms, in particular multi-row bound strengthening (Section 5.4), probing (Section 7.2) and node bound strengthening (Section 8.1).

3.3 Coefficient strengthening

Coefficient strengthening means to modify coefficients of a constraint such that the LP relaxation of the problem gets tighter inside the box defined by the bounds of the variables without affecting the set of integer solutions that satisfy the constraint. Savelsbergh [34] describes this technique in a general setting, see also Section 5.4 of this paper.

More formally, we define constraint domination as follows:

Definition 2. *Given two constraints $ax \leq b$ and $a'x \leq b'$, where a and a' denote row vectors, with integer or continuous variables $x \in \mathbb{R}^n$ that are bounded by $\ell \leq x \leq u$, then constraint $ax \leq b$* dominates *$a'x \leq b'$ if*

$$\{x \in \mathbb{R}^n | \ell \leq x \leq u, ax \leq b\} \subset \{x \in \mathbb{R}^n | \ell \leq x \leq u, a'x \leq b'\},$$

i.e., the LP-feasible region of $ax \leq b$ is strictly contained in the one of $ax' \leq b'$ within the box defined by the variable bounds.

Consider again inequality (3.3) but assume that x_k is an integer variable, $k \in I$. In contrast to the previous Section 3.2 we now need to calculate an upper bound $u_{iS} \in \mathbb{R} \cup \{\infty\}$ on the activity of $A_{iS}x_S$, i.e.,

$$A_{iS}x_S \leq u_{iS}$$

for all integer feasible solutions $x \in P_{\text{MIP}}$. Similar to the previous section, we use $u_{iS} = \sup\{A_{iS}x_S\}$ in the single-row case that is considered here.

Since x_k is an integer variable we can use u_{iS} to strengthen the coefficient of x_k in inequality (3.3). Namely, if $a_{ik} > 0$, $u_k < \infty$, and

$$a_{ik} \geq d := b_i - u_{iS} - a_{ik}(u_k - 1) > 0$$

then

$$A_{iS}x_S + (a_{ik} - d)x_k \leq b_i - du_k \tag{3.6}$$

is a valid constraint that dominates the original one in the sub-space of $x_k \in \{u_k - 1, u_k\}$, which is the only one that is relevant for this constraint under the above assumptions. The modified constraint (3.6) is equivalent to the old one (3.3) in terms of integer solutions, because for $x_k = u_k$ they are identical and for $x_k \leq u_k - 1$ the modified constraint is still redundant:

$$A_{iS}x_S + (a_{ik} - d)(u_k - 1) \leq u_{iS} + (a_{ik} - d)(u_k - 1) = b_i - du_k.$$

Constraint (3.3) is dominated by (3.6) for $x_k = u_k - 1$ because for this value the constraints read $A_{iS}x_S \leq u_{iS} + d$ and $A_{iS}x_S \leq u_{iS}$, respectively.

Table 5: Impact of disabling coefficient strengthening

		default	disable coefficient strengthening					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3159	566	563	388	431	1.02	1.02	—	—
$\geq$ 0 sec	2614	26	23	388	431	1.03	1.02	1550	1.05
$\geq$ 1 sec	1760	26	23	377	416	1.04	1.03	1282	1.05
$\geq$ 10 sec	1225	26	23	287	347	1.06	1.04	961	1.08
$\geq$ 100 sec	672	26	23	175	209	1.09	1.07	559	1.11
$\geq$ 1000 sec	218	26	23	69	75	1.14	1.13	200	1.15

Analogously, for $a_{ik} < 0$, $\ell_k > -\infty$, and

$$-a_{ik} \geq d' := b_i - u_{iS} - a_{ik}(\ell_k + 1) > 0$$

we can replace the constraint by

$$A_{iS}x_S + (a_{ik} + d')x_k \leq b_i + d'\ell_k$$

to obtain an equivalent model with a tighter LP relaxation.

Similar to bound strengthening, the computational impact of coefficient strengthening is relatively small, as can be seen in Table 5. It does not help to solve more models within the time limit, and it improves performance by only 6% in the "$\geq$ 10 sec" bracket. As noted before, this is most likely due to the fact that the reduction is also covered by the multi-row coefficient strengthening (Section 5.4) and partly by probing (Section 7.2).

3.4 Chvátal-Gomory strengthening of inequalities

We can apply the Chvátal-Gomory procedure [15] to inequalities that contain only integer variables. If the resulting inequality strictly dominates the original one (see Definition 2), we use the Chvátal-Gomory inequality instead.

More precisely, let $A_{i\cdot}x \geq b_i$, $i \in M$, be a constraint of problem (2.1) with $\operatorname{supp}(A_{i\cdot}) \subseteq I$ and $a_{ij} \geq 0$ for all $j \in N$. For negative coefficients we can complement the corresponding variable, provided that it has a finite upper bound. Moreover, assume that $\ell_j = 0$ for all $j \in \operatorname{supp}(A_{i\cdot})$. Variables x_j with a finite lower bound $\ell_j \neq 0$ can be shifted by $x'_j := x_j - \ell_j$ to get $\ell'_j = 0$.

If we can find a scalar $s \in \mathbb{R}$ with $s > 0$ such that

$$\lceil a_{ij} \cdot s \rceil \cdot b_i / \lceil b_i \cdot s \rceil \leq a_{ij}$$

Table 6: Impact of disabling Chvátal-Gomory strengthening

		default	disable Chvatal-Gomory strengthening					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3176	570	565	368	334	1.00	1.02	—	—
$\geq$ 0 sec	2624	23	18	368	334	1.00	1.03	1357	1.00
$\geq$ 1 sec	1763	23	18	354	325	1.00	1.05	1155	1.00
$\geq$ 10 sec	1216	23	18	287	261	1.00	1.04	852	1.01
$\geq$ 100 sec	660	23	18	184	159	1.00	1.01	493	1.01
$\geq$ 1000 sec	200	23	18	65	60	1.01	1.02	168	1.02

for all $j \in N$, and there exists $k \in N$ with

$$\lceil a_{ik} \cdot s \rceil \cdot b_i / \lceil b_i \cdot s \rceil < a_{ik},$$

then we replace $A_{i\cdot} x \geq b_i$ by $\lceil A_{i\cdot} \cdot s \rceil x \geq \lceil b_i \cdot s \rceil$.

Now the question is how to find the scalar s. For this, Gurobi uses a very simple heuristic. Namely, we just try values

$$s \in \{1, t/a_{\max}, t/a_{\min}, (2t-1)/(2a_{\min}) : t = 1, \ldots, 5\}$$

with $a_{\max} = \max\{|a_{ij}| : j \in N\}$ and $a_{\min} = \min\{|a_{ij}| : j \in \operatorname{supp}(A_{i\cdot})\}$.

The performance impact of Chvátal-Gomory strengthening is disappointing; see Table 6. Even though it affects many models, it does not help at all to solve more of them or to solve them faster. Nevertheless, the reduction is turned on in default Gurobi settings.

3.5 Euclidean and modular inverse reduction

Given a constraint

$$\sum_{j=1}^{n} a_{ij} x_j \circ_i b_i$$

with $x_j \in \mathbb{Z}$ for all $j = 1, \ldots, n$, $i \in M$ and $\circ_i \in \{\text{“}=\text{”}, \text{“}\leq\text{”}\}$, the *Euclidean reduction* approach divides the constraint by the greatest common divisor $d = \gcd(a_{i1}, \ldots, a_{in})$ of the coefficients, where we use the following extended definition that allows application to rational numbers, not just integers.

Definition 3. *Given $a_1, \ldots, a_n \in \mathbb{Q}$, the* greatest common divisor $d = \gcd(a_1, \ldots, a_n)$ *is the largest value $d \in \mathbb{Q}$ such that $a_j/d \in \mathbb{Z}$ for all $j = 1, \ldots, n$.*

If all coefficients in the given constraint are integral we use the Euclidean algorithm to calculate $d \in \mathbb{Z}$. One could also apply the Euclidean algorithm for fractional coefficients, but we have found that this can lead to numerical issues due to the limited precision of floating point arithmetic. Moreover, benchmarks results have shown that it degrades the performance. For this reason, Gurobi uses two simple heuristics to try to find the gcd for fractional coefficients:

1. Divide all coefficients by $a_{\min} = \min\{|a_{ij}| : j \in \operatorname{supp}(A_{i\cdot})\}$. If this leads to integer values for all coefficients, return
$$d = a_{\min} \cdot \gcd(a_{i1}/a_{\min}, \ldots, a_{in}/a_{\min}).$$
2. Multiply all coefficients by 600. If this leads to integer values for all coefficients, return
$$d = \gcd(600 \cdot a_{i1}, \ldots, 600 \cdot a_{in})/600.$$

Note that we use 600 as a multiplier because it is a multiple of many small integer values that often arise as denominators in real-world models.

Now, if $\operatorname{supp}(A_{i\cdot}) \subseteq I$ and $d = \gcd(a_{i1}, \ldots, a_{in})$ we can replace the constraint by
$$\sum_{j=1}^{n} \Big(\frac{a_{ij}}{d}\Big) x_j \circ_i \Big\lfloor \frac{b_i}{d} \Big\rfloor. \tag{3.7}$$
In case of $\circ_i =$ "=" and $b_i/d \notin \mathbb{Z}$, the problem is infeasible.

For equations we can derive some knowledge about the divisibility of the variables, say x_1 with $a_{i1} \neq 0$. Note that in this case, we can also deal with a single continuous variable x_1. Then, if $\operatorname{supp}(A_{i\cdot}) \setminus \{1\} \subseteq I$ we know that x_1 will always be a multiple of
$$d' := \gcd(a_{i2}/a_{i1}, \ldots, a_{in}/a_{i1}, b_i/a_{i1})$$
and can substitute $x_1 := d' \cdot z$ with a new, auxiliary integer variable z.

Example 1. Consider the constraint $x_1 - 3x_2 + 6x_3 = 9$ with $x_2, x_3 \in \mathbb{Z}$. We express x_1 as $x_1 = 9 + 3x_2 - 6x_3$ and see that x_1 is always an integer multiple of 3. Thus we can substitute $x_1 := 3z$ with $z \in \mathbb{Z}$ and get $z - x_2 + 2x_3 = 3$.

If all variables of the constraint are integers, $\operatorname{supp}(A_{i\cdot}) \subseteq I$, we divide the equation by d as in (3.7) so that the resulting constraint
$$a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n = b_i \tag{3.8}$$

has $b_i \in \mathbb{Z}$ and $a_{ij} \in \mathbb{Z}$ for all j, and the integers $\{a_{i1}, \ldots, a_{in}\}$ are relatively prime. Then we consider

$$\bar{d} := \gcd(a_{i2}, \ldots, a_{in}) \in \mathbb{Z}_{>0}.$$

If $\bar{d} \geq 2$ we can derive a substitution on x_1. First, if $b_i/\bar{d} \in \mathbb{Z}$, we can substitute $x_1 := \bar{d} \cdot z$ as above with auxiliary integer variable $z \in \mathbb{Z}$. Otherwise, we can use the *modular multiplicative inverse* as follows.

Definition 4. *The* modular multiplicative inverse *of $a \in \mathbb{Z}$ modulo $m \in \mathbb{Z}_{>0}$ is an integer $a^{-1|_m} \in \mathbb{Z}_{>0}$ such that $a \cdot a^{-1|_m} \equiv 1 \pmod{m}$.*

Because $\{a_{i1}, \ldots, a_{in}\}$ are relatively prime we have $\gcd(a_{i1}, \bar{d}) = 1$. Hence,

$$a_{i1} x_1 \equiv b_i \pmod{\bar{d}}$$

for all $x_1 \in \mathbb{Z}$ that are feasible for (3.8). It follows that

$$x_1 \equiv a_{i1}^{-1|_{\bar{d}}} \cdot b_i \pmod{\bar{d}}$$

and hence that we can substitute

$$x_1 := \bar{d} \cdot z + \bar{b} \quad \text{and} \quad \bar{b} \equiv a_{i1}^{-1|_{\bar{d}}} \cdot b_i \pmod{\bar{d}}$$

with $0 < \bar{b} < \bar{d}$ where $z \in \mathbb{Z}$ is a new, auxiliary variable.

Note that if $|a_{i1}| = |a_{ij}|$ for any $j \neq 1$, then $\bar{d} = 1$ due to the fact that the constraint coefficients $\{a_{i1}, \ldots, a_{in}\}$ are relatively prime. Hence, we only need to look at variables that have a unique constraint coefficient in absolute terms. Gurobi only applies the reduction to the variable with the smallest (absolute) non-zero coefficient in the constraint, and only if this variable is unique.

Example 2. Given the problem

$$\begin{array}{ll} \min & x_1 + x_2 \\ \text{s.t.} & 1867 x_1 + 1913 x_2 = 3618894 \\ & x_1, x_2 \geq 0 \end{array}$$

with $x_1, x_2 \in \mathbb{Z}$ being integer variables. Note that 1867 and 1913 are prime and hence relatively prime. Note also that 3618894/1913 is fractional. The modular multiplicative inverse of 1867 is $1867^{-1|_{1913}} = 1206$. Further, $1206 \cdot 3618894 \equiv 1009 \pmod{1913}$. Now we can substitute $x_1 = 1913z + 1009$ with $z \in \mathbb{Z}$, $z \geq 0$, which yields $3571571z + 1913x_2 = 1735091$. Dividing by

Table 7: Impact of disabling Euclidean and modular inverse reduction

		default	disable Euclidean reduction					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3167	571	569	417	413	1.01	1.01	—	—
$\geq$ 0 sec	2617	26	24	417	413	1.01	1.02	1607	1.02
$\geq$ 1 sec	1760	26	24	405	403	1.02	1.01	1317	1.02
$\geq$ 10 sec	1217	26	24	319	335	1.03	1.04	974	1.04
$\geq$ 100 sec	681	26	24	209	212	1.05	1.10	576	1.06
$\geq$ 1000 sec	213	26	24	79	69	1.06	1.10	196	1.06

the greatest common divisor 1913 results in $1867z + x_2 = 907$. The optimal solution (and obviously the only solution) is $z = 0$ and $x_2 = 907$. Inserting $z = 0$ into $x_1 = 1913z+1009$ we get $x_1 = 1009$. By using the modular inverse presolve reduction Gurobi solves the problem at the root node. Without this presolving step it takes almost two thousand branch-and-bound nodes.

The performance benefit of the Euclidean reduction is low, as can be seen in Table 7. Nevertheless, even though one may think that its applicability is relatively small, it does affect a good fraction of the models in our test set. The "faster/slower" statistics indicate that it is helping and hurting on roughly the same number of models, but overall it yields a 3% performance improvement.

3.6 Simple probing on a single equation

If the value of a binary variable in a constraint implies the value for some other variables and vice versa, we can substitute the other variables for the binary variable.

Gurobi applies this idea only to one special case that can be detected very efficiently. The more general case is covered by regular probing; see Section 7.2. More precisely, we examine equality constraints $A_{i\cdot}x = b_i$ with

$$\inf\{A_{i\cdot}x\} + \sup\{A_{i\cdot}x\} = 2b_i \tag{3.9}$$

and look for variables $x_k \in \{0, 1\}$, $k \in \operatorname{supp}(A_{i\cdot})$, such that

$$|a_{ik}| = \sup\{A_{i\cdot}x\} - b_i. \tag{3.10}$$

Since complementing variables, that is replacing x_j by $x'_j = u_j - x_j$, does not affect (3.9) and (3.10) we may assume w.l.o.g. that $a_{ij} \geq 0$ for all $j \in N$. Then, the equality constraint together with condition (3.10) implies

$$x_k = 0 \rightarrow x_j = u_j \qquad \text{for all } j \in \operatorname{supp}(A_{i\cdot}) \setminus \{k\}.$$

Table 8: Impact of disabling simple probing on single equations

		default	disable simple probing					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3180	571	570	98	102	1.00	1.00	—	—
$\geq$ 0 sec	2613	9	8	98	102	1.00	1.00	366	1.00
$\geq$ 1 sec	1746	9	8	97	101	1.00	1.00	333	1.00
$\geq$ 10 sec	1186	9	8	81	87	1.00	1.00	264	1.01
$\geq$ 100 sec	631	9	8	53	49	0.99	0.98	149	0.97
$\geq$ 1000 sec	167	9	8	26	17	1.00	0.97	54	1.00

Moreover, substituting (3.10) into (3.9) yields $|a_{ik}| = b_i - \inf\{A_{i\cdot}x\}$, and we have

$$x_k = 1 \to x_j = \ell_j \qquad \text{for all } j \in \operatorname{supp}(A_{i\cdot}) \setminus \{k\}.$$

Combining the two sets of implications, we can substitute

$$x_j := u_j - (u_j - \ell_j)x_k \qquad \text{for all } j \in \operatorname{supp}(A_{i\cdot}) \setminus \{k\}.$$

Example 3. Consider the constraint $10x_1 + x_2 + \ldots + x_{11} = 10$ with $x_1 \in \{0, 1\}$, $x_j \in \mathbb{R}$, and $0 \leq x_j \leq 1$ for $j = 2, \ldots, 11$. $x_1 = 0$ implies $x_2 = \ldots = x_{11} = 1$, and $x_1 = 1$ implies $x_2 = \ldots = x_{11} = 0$. Thus, we can substitute $x_j := 1 - x_1$ for $j = 2, \ldots, 11$.

The simple probing on single equations is one of those presolve methods that should clearly help performance. Conditions (3.9) and (3.10) are very easy to check, and if they apply, we can substitute all but one variable of the constraint and discard the constraint. Unfortunately, Table 8 tells a different story. Even though the algorithm finds reductions on about 20% of the models, it does not help at all to improve performance, neither in the number of time limit hits, nor in the "faster/slower" statistics, nor in the geometric mean over the solve times. One reason for this disappointing result is certainly that the reduction is also covered by probing, see Section 7.2. But nevertheless, this cannot be the full explanation, because performing these substitutions within the innermost presolving loop should be better than having to wait for probing, which is called rather late in the presolving process.

3.7 Special ordered set reductions

The concept of ordered sets of variables in the context of MIP was introduced by Beale and Tomlin [10]. A *special ordered set type 1 (SOS1)* is a set of variables with the requirement that at most one of the variables is non-zero.

A *special ordered set type 2 (SOS2)* is an ordered sequence of variables such that at most two variables are non-zero, and if two variables are non-zero they must be adjacent in the specified order.

Considering individual SOS constraints, we can derive the following simple reductions: for a constraint SOS1$(x_1, \ldots, x_p)$, if $\ell_k > 0$ or $u_k < 0$ for a variable x_k, $1 \leq k \leq p$, we can fix $x_j := 0$ for all $j \in \{1, \ldots, p\} \setminus \{k\}$. If $\ell_k > 0$ or $u_k < 0$ in a constraint SOS2$(x_1, \ldots, x_p)$, then we can fix $x_j := 0$ for all $j \in \{1, \ldots, p\} \setminus \{k-1, k, k+1\}$.

For SOS1 constraints we can remove variables that are fixed to zero. Note that for SOS2 constraints we can only remove variables fixed to zero that are at the beginning or the end of the sequence. Otherwise, the adjacency relation, and thus the semantics of the SOS2 constraint, would be modified.

SOS1 constraints with only one remaining variable and SOS2 constraints with at most two remaining variables can be discarded. An SOS2 constraint (x_1, x_2, x_3) that consists of only 3 variables can be translated into an SOS1 constraint (x_1, x_3).

Finally, consider an SOS1 constraint (x_1, x_2) of length 2 with objective coefficients $c_1 \geq 0$, $c_2 \geq 0$, lower bounds $\ell_1 = \ell_2 = 0$, and either both variables integer, $\{1, 2\} \subseteq I$, or both continuous, $\{1, 2\} \subseteq N \setminus I$. If for all constraints $i \in M$ we have $-a_{i1} - a_{i2} \circ_i 0$, then for any given feasible solution $x^\star$ we can decrease both variables $x_1^\star$ and $x_2^\star$ simultaneously until one of them hits its lower bound of zero. For this reason, we can discard such an SOS1 constraint, since we will always be able to post-process an otherwise feasible solution such that the SOS1 constraint gets satisfied and the objective function value does not increase. Obviously, one can apply analogous reasoning to the inverse case with $c_1, c_2 \leq 0$, $u_1 = u_2 = 0$ and $a_{i1} + a_{i2} \circ_i 0$ for all $i \in M$, but Gurobi does not implement the inverted reduction because real-world models very rarely contain SOS1 constraints on variables with negative lower bound.

The performance impact of the special ordered set reductions is hard to evaluate with our computational experiments. Our test set contains 104 models with SOS constraints, and only 48 of them are affected by our SOS specific presolve reductions, see Table 9. On this very small set of models, the reduction is very powerful, providing a performance improvement of 25%. Nevertheless, the size of the test set is too small to draw any meaningful conclusions; compare Section 2 in Achterberg and Wunderling [5].

Table 9: Impact of disabling special ordered set reductions

		default	disable SOS reductions					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3181	571	572	17	21	1.00	1.01	—	—
$\geq$ 0 sec	2606	1	2	17	21	1.00	1.01	48	1.25
$\geq$ 1 sec	1743	1	2	16	18	1.01	1.02	41	1.28
$\geq$ 10 sec	1172	1	2	16	11	1.01	1.02	35	1.26
$\geq$ 100 sec	614	1	2	9	10	1.02	1.04	22	1.58
$\geq$ 1000 sec	154	1	2	3	4	1.05	1.08	8	2.50

Table 10: Impact of disabling all single-column reductions

		default	disable single-column reductions					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3149	567	630	414	1074	1.25	1.01	—	—
$\geq$ 0 sec	2611	34	97	414	1074	1.32	1.09	2226	1.37
$\geq$ 1 sec	1811	34	97	398	974	1.45	1.11	1705	1.48
$\geq$ 10 sec	1288	34	97	296	740	1.59	1.16	1236	1.61
$\geq$ 100 sec	769	34	97	184	465	1.79	1.26	749	1.82
$\geq$ 1000 sec	326	34	97	74	220	2.36	1.46	322	2.39

4 Reductions for individual variables

In the following we will explain the Gurobi presolve reductions that deal with individual columns or variables.

The results in Table 10 suggest that this set of reductions is much more powerful than the single-row reductions: compare Table 2 on page 6. But it has to be noted that the majority of the performance improvement comes from the so-called "aggregator", which substitutes out implied free variables (see Section 4.5), while the other single-column reductions provide only a very modest speed-up.

4.1 Removal of fixed variables

Variables x_j with bounds $\ell_j = u_j$ can be removed from the problem by subtracting $A_{\cdot j}\ell_j$ from the right hand side b and by accumulating the contributions $c_j\ell_j$ to the objective function in a constant $c_0 \in \mathbb{R}$. This constant has no direct effect on the optimization process but is added to the objective function for reporting objective values and for calculating the relative optimality gap.

Similar to redundant constraints, fixed variables can be the result of other presolve reductions like bound strengthening (Section 3.2). Such vari-

ables also occur frequently as part of the original problem formulation: 860 out of the 3182 models in our test set contain fixed variables. Another important case arises in sub-MIP solves, in particular in those involving primal heuristics such as RINS, see Danna et al. [17], where potentially a large fraction of the variables are fixed before then calling the MIP solver recursively as part of a procedure for finding improved integer feasible solutions.

Unfortunately, we cannot provide meaningful benchmark results to evaluate the impact of removing fixed variables from the problem. Many important heuristic decisions during presolve and also during the overall solving process depend on the changes in the size of the model due to presolve. Thus, keeping fixed variables in the model would greatly influence those decisions, resulting in a performance degradation as a side-effect that really should not be attributed to the actual removal of the variables.

4.2 Rounding bounds of integer variables

If x_j, $j \in I$, is an integer variable with fractional lower or upper bound, this bound can be replaced by $\lceil \ell_j \rceil$ or $\lfloor u_j \rfloor$, respectively. Note also that rounding the bounds of integer variables yields tighter values for the row infima (2.2) and suprema (2.3) and can thus trigger additional reductions on other variables.

The rounding of bounds of integer variables is a mandatory step in Gurobi and cannot be disabled. Gurobi's internal MIP solving algorithms rely on the fact that integer variables have integral bounds.

4.3 Strengthen semi-continuous and semi-integer bounds

A semi-continuous variable is a variable that can take the value zero or any value between the specified lower and upper semi-continuous bounds. A lower semi-continuous bound must be finite and non-negative. An upper semi-continuous bound must be non-negative but does not need to be finite.

Let a semi-continuous variable x_j with $x_j = 0 \ \vee \ (\ell_j \leq x_j \leq u_j)$ be given. If we can prove $x_j \geq \ell'_j > 0$, then we can convert x_j into a regular continuous variable with $\max\{\ell_j, \ell'_j\} \leq x_j \leq u_j$. Conversely, if we can prove $x_j < \ell_j$ the variable must be zero and we can fix $x_j := 0$. Finally, if $\ell_j = 0$ we can discard the "semi" property of the variable and interpret the variable as a regular continuous variable.

Semi-integer variables are defined analogously to semi-continuous variables with the additional property that they need to take integer values.

Table 11: Impact of semi-continuous and semi-integer bound strengthening

bracket	models	default	disable semi-variable bound strengthening					affected	
		tilim	tilim	faster	slower	time	nodes	models	time
all	3180	571	571	5	4	1.00	1.00	—	—
$\geq$ 0 sec	2605	1	1	5	4	1.00	1.00	13	1.30
$\geq$ 1 sec	1734	1	1	5	4	1.00	1.00	13	1.30
$\geq$ 10 sec	1171	1	1	5	3	1.00	1.00	11	1.36
$\geq$ 100 sec	615	1	1	3	2	1.01	1.01	7	1.76
$\geq$ 1000 sec	155	1	1	1	1	1.03	1.05	3	4.88

We can apply the same reductions as for semi-continuous variables, but in addition semi-integer bounds can be rounded as in Section 4.2.

The computational results in Table 11 are not very meaningful: we only have 55 models with semi-continuous or semi-integer variables in our test set, and from those only 13 are affected by semi bound strengthening. Moreover, the speed-up on the affected models mainly originates from a single model, which solves in 8.3 seconds using default settings and hits the time limit without strengthening the bounds of the semi-continuous variables.

4.4 Dual fixing, substitution and bound strengthening

If the coefficients $A_{\cdot j}$ and the objective coefficient c_j of variable $j \in N$ fulfill certain conditions, we can sometimes derive variable fixings or implied bounds. In the following we assume that all constraints are either equations, $\circ_i =$ "=", or less-or-equal inequalities, $\circ_i =$ "$\leq$".

For the presolve reductions in this section we consider variables $j \in N$ that do not appear in equations, i.e., $a_{ij} = 0$ for all $i \in M$ with $\circ_i =$ "=". Then, for *dual fixing*, we distinguish two cases:

(i) $c_j \geq 0$, $a_{ij} \geq 0$ for all $i \in M$, and

(ii) $c_j \leq 0$, $a_{ij} \leq 0$ for all $i \in M$.

In case (i) we can fix $x_j := \ell_j$ if $\ell_j > -\infty$. If $\ell_j = -\infty$ and $c_j > 0$ the problem is unbounded or infeasible. For $\ell_j = -\infty$ and $c_j = 0$ we remove the variable and all constraints with $a_{ij} \neq 0$ from the problem. In a post-processing step, after an optimal solution for the presolved problem has been found, we can always select a finite value for x_j such that all constraints in which it appears are satisfied.

Case (ii) is analogous: if u_j is finite, we fix $x_j := u_j$; otherwise, the problem is unbounded or infeasible for $c_j < 0$, and for $c_j = 0$ we can remove the variable and its constraints from the problem.

If we are in case (i) except for one row $r \in M$ with $a_{rj} < 0$ we might be able to substitute x_j for some binary variable x_k. Namely, if $x_k = 0$ implies the redundancy of constraint r, i.e., $\sup\{A_{rS}\} \leq b_r$ for $S = N \setminus \{k\}$, and $x_k = 1$ implies $x_j = u_j$ (due to constraint r or other constraints), then we can substitute $x_j := \ell_j + (u_j - \ell_j) \cdot x_k$. A similar approach can be applied to case (ii), which leads to a substitution $x_j := u_j + (\ell_j - u_j) \cdot x_k$.

Example 4. Consider the mixed integer program

$$\begin{array}{rrrrrrrrrl} \min & x_1 + & x_2 + & x_3 & & & & & & \\ & 2x_1 + & 4x_2 - & 3x_3 & \leq & & 8 & & & (1) \\ & & - & x_2 - & x_3 & \leq & -4 & & & (2) \\ & -x_1 - & & x_2 + & 8x_3 & \leq & 0 & & & (3) \\ 0 \leq & x_1\,, & & x_2 & & \leq & 4 & & & \\ & & & & x_3 & \in & \{0,1\} & & & \end{array}$$

Variable x_1 only appears in constraint $r = 3$ with a negative coefficient. If $x_3 = 0$, then constraint (3) is redundant, and there is no need to set x_1 to any value larger than 0. On the other hand, $x_3 = 1$ forces $x_1 = x_2 = 4$. Hence, we can substitute $x_1 := 4x_3$.

In the general case where we are unable to fix or substitute a variable, we may still be able to derive tighter bounds using dual arguments. Given a variable x_j with $c_j \geq 0$, let

$$\begin{aligned} S &= N \setminus \{j\}, \\ M^+ &= \{i \in M : a_{ij} > 0\}, \\ M^- &= \{i \in M : a_{ij} < 0\}, \end{aligned}$$

and still assume $\circ_i =$ "$\leq$" for all $i \in M^+ \cup M^-$. Now, consider an assignment $x_j = \tilde{u}_j < u_j$ with $\tilde{u}_j \in \mathbb{Z}$ if $j \in I$. If all constraints in M^- get redundant, i.e.,

$$a_{ij}\tilde{u}_j + \sup\{A_{iS}x_S\} \leq b_i \;\text{ for all } i \in M^-,$$

then $\tilde{u}_j$ is a valid upper bound for x_j, and we can replace $u_j := \tilde{u}_j$. Note that it is easy to calculate the smallest valid $\tilde{u}_j$ or to show that no such $\tilde{u}_j$ exists.

Analogously, if $c_j \leq 0$ we may be able to update the lower bound of the variable. If $\tilde{\ell}_j > \ell_j$, $\tilde{\ell}_j \in \mathbb{Z}$ if $j \in I$, and $a_{ij}\tilde{\ell}_j + \sup\{A_{iS}x_S\} \leq b_i$ for all $i \in M^+$, we can replace $\ell_j := \tilde{\ell}_j$.

Table 12: Impact of dual fixing, substitution and bound strengthening

		default	disable dual fixing					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3162	570	576	339	417	1.02	0.97	—	—
$\geq$ 0 sec	2610	23	29	339	417	1.02	0.97	1386	1.04
$\geq$ 1 sec	1764	23	29	328	393	1.03	0.98	1068	1.04
$\geq$ 10 sec	1212	23	29	260	311	1.04	0.99	794	1.06
$\geq$ 100 sec	650	23	29	155	194	1.07	1.01	461	1.11
$\geq$ 1000 sec	213	23	29	58	77	1.25	1.31	169	1.32

Example 5. Consider the linear program

$$\begin{aligned} \min \quad & x_1 + x_2 + x_3 & & \\ & 2x_1 + 4x_2 - 3x_3 \leq 8 & & (1)\\ & - x_2 - x_3 \leq -4 & & (2)\\ & -2x_1 - 2x_2 + x_3 \leq 6 & & (3)\\ 0 \leq \quad & x_1\,, \quad x_2\,, \quad x_3 \leq 10 & & \end{aligned}$$

For variable x_1 we have $M^+ = \{1\}$ and $M^- = \{3\}$. For constraint (3) we see that $x_1 = 2$ suffices to render it redundant because $-2\cdot 2 + \sup\{-2x_2 + x_3\} = -4 + 10 \leq 6$. Thus, $x_1 \leq 2$ is a valid upper bound.

Table 12 shows the computational impact of the dual reductions of this section. About half of the models are affected, and we obtain a 4% speed-up on average in the "$\geq$ 10 sec" bracket. This performance improvement seems to originate from the subset of harder models: while the reduction even leads to a small increase in the number of nodes if we consider all models in our test set, we observe a 25% speed-up and a 31% reduction in node count on the 213 models in the "$\geq$ 1000 sec" category.

4.5 Substitute implied free variables

Let ℓ_j and u_j denote the explicit bounds of variable x_j, and $\bar{\ell}_j$ and $\bar{u}_j$ be the tightest implied bounds of problem (2.1) that can be discovered using bound strengthening of Section 3.2 without exploiting the explicit bounds of x_j. If $[\bar{\ell}_j, \bar{u}_j] \subseteq [\ell_j, u_j]$, we call x_j an *implied free variable*, which includes the case of *free variables* with $\ell_j = -\infty$ and $u_j = \infty$.

Example 6. Consider the following constraints:

$$x_1 - x_2 \leq 0 \tag{1}$$
$$x_1 - x_3 \geq 0 \tag{2}$$
$$0 \leq x_1 \leq 1 \tag{3}$$
$$0 \leq x_2 \leq 1 \tag{4}$$
$$0 \leq x_3 \leq 1 \tag{5}$$

(1) and (4) imply $x_1 \leq 1$. (2) and (5) imply $x_1 \geq 0$. Hence x_1 is an implied free variable: we could remove the bounds (3) of the variable without modifying the feasible space.

Suppose we have a constraint of the form

$$A_{iS}x_S + a_{ij}x_j = b_i$$

with $S \subseteq N \setminus \{j\}$. If x_j is an implied free continuous variable it can be substituted out of the problem by

$$x_j := \big(b_i - A_{iS}x_S\big)/a_{ij}.$$

If the variable is integer we can only perform the substitution if $S \subseteq I$ and $a_{ik}/a_{ij} \in \mathbb{Z}$ for all $k \in S$. If $b_i/a_{ij} \notin \mathbb{Z}$ in this case, then the problem is infeasible.

An implied free variable substitution can increase the number of nonzeros in the coefficient matrix A. Moreover, if the pivot element $|a_{ij}|$ is very small, applying the substitution may lead to numerical problems because of the potential round-off errors in the update operations for the matrix. For this reason, we only substitute x_j if the estimated fill-in is below some threshold and for the pivot element a_{ij} we have

$$|a_{ij}| \geq 0.01 \cdot \max\{|a_{rj}| : r \in M\} \;\text{ or }\; |a_{ij}| \geq 0.01 \cdot \max\{|a_{ik}| : k \in N\}.$$

This numerical safeguard is similar to a Markowitz type criterion for factorizing a matrix, see Markowitz [29] and Tomlin [37]. The Markowitz tolerance of 0.01 is increased to 0.5 or 0.9 for larger values of the "numerical focus" parameter. But even though these thresholds are pretty conservative, implied free variable aggregation is one of the main sources for numerical issues in the presolving step, and the user should consider to disable the aggregator for numerically challenging models.

As already mentioned above, the implied free variable substitution is the most important presolve algorithm within Gurobi's set of single-column

Table 13: Impact of implied free variable substitution

		default	disable variable substitution					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3148	567	618	340	841	1.18	1.01	—	—
$\geq$ 0 sec	2606	30	81	340	841	1.23	1.07	1800	1.34
$\geq$ 1 sec	1790	30	81	328	795	1.32	1.09	1366	1.44
$\geq$ 10 sec	1250	30	81	247	607	1.42	1.11	1003	1.54
$\geq$ 100 sec	732	30	81	147	389	1.58	1.18	608	1.73
$\geq$ 1000 sec	300	30	81	60	191	1.98	1.30	272	2.13

Table 14: Impact of disabling all multi-row reductions

		default	disable multi-row reductions					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3158	569	615	514	806	1.13	1.03	—	—
$\geq$ 0 sec	2616	32	78	514	806	1.17	1.09	2013	1.23
$\geq$ 1 sec	1788	32	78	488	743	1.24	1.11	1536	1.29
$\geq$ 10 sec	1266	32	78	346	597	1.34	1.12	1116	1.40
$\geq$ 100 sec	738	32	78	200	375	1.49	1.21	667	1.57
$\geq$ 1000 sec	286	32	78	74	158	2.02	1.41	272	2.10

reductions. Table 13 shows a significant performance degradation when the aggregator is disabled. We get 51 more time limit hits, and the average time to solve a model in the "$\geq$ 10 sec" bracket increases by 42%.

5 Reductions that consider multiple constraints at the same time

In this section we provide an overview of the methods in Gurobi that consider multiple rows at the same time to derive presolve reductions. In contrast to the single-row reductions, these are usually more complex and require work limits in order to avoid an excessive computational overhead in practice.

Table 14 shows the total performance impact of the multi-row presolve algorithms as implemented in Gurobi. Except for the redundancy detection, each of the reductions discussed in this section yields a considerable improvement, leading to a total speed-up of 34% in the "$\geq$ 10 sec" bracket. Interestingly, this is even larger than the corresponding value for the single-row reductions, compare Table 2 on page 6.

5.1 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 5. *Given a MIP* $= (M, N, I, A, b, c, \circ, \ell, u)$ *of form* (2.1) *with its set* P_{MIP} *of feasible solutions. For* $r \in M$ *let* MIP_r *be the mixed integer program obtained from MIP 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 Bixby et al. [11] and Achterberg and Wunderling [5], 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 $S = M \setminus \{r\}$, 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

Table 15: Impact of enabling dependent row detection

		default	enable dependent row detection					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3175	569	575	195	203	1.02	1.02	—	—
$\geq$ 0 sec	2615	13	19	195	203	1.02	1.02	737	1.07
$\geq$ 1 sec	1754	13	19	188	201	1.03	1.04	602	1.09
$\geq$ 10 sec	1196	13	19	151	147	1.04	1.04	430	1.11
$\geq$ 100 sec	644	13	19	95	86	1.05	1.05	232	1.17
$\geq$ 1000 sec	193	13	19	34	47	1.25	1.29	93	1.65

equation (2.3), see Section 3.1: 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 5.2. 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 [30]. 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.

Except for the mentioned cases of the single-row redundancy, the parallel row detection, and the full LP solves for very special cases, Gurobi does not apply any additional redundancy checks in default settings. The reason is that it does not help to improve the performance, as can be seed in Tables 15 to 17. Table 15 provides statistics for enabling the dependent row detection using a rank revealing LU factorization. Doing so degrades the performance by 4% in the "$\geq$ 10 sec" bracket. But note that this is not due to the overhead associated with the factorization: the node count increases by the same amount. We do not completely understand why removing redundant rows can hurt the MIP solving process, but our hypothesis is that those redundant rows can be useful as base inequalities to separate cutting planes.

Table 16 shows the impact of discarding set packing inequalities (also known as "clique inequalities") that are dominated by another constraint. Note that this is a case of integer domination, which means that removing the clique inequality can weaken the LP relaxation. If needed, however, the clique can be added back to the LP by separating it as a clique cut. The computational results indicate that this reduction neither hurts nor helps.

Table 16: Impact of enabling clique subsumption detection

		default	enable clique subsumption detection					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3178	571	565	144	164	1.00	1.01	—	—
$\geq$ 0 sec	2614	12	6	144	164	1.00	1.02	553	1.01
$\geq$ 1 sec	1751	12	6	142	162	1.00	1.02	496	1.01
$\geq$ 10 sec	1191	12	6	113	131	1.01	1.01	364	1.01
$\geq$ 100 sec	629	12	6	78	79	1.00	1.01	217	1.00
$\geq$ 1000 sec	173	12	6	27	35	1.01	1.03	80	1.03

Table 17: Impact of enabling inequality subsumption detection

		default	enable inequality subsumption detection					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3179	571	572	184	194	1.00	1.01	—	—
$\geq$ 0 sec	2616	13	14	184	194	1.00	1.01	637	1.00
$\geq$ 1 sec	1751	13	14	181	192	1.00	1.02	564	1.00
$\geq$ 10 sec	1197	13	14	150	168	1.01	1.02	439	1.02
$\geq$ 100 sec	633	13	14	102	107	1.01	1.03	269	1.01
$\geq$ 1000 sec	187	13	14	35	47	1.07	1.14	102	1.14

It affects about a third of the models in the test set, but its impact on the solve time is marginal.

The third method, which is evaluated in Table 17, identifies inequalities $ax \leq b$ that are subsets of some other constraint $ax + dy \circ b'$ with $\circ \in \{\text{“}\leq\text{”}, \text{“}=\text{”}\}$. Now, if $\inf\{dy\} \geq b' - b$ then $ax \leq b$ is dominated by $ax + dy \circ b'$ and can be discarded. On the other hand, if $\circ =$ “$\leq$” and $\sup\{dy\} \leq b' - b$ then $ax + dy \leq b'$ is dominated by $ax \leq b$. Unfortunately, even though this is a case of LP redundancy, discarding subsumed constraints does not speed up the MIP solving process either. It affects about a third of the non-trivial models but is only performance neutral.

5.2 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 [6], 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., $|\text{supp}\,(A_{\cdot j})| = 1$. 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

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

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

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

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 such "nearly parallel" rows is done together with the regular parallel row detection, see again [6]. 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.

The parallel and nearly parallel row detection provides a small performance improvement, as can be seen in Table 18. In the "$\geq$ 10 sec" bracket, it affects more than half of the models, and it reduces the average solve time on these models by 5%, which leads to an overall 3% speed-up.

Table 18: Impact of disabling parallel row detection

		default	disable parallel row detection					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3170	570	573	306	371	1.01	0.99	—	—
$\geq$ 0 sec	2609	14	17	306	371	1.02	1.00	1237	1.03
$\geq$ 1 sec	1762	14	17	298	356	1.02	1.01	1003	1.04
$\geq$ 10 sec	1204	14	17	240	281	1.03	1.01	742	1.05
$\geq$ 100 sec	656	14	17	152	170	1.04	1.03	429	1.06
$\geq$ 1000 sec	194	14	17	60	65	1.07	1.06	153	1.08

5.3 Non-zero cancellation

Adding equations to other constraints yields 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 [14]. More precisely, assume we have two rows

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

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 scalar $s \in \mathbb{R}$ such that $sA_{qS} = A_{rS}$ and $sA_{qj} \neq A_{rj}$ for all $j \in T$. Then, subtracting s times row q from row r yields the modified system:

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

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

Table 19: Impact of disabling non-zero cancellation

		default	disable non-zero cancellation					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3171	570	568	275	351	1.02	1.00	—	—
$\geq$ 0 sec	2619	23	21	275	351	1.03	1.01	1138	1.07
$\geq$ 1 sec	1768	23	21	267	337	1.04	1.01	936	1.08
$\geq$ 10 sec	1214	23	21	216	278	1.05	1.00	693	1.09
$\geq$ 100 sec	661	23	21	129	163	1.07	1.02	387	1.13
$\geq$ 1000 sec	212	23	21	51	73	1.22	1.17	151	1.32

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.

The computational impact of the non-zero cancellation in Gurobi is surprisingly large, see Table 19. More than half of the models in the "$\geq$ 10 sec" bracket are affected, and the 9% performance improvement on these models translates into a 5% overall speed-up. The fact that the node count does not change much indicates that the performance advantage originates from the sparsity and the associated speed-ups in other parts of the algorithm, in particular the linear system solves of the simplex solver.

5.4 Bound and coefficient strengthening

Savelsbergh [34] 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. As in Sections 3.2 and 3.3 we consider inequality (3.3):

$$A_{iS}x_S + a_{ik}x_k \leq b_i$$

with $a_{ik} \neq 0$ and calculate bounds $\ell_{iS}, u_{iS} \in \mathbb{R} \cup \{-\infty, \infty\}$ such that

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

for all integer feasible solutions $x \in P_{\text{MIP}}$. Using ℓ_{iS} we can potentially tighten one of the bounds of x_k, see Section 3.2. If x_k is an integer variable we can use u_{iS} to strengthen the coefficient of x_k in inequality (3.3), see Section 3.3.

Now, the important question is how to calculate the bounds

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

with reasonable computational effort, so that they are valid for all $x \in P_{\text{MIP}}$ and as tight as possible. The tighter the bounds on the activity of $A_{iS}x_S$, the better the bound and coefficient strengthening for x_k. Recall that for single-row preprocessing we just use $\ell_{iS} = \inf\{A_{iS}x_S\}$ and $u_{iS} = \sup\{A_{iS}x_S\}$. This is very cheap to calculate, but it may not be very tight.

The other extreme would be to actually calculate

$$\begin{aligned}\ell_{iS} &= \min\{A_{iS}x_S : x \in P_{\text{MIP}}\} \text{ and}\\ u_{iS} &= \max\{A_{iS}x_S : x \in P_{\text{MIP}}\},\end{aligned}$$

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_{iS} and u_{iS}:

$$\begin{aligned}\ell_{iS} &= \min\{A_{iS}x_S : x \in P_{\text{LP}}\} \text{ and}\\ u_{iS} &= \max\{A_{iS}x_S : x \in P_{\text{LP}}\}.\end{aligned}$$

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 [18] and Balas and Zemel [9].

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

$$A_{iS_1}x_{S_1} + \ldots + A_{iS_d}x_{S_d} + a_{ik}x_k \leq b_i$$

with $S_1 \cup \ldots \cup S_d = N \setminus \{k\}$ and $S_{p_1} \cap S_{p_2} = \emptyset$ for $p_1 \neq p_2$. Then we calculate individual bounds

$$\ell_{iS_p} \leq A_{iS_p}x_{S_p} \leq u_{iS_p}$$

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

$$\begin{aligned} \ell_{iS} &:= \ell_{iS_1} + \ldots + \ell_{iS_d} \text{ and} \\ u_{iS} &:= u_{iS_1} + \ldots + u_{iS_d} \end{aligned}$$

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

$$\ell_{iS_p} = \inf\{A_{iS_p}x_{S_p}\} \text{ and } u_{iS_p} = \sup\{A_{iS_p}x_{S_p}\}$$

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 [35]. Here, the "block" that we want to minimize and maximize consists of just a single variable:

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

The result directly yields stronger bounds for x_k. 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

Table 20: Impact of disabling multi-row bound and coefficient strengthening

		default	disable multi-row bound/coeff. str.					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3159	570	577	477	500	1.02	1.01	—	—
$\geq$ 0 sec	2611	27	34	477	500	1.02	1.02	1709	1.05
$\geq$ 1 sec	1764	27	34	457	469	1.03	1.03	1326	1.06
$\geq$ 10 sec	1218	27	34	340	372	1.06	1.04	953	1.08
$\geq$ 100 sec	671	27	34	201	227	1.07	1.06	539	1.11
$\geq$ 1000 sec	232	27	34	69	96	1.22	1.24	200	1.26

programming to get tighter relaxations, OBBT is usually applied very aggressively in MINLP and global optimization solvers. 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.

Table 20 illustrates the performance impact of the multi-row bound and coefficient strengthening as implemented in Gurobi. It yields a 6% speed-up in the "$\geq$ 10 sec" bracket, which is quite significant, given that this comes on top of the improvements provided by the single-row methods, compare Sections 3.2 and 3.3.

5.5 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 [7]. 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 7. 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 [25] and Savelsbergh [34]. Subsequently, we discard constraints that are now dominated by the extended set packing constraint (see Definition 2 on page 10). 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.

An interesting special case of clique merging arises for set packing constraints with a small number of elements. In this case, we can use an alternative data structure for the merging algorithm that is more efficient.

As we have seen above, a set packing constraint of length $l = |S \cup T|$ gives rise to $l(l-1)/2$ edges in the conflict graph. Storing those edges explicitly for long constraints would lead to a very large memory footprint of the algorithm. Hence, the standard clique merging algorithm in Gurobi works directly on the constraint matrix, i.e., on a conflict hyper-graph that is given by an incidence matrix with one hyper-edge for each set partitioning constraint in the model. The main operation in the clique merging algorithm is to find all neighbors of a given vertex v. Using the sparse constraint matrix, this means to go through the column of the variable x_j associated with vertex v, and for each non-zero entry in this column to scan the corresponding

Table 21: Impact of disabling clique merging

		default	disable clique merging					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3174	569	587	300	347	1.05	1.02	—	—
$\geq$ 0 sec	2617	17	35	300	347	1.07	1.05	1042	1.18
$\geq$ 1 sec	1761	17	35	291	329	1.10	1.06	881	1.21
$\geq$ 10 sec	1228	17	35	231	283	1.14	1.08	678	1.28
$\geq$ 100 sec	670	17	35	131	181	1.21	1.12	403	1.39
$\geq$ 1000 sec	211	17	35	47	74	1.58	1.42	153	1.87

row. On average, this operation touches $\bar{l}_c + \bar{l}_c \cdot \bar{l}_r$ non-zero matrix entries, with $\bar{l}_c$ and $\bar{l}_r$ being the average number of non-zeros in the columns and rows, respectively.

In the extreme case of a model that consists only of set partitioning constraints of length 2, as in Example 7, we get $3\bar{l}_c$ memory accesses on average for a neighbor scan. In particular, in each constraint we rediscover x_j, which is unnecessary. This can be rectified by extracting all short set packing constraints (Gurobi uses a limit of length 100) into a graph that stores the edges explicitly as adjacency lists. In our algorithm, we only need to store one direction of each edge, i.e., for indices $u < v$ we only store $u \to v$ but not $v \to u$. Thus, in the worst case for a model with all set packing constraints having length 100, we store 4950 edges for each constraint instead of working on two matrices (the clique matrix and its transpose) with each having 100 non-zeros per constraint. Consequently, our memory overhead is at most a factor of 24.75 compared to the incidence matrix based algorithm. The advantage is to have fewer non-zero accesses in the neighbor finding operation. For the case of set partitioning constraints of length 2, we only touch one non-zero for each neighbor, i.e., we get from $3\bar{l}_c$ down to $\bar{l}_c$ memory accesses. Typically, this directly translates into a run-time improvement of almost a factor of 3 for those extreme cases.

Table 21 shows the computational impact of the clique merging algorithms. If clique merging is disabled, we lose 28% performance on the 678 models that are affected in the "$\geq$ 10 sec" bracket. Overall, a 14% reduction in time and an 8% reduction in nodes can be observed.

Table 22: Impact of disabling all multi-column reductions

		default	disable multi-column reductions					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3173	570	572	337	459	1.04	1.02	—	—
$\geq$ 0 sec	2616	18	20	337	459	1.05	1.04	1454	1.09
$\geq$ 1 sec	1770	18	20	326	441	1.07	1.05	1135	1.12
$\geq$ 10 sec	1219	18	20	252	340	1.09	1.06	819	1.14
$\geq$ 100 sec	666	18	20	144	211	1.12	1.08	462	1.17
$\geq$ 1000 sec	210	18	20	54	79	1.21	1.14	166	1.25

6 Reductions that consider multiple variables at the same time

A few of the presolve reductions implemented in Gurobi deal with multiple columns of the problem at the same time. For a given subset $S \subseteq N$ these methods look at the matrix columns $A_{\cdot j}$, the corresponding objective coefficients c_j, bounds ℓ_j and u_j, and the integrality status of the variables $j \in S$, as well as the senses $\circ_i$ and infimum $\inf\{A_{i\cdot}x\}$ and supremum $\sup\{A_{i\cdot}x\}$ values of the rows $i \in M$ in which the variables appear with a non-zero coefficient.

As can be seen in Table 22, the multi-column reductions of Gurobi are not as successful as the multi-row presolve algorithms. They only provide a modest speed-up of 9% in the "$\geq$ 10 sec" bracket, with the largest contribution coming from the parallel column detection of Section 6.3.

6.1 Extension of dual fixing for single equations

If problem (2.1) has a specific structure, we can apply an extension of dual fixing from Section 4.4. Assume that a variable x_j appears only in one equation and could otherwise be fixed to a bound by dual reductions. If we have continuous variables in this equation that can always compensate for the fixing of the variable x_j, we can set x_j to its bound.

More formally, let $r \in M$ be a row index with $\circ_r =$ "=". For convenience, assume that $a_{rj} \geq 0$ for all $j \in N$. If this was not the case, we could just invert the variables with negative coefficient using $x'_j := -x_j$ to make the coefficient in row r positive.

Let $S_+ \subseteq \operatorname{supp}(A_{r\cdot})$ be the set of variables $s \in S_+$ in the equation with

$c_s \geq 0$ and

$$\begin{aligned} a_{is} &\geq 0 \quad \text{for all } i \in M \text{ with } \circ_i = \text{“}\leq\text{”}, \\ a_{is} &= 0 \quad \text{for all } i \in M \setminus \{r\} \text{ with } \circ_i = \text{“}=\text{”}, \end{aligned}$$

and let $S_- \subseteq \operatorname{supp}(A_{r\cdot}) \setminus S_+$ be the set of variables $s \in S_-$ in the equation with $c_s \leq 0$ and

$$\begin{aligned} a_{is} &\leq 0 \quad \text{for all } i \in M \text{ with } \circ_i = \text{“}\leq\text{”}, \\ a_{is} &= 0 \quad \text{for all } i \in M \setminus \{r\} \text{ with } \circ_i = \text{“}=\text{”}. \end{aligned}$$

We assume that no constraints with $\circ_i = \text{“}\geq\text{”}$ exist. By this definition, S_+ and S_- are the indices of the variables for which equation r is the only constraint in the problem that prevents us from pushing them to their lower or upper bounds, respectively, using the reduction of Section 4.4.

Let $S_-^C := S_- \setminus I$ be the set of continuous variables in S_- and $T_- := N \setminus S_-^C$ the set of remaining variables in the problem. Now, if

$$\inf\{A_{rT_-} x_{T_-}\} + \sup\{A_{rS_-^C} x_{S_-^C}\} \geq b_r$$

then we can fix $x_j := \ell_j$ for all $j \in S_+$, because fixing x_j to its lower bound is not hurting the objective or the feasibility of any of the constraints in $M \setminus \{r\}$ by definition of S_+, and a potential shortfall in the activity of r, i.e., $A_{r\cdot} x < b_r$, due to $x_j = \ell_j$ can always be compensated by increasing variables in S_-^C. Increasing those variables does not impair optimality or feasibility either, due to the definition of S_-.

Analogously, for $S_+^C := S_+ \setminus I$ and $T_+ := N \setminus S_+^C$, if

$$\sup\{A_{rT_+} x_{T_+}\} + \inf\{A_{rS_+^C} x_{S_+^C}\} \leq b_r$$

then we can fix $x_j := u_j$ for all $j \in S_-$.

As expected, there are just a handful of models that are affected by this presolve reduction, see Table 23. It is disabled in default settings of Gurobi 6.5, but enabling the multi-column dual fixing for equations seems to help on the 40 models where it can be applied. Nevertheless, such a small set of problem instances does not provide any meaningful results, and indeed, additional tests that we have conducted showed that the speed-up in Table 23 is just an artifact of performance variability.

Table 23: Impact of enabling multi-column dual fixing for single equations

		default	enable dual fixing extension					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3179	571	569	20	6	1.00	1.00	—	—
$\geq$ 0 sec	2605	2	0	20	6	1.00	1.00	40	0.88
$\geq$ 1 sec	1735	2	0	20	6	1.00	1.00	37	0.87
$\geq$ 10 sec	1171	2	0	19	5	1.00	1.00	35	0.86
$\geq$ 100 sec	614	2	0	13	3	0.99	1.00	24	0.81
$\geq$ 1000 sec	155	2	0	3	1	1.00	1.01	5	0.98

6.2 Fix redundant penalty variables

Let us assume problem (2.1) is given with $\circ_i =$ "$\leq$" for all $i \in M$. In this context a *penalty variable* x_p, $p \in N$, is a singleton variable with $c_p > 0$, $\text{supp}\,(A_{\cdot p}) = \{r\}$, and $a_{rp} < 0$, see also Gamrath et al. [22].

Consider the problem

$$\begin{aligned} \min \;\; c_P^T x_P + \;\; & c_V^T x_V \\ A_{rP} x_P + \; & A_{rV} x_V \leq b_r \\ & A_{TV} x_V \leq b_T \end{aligned}$$

with $\ell \leq x \leq u$, $x_j \in \mathbb{Z}$ for all $j \in I$, $N = P \cup V$, $P \cap V = \emptyset$, $c_P > 0$, and $A_{rP} < 0$. First, consider the case that all penalty variables are continuous, $P \cap I = \emptyset$. W.l.o.g. assume that $P = \{1, \ldots, p\}$ and

$$\frac{c_1}{|a_{r1}|} \leq \ldots \leq \frac{c_p}{|a_{rp}|}.$$

Now, if for some $k \in \{1, \ldots, p\}$ we have

$$\sup\{A_{rV} x_V\} + \sum_{t=1}^{k} a_{rt} u_t + \sum_{t=k+1}^{p} a_{rt} \ell_t \leq b_r,$$

then we can fix $x_{k+1} := \ell_{k+1}, \ldots, x_p := \ell_p$ because these more expensive penalty variables are never needed for feasibility of constraint r.

We can apply the same reduction for integer penalty variables if $P \subseteq I$ and all penalty variables have the same weight in the constraint, $A_{rj} = t$ for all $j \in P$ and some $t < 0$. This could even be generalized to the case of integer penalties with non-identical weights using divisibility arguments on those weights, but such a generalization is not implemented in Gurobi.

Redundant penalty variables can be fixed in about 10% of the models in our test set, see Table 24. On those models, it yields a 3 to 4% performance

Table 24: Impact of disabling redundant penalty variable fixing

		default	disable redundant penalty fixing					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3180	571	570	52	65	1.00	1.00	—	—
$\geq$ 0 sec	2610	6	5	52	65	1.00	1.01	249	1.03
$\geq$ 1 sec	1741	6	5	51	64	1.00	1.01	217	1.03
$\geq$ 10 sec	1179	6	5	44	58	1.01	1.01	173	1.04
$\geq$ 100 sec	625	6	5	28	38	1.00	1.01	99	1.02
$\geq$ 1000 sec	163	6	5	17	15	0.98	0.96	43	0.94

improvement, but one has to take this result with care, as a set of about 200 models is usually not large enough to measure such an impact precisely enough.

6.3 Parallel columns

Two columns $A_{\cdot j}$ and $A_{\cdot k}$ with $j \neq k$ of problem (2.1) are *parallel*, if there exists $\lambda \in \mathbb{R} \setminus \{0\}$ with $A_{\cdot k} = \lambda A_{\cdot j}$.

If $A_{\cdot j}$ and $A_{\cdot k}$ are parallel, $c_k = \lambda c_j$, and both variables x_j, x_k fulfill certain requirements on their variable types, we can merge x_j and x_k into a new variable y by

$$y := x_j + \lambda x_k \tag{6.1}$$

with bounds

$$\begin{aligned} \ell_y &= \begin{cases} \ell_j + \lambda \ell_k, & \text{for } \lambda > 0 \\ \ell_j + \lambda u_k, & \text{for } \lambda < 0 \end{cases} \\ u_y &= \begin{cases} u_j + \lambda u_k, & \text{for } \lambda > 0 \\ u_j + \lambda \ell_k, & \text{for } \lambda < 0 \end{cases} \end{aligned}$$

and objective coefficient $c_y = c_j$. Now we can solve the reduced problem in order to obtain a solution value for y. In a post-processing ("uncrush") step, by using (6.1) and paying attention to the bounds of x_j and x_k we can split the value of y into feasible solution values for x_j and x_k.

Note, however, that we need to take special care of integer variables, because even if the solution for y is integral we may not always be able to derive integral values for x_j and x_k.

W.l.o.g. we now assume $j \in I$. If $k \notin I$ and $|\lambda(u_k - \ell_k)| \geq 1$ we can merge x_j and x_k into a new continuous variable y, because in the post-processing step we will always be able to split the value of y into x_j and x_k such that x_j is integral.

If $k \in I$ we can merge x_j and x_k into an integer variable y if

$$\{x_j + \lambda x_k \,:\, x_j \in \{\ell_j, \ldots, u_j\}, x_k \in \{\ell_k, \ldots, u_k\}\} = \{\ell_y, \ldots, u_y\},$$

i.e., the image of $x_j + \lambda x_k$ over the domains of x_j and x_k is an interval of integers without any holes. In particular, this is true for $|\lambda| = 1$, but it may also hold for arbitrary $\lambda \in \mathbb{Z}$, depending on the bounds of x_j and x_k.

Example 8. In the linear program

$$\begin{aligned} \min\ & 2x_1 + 4x_2 + x_3 \\ \text{s.t.}\ & -x_1 - 2x_2 - x_3 \leq -10 \\ & 0 \leq x_1 \leq 3 \\ & 0 \leq x_2 \leq 4 \\ & 0 \leq x_3 \leq 5 \end{aligned}$$

the columns 1 and 2 are parallel with $A_{\cdot 2} = \lambda A_{\cdot 1}$, $c_2 = \lambda c_1$, and $\lambda = 2$. An optimal solution is given by $x^\star = (0, 2.5, 5)^T$. The bounds of

$$y := x_1 + 2x_2$$

are

$$\begin{aligned} \ell_y &= \ell_1 + \lambda \ell_2 \ = 0 \quad \text{and} \\ u_y &= u_1 + \lambda u_2 = 11. \end{aligned}$$

The resulting presolved model is

$$\begin{aligned} \min\ & 2y + x_3 \\ \text{s.t.}\ & -y - x_3 \leq -10 \\ & 0 \leq \ \ y \leq 11 \\ & 0 \leq x_3 \leq 5 \end{aligned}$$

with solution $y^\star = x_3^\star = 5$. By using the uncrush information $y := x_1 + 2x_2$ we set $x_1 = 0$ and $x_2 = 2.5$. In this example, the reduction is also possible for integer variables because any value $y \in \{0, \ldots, 11\}$ can be represented within the bounds of x_1 and x_2. The optimal solution $y^\star = x_3^\star = 5$ would be uncrushed into $x^\star = (1, 2, 5)^T$.

The detection of parallel columns uses the same code base as the one for parallel rows, see Section 5.2, which means to apply a two level hashing

Table 25: Impact of disabling parallel column detection

		default	disable parallel column detection					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3172	569	570	216	304	1.02	1.01	—	—
$\geq$ 0 sec	2616	18	19	216	304	1.02	1.01	1061	1.06
$\geq$ 1 sec	1759	18	19	209	297	1.04	1.03	849	1.08
$\geq$ 10 sec	1205	18	19	170	239	1.04	1.04	631	1.09
$\geq$ 100 sec	650	18	19	110	145	1.05	1.05	351	1.10
$\geq$ 1000 sec	199	18	19	50	59	1.08	1.02	138	1.12

algorithm and a linear scan over each set of columns with same hash values. Both the parallel row and parallel column detection are usually very fast in practice and do not need any work limits to avoid excessive running times.

The performance improvement due to parallel column detection is slightly larger than the one of parallel rows, compare Table 18 on page 30 with Table 25. We observe a 4% speed-up in the "$\geq$ 10 sec" bracket, with about half of the models affected by the reduction.

6.4 Dominated columns

This preprocessing technique analyzes and exploits dominance relations between two variables, see also Gamrath et al. [22] and the references therein. If the bounds and the integrality of both variables satisfy certain conditions, one can infer variable fixings.

Definition 6. *Let a MIP of form* (2.1) *with* $\circ_i =$ "$\leq$" *for all* $i \in M$*, and two variables* x_j *and* x_k *with* $k \neq j$ *be given.* x_j *dominates* x_k ($x_j \succ x_k$)*, if*

i) $c_j \leq c_k$,

ii) $a_{ij} \leq a_{ik}$ *for all constraints* $i \in M$*, and*

iii) $j \in N \setminus I$ *or* $k \in I$.

Condition iii) is necessary to rule out domination from integer to continuous variables.

We can transform the MIP into an equivalent problem by multiplying $A_{\cdot j}$, c_j and the swapped bounds of one variable x_j by -1. This will allow us to exploit the cases $x_j \succ -x_k$ and $-x_j \succ x_k$ as well.

Theorem 1. *Consider a MIP of form* (2.1) *with* $\circ_i =$ "$\leq$" *for all* $i \in M$*, and let* x_j, x_k *be two variables with* $j \neq k$*. Then, the following holds:*

i) If $u_j = \infty$ and $x_j \succ x_k$ then x_k can be set to ℓ_k.

ii) If $u_j = \infty$ and $x_j \succ -x_k$ then x_k can be set to u_k.

iii) If $\ell_k = -\infty$ and $x_j \succ x_k$ then x_j can be set to u_j.

iv) If $\ell_k = -\infty$ and $-x_j \succ x_k$ then x_j can be set to ℓ_j.

Proof. For case i) assume $x_j \succ x_k$ and $u_j = \infty$, and let $x^\star$ be an optimal solution. If $x_k^\star > \ell_k$ we can set $\Delta = x_k^\star - \ell_k$, increase $x_j^\star$ by Δ and decrease $x_k^\star$ by Δ. While doing this exchange by Δ we stay feasible because

$$\sum_{t \in N \setminus \{j,k\}} a_{it} x_t^\star + a_{ij}(x_j^\star + \Delta) + a_{ik}(x_k^\star - \Delta) = \underbrace{\sum_{t \in N} a_{it} x_j^\star}_{\leq b_i} + \underbrace{(a_{ij} - a_{ik})\Delta}_{\leq 0} \leq b_i$$

for all constraints $i \in M$. In addition, the objective value is not getting worse because $c^T x^\star + (c_j - c_k)\Delta \leq c^T x^\star$. Thus, if the problem has an optimal solution, there always exists one with $x_k^\star = \ell_k$. The other cases are similar to case i). □

Note that the requirement that one of the bounds is infinite is not as restrictive as it may sound. Namely, it suffices that the bound is *implied* by the constraints and the bounds of the other variables, compare also Section 4.5. Such an implied bound can be replaced with an infinite value without changing the solution space, and then we can apply the reductions of Theorem 1.

We find dominated columns by using a pair-wise comparison algorithm with some modifications to avoid useless work. We only consider variables with an infinite or implied bound as dominating columns. For each of those candidates, we find the shortest row in which it has a positive coefficient for $\leq$ inequalities or which is an equation. The other variables in this row are the candidates to be dominated. To compare two columns (vectors) for domination, we first check whether their support signatures already rule out domination; this signature is a 32 bit integer in which bit b is set if the column contains a row index i with $i \equiv b \pmod{32}$. If the signatures allow for domination, then we explicitly compare the column vectors using a linear scan.

Even though Table 26 indicates that a good fraction of the models in our test set contain dominated columns, the performance improvement of removing those variables is only marginal. We save 3% of run-time on the affected models in the "$\geq$ 10 sec" bracket, which translates into a 1% overall speed-up.

Table 26: Impact of disabling dominated column detection

		default	disable dominated column detection					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3175	571	566	244	263	1.01	1.00	—	—
$\geq$ 0 sec	2614	15	10	244	263	1.01	1.00	1026	1.02
$\geq$ 1 sec	1764	15	10	240	250	1.01	1.00	838	1.03
$\geq$ 10 sec	1202	15	10	189	188	1.01	1.02	619	1.03
$\geq$ 100 sec	644	15	10	118	119	1.00	1.02	360	1.01
$\geq$ 1000 sec	190	15	10	47	45	1.05	1.11	127	1.08

Table 27: Impact of disabling all full-problem reductions

		default	disable full-problem reductions					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3155	566	618	571	853	1.18	1.21	—	—
$\geq$ 0 sec	2621	37	89	571	853	1.23	1.27	2229	1.28
$\geq$ 1 sec	1813	37	89	534	777	1.33	1.35	1674	1.37
$\geq$ 10 sec	1299	37	89	392	612	1.45	1.45	1223	1.49
$\geq$ 100 sec	773	37	89	227	409	1.72	1.70	734	1.77
$\geq$ 1000 sec	292	37	89	85	177	2.54	2.52	288	2.57

7 Reductions that consider the whole problem

This section covers the presolve reductions of Gurobi that consider the full problem at the same time. This includes the most expensive algorithms such as probing or symmetry detection. These methods are called in the outermost loop of the Gurobi presolving procedure, which typically only runs for very few or even just one iteration.

As Table 27 shows, the performance impact of the full-problem reductions is significant. In the "$\geq$ 10 sec" bracket we observe a 45% speed-up, with probing and implied integer detection being the largest contributors.

7.1 Aggregate pairs of symmetric variables

By exploiting symmetry relations between variables we are sometimes able to substitute variables.

Definition 7. *Let $\pi : N \to N$ be a permutation of the index set N. We call π a* symmetry generator *for the MIP* (2.1)*, if*

i) $c_j = c_{\pi(j)}$, $\ell_j = \ell_{\pi(j)}$, $u_j = u_{\pi(j)}$, and $j \in I \Leftrightarrow \pi(j) \in I$ for all $j \in N$, and

ii) there exists a row permutation $\sigma : M \to M$ with $b_i = b_{\sigma(i)}$, $\circ_i = \circ_{\sigma(i)}$, and $A_{\sigma(i)\pi(j)} = A_{ij}$ for all $i \in M$ and $j \in N$.

If for all $j \in N$ with $\pi(j) \neq j$ we have $\operatorname{supp}(A_{\cdot j}) \cap \operatorname{supp}(A_{\cdot \pi(j)}) = \emptyset$*, then π is called a* non-overlapping *symmetry generator. If for all $j \in N$ with $\pi(j) \neq j$ we have $j \in N \setminus I$, then π is called a* continuous variable *symmetry generator.*

The identification of symmetries in the MIP (2.1) is done by Gurobi using an algorithm based on the ideas implemented in Saucy, see Darga et al. [19, 20]. This detection algorithm yields symmetry generators Π. Given a non-overlapping symmetry generator $\pi \in \Pi$ we can aggregate $x_j := x_{\pi(j)}$ for all $j \in N$ with $j \neq \pi(j)$, independent of whether variable x_j is integer or not, see Example 9. Gurobi performs this aggregation iteratively for all non-overlapping symmetry generators, which means that all variables in an orbit defined by these generators are aggregated into a single variable.

Additionally, for a continuous variable symmetry generator π we can aggregate $x_j := x_{\pi(j)}$ for all $j \in N$ with $j \neq \pi(j)$. Again, this means that all variables in an orbit defined by these generators are aggregated into a single variable, which corresponds to the arithmetic average of the variables in the orbit. Such an aggregation is not possible for integer variables because the arithmetic average does not preserve integrality.

Example 9. Consider the following problem with all variables binary.

$$\begin{array}{rrcrcrcrcrcrcrcl} \max & 4x_1 & + & 4x_2 & + & 2x_3 & + & 2x_4 & + & 3x_5 & + & 4x_6 & + & 5x_7 & \\ & x_1 & & & + & 4x_3 & & & + & x_5 & + & 3x_6 & + & 2x_7 & \leq & 8 \\ & & & x_2 & & & + & 4x_4 & + & x_5 & + & 3x_6 & + & 2x_7 & \leq & 8 \end{array}$$

This gives rise to the symmetry generator $\pi = (1 \leftrightarrow 2;\ 3 \leftrightarrow 4)$: if we swap x_1 with x_2 and x_3 with x_4 and then swap the two rows we get back to the original system. Neither x_1 and x_2, nor x_3 and x_4 appear in the same row. Thus, this is a non-overlapping symmetry generator. Hence, we can aggregate $x_2 := x_1$ and $x_4 := x_3$. This yields

$$\begin{array}{rrcrcrcrcrcrcl} \max & 8x_1 & + & 4x_3 & + & 3x_5 & + & 4x_6 & + & 5x_7 & & & \\ & x_1 & + & 4x_3 & + & x_5 & + & 3x_6 & + & 2x_7 & \leq & 8 \\ & x_1 & + & 4x_3 & + & x_5 & + & 3x_6 & + & 2x_7 & \leq & 8 \end{array}$$

and subsequently the parallel row detection of Section 5.2 will discard the second constraint.

Table 28: Impact of disabling symmetric variable aggregation

		default	disable symmetric variable aggregation					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3180	571	574	36	59	1.01	1.00	—	—
$\geq$ 0 sec	2609	5	8	36	59	1.02	1.00	148	1.31
$\geq$ 1 sec	1744	5	8	36	58	1.02	1.00	143	1.33
$\geq$ 10 sec	1180	5	8	32	49	1.03	1.01	130	1.36
$\geq$ 100 sec	627	5	8	24	38	1.06	1.03	88	1.50
$\geq$ 1000 sec	166	5	8	13	20	1.16	1.08	41	1.84

Table 28 provides benchmarking results for the symmetric variable aggregation. With its 3% speed-up in the "$\geq$ 10 sec" bracket the impact on the overall test set is modest, but for the about 10% of the models where it applies, the performance improvement is pretty significant. As we will also see in Section 8.2 on orbital fixing, this behavior is typical for reductions that exploit symmetry: they apply only on a relatively small subset of the models, but on those models they are very important.

7.2 Probing

The idea of probing is to set some binary variable tentatively to zero or one and derive further or stronger inequalities or better bounds, see Savelsbergh [34] and Achterberg [2].

Let x_k be a binary variable, and let x_j be an arbitrary variable with bounds $\ell_j \leq x_j \leq u_j$. Furthermore, let ℓ_j^0 and u_j^0 be the lower and upper bounds of x_j that have been deduced from setting $x_k := 0$ using bound strengthening as in Section 3.2. Correspondingly, let ℓ_j^1 and u_j^1 be the bounds of x_j calculated from setting $x_k := 1$. Then, the following observations can be made:

1. If setting $x_k = 0$ leads to an infeasible problem, we can fix $x_k := 1$. Conversely, if $x_k = 1$ is infeasible, we can fix $x_k := 0$.

2. If $\ell_j^0 = u_j^0$ and $\ell_j^1 = u_j^1$, x_j can be substituted as $x_j := \ell_j^0 + (\ell_j^1 - \ell_j^0) \cdot x_k$. Note that for $\ell_j^0 = \ell_j^1$ this means to fix $x_j := \ell_j^0$.

3. We can deduce valid global bounds of x_j by $\ell_j := \min\{\ell_j^0, \ell_j^1\}$ and $u_j := \max\{u_j^0, u_j^1\}$.

4. If none of the three cases above can be applied, then we can store valid

implications on the bounds of x_j depending on the value of x_k:

$$\begin{aligned} x_k = 0 \rightarrow x_j \geq \ell_j^0 \qquad & x_k = 1 \rightarrow x_j \geq \ell_j^1 \\ x_k = 0 \rightarrow x_j \leq u_j^0 \qquad & x_k = 1 \rightarrow x_j \leq u_j^1 \end{aligned} \tag{7.1}$$

Those implications can be used in the main solving process, for example for implied bound and MIR cut separation and for the branching variable decision.

Since tighter bounds and implications found in probing are already exploited for the domain propagation step of subsequent probing candidates, probing is sequence dependent. Moreover, probing can be very expensive and thus needs to be aborted prematurely if it does not seem to be successful enough. For this reason, two important aspects of Gurobi's probing algorithm are the order in which the probing candidates are processed and the criterion when probing is aborted.

The probing candidate order is based on the number of other binary variables that are directly implied by fixing the candidate variable to zero or one and propagating each constraint in isolation. This number can be calculated by a simple linear scan over the constraints in which the candidate participates. The more binary fixings are directly implied by a candidate, the earlier we are going to process it in the probing algorithm.

In order to avoid consuming too much time we install a probing work limit that is based on the number of columns and rows in the model. The work is counted roughly as the number of non-zero matrix entries that we touch in the domain propagation process. During probing, the work limit is dynamically adjusted based on the current success level: the base work limit is multiplied with a factor in $[0.01, 100]$ that is increased with the current number of derived variable fixings, substitutions, and bound changes.

7.2.1 Clique Coupling Constraints

An interesting problem structure that Gurobi's probing engine is exploiting is given by *clique coupling constraints*. In a preprocessing step of probing, we first identify cliques (also known as *set packing* or *set partitioning constraints*) in the constraint system. Those are constraints or implied constraints of the form

$$\sum_{j \in C^+} x_j - \sum_{j \in C^-} x_j \leq 1 - |C^-| \quad \text{or} \tag{7.2}$$

$$\sum_{j \in C^+} x_j - \sum_{j \in C^-} x_j = 1 - |C^-| \tag{7.3}$$

with binary variables x_j, $j \in C^+ \cup C^- \subseteq I$, $C^+ \cap C^- = \emptyset$. Then we partition the support of each constraint $i \in M$ into clique blocks

$$\operatorname{supp}(A_{i\cdot}) = R^i \cup C_1^i \cup \dots C_{q_i}^i$$

with cliques $C_k^i = (C_k^{i+}, C_k^{i-})$, $k = 1, \dots, q_i$, and a residual index set R^i, which means that the constraint $A_{i\cdot}x \leq b_i$ (or "$= b_i$") is expressed as

$$A_{iR^i}x_{R^i} + \sum_{k=1,\dots,q_i} A_{iC_k^i}x_{C_k^i} \leq b_i \text{ (or "}= b_i\text{")}.$$

The *clique block terms* $A_{iC_k^i}x_{C_k^i}$ are collected in a hash set with duplicates up to scaling being merged to obtain a list $T_1, \dots, T_Q$ of unique clique block terms $t_q x_{C_q}$, $q = 1, \dots, Q$, with binary variables $C_q \subseteq I$ and coefficients $t_q \in \mathbb{R}^{C_q}$. We generate a mapping $f : (i,k) \mapsto q$ for $i \in M$ and $k \in 1, \dots, q_i$ and store the corresponding scaling factors as s_k^i to obtain

$$A_{iC_k^i}x_{C_k^i} = s_k^i t_q x_{C_q}. \tag{7.4}$$

Now, for each unique clique block $q \in \{1, \dots, Q\}$ we add a *clique coupling variable* $y_q \in \mathbb{R}$ that is linked to the problem variables x via a *clique coupling constraint*

$$t_q x_{C_q} - y_q = 0 \tag{7.5}$$

and initial bounds

$$\inf\{t_q x_{C_q}\} \leq y_q \leq \sup\{t_q x_{C_q}\}.$$

Subsequently, the terms (7.4) with $q = f(i,k)$ are substituted by $s_k^i y_q$ in the constraint system. This yields an equivalent MIP formulation with additional variables and constraints that we use within probing. The reason for this transformation is that the clique coupling constraints (7.5) entail a stronger and more efficient domain propagation algorithm than regular linear constraints, because we can exploit the fact that the variables in C_q form a clique. Note that this resembles a special case of multi-row presolving, see Section 5.4.

To simplify the presentation we assume that $C_q^- = \emptyset$, i.e., all variables appear as a positive literal in the clique. Furthermore, we omit the indices and from now on consider a clique coupling constraint $tx - y = 0$ with a set of binary variables $x \in \{0,1\}^n$ and a clique coupling variable y with bounds $\ell_y \leq y \leq u_y$. If the variables x only form an inequality clique (7.2) we introduce a binary slack variable x_0 with $t_0 = 0$ to transform it into an an equality clique (7.3) so that we only need to consider this latter type.

Based on these assumptions, the propagation of clique coupling constraints exploits the following facts:

1. If $t_j < \ell_y$ or $t_j > u_y$ we can fix $x_j := 0$.
2. Valid bounds for y are $\min\{t_j | u_j = 1\} \leq y \leq \max\{t_j | u_j = 1\}$.
3. The two variables x_j that define the current bounds on y can used as the *watched literals* in a *two-watched-literals scheme*, see Moskewicz et al. [31]. This means that we only need to reconsider the clique coupling constraint for propagation if either the bounds of y have been tightened or any of the two watched literals has been fixed to zero.

Note that we do not need to propagate the clique condition (7.3), because this is implicitly done by propagating the other constraints in the system.

7.2.2 Lifting

As mentioned above, one of the products of probing is a set of implied bounds (7.1), which amongst others can later be used to derive implied bound cuts. For example, the implication $x_k = 1 \rightarrow x_j \geq \ell_j^1$ can be linearized to yield the implied bound cut $x_j - (\ell_j^1 - \ell_j)x_k \geq \ell_j$, provided that the global lower bound ℓ_j is finite.

A particularly interesting special case is an implied lower bound of a slack variable: Consider an inequality $ax \leq b$ and introduce an explicit slack variable $s \geq 0$ to obtain $ax + s = b$. Now, for an implication

$$x_k = 1 \rightarrow s \geq d > 0$$

we can derive the implied bound cut

$$s - dx_k \geq 0 \quad \Leftrightarrow \quad ax + dx_k \leq b.$$

Since the implied bound cut dominates the original constraint $ax \leq b$ (see Definition 2 on page 10), we can replace the original constraint by the cut. We say that we "lifted dx_k into the constraint".

Note that lifting is sequence dependent. Namely, lifting a variable into a constraint implicitly modifies the meaning of the corresponding slack variable. Hence, a second binary variable that implies a positive lower bound

$$x_p = 1 \rightarrow s \geq d' > 0$$

on the constraint's original slack variable can only be lifted into the constraint if $d' > d$ or x_k and x_p form a clique, $x_k + x_p \leq 1$. In the former case,

we obtain $ax + dx_k + (d' - d)x_p \leq b$, in the latter case we get the stronger lifted constraint $ax + dx_k + d'x_p \leq b$.

There are two different approaches for dealing with this sequence dependence. Until Gurobi version 6.0 we immediately lifted a variable into a constraint after probing on this variable detected the implication on the slack, implicitly updating the slack's definition for the subsequent probing and lifting. This means that the lifting sequence was predetermined by the probing candidate ordering. Moreover, this approach makes it virtually impossible to parallelize probing in a deterministic and still efficient way, because it forces the probing candidates to be evaluated in a sequential fashion.

Since Gurobi 6.5 we instead collect the lifting opportunities just like regular implied bounds, which can be done in parallel. After probing is completed, we apply the lifting in a sequential post-processing step. For each constraint $i \in M$ we consider the collected implications that provide positive lower bounds on the slack variable s_i. To simplify the presentation, we assume that those implications only originate from the $x_k = 1$ probing cases. Thus, for a given constraint $ax \leq b$ with explicit slack $ax + s = b$ we have a set of implications

$$x_k = 1 \rightarrow s > d_k$$

with binary variables x_k and $d_k > 0$ for all $k \in L \subseteq I$. To decide which variables we should lift into the constraint, we use the following heuristic, which assumes that we have constructed a clique table $\mathcal{C}$ with cliques $C \in \mathcal{C}$ as in (7.2) and (7.3) that are implicitly or explicitly given by the constraint system of the MIP. For each clique $C \in \mathcal{C}$ and each trivial clique $C = \{k\}$ for a binary variable x_k we calculate a score as the sum of the lifting coefficients

$$S_C = \sum_{k \in C} d_k,$$

with $d_k = 0$ for variables that do not entail an implication on the slack's lower bound. Given these scores, we just lift a clique C into the constraint that maximizes the score S_C. Note that by decreasing all d_k by $\max\{d_k | k \in C\}$ for the selected clique C we could iterate the lifting process if there still exists any $d_k > 0$, but this is not done in Gurobi.

Another important question regarding lifting is whether to apply it at all to a given MIP instance. We have seen multiple times in computational experiments that the impact of lifting can be pretty large in both directions: it can lead to speed-ups and degradations, depending on the problem

Table 29: Impact of disabling probing

		default	disable probing					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3155	568	579	425	550	1.07	1.08	—	—
$\geq$ 0 sec	2609	27	38	425	550	1.09	1.10	1574	1.15
$\geq$ 1 sec	1761	27	38	404	518	1.12	1.13	1265	1.18
$\geq$ 10 sec	1242	27	38	302	436	1.17	1.18	945	1.23
$\geq$ 100 sec	702	27	38	178	269	1.25	1.21	551	1.34
$\geq$ 1000 sec	230	27	38	71	100	1.33	1.17	201	1.38

instance. Even though lifting yields a provably stronger LP relaxation, it adds non-zeros to the coefficient matrix and thereby often degrades the performance and the numerical stability of the LP solves. Moreover, we conjecture that lifting may impact (positively and negatively) the separation of cutting planes. In Gurobi, we enable lifting only for models with at least a certain fraction of continuous variables or if the model seems to be challenging for other reasons. Moreover, in some cases we restrict lifting to only modify existing non-zero coefficients but not add new non-zeros.

The computational impact of Gurobi's probing procedure can be seen in Table 29. It yields an 18% reduction in branch-and-bound nodes and a 17% improvement in solve time in the "$\geq$ 10 sec" bracket. It finds reductions for a large fraction of the non-trivial models and helps to solve 11 more models within the time limit. Thus, probing is indeed an important component of MIP presolve.

7.3 Disconnected components

Consider the problem

$$\begin{array}{lrcl} \min & c_U^T x_U + & c_V^T x_V & \\ \text{s.t.} & A_{RU} x_U & \leq b_R & \\ & & A_{KV} x_V \leq b_K & \end{array} \tag{7.6}$$

with $\ell \leq x \leq u$, $x_j \in \mathbb{Z}$ for all $j \in I$, $U \cup V = N$, $U \cap V = \emptyset$, $R \cup K = M$, and $R \cap K = \emptyset$. Then, solving problem (7.6) is equivalent to solving the two individual MIPs

$$\min\{c_U^T x_U : A_{RU} x_U \leq b_R\} \quad \text{and} \quad \min\{c_V^T x_V : A_{KV} x_V \leq b_K\}$$

with corresponding bounds and integrality requirements. More generally, if a problem decomposes into multiple independent components, we can

solve each component separately and construct the final solution vector from the partial solutions corresponding to the individual components, see also Gamrath et al. [22].

One issue with the simple approach of solving the components successively is that we only obtain a feasible solution to the full problem once we have solved all components but the last and found a feasible solution to the last component. If solving any but the last component turns out to be challenging, then we cannot present any feasible solution to the user for a very long time, or we may even hit the time limit before finding a complete solution. Even though it does not increase the time to solve the MIP to optimality, this property is usually undesirable in practice.

Another challenge arises if non-zero relative or absolute gap limits are used. In the default settings, Gurobi would stop the search if the current incumbent solution's objective value is at most 0.01% away from the current global dual objective bound. Obviously, such limits should apply to the objective value of the full problem, but they cannot easily be broken down into gap limits for the individual components.

For these reasons, Gurobi employs two different strategies for dealing with models that feature disconnected components. First, during presolve, we try to solve each component except for the largest one to true optimality with zero gap. We impose pretty strict work limits for these sub-MIP calls in order to guard against the case that a single component is too hard to solve. Moreover, we disable presolving to avoid cases in which an "optimal" solution to the presolved component model turns out to violate the feasibility tolerances after uncrushing it to the master problem. For each component that we were able to solve in this manner, we fix the corresponding variables of the master problem to their optimal values.

A second decomposition algorithm is applied at the end of the root cutting plane loop if the problem has multiple independent components at this point in time. This can happen if we failed to solve any of the components in presolve, or if subsequent problem modifications in presolve or in the root cutting plane loop disaggregated the problem. This second algorithm solves the individual components in an interleaved fashion, each using a zero gap limit. It cycles through the component MIPs, processing 500 nodes for each of them per iteration. Whenever a component MIP improves its current incumbent solution, the corresponding piece of the full incumbent solution gets updated and, if feasible solutions for each component exist, a new full incumbent solution is reported to the user. The algorithm terminates if the overall gap limits are reached or if any of the component MIPs turns out to be infeasible. Moreover, if all components but one are solved to optimal-

Table 30: Impact of disabling disconnected component solving

		default	disable disconnected component solves					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3179	571	575	62	132	1.02	1.01	—	—
$\geq$ 0 sec	2612	9	13	62	132	1.03	1.03	398	1.22
$\geq$ 1 sec	1750	9	13	56	102	1.04	1.03	271	1.29
$\geq$ 10 sec	1192	9	13	51	84	1.06	1.04	213	1.36
$\geq$ 100 sec	633	9	13	32	52	1.08	1.07	123	1.52
$\geq$ 1000 sec	173	9	13	14	23	1.23	1.15	43	2.34

ity, we abort the decomposition algorithm and switch to the standard MIP solving procedure.

As can be seen in Table 30, disconnected component solving has similar properties as the symmetry handling methods of Sections 7.1 and 8.2: it applies only to a small subset of the models, but on those it provides significant speed-ups. About 20% of the models in our test set have disconnected components. These may be intrinsic to the original problem formulation, but more often they appear only after other presolve reductions have modified the problem or root node fixings from probing or reduced cost fixing discarded certain columns in the matrix. For those models, we observe a 36% performance improvement in the "$\geq$ 10 sec" bracket, which translates into an overall 6% speed-up. Note that in our experience, the majority of the disconnected components are tiny, consisting only of a handful of columns and rows. But in some cases the component sizes are non-negligible, and solving them as a separate MIP provides a significant performance boost.

7.4 Biconnected components

Consider the problem

$$\begin{aligned} \min \quad & c_U^T x_U + \quad c_V^T x_V + \quad c_k x_k \\ & A_{RU} x_U \qquad\qquad + A_{Rj} x_k \leq b_R \\ & \qquad\qquad A_{SV} x_V + A_{Sj} x_k \leq b_S \end{aligned}$$

with $\ell \leq x \leq u$, a single binary variable x_k, variables x_U and x_V of arbitrary type, $U \cup V = N \setminus \{k\}$, $U \cap V = \emptyset$, $R \cup S = M$, and $R \cap S = \emptyset$. We assume $|U| \geq |V|$. Now, if we set x_k to a specific value $x_k := \bar{x}_k \in \{0, 1\}$, then the problem splits up into two independent components

$$\min\{c_U^T x_U + c_k \bar{x}_k : A_{RU} x_U \leq b_R - A_{Rj} \bar{x}_k\}$$

Table 31: Impact of disabling biconnected component solving

bracket	models	default tilim	disable biconnected component solves tilim	faster	slower	time	nodes	affected models	affected time
all	3181	571	572	37	40	1.00	1.01	—	—
$\geq$ 0 sec	2611	6	7	37	40	1.00	1.01	166	1.01
$\geq$ 1 sec	1744	6	7	34	39	1.00	1.01	127	1.02
$\geq$ 10 sec	1180	6	7	26	29	1.00	1.01	97	1.02
$\geq$ 100 sec	621	6	7	24	18	1.00	1.01	71	0.98
$\geq$ 1000 sec	161	6	7	13	9	1.00	1.03	28	1.01

and

$$\min\{c_V^T x_V + c_k \bar{x}_k : A_{SV} x_V \leq b_S - A_{Sj} \bar{x}_k\}. \tag{7.7}$$

For such a *biconnected* problem we can apply a procedure that is similar to probing, see Section 7.2. Namely, we tentatively set the articulation variable x_k to 0 and 1, and solve the smaller sub-problem (7.7) for each of the two settings to obtain optimal solutions $\bar{x}_V^0$ and $\bar{x}_V^1$. As in the disconnected component presolving of Section 7.3 we have to use a zero gap limit for the component solves and apply a work limit to guard against excessive sub-MIP solve times.

If both sub-MIPs have been solved to optimality, we deduce implications of x_k to the variables in V, which may result either in globally valid fixings or substitutions:

1. If $x_k := 0$ renders (7.7) infeasible, we can fix $x_k := 1$, and similarly in the opposite case.
2. If $\bar{x}_j^0 = \bar{x}_j^1$ for $j \in V$, we can fix $x_j := \bar{x}_j^0$.
3. If $\bar{x}_j^0 \neq \bar{x}_j^1$ for $j \in V$, we can substitute $x_j := \bar{x}_j^0 + (\bar{x}_j^1 - \bar{x}_j^0) \cdot x_k$.

As can be seen in Table 31, biconnected components do not appear very often in the MIP models of our test set. Moreover, exploiting them does not produce big benefits, which suggests that in most cases the smaller component V is very small, and fixing or substituting all of its variables does not impact the subsequent solving process much.

7.5 Exploiting complementary slackness

By propagating the dual of the LP relaxation from a MIP in an appropriate way we can sometimes derive bounds on the dual variables and reduced costs. These bounds and reduced costs can be used to fix variables or detect implied equalities in the MIP.

Suppose $x^\star$ is a feasible solution to the primal LP

$$\min\{c^T x : Ax \geq b, x \geq 0, x \in \mathbb{R}^n\} \tag{7.8}$$

and $y^\star$ is a feasible solution to the dual LP

$$\max\{b^T y : A^T y \leq c, y \geq 0, y \in \mathbb{R}^m\}. \tag{7.9}$$

A necessary and sufficient condition for $x^\star$ and $y^\star$ to be optimal for (7.8) and (7.9), respectively, is complementary slackness. This means that for all $j \in N$ and $i \in M$ we have

(i) $x_j^\star > 0$ implies $c_j - (A_{\cdot j})^T y^\star = 0$, and

(ii) $y_i^\star > 0$ implies $A_{i\cdot} x^\star - b_i = 0$,

Let a MIP (2.1) with $\circ_i =$ "$\geq$" for all $i \in M$ and $\ell = 0$ be given, and assume that the LP relaxation of the MIP is bounded, i.e., it has a finite optimal solution. If we only consider the continuous variables $S = N \setminus I$ and apply bound strengthening on $(A_{\cdot S})^T y \leq c_S$ to get bounds on the dual variables $\bar{\ell} \leq y \leq \bar{u}$, we can transfer the implications from complementary slackness to MIPs even though they do not have in general a duality theory like linear programs. Let us first show an example and afterwards state and prove the theorem.

Example 10. Given the following problem:

$$\begin{aligned} \min\ 2.1x_1 + 5.9x_2 + 8.9x_3 - x_4 & \\ 1.05x_2 + 2.2x_3 - x_4 &\geq 1 \\ x_1 + 2.5x_2 + 2x_4 &\geq 3 \\ x_2 + x_3 - 0.1x_4 &\geq 0.5 \end{aligned}$$

with $x_1, x_2, x_3, x_4 \geq 0$ and $x_1, x_2 \in \mathbb{Z}$. By applying bound strengthening on

$$\begin{aligned} 2.2y_1 + y_3 &\leq 8.9 \\ -y_1 + 2y_2 - 0.1y_3 &\leq -1 \end{aligned}$$

and $y_1, y_2, y_3 \geq 0$ we obtain after a first step

$$\begin{aligned} 0.11 \leq\ & y_1 \leq 4.05 \\ 0 \leq\ & y_2 \leq 1.97 \\ 0 \leq\ & y_3 \leq 8.9 \end{aligned}$$

as bounds. Because $y_1 \geq \bar{\ell}_1 = 0.11$ the first constraint of the MIP can be turned into an equation. Since $c_1 - \sup\{(A_{\cdot 1})^T y\} = 2.1 - 1.97 = 0.13 > 0$ we can fix x_1 to its lower bound.

Theorem 2. *Let a MIP* (2.1) *with* $\circ_i =$ "$\geq$" *for all* $i \in M$ *and* $\ell = 0$ *be given. By considering only the continuous variables* $S = N \setminus I$ *and applying bound strengthening on* $(A_{\cdot S})^T y \leq c_S$, $y \geq 0$, *to get valid bounds* $\bar{\ell} \leq y \leq \bar{u}$, *the following holds:*

(i) If $\bar{\ell}_i > 0$ *for* $i \in M$, *we can turn constraint* i *into an equation* $A_{i\cdot}x = b_i$.

(ii) If $c_k > \sup\{(A_{\cdot k})^T y : \bar{\ell} \leq y \leq \bar{u}\}$ *for* $k \in N$, *we can fix* $x_k := 0$.

Proof. If the MIP is infeasible, then any restriction to the problem is valid. So, suppose the MIP has an optimal solution $x^\star$. We prove the two cases by contradiction.

For case (ii) assume that the condition $c_k > \sup\{(A_{\cdot k})^T y : \bar{\ell} \leq y \leq \bar{u}\}$ holds for a variable $k \in N$, but we have $x^\star_k > 0$. Then, consider the LP that results from fixing $x_j := x^\star_j$ for all $j \in I \setminus \{k\}$, which means to move the contributions of the fixed integer variables to the right hand side b. Since we only propagated dual constraints on the continuous variables and this propagation does not depend on b, the bounds $\bar{\ell} \leq y \leq \bar{u}$ are still valid. For this "fixed LP" complementary slackness has to be satisfied in order for a solution to be optimal. Thus, in any optimal solution $\tilde{x}$ to the fixed LP we have $\tilde{x}_k = 0$. But since 0 is integer, $\hat{x} := (x^\star_{I\setminus\{k\}}, \tilde{x})$ is also feasible for the MIP even if $k \in I$. The solution $\hat{x}$ has a smaller objective value than $x^\star$, which contradicts our assumption that $x^\star$ is optimal. Consequently, case (ii) must hold for any optimal solution of the MIP.

For case (i) assume that $\bar{\ell}_i > 0$ but $A_{i\cdot}x^\star > b_i$. Again, consider the LP resulting from fixing $x_j := x^\star_j$ for all $j \in I$. As before, the bounds $\bar{\ell} \leq y \leq \bar{u}$ on the dual variables are valid also for this fixed LP. But since

$$A_{i\cdot}x^\star > b_i \Leftrightarrow A_{iS}x^\star_S > b_i - A_{iI}x^\star_I$$

with $S := N \setminus I$ and $y_i \geq \bar{\ell}_i > 0$ contradict the complementary slackness conditions for the fixed LP, $x^\star_S$ cannot be optimal for the fixed LP and thus $x^\star$ cannot be optimal for the MIP. Consequently, case (i) must hold as well. □

About half of the models in our test set are affected by the complementary slackness reductions, see Table 32. On those models, the presolving procedure provides a 6% performance improvement in the "$\geq$ 10 sec" bracket, which is, in our view, surprisingly large for such a dual reduction.

Table 32: Impact of disabling complementary slackness reductions

		default	disable complementary slackness reductions					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3170	568	573	267	321	1.01	1.00	—	—
$\geq$ 0 sec	2618	21	26	267	321	1.02	1.00	1114	1.04
$\geq$ 1 sec	1763	21	26	259	312	1.02	1.01	867	1.05
$\geq$ 10 sec	1209	21	26	215	260	1.03	1.00	653	1.06
$\geq$ 100 sec	658	21	26	135	169	1.05	0.99	390	1.08
$\geq$ 1000 sec	208	21	26	63	67	1.14	1.05	153	1.19

7.6 Implied integer detection

A continuous variable that can only take integer values in any feasible solution is called an *implied integer variable*.

We use a primal and a dual method to detect implied integer variables. Primal detection means that there is an equation

$$A_{iS}x_S + a_{ij}x_j = c_i$$

with $S = N \setminus \{j\}$, integer or implied integer variables x_S, and a single continuous variable x_j, such that $c_i/a_{ij} \in \mathbb{Z}$ and $a_{is}/a_{ij} \in \mathbb{Z}$ for all $s \in S$. In this case, x_j will always take integral values in any feasible solution to the MIP and can thus be marked as implied integer.

The dual detection method can be applied on variables x_j that appear only in inequality constraints. If the MIP has an optimal solution, then it will always have an optimal solution $x^\star$ in which the continuous variables constitute a vertex solution of the "fixed LP", i.e., the LP that is given by fixing all integer variables x_k, $k \in I$, to their solution values $x_k^\star$. Consequently, in such a solution at least one of the inequalities in which x_j appears will be satisfied by equality or x_j will be at one of its bounds. Now, if $\ell_j, u_j \in \mathbb{Z}$ and the primal detection infers integrality of x_j for each inequality when treated as equation, then there exists at least one optimal solution to the MIP with $x_j \in \mathbb{Z}$. Hence, x_j can be marked as an implied integer variable.

Implied integer variables are exploited at various places in the solving process. First, cutting planes often get stronger coefficients for integer variables than for continuous variables or are not even applicable if a continuous variable is involved in the base inequality. Moreover, strong branching can be applied, the variables can be used as branching candidates, and domain propagation can exploit their integrality. On the other hand, in primal heuristics we do not need to force these variables to integer values because they will automatically become integer if we find an integer feasible solution

Table 33: Impact of disabling implied integer detection

		default	disable implied integer detection					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3172	570	591	436	532	1.06	1.08	—	—
$\geq$ 0 sec	2621	24	45	436	532	1.07	1.11	1703	1.11
$\geq$ 1 sec	1776	24	45	417	500	1.10	1.14	1367	1.13
$\geq$ 10 sec	1233	24	45	330	396	1.13	1.18	1001	1.17
$\geq$ 100 sec	698	24	45	188	252	1.25	1.34	591	1.30
$\geq$ 1000 sec	237	24	45	66	105	1.71	2.06	219	1.78

for all regular integer variables.

The computational impact of the implied integer detection is depicted in Table 33. The performance improvement is quite substantial with an 18% node count reduction and a 13% speed-up in the "$\geq$ 10 sec" bracket. Furthermore, exploiting implied integrality decreases the number of models that hit the time limit by 21.

8 Node Presolve

Node presolve is a light-weight process that is called for each sub-problem to be solved during the branch-and-bound tree search. In contrast to the root (or main) presolve that is applied before the actual solving process begins, node presolve in Gurobi does not modify the coefficient matrix A or the right hand sides b of the constraints. It only tries to tighten the local bounds of the variables in the sub-problem. Most notably, this is done by bound strengthening, but we also employ conflict propagation to exploit knowledge derived from infeasible sub-problems and orbital fixing to deal with symmetry. Since node presolve is applied for every single node in the search tree, it is very important to be efficient w.r.t. running time. Nevertheless, if it seems to be worthwhile, we even apply probing at the nodes, which can be very expensive.

Orbital fixing, bound strengthening, and conflict propagation are applied prior to solving the LP relaxation. The latter two are called in a loop, which is aborted after 5 iterations or if no further bounds could be strengthened. Probing is applied after the LP relaxation has been solved and repeated after resolving the LP if it was able to cut off the LP solution.

The total impact of node presolving is shown in Table 34. From all the different classes of presolving methods covered in this paper, node presolve provides the largest performance improvement, with bound strengthening

Table 34: Impact of disabling node presolve entirely

		default	disable node presolve					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3164	569	653	311	641	1.19	1.27	—	—
≥ 0 sec	2609	19	103	311	641	1.25	1.34	1621	1.41
≥ 1 sec	1780	19	103	303	616	1.36	1.48	1361	1.49
≥ 10 sec	1243	19	103	228	496	1.51	1.65	1024	1.65
≥ 100 sec	730	19	103	131	339	1.87	2.16	636	2.06
≥ 1000 sec	283	19	103	50	180	3.12	4.20	266	3.36

Table 35: Impact of disabling bound strengthening at local nodes

		default	disable bound strengthening at nodes					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3171	571	583	344	516	1.06	1.04	—	—
≥ 0 sec	2617	22	34	344	516	1.07	1.06	1606	1.11
≥ 1 sec	1771	22	34	336	499	1.10	1.08	1343	1.13
≥ 10 sec	1214	22	34	271	397	1.13	1.09	990	1.16
≥ 100 sec	687	22	34	163	262	1.19	1.17	591	1.23
≥ 1000 sec	219	22	34	59	101	1.30	1.31	202	1.33

and orbital fixing being the most important components. Unsurprisingly, almost all models that require branching are affected by node presolving: out of the 988 models without a path change, 963 are solved at the root node. In the "≥ 10 sec" bracket we observe a 51% speed-up, but most importantly, the number of tractable models increases by 84 due to node presolving.

8.1 Bound strengthening

Bound strengthening at the nodes is done in the same way as in the root presolve procedure, see Section 3.2. We assume that the local bounds at the parent of the current node have already been propagated and thus only need to reinvestigate the constraints in which the branching variable (or variables, if we branched on hyperplanes) appears. This may lead to tighter local bounds, which in turn triggers the reinvestigation of the constraints in which the variables with updated bounds appear.

Table 35 assesses the computational impact of node bound strengthening. With its 13% speed-up in the "≥ 10 sec" bracket it provides a larger speed-up than the root presolve bound strengthening, compare Table 4 on page 9. This is probably due to the fact that node bound strengthening catches many of the reductions that are missing in the root presolve if root bound

strengthening is disabled.

8.2 Orbital fixing

Many MIP instances contain symmetries, which essentially means that for any feasible solution there exist other solutions with the same objective value that are equivalent up to a permutation of the column and row indices of the matrix, see Definition 7 in Section 7.1. Those symmetries can significantly impair the solving process, because they lead to symmetric copies of sub-problems in the search tree that have to be solved, each of them containing equivalent solutions.

Different methods to deal with symmetry in mixed integer programming have been proposed in the literature. We refer to Margot [28] for a comprehensive survey. A recent computational study on the effectiveness of various methods was conducted by Pfetsch and Rehn [33].

One particular way of improving the solving process for models with symmetry is the so-called *orbital fixing*, which was introduced by Margot [27], see also Ostrowski [32]: Let P be the given MIP instance (2.1) and $\bar{l} \leq x \leq \bar{u}$ be the variable bounds at the local search tree node, consisting of branching decisions and implied bounds from node presolve. Orbital fixing now considers the variable orbits $\mathcal{O}$ for a symmetry group that is valid for $P \cap \{x \geq \bar{l}\}$. Namely, for an orbit $O \in \mathcal{O}$ we can tighten $x_j \leq \min\{\bar{u}_k : k \in O\}$ for all $j \in O$. Note that we could also tighten the lower bounds, but this requires more complicated bookkeeping of the origin of the current local bound changes and is not implemented in Gurobi.

Gurobi calculates a symmetry group G of the global problem P at the root node of the search tree using an algorithm based on Saucy [19, 20]. An important question for implementing orbital fixing is how to compute a symmetry group of the sub-problem $P \cap \{x \geq \bar{l}\}$ at a local node. Gurobi represents the symmetry group G as a set of *generators* $\pi : N \to N$. Usually, at a local node, we filter the generators by discarding those that are no longer valid for the local lower bounds $\bar{l}$. The remaining set of generators gives rise to a reduced subset of orbits that are locally valid. This approach is computationally cheap, but it is unable to exploit symmetry that is introduced into the problem by branching, and it often discards orbits that are at least partially still valid in the current node. Another approach is to recompute the local symmetry group from scratch at each node, which can be very time consuming. Nevertheless, it pays off for models with a certain symmetry structure and is enabled in the default settings of Gurobi if such a structure is detected.

Table 36: Impact of disabling orbital fixing

		default	disable orbital fixing					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3180	571	608	108	205	1.08	1.13	—	—
$\geq$ 0 sec	2614	10	47	108	205	1.10	1.16	510	1.61
$\geq$ 1 sec	1758	10	47	107	199	1.15	1.21	466	1.68
$\geq$ 10 sec	1215	10	47	95	169	1.20	1.27	392	1.77
$\geq$ 100 sec	666	10	47	61	128	1.36	1.48	259	2.22
$\geq$ 1000 sec	215	10	47	31	78	2.18	2.60	129	3.67

The performance impact of orbital fixing is very significant for models that contain symmetry, see Table 36. about 20% of the models are affected by orbital fixing, and on those we observe a 77% speed-up in the "$\geq$ 10 sec" bracket. Moreover, the number of time limit hits increases by 37 if orbital fixing is disabled.

8.3 Conflict propagation

Conflict analysis [1] can be used to derive conflict constraints from infeasible sub-problems of a MIP. These conflict constraints are propagated within node presolve to derive tighter local bounds for the variables.

The implementation in SCIP [2, 3] derives a set of conflicting bounds from the dual ray of the infeasible LP relaxation and applies backwards propagation to transform this initial conflict set into a more globally useful conflict constraint. Such conflict constraints are in general non-linear bound disjunction constraints, except for the case in which all variables are binary. They are propagated in SCIP using the two-watched-literals scheme, see Moskewicz et al. [31].

Gurobi's approach is slightly different. Instead of extracting one particular explanation for the infeasibility out of the dual ray, Gurobi just stores the Farkas infeasibility proof as a regular linear constraint. This is given by multiplying the dual ray y with the constraint matrix, i.e., $y^T Ax \leq y^T b$. This Farkas proof captures all possible explanations of the infeasibility that can be extracted from the dual ray. The downside is that it is usually much denser than a single explanation, and that the propagation of such a linear constraint is not as efficient as the one of bound disjunction constraints where the two-watched-literals scheme applies. Moreover, Gurobi 6.5 does not include a backwards propagation step that would be useful to further strengthen the conflict constraint.

Because propagating linear conflict constraints at each search tree node

Table 37: Impact of disabling conflict constraint propagation

		default	disable conflict propagation					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3172	571	577	189	240	1.02	1.01	—	—
$\geq$ 0 sec	2607	11	17	189	240	1.02	1.02	919	1.06
$\geq$ 1 sec	1746	11	17	188	237	1.03	1.02	798	1.07
$\geq$ 10 sec	1186	11	17	154	202	1.05	1.03	591	1.10
$\geq$ 100 sec	658	11	17	115	139	1.09	1.07	364	1.16
$\geq$ 1000 sec	183	11	17	34	52	1.14	1.14	118	1.22

can be expensive, Gurobi stores at most 100 of such constraints. When the limit of conflict constraints is reached and a new one is generated, we discard the constraint for which it was the longest time ago since it was discovered or used to derive a tighter bound in propagation.

Our intuition is that the conflict constraints in Gurobi are typically only useful in the neighborhood of the search tree node where they were discovered. For this reason, we avoid the overhead of sharing or synchronizing them between the threads of a parallel MIP search. Instead, each thread manages its own private pool of conflict constraints.

Conflict analysis as implemented in Gurobi 6.5 provides a 5% speed-up in the "$\geq$ 10 sec" bracket, as can be seen in Table 37. Even though this is non-negligible, it is a bit less than the 12% improvement that has been reported by Achterberg [2] for his implementation in SCIP. This discrepancy could arise from the less sophisticated implementation in Gurobi 6.5, which misses backwards propagation, but it certainly also comes from the different solver environment and the different test set.

8.4 Global probing

Similar to bound strengthening, probing is applied in the root presolve procedure, see Section 7.2, as well as in node presolve. Since this is a rather expensive algorithm, we impose a work limit on probing in the root presolve so that it is often aborted prematurely and some variables are not probed. Moreover, subsequent presolve reductions as well as tighter bounds found during the solving process (for example due to reduced cost fixing) may yield additional potential for probing to find more problem reductions. For this reason it makes sense to apply probing again, even if we have already probed some or all of the variables during root presolve.

In the context of node presolve, globally valid probing means to use the global bounds of the variables for the domain propagation within probing

Table 38: Impact of disabling globally valid probing at local nodes

		default	disable global probing at nodes					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3172	569	547	376	368	1.00	1.00	—	—
$\geq$ 0 sec	2634	36	14	376	368	1.00	1.00	1421	1.00
$\geq$ 1 sec	1773	36	14	366	361	1.00	1.00	1231	1.00
$\geq$ 10 sec	1229	36	14	296	296	1.01	0.99	928	1.01
$\geq$ 100 sec	675	36	14	184	181	1.01	1.02	542	1.02
$\geq$ 1000 sec	225	36	14	79	69	0.98	1.03	201	0.98

instead of the tighter local bounds at the current search tree node. During the solving process we apply globally valid probing only once for each binary variable, and only when it became fractional in a node's LP relaxation solution. In contrast to the probing procedure in root presolve, the probing in node presolve is only concerned with obtaining tighter bounds for the variables and with collecting clique and implication information. We are neither substituting variables nor are we lifting variables into constraints, because both would require complicated updates of the data structures stored during the solving process, in particular the warm start LP bases of the open search tree nodes.

In addition to the globally valid bound changes we derive statistics on the number of implied bounds from rounding the fractional LP solution value of a variable down or up, respectively. These statistics are used as secondary criterion for the branching variable selection, see also Achterberg and Berthold [4].

The globally valid probing at the nodes does not seem to provide any performance improvement, see Table 38. It is performance neutral in the number of nodes and the solve time, and it even increases the number of time limit hits. The main reason for this disappointing result is that for this benchmark test we only disabled the global probing at local nodes, while the call to the probing method at the root node during the cutting plane loop was kept. Thus, a good fraction of the integer variables that ever become fractional in LP solutions during the solving process is still subject to probing, and we collect implied bounds for them, which are important for cutting planes and branching. The results in Table 38 just indicates that the additional global probing on the variables that become fractional for the first time during the tree search phase is irrelevant for performance.

Table 39: Impact of disabling locally valid probing at local nodes

		default	disable local probing at nodes					affected	
bracket	models	tilim	tilim	faster	slower	time	nodes	models	time
all	3178	571	587	165	201	1.03	1.07	—	—
$\geq$ 0 sec	2611	9	25	165	201	1.04	1.07	734	1.12
$\geq$ 1 sec	1743	9	25	165	199	1.05	1.11	717	1.12
$\geq$ 10 sec	1189	9	25	149	189	1.07	1.15	607	1.15
$\geq$ 100 sec	649	9	25	117	137	1.12	1.23	402	1.21
$\geq$ 1000 sec	185	9	25	41	57	1.27	1.54	141	1.37

8.5 Local probing

Local probing means to exploit the local bounds of the current search tree node inside the probing procedure. Obviously, the resulting bound changes are then only valid within the sub-tree rooted at the current node. Nevertheless, local probing can sometimes be a very powerful tool to improve the running time for certain problem instances.

The probing procedure itself is almost identical to global probing, except that it uses local instead of global bounds. Another notable difference in Gurobi 6.5 is that local probing is also applied to general integer variables, where we consider the two cases $x_j \leq \lfloor \tilde{x}_j \rfloor$ and $x_j \geq \lfloor \tilde{x}_j \rfloor + 1$ with $\tilde{x}$ being the solution to the node's LP relaxation. But the main difference to global probing is that local probing can in principle be applied for every variable at every single search tree node, hence having the potential of being excessively expensive. The challenge is to identify the cases in which the benefit from tighter local bounds outweighs the costs of the probing procedure.

To drive this decision, Gurobi tracks two statistics during the solving process:

- the *work ratio* $w \in [0, 1]$, which is the work spent in local probing relative to the total work spent on solving the problem instance, and
- the *success ratio* $s \in [0, 1]$, which quantifies how often local probing was able to cut off the current LP relaxation solution relative to the total number of times probing was applied.

If $s \geq 2w$, we apply local probing on the binary and integer variables that are fractional in the current node's LP relaxation. If $w < 0.005$ or $s \geq 10w$ we apply local probing even on the variables with integral LP solution value.

Locally valid probing is an important component of Gurobi, as can be seen in Table 39. It provides a 7% speed-up in the "$\geq$ 10 sec" bracket and helps to solve 16 additional models within the time limit, but it has to be

noted that locally valid probing is very sensitive to the tuning of the dynamic work limits that are applied.

9 Conclusion

In this paper, we have given an overview on the presolve functionality in the Gurobi commercial mixed-integer programming code. This overview includes the description of more than thirty presolving techniques grouped into six classes.

Extensive computational tests over a set of about three thousand models show that presolving is extremely helpful in improving the performance of mixed integer programming solvers. By disregarding the relatively uninteresting easy models from consideration the average solve time is scaled up by a factor of nine when presolving is turned off completely. More importantly, presolving allows to solve more than 500 models from our test set that are otherwise intractable.

It is clear that we have left some unanswered questions, especially considerably more details concerning implementation of the algorithms. However, we hope that valuable ideas have been successfully demonstrated, and that perhaps some of the ideas can be used to motivate further research in this area.

Acknowledgments

This work has been supported by the Research Campus MODAL *Mathematical Optimization and Data Analysis Laboratories* funded by the Federal Ministry of Education and Research (BMBF Grant 05M14ZAM). All responsibility for the content of this publication is assumed by the authors.

References

[1] T. Achterberg. Conflict analysis in mixed integer programming. *Discrete Optimization*, 4(1):4–20, March 2007. Special issue: Mixed Integer Programming.

[2] T. Achterberg. *Constraint Integer Programming.* PhD thesis, Technische Universität Berlin, 2007.

[3] T. Achterberg. SCIP: Solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, 2009.

[4] T. Achterberg and T. Berthold. Hybrid branching. In W.-J. van Hoeve and J. Hooker, editors, *CPAIOR 2009*, Lecture Notes in Computer Science 5547, pages 309–311. Springer-Verlag, May 2009.

[5] 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.

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

[7] 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.

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

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

[10] E. M. L. Beale and J. A. Tomlin. Special facilities in a general mathematical programming system for nonconvex problems using ordered sets of variables. *Lawrence, J (ed.) Proceedings of the Fifth Intenational Conference on Operations Research*, pages 447–454, 1970.

[11] 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.

[12] 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.

[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.

[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.

[15] V. Chvátal. Edmonds polytopes and a hierarchy of combinatorial problems. *Discrete Mathematics*, 4(4):305 – 337, 1973.

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

[17] E. Danna, E. Rothberg, and C. Le Pape. Exploring relaxation induced neighborhoods to improve MIP solutions. *Mathematical Programming*, 102(1):71–90, 2005.

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

[19] P. T. Darga, M. H. Liffiton, K. A. Sakallah, and I. L. Markov. Exploiting structure in symmetry detection for cnf. In *Proceedings of the 41st annual Design Automation Conference*, pages 530–534. ACM, 2004.

[20] P. T. Darga, K. A. Sakallah, and I. L. Markov. Faster symmetry discovery using sparsity of symmetries. In *Proceedings of the 45th annual Design Automation Conference*, pages 149–154. ACM, 2008.

[21] A. Fügenschuh and A. Martin. Computational integer programming and cutting planes. In K. Aardal, G. L. Nemhauser, and R. Weismantel, editors, *Discrete Optimization*, volume 12 of *Handbooks in Operations Research and Management Science*, chapter 2, pages 69–122. Elsevier, 2005.

[22] G. Gamrath, T. Koch, A. Martin, M. Miltenberger, and D. Weninger. Progress in presolving for mixed integer programming. *Mathematical Programming Computation*, 7(4):367–398, 2015.

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

[24] 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.

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

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

[27] F. Margot. Exploiting orbits in symmetric ILP. *Mathematical Programming*, 98(1–3):3–21, 2003.

[28] F. Margot. Symmetry in integer linear programming. In *50 Years of Integer Programming 1958-2008*, pages 647–686. Springer, 2010.

[29] H. M. Markowitz. The elimination form of the inverse and its applications to linear programming. *Management Science*, 3:255–269, 1957.

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

[31] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Design Automation Conference*, July 2001.

[32] J. Ostrowski. *Symmetry in Mixed Integer Programming*. PhD thesis, Lehigh University, 2009.

[33] M. E. Pfetsch and T. Rehn. A computational comparison of symmetry handling methods for mixed integer programs. Technical report, Optimization Online, 2015.

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

[35] 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.

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

[37] J. A. Tomlin. Pivoting for size and sparsity in linear programming inversion routes. *IMA Journal of Applied Mathematics*, 10(3):289–295, 1972.