ARGONNE NATIONAL LABORATORY
9700 South Cass Avenue
Argonne, Illinois 60439

Presolving Mixed-Integer Linear Programs

Ashutosh Mahajan

Mathematics and Computer Science Division

Preprint ANL/MCS-P1752-0510

December 7, 2010

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

Contents

Presolving Mixed Integer Linear Programs [*]

Ashutosh Mahajan[†]

December 7, 2010

Abstract

We survey the techniques used for presolving Mixed-integer linear programs (MILPs). Presolving is an important component of all modern MILP solvers. It is used for simplifying a given instance, for detecting any obvious problems or errors, and for identifying structures and characteristics that are useful for solving an instance.

Keywords: Integer programming, presolving, preprocessing

AMS-MSC2000: 65K05, 90C11

1 Introduction

Presolving or preprocessing techniques constitute a broad class of methods used to transform a given instance of a Mixed-integer linear program (MILP) or to collect certain useful information about it. The main aim of presolving techniques is to simplify the given instance for solving by the more sophisticated and often time-consuming algorithms such as cutting-plane or branch-and-bound algorithms. Almost all modern solvers, both free (for instance, Achterberg, 2009; Forrest, 2010; Nemhauser et al., 1994; Ralphs et al., 2010) and commercial (for instance, FICO, 2009; Gurobi, 2009; IBM, 2009), deploy these in one way or the other. Bixby et al. (2000) claim that presolving alone speeds the solution times on benchmark instances by a factor of 2. Another important function of a presolver is to analyze the structure of the instance and collect information that will be useful when the problem is solved.

Presolving is also useful for the purpose of modeling as it can be used to check the instance for obvious errors. It can, for instance, inform the user whether certain constraints or bounds on variables make the instance infeasible or whether certain variables will have arbitrarily high values (unbounded) in a solution. Such information can help the user improve the description of the instance and correct errors. It can also sometimes tell the user whether the values used in the model may cause numerical problems for the subsequent algorithms used by the solver. As such,

[*]Preprint ANL/MCS-P1752-0510

[†]Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, mahajan@mcs.anl.gov.

some presolving is also performed by software such as AMPL (Fourer et al., 2003) that is used for creating MILP instances.

In this survey, we describe the various techniques used for basic and advanced presolving. Throughout the discussion, we assume that the original MILP is of the form

$$\begin{aligned} \min \sum_{j=1}^{n} c_j x_j & \\ \text{s.t.} \sum_{j=1}^{n} a_{ij} x_j \leq b_i, & \quad i = 1, \dots, m, \\ l_j \leq x_j \leq u_j, & \quad j = 1, 2, \dots, n, \\ x_j \in \mathbb{Z}, & \quad j \in I, \end{aligned} \tag{P}$$

where $m \in \mathbb{N}, n \in \mathbb{N}, c \in \mathbb{Q}^n, A \in \mathbb{Q}^{m \times n}, l \in \mathbb{Q}^n, u \in \mathbb{Q}^n, I \subseteq \{1, 2, \dots, n\}$ are given as inputs. We will explicitly clarify whenever a technique applies to any equality constraints ($\sum_{j=1}^{n} a_{ij} = b_i$) and not to the inequalities in (P). Some techniques for presolving MILPs are the same as those used for presolving linear programs (LPs). We also describe these techniques here because they are a starting point for more advanced techniques for presolving MILPs. Most of the presolving techniques presented here are derived from the work of Andersen and Andersen (1995), Brearley and Mitra (1975), Guignard and Spielberg (1981), and Savelsbergh (1994). Details of implementing the techniques in an MILP solver can be found in the work of Suhl and Szymanski (1994), and Achterberg (2007).

We start by describing basic presolving techniques in Section 2 and then describe more advanced techniques in Section 3. We also discuss methods for detecting structure (Section 4), and for postprocessing (Section 5).

2 Basic Presolving

Basic presolving methods are simple procedures that are directly applied to simplify (P). These methods may be called many times as subroutines in more advanced methods. Hence, implementing them efficiently is important.

2.1 Simple Rearrangements and Substitutions

Removing constraints and variables that are not necessary in the instance can reduce the amount of memory required by a computer to solve the instance and also speed the calculations. Rearranging the order of variables and constraints in the instance can speed several routines in the subsequent preprocessing and solve methods.

Removing constraints If we have $a_{ij} = 0\ \forall j \in \{1, 2, \dots, n\}$ for some $i \in \{1, 2, \dots, m\}$ and if $b_i \geq 0$, then the $i-$th constraint can be deleted. If for the same constraint $b_i < 0$, then the instance is infeasible because of this constraint. If we have a singleton row, namely, $a_{ik} \neq 0, a_{ij} = 0\ \forall j \neq k, j \in \{1, 2, \ldots, n\}$, for some $i \in \{1, 2, \ldots, m\}$ and some $k \in \{1, 2, \ldots, n\}$, then the ith constraint can be removed, and the bounds of variable x_k can be tightened if necessary.

Removing variables If we have $a_{ij} = 0\ \forall i \in \{1, 2, \ldots, m\}$ for some $j \in \{1, 2, \ldots, n\}$, then we can fix the variable x_j to l_j if $c_j \geq 0$ or to u_j if $c_j < 0$. After fixing the variable, we may have an additional constant term in the objective function. If we have a singleton column x_j, and if $j \notin I$, $a_{ij} > 0$, $l_j = -\infty$, $u_j = \infty$, $c_j \leq 0$ then we can drop the constraint i and substitute $x_j = \frac{b_i - \sum_{k:k\neq j} a_{ik}x_k}{a_{ij}}$ in the objective function. After the substitution, we have one less variable and one less constraint. The number of nonzeros in A is also reduced. The number of nonzeros in the objective function, however, may increase.

Rearrangements The variables and constraints of the instance can be rearranged depending on how the methods of solving are implemented. For example, it may be beneficial to have all binary variables stored in the end of the array of variables so that deleting them (in case we fix them) from the sparse representation of the matrix A is faster. Similarly, the constraints can be rearranged so that "constraints of interest" appear together in the arrays in which they are stored. If it is known that an interior-point method will initially be used for solving the relaxation of (P), then rows and columns may be rearranged so as to reduce the "fill in". Rothberg and Hendrickson (1998) study the effects of various such permutation techniques on interior-point methods.

2.2 Granularity

Given an instance of form (P) and a vector $\mathbf{a} \in \mathbb{Q}^n$, we define granularity of $\mathbf{a}$ as

$$g(\mathbf{a}) = \min_{x^1, x^2 \in \mathcal{P}} \left\{ \left| \sum_{j=1}^{n} a_{ij}x_j^1 - \sum_{j=1}^{n} a_{ij}x_j^2 \right| : \sum_{j=1}^{n} a_{ij}x_j^1 \neq \sum_{j=1}^{n} a_{ij}x_j^2 \right\},$$

where $\mathcal{P}$ is the set of all points satisfying the constraints of (P). Granularity refers to the minimum difference (greater than zero) between the values of activity that a constraint or objective function may have at two different feasible points. For a constraint that has only integer-constrained variables, then granularity is at least the greatest common divisor (GCD) of all the coefficients. If any of the coefficients in such a constraint are not integers, we can find a multiplier 10^K, where $K \in \mathbb{Z}^+$, to make all coefficients integers. The GCD can be divided by 10^K to get granularity. Hoffman and Padberg (1991) call this procedure Eucledean reduction. The problem can be declared infeasible if there is an equality constraint whose right-hand side is not an integer multiple of the of the GCD. For an inequality constraint $i \in \{1, 2, \ldots, m\}$, if b_i is not an integer multiple of the granularity $g(\mathbf{a_i})$, then b_i can be reduced to the nearest multiple of g_i less than b_i. So the constraint

$$3x_1 + 6x_2 - 3x_3 \leq 5.3$$

can be tightened to

$$3x_1 + 6x_2 - 3x_3 \leq 3,$$

if $x_1, x_2, x_3 \in \mathbb{Z}$. The bound tightening in this case is a special case of a Chvátal-Gomory inequality introduced by Gomory (1958). Similarly, one can find the granularity of the objective vector $\mathbf{c}$ when only integer-constrained variables have nonzero coefficients in the objective function. The difference between upper bound obtained from a feasible solution and the optimal solution value will always be at least $g(\mathbf{c})$. If the difference between an upper bound and a lower bound falls below the granularity, we can stop the branch-and-bound or cutting-plane algorithm and claim that the optimal solution has been found.

2.3 Constraint and Variable Duplication

Sometimes an instance may have duplicate constraints that are exact copies of each other. At other times, we may have constraints that are identical except for a scalar multiple. The obvious benefit of detecting such constraints is that we can reduce the memory needed for storing and solving the instance. It also helps improve the numerical stability of solution of the associated LP relaxations. Bixby and Wagner (1987), and Tomlin and Welch (1986) describe efficient ways to detect from a column-ordered format of A, whether two constraints are identical except for scalar multiples. When constraints with duplicate entries are found, we can, based on the values of the right hand sides and the scalar multiples, declare the problem infeasible or delete one of the constraints or combine the two constraints into one.

Similarly, we can detect whether two columns are alike except for a scalar multiple. However, the possible integrality constraint on one or both of these variables can make any substitution complicated. If $A_j = \alpha A_i$ and if $c_j = \alpha c_i$ for some $i, j \in \{1, 2, \ldots, n\} \setminus I, \alpha \in \mathbb{Q}$, then we can replace the two columns A_i, A_j by a single column A_k that is identical to A_i. The lower and upper bounds of the new variable x_k are $l_k = l_i + \alpha l_j, u_k = u_i + \alpha u_j$ if $\alpha > 0$ and $l_k = l_i + \alpha u_j, u_k = u_i + \alpha l_j$ otherwise. If one or both variables are integer constrained, we can use the substitution $x_k = x_i + \alpha x_j$ if we can find feasible values of x_i, x_j for each $x_k \in [l_k, u_k]$. We can also add a constraint $x_k = x_i + \alpha x_j$ to avoid such complications.

2.4 Constraint Domination

One way to identify a redundant constraint is to check whether it can be termwise dominated by another constraint. If $b_k \geq b_i$ for $i, k \in \{1, 2, \ldots, m\}$, each variable in constraints i, k is either non-negative or non-positive, $a_{kj} \leq a_{ij}$ for j such that x_j is non-negative and $a_{kj} \geq a_{ij}$ for j such that x_j is non-positive, then constraint k can be deleted because it is redundant in the presence of constraint i and the bounds on variables.

2.5 Bound Improvement

Bound improvement is one of the most important steps of presolving. If we can improve the bounds on variables or reduce b_i, then we can tighten the LP relaxation of (P). The tightest bounds on a constraint can be obtained (Savelsbergh, 1994) by solving the following problems,

$$\begin{aligned} \mathcal{L}_i = \min \sum_{j=1}^{n} & a_i^T x \\ \text{s.t. } & A^i x \le b^i, \\ & l_j \le x_j \le u_j, \quad j = 1, 2, \ldots, n \\ & x_j \in \mathbb{Z}, \quad j \in I, \end{aligned} \tag{2.1}$$

and

$$\begin{aligned} \mathcal{U}_i = \max \sum_{j=1}^{n} & a_i^T x \\ \text{s.t. } & A^i x \le b^i, \\ & l_j \le x_j \le u_j, \quad j = 1, 2, \ldots, n \\ & x_j \in \mathbb{Z}, \quad j \in I, \end{aligned} \tag{2.2}$$

where the set of constraints $A^i x \le b^i$ is obtained by deleting the ith constraint $\sum_{j=1}^{n} a_{ij} x_j \le b_i$ from (P). However, solving the bound improvement problems (2.1) or (2.2) can be as difficult as solving (P). Thus, there is a trade-off between the quality of a good bound and the time spent in finding it. The most common method of tightening the bound is to calculate bounds on $\mathcal{L}_i$ and $\mathcal{U}_i$:

$$\begin{aligned} \hat{\mathcal{L}}_i &= \sum_{j: a_{ij} > 0} a_{ij} l_j + \sum_{j: a_{ij} < 0} a_{ij} u_j, \\ \hat{\mathcal{U}}_i &= \sum_{j: a_{ij} > 0} a_{ij} u_j + \sum_{j: a_{ij} < 0} a_{ij} l_j. \end{aligned} \tag{2.3}$$

Both $\hat{\mathcal{L}}_i, \hat{\mathcal{U}}_i$ can be calculated in $\mathcal{O}(nz)$ steps, by visiting each nonzero coefficient of the A matrix.

Clearly $\hat{\mathcal{L}}_i \le \mathcal{L}_i \le \mathcal{U}_i \le \hat{\mathcal{U}}_i$. If $b_i < \mathcal{L}_i$, then the instance is infeasible. If $b_i = \mathcal{L}_i$, then all variables appearing in the constraint are forced to a bound: $x_j = l_j$ if $a_{ij} > 0$ and $x_j = u_j$ if $a_{ij} < 0$. This constraint can then be deleted. If $b_i \ge \mathcal{U}_i$, then the constraint can be deleted because it is redundant.

Similarly, we can improve the bounds on variables. The tightest lower and upper bounds for the variable x_k can be obtained by solving the following:

$$\begin{aligned} L_k = \min\ & x_k \\ \text{s.t. } & Ax \le b, \\ & l_j \le x_j \le u_j, \quad j = 1, 2, \ldots, n \\ & x_j \in \mathbb{Z}, \quad j \in I, \end{aligned} \tag{2.4}$$

and

$$\begin{aligned} U_k = \max\, & x_k \\ \text{s.t. } & Ax \le b, \\ & l_j \le x_j \le u_j, \quad j = 1, 2, \ldots, n \\ & x_j \in \mathbb{Z}, \qquad j \in I. \end{aligned} \tag{2.5}$$

Again, solving the problems (2.4) and (2.5) can be as difficult as the original problem (P). A similar trade-off then occurs on the quality of the bound and the time spent in obtaining it. The bounds $\mathcal{L}_i$ and $\mathcal{U}_i$ on constraints can also be used to improve the bounds on variables. If $a_{ij} > 0$, then let

$$\hat{U}_{ik} = \frac{b_i}{a_{ik}} - \frac{\mathcal{L}_i - a_{ik} l_k}{a_{ik}}.$$

Clearly, $\hat{U}_{ik}$ ($\ge U_k$) is an upper bound on the feasible values of x_k. If $U_k < u_k$ (or $\hat{U}_k < u_k$), then update the bound on variable x_k to U_k. If $U_k < l_k$, then the problem is infeasible. If $U_k = l_k$, then the variable x_k can be fixed to the value l_k. Similarly, if $a_{ik} < 0$, then let

$$\hat{L}_{ik} = \frac{b_i}{a_{ik}} - \frac{\mathcal{U}_i - a_{ik} l_k}{a_{ik}}.$$

Then $\hat{L}_{ik}(\le L_k)$ is also a lower bound on feasible values of x_k. If $L_k < l_k$, then update the lower bound on variable x_k to L_{ik}. If $L_{ik} > u_k$, then the problem is infeasible. If $L_{ik} = u_k$, then the variable x_k can be fixed to the value u_k. Once $\mathcal{L}_i$ and $\mathcal{U}_i$ (or their bounds $\hat{\mathcal{L}}_i, \hat{\mathcal{U}}_i$) are known, then the bounds $\hat{L}, \hat{U}$ can be calculated in $\mathcal{O}(nz)$ steps.

If the bounds are tightened using the complete models (2.1) and (2.2), then constraint domination is just a special case of bound tightening. This is no longer true, however, when the bounds are estimated from each row. In the latter case, we need only $\mathcal{O}(nz)$ calculations, where nz denotes the number of nonzeros in the A matrix.

If the bounds of a variable x_j are implied by a constraint, then the variable can be declared "free", and we can try substituting the variable out as described in Section 2.1.

2.6 Dual/Reduced-Cost Improvement

If, for a variable $x_j, j \in \{1, 2, \ldots, n\}$, $a_{ij} \ge 0\ \forall i \in \{1, 2, \ldots, m\}$, and $c_j \ge 0$, then the variable x_j can be fixed to its lower bound l_j. Similarly, if $a_{ij} \le 0\ \forall i \in \{1, 2, \ldots, m\}$ and $c_j \le 0$, then the variable x_j can be fixed to its upper bound u_j.

Sometimes, an upper bound on the optimal value of solution of (P) is known (it may be provided by the user or may be obtained from some heuristics) before presolving. In such a case, we have an additional inequality $\sum_{j=1}^{n} c_j x_j \le z_u$, where z_u is the best-known upper bound. The basic presolving techniques applicable to other constraints can be applied to this constraint as well. Valid inequalities of the form $\sum_{j=1}^{n} \hat{c}_j x_j \ge z_l$, where $z_l \in \mathbb{Q}$, can also be obtained by linear combinations of the objective function with other constraints. Such inequalities are automatically obtained from the reduced costs calculated by the simplex method. When both z_l and z_u are available as described above, we can do bound tightening on variables to obtain tighter bounds. This is equivalent to the reduced-cost fixing method described by Balas and Martin (1980).

3 Advanced Presolving

Advanced presolving procedures try to modify the problem and then call basic preprocessing procedures repeatedly to derive more information about the effects of such changes. Such procedures are also called probing. Unlike basic preprocessing techniques, probing may be expensive because one can probe on many possible changes or on combinations of several changes. The simplest changes are the changes of bounds on binary variables. We can fix such a variable to zero or 1 and then probe on the reduced problem.

3.1 Fixing Variables

When probing on a binary variable, say x_i, we first fix x_i to zero by changing its upper bound. We then perform basic preprocessing on the modified problem. If the modified problem can be shown to be infeasible, then x_i can be fixed to one in the original problem. Similarly, if the problem becomes infeasible on setting x_i to one, we can fix x_i to zero. If the problem is infeasible on setting x_i to one and zero both, we can declare that the problem is infeasible.

3.2 Improving Coefficients

If a constraint i, $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$ is redundant when a binary variable x_k is set to zero, then we can improve the coefficient a_{ik} of x_k in this constraint using the approach of Crowder et al. (1983). We first observe that changing a_{ik} does not make any difference when $x_k = 0$. When x_k is set to 1, the original constraint is equivalent to $\sum_{j=1}^{n} a_{ij}x_j - \delta_0 x_k \leq b_i - \delta_0$, for any $\delta_0 \in \mathbb{R}$. We would like to have as high a value of δ_0 as possible, while still keeping the constraint redundant at $x_k = 0$. This value of δ_0 can be obtained by solving the following problem:

$$\delta_0 = b_i - \max_{Q} \sum_{j=1}^{n} a_{ij}x_j,$$

where Q is the set of values of x such that

$$\begin{aligned} A^i x &\leq b^i, \\ x_k &= 0, \\ l_j \leq x_j &\leq u_j, \quad j = 1, 2, \ldots, n \\ x_j &\in \mathbb{Z}, \quad\ \ j \in I. \end{aligned}$$

Again, the problem of finding the optimal value of δ_0 can be as hard as solving the original problem (P). A more easily computable bound,

$$\hat{\delta}_0 = b_i - \sum_{j:j\neq k, a_{ij}>0} a_{ij}u_j - \sum_{j:j\neq k, a_{ij}<0} a_{ij}l_j,$$

can be used instead.

Similarly, if the constraint i, $\sum_{j=1}^{n} a_{ij}x_j \leq b_i$ is redundant when a binary variable x_k is set to 1, we can replace the constraint by $\sum_{j=1}^{n} a_{ij}x_j + \delta_1 x_k \leq b_i$, with

$$\delta_1 = b_i - \max_Q \sum_{j=1}^{n} a_{ij}x_j,$$

where Q is the set of values of x such that

$$\begin{aligned} A^i x &\leq b^i, \\ x_k &= 1, \\ l_j \leq x_j &\leq u_j, \quad j = 1, 2, \ldots, n \\ x_j &\in \mathbb{Z}, \quad\ \ j \in I. \end{aligned}$$

As above, we can use a bound that is computationally easier to calculate,

$$\hat{\delta}_1 = b_i - a_{ik} - \sum_{j:j\neq k, a_{ij}>0} a_{ij}u_j - \sum_{j:j\neq k, a_{ij}<0} a_{ij}l_j.$$

3.3 Deriving Implications

Implications are relations that the variable values of any optimal feasible solution must satisfy. One such implication can be found by fixing two binary variables, say, x_i, x_j, where $i, j \in I$ to zero. If the modified problem is shown to be infeasible, then we know that all solutions must satisfy the inequality $x_i + x_j \geq 1$. An equivalent way of looking at this implication is that, if we fix x_i to zero, then x_j is constrained to be 1. A more general case is when fixing x_i to zero implies $x_j = v_j, k \in \{1, 2, \ldots, n\}$, where $v_j \in \mathbb{R}$. Then the following two inequalities are valid for (P):

$$\begin{aligned} x_j &\leq v_j + (u_j - v_j)x_i \\ x_j &\geq v_j - (v_j - l_j)x_i. \end{aligned}$$

Similarly, if by fixing x_i to 1, we can fix x_j to some $v_j \in \mathbb{R}$, then we can write

$$x_i = 1 \Rightarrow x_j = v_j \iff \left\{ \begin{array}{lll} x_j & \leq & u_j - (u_j - v_j)x_i, \\ x_j & \geq & l_j + (v_j - l_j)x_i. \end{array} \right.$$

When x_j is also binary, we can use the above procedure, to obtain the following implications:

$$\begin{aligned} x_i = 0 \Rightarrow x_j = 0 &\iff x_i - x_j \geq 0, \\ x_i = 0 \Rightarrow x_j = 1 &\iff x_i + x_j \geq 1, \\ x_i = 1 \Rightarrow x_j = 0 &\iff x_i + x_j \leq 1, \\ x_i = 1 \Rightarrow x_j = 1 &\iff x_i - x_j \leq 0. \end{aligned}$$

Deriving such implications is useful not only for preprocessing but also for other techniques such as primal heuristics, branching, and generating valid inequalities, that are subsequently used in solving (P). The number of such implications can, however, be very large for certain types of problems. For example, if we have a set partitioning constraint of the form

$$\sum_{i \in S} x_i = 1,$$

where x_i is binary for $i \in S$, then the number of valid implications with only two variables would be $\mathcal{O}(|S|^2)$. Thus it is important to implement methods to store and retrieve them quickly. Several implications can also be combined to fix variables, to delete constraints, and to derive new implications, e.g., if we have implications $x_1 = 0 \Rightarrow x_2 = 0$, and $x_1 = 1 \Rightarrow x_2 = 0$, then we can fix x_2 at 0. Similarly, if a constraint is redundant when $x_1 = 0$ and also when $x_1 = 1$, then we can delete that constraint. The algorithms for deriving such new information from an existing list of implications can be studied with the help of what is called a conflict graph.

A conflict graph is a graph that shows what values of binary variables are not feasible (or optimal) for (P). The graph has two sets $X, \bar{X}$ of k vertices each, where k is the number of binary variables. For each binary variable x_i in (P), we have a vertex each in X (denoted i), and in $\bar{X}$ (denoted $\bar{i}$). A vertex $i \in X$ corresponding to variable x_i is associated with the implication "$x_i = 1$". Similarly, a vertex $i \in \bar{X}$ is associated with "$x_i = 0$". An edge between two vertices denotes a conflict, i.e., any optimal solution of (P) can not have values associated with the two vertices of any edge in the conflict graph. Figure 1 shows a conflict graph in three variables. Three implications that create the edges are $x_1 = 1 \Rightarrow x_2 = 1$, $x_3 = 1 \Rightarrow x_2 = 1$, and $x_3 = 1 \Rightarrow x_1 = 0$. The dark edges in the graph denote that either $x_i = 1$ or $x_i = 0$, and one of them necessarily holds. New implications can now be derived by studying this graph, e.g., if there is a vertex j with neighbors $i, \bar{i}$, then x_j can be fixed to 0. We refer the readers to the work of Savelsbergh (1994), and Atamtürk and Savelsbergh (2000) for algorithms that can be used to derive such implications. The latter also describe data structures for storing them efficiently. We now mention two more ways in which implications are useful for improving a formulation.

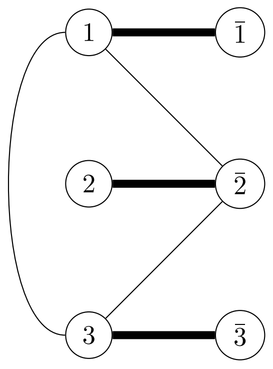

Figure 1: An example of a conflict graph

Automatic disaggregation: Savelsbergh (1994) shows that, if we add the inequalities derived from implications as above, then we automatically disaggregate each constraint of the form

$$\sum_{i \in S} x_i \le U x_k,$$

where $S \subseteq \{1, 2, \ldots, n\}, k \in \{1, 2, \ldots, n\}$, and $U \le \sum_{i \in S} u_i$, into the inequalities

$$x_i \le u_i x_k, i \in S.$$

Elimination of variables: If fixing a binary variable x_k to zero implies $x_i = r_i$ for some $i \in \{1, 2, \ldots, n\}$ and fixing x_k to one implies $x_i = s_i$ for some $r_i, s_i \in \mathbb{R}$, then we can substitute

$$x_i = r_i + (s_i - r_i)x_k.$$

The variable x_i can now be eliminated from the problem without increasing the number of nonzeros in the A matrix.

3.4 Probing on Constraints

Consider a clique inequality of the form

$$\sum_{i \in S^+} x_i - \sum_{i \in S^-} x_i \leq 1 - \left|S^-\right|,$$

where $S^+, S^- \subseteq \{1, 2, \ldots, n\}$, and all x_i are binary for $i \in S^+ \cup S^-$. The tuple $\{x_i\}, i \in S^+ \cup S^-$ can assume only $|S^+ \cup S^-| + 1$ values for any feasible point. Out of these, $|S^+ \cup S^-|$ values correspond to exactly one of $x_i, i \in S^+$ being 1 or exactly one of $x_i, i \in S^-$ being zero. All these cases are therefore considered automatically when probing on these variables. The only remaining case that is feasible for this constraint is when $x_i = 0, \forall i \in S^+$ and $x_i = 1, \forall i \in S^-$. If the problem is infeasible for this case, then the inequality can be tightened to an equality. If some other inequality is redundant, then the coefficients can be tightened following the procedure in Section 3.2.

4 Identifying Structure

Many techniques for generating valid inequalities, branching and heuristics are applicable only for particular types of constraints and variables. Yet others may be greatly improved if some special structures are known to exist. Since such techniques are used repeatedly many times in the branch-and-bound or cutting-plane methods, it is important to detect upfront any useful structures in the given MILP. Some of these structures are listed below.

- Special ordered sets of type 1 (SOS1) are of the form

$$\begin{aligned} \sum_{i \in S} x_i &= 1, \\ x_k &= \sum_{i \in S} a_i x_i, \\ x_i \in \{0, 1\}, i \in S &\subseteq \{1, 2, \ldots, n\}. \end{aligned}$$

  These can be used for special branching rules (Beale and Tomlin, 1970).

- Set partitioning ($\sum_{i \in S} x_i = \alpha, x_i \in \{0, 1\}, i \in S$), set covering ($\sum_{i \in S} \geq \alpha$), and set packing inequalities ($\sum_{i \in S} x_i \leq \alpha$) can be used for primal heuristics (Atamtürk et al., 1995) and for generating valid inequalities (Balas and Padberg, 1976).

- Knapsack inequalities including 0-1 knapsack, mixed 0-1 knapsack, integer knapsack, and mixed-integer knapsack inequalities (Atamtürk, 2005).
- Generalized upper bound constraints can be used to generate valid inequalities (Wolsey, 1990).
- Variable lower bound and upper bound constraints can also be used to generate valid inequalities (van Roy and Wolsey, 1987).

5 Postsolve

After an instance has been treated by a presolver and then solved, the user may ask for the solution and other problem characteristics. The presolver can change some instance substantially and the solution of the transformed problem may be quite different from that of the original problem. Hence the presolve routines are designed to save the changes that were made to the original instance. Andersen and Andersen (1995) suggest keeping a "stack" of all the transformations that were made by the presolver and reverting them in a last-in-first-out sequence. Sometimes, the solution obtained by reverting the changes may not remain feasible for the specified tolerance values even though the presolved solution was feasible. In such cases, the user must either decrease the tolerance limit or disable presolve.

6 Concluding Remarks

In this paper, we have surveyed many techniques for presolving. Some of these techniques are simple, like removing empty constraints, and empty columns, identifying special structures and rearranging constraints and variables. Some others are more powerful, like detecting duplicate rows and columns. They require specialized algorithms to avoid spending excessive amount of time. Then there are some methods like bound tightening, detecting infeasibility and identifying redundant constraints which in theory can be as difficult as solving the original problem itself. Heuristic methods are used to trade off the time spent in these problems against the benefits of solving them. All of these methods usually consider only bounds on variables and a single constraint. Advanced presolving methods can help simplifying the problems a lot but each probe requires repeating basic presolving methods. Further, one needs to probe over many candidate variables. In general, it is difficult to predict the optimal effort that should be put into each method of presolving. Most implementations of a presolver rely on several rules of thumb and the past experiences with these methods.

A presolver is an important component of modern MILP solvers. It can substantially reduce the time taken to solve MILPs. It is also useful for conveying to the user obvious problems with the formulation of the instance. However, the percentage of time spent by a solver in presolve routines is usually much smaller than the percentage of lines of code required to write a presolver.

Even though the techniques of presolving are simple, implementing them effectively for all types of formulations and inputs can be a challenging activity.

Acknowledgments

The author gratefully acknowledges the anonymous referee for suggestions and recommendations that helped improve this survey, and also Gail Pieper for proofreading an earlier draft. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. This work was also supported by the U.S. Department of Energy through grant DE-FG02-05ER25694.

References

Achterberg, T. (2007). *Constraint integer programming*. PhD thesis, Technical University of Berlin.

Achterberg, T. (2009). SCIP: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–49.

Andersen, E. and Andersen, K. (1995). Presolving in linear programming. *Mathematical Programming*, 71:221–245.

Atamtürk, A. (2005). Cover and pack inequalities for (mixed) integer programming. *Annals of Operations Research*, 139(1):21–38.

Atamtürk, A., Nemhauser, G. L., and Savelsbergh, M. W. P. (1995). A combined Lagrangian, linear programming and implication heuristic for large-scale set partitioning problems. *Journal of Heuristics*, 1:247–259.

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

Balas, E. and Martin, C. (1980). Pivot and complement - a heuristic for 0-1 programming. *Management Science*, 26(1):86–96.

Balas, E. and Padberg, M. W. (1976). Set partitioning: A survey. *SIAM Review*, 18(3):710–760.

Beale, E. M. L. and Tomlin, J. A. (1970). Special facilities in general mathematical programming system for non-convex problems using ordered sets of variables. In Lawrence, J., editor, *Proceedings of the Fifth International Conference on Operations Research*, pages 447–454.

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

Bixby, R. E. and Wagner, D. (1987). A note on detecting simple redundancies in linear systems. *Operations Research Letters*, 6(1):15–17.

Brearley, A. and Mitra, G. (1975). Analysis of mathematical programming problems prior to applying the Simplex algorithm. *Mathematical Programming*, 8:54–83.

Crowder, H., Johnson, E. L., and Padberg, M. (1983). Solving large scale zero-one linear programming problems. *Operations Research*, 31(4):803–834.

FICO (2009). *Xpress-Mosel user guide*. Available online at http://optimization.fico.com.

Forrest, J. (2010). *COIN-OR Branch and Cut*. Available online at https://projects.coin-or.org/Cbc.

Fourer, R., Gay, D., and Kernighan, B. (2003). *AMPL: A Modelling Language for Mathematical Programming*. Brooks/Cole Publishing Company, Pacific Grove, CA.

Gomory, R. E. (1958). Outline of an algorithm for integer solutions to linear programs. *Bulletin of American Mathematical Society*, 64:275–278.

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

Gurobi (2009). *Gurobi optimizer reference manual*. Available online at http://www.gurobi.com.

Hoffman, K. and Padberg, M. (1991). Improving LP-representations of zero-one linear programs for branch-and-cut. *ORSA Journal on Computing*, 3(2):121–134.

IBM (2009). *User's manual for CPLEX V12.1*. Available online at http://ibm.com/software/integration/optimization/cplex/.

Nemhauser, G., Savelsbergh, M., and Sigismondi, G. (1994). MINTO, a Mixed INTeger Optimizer. *Operations Research Letters*, 15(1):47–58.

Ralphs, T., Guzelsoy, M., and Mahajan, A. (2010). *SYMPHONY 5.2.3 user's manual*. Available online at https://projects.coin-or.org/SYMPHONY.

Rothberg, E. and Hendrickson, B. (1998). Sparse matrix ordering methods for interior point linear programming. *Informs Journal on Computing*, 10(1):107–113.

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

Suhl, U. H. and Szymanski, R. (1994). Supernode processing of mixed-integer models. *Computational optimization and applications*, 3:317–331.

Tomlin, J. and Welch, J. (1986). Finding duplicate rows in a linear programming model. *Operations Research Letters*, 5(1):7–11.

van Roy, T. J. and Wolsey, L. A. (1987). Solving mixed integer programming problems using automatic reformulation. *Operations Research*, 35(1):45–57.

Wolsey, L. A. (1990). Valid inequalities for 0-1 knapsacks and MIPs with generalized upper bound constraints. *Discrete Applied Mathematics*, 29:251–261.