Improving LP-Representations of Zero-One Linear Programs for Branch-and-Cut

KARLA L. HOFFMAN  
Department of Operations Research and Applied Statistics, George Mason University, Fairfax, VA 22030, BITNET: khoffman@gmuvox

MANFRED PADBERG  
Stern School of Business, New York University, New York, NY, and Laboratoire d'Econometrie, Ecole Polytechnique, Paris

(Received: October 1989; final revision received: July 1990; accepted: October 1990)

We present various techniques for automatically improving the LP-representation of general zero-one linear programming problems. These include detection of redundant rows and blatant infeasibilities, coefficient reduction using the Euclidean algorithm, optimality fixing and variable elimination. Extensions to the case where special-ordered-set constraints are present are discussed as well. A summary of the branch-and-cut approach to general zero-one problems (including flowcharts) is given. We report numerical experiments to test the effect of such preprocessing within a branch-and-cut algorithm for eleven large-scale real-world zero-one linear-programming problems. An illustrative example is included in the Appendix.

Ever since integer programming methods were used to solve practical problems, it has been well known that problems with several hundred integer variables could sometimes be solved (to optimality) quite quickly, while others with less than 100 integer variables were extremely difficult or even impossible to solve in reasonable computation times. However, in certain cases, reformulating an otherwise unsolvable problem made some of these problems tractable. There are many ways of representing an integer programming problem by linear inequalities while guaranteeing that the underlying set of feasible integer solutions is essentially unchanged. For enumerative approaches to integer programming “compact” formulations of such problems are useful, i.e., formulations involving the smallest possible number of constraints in the same set of variables. However, when linear programming is used as a component of the problem-solver, compactness considerations give no direction for improving the chances of solving the underlying integer program. Rather what is needed is a formulation of the problem for which the “gap”—i.e., the difference in the objective function values between the solutions to the linear programming relaxation and the integer program, respectively—is as small as possible.

To make the preceding precise, denote by (ZOP) the user-supplied problem

\[
\max \{ cx : Ax \leq b ; x \in \{0,1\}^n \} \quad \text{(ZOP)}
\]

where \( A \) is a \( m \times n \) matrix and \( b \) is a \( m \)-vector of rationals and denote by

\[
P_f(A,b) = \text{conv} \{ x \in \{0,1\}^n : Ax \leq b \}
\]

the convex hull of feasible solutions to (ZOP). Problem (ZOP) is then the problem of finding \( x^* \in P_f(A,b) \) such that \( cx^* = z_f \), where

\[
z_f = \max \{ cx : x \in P_f(A,b) \} \quad \text{(2)}
\]

is the (unique) optimal objective function value of (ZOP) provided it exists. Let \((D,d)\) be any \( p \times n \) matrix and \( p \)-vector of rationals, respectively, such that the polytope \( P_f(D,d) \) satisfies:

\[
P_f(D,d) \subseteq P_f(A,b) \text{ and } \exists x' \in P_f(D,d) \text{ such that } cx' = z_f.
\]

We call \((A,b)\) and \((D,d)\) equivalent representations of the given integer program \( P \) even though some alternative optimal solutions may be lost, if they exist at all. Denoting

\[
z(A,b) = \max \{ cx : Ax \leq b , 0 \leq x \leq 1 \} \quad \text{(3)}
\]

the objective function value of the naive linear programming relaxation of (ZOP), we call \((D,d)\) a tighter representation of problem (ZOP) provided that

\[
z(D,d) \leq z(A,b).
\]

The “gap” of the two respective representations of (ZOP) is \( z(A,b) - z_f \) and \( z(D,d) - z_f \), respectively. For further elaboration on LP-based combinatorial problemsolving the reader is referred to [12].

Subject classifications: Programming: integer; cutting plane/facet generation; relaxations.
Other key words: Zero-one programming, preprocessing, automatic reformulation, branch-and-cut, scientific computation.
In this paper, we present procedures that transform a “user-supplied” formulation automatically into a “tighter” equivalent representation, i.e., into one having a smaller gap. These methods require relatively little computational effort and have been shown by us to be highly effective in reducing the solution times on large zero-one programming problems. This paper continues the earlier work on “preprocessing” of zero-one linear programs of [4]. For further discussions and a historical perspective on preprocessing an formulation techniques, see [27] and also [2, 3, 4, 8-11, 15, 16, 18, 19, 21, 25, 26].

The techniques described here classify constraints, permanently fix variables, check for inconsistencies among constraints, and reduce the size of certain coefficients within the constraint matrix. Additional “new” variables are not added because we are concerned with the automatic reformulations of a problem that is already formulated. The preprocessing described here should be seen as complementing the generation of “polyhedral cuts” or new constraints which is the most effective way to tighten the linear programming relaxation of an integer program. This topic is the subject of two companion papers on constraint generation techniques based on the polyhedral theory for the symmetric traveling salesman [24] and for large zero-one problems [14], respectively.

As users employ increasingly general modeling system such as GAMS34 to construct problem formulations and extract data from bases, it is certain that automatic reformulation techniques that determine the substructures within the model, that identify inconsistencies in the data entries and that better approximate the integer polytope will become even more important than they are now.

Section 1 presents the overall description of a general preprocessor that we have implemented and tested on a variety of real-world large zero-one problems. Section 2 describes in detail each of the major components together with complexity measures, where possible. Section 3 presents an overview of our branch-and-cut optimizer, of which the preprocessor is but one component. Finally, Section 4 presents computational results illustrating its use within a branch-and-cut algorithm and in the appendix an example is given. The test problems of the computational study are available from the authors (bitnet address: KHOFFMAN@GMUVAX). An earlier version of this paper was circulated under a different title13.

1. Overall Flow of the Preprocessor

Preprocessing refers to elementary operations that can be performed automatically to tighten a given formulation. A flow chart of the preprocessing algorithm is exhibited in Figure 1. The algorithm contains seven major parts which:

- classify each row as one of four types;
- scale the A-matrix based on Euclid’s algorithm, thereby conditioning the matrix and tightening the right-hand side vector based on the pure integer structure of the problem;
- check for inactive and infeasible rows and fix variables based on feasibility requirements;
- reduce coefficients within the A matrix;
- identify redundant or inconsistent rows;
- fix variables based on optimality criteria; and
- eliminate a variable from each special-ordered-set equation, transforming the equation to an inequality and making the appropriate variable substitution in all other constraints.

In addition to these major components, the preprocessing algorithm contains the following service procedures which:

- transform the A-matrix from packed row-major form to packed column-major form;
- remove a single element from the row-major data structure and shift the matrix accordingly;
- remove an entire row from the row-major data structure; and
- order a vector in ascending order using a quick-sort algorithm.

The next section presents a detailed discussion of each of the major components of the reformulation algo-

![Flowchart of preprocessor](image-url)
rithm. This algorithm can be used as a stand-alone procedure to preprocess the user-supplied formulation in order to tighten the formulation of a zero-one linear programming problem. Section 4 of this paper will show how repeated use of the preprocessing algorithm in a tree-search/LP-based zero-one problem solver can reduce the time required to solve large zero-one problems substantially.

2. Algorithmic Description

In this section we describe the individual components of the preprocessor in detail. Note that the first five subsections assume that the constraint matrix is available in row-major format only, whereas the procedures of Sections 2.6 and 2.7 require both the row-major and column-major structure. These data structures are important for the complexity calculations that are stated in each section.

2.1. Row Classification

The row-classification procedure partitions constraints into four general types. All constraints having only zeroes, ones and minus ones are classified as either special-ordered set (SOS) constraints or invariant knapsacks. If they are of the form:

$$\sum_{j \in L} x_j - \sum_{j \in H} x_j \leq 1 - |H|$$ (5)

where L and H are disjoint index sets and |H| denotes the cardinality of the set H, they are classified as special-ordered sets. Otherwise, they are classified as invariant knapsacks. The algorithm identifies constraints having this special structure so that a cutting-plane algorithm can exploit this structure. In its present form the algorithm does not require that two distinct special-ordered sets be disjoint. It classifies invariant knapsacks as such, anticipating the possibility that sometime later (after the fixing of some variables) these constraints too will satisfy the special-ordered set requirement.

When there are nonzero coefficients in the constraint matrix other than +1, or −1, the algorithm next determines if the constraint is of the following form:

$$\sum_{j \in P} x_j \leq Mx_p$$ (6)

where M > 0 (or if the constraint can be transformed to this form by the simple substitution of $x'_j = 1 - x_j$, where necessary). Any such constraint is classified as a plant-location constraint and the variable $x_p$ is labelled the plant variable. We note that if M is greater than |P|, it will be reduced to |P| by the coefficient reduction procedure described in Section 2.4. If a constraint has the same form as (6) except that the inequality is reversed, the constraint is labelled a “reverse plant-location” constraint. Any constraint not of the preceding forms is classified as a knapsack constraint.

The classification algorithm examines sequentially each row of the matrix (which is represented in packed row-major form) and sets a flag to true whenever an element in a row is found that does not equal +1 or −1, and records the index of the associated variable. If a second such element is found in the same row, the constraint is classified as a knapsack constraint. Scanning through the nonzero elements of the row, the sum of all the positive and the negative elements are calculated and stored thereby allowing, at the completion of a single pass through the row, the classification of the row into either (1) a special-ordered set, (2) an invariant knapsack, (3) a plant-location, or (4) a knapsack constraint. The classification of a row is bounded by O(NROW) operations, where NROW is the maximum number of non-zero elements in any row. Due to the sparsity of large zero-one linear programming problem matrices, NROW is typically significantly smaller than the number of variables in the problem.

2.2. Euclidean Reduction

For any knapsack row with rational coefficients $a_j$ denote by $K_{\text{EXP}}$ the smallest number such that $a_j \times 10^{K_{\text{EXP}}}$ is an integer for all $j$. (Thus if all $a_j$ are integer, $K_{\text{EXP}} = 0$). The Euclidean reduction procedure finds this exponent and determines simultaneously the greatest common divisor, GCD, of the resulting integer data. The row is then multiplied by $10^{K_{\text{EXP}}}$ and both sides of the constraint are divided by GCD. If an equation has a non-integer right-hand side after division, the problem has no feasible integer solutions. Otherwise, if the constraint is of the "$\leq$" type ("$\geq$" type) we truncate the remainder (truncate and add one). This truncation does not eliminate any integer solutions, but makes the set of non-integer solutions to the relaxed linear-programming problem smaller, i.e., it "tightens" the L.P. relaxation problem (ZOP). In the process of performing these calculations, the largest and smallest $|a_j|$ in the row are calculated and stored as MAXAJ and MINAJ, respectively and, simultaneously, the sum of all the positive and negative $a_j$, labeled SUM1 and SUM2, respectively, are also calculated and stored. This information is used in procedures discussed in Sections 2.3 and 2.7.

Before exiting this phase, the algorithm determines whether a row, after Euclidean Reduction can be reclassified as a special-ordered set constraint or invariant knapsack by checking if ABS(MAXAJ-MINAJ) = 0.

2.3. Check for Inactive or Infeasible Rows

For all knapsack and plant-location inequalities, the algorithm puts the constraint in the following form:

$$\sum_{j \in K^+_p} a_j x_j + \sum_{j \in K^-} a_j x_j \leq a_0$$ (7)

where $K^+_p$ denotes the index set of coefficients $a_j$ with
positive values and $K_-$ denotes the index set of coefficients $a_j$ with negative values. We perform initially the same checking as described in [4] for infeasible or inactive rows.

We extend and strengthen these tests for inactive or infeasible knapsack rows when disjoint SOS-constraints are considered in conjunction with a knapsack constraint. Note that for every SOS-constraint (5) the constraint

$$\sum_{j \in L^*} x_j - \sum_{j \in H^*} x_j \leq 1 - |H^*|.$$  \hfill (8)

is valid for (ZOP) for any $L^* \subseteq L$ and $H^* \subseteq H$ and that (8) is also a SOS-constraint. Using this observation we can always construct a set of disjoint SOS-constraints from any set of overlapping SOS-constraints.

Let $S$ be the row indices of a set of disjoint SOS-constraints and for $j \in S$ denote by $L_j$ and $H_j$ the corresponding sets of positive and negative elements as in (5). Finding such a disjoint set requires one pass through the matrix, examining the indices of each SOS-constraint for conflict. Let $H_0 = \bigcup_{j \in S} H_j$, then for each $j \in H_0$ make the substitution $x_j = 1 - x_j$ in the knapsack row and in each SOS-constraint in set $S$. We now have a new knapsack constraint in the form given in (7) with $K_+$ and $K_-$ denoting the index sets of the coefficients $a_j$ with positive and negative values respectively; each SOS-row in set $S$ has been transformed to have all positive elements with a right-hand-side equal to one. Let $a^*_j(a^-_j)$, $i \in S$ be the largest (smallest) $a_j$ for all $j$ in SOS-row $i$. Set $a^*_i = 0$ if $a^*_i \leq 0$ and set $a^-_i = 0$ if $a^-_i \geq 0$. Let $T = \bigcup_{i \in S} L_i \cup H_i$. A stronger test for blatant infeasibility than that given in [4] goes as follows: If

$$\sum_{i \in S} a_i + \sum_{j \in K_+ \setminus T} a_j > a_0,$$  \hfill (9)

then constraint (7) does not have a feasible solution and the overall problem (ZOP) is blatantly infeasible. On the other hand, if

$$\sum_{i \in S} a^*_i + \sum_{j \in K_- \setminus T} a_j \leq a_0$$  \hfill (10)

then constraint (7) is inactive because every zero-one vector satisfies it. Likewise, a stronger test for the fixing of variables is given by: If for any $j \in K_+$,

$$a_j > a_0 - \sum_{i \in S} a^*_i - \sum_{j \in K_- \setminus T} a_j$$  \hfill (11)

then $x_j = 0$ in every feasible zero-one solution to (ZOP), and if for any $j \in K_- \setminus T$,

$$-a_j > a_0 - \sum_{i \in S} a^-_i - \sum_{k \in K_+ \setminus T} a_k$$  \hfill (12)

there $x_j = 1$ in every feasible solution to (ZOP).

We use logic similar to the above algorithm to examine equations for infeasibility or redundancy. If an equation has a single nonzero element $a_j$ and the right-hand side equals zero, then $x_j$ is fixed at zero and the row is deleted. Otherwise, for equations with a single non-zero element $x_j$ is fixed to one and the row is deleted if $a_j = a_0$. In any other case where $|K_+ \cup K_-| = 1$, problem (ZOP) is blatantly infeasible since there are no feasible zero-one solutions to this problem.

Consider equations having more than one nonzero element. If

$$\sum_{j \in K_+} a_j < a_0,$$  \hfill (13)

then the problem is infeasible; if

$$\sum_{j \in K_-} a_j = a_0,$$  \hfill (14)

then necessarily $x_j = 1$ for all $j \in K_+$ and $x_j = 0$ for all $j \in K_-$ in every feasible solution to (ZOP). We fix these variables correspondingly and delete the row.

The complexity of determining if a row is inactive or infeasible (and fixing variables where appropriate) is $O(NROW)$ when SOS-rows are not considered. It is of order $O(NROW*(NSOS + 1))$ when SOS rows are used to strengthen the formulation (where NSOS is the number of SOS constraints in the problem).

2.4. Coefficient Reduction

Another automatic way of tightening the LP relaxation of (ZOP) is obtained by "reducing" the size of the coefficients of individual (or several) constraints of the problem where possible. Geometrically, this corresponds to a "rotation" of the constraints so as to increase the number of zero-one solutions that satisfy them at equality. One way of performing such "coefficient reduction" goes as follows: As a first step, the algorithm transforms an inequality knapsack or plant-location constraint to the following form:

$$\sum_{j \in K} a_j x_j \geq a_0$$  \hfill (15)

where $a_j > 0$ for all $j \in K$ by making the substitution $x'_j = 1 - x_j$ where necessary. As noted in [4], if $a_j > a_0$ for some $j \in K$ then the inequality with $a_j$ replaced by $a_0$ is equivalent in terms of zero-one solutions to the original inequality. One has reduced the "size" of the coefficients $a_j$, whence the term coefficient reduction.

We extend and strengthen this test for coefficient reduction from [4] to the (frequently occurring) situation when SOS-constraints are present in conjunction with a knapsack constraint. Consider a knapsack row in the form (7) and the set of all SOS-constraint having at least two indices in common with $K_+ \cup K_-$. Using the same observation as in Section 2.3 we construct from these constraints a set of disjoint SOS-constraints that are compatible with the signs of the coefficients of the knapsack row (7), i.e. if a variable $j \in K_+$ appears in an SOS-constraint then its coefficients equals +1, while it equals -1 if $j \in K_-$. Denote by $S$ the set of row indices of such a compatible set of disjoint SOS-constraints, denote by $I_k$ the set of indices in SOS-constraint $k \in S$ and let $K = K_+ \cup K_-$. By complementing all of the variables
\( K \) it follows that the constraint set can be written as
\[
\sum_{j \in K \setminus Q} a_j x_j + \sum_{j \in Q} a_j x_j \leq a_0,
\]
\[
\sum_{j \in I_k} x_j \leq 1 \text{ for all } k \in S,
\]
where
\[
Q = K \setminus \bigcup_{k \in S} I_k \text{ and } a_j > 0 \text{ for all } j \in K.
\]
Without loss of generality we assume that \( a_0 > 0 \).

Given this constraint set, we now state an improved algorithm for coefficient reduction. This algorithm exploits the fact that at most one variable in each SOS-row can take on the value one.

**Algorithm for Coefficient Reduction When SOS-Constraints Are Present**

**Input:** Coefficients \( a_0 \) and \( a_j \) for \( j \in K \);
the sets \( Q \), \( S \), and \( I_k \) for \( k \in S \) as defined previously.

**Output:** Coefficients \( a_0 \) and \( a_j \) for \( j \in K \).

**begin**

**Initialization**

Set \( a_j^* = \max_j a_j \) for all \( k \in S \),
\( SSUM = \sum_{k \in S} a_j^* \),
\( ASUM = \sum_{j \in Q} a_j \), and \( \Delta = 0 \).

**check if constraint inactive**

if \( a_0 \geq SSUM + ASUM \) then knapsack constraint is inactive, remove it return

else

**determine if largest coefficient in SOS-row can be tightened; if so, perform reduction.** (Note: \( \Delta \) is total change in \( a_0 \)).

Set \( RHS = SSUM + ASUM - a_0 \)
for every \( k \in S \) do

if \( a_k^* > RHS \), then

Set \( \Delta = a_k^* - RHS \)
\( \Delta \leftarrow \Delta + \delta \)
for all \( j \in I_k \) do

\( a_j \leftarrow \max\{0, a_j - \delta\} \)
enddo

**reduce non-SOS coefficients in row where possible**

for all \( j \in Q \) do

if \( a_j > RHS \) then

\( \Delta \leftarrow \Delta + (a_j - RHS) \)
\( a_j \leftarrow RHS \)
enddo

**reduce \( a_0 \)**

\( a_0 \leftarrow \max\{0, a_0 - \Delta\} \)
end

**end**

To prove that the algorithm is correct one distinguishes the two cases where \( \sum_{j \in I_k} x_j = 1 \) and \( \sum_{j \in I_k} x_j = 0 \), respectively, and uses induction on \( |S| \). To illustrate the algorithm consider the example
\[
4x_1 + 95x_2 + 96x_3 \leq 100,
\]
\[
x_1 + x_2 \leq 1.
\]

Applying the algorithm in conjunction with the Euclidean reduction of Section 2.2 we obtain the constraints \( x_1 + x_2 \leq 1 \), \( x_2 + x_3 \leq 1 \), which constitutes a much "tighter" representation of the zero-one linear program than that given by the two original constraints.

As a final note on coefficient reduction, similar to the procedures by Dietrich and Escudero, we can perform additional coefficient reduction whenever there is a plant-location constraint that has the plant-variable as well as some other variables in common with a knapsack constraint. Let \( K \) be the index set of variables in the knapsack row, \( P \) be the index set of variables in the plant-location constraint (excluding the plant variable) and denote by \( d \) the objective function value of the auxiliary problem.

**Max** \[
\sum_{j \in K \setminus P} a_j x_j \mid \sum_{j \in K \setminus P} a_j x_j \leq a_0, x_j \in \{0,1\}\]

Then we can replace in (16) \( a_0 \) by \( d \) and \( a_p \) by \( a_p - (a_0 - d) \) and obtain a tighter constraint. The coefficient reduction is correct since \( x_p = 0 \) implies that \( x_j = 0 \) for all \( j \in P \), while if \( x_p = 1 \) we are subtracting a constant from both sides of (16). In the actual implementation of this coefficient reduction method one does not solve the auxiliary problem. Rather one checks whether or not \( \sum_{j \in K \setminus P} a_j < a_0 \). If this is the case, then clearly \( d = \sum_{j \in K \setminus P} a_j \) and the coefficient reduction is carried out as indicated. Similar coefficient reduction results can be obtained for constraints of forms similar to that of the plant-location constraints (e.g., \( Mx_p \leq \sum_{i \in P} x_i \)) by applying essentially the same logic.

The procedures described in Sections 2.1–2.4 are each performed on a specific row before the next row is analyzed. Each of these procedures requires O(NROW) instructions. Therefore the total complexity of all processing described so far is O(KNAP*NROW) when SOS-constraints are not considered where KNAP is the number of knapsack constraints. The improved procedures with SOS-constraints require one pass through the SOS-rows (which are already labelled as such) to obtain a disjoint compatible set. Thus the complexity of the overall algo-
rithm is considerably less than the worst case bound of \( O(KNAP*NROW*(SOS + 1)) \) where SOS is the number of SOS-rows in the problem.

2.5. Detecting Infeasibility and Simple Redundancy

Bixby and Wagner [11] present a procedure that, given a linear system \( Ax = b \), detects and removes equations that are nonzero multiples of other equations within that system. The following algorithm does the same, but it also determines whether or not two inequalities form an equation and whether or not an inequality is dominated by some other inequalities. It also checks for inconsistencies among inequalities.

A single pass through the matrix places the number of nonzero elements in each row into a vector which is then sorted in increasing order using a quick-sort routine (a quick-sort is used because the number of rows is too small to warrant more sophisticated sorting strategies). Rows with equivalent numbers of nonzero elements are compared. For each such collection of rows, all pairwise comparisons of elements are made. The algorithm determines whether or not the first element in one row is the same (in absolute value) as the first element in another, and whether the corresponding column indices match. If a match is found, a flag is set indicating if both elements are positive, both negative, if the first is positive and the second is negative, or vice versa. The algorithm moves to the next row if the first elements are not equal. If they are equal, the second elements are compared in a similar fashion. The search continues until it is determined that either all rows have been checked without a match being found, or two rows agree (up to a sign difference) in each element. If such a match is found, the right-hand-side values of these two constraints are then checked for the following possibilities:

- If both constraints are equations and the right hand sides agree, one of the two equations is removed. If they do not agree problem (ZOP) is infeasible.
- If one constraint is an equation the other one is an inequality, then either:
  - the right-hand sides are equal, and the inequality is removed; or
  - the inequality is implied by the equation and the inequality is removed;
  - otherwise, problem (ZOP) is infeasible.
- If both constraints are inequalities, then either:
  - one inequality dominates the other and the dominated constraint is removed; or
  - the two inequalities form an equation;
  - otherwise, problem (ZOP) is infeasible.

We note that a row cannot be a multiple of another row because Euclidean reduction has previously been performed on all rows.

This procedure could be expanded to examine if rows with different numbers of elements dominate one another. In this case, each row must be compared to every other row with an equal or greater number of nonzero elements. For the worst case performance estimate of the existing algorithm we assume that a comparison of every element against every other element in the matrix is required. This can occur only when all rows have an equal number of nonzero elements, and all columns except the last are identical. In general, our procedure requires much less computation, since only rows with an equal number of nonzero elements are compared and the comparison process stops as soon as two nonagreeneing elements are found. Our algorithm requires in the worst case \( O(M*NROW) \) operations after a simple sort of \( M \) numbers has been performed and this bound is very conservative.

2.6. Optimality Fixing

The fixing of variables due to optimality considerations requires the examination of columns rather than rows. Therefore, an exchange procedure transforms the matrix from packed row-major structure to packed column-major structure. The transformation from row-major to column-major structure requires two passes through the nonzero elements of the matrix. The first pass counts the number of elements in each column, thereby creating the column-pointer structure. The second pass then inserts each element into its proper position.

Once in column-major format, the procedure scans each column to determine if a given column has any nonzero entries. If the column is empty (which may happen after some rows have been eliminated by preprocessing), the corresponding variable is permanently fixed based on the sign of its objective function coefficient \( c_i \). E.g., in a maximization problem, if \( c_i \) is negative (positive), then variable \( x_i \) is permanently fixed to zero (one).)

If column \( j \) has any nonzero elements in rows corresponding to equations, then variable \( x_j \) cannot be fixed. Otherwise, let \( H \) and \( L \) be the set of row indices for variable \( x_j \) such that \( i \in H \) if row \( i \) is of the \( '\geq' \) type and \( i \in L \) otherwise. We assume that the direction of the optimization is maximization and examine only the nonzero elements \( a_i \) of column \( j \) as follows:

1. If \( c_j \geq 0 \), \( a_i > 0 \) for all \( i \in H \), and \( a_i < 0 \) for all \( i \in L \), then variable \( x_j \) is fixed to 1;
2. If \( c_j < 0 \), \( a_i > 0 \) for all \( i \in H \), variable \( x_j \) is fixed to 0.

The work of the exchange procedure requires two passes through the number of nonzeros of the constraint-matrix \( A \). The work of the optimality fixing procedure is also linear in the number of nonzeros of the constraint matrix. However, since the latter procedure does not scan every column completely, the overall work carried out by
this procedure is considerably less than its worst-case bound.

2.7. The Elimination Procedure

Polyhedral theory is easier to apply if one has full-dimen-
sional polyhedra. Having equations reduces the dimen-
sionality and it is therefore desirable to eliminate variables
from equations in order to get closer to full-dimensionality.
The elimination procedure eliminates a single variable
from each special-order-set equation, transforms the equa-
tion to an inequality, and makes the appropriate substitu-
tions in all other constraints. This operation is repeated as
long as "qualifying" special-ordered-set equations are
found. We restrict ourselves to special-ordered set equa-
tions because the elimination of variables from other types
of constraints is mathematically more complicated and, in
particular, requires the substitution, in some cases, of two
inequalities for a single equation.

We consider a special-ordered set constraint in the
following form:

\[ \sum_{j \in L} x_j - \sum_{j \in H} x_j = 1 - |H| \quad (17) \]

where \( L \) and \( H \) are disjoint index sets. The elimination of
a variable \( x_k, k \in L \), requires the removal of the variable
\( x_k \) from the data set, and the substitution

\[ x_k = 1 + \sum_{j \in H} x_j - \sum_{j \in L, j \neq k} x_j - |H| \quad (18) \]

in both the objective function and all constraints in which
the variable \( x_k \) appears with a nonzero coefficient. In
particular, the elimination of variable \( x_k \) implies that (17)
must be replaced by the two inequalities:

\[ \sum_{j \in H} x_j - \sum_{j \in L, j \neq k} x_j \leq |H|, \quad (19) \]

\[ -\sum_{j \in H} x_j + \sum_{j \in L, j \neq k} x_j \leq 1 - |H|. \quad (20) \]

Inequality (19) is redundant and is not stored, while
inequality (20) remains a SOS-constraint and is kept as
long as \( |H| + |L| \geq 3 \).

Making the substitution specified by equation (18) in
all corresponding constraints accounts for the bulk of the
computation. In order to perform these substitutions, the
elimination routines exploit the fact that both row-major
and column-major structures of the constraint matrix \( A \) of
problem (ZOP) are stored.

Given the column-major structure, determining the
rows in which the eliminated variable appears is trivial.
However, to repeatedly use this information, it is neces-
sary to update both the row-major and the column-major
structure. When there are only a few overlapping con-
straints and when these constraints have few nonzero
entries, the substitutions and appropriate shifts in the data
structures are made simultaneously to both formats. When
the overlap is large or when the special-ordered set con-
straint is relatively long, the substitutions are made only in
the row structure. (In the current version of our software
system, the corresponding parameter for overlap equals
five.) The algorithm then transforms the entire matrix
back into the column-major format before the next vari-
able is eliminated.

Whenever the substitution specified in equation 18 is
made in a special-ordered set equation, all other SOS-
equations containing \( x_j \) are flagged. The routine then
picks a nonflagged SOS-equation and a corresponding
variable to eliminate. After all SOS-equations are flagged,
the routine begins again, rechecking all equations to deter-
mine if any of the flagged SOS-equations remain SOS-
equations after the substitutions. Elimination proceeds in
this fashion until no more special-ordered-set equations
exist.

The complexity of this procedure is dependent on the
number of SOS-equations and the density of those equa-
tions. Under the worst-case assumption that all rows are
SOS-equations and that these rows are dense, the substitu-
tion procedures would require one pass through the nonze-
roes of the A-matrix (to alter the row structure) and an
additional two passes to correct the column structure (via
the exchange routine). Thus the worst-case complexity
of the algorithm, in its current form, would be \( O(M^2NZERO) \).
For general zero-one problems the complexity of this algorithm is considerably more moderate,
as exhibited by the computation times presented in the
next section. We can improve on the time required for the
elimination routines if we maintain additional list struc-
tures which store the additions and deletions from each
row and column associated with a substitution. All such
additions and deletions to the matrix can then be made in
a single pass through the A-matrix. Future research will test
the space-time tradeoffs of this alternative approach.

3. Brief Description of a Branch-and-Cut Solver

The previous section described procedures that are used
for automatically tightening the user-supplied formation of
a pure zero-one programming problem. This section will
briefly describe how such a procedure fits within the
overall branch-and-cut algorithm. For a more detailed
description see [14] and [24].

The major modules and their use in the overall
branch-and-cut search tree are shown in Figure 2, while
Figure 3 provides a flow chart of our algorithm ABC_OPT
(A Branch and Cut OPTimizer). We begin by preprocessing
the problem as described above. After reformulation
of the user-supplied problem, an upper bound \( z_{LP} \) on
the problem is determined by optimizing the linear program-
ming relaxation.

A lower bound for the problem \( z^* \) (which, initially,
is minus infinity) is obtained by calling a general zero-one
Figure 2. Flowchart of branch-and-cut search tree.

The heuristic developed by the authors. The heuristic, like any other linear-programming based heuristic, combines three fundamental principles, for such methods: rounding, infeasibility reduction and enumeration of a part of the problem up to a certain small threshold (in our case "small" means less than or equal to 16). If such steps do not produce a feasible integer solution, then the objective function is augmented with penalties, additional linear programming problems are solved and the heuristic is iterated a number of times. A technical report which details this heuristic is in preparation.

Given an upper and lower bound, one can use reduced-cost fixing, see [4] or [24] for more detail, to fix additional variables and return to the preprocessing routines to determine the implications of this fixing on other variables.

The constraint generation procedures, the engine of this branch-and-cut solver, generate valid linear inequalities that approximate facets of the convex hull of integer points and which we call polyhedral cuts. These inequalities are violated by the fractional LP solution but do not cut off any feasible integer points. The constraint generation procedures used in this code are similar to those described in [4] for knapsacks and knapsacks with associated special-ordered set constraints. One major difference between the routines for constraint generation here and those of [4] is that we project out variables both at zero and at one and a knapsack problem is solved to find the most violated minimal cover over only the fractional variables or even a subset thereof. One then sequentially lifts back first the (remaining) fractional variables not in the minimal cover, then the variables which were projected out at one, and then the variables projected out at zero. In this manner we assure that the inequalities obtained are valid for the problem and approximate the integer polytope in the area around the fractional linear programming solution. Projecting out at value zero corresponds to the usual "lifting procedure," see [20, 22, 23], while projecting out at value one is a sort of "reverse" lifting, see [28] for more detail. Using both types of projection, it then becomes unnecessary to distinguish between cuts that are generated from "minimal covers" and from "(1, k)-configurations," respectively; see [4] for unexplained terminology. (We provide in the appendix an example which illustrates this point.)

A second difference between [4] and the current constraint-generation procedures concerns the "lifting sequence." Since the lifting is done sequentially different
orderings of the variables to be lifted produce, in general, different facets. The order is determined based on both the first-order lifting coefficient and the reduced cost of the nonbasic variables. By varying the order of the variables to be lifted, we therefore obtain a number of valid inequalities from a single row that may correspond to lifted minimal cover inequalities, (1, k)-configurations and other types of constraints as well.

The violated constraints are appended to the original problem and the LP solver is again called. We iterate through this loop until one of the following cases prevails: (1) the solution is integer; (2) the LP is infeasible; (3) no additional cuts are generated (either because of our incomplete knowledge of the polyhedral structure or because of the incompleteness of our constraint-generation procedures); (4) although cuts are generated, the objective function is not increasing sufficiently, i.e. we detect a "tailing off" of the procedure; or (5) the objective function has improved substantially relative to the bounds on other nodes of the search tree; we "pause this node" and look at the more promising nodes.

If the first situation occurs and we are at the root node of the search tree, the solution obtained is optimal and the algorithm terminates. If this situation occurs within the search tree, we fathom the node, update the lower bound \( z^* \) and continue the tree-search. If the second situation occurs at the root node of the tree, the overall problem is infeasible; if within the tree, we fathom the node and continue. In the third or fourth case occurs, we will expand the tree. However, before expanding the tree we call the heuristic again in an attempt to update the lower bound \( z^* \). If the heuristic finds a better feasible solution, (or if the application of the constraint-generation procedures yielded a sufficient "gain" over the previous upper bound and \( z^* \) exists) the reduced-cost fixing and logical fixing routines are called. If sufficiently many variables are fixed, the preprocessor is called again to tighten the problem formulation. The fifth case is similar to the third and fourth cases, except that the node is not expanded, but rather paused for further examination at some later time; see [24] for a more detailed discussion of the branch-and-cut algorithm in the context of the symmetric traveling salesman problem.

Thus, at every node of the branch-and-cut algorithm, the problem is reformulated, linear programs are solved, polyhedral cuts are generated, and a heuristic is called. Normally, the only reformulation done within the branching tree is the implied "setting" of variables based on the setting of the branching variables. At the root node or, if an improved lower bound has been obtained within the tree, permanent fixing is performed on the basis of the LP reduced costs at the root node (which are stored) and the matrix is permanently altered, i.e. the preprocessor is called. We note that because the polyhedral cuts are valid throughout the tree (which is not true for traditional cutting planes, such as Gomory cuts or "intersection" cuts) inequalities that are generated in any part of the search tree remain valid "globally," i.e. across the entire search tree, and the data structures remain unchanged when we move from one branch of the search tree to another unless the preprocessor is invoked.

As is readily apparent from the above description, the reformulation stage is not merely an initial tightening of the user's formulation but a set of procedures that is routinely called within the overall branch-and-cut algorithm to solve large zero-one problems. The next section will provide our computational experience with this algorithm.

4. Computational Results

Section 3 described the procedures that we use for automatically tightening a user-supplied formulation of a pure zero-one linear programming problem. This section describes our computational experience with these routines when they are incorporated within an experimental computer software system ABC\_OPT which uses a branch-and-cut strategy for solving such problems. This software package is written entirely in FORTRAN and has been tested on Control Data, Digital and IBM mainframes. As discussed in Section 4, the software system employs a tree-search strategy for the solution of integer-programming problems that is radically different from the traditional branch-and-bound method for integer programming, see Table I for a comparison of the nodes of the search tree generated by the software systems PIPEX of [4] and our new system ABC\_OPT, respectively, for a subset of our test set. Besides the LP solver, the major components of ABC\_OPT consist of the preprocessor described here, a heuristic algorithm and a constraint-generator. All three components are invoked repeatedly across all branches of the search tree that is generated by ABC\_OPT, see Figure 2. A complete description of the software system can be found in a companion paper [14].

Our test set consists of the set of zero-one linear programming problems that appear in Crowder, Johnson and Padberg[4] except for problems P1550, P1939 and P2655 which were not available to us. In addition, we present results on four capital budgeting problems, labeled C0789, C1346, C1056 and C6000 provided to us by a telecommunications corporation.

Tables II through V summarize the results of the initial preprocessing step, i.e. the effect of our routines on the user-supplied formulation only. The computational experiments described here were executed on a VAX8800 using one of its two processors (making these results equivalent to running times on a VAX8700). All routines were written in ANSI FORTRAN and compiled using the VAX VMS FORTRAN compiler with default level opti-
Table I
Comparison of PIPEX and ABC\_OPT

<table>
<thead>
<tr>
<th>Problem</th>
<th>Variables</th>
<th>Constraints</th>
<th>PIPEX</th>
<th>ABC_OPT</th>
<th>Percent Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0033</td>
<td>33</td>
<td>15</td>
<td>113</td>
<td>8</td>
<td>94.0</td>
</tr>
<tr>
<td>P0040</td>
<td>40</td>
<td>23</td>
<td>11</td>
<td>0</td>
<td>100.0</td>
</tr>
<tr>
<td>P0201</td>
<td>201</td>
<td>133</td>
<td>1116</td>
<td>346</td>
<td>69.0</td>
</tr>
<tr>
<td>P0282</td>
<td>282</td>
<td>241</td>
<td>1862</td>
<td>8</td>
<td>99.6</td>
</tr>
<tr>
<td>P0291</td>
<td>291</td>
<td>252</td>
<td>87</td>
<td>4</td>
<td>95.4</td>
</tr>
<tr>
<td>P0548</td>
<td>548</td>
<td>176</td>
<td>36</td>
<td>0</td>
<td>100.0</td>
</tr>
<tr>
<td>P2756</td>
<td>2757</td>
<td>756</td>
<td>2392</td>
<td>10</td>
<td>99.6</td>
</tr>
</tbody>
</table>

\*PIPEX nodes generated by MPSX/MIP370. After problem reduction, variable fixing and constraint generation when no additional cuts could be generated, the augmented problem was sent to MPSX/MIP370 for resolution.

Table II
Effect of Preprocessing on Number of Variables

<table>
<thead>
<tr>
<th>Problem</th>
<th>No. Original Variables</th>
<th>No. Variables Fixed</th>
<th>No. Variables Eliminated</th>
<th>No. Variables Active</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0033</td>
<td>33</td>
<td>1</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>P0040</td>
<td>40</td>
<td>10</td>
<td>10</td>
<td>20</td>
</tr>
<tr>
<td>P0201</td>
<td>201</td>
<td>10</td>
<td>7</td>
<td>184</td>
</tr>
<tr>
<td>P0282</td>
<td>282</td>
<td>80</td>
<td>0</td>
<td>202</td>
</tr>
<tr>
<td>P0291</td>
<td>291</td>
<td>188</td>
<td>0</td>
<td>103</td>
</tr>
<tr>
<td>P0548</td>
<td>548</td>
<td>54</td>
<td>0</td>
<td>494</td>
</tr>
<tr>
<td>P2756</td>
<td>2756</td>
<td>52</td>
<td>0</td>
<td>2704</td>
</tr>
<tr>
<td>C0789</td>
<td>781</td>
<td>481</td>
<td>239</td>
<td>61</td>
</tr>
<tr>
<td>C1346</td>
<td>1346</td>
<td>937</td>
<td>343</td>
<td>66</td>
</tr>
<tr>
<td>C1056</td>
<td>1056</td>
<td>569</td>
<td>391</td>
<td>96</td>
</tr>
<tr>
<td>C6000</td>
<td>6000</td>
<td>25</td>
<td>125</td>
<td>5850</td>
</tr>
</tbody>
</table>

Table III
Effect of Preprocessing on Types of Rows within Problem

<table>
<thead>
<tr>
<th>Problem</th>
<th>Total</th>
<th>SOS</th>
<th>INVAR</th>
<th>PLANT</th>
<th>KNAP</th>
<th>SOS</th>
<th>INVAR</th>
<th>PLANT</th>
<th>KNAP</th>
<th>Total</th>
<th>No. of Rows Dropped</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0033</td>
<td>15</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>8</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>P0040</td>
<td>23</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>3</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>P0201</td>
<td>133</td>
<td>26</td>
<td>74</td>
<td>0</td>
<td>33</td>
<td>28</td>
<td>30</td>
<td>0</td>
<td>33</td>
<td>111</td>
<td>22</td>
</tr>
<tr>
<td>P0282</td>
<td>241</td>
<td>177</td>
<td>0</td>
<td>8</td>
<td>56</td>
<td>117</td>
<td>0</td>
<td>8</td>
<td>36</td>
<td>161</td>
<td>80</td>
</tr>
<tr>
<td>P0291</td>
<td>252</td>
<td>64</td>
<td>0</td>
<td>5</td>
<td>107</td>
<td>52</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>66</td>
<td>186</td>
</tr>
<tr>
<td>P0548</td>
<td>176</td>
<td>62</td>
<td>0</td>
<td>103</td>
<td>387</td>
<td>352</td>
<td>1</td>
<td>16</td>
<td>367</td>
<td>737</td>
<td>19</td>
</tr>
<tr>
<td>P2756</td>
<td>756</td>
<td>352</td>
<td>0</td>
<td>16</td>
<td>387</td>
<td>352</td>
<td>1</td>
<td>16</td>
<td>367</td>
<td>737</td>
<td>19</td>
</tr>
<tr>
<td>C0789</td>
<td>734</td>
<td>727</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>15</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>17</td>
<td>717</td>
</tr>
<tr>
<td>C1346</td>
<td>1283</td>
<td>1276</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>23</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>25</td>
<td>1258</td>
</tr>
<tr>
<td>C1056</td>
<td>694</td>
<td>687</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>278</td>
<td>31</td>
<td>0</td>
<td>3</td>
<td>312</td>
<td>382</td>
</tr>
<tr>
<td>C6000</td>
<td>2176</td>
<td>2169</td>
<td>0</td>
<td>0</td>
<td>7</td>
<td>2093</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2095</td>
<td>81</td>
</tr>
</tbody>
</table>
Table IV
Additional Statistics on Effect of Preprocessing

<table>
<thead>
<tr>
<th>Problem</th>
<th>STANDEV</th>
<th>DENSITY</th>
<th>MEANPROF</th>
<th>SDPROF</th>
<th>STANDEV</th>
<th>DENSITY</th>
<th>MEANPROF</th>
<th>SDPROF</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0033</td>
<td>89.3</td>
<td>19.8%</td>
<td>220.5</td>
<td>115.6</td>
<td>11.9</td>
<td>20.0%</td>
<td>228.5</td>
<td>102.6</td>
</tr>
<tr>
<td>P0040</td>
<td>271.3</td>
<td>12.0%</td>
<td>6633.3</td>
<td>875.0</td>
<td>267.7</td>
<td>15.4%</td>
<td>675.4</td>
<td>532.1</td>
</tr>
<tr>
<td>P0201</td>
<td>11.8</td>
<td>7.1%</td>
<td>497.0</td>
<td>815.0</td>
<td>2.1</td>
<td>8.7%</td>
<td>281.4</td>
<td>240.0</td>
</tr>
<tr>
<td>P0282</td>
<td>44.7</td>
<td>2.9%</td>
<td>4619.2</td>
<td>17131.3</td>
<td>41.8</td>
<td>3.9%</td>
<td>6323.1</td>
<td>19993.2</td>
</tr>
<tr>
<td>P0291</td>
<td>4.1</td>
<td>2.8%</td>
<td>1179.2</td>
<td>4766.5</td>
<td>43.8</td>
<td>5.1%</td>
<td>3249.1</td>
<td>7607.9</td>
</tr>
<tr>
<td>P0548</td>
<td>2592.3</td>
<td>0.4%</td>
<td>176.6</td>
<td>581.4</td>
<td>87.6</td>
<td>1.9%</td>
<td>189.1</td>
<td>669.3</td>
</tr>
<tr>
<td>P2756</td>
<td>444.2</td>
<td>0.8%</td>
<td>116.8</td>
<td>640.5</td>
<td>162.1</td>
<td>0.4%</td>
<td>116.8</td>
<td>640.5</td>
</tr>
<tr>
<td>C0789</td>
<td>1502.9</td>
<td>0.5%</td>
<td>459.6</td>
<td>1890.1</td>
<td>1071.0</td>
<td>2.4%</td>
<td>285.9</td>
<td>344.4</td>
</tr>
<tr>
<td>C1346</td>
<td>1842.6</td>
<td>0.6%</td>
<td>646.1</td>
<td>1854.6</td>
<td>1427.7</td>
<td>1.3%</td>
<td>1872.5</td>
<td>5004.6</td>
</tr>
<tr>
<td>C1056</td>
<td>568.8</td>
<td>1.2%</td>
<td>391.7</td>
<td>1525.8</td>
<td>490.7</td>
<td>0.8%</td>
<td>1044.9</td>
<td>145.4</td>
</tr>
<tr>
<td>C6000</td>
<td>2476.3</td>
<td>0.4%</td>
<td>216.1</td>
<td>3421.8</td>
<td>2408.5</td>
<td>0.1%</td>
<td>2189.6</td>
<td>3440.5</td>
</tr>
</tbody>
</table>

Table V
Effect of Preprocessing on LP-Value

<table>
<thead>
<tr>
<th>Problem</th>
<th>Original LP Value</th>
<th>LP Value after Preprocessing</th>
<th>True IP Value</th>
<th>Time for Preprocessing in Seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0033</td>
<td>2520.6</td>
<td>2828.3</td>
<td>3089.0</td>
<td>0.1</td>
</tr>
<tr>
<td>P0040</td>
<td>61796.5</td>
<td>61829.1</td>
<td>62027.0</td>
<td>0.1</td>
</tr>
<tr>
<td>P0201</td>
<td>6875.0</td>
<td>7125.0</td>
<td>7615.0</td>
<td>2.6</td>
</tr>
<tr>
<td>P0282</td>
<td>176867.5</td>
<td>180000.3</td>
<td>258411.0</td>
<td>3.5</td>
</tr>
<tr>
<td>P0291</td>
<td>1705.1</td>
<td>2925.8</td>
<td>5223.75</td>
<td>1.6</td>
</tr>
<tr>
<td>P0548</td>
<td>315.3</td>
<td>3126.3</td>
<td>8691.0</td>
<td>1.9</td>
</tr>
<tr>
<td>P2756</td>
<td>2688.8</td>
<td>2701.1</td>
<td>3124.0</td>
<td>13.0</td>
</tr>
<tr>
<td>C0789</td>
<td>-102345.0</td>
<td>-102345.1</td>
<td>-102397.0</td>
<td>15.7</td>
</tr>
<tr>
<td>C1346</td>
<td>88319.0</td>
<td>88319.0</td>
<td>88281.0</td>
<td>37.4</td>
</tr>
<tr>
<td>C1056</td>
<td>111010.8</td>
<td>111010.8</td>
<td>111001.2</td>
<td>12.8</td>
</tr>
<tr>
<td>C6000</td>
<td>2451541.3</td>
<td>2451541.3</td>
<td>2451377.0</td>
<td>110.4</td>
</tr>
</tbody>
</table>

Optimization. The LP-solver is XMP<sup>190</sup> using the dual-simplex method exclusively. Table II reports the number of variables still active after preprocessing. The column No. Variables Fixed refers to the number of variables fixed permanently to either zero or one, and the column No. Variables Eliminated refers to the number of variables eliminated in the elimination routines. Table II reports the number and type of rows removed by the preprocessing routines. The columns headed TOTAL, SOS, INVAR, PLANT and KNAP contain the total number of constraints, the number of special-ordered set constraints, the number of invariant knapsack constraints, the number of plant-location constraints, and the number of knapsack constraints, respectively.

Table IV summarizes the problem statistics of the test set before and after preprocessing. The column headed STANDEV states the average standard deviation of the absolute values of the nonzero elements of the knapsack rows of the constraint matrix. The columns headed DENSITY, MEANPROF, SDPROF contain the density of the matrix, the mean value of the profit array and the standard deviation of the profit array, respectively.

As stated in the introduction, we measure the improvement in the formulation of problem (ZOP) due to preprocessing by the difference in the objective function values of the linear programming relaxations of the two formulations. Table V reports the objective function value of the linear programming relaxation for the user-supplied formulation and the objective function value of the problem after preprocessing. Even for the capital budgeting problems where the linear programming relaxation value did not change, the preprocessing procedures eliminated most of the rows and permanently fixed a majority of the variables for C0789, C1346 and C1056, allowing the constraint generation routine to concentrate on the “core” of the integer problem. Table V also reports the combined time required for the two passes through the preprocessor and for the single call to the elimination procedure. We
execute the preprocessor twice whenever it is called because variables may have been fixed toward the end of the procedure (i.e., the fixing occurred when analyzing the last rows of the user-supplied formulation). This fixing could imply further reductions in preceding rows. We therefore execute the preprocessor routine once, call the elimination routine and then execute the preprocessor a second time.

The times shown are CPU seconds running in an interactive environment. In all cases, the reformulation and elimination stages required less than two minutes of processing time per problem. We note that the CJP-problems are all minimization problems while the capital budgeting problems are maximization problems.

Table VI illustrates the effect of the preprocessing routines on the overall time and effort taken to solve to optimality large-scale zero-one linear programming problems. In Table VI, NODES refers to the total number of nodes in the search tree and the column headed PASS state the number of times the software system ABC_OPT invoked the preprocessor during the entire search process. It is invoked whenever the number of variables fixed is greater than some percentage of the total number of active variables, where active variables is the number of non-fixed variables when the preprocessor was last called. This percentage is under the control of the user, but for this test it was equal to 5%. The branch-and-cut optimizer fixes variables based on feasibility and optimality considerations whenever possible in the search tree. Variables are fixed permanently based on the permanent fixing of other variables and conditionally based on the conditional fixing of branching variables. In order to test the effectiveness of the preprocessor, the preprocessing and related "logical" fixing subroutines within the code were turned off. We did, however, allow the permanent fixing of variables based on reduced cost considerations (see [4] and [6]) since such fixing is not based on the logic of preprocessing. Table VI illustrates that on small, tractable problems, the preprocessing has little effect on the overall time or search tree. On the larger problems, however, the preprocessing alters significantly the effort taken to solve these problems, as illustrated by both the number of nodes required to solve the problem and the amount of time.

Another issue that we have investigated concerns the questions of whether or not preprocessing alone is likely to make problems previously unsolvable by branch-and-bound codes solvable using only a subset of the features of the branch-and-cut optimizer ABC_OPT. Table VII shows the total time and number of nodes the software system ABC_OPT required to solve our problems when a best-node branching strategy is applied but when the cut-generator was turned off. We did not even attempt to solve the test problems by branch-and-bound without preprocessing as the major portion of the CJP-test set was previously unsolvable using a state-of-the-art commercial standard branch-and-bound code. We allowed the heuristic to be called at each node in the search tree to examine if a branch-and-bound code which repeatedly uses such heuristics coupled with preprocessing and conditional and permanent fixing of variables based on the reduced-cost information, and on logical and optimality considerations might be sufficient to make these problems solvable.

The results presented in Table VII support the premise that constraint generation is needed to make these problems solvable in reasonable times. The preprocessing does, however, lessen the overall computational effort of the branch-and-cut solver. The results of our experiment indicate there is a symbiotic relationship among the preprocessor, the heuristic procedure and the constraint generator in the sense that each is strengthened by the information provided by the others, especially when all three are invoked repeatedly across the different branches of the search tree.

### A. Appendix

Consider the zero-one knapsack problem with rational data:

\[
\text{max } 1200.x_1 + 1300.x_2 + 1300.x_3 + 1200.x_4 + 899.x_5 \\
+ 999.x_6 + 899.x_7 + 1099.x_8 + 5499.x_9 - 545.x_{10}
\]

subject to:

\[
(KP)
\]

\[
6.0.x_1 + 6.5.x_2 + 6.5.x_3 + 6.0.x_4 + 4.5.x_5 + 5.0.x_6 \\
+ 4.5.x_7 + 5.5.x_8 + 20.1.x_9 + 9.3.x_{10} \leq 19.9
\]

\[x_j \in \{0, 1\} \text{ for } j = 1, \ldots, 10.\]

The preprocessor routines transform \((KP)\) into a knapsack with integer data, find that Euclidean reduction does not apply since the greatest common division equals
Table VII
Run Effort Without Cutting Planes

<table>
<thead>
<tr>
<th>Problem</th>
<th>Best Upper Bound Found</th>
<th>Best Lower Bound Found</th>
<th>Optimal Value</th>
<th>Total Nodes</th>
<th>Total Passes</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0033</td>
<td>3089.00</td>
<td>3089.00</td>
<td>3089.00</td>
<td>412</td>
<td>2</td>
<td>351.7</td>
</tr>
<tr>
<td>P0040</td>
<td>62027.00</td>
<td>62027.00</td>
<td>62027.00</td>
<td>76</td>
<td>7</td>
<td>71.3</td>
</tr>
<tr>
<td>P0201</td>
<td>7615.00</td>
<td>7615.00</td>
<td>7615.00</td>
<td>1558</td>
<td>1</td>
<td>25948.6</td>
</tr>
<tr>
<td>P0282</td>
<td>258411.00</td>
<td>258411.00</td>
<td>258411.00</td>
<td>140</td>
<td>3</td>
<td>1693.3</td>
</tr>
<tr>
<td>P0291</td>
<td>5223.75</td>
<td>5223.75</td>
<td>5223.75</td>
<td>44</td>
<td>4</td>
<td>168.1</td>
</tr>
<tr>
<td>P0548</td>
<td>24617.00</td>
<td>3369.8</td>
<td>8691.00</td>
<td>&gt;15000</td>
<td>1</td>
<td>79449.9</td>
</tr>
<tr>
<td>P2756</td>
<td>6778.00</td>
<td>2706.20</td>
<td>3124.00</td>
<td>&gt;15000</td>
<td>2</td>
<td>155541.6</td>
</tr>
<tr>
<td>C0789</td>
<td>102397.00</td>
<td>102397.00</td>
<td>102397.00</td>
<td>8</td>
<td>1</td>
<td>277.6</td>
</tr>
<tr>
<td>C1346</td>
<td>88281.00</td>
<td>88281.00</td>
<td>88281.00</td>
<td>16</td>
<td>1</td>
<td>683.8</td>
</tr>
<tr>
<td>C1056</td>
<td>111001.20</td>
<td>111001.20</td>
<td>111011.20</td>
<td>38</td>
<td>1</td>
<td>542.7</td>
</tr>
<tr>
<td>C6000</td>
<td>2451377.00</td>
<td>2451377.00</td>
<td>2451377.00</td>
<td>1270</td>
<td>18</td>
<td>9058.3</td>
</tr>
</tbody>
</table>

*Heuristic procedure called at each node, preprocessing and variable fixing performed where possible.

1, detect that variable \( x_6 \) equals zero in every feasible solution and the optimality fixing routine finds that \( x_{10} \) equals zero in some optimal solution to \((KP)\) (indeed \( x_{10} = 0 \) in every optimal solution to \((KP)\)). In the second pass through the preprocessor routines, the Euclidean reduction routine finds a greatest common divisor of 5 and reduces \((KP)\) accordingly, thereby "tightening" the LP relaxation of \((KP)\). (Integrality of the data is desirable, but not a must, for the constraint-generation procedures—the LP solver is assumed to do a proper scaling on its own.) We thus get the problem:

\[
\begin{align*}
\text{max} & \quad 1200x_1 + 1300x_2 + 1300x_3 + 1200x_4 + 899x_5 \\
& + 999x_6 + 899x_7 + 1099x_8 \\
\text{subject to:} & \quad \begin{cases} 
(KP^*) \\
12x_1 + 13x_2 + 13x_3 + 12x_4 + 9x_5 + 10x_6 + 9x_7 \\
+ 11x_8 \leq 39 \\
x_j \in \{0, 1\} \text{ for } j = 1, \ldots, 8.
\end{cases}
\end{align*}
\]

which is - or course- the example of the appendix of [4]. Problem \((KP^*)\) is a tighter representation of \((KP)\) which, however, is equivalent to \((KP)\) in terms of its (optimal) zero-one solutions, see the general discussion of this point in the introduction. The software system ABC_\text{OPT} solves now the linear program associated with \((KP^*)\) and finds the optimal solution \(x_1 = \frac{1}{12}, x_2 = x_3 = x_4 = 1, x_5 = x_6 = x_7 = x_8 = 0\) having an objective function value of 3900. The heuristic is called and finds a rounded "Dantzig solution" (or "greedy" solution) \(x_2 = x_3 = x_4 = 1, x_1 = x_5 = x_6 = x_7 = x_8 = 0\) with an objective function value of 3800. The reduced-cost fixed routine does not fix any of the variables and thus, rather than returning to the preprocessor, ABC_\text{OPT} calls the constraint generation procedure with finds the following five valid inequalities:

\[
\begin{align*}
& x_1 + x_2 + x_3 + x_4 + x_5 + 0x_6 + x_7 + x_8 \leq 3 \\
& x_1 + x_2 + x_3 + x_4 + x_5 + 0x_6 + x_7 + x_8 \leq 3 \\
& x_1 + x_2 + x_3 + x_4 + x_5 + x_7 + 0x_6 \leq 3 \\
& x_1 + x_2 + 2x_3 + x_4 + x_5 + x_6 + x_7 + x_8 \leq 4 \\
& x_1 + x_2 + x_3 + 2x_4 + x_5 + x_6 + x_7 + x_8 \leq 4
\end{align*}
\]

the first three of which are lifted "minimal covers" inequalities and the remaining two of which are "\((1, k)\)-configurations" inequalities. The latter two inequalities are obtained by projecting out the variables \(x_3\) and \(x_4\) at one. Reversing the projection, one obtains the two inequalities depending upon the order in which variables \(x_3\) and \(x_4\) are re-introduced ("lifted") into the problem. ABC_\text{OPT} then adds the five inequalities to the LP relaxation of \((KP^*)\) and reoptimizes the resulting linear program. The corresponding optimal solution is given by \(x_1 = x_5 = 0, x_2 = 1, x_3 = x_4 = \frac{1}{3}, x_6 = x_7 = x_8 = \frac{1}{3}\) with an objective function value of 3899.4. In view of the "small" gain in the objective function value, ABC_\text{OPT} does not return to the fixing routines (which would then call the preprocessor routines, etc.) but rather ABC_\text{OPT} "stays" in the constraint generation procedure. This procedure finds the two additional inequalities:

\[
\begin{align*}
& x_1 + x_2 + x_3 + x_4 + 0x_5 + x_6 + x_7 + x_8 \leq 3 \\
& x_1 + 2x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 \leq 4
\end{align*}
\]

which chop off the last LP optimum. ABC_\text{OPT} reoptimizes the resulting linear program and terminates with the optimal solution \(x_1 = x_2 = x_3 = x_4 = 0, x_5 = x_6 = x_7 = x_8 = 1\) which has an objective function value of 3896. Since the solution is zero-one valued, it is an optimal solution to \((KP^*)\) and thus for the initial problem \((KP)\). Incidentally, all of the inequalities found by ABC_\text{OPT}
define facets of the associated knapsack polytope in this case. In general, this need not be the case, however, since our calculation of the lifting coefficients is approximate—for the sake of speed, evidently. The entire calculations for the above consume less than 0.5 seconds on a VAX8800.

ACKNOWLEDGMENTS

This research was supported in part by grants from the Office of Naval Research (N00014-87-K-0141) and the National Science Foundation (ECS-8615438).

REFERENCES