

Solving Large-Scale Zero-One Linear Programming Problems

Author(s): Harlan Crowder, Ellis L. Johnson and Manfred Padberg

Source: *Operations Research*, Sep. – Oct., 1983, Vol. 31, No. 5 (Sep. – Oct., 1983), pp. 803–834

Published by: INFORMS

Stable URL: <https://www.jstor.org/stable/170888>

**REFERENCES**

Linked references are available on JSTOR for this article:

[https://www.jstor.org/stable/170888?seq=1&cid=pdf-reference#references\\_tab\\_contents](https://www.jstor.org/stable/170888?seq=1&cid=pdf-reference#references_tab_contents)

You may need to log in to JSTOR to access the linked references.

---

JSTOR is a not-for-profit service that helps scholars, researchers, and students discover, use, and build upon a wide range of content in a trusted digital archive. We use information technology and tools to increase productivity and facilitate new forms of scholarship. For more information about JSTOR, please contact [support@jstor.org](mailto:support@jstor.org).

Your use of the JSTOR archive indicates your acceptance of the Terms & Conditions of Use, available at <https://about.jstor.org/terms>



INFORMS is collaborating with JSTOR to digitize, preserve and extend access to *Operations Research*

JSTOR

# Solving Large-Scale Zero-One Linear Programming Problems

HARLAN CROWDER AND ELLIS L. JOHNSON

*IBM Thomas J. Watson Research Center, Yorktown Heights, New York*

MANFRED PADBERG

*New York University, New York, New York*

(Received July 1981; accepted October 1982)

In this paper we report on the *solution to optimality* of 10 large-scale zero-one linear programming problems. All problem data come from real-world industrial applications and are characterized by *sparse* constraint matrices with rational data. About half of the sample problems have no apparent special structure; the remainder show structural characteristics that our computational procedures do not exploit directly. By today's standards, our methodology produced impressive computational results, particularly on sparse problems having no apparent special structure. The computational results on problems with up to 2750 variables strongly confirm our hypothesis that a combination of problem preprocessing, cutting planes, and clever branch-and-bound techniques permit the optimization of sparse large-scale zero-one linear programming problems, even those with no apparent special structure, in reasonable computation times. Our results indicate that cutting-planes related to the facets of the underlying polytope are an indispensable tool for the exact solution of this class of problem. To arrive at these conclusions, we designed an experimental computer system PIPX that uses the IBM linear programming system MPSX/370 and the IBM integer programming system MIP/370 as building blocks. The entire system is automatic and requires no manual intervention.

---

THE MANAGEMENT science and operations research literature abounds with problems that can be appropriately formulated using zero-one variables that must satisfy certain linear constraints. Researchers and practitioners generally assume, however, that—while useful for the analysis of real-world problem situations—zero-one problems of larger-scale are not amenable to solutions by exact methods. This sentiment is in part due to the fact that the general zero-one problem with linear constraints and linear objective function is among the NP-complete or “hard” combinatorial optimization problems for which, to date, no technically good algorithms are available. In this paper we report on

*Subject classification:* 625 problem preprocessing, brand and bound, cutting planes, 628 facets of zero-one linear programs.

803

**Operations Research**

Vol. 31, No. 5, September–October 1983

0030-364X/83/3105-0803 \$01.25

© 1983 Operations Research Society of America

the *solution to optimality* of 10 large-scale zero-one linear programming problems. Two problems in our study have 33 and 40 zero-one variables, respectively; the remaining eight problems have between 200 and 2750 zero-one variables. All problem data come from real-world industrial applications and were given to us from various sources within and outside of the IBM Corporation. All problems are characterized by *sparse* constraint matrices with rational data. About half of the sample problems have no apparent special structure; the remainder show structural characteristics that are not exploited directly by our computational procedures. While we have no comparative data on previous solution attempts, most of the 10 problems were originally considered not amenable to exact solution in economically feasible solution times. For all these test problems, we have computed a zero-one solution *and* proved it to be optimal with respect to the given linear objective function.

The test problems were solved on an IBM 370/168 at the Thomas J. Watson Research Center, Yorktown Heights, New York. All test problems were solved in less than 1 hour of CPU time; indeed, most problems were solved in less than 10 CPU minutes. The experimental software package PIPX that we designed to solve pure zero-one programming problems uses the IBM linear programming software system MPSX/370 (IBM [1979]) and the branch-and-bound integer programming package MIP/370 (IBM [1975]) as “building blocks.” (It goes without saying that *any comparable* commercial or noncommercial software packages for linear and integer programming could be used in lieu of MPSX/370 and MIP/370.)

The methodology incorporated in PIPX includes automatic problem preprocessing and constraint generation. Essentially, problem preprocessing “inspects” the user-supplied formulation of a zero-one linear program and improves upon the associated linear programming formulation by “tightening” the constraint set, by spotting variables that can be fixed at either zero or one, and by determining constraints of the problem that are rendered inactive by the previous data manipulations. Constraint generation essentially generates cutting-planes that are satisfied by zero-one solutions of the problem and that chop off part of the feasible set of the linear programming relaxation. The cutting planes that we use, however, are not the traditional cutting planes found in textbooks on integer programming; rather, we have implemented recent theoretical results regarding the facial structure of zero-one polytopes; see Padberg [1979] for a related survey and for detailed references. In addition, we have implemented a branch-and-bound strategy for MIP/370 that is designed to quickly find good integer solutions. This procedure is, in most cases, used repeatedly and utilizes information contained in the reduced costs associated with the optimal solution of the linear

programming relaxation to fix variables to zero or one. The PIPX computer system is automatic and requires no manual intervention.

The objective of this computational study is to establish the usefulness of these methodological advances—when combined with clever branch-and-bound strategies—for the automatic solution of sparse large-scale zero-one linear programming problems and, most of all, to demonstrate that today's technology permits one to solve such problems, even when they have no apparent special structure, by *exact* methods in *economically feasible* solution times. The results of this study support this hypothesis and thus agree with the results of a similar study on the symmetric traveling salesman problem (see Crowder and Padberg [1980]) where we made substantial use of the special structure of the problem and arrived at the same conclusion for this class of particularly hard combinatorial optimization problems.

The zero-one programming problems that we consider here have the following form:

$$(P) \quad \text{minimize } \{cx \mid Ax \leq b, x_j = 0 \text{ or } 1 \text{ for } j = 1, \dots, n\}$$

where  $A$  is an  $m$ -by- $n$  matrix with arbitrary rational entries, and  $b$  and  $c$  are vectors of length  $m$  and  $n$ , respectively, with rational entries. In all test problems, the costs  $c_j$  are nonnegative numbers. The problems that we are interested in have a *sparse* constraint matrix  $A$ , i.e., the total number of nonzero elements  $a_{ij}$  of  $A$  divided by the product of  $m$  and  $n$  is typically less than 0.05. Furthermore, while  $A$  is permitted to have rows with entries equal to +1, -1 and 0 only—such as, for example, given by special ordered set constraints—the problems that we address here primarily have a substantial number of rows which the nonzero elements of  $A$  have a significant variance within each row. Indeed, if  $A$  has only +1, -1 and 0 entries, or if all but a few rows of  $A$  have this property, then neither problem preprocessing nor constraint generation as discussed in this study can be expected to be very effective. This, indeed, is the case for several of our test problems having this property. However, as far as constraint generation is concerned, the recent theoretical results concerning the facial structure of the polytopes associated with such problems, or with zero-one problems having a *dense* constraint matrix, suggest quite different methods of constraint generation than the ones that we have incorporated in our software package PIPX. Our numerical results show that for such problems, improvements in the branch-and-bound procedure can offer substantial computational gains.

In Section 1 we discuss the details of the problem preprocessing routine and indicate some promising avenues for extending PIPX. In Section 2 we describe our constraint generation procedure and state some interesting theoretical problems that our computational work suggests and that,

to date, have not yet been resolved rigorously. The constraint generation procedure is illustrated by a numerical example in the Appendix to this paper. In Section 3 we describe the special features of the branch-and-bound strategy that we used for MIP/370 and discuss how we use the reduced costs of the linear programming relaxation to speed up the branch-and-bound phase. In Section 4 we describe how we integrated these various components to form the PIPX experimental computer system for the solution of pure zero-one programming problems. Section 5 describes our computational experiments in detail. Finally, in Section 6, we draw some general conclusions from our computational work.

## 1. PROBLEM PREPROCESSING

Many researchers have suggested that it is desirable to automatically make *ad hoc* improvements to formulations of integer programming problems prior to using linear programming-based solution methods. To obtain improved formulations, various possible strategies can be pursued, ranging from complicated search procedures to rather straightforward, but nevertheless, computationally effective methods. In our computer program for the solution of pure zero-one programming problems we have incorporated only the more obvious possibilities that we describe briefly below. The fact that our uncluttered, but effective, procedure produced favorable results indicates that the topic of problem preprocessing deserves further attention.

### 1.1. Constraint Classification

When the data of problem (P), described in the introduction, are read into the computer, we first classify the inequalities into two types: type 1 constraints are *special ordered set constraints*, i.e., constraints of the type

$$\sum_{j \in L} x_j - \sum_{j \in H} x_j \leq 1 - |H| \quad (1.1)$$

where  $L$  and  $H$  are disjoint index sets and  $|H|$  denotes the cardinality of the set  $H$ . Clearly  $x_j = 1$  for some  $j \in L$  implies  $x_k = 0$  for all  $k \in L$ ,  $k \neq j$ , and  $x_k = 1$  for all  $k \in H$ ; while  $x_j = 0$  for some  $j \in H$  implies  $x_k = 1$  for all  $k \in H$ ,  $k \neq j$ , and  $x_k = 0$  for all  $k \in L$ . In the present form of our computer program, we do not require that two distinct special ordered set constraints be disjoint. Note that our definition of special ordered set constraints is less general than the definition given by IBM [1975]. Type 2 constraints are all other constraints of problem (P).

### 1.2. Variable Fixing and Blatant Infeasibility Check

Suppose, for notational simplicity, that we write a type 2 constraint of

problem (P) as

$$\sum_{j \in N_+} a_j x_j + \sum_{j \in N_-} a_j x_j \leq a_0 \quad (1.2)$$

where  $N_+$  denotes the index set of coefficients  $a_j$  with positive value and  $N_-$  denotes the index set of coefficients  $a_j$  with negative value. Clearly, if

$$\sum_{j \in N_-} a_j > a_0 \quad (1.3)$$

holds, then constraint (1.2) does not have a feasible solution and the overall problem (P), of which (1.2) is but one constraint, is *blatantly infeasible*. On the other hand, if

$$\sum_{j \in N_+} a_j \leq a_0 \quad (1.4)$$

holds, then constraint (1.2) is *inactive* because every possible zero-one vector  $x$  satisfies it. Such an inequality can be dropped from the constraint set of (P) because it does not exclude any zero-one solution. Let  $j \in N_+$  and suppose that

$$a_j > a_0 - \sum_{k \in N_-} a_k \quad (1.5)$$

holds, then  $x_j = 0$  in every feasible zero-one solution to (1.2) and we can fix variable  $x_j$  at the value 0 and drop it from problem (P). Likewise, if for some  $j \in N_-$  we have

$$-a_j > a_0 - \sum_{k \in N_-} a_k \quad (1.6)$$

then  $x_j = 1$  holds in every feasible zero-one solution to (1.2). We can fix variable  $x_j$  at value 1, adjust the right-hand-side vector  $b$  of (P) and drop the variable  $x_j$  from problem (P). If a variable that is fixed at value 1 also appears in a type 1 constraint with a positive coefficient, the remaining variables in this special ordered set are fixed as discussed in Section 1.1; a similar argument holds if a variable that is fixed at value 0 appears also in a type 1 constraint with a negative coefficient. All type 2 constraints of problem (P) are examined one at a time in the order they appear in the formulation.

### 1.3. Coefficient Reduction

Consider an arbitrary linear inequality in the form

$$\sum_{j=1}^p a_j x_j \geq a_0 \quad (1.7)$$

where all  $a_j$  for  $j = 1, \dots, p$  are *positive*. If we have  $a_k > a_0$  for some  $k \in \{1, \dots, p\}$ , then we can replace  $a_k$  by  $a_0$  and the inequality

$$a_0 x_k + \sum_{j=1, j \neq k}^p a_j x_j \geq a_0 \quad (1.8)$$

has the same solution set in terms of zero-one solutions as (1.7), but

fewer real solutions in the unit-hypercube. Thus (1.8) is a “tighter” inequality than (1.7) for the associated linear programming relaxation. Of course, the constraints of (P) are not always of the form (1.7), but using the substitution  $x_j' = 1 - x_j$  where necessary, we can bring every constraint of (P) into this form, apply this reasoning and check each coefficient of each type 2 constraint for a possible coefficient reduction. Then the transformation is reversed and the changed coefficients are recorded. The details of this procedure as they apply to problem (P) are easy to derive and are omitted here. The computer implementation is straightforward and the procedure has proved extremely efficient on real-world problems when the user employed the various “Big  $M$ ” formulation devices that can be found—typically without any discussion of the impact of the choice of magnitude of  $M$  upon computation—in standard texts on integer programming.

#### 1.4. Further Possible Enhancements

Clearly, the coefficient reduction described above can be improved by taking into account special ordered set constraints. While such an extension is not difficult to implement, it is not incorporated in our present software system. The fixing of variables as described in Section 1.2 can be extended to detect, for example, constraints of the form

$$x_i + x_k \leq 1 \quad (1.9)$$

which could be extremely useful in limiting and guiding the search for zero-one solutions in the branch-and-bound phase. These inequalities are called *degree-two constraints*; see, e.g., the survey article by Spielberg [1979]. Johnson and Padberg [1982] study systems of inequalities in zero-one variables with exactly two nonzero coefficients per inequality from a theoretical point of view and derive strong *clique-inequalities* that can be used in computation. Furthermore, there is no reason to limit oneself to the derivation of such implications from single inequalities only; for instance, Hammer et al. [1975] derive implications of the form (1.9) from all pairs of the original constraints. It is, however, clear that any additional work to *preprocess* a given problem of the form (P) must be weighed against the possible benefits that are obtained in terms of an improved linear programming formulation of the problem and against the possible benefits that are obtained of guiding and limiting the search for a zero-one solution in the branch-and-bound phase of a solution procedure. While an implementation of the latter requires more flexible branch-and-bound codes than exist today—at least as regards commercially available codes for branch-and-bound—some of these possible benefits can be reaped as well by the generation of cutting planes that we discuss in the next section.



## 2. CONSTRAINT GENERATION

Considerable research activity in the past 10 years has focused on the facial structure of zero-one polytopes; see Padberg for a related survey article and further references. Briefly, the related studies are concerned with valid inequalities for various zero-one linear programming problems that are *best possible* in the sense that they are required in any representation of the convexified solution set of the problem under consideration by way of linear inequalities. While a theoretically satisfactory and computationally useful characterization of *all* linear inequalities that describe the convexified solution set of the constraints of problem (P) will probably prove to be elusive, partial results in that direction have been obtained to date and, as the experimental results of this paper demonstrate, these partial results do an extremely useful job in the numerical solution of problems with a sparse matrix  $A$  and no apparent special structure.

### 2.1. Sparse Zero-One Problems and Knapsack Problems

The zero-one problem (P) with a single linear constraint, i.e., the case where  $m = 1$ , is called the *knapsack problem* and the facial structure of the associated polytope has been thoroughly studied, even though a complete list of the linear inequalities that define the knapsack polytope remains unknown. Let  $(a^i, b_i)$  denote the  $i$ th row of  $(A, b)$  and assume for simplicity that all constraints of (P) are type 2 constraints. Let

$$P_I^i = \text{conv}\{x \in \mathcal{R}^n \mid a^i x \leq b_i, x_j = 0 \text{ or } 1 \text{ for } j = 1, \dots, n\} \quad (2.1)$$

denote the convex hull (i.e., the convexified solution set; in short: *conv*) of the zero-one solutions to the single inequality  $a^i x \leq b_i$  where  $i \in \{1, \dots, m\}$ .  $P_I^i$  is the knapsack polytope associated with constraint  $i$  of problem (P). Likewise, we let

$$P_I = \text{conv}\{x \in \mathcal{R}^n \mid Ax \leq b, x_j = 0 \text{ or } 1 \text{ for } j = 1, \dots, n\} \quad (2.2)$$

denote the convex hull of zero-one solutions to the entire constraint set of problem (P).  $P_I$  is the zero-one polytope associated with problem (P) and clearly we have

$$P_I \subseteq \bigcap_{i=1}^m P_I^i \quad (2.3)$$

i.e.,  $P_I$  is contained in the intersection of all of the knapsack polytopes  $P_I^i$ ,  $i = 1, \dots, m$ . Equality in (2.3) does *not*, in general, hold, but *does* hold if, for example, problem (P) decomposes totally into  $m$  knapsack problems. This is the case if every variable  $x_j$  appears in exactly one of the  $m$  constraints  $a^i x \leq b_i$  for  $i = 1, \dots, m$ . If we have a large-scale zero-one programming problem with a *sparse* matrix  $A$  and with *no apparent*



*special structure*, it is reasonable to expect that the intersection of the  $m$  knapsack polytopes  $P_I^j$  provides a fairly good approximation to the zero-one polytope  $P_I$  over which we wish to minimize a linear objective function. This is our working hypothesis and our computational results confirm that it is a reasonable assumption. Practically speaking, this assumption permits us to concentrate on the individual rows of the constraint set of problem (P) when we derive valid inequalities for the polytope  $P_I$ . While the proceeding is clearly *valid* for any zero-one problem (P), it cannot be expected to lead to substantial computational gains if, for instance, the matrix  $A$  is rather dense or if almost all entries in  $A$  equal +1, -1 and 0. In these two cases the constraint generation must be done differently; see Padberg. In the latter case, e.g., if *all* components of a row  $a^i$  of  $A$  equal +1, -1 and 0, then we know that

$$P_I^j = \{x \in \mathcal{R}^n \mid a^i x \leq b_i, 0 \leq x_j \leq 1 \text{ for } j = 1, \dots, n\} \quad (2.4)$$

holds, provided that  $b_i$  is integer, and no further improvement in terms of approximating  $P_I$  by way of linear inequalities can be obtained from using such a row *individually*. On the other hand, if the matrix  $A$  is *dense*, then the different rows of  $A$  interact and cutting planes from individual rows of  $A$ , while certainly valid and in some instances useful, cannot be expected to produce the same impressive results that we get for sparse large-scale zero-one problems with no apparent special structure.

## 2.2. Facets of the Knapsack Polytope

We consider a single inequality  $a^i x \leq b_i$  of the constraint system of problem (P) and, using the variable substitution  $x_j = 1 - x_j$  where necessary, we bring the inequality into a form where all nonzero coefficients are positive. Dropping the index  $i$  for notational convenience, we therefore consider a linear inequality

$$\sum_{j \in K} a_j x_j \leq a_0 \quad (2.5)$$

where the coefficients  $a_j$  are positive rational numbers and the variables  $x_j$  assume the values zero or one. Let  $S \subseteq K$  be such that

$$\sum_{j \in S} a_j > a_0 \quad \text{and} \quad \sum_{j \in S} a_j - a_k \leq a_0, \quad \text{for all } k \in S \quad (2.6)$$

hold. Then  $S$  is called a *minimal cover* with respect to (2.5) and obviously every zero-one solution to (2.5) satisfies the inequality

$$\sum_{j \in S} x_j \leq |S| - 1 \quad (2.7)$$

where  $|S|$  denotes the cardinality of the set  $S$ . Suppose, next, that  $S^* \subseteq K$  and  $t \in K - S^*$  satisfy

$$\sum_{j \in S^*} a_j \leq a_0 \text{ and} \quad (2.8)$$

$Q \cup \{t\}$  is a minimal cover for every  $Q \subseteq S^*$  with  $|Q| = k$

where  $k$  is an integer number satisfying  $2 \leq k \leq |S^*|$ . Due to the one-element role of the index  $t$  and because  $k$  is some integer number, the set  $S^* \cup \{t\}$  is called a  $(1, k)$ -configuration with respect to (2.5). Every zero-one solution to (2.5) satisfies the following set of inequalities that are associated with a  $(1, k)$ -configuration:

$$(r - k + 1)x_t + \sum_{j \in T(r)} x_j \leq r \quad (2.9)$$

where  $T(r) \subseteq S^*$  varies over all subsets of cardinality  $r$  of  $S^*$  and  $r$  varies over all integers from  $k$  to  $|S^*|$ , inclusively. If  $k = |S^*|$  holds in (2.8), then a  $(1, k)$ -configuration is a minimal cover; thus, in general, the class of inequalities associated with  $(1, k)$ -configurations properly contains the class of inequalities associated with minimal covers. Both inequalities (2.7) and (2.9) are *best possible* ones if  $K = S$  holds in the first case, or if  $K = S^* \cup \{t\}$  holds in the second case, i.e., in these cases the respective inequalities define *facets* of the associated knapsack polytope; see Padberg. In general, however, these inequalities must be “lifted” to obtain facets of the knapsack polytope associated with (2.5), i.e., they must be extended appropriately to the variables  $x_j$  with index  $j$  in  $K - S$ , or with index  $j$  in  $K - S^* - \{t\}$ , respectively.

The extension is done over the respective variables by the following recursive *lifting procedure*: Initially we set  $f_j = 1$  for all  $j \in S$ ,  $f_0 = |S| - 1$  or  $f_j = 1$  for all  $j \in S^*$ ,  $f_t = (r - k + 1)$ ,  $f_0 = r$  and  $S = S^* \cup \{t\}$ , respectively. For the iterative step, let  $k \in K - S$  and determine

$$\begin{aligned} z_k &= \max\{\sum_{j \in S} f_j x_j \mid \sum_{j \in S} a_j x_j \leq a_0 - a_k, \\ &\quad x_j = 0 \text{ or } 1 \text{ for all } j \in S\}. \end{aligned} \quad (2.10)$$

Define  $f_k = f_0 - z_k$ , redefine  $S$  to be  $S \cup \{k\}$ , and repeat until  $K - S$  is empty. The resulting inequality  $fx \leq f_0$  defines a facet for the polytope  $P_j^i$  associated with (2.5). Obviously, the lifting procedure requires the solution of several zero-one problems. In the next section we discuss how to identify suitable minimal cover inequalities and return later to the question of how to efficiently approximate the lifting procedure.

We note that the *support* of an inequality obtained by lifting (2.7) or (2.9) is contained in the support of the inequality (2.5), i.e.,  $f_j \neq 0$  holds only if  $a_j \neq 0$  holds. Therefore, the inequalities that we generate preserve the sparsity of the constraint matrix. This is an additional difference between our methods and the traditional cutting planes described in the textbooks on integer programming. The traditional cutting planes are typically rather dense and—as integer programming folklore has it—lead to explosive storage requirements. The results in Section 5 show that the constraints that we generate lead to only moderate increases in storage requirements.

### 2.3. Constraint Identification

To use the methods described above in actual computation we could, of course, simply generate *a priori all* possible minimal covers and  $(1,k)$ -configurations for each row of the problem (P), extend the respective inequalities in some fashion as, e.g., discussed above, append them to the linear constraints of (P), and proceed to solve the *augmented* linear programming problem. Clearly this would be wasteful and, for large-scale problems, impossible to implement: The number of possible  $(1,k)$ -configurations for (2.5) is evidently exponential in the number of variables of the constraint (2.5) and thus storage requirements would simply explode. Thus we have no alternative but to generate such constraints on the “fly,” i.e. generate them in the course of computation as they are needed. To this end, we start by solving the linear program

$$\min\{cx \mid Ax \leq b, 0 \leq x_j \leq 1 \text{ for } j = 1, \dots, n\} \quad (2.11)$$

and obtain an optimal solution  $\bar{x}$ . If  $\bar{x}$  is a zero-one solution, we stop:  $\bar{x}$  solves the problem (P). Otherwise, we take any type 2 row of  $A$  and solve the following problem

**CONSTRAINT IDENTIFICATION PROBLEM.** *Given  $\bar{x}$  find a minimal cover inequality (2.7) or a  $(1,k)$ -configuration inequality (2.9) that chops off  $\bar{x}$ , if such an inequality exists.*

The constraint identification problem is solved, in turn, for *each* row of the original constraint matrix  $A$  that qualifies, the resulting inequalities that are identified are “lifted” and appended to the linear programming problem, the augmented problem is reoptimized and the procedure is repeated until we find a zero-one solution, until no more constraint is found, or until the gain in the objective function value becomes too small, e.g., is less than one unit in terms of the units of the objective function. And, of course, when one of the latter two possibilities occur, we resort to branch-and-bound.

The question is then to devise a procedure that solves the constraint identification problem. Using the notation of Section 2.2, consider the zero-one knapsack problem

$$\min\{\sum_{j \in K} (1 - \bar{x}_j)s_j \mid \sum_{j \in K} a_j s_j > a_0, s_j = 0 \text{ or } 1 \text{ for all } j \in K\} \quad (2.12)$$

where the inequality is strict in the knapsack constraint. It follows that there is a minimal cover inequality (2.7) that chops off  $\bar{x}$  if and only if the optimal objective function value of (2.12) is less than one. For suppose that there is a minimal cover  $S \subseteq K$  that chops off  $\bar{x}$ . Setting  $s_j = 1$  for all  $j \in S$ ,  $s_j = 0$  for all  $j \in K - S$  we find that (2.12) has an optimal objective function value less than one. To prove the reverse direction, observe that  $0 \leq \bar{x}_j \leq 1$  for all  $j = 1, \dots, n$  implies that the objective

function coefficients in (2.12) are nonnegative, and hence among the optimal solutions to (2.12), at least one defines a minimal cover for (2.5). Let  $S$  be the set of variables with value 1 in such a solution. If the optimum value of the objective function value of (2.12) is less than 1, then the corresponding inequality (2.7) chops off  $\bar{x}$ . As can easily be verified, problem (2.12) is constructed in such a manner that its solution finds a *most violated* minimum cover inequality.

Our formulation of the constraint identification problem for minimal cover inequalities involves the solution of a zero-one knapsack problem for which, to date, no technically good algorithm is known. We *conjecture* that—using a different approach than the one outlined here—violated minimal cover inequalities can be identified by a polynomially bounded algorithm. For  $(1,k)$ -configurations, we do not, at present, even have a formulation of the constraint identification problem in a tractable form such as (2.12). Rather, we currently use an ad hoc procedure for finding  $(1,k)$ -configurations that can be improved upon.

## 2.4. Implementation of Constraint Identification

To implement the theoretical development outlined above, we have written a FORTRAN program that does the constraint generation. Given the current solution  $\bar{x}$ , we scan each type 2 constraint and set up auxiliary data structures to minimize the actual computational effort. First, using the substitution  $x_j' = 1 - x_j$ , where necessary, we bring the row under consideration into the form (2.5) with all nonnegative coefficients. This substitution changes the solution vector from  $\bar{x}$  to, say,  $\bar{y}$ . Next, define

$$K_1 = \{j \in N \mid \bar{y}_j > 0, a_{ij} \neq 0\}, K_0 = \{j \in N \mid \bar{y}_j = 0, a_{ij} \neq 0\} \quad (2.13)$$

where  $N = \{1, \dots, n\}$  and  $i$  is the index of the current type 2 constraint. Clearly to solve the constraint identification problem, it is sufficient to work only with the data in  $K_1$ , which is a substantial advantage since  $|K_1|$  is usually small for sparse problems. If  $K_1$  is empty, if  $j \in K_1$  implies  $\bar{y}_j = 1$ , or if  $j \in K_1$  implies  $|a_{ij}| = 1$ , then there is nothing to do in row  $i$  and we process the next type 2 constraint. Otherwise, we solve problem (2.12) with  $K$  replaced by  $K_1$  and  $\bar{x}$  replaced by  $\bar{y}$ , respectively, as a *linear program*, i.e., we replace the constraints  $s_j = 0$  or  $1$  by  $0 \leq s_j \leq 1$  for all  $j \in K_1$ . This problem is a linear knapsack problem which is solved by Dantzig's method [1963] (p. 517). Rounding the fractional variable to 1, we obtain a cover for the knapsack which we proceed to change into a minimal cover by dropping some of the variables to zero. We call  $S$  the index set of the variables in the minimal cover.

Having obtained a minimal cover  $S$ , we proceed to sort the variables in  $S$  by increasing order of magnitude of the coefficients  $|a_{ij}|$ , i.e., by

decreasing ratio  $1/|a_{ij}|$ . This sorting is done in order to have a start for the lifting procedure. If  $K_1 - S$  is nonempty, we extend the minimal cover inequality (2.7) to the variables in  $K_1 - S$  using the lifting procedure described in Section 2.2. As in the previous case, we relax the zero-one problem (2.10) to its associated linear program, i.e., we replace  $x_j = 0$  or 1 by  $0 \leq x_j \leq 1$  for all  $j \in S$ . Again, this yields a linear knapsack problem that we solve by Dantzig's method. Let  $\bar{z}_k$  denote the optimal objective function value of this linear program and let  $z_k^*$  denote the value obtained from  $\bar{z}_k$  by truncating it to its integer part. Clearly  $z_k$ , the optimal objective function of (2.10), satisfies  $z_k \leq z_k^*$ . We define  $f_k$  to be  $f_0 - z_k^*$ . Because  $z_k \leq z_k^*$  holds, we *underestimate* the "true" extension coefficient and thus the extension is valid because of the nonnegativity of the constraint. The possible error that we make is small since the coefficients of minimal cover inequalities are all less than  $|S|$  and, typically,  $|S|$  is small. If the resulting extension coefficient  $f_k$  is positive, we merge variable  $k$  into the list of variables according to its ratio  $f_k/|a_{ik}|$  and iterate until all variables in  $K_1 - S$  are processed. If  $f_k$  equals zero, the variable  $k$  need no longer be considered in the lifting procedure. Note that the iterative step of the lifting procedure can be executed very quickly because the "next" step continues to use the ordered list of variables of the previous step in the Dantzig solution to the linear program associated with (2.10).

After having processed all variables in  $K_1 - S$ , or if  $K_1 - S$  is empty, we check whether or not the resulting (extended) minimal cover inequality chops off  $\bar{y}$ . If this is not the case, we try to find a  $(1,k)$ -configuration; see the discussion to follow. Otherwise, we extend the inequality to the variables with index in  $K_0$  using the lifting procedure in the relaxed form as described previously. After having obtained a violated inequality, we store it and proceed with the next available type 2 constraint.

The identification of  $(1,k)$ -configurations in our present implementation is preliminary at best and goes as follows: We start with a minimal cover  $S$  identified earlier and define the index  $t$  of the  $(1,k)$ -configuration to be the index with the largest  $|a_{ij}|$  for all  $j \in S$ . If this does not yield a unique index, we consider the next type 2 constraint. Otherwise, we set  $S^* = S - \{t\}$  and  $k = |S| - 1$ . (Note that (2.8) is satisfied.) We scan the indices in  $K_1 - S^* - \{t\}$  one by one. Let  $i \in K_1 - S^* - \{t\}$  be the current index, and check whether or not (2.8) is valid with  $S^*$  replaced by  $S^* \cup \{i\}$ . If not, we scan the next index. If yes, we replace  $S^*$  by  $S^* \cup \{i\}$ . We check whether or not the inequality (2.9) with  $r = |S^*|$  chops off the solution  $\bar{y}$ . If not, we scan the next index. If yes, we save  $S^*$  for further use and extend the inequality found to the other variables in  $K_1 - S^* - \{t\}$  using the lifting procedure described above. After extending the inequality to the variables in  $K_0$  as well, we store the resulting constraint. Then we iterate by scanning the next index. After scanning all indices,

we process the next type 2 constraint. Note that while  $t$  and  $k$  remain fixed in this computation, the size of  $S^*$  generally increases. For each value of  $|S^*|$ , however, only one violated inequality is generated.

The order of computations thus always involves as a first step the identification of a minimal cover. If successful, we take the next type 2 constraint. If not successful, we look for a  $(1,k)$ -configuration, and so on. At the end of the constraint generation phase, we thus have a list of violated inequalities from minimal covers and/or  $(1,k)$ -configurations from the individual rows of the problem (P). These new constraints are then added to the constraint set of the linear programming problem. The constraints generated in the course of this computation are not used for further constraint generation; they are simply appended to the linear program and not further classified.

Special ordered set constraints (type 1) are not fully exploited in our current implementation. At present, we use them in the following fashion: When setting up a type 2 constraint for constraint generation, we check whether or not the “first” variable with a nonzero coefficient is contained in some special ordered set (SOS), with “first” being defined relative to the numbering order used. If so, we check whether or not the current solution  $\bar{x}$  satisfies the corresponding SOS as an equality. If not, or if the variable is not contained in any SOS, we check the next variable in the numbering order. Otherwise we determine the smallest coefficient in the overlap of the SOS and the current constraint having a positive  $\bar{x}$  component, take its absolute value and add the SOS multiplied by this number to the current constraint. We flag all variables of the current type 2 constraint that are contained in this SOS constraint as having been processed, and continue the process with the next unflagged variable of the *original* type 2 constraint. It is possible that we create new nonzeros in the constraint this way and this is taken into consideration. In other words, we form a *surrogate constraint* from the original type 2 constraint and a nonoverlapping subset of SOS constraints that are tight at the current solution. This surrogate constraint is then used for constraint generation in lieu of the original one. Evidently, one can do better than forming a surrogate constraint and in a recent paper Johnson and Padberg [1981] have generalized the Dantzig method to solve linear knapsack problems with SOS constraints. Also, Johnson and Padberg studied some properties of the associated zero-one polytope. These results were not available when we began to work on this study and thus have not been incorporated in our implementation.

### 3. BRANCH-AND-BOUND METHODS

We now describe how we modified the standard branch-and-bound algorithm of MIP/370 and applied it to the augmented zero-one problem



described in Section 2.3. In most cases, the augmented problem will have the desired characteristic that the gap between the continuous linear program optimal objective function value and the optimal zero-one objective function value is relatively small, thereby making the branch-and-bound search for an optimal integer solution easier. In order to further facilitate the search, we make the following modifications and enhancements to the standard MIP/370 algorithm:

- The computation of an *ad hoc* upper bound on the optimal zero-one solution.
- The PIP strategy for quickly finding integer solutions.
- The use of continuous reduced cost implications to fix variables.

### 3.1. The Ad Hoc Upper Bound

Since at present we have no reasonable heuristic for general large-scale zero-one linear programs that could be used to determine a true upper bound for the optimal zero-one solution, we compute an *ad hoc* bound for the zero-one optimum from the objective function value of the augmented linear programming relaxation. This bound is assigned to the MIP/370 variable *XMxDROP*, which is the value of the objective function beyond which a branch or node of the branch-and-bound tree is dropped forever; see IBM [1975]. The value for the computed *ad hoc* bound is

$$\text{XMxDROP} = \text{XPCTGAP} * \text{XFUNCT} \quad (3.1)$$

where *XFUNCT* is the value of the optimal continuous linear program objective function and *XPCTGAP* is a parameter with value in the range 1.1 to 1.25.

The assumption underlying this rather simple bounding mechanism is that the gap between the augmented continuous linear program optimal objective function value and the optimal zero-one objective function value is small. Furthermore, if, for a particular problem, the optimal zero-one solution is not within 10 to 25% of the continuous solution, then presumably the preprocessing and constraint generation procedures have not been effective for the problem at hand, and we have no desire to exhaust the branch-and-bound tree by MIP/370 processing for a truly large-scale problem. Should our assumption be wrong, then we can at least expect to complete the run for the problem at hand with the additional useful information that the optimal zero-one objective function value is greater than our computed bound.

As indicated in the computational experiments in Section 5, for those problems for which the preprocessing and constraint generation procedures were effective, the aforementioned objective function gap always fell within the above limit and (3.1) proved to be a valid bounding



mechanism. Indeed, in all of our test problems, it sufficed to choose XPCTGAP equal to 1.1.

### 3.2. The PIP Strategy

The PIP branch-and-bound strategy is a way of guiding MIP/370 to quickly find integer solutions. The primary assumption of the PIP strategy is that, as previously mentioned, the gap between the optimal continuous linear program solution value and the optimal zero-one solution value is relatively small, i.e., that the *augmented* linear program provides a fairly good approximation to the zero-one polytope in the neighborhood of an optimal zero-one solution.

To understand the PIP strategy, we must introduce some MIP/370 terminology. (This discussion is applicable to minimization problems.) The following MIP/370 variables, all of which contain floating-point numeric values, are related, as indicated, to how MIP/370 searches for integer solutions:

- MXMDROP: As indicated previously, any node or branch whose objective function value exceeds MXMDROP is abandoned forever in the search.
- MXMBESTF: At any time during the search, MXMBESTF contains the best (minimal) functional value of the waiting nodes. Therefore, the best integer solution that can still be expected cannot have a functional value less than MXMBESTF.
- MXMSCAN: A node in the branch-and-bound tree is *postponed* (it will not be considered for branching) if its objective function value is greater than MXMBESTF + MXMSCAN. Note: a node postponed as a branching candidate because of this rule may later become eligible if MXMSCAN or MXMBESTF is increased.

Figure 1 illustrates the relationship between MXMDROP, MXMBESTF and MXMSCAN.

Areas A, B and C in Figure 1 show the possible positioning of the objective function values of nodes in the branch-and-bound tree with respect to MXMDROP, MXMBESTF and MXMSCAN. Nodes in area A are eligible candidates for branching; nodes in area B are postponed and are not candidates for branching; area C contains no nodes because nodes and branches with objective function values greater than MXMDROP are dropped from the tree.

The PIP strategy starts with

$$\text{MXMSCAN} = 0.1 * (\text{MXMDROP} - \text{MXMBESTF}) \quad (3.2)$$

In other words, area *A* in Figure 1 is small compared to area *B*; resulting in nodes with very good objective function values being the branch node candidates. *XMxBESTF* is gradually increased until every waiting node is a candidate for branching. That is, area *A* in Figure 1 includes all nodes between *XMxBESTF* and *XMxDROP* and area *B* is empty. At the same time that *XMxBESTF* is increasing, a higher priority is given to nodes based on their integer feasibility; see IBM [1975] as amended by IBM Technical Newsletter SN19-1132, April 30, 1979, p. 146.2. After a predetermined number of nodes have been developed and if no integer solution has been found *XMxBESTF* is reset to the value indicated in (3.2) and the process is repeated.

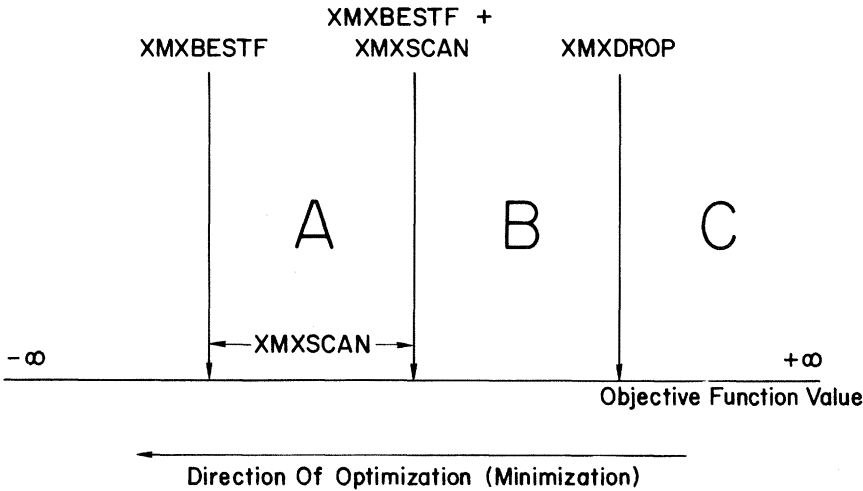


Figure 1. Relationship between MIP/370 variables.

3.3. Continuous Reduced Cost Implications

In a pure zero-one minimization linear programming problem, given the continuous optimal objective function  $Z_{lp}$ , the continuous optimal reduced costs  $d_j$ , and a true upper bound on the zero-one optimal objective function value  $Z^+$ , the following implications hold for variables  $x_j$  which are nonbasic in the continuous optimal solution:

- 1. If  $x_j = 0$  in the continuous solution and  $Z^+ - Z_{lp} < d_j$  then  $x_j = 0$  in every optimal zero-one solution.
- 2. If  $x_j = 1$  in the continuous solution and  $Z^+ - Z_{lp} < -d_j$  then  $x_j = 1$  in every optimal zero-one solution.

In the branch-and-bound routine, the above test is made each time a zero-one solution is obtained and  $Z^+$  is set equal to the objective function

value of the zero-one solution that was found. If more than a fixed percentage, typically 10%, of the current active variables can be fixed, the problem is revised by setting the indicated variables to zero or one, and the procedure is restarted by performing preprocessing and constraint generation on the new problem. In this case, the previously generated additional cutting planes are purged from the computer memory, i.e., we restart from scratch on the *reduced original* problem. We do this because it is easier to implement and because the generation of constraints can be done rather quickly—the generation of cutting planes definitely was not a bottleneck in our computations.

It would seem, however, that it is much more efficient to be able to fix the indicated variables in the current branch-and-bound tree and continue the search, rather than restarting from the beginning with the new problem after revision. Unfortunately, there is no way to perform such a global function in MPSX/370. The fact that we are willing to abandon a search tree and restart the entire procedure with a reduced problem reflects our philosophy that we would rather do almost *anything* than perform branch and bound. Our philosophy is not a prejudice; it is based on our experience solving this type of problem. The excellent computational times reported in Section 5 confirm our strategy.

#### 4. THE PIPX COMPUTER SYSTEM

PIPX is an experimental computer software system, written in PLI, which calls MPSX-MIP/370 procedures and FORTRAN routines to perform the preprocessing, constraint generation, and branch-and-bound functions described in Sections 1, 2 and 3. It is another example of the successful use of the algorithmic and data handling features of a state-of-the-art, commercial linear programming package as a building block for larger application systems; see, e.g., Crowder and Padberg [1980] and Ho and Louie [1981].

PIPX consists of 22 external procedures (6 written in PLI, 5 in FORTRAN, and 11 being MPSX-MIP/370 procedures). Table I summarizes the name, purpose, calling procedure and procedures called for the non-MPSX-MIP/370 procedures in PIPX. For a more detailed description of MPSX-MIP/370 procedures, see IBM [1975, 1979].

PIPX requires two types of data storage: MPSX/370 data storage, used by the computational routines of MPSX-MIP/370, and PIPX internal data storage. The MPSX/370 data storage component is allocated automatically by the SETUP routine and consists mainly of work regions and buffers; see IBM [1979]. PIPX internal data storage is allocated dynamically, based on problem size and storage availability, and includes two sparse representations of the linear programming

problem (row-major order and column-major order schemes) and utility storage used for the problem preprocessing and constraint generation functions.

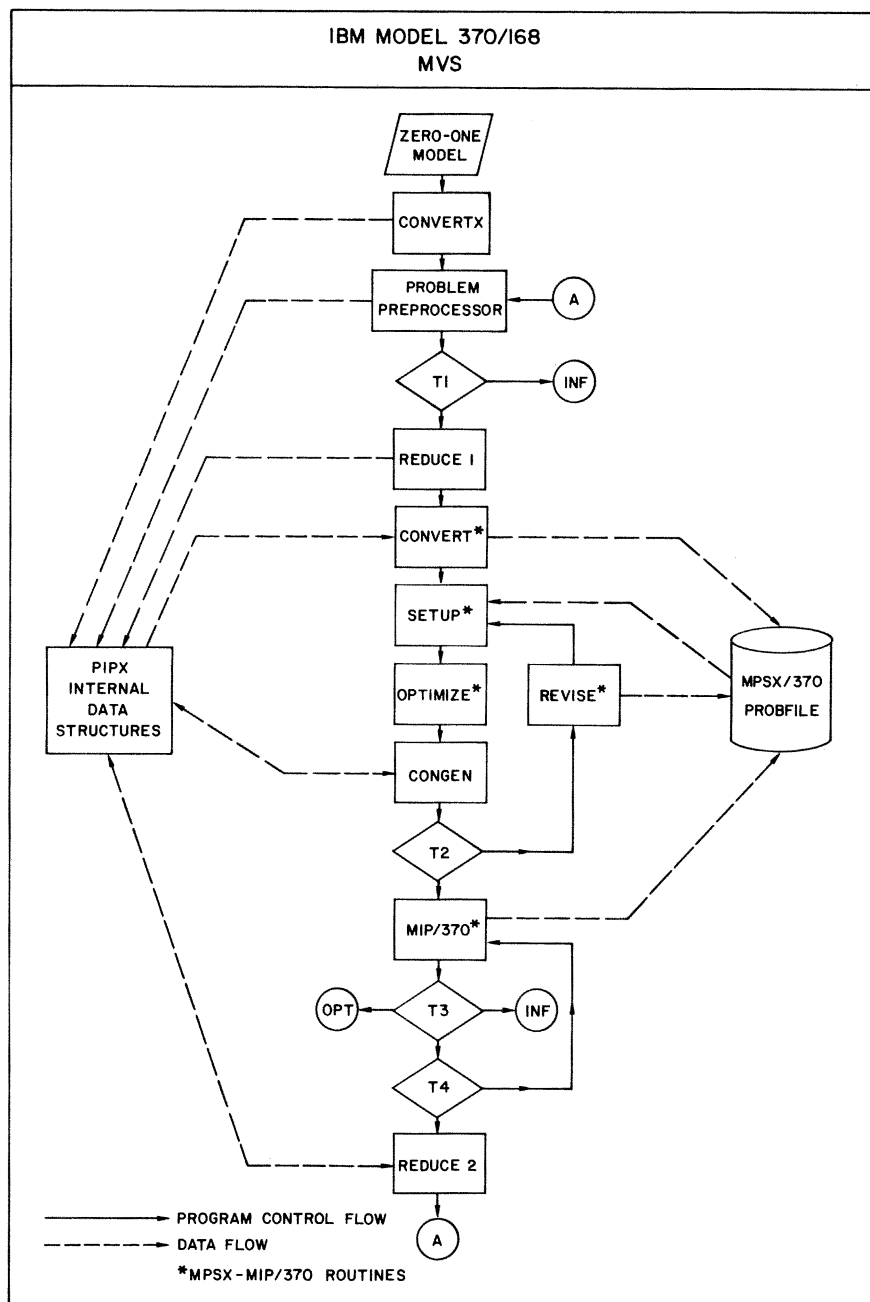
Figure 2 illustrates the overall program control and data flow for PIPX. A more detailed functional description of the system, with emphasis on

TABLE I  
PIPX EXTERNAL PROCEDURES

Name	Purpose	Called by	Calls to procedures in	
			PIPX	MPSX-MIP/370
OPTIPI	main procedure	—	PIPIN CORDER ROWFOR DETSOS REDCOF PIPREC PIPRV PIPCVT PIPCGN	TIME CONVERT SOLUTION SELIST MIXSTART MIXFLOW MIXSTATS
PIPIN	reads the problem in SHARE format from MPSX/370 file SYSIN and sets up intermediate sparse data structures	OPTIPI	—	—
CORDER	called after PIPIN to set up the sparse column representation of the linear programming tableau	OPTIPI PIPREC	—	—
ROWFOR	called after PIPIN to set up the sparse row representation of the linear programming tableau	OPTIPI PIPREC	—	—
DETSOS	SOS constraint determination and infeasibility checking	OPTIPI	—	—
REDCOF	coefficient reduction	OPTIPI	—	—
PIPREC	elimination of inactive rows and fixed variables prior to calling MPSX/370 procedure REVISE	OPTIPI	CORDER ROWFOR	—
PIPRV	computes intra-row variance for active, non-SOS constraints	OPTIPI	—	—
PIPCVT	builds a PLI structure interface to MPSX/370 routine CONVERT	OPTIPI	—	—
PIPCGN	iteratively solves linear programming problems augmented with constraints	OPTIPI	CONGEN	SETUP RESTORE OPTIMIZE SOLUTION REVISE
CONGEN	constraint generation	PIPCGN	—	—

the program control and data manipulation interface to MPSX/370, will appear in a subsequent publication. The following brief functional overview of PIPX serves as an annotation of Figure 2.

1. CONVERT reads the zero-one linear programming problem, in SHARE format (IBM [1979], p. 193), from a data file. The system allocates and initializes the appropriate internal data structures,



**Figure 2.** PIPX control and data flow.

including row-major order and column-major representations of the problem.

2. The PROBLEM PREPROCESSOR uses the internal data structures to perform the functions described in detail in Section 1.
  3. Decision T1 determines if the problem has been found infeasible in the PROBLEM PREPROCESSOR. If so, PIPX terminates.
  4. REDUCE1 updates the internal representation of the problem, eliminating variables that have been fixed and constraints that have become inactive due to the actions of the PROBLEM PREPROCESSOR.
  5. CONVERT, an MPSX/370 procedure, uses the internal representation of the problem to create the MPSX/370 representation on PROBLEM (the problem file used by MPSX/370).
  6. SETUP, an MPSX/370 procedure, reads the problem from PROBLEM and creates the MPSX/370 in-storage data structures.
  7. OPTIMIZE, an MPSX/370 procedure, solves the continuous linear programming problem.
  8. CONGEN then processes the problem and the optimal continuous solution in the manner described in Section 2.4. The actions determined by T2 are either
    - i) Update the problem with the new constraints, using the MPSX/370 procedure REVISE, and restart the sequence from the SETUP step, or
    - ii) Determine that no constraints have been generated, or that the constraints generated do not have a significant effect of the linear programming objective function value; in either case, the constraint generation phase is terminated.
- Note:* In most cases, the SETUP, OPTIMIZE, CONGEN, REVISE sequence is repeated several times, resulting in successive linear programming problems with increasingly greater (in the case of minimization) objective function values; see Section 5.
9. MIP/370 solves the augmented zero-one linear programming problem; Section 3 specifies details of the solution strategy.
  10. Decision T3 is invoked when MIP/370 terminates or when it finds an integer solution. The possible actions are
    - i) PIPX terminates with an optimal solution, or
    - ii) PIPX terminates because the problem is zero-one infeasible, or
    - iii) A nonoptimal integer solution is found in which case decision T4 is invoked. Decision T4 determines if the reduced costs from the continuous optimal solution can be used to fix variables and thus reduce the problem; see Section 3.3. If this reduction is possible, the problem is processed by REDUCE2, and the PIPX program control sequence is restarted from the PROBLEM

PREPROCESSOR step. Otherwise, MIP/370 branch-and-bound processing continues.

## 5. COMPUTATIONAL EXPERIMENTS

We applied the PIPX computer system to a set of real large-scale zero-one linear programming problems. Table II summarizes the main characteristics of the test problem set. The columns headed VARS, ROWS and NONSOS contain the number of variables, the total number of constraints, and the number of those constraints that are *not* of the special ordered set type. The column headed VARIANCE indicates the *average* standard deviation of the absolute values of the nonzero elements of the non-SOS rows of the constraint matrix after preprocessing; the number in parentheses is the standard deviation of this aggregate statis-

TABLE II  
TEST PROBLEM SUMMARY

NAME	VARS	ROWS	NON-SOS	VARIANCE*	DENS	$Z_{lp}$	$Z_{ip}$
P0033	33	16	11	49.6 (33.9)	17.4	2520.6	3089.0
P0040	40	24	13	39.6 (92.8)	11.3	61796.5	62027.0
P0201	201	134	107	0.7 (1.0)	5.0	6875.0	7615.0
P0282	282	242	44	34.3 (16.2)	2.0	176867.5	258411.0
P0291	291	253	14	3.5 (2.3)	1.9	1705.1	5223.75
P0548	548	177	94	123.9 (88.3)	1.8	315.3	8691.0
P1550	1550	94	2	8.4 (8.4)	7.4	1706.5	1708.0
P1939	1939	109	2	3.5 (3.5)	4.8	2051.1	2066.0
P2655	2655	147	2	2.3 (2.3)	3.4	6532.1	6548.0
P2756	2756	756	386	149.2 (89.4)	0.4	2688.7	3124.0

\*After problem preprocessing

tic. Thus, for example, P2756 has 756 rows out of which 386 are non-SOS rows. Computing for each non-SOS row of P2756 the standard deviation of the absolute values of its nonzero elements and averaging over all 386 rows yields an average intra-row standard deviation of 149.2, i.e., a substantial variation of the nonzero coefficients; the number 89.4 indicates that the variation is substantial *among* the non-SOS rows of the constraint matrix, but is somewhat less on average than the variation *within* each non-SOS row. In problems P1550, P1939 and P2655 these figures indicate that only one of the two non-SOS rows contributes to the standard deviation, the other row having identical nonzero elements. Indeed, these problems are set partitioning problems with two additional constraints; one of which constrains the sum of all variables to be a certain number, the other one being a knapsack constraint with positive coefficients which, however, have little variation. The column headed DENS specifies the density of the original constraint matrix plus the



cost row; i.e., the total number of nonzero elements of  $A$  and  $c$  divided by the product of  $m$  and  $n$  and multiplied by 100. Finally,  $Z_{lp}$  denotes the optimal objective function value of the linear programming relaxation of problem (P) in original form, i.e., prior to applying any of the PIPX functions, while  $Z_{ip}$  denotes the optimal zero-one objective function value.

Four of the test problems used in this study are proprietary and cannot be released. The remainder of the problems, as well as other similar problems, will be made available by the authors within a reasonable time limit.

The computational experiments described here were executed on the IBM 370/168 computer, running the MVS operating system, at the Thomas J. Watson Research Center in Yorktown Heights, New York. FORTRAN programs were compiled using the FORTRAN H Extended Compiler. PLI programs were compiled using the PLI Optimizing Compiler.

TABLE III  
PROBLEM PREPROCESSOR SUMMARY

NAME	ORIGINAL PROBLEM		ELIMINATED		TIME*
	ROWS	VARS	ROWS	VARS	
P0033	16	33	0	0	0.01
P0040	24	40	0	0	0.02
P0201	134	201	0	6	0.01
P0282	242	282	20	0	0.02
P0291	253	291	47	1	0.02
P0548	177	548	20	21	0.03
P1550	94	1550	0	0	0.19
P1939	109	1939	0	0	0.18
P2655	147	2655	0	0	0.30
P2756	756	2756	17	22	0.50

\*CPU minutes, IBM 370/168 - MVS

5.1. Zero-One Preprocessing Phase

The zero-one preprocessor is the initial computational phase in the PIPX system. As indicated in Section 1, the effect of the preprocessor is to tighten the user-supplied linear programming formulation of the zero-one problem, to eliminate inactive constraints, and to fix variables at either zero or one. Table III summarizes the results of the initial preprocessor step on the test problem set. The column headed ORIGINAL PROBLEM specifies the number of rows and variables in the problem before preprocessing. The column headed ELIMINATED gives the number of rows and variables that the preprocessor successfully eliminated from the problem. The time for the initial preprocessor phase, in CPU minutes, is given under the TIME column.

In several cases, the zero-one preprocessor was reapplied to a revised problem after variables were eliminated using the continuous reduced cost implications (see Section 2.3). We have omitted the detailed results of any second and subsequent applications of the preprocessor procedure because such data are not crucial to this discussion. Table VII in Section 5.4, however, gives the number of major passes of the PIPX system for each test problem, and hence the number of times the system invoked the zero-one preprocessor.

## 5.2. Constraint Generation Phase

The constraint generation procedure is the second computational phase of the PIPX system; it operates on the preprocessed problem to produce and solve a linear programming problem with a better (greater in the

TABLE IV  
CONSTRAINT GENERATION SUMMARY

NAME	PREPROCESSED PROBLEM				AUGMENTED PROB				TIME*
	VARS	ROWS	DENS	$Z_{lp}^*$	PASS	ROWS	DENS	$Z_{lp}^{**}$	
P0033	33	16	17.4	2819.4	6	36	12.1	3065.3	0.12
P0040	40	24	11.3	61829.1	4	29	13.8	61862.8	0.12
P0201	195	134	5.0	7125.0	1	139	5.0	7125.0	0.16
P0282	282	222	1.5	176867.5	14	462	1.3	255033.1	1.00
P0291	290	206	1.3	1749.9	6	278	2.5	5022.7	0.33
P0548	527	157	1.9	3125.9	9	296	1.3	8643.5	0.57
P1550	1550	94	7.4	1706.5	1	94	7.4	1706.5	0.81
P1939	1939	109	4.8	2051.1	1	110	4.8	2051.1	0.74
P2655	2655	147	3.4	6532.1	2	149	3.4	6535.0	2.04
P2756	2734	739	0.4	2701.1	8	1065	0.4	3115.3	3.06

\*CPU minutes, IBM 370/168 - MVS

minimization case) optimal continuous objective function value (see Section 2). Table IV summarizes the results of the first constraint generation step on the test problem set. The columns headed PRE-PROCESSED PROBLEM specify the number of constraints, the number of variables, the density of the problem, and optimal continuous objective function value  $Z_{lp}^*$  of the problem produced by the preprocessor, before constraint generation. The AUGMENTED PROBLEM section of the table indicates the number of constraint generation "passes," i.e., the number of intermediate linear programming problems required, the number of constraints in the final linear programming problem produced, the density of its constraint matrix including the cost row, and its optimal continuous objective function value  $Z_{lp}^{**}$ . The time for the initial constraint generation phase, in CPU minutes, is given under the heading TIME. This time includes the solution and reoptimization of all linear

programming calculations in the constraint generation phase, including the original linear program.

As in the case of the preprocessing phase, fixing variables using the continuous reduced cost implications in the branch-and-bound phase causes the constraint generation procedure to be reapplied to the revised problem. As before, we omit the details of second and subsequent applications of the constraint generation procedure. The real measure of the effectiveness of the constraint generation procedure is determined by how much it closes the "gap" between the optimal linear programming relaxation objective function value and the optimal zero-one objective function value. Table V summarizes these data for the first application of the constraint generation procedure. The column headed  $\Delta$ ROWS specifies the number of constraints generated by the procedure. The GAP is the difference between the optimal zero-one objective function value

TABLE V  
EFFECT OF CONSTRAINT GENERATION

NAME	$\Delta$ ROWS	GAP	RATIO
P0033	20	269.6	0.92
P0040	5	197.9	0.17
P0201	5	490.0	0.00
P0282	240	81543.5	0.96
P0291	72	3473.8	0.94
P0548	139	5565.1	0.99
P1550	0	1.5	0.00
P1939	1	14.9	0.00
P2655	2	15.9	0.19
P2756	326	422.9	0.98

and the optimal continuous objective function value before constraint generation, but after problem preprocessing, i.e., the GAP is  $Z_{ip} - Z_{lp}^*$ . The figure RATIO is computed as  $\text{RATIO} = (Z_{lp}^{**} - Z_{lp}^*) / (Z_{ip} - Z_{lp}^*)$ . Thus RATIO measures the fraction of the GAP that was closed due to the generation of cutting planes. For example, the application of constraint generation to problem P2756 produced 326 new constraints and closed 98% of the gap.

While we have not gathered data on the numbers of the two types of constraints (minimum covers and  $(1,k)$ -configurations, respectively), that our computer programs generated, runs made at an earlier state of the experiment revealed that roughly 5 to 10% of the constraints generated were  $(1,k)$ -configurations. The low percentage is not surprising because of the incompleteness of our identification procedure for  $(1,k)$ -configurations. Originally, however, we worked with inequalities from minimum covers only and results improved markedly when we used both minimum

covers and  $(1,k)$ -configurations. The reason for the improvement rests partially with the value of the inequalities (2.9) for the branch-and-bound phase: If the key variable  $x_t$  in a  $(1,k)$ -configuration is set equal to the value one, inequality (2.9) strongly constrains the sum of the other variables. Because minimum covers are just a special form of  $(1,k)$ -configurations, however, we did not study the relative merit of one type over the other in any detail.

### 5.3. Branch-and-Bound Phase

The application of MIP/370 to perform branch-and-bound is the third computational phase in the PIPX system. Because of the necessity to revise a problem after variables are fixed using continuous reduced cost implications, most of the test problems required more than one applica-

TABLE VI  
BRANCH-AND-BOUND SUMMARY

NAME	BRANCH-AND-BOUND		DJ FIX		
	SOLUTIONS	NODES	PASSES	$\Delta$ ROW	$\Delta$ VAR
P0033	2	113	0	0	0
P0040	1	11	1	2	15
P0201	2	1116	0	0	0
P0282	4	1862	0	0	0
P0291	1	87	1	164	127
P0548	1	36	1	66	261
P1550	2	10	2	0	1442
P1939	2	334	2	0	1746
P2655	5	214	2	0	1967
P2756	3	2392	1	260	1341

tion of the branch-and-bound procedure. Table VI summarizes the branch-and-bound phase and the related continuous reduced cost implication results. BRANCH-AND-BOUND lists the total number of zero-one solutions obtained (both optimal and nonoptimal) and the total number of branch-and-bound tree nodes produced for all applications of the branch-and-bound procedure. The columns headed DJ FIX specify the statistics for the continuous reduced cost implication procedure. The PASSES column shows the total number of times the continuous reduced costs were used to reduce the problem. The  $\Delta$ ROW and  $\Delta$ VAR columns give the total number of constraints and variables eliminated due to continuous reduced cost implications. Since the problem is subsequently processed again by the preprocessor, additional variables are fixed and additional constraints are dropped. These statistics are included in Table VI.

5.4. PIPX Execution Summary

Table VII summarizes the time, in CPU minutes, for PIPX system execution on the test problem set. PIPX Passes is the number of major iterations performed by PIPX (see Section 4). The columns headed TIME specify the aggregate times spent in each of the three main PIPX computational phases, and the total time required to solve the test problems.

6. CONCLUSIONS

All test problems considered during the course of this study were solved to optimality. The methodology that we propose for the solution of large-scale zero-one linear programming problems produced—by today’s stand-

TABLE VII  
PIPX EXECUTION SUMMARY

NAME	PIPX Passes	TIME*			
		Pre-processor	LP/Constraint Generation	Branch- and-Bound	Total
P0033	1	0.01	0.12	0.56	0.69
P0040	2	0.03	0.13	0.02	0.18
P0201	1	0.01	0.16	9.90	10.07
P0282	1	0.02	1.00	11.70	12.72
P0291	2	0.04	0.51	0.30	0.85
P0548	2	0.12	0.70	0.09	0.91
P1550	3	0.50	0.90	0.10	1.50
P1939	3	0.49	0.86	13.64	14.99
P2655	3	0.82	3.23	2.70	6.75
P2756	2	1.07	4.03	49.32	54.42

\*CPU minutes, IBM 370/168 - MVS

ards—impressive computational results, particularly on sparse problems having no apparent special structure. The test problems were given to us from various sources within and outside of the IBM Corporation and, while we have no comparative data on previous solution attempts, most were originally considered not amenable to exact solution in economically feasible computation times. The computational results reported in Section 5 contradict this sentiment and strongly confirm our hypothesis that a combination of problem preprocessing, cutting planes, and clever branch-and-bound technique permits the optimization of sparse large-scale zero-one linear programming problems, even when no apparent special structure is present, in reasonable computation times. Our results indicate that cutting-planes related to the facets of the underlying polytope are an indispensable tool for the exact solution of this class of

problem, and they agree with the results of an earlier study (Crowder and Padberg) of cutting planes for the symmetric traveling salesman problems. A case in point is the problem P0548 where the preprocessing closed a considerable portion of the gap between the continuous objective function value of the user-supplied linear programming formulation and the integer objective function value. Undoubtedly, however, preprocessing alone would not have permitted us to optimize P0548 in about one minute of CPU time. The remaining gap is still much too large to permit one to expect completion of the branch-and-bound phase within a reasonable time limit. Indeed, we originally attempted to solve this problem without the use of cutting planes and had to give up after letting the problem run for several CPU hours. After identifying cutting planes, the gap after preprocessing was reduced to a small fraction of the original value; this, combined with branch-and-bound improvements, permitted us to solve the problem—quite unexpectedly—in less than one minute of CPU time. This observation is not a singular occurrence; it is supported by the results of several other successful runs including problem P2756 having 2756 zero-one variables.

The problem characteristics given in Table II reveal that the cutting planes considered here work best when we have both a low density *and* a substantial variance among the nonzero coefficients of the constraint matrix. To measure this variance, we use the statistics given by the preprocessed version of the problem and *not* the user-supplied version, because the cutting planes are generated from the preprocessed problem. Whenever these criteria are not met, the constraint generation as implemented in our experimental software system PIPX did not have a substantial impact upon our computations. In these cases, we did obtain the impressive computation times reported for the problems considered in this paper, because of the various (in part probably known) improvements that we implemented over the standard MIP/370 algorithm, the most important of which, in our view, is the use of the reduced cost implications for limiting the search; see the discussion in Section 3. However, problem P0033 is a notable exception indicating that even for dense problems one can *sometimes* expect substantial improvements of the linear programming relaxation by the use of cutting planes derived from the individual rows of the constraint matrix. As mentioned in Section 2, it would be wrong to conclude that cutting planes cannot be used as an aid in solving problems having a dense matrix or an insignificant variance among the nonzero coefficients of the constraint matrix; rather, for this type problem, the constraint generation must be carried out differently from the way we currently generate cutting planes.

In the course of this study, we encountered two major difficulties in implementation. The first was the constraint identification procedure

discussed in Section 2.3. While the method used here to identify violated constraints is an ad hoc procedure, it appears to be very effective in actual computation; it is easy to implement and we do not hesitate to advocate its use for the solution of sparse large-scale problems displaying a substantial intra-row variance of the nonzero coefficients. It is clear, however, that special ordered set (SOS) constraints, when present, should be taken into consideration explicitly for the constraint generation as well as for the preprocessing of zero-one problems. Our experience with the use of such constraints indicates that it is preferable to work with a *flexible* SOS structure rather than with a *fixed* or *user-supplied* SOS structure, particularly in identifying constraints. Being able to solve the identification problem discussed in Section 2 rigorously by a polynomially bounded algorithm should go a long way toward making the exact solution of sparse large-scale pure zero-one linear programming problems a routine task.

The second major difficulty was the design and implementation of an effective and efficient interface between our computational procedures and MPSX-MIP/370. Our experimental software system PIPX was concerned not with only performing the mathematical procedures described previously, but also with the management of storage and control flow in concert with MPSX-MIP/370. The use of the Extended Control Language (ECL) of MPSX/370 (ECL is, in fact, a superset of PLI) facilitated this task. For simple applications, the design and implementation of ECL programs for utilizing MPSX-MIP/370 functions is usually straightforward. For an application on the scale of PIPX, however, the task is more complex and requires a certain amount of "experimentation" to arrive at an effective and efficient system. The fact that we were able to realize such a system demonstrates anew the feasibility of using commercial linear programming package components as building blocks in specialized applications.

Finally, we are confident that the exact solution method discussed in this paper permits one to solve substantially larger zero-one linear programming problems than those considered in this study, in economically feasible computation times. We limited ourselves to the solution of real-world models because we seriously doubt the validity of testing such methods on randomly generated problems.

## APPENDIX

We illustrate by way of a numerical example the constraint generation described in Section 2. Consider the zero-one knapsack problem:

$$\begin{aligned} \max \quad & 1200x_1 + 1300x_2 + 1300x_3 + 1200x_4 + 899x_5 + 999x_6 \\ & + 899x_7 + 1099x_8 \end{aligned}$$



subject to

$$12x_1 + 13x_2 + 13x_3 + 12x_4 + 9x_5 + 10x_6 + 9x_7 + 11x_8 \leq 39$$

$$x_i = 0 \quad \text{or} \quad 1 \quad \text{for} \quad i = 1, \dots, 8.$$

Using an ad hoc argument, one can readily see that the optimal zero-one solution to this problem is given by  $x_1 = x_2 = x_3 = x_4 = 0$ ,  $x_5 = x_6 = x_7 = x_8 = 1$ . Besides illustrating the concepts of Section 2, we will next prove optimality of this solution by a linear programming argument. We start by replacing the conditions  $x_i = 0$  or  $1$  for  $i = 1, \dots, 8$  by  $0 \leq x_i \leq 1$  for  $i = 1, \dots, 8$ . The Dantzig solution to the resulting continuous knapsack problem is given by  $x_1 = x_2 = x_3 = 1$ ,  $x_4 = 1/12$ ,  $x_5 = x_6 = x_7 = x_8 = 0$  and has an objective function value of  $Z_{LP} = 3900$ . To find a violated minimum cover (2.7), we solve (2.12), i.e. we solve

$$\min 0s_1 + 0s_2 + 0s_3 + (1/12)s_4 + s_5 + s_6 + s_7 + s_8$$

subject to

$$12s_1 + 13s_2 + 13s_3 + 12s_4 + 9s_5 + 10s_6 + 9s_7 + 11s_8 \geq 40$$

$$s_i = 0 \quad \text{or} \quad 1 \quad \text{for} \quad i = 1, \dots, 8,$$

approximately by replacing the integrality constraints by  $0 \leq s_i \leq 1$  for  $i = 1, \dots, 8$ . Note that the data of the constraint are integer and thus  $>39$  is replaced by  $\geq 40$ . (For *rational* data, we would have to replace  $>39$  by  $\geq 39 + \epsilon$  where  $\epsilon$  is a small positive number.) The optimal solution is  $s_1 = s_2 = s_3 = 1$ ,  $s_4 = 2/12$ ,  $s_5 = s_6 = s_7 = s_8 = 0$  and rounding yields  $S = \{1, 2, 3, 4\}$  which defines a minimal cover. Since

$$\sum_{j \in S} x_j = 3^{1/12} > 3$$

the minimum cover inequality (2.7) chops off the Dantzig solution. We extend the inequality using the linear programming version of (2.10) and  $K - S = \{5, 6, 7, 8\}$  in this ordering of the indices. Carrying out the first step, we have to find

$$\tilde{z}_5 = \max \sum_{j=1}^4 x_j$$

subject to

$$12x_1 + 13x_2 + 13x_3 + 12x_4 \leq 39 - a_5 = 30$$

$$0 \leq x_j \leq 1 \quad \text{for} \quad j = 1, \dots, 4$$

and find  $\tilde{z}_5 = 2^{6/13}$  and thus the rounded value  $z_5^* = 2$  yields a lifting coefficient of  $3 - 2 = 1$  for variable  $x_5$ . Repeating for variable  $x_6$ , we have to find

$$\tilde{z}_6 = \max \sum_{j=1}^5 x_j$$

subject to

$$12x_1 + 13x_2 + 13x_3 + 12x_4 + 9x_5 \leq 39 - a_6 = 29$$

$$0 \leq x_i \leq 1 \quad \text{for } i = 1, \dots, 5$$

and obtain  $\bar{z}_6 = 2\frac{2}{3}$ . Consequently,  $z_6^* = 2$  and variable  $x_6$  obtains a lifting coefficient of 1. Repeating for variable,  $x_7$ , we obtain  $\bar{z}_7 = 2\frac{11}{12}$  and thus  $z_7^*$  yields a lifting coefficient of 1 for variable  $x_7$ . Finally, repeating for variable  $x_8$ , we find  $\bar{z}_8 = 3$  and thus a lifting coefficient of zero. We have thus found the lifted inequality

$$x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + 0x_8 \leq 3,$$

which, incidentally, defines a facet of the knapsack polytope. We have thus an inequality that chops off the current linear programming optimum of the continuous example knapsack problem. Rather than stopping at this point and reoptimizing the linear program, we notice that the original minimum cover has four liftings all of which chop off the continuous optimum. Moreover, by a simple labeling procedure, we can find all four of them. Adding the four lifted inequalities, we get the following linear program to be reoptimized:

$$\begin{aligned} \max \quad & 1200x_1 + 1300x_2 + 1300x_3 + 1200x_4 + 899x_5 + 999x_6 \\ & + 899x_7 + 1099x_8 \end{aligned}$$

subject to

$$\begin{aligned} & 12x_1 + 13x_2 + 13x_3 + 12x_4 + 9x_5 + 10x_6 + 9x_7 + 11x_8 \leq 39 \\ (LP) \quad & x_1 + x_2 + x_3 + x_4 + 0x_5 + x_6 + x_7 + x_8 \leq 3 \\ & x_1 + x_2 + x_3 + x_4 + x_5 + 0x_6 + x_7 + x_8 \leq 3 \\ & x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + 0x_7 + x_8 \leq 3 \\ & x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + 0x_8 \leq 3 \\ & 0 \leq x_i \leq 1 \quad \text{for } i = 1, \dots, 8. \end{aligned}$$

A suboptimal solution to (LP) is  $x_1 = x_3 = x_4 = 0$ ,  $x_2 = 1$ ,  $x_5 = x_6 = x_7 = x_8 = \frac{2}{3}$  which has an objective function value of  $z_{LP}^* = 3897\frac{1}{3}$  and will be used for illustration. Setting up problem (2.12) yields the problem

$$\min s_1 + 0s_2 + s_3 + s_4 + \frac{1}{3}(s_5 + s_6 + s_7 + s_8)$$

subject to

$$\begin{aligned} & 12s_1 + 13s_2 + 13s_3 + 12s_4 + 9s_5 + 10s_6 + 9s_7 + 11s_8 \geq 40 \\ & s_i = 0 \quad \text{or} \quad 1 \quad \text{for } i = 1, \dots, 8. \end{aligned}$$

Solving this problem approximately as a linear program yields the optimal solution  $s_2 = s_6 = s_8 = 1$ ,  $s_7 = 2/3$ ,  $s_1 = s_3 = s_4 = s_5 = 0$  and rounding yields  $S = \{2, 6, 7, 8\}$  which defines a minimum cover. However,

$$\sum_{j \in S} x_j = 3$$

and thus the minimum cover inequality does not chop off the continuous optimum (nor does a lifting of it do the job). We thus look for a  $(1, k)$ -configuration: since  $a_2 > a_j$  for  $j = 6, 7, 8$ , we set  $t = 2$ , and  $k = 3$ . There is only one unscanned index  $i = 5$  and the associated inequality (2.9) is

$$2x_2 + x_5 + x_6 + x_7 + x_8 \leq 4$$

which chops off the continuous optimum. Extending the inequality to the remaining variables  $x_1$ ,  $x_3$ , and  $x_4$  as before, we get the inequality

$$x_1 + 2x_2 + x_3 + x_4 + x_5 + x_6 + x_7 + x_8 \leq 4.$$

We adjoin this inequality to  $(LP)$  and reoptimize. We obtain an "alternate" optimum solution  $x_1 = x_2 = x_4 = 0$ ,  $x_3 = 1$ ,  $x_5 = x_6 = x_7 = x_8 = 2/3$  with objective function value  $z_{LP}^* = 3897\frac{1}{3}$ . In our computer program we would stop at this point and resort to branch-and-bound to finish the problem off, because the last cut did not improve the objective function. For matters of completeness we note, however, that repeating the constraint identification procedure produces the (lifted) inequality

$$x_1 + x_2 + 2x_3 + x_4 + x_5 + x_6 + x_7 + x_8 \leq 4$$

from the  $(1, k)$ -configuration with  $t = 3$ ,  $S^* = \{5, 6, 7, 8\}$  and  $k = 3$  which chops off the continuous optimum and which is found as before. Moreover, reoptimizing the linear programming over all constraints that have been identified yields  $x_1 = x_2 = x_3 = x_4 = 0$ ,  $x_5 = x_6 = x_7 = x_8 = 1$  with an objective function value  $z = 3896$  as an optimal solution and, because the solution is integer, branch-and-bound is superfluous. Finally to facilitate the checking of optimality of this solution, we give the respective dual solution in the ordering of the constraints as suggested by the above:  $v_1 = 0$ ,  $v_2 = 397$ ,  $v_3 = 297$ ,  $v_4 = 397$ ,  $v_5 = 197$ ,  $v_6 = 4$ ,  $v_7 = 4$ . All other dual variables belonging to the remainder of the totality of all linear constraints of this particular knapsack problem, including the upper bounds, are zero.

## REFERENCES

- CROWDER, H., AND M. W. PADBERG. 1980. Solving Large-Scale Symmetric Travelling Salesman Problems to Optimality. *Mgmt. Sci.* **26**, 495-509.  
 DANTZIG, G. B. 1963. *Linear Programming and Extensions*. Princeton University Press, Princeton, N.J.

- HAMMER, P. L., M. W. PADBERG AND U. N. PELED. 1975. Constraint Pairing in Integer Programming. *INFOR—Can. J. Opns. Res.* **13**, 68–81.
- HO, J. K., AND E. LOUÏE. 1981. An Advanced Implementation of the Dantzig-Wolfe Decomposition Algorithm for Linear Programming. *Math. Program.* **20**, 303–326.
- IBM Mixed Integer Programming/370 (MIP/370) Program Reference Manual.* 1975. Form number SH19-1099, IBM Corporation.
- IBM Mathematical Programming System Extended/370 (MPSX/370) Program Reference Manual.* 1979. Form number SH19-1095, IBM Corporation.
- JOHNSON, E. L., AND M. W. PADBERG. 1982. Degree-Two Inequalities, Clique Facets and Bipartite Graphs. *Ann. Discrete Math.* **16**, 169–187.
- JOHNSON, E. L., AND M. W. PADBERG. 1981. A Note on the Knapsack Problem with Special Ordered Sets. *Opns. Res. Lett.* **1**, 38–43.
- PADBERG, M. W. 1979. Covering, Packing and Knapsack Problems. *Ann. Discrete Math.* **4**, 265–287.
- SPIELBERG, K. 1979. Enumerative Methods in Integer Programming. *Ann. Discrete Math.* **5**, 139–183.