NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

SET COVERING ALGORITHMS USING CUTTING PLANES, HEURISTICS, AND SUBGRADIENT OPTIMIZATION: A COMPUTATIONAL STUDY

by

Egon 5alas & Andrew Ho

DRC-70-5-79

September 1979

Graduate School of Industrial Administration
Carnegie-Mellon University
Pittsburgh, PA 15213

This report was prepared as part of the activities of the Management Sciences Research Group, Carnegie-Mellon University, under Grant MCS76-12026 A02 of the National Science Foundation and Contract N0014-75-C-0621 NR 047-048 with the U.S. Office of Naval Research. Reproduction in whole or in part is permitted for any purpose of the U.S. Government.

620.0042
C28d
DRC ;U5-79

Abstract

We report on the implementation and computational testing of several versions of a set covering algorithm, based on the family of cutting planes from conditional bounds discussed in the companion paper [2]. The algorithm uses a set of heuristics to find prime covers, another set of heuristics to find feasible solutions to the dual linear program which are needed to generate cuts, and subgradient optimization to find lower bounds. It also uses implicit enumeration with some new branching rules. Each of the ingredients was implemented and tested in several versions. The variant of the algorithm that emerged as best was run on 55 randomly generated test problems (20 of them from the literature), with up to 200 constraints and 2000 variables. The results show the algorithm to be more reliable and efficient than earlier procedures on large, sparse set covering problems.

University Libraries
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

SET COVERING ALGORITHMS USING CUTTING PLANES, HEURISTICS, AND SUBGRADIENT OPTIMIZATION: A COMPUTATIONAL STUDY

by

Egoa Balas and Andrew Ho

1. Introduction: Cutting Planes from Conditional Bounds.

In this paper we report on the implementation and computational testing of an algorithm, or rather a class of algorithms, based on the cutting planes from conditional bounds introduced in [1], and also using as ingredients several heuristics for finding feasible primal and dual solutions, subgradient optimization of a Lagrangean function, and implicit enumeration with some new branching rules.

The family of cutting planes from conditional bounds is briefly described in this section; a more detailed discussion of the properties of these cuts is to be found in the companion paper [2]. In section 2 we describe the general features of the algorithm whose versions we implemented and tested. Sections 3-7 discuss **each** of the ingredients of our procedure, with comparisons of various versions based on computational testing. Finally, section 8 summarizes our computational experience with the algorithm as a whole. While the discussion focuses on a particular instance of the algorithms in the class considered, namely the one that emerged as the most successful from our computational experiments, we also discuss possible variations wherever it seems useful.

As one can see from the computational results presented in section 8, the algorithm discussed here is a reliable and efficient tool for solving large, sparse set covering problems of the kind that frequently occur in practice. With a time limit of 10 minutes on the DEC 20/50, we have solved

all but one of a set of 50 randomly generated set covering problems with up to 200 constraints, 2000 variables, and 8000 nonzero matrix entries (here "solving[11] means finding an optimal solution and proving its optimality), never generating a branch and bound tree with more than 50 nodes. We know of no other approach with comparable performance. For problems that are too large to be solved within a reasonable time limit, the procedure usually finds good feasible solutions, with a bound on the distance from the optimum (for the one unsolved problem, this bound was 2.37o). We also tested the algorithm on a set of 57o density problems, but as density increases, the performance of the algorithm tends to decline.

We consider the set covering problem

(SC) $\min Ccx|Ax\geq e,\ xe\{0,l\}^a\}$

where $A = (a_{ij})$ is an m X n 0-1 matrix, and e * (1,.,.,1) has m components. Let a^i and a_j denote the i-th row and the j-th column of A, and let $M = \{l,...,m\}$, $N =* \{l,...,n\}$. We denote

$$M_j = UeMla_i.^1\}, \quad jeN; \quad N_{\pm} = \{jeN\ a\ J^1\}, \quad ieM.$$

We will also use the pair of dual linear programs

(L) $\min (cx|Ax\geq e,\ x\geq 0\}$

and

(D) $\max (ueJuA\leq c,\ u\geq 0\}$,

associated with (SC).

A vector $xe\{0,l\}^a$ satisfying $Ax\geq e$ is called a cover, and $S(x) = \{jeN|x_j=1\}$ its support. A cover whose support is minimal, is prime. For a cover x, we denote $T(x) = \{ieM|a^i x=1\}$.

The theory underlying the family of cutting planes from conditional

bounds can be summarized as follows (for proofs of these statements and further elaboration see the companion paper [2]).

Let z_U be some known upper bound on the value of (SC), and let u be any feasible solution to (D), with s=c-uA, such that the condition

$$(1) \qquad \sum_{US} s_j \geq z_U - ue$$

is satisfied for some $S \subseteq I$, $S = (j(1),\ldots,j(p))$. Further, let Q_i, $i=1,\ldots,p$, be any collection of subsets of N satisfying

$$(2) \qquad \sum_{i \mid j \in Q_i}^{P} s_{j(i)} \leq s_j, \quad j \in N.$$

Then every cover x such that $cx<z_U$ satisfies the disjunction

$$(3) \qquad \bigvee_{i=1}^{P} (x_j \geq 0, \quad j \in Q_t).$$

Further, for any choice of indices $h(i) \in M$, $i=1,\ldots,p$, the disjunction (3) implies the inequality

$$(4) \qquad \sum_{j \in W} x_j \geq 1$$

where

$$W = \bigcup_{i=1}^{P} (N_{j(i)} \setminus Q_i).$$

Finally, if $j(i) \in Q_i$, $i=1,\ldots,p$, and if $\bar{x}$ is a cover such that $S \subseteq S(\bar{x})$ and $h(i) \in T(\bar{x}) \cap M_{j(i)}$, $i=1,\ldots,p$, then the inequality (4) cuts off $\bar{x}$ and defines a facet of

$$P^* = \{x \in R^n \mid Ax \geq e, \ \sum_{j \in W} x_j \geq 1, \ x \geq 0, \ x_j \text{ integer}, \ j \in N\}.$$

Using the above results, one can generate a sequence of cutting planes that are all distinct from each other, by generating a sequence of covers

x and feasible solutions u to (D). The covers x provide upper bounds, while the vectors u provide lower bounds on the value of (SC). Since every inequality generated cuts off a cover satisfying all previously generated inequalities, and the number of distinct covers is finite, the procedure ends in a finite number of iterations, with an optimal cover at hand.

2. Outline of the Algorithm.

The algorithm alternates between two sets of heuristics, one of which finds a "good" prime cover x for the current problem and a (possibly improved) upper bound, while the other generates a feasible solution to (D) satisfying (1) for S=S(x), and from it a cutting plane that cuts off x, as well as a (possibly improved) lower bound. Whenever a disjunction (3) is obtained with p=l, all the variables indexed by Q_1 are set to 0. The second set of heuristics is periodically supplemented by subgradient optimization to obtain sharper lower bounds. Though this procedure in itself must find an optimal cover in a finite number of iterations, for large problems this may take too many cuts. Therefore, as soon as the rate of improvement in the bounds decreases beyond a certain value, the algorithm branches.

A schematic flowchart of the algorithm is given in figure 1. PRIMAL designates the set of heuristics used for finding prime covers (feasible primal solutions), DUAL the heuristics used for finding feasible dual solutions. TEST is the routine for fixing variables at 0. CUT generates a cutting plane violated by the current cover. SGBAD uses subgradient optimization in an attempt to find an improved dual solution and lower bound. BRANCH is the branching routine, which breaks up the current

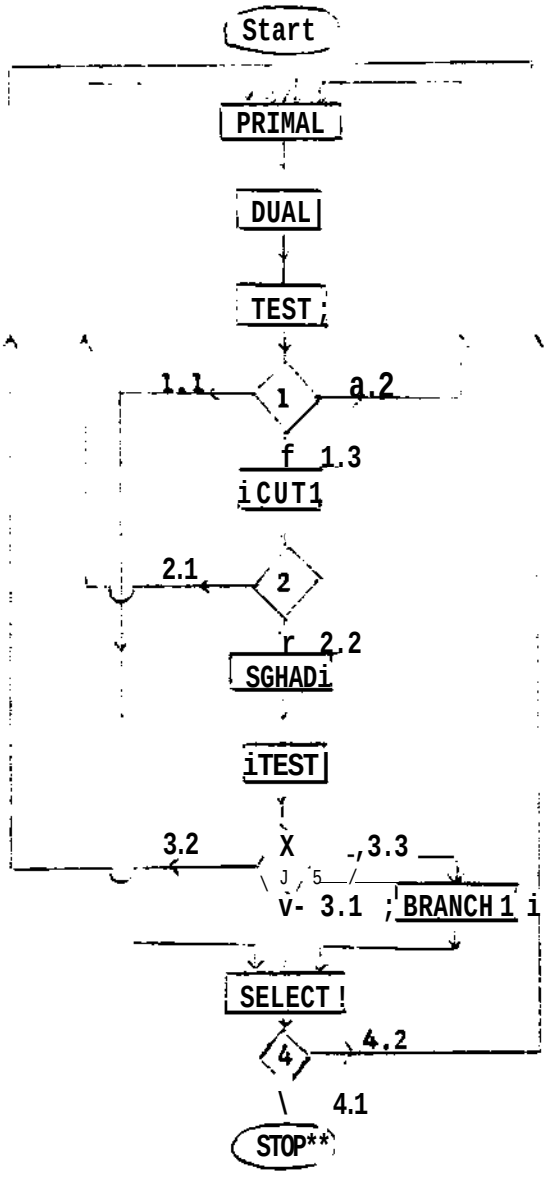
Start
PRIMAL
DUAL
TEST ;
1.1
1
a.2
f 1.3
iCUT1
2.1
2
r 2.2
SGHADi
iTEST
3.2
X
3.3
J 5
V- 3.1 ; BRANCH 1 i
SELECT !
4
4.2
4.1
STOP**

Figure 1.

problem into a number of subproblems, while SELECT chooses a new subproblem to be processed.

The four decision boxes of the flowchart can be described as follows. Let z_U and z_L be the current upper and lower bound, respectively, on the value of (SC).

1. If $z_L \geq z_U$, the current subproblem is fathomed (1.1). If $z^{\wedge} < z_{JJ}$ and some variable belonging to the last prime cover has been fixed at 0, a new cover needs to be found (1.2). Otherwise, a cut is generated (1.3).

2. After adding a cut, the algorithm returns to PRIMAL (2.1) unless the iteration counter is a multiple of &; in which case (2.2) it uses subgradient optimization in an attempt to improve upon z_{Li}. On the basis of some experimentation, we set $|M|/10 \leq ct \leq |M|/20$,

3. If $z^L \geq z^U$, the current subproblem is fathomed (3.1). If $z_T < Z^U$ but the gap $z^U - L$ has decreased by at least s>0 during the last 0 iterations, we continue the iterative process (3.2). Otherwise, we branch (3.3). Again, following some numerical experiments, we use e = 0.5 and 3 = 4a, with <* as defined in 2.

4. If there are no active subproblems, the algorithm stops: the cover associated with z^U is optimal (4.1). Otherwise, it applies the iterative procedure to the selected subproblem (4.2).

In the next five sections we discuss each of the ingredients of the algorithm in some detail on the basis of computational testing of several versions. After that we report on our computational experience with the algorithm as a whole.

3. Primal Heuristics.

The heuristics we use to generate prime covers are of the "greedy"

type, in that they construct a cover by a sequence of steps, each of which consists of the selection of a variable x_j that minimizes a certain function f of the coefficients of x_j. They differ in the function f used to evaluate the variables. If k_j denotes the number of positive coefficients of x_j in those rows of the current constraint set not yet covered, the general form of the evaluation function is $f(c_j,k_j)$.

Since it is computationally cheaper to consider only a subset of variables at a time and since every row must be covered anyhow, i.e., the cover to be constructed must contain at least one of the variables having positive coefficient in any given row, we restrict the choice at each step to the set of variables having a positive coefficient in some specified row i*eM. Denoting by R the set of rows not yet covered and by S the support of the cover to be constructed, the basic procedure that we use can be stated as follows.

Step 0. Set R = M, S • 0, t = 1, and go to 1.

Step 1. If R - 0, go to 2.

Otherwise, let k_j = [M-flRJ, choose i*eR, and choose j(t) such that

$$f(c_{j(t)}, k_{j(t)}) = \min_{j \in N_{i_*}} f(c_j,k_j).$$

Set R- R\W$_{j(t)}$, S - SU£j(t)}, t -t+1, and go to 1.

Step 2. Consider the elements igS in order, and if S\{i} is the support of a cover, set S - S\(i}. When all icS have been considered, S defines a prime cover.

As to the choice of i^ in Step 1, the criterion that suggests itself is

$$|N_{i_*} I - \min_{i \in R} JN_i|.$$

Rather than implement this choice rule directly, which would be costly, we approximate it by ordering once and for all the rows of the initial coefficient matrix A according to decreasing $|N_i|$, and always choosing i_* as the last element of the ordered set R. Since the cuts generated in the procedure also tend to have a decreasing number of 1's, i.e., later cuts tend to have fewer 1's than earlier cuts, this rule comes sufficiently close to choosing the row with the minimum number of 1's.

If the set N_{i_*} in step 1 is replaced by N, i.e., the choice is not restricted to a certain row, and step 2 is removed, i.e., the procedure is allowed to stop whenever a cover is obtained, whether prime or not, then the above procedure with $f(c_j,k_j) = c_j/k_j$ is the greedy heuristic shown by Chvátal [3] to have the following property: if z_{opt} is the value of (SC) and z_{heu} is the value of a solution found by the heuristic, then

$$z_{heu}/z_{opt} \leq \sum_{j=1}^{d} \frac{1}{j}$$

where

$$d = \max_{j \in N} |M_j|,$$

and this bound is best possible. From a practical point of view this bound is very poor, and it can be shown [7] that there is no better bound for any function f used in the above procedure. However, proving this statement requires the construction of examples for which the worst case bound is attained, and every function f requires a different example. This suggests as a practical remedy against the poor worst-case performance of the heuristic, the intermittent use of several functions f rather than a single one. This idea has been implemented and tested with reasonably good results. The following five functions $f(c_j,k_j)$ were considered:

(i) c_j ; (ii) c_j/k_j ; (iii) $c_j/\log_2 k_j$; (iv) $c_j/\log_2 \hat{}_j$;
(v) $c_j/k_j \ln k_j$. In cases (iii) and (iv), $\log_2 k_j$ is to be replaced by 1 when $k_j \gg 1$; and in case (v) $\ln k_j$ is to be replaced by 1 when $k_j = 1$ or 2.

Using (i) amounts to simply choosing the lowest-cost variable at each step. Criterion (ii) minimizes the unit cost of covering an uncovered row. The functions (iii), (iv) and (v) select the same variable as function (ii) whenever $c_j = 1$, $j \in N$, but otherwise (iii) assigns less weight, while (v) assigns more and (iv) even more weight to the number k_j of rows covered, versus the cost c_j.

The five functions were tested on a set of 13 randomly generated problems with $100 \leq m \leq 200$, $100 \leq n \leq 1000$, and 2% density. Though none of them emerged as uniformly dominating any of the others in terms of the quality of the solutions obtained, (iii) scored best and (ii) second best, in the sense that of the 13 problems, (iii) gave the best solution in 6 cases, (ii) in 5 cases. As to the other functions, the best solution was found by (i) and (v) in 3 cases each, and by (iv) in 2 cases (the sum of these numbers exceeds 13, since often more than one function gave a best solution). Table 1 shows the % deviation from the optimum, of the solution found by each function for each problem. The numbers in the first column are those under which the problems are listed in section 8, where they are also described in more detail. The best solution found by any of the 5 functions never deviated from the optimum by more than 10.8%.

The above described procedure can be amended so as to produce, at little extra cost, more than one cover, the best of which can then be retained. This is done as follows. We first use the above heuristic to find a cover. Then we consider the variables in the order of their inclusion into the cover, and remove from the cover all those that have a positive coefficient in at least one oversatisfied constraint. Next we

Table 1. Deviation from optimum (in %) of the values found by primal heuristics.

Problem data			Function used with heuristic				
No.	m	n	c_j	c_j/k_j	$c_j/\log_2 k_j$	$c_j/k_j \log_2 k_j$	$c_j/k_j \ln k_j$
2.2	200	413	5.4	5.0	6.2	16.3	5.2
2.3	200	300	3.1	8.8	10.0	14.1	13.0
2.4	200	500	11.4	11.7	2.3	14.5	14.5
3.1	100	100	1.6	0.5	1.6	0.5	0.5
3.2	100	200	6.9	3.5	7.6	7.6	6.9
3.3	100	300	3.7	16.4	3.1	12.0	16.4
3.4	100	400	16.2	6.5	10.3	9.1	9.7
3.5	100	500	11.7	28.2	10.8	31.1	28.8
3.6	100	600	6.4	17.6	4.9	23.1	18.7
3.7	100	700	4.7	8.0	6.7	13.7	11.5
3.8	100	800	4.6	2.4	3.5	2.4	2.4
3.9	100	900	7.3	15.9	7.3	9.2	6.4
3.10	100	1000	6.8	1.5	1.5	1.5	1.5

complete the cover again using the heuristic. We continue in this fashion until either a cover is generated which does not oversatisfy any of the constraints, or the number of covers generated is some specified integer a. To find the most desirable value for a, we have applied this procedure with o«10 to each of the above described 13 problems, with each of the 5 functions discussed. Somewhat surprisingly, we found that of the 13 X 5 = 65 cases, the best of the 10 covers generated was the first one in 22 instances, the second one in 34 instances, the third one in 8 instances, and in 1 instance it was the fifth one. In other words, only once was there an improvement after the third cover was found, and this in spite of the fact that the best cover found was not optimal in any of the 65 cases.

Consequently, we set o*3, and use function (iii) for the first cover, a different function ((i) or (ii)) for the second cover, and again a

different function for the third cover. This way the procedure is still computationally cheap, and yields considerably better results than the version that produces only one cover. We call this procedure PRIMAL 1.

The general algorithm discussed in section 1 at times fixes variables at 0. Whenever some variable belonging to the current cover gets fixed at 0, we have to generate a new cover. Rather than starting from scratch, in such situations we start with the partial cover at hand, and complete the cover by using the procedure discussed above. This version of the heuristic we call PRIMAL 2.

When the dual heuristics to be discussed in the next section produce a vector (u,s) such that (1) does not hold for S=*S(x), where x is the current cover, either the dual solution u must be weakened (see the next section), or else the cover x must be replaced by another one, say $\bar{x}$, such that (1) is satisfied for S=S($\bar{x}$). PRIMAL 3 was designed to accomplish this starting with the cover at hand. It introduces into the cover additional variables x_j such that $s_j(X)$, in order of increasing values of f (as defined in (iii)), and removes variables in order of increasing values of s_j so as to keep the cover prime. While it is not guaranteed to succeed, the percentage of failure is sufficiently low to justify the use of PRIMAL 3. When it fails, the dual solution u must be weakened as explained in the next section.

Whenever a new cut is generated, the last cover satisfies all the constraints except for the newly added one. Since it is much cheaper to obtain a new cover from the old one than to construct a new one from scratch, a special heuristic was implemented for this purpose. PRIMAL 4 adds to the current cover a number *c* of variables with positive coefficients

in the cut just generated, chosen in order of increasing c_j, and then removes from the cover redundant variables so as to make it prime. Computational experiments with c=l,...,5 have shown c=2 to give the best results.

Finally, every time we apply the subgradient method to obtain an improved lower bound, we also generate a new cover by using the reduced costs s_j produced by that procedure. This is done by subroutine PRIMAL 5, by setting $x_j = 1$ if $s_j = 0$, $x_j \leq 0$ otherwise. The resulting vector either is a cover, or else if row i is uncovered, then $s_j > 0$, $Vj \in N_i$, and variable u_i can be increased to $u_\# + \min s_j$. This creates at least one new reduced cost s_j equal to 0, and for each such k we set $x_k = 1$. We proceed this way until every row is covered, after which we make sure the cover is prime, like in Step 2 of PRIMAL 1. PRIMAL 5 produces consistently better covers (hence better upper bounds) than any of the other 4 procedures; but obtaining the reduced costs by subgradient optimization is many times more expensive than producing them by the dual heuristics, as will be discussed below.

While the conditions for using PRIMAL 2, 3 and 5 have been spelled out, the choice between PRIMAL 1 and 4 seems open at this point. PRIMAL 4 is computationally cheaper, but it often produces a cover that differs very little from the previous one. PRIMAL 1 is more expensive, but yields a genuinely new cover. We follow the strategy of using PRIMAL 1 to obtain the first cover, then using PRIMAL 4, except for every 9-th iteration, when we again use PRIMAL 1. As to the value of 0, computational experiments have led us to start with 9=1 and then set 9 «- min {9+1,7}. In other words, at the beginning we return to PRIMAL 1 more frequently, then at regular intervals of, say, 7 iterations.

3. Dual Heuristics.

The purpose of the dual heuristics is to find, at a low computational

cost, a feasible solution to the dual linear program (D) associated with the current problem, with as high an objective function value (lower bound on the value of (D), hence of (SC)), as possible. In addition, the dual solution u and its associated reduced cost vector s=c-uA have to satisfy condition (1) for S=S(x), where x is the current cover. Again, we use a procedure of the "greedy" type, which considers the variables in some prescribed order and assigns to each one the maximum value that can be assigned without violating some constraint or changing some earlier value assignment. Since it is known (see **[2]**, Theorem 4) that for a feasible solution u to (D), the vector s • c - uA satisfies (1) for S«S(x) if u satisfies u(Ax-c)=O, in considering the variables u_i priority is given to ieT(x), where T(x) = {ieM|a^ix = l}. Denoting by R the index set of the dual variables (rows) not **yet** assigned a value (ordered in accordance with M), the **basic procedure is as follows.**

Step 0. Set R • MfTT(x), s=c, t«l, and go to 1.

Step 1, If R = 0, go to 2. Otherwise, let I **c** R, choose i(t) such that

$$|N_{i(t)}| = \min_{i \in I} |N_i|,$$

and let

$$u_{i(t)} = \min_{j \in N_{i(t)}} s_j.$$

Then set

$$s_j \leftarrow \begin{cases} s_j - u_{i(t)} t(t) & j \in N \backslash N_{i(t)}, \\ s_j & \end{cases}$$

R - R\£i(t)}, t - t + 1, and go to 1.

Step 2. If Step 2 is entered for the first time, set R «-M\T(x) and go to 1. Otherwise, stop: u is a feasible solution to (D).

Restricting the choice of i(t) to a subset I of R has the sole

purpose of nsaking this choice computationally less costly, at the risk of sacrificing some quality. Since the rows of the original matrix A are ordered according to decreasing $|N_i($ and the cuts generated also tend to have progressively fewer l^fs, we define I as the union of (i) the last element of MgflR, where M_0 is the row index set of the original constraint matrix, and (ii) the last A. elements of R, where X = min {|R|, V"$|^M{}_o|$}* This makes sure that the choice rarely — if at all -- misses the minimum of (N_i | over all i«R. This procedure we call DUAL 1.

A feasible solution to the current (D) remains feasible after adding a new cut to (SC), i.e., a new column to the constraint set of (D), but usually ceases to be feasible if the new variable is assigned a positive value. On the other hand, if the new variable is set to 0, the solution remains unchanged. Furthermore, if a new solution is generated from scratch by the same heuristic, it is often identical to the previous one. DUAL 2 is a version of the heuristic that starts with the infeasible solution obtained from the last feasible solution u by assigning a value of 1 to the new variable associated with the last cut. This guarantees that the dual solution to be obtained will differ from all the previous ones. Next the remaining variables are considered in a certain order, and set to 0 (or, in the case of the last variable, to the maximum allowable value), until the solution becomes feasible. The order in which the variables u_i are considered is that of decreasing number of positive coefficients in constraints of (D) that are violated, with priority given to variables u_i corresponding to primal constraints that are oversatisfied by the current cover.

Finally, a vector (u,s) generated by DUAL 1 or DUAL 2 may violate condition (1) for S=S(x), where x is the current cover. In such cases the

algorithm goes to PRIMAL 3, in an attempt to find a new cover $\tilde{x}$ such that (1) holds for S=S($\tilde{x}$). Though our computational experience has been that PRIMAL 3 seldom fails to find such a cover, failure does sometimes occur. In such cases we use DUAL 3 to modify the solution u, while weakening it as little as possible, so as to satisfy (1). DUAL 3 considers the variables u_i, ieM\T(x), in decreasing order of JHjPSCx)!, and successively reduces their value until (1) is satisfied for S=S(x). Since this latter condition always holds when all u_i ietf\T(x), are set to 0, the procedure always ends with a solution having the required property.

While DUAL 3 is used under clearly defined circumstances, DUAL 1 and DUAL 2 can be used intermittently. DUAL 2 is computationally cheaper than DUAL 1 and guarantees a new solution, but DUAL 1 is more likely to produce an improved lower bound. We start with DUAL 1 and then use DUAL 2, except for every ^-th iteration, when we again use DUAL 1. Based on some computational experimentation, we set {j,¾«

4. Subgradient Optimization,

While the dual heuristics provide reasonably good feasible solutions to (D) at a low computational cost, a sharper lower bound could of course be obtained by solving to optimality the linear program (D). After sufficient cuts have been added to the constraint set of (SC), the value of the current problem (D) may exceed z_U, thus bringing the procedure to an end. However, the computational effort involved in solving (D) by the simplex method is considerable, and increases at least quadratically with the number of cuts added to the constraint set of (SC). On the other hand, one can use subgradient optimization to find an optimal or near-optimal solution to (D) at a computational cost that seems to increase only linearly

with the number of cuts added. This is what we are doing periodically in order to generate lower bounds stronger than those obtained by the dual heuristics. Though the subgradient method consistently produces a stronger bound than the heuristics, the cuts derived from the dual solution obtained this way tend to be weaker than those derived from the heuristically generated dual solutions. This is so because the reduced costs obtained by the subgradient method do not usually satisfy (1), and the dual solution u (together with the bound ue) has to be weakened in order to get (1) satisfied.

The subgradient method that we use is a specialization of the general procedure discussed, for instance, in [6] or [5] . We wish to find or approximate

$$\max_{u \in F} \min_{x \in G} L(x,u) * ex + ue - uAx,$$

where F and G are suitable relaxations (supersets) of the feasible sets of (D) and (L) respectively, i.e.,

$$F \supset \{u \mid uA \leq c,\ 0 \leq u_t \leq \bar{u}_{if}\ \forall i\}$$

and

$$G \Rightarrow \{x \mid Ax \geq e,\ 0 \leq x_j \leq 1,\ \forall j\},$$

where $\bar{u}_i = \min_{j \in N_t} c_j$, $\forall i$.

During every subgradient iteration, given the current vector u , we

solve the problem

$$(6) \qquad \min_{x \in G} L(x,u^k),$$

and if $x(u^k)$ denotes an optimal solution to (6), we put $d^k = e - Ax(u^k)$,

$$u^{k+1} = P_F(u^k + t_k d^k).$$

Here the direction vector d^k is a subgradient of $L(x,u)$ at $u = u^k$, the scalar t_k is the step length, while $P_F(g)$ is the projection of the vector g on F. The step length is of the form

$$t_k = \frac{\lambda_k \, (z_U - L(x(u^k),u^k))}{\|d^k\|^2}$$

where $0 < \lambda_k \leq 2$ and the double bars denote the Euclidean norm; and if we take the relaxation of the feasible set of (D) to be $F = \{u \mid 0 \leq u \leq \bar{u}\}$, then the projection of $g = u^k + t_k d^k$ on F is

$$P_F(g_i) = \begin{cases} \bar{u}_i & \text{if } g_i > \bar{u}_i \\ g_i & \text{if } 0 \leq g_i \leq \bar{u}_i \\ 0 & \text{if } g_i < 0. \end{cases}$$

We tried two different relaxations G of the feasible set of (L). The first one,

$$G_1 = \{x \in R^n \mid 0 \leq x_j \leq 1, \; j \in N\},$$

makes the solution of (6) trivial: the optimum is attained at

$$x_j(u^k) \quad \begin{cases} = 0 & \text{if } u^k a_j < c_j \\ \in [0,1] & \text{if } u^k a_j = c_j \\ = 1 & \text{if } u^k a_j > c_j \end{cases}$$

For $j \in N$ such that $u^k a_j = c_j$, where any $x_j \in [0,1]$ is optimal, we have tried to set $x_j = 0$, 1/2 and 1, and $x_j = 1$ gave on the average slightly better

results; so this is what we use.

For the second relaxation, we choose a (maximal) subset $\bar{M} \subset M$ such that $N_i \cap N_k \neq 0$, $\forall i,k \in \bar{M}$, $i \neq k$, and define

$$G_2 = \left\{ x \in R^n \;\middle|\; \begin{array}{ll} \sum_{j \in N_i} x_j \geq 1, & i \in \bar{M} \\ \sum_{j \in N} d_j x_j \geq d_0 & \\ 0 \leq x_j \leq 1, & j \in N \end{array} \right\}$$

where

$$d_j = \begin{cases} -1 & \text{if } j \in \bigcup_{i \in \bar{M}} N_i \\ 1 & \text{otherwise} \end{cases}$$

and $d_0 = |M \setminus \bar{M}|$.

The idea of using the inequalities defined by a family of disjoint subsets $N_i \subset N$, $i \in \bar{M}$, is borrowed from J. Etcheberry [4]. The extra inequality that we add, which is the sum of the remaining inequalities of $Ax \geq e$, usually makes G_2 considerably more constrained.

While G_2 is a tighter relaxation than G_1, it is also one for which solving (6) is considerably more expensive. We therefore restrict ourselves, when using G_2, to approximating the optimum of (6) by a fast heuristic. In this version, the subgradient procedure using G_2 is about 1.2-1.3 times more time consuming than the one using G_1, but it also tends to be more reliable and to occasionally produce a slightly better bound. A comparison on 5 randomly generated 200 x 2000 problems (with 8000 nonzero matrix entries) showed the version using G_2 to generate 0.61 times the number of nodes (of the search tree) generated by the version using G_1, and to require 0.85 times the amount of total time required by the latter.

Currently the main version of our algorithm uses $Q^{\wedge}$.

We have tested various strategies for choosing the value of the parameter λ_k in the definition of the step length t_k, and ended up with setting $\lambda_k * 2$ at the start, and then dividing the current λ_k by 1/2 whenever there is no improvement in the value of the Lagrangean for 7 consecutive iterations of SGRAD.

To start the subgradient optimization procedure, one needs an initial solution u^0. We use for this purpose the vector u obtained fron the dual heuristics when we apply SGRAD the first time to a problem; then at subsequent applications of SGRAD we use as u^0 the dual solution obtained in the last application of SGRAD, which is usually considerably better than the one obtained from the dual heuristics. The quality of the starting solution apparently makes a great difference in the computational effort involved in SGRAD: the first application of SGRAD takes about 6 times the computational effort required by subsequent applications to the same problem (amended with cuts).

As to the overall usefulness of the subgradient method in our algorithm, our experience has been that though it is computationally more expensive than the dual heuristics by 1 and often 2 orders of magnitude, subgradient optimization nevertheless pays off. On the one hand, it produces consistently better lower bounds than the heuristics, by a margin that tends to increase with problem size; on the other, it provides a set of reduced costs that can be used by PRIMAL 5 to generate consistently better covers, and hence better upper bounds, than the other primal heuristics. These findings are illustrated in Table 2. The problems listed there are all randomly generated, 2% density set covering problems, described in more detail in section 8.

Table 2. Improvement in bounds (in %) due to SGRAD and PRIMAL 5.

Problem data			Improvement (7.)		SGRAD iterations to obtain lower bound
No.	m	n	Lower bound SGRAD versus DUAL 1	Upper bound PRIMAL 5 versus PRIMAL 1	
2.2	200	413	8.18	1.84	172
2.3	200	300	3.37	3.02	119
2.4	200	500	11.27	4.04	78
3.1	100	100	1.21	1.53	41
3.2	100	200	1.57	3.69	23
3.3	100	300	4.90	3.05	82
3.4	100	400	6.07	9.30	80
3.5	100	500	0.61	9.08	44
3.6	100	600	7.39	3.65	178
3.7	100	700	9.35	3.63	139
3.8	100	800	15.13	3.15	164
3.9	100	900	2.87	6.77	263
3.10	100	1000	6.58	2.11	93
5.1	200	2000	13.82	8.90	93
5.2	200	2000	15.98	0	176
5.3	200	2000	15.90	6.22	96
5.4	200	2000	14.95	0.78	148
5.5	200	2000	15.90	6.19	97
5.6	200	2000	12.38	5.31	99
5.7	200	2000	19.18	0.97	146
5.8	200	2000	16.25	7.89	183
5.9	200	2000	8.74	3.37	123
5.10	200	2000	12.29	6.03	92

Table 3. Average time per application of DUAL 1 or 2 and SGRAD

DUAL 1 or 2			SGRAD		
No, of times used*	Total time** spent	Average time for one application	No. of times used*	Total time** spent	Average time for one application
1,638	207.4	0.127	181	623.3	3.44

* Total for all 23 problems of Table 2

** DEC 20/50 seconds

Table 4. Cutting plane algorithm with and without SGRAD

Problem data			Without SGBAD		With SGRAD*	
No.	m	n	No. of cuts	Time**	No. of cuts	Time**
1.1	15	32	3	0.30	0	0.73
1.2	30	30	12	0.58	0	0.77
1.3	30	40	14	0.77	18	1.28
1.4	30	50	28	1.44	10	1.57
1.5	30	60	115	9.44	0	0.76
1.6	30	60	77	4.75	29	2.00
1.7	30	70	>438	>120.00	0	1.27
1.8	30	70	>480	>120.00	0	0.87
1.9	30	80	211	23.35	0	1.12
1.10	30	80	47	3.40	6	1.23
1.11	30	90	>406	>120.00	72	6.60
1.12	30	90	>496	>120.00	0	1.16

* SGRAD applied at first and every 10-th iteration

** DEC 20/50 seconds

Table 3 shows the average time needed for one application of the heuristic DUAL 1 or 2, and one application of SGRAD, with the averages taken over all applications to all of the 23 problems of Table 2. The comparison shows SGRAD to be about 27 times (3.44:0.127 = 27) as time consuming as DUAL 1 or 2. The factor 27 is$_y$.however, an average: as mentioned earlier, the first application of SGRAD to a problem is about 6 times as expensive as subsequent applications to the same problem (with added inequalities); in particular, the first application of SGRAD requires about 100 times more time than an application of DUAL 1 or 2, while subsequent applications to amended versions of the same problem require on the average 15 times as much time as DUAL 1 or 2.

Finally, to support our contention that in spite of these great time differences the use of SGRAD pays off, we show in Table 4 the outcome of the cutting plane procedure with and without SGRAD, on a set of 12 set covering problems taken from the literature and described in section 8. These problems, all of which except for 1.1 have unit costs, were particularly hard for the cutting plane algorithm without SGRAD, which failed to solve 4 of them within a time limit of 2 minutes.

6. Fixing Variables and Generating Cuts.

Every time a new solution u to (D) is obtained either by one of the heuristics or by subgradient optimization, the algorithm searches for variables x_j such that $s_j \geq z^\wedge - ue$, and fixes them at 0, removing the corresponding indices from N.

Intuitively, one would be inclined to think that this feature of the algorithm becomes operative only after many iterations, when the gap $z_{\overline{U}}$-$z_{\underline{L}}$ has been narrowed down considerably. This, however, was not the

case on the randomly generated problems that we solved. Substantial numbers of variables were usually fixed at 0 quite early in the procedure, and by the time the first branching occurred, the number of variables left was almost always close to the number m of initial constraints, as can be seen from the data of section 8.

To generate a cut, the algorithm uses a subroutine that implements the procedure discussed in [2]. Let x be the current cover, S(x) and T(x) defined as above, and z_U the current upper bound.

Step 0. Set W = 0, S = £jeS(x)|s_jX)}, y = ue, t = 1, and go to 1.

Step 1. Let

$$v_t = \min \{\max_{j \in S} s_j, \min_{j \in S} \{s_j | s_j \geq z_{TT} - y\}\},$$

$$J = [j_\epsilon S \ | s_j = v_t\}, \quad Q \ll \{jcNJs_j \geq v_t\}, \quad M_J - \bigcup_{j \in J} M_j.$$

Choose i(t) such that

$$|N_{i_k t_j} AQUW| * \min_{i c T(x) r M_T} |N_L, \backslash Q 0 W|$$

and let (j(t)} = JfTH$_{1(t)}$.

The.n set W - W\J($N_{i(t)}$\Q), y - y + s_j(t). If y $\geq z^\wedge$, go to 2. Otherwise set S - S\[j(t)},

$$s_j \leftarrow \begin{cases} s_j - s_{j(t)} & j \epsilon Q \cap N_{i(t)} \\ s_j & \text{otherwise,} \end{cases}$$

t «- t + 1, and go to 1.

Step 2. Add to (SC) the inequality

$$\sum_{j \epsilon W} x_j \geq 1.$$

This procedure terminates after a number of iterations equal to the number of jeS(x) such that s_j > 0, with an inequality satisfied by every

cover better than the one associated with z_U, and violated by the last cover x. Typically, the cuts tend to become successively stronger during the procedure, the last few cuts often having just one or two l's. The total number of cuts required to solve a m x n problem increases with both m and n. For the randomly generated sparse problems solved in our experiment the number of cuts needed was typically less than 3m or n/3, as can be seen from the results of section 8. This of course refers to the number of cuts required when the cuts are used within the framework of our algorithm in the class discussed here, which also uses implicit enumeration. The cuts by themselves, without branching (but with .periodic use of SGRAD) were able to solve all of the 20 test problems from the literature that we could obtain, and all but one of the 10 test problems with m = 100 and $100 \leq n \leq 1000$ that we generated, as shown in section 8. As to the larger problems, six of the ten 200 x 1000 problems and four of the ten-200 x 2000 problems were solved by cutting planes only, without branching (see section 8 for details).

7. Branching and Node Selection.

As mentioned in section 2, we branch whenever the gap $z_U - z_L$ decreases by less than e » 0.5 during a sequence of 4a iterations, where or is the frequency of applying SGRAD (every a-th iteration). The actual rule we use is slightly more sophisticated: if for 3a iterations the decrease in $z_U - z_L$ is less than e^s 0.5, then we branch at the first iteration where the current (say the k-th) dual solution u gives a bound greater than a weighted sum of the earlier bounds, namely where

$$u^k e > \sum_{i=1}^{k-1} (\cdot)^{k-i} u^i e.$$

If this requirement is not met for any of the next α iterations, and the decrease in the gap is still less than $\epsilon = 0.5$, then (i.e., after a total of 4α iterations) we branch.

We use two branching rules intermittently. The first one is based on a disjunction (3), which is strenghened to (7) when used for branching, so as to partition the feasible set:

$$(7) \qquad \bigvee_{i=1}^{p} (x_j=0,\ j\in Q_i\ ;\ \sum_{j\in Q_k} x_j \geq 1,\ k=1,\ldots,i-1)$$

The sets Q_i for the disjunction (7) are constructed by a procedure similar to Step 1 of the cut generating routine. The use of the same criterion (of minimizing $N_{h(i)}\backslash Q_i$) as in the cut generating routine is motivated by the attempt to guarantee that each subproblem created by the branching will have at least one inequality with as few 1's as possible.

The second branching rule is a variation of the one proposed by J. Etcheberry [4]. We choose two row indices $i,k\in M$, such that i is the last element in the ordered set M, and $k \neq i$, with

$$|N_k \cap N_i| = \min_{N_h \cap N_i \neq \emptyset} |N_h \cap N_i|,$$

and then branch on the disjunction

$$(8) \qquad (x_j=0,\ j\in N_i \cap N_k) \vee \left(\sum_{j\in N_i \cap N_k} x_j \geq 1\right).$$

Whenever $|N_i \cap N_k| = 1$, which is usually the case, the second term becomes $x_j=1$, where $\{j\}=N_i \cap N_k$, and (8) becomes a special case of the usual branching dichotomy $(x_j=0)\vee(x_j=1)$. However, a comparison of the rule based on (8) with the usual branching dichotomy combined with a choice rule for the branching variable different from the one used here, has shown (8) to be on the average considerably better.

The choice between rule 1 (using the disjunction (7)) and rule 2 (using the disjunction (8)) is dictated by the following considerations. Since none of the two rules dominates the other, i.e., at certain nodes it may be more advantageous to use rule 1, whereas at other nodes (of the same search tree) rule 2 may be preferable, we introduce a measure of relative efficiency of rule 1 (as compared to the usual dichotomy), and then choose rule 1 whenever it meets the efficiency requirements. With the traditional dichotomy, the k-th level of the (binary) search tree contains 2^k nodes, with k variables fixed at each node, i.e., a total of $k2^k$ variables fixed. In other words, in order to fix $f = k2^k$ variables by generating a breadth-first search tree, one has to break up the feasible set into p=2 subsets. Substituting for k in the expression for f, we find that with the usual dichotomy, breaking up the feasible set into p subsets (all on the same level of the search tree) makes it possible to fix a total of $f - p\log_2 p$ variables. Therefore, in order to use branching rule 1, i.e., the disjunction (7), which breaks up the feasible set into p subsets, we require that

(i) $\quad i_i|Q_j| \geq cp\,p\log_2 p,$

i.e., that the number of variables fixed on all branches be greater than cp times $p\log_2 p$. As to the value of the parameter cp, after some experimentation we found that $cp \leq 1$ implies that disjunction (7) will be preferred to (8) about 2/3 of the time, while $cp \geq 3$ implies the opposite. A judicious mix of the two rules requires $1 \leq cp \leq 3$. The current version of the algorithm uses cp = 1.

Besides condition (i), we have also found it useful to require that (ii)

there be at most one singleton among the sets Q_i, i=1,...,p, and that (iii) p not exceed a specified constant, which we usually set to 8. Whenever conditions (i), (ii) and (iii) are met, we use disjunction (7); otherwise we use disjunction (8).

Our node selection rule is LIFO; i.e., whenever available, we choose one of the nodes created by the last branching. When rule 1 is used, we choose the p nodes created by disjunction (7) in order of decreasing $|Q_i|$; and when rule 2 is used, we choose first the node defined by $x_j=0$, $j \in N_1^{HN}$, then the other. When a node is fathomed, we first look for an unfathomed brother node, and if none is available, we go to the father node.

Table 5 contains information concerning our branching rules. The problems listed are all those among the 200 X 1000 and 200 x 2000, 2% density problems, (i.e., among the problems in sets 4 and 5), whose solution required branching. The criterion for choosing the branching rule was the one described above, with cp = 1. For each problem, the table gives the number of branchings according to rule 1 and rule 2. Further, each branching according to rule 1 is described by a sequence of numbers in parantheses; where the length of the sequence is the number of branches created, and each number in the sequence is the number of variables fixed at 0 on the corresponding branch. Thus, for solving problem 4.4 (with m = 200, n » 1000, density 2%), the code branched 3 times, using each time rule 1 (disjunction (7)). The first branching created 5 new nodes (subproblems), with 58 variables fixed at 0 on the first branch, 53 on the second, 44 on the third, etc.

A typical search tree is illustrated in Fig. 2: it is the one corresponding to problem 5.1. The symbol (f) means a branching based on rule 2. The numbers on the arcs stand for the variables fixed by branching rule 1. At node 0, the algorithm generates 30 cuts, then branches according

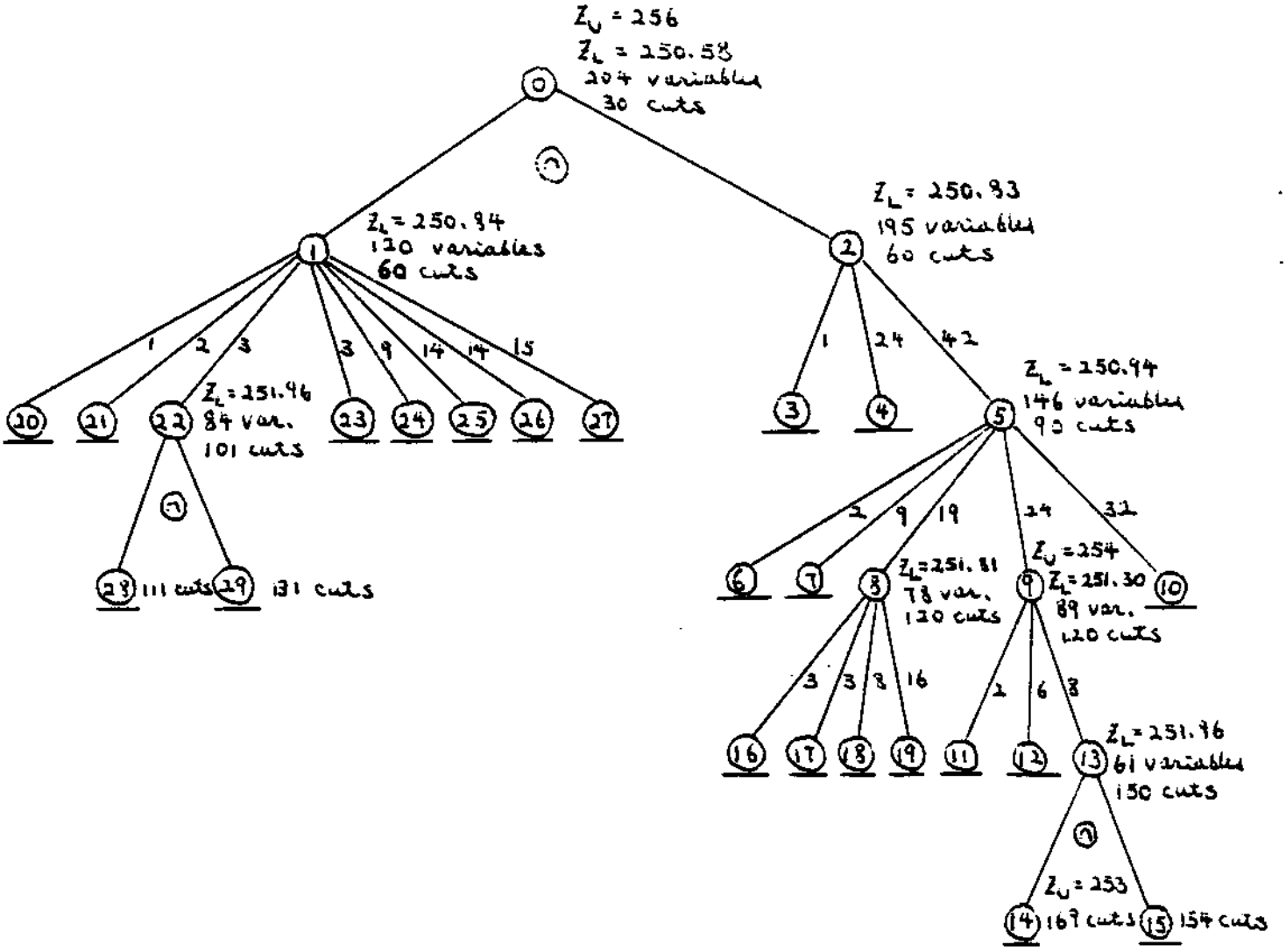
Z_U = 256
Z_L = 250.58
204 variables
30 cuts
0
1
Z_L = 250.84
120 variables
60 cuts
2
Z_L = 250.93
195 variables
60 cuts
1 2 3 3 9 14 14 15
1 24 42
10 21 22 23 24 25 26 27
3 4
Z_L = 251.96
84 var.
101 cuts
Z_L = 250.94
146 variables
90 cuts
7
2 9 19 24 32
28 111 cuts 29 131 cuts
6 7 8 9 10
Z_L = 251.81
78 var.
120 cuts
Z_U = 254
Z_L = 251.30
89 var.
120 cuts
3 3 3 16 2 6 8
16 17 18 19 11 12 13
Z_L = 251.96
61 variables
150 cuts
Z_U = 253
14 169 cuts 15 154 cuts

Fig. 2.

Table 5. Information on the use of the two branching rules

Problem data			No. of branchings according to		No. of variables fixed on each branch, for every branching according to Rule 1
No.	m	a	Rule 1	Rule 2 i	
3.4	200	1000	3	0	(58,53,44,15,6), (15,7,5), (15,6,5,4)
3.6	200	1000	6	1	(20,10,8,7,6,1),(24,12,10,9,5,3,1), (7,5,2,2), (14,4,2,1), (16,14,11,4,4), (6,1)
3.8	200	1000	2	2	(26,13,7,5,2,2), (8,7,1)
3.9	200	1000	6	5	(40,21,10,9,8,4,2),(18,15,10,3,2,1), (14,13,13,13,2), (15,4,2), (25,1), (10,3,1)
4.1	200	2000	5	3	(42,24,1),(15,14,14,9,3,3,2,1), (32,24,19,9,2), (8,6,2), (16,8,3,3)
4.4	200	2000	9	3	(35,33,33,27,6), (50,26,3,2), (40,6,5,2,1), (16,10,5,4,1), (20,16,10,5,1), (41,12,9,5,3) (7,7,6,2,1), (21,4,2), (30,16,5,3,2)
4.7	200	2000	2	4	(5,1), (36,11,5,5)
4.8	200	2000	4	2	(35,9,3,3,2,1), (23,8,8,3,2), (9,4,4,2) (20,13,9,5,3,3,3,2)
4.9	200	2000	1	0	(44,20,17,10,3,2)

to rule 2, creating 2 new nodes. Prior to branching, the upper and lower bounds are z_U=256 and z_L=250.58 respectively, and there are 204 variables, i.e., 1796 variables have been fixed at 0. Next the algorithm selects node 2, generates another 30 cuts, improves the lower bound to z_T=250.83, and fixes another 9 variables before branching, leaving 195. This time the branching is according to rule 1, and 3 new nodes are created, with 1, 24 and 42 new variables fixed at nodes 3, 4 and 5 respectively. Node 5 is chosen next, etc.

8. Computational Experience with the Algorithm as a Whole.

We have tested the algorithm as a whole on 6 sets of problems, that

we now describe. The problems are labelled with two numbers separated by a dot. The first number is the set to which the problem belongs, the second one distinguishes the problems within the same set. Thus 2.3 is the third problem in set 2.

Sets 1 and 2, containing 12 and 8 problems respectively, are from Salkin and Koncal [9], who account for their origin as follows. Problem 1.1 was obtained from C. E. Lemke, problem 1,8 from IBM Buffalo, and the remaining problems in set 1 from A. M. Geoffrion. All the problems in set 1, except for 1.1, have unit costs. They all have randomly generated coefficient matrices with 7% density.

Problem 2.1 is attributed by Salkin and Koncal [9] to American Airlines, problems 2.6 and 2.7 to IBM New York, while the remaining problems in set 2 were randomly generated by H. M. Salkin. The problems in set 2 have coefficient matrices whose density varies between 2% and 11%, and randomly generated costs in the range [0,99].

Sets 3, 4, 5 contain 10 problems each, randomly generated by the second author, with coefficient matrices of *2%* density, subject to the requirement that every column has at least one, and every row at least two, nonzero entries. The costs are randomly generated from the range [1,100].

Finally, set 6 contains 5 problems, also randomly generated by the second author subject to the same conditions, with costs from the same range, but with coefficient matrices of 5% density.

Table 6 compares the performance of our algorithm with two other procedures, that of Salkin and Koncal [9], and of Lemke, Salkin and Soielberg [8], on the 20 Salkin-Koncal problems (sets 1 and 2). The procedure used by Salkin and Koncal is Gomory's all integer cutting plane algorithm, while the

Table 6. Comparison of algorithms on 20 problems from the literature

Profcilem data			Salkin-Koncal [9] Time*	Lemke-Salkin-Spielberg [8] Time**	Algorithm of section 2 (without branching)		
						Time***	
No.	m	n			No. of cuts	Total	SGRAD
l.l	15	32	0.51	2.7	0	0.73	0.46
1.2	30	30	0.41	5.3	0	0.77	0.46
1.3	30	40	0.78	19.7	18	1.28	0.63
1.4	30	50	16.33	21.6	10	1.57	1.01
1.5	30	60	2.47	} 18.0+	0	0.76	0.36
1.6	30	60	10.07		29	2.00	0.96
1.7	30	70	5.66	) 20.4+	0	1.27	0.94
1.8	30	70	4.68		0	0.87	0.39
1.9	30	80	5.99	j31.2+	0	1.12	0.73
1.10	30	80	6.83		6	1.23	0.73
1.11	30	90	16.99	j 30.6+	0	1.16	0.79
1.12	30	90	19.16		72	6.60	1.76
1.1	104	133	5.70	424.0	22	8.03	4.61
2.2	200	413	26.71	625.9	6	12.00	7.10
2.3	200	300	15.90	461.3	0	6.10	4.12
2.4	200	500	22.70	803.5	0	6.23	3.74
2.5	50	450	>120.00	144.5	0	2.06	1.24
2.6	36	455	18.64	35.5	0	1.76	0.55
2.7	46	683	117.85	56.9	10	5.94	3.35
2.8	50	905	117.87	670.0	0	5.12	3.34

* UNIVAC 1108 seconds (about the same speed as DEC 20/50)
IBM 360/50 seconds (4-5 times slower than DEC 20/50)
DEC 20/50 seconds

\+ Average time for the two problems of the same size

++ Time limit exceeded

algorithm of Lemke, Salkin and Spielberg is a specialized implicit enumeration procedure, with an imbedded linear program. Our algorithm solved each of these problems without branching, and on the larger problems of set 2 its performance was an order of magnitude better than that of the other two procedures. Note that about 1/2 of the total time (in some cases more, in others less than 1/2) was spent on SGRAD. The number of times the subgradient procedure was used can be calculated by dividing the number of cuts by 10, and adding 1 to the result. The rest of the time was spent on primal and dual heuristics and cut generation.

Note also that 7 of the 12 problems in set 1, and 5 of the 8 problems in set 2, did not require any cuts. This does not necessarily imply that the linear programming relaxation of (SC) had an integer solution in these cases, but it does imply that the gap between the linear programming optimum and the integer optimum was less that 1. This small gap apparently did not make most of these problems easy for the other two procedures, as evidenced by their performance on problems 1.11, 2.3, 2.4, 2.5, 2.7 and 2.8. Our procedure, however, can take advantage of the small gap due to the use of the primal heuristics.

Table 7 shows the performance of our algorithm, still without branching, on the 10 randomly generated problems of set 3. Note that 6 of these problems did not require cuts. As to the remaining 4 problems, one of them (3.5) required only 4 cuts, while the other three required large numbers of cuts and one of them (3.8) was actually not solved within the time limit of 5 minutes. Had we used the full algorithm (with branching) on these 3 problems, the number of cuts and the time needed would in all likelihood have been smaller. However, we ran the full version of the

Table 7. Algorithm of section 2 without branching

Problem data			No. of	Time*	
No.	m	n	cuts	Total	SGRAD
3.1	100	100	0	1.21	0.57
3.2	100	200	0	1.26	0.48
3.3	100	300	0	3.04	1.85
3.4	100	400	0	3.22	2.13
3.5	100	500	4	3.95	1.51
3.6	100	600	146	42.61	19.03
3.7	100	700	59	24.03	14.38
3.8**	100	800	682	>300.00	65.06
3.9	100	900	0	13.27	11.18
3.10	100	1000	0	8.30	6.00

* DEC 20/50 seconds
** Time limit exceeded

algorithm only on problem 3.8, with the outcome that the problem was solved in 92.24 seconds, with a search tree of 30 nodes and a total of 362 cuts.

Table 8 describes the performance of our algorithm (in its complete version) on the 20 randomly generated test problems of sets 4 and 5. It shows the value z_{opt} of the optimum; the upper and lower bounds, as well as the number of variables left, before the algorithm first branched; the number of nodes and cuts, and, finally, the total time and the time spent on subgradient optimization. Six out of the 10 problems in set 4, and 4 out of the 10 problems in set 5, did not require any branching. Of those problems that did not require branching, 4 in set 4 and 2 in set 5 did not require cuts either. These 6 problems had a gap of less than 1 between the linear programming optimum and the integer optimum, and for some of them the linear programming optimum may actually be integer (since we don't use the simplex method, we do not necessarily discover this when we solve a problem).

Table 8. Complete algorithm on 2% density problems

(m = 200; n = 1000 for set 4, n = 2000 for set 5)

No.	z_{opt} = z_{up} = z_u	Before first branching			No. of nodes in search tree	No. of cuts	Time*	
		z_U	z_L	No. of variables left			Total	SGRAD
4.1	429	429	429	0	1	20	31.88	24.04
4.2	512	512	512	0	1	0	18.62	13.54
4.3	516	516	516	0	1	22	26.02	16.29
4.4	494	507	493.77	243	13	119	81.28	50.34
4.5	512	512	512	0	1	0	13.33	8.40
4.6	560	572	556.83	258	31	580	316.22	179.28
U.7	430	430	430	0	1	0	19.59	14.11
i4.8	492	492	478.78	99	14	274	167.15	110.52
;4.9	641	648	636.57	224	37	686	416.46	215.41
4.10	514	514	514	0	1	0	22.34	16.50
5.1	253	256	250.58	204	30	473	327.89	181.20
;5.2**	307+	315	299.32	408	51++	625	>600.00	206.81
5.3	226	226'	226	0	1	0	26.87	15.83
5.4	242	247	240.29	258	49	765	393.22	133.47
5.5	211	211	211	0	1	15	38.73	24.31
5.6	213	213	213	0	1	10	32.71	19.47
5.7	293	296	291.02	173	15	298	248.65	152.62
5.8	288	288	286.09	125	28	413	241.42	108.72
5.9	279	281	276.21	181	7	118	140.61	94.60
5.10	265	265	265	0	1	0	25.89	15.38

* DEC 20/50 seconds
** Time limit exceeded
\+ Best solution found before exceeding time limit
-H- 51 nodes generated, of which 30 fathomed

As to the remaining problems in sets 4 and 5, they were solved with a reasonable computational effort, in terms of the number of nodes in the search tree (never more than 50), the number of cutting planes (several hundred at most), as well as in terms of computing time (between about 20 seconds and 7 minutes), except for problem 5.2, which could not be solved within the time limit of 10 minutes. The best solution found for this problem, with a value of 307, is at most 2.33% worse than the optimum, since <299.32> is the lower bound found before the first branching occurred.

From Table 8 one can see again that the subgradient procedure in most cases takes up between 1/2 and 2/3 of the computational effort. The time needed to solve a problem strongly depends on the number of variables left before one has to branch: there is a high positive correlation between this number, and the number of nodes in the search tree. There is an even higher correlation, of course, between the number of nodes.in the search tree and the total time needed to solve the problem. On the other hand, cuts are cheap to generate, and the number of cuts affects the total time mainly through the fact that after every *a* cuts the subgradient procedure is applied (which in turn is costly). This can be seen, for instance, by looking at the 4 problems in set 5 that required no branching. Problems 5.3 and 5.10, which required no cuts either, took 26-27 seconds to be solved. Problems 5.6 and 5.5, which required 10 and 15 cuts respectively, required only 33 and 39 seconds respectively, i.e., about-1.24-1.45 times the time required for problems 5.3 and 5.10. The reason for this is that the subgradient procedure was applied once to each of problems 5.3 and 5.10, and twice to each of problems 5.6 and 5.5 The reason the computational effort increased less than twice for the second pair of problems, is that

SGRAD is the most time consuming when applied for the first time to a problem, as discussed in section 5.

As can be seen from the above computational experience, the algorithm discussed here is a reasonably reliable, efficient tool for solving large, sparse set covering problems, as well as for finding good approximate solutions to problems that are too hard to be solved exactly. However, the strength of the family of cuts from conditional bounds strongly depends on sparsity. As problem density increases, the strength of the cuts diminishes, and so does the efficiency of this algorithm, at least in its versions that we have tested. For relatively small problem sizes, the algorithm can cope well with somewhat higher density, as illustrated by problem sets 1 (7% density) and 2 (2% - 11% density), on which it clearly outperformed the other two procedures that had been tried on those problems. But for larger problems, this is unlikely to be the case. To see how fast the algorithm's performance declines with problem density, we have run the code on a set of 5 randomly generated 200 X 1000 problems with 5% density (problem set 6). The results are shown in Table 9.

Table 9. Complete algorithm on 5% density problems

(m « 200, n = 1000)

No.	Best z_U	Before first branching			No. of nodes in search tree		No. of cuts	Time*	
		«D	z_L	No. of variables left	Generated	Fathomed		Total	SGRAD
6.1+	141	143	132.07	189	143	116	2160	>1800	842
6.2+	149	157	139.83	244	173	153	2412	>1800	920
6.3	145	145	139.60	117	163	163	2481	1548	787
6.4	131	135	128.05	147	21	21	275	194	106
6.5+	161	173	152.52	278	161	140	2425	>1800	884

* DEC 20/50 seconds

\+ Time limit exceeded

Two of the 5 problems were solved within the 30 minutes time limit. By no means accidentally, these two happen to be the problems with the smallest numbers of variables left before branching. For the remaining 3 problems, the best solution found is guaranteed to be within 6%, 6.4% and 5.2% respectively, of the optimum, though it could of course be much closer. Though the algorithm exceeded the time limit before finishing 3 of the five problems, from the data of Table 6 (ratio of fathomed versus unfathomed nodes, ratio between numbers of variables left before branching for the unsolved and the solved problems), it seems that all the problems could be solved by a not too drastic extension of the time limit. In general, the data of Tables 7, 8 and 9 show a certain reliability of the procedure on randomly generated problems: the results are not wildly erratic, as it so often happens with integer programming algorithms.

It is possible that a different version of our approach, that would generate a larger number of cuts but retain only the stronger ones, and rely more heavily on branching from a disjunction (7), would be more successful on higher density problems. It is also possible, even highly probable, that a different backtracking rule, that would lead to earlier processing of the nodes with the best lower bounds, would provide a considerably better bound on the quality of the solution obtained when the procedure stops prematurely because of the time limit. These ideas, however, have not yet been tested.

References

[1] E. Balas, "Set Covering with Cutting Planes from Conditional Bounds." MSRR No. 399, Carnegie-Mellon University, July 1976.

[2] E. Balas, "Cutting Planes from Conditional Bounds: A New Approach to Set Covering." MSRR No. 437, Carnegie-Mellon University, July 1979.

[3] V. ChWtal, "A Greedy Heuristic for the Set Covering Problem." Publication 284, Department d^f Infonnatique et de Recherche Op^rationnelle, University de Montreal, 1978.

[4] J. Etcheberry, "The Set Covering Problem: A New Implicit Enumeration Algorithm." Operations Research, 25, 1977, p. 760-772.

[5] J. L. Goffin, "On the Convergence Rates of Subgradient Optimization Methods." Mathematical Programihg« 13, 1977, p. 329-348.

[6] M. Held, P. Wolfe and H. P. Crowder, "Validation of Subgradient Optimization." Mathematical Programming. £, 1974, p. 62-88.

[7] A. Ho, "Worst Case Analysis of a Class of Set Covering Heuristics." GSIA, Carnegie-Mellon University, June 1979.

[8] C. E. Lemke, H. M. Salkin and K. Spielberg, "Set Covering by Single Branch Enumeration with Linear Programming Subproblems." Operations Research, 19, 1971, p. 998-1022.

[9] H. M. Salkin and R. D. Koncal, "Set Covering by an All-Integer Algorithm: Computational Experience." ACM Journal, 20, 1973, p. 189-193.