SIAM J. CONTROL
Vol. 7, No. 2, May 1969

INTEGER PROGRAMMING OVER A FINITE ADDITIVE GROUP*

FRED GLOVER†

Abstract. An algorithm is given for solving an integer program over an additive group. Computation times appear to grow more favorably with increases in the number of variables and group elements than with the dynamic programming approach proposed by Gomory. A new property satisfied by optimal solutions to the group problem is established by reference to the structure of the algorithm. Extension of the algorithm to the general integer programming problem is developed in a sequel.

1. Introduction.

In this paper we give an algorithm for solving the problem:

$$\text{(I)} \quad \text{Minimize} \quad \sum_{j=1}^{n} c_j x_j$$

$$\text{subject to} \quad \sum_{j=1}^{n} \alpha_j x_j = \alpha_0, \quad x_j \geqq 0 \quad \text{and integer,} \quad j = 1, \cdots, n,$$

where the c_j are nonnegative scalar constants, and α_0 and the $\alpha_j, j = 1, \cdots, n$, are elements of a finite additive group. We sometimes also refer to $\sum_{j=1}^{n} c_j x_j$ in matrix notation as cx, where $c = (c_1, c_2, \cdots, c_n)$ and $x = (x_1, x_2, \cdots, x_n)$. An example of (I) is the problem:

$$\text{(I')} \quad \text{Minimize} \quad 3x_1 + 7x_2 + 4x_3$$
$$\text{subject to} \quad 8x_1 + 3x_2 + 5x_3 \equiv 6 \pmod{11},$$
$$x_1, x_2, x_3 \geqq 0 \quad \text{and integer.}$$

Alternatively, consider the linear integer programming problem:

$$\text{(II)} \quad \text{Minimize} \quad \sum_{j=1}^{n} c'_j x'_j$$

$$\text{subject to} \quad \sum_{j=1}^{n} a'_{ij} x'_j + y'_i = b'_i, \quad i = 1, \cdots, m,$$

$$x'_j, y'_i \geqq 0 \quad \text{and integer for all } i, j,$$

where c'_j, a'_{ij} and b'_i are integer constants.

Applying the simplex method to (II) without the integer restriction on the x'_j and y'_i yields an equivalent representation:[1]

$$\text{(II')} \quad \text{Minimize} \quad \sum_{j=1}^{n} c_j x_j$$

$$\text{subject to} \quad \sum_{j=1}^{n} a_{ij} x_j + y_i = b_i, \quad i = 1, \cdots, m,$$

* Received by the editors November 16, 1966, and in final revised form December 4, 1968.

† Department of Statistics and Operations Research, The University of Texas at Austin, Austin, Texas 78712. This research was supported in part by the U.S. Office of Naval Research under Contract NONR 760(24) NR 047-048.

[1] It is assumed an optimal continuous solution exists.

where the c_j, a_{ij} and b_i are rational, the x_j and y_i are obtained by renaming the x'_j and y'_i (e.g., $x_1 = y'_2, x_2 = x'_4$, etc.) and $c_j \geqq 0, b_i \geqq 0$ for all i and j. Problem (II') then becomes an instance of (I) by dropping the restriction $y_i \geqq 0$, giving:[2]

$$\text{(III)} \quad \text{Minimize} \quad \sum_{j=1}^{n} c_j x_j$$

$$\text{subject to} \quad \sum_{j=1}^{n} a_{ij} x_j \equiv b_i \,(\text{mod } 1), \quad i = 1, \cdots, m.$$

The significance of (I) lies in the fact that under certain conditions, its solution gives an optimal solution to (II) (see § 7).

It frequently happens that all of the constraints

$$\sum_{j=1}^{n} a_{ij} x_j \equiv b_i \,(\text{mod } 1), \quad i = 1, \cdots, m,$$

can be replaced by a single constraint

$$\sum_{j=1}^{n} \alpha_j x_j \equiv \alpha_0 \,(\text{mod } D),$$

where D, α_0 and the α_j are integer constants, so that (III) becomes equivalent to the class of problems whose form is illustrated by (I').

A variety of such problems containing from 50 to 1500 variables and from 100 to 4500 group elements have been solved with the algorithm of this paper. Computational results are reported in § 8.

2. Methods for solving (I). Two methods have been proposed for solving (I) other than the method of this paper. The first, due to Ralph Gomory [4], is based upon a dynamic programming recursion for the knapsack problem developed by Gilmore and Gomory [1]. Refinements in this approach have also been suggested by W. W. White [8]. Computation time is estimated to be proportional to nD, where n is the number of variables and D the order of the additive group.

The second method, due to Jeremy Shapiro [6], is based on a dynamic programming recursion for the knapsack problem developed by Shapiro and Wagner [7]. No estimates of computation time are available for this method, although the method appears intuitively to be quite promising.

The method of this paper takes a different approach that departs from the dynamic programming framework. An appeal to the structure of the algorithm establishes a new property satisfied by optimal solutions to (I) (see § 7). Computation times for the method, as reported in § 8, appear to depend somewhat more favorably on n and D than a direct proportionality to nD.

This method can be considered a dual method, in that optimal solutions are generated for a sequence of right-hand sides, until a feasible solution is found.

---

[2] That (III) is in fact an instance of (I) derives from the elegant theory developed by Gomory in [3], [4].

3. Simplified version of the algorithm. We describe three versions of the algorithm in this and the next two sections, beginning with the simple and working toward the more complex (and more efficient). Formal justification of the principal ideas and claims is deferred to § 6.

To begin with, we eliminate degeneracy by assuming $c_j > 0$. If $c_j \geqq 0$ is rational, this can be ensured as follows: multiply c by a positive integer large enough to make all components integer in the resulting new c. Then replace all $c_j = 0$ by $c_j = 1/P$, where P is a number such that $\sum_{c_j=0} x_j \leqq P - 1$. Thus any feasible adjustments of $\{x_j \mid c_j \text{ was zero}\}$ cannot change the objective as much as a unit change in any x_j. In particular, it suffices to let $P = D$ (see § 7).[3]

The algorithm generates a sequence of solutions (vectors of nonnegative integers) $x(1), x(2), \cdots, x(i), \cdots$, where $x(i)$ is the vector $(x_1^i, x_2^i, \cdots x_n^i)$. Associated with $x(i)$ is the "cost" $c(i) = \sum_{j=1}^n c_j x_j^i$ and the group element $\alpha(i) = \sum_{j=1}^n \alpha_j x_j^i$. If $\alpha(i) = \alpha_0$, then $x(i)$ is a *feasible* solution to (I). Each $x(i)$ is generated from an earlier solution $x(p)$, called the *predecessor* of $x(i)$, by incrementing one of the components of $x(p)$ by one. Thus if x_r^p is the component of $x(p)$ that is incremented to give $x(i)$, then we may write $x(i) = x(p) + e_r$, where e_r denotes the vector with 1 in the rth component and 0's elsewhere. We observe that $c(i) = c(p) + c_r$ and $\alpha(i) = \alpha(p) + \alpha_r$.

We construct the sequence of solutions to satisfy the following conditions:

(i) If $p \neq q$, then $x(p) \neq x(q)$.

(ii) If $p < q$, then $c(p) \leqq c(q)$.

(iii) $x(i)$ is an optimal solution to (I) when α_0 is replaced by $\alpha(i)$.

(iv) The solution sequence is finite, and $\alpha(i) = \alpha_0$ for some $x(i)$ if and only if problem (I) has a feasible solution.

If we alternately interpret the α_j as ordinary column vectors, our strategy in generating the $x(i)$ may be seen to correspond quite closely to the strategy of the dual simplex method in solving the ordinary linear programming problem. In fact, the successive basic solutions determined by the pivot rules of the dual simplex method satisfy exactly the same four conditions.

We shall introduce several of the fundamental ideas of the algorithm (in a simplified form) by means of an example. Consider the problem:

$$\begin{aligned} &\text{Minimize} && 3x_1 + 7x_2 + 4x_3 \\ &\text{subject to} && 8x_1 + 3x_2 + 5x_3 \equiv 6 \pmod{11} \end{aligned}$$

given in § 1 as an instance of (I).

Table 1 shows a sequence of solutions $x(i)$ generated by the algorithm.[4] Included in the table are the costs $c(i)$, group elements $\alpha(i)$, and the indices p_i and r_i from which one may verify the relations

$$x(i) = x(p) + e_r,$$

$$c(i) = c(p) + c_r,$$

$$\alpha(i) = \alpha(p) + \alpha_r$$

[3] Here (as earlier), and throughout the paper, we let D denote the number of elements in the additive group.

[4] The specific rules of the algorithm follow the example.

for $p = p_i$ and $r = r_i$, where p_i names the predecessor of solution i, and r_i names the variable which was incremented to get $x(i)$ from $x(p_i)$.

Note that the starting solution, $x(1)$, is the 0-vector. Because $x(1)$ has no predecessor, r_1 and p_1 have not been assigned values.

TABLE 1

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$c(i)$	0	3	4	6	7	7	8	9	10	10	11	11	12	12	13	13	14	14	14
$\alpha(i)$	0	8	5	5	2	3	10	2	10	0	7	8	10	4	7	8	4	5	6
r_i		1	3	1	1	2	3	1	1	1	1	2	1	3	1	1	1	1	2
p_i		1	1	2	3	1	3	4	5	6	7	3	8	7	9	10	11	12	6
x_1^i	0	1	0	2	1	0	0	3	2	1	1	0	4	0	3	2	2	1	0
x_2^i	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	1	0	1	2
x_3^i	0	0	1	0	1	0	2	0	1	0	2	1	0	3	1	0	2	1	0
				x				*	x	x		x	*		*	*	x	*	

To identify the contribution of each variable x_j to the generation of Table 1, we define a transition index t_j which names the next solution from which x_j will be incremented. That is, if $x(1), x(2), \cdots, x(k-1)$ denote the solutions currently generated, then $x(k)$ will be one of the solutions $x(t_1) + e_1, x(t_2) + e_2, \cdots, x(t_n) + e_n$. That is, t_j will be the predecessor the next time x_j gets incremented. All of the t_j are initially set equal to 1, so that $x(2)$ will be one of the solutions $e_1, e_2, \cdots, e_n$.

As soon as x_j is incremented, i.e., when $x(t_j) + e_j = x(k)$, then the predecessor name is changed. The next time x_j gets incremented, its predecessor will be $\bar{t}_j$ instead of t_j, where[5]

$$\bar{t}_j = \min \{i : i > t_j \text{ and } r_i \geqq j\}.$$

All that remains for the determination of $x(k)$ is the selection of the particular index r for which $x(k) = x(t_r) + e_r$. To do this we define a *next cost* $N_j = c(t_j) + c_j$ for each j. N_j is the cost $c(k)$ when $x(k)$ is generated from the predecessor $x(t_j)$, i.e., when

$$x(k) = x(t_j) + e_j.$$

Then we select the index r by

$$N_r = \min \{N_1, N_2, \cdots, N_n\}$$

and set $x(k) = x(t_r) + e_r$.

We summarize our foregoing remarks in the following description of the procedure as developed to this point.

[5] The reason for the stipulation $r_i \geqq j$ is to avoid duplications. For example, without this restriction the solution $x(5) = e_1 + e_3$ in Table 1 could be generated both as $x(2) + e_3$ and $x(3) + e_1$, since $x(3) = e_3$.

SIMPLIFIED ALGORITHM.

1. Begin with $x(1) = 0$ and $t_j = 1$ for $j = 1, \cdots, n$. Designate the solutions currently generated by $x(i)$, $i = 1, \cdots, k - 1$.

2. To generate the next solution $x(k)$, select r to be one of the j's, $j = 1, \cdots, n$, by the rule $N_r = \min(N_j)$. If more than one index j is a candidate for r by this criterion, let r be the smallest of these indices. Then let $x(k) = x(t_r) + e_r$.

3. Update t_r by setting it equal to its next value $_r$ (note $\bar{t}_r \leqslant k$) and repeat the foregoing process.

The reader may verify that this algorithm generates the sequence of columns of Table 1. To facilitate this verification, Table 2 supplies the successive values assumed by the t_j and the N_j.

The entries for portions of the table left blank are the same as the nearest preceding entries in the same column. The value of r ($N_r = \min\{N_j\}$) at each stage

TABLE 2

	j / c_j	1 2 3 / 3 7 4	
1	t_j	1 1 1	$r = 1$
	N_j	3 7 4	
2	t_j	2	$r = 3$
	N_j	6	
3	t_j	3	$r = 1$
	N_j	8	
4	t_j	3	$r = 1$
	N_j	7	
5	t_j	4	$r = 2$
	N_j	9	
6	t_j	3	$r = 3$
	N_j	11	
7	t_j	7	$r = 1$
	N_j	12	
8	t_j	5	$r = 1$
	N_j	10	
9	t_j	6	$r = 1$
	N_j	10	
10	t_j	7	$r = 1$
	N_j	11	
11	t_j	8	$r = 2$
	N_j	12	
12	t_j	6	$r = 1$
	N_j	14	
13	t_j	9	$r = 3$
	N_j	13	
14	t_j	14	$r = 1$
	N_j	16	
15	t_j	10	$r = 1$
	N_j	13	
16	t_j	11	$r = 1$
	N_j	14	
17	t_j	12	$r = 1$
	N_j	14	
18	t_j	13	$r = 2$
	N_j	15	

is indicated to the right of the appropriate portion of Table 2. It may be noted that the amount of computation required in going from one iteration to the next is very small.

This simplified procedure generates solutions that are unnecessary for solving (I), and a glance at Table 1 discloses a variety of them: $x(4)$, $x(8)$, $x(9)$, $x(10)$, $x(12)$, $x(13)$, $x(15)$, $x(16)$, $x(17)$ and $x(18)$. These solutions are *dominated* in the sense that other solutions generated earlier in the table give the same $\alpha(i)$ with as good or better values for $c(i)$. Since these solutions are superfluous, they can be dropped.

There is of course no gain in dropping the dominated solutions at this stage, since the work devoted to generating them has already been done. However, if each solution is checked to see if it is dominated before it is added to the table, then the outcome is somewhat different. The x's and *'s beneath Table 1 show the columns that would never have entered the table. The x's are attached to columns that would have been checked for inclusion in the table, but rejected, and the *'s are attached to columns that never would have been checked or generated at all since they are descendants of other dropped columns.

It is not evident that solutions can be dropped legitimately at the point at which they are discovered to be dominated, unless they are dominated by a solution with a strictly lower cost. In fact, it can be shown that dropping dominated solutions can cause the method never to generate a feasible solution to (I), let alone an optimal one, if an improper tie-breaking rule is used in the choice of r at instruction 2.

A disguised complexity in the process of dropping solutions arises from the fact that some of the t_j's can thereby become "undefined." On the other hand, from an ability to drop solutions also comes an ability to impose bounds on variables, thereby further limiting the number of solutions examined.

The procedural details for accommodating these facts are given in the next section.

4. Procedures for handling dominated solutions and upper bounds. To supplement our previous remarks we define a list $G(k)$, $k = 1, 2, \cdots, D$, where $G(k) = 0$ if none of the $x(i)$ currently generated gives $\alpha(i) = g_k$ (g_k denotes the kth group element). Otherwise, if $\alpha(p) = g_k$ for some p, then $G(k) = p$. $G(k)$ names the solution index (or "iteration") p for which the right-hand-side element, $\alpha(p)$, is g_k.

The use of the G-list in dropping dominated solutions is as follows. When preparing to generate the solution $x(k) = x(t_r) + e_r$, identify the group element g_h such that $g_h = \alpha(t_r) + \alpha_r$. Then $x(t_r) + e_r$ is permitted to be generated as $x(k)$ only if $G(h) = 0$, whereupon $G(h)$ is set equal to k. Otherwise, if $G(h) \geqq 1$, then $x(t_r) + e_r$ is dominated by the previously generated solution $x(i)$ for $i = G(h)$, and thus is not recorded in the table. Note that x_r might eventually be incremented even if $x(t_r) + e_r$ is dominated at iteration k. Therefore, whether or not it is dominated, the next step is to find the next value $\bar{t}_r$ for t_r. We now define

$$\bar{t}_r = \min \{i : i > t_r, r \leqq r_i \text{ and } G(q) = 0,$$

where g_q denotes the group element $\alpha(i) + \alpha_r\}$.

As already intimated, there may not be a next value for t_r that satisfies this definition. Thus, we introduce the set $T = \{j : t_j \text{ is defined}\}$. Initially, T contains all the j, $j = 1, \cdots, n$ (since $t_j = 1$ for all j). Thereafter, the composition of T can vary. But from the results of § 6, T cannot become empty unless (I) has no solution.

We now summarize these remarks by describing an algorithm for (I) that accommodates dominated solutions.

To begin, let $T = \{j : j = 1, \cdots, n\}$, $t_j = 1$ for all $j \in T$, $G(h) = 0$ for $h = 1, \cdots, D - 1$ and $G(D) = 1$, where g_D is the "0" group element, generated by the starting solution $x(1) = 0$. If $a_0 = 0$, the problem is trivially solved by $x(1)$.

Otherwise, we denote the solutions generated at the current stage of the method by $x(1), \cdots, x(k-1)$ and the next step is to generate $x(k)$.

ALGORITHM FOR (I).

1. If T is empty, problem (I) has no solution. Otherwise, identify the index r such that

$$N_r = \min_{j \in T} \{N_j\}.$$

If more than one j qualifies to be r, let r be the smallest of the qualifying indices.

2. Let g_h denote the group element given by $g_h = \alpha(t_r) + \alpha_r$.

(i) If $G(h) \geqq 1$, do nothing at instruction 2. Go to instruction 3.

(ii) If $G(h) = 0$, indicating that g_h has not previously been generated, generate the solution $x(k) = x(t_r) + e_r$ and let $G(h) = k$. If $\alpha(k) = \alpha_0$, $x(k)$ is optimal for (I) and the method stops.

3. Update t_r to its next value $\bar{t}_r$ (using the expanded definition of this section). If the updated value of t_r does not exist, remove r from T.

4. If a new solution $x(k)$ was *not* generated at instruction 2, then return to step 1 to pick up the next smallest N_j. But if a new $x(k)$ was generated, check whether any of the $j \notin T$ can be returned to T; i.e., whether $j \leqq r_k$ and $G(h) = 0$ for g_h given by $g_h = \alpha(k) + \alpha_j$. Let $t_j = k$ for all such j added back to T, and then return to step 1 to generate $x(k)$ for the next larger value of k.

We illustrate the algorithm above by applying it to the problem:

$$\begin{aligned} &\text{Minimize} \quad 3x_1 + 4x_2 + 5x_3 + 7x_4 \\ &\text{subject to} \quad 5x_1 + 9x_2 + 3x_3 + 4x_4 \equiv 1 \pmod{10}. \end{aligned}$$

Table 3 gives the sequence of solutions generated by the algorithm.

TABLE 3

i	1	2	3	4	5	6	7	8	9	10
$c(i)$	0	3	4	5	7	8	9	10	12	13
$\alpha(i)$	0	5	9	3	4	8	2	6	7	1
r_i		1	2	3	1	1	2	3	1	1
p_i		1	1	1	3	4	4	4	7	8
x_r^i	0	1	1	1	1	1	1	2	1	1
Σx_j^i	0	1	1	1	2	2	2	2	3	3

Notice that in place of recording the vector $x(i)$ for each column, as in the earlier example, we have instead recorded $\sum x_j^i$ and the value of the single variable x_r^i (for $r = r_i$). The formula for determining these values is as follows. Let $x(p)$ denote the predecessor of $x(i)$, i.e., $x(i) = x(p) + e_r$. Then $\sum x_j^i = \sum x_j^p + 1$, and $x_r^i = x_r^p + 1$ if $r_p = r$ and $x_r^i = 1$ otherwise.

The successive iterations of the algorithm that produced the columns of this table are summarized in Table 4. As before, entries for portions left blank are the same as the nearest preceding entries in the same column.

TABLE 4

	j	1	2	3	4	
	c_j	3	4	5	7	
	α_j	5	9	3	4	
1	t_j	1	1	1	1	
	N_j	3	4	5	7	$r = 1$
	$\alpha(t_j) + \alpha_j$	5	9	3	4	
2	t_j	*				
	N_j	*				$r = 2$
	$\alpha(t_j) + \alpha_j$	*				
3	t_j	3	3			
	N_j	7	8			$r = 3$
	$\alpha(t_j) + \alpha_j$	4	8			
4	t_j			4		
	N_j			10		$r = 1$
	$\alpha(t_j) + \alpha_j$			6		
5	t_j	4			*	$r = 4$
	N_j	8			*	$r = 1$
	$\alpha(t_j) + \alpha_j$	8			*	
6	t_j	*	4			$r = 2$
	N_j	*	9			$r = 2$
	$\alpha(t_j) + \alpha_j$	*	2			
7	t_j	7	7			
	N_j	12	13			$r = 3$
	$\alpha(t_j) + \alpha_j$	7	1			
8	t_j			*		
	N_j			*		$r = 1$
	$\alpha(t_j) + \alpha_j$			*		
9	t_j	8				
	N_j	13				$r = 1$
	$\alpha(t_j) + \alpha_j$	1				

The steps of the algorithm can be traced from the tables as follows. From step 1 of Table 4, $r = 1$, producing column 2 of Table 3. Thereupon, the next value for t_1 would ordinarily be 2, except that $\alpha(2) + \alpha_1 = 10 \equiv 0$, and 0 has already been generated ($\alpha(1) = 0$). Consequently, since there are no other possible values for t_1, it becomes undefined, as indicated by the asterisks in step 2.

At step 2, $r = 2$, producing column 3 of Table 3. t_1 becomes defined again ($t_1 = 3$) and the next value of t_2 is determined ($t_2 = 3$), as shown in step 3 of Table 4.

Step 4 of Table 4 is generated routinely. At step 5, however, $r = 4$ is indicated, except that $\alpha(1) + \alpha_4 = 4$ has by now been generated ($\alpha(5) = 4$), and hence the next permissible value is sought for t_4. There is none, and so t_4 becomes undefined (as indicated by the asterisks). The actual value of r at step 5 is therefore $r = 1$.

At step 6, $r = 2$ is indicated, but $\alpha(3) + \alpha_2 = 8$ has been generated ($\alpha(6) = 8$). Thus, the next value of t_2 is determined, giving $t_2 = 4$. In this case, N_2 is still minimum, and so $r = 2$ gives the correct value of r. Steps 7, 8 and 9 of Table 4 are determined similarly.

The optimal x-vector can be recovered as follows. Begin with $x = 0$ and i the index of the optimal (last) column of Table 3.

Let $x_r = x + x_r^i e_r$ (for $r = r_i$) and identify the group element $g_h = \alpha(i) - x_r^i \alpha_r$. From the G-list (or simply by scanning back through the table) locate the column i for which $g_h = \alpha(i)$ (i.e., set $i = G(h)$), and then repeat this procedure until $i = 1$.

The foregoing also works by replacing x_r^i with 1 at each step, so that the x_r^i values need not have been recorded for this purpose. Using either procedure, we see that an optimal solution is given by $x = (1 \quad 0 \quad 2 \quad 0)$.

The x_r^i and $\sum x_r^i$ values are not needed to recover the optimal solution, but can be used to serve another more fruitful purpose. Specifically, whenever a solution $x(t_r) + e_r$ is rejected as the next $x(k)$ at instruction 2, postpone this rejection, temporarily designating $x(k) = x(t_r) + e_r$. Then if $x_r^k = \sum x_j^k$, it follows that[6] $x(k) = x_r^k e_r$. Moreover, since $x(k)$ is dominated, one may reasonably guess that x will satisfy $x_r \leqq x_r^k - 1$ in all undominated solutions; this is shown to be true in Lemma 5, § 6.

We denote the upper bound so determined for x_r by U_r. After checking for such a bound, we discard, without being recorded, the dominated solution temporarily designated $x(k)$, and the process continues.

Similarly, one may check to establish an upper bound for x_r at instruction 3 when seeking the updated value $\bar{t}_r$ for t_r, since the chance to identify dominated solutions also arises there.

To make use of the upper bounds U_j thus determined, one expands the definition of $\bar{t}_j$ to

$$\bar{t}_j = \min \{i : i > t_j, j \leqq r_i, x_j^i < U_j \text{ and } G(h) = 0, \text{ where } g_h = \alpha(i) + \alpha_j\}.$$

[6] A slightly quicker way to check whether $x(k) = x_r^k e_r$ is to record a flag for each $x(k)$ which takes the value 0 if $x(k) = x_r^k e_r$ and 1 otherwise. The flag for a successor $x(q)$ of $x(k)$ is the same as for $x(k)$ if $r_q = r_k$, and is 1 if $r_q \neq r_k$.

The stipulation[7] $x_j^i < U_j$ is easily checked after checking for $j \leqq r_i$, since $x_j^i = 0$ if $j < r_i$, and x_j^i is precisely the value recorded in the table as "x_r^i" if $j = r_i$.

Had such upper bounds been computed in generating Tables 3 and 4, $U_1 = 1$ would have been determined at step 2 of Table 4, $U_4 = 0$ at step 5 and $U_3 = 2$ at step 8. Also, accounting for $U_1 = 1$ would have avoided two attempts to determine a next value for t_1 in going from step 5 to 6, and accounting for $U_4 = 0$ would have avoided repeated checks to see whether t_4 should become defined once again after step 5.

5. An accelerated version of the algorithm. We now show how to solve problem (I) by generating only a subset of the $x(i)$ produced by the algorithm in § 4.

First note that the algorithm in § 4 generates optimal solutions $x(i)$ in order of increasing cost, stopping when a solution with the desired right-hand side is reached. Let x^* be the optimal x-vector which is to be generated by the algorithm. Consider two nonnegative integer vectors x^a and x^b such that $x^* = x^a + x^b$ and $|cx^b - cx^a| = \min |cx'' - cx'|$, where x' and x'' range over all pairs of nonnegative integer x that sum to x^*. We prove in § 6 (Lemmas 6 and 7) that vectors qualifying to be x^a and x^b will be generated by the algorithm. Consequently, we hereafter denote these solutions by $x(a)$ and $x(b)$, where, say, $a < b$.

Since $c(a)$ and $c(b)$ are either equal or nearly so, it may be expected that $x(a)$ and $x(b)$ will be generated somewhat before x^*. But since $x^* = x(a) + x(b)$, it would be possible to stop immediately after generating $x(b)$, eliminating the generation of all subsequent solutions.

Let $\gamma = c(k) - c(p)$, where k and p are candidates for b and a, so that $x(k) + x(p) = x^* \cdot \gamma$ is $\geqq 0$ because $k > p$. The accelerated algorithm will generate candidates for $x(a)$ and $x(b)$ in order of increasing (or nondecreasing) γ. The trick is to know when the optimal $x(k)$ and $x(p)$, namely $x(a)$ and $x(b)$, have been generated.

Clearly, the first step is to check each time a new $x(k)$ is generated at instruction 2 to determine whether $G(q) \geqq 1$, where g_q is the group element $\alpha_0 - \alpha(k)$. If so, $x(k) + x(p)$ is a feasible solution to (I), where $p = G(q)$ $(\leqq k)$. However, $x(p) + x(k)$ may not be optimal, and, in general, several feasible solutions to (I) may be found by repeating this check for successive $x(k)$.

Let x' denote the best of these solutions. Now, if x' is not optimal, then $c(b) + c(a) < cx'$. Furthermore, until $x(b)$ is finally generated, it must be true that $c(b) \geqq N_r$, because candidates for $x(b)$ are generated in order of increasing cost. Consequently, $c(a) < cx' - N_r$, and an upper bound for $c(s)$ is found by identifying the largest $c(i)$ (call it $c(a')$) such that $c(i) < cx' - N_r$. Since $c(b) \geqq N_r$ and $c(a) \leqq c(a')$, it must be that $c(b) - c(a) \geqq N_r - c(a')$. Moreover, since $c(b)$ and $c(a)$ are as nearly the same as possible, it is also evident that $c(b) - c(a) \leqq c_m$, where $c_m = \max \{c_j\}$. Thus once $N_r - c(a') > c_m$ becomes satisfied, $x(b)$ has already been generated and x' is the optimal solution.

[7] The handling of the U_j can alternately be accommodated by requiring "$j \leqq d_i$" instead of "$j \leqq r_i$ and $x_j^i < U_j$" in the definition of the t_j, where $d_i = r_i - 1$ if $x_r^i = U_r$ $(r = r_i)$, and $d_i = r_i$ otherwise.

Frequently, a smaller value can be given for c_m than $\max\{c_j\}$, thereby permitting earlier termination of the algorithm. To see this, let $\delta = c(b) - c(a)$. Then if x' is not optimal, $2c(a) + \delta < cx'$, and hence $\delta < cx' - 2c(a)$. Suppose each α_j is scanned before starting the algorithm and if $\alpha_j = \alpha_0$, the solution $x = e_j$ is admitted as a candidate for x'. This implies that if x' is not optimal, it is also true that $x(a) \neq 0$, and hence $c(a) \geqq c(1)$. Thus $\delta < cx' - 2\,c(1)$, and c_m may alternately be given by $c_m = \max\{c_j : c_j < cx' - 2\,c(1)\}$.

The cutoff level thus determined will generally succeed in stopping the algorithm considerably in advance of generating x^*. However, the whole process becomes more effective by dropping solutions that would ordinarily be retained. Specifically, when $x(p) + x(k)$ is found to be feasible for (I), both $x(p)$ and $x(k)$ and all their successors can be eliminated from further consideration (Lemma 7, § 6). This is easily accomplished for $x(k)$ simply by not recording it in the table (although $G(h)$ is assigned some positive value to permit solutions dominated by $x(k)$ to be dropped). To prevent additional successors from being generated it suffices to set $r_p = 0$ (or $r_p = -r_p$ in case it is desired to recover the value of r_p later). Similarly, one may locate successors $x(i)$ of $x(p)$ that are already generated, and set $r_i = 0$ (or $r_i = -r_i$) to assure that no more of *their* successors will be generated. Clearly this process can be carried out for as many generations of descendants of $x(p)$ as desired. However, the chances of finding a descendant of $x(p)$ beyond an immediate successor are probably remote.

The main content of the foregoing discussion can be summarized by prescribing the following changes in the constructions of the algorithm as stated in § 4.

Change in instruction 1. If no feasible solution for (I) has previously been found, the instruction remains unchanged. Otherwise, let x' denote the best solution found. Identify N_r (as before), let $c(a') = \max\{c(i) : c(i) < cx' - N_r\}$, and let $c_m = \max\{c_j\}$ (or $c_m = \max\{c_j : c_j < cx' - 2\,c(1)\}$ if the solutions $x = e_j$, $j = 1, \cdots, n$, have been included as candidates for x'). If one of the following conditions holds, then x' is optimal for (I) and the method stops:

(i) T is empty;

(ii) $c(a')$ or c_m does not exist;

(iii) $N_r - c(a') > c_m$.

Change in instruction 2. If 2(i) is applicable, the instruction is unchanged. If instruction 2(ii) is applicable, set $G(h) = k$ (as before) but postpone all other work involved in recording $x(k)$. Identify the group element $g_q = \alpha_0 - \alpha(k)$ and the index $p = G(q)$. If $p = 0$, the generation of $x(k)$ is recorded, as before, and nothing further is done. But if $p \geqq 1$, then $x(p) + x(k)$ is feasible for (I) and is designated the new x' if $c(p) + c(k) < cx'$ (letting $cx' = \infty$ if x' does not exist). Furthermore, the generation of $x(k)$ is not recorded, and if $p \neq k$, r_p is set equal to 0 (or to $-r_p$) to prevent generating new successors of $x(p)$. (One may also replace r_i by $-r_i$ for successors $x(i)$ of $x(p)$ already generated, and similarly for *their* successors, etc.)

Except for these changes, the algorithm remains the same as before. Note that the dropping of $x(k)$ and $x(p)$ specified by the changed instruction 2 may

provide upper bounds for some of the x_j in the manner described in the latter part of § 4.

We trace the course taken by the accelerated version of the algorithm by examining Table 3. Since the accelerated version is the same as the original except for checking for new solutions $x(k) + x(p)$ and dropping their successors, we confine ourselves to determining the effect of these operations on the columns of the table.

The first candidates found for $x(a)$ and $x(b)$ are $x(4)$ and $x(6)$—$\alpha(4) + \alpha(6) = 11 \equiv 1$—yielding $cx' = c(4) + c(6) = 13$. We shall now verify that this is optimal. First, no successors of $x(4)$ or $x(6)$ need be generated. Thus, changing r_4 from 3 to 0 and masking over the column for $x(6)$ (which is not actually generated by the accelerated method) insures that $x(7)$ will be bypassed. Also, $x(8)$ need not be generated, since it is also a successor of $x(4)$. However, to avoid its generation would ordinarily require checking the current t_j and updating t_3 which is found to equal 4. But the method stops without generating $x(8)$ by checking the relation $N_r - c(a') > c_m$. Specifically, upon preparing to generate $x(8)$, $N_r = 10$; hence $c(a') < 13 - 10$, giving $c(a') = 0$. Also, $c_m = 7$, and the relation becomes $10 - 0 > 7$, which is true, thus signaling optimality and directing the method to stop.

The optimality of $x(4) + x(6)$ can also be verified more quickly if the preliminary scanning is used to admit each e_j as a candidate for x' (none of the e_j qualify).

Then $c_m < cx' - 2c(1) = 13 - 6$, giving $c_m = c_3 = 5$. Before updating N_r from 9 to 10, the relation $N_r - c(a') > c_m$ is $9 - 3 > 5$, the validity of which again signals optimality.

6. Theorems and proofs. We refer to the simplified (incomplete) form of the algorithm given in § 3 by stipulating that no solutions are dropped, and the complete form of the algorithm (including the use of upper bound restrictions) by stipulating that solutions *are* dropped.

LEMMA 1. *If no solutions are dropped and the algorithm is not permitted to stop upon generating α_0, then the method will generate every x-vector having finite components.*

Proof. Note that $c > 0$ implies every $j, j = 1, \cdots, n$, will be selected as r at finite intervals. Suppose $x = x'$ is not generated. Then neither would the method generate $x(i) = x' - e_u$, where u is the first nonzero component of x'. For clearly $u \leqq r_i$, which means t_u must eventually be set equal to r_i and hence x' generated. Repeating this argument implies that 0 is not generated, contrary to $x(1) = 0$.

LEMMA 2. *No solution is generated twice, whether or not some solutions are dropped.*

Proof. Let $x(q)$ be the first solution that duplicates a previous one, say $x(p)$. Then for some $h < q$ and $k < p$, $x(q)$ was generated as $x(h) + e_r$ and $x(p)$ was generated as $x(k) + e_r$, where r is the first nonzero component of $x(q)$ and $x(p)$. Thus $x(k) = x(h)$, and since $x(q)$ is the first duplicating solution, $h = k$. When $x(p)$ was generated $t_r = h$, and then t_r was increased, never to be decreased. Consequently, $x(q)$ could not have been generated from $x(h)$, contrary to assumption.

LEMMA 3. *If no solutions are dropped,* $p < q$ *implies*[8]

(i) $c(p) < c(q)$ *or*

(ii) $c(p) = c(q)$ *and* $x(p) \overset{l}{>} x(q)$.

Proof. Note that either (i) or (ii) is satisfied for $q = 2$ and $p < q$ (hence $p = 1$). Suppose the lemma is true for all $q < k$ and $p < q$. We prove it true for $q = k$ and $p = h < k$. Write $x(k) = x(k') + e_u$, $x(h) = x(h') + e_v$. When $x(h)$ was generated, $c(h) = N_v$ and either $N_v < N_u$ or $N_v = N_u$ and $v < u$. If t_u (and N_u) are the same when $x(k)$ is generated as when $x(h)$ was generated, then the proof is immediate. If t_u and N_u change, then let N'_u be the new N_u when generating $x(k)$. We have $N'_u = c(t'_u) + c_u$ and $N_u = c(t_u) + c_u$. But $t_u < t'_u < k$ and, by hypothesis, (i) or (ii) holds relative to $p = t_u$ and $q = t'_u$. It follows immediately that (i) or (ii) must also hold relative to $p = h$ and $q = k$.

LEMMA 4. *Let* S *denote the sequence of solutions generated by the simplified algorithm and suppose this algorithm is modified so that occasionally solutions are not generated but bypassed (according to any rule whatsoever). The resulting sequence of solutions* S' *is a subsequence of* S *(i.e., contains a subset of the solutions of* S *in the same relative order).*

Proof. It is evident from Lemma 1 that S and S' are well-defined. Denote those $x(i)$ in S by $x^1(i)$ and those $x(i)$ in S' by $x^2(i)$. Let $\hat{S}$ be the subsequence of S obtained by deleting from S each $x^1(i)$ such that $x^1(i) \neq x^2(k) \in S'$. We note by Lemmas 2 and 3 that the components of $\hat{S}$ must be a permutation of those of S'. Thus, designate the smallest i such that $x^1(i) \in \hat{S}$ by $\bar{0}$, the next smallest i by $\bar{1}$, and so on. Then we wish to prove that $x^1(\bar{i}) = x^2(i)$ for all $x^2(i) \in S'$. Suppose otherwise, and let $p = \min\{i : x^1(\bar{i}) \neq x^2(i)\}$. Also identify the indices q and r such that $x^1(\bar{p}) = x^2(q)$ and $x^2(p) = x^1(\bar{r})$. It is assured by Lemma 2 that $q, r > p$. Since $\bar{i} = \bar{s}$ if and only if $i = s$ for all i and s, we have $x^1(\bar{p}) = x^1(\bar{h}) + e_u$ for some u and some $h < p$, and $x^2(p) = x^2(k) + e_v$ for some v and some $k < p$. Now, $x^1(\bar{p})$ was generated before $x^1(\bar{r})$, but $x^1(\bar{r}) = x^2(p)$ implies $x^1(\bar{r}) = x^1(\bar{k}) + e_v$. Since $k < p$, this means that when $x^1(\bar{p})$ was generated, $t_v\,(= \bar{k})$ was well-defined ($v \in T$). Thus there was a choice to make between generating $x^1(\bar{p})$ and $x^1(\bar{r})$. Similarly, $x^2(p)$ was generated before $x^2(q)$, but $x^2(q) = x^2(h) + e_u$, so that, by analogous reasoning, there was a choice to make between generating $x^2(p)$ and $x^2(q)$ when $x^2(p)$ was generated in S'. But $\bar{p} < \bar{r}$ thus implies $q < p$, providing a contradiction.

Remark. Lemma 4 establishes the validity of Lemma 3 for the case when solutions are dropped.

LEMMA 5. *If a solution* x' *is dropped at instructions* 2 *or* 3 *of the complete algorithm, then there is no vector* $x^* \geqq x'$ *that is a lexicographically largest optimal solution.*

Proof. Suppose this lemma is false, and let x' be the first solution dropped that has a lexicographically largest optimal descendant. x' is dropped because there is a solution $x(i)$ already generated such that $c(i) \leqq cx'$ and $\alpha(i) = \sum \alpha_j x'_j$. Let

[8] A vector y is defined to be *lexicographically larger* than a vector z, written $y \overset{l}{>} z$, if the first nonzero component of $y - z$ is positive.

$x^* = x' + x''$ be the lexicographically largest optimal solution. Since $\alpha(i) = \sum \alpha_j x'_j$ and $c(i) \leqq cx'$ it must be true that $x(i) + x''$ is also optimal. Moreover, $c(i) = cx'$. But then $x(i)$ is lexicographically larger than x' (since x' would have been generated later than $x(i)$) and in turn $x(i) + x''$ is lexicographically larger than $x' + x''$, contrary to assumption.

Lemmas 1 to 5 immediately imply the next theorem.

THEOREM 1. *The algorithm of* § 4 *yields an optimal solution to* (I) *or verifies that no feasible solution exists, after generating at most D solutions* $x(i)$, *each of which is optimal for* (I) *with* α_0 *replaced by* $\alpha(i)$.

The succeeding results refer to the accelerated algorithm of § 5.

LEMMA 6. *Let* $x(a)$ *and* $x(b)$, $a < b$, *be two solutions such that* $x(a) + x(b)$ *is optimal for* (I), *and, moreover, let* b *be the least index* ($b \geqq a$) *for which two such solutions can be found. Then* $c(b) - c(a) \leqq cx' - cx''$ *for all solutions* $x', x'' \geqq 0$ *such that* $x' + x''$ *is optimal for* (I) *and* $cx' \geqq cx''$.

Proof. Let x^q and x^p be two solutions qualifying as x' and x'' and minimizing $cx' - cx''$. Thus the lemma asserts $c(b) - c(a) = cx^q - cx^p$. The lemma is trivially true for $cx^p = 0$; hence suppose $cx^p > 0$.

Let $\alpha^p = \sum \alpha_j x_{pj}$ and $\alpha^q = \sum \alpha_j x_{qj}$, where x_{pj} and x_{qj} are the jth components of x^p and x^q. Define problem (I^p) to be the same as (I) with α^p replacing α_0 and (I^q) to be the same as (I) with α^q replacing α_0. Since $cx^p, cx^q < c(x^p + x^q)$, it follows from Theorem 1 and Lemma 3 that there exist $x(p)$ and $x(q)$ generated in the process of solving (I) such that $x(p)$ is optimal for (I^p) and $x(q)$ is optimal for (I^q). Suppose $p \leqq q$. By assumption $b \leqq q$, and hence $c(b) \leqq c(q)$. But since $c(q) + c(p) = c(a) + c(b)$, it follows that $c(a) \geqq c(p)$ and hence $c(q) - c(p) = c(b) - c(a)$, proving the lemma.

LEMMA 7. *Let* $x(a)$ *and* $x(b)$ *be as in Lemma* 6, *and suppose there are solutions* $x(h)$ *and* $x(k)$, $h \leqq k < b$, *such that* $x(h) + x(k)$ *is feasible for* (I). *Then there is no vector* $z \geqq 0$ *that satisfies one or more of the following four conditions*:

(i) $x(h) + z = x(a)$,
(ii) $x(h) + z = x(b)$,
(iii) $x(k) + z = x(a)$,
(iv) $x(k) + z = x(b)$.

Proof. Since $k < b$, $x(h) + x(k)$ is not optimal, and hence $c(h) + c(k) > c(a) + c(b)$. Condition (i) implies $x(h) + z + x(b)$ is optimal, hence there exists $x(v)$ such that $x(v) + x(h)$ is optimal and $c(v) < c(k)$. Consequently $v < k$, and $x(v)$ and $x(k)$ qualify to be $x(a)$ and $x(b)$, contrary to $k < b$. Conditions (ii), (iii) and (iv) lead to similar contradictions.

Lemmas 6 and 7 establish the next theorem.

THEOREM 2. *The accelerated algorithm will find an optimal solution if one exists and, in particular, will generate* $x(a)$ *and* $x(b)$ *of Lemma* 6.

7. Properties of optimal solutions. The characteristics of the solution sequence $x(1), x(2), \cdots$, generated by the algorithm, make several properties of optimal solutions to (I) immediately evident. For example, let n_j denote the order of the

subgroup generated by all multiples of α_j. Then, since $n_j\alpha_j$ is the 0-element, the solution $n_j e_j$ is dominated by $x(1)$, and $x_j^i \leqq n_j - 1$ holds for every $x(i)$ generated. The existence of optimal solutions with this property is proved by Gomory in [4].

Moreover, there are at most D of the $x(i)$ (including $x(1) = 0$), and the sum of the variables in each is only one more than in its predecessor. Thus it is evident that $\sum x_j^i \leqq D - 1$ for all $x(i)$. The existence of optimal solutions with this property is also proved by Gomory in [4].

We see that solutions satisfying *both* of the two foregoing properties exist and, in fact, are the only solutions generated by the algorithm.

More recently Gomory has proved that optimal solutions may be found that satisfy[9] $\prod_{j=1}^{n}(x_j + 1) \leqq D$. It may be observed that this property is somewhat stronger than $\sum x_j^i \leqq D - 1$.

We shall prove a different property that is also considerably stronger than $\sum x_j^i \leqq D - 1$ by a direct appeal to the structure of the algorithm.

THEOREM 3. *Let* (I^i) *denote problem* (I) *with* α_0 *replaced by* g_i *for* $i = 1, \cdots, D$. *Then there exists a set of optimal solutions* x^1 *for* $(\mathrm{I}^1), x^2$ *for* $(\mathrm{I}^2), \cdots, x^D$ *for* (I^D), *such that* $\max\{x_1^1, x_1^2, \cdots, x_1^D\} + \max\{x_2^1, x_2^2, \cdots, x_2^D\} + \cdots + \max\{x_n^1, x_n^2, \cdots, x_n^D\} \leqq D - 1$.

Proof. Label the group elements to correspond to the $\alpha(i)$ generated by the algorithm; i.e., $g_1 = \alpha(1)$, $g_2 = \alpha(2), \cdots, g_D = \alpha(D)$, where $\alpha_0 = g_D$. Then the solutions x_i specified by the theorem are precisely the $x(i)$ generated by the algorithm of § 4. To see this, let $U_j = \max\{x_j^1, x_j^2, \cdots, x_j^D\}$ for $j = 1, \cdots, n$. The theorem asserts $\sum U_j \leqq D - 1$. Beginning with $j = 1$, delete each $x(i)$ which is derived from its predecessor by incrementing only x_1. No solution $x(k)$ is deleted in which any of the components $x_2^k, x_3^k, \cdots, x_n^k$ differs from the corresponding component of the predecessor of $x(k)$. Consequently, $x_2, x_3, \cdots, x_n$ attain their maximum values in the undeleted solutions. There are at least U_1 of the $x(i)$ to be deleted, leaving at most $D - U_1$ solutions behind (one of which is $x(1) = 0$). Moreover, in each remaining $x(k)$, $\sum_{j=2}^{n} x_j^k$ is only one larger than in one of the preceding $x(k)$. Thus since there are at most $D - 1 - U_1$ solutions other than $x(1) = 0$, $\sum_{j=2}^{n} x_j^k \leqq D - 1 - U_1$. We repeat this process for $j = 2, \cdots, n$. At each step r, $\sum_{j=r+1}^{n} x_j^k \leqq D - 1 - \sum_{j=1}^{r} U_j$. Finally, we obtain $0 \leqq D - 1 - \sum_{j=1}^{n} U_j$, or $\sum_{j=1}^{n} U_j \leqq D - 1$, as claimed.

The U_j in the proof of the preceding theorem constitute upper bounds for the x_j that apply regardless of which group element α_0 happens to be. One way to determine such a set of U_j is to apply the algorithm until every g_i is generated, and then compute $\max\{x_j^1, \cdots, x_j^D\}$ for each j. There is also a second much faster way. Suppose the definition of $\bar{t}_j$ is simplified so that, when $x(t_j) + e_j$ is generated as the solution $x(k)$, j is set equal to k. Further suppose the method is stopped only when T becomes empty. This "modified" version of the algorithm has the following features.

(i) Only solutions of the form $1e_j, 2e_j, 3e_j, \cdots$ are generated for each j.

(ii) As soon as a solution he_j is dropped (checked but not generated), j is removed from T and never returns. At this point U_j can be recorded as $h - 1$.

[9] Reported by W. W. White in [8].

It may be observed that the foregoing method will usually generate fewer (and never more) than the D solutions required to determine the U_j with the unmodified algorithm. Moreover, all comparison operations in determining the next value $\bar{t}_j$ of t_j are eliminated. No upper bounds are checked for the variables, since a variable drops from T as soon as its upper bound is attained and verified. Because of this, T also tends to shrink more rapidly than with the unmodified algorithm, reducing the number of effective problem variables. Finally, there is no need to check those $j \notin T$ to see if they should be put back into T.

By Lemma 4, the sequence of solutions generated is a subsequence of that generated if no solutions are dropped. (Here some of the solutions are "dropped" by the restrictive definition of $\bar{t}_j$.) Solutions bypassed due to dominance considerations are therefore truly dominated and would not be generated in any case. Consequently, the U_j are valid (although possibly not as restrictive as those obtained from the sequence of $x(i)$ generated by the unmodified algorithm) and satisfy $\sum U_j \leqq D - 1$ by the proof of Theorem 3.

8. Computational experience. Roughly five hundred problems have been solved with the algorithm, containing from 50 to 1500 variables and from 100 to 4500 group elements. The problems all have the form

$$\text{Minimize} \quad \sum_{j=1}^{n} c_j x_j$$

$$\text{subject to} \quad \sum_{j=1}^{n} \alpha_j x_j \equiv \alpha_0 \pmod{D}, \quad x_j \geqq 0 \text{ and integer},$$

where the c_j and α_j are positive integers.

The c_j were randomly generated to lie within a specified interval, and several different intervals were tested to determine the effect on computation times.

The α_j were generated by selecting α_1 randomly from the set $S = \{1, 2, \cdots, D - 1\}$, α_2 randomly from the set $S - \{\alpha_1\}$, α_3 randomly from the set $S - \{\alpha_1, \alpha_2\}$, and so on. Thus, $\alpha_p = \alpha_q$ for $p \neq q$ was avoided, although one might expect this situation to arise in practice, thereby making it possible to reduce the number of problem variables.

Representative tables of computation times follow. All times reported are in seconds of central processing time on the CDC 6600.[10] The c_j were arranged in ascending order before starting the algorithm but the time for this preliminary ordering is not included.

The tables are headed with the symbols n, D, Total, Av., Fast and Ratio. "Total" gives the time for the algorithm of § 4 to solve problem (I) for every value of α_0. That is, the algorithm is permitted to continue until $\alpha(1), \alpha(2), \cdots, \alpha(D)$ are all generated. Since, in practice, one will often be interested in solving (I) for a

[10] The code was written in FORTRAN IV.

particular value of α_0, the times to solve (I) for[11] $\alpha_0 = \alpha(D/5), \alpha(2D/5), \cdots, \alpha(D)$ were averaged to give an idea of expected computation time, and this average appears in the column headed "Av."

The "Fast" column gives the computation time for solving problem (I) with the accelerated version of the algorithm. The accelerated version was applied to (I) with $\alpha_0 = \alpha(D)$ (thus requiring more computation than with α_0 at any other value).

The "Ratio" column gives the ratio of the "Fast" column to the "Total" column, indicating the relative efficiency of the accelerated version to the version of the algorithm of § 4.

From Tables 5, 6 and 7 it may be seen that computation times tend to become longer as the relative difference between the largest and smallest c_j decreases.[12] In Tables 5 and 6 the effect of holding n constant and increasing D is an almost exactly proportional increase in the "Total" times. The increase in "Total" times in Table 7 for n and $D \geqq 1000$ is somewhat less than proportional to increases in D.

TABLE 5

$1 \leqq c_j \leqq 400$					
n	D	Total	Av.	Fast	Ratio
500	501	.148	.096	.098	.662
	1002	.270	.175	.197	.730
	1503	.421	.274	.334	.793
1000	1000	.245	.176	.154	.629
	2002	.452	.307	.382	.845
	3003	.680	.479	.577	.849
1500	1501	.439	.288	.333	.759
	3002	.888	.559	.555	.625
	4503	1.274	.838	.855	.671

TABLE 6

$301 \leqq c_j \leqq 700$					
n	D	Total	Av.	Fast	Ratio
500	501	.739	.507	.376	.509
	1002	1.326	.843	.409	.308
	1503	2.297	1.237	.645	.281
1000	1001	2.537	1.858	1.092	.430
	2002	4.388	2.996	1.294	.295
	3003	6.663	3.972	2.081	.312
1500	1501	4.481	3.485	1.724	.385
	3002	8.790	5.565	2.252	.256
	4503	13.916	7.996	3.717	.267

[11] The numbers $D/5$, $2D/5$, $\cdots$ were of course replaced with their nearest integers.

[12] This may be due in part to a less than optimal computer subroutine for determining N_r at each iteration.

TABLE 7

$601 \leqq c_j \leqq 1000$					
n	D	Total	Av.	Fast	Ratio
500	501	1.226	1.017	.445	.363
	1002	2.041	1.508	.509	.249
	1503	3.146	2.060	1.207	.384
1000	1001	4.749	3.999	5.393	1.136*
	2002	6.373	5.194	1.532	.240
	3003	9.592	6.738	2.567	.268
1500	1501	8.240	7.657	1.061	.129
	3002	12.006	10.286	2.389	.199
	4503	15.471	12.201	4.544	.294

TABLE 8

n	D	Total	Fast	Ratio	
50	100	.050	.020	.400	$1 \leqq c_j \leqq 40$
	150	.061	.024	.393	
100	200	.081	.035	.436	
50	100	.056	.029	.516	$31 \leqq c_j \leqq 70$
	150	.072	.033	.458	
100	200	.153	.035	.228	
50	100	.065	.013	.200	$61 \leqq c_j \leqq 100$
	150	.088	.019	.215	
100	200	.164	.044	.268	

Also, for the ranges of c_j in which the computation times are longer (Tables 6 and 7), the "Av." and "Fast" times become increasingly favorable relative to the "Total" times. The superiority of the accelerated version of the algorithm is quite evident from the fact that the "Fast" times in Tables 6 and 7 are not only better than the "Total" times, but are also considerably better than the "Av." times. An exception occurs for $n = 1000$ and $D = 1001$ in Table 7, as indicated by the asterisk beside the "Ratio" entry. The reason for this exceptional divergence from the pattern evident in the other entries is not known.

While computation times appear to increase roughly in proportion to increases in D, they do not increase in proportion to increases in n. For example, in Table 5, a proportional increase in computation time would lead one to expect the "Total" and "Fast" times for $n = 1500$ and $D = 1501$ to be roughly 1.2 and .99 seconds

(multiplying the times for $n = 500$ and $D = 1503$ by three). In contrast, they are actually .439 and .333 seconds.

More dramatic examples arise by comparing the times[13] of Table 8 to those of Tables 5, 6 and 7. At first glance the Table 8 times are very small, since most of the "Total" times are under .09 seconds and most of the "Fast" times are under .04 seconds. However, if the computational time were to increase in proportion to nD, as in the group algorithm of [4], the times for "corresponding" ranges of c_j would be greater by a factor of from 4 to 40 than the times appearing in Tables 5, 6 and 7. For example, extrapolating from $n = 50$ and $D = 150$ for $1 \leqq c_j \leqq 40$ would give a "Total" time of 54.900 seconds for $n = 1500$, $D = 4503$ in Table 5, as compared to 1.274 seconds.

Such comparisons do not yield a precise formula linking computation times and increases in n and D, both because of the effect of different ranges of c_j and because of probable shortcomings of the computer code in determining $N_r = \min \{c_j\}$ and in determining the current composition of the set T.[14] Nevertheless, without attempting to be definitive, the tables do establish definite patterns in the performance of the algorithm: in particular, that the accelerated version of the algorithm is distinctly superior to the version of § 4 and that computation times for both versions increase at a considerably more favorable rate than nD.

Acknowledgment. I am indebted to Sam Ginsburg of the University of California, Berkeley, and to Professor Stanley Zionts of the State University of New York at Buffalo for suggestions that have improved the exposition of this paper.

REFERENCES

[1] P. C. GILMORE AND R. E. GOMORY, *Multistage cutting stock problems of two and more dimensions*, Operations Res., 13 (1965).

[2] FRED GLOVER, *An algorithm for solving the linear integer programming problem over a finite additive group, with extensions to solving general linear and certain non-linear integer problems*, Rep. ORC 66–29, Operations Research Center, University of California, Berkeley, 1966.

[3] R. E. GOMORY, *An algorithm for integer solutions to linear programs*, Recent Advances in Mathematical Programming, R. L. Graves and P. Wolfe, eds., McGraw-Hill, New York, 1963.

[4] ———, *On the relation between integer and noninteger solutions to linear programs*, Proc. Nat. Acad. Sci. U.S.A., 53 (1965), pp. 260–265.

[5] ———, *Faces of an integer polyhedron*, Ibid., 57 (1967), pp. 16–18.

[6] JEREMY F. SHAPIRO, *Dynamic programming algorithms for the integer programming problem. I: The integer programming problem viewed as a knapsack type problem*, Operations Res., 16 (1968), pp. 103–121.

[7] J. F. SHAPIRO AND H. M. WAGNER, *A finite renewal algorithm for the knapsack and turnpike models*, Ibid., 15 (1967), pp. 319–341.

[8] W. W. WHITE, *On a group theoretic approach to linear integer programming*, Rep. ORC 66–27, Operations Research Center, University of California, Berkeley, 1966.

[13] "Av." times were not computed for Table 8.

[14] However, the latter should probably penalize computation times more for larger problems than for smaller ones. A more sophisticated computer code is currently being designed for a sequel to this paper which extends the group algorithm to the general integer programming problem.