An Algorithm to Solve Minimal Cover Problems

KEIICHI ABE* AND TERUO FUKUMURA*

1. *Introduction*

A combinatorial algorithm to solve minimal cover problems is described here. These problems are encountered in minimization of Boolean functions and considered as the simplest but the most difficult ones among integer programming problems. Although many combinatorial algorithms for integer programming have been reported, it seems that a decisively good one is not yet found. In fact, the efficiency of an algorithm depends remarkably on the properties of the problems. This is the reason why we confine ourselves to minimal cover problems, a special class of integer programming problems. We expect that the effort to construct an algorithm for these problems will bring us many useful informations about solving the more general problems as well as a fast and specialized algorithm itself.

2. *The Formulation of Minimal Cover Problems*

A minimal cover problem is formulated as follows:

Under the constraints

$$\sum_{j=1}^{n} c_{ij}x_j \geqq 1 \quad (i=1,2,\ldots,m) \tag{1}$$

where $c_{ij}=0$ or 1 $(i=1,2,\ldots,m;\ j=1,2,\ldots,n)$ and

$$x_j=0 \text{ or } 1 \quad (j=1,2,\ldots,n), \tag{2}$$

minimize the object function

$$y=\sum_{j=1}^{n} x_j. \tag{3}$$

3. *Algorithm*

To solve the above problem, we use a matrix $C=(c_{ij})$ and introduce the following vectors. We denote the states of rows by

$$U=(u_1,u_2,\ldots,u_m)$$
$$u_i=0 \text{ or } \pm 1 \quad (i=1,2,\ldots,m) \tag{4}$$

and those of columns by

$$V=(v_1,v_2,\ldots,v_n)$$
$$v_j=0 \text{ or } \pm 1 \quad (j=1,2,\ldots,n). \tag{5}$$

The values of the component variables of these vectors bear the following meanings.

This paper first appeared in Japanese in Joho Shori (the Journal of the Information Processing Society of Japan), Vol. 10, No. 4 (1969), pp. 227-234.

* Faculty of Engineering, Nagoya University.

$$u_i=\begin{cases}1: & \text{the } i\text{-th constraint is satisfied.}\\ -1: & \text{the } i\text{-th constraint needs no longer to be satisfied.}\\ 0: & \text{otherwise.}\end{cases}$$

$$v_j=\begin{cases}1: & \text{the } j\text{-th variable } x_j \text{ is set to 1. (This corresponds to the selection of the } j\text{-th column in our algorithm.)}\\ -1: & \text{the } j\text{-th variable } x_j \text{ needs no longer to be set to 1.}\\ 0: & \text{otherwise.}\end{cases}$$

Initially, both vectors are set to zero vectors. If all u_i's become non-zero, then a feasible solution is obtainable by converting v_j to x_j according to the equation

$$x_j=\begin{cases}1 & \text{if } v_j=1\\ 0 & \text{otherwise.}\end{cases} \qquad (6)$$

3.1 *The Reduction of the Matrix*

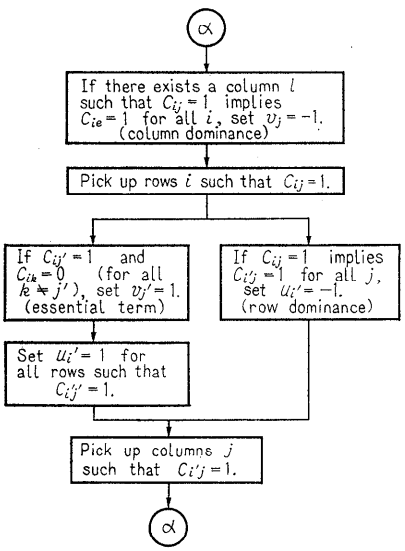
As is generally known, solving a minimal cover problem starts with the reduction of the matrix C by repeating the following three reduction rules,*

(i) essential term

(ii) row dominance

(iii) column dominance.

A use of one rule may affect possible use of other rules, e.g. if an essential term reduction is made, the possibilities of column dominance reduction occur anew. Therefore, we have to restrict the range of affection, as summarized in Fig. 1.



Eig. 1. Algorithm of matrix reduction.

* These rules are explained, for example, in [1]. It should be noticed that we adopt rows and columns contrary to custom for the ease of printing the solutions in our program.

3.2 *Branch and Bound Method*

When no more reduction rule can be applied to the matrix, then the problem is to be solved by means of branch and bound method [2]. In this paper, we adopted the branching technique in connection with constraints, which can be expressed as follows.

(1) [Setting a branching point.] Select a row i^* whose state u_{i^*} is 0, and alter the state to 1.

(2) [Branching to a column.] Select a column j^*, for which $c_{i^*j^*}=1$ and $v_{j^*}=0$. Alter the state v_{j^*} to 1.

(3) [Recursions.]

(3a) If, selecting the column j^*, all the states of rows are ± 1, a feasible solution is obtained. Recording the solution, return to the last branching point and branch into another column not selected yet. If there remains no column to be selected, return to the next previous branching point and so on.

(3b) If, after selecting the column j^*, there remain rows whose states are 0, go to the next step and branch again.

Considering the situations after selection of the column j^* and alteration of the state v_{j^*} from 0 to 1 in (2), they are just the same as the situations after altering the state $v_{j'}$ by essential term reduction rule in Fig. 1. Therefore we should modify the above algorithm as follows.

"Don't go at once from (2) to (3), but go from (2) via (2)′ to (3).

(2)′ Apply the reduction rules."

Similarly, when we return to a branching point, we should alter the state of the last selected column to -1 (this procedure is necessary for avoiding a double searching), and apply the reduction rules for this alteration. After all possible reductions are made, go to (2).

These two modifications of algorithm brought us enormous saving of computing time.

No sophisticated bounding criterion is employed in our algorithm. Only the number of columns for which $v_j=1$ is always compared with the minimum value of object function given by the feasible solutions already obtained.

The resulting branch and bound algorithm is shown in Fig. 2.

4. *Computer Simulations*

We wrote the algorithm discussed above in the form of a Fortran program and several examples were solved by a digital computer (HITAC 5020E). We did not make use of bit processing instructions because of machine compatibilities. The results of simulations are summarized in Table 1. All the problems are given in the reduced form. Problems No. 1～5 are encountered in minimization of Boolean functions, but No. 6～10 are made artificially by means of random numbers.

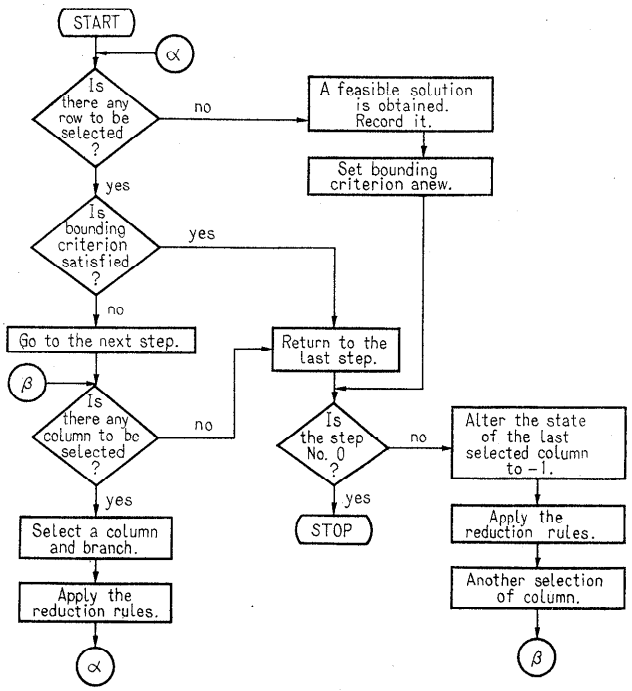


Fig. 2. Branch and bound algorithm.

Table 1 The results of computer simulations.

Problem No.	Number of rows m	Number of columns n	Number of 1's per row	Computing time (sec)	
				CPU time	USE time
1	3	3	2	0	0.167
2	6	6	2	0	0.251
3	11	11	2.3	1	0.447
4	26	26	3.8	0	0.990
5	34	33	2.9	2	1.104
6	35	15	3	2	2.689
7	30	30	3	3	3.052
8	30	30	3	3	3.332
9	50	50	3	229	232
10	50	50	3	170	171

5. *Discussions and Conclusions*

Combinatorial methods to solve integer programming problems give us a typical question in heuristics. These methods are, essentially, the exhaustive search of all possible cases, and the efficiency of an algorithm depends largely on the efficiency of the exhaustion. It seems plausible that before branching it is recommended to foresee as far as possible to select the better branch.

But is it always the case? The time to foresee is consumed each time branching takes place, while the time saving due to it occurs only on rare occasions. For this reason, the efficiency of a combinatorial algorithm depends strikingly on the properties of problems and a sophisticated procedure adaptable to the type of given problem is desired. The generalization of the algorithm described in this paper to solve more general integer programming problems must be studied from this point of view.

References

[1] McCluskey, E. J., Introduction to the Theory of Switching Circuits, McGraw-Hill Inc. (1965).

[2] Lawler, E. L. and D. E. Wood, Branch and Bound Methods: A Survey, *Opns. Res. 14*, p. 699 (1966).