An Algorithm to Derive the Complement of a Binary Function with Multiple-Valued Inputs

TSUTOMU SASAO, MEMBER, IEEE

***Abstract* — A recursive algorithm to obtain a complement of a sum-of-products expression for a binary function with p-valued inputs is presented. It produces at most $p^n/2$ products for n-variable functions, whereas a conventional elementary algorithm produces $O(t^n \cdot n^{(1-t)/2})$ products where $t = 2^p - 1$. It is 10–20 times faster than the elementary one when $p = 2$ and $n = 8$. For large practical problems, it produces many fewer products than the disjoint sharp algorithm used by MINI. Applications of the algorithm to PLA minimization are also presented.**

***Index Terms* — Complement of logical expression, logic design, minimization of logical expressions, multiple valued logic, prime implicants, programmable logic array, switching theory.**

I. Introduction

As an elementary method to produce the complement of a sum-of-products expression $\mathcal{F}$ as a sum-of-products expression, Algorithm 2.1 described in Section II is well known. However, this method becomes quite inefficient when the number of input variables is large because it produces all the prime implicants of $\overline{\mathcal{F}}$. For example, the elementary method generates $O(3^n/n)$ products for a class of n-variable switching functions (two-valued input binary functions) [1], whereas the algorithm presented in this paper will generate at most 2^{n-1} products. The new algorithm is about 10–20 times faster than the elementary one for switching function of 8 variables.

A binary function with multiple-valued inputs is useful in designing programmable logic arrays with decoders [2], [3] and other circuits [4]. It is a mapping $F\colon \times_{i=1}^{n} P_i \rightarrow B$ where $P_i = \{0, 1, \cdots, p_i - 1\}$ and $B = \{0, 1\}$. When $P_i = \{0, 1\}$ for $i = 1, 2, \cdots, n$, F is an ordinary switching function, in other words, a binary function with two-valued inputs. As is well known, a minimum sum-of-products expression for a switching function corresponds to a minimum two-level AND–OR network, or a minimum standard PLA [3] (a standard PLA is also called a two-level PLA [2]). When $p_i = 2^t$ for $i = 1, 2, \cdots, n$ where $t \geqq 2$, F represents a PLA with t-bit decoders [2] (which is also called a decoded PLA [3]), and a minimum sum-of-products expression for F corresponds to a minimum PLA with t-bit decoders.

A binary function with multiple-valued inputs can also represent a multiple-output function. Consider a set of m switching functions

$$f_0(x_1, x_2, \cdots, x_n),$$
$$f_1(x_1, x_2, \cdots, x_n),$$
$$\vdots$$
$$f_{m-1}(x_1, x_2, \cdots, x_n).$$

Define a binary function $F(x_1, x_2, \cdots, x_n, y)$ such that

$$F(x_1, x_2, \cdots, x_n, y) = f_y(x_1, x_2, \cdots, x_n)$$

where

$$y = 0, 1, \cdots, \quad \text{and} \quad m - 1.$$

Then F represents all $f_0, f_1, \cdots, f_{m-1}$ and consequently a standard PLA with n inputs and m outputs [14]. Note that the variable y takes m different values. Functions for multiple-output PLA's with decoders can be defined similarly, and in this case the inputs $x_{i_1}, x_{i_2}, \cdots, x_{i_t}$ of each decoder can be regarded as an input $X_i = (x_{i_1}, x_{i_2}, \cdots, x_{i_t})$ whose values are $(0, 0, \cdots, 0)$, $(0, 0, \cdots, 1)$, $\cdots$, and $(1, 1, \cdots, 1)$.

Most programs for PLA minimization require checking whether or not a product c is an implicant of a function a large number of times. This can be efficiently done by examining whether $c \cdot \overline{F} \equiv 0$ or not. A fast complementation algorithm which produces as few products as possible has been desired because some PLA minimization algorithms such as MINI [5] and Espresso [6] use the complement of the given function at the initial stage of minimization. In MINI and Espresso, the required memory size primarily depends on the number of products in the given expression and its complement. The disjoint sharp algorithm used in MINI often requires an excessive memory space in computing the complement, and accordingly the initial phase of the minimization cannot be completed. The algorithm proposed in this paper will produce many fewer products than the disjoint sharp algorithm in a comparable computation time. It has been now incorporated into MINI and other systems and has been effectively used to design logical networks.

II. An Elementary Method for Complementation

Let us define some concepts in multiple-valued logic.

Definition 2.1: Let X_i be a variable which takes any value in the set $P = \{0, 1, \cdots, p_i - 1\}$. $X_i^{S_i}$ is a *literal* of X_i when $S_i \subseteq P_i$. $X_i^{S_i}$ represents a function such that

Manuscript received August 2, 1983; revised March 22, 1984 and July 18, 1984.

The author is with the Department of Electronic Engineering, Osaka University, Osaka 565, Japan.

$$X_i^{S_i} = \begin{cases} 0 & \text{if } X_i \notin S_i \\ 1 & \text{if } X_i \in S_i \end{cases}$$

where $S_i \subseteq P_i$.

Definition 2.2: A product of literals $X_1^{S_1} \cdot X_2^{S_2} \cdots X_n^{S_n}$ is called a *product*. A sum of products is called *a sum-of-products expression*.

When $S_i = P_i$, a literal $X_i^{S_i}$ denotes the constant 1 function and may be deleted from the product.

Theorem 2.1 [2]: An arbitrary function can be represented by a sum-of-products expression

$$\mathcal{F}(X_1, X_2, \cdots, X_n) = \bigvee_{(S_1, S_2, \cdots, S_n)} X_1^{S_1} \cdot X_2^{S_2} \cdots X_n^{S_n}$$

where $S_i \subseteq P_i$.

From here, F (a capital letter) represents a binary function and $\mathcal{F}$ (a script letter) represents its sum-of-products expression.

Definition 2.3: Let E be a product. If F is equal to one whenever E is equal to one, then E is *an implicant* of F and this is denoted by $E \leq F$. E is called *a prime implicant* if $E \leq F$ and there is no product E_1 such that $E \leq E_1 \leq F$ and $E_1 \neq E$.

Lemma 2.1: Let F, G, and H be binary functions.

$$\overline{F \cdot G} = \overline{F} \vee \overline{G}, \overline{F \vee G} = \overline{F} \cdot \overline{G} \qquad \text{(De Morgan's law)}.$$

$$(F \vee G) \cdot H = F \cdot H \vee G \cdot H \qquad \text{(Distributive law)}.$$

$$F \vee F \cdot G = F \qquad \text{(Absorption law)}.$$

As an elementary method to produce the complement of sum-of-products expression as a sum-of-products expression, the following is well known.

Algorithm 2.1 (Elementary Complementation Algorithm):

1) By using De Morgan's law, produce the complement of a given sum-of-products expression as a product-of-sums expression.

2) By using the distributive law, expand the expression into a sum-of-products expression. Delete null products (if $A \cap B = \emptyset$ then $X^A \cdot X^B = \emptyset$) and redundant literals (if $A \supseteq B$ then $X^A \cdot X^B = X^B$).

3) By using the absorption law, drop subsumming products (if $E_1 \leq E_2$ then $E_1 \vee E_2 = E_2$).

Example 2.1: Consider a binary function

$$F\colon \{0,1\} \times \{0,1,2\} \times \{0,1,2,3\} \rightarrow \{0,1\}$$

and its expression

$$\mathcal{F} = X_1^0 \cdot X_2^1 \cdot X_3^{\{1,3\}} \vee X_1^1 \cdot X_2^{\{0,2\}} \cdot X_3^{\{1,2\}} \vee X_2^{\{1,2\}} \cdot X_3^1 .$$

In the above expression, for simplicity, $X_1^{\{0\}}$ and $X_1^{\{1\}}$ are represented by X_1^0 and X_1^1, respectively. Let us obtain the complement of $\mathcal{F}$ by Algorithm 2.1. First, by De Morgan's law, convert it into a product-of-sums expression

$$\overline{\mathcal{F}} = (X_1^1 \vee X_2^{\{0,2\}} \vee X_3^{\{0,2\}}) \cdot (X_1^0 \vee X_2^1 \vee X_3^{\{0,3\}}) \cdot (X_2^0 \vee X_3^{\{0,2,3\}}) .$$

Second, by the distributive law, we have the following:

$$\overline{\mathcal{F}} = (X_1^1 \cdot X_2^1 \vee X_1^1 \cdot X_3^{\{0,3\}} \vee X_1^0 \cdot X_2^{\{0,2\}} \vee X_2^{\{0,2\}} \cdot X_3^{\{0,3\}} \vee X_1^0 \cdot X_3^{\{0,2\}} \vee X_2^1 \cdot X_3^{\{0,2\}} \vee X_3^0) \cdot (X_2^0 \vee X_3^{\{0,2,3\}}) .$$

In the above expression, $X_1^1 \cdot X_1^0$ and $X_2^{\{0,2\}} \cdot X_2^1$ are omitted because they are null products. By using the distributive law again, we have the sum-of-products expression

$$\begin{aligned} \overline{\mathcal{F}} = {} & X_1^1 \cdot X_2^0 \cdot X_3^{\{0,3\}} \vee X_1^0 \cdot X_2^0 \vee X_2^0 \cdot X_3^{\{0,3\}} \\ & \vee X_1^0 \cdot X_2^0 \cdot X_3^{\{0,2\}} \vee X_2^0 \cdot X_3^0 \\ & \vee X_1^1 \cdot X_2^1 \cdot X_3^{\{0,2,3\}} \vee X_1^1 \cdot X_3^{\{0,3\}} \\ & \vee X_1^0 \cdot X_2^{\{0,2\}} \cdot X_3^{\{0,2,3\}} \vee X_2^{\{0,2\}} \cdot X_3^{\{0,3\}} \\ & \vee X_1^0 \cdot X_3^{\{0,2\}} \vee X_2^1 \cdot X_3^{\{0,2\}} \vee X_3^0 . \end{aligned}$$

Third, by the absorption law, we can delete some products such as $X_1^1 \cdot X_2^0 \cdot X_3^{\{0,3\}}$ and $X_2^0 \cdot X_3^0$ because $X_2^0 \cdot X_3^0, \leq X_3^0$ and $X_1^1 \cdot X_2^0 \cdot X_3^{\{0,3\}} \leq X_1^1 \cdot X_3^{\{0,3\}}$. Hence, we have the sum-of-products expression

$$\begin{aligned} \overline{\mathcal{F}} = {} & X_1^0 \cdot X_2^0 \vee X_1^1 \cdot X_2^1 \cdot X_3^{\{0,2,3\}} \vee X_1^1 \cdot X_3^{\{0,3\}} \\ & \vee X_1^0 \cdot X_2^{\{0,2\}} \cdot X_3^{\{0,2,3\}} \vee X_2^{\{0,2\}} \cdot X_3^{\{0,3\}} \\ & \vee X_1^0 \cdot X_3^{\{0,2\}} \vee X_2^1 \cdot X_3^{\{0,2\}} \vee X_3^0 . \end{aligned}$$

Note that $\overline{\mathcal{F}}$ now contains 8 products. (End of Example)

Straightforward application of Algorithm 2.1 is quite inefficient. It might be made more efficient by simplifying intermediate results by using the absorption law or by changing the order of expansion, but Algorithm 2.1 often generates an excessive number of products. This is because Algorithm 2.1 generates all the prime implicants of a function $\overline{F}$, as stated in the following theorem.

Theorem 2.1: Let $\mathcal{F}$ be a sum-of-products expression of a function F and $\overline{\mathcal{F}}$ be an expression obtained by Algorithm 2.1. Then, $\overline{\mathcal{F}}$ contains all the prime implicants of $\overline{F}$.

The proof can be obtained in a manner similar to the case of a switching function [7].

As to the maximal number of the prime implicants of functions, the following theorem is known [8].

Theorem 2.2: Let $z(n,p)$ be the maximal number of the prime implicants of a binary function with p-valued inputs; there exists a positive constant K such that $K(t^n \cdot n^{(1-t)/2}) \leq z(n,p)$ where $t = 2^p - 1$.

The above theorem shows that, in the case of switching functions (i.e., $p = 2$), there exists an n-variable function which has $O(3^n/n)$ prime implicants and in the case of binary functions with four-valued inputs (i.e., $p = 4$) there exists a function which has $O(15^n/n^7)$ prime implicants.

III. A Fast Complementation Algorithm

The complementation algorithm in this paper contains the following restriction operation.

Definition 3.1: Let $c = X_1^{S_1} \cdot X_2^{S_2} \cdots X_n^{S_n}$ be a product. *A restriction of a function F to c,* denoted by $F(|c)$, is the function whose domain is restricted to the minterms of c. In other words, $F(|c)$ denotes the restricted function

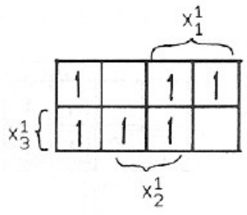

Fig. 1. Example 3.1.

$F(x_1, x_2, \cdots, x_n)$ where $x_1 \in S_1$, $x_2 \in S_2, \cdots$, and $x_n \in S_n$.

Example 3.1: Consider the function F shown in Fig. 1 where $P_1 = P_2 = P_3 = \{0, 1\}$. Note that F has the domain $\{0, 1\} \times \{0, 1\} \times \{0, 1\}$ and the universal cube for F is $X_1^{\{0,1\}} \cdot X_2^{\{0,1\}} \cdot X_3^{\{0,1\}}$. Given $c = X_1^{\{0,1\}} \cdot X_2^1 \cdot X_3^{\{0,1\}}$, $F(|c)$ denotes the function shown in Fig. 2. Now $F(|c)$ has the domain $\{0, 1\} \times \{1\} \times \{0, 1\}$ and the universal cube for $F(|c)$ is $X_1^{\{0,1\}} \cdot X_2^{\{1\}} \cdot X_3^{\{0,1\}}$. (End of Example)

As shown in the above example, $F(|c)$ has a different universal cube than F. Also, each function $F(|c_i)$ has a different universal cube for a different c_i

When we calculate the complement by a computer, a function can be conveniently represented by positional cubes [11], [12] as illustrated in the following example, and logic operations can be performed based on these cubes.

Example 3.2: Consider the function F of Example 3.1 and its expression

$$\mathscr{F}(X_1, X_2, X_3) = X_1^0 \cdot X_2^0 \vee X_1^0 \cdot X_3^1 \vee X_2^1 \cdot X_3^1 \vee X_1^1 \cdot X_2^1 \vee X_1^1 \cdot X_3^0$$

where $P_1 = P_2 = P_3 = \{0, 1\}$. Then the positional cube for $\mathscr{F}$ is

$$\begin{array}{cccc} & X_1 & X_2 & X_3 \\ & 01 & 01 & 01 \\ \hline & 10 & 10 & 11 \\ & 10 & 11 & 01 \\ \mathscr{F} = & 11 & 01 & 01 \\ & 01 & 01 & 11 \\ & 01 & 11 & 10 \end{array}$$

In the above cube, 10 denotes X_i^0, 01 denotes X_i^1, and 11 denotes $X_i^{\{0,1\}}$. The positional cube for the universal cube is

$$U = \{11 \quad 11 \quad 11\}.$$

Consider a cube $c = X_1^{\{0,1\}} \cdot X_2^1 \cdot X_3^{\{0,1\}}$. The positional cube for c is

$$c = \{11 \quad 01 \quad 11\}.$$

Logical AND of $\mathscr{F}$ and c is obtained by bit-by-bit AND operation of the cubes for $\mathscr{F}$ and c

$$\mathscr{F} \cdot c = \begin{bmatrix} 10 & 00 & 11 \\ 10 & 01 & 01 \\ 11 & 01 & 01 \\ 01 & 01 & 11 \\ 01 & 01 & 10 \end{bmatrix}$$

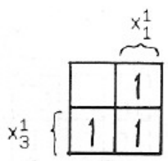

Fig. 2. Example 3.1.

where the top row may be deleted because it represents a null product.

Logical OR of $\mathscr{F}$ and c is obtained by the concatination of the cubes for $\mathscr{F}$ and c (the cube for c is appended to the bottom row in the following):

$$\mathscr{F} \vee c = \begin{bmatrix} 10 & 10 & 11 \\ 10 & 11 & 01 \\ 11 & 01 & 11 \\ 01 & 01 & 11 \\ 01 & 11 & 10 \\ 11 & 01 & 11 \end{bmatrix}$$

(End of Example)

As can be seen from the above example, when two expressions such as $\mathscr{F}$ and c have the same number of bit positions in cubes, logical AND and logical OR of these expressions can be performed easily. But the restriction operation in Definition 3.1 changes the number of bit positions in cubes. This is unsuitable for the computer processing based on the above positional cubes. Therefore, we introduce a new definition of restriction operation. In the following definition, operation 1) produces an expression without changing the number of bit positions in cubes, and operation 2) makes this expression represent the same function as the one which can be obtained by Definition 3.1.

Definition 3.2: Let $c = X_1^{S_1} \cdot X_2^{S_2} \cdots X_n^{S_n}$ be a product. *A cube restriction of an expression* $\mathscr{F}$ *to* c is obtained as follows and denoted by $\mathscr{F}(|c)$.

1) Make a logical product of $\mathscr{F}$ and c, and delete null products.

2) Replace each product $X_1^{T_1} \cdot X_2^{T_2} \cdots X_n^{T_n}$ in the expression obtained by 1) with

$$X_1^{(T_1 \cup \bar{S}_1)} \cdot X_2^{(T_2 \cup \bar{S}_2)} \cdots X_n^{(T_n \cup \bar{S}_n)}.$$

Example 3.3: Consider the expression shown Fig. 3

$$\mathscr{F} = X_1^0 \cdot X_2^0 \vee X_1^0 \cdot X_3^1 \vee X_2^1 \cdot X_3^1 \vee X_1^1 \cdot X_2^1 \vee X_1^1 \cdot X_3^0$$

where each product is shown by a loop in Fig. 3. Suppose $c = X_1^{S_1} \cdot X_2^{S_2} \cdot X_3^{S_3}$ is given where $S_1 = \{0, 1\}$, $S_2 = \{1\}$, and $S_3 = \{0, 1\}$. $\mathscr{F}(|c)$ can be obtained by Definition 3.2 as follows.

1) Logical AND of $\mathscr{F}$ and c is given by

$$\mathscr{F} \cdot c = X_1^0 \cdot X_2^1 \cdot X_3^1 \vee X_1^{\{0,1\}} \cdot X_2^1 \cdot X_3^1 \vee X_1^1 \cdot X_2^1 \cdot X_3^{\{0,1\}} \vee X_1^1 \cdot X_2^1 \cdot X_3^0.$$

2) Replacing each product $X_1^{T_1} \cdot X_2^{T_2} \cdot X_3^{T_3}$ by

$$X_1^{(T_1 \cup \bar{S}_1)} \cdot X_2^{(T_2 \cup \bar{S}_2)} \cdot X_3^{(T_3 \cup \bar{S}_3)} = X_1^{T_1} \cdot X_2^{(T_2 \cup \{0\})} \cdot X_3^{T_3},$$

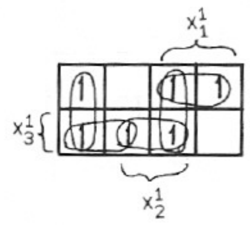

Fig. 3. Example 3.3.

we have

$$\mathcal{F}(|c) = X_1^0 \cdot X_2^{\{0,1\}} \cdot X_3^1 \vee X_1^{\{0,1\}} \cdot X_2^{\{0,1\}} \cdot X_3^1 \vee X_1^1 \cdot X_2^{\{0,1\}} \cdot X_3^{\{0,1\}} \vee X_1^1 \cdot X_2^{\{0,1\}} \cdot X_3^0.$$

Fig. 4 shows the map for $\mathcal{F}(|c)$. Note that because of $X_2^{\{0,1\}} = 1$, $\mathcal{F}(|c)$ is equivalent to $X_1^0 \cdot X_3^1 \vee X_3^1 \vee X_1^1 \vee X_1^1 \cdot X_3^0$.

We can also obtain $\mathcal{F}(|c)$ by using positional cubes as follows.

1) By making logical AND operation of $\mathcal{F}$ and c, and by deleting a null product, we have

$$\mathcal{F} \cdot c = \begin{bmatrix} 10 & 01 & 01 \\ 11 & 01 & 01 \\ 01 & 01 & 11 \\ 01 & 01 & 10 \end{bmatrix}$$

2) By replacing each product $X_1^{T_1} \cdot X_2^{T_2} \cdot X_3^{T_3}$ (i.e., each row in 1)) with

$$X_1^{(T_1 \cup \bar{S}_1)} \cdot X_2^{(T_2 \cup \bar{S}_2)} \cdot X_3^{(T_3 \cup \bar{S}_3)},$$

we have

$$\mathcal{F}(|c) = \begin{bmatrix} 10 & 11 & 01 \\ 11 & 11 & 01 \\ 01 & 11 & 11 \\ 01 & 11 & 10 \end{bmatrix}$$

If we use Definition 3.1 instead of Definition 3.2, the restriction of F becomes as follows:

X_1	X_2	X_3
10	1	01
11	1	01
01	1	11
01	1	10

Fig. 5 shows the map for the above cubes. Here, notice that the number of bit positions in each cubes is reduced to 5. Thus, it becomes difficult to perform logical AND or OR of the above cubes (with 5 bit positions) and other cubes (with 6 bit positions).

Notice that Figs. 4 and 5 represent the same function $X_1^0 \cdot X_3^1 \vee X_3^1 \vee X_1^1 \vee X_1^1 \cdot X_3^0$; however, they are based on different universal cubes, i.e., the maps which can be expressed by cubes with different bit positions.

(End of Example)

Lemma 3.1: $c \cdot \mathcal{F} = c \cdot \mathcal{F}(|c)$.

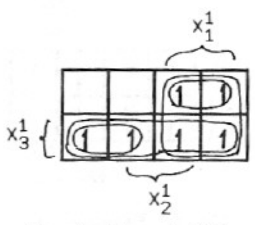

Fig. 4. Example 3.3.

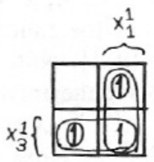

Fig. 5. Example 3.3.

Proof: This is obvious from Definition 3.1. (Q.E.D.)

Lemma 3.2: Let E_t $(t = 0, 1, \cdots, k)$ be arbitrary products, and $\mathcal{F}$ be an arbitrary expression. If $\bigvee_{t=0}^{k} E_t \equiv 1$ and $E_i \cdot E_j = \emptyset$ for $i \neq j$, then $\mathcal{F} = \bigvee_{t=0}^{k} E_t \cdot \mathcal{F}_t$ where $\mathcal{F}_t = \mathcal{F}(|E_t)$. The complement of $\mathcal{F}$ is given by $\overline{\mathcal{F}} = \bigvee_{t=0}^{k} E_t \cdot \overline{\mathcal{F}}_t$.

Proof:

$$\mathcal{F} = \left(\bigvee_{t=0}^{k} E_t\right) \cdot \mathcal{F} = \bigvee_{t=0}^{k} (E_t \cdot \mathcal{F}).$$

By Lemma 3.1, we have

$$\mathcal{F} = \bigvee_{t=0}^{k} E_t \cdot \mathcal{F}(|E_t) = \bigvee_{t=0}^{k} E_t \cdot \mathcal{F}_t.$$

By the complementation theorem of Hong-Ostapko [10], we have

$$\overline{\mathcal{F}} = \bigvee_{t=0}^{k} E_t \cdot \overline{\mathcal{F}}_t. \qquad \text{Q.E.D.}$$

Note that the first part of Lemma 3.2 is a generalization of the Shannon expansion. In fact, let $E_0 = X_i^{\{0\}}$, $E_1 = X_i^{\{1\}}, \cdots$, and $E_{p_i-1} = X_i^{\{p_i-1\}}$, and we have

$$\mathcal{F} = X_i^{\{0\}} \cdot \mathcal{F}_0 \vee X_i^{\{1\}} \cdot \mathcal{F}_1 \vee \cdots \vee X_i^{\{p_i-1\}} \cdot \mathcal{F}_{p_i-1}$$

where $\mathcal{F}_t = \mathcal{F}(|X_i^t)$.

Theorem 3.1: The complement of an arbitrary expression $\mathcal{F}$ is given by

$$\overline{\mathcal{F}} = \bigvee_{t=0}^{p_i-1} X_i^t \cdot \overline{\mathcal{F}}_t$$

where $\mathcal{F}_t = \mathcal{F}(|X_i^t)$ $(t = 0, 1, \cdots, p_i - 1)$.

Proof: Let $E_t = X_i^t$ $(t = 0, 1, \cdots, k{:}\ k = p_i - 1)$ in Lemma 3.2. Because E_t $(t = 0, 1, \cdots, k)$ satisfy the conditions of Lemma 3.2, we can easily obtain $\overline{\mathcal{F}} = \bigvee_{t=0}^{k} (E_t \cdot \overline{\mathcal{F}(|E_t)})$. (Q.E.D.)

Theorem 3.2: Let a function be represented by $\mathcal{F} = X_1^{S_1} \cdot X_2^{S_2} \cdots X_r^{S_r} \vee \mathcal{G}$; then the complement of $\mathcal{F}$ is given by

$$\overline{\mathcal{F}} = X_1^{\bar{S}_1} \cdot \overline{\mathcal{G}}_1 \vee X_1^{S_1} \cdot X_2^{\bar{S}_2} \cdot \overline{\mathcal{G}}_2 \vee \cdots \vee X_1^{S_1} \cdot X_2^{S_2} \cdots X_r^{\bar{S}_r} \cdot \overline{\mathcal{G}}_r$$

where $\mathcal{G}_t = X_1^{S_1} \cdot X_2^{S_2} \cdots X_t^{\bar{S}_t} \cdot \mathcal{G}$ and $t = 1, 2, \cdots, r$.

Proof: Let $E_0 = X_1^{S_1} \cdot X_2^{S_2} \cdots X_r^{S_r}$, and $E_t = X_1^{S_1} \cdot X_2^{S_2} \cdots X_t^{\bar{S}_t}$ $(t = 1, 2, \cdots, r)$ in Lemma 3.2. Because $E_t(t = 0, 1, \cdots, r)$ satisfy the conditions of Lemma 3.2, we have

$$\overline{\mathcal{F}} = \bigvee_{t=0}^{r} E_t \cdot \overline{\mathcal{F}(|E_t)} = \bigvee_{t=0}^{r} E_t \cdot \overline{E_t \cdot \mathcal{F}(|E_t)}\,.$$

Because $E_t \cdot \mathcal{F}(|E_t) = E_t \cdot \mathcal{F}$ by Lemma 3.1, we have

$$\mathcal{F} = \bigvee_{t=0}^{r} E_t \cdot \overline{E_t \cdot \mathcal{F}}\,.$$

Because $E_0 \cdot \overline{E_0 \mathcal{F}} = 0$ and $E_t \cdot \mathcal{F} = E_t \cdot \mathcal{G} = \mathcal{G}_t$ for $t = 1, 2, \cdots, r$, we have

$$\overline{\mathcal{F}} = \bigvee_{t=1}^{r} E_t \cdot \overline{\mathcal{G}}_t\,.$$

(Q.E.D.)

By iteratively applying Theorems 3.1 and 3.2, we can decompose the problem of complementation of a function into ones with less variables. We will continue the decomposition iteratively until all the functions become trivial ones. The following algorithm was developed after many experiments using large practical problems.

Algorithm 3.1 (Fast Complementation Algorithm): Let $\mathcal{F}$ be a given sum-of-products expression of F. Use the following rules recursively.

Rule 1) If $\mathcal{F}$ is a constant: i.e., if $\mathcal{F} = 1$, then $\overline{\mathcal{F}} = 0$, and if $\mathcal{F} = 0$, then $\overline{\mathcal{F}} = 1$.

Rule 2) If $\mathcal{F}$ depends on only one variable, i.e., if $\mathcal{F} = X_1^{S_a} \vee X_1^{S_b} \vee \cdots \vee X_1^{S_z}$ then $\overline{\mathcal{F}} = X_1^{\bar{S}}$ where $S = S_a \cup S_b \cup \cdots \cup S_z$ and $S_a, S_b, \cdots, S_z \subseteq P_1$.

Rule 3) If $\mathcal{F}$ consists of one product, i.e., if $\mathcal{F} = X_1^{S_1} \cdot X_2^{S_2} \cdots X_k^{S_k}$ then $\overline{\mathcal{F}} = X_1^{\bar{S}_1} \vee X_1^{S_1} \cdot X_2^{\bar{S}_2} \vee \cdots \vee X_1^{S_1} \cdot X_2^{S_2} \cdots X_{k-1}^{S_{k-1}} \cdot X_k^{\bar{S}_k}$.

Rule 4) If $\mathcal{F}$ has common factor, i.e., if $\mathcal{F}$ can be written as $\mathcal{F} = X_1^{S_1} \cdot X_2^{S_2} \cdots X_k^{S_k} \cdot \mathcal{G}$ by renaming variables where $\mathcal{G}$ does not contain variables $X_1, X_2, \cdots, X_k$, then

$$\overline{\mathcal{F}} = X_1^{\bar{S}_1} \vee X_1^{S_1} \cdot X_2^{\bar{S}_2} \vee \cdots \vee X_1^{S_1} \cdot X_2^{S_2} \cdots X_k^{\bar{S}_k} \vee X_1^{S_1} \cdot X_2^{S_2} \cdots X_k^{S_k} \cdot \overline{\mathcal{G}}\,.$$

Rule 5) If $\mathcal{F}$ can be decomposed with a variable X_i, i.e., if $\mathcal{F}$ can be written as

$$\mathcal{F} = X_i^0 \cdot \mathcal{G}_0 \vee X_i^1 \cdot \mathcal{G}_1 \vee \cdots \vee X_i^{p_i-1} \cdot \mathcal{G}_{p_i-1}$$

where $\mathcal{G}_k$ $(k = 0, 1, \cdots, p_i - 1)$ do not contain the variable X_i, then by Theorem 3.1 we have

$$\overline{\mathcal{F}} = X_i^0 \cdot \overline{\mathcal{G}}_0 \vee X_i^1 \cdot \overline{\mathcal{G}}_1 \vee \cdots \vee X_i^{p_i-1} \cdot \overline{\mathcal{G}}_{p_i-1}\,.$$

We use this rule only when every product of $\mathcal{F}$ has the literal $X_i^0, X_i^1, \cdots$, or $X_i^{p_i-1}$ for some i, and $\mathcal{G}_k \neq 0$ $(k = 0, 1, \cdots, p_i - 1)$.

Rule 6) If Rules 1)–5) cannot be applied, then $\mathcal{F}$ is a sum-of-products expression consisting of at least two products. By renaming the variables, it can be written as

$$\mathcal{F} = X_1^{S_1} \cdot X_2^{S_2} \cdot \cdots \cdot X_k^{S_k} \vee \mathcal{G}\,.$$

By Theorem 3.2, $\overline{\mathcal{F}}$ is given by

$$\overline{\mathcal{F}} = X_1^{\bar{S}_1} \cdot \overline{\mathcal{G}}_1 \vee X_1^{S_1} \cdot X_2^{\bar{S}_2} \cdot \overline{\mathcal{G}}_2 \vee \cdots \vee X_1^{S_1} \cdot X_2^{S_2} \cdots X_{k-1}^{S_{k-1}} \cdot X_k^{\bar{S}_k} \cdot \overline{\mathcal{G}}_k$$

where $\mathcal{G}_i = X_1^{S_1} \cdot X_2^{S_2} \cdots X_i^{\bar{S}_i} \cdot \mathcal{G}$.

Definition 3.2: A sum-of-products expression is *disjoint* if all products are mutually disjoint; i.e.,

$$\mathcal{F} = E_1 (\text{i.e., } s = 1) \quad \text{or} \quad \mathcal{F} = E_1 \vee E_2 \vee \cdots \vee E_s$$

where $E_i \cdot E_j = 0$ for $i \neq j$.

Theorem 3.3: Algorithm 3.1 generates disjoint sum-of-products expression for $\overline{F}$.

Proof: By Theorems 3.1 and 3.2, it is clear that Algorithm 3.1 generates the complement of $\mathcal{F}$. It is also easy to see that all the rules generate disjoint expressions.

(Q.E.D.)

As shown in Theorem 3.3, Algorithm 3.1 generates a disjoint sum-of-products expression. Disjoint sum-of-products expressions have many desirable properties, which are not utilized in the rest of this paper. (For example, if $\mathcal{F}_1$ and $\mathcal{F}_2$ are disjoint sum-of-products expressions, then $\mathcal{F}_3 = \mathcal{F}_1 \cdot \mathcal{F}_2$, obtained by using the distributive law only, is also disjoint sum-of-products expression. In disjoint sum-of-products expression, we can easily calculate the number of minterms of the function. We can quickly check the implication relation $\mathcal{F}_1 \leq \mathcal{F}_2$ by counting the numbers of minterms in the cubes of $\mathcal{F}_1$ and $\mathcal{F}_3$ because $\mathcal{F}_1 \leq F_2$ holds if and only if the number of minterms in $\mathcal{F}_1$ is equal to the number of minterms in $\mathcal{F}_3$.)

The number of products in $\overline{\mathcal{F}}$ is denoted by $t(\overline{\mathcal{F}})$. Because the number of products in a disjoint sum-of-products expression does not exceed the total number of the minterms in the universal cube, we have $t(\overline{\mathcal{F}}) \leqq \Pi_{i=1}^{n} p_i$ where $\overline{\mathcal{F}}$ is obtained by Algorithm 3.1.

Theorem 3.4: Let $\overline{\mathcal{F}}$ be a sum-of-products expression obtained by Algorithm 3.1; then $t(\overline{\mathcal{F}}) \leqq (\Pi_{i=1}^{n} p_i)/2$.

Proof: Let n be the number of variables of F. The proof will be done by the induction on n.

Corresponding to the rules in Algorithm 3.1, we have the following.

Rule 1) When $n = 0$: $t(\overline{\mathcal{F}}) \leqq 1$.

Rule 2) When $n = 1$: $t(\overline{\mathcal{F}}) \leqq 1$ and the theorem holds for $n = 1$.

Rule 3) When $\mathcal{F}$ consists of one product, $X_1^{S_1} \cdot X_2^{S_2} \cdots X_k^{S_k}$, then, $t(\overline{\mathcal{F}}) = k \leqq n \leqq 2^{n-1}$, and the theorem holds $(n \geqq 2)$.

From here, suppose that the theorem holds for the restriction of F, and any function with $n - 1$ or less variables.

Rule 4) When $\mathcal{F}$ has a common factor, i.e., $\mathcal{F}$ can be written as follows by renaming the variables:

$$\mathcal{F} = X_1^{S_1} \cdot X_2^{S_2} \cdots X_k^{S_k} \cdot \mathcal{G}\,,$$

then we have the following relation:

$$t(\overline{\mathcal{F}}) = t(X_1^{\bar{S}_1} \vee X_1^{S_1} \cdot X_2^{\bar{S}_2} \vee \cdots \vee X_1^{S_1} \cdot X_2^{S_2} \cdots X_k^{\bar{S}_k}) + t(\overline{\mathcal{G}})\,.$$

Since $\mathcal{G}$ does not contain the variables $X_1, X_2, \cdots, X_k$, it has at most $(n - k)$ variables. By the induction hypothesis,

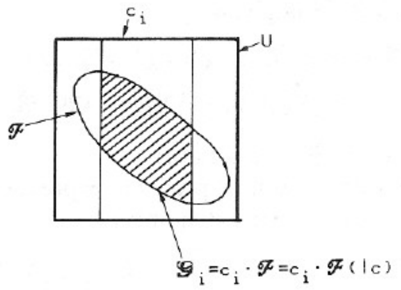
c_i
U
$\mathscr{F}$
$\mathscr{G}_i = c_i \cdot \mathscr{F} = c_i \cdot \mathscr{F}(|c)$

Fig. 6. Proof of Theorem 3.4.

we have

$$t(\overline{\mathscr{G}}) \leqq \frac{1}{2} \prod_{i=k+1}^{n} p_i .$$

Thus,

$$t(\overline{\mathscr{F}}) \leqq k + \frac{1}{2} \prod_{i=k+1}^{n} p_i \leqq \frac{1}{2} \prod_{i=1}^{n} p_i$$

and the theorem holds.

Rule 5) When $\mathscr{F}$ can be decomposed with respect to X_i; i.e., $\mathscr{F}$ can be written as

$$\mathscr{F} = X_1^0 \cdot \mathscr{G}_0 \vee X_1^1 \cdot \mathscr{G}_1 \vee \cdots \vee X_1^{p_1-1} \cdot \mathscr{G}_{p_1-1},$$

by renaming the variables we have

$$t(\overline{\mathscr{F}}) = \sum_{k=0}^{p_1-1} t(\overline{\mathscr{G}}_k)$$

where $\mathscr{G}_k$ $(k = 0, 1, \cdots, p_{i-1})$ do not contain variable X_1.

By the induction hypothesis, $t(\overline{\mathscr{G}}_k) \leqq (\prod_{j=2}^{n} p_j)/2$. Hence,

$$t(\overline{\mathscr{F}}) \leqq p_1 \times \frac{1}{2} \prod_{j=2}^{n} p_j = \frac{1}{2} \prod_{j=1}^{n} p_j ,$$

and the theorem holds.

Rule 6) When Rules 1)–5) cannot be applied, $\mathscr{F}$ can be written as $\mathscr{F} = X_1^{S_1} \cdot X_2^{S_2} \cdots X_k^{S_k} \vee \mathscr{G}$ by renaming the variables. Because $\overline{\mathscr{F}} = X_1^{\overline{S}_1} \cdot \overline{\mathscr{G}}_1 \vee X_1^{S_1} \cdot X_2^{\overline{S}_2} \overline{\mathscr{G}}_2 \vee \cdots \vee X_1^{S_1} \cdot X_2^{S_2} \cdots X_k^{\overline{S}_k} \cdot \overline{\mathscr{G}}_k$, we have $t(\overline{\mathscr{F}}) = \sum_{i=1}^{k} t(X_1^{S_1} \cdot X_2^{S_2} \cdots X_i^{\overline{S}_i} \cdot \overline{\mathscr{G}}_i)$.

Let $U = X_1^{P_1} \cdot X_2^{P_2} \cdots X_n^{P_n}$; then U denotes the universal cube for $\mathscr{F}$. Because $\overline{\mathscr{F}} = U \cdot \overline{\mathscr{F}}$, the theorem can be restated as follows:

$$t(U \cdot \overline{\mathscr{F}}) \leqq \frac{1}{2} \qquad \text{(number of minterms in } U).$$

By the induction hypothesis, the theorem holds for the restriction of $\mathscr{F}$. Therefore, we have

$$t(c_i \cdot \overline{\mathscr{G}}_i) \leqq \frac{1}{2} \qquad \text{(number of minterms in } c_i)$$

where $\mathscr{G}_i = c_i \cdot \mathscr{F} = c_i \cdot \mathscr{F}(|c_i)$, and $c_i = X_1^{S_1} \cdot X_2^{S_2} \cdots X_i^{\overline{S}_i} \cdot X_{i+1}^{P_{i+1}} \cdots X_n^{P_n}$. In this case, c_i can be regarded as the universal cube for $\mathscr{G}_i$ (see Fig. 6) as U was for $\mathscr{F}$. Hence, we have

$$t(c_i \cdot \overline{\mathscr{G}}_i) \leqq \frac{1}{2} \left(\prod_{r=1}^{i-1} a_r \right) \cdot (p_i - a_i) \cdot \left(\prod_{r=i+1}^{n} p_r \right).$$

Let $b_r = a_r/p_r$ $(r = 1, 2, \cdots, i - 1)$; we have

$$t(c_i \cdot \overline{\mathscr{G}}_i) \leqq \frac{1}{2} \left(\prod_{r=1}^{n} p_r \right) \cdot \left(\prod_{r=1}^{i-1} b_r \right) \cdot (1 - b_i).$$

Therefore,

$$\begin{aligned} t(\overline{\mathscr{F}}) &\leqq \sum_{i=1}^{k} \frac{1}{2} \left(\prod_{r=1}^{n} p_r \right) \cdot \left(\prod_{r=1}^{i-1} b_r \right) \cdot (1 - b_i) \leqq \frac{1}{2} \left(\prod_{r=1}^{n} p_r \right) \\ &\quad \cdot \{(1 - b_1) + b_1 \cdot (1 - b_2) + \cdots + b_1 \\ &\qquad \cdot b_2 \cdots b_{k-1} \cdot (1 - b_k)\} \leqq \frac{1}{2} \left(\prod_{r=1}^{n} p_r \right) \\ &\quad \cdot (1 - b_1 \cdot b_2 \cdots b_k). \end{aligned}$$

Since $b_r > 0$, we have $t(\overline{\mathscr{F}}) < 1/2(\prod_{i=1}^{n} p_i)$ and the theorem holds.

We have exhausted all possible cases and proved the theorem by induction. (Q.E.D.)

Table I shows values of G_p, the average number of the prime implicants for functions with p-valued inputs, and $p^n/2$ for $p = 2$ and $p = 4$ [8]. (Statistical data for $p = 2$ are also shown in [9, p. 123].) By Theorem 2.1, we can see that Algorithm 2.1 produces $G_p(n)$ products on the average. On the other hand, by Theorem 3.4, Algorithm 3.1 produces at most $p^n/2$ products. From Table I, we can see that Algorithm 3.1 produces fewer products than Algorithm 2.1, when $p = 2$ and $n \geqq 10$ or when $p = 4$ and $n \geqq 4$.

In Rule 6) of Algorithm 3.1, the selection of products and the ordering of variables drastically influences the efficiency of the algorithm. After doing many experiments using practical PLA's, the following heuristics have been obtained.

Heuristic 3.1 (Selection of Products): For each product $E = X_1^{S_1} \cdot X_2^{S_2} \cdots X_n^{S_n}$ in the expression $\mathscr{F}$, calculate $L(E)$ = (number of variables such that $S_i \neq P_i$). Choose a product with the minimum $L(E)$. If there is more than one candidate, choose one with the maximum $\sum_{i=1}^{n} |S_i|$.

(Explanation of Heuristic 3.1): Let $E = X_1^{S_1} \cdot X_2^{S_2} \cdots X_n^{S_n}$. If $S_i = P_i$, then literal $X_i^{S_i}$ denotes a constant 1 and can be deleted from E. Thus, $L(E)$ denotes the actual number of literals in E. In Rule 6, we can minimize the number of the subfunctions $\mathscr{G}_i$ by minimizing $L(E) = k$.

(End of the Explanation)

Heuristic 3.2 (Ordering of the Variables): Rearrange the variables in the product according to the order shown in the ORD, which is obtained by the following algorithm.

Algorithm 3.2 (Ordering of the Variables): Suppose that $\mathscr{F}$ can be written as $\mathscr{F} = c \vee \mathscr{G}$ where $c = X_1^{S_1} \cdot X_2^{S_2} \cdots X_n^{S_n}$, $\mathscr{G} = g_1 \vee g_2 \vee \cdots \vee g_m$. Suppose that each g_i has a form $g_i = X_1^{T_1(i)} \cdot X_2^{T_2(i)} \cdots X_n^{T_n(i)}$ where $T_j(i) \subseteq P_j$ for $i = 1, 2, \cdots, m$, and $j = 1, 2, \cdots, n$.

1) Let $H0$ and $H1$ be 0-1 matrices with m rows and n columns as follows:

$$H0[i,j] = \begin{cases} 1 & \text{if } \overline{S}_i \cap T_i(j) \neq \emptyset \\ 0 & \text{if } \overline{S}_i \cap T_i(j) = \emptyset \end{cases}$$

$$H1[i,j] = \begin{cases} 1 & \text{if } S_i \cap T_i(j) \neq \emptyset \\ 0 & \text{if } S_i \cap T_i(j) = \emptyset . \end{cases}$$

TABLE I
COMPARISON OF $G_p(n)$ TO $p^n/2$

	n	6	8	10	12	14
When p=2	$G_2(n)$	24	118	585	2,902	14,255
	$2^n/2$	32	128	512	2,048	8,192
	n	3	4	5	6	7
When p=4	$G_4(n)$	24	136	758	4,095	21,565
	$4^n/2$	32	128	512	2,048	8,192

2) Initialize index I to 1.

3) Let $h0$ and $h1$ be vectors as follows:

$$h0[j] = \sum_i H0[i,j]$$

and

$$h1[j] = \sum_i H1[i,j]$$

where Σ_i denotes an arithmetic sum for all i.

4) Let h be a vector as follows:

$$h[j] = \min\{h0[j], h1[j]\}.$$

5) Find J such that $h[j]$ is the minimum.

Set Jth element of a one-dimensional array with n elements, ORD, to the number stored in I.

Increment I.

6) Find the rows such that $H1[i,J] = 0$.

From $H0$ and $H1$, delete these rows and delete column J.

7) If $H0$ and $H1$ are null, then stop. Else go to step 3).

(Explanation of Heuristic 3.2): Consider the expression in Rule 6):

$$\begin{aligned}\overline{\mathcal{F}} = {} & X_1^{\overline{S}_1} \cdot \overline{\mathcal{G}}_1 \vee X_1^{S_1} \cdot X_2^{\overline{S}_2} \cdot \overline{\mathcal{G}}_2 \vee \cdots \\ & \vee X_1^{S_1} \cdot X_2^{S_2} \cdots X_{k-1}^{S_{k-1}} \cdot X_k^{\overline{S}_k} \cdot \overline{\mathcal{G}}_k .\end{aligned}$$

We want to obtain an ordering of variables $X_1, X_2, \cdots, X_k$ which makes $\overline{\mathcal{F}}$ as simple as possible, but this is not so easy for this general case. So in Heuristic 3.2, let us consider the following simpler problem.

First, rewrite $\mathcal{F}$ as follows:

$$\mathcal{F} = X_j^{\overline{S}_j} \cdot \mathcal{G}_1^* \vee X_j^{S_j} \cdot \mathcal{G}_2^*$$

where $\mathcal{G}_1^* = \mathcal{F}(|X_j^{\overline{S}_j})$ and $\mathcal{G}_2^* = \mathcal{F}(|X_j^{S_j})$. Note that by Lemma 3.2

$$\overline{\mathcal{F}} = X_j^{\overline{S}_j} \cdot \overline{\mathcal{G}}_1^* \vee X_j^{S_j} \cdot \overline{\mathcal{G}}_2^* .$$

Here, let us consider the problem to find a variable j which makes either $\mathcal{G}_1^*$ or $\mathcal{G}_2^*$ as simple as possible. When both $t(\mathcal{G}_1)^*$ and $t(\mathcal{G}_2)^*$ are small, we can expect that $t(\overline{\mathcal{F}})$ is also small because of the following empirical rule. "The smaller $t(\mathcal{G}_i^*)$, the smaller $t(\overline{\mathcal{G}}_i^*)$ $(t = 1$ and $2)$."

When $t(\mathcal{G}_1^*)$ is small but $t(\mathcal{G}_2^*)$ is large, we can expect that $t(\overline{\mathcal{G}}_1^*)$ is small. For $\mathcal{G}_2^*$, it is also possible to make the complement small by finding the best ordering of variables by the iterative application of this procedure. When $t(\mathcal{G}_1^*)$ is large, but $t(\mathcal{G}_2^*)$ is small, we can do similar things for $\mathcal{G}_1^*$ to make the complement as simple as possible.

In Step (4), $h0[j]$ shows the number of products in $\mathcal{G}_1^*$ and $h1[i]$ shows the number of products in $\mathcal{G}_2^*$ when $X_j^{S_j}$ is used to expand $\mathcal{F}$. Therefore, we will find a variable X_J which makes $h[j]$ minimum. Then, X_J is the first variable to expand.

In Step (6), from $\mathcal{G}$ shown in the beginning of Algorithm 3.2, we delete products which are not element of $\mathcal{G}_2^*$.

For $\mathcal{G}_2^*$, we will use the same technique repeatedly until the ordering of all the remaining variables are determined.

(End of Explanation)

Although Heuristic 3.2 is more complicated than that of [8], it usually produces fewer products for practical problems with many inputs.

Example 3.1: Consider the expression shown in Example 2.1

$$\mathcal{F} = X_1^0 \cdot X_2^1 \cdot X_3^{\{1,3\}} \vee X_1^1 \cdot X_2^{\{0,2\}} \cdot X_3^{\{1,2\}} \vee X_2^{\{1,2\}} \cdot X_3^1 .$$

Because we cannot use Rules 1)–5), we use Rule 6). By Heuristic 3.1, we choose the product $X_2^{\{1,2\}} \cdot X_3^1$ because it is the product with the least number of literals. Thus, $\mathcal{F}$ is written as follows:

$$\mathcal{F} = X_2^{\{1,2\}} \cdot X_3^1 \vee \mathcal{G}$$

where

$$\mathcal{G} = X_1^0 \cdot X_2^1 \cdot X_3^{\{1,3\}} \vee X_1^1 \cdot X_2^{\{0,2\}} \cdot X_3^{\{1,2\}}.$$

Then, we have to obtain the ordering of variables by Heuristic 3.2. The obtained order of expansion is first X_2 and then X_3 (the details will be shown in Example 3.2.) The complement of $\mathcal{F}$ is given by

$$\begin{aligned}\overline{\mathcal{F}} &= (X_2^0 \vee X_2^{\{1,2\}} \cdot X_3^{\{0,2,3\}}) \cdot \overline{\mathcal{G}} \\ &= X_2^0 \cdot \overline{\mathcal{G}}_1 \vee X_2^{\{1,2\}} \cdot X_3^{\{0,2,3\}} \cdot \overline{\mathcal{G}}_2\end{aligned}$$

where

$$\mathcal{G}_1 = X_2^0 \cdot \mathcal{G} = X_1^1 \cdot X_2^0 \cdot X_3^{\{1,2\}}$$

and

$$\mathcal{G}_2 = X_2^{\{1,2\}} \cdot X_3^{\{0,2,3\}} \cdot \mathcal{G} = X_1^0 \cdot X_2^1 \cdot X_3^3 \vee X_1^1 \cdot X_2^2 \cdot X_3^2 .$$

Next, let us obtain $\overline{\mathcal{G}}_1$ and $\overline{\mathcal{G}}_2$.

$\mathcal{G}_1$ consists of one product. By Rule 3), we have

$$\overline{\mathcal{G}}_1 = X_1^0 \vee X_1^1 \cdot X_2^{\{1,2\}} \vee X_1^1 \cdot X_2^0 \cdot X_3^{\{0,3\}}.$$

By Rule 5), $\overline{\mathcal{G}}_2$ can be written as

$$\begin{aligned}\overline{\mathcal{G}}_2 = {} & X_1^0 \cdot (X_2^{\{0,2\}} \vee X_2^1 \cdot X_3^{\{0,1,2\}}) \\ & \vee X_1^1 \cdot (X_2^{\{0,1\}} \vee X_2^2 \cdot X_3^{\{0,1,3\}}) .\end{aligned}$$

Hence,

$$\begin{aligned}\overline{\mathcal{F}} = {} & X_2^0 \cdot (X_1^0 \vee X_1^1 \cdot X_2^{\{1,2\}} \vee X_1^1 \cdot X_2^0 \cdot X_3^{\{0,3\}}) \\ & \vee X_2^{\{1,2\}} \cdot X_3^{\{0,2,3\}} \cdot (X_1^0 \cdot (X_2^{\{0,2\}} \vee X_2^1 \\ & \cdot X_3^{\{0,1,2\}}) \vee X_1^1 \cdot (X_2^{\{0,1\}} \vee X_2^2 \cdot X_3^{\{0,1,3\}}) . \\ = {} & X_1^0 \cdot X_2^0 \vee X_1^1 \cdot X_2^0 \cdot X_3^{\{0,3\}} \vee X_1^0 \cdot X_2^2 \cdot X_3^{\{0,2,3\}} \\ & \vee X_1^0 \cdot X_2^1 \cdot X_3^{\{0,2\}} \vee X_1^1 \cdot X_2^1 \cdot X_3^{\{0,2,3\}} \\ & \vee X_1^1 \cdot X_2^2 \cdot X_3^{\{0,3\}}.\end{aligned}$$

Note that $\overline{\mathcal{F}}$ contains 6 products. (End of Example)

TABLE II
NUMBER OF PRODUCTS IN THE COMPLEMENTS AND THEIR COMPUTATION TIME FOR RANDOMLY GENERATED FUNCTIONS

Input Data	Number of minterms in F	$t(\mathcal{F})$	Algorithm 2.1		Disjoint sharp		Algorithm 3.1	
			CPU time (sec)	$t(\overline{\mathcal{F}})$	CPU time (sec)	$t(\overline{\mathcal{F}})$	CPU time (sec)	$t(\overline{\mathcal{F}})$
p=2 n=8	32	23	11.89	180	.69	64	.72	55
	64	43	24.19	190	1.36	86	1.20	73
	96	50	21.41	139	1.59	82	1.60	77
	128	61	21.76	112	1.98	81	1.77	67
p=4 n=4	32	20	9.98	231	.34	42	.47	37
	64	35	20.19	252	.61	52	.73	52
	96	43	21.19	167	.77	57	.94	54
	128	50	17.15	142	.90	52	.93	51

F is a function with n p-valued inputs.

$\mathcal{F}$: sum-of-products expression for F. $t(\mathcal{F})$: number of products in $\mathcal{F}$.

$\overline{\mathcal{F}}$: sum-of-products expression for $\bar{F}$. $t(\overline{\mathcal{F}})$: number of products in $\overline{\mathcal{F}}$.

Example 3.2: Let us obtain the ordering of the variables shown in Example 3.1.

1) $c = X_1^{\{0,1\}} \cdot X_2^{\{1,2\}} \cdot X_3^1$

$$H0 = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix}$$

$$H1 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

2) $I = 1$.

3) $h0 = [0 \quad 1 \quad 2]$
$h1 = [2 \quad 2 \quad 2]$.

4) $h = [0 \quad 1 \quad 2]$.

5) Because $h[1]$ is the minimum, ORD[1] = 1. $I = 2$.

6) There is no row such that $H1[i, 1] = 0$. Delete column 1 from $H0$ and $H1$.

3) $(h0[2] \quad h0[3]) = [1 \quad 2]$
$(h1[2] \quad h1[3]) = [2 \quad 2]$.

4) $(h[2] \quad h[3]) = [1 \quad 2]$.

5) Because $h[2]$ is minimum, ORD[2] = 2. $I = 3$.

6) There is no row such that $H1[i, 2] = 0$. Delete column 2. Similarly, we have ORD[3] = 3.

Hence, ORD = [1 2 3]. (End of Example)

IV. EXPERIMENTAL RESULTS

Algorithm 3.1 has been programmed in APL and compared to other algorithms written in APL.

1) Table II compares Algorithm 2.1 based on sharp algorithm [11], [12], the disjoint sharp algorithm used in MINI [5], and Algorithm 3.1. Computational experiments were done as follows.

First, truth tables for 8-variable switching functions were randomly generated. Then, the functions were simplified by the distance-one-merge algorithm of MINI [5]. $t(\mathcal{F})$ denotes the number of the products in a simplified expression. Lastly, the complements of the expressions were obtained. $t(\overline{\mathcal{F}})$ denotes the number of products in the complement $\overline{\mathcal{F}}$. Table II shows that the disjoint sharp algorithm and Algorithm 3.1 are 10–20 times faster than Algorithm 2.1 and generate many fewer products (see the entries for $n = 8$ and $p = 2$). Also, the truth tables of 8-variable switching functions were decoded to make 4-variable binary functions of 4-valued inputs (i.e., a PLA with two-bit decoders). Also in this case, the disjoint sharp and Algorithm 3.1 were faster and produced many fewer products (see the entries for $n = 4$ and $p = 4$).

2) Table III shows the comparison of disjoint sharp, Algorithm 3.1, and a fast recursive complementation algorithm used by Espresso [13]. Control logic networks for microprocessors were used to compare the performance of these three algorithms [14]. For example, see the entries for $D2$, which is for an 8-input 7-output network. For standard PLA (two-level PLA), a simplified expression has 43 products. Also, for a PLA with two-bit decoders [2], [3], a simplified expression has 42 products. Table III shows that Algorithm 3.1 generates fewer products than other algorithms. This is a desirable property because in MINI or Espresso the algorithms often produce an excessive number of products which makes computing the initial phase of minimization impossible for large practical problems. Note that the complementation algorithm used by Espresso does not treat functions with multiple-valued variables, i.e., decoded PLA's. Also, it uses different data structure from Algorithm 3.1. The algorithm of Espresso usually produces many more products than Algorithm 3.1 because the former is designed to obtain the complement quickly regardless of the number of the products, whereas Algorithm 3.1 tries to obtain the complement as few as possible.

V. APPLICATION

Since Algorithm 3.1 usually produces many fewer products than the disjoint sharp algorithm in a comparable computation time, we exclusively use Algorithm 3.1 instead of disjoint sharp in our PLA minimization system [14].

1) Complementation of $\mathcal{F}$ (discussed in Sections I and IV): We have developed both APL and Fortran versions of MINI systems, which run on main frame computers with large memory spaces, as well as a MINI which runs on a personal computers with 8086 used. In the latter case, the limitation on the memory space is very severe. Heuristic 3.2 is the key of the algorithm which minimizes large practical

TABLE III
NUMBER OF PRODUCTS IN THE COMPLEMENTS AND THEIR COMPUTATION TIME FOR CONTROL LOGIC NETWORKS FOR MICROPROCESSORS

Input data					Disjoint sharp		Algorithm 3.1		Algorithm of ESPRESSO [6],[13]	
Circuit name	n	p	m	$t(\mathcal{F})$	CPU time (sec)	$t(\overline{\mathcal{F}})$	CPU time (sec)	$t(\overline{\mathcal{F}})$	CPU time (sec)	$t(\overline{\mathcal{F}})$
D2	8	2	7	43	1.57	125	1.71	68	.49	86
	4	4	7	42	.94	106	1.51	80	---	---
R1	8	2	31	33	1.33	123	1.06	31	.47	142
	4	4	31	32	.82	63	.87	33	---	---
I1	16	2	17	110	8.51	333	4.87	211	6.20	1186
	8	4	17	103	4.86	288	3.63	208	---	---
I4	32	2	20	222	60.76	3042	20.08	596	3.66	499
	16	4	20	204	30.23	1633	27.41	851	---	---
I5	24	2	14	62	7.37	918	6.51	271	1.78	415
	12	4	14	61	4.49	1100	6.49	310	---	---
A2	10	2	8	89	4.46	228	7.12	185	.91	192
	5	4	8	83	2.62	216	5.91	162	---	---

F is a function with n p-valued input and m outputs.
For the definition of other symbols, see the notes of Table II.

PLA's on a small memory space.

2) *REDUCE algorithm used in MINI [5]:* The original version of REDUCE algorithm in MINI is very time consuming. By using Algorithm 3.1, it becomes 10–20 times faster than the original one.

3) *Output phase optimization of PLA's [14]:* The complement of the multiple-output functions is required to obtain a near optimal output phase assignment.

4) *Essential prime implicant detection [14]:* In order to find essential prime implicants [9], we have to check the relation $c \leq \mathcal{H}$ many times. Algorithm 3.1 can be modified to check quickly whether $c \leq \mathcal{H}$ or not (see also [15]).

Other application to logic design are as follows.

5) *Verification of the equivalence of two logical expressions [16]:* Boolean comparison is a design verification technique in which two logic networks are compared for functional equivalence. It was used on the IBM 3081 project to establish that hardware flowcharts and the details of hardware logic design were functionally equivalent. Usually, equivalence of two logical expressions $\mathcal{F}$ and $\mathcal{G}$ is verified by the following method [11]:

$$\mathcal{F} \equiv \mathcal{G} \leftrightarrow \{\mathcal{F} \# \mathcal{G} = \emptyset \quad \text{and} \quad \mathcal{G} \# \mathcal{F} = \emptyset\}.$$

By slightly modifying Algorithm 3.1, we can more efficiently verify $\mathcal{F} \cdot \overline{\mathcal{G}} = \emptyset$ and $\mathcal{G} \cdot \overline{\mathcal{F}} = \emptyset$ (see also [15]).

6) *Test generation:* Boolean difference [17] can be calculated efficiently by using Algorithm 3.1. (Recently, a fast program to obtain the test for PLA's was developed, which also uses a partitioned algorithm [18].)

7) *Conversion of a product-of-sums expression into a sum-of-products form:* It is known that the straightforward method is usually very time consuming (see [9, p. 106]). Algorithm 3.1 is also useful for such conversion.

VI. CONCLUSIONS

This paper derived Algorithm 3.1 which generates at most $p^n/2$ products. Our experiments concluded that Algorithm 3.1 is 10–20 times faster than the conventional elementary method when $n = 8$ and $p = 2$, and Algorithm 3.1 produces many fewer products than the disjoint sharp algorithm used by MINI for large practical problems.

ACKNOWLEDGMENT

Part of this work was done while the author was a Visiting Scientist at the IBM T. J. Watson Research Center. The author is grateful to Dr. R. K. Brayton who provided all the data for the control logic networks as well as the results of his program Espresso, and also Dr. S. J. Hong who supplied the original APL MINI program.

REFERENCES

[1] B. Dunham and R. Fridshal, "The problem of simplifying logical expressions," *J. Symbol. Logic,* vol. 24, pp. 17–19, 1959.

[2] T. Sasao, "Multiple-valued decomposition of generalized Boolean functions and the complexity of programmable logic arrays," *IEEE Trans. Comput.,* vol. C-30, pp. 635–643, Sept. 1981.

[3] S. Muroga, *VLSI System Design.* New York: Wiley-Interscience, 1982.

[4] M. Davio, J. P. Deschamps, and A. Thayse, *Discrete and Switching Functions.* New York: George and McGraw-Hill, 1978.

[5] S. J. Hong, R. G. Cain, and D. L. Ostapko, "MINI: A heuristic approach for logic minimization," *IBM J. Res. Devel.,* pp. 443–458, Sept. 1974.

[6] R. K. Brayton, G. D. Hachtel, L. A. Hemachandra, A. R. Newton, and A. L. M. Sangiovanni-Vincentelli, "A comparison of logic minimization strategies using ESPRESSO: An APL program package for partitioned logic minimization," in *Proc. 1982 Int. Symp. Circ. Syst.,* May 1982, pp. 42–48.

[7] R. J. Nelson, "Simplest normal truth function," *J. Symbol. Logic,* vol. 20, pp. 105–108, June 1954.

[8] T. Sasao, "A fast complementation algorithm for sum-of-products expressions of multiple-valued input binary functions," in *Proc. 13th Int. Symp. Multiple-Valued Logic,* May 1983, pp. 103–110.

[9] S. Muroga, *Logic design and Switching Theory.* New York: Wiley-Interscience, 1979, p. 106.

[10] S. J. Hong and D. L. Ostapko, "On complementation of Boolean functions," *IEEE Trans. Comput.,* vol. C-21, p. 1022, 1972.

[11] D. L. Dietmeyer, *Logic Design of Digital Systems (2nd ed.).* Boston, MA: Allyn and Bacon, 1978.

[12] S. Y. H. Su and P. Y. Cheung, "Computer simplification of multi-valued switching functions," in *Computer Science and Multiple-Valued Logic*. Amsterdam, The Netherlands: North-Holland, 1977, pp. 189–220.

[13] R. K. Brayton, J. D. Cohen, G. D. Hachtel, B. M. Tragger, and D. Y. Y. Yun, "Fast recursive Boolean function manipulation," in *Proc. 1982 Int. Symp. Circ. Syst.*, May 1982, pp. 58–62.

[14] T. Sasao, "Input variable assignment and output phase optimization of PLA's," *IEEE Trans. Comput.*, vol. C-33, pp. 879–894, Oct. 1984.

[15] ——, "Tautology checking algorithm for multiple-valued input binary functions and their application," in *Proc. 14th Int. Symp. Multiple-Valued Logic*, May 1984, pp. 242–250.

[16] G. L. Smith, R. J. Bahnsen, and H. Halliwell, "Boolean comparison of hardware and flowcharts," *IBM J. Res. Devel.*, pp. 106–116, Jan. 1982.

[17] F. F. Sellers, M. Y. Hsiao, and C. L. Bearnson, "Analyzing errors with Boolean difference," *IEEE Trans. COmput.*, vol. C-18, pp. 678–683, 1968.

[18] F. Somenzi, S. Gai, M. Mezzalama, and P. Prinetto, "Part: Programmable ARray Testing based on a PARTitioning algorithm," *IEEE Trans. Comput. Aided Design Integ. Circ. Syst.*, vol. CAD-3, no. 2, pp. 143–149, Apr. 1984.

Tsutomu Sasao (S'72–M'77) was born in Osaka, Japan, on January 26, 1950. He received the B.E., M.E., and Ph.D. degrees in electronic engineering from Osaka University, Osaka, Japan, in 1972, 1974, and 1977, respectively.

Since 1977 he has been with Osaka University. His research interests include design automation of digital systems, switching theory, and application of microprocessors. He specializes in the design of PLA and application of multiple-valued logic to the design automation. From February 1982, he spent a year as a Visiting Scientist at the IBM T. J. Watson Research Center, Yorktown Heights, NY, where he developed a PLA minimization system.

Dr. Sasao served as Asia Area Program Chairman of the 1984 International Syposium on Multiple-Valued Logic, and is currently a member of the Executive Committee of the IEEE Computer Society Technical Committee on Multiple-Valued Logic. He has published three books on switching theory and logical design in Japanese. He is a member of the Institute of Electronics and Communication Engineers of Japan. He received the NIWA Memorial Award in 1979.