A FAST COMPLEMENTATION ALGORITHM FOR SUM-OF-PRODUCTS EXPRESSIONS OF MULTIPLE-VALUED INPUT BINARY FUNCTIONS

Tsutomu Sasao

Mathematical Sciences Department
IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

and

Department of Electronics Engineering
Osaka University
Osaka 565, Japan

ABSTRACT: A recursive algorithm to obtain a complement of a sum-of-products expression for a binary function of p-valued input variables is presented. It produces at most $p^n/2$ products for n-variables functions, whereas an elementary algorithm produces $O(t^n \cdot n^{(1-t)/2})$ products where $t = 2^p - 1$. It is 10 ~ 30 times faster then the elementary one when p=2 and n=8.

I. Introduction:

As an elementary method to obtain a complement of a sum-of-products expression for f, the following is well known.

1. By using De Morgan's law, obtain a product-of-sums expression for $\bar{f}$.
2. By using the distributive law, obtain a sum-of-products expression for $\bar{f}$.
3. By using the absorption law, simplify the sum-of-products expression.

However, this method becomes quite inefficient when the number of input variables is large, because it will produce all the prime implicants of $\bar{f}$. For example, the elementary method will generate $O(3^n/n)$ products for a class of n-variable switching functions (two-valued input binary functions) [1], whereas the presented algorithm will generate at most 2^{n-1} products. The new algorithm is about 10 ~ 30 times faster than the elementary one for switching functions of 8-variables.

Binary functions are useful in designing programmable logic arrays with decoders [2] and other circuits [3], [4]. Simplification of the expressions for the binary functions will reduce the complexities of circuits. A fast complementation algorithm has been desired because practical minimization algorithms such as MINI [5] and ESPRESSO [6] require the complement of the given function.

The proposed algorithm has been incorporated into MINI and other systems and has been effectively used to design logical circuits.

II. Definitions and an Elementary Method for Complementation

Definition 2.1[4]: A mapping $\times_{i=1}^{n} P_i \rightarrow B$ is called a multiple-valued input binary function, where $P_i = \{0,1,\ldots, p_i-1\}$, and $B = \{0, 1\}$.

Definition 2.2: Let X_i be a variable on P_i. $X_i^{S_i}$ is a literal of X_i when $S_i \subseteq P_i$.
$X_i^{S_i}$ represent a function

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

Definition 2.3: A product of literals $X_1^{S_1} \cdot X_2^{S_2} \cdot \ldots \cdot X_n^{S_n}$ is called a product. A sum of products is called a sum-of-products expression.

Theorem 2.1[2]: An arbitrary binary function $\times_{i=1}^{n} P_i \rightarrow B$ can be represented by a sum-of-products expression

$$f(X_1, X_2, \ldots, X_n) = \bigvee_{(S_1,S_2,\ldots,S_n)} X_1^{S_1} \cdot X_2^{S_2} \cdot \ldots \cdot X_n^{S_n}$$

where $S_i \subseteq P_i$.

Definition 2.4: Let E be a product. E is called a prime implicant if $E \leq f$ and E is maximal (i.e. there is no E_1 such that $E, E_1 \leq f$).

Lemma 2.1: Let f, g and h be binary functions.

$\overline{f \cdot g} = \bar{f} \vee \bar{g}$ (De Morgan's law)

$(f \vee g) \cdot h = f \cdot h \vee g \cdot h$ (Distributive law)

$f \vee f \cdot g = f$ (Absorption law)

As an elementary method to obtain a complement of

sum-of-products expression, the following is well known.

Algorithm 2.1:

1. By using De Morgan's law, convert a complement of a given expression into a product-of-sums form.

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

3. By using the absorption law, drop subsuming products ($p \vee pq = p$).

Example 2.1:

Consider a binary function

$$f: \{0,1\} \times \{0,1,2\} \times \{0,1,2,3\} \rightarrow \{0,1\}$$

and an expression

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

Let's obtain a complement of $\mathcal{F}$ by Algorithm 2.1. First, by De Morgan's law, convert it into a product-of-sums form.

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

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

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

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

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

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

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

Note that $\bar{\mathcal{F}}_1$ contains 8 products.

(End of Example)

Straightforward application of Algorithm 2.1 is quite inefficient. It might be made more efficient by the simplifying the intermediate results by using absorption law or by changing the order of expansion. However, in any case, Algorithm 2.1 will generate many products. This is because that Algorithm 2.1 will generate all the prime implicants of $\bar{f}$, which is stated by the following theorem.

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

(Proof). Similar to the switching function [7]

(Q.E.D)

III. A Fast Complementation Algorithm

By extending Shannon's expansion theorem to binary functions and applying the complementation theorem of Hong-Ostapko [9], we have the following:

Lemma 3.1: Let a binary function be represented by

$$f = X^0 \cdot f_0 \vee X^1 \cdot f_1 \vee \ldots \vee X^{p-1} \cdot f_{p-1}.$$

Then a complement of f is given by

$$\bar{f} = X^0 \bar{f}_0 \vee X^1 \cdot \bar{f}_1 \vee \ldots \vee X^{p-1} \cdot \bar{f}_{p-1}.$$

where $f_i = f(X \leftarrow i)$.

By using Lemma 3.1 recursively, we can obtain a complement of an expression. It is possible to make an algorithm to generate at most $\prod_{i=1}^{n} p_i/(\max\{p_i\})$ products. However, experiments showed that a program simply based on Lemma 3.1 was not so fast for large practical problems. One reason for it is that most variables appear in a small number of products (i.e., a lot of "don't cares" in the array). Therefore, for the practical problems, the following algorithm has been developed.

Algorithm 3.1: Let $\mathcal{F}$ be a given expression. Use the following rules recursively.

Rule 1. If $\mathcal{F}$ is a constant:

If $\mathcal{F} = 1$, then $\bar{\mathcal{F}} = 0$

If $\mathcal{F} = 0$, then $\overline{\mathcal{F}} = 1$.

Rule 2. If $\mathcal{F}$ depends on only one-variable: i.e. if

$\mathcal{F} = X_1^{s_a} \vee X_1^{s_b} \vee \ldots \vee X_1^{s_z}$ then $\overline{\mathcal{F}} = X_1^{\bar{s}}$ where

$S = S_a \cup S_b \cup \ldots \cup S_z$.

Rule 3. If $\mathcal{F}$ consists of one product i.e., if

$\mathcal{F} = X_1^{s_1} \cdot X_2^{s_2} \ldots X_\ell^{s_\ell}$ then

$\overline{\mathcal{F}} = X_1^{\bar{s}_1} \vee X_1^{s_1} \cdot X_2^{\bar{s}_2} \vee \ldots \vee X_1^{s_1} \cdot X_2^{s_2} \ldots X_{\ell-1}^{s_{\ell-1}} \cdot X_\ell^{\bar{s}_\ell}$

Rule 4. If $\mathcal{F}$ has a common factor, i.e. if $\mathcal{F}$ can be written as

$\mathcal{F} = X_1^{s_1} X_2^{s_2} \ldots X_\ell^{s_\ell} \cdot \mathcal{G}$.

by renaming variables, where $\mathcal{G}$ does not contain variables $X_1, X_2, \ldots, X_\ell$, then

$\overline{\mathcal{F}} = X_1^{\bar{s}_1} \vee X_1^{s_1} X_2^{\bar{s}_2} \vee \ldots \vee X_1^{s_1} X_2^{s_2} \ldots X_\ell^{\bar{s}_\ell} \overline{\mathcal{G}}$.

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

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

then

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

where $\mathcal{G}_k$ $(k = 0, \ldots, p_i - 1)$ do not contain the variable X_i.

Rule 6. Otherwise, $\mathcal{F}$ can be written as

$\mathcal{F} = X_1^{s_1} \cdot X_2^{s_2} \cdot \ldots \cdot X_\ell^{s_\ell} \vee \mathcal{G}$

by renaming the variables. Then $\overline{\mathcal{F}}$ is given by

$\overline{\mathcal{F}} = X_1^{\bar{s}_1} \overline{\mathcal{G}}_1 \vee X_1^{s_1} X_2^{\bar{s}_2} \overline{\mathcal{G}}_2 \vee \ldots \vee X_1^{s_1} X_2^{s_2} \ldots X_{\ell-1}^{s_{\ell-1}} X_\ell^{\bar{s}_\ell} \overline{\mathcal{G}}_\ell$

where $\mathcal{G}_i$ is obtained from $(X_1^{s_1} \cdot X_2^{s_2} \cdot \ldots \cdot X_i^{\bar{s}_i}) \wedge \mathcal{G}$ by deleting null products.

Definition 3.1: A sum-of-products expression is disjoint if all products are mutually disjoint, i.e.,

$\mathcal{F} = \alpha_1 \vee \alpha_2 \vee \ldots \vee \alpha_s$

$\alpha_i \cdot \alpha_j \equiv 0 \ (i \neq j)$ or $s = 1$

Theorem 3.1: Algorithm 3.1 generates disjoint sum-of-products expression for f. The number of products in $\mathcal{F}$ is denoted by $t(\mathcal{F})$. By Theorem 3.1, it is clear that

$$t(\overline{\mathcal{F}}) \leq \prod_{i=1}^{n} p_i$$

where $\overline{\mathcal{F}}$ is obtained by Algorithm 3.1.

Theorem 3.2: Let $\mathcal{F}_n$ be a sum-of-products expression for

$$f_n: \ \bigtimes_{i=1}^{n} P_i \to B.$$

Let $\overline{\mathcal{F}}_n$ be an expression obtained by Algorithm 3.1, then

$$t(\overline{\mathcal{F}}_n) \leq \frac{1}{2} \prod_{i=1}^{n} p_i$$

(Proof). Proof will be done by the induction on n and restriction of f_n.

Rule 1. When $n = 0$: $t(\overline{\mathcal{F}}_0) \leq 1$ and the theorem holds for $n = 0$.

Rule 2. When $n = 1$: $t(\overline{\mathcal{F}}_1) \leq 1$ and the theorem hold for $n = 1$.

Rule 3. When $\mathcal{F}$ consists of one product: $X_1^{s_1} \cdot X_2^{s_2} \cdot \ldots \cdot X_\ell^{s_\ell}$. Then, $t(\overline{\mathcal{F}}_n) = \ell \leq n \leq 2^{n-1}$, and the theorem holds $(n > 2)$.
From here, suppose that the theorem holds for $n-1, n-2, \ldots, 1, 0$, and for the restriction of f_n.

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

$\mathcal{F}_n = X_1^{s_1} \cdot X_2^{s_2} \cdot \ldots \cdot X_\ell^{s_\ell} \cdot \mathcal{G}$

$t(\overline{\mathcal{F}}_n) = t(X_1^{\bar{s}_1} \vee X_1^{s_1} \cdot X_2^{\bar{s}_2} \vee \ldots \vee X_1^{s_1} \cdot X_2^{s_2} \cdot \ldots \cdot X_\ell^{\bar{s}_\ell}) + t(\overline{\mathcal{G}})$.

Since $\mathcal{G}$ does not contain the variables $X_1, X_2, \ldots, X_\ell$, it has at most $(n-\ell)$ variables. By the hypothesis of induction $t(\overline{\mathcal{G}}) \leq \frac{1}{2} \prod_{i=\ell+1}^{n} p_i$. Hence $t(\overline{\mathcal{F}}) \leq \ell + \frac{1}{2} \prod_{i=\ell-1}^{n} p_i \leq \frac{1}{2} \prod_{i=1}^{n} p_i$ and the theorem holds.

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

$\mathcal{F}_n = X_1^0 \cdot \mathcal{G}_0 \vee X_1^1 \cdot \mathcal{G}_1 \vee \ldots \vee X_1^{p_1-1} \cdot \mathcal{G}_{p_1-1}$

by renaming the variables:

$$t(\mathcal{F}_n) = \sum_{i=0}^{p_1-1} t(\mathcal{G}_i).$$

By the hypothesis of induction $t(\mathcal{G}_i) \leq \frac{1}{2} \prod_{j=2}^{n} p_j$ Hence $t(\mathcal{F}_n) \leq p_1 \times \frac{1}{2} \prod_{j=2}^{n} p_j = \frac{1}{2} \prod_{j=1}^{n} p_j$ and the theorem holds.

Rule 6. Otherwise, $\mathcal{F}$ can be written as

$$\mathcal{F} = X_1^{S_1} \cdot X_2^{S_2} \cdot \ldots \cdot X_\ell^{S_\ell} \vee \mathcal{G}$$

by renaming the variables. Because

$$\overline{\mathcal{F}} = X_1^{\bar{S}_1} \cdot \overline{\mathcal{G}}_1 \vee X_1^{S_1} \cdot X_2^{\bar{S}_2} \cdot \overline{\mathcal{G}}_2 \vee \ldots \vee X_1^{S_1} \cdot X_2^{S_2} \cdot \ldots \cdot X_\ell^{\bar{S}_\ell} \cdot \overline{\mathcal{G}}_\ell,$$
$$t(\overline{\mathcal{F}}) = \sum_{i=1}^{\ell} t(\overline{\mathcal{G}}_i).$$

$\mathcal{G}_i$ is obtained from $(X_1^{S_1} \cdot X_2^{S_2} \cdot \ldots \cdot X_\ell^{\bar{S}_\ell}) \wedge \mathcal{G}$ by deleting null products, and has a common factor $X_1^{S_1} \cdot X_2^{S_2} \cdot \ldots \cdot X_k^{\bar{S}_k}$. Let $|S_k| = a_k$, where $1 \leq a_k \leq p_k - 1$. In $\mathcal{G}_i$, X_k takes at most a_k distinct values; in other words, $\mathcal{G}_i$ represents a restriction of f_n:

$$\{0,1,\ldots,a_1-1\} \times \{0,1,\ldots,a_2-1\} \times \ldots \times \{0,1,\ldots,(p_i-a_i)-1\} \times \underset{k=i+1}{\overset{n}{\times}} P_k \to B$$

By the hypothesis of induction,

$$t(\mathcal{G}_i) \leq \frac{1}{2}\left(\prod_{k=1}^{i-1} a_k\right) \cdot (p_i - a_i) \times \left(\prod_{k=i+1}^{n} P_k\right).$$

Let $b_i = a_i/p_i$ and we have

$$t(\overline{\mathcal{G}}_i) \leq \frac{1}{2} \prod_{k=1}^{n} P_k \cdot \left(\prod_{k=1}^{i-1} b_k\right) \times (1-b_i)$$

Hence

$$t(\overline{\mathcal{F}}) \leq \frac{1}{2} \prod_{k=1}^{n} p_k \{(1-b_1) + b_1(1-b_2) + \ldots + b_1 b_2 \ldots b_{\ell-1}(1-b_\ell)\} \leq \frac{1}{2} \prod_{k=1}^{n} p_k$$

and the theorem holds.

We have exhausted all possible cases and proved the theorem by induction.

(Q.E.D)

In Rule 6 of Algorithm 3.1, the selection of the products and the ordering of the variables influence the efficiency of the algorithm. After doing a lot of experiments on practical circuits, we use the following heuristics.

Heuristic 3.1:

1. Which product to select: Choose one with the least number of literals (i.e., the number of literals such that $|S_i| \neq p_i$) If a tie occurs, choose one with maximal $\sum_{i=1}^{\ell} |S_i|$, where $X_1^{S_1} \cdot X_2^{S_2} \cdot \ldots \cdot X_\ell^{S_\ell}$.

2. The ordering of variables: Expand a product in the ascending order of $|S_i|/p_i$. If a tie occurs, expand first using the variable with the smallest α_i = (sum of number of 1's of each part in bit representation of $\mathcal{G}$)$/p_i$.

Example 3.1: Consider the function shown in Example 2.1:

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

First use Rule 6. $X_2^{\{1,2\}} \cdot X_3^1$ is a product with least number of literals. $\mathcal{F}$ is written as follows:

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

Because $\frac{|S_2|}{p_2} = \frac{2}{3}$ and $\frac{|S_3|}{p_3} = \frac{1}{4}$, expand it in the order of X_3 and X_2.

$\overline{\mathcal{F}} = (X_3^{\{0,2,3\}} \vee X_2^0 \cdot X_3^1) \cdot \overline{\mathcal{G}} = X_3^{\{0,2,3\}} \cdot \overline{\mathcal{G}}_1 \vee X_2^0 \cdot X_3^1 \cdot \overline{\mathcal{G}}_2$
where $\mathcal{G}_1 = X_3^{\{0,2,3\}} \cdot \mathcal{G} = X_1^0 \cdot X_2^1 \cdot X_3^3 \vee X_1^1 \cdot X_2^{\{0,2\}} \cdot X_3^2$ and
$\mathcal{G}_2 = X_2^0 \cdot X_3^1 \cdot \mathcal{G} = X_1^0 \cdot X_2^0 \cdot X_3^1$.

Next let's obtain $\overline{\mathcal{G}}_1$ and $\overline{\mathcal{G}}_2$ recursively. $\mathcal{G}_1$ can be written as $\mathcal{G}_1 = X_1^0 \cdot (X_2^1 \cdot X_3^3) \vee X_1^1 \cdot (X_2^{\{0,2\}} \cdot X_3^2)$. $\mathcal{G}_1$ can be decomposed with respect to X_1. By Rule 5, we have

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

$\mathcal{G}_2$ consists of one product and by Rule 3, $\overline{\mathcal{G}}_2 = X_1^1 \vee X_1^0 \cdot X_2^{\{1,2\}} \vee X_1^0 \cdot X_2^0 \cdot X_3^{\{0,2,3\}}$.

Hence

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

Note that $\overline{\mathcal{F}}$ contains 5 products.

(End of Example)

IV. Experimental Results:

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

1. Table 4.1 shows the comparison of Algorithm 2.1 (U_n # F), the disjoint sharp algorithm of MINI [5], and Algorithm 3.1. U_n denotes a universal cube. (U_n # $\mathcal{F}$) can be considered as an implementation of Algorithm 2.1. ⓗ is similar to #, but will generate disjoint sum-of-products expressions.

First, truth tables for 8-variable switching functions were randomly generated. Then, the functions were simplified by the distance-one-merge algorithm [5]. $t(\mathcal{F})$ denotes the number of products in a simplified expression. Lastly, the complement of the expressions were obtained. $t(\overline{\mathcal{F}})$ denotes the number of products in the complement $\overline{\mathcal{F}}$. Table 4.1 shows that the disjoint sharp (#) algorithm and Algorithm 3.1 are 10 ~ 30 times faster than algorithm 2.1 (D # F) and will generate simpler expressions. (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 variables. Also in this case, Disjoint sharp and Algorithm 3.1 were faster and produced simpler expressions. (See the entries for n = 4 and p = 4.)

2. Table 4.2 shows the comparison of Disjoint sharp (U_n(#)F) and Algorithm 3.1. Control circuits for microprocessors were used to compare the performance of two algorithms. For example see the entries for D2. D2 is an 8-input 7-output circuit. A characteristic function for a two-level PLA [2] is a mapping

   f: $P^8 \times M \to B$; P = {0,1}, M = {0,1,...,6}, B = {0,1}

   A simplified expression $\mathcal{F}$ for f has 43 products. Also, a characteristic function for a PLA with two-hit decoders [2] is a mapping.

   f: $P^4 \times M \to B$; P = {0,1,2,3}, M = {0,1,...,6}, B = {0,1}

   A simplified expression $\mathcal{F}$ for f has 42 products. Table 4.2 shows that Algorithm 3.1 generates simpler solutions (fewer products) than U_n(#)F. This is a desirable property because in MINI, (U_n (#) F) often produces an excessive number of products which prevents completing the initial phase of computing the complement for large problems.

3. Recently, R.K. Brayton et. al have independently developed a fast complementation algorithm [13]. It is for ordinary multiple-output switching functions only, and cannot treat multiple-valued variables. It is difficult to compare the performance of their algorithm with Algorithm 3.1 because of different data structures. In most cases, Algorithm 3.1 produced comparable solutions, but took longer time.

Table 4.1: Numbers of products in complement expressions and their computation time for Sharp,

		Algorithm 2.1		Disjoint Sharp		Algorithm 3.1	
		U_n # F		U_n(#)F			
	U	$t(\mathcal{F})$	CPU time (sec)	$t(\overline{\mathcal{F}})$	CPU time (sec)	$t(\overline{\mathcal{F}})$	CPU time (sec)	$t(\overline{\mathcal{F}})$
	32	23	38.814	171	2.098	67	1.167	51
p=2	64	39	57.631	203	3.186	82	4.493	68
n=8	96	57	65.232	163	4.191	87	2.925	79
	128	57	59.472	116	4.239	73	2.930	63
	32	21	15.020	243	0.701	47	0.735	41
p=4	64	33	22.494	207	1.436	54	0.883	56
n=4	96	45	43.494	204	2.485	56	1.647	54
	128	53	29.927	131	2.583	56	1.624	56

f: $P^n \to B$; P = {0,1,..., p − 1} $u = |f^{-1}(1)|$.

$\mathcal{F}$: sum-of-products expression for f; $\overline{\mathcal{F}}$: sum-of-products expression for $\bar{f}$.

$t(\mathcal{F})$: Number of products in $\mathcal{F}$; $t(\overline{\mathcal{F}})$: number of products in $\overline{\mathcal{F}}$.

Table 4.2: Numbers of products in complement expressions and their computation time for Disjoint sharp and Algorithm 3.1.

					Disjoint Sharp $U_n\#F$		Algorithm 3.1	
Circuit name	n	p	m	$t(\mathcal{F})$	CPU Time (sec)	$t(\overline{\mathcal{F}})$	CPU Time (sec)	$t(\overline{\mathcal{F}})$
D2	8	2	7	43	1.904	125	1.548	67
	4	4	7	42	1.166	106	1.569	95
R1	8	2	31	33	1.652	123	1.231	31
	4	4	31	32	1.016	63	.976	40
I1	16	2	17	110	10.162	333	5.429	202
	8	4	17	103	5.779	288	4.720	229
I4	32	2	20	222	64.954	3042	23.375	651
	16	4	20	204	33.841	1633	32.038	1148
I5	24	2	14	62	8.783	918	7.460	255
	12	4	14	61	5.287	1100	19.353	667
A2	10	2	8	89	5.372	228	6.417	188
	5	4	8	83	3.124	216	5.417	165

f: $P^n \times M \rightarrow B$, $P = \{0,1,\ldots, p-1\}$ $M = \{0,1,\ldots, m-1\}$

$\mathcal{F}$: sum-of-products expression for f; $\overline{\mathcal{F}}$: sum-of-products expression for $\bar{f}$.

$t(\mathcal{F})$: Number of products in $\mathcal{F}$; $t(\overline{\mathcal{F}})$: number of products in $\overline{\mathcal{F}}$.

V. Conclusions

1. The elementary method to obtain the complement of sum-of-product expression for f will generate all the prime implicants of $\bar{f}$, and is quite inefficient.

2. The average number of prime implicants for binary functions $\{0,1,..., p-1\}^n \rightarrow B$ is larger than $1/2\ p^n$ for large n.

3. Algorithm 3.1 will generate at most $1/2\ p^n$ products. It is $10 \sim 30$ times faster than the elementary one when $n = 8$ and $p = 2$.

4. Algorithm 4.1 produces fewer products than the disjoint sharp algorithm used by MINI for large practical problems.

Acknowledgement

The author is grateful to Dr. R.K. Brayton and Dr. S.J. Hong for their technical work. He also thanks Mrs. B. White for typing the manuscript.

References

[1] B. Dunham and R. Fridshal, "The problem of simplifying logical expressions", Journal of Symbolic 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. on Comput., Vol. C-30, No. 9, pp. 635-643, Sept. 1981.

[3] T. Sasao, "An application of multiple-valued logic to a design of masterslice gate array LSI", Proceedings of the 12th International Symposium on Multiple-Valued Logic, May 1982.

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

[5] S.J. Hong, R.G Cain and D.L. Ostapko, "MINI: A heuristic approach for logic minimization", IBM Res. Develop., Vol. 18, pp. 443-458, Sept. 1974.

[6] R.K. Brayton et al., "A comparison of logic minimization strategies using ESPRESSO: An APL Program package for partitioned logic minimization," Proc. 1982 International Symposium on Circuits and Systems, pp. 42-48, May 1982.

[7] R.J. Nelson, "Simplest normal truth function", J. Symbolic Logic, Vol. 20, pp. 105-108, June 1954.

[8] T. Sasao and H. Terada, "Multiple-valued logic and the design of programmable logic arrays with decoders", Proc. 9th International Symposium on Multiple-valued Logic, May 1979.

[9] S.J. Hong and D.L. Ostapko, "On complementation of Boolean functions", IEEE Trans. on Comput., Vol. C-21, p. 1072, 1972.

[10] D.L. Dietmeyer, *Logic Design of Digital Systems*, (second edition), Allyn and Bacon, Inc., Boston, 1978.

[11] S.Y.H. Su and P.T. Cheung, "Computer simplification of multi-valued switching functions", in *Computer Science and Multiple-Valued Logic*, North-Holland, pp. 189 ~ 220, 1977.

[12] T. Sasao et al., "A fast complementation algorithm for sum-of-products expressions", (in Japanese) Technical Group on Automata and Languages, IECE Japan, Jan. 22, 1981.

[13] R.K. Brayton et al., "Fast recursive Boolean function manipulation", Proc. 1982 International Symposium on Circuit and Systems, pp. 58-62, May 1982.

Appendix

As to the maximum number of the prime implicants of binary functions, the following are known.

Lemma A.1[8]: Let $\mu(n, p)$ be the maximum number of prime implicants of binary functions $P^n \to B$ where $P = \{0,1,..., p\}$. Define $t = 2^p-1$ and $m = \frac{n}{2^p-1}$. Then

$$(n!)/(m!)^t \leq \mu(n, p)$$

For example for $n=15$ and $p=4$, we have $\mu(p, n) \geq 15! \simeq 1.3\times 10^{12}$.

Theorem A.2[8]: For fixed p, there exists a positive constant K such that

$$K\cdot(t^n/n^{(1-t)/2}) \leq \mu(n, p)$$

where $t = 2^p-1$.

As to the average number of the prime implicants of binary function, we have the following:

Theorem A.3: Let f be a binary function

$$f: P_i^n \to B, \quad \text{where} \quad P_i = \{0,1,..., p_i-1\} \quad \text{and}$$

$B = \{0,1\}$, $u = |f^{-1}(1)|$ is a weight of f. The average number of the prime implicants of f with weight u is given by the following:

$$G_p(n,u) =$$

$$\frac{1}{F^{(u)}} \sum_{\mathbf{s}} C^{\mathbf{s}} \sum_{t=0}^{\eta(\mathbf{p},\mathbf{s})} (-1)^t \cdot \sum_{\mathbf{t}} \lambda(\mathbf{p},\mathbf{s},\mathbf{t})\cdot \binom{w-w(t,s)}{u-w(t,s)}$$

where $\mathbf{p} = (p_1,p_2,..., p_n)$, $\mathbf{s} = (s_1,s_2,..., s_n)$, $\mathbf{s} \leq \mathbf{p}$.

$$C^{\mathbf{s}} = \prod_{i=1}^{n} \binom{p_i}{s_i}, \quad F^{(u)} = \binom{w}{u}, \quad W = \prod_{i=1}^{n} p_i,$$

$\eta(\mathbf{p},\mathbf{s}) = \sum_{i=1}^{n} (p_i-s_i)$, $\mathbf{t} = (t_1,t_2,..., t_n)$ is a partition of t, and $t_i \leq p_i-s_i$,

$$\lambda(\mathbf{p},\mathbf{s},\mathbf{t}) = \prod_{i=1}^{n} \binom{p_i-s_i}{t_i}, \quad w(t,s) = \xi(\mathbf{s})\left(1 + \sum_{i=1}^{n} \frac{t_i}{s_i}\right)$$

and $\xi(\mathbf{s}) = \prod_{i=1}^{n} s_i$.

(proof.) Omitted.

Theorem A.4: Average number of the prime implicants of p-valued input binary functions is given by the following:

$$G_p(n) = \sum_{\mathbf{k}} C^{\mathbf{k}}\cdot 2^{-w(\mathbf{k})}\cdot \prod_{i=1}^{p-1} (1-2^{-w(\mathbf{k})/i})^{a_i},$$

where $\mathbf{k} = (k_1,k_2,..., k_p)$ is a partition of n.

$$w(\mathbf{k}) = \prod_{i=1}^{n} (i)^{k_i}, \quad C^{(\mathbf{k})} = (n!) \prod_{i=1}^{p} \frac{1}{k_i!}\binom{p}{i}^{k_i}, \text{ and } a_i = k_i(p-i).$$

(proof.) Omitted.

For example, for $n=15$ and $p=4$, we have $G_p(n) \cong 7\times 10^9$.

The algorithm in section III will generate at most $1/2\ p^n$ products. For example, for $n=15$ and $p=4$, $1/2\ p^n \simeq 5\times 10^8$. This shows that the algorithm generates at least 14 times less products than the elementary one. Table A.1 compares $G_p(n)$ and $1/2\ p^n$ for $p=2$ and $p=4$.

Table A.1 Comparison with $G_p(n)$ and $1/2\ p^n$

n	6	8	10	12	14
$G_2(n)$	24	118	585	2902	14225
$2^n/2$	32	128	512	2048	8192

n	3	4	5	6	7
$G_4(n)$	24	136	758	4095	21565
$4^n/2$	32	128	512	2048	8192