ON THE GENERATION OF PRIME IMPLICANTS

Bernd Reusch

TR 76-266

January 1976

Department of Computer Science
Cornell University
Ithaca, New York 14853

+This research was partially supported by NSF grant DCR 75-09433.

++On sabbatical leave from Universität Dortmund, Abteilung Informatik,
46 Dortmund 50, Postfach 500500, W. Germany.
ON THE GENERATION OF PRIME IMPLICANTS

Bernd Reusch

1. Introduction

The main purpose of this paper is to show that certain normal forms for Boolean functions as well as a number of the most basic methods for generation of prime implicants can very easily be derived using certain notations, basic theorems and proof techniques. We hope to make available material in a short and readable way, which is otherwise scattered over a large number of original papers and, in our opinion, not properly presented in textbooks on Switching Theory.

The next section is used to introduce our notion of Boolean functions, Boolean formulas and subfunctions. The latter is similar to the one used in [5]. Theorem 2.1 proves to be central for this section, a weaker form of it can also be found in [5]. From this theorem are easily derived as corollaries the Shannon-decompositions (which were well known in the 19th century!) as well as the minterm- and maxterm-normal forms for Boolean functions.

Section 3 is devoted to the discussion of various algorithms for generation prime implicants of Boolean functions. We did not concentrate on variations of basic methods, which were introduced

+This research was partially supported by NSF grant DCR 75-09433.

++On sabbatical leave from Universität Dortmund, Abteilung Informatik, 46 Dortmund 50, Postfach 500500, W. Germany.
for better implementations, as for instance in [11,12,17,18,44], nor did we want to deal with the well known topological terminology in [29,30,44], which is nicely summarized in [43]. Also multiple output prime implicants or multi-level prime implicants were not our deal.

The basic result of this section is Theorem 3.1 and its corollary. Brown in [5] mentions a theorem of Blake [4], which is based on the same idea. Our theorem is proven to be the basis for all product-methods [1,3,5,7,8,14,21,22,41] whereas its corollary, independently found in [26,27], is basic for all consensus-algorithms [13,15,17,24,25,26,40,41] and variations of them [2,10,11,15,16,17,18,19]. Some tree-methods and other decomposition-methods [6,7,20,26,32] can also be proven by Lemma 3.1.

The last section gives some hints on the complexity of the algorithms discussed. Worst-case considerations are shown to be related to the celebrated (P = NP)-question.

Although we have emphasized the tutorial nature of this paper, we want to point out that a number of things are new. First of all practically all proofs are new. Theorem 2.1 and Theorem 3.1, to our best knowledge, have not been reported in this general form elsewhere. The method of exclusion in Section 3, if is a fair generalization of an equally named method by Tison [40], the proof of which is reported to having been subject to some doubts [42].
Our list of references is by no means complete. To compute a complete list of the enormous number of papers on the generation of prime implicants is not an easy task and must be left to someone else.

2. Boolean functions and formulas.

In this section we want to introduce briefly the basic notions of Boolean functions and formulas as well as their inter-relationship. We will use what we think to be a rather elegant notation to state and prove some fundamental representation theorems.

**Def. 2.1:** A total mapping $f: \{0,1\}^n \rightarrow \{0,1\}$ is called a Boolean function. $n$ is called the argument length of $f$.

The set $\{0,1\}$ carries a simple algebraic structure, which is easily extended to the set of Boolean functions in the usual way.

<table>
<thead>
<tr>
<th></th>
<th>a+b</th>
<th>a\cdot b</th>
<th>$\bar{a}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

$\{0,1\}$ together with these operations is a Boolean lattice.

**Def. 2.2:** Let $f$ and $g$ be Boolean functions with the same argument length $n$. Then

$(f+g)(b) := f(b) + g(b)$

$(f \cdot g)(b) := f(b) \cdot g(b)$

$f(b) := f(b)\bar{b}$

for all $b \in \{0,1\}^n$. 
We have not introduced new symbols for the operations "+", "-" and "," on Boolean functions because of their close relation to those on \( \{0,1\} \) and since confusion is unlikely to arise.

Boolean functions of equal argument length together with the operations of Def. 2.2 form a Boolean lattice, the greatest and smallest elements of which are denoted by 1 and 0 respectively, where \( 1(b) = 1 \) and \( 0(b) = 0 \) for all \( b \in \{0,1\}^n \).

Boolean functions can be described by Boolean formulæ which are defined as follows:

**Def. 2.3:** Boolean formulæ are the strings generated by the following grammar:

- **Startsymbol:** \( S \)
- **Nonterminal:** \( B \)
- **Terminals:** \( 0, 1, x_1, \ldots, x_n, +, -, (, ) \)

**Productions:**

- \( S \rightarrow B \)
- \( B \rightarrow (B) \cdot (B) \)
- \( B \rightarrow (B) + (B) \)
- \( B \rightarrow (\overline{B}) \)
- \( B \rightarrow x_i \) for \( i = 1, \ldots, n \)

For instance \( (((x_1)(x_2)) + ((x_1)(x_2))) \) is a Boolean formula, which is convenient to be written as \( \overline{x_1}x_2 + x\overline{x_2} \) with the understanding that "," binds stronger than "+". The set of Boolean formulæ itself forms an algebra, the free \( \cdot, +, \rightarrow \)-algebra over \( \{x_1, \ldots, x_n\} \), since if \( A \) and \( B \) are Boolean formulæ so are \( (A)+(B) \),
(A) • (B) and (Ā).

This free algebra is homomorphically mapped onto the Boolean lattice of Boolean functions, if we assign "a meaning" to each formula in the following way.

Def. 2.4: The Boolean formula A is mapped onto the Boolean functions <A>, where <A> is defined recursively as follows:

Let A and B be Boolean formulas, then

<\(A + B\)> := <A> + <B>

<\(A \cdot B\)> := <A> • <B>

<\(\bar{A}\)> := <Ā>

<0> := |0|

<1> := 1

\(<x_j>(b_1,\ldots,b_n) := b_j\), for all \(b_j \in \{0,1\}\) and \(i = 1,\ldots,n\)

We say f is represented by A if \(f = <A>\). By definition, every Boolean formula represents some Boolean function. We will show, that every Boolean function is represented by some Boolean formula also.

It is convenient for our purposes to identify certain formulas.

Def. 2.5: Let \(B_1, B_2, B_3, B_4\) be arbitrary Boolean formulas.

1. \(B_1 \cdot (B_2) \equiv (B_2) \cdot (B_1)\)

\((B_1) + (B_2) \equiv (B_2) + (B_1)\)

\(((B_1) \cdot (B_2)) \cdot (B_3) \equiv (B_1) \cdot ((B_2) \cdot (B_3))\)
\[(B_1 + (B_2)) + (B_3) \equiv (B_1 + ((B_2) + (B_3)))\]

\[(B_1 + (B_1)) \equiv B_1\]

\[(B_1 \cdot (B_1)) \equiv B_1\]

\[B_1 \equiv B_1\]

2. If \(B_1 \equiv B_2\), then

\[(B_3) \cdot (B_1) \equiv (B_3) \cdot (B_2)\]

\[(B_3) + (B_1) \equiv (B_3) + (B_2)\]

\[B_1 \equiv B_2\]

3. If \(B_1 \equiv B_2\) and \(B_2 \equiv B_3\) according to 1. or 2.,

then \(B_1 \equiv B_3\).

It can be shown that \(\equiv\) is a congruence relation on the set of Boolean formulas. Hence operations can be defined on the equivalence classes, such that the new algebra is a homomorphic image of the algebra of Boolean formulas and the Boolean lattice of Boolean functions is still a homomorphic image of it. This simply means that we did not identify Boolean formulas by def. 2.5, which represent different Boolean functions.

In the sequel we call the equivalence classes defined above formulas for short. Associativity saves us brackets and we need not write out repetitions.

Let us now come back to the question whether every Boolean function is represented by some Boolean formula. We need the following definitions and simple facts.
Def. 2.6:  

a) \( \hat{x}_{i_1} \ldots \hat{x}_{i_k} \) is a product-term, if
\[ i_j \neq i_k \text{ for } j \neq k \text{ and } \hat{x}_{i_j} (x_{i_j}, \overline{x}_{i_j}) \]

b) \( \hat{x}_{i_1} + \ldots + \hat{x}_{i_k} \) is a sum-term, if
\[ i_j \neq i_k \text{ for } j \neq k \text{ and } \hat{x}_{i_j} (x_{i_j}, \overline{x}_{i_j}) \]

We add 0 and 1 to be special product-terms as well as special sum-terms.

c) \( \hat{x}_{i_1} \ldots \hat{x}_{i_k} \) is a minterm in the symbols 
\( x_1, \ldots, x_n \), if it is a product-term and 
\( \{i_1, \ldots, i_k\} = \{1, \ldots, n\} \)

d) \( \hat{x}_{i_1} + \ldots + \hat{x}_{i_k} \) is a maxterm in the symbols 
\( x_1, \ldots, x_n \), if it is a sum-term and \( \{i_1, \ldots, i_k\} = \{1, \ldots, n\} \)

e) Let \( P = \hat{x}_{i_1} \ldots \hat{x}_{i_k} \) and \( Q = \hat{x}_{j_1} \ldots \hat{x}_{j_l} \) be product terms. \( P \) is a shortening of \( Q \), if
\( \{i_1, \ldots, i_k\} \subseteq \{j_1, \ldots, j_l\} \) and \( \hat{x}_{i_m} = \hat{x}_{j_n} \) for all
\( i_m = j_n \).

\( P \) is a proper shortening of \( Q \), if it is a shortening and \( \{i_1, \ldots, i_k\} \neq \{j_1, \ldots, j_l\}. \)
Any product term $P$ is a shortening of 0 and 1 is a shortening of every product term.

Product-terms, sum-terms, minterms and maxterms so defined are special Boolean formulas and as such represent Boolean functions. We will now associate with product-terms another Boolean function.

**Def. 2.7:** Notation: $x_i^0 := \overline{x_i}$, $x_i^1 := x_i$.

Let $t = x_{i_1}^{e_1} \ldots x_{i_r}^{e_r}$ be a product-term and $b = (b_1, \ldots, b_n) \in \{0,1\}^n$. We define

$$<t=1>(b) := (b_1', \ldots, b_n')$$

where $b_j' := e_j$ for $j = 1, \ldots, r$ and $b_k' := b_k$ for $k \in \{i_1, \ldots, i_r\}$.

The functions $<t>$ and $<t=1>$ are different but closely related.

**Lemma 2.1:** Let $t$ be a product-term. Then

$$<t>(b) = 1 \text{ iff } <t=1>(b) = b$$

**Proof:** We have $t = x_{i_1}^{e_1} \ldots x_{i_r}^{e_r}$, $b = (b_1, \ldots, b_n)$ and

$$<t=1>(b) = (b_1', \ldots, b_n')$$. The following statements are pairwise equivalent:

a) $<t>(b) = 1$; b) $<x_{i_j}^{e_j}>(b) = 1$

for all $j = 1, \ldots, r$;

c) $b_{i_j} = e_j$ for $j = 1, \ldots, r$;

d) $b_{i_j} = b_j'$ for $j = 1, \ldots, r$;

e) $<t=1>(b) = b$. 
Before stating the main theorem of this section we have to introduce the notion of subfunctions.

**Def. 2.8:** Let \( f \) be a Boolean function and \( t \) a product-term.

The subfunction of \( f \) with respect to \( t \) is defined by

\[
f_t := f \circ <t=1>,
\]

where \( f \circ <t=1> \) means: apply \( <t=1> \) first and then \( f \) to the result.

**Theorem 2.1:** Let \( f \) be a Boolean function and let \( t_1, \ldots, t_k \)

be product-terms such that

\[
\sum_{i=1}^{k} t_i = 1.
\]

Then

\[
a) \quad f = \sum_{i=1}^{k} <t_i>f_{t_i}
\]

\[
b) \quad f = \prod_{i=1}^{k} (\overline{t_i} + f_{t_i})
\]

**Proof:** For any \( b \in \{0,1\}^n \) there is an \( i \) such that \( <t_i,b> = 1 \)

because of \( \sum_{i=1}^{k} t_i = 1 \). For this \( i \) we have \( <t_i=1,b> = b \)

by the previous lemma and therefore \( f_{t_i}(b) = f(<t_i=1>b)) = f(b) \). For every \( b \in \{0,1\}^n \) we define \( I_b := \{i|<t_i,b> = 1\} \)

and \( J_b := \{i|<t_i,b> = 0\} \).
a) \[ \left( \sum_{i=1}^{k} \langle t_i, f_{t_i} \rangle \right) (b) = \sum_{i \in I_b} \langle t_i, f_{t_i} \rangle (b) + \sum_{i \in J_b} \langle t_i, f_{t_i} \rangle (b) \]

\[ = \sum_{i \in I_b} f_{t_i} (b) = f(b) \]

b) \[ \prod_{i=1}^{k} \left( \langle \overline{t_i}, (b) + f_{t_i} (b) \right) = \]

\[ = \prod_{i \in I_b} (\langle \overline{t_i}, (b) + f_{t_i} (b) \rangle) \prod_{i \in J_b} (\langle \overline{t_i}, (b) + f_{t_i} (b) \rangle) \]

\[ = \prod_{i \in I_b} f_{t_i} (b) = f(b) . \]

Two very well known formulas can very easily be derived from the general one given in this theorem.

**Lemma 2.2:** (Shannon-decomposition)

Let \( f \) be a Boolean function and \( x \) one of its variables. Then

a) \( f = \langle x \rangle f_x + \langle \overline{x} \rangle f_{\overline{x}} \)

b) \( f = (\langle x \rangle + f_x) (\langle \overline{x} \rangle + f_{\overline{x}}) \)

**Proof:** This is a special case of Theorem 2.1 if we show \( \langle \overline{x} \rangle = \langle \overline{\overline{x}} \rangle \) and \( x + \overline{x} = 1 \). Both are trivial.
Lemma 2.3: Let \( f \) be a Boolean function in the variables 
\( x_1, \ldots, x_n \).

a) (Minterm-normalform)

\[
f = \sum_{(e_1, \ldots, e_n) \in \{0,1\}^n} \prod_{i=1}^n x_i^{e_i} f(e_1, \ldots, e_n)
\]

b) (Maxterm-normalform)

\[
f = \prod_{(e_1, \ldots, e_n) \in \{0,1\}^n} \left( \sum_{j=1}^n \overline{x_j}^{e_j} + x_n^{e_n} + f(e_1, \ldots, e_n) \right)
\]

Proof: In order to apply Theorem 2.1, we only have to show
the following three simple facts.

1) Let \( b = (e_1, \ldots, e_n) \). Then \( \sum_{i=1}^n \prod_{j=1}^n x_j^{e_j} = 1 \) (by) = b

and therefore \( \sum_{(e_1, \ldots, e_n) \in \{0,1\}^n} x_1^{e_1} \ldots x_n^{e_n} (b) = 1 \). Thus

\[
\sum_{(e_1, \ldots, e_n) \in \{0,1\}^n} x_1^{e_1} \ldots x_n^{e_n} (b) = 1.
\]

Since \( b \) was arbitrary we conclude

\[
\sum_{(e_1, \ldots, e_n) \in \{0,1\}^n} x_1^{e_1} \ldots x_n^{e_n} = 1.
\]

2) Let \( t = x_1^{e_1} \ldots x_n^{e_n} \). Then \( f_t(b) = \)

\[
f_t(b) = f_1 \prod_{j=1}^n \left( \sum_{i=1}^n \overline{x_j} \ overline{x}^{e_j} + x_n^{e_n} + f(e_1, \ldots, e_n) \right)
\]

for any \( b \in \{0,1\}^n \).
3) \[
\langle \mathbf{x}_1 \ldots \mathbf{x}_n \rangle = \langle \mathbf{e}_1 \rangle + \ldots + \langle \mathbf{e}_n \rangle = \langle \overline{\mathbf{x}}_1 \rangle + \ldots + \langle \overline{\mathbf{x}}_n \rangle
\]

Lemma 2.3 also solves a question we raised earlier.

**Corollary:** For every Boolean function there is a Boolean formula representing it.

The notations and methods developed in this section will be used to examine various algorithms to generate all the prime implicants of a given Boolean function \( f \).

3. Algorithms to generate prime implicants.

In this section various well-known algorithms to generate prime implicants for Boolean functions given in some sum-of-products form are presented and their correctness is proved in a rather uniform manner.

**Def. 3.1:** Let \( f \) be a Boolean function and \( P \) a product-term.

a) \( P \) is an implicant of \( f \), if \( \langle P \rangle (b) = 1 \) implies \( f(b) = 1 \) for all \( b \in \{0,1\}^n \). Notation: \( \langle P \rangle \rightarrow f \).

b) \( P \) is a prime implicant of \( f \), if
\[
\langle P \rangle \rightarrow \langle Q \rangle \rightarrow f \text{ implies } P = Q \text{ for all product terms } Q.
\]

\( P \) is a prime implicant of \( f \) simply if it is an implicant of \( f \) and no proper shortening of \( P \) has this property. The following will turn out to be a basic theorem for proving several algorithms for generating prime implicants correct. As a notation we use \( P \ast Q \) to denote the product term we get from...
product-terms \( P \) and \( Q \) by concatenation and then removing repetitions and contradictions. For example \( x_1 x_2 \bar{x}_2 x_3 = x_1 x_2 x_3 \) but \( x_1 \bar{x}_2 x_2 x_3 = 0 \).

**Theorem 3.1:** Let \( f = \bigcap_{i=1}^{\ell} f_i \) and \( P \) be a prime implicant of \( f \). Then \( P = P_1 \ldots P_\ell \) where \( P_i \) is a prime implicant of \( f_i \), \( i=1,\ldots,\ell \).

**Proof:** From \( <P>(b)=1 \) we conclude \( f(b)=1 \) and \( f_i(b)=1 \) for \( i=1,\ldots,\ell \). Therefore \( P \) is an implicant for all \( f_i \)'s.

But then there exist prime implicants \( P_i \) of \( f_i \) such that \( <P> <P_i> \), \( i=1,\ldots,\ell \) which induces \( <P> <P_1 \ldots P_\ell> \).

If we now can show \( <P_1 \ldots P_\ell> \) is a prime implicant, we can conclude \( P = P_1 \ldots P_\ell \) by the definition of prime implicants.

If \( <P_1 \ldots P_\ell>(b)=1 \), then \( <P_i>(b)=1 \) and hence \( f_i(b)=1 \) for \( i=1,\ldots,\ell \). But this immediately gives \( f(b)=1 \) and hence \( P_1 \ldots P_\ell \) is an implicant of \( f \).

A simple corollary of this theorem will also prove to be very helpful. It shows how prime implicants of a function \( f \) depend on the prime implicants of its subfunctions with respect to product-forms \( x \) and \( \bar{x} \), where \( x \) is a variable of the function. This result will be used later for straightforward inductions on the number of variables.

**Lemma 3.1:** Let \( f \) be a Boolean function, \( x \) one of its variables and \( P \) a prime implicant of \( f \). Then one of the following statements is valid:
1) $P = \overline{x}P_0$ and $P_0$ is a prime implicant of $f_x$.

2) $P = xP_1$ and $P_1$ is a prime implicant of $f_x$.

3) $P = P_0P_1$ and $P_0$ is a prime implicant of $f_x$ and $P_1$ a prime implicant of $f_x$.

Proof: We know $f = (\overline{x} + f_x)(x + f_x^{-})$. To apply Theorem 3.1, we only have to show that if $Q$ is a prime implicant of $(\overline{x} + f_x)$ then either $Q = \overline{x}$ or $Q$ is a prime implicant of $f_x$ and the related statement for $(x + f_x^{-})$. If $Q \neq \overline{x}$, then $Q \neq \overline{x}Q'$ and $Q \neq xQ'$ since otherwise $Q$ could not be a prime implicant. Therefore $Q(b) = (Qo<x=1>)(b)$ for all $b$. Assume $Q(b) = 1$. Then we know $(Qo<x=1>)(b) = 1$ also and $\overline{x}(x=1)(b)) = 0$. But since $Q$ is an implicant of $(\overline{x} + f_x)$ we conclude $f_x(x=1)(b)) = 1$. Therefore $(f_0<x=1>o<x=1>)(b) = (f_0<x=1>)(b) = f_x(b) = 1$ and $Q$ is an implicant of $f_x$. Clearly it must be a prime implicant since implicants of $f_x$ also are implicants of $(\overline{x} + f_x)$.

Both results of this section imply simple algorithms to generate all prime implicants. We will give a recursive "schema" for algorithms which can be derived from Theorem 3.1.

Algorithm 0:

(1) Represent $f$ as $f = \prod_{i=1}^{k} f_i$.

(2) $k = 1$

(3) Are the prime implicants of $f_k$ known?

no: Apply Algorithm 0 to $f_k$. 
(4) Remember the prime implicants of \( f_k \).

(5) \( l = k? \)
   yes: goto (7)

(6) \( l := l+1, \) goto (3)

(7) Construct all possible \( P_1 \cdots \* P_k \) where \( P_k \) is prime
    implicant of \( f_k \).

(8) Remove all product-terms for which there are proper shortenings
    in the list.

(9) Stop.

In fact a number of known algorithms more or less use this
schema. There are two points in Algorithm 0 where variations
come in.

a) How does one represent \( f \) as a product?
b) When are the prime implicants of \( f_k \) "known"?

We will now introduce a number of methods to generate
prime implicants, the first two of which will be instances of
what is said above. The starting point for all methods will be
a sum-of-products representation of the Boolean function under
consideration.

a) Method of double_product\[3,4,21,22,29]\.

If \( f \) is given in a product of sums form, then merely
"multiplying out" and removing terms if shortenings also appear
in the resulting sum-of-products form will result in the sum of
all prime implicants. This follows from Theorem 3.1 and the
fact that implicants of length one of functions \( f \neq 1 \) are prime
implicants. Converting a given sum-of-products form into a
product-of-sums form can be done by using the rule \(<a+bc> =
<(a+b)(a+c)>\) repeatedly. The methods name stems from a way to
do these operations, which is demonstrated by the following
example.

\[
\begin{align*}
\text{f is given by} & \quad x_1 \bar{x}_2 + x_2 \bar{x}_3 + \bar{x}_1 \bar{x}_2 x_4 \\
\text{replace "+" by "." and vice versa} & \quad (x_1 + \bar{x}_2)(x_2 + \bar{x}_3)(\bar{x}_1 + \bar{x}_2 + x_4) \\
\text{multiply out} & \quad \bar{x}_2 \bar{x}_3 + x_1 \bar{x}_2 x_4 + x_1 \bar{x}_3 x_4 \\
\text{replace "+" by "." and vice versa} & \quad (\bar{x}_2 + \bar{x}_3)(x_1 + x_2 + x_4)(x_1 + \bar{x}_3 + x_4) \\
\text{multiply out} & \quad \bar{x}_2 \bar{x}_3 + x_1 \bar{x}_2 + \bar{x}_2 x_4 + x_1 \bar{x}_3 + \bar{x}_3.
\end{align*}
\]

The same method sometimes is called the "method of double
dualization". The "method of double inversion" is practically
the same. If \( f \) is given by a sum-of-products form, then a sum-
of-products form of \( \bar{f} \) is computed by using deMorgan's law and
after that the same method is used to compute a sum-of-products
form of \( f = \bar{\bar{f}} \) from \( \bar{f} \). Again by Theorem 3.1 the result is the
sum of all prime implicants of \( f \).

There is a version which is less easy to detect namely the
one of "sharp products", which we will formulate next.

\text{Def. 3.2:} Let \( Q \) and \( P = x_{i_1}^{e_1} \ldots x_{i_k}^{e_k} \) be product terms. Then
\[
Q \# P := \{Q \star x_{i_j}^{e_j} | j = 1, \ldots, k\}
\]
Let $L = \{Q_1, \ldots, Q_m\}$ be a list of product terms. $Q_k$ is called **removable** in $L$, if there is a shortening of $Q_k$ contained in $L$.

**Def. 3.3:** The operation $1\#L$ on $L = \{Q_1, \ldots, Q_m\}$ is defined by the following algorithm

1. $L_0 := \{Q_1, \ldots, Q_m\}$, list of product terms.
2. $L_1 := 1\#Q_1$
3. $i := 1$
4. $L_{i+1} := \{R \# Q_{i+1} | R \in L_i\}$
5. Delete removable terms in $L_{i+1}$, call the result $L_{i+1}^*$
6. $i+1 = m^2$
   - no: $i := i+1$, goto (9)
7. $1\#L_0 := L_m$

Using $1\#L_0$ we are now able to formulate a short algorithm to generate prime implicants.

**Algorithm 1:**

1. $L_0 := \{Q_1, \ldots, Q_m\}$, list of product terms.
2. $L_E := 1\#(1\#L_0)$.

For easier writing we use $\langle L \rangle$ for the Boolean function $\langle R_1^+ \ldots + R_k \rangle$, where $L = \{R_1, \ldots, R_k\}$. 
Theorem 3.2: Let $L_E$ be the final list of an application of algorithm 1 to $L_0$. If $P$ is a prime implicant of $<L_0>$, then $P \in L_E$.

Proof: We only have to show, that $1\#L_0$ has the same effect on $L_0$ as "replace "+" by "." and vice versa, multiply out and delete removable terms". We do this by induction on the number of terms in $L_0$.

a) $L_0 = \{Q_1\} = \{x_{i1}^{e_1} ... x_{ik}^{e_k}\}$. In this case

$1\#L_0 = \{x_{i1}^{e_1}, ..., x_{ik}^{e_k}\}$ and everything is fine.

b) $L_0 = \{Q_1, ..., Q_m\}$ and $1\#\{Q_1, ..., Q_{m-1}\} = \{P_1, ..., P_k\}$.

Then

$1\#L_0 = \{P_1 \#Q_m, P_2 \#Q_m, ..., P_k \#Q_m\}$ and we conclude by the definition of $\#$.

Let us demonstrate the method by our previous example.

$L_0 = \{x_1 \bar{x}_2, x_2 \bar{x}_3, \bar{x}_1 x_2 x_4\}$

$L_1 = \{x_1, \bar{x}_2\}$

$L_2 = \{x_1 x_2, x_1 \bar{x}_3, \bar{x}_2 \bar{x}_3\}$

$L_3 = \{x_1 x_2 x_4, \bar{x}_2 \bar{x}_3, x_1 \bar{x}_3 x_4\}$

Now the computation of $1\#L_0$ is completed.
\[ L'_0 = \{ x_1x_2x_4, \overline{x}_2\overline{x}_3, x_1\overline{x}_3x_4 \} \]

\[ L'_1 = \{ x_1, x_2, x_4 \} \]

\[ L'_2 = \{ x_1\overline{x}_2, x_1\overline{x}_3, x_2\overline{x}_3, \overline{x}_2x_4, \overline{x}_3x_4 \} \]

\[ L'_3 = \{ x_1\overline{x}_2, x_1\overline{x}_3, x_2\overline{x}_3, \overline{x}_2x_4, \overline{x}_3x_4 \} \]

This is the result of \( 1\#L'_0 = 1\#(1\#L_0) \).

b) A tree_method \([5, 26, 27]\).

We now want to use Theorem 2.1 directly to generate prime implicants using the scheme of Algorithm 0.

a) We decompose \( f = \prod_{i=1}^{k} (t_i^+ + t_i^-) \) where \( \Sigma t_i^+ = 1 \).

b) The prime implicants of \( f_0^k \) are known if \( f_0^k \) is represented in sum of products form, where the products consist of 1, 0, \( x \) or \( \overline{x} \).

The simplest set of terms in a) is \( \{ x, \overline{x} \} \) and in b) more complicated stopping rules may be used. Let us again use our previous example to illustrate the method.

1) \( f = <x_1\overline{x}_2 + x_2\overline{x}_3 + \overline{x}_1x_2x_4> \)

2) Choose \( \{ x_1, \overline{x}_1 \} \)

3) \( f = <(x_1 + x_2\overline{x}_3 + \overline{x}_2x_4)(\overline{x}_1 + \overline{x}_2 + x_2\overline{x}_3)> \)

4) Choose \( \{ x_2, \overline{x}_2 \} \)
5) \[ f = \left( \overline{x_2} + x_1 + \overline{x_3} \right) (x_2 + x_1 + x_4) (\overline{x_2} + \overline{x_1} + x_3) (x_2 + \overline{x_1} + 1) \]

6) \[ f = \left( x_1 + \overline{x_2} x_4 + x_2 \overline{x_3} + \overline{x_3} x_4 \right) (\overline{x_2} + \overline{x_1} + \overline{x_3}) \]
\[ = \langle \overline{x_2} x_4 + x_2 \overline{x_3} + \overline{x_3} x_4 + x_1 \overline{x_2} + x_1 \overline{x_3} \rangle \]

The method is called tree method because of a slightly different version which uses the notation of a tree.

Instead of Theorem 2.1 and Theorem 3.1 we may use their corollary, namely Lemma 3.1. Here we have to compute prime implicants of subfunctions and then go back to the function.

The root of the tree is \( \hat{f} \). If \( f \) is a node of the tree and the prime implicants of \( \hat{f} \) are not known, then the tree is continued by

\[ \hat{f} \]
\[ \overline{x} \]
\[ x \]
\[ \overline{f_x} \]
\[ f_x \]

where \( x \) is a variable of \( \hat{f} \). The prime implicants of \( \hat{f} \) are known, if \( \hat{f} \) is given by a single term. To use our previous example:

\[ f = \langle x_1 \overline{x_2} + x_2 \overline{x_3} + \overline{x_1} x_2 x_4 \rangle \]
\[ = \langle x_1 A + \overline{x_1} B \rangle \text{ ~where~} \]
\[ A = \overline{x_2} + x_2 \overline{x_3} \]
\[ B = x_2 \overline{x_3} + \overline{x_2} x_4 \]
\(<A> = \langle \overline{x_2}, 1 \rangle + x_2 \cdot C\>
\(<B> = \langle \overline{x_2}, D \rangle + x_2 \cdot E\>

where

\[ C = \overline{x_3} \]
\[ D = x_4 \]
\[ E = \overline{x_3} \]

Hence we have the following tree:

Now we work back to generate the prime implicants of A, B

and finally f.

A: \(\{\overline{x_2}\} \cdot \{1\} \cup \{x_2\} \cdot \{\overline{x_3}\} \cup \{\overline{x_3}\} \cdot \{1\} = \{\overline{x_2}, x_2 \overline{x_3}, \overline{x_3}\}\)

after deletions: \(\{\overline{x_2}, \overline{x_3}\}\)

B: \(\{\overline{x_2}\} \cdot \{x_4\} \cup \{x_2\} \cdot \{\overline{x_3}\} \cup \{x_4\} \cdot \{\overline{x_3}\} = \{\overline{x_2} x_4, x_2 \overline{x_3}, \overline{x_3} x_4\}\)

f: \(\{\overline{x_1}\} \cdot \{\overline{x_2} x_4, x_2 \overline{x_3}, \overline{x_3} x_4\} \cup \{x_1\} \cdot \{\overline{x_2}, \overline{x_3}\} \cup \{\overline{x_2} x_4, x_2 \overline{x_3}, \overline{x_3} x_4\} \cdot \{\overline{x_2}, \overline{x_3}\}\)

\[= \{\overline{x_2} x_4, \overline{x_1} x_2 \overline{x_3}, \overline{x_1} \overline{x_3} x_4, x_1 \overline{x_2}, x_1 \overline{x_3}, x_2 x_4, x_2 \overline{x_3}, \overline{x_3} x_4\}\]

after deletions: \(\{x_1 \overline{x_2}, x_1 \overline{x_3}, x_2 x_4, x_2 \overline{x_3}, \overline{x_3} x_4\}\).
Improvements of the method are possible if we allow rules to modify the lists appearing at the nodes or if we use more complicated stopping rules.

c) Method of iterated consensus [4,15,24,31,41].

Although the previous methods are very simple to understand and prove, their execution may take too much time. The method of iterated consensus uses a more complicated basic operation to overcome this difficulty.

Def. 3.4: a) Let \( P = xP' \) and \( Q = \bar{x}Q' \) be product-terms.
\[
\hat{c}(P, Q) := P' * Q'
\]

b) Let \( P \) and \( Q \) be product-terms.
\[
c(P, Q) := \begin{cases} 
\hat{c}(P, Q), \text{if it exists for some } x \\
0, \text{otherwise}
\end{cases}
\]

c\( (P, Q) \) is called the consensus of \( P \) and \( Q \).

Lemma 3.3: \( <P+Q> = <P+Q+c(P, Q)> \).

Proof: We have to show \( <c(P, Q)> + <P+Q> \). Assume \( <c(P, Q)> (b) = 1 \).

Then there is a variable \( x \) such that \( P = xP' \) and \( Q = \bar{x}Q' \) and \( <P'> (b) = 1 \) as well as \( <Q'> (b) = 1 \). Now either \( <x = 1> (b) = b \) or \( <\bar{x} = 1> (b) = b \). In the first case we get \( <P> (b) = 1 \) in the second \( <Q> (b) = 1 \).
Let $L = \{Q_1, \ldots, Q_m\}$ be a list of product terms. The consensus $c(Q_i, Q_j)$ is called useful in $L$, if it is not removable in $L \cup \{c(Q_i, Q_j)\}$.

Algorithm 3:

1. $L_0 := \{Q_1, \ldots, Q_m\}$, list of product-terms.
2. $i := 0$
3. Is there a useful consensus $c(Q_k, Q_{k'})$ in $L_i$?
   - no: goto 7
4. $L_{i+1} := L_i \cup \{c(Q_k, Q_{k'})\}$
5. Delete removable terms in $L_{i+1}'$, call the result $L_{i+1}$.
6. $L_{i+1} = L_i$?
   - no: $i := i+1$, goto 3
7. $L_E := L_i$, stop.

The algorithm stops on every list $L_0$, simply because no new variables are introduced and there are only finitely many product terms for a given set of variables. The same holds true for other algorithms to follow. Hence we prove only "partial correctness", i.e. we prove that all prime implicants are generated under the assumption that the algorithm halts.

**Theorem 3.3:** Let $L_E = \{P_1, \ldots, P_n\}$ be the final list of an application of Algorithm 3 to the list $L_0 = \{Q_1, \ldots, Q_k\}$. If $P$ is a prime implicant of $f = <Q_1 + \ldots + Q_k>$, then $P \in L_E$. 
Proof: By Lemma 3.3 we know that \( f = \langle P_1 + \ldots + P_n \rangle \). Hence it is sufficient to show that every list which is invariant under the algorithm contains exactly the prime implicants of its associated function.

We prove by induction on the number of variables in \( L_E \).

a) \( \{0\}, \{1\}, \{x\}, \{\bar{x}\} \) are the only cases of final lists with one or less variables. They contain all prime implicants of the associated functions.

b) Let the theorem be true for lists containing up to \( n \) variables, let \( L_E \) be a list with \( n+1 \) variables and let \( P = xp' \) be a prime implicant. Now we sort \( L_E \) according to \( x \) and get

\[
L_E = \{xA_1', \ldots, xA_k', A_{k+1}', \ldots, A_{l}', \bar{x}c_1', \ldots, \bar{x}c_m'\}
\]

where \( A_{k+1}', \ldots, A_{l}' \) do not contain \( x \) or \( \bar{x} \).

\( P' \) is a prime implicant of \( \langle A_1 + \ldots + A_k \rangle \) by Lemma 3.1 and from \( P' \in \{A_1', \ldots, A_k'\} = L_1 \) we can conclude \( P \in L_1 \). Assume \( P' \notin L_1 \). Since \( L_1 \) contains only \( n \) variables this means the is a useful consensus \( c(A_i, A_j) \) in \( L_1 \). But then \( xc(A_i, A_j) \) is a useful consensus in \( L_1 \) which is a contradiction. The argument for the case \( P = \bar{x}P' \) runs similarly. If \( P = 1 \) then \( P \) itself is prime implicant of \( f \) and
\( f_x \) for any variable \( x \). Hence either the argument runs as above, or \( L_{\bar{E}} \supseteq \{x, \bar{x}\} \), whence there is a useful consensus again.

The method of iterated consensus gives some freedom in Step 3 which useful consensus to add. An actual implementation of the algorithm has to give some rules which one is to be selected and depending on the rules given, there will be generated many or less many intermediate terms. We will see some algorithms where rather "good" rules are given.

d) _Method of Quine-McCluskey \([15, 23]\)_

In this section we will improve the iterated consensus for the case where the start list contains no product-terms but minterms.

**Def. 3.5:** a) Let \( P \) and \( Q \) be product-terms such that \( P = xR \) and \( Q = \bar{x}R \) for some variable \( x \).

\[
\hat{d}(P, Q) := R
\]

b) Let \( P \) and \( Q \) be product-terms

\[
d(P, Q) := \begin{cases} 
\hat{d}(P, Q), & \text{if it exists for some } x \\
0, & \text{otherwise}
\end{cases}
\]

\( d(P, Q) \) is called simple consensus of \( P \) and \( Q \).

**Algorithm 4:**

1. \( L_0 := \{Q_1, \ldots, Q_m\} \), list of product-terms
(2) \( i := 0 \)

(3) \( L_{i+1} = L_i \cup \{d(R_j, R_k) | R_j, R_k \in L_i \} \)

(4) \( L_{i+1} = L_i \)?
   no: \( i := i+1 \), goto (3)

(5) Delete all removable terms in \( L_i \), call the result \( L_E \)

(6) Stop.

**Theorem 3.4:** Let \( L_E = \{P_1, \ldots, P_n\} \) be the final list of an application of Algorithm 4 to the list \( L_0 = \{Q_1, \ldots, Q_m\} \) consisting of minterms only. If \( P \) is a prime implicant of \( f = Q_1 + \cdots + Q_m \), then \( P \in L_E \).

**Proof:** We can translate the proof for the method of iterated consensus almost word by word for the final list \( L_i \) before cancelling removable terms using \( d(R_j, R_k) \) instead of \( c(Q_k, Q_l) \) there. We will give a second very short proof here.

Let there be \( n \) variables. It is sufficient to show that \( L_i \) contains all implicants of length \( n-i \).

a) \( i=0 \) is trivial, because we start with minterms.

b) Let it be true for \( i \leq k \) and let \( R \) be an implicant of the length \( n-(k+1) \) where \( x \) and \( \bar{x} \) are not contained in \( R \). From that we know \( xR \in L_k \) and \( \bar{x}R \in L_k \). But 
\[ d(xR, \bar{x}R) = R \] and therefore \( R \in L_{k+1} \).
Method of successive extraction [40]

In this method, consensus is taken with respect to one variable after the other, no repetition is necessary.

Algorithm 5:

1. \( L_1 = \{Q_1, \ldots, Q_m\} \), list of product-terms in \( n \) variables

2. \( i := 1 \)

3. Sort \( L_i \) to be

   \( L_i = \{\bar{x}_1A_1, \ldots, \bar{x}_kA_k, x_1B_1, \ldots, x_kB, C_1, \ldots, C_p\} \)

4. \( L_{i+1}' := L_i \cup \{A_j^{*}B_q^{*} \mid j = 1, \ldots, k, q = 1, \ldots, l\} \)

5. Delete removable terms in \( L_{i+1}' \), call the result \( L_{i+1} \).

6. \( i = n? \)

   no: \( i := i+1 \), go to 3.

7. \( L_E := L_i \), stop.

Theorem 3.5: Let \( L_E \) be the final list of an application of algorithm 5 to \( L_1 = \{Q_1, \ldots, Q_m\} \). If \( P \) is a prime implicant of \( f = Q_1 + \ldots + Q_m \), then \( P \in L_E \).

Proof: By induction on the number of variables in \( L_1 \).

a) If there is one variable, then \((x), (\bar{x})\) and \((x, \bar{x})\) are the only possibilities for \( L_1 \). In all three cases Algorithm 5 gives the correct result.
b) Let the theorem be true for up to n variables and
   \[ L_1 = \{ \tilde{x}_{n+1}A_1', \ldots, \tilde{x}_{n+1}A_k', x_{n+1}B_1', \ldots, x_{n+1}B_k, C_1', \ldots, C_p \} \]
   be a start list with n+1 variables. Now consider
   \[ L'_1 = \{ A_1', \ldots, A_k', C_1', \ldots, C_p \} \]
   \[ L''_1 = \{ B_1', \ldots, B_k', C_1', \ldots, C_p \} \]
   which correspond to subfunctions according to \( \tilde{x}_{n+1} \)
   and \( x_{n+1} \). Applying Algorithm 5 to \( L'_1 \) and \( L''_1 \) would result
   in \[ L'_E = \{ P_1', \ldots, P_a', P_1', \ldots, P_b \} \]
   \[ L''_E = \{ P_1'', \ldots, P_c', P_1', \ldots, P_b \} \]
   where \( (P_1', \ldots, P_a') \cap (P_1'', \ldots, P_c') = \emptyset \).
   By assumption \( L'_E \)
   and \( L''_E \) contain all prime implicants of the subfunctions.
   But \( L_1 \) will be transferred after n steps into
   \[ L_n = \{ \tilde{x}P_1', \ldots, \tilde{x}P_a', xP_1'', \ldots, xP_c', P_1', \ldots, P_b \} \]
   and by Lemma 3.1 finally \( L_{n+1} = L_E \) will contain all
   prime implicants of \( f = <Q_1 + \ldots + Q_m> \).

f) Method of exclusion [40].

In this method, every term is used only once to form consensus
with other terms. We use two different lists during the computation.

Algorithm 6:

1. \( A_0 = \emptyset, L_0 = \{ Q_1, \ldots, Q_m \} \), lists of product-terms.
2. \( i:=0 \)
3. Choose \( R_i \in L_i \)
4. \( A_{i+1} := A_i \cup \{ R_i \} \)
   \( L_{i+1} := L_i \cup \{ c(R_i, S) \mid 0 \leq S \in L_i \} \)
(5) Delete terms in \( L_i' + 1 \) which are removable in \( L_i' + 1 \cup A_i' + 1 \).
    Call the result \( L_i + 1 \).

Delete terms in \( A_i' + 1 \) which are removable in \( L_i + 1 \cup A_i' + 1 \).
    Call the result \( A_i + 1 \).

(6) \( L_{i+1} = \emptyset \)?
    no: \( i:= i+1 \), goto (3).

(7) \( L_{i+1} := A_i + 1 \), stop.

We want to show, that every prime implicant of \( f = \langle L_0 \rangle \) is contained in \( L_{i+1} \). The following lemma will be used.

**Lemma 3.4:** Let \( A_i \), \( L_i \) be the lists generated by Algorithm 6.
    If \( P \) is a prime implicant of \( \langle A_i \cup L_i \rangle \) and \( P \notin A_i \),
    then \( P \) is an implicant of \( \langle L_i \rangle \).

**Proof:** We prove by induction on \( i \), where the case \( i = 0 \) is trivial.
    If \( P \notin A_{i+1} \), then \( P \notin A_i \), since \( P \) is prime. By assumption
    \( P \) is an implicant of \( \langle L_i \rangle \), i.e. for every \( b \in \{0,1\}^n \) such
    that \( \langle P \rangle (b) = 1 \) there is a \( Q_b \in L_i \) such that \( \langle Q_b \rangle (b) = 1 \). We
    have to find a \( Q_b' \in L_{i+1} \) with the same property. If \( Q_b \in L_{i+1} \),
    then \( Q_b' = Q_b \). If \( Q_b \notin L_{i+1} \), then there is a shortest
    \( Q_b' \in L_{i+1} \cup A_{i+1} \) removing \( Q_b \). We also know \( \langle Q_b' \rangle (b) = 1 \).
    Assume \( Q_b' \notin L_{i+1} \). Since it was shortest, it cannot have
    been removed and therefore \( Q_b' \in A_{i+1} \subset A_i \cup L_i \). But then
    \( Q_b \in L_i \), because \( Q_b' \) removed it one step earlier. This is
    a contradiction and hence \( Q_b' \in L_{i+1} \).
Theorem 3.6: Let \( L_p \) be the final list of an application of Algorithm 6 to \( L_0 \). If \( P \) is a prime implicant of \( <L_0> \), then \( P \notin L_p \).

Proof: If \( L_1 = \emptyset \) and \( P \notin A_1 \) for a prime implicant \( P \) of \( <L_0> \), then by the previous lemma \( P \) is an implicant of \( <\emptyset> = 0 \).
This is a contradiction unless \( L_0 = \emptyset \).

g) Prime implicants of subfunctions [28].

In some applications Boolean functions depend on two types of variables, a set of "parameters" \( a_n', \ldots, a_k \) and a set of "variables" \( x_1', \ldots, x_n \). The problem is to find all prime implicants of functions in the variables \( x_1', \ldots, x_n \) which can be obtained by fixing \( a_1', \ldots, a_k \) to values from \( \{0, 1\} \). In the language used in this paper, this is to find all prime implicants of functions \( f_t \), where \( t \) is a minterm in \( a_1', \ldots, a_k \).

We will attack the more general problem of finding all prime implicants of \( f_t \) for given Boolean function \( f \) and arbitrary product term \( t \) in its variables.

Theorem 3.7: Let \( f \) be a Boolean function and \( t \) a product term in the variables \( x_1', \ldots, x_n \). If \( P \) is a prime implicant of \( f_t \), then there is a prime implicant of \( QP \) of \( f \), such that \( <Q> \circ <t=1> = 1 \), i.e. \( Q \) is a shortening of \( t \).

Proof: We will prove by induction on the length of \( t \). If \( t = x_i \) or \( t = \overline{x_i} \) we consult Lemma 3.1 and obtain: if \( P \) is a prime implicant of \( f_{x_i} \), then either is \( x_i P \)
a prime implicant of f, or there is a prime implicant
P_0 of f_{x_i}, such that P \cdot P_0 is a shortening of x_iP
Since P is prime, we conclude P \cdot P_0 = P = 1 \cdot P is
a prime implicant of f in this case. Thus our theorem
holds for product terms of length 1. Let it hold for
product terms up to length k and let t = t' \cdot x_i be a
product term of length k+1.

We observe that f_t = (f_{t'}, x_i). Hence by the
discussion above, if P is a prime implicant of f_{t'}, then
either x_iP or P is a prime implicant of f_{t'}.
By assumption
we then have a shortening Q' of t' such that Q'x_iP or
Q'P is a prime implicant of f. Hence Q = Q' or Q = x_iQ'
is the desired term.

Using this result, we obtain the prime implicants of f_t
in the following way.

Algorithm 7:
(1) L_0 := \{Q_1, \ldots, Q_m\}, list of prime implicants of <L_0>.
(2) L_1 := \{Q_j | j=1, \ldots, m, \; t \cdot Q_j \neq 0\}
(3) L_2 := \{Q'_j | Q_j \in L_1, \; Q_j = Q'_j Q_j, \; t \cdot Q'_j = t, \; t \cdot Q'_j = tQ'_j\}
(4) Delete removable terms in L_2, call the result L_E.
Let us demonstrate the method by an example.

a) \( L_0 = \{ \bar{x}_2 x_4, x_3 \bar{x}_4, x_2 x_3, x_1 x_2 \bar{x}_4, \bar{x}_1 x_3 \} \)

\[ t = \bar{x}_1 \bar{x}_2 \]

\( L_1 = \{ \bar{x}_2 x_4, x_3 \bar{x}_4, x_3 \} = \{ x_4 \cdot \bar{x}_2, x_3 \bar{x}_4 \cdot x_3 \cdot 1 \} \)

\( L_2 = \{ x_4, x_3 \bar{x}_4, x_3 \} \)

\( L_E = \{ x_4, x_3 \} \)

b) \( L_0 \) as in a)

\[ t = x_1 x_3 \]

\( L_1 = \{ \bar{x}_2 x_4, x_3 \bar{x}_4, x_2 x_3, x_1 x_2 \bar{x}_4 \} = \{ \bar{x}_2 x_4 \cdot 1, x_4 \cdot x_3, \bar{x}_2 \cdot x_3, x_2 \bar{x}_4 \cdot x_1 \} \)

\( L_2 = \{ \bar{x}_2 x_4, \bar{x}_4, x_2 \bar{x}_4 \} \)

\( L_E = \{ \bar{x}_2, \bar{x}_4 \} \)
4. Remarks on running times.

Some experiments have been conducted with programmed versions of algorithms presented in this paper at the University of Dortmund. Although no attempt was made for serious measurement, almost all experiments uniformly indicated the following ranking from fast to slow: successive extraction, method of exclusion, tree-method, double product, method of exclusion.

Another method to evaluate the performance of algorithms is consideration of worst cases. This leads us to a celebrated open question in theoretical computer science, namely the so-called (P = NP)—problem (for a presentation and discussion of this problem, see for instance: Aho, Hopcroft, Ullman, The Design and Analysis of Computer Algorithms, Addison Wesley, 1974). One version of it asks whether there is a "good" algorithm to decide a given sum-of-products form to denote the l-function, i.e. to be a tautology. An algorithm is "good", if it takes a total number of reasonably elementary steps which is polynomial in the number of literals in the original list. Although there is no proof yet, it is generally believed that there is no "good" algorithm for the tautology problem.

Now, if there would be a "good" algorithm for generating prime implicants, we could use it for the tautology problem: 1 is the one and only prime implicant of the l-function. On the other hand, if there is a "good" algorithm for the tautology-problem then there is at least a "good" algorithm to decide for any given product term whether it is an implicant. This is shown by our next lemma [34].
Lemma 4.1: Let $Q_1, \ldots, Q_m, t$ be product terms and $L_1, L_2$

as in Algorithm 7.

$t$ is an implicant of $<Q_1 + \ldots + Q_m>$ iff $<L_2> = 1$.

Proof: Assume $<L_2> = 1$ and $<t>(b) = 1$. From $<L_2> = 1$ we

conclude the existence of $Q'_j \in L_2$ such that $<Q'_j>(b) = 1$.

But now $<Q'_j>(b) = <Q'_j \bar{Q}_j>(b) = <Q'_j Q_j t>(b) = <Q'_j>(b)$. 

$<t>(b) = 1$ and therefore $<Q_1 + \ldots + Q_m>(b) = 1$. Now

assume $t$ to be an implicant of $<Q_1 + \ldots + Q_m>$ and

$<L_2> \neq 1$. Then there is a $b$ such that $<L_2>(b) = 0$,

which means $<Q'_j>(b) = 0$ for all $Q'_j \notin L_2$.

This property is conserved if we change $b$ to $b' = <t=1>(b)$

by the construction of $L_2$. But now $<t>(b') = 1$ and hence

by assumption $<Q_1 + \ldots + Q_m>(b') = 1$. On the other

hand $<Q'_j>(b') = 0$ for $Q'_j \notin L_1$ and $<Q'_j>(b') = <Q'_j \bar{Q}_j>(b') = 0$

for $Q'_j \notin L_1$ which is a contradiction.

The discussion above indicates, that most probably there

is no "good" algorithm for generating all the prime implicants

of a given Boolean function. It does not mean at all that it

is easy to prove for a particular algorithm that it is not a

"good" one. However Z. Galil has shown (Z. Galil, The Complexity

of Resolution Procedures for Theorem Proving in the Propositional

Calculus, Cornell University, Dept. of Computer Science, TR 75-239,

1975) that any algorithm using consensus as basic operation needs

exponentially many steps, if a variable which is once eliminated
in the computation of a particular prime implicant, is not introduced again in that computation.
References


