Recognition of Fanout-free Functions*

Tsung-Lin Lee   Chun-Yao Wang

Department of Computer Science,
National Tsing Hua University,
HsinChu, Taiwan 300 R.O.C.
johnny@nthucad.cs.nthu.edu.tw; wcyao@cs.nthu.edu.tw

Abstract

Factoring is a logic minimization technique to represent a Boolean function in an equivalent function with minimum literals. When realizing the circuit, a function represented in a more compact form has smaller area. Some Boolean functions even have equivalent forms where each variable appears exactly once, which are known as fanout-free functions. *John P. Hayes* [4] had devised an algorithm to determine if a function can be fanout-free and construct the circuit if fanout-free realization exists. In this paper, we propose a property and an efficient technique to accelerate this algorithm. With our improvements, execution time of this algorithm is more competitive with the state-of-the-art method [3].

1 Introduction

There are many techniques of logic minimization, such as decomposition, substitution, factoring, and etc. By applying factoring to a Boolean function, a logically equivalent function with the minimum number of literals can be derived. Some Boolean functions even have factored forms with each variable appears exactly once, which are called fanout-free functions.

The corresponding circuit of a fanout-free function is a tree-like structure, which is favored in the application of automatic test-pattern generation (ATPG) due to its simplicity [5]. Tree-like circuits are also preferred in technology mapping algorithms, such as DAGON [6]. It also benefits the evaluation of the testability of a design under test (DUT).

Many methods of recognizing fanout-free functions are proposed in recent works [8][2][3]. *John P. Hayes* first introduced an algorithm to recognize fanout-free functions [4]. However, this algorithm is based on examination of the adjacency relation of input variables, which are originally achieved by performing equivalence checking of cofactors of each variable. As a result, it is a computation intensive procedure as described in more detail in Section III.

We propose a property *disappearance* on adjacency relation checking and find it can be applied to significantly accelerate *Hayes'* algorithm. With this property, we can examine the adjacency relation without performing equivalence checking. These will be described in Section IV.

With our improvements on *Hayes'* algorithm, we make it more competitive with the state-of-the-art method proposed in [3]. Experimental results will be shown in Section V, and Section VI concludes this work.

*This work was supported in part by the National Science Council of R.O.C. under Grant NSC94-2220-E-007-012.

2 Preliminaries

In this section, we will describe notations and fundamental concepts used in this paper.

Cofactor

Given an n-input Boolean function $f(X)$, where $X = \{x_1, x_2, \cdots, x_n\}$. Cofactor of f with respect to $x_i = c$ is denoted as $f(x_i = c)$ and represents the function that variable x_i was assigned to a constant c, c=0 or 1. For example, if $f(X) = x_1 + x_2$, then $f(x_1 = 0) = x_2$ and $f(x_1 = 1) = 1$.

Unate Functions

Given an n-input Boolean function $f(X)$, where $X = \{x_1, x_2, \cdots, x_n\}$. f is positive unate in variable x_i if $f(x_i = 1) = 1$ whenever $f(x_i = 0) = 1$. In other words, x_i appears in positive phase. Similarly, f is negative unate in variable x_i if $f(x_i = 0) = 1$ whenever $f(x_i = 1) = 1$. A function is unate if all its variables are unate [1]. For example, the function $f = \overline{x}_1x_2 + \overline{x}_1x_3$ is a unate function since it is positive unate in x_2 and x_3, and negative unate in x_1, i.e., all three variables are unate.

Without loss of generality, we only discuss positive Boolean functions in this paper. A positive Boolean function is a Boolean function in which all its variables are positive unate. If a function has a negative unate variable, we can substitute a positive unate variable for this negative unate variable. For example, the function $g(x_1, x_2) = x_1 + \overline{x}_2$ can be viewed as $f(x_1, \overline{x}_2) = x_1 + x_2$.

Fanout-free Functions

A fanout-free function is also known as a read-once function which has a factored form that each variable appears exactly once [3]. For example, the function $f = x_1x_2 + x_1x_3$ is a fanout-free function since it has a factored form $f = x_1(x_2 + x_3)$ in which each variable appears exactly once.

It is obviously that fanout-free functions must be unate functions. If a function has some non-unate variable, this variable will appear in two phases. Thus, this variable has to be fanouted to realize the function. For example, variable x_1 has to be fanouted in function $f = x_1x_2 + \overline{x}_1x_3$.

Simple Disjunctive Decomposition

Simple disjunctive decomposition extracts a single-output sub-function whose input variable set is disjunctive from the other input variables [7]. Thus, a function $f(X)$ has a simple disjunctive decomposition form $g(h(Y), Z)$ where $X = Y \bigcup Z$ and $Y \bigcap Z = \phi$. Figure 1 illustrates $f(X)$=$g(h(Y), Z)$.

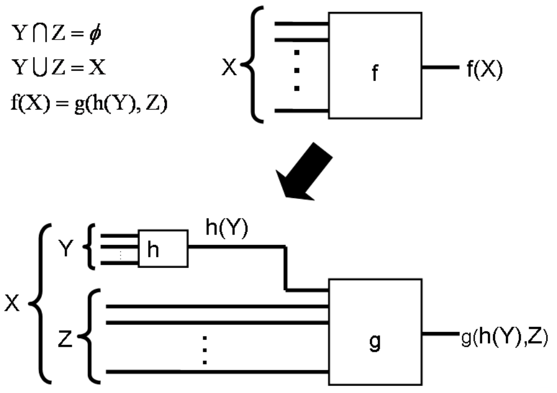

Figure 1: Simple Disjunctive Decomposition

Adjacency Relation

This relation is proposed by John P. Hayes [4]. Its definition is different from that used in graph theory. Given a function $f(X)$, where $X = \{x_1, x_2, \cdots, x_n\}$. x_i and x_j are adjacent if $f(x_i = a) = f(x_j = a)$ for some a, a=0 or 1. It will be denoted by $=_a$, e.g., $x_i =_a x_j$. Adjacency relation obviously is a reflexive and symmetric relation. It also has been proven that it is a transitive relation [4]. Thus, adjacency relation is an equivalence relation. Take f=$x_1x_2x_3x_4$+$x_1x_2x_3x_5$+x_4x_6+x_5x_6 for example, by definition, we compare cofactors of f w.r.t. each variable to see if any $f(x_i = a) = f(x_j = a)$ exists. Since $f(x_1 = 0) = f(x_2 = 0) = f(x_3 = 0) = x_4x_6 + x_5x_6$, $x_1 =_0 x_2 =_0 x_3$. Also, since $f(x_4 = 1) = f(x_5 = 1) = x_1x_2x_3 + x_6$, $x_4 =_1 x_5$. Thus, we obtain three adjacent classes $\{x_1, x_2, x_3\}$, $\{x_4, x_5\}$, and $\{x_6\}$.

3 Previous Works

Several approaches have been proposed to recognizing fanout-free functions [4][8][2][3]. Our approach is an improvement on John P. Hayes' fanout-free realization algorithm [4]. Here we only review Hayes' algorithm. In the following sections, we will call it JPH's procedure.

Hayes proposed a theorem that "Let the variables of $f(X)$ be partitioned into blocks $X_1, X_2, \cdots, X_m$ by the adjacency relation. There exists a set of m elementary functions[1] $\varphi_1(X_1)$,$\varphi_2(X_2)$,$\cdots$,$\varphi_m(X_m)$, and an m-variable function F such that $f(X) = F(\varphi_1(X_1), \varphi_2(X_2), \cdots, \varphi_m(X_m))$". We explain its concepts here.

Since adjacency relation is an equivalence relation, it can partition a set into disjoint subsets. By examining the adjacency relation among variables of a function, we can get disjoint subsets of variables. Each subset forms a simple disjunctive decomposition. A corresponding elementary function is composed of a subset of variables. If a subset is obtained by assigning 0 to its variables, then its corresponding elementary function is AND function. Similarly, the elementary function is OR function if variables are adjacent by assigning 1 to themselves.[2] As shown in Figure 2, the functions $\varphi_1(X_1)$, $\varphi_2(X_2)$, $\cdots$, and $\varphi_m(X_m)$ become fanout-free.

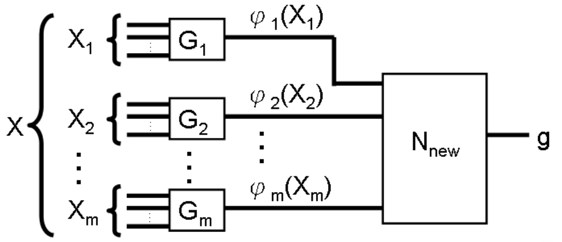

Figure 2: Disjunctive Decomp. with Adjacency Relation

By viewing the output of the elementary function as an input variable to the remaining circuit as shown in Figure 2, the remaining circuit forms a new function, N_{new}. This new function (corresponding decomposition function) can be derived from the original function via truth table construction.

This theorem forms the underpinning of JPH's procedure. It will realize the fanout-free function level by level. If none of variables are adjacent, then this function does not have any simple disjunctive decomposition. In other words, we cannot extract any elementary function from this function. The following are the steps of the JPH's procedure.

JPH's procedure: To find a fanout-free realization of $f(X)$ if it exists.

[1]An elementary function is either a sum (OR) function or a product (AND) function.

[2]This conclusion is based on the assumption that we discuss positive Boolean functions.

Step 1 Let $f_i(X_i) = f(X)$.

Step 2 Examine the adjacency relation of X_i. If X_i are mutually adjacent, go to Step 4. If none of X_i are adjacent, go to Step 5. Otherwise go to Step 3.

Step 3 We have adjacent classes $\{X_{i_j}\}$ and associated elementary functions $\{\varphi_{i_j}(X_{i_j})\}$. Construct the corresponding decomposition function $f_{i+1}(\varphi_{i_1}(X_{i_1}), \varphi_{i_2}(X_{i_2}), \cdots, \varphi_{i_r}(X_{i_r})) = f_i(X_i)$. Replace f_i by f_{i+1}. Go to Step 2.

Step 4 $f_i(X_i)$ is the fanout-free realization of $f(X)$.

Step 5 $f(X)$ does not have a fanout-free realization.

The corresponding flow chart of JPH's procedure is shown in Figure 3. In Step 2 of the procedure, adjacent classes X_{i_j} are determined, where $\{X_{i_j}\} = X_i$. Each adjacent class will be associated with a corresponding elementary function $\varphi_{i_j}(X_{i_j})$ in Step 3.

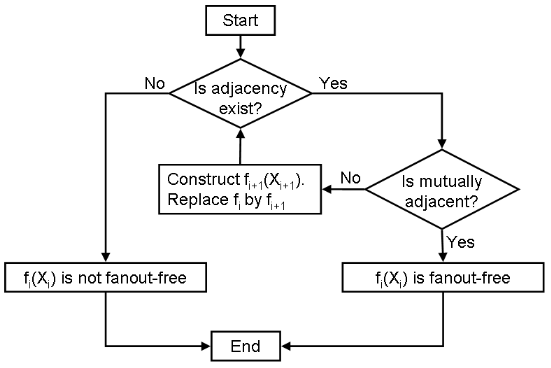
Start
Is adjacency exist?
No
Yes
Construct $f_{i+1}(X_{i+1})$. Replace f_i by f_{i+1}
No
Is mutually adjacent?
Yes
$f_i(X_i)$ is not fanout-free
$f_i(X_i)$ is fanout-free
End

Figure 3: Flow Chart of JPH's Procedure

Here we demonstrate the first iteration of JPH's procedure with $f{=}x_1x_2x_3x_4{+}x_1x_2x_3x_5{+}x_4x_6{+}x_5x_6$. At the beginning, let $f_1{=}f$. It was examined $x_1 =_0 x_2 =_0 x_3$ and $x_4 =_1 x_5$ as mentioned in Section II. The associated elementary functions are $\varphi_{1_1} = x_1x_2x_3$, $\varphi_{1_2} = x_4 + x_5$, and $\varphi_{1_3} = x_6$. A truth table for f_2, as shown in Table 1, can be determined from $\{\varphi_{1_j}\}$ and f_1. Thus, we obtain the corresponding decomposition function

$$\begin{aligned} f_2 &= \overline{\varphi}_{1_1}\varphi_{1_2}\varphi_{1_3} + \varphi_{1_1}\varphi_{1_2}\overline{\varphi}_{1_1} + \varphi_{1_1}\varphi_{1_2}\varphi_{1_3} \\ &= \varphi_{1_1}\varphi_{1_2} + \varphi_{1_2}\varphi_{1_3} \end{aligned}$$

from Table 1. After replacing f_1 by f_2, we continue dealing with f_2. The circuit after the first iteration is shown in Figure 4.

4 Our Approach

We propose a *disappearance* property which could be used to accelerate the examination of adjacency relation. We also find an efficient way to constructing the corresponding decomposition function, N_{new}, without using truth table.

Table 1: Determination of f_2

φ_{1_1}	φ_{1_2}	φ_{1_3}	$f_2 = f_1$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

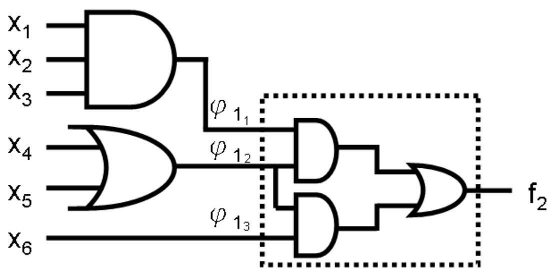

Figure 4: The 1st iteration of JPH's Procedure on $f = x_1x_2x_3x_4 + x_1x_2x_3x_5 + x_4x_6 + x_5x_6$

4.1 Disappearance Property

Given a function $f(X)$, where $X = \{x_1, x_2, \cdots, x_n\}$. $x_i =_a x_j$ implies the cofactor $f(x_i = a)$ is independent of x_j and the cofactor $f(x_j = a)$ is independent of x_i. Intuitively speaking, x_j disappears in the cofactor $f(x_i = a)$, and vice versa. Thus we conclude that "*adjacency* $\rightarrow$ *disappearance*".

Lemma 1. Let x_i and x_j be two distinct variables of $f(X)$. If $x_i{=_a}x_j$, then x_j disappears in the function $f(x_i = a)$ and x_i disappears in the function $f(x_j = a)$.
Proof. $x_i{=_a}x_j$ implies $f(x_i = a){=}f(x_j = a)$. If x_j appears in $f(x_i = a)$, without loss of generality, we can assume $f(x_i = a) = x_jA + Z$, where A and Z are independent of x_j. $f(x_i = a) = x_jA + Z \neq f(x_j = a)$. It leads to a conflict. Thus, there should be no terms containing x_j since x_j is assigned to a constant a. Therefore x_j disappears in the function $f(x_i = a)$. The same rule applies to "x_i disappears in the function $f(x_j = a)$". Q.E.D.

We can use the contrapositive of "*adjacency* $\rightarrow$ *disappearance*", which is "*appearance* $\rightarrow$ *non* $-$ *adjacency*", to accelerate JPH's procedure. If x_j appears in function $f(x_i = a)$, then $x_i \neq_a x_j$. This property can be used as a filter when we are examining the adjacency relation. We drop the variables which cannot be adjacent, and only check variables which are possibly adjacent.

However, it is still time-consuming to examine whether two variables are adjacent. This is because checking whether $f(x_i = a)$ is equal to $f(x_j = a)$ involves equivalence checking of two cofactors. Thus, we would like to know if "*disappearance* → *adjacency*" holds? Formally speaking, we want to prove that "if x_j disappears in the function $f(x_i = a)$ and x_i disappears in the function $f(x_j = a)$, then $x_i =_a x_j$". This is stated in Lemma 2.

Lemma 2. Let x_i and x_j be two distinct variables of $f(X)$. If x_j disappears in the function $f(x_i = a)$ and x_i disappears in the function $f(x_j = a)$, then $x_i =_a x_j$.

Proof. Without loss of generality, we assume $f = x_i x_j A + x_i B + x_j C + (x_i + x_j)D + E$, where A, B, C, D, and E are independent of x_i and x_j.

Case 1: (for $x_i =_0 x_j$)

$f(x_i = 0) = x_j C + x_j D + E$. Since x_j disappears in $f(x_i = 0)$ by the given condition, C and D must be 0. $f(x_j = 0) = x_i B + x_i D + E$. Since x_i disappears in $f(x_j = 0)$ by the given condition, B and D must be 0. Combining these two implications, $B = C = D = 0$. Thus, $f = x_i x_j A + E$. $f(x_i = 0) = E = f(x_j = 0)$, and $x_i =_0 x_j$.

Case 2: (for $x_i =_1 x_j$)

$f(x_i = 1) = x_j A + B + x_j C + D + E$. Since x_j disappears in $f(x_i = 1)$ by the given condition, A and C must be 0. $f(x_j = 1) = x_i A + x_i B + C + D + E$. Since x_i disappears in $f(x_j = 1)$ by the given condition, A and B must be 0. Combining these two implications, $A = B = C = 0$. Thus, $f = (x_i + x_j)D + E$. $f(x_i = 1) = D + E = f(x_j = 1)$, and $x_i =_1 x_j$.

By Case 1 and 2, we can conclude that "*disappearance* → *adjacency*". Q.E.D.

Theorem 3. Let x_i and x_j be two distinct variables of $f(X)$. $x_i =_a x_j$ if and only if x_j disappears in the function $f(x_i = a)$ and x_i disappears in the function $f(x_j = a)$.

Proof. This theorem can be restated as "*adjacency* ⇔ *disappearance*". The sufficient condition has been proved in Lemma 2, and the necessary condition has been proved in Lemma 1. Q.E.D.

By Theorem 3, we have the *disappearance* property, which is "*adjacency*⇔*disappearance*". Since adjacency relation is an equivalence relation, it has a property of transitivity. Thus, $x_i =_a x_j$ and $x_j =_a x_k$ implies $x_i =_a x_k$. Although we simply prove the *disappearance* property with two variables, it can be easily extended to multiple variables.

$$\begin{aligned}
f(x_1 = 0) &= x_4x_6 + x_5x_6 \\
f(x_2 = 0) &= x_4x_6 + x_5x_6 \\
f(x_3 = 0) &= x_4x_6 + x_5x_6 \\
f(x_4 = 0) &= x_1x_2x_3x_5 + x_5x_6 \\
f(x_5 = 0) &= x_1x_2x_3x_4 + x_4x_6 \\
f(x_6 = 0) &= x_1x_2x_3x_4 + x_1x_2x_3x_5
\end{aligned}$$

(a)

$$\begin{array}{c} \\ f(x_1=0) \\ f(x_2=0) \\ f(x_3=0) \\ f(x_4=0) \\ f(x_5=0) \\ f(x_6=0) \end{array}
\begin{array}{c} \begin{array}{cccccc} x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \end{array} \\
\left(\begin{array}{cccccc}
0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 \\
1 & 1 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 & 0 & 1 \\
1 & 1 & 1 & 1 & 1 & 0
\end{array}\right)\end{array}$$

(b)

Figure 5: (a) Equations of Neg. Cofactors; (b) Matrix; of $f = x_1x_2x_3x_4 + x_1x_2x_3x_5 + x_4x_6 + x_5x_6$

4.2 Improvement on Examining the Adjacency Relation

With the *disappearance* property, we can examine the adjacency relation without conducting equivalence checking. If x_i disappears in $f(x_j = a)$, then we check whether x_j disappears in $f(x_i = a)$. If it does, we can infer $x_i =_a x_j$.

We demonstrate how to use *disappearance* property to accelerate JPH's procedure with the same 6-variable function $f = x_1x_2x_3x_4 + x_1x_2x_3x_5 + x_4x_6 + x_5x_6$. First, we list negative cofactors of f with respect to each variable as shown in Figure 5(a).

Second, as shown in Figure 5(b), we build a 6×6 matrix with rows and columns labeled by $f(x_i = 0)$ and x_j respectively. The matrix is filled with element 1 or 0 in position $(f(x_i = 0), x_j)$ according to whether x_j appears in $f(x_i = 0)$. If x_j appears in $f(x_i = 0)$, 1 is filled into $(f(x_i = 0), x_j)$; otherwise, 0 is filled into $(f(x_i = 0), x_j)$. This matrix has 0s on the diagonal because x_i always disappears in $f(x_i = 0)$.

Then we check this matrix row by row. Take the first row for example, x_2 and x_3 are suspected to be adjacent to x_1 because $(f(x_1 = 0), x_2)$ and $(f(x_1 = 0), x_3)$ are 0s, i.e., they disappear in $f(x_1 = 0)$. So we check whether these corresponding positions $(f(x_2 = 0), x_1)$ and $(f(x_3 = 0), x_1)$ are 0s or not.

If both of them are 0, we can infer that $x_1 =_0 x_2 =_0 x_3$. Checking $(f(x_2 = 0), x_3)$ and $(f(x_3 = 0), x_2)$ is redundant since $x_1 =_0 x_2$ and $x_1 =_0 x_3$ implies $x_2 =_0 x_3$. They definitely will be 0s. Furthermore, checking the other elements of row $f(x_2 = 0)$ is unnecessary, so is row $f(x_3 = 0)$. The reason is if $x_i \neq_a x_1$ and $x_1 =_a x_2 =_a x_3$, then $x_i \neq_a x_2$ and $x_i \neq_a x_3$.

Next, we list positive cofactors of f with respect to each variable first and build its corresponding matrix. Following the same process, we find that $x_4 =_1 x_5$. Now we have three adjacent classes $\{x_1, x_2, x_3\}$, $\{x_4, x_5\}$, and $\{x_6\}$ with their elementary functions $\varphi_1 = x_1x_2x_3$, $\varphi_2 = (x_4 + x_5)$, and $\varphi_3 = x_6$, respectively. Figure 6 illustrates the circuit after the 1st iteration of the algorithm.

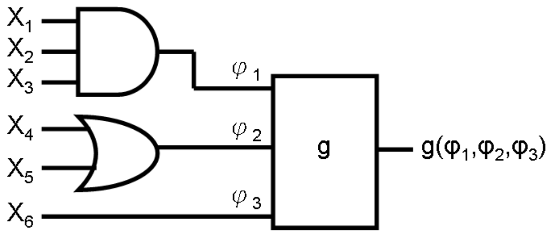

Figure 6: The Circuit after the 1st Iteration of the Algorithm on $f = x_1x_2x_3x_4 + x_1x_2x_3x_5 + x_4x_6 + x_5x_6$

4.3 Improvement on Constructing the New Function

Instead of using truth table to construct the new function, we propose a more efficient way. The definition of factoring is to express a term as the product of two terms such as $g{=}\varphi A{+}Z$. If $x_1{=}_0x_2{=}_0\cdots{=}_0x_r$, the associated elementary function is $\varphi{=}x_1x_2\cdots x_r$. The decomposed function would be $f{=}x_1x_2\cdots x_rA{+}Z{=}\varphi A{+}Z{=}g$. Thus g can be derived by substituting φ for x_i, $\forall\, i \in \{1, 2, \cdots, r\}$.

$$\begin{aligned} g(\varphi, A, Z) &= f(x_i = \varphi, A, Z) \quad \forall\, i \in \{1, 2, \cdots, r\} \\ &= \varphi\varphi\cdots\varphi A + Z \qquad \text{(absorption law)} \\ &= \varphi A + Z. \end{aligned}$$

The same rule applies to $x_1{=}_1x_2{=}_1\cdots{=}_1x_r$.

$$\begin{aligned} g(\varphi, A, Z) &= f(x_i = \varphi, A, Z) \\ &= (\varphi + \varphi + \cdots + \varphi)A + Z \\ &= \varphi A + Z. \end{aligned}$$

Here we continue using $f = x_1x_2x_3x_4 + x_1x_2x_3x_5 + x_4x_6 + x_5x_6$ as an example. It has been examined that $x_1 =_0 x_2 =_0 x_3$ and $x_4 =_1 x_5$. Thus $\varphi_1 = x_1x_2x_3$, $\varphi_2 = (x_4{+}x_5)$, and $\varphi_3 = x_6$. The new function $g(\varphi_1, \varphi_2, \varphi_3) = f(x_1 = \varphi_1, x_2 = \varphi_1, x_3 = \varphi_1, x_4 = \varphi_2, x_5 = \varphi_2, x_6 = \varphi_3) = \varphi_1\varphi_1\varphi_1\varphi_2 + \varphi_1\varphi_1\varphi_1\varphi_2 + \varphi_2\varphi_3 + \varphi_2\varphi_3 = \varphi_1\varphi_2 + \varphi_2\varphi_3$ can be constructed straightforward.

4.4 Time Complexity Analysis

Here we define some notations for analyzing time complexity of JPH's procedure and our improved one. Assume a function F is represented in SOP form. It is consisted of N variables and K products. Thus literals of F is about $N \times K$.

Examining the Adjacency Relation

Obviously, N^2 times of equivalence checking should be done to examine the adjacency relation of variables in JPH's procedure. A traditional technique of equivalence checking takes about $O(2^N)$. Hence this stage in JPH's procedure is bounded by $O(N^2 2^N)$.[3]

Constructing and checking matrix are main operations in this stage of our improved procedure. Constructing matrix needs to scan N equations, where each equation is about $N \times K$ literals. Checking matrix has to scan N^2 position of a matrix in worst case. Thus the time complexity is $O(N^2K + N^2) \approx O(N^2K)$.

Constructing the New Function

Instead of enumerating a truth table, which is about $O(2^N)$, we use substitutions in this stage. Hence the time complexity is $O(1)$.

Overall Analysis

From the analysis above, we find that examining the adjacency relation is the most critical part. Thus the time complexity of our improved procedure is bounded by $O(N^2K)$. As a result, our improved procedure is a more efficient approach.

4.5 A Go-through Example

To realize the function f to be fanout-free, we examine adjacent classes of $g(\varphi_1, \varphi_2, \varphi_3)$ again. We get $\varphi_1 =_1 \varphi_3$, so we have $\theta_1 = \varphi_1 + \varphi_3$ and $\theta_2 = \varphi_2$. We construct the new function $h(\theta_1, \theta_2){=}g(\varphi_1 = \theta_1, \varphi_2 = \theta_1, \varphi_3 = \theta_2){=}\theta_1\theta_2$.

Now we examine the adjacency relation among variables of $h(\theta_1, \theta_2)$. We find there is only one adjacent class in h, i.e., variables of h are mutually adjacent. So we recognize the function $f = x_1x_2x_3x_4 + x_1x_2x_3x_5 + x_4x_6 + x_5x_6$ as a fanout-free function which is $h{=}\theta_1\theta_2{=}(\varphi_1 + \varphi_3)\varphi_2{=}(x_1x_2x_3 + x_6)(x_4 + x_5){=}f$. Figure 7 shows each iteration of recognizing $f = (x_1x_2x_3 + x_6)(x_4 + x_5)$.

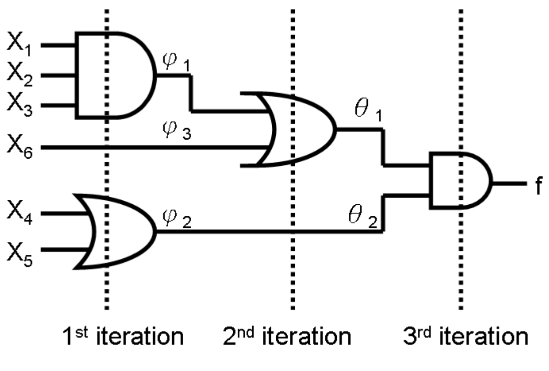

Figure 7: Each Iteration of Recognizing $f = (x_1x_2x_3 + x_6)(x_4 + x_5)$

[3]We omit the time to evaluate positive and negative cofactors.

Table 2: Experimental Results

Name	lits(sop)	#vars	CPU Time (s) IROF	JPH	Ours
l2_b10	10240	20	0.30	4	0.11
l4_b3	3072	24	0.10	6	0.08
l4_b6	7290	24	0.21	277	0.18
l6_b4	672	20	0.02	3	0.02
l6_b8a	132	52	0.02	>1hr	0.29
l6_b8b	24192	52	0.74	>1hr	2.08
l8_b5	3380	29	0.09	>1hr	0.11
l10_b3	2160	30	0.07	>1hr	0.16
l14_b3	6720	42	0.20	>1hr	0.72

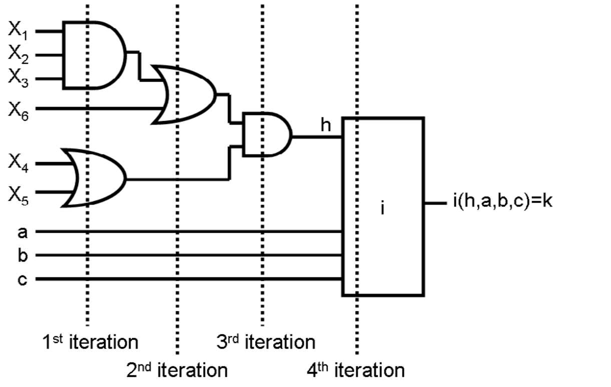

Figure 8: The Circuit of $k = (x_1x_2x_3 + x_6)(x_4 + x_5)(ab + bc + ac)$ after Our Recognition

5 Experimental Results and Analysis

We have implemented JPH's procedure [4] with our improvements within SIS [9] environment. We adopted Boolean functions proposed in [3] as the benchmarks and compared the results with that of the state-of-the-art IROF algorithm [3]. Experimental results are listed in Table 2. The first column shows the name of the function. The second column shows the number of literals in the sum-of-product form. The third column shows the number of variables of the function. The column labels "IROF" is empirical results of the IROF algorithm which were reported in [3]. The last two columns are the re-implemented JPH and our improved JPH results that ran by a Sun Blade 2500 machine. The CPU time is measured in second. These three algorithms all recognize an identical fanout-free function for each benchmark. So Table 2 shows the comparison on CPU time. According to Table 2, our approach improves the efficiency of original JPH's procedure and is more competitive with the state-of-the-art IROF algorithm [3].

Besides recognizing fanout-free functions, our method has advantage on recognizing a fanout function, while IROF gets nothing. Consider the function k=$(x_1x_2x_3 + x_6)(x_4 + x_5)(ab + bc + ac)$ which is not fanout-free due to the term $(ab + bc + ac)$. Our method will produce a partially fanout-free circuit as shown in Figure 8. IROF will just return k is not a fanout-free function.

6 Conclusions

Functions which have fanout-free forms can be realized with minimum literals. *John P. Hayes* had proposed an algorithm to recognize fanout-free functions based on adjacency of the input variables. We discover the *disappearance* property on the adjacency relation checking and apply it to accelerate JPH's procedure. Experimental results demonstrate that our improvements make JPH's procedure more competitive with the state-of-the-art method. Our method also produce a partially fanout-free function when recognizing a fanout function.

References

[1] Robert K. Brayton, Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis", *Kluwer Academic Publishers*, 1984.

[2] Martin C. Golumbic and Aviad Mintz, "Factoring Logic Functions using Graph Partitioning", *Proceedings of the 1999 IEEE/ACM International Conference on Computer-aided Design (ICCAD'1999)*, pp. 195-198, 1999.

[3] Martin C. Golumbic, Aviad Mintz, and Udi Rotics, "Factoring and Recognition of Read-Once Functions using Cographs and Normality", *Proceedings of the 38th Design Automation Conference (DAC'2001)*, pp. 109-114, 2001.

[4] John P. Hayes, "The Fanout Structure of Switching Functions", *Journal of the ACM*, 22:551-571, 1975.

[5] Michael John and Sebastian Smith, "Application-Specific Integrated Circuits", *Addison-Wesley Publishing Company*, 1997.

[6] Kurt Keutzer, "DAGON: Technology Mapping and Local Optimization", *Proceedings of the 24th Design Automation Conference (DAC'1987)*, pp. 341-347, 1987.

[7] Shin-Ichi Minato and Giovanni De Micheli, "Finding All Simple Disjunctive Decompositions Using Irredundant Sum-of-Products Forms", *Proceedings of the 1998 IEEE/ACM International Conference on Computer-aided Design (ICCAD'1998)*, pp. 111-117, 1998.

[8] Joram Pe'er and Ron Y. Pinter, "Minimal Decomposition of Boolean Functions Using Non-Repeating Literal Trees", *Proceedings of the IFIP Workshop on Logic and Architecture Synthesis*, pp. 129-139, 1995.

[9] Ellen M. Sentovich, Kanwar Jit Singh, Luciano Lavagno, Cho Moon, Rajeev Murgai, Alexander Saldanha, Hamid Savoj, Paul R. Stephan, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis", *Memorandum No. UCB/ERL M92/41*, 1992.