New Methods to Find Optimal Non-Disjoint Bi-Decompositions

Shigeru Yamashita Hiroshi Sawada Akira Nagoya

NTT Communication Science Laboratories
2-Chome, Hikaridai, Seika-cho, Soraku-gun, Kyoto 619-02 Japan
Tel: +81-774-95-{1867, 1866, 1860}
Fax: +81-774-95-1876
e-mail: {ger, sawada, nagoya}@cslab.kecl.ntt.co.jp

Abstract— This paper presents new efficient methods to find "optimal bi-decomposition" forms of logic functions. An "optimal bi-decomposition" form of $f(X)$ is $f = \alpha(g_1(X^1), g_2(X^2))$ where the total number of variables in X^1 and X^2 is the smallest among all bi-decomposition forms of f. We consider two methods; one's decomposition form is $(g_1 \cdot g_2)$ and the other's is $(g_1 \oplus g_2)$. The proposed methods can find one of the existing "optimal" decomposition forms efficiently based on the Branch-and-Bound algorithm. These methods can decompose incompletely specified functions. Preliminary experimental results show that the proposed methods can construct networks with fewer levels than conventional methods.

I. Introduction

When implementing a combinational logic function using a given technology, the desired function must be decomposed or factorized to smaller functions so that the decomposed functions can fit onto the implementation primitives of the technology. Many decomposition methods, therefore, have been proposed. Among them, AND/OR factoring and weak division [1] are supreme methods when expressions are in sum-of-product forms. In some cases, however, other approaches produce better results. For example, factoring with XOR can express some logic functions simpler than AND/OR factoring [2, 3]. As for the synthesis of LUT (Look-Up Table) networks, functional decomposition [4] based methods can often produce better results [5].

Most of the previously proposed functional decomposition methods have been based on Roth-Karp decomposition [6], and thus they decompose function f to the following form: $f = \alpha(g_1(X^B), \ldots, g_t(X^B), X^F) = \alpha(\vec{g}(X^B), X^F)$, where X^B and X^F are sets of variables. We can think of another strategy for functional decomposition: function f is decomposed into only two functions as $f = \alpha(g_1(X^1), g_2(X^2))$, where X^1 and X^2 are sets of variables. This decomposition is called bi-decomposition [7]. If X^1 and X^2 are disjoint, the bi-decomposition form can be found very quickly [7]. In some cases, a "non-disjoint" bi-decomposition form can provide the best decomposition. (An example will be shown in Section II.)

The methods proposed in [8, 9] can find non-disjoint bi-decomposition forms efficiently using the notion of "groupability". The methods can find a bi-decomposition form for given X^1 and X^2, but they still have a problem selecting the best X^1 and X^2.

In this paper, we propose new efficient methods to find "optimal" non-disjoint bi-decomposition forms of incompletely specified functions. Here, "optimal" means that the total number of variables in X^1 and X^2 is the smallest among all bi-decomposition forms. This meaning is thought to be adequate for the synthesis of LUT networks, because an LUT can realize a complex function if the number of input variables does not exceed the maximum number of inputs of the LUT. We think our methods can provide a solution to the problem of how to select the best X^1 and X^2, especially in LUT network synthesis.

This paper is organized as follows. In Section II, we explain non-disjoint bi-decomposition and formulate our problem. In Section III, we propose the novel methods to find "optimal" bi-decomposition forms. We present preliminary experimental results in Section IV. Section V concludes this paper.

II. Preliminaries

A. Non-Disjoint Bi-Decomposition

The decomposition form $f = \alpha(g_1(X^1), g_2(X^2))$ is called a bi-decomposition form [7]. If X^1 and X^2 are disjoint, it is called a "disjoint" bi-decomposition form. If X^1 and X^2 are not disjoint, it is called a "non-disjoint" bi-decomposition form.

Disjoint bi-decomposition forms are very useful for logic synthesis, and they can be found quickly [7]. However, there are functions that can be decomposed efficiently only by non-disjoint bi-decomposition. For example, suppose we want to decompose $(x_1 + x_2 + x_3) \cdot (x_2 \oplus x_3 \oplus x_4) \cdot (x_1 \oplus x_3 \oplus x_5)$. With disjoint bi-decomposition, we cannot decompose the function. With the recursive use of non-disjoint bi-decomposition, however, we can

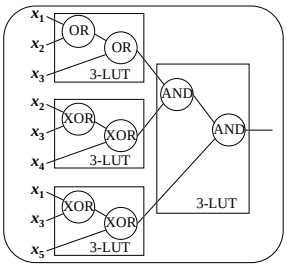

Fig. 1. An LUT Network Obtained by Non-Disjunctive Decomposition

decompose the function as shown in Fig. 1. If we want to realize the function by 3-input LUTs, we can get an LUT network as shown in Fig. 1. This network is the same as a straightforward realization from the expression. With a Roth-Karp decomposition based method, we can not find such a good decomposition form of this example.

B. Problem Formulation

In LUT network synthesis, one of the costs of a function is the number of variables which the function depends on. Therefore, we define an "**optimal**" bi-decomposition form as follows:

Definition 1 $f(X) = \alpha(g_1(X^1), g_2(X^2))$ is called an "**optimal**" bi-decomposition form if the total number of variables in X^1 and X^2 is the smallest among all bi-decomposition forms of f. □

If X^1 (or X^2) is an empty set, the decomposition is a *trivial* decomposition, which we ignore in this paper. For example, $f(X) = 1 \cdot f(X)$ is a trivial decomposition, and X^1 is an empty set. Even if the total number of variables in X^1 and X^2 of a trivial decomposition form is the smallest among all bi-decomposition forms, the trivial decomposition is *not* called optimal. Note that an optimal bi-decomposition of a function may be disjoint or non-disjoint depending on the function, and there may be no optimal bi-decomposition forms for some functions.

To find a bi-decomposition form $\alpha(g_1(X^1), g_2(X^2))$, we need to consider only three decomposition forms: $(g_1(X^1) \cdot g_2(X^2))$, $(g_1(X^1) + g_2(X^2))$, and $(g_1(X^1) \oplus g_2(X^2))$, which we call the "*AND-Decomposition*", "*OR-Decomposition*", and "*XOR-Decomposition*" forms, respectively. Since f can be decomposed into an *OR-Decomposition* form iff $\overline{f}$ can be decomposed into an *AND-Decomposition* form, we only consider *AND-Decomposition* and *XOR-Decomposition* in this paper.

For an incompletely specified function f, the goals of this paper are:

1. Finding an optimal *AND-Decomposition* form, and
2. Finding an optimal *XOR-Decomposition* form.

Although there may be more than two optimal bi-decomposition forms, our problem is to find just one of them.

III. Optimal Non-Disjoint Bi-Decompositions

In this section, we present efficient methods to find optimal *AND-Decomposition* and *XOR-Decomposition* forms.

A. Optimal AND-Decomposition

Here, we present a method to decompose incompletely specified function f into an optimal bi-decomposition form: $f(X) = g_1(X) \cdot g_2(X)$.

First we generate an initial solution (g_1, g_2) as will be mentioned later. From the initial solution, the recursive procedure "*DecompAND*" shown in Fig. 2 improves the solution to produce an optimal *AND-Decomposition* form of f based on the Branch-and-Bound algorithm. Although there may be more than two optimal solutions according to our definition in Section II, the procedure only finds one of them.

A.1 Definitions for *DecompAND*

In the procedure *DecompAND*, g_1 and g_2 are treated as four-valued functions whose values are 0, 1, $*0$ or $*$. $*$ means a usual don't care. $*0$, which is introduced in this paper, means that g_1 (g_2) can be treated as a usual don't care if g_2 (g_1) is treated as 0, and g_1 (g_2) must be 0 if g_2 (g_1) is treated as 1. In other words, $*0$ means that at least one of the functions must be 0. The following condition is maintained throughout *DecompAND*.

- $g_1(a) = *0$ iff $g_2(a) = *0$. (a is a minterm.)

This conditions means that if we change g_1 so that $(g_1(a) = *0)$ is changed to $(g_1(a) = 1)$, we also change g_2 so that $(g_2(a) = *0)$ is changed to $(g_2(a) = 0)$, and if we change g_1 so that $(g_1(a) = *0)$ is changed to $(g_1(a) = 0)$, we also change g_2 so that $(g_2(a) = *0)$ is changed to $(g_2(a) = *)$. The introduction of $*0$ makes it possible to find an optimal solution based on the Branch-and-Bound algorithm. The following definitions of functions and operations are used in our method. In the definitions, g is a four-valued function, f is an incompletely specified function (three-valued) or a four-valued function, and h is a completely specified function (two-valued).

- $ON(f)$ means a characteristic function that represents a set of minterms $\{a \mid f(a) = 1\}$.
- $OFF(f)$ means a characteristic function that represents a set of minterms $\{a \mid f(a) = 0\}$.
- $DC0(g)$ means a characteristic function that represents a set of minterms $\{a \mid g(a) = *0\}$.
- $DC(f)$ means a characteristic function that represents a set of minterms $\{a \mid f(a) = *\}$.
- h_{x_i} and $h_{\overline{x}_i}$ mean the positive and negative cofactors of h with respect to x_i, respectively.

```
1    best_g1, best_g2; /* global variables that store the best solution */
2    no_depend_best_g1, no_depend_best_g2; /* global variables that store the best solution */
3  DecompAND(g1, g2, no_depend_g1, no_depend_g2, elim_var_set) {
4        /* num(A) means the number of variables in A */
5    if ( (num(elim_var_set) + num(no_depend_g1) + num(no_depend_g2)) ≤
6        (num(no_depend_best_g1) + num(no_depend_best_g2)) ) return; /* the search space is pruned */
7    Take out one variable as x_i from elim_var_set;
8    back_up_g1 = g1; /* preserve g1 and g2 because ElimVarAnd may change g1 and g2 */
9    back_up_g2 = g2;
10   if ( (ElimVarAnd(x_i, g1, g2, no_depend_g2)) == TRUE){ /* Choice 1 */
11       no_depend_g1 = no_depend_g1 ∪ {x_i};
12       if((num(no_depend_g1) + num(no_depend_g2)) > (num(no_depend_best_g1) + num(no_depend_best_g2))){
13           best_g1 = g1; /* overwrite the best solution */
14           best_g2 = g2;
15           no_depend_best_g1 = no_depend_g1;
16           no_depend_best_g2 = no_depend_g2;
17       }
18       DecompAND(g1, g2, no_depend_g1, no_depend_g2, elim_var_set);
19   }
20   g1 = back_up_g1; /* restore g1 and g2 */
21   g2 = back_up_g2;
22   if ( (ElimVarAnd(x_i, g2, g1, no_depend_g1)) == TRUE){ /* Choice 2 */
23       no_depend_g2 = no_depend_g2 ∪ {x_i};
24       if((num(no_depend_g1) + num(no_depend_g2)) > (num(no_depend_best_g1) + num(no_depend_best_g2))){
25           best_g1 = g1; /* overwrite the best solution */
26           best_g2 = g2;
27           no_depend_best_g1 = no_depend_g1;
28           no_depend_best_g2 = no_depend_g2;
29       }
30       DecompAND(g1, g2, no_depend_g1, no_depend_g2, elim_var_set);
31   }
32   g1 = back_up_g1; /* restore g1 and g2 */
33   g2 = back_up_g2;
34   DecompAND(g1, g2, no_depend_g1, no_depend_g2, elim_var_set); /* Choice 3 */
35 }
```

Fig. 2. *DecompAND*

- $Smooth(x_i, h)$ means $h_{x_i} + h_{\overline{x}_i}$.
- $SmoothSet(var\_set, h)$ means a function that is obtained by applying successively $Smooth(x_i, h)$ to h for all x_i in var_set.
- $EnlargeON(g, h)$ means an operation to change g so that $ON(g)$ is changed to $ON(g) + h$. (Note that $OFF(g)$, $DC0(g)$ and $DC(g)$ must be changed to proper functions at the same time. Similar operations are also needed at $EnlargeOFF(g, h)$ and $EnlargeDC(g, h)$.)
- $EnlargeOFF(g, h)$ means an operation to change g so that $OFF(g)$ is changed to $OFF(g) + h$.
- $EnlargeDC(g, h)$ means an operation to change g so that $DC(g)$ is changed to $DC(g) + h$.

Note that $ON(g)$, $OFF(g)$, $DC0(g)$ and $DC(g)$ are completely specified functions, whereas g is a four-valued function.

The definitions of the variables used in *DecompAND* are as follows.

- $best\_g_1$ and $best\_g_2$ represent g_1 and g_2 in an optimal solution, respectively.
- *no_depend_(function)* represents a set of variables that have already been eliminated explicitly from the dependency of *function*.
- *elim_var_set* represents a set of variables that have the possibility of being eliminated from the dependency of g_1 or g_2.

A.2 *DecompAND*

Now, we explain the procedure *DecompAND* using an example. Suppose we want to find an optimal *AND-Decomposition* of function f: the ON-set is $(x_2 + \overline{x}_4) \cdot (x_1 \cdot x_4 + \overline{x}_1 \cdot \overline{x}_3)$ and the DC-set is $x_3 \cdot (x_2 \cdot \overline{x}_4 + x_1 \cdot \overline{x}_2 \cdot x_4)$. The truth table of f is shown at the top left-hand corner of Fig. 3. At first $best\_g_1$ and $best\_g_2$ are set to *null*. $no\_depend\_best\_g_1$ and $no\_depend\_best\_g_2$ are set to $\emptyset$. *elim_var_set* is set to the set of variables which f depends on. In the example, it is set to $\{x_1, x_2, x_3, x_4\}$. Initial g_1 (g_2 is the same) is produced to satisfy $ON(g_1) = ON(f)$, $DC0(g_1) = OFF(f)$, and $DC(g_1) = DC(f)$. Clearly,

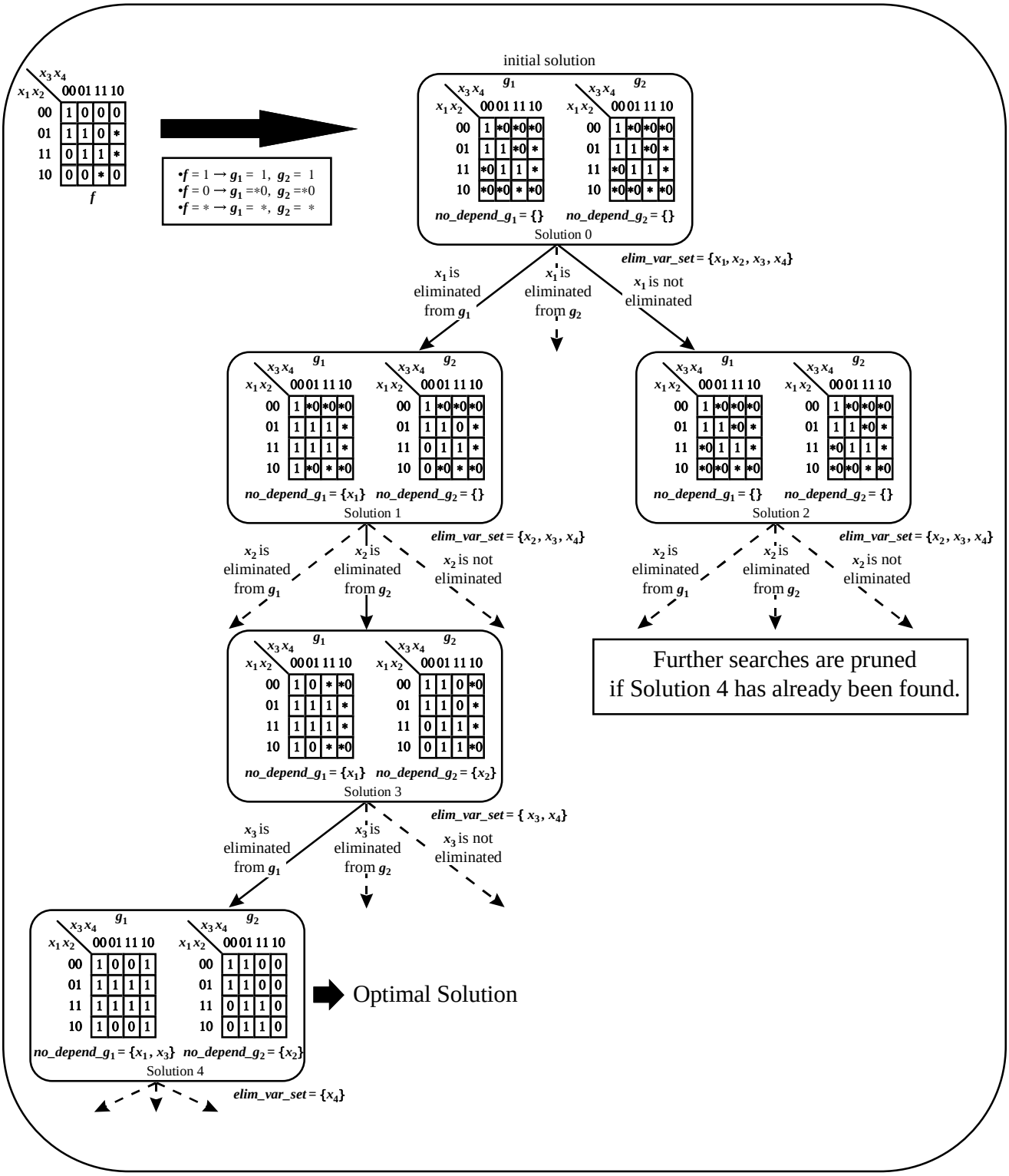

Fig. 3. A Search Tree for Finding an Optimal *AND-Decomposition*

this initial solution satisfies $f(X) = g_1(X) \cdot g_2(X)$. In the example, the initial solution is shown as "Solution 0" in Fig. 3. In the initial solution, *no_depend_*g_1 and *no_depend_*g_2 are $\emptyset$. Initial g_1 and g_2 have $*$s and $*$0s. By specifying these values to 1 or 0, *DecompAND* eliminates variables from the dependency of g_1 or g_2. The search tree of the example is shown in Fig. 3. For instance, from "Solution 0" to "Solution 1" x_1 is eliminated from the dependency of g_1, and x_1 is added to *no_depend_*g_1. If x_1 is eliminated from the dependency of g_1, x_1 cannot be eliminated from the dependency of g_2, because f depends on x_1. Therefore, when x_1 is eliminated from the dependency of g_1, we delete x_1 from *elim_var_set*.

From an intermediate solution, we have three choices to search further:

Choice 1 We eliminate a variable (x_i) in *elim_var_set* from the dependency of g_1, and then search further. (This corresponds to line 18 in Fig. 2 and the left arrow from each solution in Fig. 3.)

Choice 2 We eliminate a variable (x_i) in *elim_var_set* from the dependency of g_2, and then search further. (This corresponds to line 30 in Fig. 2 and the middle arrow from each solution in Fig. 3.)

Choice 3 We do not eliminate a variable (x_i) in *elim_var_set* from the dependency of either g_1 and g_2, and then search further. (This corresponds to line 34 in Fig. 2 and the right arrow from each solution in Fig. 3.)

If we select Choice 1 or 2, we must eliminate x_i from the dependency of g_1 or g_2, respectively. This is done by the procedure *ElimVarAnd* which is mentioned in the next section.

If we successfully eliminate a variable from an intermediate solution, and a new solution is better than the best solution: (*best_*g_1, *best_*g_2), we overwrite the best solution at lines 13 to 16, and 25 to 28 in Fig. 2.

Let us consider when a search is pruned. For example, if "Solution 4" in Fig. 3 has already been found, we do not need to search further from "Solution 2". The reason is as follows. The total number of variables in *no_depend_*g_1 and *no_depend_*g_2 of "Solution 4" and that of "Solution 2" are three and zero, respectively. From "Solution 2", we can eliminate three variables from the dependency of g_1 or g_2 at best because the number of variables in *elim_var_set* of "Solution 2" is three. Therefore, we cannot find a better solution than "Solution 4" in the search space from "Solution 2". This check is done at lines 5 and 6 in Fig. 2.

A.3 *ElimVarAnd*

The procedure *ElimVarAnd*(x_i, g_1, g_2, *no_depend_*g_2) used in *DecompAND* eliminates x_i from the dependency of g_1. This is shown in Fig. 4. Of course, the following conditions must be kept throughout the operations of *ElimVarAnd*.

Condition 1 $f = g_1 \cdot g_2$.

Condition 2 g_1 does not depend on the variables in *no_depend_*g_1.

Condition 3 g_2 does not depend on the variables in *no_depend_*g_2.

For example, from "Solution 3" in Fig. 3, x_3 is eliminated from the dependency of g_1 to get "Solution 4" as follows. At first we calculate what parts of $DC(g_1)$ and $DC0(g_1)$ must be changed to $ON(g_1)$ so that g_1 does not depend on x_i. This calculation corresponds to *change*1_g_1 at line 5 in Fig. 4. Next, we calculate what parts of $DC(g_1)$ and $DC0(g_1)$ must be changed to $OFF(g_1)$ so that g_1 does not depend on x_i. This calculation corresponds to *change*0_g_1 at line 6 in Fig. 4. In the example, these functions are calculated as "*change*1_g_1" and "*change*0_g_1" in Fig. 5, respectively.

Then, we change g_1 by $EnlargeON(g_1, change1\_g_1)$ and $EnlargeOFF(g_1, change0\_g_1)$ at lines 13 and 14 in Fig. 4. Although we change g_1 by the above operations, we cannot eliminate x_i from the dependency of g_1 when $(Smooth(x_i, ON(g_1)) \cdot Smooth(x_i, OFF(g_1)))$ is not the constant 0 function. This is checked at line 3 in Fig. 4, and *FALSE* is returned if x_i cannot be eliminated. For example, we cannot eliminate x_4 from g_1 of "Solution 3" in Fig. 3. This is because $Smooth(x_4, ON(g_1)) \cdot Smooth(x_4, OFF(g_1))$ (shown as "*obstacle*" in Fig. 5) is not the constant 0 function. Therefore, *ElimVarAnd* returns *FALSE* in this case. Indeed the parts of $ON(g_1)$ and $OFF(g_1)$ masked by $ON($"*obstacle*"$)$ make it impossible to eliminate x_4 from the dependency of g_1 of "Solution 3".

After these operations, g_1 is changed to a function which does not depend on x_3. This function is shown as "*new_*g_1" in Fig. 5. However, remember that $*0$ is not a usual don't care, and therefore, we need some more operations as follows to satisfy Condition 1. If a part of $DC0(g_1)$ is changed to $ON(g_1)$ by $EnlargeON(g_1, change1\_g_1)$, the corresponding part of $DC0(g_2)$ must be changed to $OFF(g_2)$. The corresponding part of $DC0(g_2)$ is calculated as $must\_change0\_g_2 = change1\_g_1 \cdot DC0(g_1)$ (at line 8 in Fig. 4). This function is shown as "first *must_change*0_g_2" in Fig. 5. Therefore, $EnlargeOFF(g_2, must\_change0\_g_2)$ at line 15 in Fig. 4 is needed.

By the operations mentioned above, we can get a new solution: (*new_*g_1, half-finished *new_*g_2), where *new_*g_1 does not depend on x_3 and the solution satisfies Condition 1. *new_*g_1 and half-finished *new_*g_2 are shown in Fig. 5. Here, let us check Conditions 2 and 3. Since g_1 does not depend on the variables in *no_depend_*g_1, *change*1_g_1 and *change*0_g_1 do

```
1   ElimVarAnd(x_i, g_1, g_2, no_depend_g_2){
2           /* to check if there is a possibility of eliminating x_i from g_1 */
3       if ((Smooth(x_i, ON(g_1)) · Smooth(x_i, OFF(g_1))) != the constant 0 function) return FALSE;
4           /* to calculate what parts of DC(g_1) and DC0(g_1) must be changed */
5       change1_g_1 = (DC(g_1) + DC0(g_1)) · Smooth(x_i, ON(g_1)); /* needs to be changed to 1 */
6       change0_g_1 = (DC(g_1) + DC0(g_1)) · Smooth(x_i, OFF(g_1)); /* needs to be changed to 0 */
7           /* to calculate what parts of DC0(g_2) must be changed for Condition 1*/
8       must_change0_g_2 = change1_g_1 · DC0(g_1); /* DC0(g_1) is the same as DC0(g_2) */
9           /* to enlarge must_change0_g_2 for Condition 3*/
10      must_change0_g_2 = SmoothSet(no_depend_g_2, must_change0_g_2) ;
11          /* to check if must_change0_g_2 includes ON(g_2) */
12      if ((ON(g_2) · must_change0_g_2) != the constant 0 function) return FALSE;
13      EnlargeON(g_1, change1_g_1);
14      EnlargeOFF(g_1, change0_g_1);
15      EnlargeOFF(g_2, must_change0_g_2);
16      EnlargeDC(g_1, (DC0(g_1) · OFF(g_2)));
17      EnlargeDC(g_2, (DC0(g_2) · OFF(g_1)));
18      return TRUE;
19  }
```

Fig. 4. *ElimVarAnd*

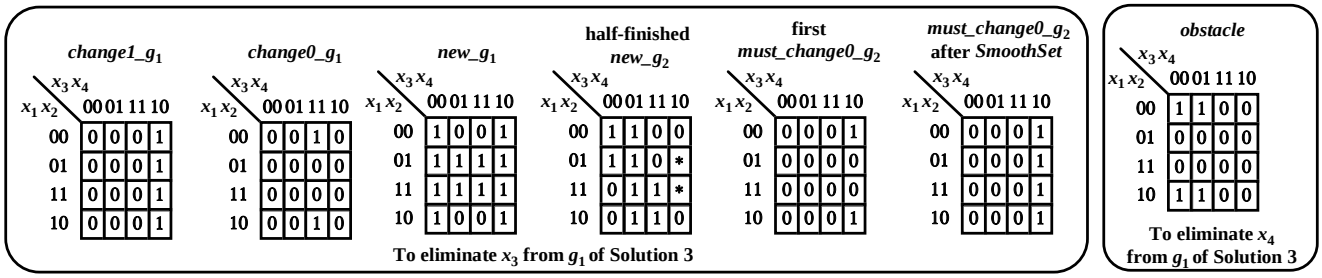

Fig. 5. Functions to Explain *ElimVarAnd*

not depend on the variables in $no\_depend\_g_1$. Therefore, new_g_1 still does not depend on the variables in $no\_depend\_g_1$ after $EnlargeON(g_1, change1\_g_1)$ and $EnlargeOFF(g_1, change0\_g_1)$, that is, Condition 2 is satisfied. However, Condition 3 is not satisfied because there is no guarantee that $must\_change0\_g_2$ does not depend on the variables in $no\_depend\_g_2$. In the example, $must\_change0\_g_2$ (see "first $must\_change0\_g_2$" in Fig. 5) and half-finished new_g_2 indeed depend on x_2 which is a variable in $no\_depend\_g_2$. Thus, if we need to change a part of $DC0(g_2)$ to $OFF(g_2)$, we must also change the corresponding part obtained by $SmoothSet(no\_depend\_g_2, the\ part)$ to $OFF(g_2)$ so that g_2 does not depend on the variables in $no\_depend\_g_2$. In the example, $SmoothSet(no\_depend\_g_2, must\_change0\_g_2)$ is calculated as "$must\_change0\_g_2$ after $SmoothSet$" in Fig. 5. Therefore, the calculation at line 10 in Fig. 4 is needed.

In some cases, $must\_change0\_g_2$ after $SmoothSet$ is a function that includes a part of $ON(g_2)$. In such cases, we cannot do $EnlargeOFF(g_2, must\_change0\_g_2)$ at line 15 in Fig. 4 while keeping Condition 1. Therefore, in such cases, x_i cannot be eliminated from g_1 while keeping the above three conditions, and *ElimVarAnd* returns *FALSE* (at line 12 in Fig. 4). If during the above operations a part of $DC0(g_1)$ ($DC0(g_2)$) is changed to $OFF(g_1)$ ($OFF(g_2)$), the corresponding part of $DC0(g_2)$ ($DC0(g_1)$) can be changed to don't care (usual don't care). Therefore, we can enlarge the don't cares of g_1 and g_2 by $EnlargeDC(g_1, (DC0(g_1) \cdot OFF(g_2)))$ at line 16 in Fig. 4 and $EnlargeDC(g_2, (DC0(g_2) \cdot OFF(g_1)))$ at line 17 in Fig. 4, respectively. In the example, finally we get g_1 and g_2 of "Solution 4" in Fig. 3.

In this way, *DecompAND* eliminates variables one by one from the dependency of g_1 and g_2 to find an optimal *AND-Decomposition*. In the example, "Solution 4" is an optimal *AND-Decomposition* where $g_1 = (x_2 + \overline{x}_4)$ and $g_2 = (x_1 \cdot x_4 + \overline{x}_1 \cdot \overline{x}_3)$.

It is clear that *DecompAND* can find one of the existing optimal solutions by elimination of any order of variables. The order only affects the execution time and which solution is found among the optimal solutions if there are more than two optimal solutions.

B. Optimal XOR-Decomposition

Here, we present a method to decompose incompletely

```
1   ElimVarXor(x_i, g_1, g_2, no_depend_g_1, no_depend_g_2){
2           /* to calculate what parts of ON(g_1) and OFF(g_1) must be changed */
3       depending_x_i_g_1 = Smooth(x_i, ON(g_1)) · Smooth(x_i, OFF(g_1));
4       no_depend_All = no_depend_g_1 + no_depend_g_2;
5           /* to enlarge depending_x_i_g_1 for Conditions 2 and 3*/
6       depending_x_i_g_1 = SmoothSet(no_depend_All, depending_x_i_g_1) ;
7           /* to check if depending_x_i_g_1 after SmoothSet includes the part where g_1 must not be reversed to eliminate x_i */
8       if ( ((depending_x_i_g_1) · Consensus(x_i, ON(g_1))) != the constant 0 function) return FALSE;
9       if ( ((depending_x_i_g_1) · Consensus(x_i, OFF(g_1))) != the constant 0 function) return FALSE;
10      must_reverse = {depending_x_i_g_1}_{x̄_i};
11      Reverse(g_1, must_reverse);
12      Reverse(g_2, must_reverse);
13      change0 = Smooth(x_i, OFF(g_1)) · DC(g_1);
14      change1 = Smooth(x_i, ON(g_1)) · DC(g_1);
15      EnlargeOFF(g_1, change0);
16      EnlargeON(g_1, change1);
17      return TRUE;
18  }
```

Fig. 6. *ElimVarXor*

specified function f into an optimal bi-decomposition form: $f(X) = g_1(X) \oplus g_2(X)$. The whole procedure is called *DecompXOR*, which is almost the same as *DecompAND*. The different points are as follows:

- How to eliminate the variable dependency from g_1 and g_2 is different. Therefore, in *DecompXOR*, *ElimVarAnd* is replaced with "*ElimVarXor*" (mentioned later).
- In *ElimVarXor*, we do not use $*0$ but only $*$ (usual don't care). Therefore, in *DecompXOR*, g_1 and g_2 are usual incompletely specified functions (three-valued).
- The initial solution is different. Initial g_1 is the same as f. Initial g_2 is the constant 0 function with the same don't cares as f. Clearly, this initial solution satisfies $f(X) = g_1(X) \oplus g_2(X)$.

From the initial solution, we eliminate the variable dependency from g_1 or g_2 by *ElimVarXor* in Fig. 6. We explain procedure *DecompXOR* using an example. Suppose we want to find an optimal *XOR-Decomposition* of function f: the ON-set is $(x_2 + \overline{x}_4) \oplus (x_1 \cdot x_4 + \overline{x}_1 \cdot \overline{x}_3)$ and the DC-set is $x_3 \cdot (x_2 \cdot \overline{x}_4 + \overline{x}_1 \cdot \overline{x}_2 \cdot x_4)$. The truth table of f is shown at the top left-hand corner of Fig. 7. The search tree of the example is shown in Fig. 7. The following additional definitions of a function and an operation are used in procedure *ElimVarXor*.

- $Consensus(x_i, h)$ means $h_{x_i} \cdot h_{\overline{x}_i}$.
- $Reverse(g, h)$ means an operation to *reverse* the values of g's part which is masked by $ON(h)$. (*reverse* is an operation to change 1 to 0, 0 to 1, and $*$ to $*$.)

Here, we explain procedure *ElimVarXor*. Of course, the following conditions must be kept throughout the operations of *ElimVarXor*.

Condition 1 $f = g_1 \oplus g_2$.

Condition 2 g_1 does not depend on the variables in *no_depend_g*$_1$.

Condition 3 g_2 does not depend on the variables in *no_depend_g*$_2$.

To satisfy Condition 1, we can only do a pair of operations: $Reverse(g_1, h)$ and $Reverse(g_2, h)$. This is because although we *reverse* the values of some parts of g_1, $(g_1 \oplus g_2)$ does not change if the values of the same parts of g_2 are also *reversed*. Of course, we can freely change $*$s of g_1 and g_2 to 1 or 0.

At first, we calculate what parts of $ON(g_1)$ and $OFF(g_1)$ cause g_1 to depend on x_i. This calculation corresponds to *depending_x*$_i$*_g*$_1$ at line 3 in Fig. 6. x_i can be eliminated from the dependency of g_1, if we do $Reverse(g_1, must\_reverse)$, where *must_reverse* satisfies the following two conditions which we call "*Conditions for must_reverse*":

- $Smooth(x_i, must\_reverse) =$ *depending_x*$_i$*_g*$_1$.
- $Consensus(x_i, must\_reverse)$ = the constant 0 function.

There are many candidates for *must_reverse* that satisfy the "*Conditions for must_reverse*". Among them we chose $\{$*depending_x*$_i$*_g*$_1\}_{\overline{x}_i}$ at line 10 in Fig. 6. For example, when we eliminate x_3 from the dependency of g_1 of "Solution 3" in Fig. 7, *depending_x*$_3$*_g*$_1$ and $\{$*depending_x*$_3$*_g*$_1\}_{\overline{x}_3}$ are calculated as shown in Fig. 8. Then, we change g_1 and g_2 by $Reverse(g_1, must\_reverse)$ and $Reverse(g_2, must\_reverse)$ at lines 11 and 12 in Fig. 6. To add to these operations, we must change a part of $DC(g_1)$ to $ON(g_1)$ or $OFF(g_1)$ so that g_1 does not depend on x_i. This is done at lines 13 to 16 in Fig. 6. For example, when we eliminate x_3 from the dependency

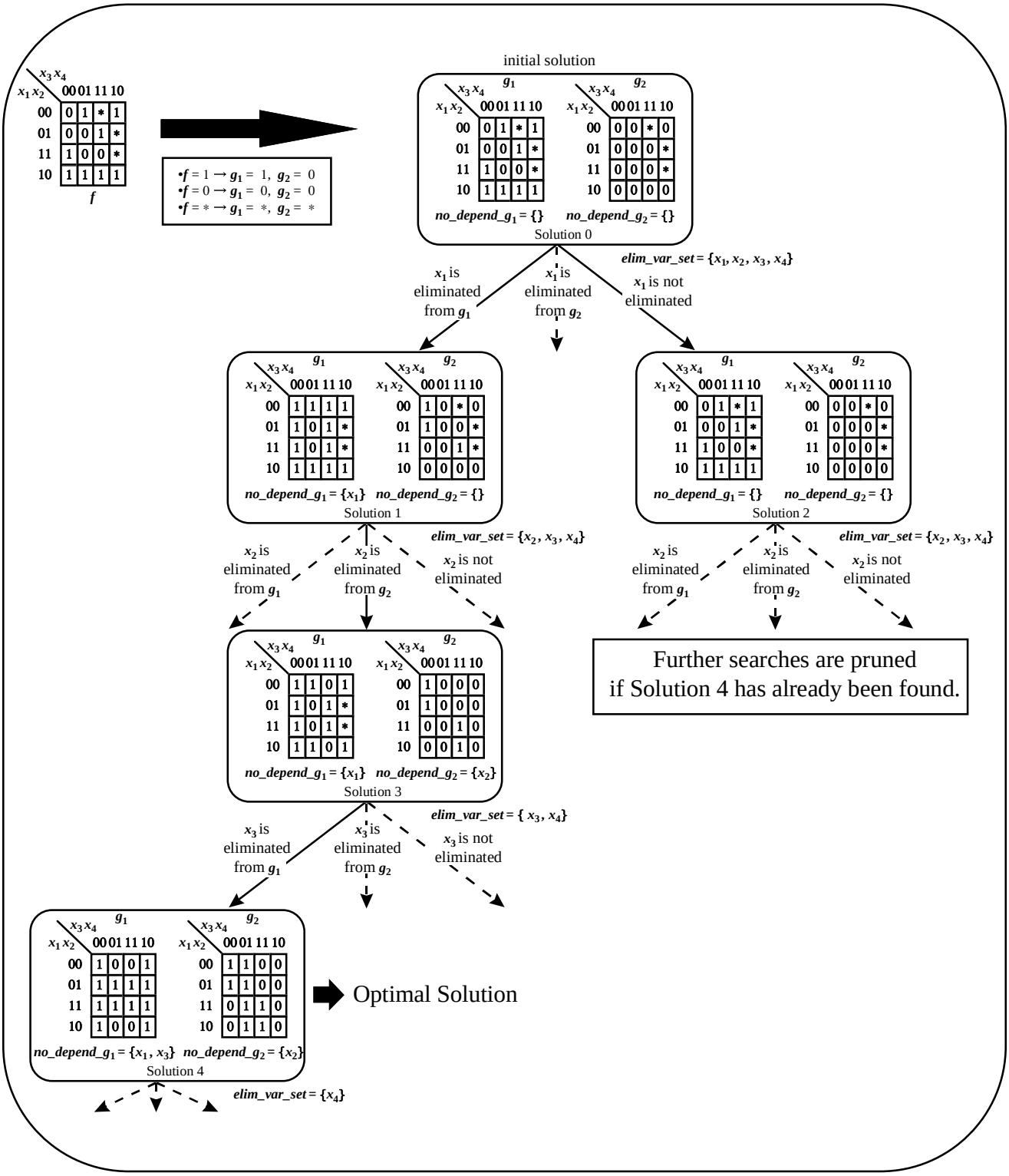

Fig. 7. A Search Tree for Finding an Optimal *XOR-Decomposition*

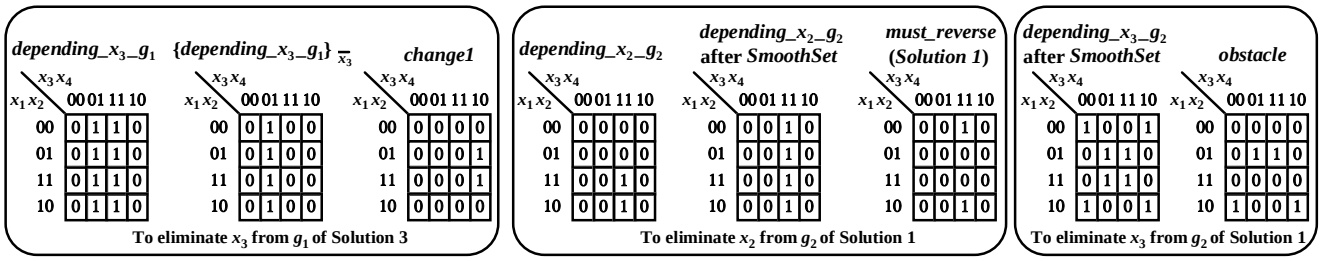

Fig. 8. Functions to Explain *ElimVarXor*

of g_1 of "Solution 3" in Fig. 7, *change*0 is the constant 0 function and *change*1 is calculated as shown in Fig. 8. After these operations we get "Solution 4" in Fig. 7, which is an optimal solution in the example.

For Conditions 2 and 3, *must_reverse* must not depend on the variables in *no_depend_g*$_1$ and *no_depend_g*$_2$. Therefore, line 6 in Fig. 6 is needed. For example, when we eliminate x_2 from the dependency of g_2 of "Solution 1" in Fig. 7, *depending_x*$_2$*_g*$_2$ at line 3 in Fig. 6 is calculated as "*depending_x*$_2$*_g*$_2$" in Fig. 8. This function depends on x_1, and therefore, we must *Smooth* this function with respect to x_1 and get *depending_x*$_2$*_g*$_2$ at line 6 in Fig. 6 as "*depending_x*$_2$*_g*$_2$ after *SmoothSet*" in Fig. 8. In this case, *must_reverse* at line 10 in Fig. 6 is calculated as "*must_reverse* (Solution 1)" in Fig. 8 and we successfully eliminate x_2 from the dependency of g_2 to get "Solution 3".

Here, we consider a case where *ElimVarXor* fails. If we want to eliminate x_3 from the dependency of g_2 of "Solution 1" in Fig. 7, *depending_x*$_3$*_g*$_2$ at line 6 in Fig. 6 is calculated as "*depending_x*$_3$*_g*$_2$ after *SmoothSet*" in Fig. 8. In this case, $((depending\_x_3\_g_2) \cdot Consensus(x_3, OFF(g_2)))$ is calculated as "*obstacle*" in Fig. 8. *obstacle* is not the constant 0 function. Therefore, there is no *must_reverse*, where *must_reverse* satisfies the "*Conditions for must_reverse*" and $Reverse(g_2, must\_reverse)$ can eliminate x_3 from the dependency of g_2. This check is done at line 9 in Fig. 6. Line 8 in Fig. 6 is also needed.

There are many candidates for *must_reverse* that satisfy the "*Conditions for must_reverse*". Among them we chose $\{depending\_x_i\_g_1\}_{\overline{x}_i}$. If we choose another *must_reverse* among the candidates, we get another solution (g_1', g_2'). However, the variables which g_1' and g_2' depend on are the same for all candidates. Therefore, *DecompXOR* can find one of the existing optimal solutions even though we choose $\{depending\_x_i\_g_1\}_{\overline{x}_i}$ among the many candidates for *must_reverse*. (The proof is omitted due to space limitations.) Like *DecompAND*, *DecompXOR* can find one of the existing optimal solutions by elimination of any order of variables. (The proof is also omitted due to space limitations.)

IV. Experimental Results

We know the methods presented here find optimal bi-decomposition forms of logic functions. However, it is not certain that the methods can produce good circuits in reasonable time. Therefore, we performed a preliminary experiment on MCNC [10] benchmark circuits. The experiment was to generate a network with two-input nodes as [9] to compare the proposed methods with the other bi-decomposition method proposed in [9]. For each output function of a benchmark circuit, we decomposed the function recursively to two-input functions. The decomposition of function f was done as follows:

- Find an optimal AND-Decomposition of f.
- Find an optimal AND-Decomposition of $\overline{f}$. (This means an OR-Decomposition of f.)
- Find an optimal XOR-Decomposition of f.
- Select the best decomposition among the above three.
- If there is no *non-trivial* bi-decomposition of f, f is decomposed by Shannon Expansion.

Table I shows a comparison of our results and other results. The other results were taken from [9] and generated by MIS [1]. (The network levels were not reported in [9], therefore, we produced networks by MIS to examine the network levels. The script of MIS was the same as in [9].) In Table I, "nodes", "lev" and "CPU" show the number of two-input nodes, the network levels and the CPU run-time (sec.) on a Sun Ultra 2 2200, respectively. "ratio" shows the ratios of the results of [9] and our method to those of MIS.

Although our results were produced without the multiple use of internal nodes, unlike [9] and MIS, the number of nodes is almost the same. Moreover, as for the network levels, the networks obtained by our method had fewer levels than those obtained by MIS.

In the experiment, our method could not find *non-trivial* bi-decompositions in some cases. In other words, for some functions, we cannot find a decomposition form: $f(X) = \alpha(g_1(X^1), g_2(X^2))$ where the number of variables

TABLE I
DECOMPOSITION TO TWO-INPUT NODES

Circuits	[9]	MIS		Ours		
	nodes	nodes	lev	nodes	lev	CPU
5xp1	73	100	22	77	7	0.28
9symml	199	243	14	202	10	1.04
con1	15	17	4	16	4	0.03
duke2	550	641	26	759	11	30.80
e64	390	877	11	655	7	5.53
f51m	60	99	26	65	6	0.12
misex1	56	42	12	74	6	0.11
misex2	109	127	8	158	5	0.16
misex3c	757	457	45	820	18	78.36
rd53	21	32	9	30	5	0.05
rd73	64	93	15	100	8	0.32
rd84	80	169	18	172	10	0.70
sao2	117	144	28	158	10	1.01
z4ml	16	46	13	69	9	0.18
total	2507	3087	251	3355	116	-
ratio	0.81	1.00	1.00	1.08	0.46	-

in X_1 and that in X_2 are both less than the number in X. We cannot decompose such functions by bi-decomposition. In the experiment, therefore, we only used Shannon Expansion for such functions. Unfortunately, we think this use of Shannon Expansion made our results worse. In such cases we must use another decomposition method, such as general decomposition: $f = \alpha(\vec{g}(X^B), X^F)$.

In the proposed methods, we do not share internal nodes among several functions. We think this was another reason causing our results to be worse. Therefore, we must combine the proposed methods with the multiple use of internal nodes. We believe the sharing of internal nodes is not so difficult. For example, we can think of the following two strategies.

- After all decompositions, we can check whether a node can be replaced with another node by the method proposed in [11].
- When f is decomposed to $\alpha(g_1(X^1), g_2(X^2))$, we can check whether an existing function can be used as g_1 (or g_2) by the boolean resubstitution and the support minimization technique proposed in [5].

V. CONCLUSION AND FUTURE WORK

We have presented new efficient methods to find "optimal" non-disjoint bi-decomposition AND and XOR forms. Our methods have the following properties.

- They can decompose incompletely specified functions.
- They eliminate variables one by one from the dependency of g_1 and g_2 of an intermediate solution.
- The Branch-and-Bound algorithm is used to find an optimal solution.

In this paper, a decomposition: $f(X) = \alpha(g_1(X^1), g_2(X^2))$ is "optimal" if the total number of variables in X^1 and X^2 is the smallest. Of course, there is no guarantee that the final decomposed networks are optimal even if we adopt "optimal" decompositions at intermediate decompositions. We think, however, our meaning of "optimal" is adequate if we want to decompose functions to LUTs.

There are some functions that do not have *non-trivial* bi-decompositions. These functions cannot be decomposed by bi-decomposition; in the experiment, only Shannon Expansion was used for them, which is not such a good strategy. Therefore, we must develop the proposed methods by combining them with other decomposition methods. According to the experimental results, we think the proposed methods decompose functions to networks with few levels. Therefore, we plan to utilize the proposed methods to generate initial LUT networks with few levels. At this time, we do not share internal nodes among several functions in the proposed methods. Therefore, we have yet to combine the proposed methods with the multiple use of internal nodes.

REFERENCES

[1] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, "MIS: a multiple-level logic optimization system," *IEEE Trans. CAD*, vol. CAD-6, pp. 1062–1081, Nov. 1987.

[2] G. Lee, D. Bang, and J. Saul, "Synthesis of LUT-type FPGAs using AND/OR/EXOR representations," in *Proc. SASIMI*, pp. 74–77, Nov. 1996.

[3] Y. Matsunaga, "An attempt to factor logic functions using exclusive-or decomposition," in *Proc. SASIMI*, pp. 78–83, Nov. 1996.

[4] R. L. Ashenhurst, "The decomposition of switching functions," in *Proceedings of an International Symposium on the Theory of Switching*, pp. 74–116, Apr. 1957.

[5] H. Sawada, T. Suyama, and A. Nagoya, "Logic synthesis for look-up table based FPGAs using functional decomposition and support minimization," in *Proc. ICCAD*, pp. 353–358, Nov. 1995.

[6] J. P. Roth and R. M. Karp, "Minimization over boolean graphs," *IBM journal*, pp. 227–238, Apr. 1962.

[7] T. Sasao and J. T. Butler, "On bi-decompositions of logic functions," in *Notes of International Workshop on Logic Synthesis (IWLS'97)*, May 1997.

[8] D. Bochmann, F. Dresig, and B. Steinbach, "A new decomposition method for multilevel circuit design," in *Proc. EDAC*, pp. 374–377, Feb. 1991.

[9] B. Steinbach and A. Wereszczynski, "Synthesis of multi-level circuits using EXOR-gates," in *Proc. of Reed-Muller'95*, pp. 161–168, Aug. 1995.

[10] S. Yang, "Logic synthesis and optimization benchmarks user guide version 3.0," *MCNC*, Jan. 1991.

[11] S. Yamashita, H. Sawada, and A. Nagoya, "A new method to express functional permissibilities for LUT based FPGAs and its applications," in *Proc. ICCAD*, pp. 254–261, Nov. 1996.