An Exact Optimization Algorithm for Linear Decomposition of Index Generation Functions

Shinobu Nagayama* Tsutomu Sasao† Jon T. Butler‡

*Dept. of Computer and Network Eng., Hiroshima City University, Hiroshima, JAPAN
†Dept. of Computer Science, Meiji University, Kawasaki, JAPAN
‡Dept. of Electr. and Comp. Eng., Naval Postgraduate School, Monterey, CA USA

***Abstract*—This paper proposes an exact optimization algorithm based on a branch and bound method for linear decomposition of index generation functions. The proposed algorithm efficiently finds the optimum linear decomposition of an index generation function by pruning non-optimum solutions using effective branch and bound strategies. The branch strategy is based on our previous heuristic [2] using a balanced decision tree, and the bound is based on a lower bound on the number of variables needed for linear decomposition. Experimental results using a benchmark index generation function show its optimum linear decompositions and effectiveness of the strategies.**

***Index Terms*—Index generation functions; linear decomposition; incompletely specified functions; logic design; exact optimization algorithm; branch and bound method.**

1. Introduction

Pattern matching and text search are basic operations used in many applications, such as detection of computer viruses and packet classification. These operations can be logically modeled as *index generation functions* [4], [5]. Since index generation functions are frequently updated particularly in the above network applications, a memory-based design of index generation functions is desired.

To design index generation functions using memory efficiently, a design method using *linear decomposition* [1], [3] of index generation functions has been proposed [7]. This method realizes an index generation function $f(x_1, x_2, \ldots, x_n)$ using two blocks L and G, as shown in Fig. 1. The first block L realizes linear functions y_i $(i = 1, 2, \ldots, p)$ with EXOR gates, registers, and multiplexers, and the second one G realizes a general function with a $(2^p \times q)$-bit memory [7], where p is the number of linear functions, and q is the number of bits needed to represent function values.

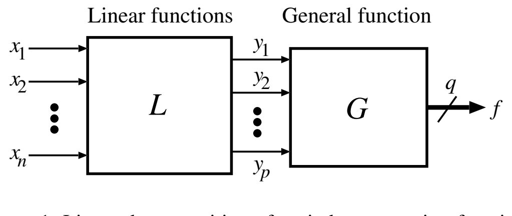

Figure 1. Linear decomposition of an index generation function.

In this design method, minimization of p is important to reduce size of the memory for G. Thus, various minimization algorithms have been proposed [2], [6], [7], [9], [10], [11], [12]. Most of them are heuristic methods to find a good linear decomposition of large index generation functions efficiently. However, devising an efficient exact optimization algorithm is not only academically but also practically significant. Although an exact optimization algorithm using a SAT solver, in which the problem is reduced to a SAT problem, has been proposed [11], [12], as far as we know, few dedicated algorithms for linear decomposition of index generation functions have been proposed.

Hence, in this paper, we propose an exact optimization algorithm based on a branch and bound method dedicated to linear decomposition of index generation functions. The proposed algorithm efficiently finds the optimum linear decomposition by pruning non-optimum solutions using effective branch and bound strategies. The branch strategy is based on an efficient heuristic using a balanced decision tree [2], and the bound is based on a lower bound on the number of variables for linear decomposition.

The rest of this paper is organized as follows: Section 2 defines index generation functions and linear decomposition. Section 3 formulates the minimization problem for the number of linear functions, derives a lower bound on the number of variables for linear decomposition, and shows our exact optimization algorithm to solve the problem. Section 4 shows experimental results using some benchmark index generation functions, and Section 5 concludes the paper.

2. Preliminaries

We briefly define index generation functions [4], [5] and their linear decompositions [1], [3], [7].

Definition 1. An **incompletely specified index generation function**, or simply **index generation function**, $f(x_1, x_2, \ldots, x_n)$ is a multi-valued function, where k assignments of values to binary variables $x_1, x_2, \ldots,$ and

TABLE 1. EXAMPLE OF INDEX GENERATION FUNCTION.

Registered vectors				indices
x_1	x_2	x_3	x_4	f
0	0	0	1	1
0	0	1	0	2
0	1	0	0	3
1	1	0	1	4

x_n map to $K = \{1,2,\ldots,k\}$. That is, the variables of f are binary-valued, while f is k-valued. Further, there is a one-to-one relationship between the k assignments of values to $x_1, x_2, \ldots$, and x_n and K. Other assignments are left unspecified. The k assignments of values to $x_1, x_2, \ldots$, and x_n are called a set of **registered vectors**. K is called a set of **indices**. $k = |K|$ is called **weight** of the index generation function f.

Example 1. Table 1 shows a 4-variable index generation function with weight four. Note that, in this function, input values other than 0001, 0010, 0100, and 1101 are NOT assigned to any function values. (End of Example)

Definition 2. Let $K = \{1,2,\ldots,k\}$ be a set of indices of an index generation function. If $K = S_1 \cup S_2 \cup \ldots \cup S_u$, each $S_i \neq \emptyset$, and $S_i \cap S_j = \emptyset$ $(i \neq j)$, then $\mathcal{P} = \{S_1, S_2, \ldots, S_u\}$ is a **partition** of the set of indices K. When all the subsets S_i are singletons (i.e., $|S_i| = 1$), $|\mathcal{P}| = |K| = k$.

An arbitrary n-variable index generation function with weight k can be realized by a $(2^n \times q)$-bit memory, where $q = \lceil \log_2(k+1) \rceil$. To reduce the memory size, linear decomposition is effective [7].

Definition 3. **Linear decomposition** of an index generation function $f(x_1, x_2, \ldots, x_n)$ is a representation of f using a general function $g(y_1, y_2, \ldots, y_p)$ and linear functions y_i:

$$y_i(x_1, x_2, \ldots, x_n) = a_{i1}x_1 \oplus a_{i2}x_2 \oplus \ldots \oplus a_{in}x_n \quad (i = 1,2,\ldots,p),$$

where $a_{ij} \in \{0,1\}$ $(j = 1,2,\ldots,n)$, and, for all registered vectors of the index generation function, the following holds:

$$f(x_1, x_2, \ldots, x_n) = g(y_1, y_2, \ldots, y_p).$$

Each y_i is called a **compound variable**. For each y_i, $\sum_{j=1}^{n} a_{ij}$ is called a **compound degree** of y_i, denoted by $deg(y_i)$, where a_{ij} is viewed as an integer, and $\sum$ is an integer sum.

Definition 4. An inverse function of a general function $z = g(y_1, y_2, \ldots, y_p)$ in a linear decomposition is a mapping from $K = \{1,2,\ldots,k\}$ to a set of p-bit vectors B^p, denoted by $g^{-1}(z)$. In this inverse function $g^{-1}(z)$, a mapping obtained by focusing only on the i-th bit of the p-bit vectors: $K \to \{0,1\}$ is called an **inverse function to a compound variable** y_i, denoted by $(g^{-1})_i(z)$.

Definition 5. Let $ON(y_i) = \{z \mid z \in K, (g^{-1})_i(z) = 1\}$, where $K = \{1,2,\ldots,k\}$ and $(g^{-1})_i(z)$ is an inverse function of $g(y_1, y_2, \ldots, y_n)$ to y_i. $|ON(y_i)|$ is called the **cardinality of** y_i or informally the **number of 1s included in** y_i.

TABLE 2. GENERAL FUNCTIONS g_1 AND g_2 IN LINEAR DECOMPOSITION OF f.

y_1	y_2	g_1	g_2
0	0	1	2
0	1	2	1
1	0	3	3
1	1	4	4

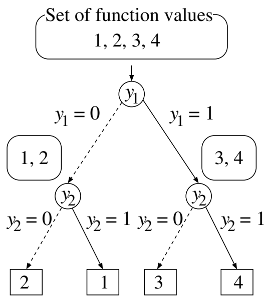

Figure 2. Point of view as a binary decision tree.

Example 2. The index generation function f in Example 1 can be represented by $y_1 = x_2$, $y_2 = x_1 \oplus x_3$, and $g_1(y_1, y_2)$ shown in Table 2. In this case, $deg(y_1) = 1$ and $deg(y_2) = 2$, respectively. f can be also represented by $y_1 = x_2$, $y_2 = x_4$, and $g_2(y_1, y_2)$ in the same table. In this case, both $deg(y_1)$ and $deg(y_2)$ are 1. In either case, f can be realized by the architecture in Fig. 1 with a $(2^2 \times 3)$-bit memory.

For $g_2(y_1, y_2)$ in Table 2, its inverse functions to y_1 and y_2 are $(g_2^{-1})_1(z)$ and $(g_2^{-1})_2(z)$, respectively. We have $(g_2^{-1})_1(2) = 0$, $(g_2^{-1})_1(1) = 0$, $(g_2^{-1})_1(3) = 1$, and $(g_2^{-1})_1(4) = 1$. Similarly, $(g_2^{-1})_2(2) = 0$, $(g_2^{-1})_2(1) = 1$, $(g_2^{-1})_2(3) = 0$, and $(g_2^{-1})_2(4) = 1$. The cardinalities of both y_1 and y_2 are 2. (End of Example)

In this way, by using linear decomposition, memory size needed to realize an index generation function can be reduced significantly. But, to realize a compound variable with compound degree d, $(d-1)$ 2-input EXOR gates are required. Thus, a lower compound degree is desirable when the memory size is equal.

3. Minimization of Number of Linear Functions

This section formulates the minimization problem of the number of linear functions, and presents an exact minimization algorithm to solve the problem.

3.1. Formulation of Minimization Problem

Since the architecture in Fig. 1 realizes an index generation function with EXOR gates, registers, multiplexers, and a $(2^p \times q)$-bit memory, to obtain an optimum realization of an index generation function, we have to solve the following problem:

Algorithm 1. Overview of the proposed algorithm

```
Input: an index generation function with weight $k$ and
an upper bound $t$ on compound degrees
Output: a set of compound variables and its size $h_{min}$
---
Let $\mathcal{P} = \{K\}, h = 0$, and iterate the following recursively.
min_search($\mathcal{P}, h$) {
    if ($|\mathcal{P}| = k$) { update_solution($h$); return; }
    if (bound_condition($\mathcal{P}, h$) is satisfied) return;
    branch($\mathcal{P}$, $t$, $h$);
}
```

Problem 1. Given an index generation function f and an integer t, find a linear decomposition of f such that the number of linear functions p is the minimum, and compound degrees are at most t.

The constraint on compound degrees t is given not only for reduction of solution space, but also for reduction of delay and area of the circuit L to realize linear functions.

Example 3. For linear decompositions of f in Example 2, the decomposition with $y_1 = x_2$, $y_2 = x_4$, and $g_2(y_1, y_2)$ is optimum when $t = 1$. (End of Example)

3.2. Exact Minimization Algorithm Based on Branch and Bound Method

Problem 1 can be considered as the problem of minimizing the height of a binary decision tree constructed by compound variables [2].

Example 4. Fig. 2 shows a binary decision tree of the smallest height that divides the set of indices into singletons by compound variables y_1 and y_2. This corresponds to g_2 in Table 2. (End of Example)

Thus, this subsection proposes an algorithm to find a binary decision tree with the smallest height. The proposed algorithm constructs binary decision trees in a top-down manner by selecting a compound variable one by one, and finds the best one by comparing their heights. As shown in Algorithm 1, it is based on a branch and bound method, and prunes clearly non-optimum solutions.

Algorithm 1 searches for a solution recursively while constructing a binary decision tree with height h. When $|\mathcal{P}| = k$ (i.e., the set of indices is partitioned into singletons), a solution (a set of h compound variables) is obtained. The procedure *update_solution*() compares the obtained solution with the current solution, and updates the current solution if the obtained one is better.

The procedure *branch*() explores the solution space by selecting a compound variable, and the procedure *bound_condition*() detects ineffective solution search and prunes it. Performance of branch and bound methods strongly depends on strategies for branch and bound. In the following subsections, we explain our strategies for branch and bound.

Algorithm 2. Overview of the branch process

```
Input: an index generation function, a partition of indices $\mathcal{P}$, an upper bound $t$ on compound degrees, and a
tree height $h$
Process: Search for solutions recursively by selecting a
compound variable
---
branch($\mathcal{P}$, $t$, $h$) {
    // $y$ is an already selected compound variable.
    for (each of $x_1, x_2, \ldots,$ and $x_n$)
        Compute $cost_1(\mathcal{P}, y \oplus x_i)$ and $cost_2(\mathcal{P}, y \oplus x_i)$;
    Sort $x_i$ in their ascending order;
    for (each $x_i$ in the order) {
        $y = y \oplus x_i$;
        if ($t > 1$) branch($\mathcal{P}$, $t - 1$, $h$);
        Divide each $S \in \mathcal{P}$ with $y$;
        min_search($\mathcal{P}, h + 1$);
        Combine each $S \in \mathcal{P}$ divided by $y$;
        $y = y \oplus x_i$;        // $x_i$ is taken out of $y$.
    }
}
```

3.2.1. Branch Strategy.

To prune non-optimal solutions efficiently, finding a good solution in an earlier stage is important. Thus, we select a good compound variable using the following cost function that has been proposed for our previous heuristic [2]:

$$cost_1(\mathcal{P}, y_i) = \sqrt{\sum_{S \in \mathcal{P}} \left(\frac{|S|}{2} - |S \cap ON(y_i)| \right)^2},$$

where $\mathcal{P}$ is a partition of a set of indices with already selected compound variables. In addition, when the cost functions of partitions by y_i are equal, the following cost function is used:

$$cost_2(\mathcal{P}, y_i) = \max_{S \in \mathcal{P}} \{\max\{|S \cap ON(y_i)|, |S \setminus ON(y_i)|\}\}$$

to select a compound variable that divides the largest subset into smaller subsets.

As mentioned before, Problem 1 can be considered as a minimization problem of the height of a binary decision tree, and thus, we have proposed the heuristic to produce a balanced tree using the cost functions. A compound variable y_i minimizing the cost functions tends to divide each subset into halves and be included in the optimum solution. Thus, the proposed exact minimization algorithm can find a good solution quickly by selecting compound variables in ascending order of values of the cost functions.

Algorithm 2 shows an overview of the branch process. It compounds original variables x_i recursively without overlapping a variable, and branches for solution search while dividing sets of indices with a compound variable y. After the branch is backtracked, sets divided by y are combined, and x_i is taken out of the compound variable y for the next search. In this branch strategy, compound variables with larger compound degree are searched prior to those with smaller compound degree. This is because compound

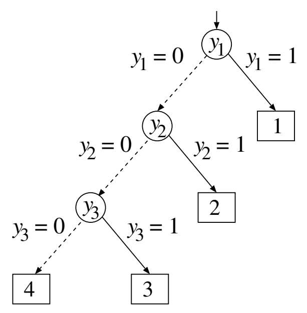

Figure 3. Imbalanced decision tree with height 3.

Algorithm 3. Upper bound on the number of 1s in a variable

```
Input: an index generation function with weight $k$ and
an upper bound $t$ on compound degrees
Output: the upper bound $c_{upper}$
-----
Sort $x_i$ in descending order of $|ON(x_i)|$;
$c_{upper} = 0$;
for (each of top-$t$ $x_i$'s in the order)
    $c_{upper} = c_{upper} + |ON(x_i)|$;
$c_{upper} = \min(c_{upper}, \frac{k}{2})$;
```

variables with larger compound degree tend to have smaller values of the cost functions [2].

3.2.2. Bound Strategy.

To determine whether the current solution search can be pruned or not, we need a good lower bound on the number of compound variables. The lower bound $\lceil \log_2(m) \rceil$ shown in [5] is a good estimate of the number of compound variables needed to divide a set with m indices into m singletons when the number of 1s in compound variables is $\frac{m}{2}$. However, when the number of 1s in compound variables is much smaller, a better (larger) lower bound is possible.

Example 5. Since both y_1 and y_2 in Example 2 have two 1s that are a half number of indices 4, a balanced tree with height $\log_2(4) = 2$ shown in Fig. 2 is obtained. On the other hand, when each compound variable has only one 1, an imbalanced tree with height 3 like Fig. 3 is obtained. (End of Example)

To better estimate the number of compound variables, in such a case, we use the following lower bound:

Theorem 1. Let m be the number of indices in a set, and c be the number of 1s in compound variables. When $c < \frac{m}{2}$, at least

$$lower(m,c) = \left\lfloor \frac{m}{c} \right\rfloor + \lceil \log_2(c) \rceil - 1$$

compound variables are needed to divide the set into m singletons.

(Proof) See Appendix-A.

To estimate the number of compound variables using this lower bound, a good estimate for the number of 1s in compound variables is needed as well. But, when the estimate for the number of 1s is smaller than the actual number of 1s, the lower bound in Theorem 1 does not hold even if the estimate is close to the actual number. In that case, the optimum solution can be missed. Thus, an upper bound on the number of 1s in compound variables is needed.

Algorithm 4. Overview of the bound process

```
Input: a partition of indices $\mathcal{P}$, a current tree height $h$,
the minimum tree height so far $h_{min}$
Output: the bound condition is satisfied or not
-----
bound_condition($\mathcal{P}$, $h$) {
    $m = \max_{S \in \mathcal{P}}\{|S|\}$;
    if ($2 \times c_{upper} < m$) {
        if ($h_{min} \leq h + \max(lower(m, c_{upper}), \lceil \log_2(m) \rceil)$)
            return satisfied;
    }
    else {
        if ($h_{min} \leq h + \lceil \log_2(m) \rceil$)
            return satisfied;
    }
    return unsatisfied;
}
```

Since a compound variable is produced by applying the EXOR operation to some original variables x_i, the number of 1s in a compound variable does not exceed the sum of the number of 1s in x_i's. When t original variables are compounded, the upper bound on the number of 1s in a compound variable is obtained by summing the 1s in t original variables in descending order of $|ON(x_i)|$, as shown in Algorithm 3. If the sum of $|ON(x_i)|$ exceeds $\frac{k}{2}$, the upper bound is held down to $\frac{k}{2}$. This is because $\frac{k}{2}$ is the best number to divide the set of indices into halves, and the smallest number of compound variables $\lceil \log_2(k) \rceil$ can be achieved in that case. We run Algorithm 3 only once just after registered vectors of an index generation function are read in.

Algorithm 4 shows an overview of the bound process based on the lower bound in Theorem 1. As search in the branch and bound method proceeds, m in Algorithm 4 gets smaller, and thus, $\lceil \log_2(m) \rceil$ tends to be applied as a lower bound. However, at earlier stages of solution search, the larger lower bound $lower(m, c_{upper})$ is applied. Thus, unpromising solutions are pruned at earlier stages by using this strategy, resulting in significant reduction of the search space. When both m and c are small (e.g., $m = 5$ and $c = 2$), $\lceil \log_2(m) \rceil$ can be larger than $lower(m,c)$. Even in such a case, we prune the search space efficiently by choosing the larger bound.

4. Experimental Results

The proposed exact minimization algorithm is implemented in the C language, and run on the following computer environment: CPU: Intel Core2 Quad Q6600 2.4GHz, memory: 4GB, OS: CentOS 5.7, and C-compiler: gcc -O2 (version 4.1.2).

TABLE 3. COMPARISON OF B&B METHODS IN TERMS OF SEARCH SPACE.

	Compound degrees				
	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$
Solutions h_{min}	9	6	5	4	4
method 1)	293,909	2,101,815	153,081	91	19
method 2)	262,303	1,975,994	**151,773**	**4**	**4**
method 3)	**9**	2,101,173	153,081	91	19
method 4)	**9**	**1,975,364**	**151,773**	**4**	**4**

TABLE 4. COMPUTATION TIME IN SECONDS OF B&B METHODS FOR 1-OUT-OF-10.

	Compound degrees				
	$t = 1$	$t = 2$	$t = 3$	$t = 4$	$t = 5$
method 1)	2.09	**209.24**	113.13	0.55	0.58
method 2)	2.18	231.37	130.38	**0.02**	**0.14**
method 3)	<**0.01***	213.63	**109.65**	0.55	0.60
method 4)	<**0.01***	216.62	127.32	**0.02**	**0.14**

*They were shorter than 1 msec., but could not be obtained precisely due to precision of the timer.

4.1. On Effectiveness of Branch and Bound Strategies

To evaluate the effectiveness of the proposed branch and bound strategies, we compare search space in the following four combinations of strategies:

1) Branch: natural order of original variables (i.e., $x_1, x_2, \ldots, x_n$).
   Bound: only $\lceil \log_2(m) \rceil$.
2) Branch: ascending order of $cost_1$ and $cost_2$.
   Bound: only $\lceil \log_2(m) \rceil$.
3) Branch: natural order of original variables.
   Bound: Theorem 1 and $\lceil \log_2(m) \rceil$.
4) Branch: ascending order of $cost_1$ and $cost_2$.
   Bound: Theorem 1 and $\lceil \log_2(m) \rceil$.

The first one uses naive strategies for both branch and bound, the second one uses only the proposed strategy for branch, the third one uses only the proposed strategy for bound, and the fourth one uses the proposed strategies for both branch and bound. Table 3 shows the number of times that the procedure *branch()* is invoked in each branch and bound method for a benchmark "1-out-of-10 code" with weight $k = 10$ shown in [7].

As shown in Table 3, the proposed branch strategy works effectively when t is large. On the other hand, the proposed bound strategy works effectively when t is small. And, by taking advantages of both the strategies, we can get a bigger effect. In the table, the number of times that branch() is invoked in the proposed algorithm (method 4) is equal to the optimum solution h_{min} when $t = 1, 4$, and 5. This means that the optimum solution is found by the first search, and all other searches are pruned. This also means that our balanced tree based heuristic [2] finds the optimum solutions in those cases since the branch strategy is based on the heuristic.

In fact, the heuristic [2] finds the optimum solutions for this function when $t = 1$ to 5 except for $t = 2$. When $t = 2$, it produces 7 compound variables while the smallest number of compound variables is 6. From these results, we can see that for $t = 2$, it is hard to find the optimum solution, and the heuristic still has room for improvement.

4.2. On Computation Time

Although the proposed strategies reduce search space significantly as shown in Table 3, its benefit can be canceled out if computational overheads of the strategies are large. To show that the overheads are small and reduction of search space leads to shortening of computation time, we compare computation time of the four methods shown in the previous subsection. Table 4 shows computation time, in seconds, of the four methods for the same benchmark "1-out-of-10 code".

When $t = 1, 4$, and 5, the method 4) finds the optimum solution in the shortest computation time among the four methods because the proposed strategies significantly reduce search space. As shown in Table 3, the method 4) reduces search space of the method 1) by about 96% and 79%, respectively when $t = 4$ and 5. Similarly, computation time is also reduced by about 96% and 75%, respectively when $t = 4$ and 5. This means that computational overheads of the strategies are small. However, when $t = 2$ and 3, the method 4) requires longer computation time because of the overheads of the strategies. This is not because the overheads are large, but because the method 4) reduces search space of the method 1) by only about 6% and 1%, respectively when $t = 2$ and 3.

Table 5 shows computation time, in seconds, of the proposed method for other benchmarks. Since the time complexity of the proposed method is an exponential function of the number of original variables n, we could not obtain optimum solutions in many cases when n is larger than 10. However, when the proposed strategies work effectively and reduce search space significantly, the proposed method can find optimum solutions quickly even for such larger n.

5. Conclusion and Comments

This paper proposes a branch and bound method to exactly minimize the number of compound variables for linear decomposition of index generation functions. By taking advantages of techniques used in the balanced tree based heuristic, the proposed method efficiently reduces search space of solutions with small overheads, and finds the optimum solution quickly. Experimental results show that the proposed method has potential to find optimum solutions with reasonable computation time.

As shown in Table 3, search space is not reduced significantly by the proposed method when $t = 2$ and $t = 3$. In these cases, the proposed strategies could not be very efficient. We will analyze these cases in more detail, and improve the bound and branch strategies as our future work. In addition, we will study on optimum scheduling of compound degrees. In the current method, compound variables with the largest compound degree are searched first. However,

TABLE 5. COMPUTATION TIME IN SECONDS OF THE PROPOSED METHOD FOR OTHER BENCHMARKS.

	$t=1$		$t=2$		$t=3$		$t=4$		$t=5$	
Benchmarks	h_{min} (space)	Time	h_{min} (space)	Time	h_{min} (space)	Time	h_{min} (space)	Time	h_{min} (space)	Time
1-out-of-10	9 (9)	<0.01*	6 (1,975,364)	216.62	5 (151,773)	127.32	4 (4)	0.02	4 (4)	0.14
1-out-of-12	11 (11)	<0.01*	–	–	–	–	–	–	4 (4)	0.46
1-out-of-16	15 (15)	<0.01*	–	–	–	–	–	–	5 (5)	3.68
2-out-of-16	–	–	–	–	–	–	8 (8)	2.76	8 (9)	36.32
3-out-of-16	–	–	–	–	–	–	10 (10)	16.17	–	–

*They were shorter than 1 msec., but could not be obtained precisely due to precision of the timer.
–: We quit the computation because it took more than $3,600$ seconds.

this way is not always optimum. By changing the search order of compound degrees, more efficient branch can be expected.

Acknowledgments

This research is partly supported by the JSPS KAKENHI Grant (C), No.16K00079. The reviewers' comments were helpful in improving the paper.

References

[1] R. J. Lechner, "Harmonic analysis of switching functions," in A. Mukhopadhyay (ed.), *Recent Developments in Switching Theory*, Academic Press, New York, Chapter V, pp. 121–228, 1971.

[2] S. Nagayama, T. Sasao, and J. T. Butler, "An efficient heuristic for linear decomposition of index generation functions," *46th International Symposium on Multiple-Valued Logic*, pp. 96–101, May, 2016.

[3] E. I. Nechiporuk, "On the synthesis of networks using linear transformations of variables," *Dokl, AN SSSR*, Vol. 123, No. 4, pp. 610–612, Dec., 1958.

[4] T. Sasao, *Memory-Based Logic Synthesis*, Springer, 2011.

[5] T. Sasao, "Index generation functions: recent developments (invited paper)," *41st International Symposium on Multiple-Valued Logic*, pp. 1–9, May 2011.

[6] T. Sasao, "Linear transformations for variable reduction," *Reed-Muller Workshop 2011*, May 2011.

[7] T. Sasao, "Linear decomposition of index generation functions," *17th Asia and South Pacific Design Automation Conference*, pp. 781–788, Jan. 2012.

[8] T. Sasao, Y. Urano, and Y. Iguchi, "A lower bound on the number of variables to represent incompletely specified index generation functions," *44th International Symposium on Multiple-Valued Logic*, pp. 7–12, May 2014.

[9] T. Sasao, Y. Urano, and Y. Iguchi, "A method to find linear decompositions for incompletely specified index generation functions using difference matrix," *IEICE Transactions on Fundamentals*, Vol. E97-A, No. 12, pp. 2427–2433, Dec. 2014.

[10] T. Sasao, "A reduction method for the number of variables to represent index generation functions: s-min method," *45th International Symposium on Multiple-Valued Logic*, pp. 164–169, May 2015.

[11] T. Sasao, I. Fumishi, and Y. Iguchi, "A method to minimize variables for incompletely specified index generation functions using a SAT solver," *International Workshop on Logic and Synthesis*, pp. 161–167, June 2015.

[12] T. Sasao, I. Fumishi, and Y. Iguchi, "On an exact minimization of variables for incompletely specified index generation functions using SAT," *Note on Multiple-Valued Logic in Japan*, Vol.38, No.3, pp. 1–8, Sept. 2015.

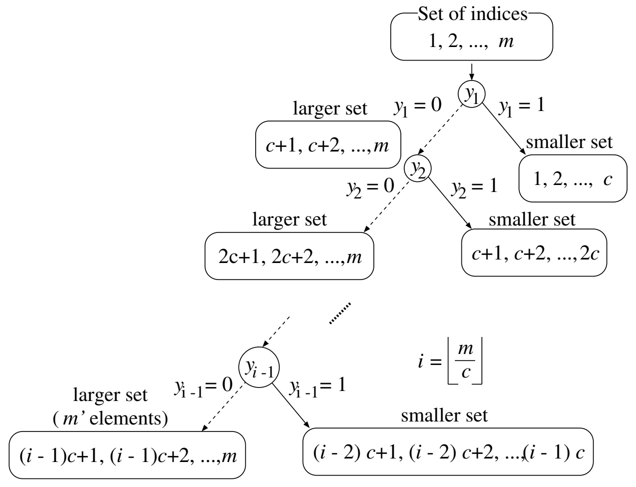

Figure A.1. Unbalanced divisions of a set of indices.

Appendix

1. Proof for Theorem 1

When $o < \frac{m}{2}$, a set of indices is divided into unbalanced subsets, as shown in Fig. A.1. This unbalanced division is iterated until the size of larger subset is smaller than $2 \times o$. In this iteration, at least

$$\left\lfloor \frac{m}{c} \right\rfloor - 1 \qquad \text{(A.1)}$$

compound variables are required. This is an ideal case where only a larger subset in each division is divided by separating c indices from the subset. However, smaller subsets have to be also divided to finally obtain singletons of indices. Thus, more compound variables are actually required.

After the unbalanced divisions, the larger subset has m' indices, where m' is an integer satisfying the following:

$$c \leq m' < 2 \times c.$$

Thus, we have the following relation:

$$\lceil \log_2(c) \rceil \leq \lceil \log_2(m') \rceil. \qquad \text{(A.2)}$$

By summing (A.1) and the left side of (A.2), we have the lower bound

$$\left\lfloor \frac{m}{c} \right\rfloor + \lceil \log_2(c) \rceil - 1$$

∎