A Dynamic Programming Based Method for Optimum 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*—The problem addressed in this paper is minimization of the number of linear functions in a linear decomposition. This paper proposes an exact minimization method based on dynamic programming for index generation functions. The proposed method searches for an optimum solution while recursively dividing an index set of a given index generation function. To use partial solutions efficiently in solution search, the proposed method represents partitions of an index set compactly and uniquely by zero-suppressed binary decision diagrams (ZDDs). Existing methods based on a branch-and-bound approach search for a solution sequentially in a depth-first manner. On the other hand, the proposed method searches for multiple partial solutions in parallel in a breadth-first manner. Thus, once a solution is found, we can terminate the search process. This is because the depth of searches corresponds to the number of linear functions. Experimental results using benchmark index generation functions show the effectiveness of the proposed method.**

***Keywords*-Index generation functions; functional decomposition; linear decomposition; zero-suppressed binary decision diagrams; logic design; dynamic programming method.**

I. Introduction

Index generation functions [7], [8] are multiple-valued functions that map certain assignments of values to variables (called registered vectors) to unique multiple-valued indices. Such functions have a wide range of applications, including computer viruses detection and packet classification, since the functions can be used to check whether an input vector matches a registered one. In these index search applications, indices to be searched, such as virus patterns and rules for classification, are frequently updated. Thus, quick updating of functions, as well as high processing speed, is required. To satisfy the design requirements, memory based designs of index generation functions show promise [7].

Among the designs, one using *linear decomposition* of index generation functions [10] is efficient. Linear decomposition [1], [6] decomposes an index generation function $f(x_1, x_2, \ldots, x_n)$ into two parts: L and G, as shown in Fig. 1. The first part L produces linear functions y_i $(i = 1, 2, \ldots, p)$ from inputs $x_1, x_2, \ldots, x_n$ of f. Then, the second part G generates a function value (i.e., an index) of f from the p linear functions y_i. L is a programmable circuit [10] consisting of EXOR gates, registers, and multiplexers. G is a $(2^p \times q)$-bit memory, where q is the number of bits needed to represent indices of f.

Since the memory size of G grows exponentially with p, the number of linear functions, minimization of p is important. To minimize the number of linear functions in a linear decomposition of index generation functions, various methods [3], [4], [5], [9], [10], [12], [13], [14], [15] have been proposed. Among them, this paper focuses on exact minimization methods [4], [5], [14], [15]. Although heuristic methods are more scalable, devising an efficient exact minimization method is not only academically but also practically significant. This is because it becomes a basis for evaluating the quality of solutions produced by heuristic methods.

Since the methods proposed in [14], [15] reduce the linear decomposition problem to a SAT problem, and solve it using a SAT solver, they no longer have much room for improvement unless the SAT solver is improved. On the other hand, the methods proposed in [4], [5] directly solve the linear decomposition problem based on a branch-and-bound approach. However, they are still slow. This is because, in a branch-and-bound approach, a solution is sequentially searched in a depth-first manner.

This paper proposes an exact minimization method based on dynamic programming. Since the proposed method searches for multiple partial solutions in parallel in a breadth-first manner, we can terminate the search process once a solution is found. By representing a characteristic of partial solutions compactly and uniquely using a zero-suppressed binary decision diagram (ZDD), partial solutions can be classified into *equivalence classes* with their charac-

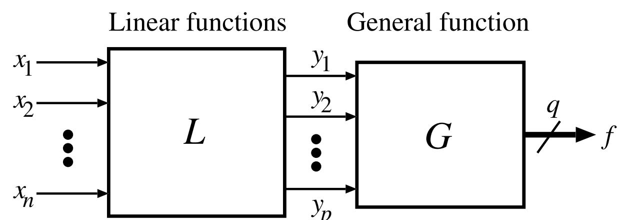


Figure 1. Linear decomposition of an index generation function [10].

Table I
EXAMPLE OF INDEX GENERATION FUNCTION [4].

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

teristics, and thus, the proposed method can search for an exact solution efficiently.

The rest of this paper is organized as follows: Section II defines index generation functions and linear decomposition. Section III formulates the minimization problem for the number of linear functions, provides a brief overview of the existing exact minimization methods based on branch-and-bound approach [4], [5], and proposes a dynamic programming based method using ZDDs. Section IV shows experimental results using some benchmark index generation functions, and Section V concludes the paper.

II. PRELIMINARIES

A. Index Generation Functions and Linear Decomposition

We briefly define index generation functions [7], [8] and their linear decompositions [1], [6], [10].

Definition 1: An **incompletely specified index generation function**, or simply **index generation function**, $f(x_1, x_2, \ldots, x_n)$ is a multiple-valued function, where k assignments of values to binary variables $x_1, x_2, \ldots, x_n$ 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 $x_1, x_2, \ldots, x_n$ and K. Other assignments are left unspecified. The k assignments of values to $x_1, x_2, \ldots, x_n$ are called the **registered vectors**. K is called the set of **indices**. $k = |K|$ is called the **weight** of the index generation function f.

Example 1: Table I 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. ∎

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 [10].

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:

$$y_i(x_1, x_2, \ldots, x_n) = a_{i1}x_1 \oplus a_{i2}x_2 \oplus \ldots \oplus a_{in}x_n,$$

Table II
GENERAL FUNCTIONS g_1 AND g_2 IN LINEAR DECOMPOSITION OF f [4].

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

where $i \in \{1, 2, \ldots, p\}$, $a_{ij} \in \{0, 1\}$ $(j \in \{1, 2, \ldots, n\})$, 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 $\sum$ as an integer sum.

Example 2: The index generation function f in Example 1 can be represented using two linear functions $y_1 = x_2$ and $y_2 = x_1 \oplus x_3$, and a function $g_1(y_1, y_2)$ shown in Table II. In this case, $deg(y_1) = 1$ and $deg(y_2) = 2$. 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, while a $(2^4 \times 3)$-bit memory is needed to directly realize f without linear decomposition. ∎

In this way, linear decomposition can significantly reduce memory size needed to realize an index generation function. But, in linear decomposition, not only memory, but also EXOR gates, registers, and multiplexers are required to realize a compound variable (i.e., block L in Fig. 1). Since the circuit size of L depends on compound degrees, lower compound degrees are desirable when the memory size is equal.

B. Zero-suppressed Binary Decision Diagrams (ZDDs)

In this subsection, we briefly define ZDDs [2].

Definition 4: A ***zero-suppressed binary decision diagram (ZDD)*** is a rooted directed acyclic graph (DAG) representing a logic function. It consists of two terminal nodes representing function values 0 and 1, and nonterminal nodes representing input variables. Each nonterminal node has two outgoing edges, a 0-edge and a 1-edge, that correspond to the values of the input variables. Neither terminal node has outgoing edges.

A ZDD is obtained by repeatedly applying the Shannon expansion $f = \overline{x_i}f_0 \vee x_i f_1$ to a logic function, where $f_0 = f(0 \to x_i)$, and $f_1 = f(1 \to x_i)$, and by applying the following two reduction rules:

1) Coalesce equivalent sub-graphs.
2) Delete nonterminal nodes v whose 1-edge points to the terminal node representing 0, and redirect edges pointing to v to its child node u that is pointed by the 0-edge of v.

As is well known, ZDDs can represent sets of combinations compactly and uniquely by using characteristic

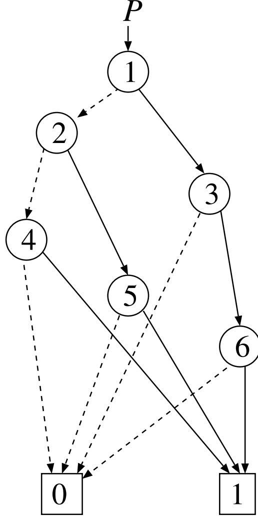


Figure 2. [5] ZDD for $\mathcal{P} = \{\{1,3,6\},\{2,5\},\{4\}\}$.

functions of sets [2]. Similarly, a partition of an index set $\mathcal{P} = \{S_1, S_2, \ldots, S_u\}$ can be also represented compactly and uniquely using a ZDD. In ZDDs for $\mathcal{P}$, each node corresponds to an index, and its 1-edge and 0-edge represent whether the index is included in a subset S_i, or not. A 1-path in a ZDD from the root node to the terminal node 1 represents a subset S_i.

Example 3: Let an index set be $K = \{1,2,3,4,5,6\}$, and a partition of K be $\mathcal{P} = \{\{1,3,6\},\{2,5\},\{4\}\}$. Fig. 2 shows a ZDD for $\mathcal{P}$. In Fig. 2, dashed lines and solid lines denote 0-edges and 1-edges, respectively. The number of nonterminal nodes is 6. □

Theorem 1: [5] Let an index set be $K = \{1,2,\ldots,k\}$. For any partition $\mathcal{P}$ of K, the number of nonterminal nodes in a ZDD for $\mathcal{P}$ is k, regardless of the order of indices.

III. Optimization of Linear Decompositions

This section formulates the minimization problem of the number of linear functions, and shows exact minimization methods to solve the problem.

A. Formulation of Minimization Problem

The linear decomposition problem of index generation functions is formulated as follows:

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 minimum, and compound degrees are at most t.

By solving this problem, we can realize f using the architecture in Fig. 1 with the minimum memory size. The architecture has not only a memory, but also EXOR gates, registers, and multiplexers for the circuit L realizing the linear functions. To obtain an optimum design considering balance of the circuit size of L and the memory size, the compound degree t is given as the constraint of the problem.

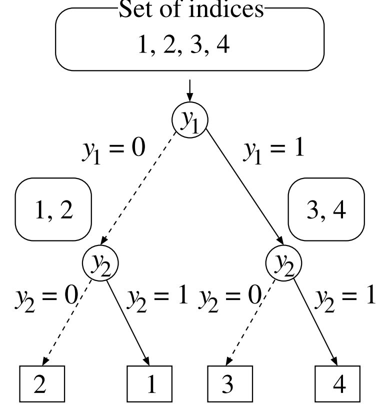


Figure 3. Binary decision tree for g_2 of Table II [5].

Example 4: For the index generation function f in Example 2, the linear decomposition with $y_1 = x_2$, $y_2 = x_4$, and $g_2(y_1, y_2)$ is optimum when $t = 1$. □

B. Existing Methods Based on a Branch-&-Bound Approach

Problem 1 can be considered as a tree height minimization problem of a binary decision tree that recursively divides a set of indices into singletons by compound variables [3].

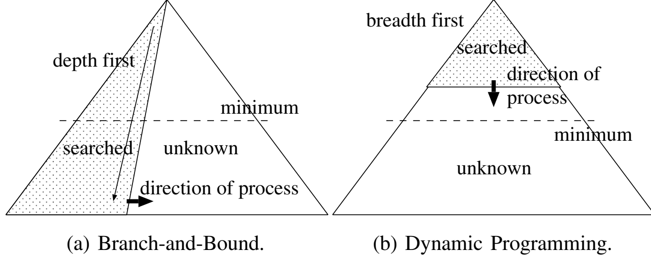
Example 5: Fig. 3 shows a binary decision tree representing g_2 in Table II. The tree divides the set of indices into singletons by compound variables y_1 and y_2. The tree height corresponds to the number of compound variables. □

Existing methods [4], [5] search for a binary decision tree with the smallest height by a branch-and-bound approach. A binary decision tree is constructed in a top-down manner while dividing a given set of indices recursively by selecting a compound variable one by one. Since the methods are based on a branch-and-bound approach, trees are constructed sequentially in a depth-first manner to find the smallest height tree.

The method [5] uses ZDDs to prune redundant solution searches. In this method, ZDDs represent partitions of indices, and ZDDs are stored along with the smallest number of compound variables needed to make partitions of indices into singletons. ZDDs are used to quickly check whether partitions have been already searched or not. If a partition has been previously searched, the stored number of compound variables is used as a partial solution to avoid search of the same solution.

C. Proposed Method Based on Dynamic Programming

Since existing methods [4], [5] are based on a branch-and-bound approach, a solution (a combination of compound variables) is sequentially searched in a depth-first manner, as shown in Fig. 4(a). Even if a minimum solution has been obtained at an early stage of search, the search process cannot be terminated because we cannot decide whether it is minimum or not unless all the solutions are searched. In addition, during solution search, the same partition of



(a) Branch-and-Bound. (b) Dynamic Programming.

Figure 4. Overviews of solution search approaches.

indices tends to reappear [5], and thus, the existing methods are slow.

To search for an optimum solution more efficiently, we propose an exact optimization method based on a dynamic programming approach. In dynamic programming approaches, partial solutions are usually stored to solve a problem, but the proposed method stores *partitions of indices*, instead of partial solutions. This is because partitions of indices are a characteristic representing partial solutions, and multiple partial solutions (combinations of compound variables) tend to have the same partition of indices. By merging multiple partial solutions into a single partition of indices, partial solutions can be classified into *equivalence classes*, and thus, the search space for a solution can be reduced.

The proposed method searches for multiple partial solutions in parallel in a breadth-first manner, as shown in Fig. 4(b), by adding a compound variable in turn to divide a set of indices. Once a solution is found (a set of indices is divided into singletons). we can terminate the search process since it is clear that the number of compound variables is the smallest to reach the solution.

Algorithm 1 shows the overview of the proposed method based on the dynamic programming approach using ZDDs. In the algorithm, C denotes a set of partitions of indices, and its element $\mathcal{P}$ is represented by a ZDD. By updating C while selecting an *irredundant compound variable*[*] in turn, the algorithm searches for a solution. The procedure *divide_sets*() divides subsets of indices S_i in $\mathcal{P} = \{S_1, S_2, \ldots, S_u\}$ by a compound variable y using ZDDs. It traverses all the 1-paths in a ZDD for $\mathcal{P}$, and constructs another ZDD for $\mathcal{P}'$ using y, *Change*, and *Union* operations that are basic operations in a ZDD package [2]. The time complexity of the procedure strongly depends on the number of paths in a ZDD.

Algorithm 1: Overview of the proposed method

```
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
-------------------------------------------------------
min_DPsearch( ) {
  Let an initial set of partitions be C = {P},
   where P = {K};
  for (h = 1; a solution has not been obtained; h++) {
    for (each P ∈ C: current set of partitions) {
      for (each irredundant compound variable y) {
        P' = divide_sets(P, y);
        N = store_as_next(P', y, h);
      }
    }
    Update the set of partitions C with N;
  }
  return h − 1 and a set of y's as the solution;
}
```

Theorem 2: Let an index set be $K = \{1, 2, \ldots, k\}$. For any partition $\mathcal{P}$ of K, the number of paths in a ZDD for $\mathcal{P}$ is exactly $k+1$, regardless of the order of indices.

Proof: From Theorem 1, the number of nonterminal nodes is k, and no nonterminal nodes are not shared. Thus, a ZDD for $\mathcal{P}$ can be considered as a binary tree if terminal nodes are not shared. In binary trees, the number of leaves (terminal nodes) corresponds to the number of paths. Since the number of leaves is $k+1$ when the number of branch nodes (nonterminal nodes) is k, we have the theorem. ∎

Thus, the time complexity of *divide_sets*() is $O(k)$. Therefore, it is fast.

The procedure *store_as_next*() screens the obtained partition $\mathcal{P}'$ to store it as a set of partitions for the next stage of search. Partitions $\mathcal{P}'$ are discarded if one of the following two conditions holds:

1) $\mathcal{P}'$ has already appeared.
2) Sum of h and the *lower bound* on the number of compound variables needed to divide $\mathcal{P}'$ into singletons is not smaller than the *upper bound* on the minimum solution.

1) Since partitions are represented by ZDDs, this checking (equivalence checking) is performed on ZDDs. It is what ZDDs do best, and is done in $O(1)$ time.

2) To discard ineffective partitions as much as possible, we use Theorem 3 as the *lower bound*, and a result of the heuristic method [4] as the *upper bound*. That is, in the proposed method, the heuristic method and then the dynamic programming approach are applied to the problem.

Theorem 3: [4] 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.

Although a key issue of dynamic programming approaches is the *space complexity*, the proposed method overcomes the issue by discarding ineffective partial solutions and by using ZDDs to represent partitions of indices compactly.

[*]A compound varible is redundant for solution search if it has been already used to partition of indices, or it is equivalent to another one due to the commutative law of EXOR.

Table III
COMPUTATION TIME OF METHODS (IN SECONDS).

Benchmark functions	t	h_{heu}	h_{min}	Existing [4]	Existing [5]	Proposed
1-out-of-10 ($k=10$)	1	9	9	*<0.01	*<0.01	*<0.01
	2	6	6	216.62	0.84	**0.10**
	3	5	5	127.62	1.12	**0.18**
	4	4	4	0.02	*<0.01	*<0.01
	5	4	4	0.14	*<0.01	*<0.01
1-out-of-12 ($k=12$)	1	11	11	*<0.01	*<0.01	*<0.01
	4	5	5	†	18.34	**1.73**
1-out-of-16 ($k=16$)	1	15	15	*<0.01	*<0.01	*<0.01
	5	5	5	3.68	0.18	**0.02**
2-out-of-16 ($k=120$)	4	8	8	2.76	5.24	**0.09**
	5	9	8	36.32	14.13	†
3-out-of-16 ($k=560$)	4	10	10	16.17	207.65	**0.56**

h_{heu}: results obtained by the heuristic [3].
* Time is less than 0.01 sec..
† Computation was terminated when it exceeded one hour.

Table IV
TOTAL NUMBER OF NONTERMINAL NODES IN ZDDS.

Benchmark functions	t	Existing [5] No. of ZDDs	Existing [5] No. of nodes	Proposed No. of ZDDs	Proposed No. of nodes
1-out-of-10	1	9	59	**1**	**10**
	2	5,310	11,564	**5,261**	**11,369**
	3	2,268	4,765	**2,266**	**4,751**
	4	4	36	**1**	**10**
	5	4	38	**1**	**10**
1-out-of-12	1	11	84	**1**	**12**
	4	6,274	13,451	**6,271**	**13,429**
1-out-of-16	1	15	144	**1**	**16**
	5	5	67	**1**	**16**
2-out-of-16	4	8	939	**1**	**120**
	5	9	1,069	†	†
3-out-of-16	4	10	5,595	**1**	**560**

† Computation was terminated when it exceeded one hour.

IV. EXPERIMENTAL RESULTS

The proposed exact minimization method 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 Linux, and C-compiler: gcc -O2 (version 4.1.2).

A. On Computation Time

To evaluate the efficiency of the proposed method, we compare the proposed method with the existing methods [4], [5], in terms of computation time. Table III shows computation time, in seconds, of each method for some benchmark index generation functions shown in [10]. We used the same benchmark functions as in [4], [5]. The bold values in Table III show where the proposed method significantly outperforms the other two methods.

As shown in Table III, the computation time of the proposed method is a few orders of magnitude shorter than computation time of the existing ones. In particular, for "2-out-of-16" with $t = 4$ and "3-out-of-16" with $t = 4$, the proposed method is much faster than the existing ones. From these results, we can see that the proposed method yields a significant effect on reduction of the redundant solution searching time. Further, overhead of the dynamic programing approach and ZDD operations are negligible small.

However, for "2-out-of-16" with $t = 5$, the proposed method could not obtain the optimum solution in a reasonable computation time due to the large search space. In this case, the heuristic method [3] does not produce the optimum result for this function, and the proposed method cannot prune partial solutions sufficiently.

B. On Number of Nonterminal Nodes

Table IV shows the number of ZDDs and the total number of nonterminal nodes in ZDDs during a solution search of the existing method [5] and the proposed method. These results show the space (memory size) complexity of the methods. Note that the "No. of nodes" columns do not include unused nodes after ZDD operations (that is, Table IV shows the number of nodes after *garbage collection* is applied).

As shown in Table III, for almost all functions, the heuristic method [3] finds the optimum solution. Thus, the proposed dynamic programming approach is used only to verify that there exists no other result better than the result obtained by the heuristic method. For functions where the number of ZDDs is 1 in the column "No. of ZDDs" under "Proposed" in Table IV, only a ZDD for the initial partition $\mathcal{P} = \{K\}$ is constructed, and any other partitions are pruned. For other functions, ZDDs other than the initial one are constructed, but the proposed method produces fewer ZDDs than the existing method. This is because the proposed method merges partial solutions since it uses the same lower bound and upper bound as the existing method.

From these results, we can see that the space complexity of the proposed method is as low as that of the existing method based on a branch-and-bound approach.

C. For Randomly Generated Functions

To investigate the efficiency of the proposed method further, index generation functions were randomly generated using *k social security and tax numbers* (SST numbers) with different number of digits and different k [3]. Table V shows the results for random SST numbers.

When the heuristic used in the proposed method finds a good initial solution, the proposed method reduces the search space of solutions significantly, and finds an optimum solution quickly even for randomly generated functions with many variables. However, when the number of indices is large, the proposed method tends to require long computation time, as shown in Table V. This is because the number of different partitions of indices becomes large as the number of indices becomes large. Since the proposed dynamic programming approach stores *partitions of indices* instead of partial solutions, its space and time complexities depend on the number of different partitions more strongly than the number of variables.

Improving the heuristic method and the lower bound in the proposed method would be helpful to find optimum

Table V
RESULTS FOR RANDOM SST NUMBERS.

Random SST numbers	t	h_{heu}	h_{min}	No. of ZDDs	No. of nodes	Time (msec.)
3-digit SST	1	6	5	1,684	268	10
($n = 12$)	2	5	5	157	1,013	20
($k = 15$)	3	5	4	7,283	33,029	1,100
	4	4	4	1	15	*<10
	5	4	4	1	20	*<10
5-digit SST	1	5	5	13	146	*<10
($n = 20$)	2	5	4	1,700	8,605	260
($k = 15$)	3	4	4	1	15	*<10
	4	4	4	1	15	20
	5	4	4	1	15	70
7-digit SST	1	5	5	16	173	*<10
($n = 28$)	2	5	4	13,543	61,119	2,020
($k = 15$)	3	4	4	1	15	10
	4	4	4	1	15	50
	5	4	4	1	15	250
3-digit SST	1	5	5	1	20	*<10
($n = 12$)	2	6	5	27,460	201,033	3,260
($k = 20$)	3	6	†	†	†	†
	4	6	†	†	†	†
	5	5	5	1	20	10
5-digit SST	1	6	5	348	3,097	10
($n = 20$)	2	5	5	1	20	10
($k = 20$)	3	5	5	1	20	10
	4	5	5	1	20	30
	5	5	5	1	20	120
7-digit SST	1	6	5	825	6,739	40
($n = 28$)	2	5	5	1	20	10
($k = 20$)	3	5	5	1	20	10
	4	5	5	1	20	100
	5	5	5	1	20	550
3-digit SST	1	7	6	505	5,442	10
($n = 12$)	2	6	5	2,478	22,957	580
($k = 25$)	3	6	†	†	†	†
	4	6	†	†	†	†
	5	6	†	†	†	†
5-digit SST	1	6	6	84	1,253	*<10
($n = 20$)	2	6	†	†	†	†
($k = 25$)	3	6	†	†	†	†
	4	6	†	†	†	†
	5	6	†	†	†	†
7-digit SST	1	6	6	188	2,493	10
($n = 28$)	2	6	†	†	†	†
($k = 25$)	3	6	†	†	†	†
	4	6	†	†	†	†
	5	6	†	†	†	†

h_{heu}: results obtained by the heuristic [3].
* Time is less than 10 msec..
† Computation was terminated when it exceeded one hour.

solutions for larger index generation functions. That is our future work.

V. CONCLUSION AND COMMENTS

This paper proposes an exact optimization method based on dynamic programming using ZDDs for the linear decomposition problem of index generation functions. The proposed method reduces search space for a solution by storing *partitions of indices* and merging multiple partial solutions. Since the proposed method searches for partial solutions in parallel in a breadth-first manner, the search process can be terminated once a solution is found. By using the heuristic method to produce an initial solution, the proposed method reduces the search space size of solutions significantly, and finds optimum solutions quickly. Experimental results show that the proposed method can find optimum solutions faster than the existing methods.

The proposed method still has a drawback, as shown in the experiments. In branch-and-bound approaches, a solution is updated during solution search since solutions are searched in a depth-first manner. On the other hand, in the proposed method based on dynamic programming, a solution is not updated, and thus, partial solutions are pruned only by the initial solution. Therefore, the search space size can be large when the number of indices is large. To overcome the drawback, we will study how to improve the initial solution and the lower bound in the proposed method.

ACKNOWLEDGMENTS

This research is partly supported by the JSPS KAKENHI Grant (C), No.16K00079, 2018. 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. Minato, "Zero-suppressed BDDs for set manipulation in combinatorial problems," *Proc. 30th Design Automation Conference*, pp. 272–277, 1993.

[3] 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.

[4] S. Nagayama, T. Sasao, and J. T. Butler, "An exact optimization algorithm for linear decomposition of index generation functions," *47th International Symposium on Multiple-Valued Logic*, pp. 161–166, May 2017.

[5] S. Nagayama, T. Sasao, and J. T. Butler, "An exact optimization method using ZDDs for linear decomposition of index generation functions," *48th International Symposium on Multiple-Valued Logic*, pp. 144–149, May 2018.

[6] 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 (in Russian).

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

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

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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 (in Japanese).