Abstract—This paper proposes a heuristic algorithm for linear decomposition of index generation functions using a balanced decision tree. The proposed algorithm finds a good linear decomposition of an index generation function by recursively dividing a set of its function values into two subsets with balanced size. Since the proposed algorithm is fast and requires a small amount of memory, it is applicable even to large index generation functions that cannot be solved in a reasonable time by existing algorithms. This paper shows time and space complexities of the proposed algorithm, and experimental results using some large examples to show its efficiency.

Keywords—Heuristic algorithm; balanced decision tree; linear decomposition; index generation functions; logic design.

I. INTRODUCTION

Pattern matching and text search are basic operations used in many applications, such as detection of computer viruses and packet classification. Logical behavior of these operations can be specified as index generation functions [2], [3]. 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 method using linear decomposition [1], [10] of index generation functions has been proposed [5]. This method realizes an index generation function \( f(X) \) using two blocks \( L \) and \( G \), as shown in Fig. 1. The first block \( L \) realizes linear functions with EXOR gates, registers, and multiplexers, and the second one \( G \) realizes a general function with a \( (2^p \times q) \)-bit memory, where \( p \) is the number of linear functions, and \( q \) is the number of bits needed to represent function values.

In this design method, minimization of \( p \) is important to reduce size of the memory for \( G \). Thus, various minimization algorithms have been proposed [4], [5], [7], [8]. However, for larger index generation functions, more efficient minimization algorithms are still required. Hence, in this paper, we propose a heuristic algorithm with smaller time and space complexities than the existing algorithms. The proposed algorithm is useful not only to find good linear decompositions of large index generation functions, but also to investigate a trade-off between complexity of \( L \) and memory size of \( G \) for large index generation functions.

The rest of this paper is organized as follows: Section II defines index generation functions and linear decomposition. Section III formulates the minimization problem of the number of linear functions, and shows our heuristic algorithm to solve it. Section IV shows experimental results using some practical examples, and Section V concludes the paper.

II. PRELIMINARIES

We briefly define index generation functions [2], [3] and their linear decompositions [1], [5], [10].

**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 \( x_n \) 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 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. (End of Example)

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 [5].

Definition 2: 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 \( y_i \):

\[
y_i(x_1, x_2, \ldots, x_n) = c_{i1}x_1 + c_{i2}x_2 + \ldots + c_{in}x_n
\]

where \( c_{ij} \in \{0, 1\} \ (j = 1, 2, \ldots, p) \), 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 \( y_i \), \( \sum_{j=1}^{n} c_{ij} \) is called a compound degree of \( y_i \), denoted by \( \text{deg}(y_i) \), where \( c_{ij} \) is viewed as an integer, and \( \Sigma \) is an ordinary integer sum.

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 II. In this case, \( \text{deg}(y_1) = 1 \) and \( \text{deg}(y_2) = 2 \), 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 \( \text{deg}(y_1) \) and \( \text{deg}(y_2) \) are 1. In either case, \( f \) can be realized by the architecture in Fig. 1 with a \((2^3 \times 3)\)-bit memory.

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

III. MINIMIZATION OF NUMBER OF LINEAR FUNCTIONS

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

A. Formulation of Minimization Problem

Since the architecture in Fig. 1 realizes an index generation function with EXOR gates 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:

Problem 1: Given an index generation function \( f \) and the maximum compound degree \( t \), find a linear decomposition of \( f \) such that the number of linear functions \( p \) is the minimum, and compound degrees are at most \( t \). For linear decompositions with the same value of \( p \), the one with the lowest compound degree is the optimum.

The maximum compound degree \( t \) 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 \( y = x_2, y_2 = x_4 \), and \( g_2(y_1, y_2) \) is the optimum. (End of Example)

B. Heuristic Algorithm for Minimization Problem

Since the solution space of Problem 1 is too large to solve the problem exactly, various heuristic algorithms have been proposed [4], [5], [7], [8]. However, for larger index generation functions, a heuristic algorithm that finds a good linear decomposition with smaller time and space complexities is still required. To reduce both time and space complexities, we propose a heuristic algorithm that searches only promising linear decompositions efficiently.

To solve Problem 1, we have to find the smallest set of compound variables such that \( k \) distinct combinations of values of the compound variables have a one-to-one correspondence to a set of indices for \( f \). In other words, we have to find the fewest compound variables that divide a set of indices into singletons (sets consisting of exactly one index). This corresponds to constructing a binary decision tree with the smallest height. This is the key idea for our heuristic algorithm.

Example 4: As shown in Fig. 2, finding the optimum linear decomposition in Example 3 can be considered as constructing a binary decision tree with the smallest height that divides a set of indices into singletons by compound variables \( y_1 \) and \( y_2 \). (End of Example)

Since a binary decision tree with the smallest height is a balanced decision tree, we propose a heuristic algorithm to construct a balanced decision tree using compound variables. To do this, a set of indices is divided into two subsets with balanced size recursively by compound variables. Before
describing requirements to find such compound variables, we define some terms and notations.

**Definition 3:** 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: \( K \rightarrow \{0, 1\} \) is called an inverse function to a compound variable \( y_i \), denoted by \( (g^{-1})_i(z) \).

**Definition 4:** 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 \( y_i \).

**Example 5:** For \( g_2(y_1, y_2) \) in Table II, 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.

**Theorem 1:** An index generation function with weight \( k = 2^m \), where \( m \) is a natural number, can be represented by a completely balanced binary decision tree with \( m \) compound variables, if and only if there exist \( m \) compound variables satisfying the following requirement: For all subsets \( Y \) of the set of the \( m \) compound variables, (1) holds,

\[
\bigcap_{y_i \in Y} ON(y_i) = 2^{m-h}, \tag{1}
\]

where \( h = |Y| \).

(Proof) See Appendix-A.

Although compound variables satisfying the above requirements are ideal to construct a balanced decision tree, weights of index generation functions are not always \( 2^m \), and only limited functions have such compound variables. In addition, finding such \( m \) compound variables is hard. Thus, to construct a balanced decision tree, we heuristically select a compound variable closer to the ideal one satisfying the above requirements by minimizing the following cost function over the unselected compound variables:

\[
cost(P, y_i) = \sqrt{\sum_{S \subseteq P} \left( \frac{|S|}{2} - |S \cap ON(y_i)| \right)^2}, \tag{2}
\]

where \( P \) is a partition of a set of indices with already selected compound variables. Initially, when there is no selected compound variables, \( P \) is the trivial partition consisting of a single block containing all indices.

This cost function is defined with the view of a Euclidean distance (2-norm) between \( y_i \) and an optimum compound variable that divides all subsets into halves. A compound variable with a small cost function tends to be a member of the optimum set of variables. Algorithm 1 shows a heuristic to find a good compound variable using the cost function. In the algorithm, \( P \) is a partition of a set of indices with already selected compound variables, and \( t \) is the maximum compound degree.

Since Algorithm 1 selects promising variables \( x_i \) using the cost function, and compunds only those variables, it can find a good compound variable with small time and space complexities. In Steps 2 and 4, when the cost function is equal, the algorithm selects a compound variable that divides subsets into smaller subsets. That is, the following is used as the second cost function:

\[
\max_{S \subseteq P} \left( \max(|S \cap ON(y_i)|, |S \setminus ON(y_i)|) \right).
\]

In addition, a compound variable with smaller compound degree is prioritized since the algorithm begins with the smallest compound degree.

By using Algorithm 1 iteratively, we can find a good linear decomposition. Algorithm 2 shows a heuristic to find a good linear decomposition using Algorithm 1. Algorithm 2 divides a set of indices iteratively using compound variables selected by Algorithm 1, and it terminates when a set of indices is divided into singletons.

**C. Time and Space Complexities of Heuristic Algorithm**

Since the cost function \( cost(P, y_i) \) is computed by checking which subset \( S \in P \) each index belongs to and whether it belongs to \( ON(y_i) \), its time complexity is \( O(k) \). In Step 2 of Algorithm 1, the cost function is invoked \( n \) times to find
the best $x_i$ among $x_1$ to $x_n$. Since Step 2 is iterated $t$ times, the time complexity of Algorithm 1 is

$$O(k) \times n \times t = O(knt).$$

Similarly to the cost function, the time complexity for dividing subsets of $P$ in Step 3 of Algorithm 2 is $O(k)$. Algorithm 2 invokes this computation and Algorithm 1 iteratively until $|P| = k$. Since the number of iterations in Algorithm 2 is $k - 1$ in the worst case, its time complexity is $(O(knt) + O(k)) \times (k - 1) = O(k^2nt)$. In this case, an extremely imbalanced decision tree is constructed as shown in Fig. 3. However, it rarely happens because the algorithm intends to construct a balanced decision tree. Thus, the number of iterations is $\log(k)$ on the average, resulting in a time complexity of $O(nk\log(k))$. Since $t$ is a small constraint parameter rather than the size of the problem, it can be considered as a constant. Therefore, the time complexity of Algorithm 2 is

$$O(nk\log(k)).$$

Memory sizes to store subsets of indices and to store selected compound variables are $O(k)$ and $O(n)$, respectively. On the other hand, memory size to store given registered vectors is $O(kn)$. Since other working spaces require much less memory size, the space complexity of Algorithm 2 is $O(\text{kn})$.

Table III compares our heuristic algorithm with existing algorithms, in terms of time and space complexities. Table III shows that our heuristic algorithm can solve even larger instances of Problem 1 (e.g., $n = 40$ and $k = 1,000,000$) with a computation time that is several orders of magnitude smaller and with smaller memory size than previous algorithms.

### IV. Experimental Results

The proposed heuristic algorithm is implemented in the C language, and run on the following computer environment: CPU: Intel Core2Quad Q6600 2.4GHz, memory: 4GB, OS: CentOS 5.7, and C-compiler: gcc -O2 (version 4.1.2).

#### A. On Quality of Solutions

Among the existing algorithms in Table III, the algorithm presented in ASP-DAC2012 [5] produces the best solutions (i.e., the smallest numbers of compound variables). Thus, we compare our heuristic algorithm with it in terms of quality of solutions. Table IV compares the numbers of compound variables selected by both algorithms for some benchmarks shown in [5].

Even though the search space of our heuristic algorithm is much smaller than that of the existing algorithm, the number of compound variables selected by our heuristic algorithm is not much larger than that selected by the existing algorithm, as shown in Table IV. Particularly, for the benchmark of 1-out-of-20 code, our heuristic algorithm found the exact minimum number of compound variables [9] when $t = 1$ to 8 except for $t = 2$. This shows that our algorithm finds good solutions efficiently by pruning unpromising solutions heuristically.

#### B. Results for Large Problems

To show that our algorithm can be applied to larger problems, we used the following three examples: 1) random social security and tax numbers (SST numbers) in Japan [11]; 2) the bible [15]; and 3) the US constitution [16] including amendments [17], [18]. 1) is used as a numeric example with large $k$, and 2) and 3) are examples of text search with large $n$. For how to generate index generation functions from these examples, see Appendix-B. Table V shows computation time of our algorithm and the number of compound variables for these examples.

For each example function, we can predict the number of compound variables using Property 1 shown in [6].

**Property 1:** [6] When $n$ is sufficiently large and $k << 2^n$, most index generation functions with weight $k$ can be
represented by $L - 1$, $L$, or $L + 1$ compound variables, where $L = 2 \lfloor \log_2 (k + 1) \rfloor - 4$.

For the function of the SST numbers, the predicted number of compound variables is $L - 1 = 2 \times \lfloor \log_2 (1,000,001) \rfloor - 5 = 35$; for the function of the bible, it is $L - 1 = 2 \times \lfloor \log_2 (20,828) \rfloor - 5 = 25$; and for the function of the US constitution, it is $L - 1 = 2 \times \lfloor \log_2 (254) \rfloor - 5 = 11$. As shown in Table V, our algorithm achieves those numbers or even smaller numbers when $t \geq 3$ for the SST numbers, $t \geq 5$ for the bible, and $t \geq 3$ for the US constitution.

These results show that even if $n$ and $k$ are large, our algorithm finds a good solution within a practical time.

C. Number of Compound Variables vs. Compound Degree

Fig. 4 shows distributions of $|ON(x_i)|$ for the example index generation functions. In the figure, the horizontal axis denotes ratios of the number of 1s included in original variables $x_i$ in registered vectors, and the vertical axis denotes ratios of the number of $x_i$ having the same ratio of $|ON(x_i)|$.

As shown in Fig. 4, the example functions have few variables $x_i$ with $|ON(x_i)|/k \simeq 0.5$ that can divide a set of indices into halves. Thus, many variables are required when $t = 1$, as shown in Table V. However, for such functions, we can produce variables with $|ON(x_i)|/k \simeq 0.5$ by increasing the compound degree, and thus, the number of variables can be reduced. As shown in Table V, however, it is not reduced so much for $t > 5$. We think this is because variables with $|ON(x_i)|/k \simeq 0.5$ are obtained sufficiently when $t = 5$. This means that practically effective compound degree is at most 5 for the example index generation functions.

V. Conclusion and Comments

This paper proposes a balanced decision tree based heuristic algorithm to minimize the number of compound variables for linear decomposition of index generation functions. Since time and space complexities of the proposed algorithm are smaller than those of existing heuristic algorithms, it can be applied to larger index generation functions. Experimental results show that the proposed algorithm finds a good solution that is close to the best solution ever found, even though its search space is much smaller. And, this paper also shows a relation between the number of compound variables and compound degrees $t$, and shows that the number of compound variables is reduced by increasing $t$ until $t = 5$.

The proposed heuristic algorithm would be helpful for exact minimization algorithm based on a branch-and-bound method because we can prune unpromising solutions using the heuristic. We will study on an exact minimization algorithm based on a branch-and-bound method. In addition, we will study on a more efficient cost function than (2).

Acknowledgments

This research is partly supported by the JSPS Kaken (C), 25330071 and 26330072, and Hiroshima City University Grant for Academic Research (General Studies), 2015.

References

A. Proof for Theorem 1

(if) Assume that there exist \( m \) compound variables satisfying the requirement. Then, since for any compound variable \( y_i \), \( |ON(y_i)| = 2^m/2 \) holds, each \( y_i \) can divide a set of indices into halves by \( y_i = 0 \) and \( y_i = 1 \). And, each subset of \( 2^{m-1} \) indices obtained by partition with \( l \) variables can be divided into halves furthermore by \( y_i = 0 \) and \( y_i = 1 \), as shown in Fig. 2. Thus, for any compound variable \( y_i \), \( |ON(y_i)| = 2^m/2 \) holds. And, since a set of indices is divided into equal-sized subsets recursively, for any \( h \) variables, \( \bigcap_{i=1}^{h} ON(y_i) = 2^m/2^h \) holds.

B. How to Generate Index Generation Functions

In 2015, Japan introduced the new “social security and tax number” (SST number) to replace the old “resident’s identification number” (RIN) [11]. The SST number is a 12-digit decimal number \((d_{11} d_{10} \ldots d_1 d_0)_{10}\), and it consists of a single check digit \(d_0\) and an 11-digit number \((d_{11} d_{10} \ldots d_1)_{10}\) that is generated from the resident’s 11-digit RIN [12]. The RIN, in turn, consists of a single check digit and a 10-digit number that is randomly generated to prevent from identifying an individual from the number [14]. Thus, we randomly generated a 11-digit number, and attached a check digit to its least significant digit to generate an SST number. The check digit \(d_0\) is obtained by the following computation [13]:

\[
d_0 = \begin{cases} 
0 & (r \leq 1) \\
11 - r & \text{(otherwise)}
\end{cases}
\]

\[
r = \left( \sum_{i=7}^{11} d_i \times (i-5) + \sum_{i=1}^{6} d_i \times (i+1) \right) \pmod{11}.
\]

We randomly generated 1,000,000 distinct SST numbers, and assigned an index from 1 to 1,000,000 to each number. By converting each digit into a 4-bit number, we generated 1,000,000 registered vectors, each with \(4 \times 12 = 48\) bits.

The bible [15] consists of 31,102 verses, and we took the first 80 characters from each verse excluding its reference number and verses shorter than 80 characters. Then, we obtained 20,827 distinct strings of the characters by removing the duplicated strings. By assigning an index from 1 to 20,827 to each string, and converting each character into a 7-bit binary number using the ASCII code, we generated the second index generation function.

The US constitution [16] consists of 256 sentences, including amendments [17], [18] but excluding headings. Similarly to the bible, we took the first 216 characters from each sentence, and then, we obtained 253 strings by removing the duplicated strings. For sentences shorter than 216 characters, blanks are padded to make their length 216. By assigning an index from 1 to 253 to each string, and converting each character into a 7-bit binary number using the ASCII code, we generated the third index generation function.