1 INTRODUCTION

Binary decision diagrams (BDDs) [5] and multi-valued decision diagrams (MDDs) [15] are extensively used in logic synthesis [10], logic simulation [1, 13, 17], software synthesis [2, 14], and pass transistor logic (PTL) [3, 29, 30]. These applications use decision diagrams to evaluate logic functions, and the evaluation time is proportional to the average path length (APL) in the decision diagram. Therefore, minimization of the APL leads to faster evaluation of the logic function. Particularly, in logic simulation using decision diagrams [1, 13, 17], minimization of the APL reduces the simulation time substantially because logic functions are evaluated many times with different test vectors.

Minimization of the APL can also be applied to logic synthesis. A method for functional decomposition [32] uses BDDs to detect Boolean divisors. The quality of a divisor is measured by the number of don’t-cares it provides for the
**Exact and Heuristic Minimization of the Average Path Length in Decision Diagrams**

Naval Postgraduate School, Department of Electrical and Computer Engineering, Monterey, CA, 93943

**Approved for public release; distribution unlimited.**

**16. SECURITY CLASSIFICATION OF:**

<table>
<thead>
<tr>
<th>a. REPORT</th>
<th>b. ABSTRACT</th>
<th>c. THIS PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>unclassified</td>
<td>unclassified</td>
<td>unclassified</td>
</tr>
</tbody>
</table>

**19a. NAME OF RESPONSIBLE PERSON**

<table>
<thead>
<tr>
<th>a. REPORT</th>
</tr>
</thead>
<tbody>
<tr>
<td>unclassified</td>
</tr>
</tbody>
</table>

**19. NUMBER OF RESPONSIBLE PERSON**

Standard Form 298 (Rev. 8-98)  
Prescribed by ANSI Std Z39-18
minimization of the quotient. The don’t-cares are generated by the paths in the BDD that lead to the terminal nodes. The shorter the paths, the more don’t-care minterms they contain. Therefore, minimizing the APL in BDDs can improve the quality of decomposition.

In pass transistor logic (PTL) synthesis, the circuits are derived directly from BDDs representing logic functions. In this case, the longer paths in BDDs cause larger voltage drop and larger delay. This problem can be solved by inserting buffers in long paths [3]. Minimizing the APL in the BDD can reduce the number of buffers that must be inserted.

In this paper, we propose an exact APL minimization algorithm based on the branch-and-bound algorithm. This algorithm finds an optimum variable order much faster than exhaustive search, which enumerates all possible variable orders. However, the exact method is time-consuming for functions with many inputs. To minimize the APL of such functions in a reasonable time, we propose a heuristic algorithm based on dynamic variable reordering.

This paper is organized as follows. Section 2 contains the necessary terminology and definitions. Section 3 introduces lower bounds on the APL. Section 4 proposes an exact and a heuristic minimization algorithm for the APL. Section 5 considers the paired ordering of binary variables. Section 6 shows the efficiency of the algorithms using benchmark functions. Experimental results for 2-valued cases and 4-valued cases are shown. The Appendix includes the proofs of theorems.

2 PRELIMINARIES

We assume that the reader is familiar with the basic terminology of reduced ordered binary decision diagrams (ROBDDs) [5] and reduced ordered multi-valued decision diagrams (ROMDDs) [15]. In the following, a BDD and an MDD mean an ROBDD and an ROMDD. DD means either BDD or MDD.

Definition 2.1 Let \( x \) be an \( r \)-valued variable, and let \( c \in \{0, 1, \ldots, r-1\} \). Then, \( P(x = c) \) denotes the probability that \( x \) has value \( c \).

Definition 2.2 In a DD, a sequence of edges and non-terminal nodes leading from the root node to a terminal node is a path. The number of edges in the path is the path length.

Note that the sequence of edges in a path \( p_i \) of a DD corresponds to an assignment of values \( a_i \) to the specific variables associated with those edges in the DD. We say that such an assignment \( a_i \) selects path \( p_i \). Similarly, if an assignment of values \( c_i \) to all variables agrees with \( a_i \) for all variables assigned in \( a_i \), we also say \( c_i \) selects path \( p_i \).

Definition 2.3 In a DD for an \( n \)-variable function, the path probability of a path \( p_i \), denoted by \( PP(p_i) \), is the probability that the path \( p_i \) is selected in all assignments of values to the \( r \)-valued variables. \( PP(p_i) \) is given by

\[
PP(p_i) = \sum_{\bar{c} \in C_i} P(x_1 = c_1) \times P(x_2 = c_2) \times \ldots \times P(x_n = c_n),
\]

where \( C_i \) denotes a set of assignments of values to the variables selecting the path \( p_i \), \( \bar{c} = (c_1, c_2, \ldots, c_n) \), each \( c_j \in \{0, 1, \ldots, r-1\} \), and \( P(x_j = c_j) \) is the probability \( x_j \) has value \( c_j \).

Definition 2.4 The average path length, or \( \text{APL} \), in a DD is given by:

\[
\text{APL} = \frac{N}{i=1} PP(p_i) \times l_i,
\]

where \( i \) indexes the paths, \( N \) denotes the number of paths, and \( l_i \) denotes the path length of path \( p_i \).

Definition 2.5 The node traversing probability of a node \( v \), denoted by \( NTP(v) \), is the probability that an assignment of values to the variables selects a path that includes the node \( v \).

Definition 2.6 The edge traversing probability of an edge \( e \), denoted by \( ETP(e) \), is the probability that an assignment of values to the variables selects a path that includes the edge \( e \).

Note that the node traversing probability of the root node in a decision diagram is 1.0, since all paths start from the root node.
By using node traversing probabilities, we can compute this APL as follows: First, we have $\text{NTP}(v_1) = 1.00$ for root node $v_1$. Then, $\text{NTP}(v_2) = \text{ETP}(e_1) = P(x_1 = 0) \times \text{NTP}(v_1) = 0.50$ and $\text{NTP}(v_3) = \text{ETP}(e_2) = P(x_1 = 1) \times \text{NTP}(v_1) = 0.50$. Similarly,

\[
\begin{align*}
\text{NTP}(v_4) &= P(x_2 = 1) \times \text{NTP}(v_2) + P(x_2 = 0) \times \text{NTP}(v_3) = 0.50, \\
\text{NTP}(v_5) &= P(x_2 = 1) \times \text{NTP}(v_3) = 0.25, \quad \text{and} \\
\text{NTP}(v_6) &= P(x_3 = 1) \times \text{NTP}(v_4) + P(x_3 = 0) \times \text{NTP}(v_5) = 0.375.
\end{align*}
\]

**Lemma 2.1** [27] The node traversing probability of node $v$ is the sum of the edge traversing probabilities of all incoming edges to $v$. Also, the node traversing probability of node $v$ is the sum of the edge traversing probabilities of all outgoing edges from $v$.

**Proof.** See Appendix.

From Lemma 2.1, the following relation holds:

\[
\text{ETP}(e) = P(x = c) \times \text{NTP}(v),
\]

where $P(x = c)$ is the probability $x$ has a value $c$, $v$ is a node representing a variable $x$, and $e$ is an outgoing edge corresponding to the value $c$ of $v$.

**Theorem 2.1** [27] The APL is equal to the sum of the edge traversing probabilities of all edges. Also, the APL is equal to the sum of the node traversing probabilities of all the non-terminal nodes.

**Proof.** See Appendix.

From Theorem 2.1, we have the following:

\[
\text{APL} = \sum_{i=1}^{N_e} \text{ETP}(e_i) = \sum_{j=1}^{N_v} \text{NTP}(v_j),
\]

where $N_e$ and $N_v$ denote the number of edges and non-terminal nodes, respectively.

**Example 2.1** Consider the BDD in Fig. 1(a), where solid lines and dotted lines denote 1-edges and 0-edges, respectively. For simplicity, assume that $P(x_i = 0) = P(x_i = 1) = 0.50$ ($i = 1, 2, 3, 4$). This BDD has 10 different paths: path $p_1$ is $(v_1, e_1, v_2, e_3)$, path $p_2$ is $(v_1, e_1, v_2, e_4, v_4, e_7)$, ..., and path $p_{10}$ is $(v_1, e_2, v_3, e_5, v_5, e_{10})$. The PP $p_i$ and path length of each path $p_i$ are listed in Fig. 1(b). Therefore, by Definition 2.4,

\[
\text{APL} = \sum_{i=1}^{10} \text{PP}(p_i) \times l_i = 3.125.
\]

By using node traversing probabilities, we can compute this APL as follows: First, we have $\text{NTP}(v_1) = 1.00$ for root node $v_1$. Then, $\text{NTP}(v_2) = \text{ETP}(e_1) = P(x_1 = 0) \times \text{NTP}(v_1) = 0.50$ and $\text{NTP}(v_3) = \text{ETP}(e_2) = P(x_1 = 1) \times \text{NTP}(v_1) = 0.50$. Similarly,
Thus, we obtain

\[ APL = \sum_{i=1}^{6} NT P(v_i) = 3.125. \]

Similarly, we can compute the APL using the edge traversing probabilities.

(End of Example)

Consider a multiple-output function \( F = (f_0, f_1, \ldots, f_m) : \mathbb{R}^n \rightarrow \mathbb{R}^m \), where \( R = \{0, 1, \ldots, r-1\} \), and \( n \) and \( m \) denote the number of input and output variables, respectively. In this paper, we use shared MDDs (SMDDs) to represent multiple-output function \( F \). For reasons that will be clear later, we view the APL of an SMDD as the sum of the APLs of the individual MDDs for each component function \( f_i \).

### 3 LOWER BOUNDS ON APL

In this section, we derive lower bounds on the APL. Such bounds are used to reduce the computation time in the algorithm, as discussed later.

**Definition 3.1** Suppose an MDD is partitioned into two parts as shown in Fig. 2. Here, \( X_{\text{upper}} \) denotes the variables above or in level \( i \), \( X_{\text{lower}} \) denotes the variables below or in level \( i+1 \), and \( \text{Cut}(i) \) denotes a set of edges connecting the nodes above or in level \( i \) with the nodes below or in level \( i+1 \).

Note that the nodes are indexed by \( i \) starting with the root node at level 1. The nodes just below have \( i = 2 \), etc..

**Definition 3.2** \( ETP(\text{Cut}(i)) \) denotes the sum of edge traversing probabilities of edges in \( \text{Cut}(i) \), and is given by

\[ ETP(\text{Cut}(i)) = \sum_{e \in \text{Cut}(i)} ETP(e). \]

**Lemma 3.1** Suppose an SMDD represents a multiple-output function \( F = (f_0, f_1, \ldots, f_m) \). Then,

\[ ETP(\text{Cut}(i)) = m_U, \]

where \( m_U \) is the number of the root nodes of the multiple-output function \( F \) above or in level \( i \).

**Proof.** See Appendix.

**Corollary 3.1** Suppose an MDD represents a single-output function \( f \). Then,

\[ ETP(\text{Cut}(i)) = 1.0. \]

**Lemma 3.2** Let

\[ \text{Cut}'(i) = \{ e | e \in \text{Cut}(i), \text{ such that } e \text{ is incident to only non-terminal nodes} \}. \]

Then, for every permutation of \( X_{\text{upper}} \),

\[ ETP(\text{Cut}'(i)) = c_i, \]

where \( c_i \leq m_U \).
Proof. See Appendix.

**Theorem 3.1** Consider an SMDD for multiple-output function $F$. Let $L$ be the sum of the node traversing probabilities of the non-terminal nodes below or in level $i + 1$. Let $m_L$ be the number of root nodes for $F$ below or in level $i + 1$. Then, for any permutation of $X_{\text{lower}}$ and any permutation of $X_{\text{upper}}$,

$$ETP(Cut'(i)) + m_L \leq L.$$  

Proof. See Appendix.

**Theorem 3.2** Consider an SMDD for multiple-output function $F$. Let $U$ be the sum of the node traversing probabilities of the non-terminal nodes above or in level $i$. When the order of $X_{\text{upper}}$ is fixed,

$$U + ETP(Cut'(i)) + m_L \leq APL.$$  

Proof. See Appendix.

**Corollary 3.2** Consider an SMDD of multiple-output function $F$. Let $U$ and $L$ be the sums of the node traversing probabilities of the non-terminal nodes above and below or in level $i$, respectively. Then

$$\max\{L, U\} \leq APL.$$  

### 4 MINIMIZATION OF APL

Since the APL in a DD (BDD or MDD) depends on the variable order, the APL minimization problem can be formulated as follows:

**Problem 4.1** Given a DD for a logic function $f$, find a variable order that produces the minimum APL.

#### 4.1 Change of the APL during Swapping Two Adjacent Variables

Our APL minimization algorithms go from one variable order to another variable order by a sequence of steps that swap pairs of adjacent variables. A part of the algorithms that has a significant effect on computation time is updating the APL after swapping each pair of adjacent variables. This section describes a fast method to update the APL after the swap of two adjacent variables.

**Theorem 4.1** Let $U$ be the sum of the node traversing probabilities of non-terminal nodes above or in level $i - 1$, and let $L$ be the sum of the node traversing probabilities of non-terminal nodes below or in level $i + 2$. Then, after the variable swap of level $i$ with level $i + 1$, $U$ and $L$ remain unchanged.

Proof. See Appendix.

Theorem 4.1 shows that the previously computed node traversing probabilities need not be repeated in computing the new APL caused by the swap of two adjacent variables. Fig. 3 illustrates a subgraph of level $i$ and level $i + 1$ in the BDD when two adjacent variables are interchanged. Since the principles of variable swap for the binary case and the multi-valued case are the same, we describe only the binary case. The details of variable swaps for the multi-valued case are discussed in [18]. A subgraph composed of BDD nodes involved in the variable swap belongs to one of the six classes shown in Fig. 3. For each class, the figure on the left occurs before the swap, while the figure on the right occurs as a result of the swap. In Fig. 3, only cases (e) and (f) do not change the APL, while other cases change the APL. For example, in case (a), the node traversing probabilities of nodes $v_2$ and $v_3$ are changed as a result of the swap. Before the swap, the node traversing probabilities of $v_2$ and $v_3$ are given by:

\[
\begin{align*}
NTP(v_2) &= ETP(e_0) = P(x_i = 0) \times NTP(v_1) \\
NTP(v_3) &= ETP(e_1) = P(x_i = 1) \times NTP(v_1),
\end{align*}
\]

where $e_0$ and $e_1$ denote the edges from $v_1$ to $v_2$ and from $v_1$ to $v_3$, respectively. On the other hand, after the swap, the node traversing probabilities of $v_2$ and $v_3$ are:

\[
\begin{align*}
NTP(v_2) &= P(x_{i+1} = 0) \times NTP(v_1) \\
NTP(v_3) &= P(x_{i+1} = 1) \times NTP(v_1),
\end{align*}
\]
When $P(x_i = 0) = P(x_{i+1} = 0)$ and $P(x_i = 1) = P(x_{i+1} = 1)$, the node traversing probabilities of $v_2$ and $v_3$ do not change after the swap. Therefore, in case (a), the APL is changed by the edge traversing probabilities of outgoing edges from $v_1$.

Similarly, in other cases except for (e) and (f), the APL is changed by the edge traversing probabilities of outgoing edges from the root node of a subgraph. Note that from Theorem 4.1, we consider only the edges from the root node to nodes in level $i + 1$ to update the APL.

We summarize the strategy for updating the APL as follows:

1. Before the swap, for each subgraph involved in the swap, the edge traversing probabilities of the edges from the root node of a subgraph to nodes in level $i + 1$ are subtracted from 1) the APL and from 2) the node traversing probabilities of nodes in level $i + 1$.

2. After the swap, for each subgraph, the edge traversing probabilities of edges from the root node of a subgraph to nodes in level $i + 1$ are re-calculated.

3. The calculated edge traversing probabilities are added to 1) the APL and to 2) the node traversing probabilities of nodes in level $i + 1$.

**Example 4.1** Fig. 4 shows BDDs for logic function $f = x_1x_4 \lor x_2x_4 \lor x_3$. Fig. 4(a) shows the BDD with the variable order $(x_1, x_2, x_3, x_4)$, top to bottom. For simplicity, assume that $P(x_i = 0) = P(x_i = 1) = 0.50 \ (i = 1, 2, 3, 4)$. Then, the APL of the BDD in Fig. 4(a) is 2.875. In this BDD, we consider the swap of variables $x_2$ and $x_3$. During such a swap, case (b) applies to node $v_2$ and case (f) applies to node $v_4$. Performing the swap leads to the BDD shown in Fig. 4(b). Note that the swap decreases the APL by 0.25 because the node $v_4$ after the swap does not have the incoming edge from node $v_2$. The node traversing probabilities associated with nodes $v_2$ and $v_3$ do not change. The overall APL decreases from 2.875 to 2.625.

(End of Example)

**Example 4.2** Fig. 5(a) shows the BDD with the variable order $(x_2, x_3, x_1)$ for logic function $f = x_1(x_2 \lor x_3)$. Assume that

$$P(x_1 = 0) = 0.6, \quad P(x_1 = 1) = 0.4,$$
$$P(x_2 = 0) = 0.3, \quad P(x_2 = 1) = 0.7,$$
$$P(x_3 = 0) = 0.8, \quad P(x_3 = 1) = 0.2.$$

The APL of the BDD in Fig. 5(a) is 2.06. For the swap of variables $x_3$ and $x_1$, case (d) applies to node $v_2$ and case (f) applies to node $v_3$. Performing this swap yields the BDD shown in Fig. 5(b). It changes the node traversing probabilities of $v_3$ and $v_4$ (a new node). Before the swap, the edge traversing probability of edge from $v_2$ to $v_3$, 0.06, is subtracted from the APL and from the node traversing probabilities of $v_3$. After the swap, the edge traversing probability of edge from $v_2$ to $v_4$, 0.12, is added to the APL and to $v_4$. The overall APL increases from 2.06 to 2.12.

(End of Example)
4.2 Symmetric Variables

Definition 4.1 A logic function \( f (x_1, x_2, \ldots, x_i, \ldots, x_j, \ldots, x_n) \) is symmetric with respect to \( x_i \) and \( x_j \) if the interchange of \( x_i \) and \( x_j \) does not change \( f \). \( x_i \) and \( x_j \) are called symmetric variables.

In a DD, swapping symmetric variables \( x_i \) and \( x_j \) does not change the graph structure.

Definition 4.2 Let \( \pi_1 \) and \( \pi_2 \) be permutations of the variables. If the positions of variables in \( \pi_1 \) are the same as in \( \pi_2 \) except for symmetric variables, \( \pi_1 \) and \( \pi_2 \) are called symmetric orders.

Since symmetric orders produce DDs with the same graph structure, the DDs have the same APL when \( P(x_i = 0) = P(x_j = 0) \), \( P(x_i = 1) = P(x_j = 1) \), \( \ldots \), and \( P(x_i = r - 1) = P(x_j = r - 1) \) for symmetric variables \( x_i \) and \( x_j \). Therefore, in such a case, detection of symmetric orders can reduce the computation time for an APL minimization algorithm.

Example 4.3 Consider the logic function \( f = x_1 x_4 \lor x_2 x_4 \lor x_3 \) (Fig. 4). Let variable orders \( \pi_1 \) and \( \pi_2 \) be \( (x_1, x_2, x_3, x_4) \) and \( (x_2, x_1, x_3, x_4) \), respectively. Since \( x_1 \) and \( x_2 \) are symmetric variables, \( \pi_1 \) and \( \pi_2 \) are symmetric orders. The BDDs for the two orders are the same except the labels \( x_1 \) and \( x_2 \) are interchanged, and have the same APL and the same number of nodes. (End of Example)
4.3 Exact Minimization Algorithm

Fig. 6 shows a pseudo-code to solve Problem 4.1. This algorithm finds an optimum solution using a branch-and-bound method, similar to the top-down algorithm (JANUS) in [9]. JANUS [9] uses the number of nodes in a BDD as the cost function, while our algorithm uses the APL of a DD (BDD or MDD) as the cost function. By using the node traversing probability (NTP), the changes in APL can be calculated at each node locally. This locality of computation allows a top-down algorithm. To our knowledge, this is the first time an APL minimization algorithm based on branch-and-bound has been proposed. This algorithm finds an optimum variable order much faster than the exhaustive search method, which enumerates all possible variable orders. In lines 11 and 31 of Fig. 6, procedure ordering changes the variable order of the DD into the given order from the top to the specified level. For example, let the current variable order be \((x_1, x_2, x_3, x_4, x_5)\). We seek the order \((x_5, x_4)\) at level two. That is, we seek \((x_5, x_4, *, *, *)\), where \(*, *, *\) represents \(x_1, x_2, \) and \(x_3\) in some order. Then, procedure ordering\((DD, (x_5, x_4), 2)\) obtains the order \((x_5, x_4, x_1, x_2, x_3)\) in 7 swaps from the order \((x_1, x_2, x_3, x_4, x_5)\).

Procedure symmetry\_check in line 15 checks symmetry of adjacent variables [22]. When the variable order of \(X'_\text{sub}\) which has already been stored in array \("order[X'_\text{sub}]\"\) as a candidate, and the current variable order of the DD are symmetric, and all \(P(x = c)\)s are the same for the symmetric variables, the current order is excluded from the set of candidates. In line 19, Theorem 3.2 is used to eliminate the unneeded variable exchanges to reduce computation time. In line 21, NTP\((level)\) denotes the sum of the node traversing probabilities of the nodes on the given level \((level)\). The initial values of array cost
in Fig. 6 are set to infinity.

4.4 Heuristic Minimization Algorithm

The exact minimization algorithm in Fig. 6 obtains an optimum solution for Problem 4.1. However, when the number of input variables is large, finding the optimum variable order may require much computation time.

In this section, we show a heuristic minimization method using variable sifting [23]. The sifting algorithm repeatedly performs the following basic steps:

1. Change the variable order.
2. Compute a cost.

The proposed sifting algorithm uses APL as the cost function. It was shown in Section 4.1 that the APL can be efficiently updated after the swap of two adjacent variables. As a result, the time needed to compute the cost in our sifting algorithm is comparable to the time needed to update the number of nodes in the classical sifting algorithm, which minimizes the number of nodes. Fig. 7 shows the pseudo-code of the heuristic minimization algorithm. In this algorithm, each variable $x_i$ is sifted across all possible positions to determine its best position. First, $x_i$ is sifted in one direction to the closer extreme (top or bottom). Then, $x_i$ is sifted in the opposite direction to the other extreme. In lines 10 and 20 of Fig. 7, Corollary 3.2 is used to eliminate unneeded sifting of $x_i$. When variable $x_i$ moves down to the bottom, we use $U$ equal to the sum of the

```c
1: sifting_APL (DD, #rounds of sifting R) {
2:     cost = APL for initial DD ;
3:     for (r = 0; r < R; r++) {
4:         for (each $x_i \in X$) {
5:             start = current position of $x_i$ ;
6:             best_p = start ;
7:             for (each position $p$ from start to the closer extreme) {
8:                 Move $x_i$ to $p$ ;
9:                 Update $U$ (or $L$) ;
10:                if ($cost \leq U$ (or $L$))
11:                    break ;
12:                if ($APL < cost$) {
13:                    cost = APL ;
14:                    best_p = p ;
15:                }
16:            }
17:            for (each position $p$ to the other extreme) {
18:                Move $x_i$ to $p$ ;
19:                Update $U$ (or $L$) ;
20:                if ($cost \leq U$ (or $L$))
21:                    break ;
22:                if ($APL < cost$) {
23:                    cost = APL ;
24:                    best_p = p ;
25:                }
26:            }
27:            Move $x_i$ to best_p ;
28:        }
29:    }
30: }
```

Figure 7: Heuristic APL minimization algorithm.
node traversing probabilities of the nodes above \( x_i \). If \( \text{cost} \leq U \), sifting of \( x_i \) further down to the bottom cannot lead to a smaller APL than \( \text{cost} \). In such cases, there is no need to continue sifting to the bottom. Similarly, when variable \( x_i \) moves up to the top, we use \( L \) equal to the sum of the node traversing probabilities of the nodes below \( x_i \). This lower bound for the APL is similar to the one introduced for the number of nodes during the classical sifting [8].

4.5 Initial Ordering of the Binary Variables

The initial ordering of variables influences the effectiveness of the heuristic minimization algorithm described in the previous section. An analysis of variable orders that produces the minimal APL in several known classes of functions [6, 28] leads to a heuristic to find a good initial variable order. In this section, we propose an initial variable order using Walsh spectrum [12] for binary logic functions.

The value of a first-order Walsh spectral coefficient expresses the correlation between the variable value with the function value. For \( n \)-variable logic function \( f(X) \), the first-order Walsh spectral coefficient can be computed as follows [7]:

\[
R_j = \frac{|\bar{x}_i \oplus f|}{2^n - 1} - 1, 
\]

where \(|\bar{x}_i \oplus f|\) denotes the number of assignments of values to the variables \( X \) such that the values of \( x_i \) and \( f(X) \) are equal. The initial variable order is found by placing the variables in descending order of the absolute value of \( R_j \). For variables with identical absolute values of \( R_j \), we arbitrarily choose the order.

All spectral coefficients can be computed by scanning the nodes beginning at the root node and ending on the terminal nodes using a fast algorithm [31]. The first-order coefficients can be computed by a simplified version of the general algorithm.

**Example 4.4** Consider the logic function \( f = x_1x_4 \lor x_2x_4 \lor x_3 \) in Example 4.1. For each binary variable \( x_i \), the value of \(|\bar{x}_i \oplus f|\) is given by:

\[
|\bar{x}_1 \oplus f| = 9, \quad |\bar{x}_2 \oplus f| = 9, \quad |\bar{x}_3 \oplus f| = 13, \quad |\bar{x}_4 \oplus f| = 11. 
\]

The value of each \( R_j \) corresponding to \( x_i \) is as follows:

\[
R_1 = \frac{1}{8}, \quad R_2 = \frac{1}{8}, \quad R_3 = \frac{5}{8}, \quad R_4 = \frac{3}{8}. 
\]

Therefore, we have an initial variable order \( x_3, x_4, x_1, x_2 \), and APL = 1.875. This is the minimum APL for \( f \). (End of Example)

5 PAIRED ORDERING OF BINARY VARIABLES

Unfortunately, there are no standard benchmark functions for multi-valued logic. Thus, 4-valued input 2-valued output functions obtained by pairing binary variables of 2-valued benchmark functions are often used for experiments for the multi-valued case [11, 18, 24]. Especially, [11, 24] show that 4-valued MDDs can represent binary logic functions more compactly than BDDs, by considering paired orderings of binary variables. 4-valued MDDs can be implemented efficiently using LUT-based FPGAs. In Section 6.2, we also use the 4-valued input 2-valued output functions for experiments of the multi-valued case, and show that 4-valued MDDs can reduce the APL, as well as the number of nodes efficiently. To do this, in this section, we define a paired ordering of binary variables.

**Definition 5.1** Let \( f(X) \) be a 2-valued logic function, where \( X = (x_1, x_2, \ldots, x_n) \) is an ordered set of binary variables. Let \( \{X\} \) denote the unordered set of variables in \( X \). Let \( X_i \subseteq X \). If \( \{X\} = \{X_1\} \cup \{X_2\} \cup \ldots \cup \{X_n\}, \{X_i\} \cap \{X_j\} = \emptyset \) \((i \neq j)\), and \(|X| = 2\), then \((X_1, X_2, \ldots, X_n)\) is a **paired ordering** of binary variables \( X \), and each \( X_i \) can be represented as a 4-valued variable. And then, a 2-valued logic function \( f(X) \) can be represented by the mapping \( f(X_1, X_2, \ldots, X_n): \{0, 1, 2, 3\}^n \to \{0, 1\} \).

For \( n \)-variable functions, if \( n < 2u \) (i.e. \( n \) is an odd number), we use an additional redundant binary variable called a **dummy variable**. The set of binary variables with the dummy variable if it exists, is denoted by \( \{X\}' = \{x_1, x_2, \ldots, x_n, x_{n+1}\} \), where \(|X'| = n + 1 = 2u\). Note that \( f \) is independent of \( x_{n+1} \).
Theorem 5.1 The number of different paired orderings of binary variables $X = (x_1, x_2, \ldots, x_n)$ to consider is
\[ \frac{n!}{2^u}, \]
where $u$ denotes the number of 4-valued variables obtained by the pairing, and is given by
\[ u = \frac{n}{2}. \]
Note that we assume that $n = 2u$.\(^2\)

Proof. See Appendix.

In [11, 24], heuristic paired ordering algorithms for node minimization have been proposed. However, in this paper, we consider the paired ordering algorithms for APL minimization. We formulate the APL minimization problem considering the paired orderings of binary variables as follows:

Problem 5.1 Given a binary logic function $f(X)$, find a paired ordering of binary variables $X$ that produces an MDD with the minimum APL.

5.1 Exact Paired Ordering Algorithm

Fig. 8 shows a pseudo-code to solve Problem 5.1. This algorithm finds an optimum solution for Problem 5.1 using branch-and-bound, similar to the algorithm in Fig. 6. In lines 6 and 38 of Fig. 8, procedure pairing produces an MDD for 4-valued input 2-valued output function by making pairs of binary variables from the given variable order. In line 7, $\text{min} \cdot \text{apl}$ is set to the APL of MDD obtained by making pairs of binary variables from the variable order of given BDD. In line 26, $\text{pairing}_\text{NTP}(\text{level})$ denotes the sum of the node traversing probabilities of the MDD nodes obtained by pairing binary variables in level and level + 1.

5.2 Heuristic Paired Ordering Algorithm

In this section, we show a heuristic paired ordering algorithm. The heuristic paired ordering algorithm, called \textit{pair-sifting algorithm}, consists of the following four basic steps:

1. Apply the sifting algorithm for APL minimization presented in Section 4.4 to the BDD of the given binary logic function.
2. Make pairs of binary variables from the variable order obtained by the sifting algorithm.
3. Construct an MDD for the 4-valued input 2-valued output function.
4. Apply the sifting algorithm to the MDD.

This strategy is similar to one used in [24], which minimizes the number of nodes in an MDD.

6 EXPERIMENTAL RESULTS

Experiments using MCNC benchmarks were conducted in the following environment:

- CPU: Pentium4 Xeon 2.8GHz
- L1 Cache: 32KB
- L2 Cache: 512KB
- Main Memory: 4GB
- Operating System: redhat (Linux 7.3)
- C-Compiler: gcc -O2

In this section, we assume that $P(x_i = 0) = P(x_i = 1) = \ldots = P(x_i = r - 1) = \frac{1}{r}$ for $r$-valued input functions.

\(^2\)When $n$ is an odd number, we use a dummy variable.
1: pairing\_APL (BDD, binary variables $X$, # inputs $n$) \{ 
2: $X_{\text{sub}} = \emptyset$ ;
3: cost[$X_{\text{sub}}$] = 0 ;
4: order[$X_{\text{sub}}$] = $\emptyset$ ;
5: $S_{\text{next}} = \{X_{\text{sub}}\}$ ;
6: pairing($X$, order[$X$]) ;
7: min\_apl = APL for initial MDD obtained by pairing ;
8: for (level = 1; level $\leq n$; level += 2) \{ 
9: $S_{\text{cur}} = S_{\text{next}}$ ;
10: $S_{\text{next}} = \emptyset$ ;
11: for (each $X_{\text{sub}} \in S_{\text{cur}}$) \{ 
12: ordering(BDD, order[$X_{\text{sub}}$], level - 1) ;
13: for (each $x_i \in \{X \setminus X_{\text{sub}}\}$) \{ 
14: $X'_{\text{sub}} = X_{\text{sub}} \cup \{x_i\}$ ;
15: Move $x_i$ to level ;
16: symmetry\_check(BDD, level) ;
17: for (each $x_j \in \{X \setminus X'_{\text{sub}}\}$) \{ 
18: $X''_{\text{sub}} = X'_{\text{sub}} \cup \{x_j\}$ ;
19: Move $x_j$ to level + 1 ;
20: symmetry\_check(BDD, level + 1) ;
21: if (order[$X''_{\text{sub}}$] and current order are symmetric & all $P(x = c)$s are same) 
22: continue ;
23: Update min\_apl ;
24: if (lower\_bound(level + 1) > min\_apl) 
25: continue ;
26: new\_cost = cost[$X'_{\text{sub}}$] + pairing\_NTP(level) ;
27: if (new\_cost < cost[$X''_{\text{sub}}$]) \{ 
28: cost[$X''_{\text{sub}}$] = new\_cost ;
29: order[$X''_{\text{sub}}$] = current order ;
30: if ($X''_{\text{sub}} \notin S_{\text{next}}$) 
31: $S_{\text{next}} = S_{\text{next}} \cup \{X''_{\text{sub}}\}$ ;
32: \}
33: \}
34: \}
35: \}
36: \}
37: ordering(BDD, order[$X$], $n$) ;
38: pairing($X$, order[$X$]) ;
39: \}

Figure 8: Exact paired ordering algorithm for APL minimization.

### 6.1 Binary Case

Table 1 compares the number of nodes and APL of BDDs optimized using four different methods: (a) exact minimization of the number of nodes; (b) exact minimization of the APL; (c) the algorithm in [16]; and (d) the heuristic APL minimization algorithm presented in this paper. In the table, Name lists the names of benchmark functions. In and Out lists the numbers of input variables and single-output functions, respectively. Columns Nodes contain the number of non-terminal nodes. Columns Time contain the CPU time of three algorithms coded by us, in seconds. Unfortunately, the CPU time of the algorithm in [16] is unavailable. Columns “(a) Min\_Nodes”, “(b) Min\_APL”, “(c) Liu [16]”; and “(d) sifting” show the exact nodes minimization algorithm in [9], the exact APL minimization algorithm in Section 4.3, the heuristic APL minimization in [16], and the heuristic APL minimization in Section 4.4, respectively. Initial variable order for “(d) sifting” was obtained using Walsh spectrum described in Section 4.5. The BDDs in this table use complemented edges. Table 1 includes the same benchmark functions as the experiment in [16] except for incompletely specified functions. We omitted incompletely specified functions because the number of nodes and the APL in BDDs for incompletely specified functions depend on the
Table 1: Minimization of APL for individual BDDs

<table>
<thead>
<tr>
<th>Name</th>
<th>In</th>
<th>Out</th>
<th>(a) Min_Nodes</th>
<th>(b) Min_APL</th>
<th>(c) Liu [16]</th>
<th>(d) Sifting</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Nodes</td>
<td>APL</td>
<td>Time</td>
<td>Nodes</td>
</tr>
<tr>
<td>5xp1</td>
<td>7</td>
<td>10</td>
<td>66</td>
<td>34.13</td>
<td>0.01</td>
<td>81</td>
</tr>
<tr>
<td>alu4</td>
<td>14</td>
<td>8</td>
<td>448</td>
<td>41.75</td>
<td>22.76</td>
<td>547</td>
</tr>
<tr>
<td>b12</td>
<td>15</td>
<td>9</td>
<td>64</td>
<td>23.86</td>
<td>0.03</td>
<td>68</td>
</tr>
<tr>
<td>con1</td>
<td>7</td>
<td>2</td>
<td>14</td>
<td>6.06</td>
<td>0.01</td>
<td>16</td>
</tr>
<tr>
<td>cordic</td>
<td>23</td>
<td>2</td>
<td>73</td>
<td>13.74</td>
<td>416.57</td>
<td>89</td>
</tr>
<tr>
<td>sao2</td>
<td>10</td>
<td>4</td>
<td>99</td>
<td>10.90</td>
<td>0.26</td>
<td>116</td>
</tr>
<tr>
<td>vg2</td>
<td>25</td>
<td>8</td>
<td>202</td>
<td>31.00</td>
<td>6431.83</td>
<td>222</td>
</tr>
<tr>
<td>misex1</td>
<td>8</td>
<td>7</td>
<td>54</td>
<td>23.22</td>
<td>0.01</td>
<td>57</td>
</tr>
<tr>
<td>cm150a</td>
<td>21</td>
<td>1</td>
<td>32</td>
<td>3.50</td>
<td>1106.23</td>
<td>32</td>
</tr>
<tr>
<td>cm151a</td>
<td>12</td>
<td>2</td>
<td>32</td>
<td>6.00</td>
<td>0.38</td>
<td>32</td>
</tr>
<tr>
<td>cm162a</td>
<td>14</td>
<td>5</td>
<td>41</td>
<td>11.76</td>
<td>0.06</td>
<td>52</td>
</tr>
<tr>
<td>cm163a</td>
<td>16</td>
<td>5</td>
<td>35</td>
<td>11.70</td>
<td>0.01</td>
<td>38</td>
</tr>
<tr>
<td>cm85a</td>
<td>11</td>
<td>3</td>
<td>38</td>
<td>7.72</td>
<td>0.05</td>
<td>38</td>
</tr>
<tr>
<td>mux</td>
<td>21</td>
<td>1</td>
<td>32</td>
<td>3.50</td>
<td>1098.72</td>
<td>32</td>
</tr>
<tr>
<td>z4ml</td>
<td>7</td>
<td>4</td>
<td>28</td>
<td>18.25</td>
<td>0.01</td>
<td>30</td>
</tr>
<tr>
<td>ft1m</td>
<td>8</td>
<td>8</td>
<td>51</td>
<td>28.08</td>
<td>0.01</td>
<td>65</td>
</tr>
<tr>
<td>pcle</td>
<td>19</td>
<td>9</td>
<td>79</td>
<td>22.50</td>
<td>0.11</td>
<td>84</td>
</tr>
<tr>
<td>Average</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.12</td>
<td>0.95</td>
<td>0.93</td>
</tr>
</tbody>
</table>
Table 2: Minimization of APL for shared BDDs for larger functions

| Name | In | Out | classical sifting |  | Coef. | | Nodes | APL | Time | | Nodes | APL | Time | | Nodes | APL | Time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| C432 | 36 | 7 | 1063 | 86.58 | 0.01 | 1081 | 86.24 | 0.15 | 1899 | 82.09 | 0.83 |
| C499 | 41 | 32 | 25873 | 782.66 | 0.02 | 32105 | 641.16 | 7.12 | 32105 | 641.16 | 7.11 |
| C800 | 60 | 33 | 4122 | 140.42 | 0.01 | 41701 | 123.85 | 4.48 | 91767 | 122.22 | 52.12 |
| C908 | 33 | 25 | 5532 | 284.65 | 0.01 | 16634 | 179.20 | 0.96 | 13868 | 171.96 | 2.73 |
| C2670 | 233 | 140 | 1882 | 303.34 | 0.05 | 2755 | 278.17 | 1.30 | * | * | * |
| C3540 | 50 | 22 | 24231 | 208.44 | 0.10 | 25162 | 208.44 | 7.44 | 56989 | 212.73 | 75.21 |
| C5315 | 178 | 123 | 1728 | 460.78 | 0.05 | 1820 | 446.26 | 0.26 | * | * | * |
| C7552 | 207 | 108 | 2212 | 485.03 | 0.05 | 2207 | 471.54 | 0.87 | * | * | * |
| apex3 | 54 | 50 | 931 | 188.58 | 0.01 | 900 | 158.82 | 0.04 | 905 | 158.73 | 0.03 |
| apex7 | 49 | 37 | 242 | 113.88 | 0.01 | 277 | 82.44 | 0.01 | 280 | 82.45 | 0.02 |
| b9 | 41 | 21 | 108 | 61.16 | 0.01 | 131 | 55.25 | 0.01 | 129 | 55.39 | 0.01 |
| dalu | 75 | 16 | 688 | 102.67 | 0.01 | 990 | 78.81 | 0.08 | 1069 | 78.81 | 35.31 |
| des | 256 | 245 | 3297 | 1209.50 | 0.18 | 3343 | 1081.13 | 0.47 | 3886 | 1077.63 | 2.15 |
| duke2 | 22 | 29 | 360 | 87.89 | 0.01 | 386 | 77.52 | 0.01 | 392 | 77.52 | 0.02 |
| e64 | 65 | 50 | 128 | 128.00 | 0.01 | 128 | 128.00 | 0.01 | 573 | 128.00 | 0.05 |
| ex4 | 128 | 28 | 497 | 51.38 | 0.01 | 629 | 47.26 | 0.02 | 630 | 47.26 | 0.03 |
| frg2 | 143 | 139 | 1379 | 607.00 | 0.04 | 1580 | 322.89 | 0.15 | 2189 | 321.75 | 0.23 |
| k2 | 45 | 45 | 1257 | 181.80 | 0.01 | 1426 | 177.52 | 0.07 | 1418 | 177.50 | 0.10 |
| rot | 135 | 107 | 7891 | 446.47 | 0.05 | 16164 | 312.08 | 5.61 | 18503 | 308.68 | 30.34 |

| Name | In | Out | classical sifting |  | Coef. | | Nodes | APL | Time | | Nodes | APL | Time | | Nodes | APL | Time |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

Average | 1.00 | 1.00 | 0.03 | 1.87 | 0.85 | 1.53 | 3.01 | 0.84 | 12.89 |

* Memory overflow precluded computation of these values.
assignment of values to don’t cares, as well as the variable order. To make our results compatible with the results in [16], we optimized each output of the multiple-output benchmark functions independently, and obtained the sum of the values over all outputs. Thus, the number of nodes and APL in Table 1 are different from those of the SBDD. Two rounds of sifting are performed in all experiments. The row labeled Average of ratios represents the normalized averages for Nodes, APL, and Time assuming the values of “(a) Min\_Nodes” to be 1.00. The columns “(b) Min\_APL”, “(c) Liu [16]”, and “(d) sifting” of this row contains the relative values to the results of “(a) Min\_Nodes”.

The heuristic method in [16] obtained BDDs with the exact minimum APLs in 5 out of 17 benchmark functions. However, for alu4, cm151a, and cm85a, the algorithm in [16] obtained BDDs with much larger APLs than the exact minimum APLs. On the other hand, our heuristic method in Section 4.4 obtained BDDs with the exact minimum APLs in 11 out of 17 benchmark functions. For five of the remaining functions, the APLs in the column labeled “(d) sifting” are smaller than or equal to the APLs in “(c) Liu [16]”. For cm162a, our sifting algorithm obtained BDDs with slightly larger APLs than the exact minimum APLs.

An exhaustive search algorithm finds the minimum APLs for the functions with up to 14 inputs within a reasonable computation time. Meanwhile, our exact minimization algorithm in Section 4.3 found the minimum APL for functions with 25 inputs (vg2) within a reasonable computation time.

Table 2 shows the results for larger MCNC benchmarks and the effectiveness of the initial variable order using the Walsh spectrum. In this table, we used SBDDs with complemented edges for multiple-output functions. In Table 2, the column “classical sifting” shows the number of nodes and APL for BDDs obtained by the sifting algorithm [23] which minimizes the number of nodes in BDD. The column “Without Walsh spectrum” shows the results of our sifting algorithm, which minimizes the APL, where the initial variable orders are the variable orders of BDDs obtained by “classical sifting”. And, the column “With Walsh spectrum” shows the results of our sifting algorithm, where the initial variable orders were obtained using Walsh spectrum shown in Section 4.5. The column “Coeff. Time” denotes the CPU time needed to calculate the values of first-order Walsh spectral coefficients $R_i$, in seconds. Unfortunately, for $C2670$, $C5315$, and $C7552$, BDDs with the initial variable orders could not be constructed due to memory overflow. The row labeled Average represents average of Time and normalized averages of Nodes and APL assuming the values of “classical sifting” to be 1.00. The columns “Without Walsh spectrum” and “With Walsh spectrum” show the relative values to the results of “classical sifting”.

For some benchmark functions, for example, $C1908$, $frg2$, and $rot$, the APLs are reduced drastically. For $C7552$, the number of nodes is reduced as a byproduct of the APL minimization. However, for most functions, the number of nodes is increased by the APL minimization. The comparison of “Without Walsh spectrum” and “With Walsh spectrum” shows the effectiveness of the initial variable order using Walsh spectrum. For 8 out of 19 benchmark functions, the APLs in the column “With Walsh spectrum” are smaller than the APLs in “Without Walsh spectrum”. The computation time to calculate the values of $R_i$ is short.

However, for most functions, the computation times of sifting for “With Walsh spectrum” are significantly longer than that for “Without Walsh spectrum” because the number of nodes in BDD with initial variable order computed using Walsh spectrum is large. When the number of nodes in the BDD is large, swapping one pair of adjacent variables takes a longer time because the time needed for the swap is roughly proportional to the number of nodes present on the given levels in the BDD.

Tables 1 and 2 show that the proposed heuristic minimization minimizes the APL in short computation time. For small benchmark functions in Table 1, the heuristic minimization could obtain BDDs with near-minimum APLs. For large benchmark functions in Table 2, the heuristic algorithm reduces APLs to 84% on the average.

### 6.2 Multi-Valued Case

There are no standard benchmark functions for multi-valued logic. Thus, by pairing binary variables of 2-valued benchmark functions, we obtained 4-valued input 2-valued output functions. Table 3 compares the number of nodes and the APLs of the BDDs and the MDDs optimized using four algorithms: (a) exact minimization of the APL for BDDs; (b) exact minimization of the number of nodes for MDDs; (c) exact minimization of the APL for MDDs; and (d) the heuristic APL minimization algorithm for MDDs. Columns “BDD”, “Min\_Nodes”, “Min\_APL”, and “pair-sifting” denote the exact APL minimization algorithm in Section 4.3, the exact paired ordering algorithm for node minimization, the exact paired ordering algorithm for APL minimization in Section 5.1, and the pair-sifting algorithm in Section 5.2, respectively. The symbol “*” in this table denotes that the results could not be obtained because of memory overflow. The bottom row labeled Average represents normalized averages of Nodes, APL, and Time for all functions except for 4 functions (cordic, cm150a, mux, pele), where the values of “BDD” are set to 1.00. The SBDDs and SMDDs in this table do not use complemented edges. Note that the values (Nodes, APL, Time) of BDDs in this table are different from the values in Table 1, because in this table, SBDDs
Table 3: Comparison of APLs for SBDDs and SMDDs

<table>
<thead>
<tr>
<th>Name</th>
<th>BDD</th>
<th></th>
<th></th>
<th></th>
<th>MDD</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nodes</td>
<td>APL</td>
<td>Time</td>
<td>Min_Nodes</td>
<td>APL</td>
<td>Time</td>
<td>Min_APL</td>
<td>pair-sifting</td>
</tr>
<tr>
<td>---------</td>
<td>-------</td>
<td>---------</td>
<td>---------</td>
<td>-----------</td>
<td>-----</td>
<td>---------</td>
<td>---------</td>
<td>-------------</td>
</tr>
<tr>
<td>5xp1</td>
<td>68</td>
<td>32.00</td>
<td>0.01</td>
<td>37</td>
<td>20.69</td>
<td>0.01</td>
<td>37</td>
<td>20.69</td>
</tr>
<tr>
<td>alu4</td>
<td>462</td>
<td>40.70</td>
<td>47.96</td>
<td>37</td>
<td>20.69</td>
<td>0.01</td>
<td>37</td>
<td>27.00</td>
</tr>
<tr>
<td>b12</td>
<td>66</td>
<td>22.77</td>
<td>2.26</td>
<td>37</td>
<td>20.69</td>
<td>0.01</td>
<td>37</td>
<td>19.00</td>
</tr>
<tr>
<td>con1</td>
<td>18</td>
<td>6.31</td>
<td>0.01</td>
<td>46</td>
<td>9.98</td>
<td>0.21</td>
<td>58</td>
<td>6.57</td>
</tr>
<tr>
<td>cordic</td>
<td>94</td>
<td>9.46</td>
<td>39.96</td>
<td>37</td>
<td>20.69</td>
<td>0.01</td>
<td>37</td>
<td>15.41</td>
</tr>
<tr>
<td>sao2</td>
<td>102</td>
<td>10.64</td>
<td>0.12</td>
<td>22</td>
<td>15.50</td>
<td>0.01</td>
<td>28</td>
<td>12.75</td>
</tr>
<tr>
<td>misex1</td>
<td>46</td>
<td>22.84</td>
<td>0.02</td>
<td>46</td>
<td>9.98</td>
<td>0.21</td>
<td>58</td>
<td>6.57</td>
</tr>
<tr>
<td>cm150a</td>
<td>32</td>
<td>3.50</td>
<td>59.72</td>
<td>37</td>
<td>7.72</td>
<td>0.08</td>
<td>15</td>
<td>6.28</td>
</tr>
<tr>
<td>cm151a</td>
<td>32</td>
<td>6.00</td>
<td>0.47</td>
<td>16</td>
<td>5.50</td>
<td>0.13</td>
<td>22</td>
<td>4.00</td>
</tr>
<tr>
<td>cm162a</td>
<td>38</td>
<td>11.70</td>
<td>0.86</td>
<td>19</td>
<td>8.73</td>
<td>0.55</td>
<td>27</td>
<td>8.44</td>
</tr>
<tr>
<td>cm163a</td>
<td>40</td>
<td>11.70</td>
<td>1.98</td>
<td>19</td>
<td>12.25</td>
<td>19.60</td>
<td>22</td>
<td>8.16</td>
</tr>
<tr>
<td>cm85a</td>
<td>37</td>
<td>7.72</td>
<td>0.08</td>
<td>15</td>
<td>6.28</td>
<td>0.08</td>
<td>16</td>
<td>5.38</td>
</tr>
<tr>
<td>mux</td>
<td>32</td>
<td>3.50</td>
<td>6334</td>
<td>10</td>
<td>9.13</td>
<td>0.01</td>
<td>10</td>
<td>9.13</td>
</tr>
<tr>
<td>z4ml</td>
<td>26</td>
<td>16.38</td>
<td>0.01</td>
<td>39</td>
<td>18.81</td>
<td>0.02</td>
<td>41</td>
<td>17.38</td>
</tr>
<tr>
<td>f51m</td>
<td>76</td>
<td>28.02</td>
<td>0.03</td>
<td>50</td>
<td>16.50</td>
<td>0.01</td>
<td>48</td>
<td>20.92</td>
</tr>
<tr>
<td>pcle</td>
<td>83</td>
<td>22.50</td>
<td>34.46</td>
<td>50</td>
<td>16.50</td>
<td>0.01</td>
<td>48</td>
<td>20.92</td>
</tr>
<tr>
<td>Average</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>0.50</td>
<td>0.79</td>
<td>15.89</td>
<td>0.59</td>
<td>0.64</td>
</tr>
</tbody>
</table>

* Memory overflow precluded computation of these values.
Table 4: Minimization of APL for SMDDs for larger functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Node pair-sifting [24]</th>
<th>APL pair-sifting</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nodes</td>
<td>APL</td>
</tr>
<tr>
<td>C432</td>
<td>617</td>
<td>59.84</td>
</tr>
<tr>
<td>C499</td>
<td>13541</td>
<td>407.23</td>
</tr>
<tr>
<td>C880</td>
<td>3025</td>
<td>118.99</td>
</tr>
<tr>
<td>C1908</td>
<td>4390</td>
<td>167.42</td>
</tr>
<tr>
<td>C2670</td>
<td>2336</td>
<td>276.19</td>
</tr>
<tr>
<td>C3540</td>
<td>2292</td>
<td>155.06</td>
</tr>
<tr>
<td>C5315</td>
<td>1947</td>
<td>398.53</td>
</tr>
<tr>
<td>C7552</td>
<td>22519</td>
<td>155.06</td>
</tr>
<tr>
<td>apex3</td>
<td>628</td>
<td>143.66</td>
</tr>
<tr>
<td>apex7</td>
<td>200</td>
<td>99.82</td>
</tr>
<tr>
<td>b9</td>
<td>126</td>
<td>55.90</td>
</tr>
<tr>
<td>dalu</td>
<td>523</td>
<td>70.55</td>
</tr>
<tr>
<td>des</td>
<td>2685</td>
<td>934.38</td>
</tr>
<tr>
<td>duke2</td>
<td>272</td>
<td>65.99</td>
</tr>
<tr>
<td>e64</td>
<td>96</td>
<td>96.67</td>
</tr>
<tr>
<td>ex4</td>
<td>420</td>
<td>46.00</td>
</tr>
<tr>
<td>frg2</td>
<td>1179</td>
<td>544.34</td>
</tr>
<tr>
<td>k2</td>
<td>1055</td>
<td>168.17</td>
</tr>
<tr>
<td>rot</td>
<td>5615</td>
<td>393.57</td>
</tr>
<tr>
<td>Average</td>
<td>1.00</td>
<td>1.00</td>
</tr>
</tbody>
</table>

The pair-sifting algorithm obtained MDDs with the exact minimum APL for 4 functions. On the average, the pair-sifting algorithm reduced the APL to 70% of “BDD”. For con1, the pair-sifting algorithm obtained larger APL than that of “Min Nodes” due to heuristic pairing algorithm. However, this algorithm can obtain a smaller APL and fewer nodes than those of the corresponding BDD. Although the exact paired ordering algorithms for nodes and APL can reduce both nodes and APL drastically, they are time-consuming. On the other hand, the pair-sifting algorithm quickly reduces both Nodes and APL.

Table 4 shows the results for larger MCNC benchmark functions. Similarly, we obtained 4-valued input 2-valued output functions by pairing binary variables. Column “Node pair-sifting [24]” denotes the heuristic paired ordering algorithm for the node minimization method proposed in [24]. The number of nodes in an MDD obtained by the paired ordering algorithm for node minimization is smaller than or equal to the corresponding BDD [24]. However, since the MDDs in this table do not use the complemented edges, some MDDs are larger than the BDDs with complemented edges in Table 2. The bottom row labeled Average represents normalized averages of Nodes, APL, and Time assuming the values of “Node pair-sifting [24]” to be 1.00.

For these benchmark functions, the pair-sifting algorithm reduced the APL to 83% of “Node pair-sifting [24]”, on average. Especially, for frg2, the APL was reduced to 48% of “Node pair-sifting [24]”. The pair-sifting algorithm cannot always find an MDD with the minimum APL, because it is a heuristic algorithm. For C3540 and C7552, the APLs are slightly larger than that in “Node pair-sifting [24]”. However, Tables 2 and 4 show that the pair-sifting algorithm can find an MDD with smaller APL than APL of corresponding BDD.

7 CONCLUSION AND COMMENTS

We have proposed an exact and a heuristic algorithm for the minimization of the APL in BDDs and MDDs. The experimental results using MCNC benchmark functions show that: 1) The exact minimization algorithm finds BDDs with the minimum APL for the function with up to 25 input variables within a reasonable computation time. 2) Using the node and edge traversing probabilities to compute and update the APLs after the swap of two adjacent variables, the proposed sifting algorithm can heuristically minimize the APLs as fast as classical sifting, which minimizes the number of nodes. 3) Using an initial variable order computed using Walsh spectral coefficients increases the quality of the results of APL minimization algorithms. However, in some cases the initial variable order leads to BDDs with a large number of nodes, which slows down
APL minimization. 4) MDDs produced by pairing binary variables have smaller APL and fewer nodes than corresponding BDDs.

ACKNOWLEDGMENTS

This research is partly supported by the Grant in Aid for Scientific Research of the Japan Society for the Promotion of Science (JSPS), and the funds from Ministry of Education, Culture, Sports, Science, and Technology (MEXT) via Kitakyushu innovative cluster project. We also thank the reviewers for constructive comments.

REFERENCES

APPENDIX

Proof for Lemma 2.1. We prove only the first statement; the proof for the second statement is similar. Consider a node \( v \). Any path that includes an incoming edge to \( v \) includes \( v \). Conversely, any path that includes \( v \) includes an incoming edge to \( v \). It follows that any assignment of values to the variables that corresponds to a path through \( v \) contributes to the node traversing probability of \( v \) an amount that is identical to the amount contributed to the edge traversing probability of an incoming edge to \( v \). It follows that the node traversing probability of \( v \) is equal to the sum of edge traversing probabilities of all incoming edges to \( v \). \( \square \)

Proof for Theorem 2.1. We prove only the first statement; the proof for the second statement is similar. From Definition 2.6, we have

\[
ETP(e) = \sum_{p \in SP(e)} PP(p),
\] (1)
where \(SP(e)\) is a set of paths including the edge \(e\). We prove the following

\[
APL = \sum_{i=1}^{N_e} ETP(e_i),
\]

(2)

where \(N_e\) denotes the number of edges in a DD. From (1), (2) can be transformed as follows:

\[
APL = \sum_{i=1}^{N_e} \sum_{p \in SP(e_i)} PP(p)
\]

(3)

From Definition 2.4, we have

\[
APL = \sum_{i=1}^{N} PP(p_i) \times l_i
\]

\[
= \sum_{i=1}^{N} \sum_{j=1}^{l_i} PP(p_i)
\]

(4)

Although (3) and (4) use different computational approaches, they obviously compute the same value.

Proof for Lemma 3.1. An SMDD for \(F = (f_0, f_1, \ldots, f_m)\) is traversed \(m_U \cdot U\) times to evaluate multiple-output function \(F\). Since \(m_U\) root nodes are located above or in level \(i\), \(m_U\) traversals via edges in \(Cut(i)\) are performed while evaluating the multiple-output function. Therefore, we have \(ETP(Cut(i)) = m_U\).

Proof for Lemma 3.2. From Lemma 2.1, the following relation holds:

\[
ETP(Cut'(i)) = \sum_{v \in V_c} NTP(v),
\]

where \(V_c\) denotes a set of non-terminal nodes representing the cofactors with respect to \(X_{upper}\). The probability of the occurrence of the cofactor depends only on the function and not the order of \(X_{upper}\). Since \(Cut'(i)\) does not include the edges to terminal nodes, the upper bound of \(m_U\) on \(c_i\) follows from Lemma 3.1.

Proof for Theorem 3.1. All nodes representing cofactors with respect to the variables in \(X_{upper}\) and \(m_L\) root nodes are situated below or in level \(i+1\). Thus, \(L\) includes the node traversing probabilities of these nodes.

Proof for Theorem 3.2. Let \(L\) be the sum of the node traversing probabilities of the non-terminal nodes below or in level \(i+1\). From Theorem 2.1, we have

\[
APL = U + L.
\]

Then, from Theorem 3.1, for any permutation of \(X_{lower}\),

\[
APL \geq U + ETP(Cut'(i)) + m_L.
\]

Proof for Theorem 4.1. The variable swap of level \(i\) and level \(i+1\) does not influence the graph structure except for levels \(i\) and \(i+1\) because of the locality of the swap operation. Thus, it is clear that \(U\) remains unchanged. From Lemma 2.1, \(L\) is obtained by the sum of \(ETP(Cut'(i+1))\) and \(ETP(E_{lower})\), where

\[
Cut'(i+1) = \{e \mid e \in Cut(i+1), e \text{ is incident to a non-terminal node}\},
\]

\[
E_{lower} = \{e \mid e \text{ is an edge situated below or in level } i+2\},
\]

\[
ETP(Cut'(i+1)) = \sum_{e \in Cut'(i+1)} ETP(e),
\]

\[
ETP(E_{lower}) = \sum_{e \in E_{lower}} ETP(e).
\]
By Lemma 3.2, $ETP(Cut'(i+1))$ is an invariant. $ETP(E_{lower})$ remains unchanged because of the invariance of $ETP(Cut'(i+1))$ and the locality of the swap operation. Therefore, $L$ also remains unchanged. □

**Proof for Theorem 5.1.** The number of different permutations of binary variables $X$ is $n!$. Since from Definition 5.1, the binary variables $X$ are partitioned into the unordered sets $\{X_1\}, \{X_2\}, \ldots, \{X_n\}$, the order of binary variables in each $\{X_i\}$ is not important. The number of different permutations of two binary variables in each $\{X_i\}$ is 2. Therefore, we have the theorem. □