# Representations and Evaluations of Logic Functions using Multi-valued Decision Diagrams

Shinobu Nagayama

# Abstract

Binary decision diagrams (BDDs) and multi-valued decision diagrams (MDDs) are extensively used in logic synthesis, formal verification, logic simulation, pass transistor logic (PTL), software synthesis, etc.. In these applications using decision diagrams (DDs), proper optimizations of DDs are required to reduce the memory sizes and runtimes. Particularly, in software synthesis, intensive optimization of DDs is required to generate a compact and fact program code. The purpose of most optimization algorithms for DDs is minimization of the number of nodes in DDs. Minimization of the number of nodes results in reduction of memory size. However, logic simulation and software synthesis require shorter evaluation time of logic functions, as well as smaller memory size. In evaluation of logic functions using DDs, the evaluation time depends on the path length of DDs. Therefore, in logic simulation and software synthesis, minimization of the path length is important, as well as minimization of the number of nodes. This thesis proposes the optimization algorithms for DDs that minimize the memory size, average path length (APL), or both of them.

Since the graph structures of DDs depend on the variable order, the number of nodes and APLs for DDs can be reduced by changing variable order. Chapter 3 proposes APL minimization algorithms for DDs considering only variable orderings. The APL minimization algorithms proposed in Chapter 3 yield an improvement over an existing algorithm in both APL and runtime. However, the APL minimization algorithms considering only variable orderings often increase the number of nodes, since a variable order that minimizes the APL is often different from the variable order that minimizes the number of nodes.

Next, we use MDDs to reduce the memory sizes and APLs furthermore. MDDs are usually used to represent multi-valued logic functions. However, we use MDDs to represent binary logic functions. When MDDs are used to represent binary logic functions, we can use an additional optimization approach, which is a partition of binary variables. To represent binary logic functions using MDDs, we partition the binary variables into groups, and we treat each group as a multi-valued variable. Chapter 4 shows the relations between the values of $k$ and the number of nodes, memory size, path length, and area-time complexity for quasi-reduced

MDD($k$) (QRMDD($k$)), and derives the optimum values of $k$ for each application. For many benchmark functions, the numbers of nodes and path lengths for QRMDD($k$)s are inversely proportional to the value of $k$. Therefore, the numbers of nodes for QRMDD($k$)s can be reduced with increasing the value of $k$. However, the memory size of each node in QRMDD($k$) increases with $2^k$. By experiments, we show that the memory sizes for QRMDD($k$)s take their minimum when $k = 2$. To obtain the optimum values of $k$ considering both memory size and path length, we introduce the area-time complexity. By experiments, we show that when both the memory size and path length are equally important, the optimum value of $k$ is 3 or 4. On the other hand, when the path length is more important than the memory size, the optimum value of $k$ is 4, 5 or 6.

In MDD($k$)s representing binary logic functions, the binary variables are partitioned into the groups with $k$ binary variables. On the other hand, in heterogeneous MDDs, the binary variables can be partitioned into the groups with different numbers of binary variables. Therefore, the memory sizes and APLs of heterogeneous MDDs depend on the partition of binary variables, as well as the order of binary variables. Chapter 5 proposes the memory size and APL minimization algorithms for heterogeneous MDDs that consider both orderings and partitions of binary variables. By considering both orderings and partitions of binary variables, heterogeneous MDDs can represent logic functions with smaller memory sizes than free BDDs (FBDDs) and smaller APLs than ordered BDDs (OBDDs), and the APLs of heterogeneous MDDs can be reduced by a half of BDDs without increasing memory size. Heterogeneous MDDs have smaller area-time complexities than MDD($k$)s, since heterogeneous MDDs allow more flexible partition of binary variables than MDD($k$)s.

# List of Publications by Author

## Journal Papers

1. Shinobu Nagayama and Tsutomu Sasao, "Compact representations of logic functions using heterogeneous MDDs," *IEICE Transactions on Fundamentals of Electronics*, Vol. E86-A, No. 12, pp. 3168–3175, December 2003.

2. Shinobu Nagayama, Tsutomu Sasao, Yukihiro Iguchi, and Munehiro Matsuura, "Area-time complexities of multi-valued decision diagrams," *IEICE Transactions on Fundamentals of Electronics*, Vol. E87-A, No. 5, pp. 1020–1028, May 2004.

3. Shinobu Nagayama, Alan Mishchenko, Tsutomu Sasao, and Jon T. Butler, "Exact and heuristic minimization of the average path length in decision diagrams," *Journal of Multiple-Valued Logic and Soft Computing*, (accepted for publication).

4. Hui Qin, Tsutomu Sasao, Munehiro Matsuura, Shinobu Nagayama, Kazuyuki Nakamura, and Yukihiro Iguchi, "A realization of multiple-output functions by a look-up table ring," *IEICE Transactions on Fundamentals of Electronics*, Vol. E87-A, No. 12, December 2004, (accepted for publication).

## International Conference and Workshop Papers

1. Tsutomu Sasao, Munehiro Matsuura, Yukihiro Iguchi, and Shinobu Nagayama "Compact BDD representations for multiple-output functions and their applications to embedded system," *IFIP VLSI-SOC'01*, pp. 406–411, December 2001.

2. Shinobu Nagayama, Tsutomu Sasao, Yukihiro Iguchi, and Munehiro Matsuura, "Representations of logic functions using QRMDDs," *32nd IEEE International Symposium on Multiple-Valued Logic (ISMVL 2002)*, pp. 261–267, May 2002.

3. Shinobu Nagayama and Tsutomu Sasao, "Code generation for embedded systems using heterogeneous MDDs," *The 11th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI 2003)*, pp. 258–264, April 2003.

4. Shinobu Nagayama and Tsutomu Sasao, "Compact representations of logic functions using heterogeneous MDDs," *33rd IEEE International Symposium on Multiple-Valued Logic (ISMVL 2003)*, pp. 247–255, May 2003.

5. Shinobu Nagayama, Alan Mishchenko, Tsutomu Sasao, and Jon T. Butler, "Minimization of average path length in BDDs by variable reordering," *12th International Workshop on Logic and Synthesis (IWLS 2003)*, pp. 207–213, May 2003.

6. Shinobu Nagayama and Tsutomu Sasao, "Minimization of memory size for heterogeneous MDDs," *Asia and South Pacific Design Automation Conference (ASP-DAC'2004)*, pp. 872–875, January 2004.

7. Shinobu Nagayama and Tsutomu Sasao, "On the minimization of average path lengths for heterogeneous MDDs," *34th IEEE International Symposium on Multiple-Valued Logic (ISMVL 2004)*, pp. 216–222, May 2004.

8. Shinobu Nagayama and Tsutomu Sasao, "On the minimization of longest path length for decision diagrams," *13th International Workshop on Logic and Synthesis (IWLS 2004)*, pp. 28–35, June 2004.

## Japanese Domestic Workshop Papers

1. Munehiro Matsuura, Tsutomu Sasao, Yukihiro Iguchi, and Shinobu Nagayama, "Compact representations of BDDs for multiple-output functions and their optimization," (in Japanese) *IEICE Technical Report*, VLD2001-100, November 2001.

2. Shinobu Nagayama, Tsutomu Sasao, Yukihiro Iguchi, and Munehiro Matsuura, "Representations of logic functions using QRMDDs," (in Japanese) *IEICE Technical Report*, VLD2001-142, January 2002.

3. Shinobu Nagayama and Tsutomu Sasao, "Compact representations of logic functions using heterogeneous MDDs," (in Japanese) *IEICE Technical Report*, VLD2002-98, pp. 97–102, November 2002.

4. Qin Hui, Tsutomu Sasao, Munehiro Matsuura, Shinobu Nagayama, Kazuyuki Nakamura, and Yukihiro Iguchi, "On a sequential look-up table cascade," *7th System LSI workshop*, pp. 311–314, November 2003.

5. Shinobu Nagayama and Tsutomu Sasao, "Minimization of average path lengths for heterogeneous MDDs," (in Japanese) *IEICE Technical Report*, VLD2003-107, pp. 223–228, November 2003.

6. Qin Hui, Tsutomu Sasao, Munehiro Matsuura, Shinobu Nagayama, Kazuyuki Nakamura, and Yukihiro Iguchi, "Realization of multiple-output functions by sequential look-up table cascades," *IEICE Technical Report*, VLD2003-127, pp. 13–18, January 2004.

# Contents

**Abstract** i

**List of Publications by Author** iii

**List of Abbreviations and Symbols** ix

**1 Introduction** 1

1.1 Backgrounds and Purposes of Research . . . . . . . . . . . . . . . . . . . . . 1

1.2 Organization of Thesis . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 3

**2 Preliminary** 4

2.1 Logic Functions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 4

2.2 Partition of Binary Variables . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.3 Decision Diagrams (DDs) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.4 Average Path Lengths (APLs) . . . . . . . . . . . . . . . . . . . . . . . . . . 7

**3 Minimization of APL in DDs by Variable Ordering** 8

3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

3.2 Definitions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

3.3 Efficient Computation of APLs . . . . . . . . . . . . . . . . . . . . . . . . . 9

3.4 Lower Bounds on APL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

3.5 APL Minimization Algorithms . . . . . . . . . . . . . . . . . . . . . . . . . 13

3.5.1 Change of the APL during Swapping Two Adjacent Variables . . . . . 13

3.5.2 Symmetric Variables . . . . . . . . . . . . . . . . . . . . . . . . . . 16

3.5.3 Exact Minimization Algorithm . . . . . . . . . . . . . . . . . . . . . 17

3.5.4 Heuristic Minimization Algorithm . . . . . . . . . . . . . . . . . . . 19

3.5.5 Initial Ordering of the Binary Variables . . . . . . . . . . . . . . . . 21

3.6 Experimental Results . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

3.7 Conclusion and Comments . . . . . . . . . . 25

**4 Area-Time Complexities of QRMDD($k$)s** **26**

4.1 Introduction . . . . . . . . . . 26

4.2 Definitions . . . . . . . . . . 27

4.3 Number of Nodes in QRMDD($k$) . . . . . . . . . . 28

4.3.1 Number of Nodes for General Functions . . . . . . . . . . 29

4.3.2 Number of Nodes for Benchmark Functions . . . . . . . . . . 30

4.3.3 Number of Nodes for Randomly Generated Functions . . . . . . . . . . 33

4.4 Area-Time Complexity of QRMDD($k$) . . . . . . . . . . 34

4.4.1 Memory Size for QRMDD($k$) . . . . . . . . . . 34

4.4.2 Area-Time Complexity of QRMDD($k$)s . . . . . . . . . . 36

4.4.3 Experimental Results . . . . . . . . . . 36

4.4.4 Analysis for the Functions that Satisfy Property 4.1 . . . . . . . . . . 37

4.5 Conclusion and Comments . . . . . . . . . . 38

**5 Heterogeneous MDDs and Their Optimization Algorithms** **39**

5.1 Introduction . . . . . . . . . . 39

5.2 Definitions . . . . . . . . . . 40

5.3 Number of Heterogeneous MDDs . . . . . . . . . . 42

5.4 Memory Size Minimization Algorithms . . . . . . . . . . 44

5.4.1 Bounds on Memory Size of Heterogeneous MDDs . . . . . . . . . . 44

5.4.2 Partition Algorithm for Memory Size Minimization . . . . . . . . . . 45

5.4.3 Exact Memory Size Minimization Algorithm . . . . . . . . . . 47

5.4.4 Heuristic Memory Size Minimization Algorithm . . . . . . . . . . 48

5.5 APL Minimization Algorithms . . . . . . . . . . 48

5.5.1 Partition Algorithm for APL Minimization . . . . . . . . . . 50

5.5.2 Exact APL Minimization Algorithm . . . . . . . . . . 51

5.5.3 Heuristic APL Minimization Algorithm . . . . . . . . . . 52

5.6 Experimental Results . . . . . . . . . . 53

5.6.1 Comparison with FBDDs . . . . . . . . . . 55

5.6.2 Comparison with ROBDDs . . . . . . . . . . 57

5.6.3 Comparison of Computation Time for Algorithms . . . . . . . . . . 59

5.6.4 Comparison with MDD($k$)s . . . . . . . . . . 59

5.7 Conclusion and Comments . . . . . . . . . . 62

**6 Conclusion** **65**

**Acknowledgements** **67**

**References** **69**

**Appendix** **77**

A. Proofs of Theorems in Chapter 3 . . . . . . . . . . . . . . . . . . . . . . . . . 77

B. Proofs of Theorems in Chapter 4 . . . . . . . . . . . . . . . . . . . . . . . . . 80

C. Proofs of Theorems in Chapter 5 . . . . . . . . . . . . . . . . . . . . . . . . . 81

# List of Abbreviations and Symbols

## Abbreviations

| | |
|---|---|
| APL | Average Path Length |
| BDD | Binary Decision Diagram |
| CPU | Central Processing Unit |
| DAG | Directed Acyclic Graph |
| DD | Decision Diagram |
| ECFN | Encoded Characteristic Function for Non-zero output |
| ETP | Edge Traversing Probability |
| FBDD | Free Binary Decision Diagram |
| FPGA | Field Programmable Gate Array |
| LUT | Look-Up Table |
| MDD | Multi-valued Decision Diagram |
| NTP | Node Traversing Probability |
| OBDD | Ordered Binary Decision Diagram |
| PDA | Personal Digital Assistance |
| PP | Path Probability |
| PTL | Pass Transistor Logic |
| ROBDD | Reduced Ordered Binary Decision Diagram |
| ROMDD | Reduced Ordered Multi-valued Decision Diagram |
| QRBDD | Quasi-Reduced ordered Binary Decision Diagram |
| QRMDD | Quasi-Reduced ordered Multi-valued Decision Diagram |
| SBDD | Shared Binary Decision Diagram |
| SDD | Shared Decision Diagram |
| SMDD | Shared Multi-valued Decision Diagram |

# Symbols

| | |
|---|---|
| $f$ | single-output logic function |
| $F$ | multiple-output logic function |
| $m$ | the number of outputs |
| $n$ | the number of inputs or the number of variables |
| $x$ | variable |
| $X$ | ordered set of variables or super variable |
| $k$ | the size of a $X$ or the number of variables in a $X$ |
| $\vec{a}$ | vector $(a_1, a_2, \ldots, a_n)$ |
| $e$ | edge in a DD |
| $v$ | non-terminal node in a DD |
| $d$ | density for a logic function |
| $\eta$ | normalized difference |
| $a \simeq b$ | $a$ and $b$ are nearly equal |
| $\lceil a \rceil$ | the smallest integer that is larger than $a$ |
| $N_{fix}$ | the number of different fixed-order partitions |
| $N_{non-fix}$ | the number of different non-fixed-order partitions |
| $S_n$ | the number of different FBDDs |
| $O$ | order notation showing the computational complexity |

# Chapter 1

# Introduction

## 1.1 Backgrounds and Purposes of Research

Binary decision diagrams (BDDs) [11] and multi-valued decision diagrams (MDDs) [3, 26, 37, 38] are extensively used for representations of logic functions in logic synthesis [5, 17, 32, 42, 80], formal verification [15, 56, 57], logic simulation [1, 22, 34, 35], pass transistor logic (PTL) [4, 69, 70], software synthesis [2, 25, 27, 45], etc.. For example, in logic synthesis, decision diagrams (DDs) are used for compact representation of a given logic function, for efficient Boolean operations, and for functional decomposition [5, 11, 17, 32, 42, 80]. Since the memory size and runtime needed for logic synthesis depend on the sizes of DDs, minimization of the size of DDs is required to reduce them. In logic simulation [1, 22, 34, 35], DDs are used to evaluate the logic functions quickly. Since the evaluation time for logic simulation depends on the path length of DDs, minimization of path length of DDs is required to reduce the design verification time. In software synthesis [2, 25, 27, 45], DDs are used to generate a program code, such as branching program [78] that can be derived directly from DDs by translating each node in DDs into a fragment of program code. Since the sizes and runtimes for the generated codes depend on the sizes and path length of DDs, minimizations of size and path length of DDs are required to generate compact and fast codes. Particularly, in software synthesis for embedded systems (e.g. consumer electronics, vehicle control, personal digital assistance (PDA), cellular phone, etc..), intensive optimization of DDs is required to generate a code satisfying the memory size limitation and timing limitation for systems. Thus, in various applications, the optimizations of DDs are key issue. This thesis focuses on the optimizations of DDs that are useful for various applications. In optimizations of DDs, the following approaches are well-known.

- Variable ordering [14, 17, 18, 19, 24, 38, 39, 59, 74]

- Complemented edges [6, 40]
- Assignment of values to don't cares [33, 54, 71]

Among them, the most widely used and effective approach is variable ordering approach. The paper [11] shows that the size of BDDs can be varied from linear to exponential of the number of input variables by changing variable order. Therefore, considering the variable orderings is important and effective to optimize DDs.

The purpose of most existing optimization algorithms for DDs is minimization of the number of nodes in DDs [14, 17, 18, 19, 20, 24, 38, 39, 59, 74]. Although minimization of the number of nodes results in reduction of size for DDs, it is not directly related to the reduction of the evaluation time of logic functions. Since the logic functions that are represented using DDs are evaluated by traversing DDs from the root node to a terminal node, the evaluation time depends on the path length in DDs. Thus, minimization of path length is important to reduce the evaluation time of logic functions. For example, logic simulation requires shorter evaluation time of logic functions, as well as smaller size of DDs. Therefore, in such applications, minimization of the path length is important, as well as minimization of the number of nodes. Since the graph structures of DDs depend on the variable order, the number of nodes and path length in DDs can be reduced by changing the variable order. This thesis proposes the algorithms for minimization of path length in DDs considering only the variable orderings.

MDDs are usually used to represent multi-valued logic functions, and are usually optimized by changing variable order [38, 39]. However, when MDDs are used to represent binary logic functions, an additional optimization approach can be used. To represent binary logic functions using MDDs, binary variables are partitioned into some groups, and each group is treated as a multi-valued variables. In this case, the graph structures of MDDs depend on the size of groups (i.e. the number of binary variables in a group) and the partition of binary variables, as well as the variable order. The papers [20, 62] present the optimization algorithm for pairing binary variables. Since these papers focus on the logic design for the field programmable gate arrays (FPGAs) with 6-input look-up tables (LUTs) as an application using MDDs, the size of groups is set to two and MDDs are optimized. However, this thesis assumes that size of groups can be changed, and discusses on the size of groups that optimizes MDDs. The paper [34] claims that when the size of groups is five, the best performance for the logic simulator using MDDs can be obtained. However, this paper does not show any theoretical or experimental justification. This thesis shows the optimum size of groups by experimental results using many benchmark functions.

When the binary variables are partitioned into groups, in many cases, groups have the same number of binary variables. However, in a heterogeneous MDD proposed in this thesis, the groups can have the different number of binary variables. Thus, heterogeneous MDDs allow more flexible partition of binary variables than MDD($k$)s that have groups with $k$ binary variables, and in heterogeneous MDDs, both orderings and partitions of binary variables can be optimized to minimize memory size and path length.

## 1.2 Organization of Thesis

This thesis consists of six chapters. Each chapter is organized as follows.

Chapter 2 defines basic terminologies, assumptions, and computational model used in this thesis.

Chapter 3 proposes APL minimization algorithms for DDs considering only variable orders. Experimental results in Chapter 3 show that the proposed APL minimization algorithms yield an improvement over an existing algorithm in both APL and runtime, and the APL minimization algorithms considering only variable orders often increase the number of nodes.

Chapter 4 shows the relations between the values of $k$ and the number of nodes, memory size, path length, and area-time complexity [8, 76] for QRMDD($k$), and derives the optimum values of $k$ for each application.

Chapter 5 proposes the memory size and APL minimization algorithms for heterogeneous MDDs that consider both orderings and partitions of binary variables. Experimental results in Chapter 5 show that by considering both orderings and partitions of binary variables, heterogeneous MDDs can represent logic functions with smaller memory sizes than FBDDs and smaller APLs than OBDDs, the APLs of heterogeneous MDDs can be reduced by a half of BDDs without increasing memory size, and heterogeneous MDDs have smaller area-time complexities than MDD($k$)s.

Chapter 6 concludes this thesis.

# Chapter 2

# Preliminary

This chapter defines basic terminologies used in this thesis.

## 2.1 Logic Functions

**Definition 2.1** A **logic function**, denoted by $f(x_1,x_2,\ldots,x_n)$ or simply $f$, is a mapping:

$$f(x_1,x_2,\ldots,x_n) : \{0,1,\ldots,r-1\}^n \rightarrow \{0,1,\ldots,r-1\},$$

where each $x_i$ is called a variable. When $r = 2$, a logic function is a **binary logic function** that is a mapping:

$$f(x_1,x_2,\ldots,x_n) : \{0,1\}^n \rightarrow \{0,1\},$$

where each $x_i$ is called a **binary variable**. When $r > 2$, a logic function is a **multi-valued logic function**, and each $x_i$ is called a **multi-valued variable**.

**Definition 2.2** A **multiple-output logic function** $F = (f_0, f_1, \ldots, f_{m-1})$ is a mapping:

$$F : \{0,1,\ldots,r-1\}^n \rightarrow \{0,1,\ldots,r-1\}^m.$$

Specially, when $m = 1$, it is called **single-output logic function**.

**Definition 2.3** Let $S \subseteq \{0,1,\ldots,r-1\}$. Then, $x^S$ is a **literal** of variable $x$.

**Definition 2.4** **Shannon expansion** of a logic function $f$ with respect to a variable $x_i$ is:

$$f(x_1,x_2,\ldots,x_n) = \bigvee_{j=0}^{r-1} x_i^j \cdot f(x_1,x_2,\ldots,x_{i-1},j,x_{i+1},\ldots,x_n),$$

and each $f(x_1,x_2,\ldots,x_{i-1},j,x_{i+1},\ldots,x_n)$ is called a **cofactor** of $f$ with respect to $x_i$.

In this thesis, we assume that a given logic function is completely specified and has no redundant variables.

## 2.2 Partition of Binary Variables

**Definition 2.5** Let $f(X)$ be a binary logic function, where $X = (x_1, x_2, \dots, x_n)$ is an ordered set of binary variables. Let $\{X\}$ denote the unordered set of variables in $X$. Let $X_i \subseteq X$. If $\{X\} = \{X_1\} \cup \{X_2\} \cup \dots \cup \{X_u\}$, $\{X_i\} \neq \phi$, and $\{X_i\} \cap \{X_j\} = \phi$ $(i \neq j)$, then $(X_1, X_2, \dots, X_u)$ is a **partition** of $X$. $X_i$ is called a **super variable**. If $|X_i| = k_i$ $(i = 1, 2, \dots, u)$ and $k_1 + k_2 + \dots + k_u = n$, then a binary logic function $f(X)$ can be represented by a **multi-valued input two-valued output logic function** that is a mapping $f(X_1, X_2, \dots, X_u)$: $R_1 \times R_2 \times R_3 \times \dots \times R_u \to B$, where $R_i = \{0, 1, 2, \dots, 2^{k_i} - 1\}$ and $B = \{0, 1\}$.

**Definition 2.6** A **fixed-order partition** of $X = (x_1, x_2, \dots, x_n)$ is a partition $(X_1, X_2, \dots, X_u)$, where

$$\begin{aligned} X_1 &= (x_1, x_2, \dots, x_{k_1}), \\ X_2 &= (x_{k_1+1}, x_{k_1+2}, \dots, x_{k_1+k_2}), \\ \dots \\ X_u &= (x_{k_1+k_2+\dots+k_{u-1}+1}, x_{k_1+k_2+\dots+k_{u-1}+2}, \dots, x_{n-1}, x_n), \end{aligned}$$

and $|X_i| = k_i$. That is, in the fixed-order partition of $X$, the order of variables $(x_1, x_2, \dots, x_n)$ is fixed.

When the order of variables is not fixed, we call the partition **non-fixed-order partition.** In this thesis, a **partition** means **fixed-order partition** unless stated otherwise.

**Example 2.1** Consider $(X_1, X_2)$, which is a fixed-order partition of $X$, where $X = (x_1, x_2, x_3, x_4, x_5)$ and each $x_i$ is a binary variable. When $X_1 = (x_1, x_2)$ and $X_2 = (x_3, x_4, x_5)$, we have $k_1 = 2$, $k_2 = 3$, $P_1 = \{0, 1, 2, 3\}$, and $P_2 = \{0, 1, \dots, 7\}$. Note that $X_1$ takes 4 values, and $X_2$ takes 8 values. So, a 5-variable binary logic function $f(X)$ can be represented by the multi-valued input two-valued output function $f(X_1, X_2)$: $R_1 \times R_2 \to B$. (End of Example)

## 2.3 Decision Diagrams (DDs)

**Definition 2.7** A **decision diagram (DD)** is a rooted directed acyclic graph (DAG) $G(V, E)$ representing a logic function $f$, where $V$ and $E$ denote sets of vertices and edges in $G$, respectively. Specially, vertices in $G$ are called **nodes** in the DD, nodes without outgoing edges are **terminal nodes,** and nodes with outgoing edges are **non-terminal nodes.** Each terminal node is labeled with a value of $f$, and each non-terminal node is labeled with a variable.

**Definition 2.8** In a DD, the **number of nodes in the DD**, denoted by $nodes$(DD), is the sum of all non-terminal nodes.

**Definition 2.9** A DD can be obtained by applying Shannon expansion repeatedly to a logic function $f$, and in such case, each non-terminal node labeled with a variable $x_i$ has some outgoing edges which refer to succeeding nodes representing cofactors of $f$ with respect to $x_i$. When all non-terminal nodes in the DD have two outgoing edges, the DD is called **binary decision diagram (BDD)**. On the other hand, when all non-terminal nodes have more than two outgoing edges, the DD is called **multi-valued decision diagram (MDD)**.

In this thesis, DD means either BDD or MDD.

**Definition 2.10** A **variable order** of DD is the order of variables that were used for Shannon expansion.

**Definition 2.11** In a DD, a sequence of edges and non-terminal nodes leading from a root node to a terminal node is a **path**.

**Definition 2.12** An **ordered BDD (OBDD)** has the same variable order on any path. On the other hand, a **free BDD (FBDD)** allows the different variable orders along each path.

**Definition 2.13** A **reduced ordered BDD (ROBDD)** is derived by applying the following two reduction rules to an OBDD:

1. Share equivalent sub-graphs.
2. If all the outgoing edges of a non-terminal node $v$ refer to the same succeeding node $u$, then delete $v$ and connect the incoming edges of $v$ to $u$.

A **quasi-reduced ordered BDD (QRBDD)** is derived by applying only the above reduction rule 1.

A **reduced ordered MDD (ROMDD)** and a **quasi-reduced ordered MDD (QRMDD)** can be defined similarly.

In this thesis, BDD and MDD means ROBDD and ROMDD, unless stated otherwise.

## 2.4 Average Path Lengths (APLs)

**Definition 2.14** A **path length** is the number of edges in the path.

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

**Definition 2.16** 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

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

where $C_i$ denotes a set of assignments of values to the variables selecting the path $p_i$, $\vec{c} = (c_1, c_2, \ldots, c_n)$, 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 2.17** The **average path length**, or **APL**, of a DD is given by:

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

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

In this thesis, we assume the following computation model:

1. The logic functions are evaluated by traversing DDs from the root node to a terminal node according to values of variables.

2. Encoded input values are available, and their access time is negligible. For example, when $X_1 = (x_1, x_2, x_3, x_4) = (1,0,0,1)$, $X_1 = 9$ is immediately available as an input to the super variable.

3. Most of computation time is devoted to accessing nodes.

4. The evaluation time for all DD nodes are the same.

In this case, the average evaluation time of a DD is proportional to the APL of the DD. Thus, in this model, we can use the APL to compare the evaluation times of different types of DDs.

# Chapter 3

# Minimization of APL in DDs by Variable Ordering

This chapter proposes APL minimization algorithms for DDs considering only variable orders.

## 3.1 Introduction

In applications using DDs to evaluate logic functions, the average evaluation time is proportional to the APL in the DD. Therefore, minimization of the APL leads to faster evaluation of the logic function. Particularly, in logic simulation using DDs [1, 22, 34, 35], 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 [80] uses BDDs to detect Boolean divisors. The quality of a divisor is measured by the number of don't-cares it provides for the 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 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 [4]. Obviously, minimizing the APL in the BDD can reduce the number of buffers that must be inserted.

In this chapter, 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 chapter is organized as follow. Section 3.2 contains the necessary terminology and definitions. Section 3.3 shows the efficient computation method of the APLs. Section 3.4 introduces lower bounds on the APL. Section 3.5 proposes an exact and a heuristic minimization algorithm for the APL. Section 3.6 shows the efficiency of the algorithms using benchmark functions.

## 3.2 Definitions

This section provides definitions used in this chapter.

**Definition 3.1** 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 3.2** 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 DD for a single-output function is 1.0, since all paths start from the root node.

In this chapter, we use shared DD (SDD) to represent a multiple-output function $F = (f_0, f_1, \ldots, f_{m-1})$ [40]. For reasons that will be clear later, we view the APL of an SDD as the sum of the APLs of the individual DDs or for each component logic function $f_i$.

## 3.3 Efficient Computation of APLs

This section provides the efficient computation method of APLs. This computation method plays an important part of APL minimization algorithms proposed in this chapter.

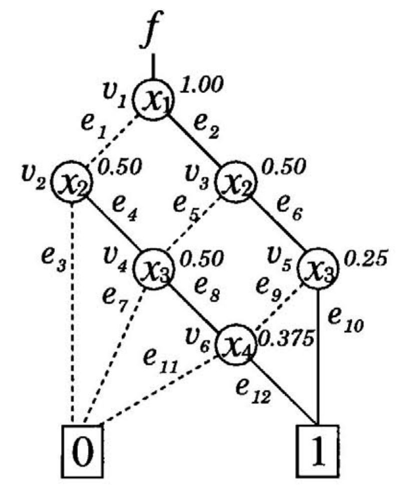
**Lemma 3.1** [67] The node traversing probability of node $v$ is the sum of the edge traversing probabilities of all incoming edges to $v$. Also, the node traversing probability of node $v$ is the sum of the edge traversing probabilities of all outgoing edges from $v$.

**Proof** See Appendix.

From Lemma 3.1, the following relation holds:

$$ETP(e) = P(x = c) \times NTP(v),$$

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 a value $c$ of $v$.



(a) BDD

| Path $p_i$ | $PP(p_i)$ | Path length $l_i$ |
|---|---|---|
| $p_1$ | 0.25 | 2 |
| $p_2$ | 0.125 | 3 |
| $p_3$ | 0.0625 | 4 |
| $p_4$ | 0.0625 | 4 |
| $p_5$ | 0.125 | 3 |
| $p_6$ | 0.0625 | 4 |
| $p_7$ | 0.0625 | 4 |
| $p_8$ | 0.0625 | 4 |
| $p_9$ | 0.0625 | 4 |
| $p_{10}$ | 0.125 | 3 |

(b) PPs and path lengths

Figure 3.1: Example of node traversing probability in a BDD.

**Theorem 3.1** [67] 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 3.1, we have the following:

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

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

**Example 3.1** Consider the BDD in Fig. 3.1(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 $(v_1, e_1, v_2, e_4, v_4, e_7), \ldots$, and path $p_{10}$ is $(v_1, e_2, v_3, e_5, v_5, e_{10})$. The $PP(p_i)$ and path length $l_i$ of each path $p_i$ are listed in Fig. 3.1(b). Therefore, by Definition 2.17,

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

By using node traversing probabilities, we can compute this APL as follows: First, we have $NTP(v_1) = 1.00$ for root node $v_1$. Then, $NTP(v_2) = ETP(e_1) = P(x_1 = 0) \times NTP(v_1) = 0.50$

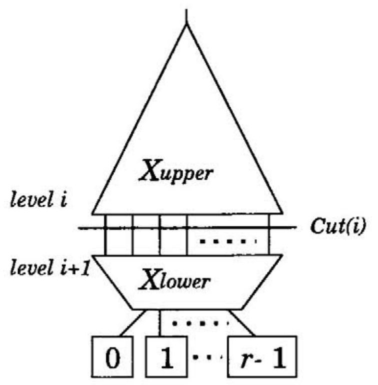


Figure 3.2: Partition of DD.

and $NTP(v_3) = ETP(e_2) = P(x_1 = 1) \times NTP(v_1) = 0.50$. Similarly,

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

Thus, we obtain

$$APL = \sum_{i=1}^{6} NTP(v_i) = 3.125.$$

Similarly, we can compute the APL using the edge traversing probabilities. (End of Example)

## 3.4 Lower Bounds on APL

In this section, we derive lower bounds on the APL. Such bounds result in a reduction of the computation time in the algorithm, as discussed later.

**Definition 3.3** Suppose a DD is partitioned into two parts as shown in Fig. 3.2. Here, $X_{upper}$ denotes the variables above or in level $i$, $X_{lower}$ denotes the variables below or in level $i+1$, and $Cut(i)$ 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 3.4** $ETP(Cut(i))$ denotes the sum of edge traversing probabilities of edges in $Cut(i)$, and is given by

$$ETP(Cut(i)) = \sum_{e \in Cut(i)} ETP(e).$$

**Lemma 3.2** Suppose an SDD represents a multiple-output logic function $F$. Then,

$$ETP(Cut(i)) = m_U,$$

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 a DD represents a single-output function $f$. Then,

$$ETP(Cut(i)) = 1.0.$$

**Lemma 3.3** Let

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

Then, for every permutation of $X_{upper}$,

$$ETP(Cut'(i)) = c_i,$$

where $c_i \leq m_U$.

**Proof** See Appendix.

**Theorem 3.2** Consider an SDD 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 $X_{lower}$ and any permutation of $X_{upper}$,

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

**Proof** See Appendix.

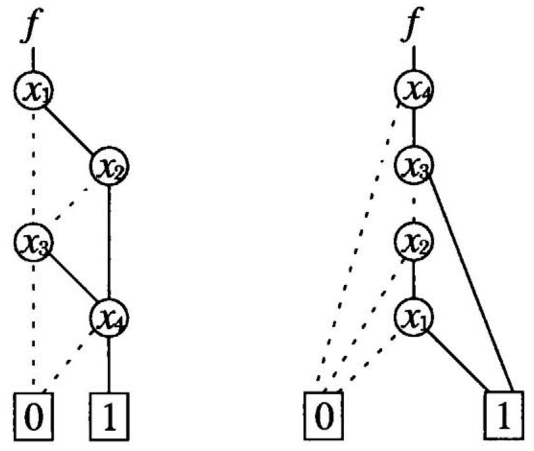
**Theorem 3.3** Consider an SDD for multiple-output function $F$. Let $U$ be the sum of the node traversing probabilities of the non-terminal nodes above or in level $i$. When the order of $X_{upper}$ is fixed,

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

**Proof** See Appendix.

**Corollary 3.2** Consider an SDD 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. If the variable order of the SDD is fixed,

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



(a) $APL = 2.875$ (b) $APL = 1.875$

Figure 3.3: Relation between the variable orders and the APLs.

## 3.5 APL Minimization Algorithms

**Example 3.2** Consider a binary logic function $f = x_4(x_3 \vee x_2x_1)$, and assume that $P(x_i = 0) = P(x_i = 1) = 0.50$ $(i = 1,2,3,4)$. When the variable order of BDD for $f$ is $(x_1,x_2,x_3,x_4)$, we have the BDD shown in Fig. 3.3(a). For the BDD in Fig. 3.3(a), $nodes(\text{BDD}) = 4$, $APL = 2.875$. When the variable order is $(x_4,x_3,x_2,x_1)$, we have the BDD in Fig. 3.3(b), where $nodes(\text{BDD}) = 4$, $APL = 1.875$. Note that the the numbers of nodes of two BDDs are minimum. (End of Example)

As shown in Fig. 3.3, since the APL in a DD depends on the variable order, the APL minimization problem can be formulated as follows:

**Problem 3.1** Given a DD for a logic function $f$, find a variable order that produces a DD with the minimum APL.

### 3.5.1 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.

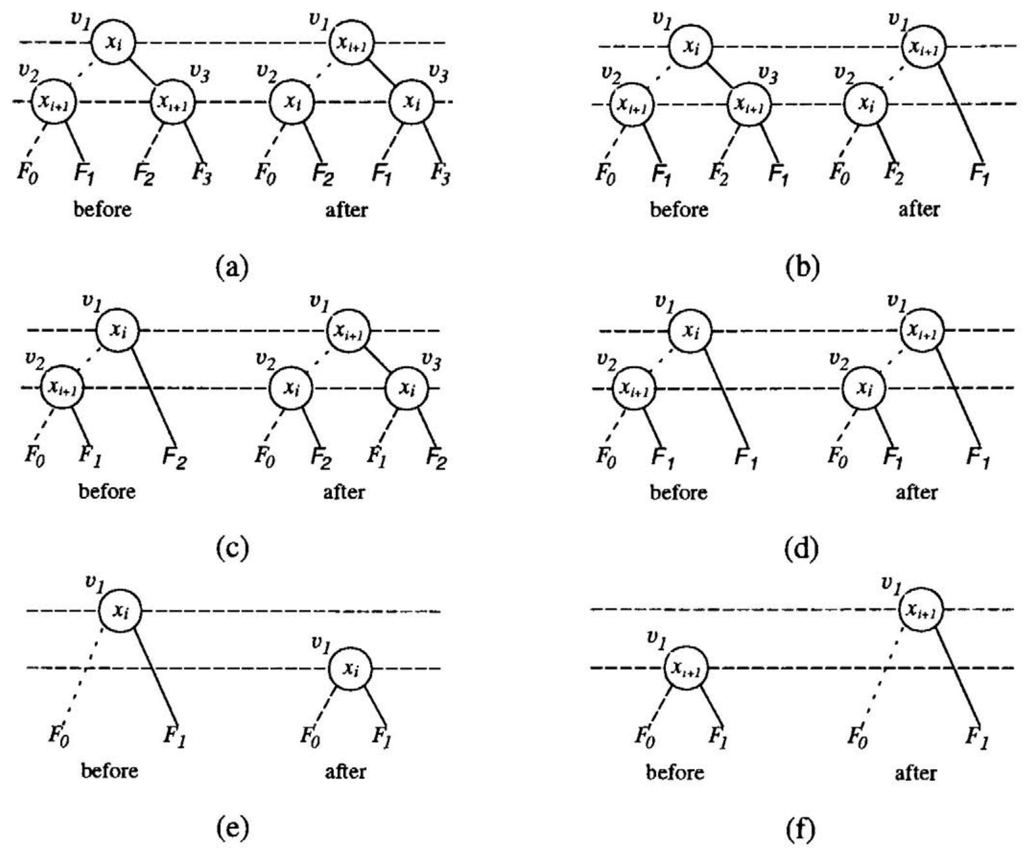


Figure 3.4: Six cases of exchanging two adjacent variables.

**Theorem 3.4** 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 3.4 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. 3.4 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 [38]. A subgraph composed of BDD nodes involved in the variable swap belongs to one of the six classes shown in Fig. 3.4. 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. 3.4, 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:

$$\begin{aligned} NTP(v_2) &= ETP(e_0) = P(x_i = 0) \times NTP(v_1) \\ NTP(v_3) &= ETP(e_1) = P(x_i = 1) \times NTP(v_1), \end{aligned}$$

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:

$$\begin{aligned} NTP(v_2) &= P(x_{i+1} = 0) \times NTP(v_1) \\ NTP(v_3) &= P(x_{i+1} = 1) \times NTP(v_1). \end{aligned}$$

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 3.4, 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 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 3.3** Fig. 3.5 shows BDDs for a binary logic function $f = x_1x_4 \vee x_2x_4 \vee x_3$. Fig. 3.5(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. 3.5(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. 3.5(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

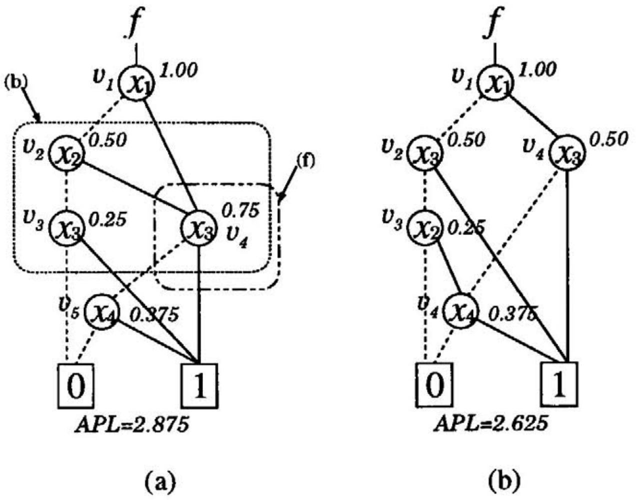


Figure 3.5: Example of the update of the APL

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 3.4** Fig. 3.6(a) shows the BDD with the variable order $(x_2, x_3, x_1)$ for logic function $f = x_1(x_2 \vee x_3)$. Assume that

$$\begin{aligned} P(x_1 = 0) &= 0.6, \quad P(x_1 = 1) = 0.4, \\ P(x_2 = 0) &= 0.3, \quad P(x_2 = 1) = 0.7, \\ P(x_3 = 0) &= 0.8, \quad P(x_3 = 1) = 0.2. \end{aligned}$$

The APL of the BDD in Fig. 3.6(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. 3.6(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)

### 3.5.2 Symmetric Variables

**Definition 3.5** 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.**

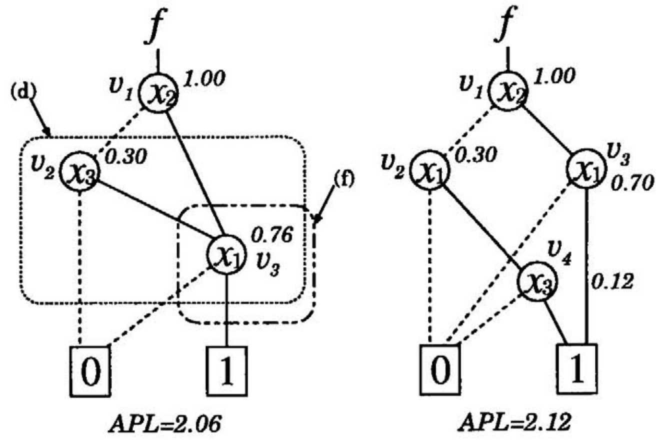


Figure 3.6: Another example of the update of the APL

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

**Definition 3.6** 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 3.5** Consider the logic function $f = x_1x_4 \vee x_2x_4 \vee x_3$ (Fig. 3.5). 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)

## 3.5.3 Exact Minimization Algorithm

Fig. 3.7 shows a pseudo-code to solve Problem 3.1. This algorithm finds an optimum solution using a branch-and-bound method, similar to the top-down algorithm (*JANUS*) in [16]. *JANUS* [16] 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

**Algorithm 3.1**

```
minimize_APL (DD, input variables $X$, # inputs $n$) {
    $X_{sub} = \phi$ ;
    cost[$X_{sub}$] = 0 ;
    order[$X_{sub}$] = $\phi$ ;
    $S_{next} = \{X_{sub}\}$ ;
    $min\_apl$ = APL for initial DD ;
    for ($level$ = 1; $level \leq n$; $level$++) {
        $S_{cur} = S_{next}$ ;
        $S_{next} = \phi$ ;
        for (each $X_{sub} \in S_{cur}$) {
            ordering(DD, order[$X_{sub}$], $level$   1) ;
            for (each $x_i \in \{X \setminus X_{sub}\}$) {
                $X'_{sub} = X_{sub} \cup \{x_i\}$ ;
                Move $x_i$ to $level$ ;
                symmetry_check(DD, $level$) ;
                if (order[$X'_{sub}$] and current order are symmetric && all $P(x = c)$s are same)
                    continue ;
                Update $min\_apl$ ;
                if (**lower_bound**($level$) > $min\_apl$)
                    continue ;
                $new\_cost$ = cost[$X_{sub}$] + $NTP(level)$ ;
                if ($new\_cost$ < cost[$X'_{sub}$]) {
                    cost[$X'_{sub}$] = $new\_cost$ ;
                    order[$X'_{sub}$] = current order ;
                    if ($X'_{sub} \notin S_{next}$)
                        $S_{next} = S_{next} \cup \{X'_{sub}\}$ ;
                }
            }
        }
    }
    ordering(DD, order[$X$], $n$) ;
}
```

Figure 3.7: Exact APL minimization algorithm by variable ordering.

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. 3.7, 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 [53]. When the variable order of $X'_{sub}$, which has already been stored in array "order[$X'_{sub}$]" as a candidate, and the current variable order of the DD are symmetric, and all $P(x=c)$s are same for the symmetric variables, the current order is excluded from candidates. In line 19, Theorem 3.3 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. 3.7 are set to infinity.

### 3.5.4 Heuristic Minimization Algorithm

The algorithm in Fig. 3.7 obtains an optimum solution for Problem 3.1. 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 [59]. 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 3.5.1 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. 3.8 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. 3.8, Corollary 3.2 is used to eliminate unneeded

**Algorithm 3.2**

```
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 3.8: Heuristic APL minimization algorithm by variable ordering.

sifting of $x_i$. When variable $x_i$ moves down to the bottom, we use $U$ equal to the sum of the node traversing probabilities of the nodes above $x_i$. If $cost \leq U$, sifting of $x_i$ further down to the bottom cannot lead to a smaller APL than $cost$. 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 [14].

### 3.5.5 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 [12, 68] leads to a heuristic to find a good initial variable order. In this section, we propose an initial variable order using Walsh spectrum [21] 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 [13]:

$$R_i = \frac{|\bar{x}_i \oplus f|}{2^{n-1}} - 1,$$

where $|\bar{x}_i \oplus f|$ denotes the number of assignments of values to the variables $X$ 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 $R_i$. For variables with identical absolute values of $R_i$, 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 [77]. The first-order coefficients can be computed by a simplified version of the general algorithm.

**Example 3.6** Consider the binary logic function $f = x_1x_4 \vee x_2x_4 \vee x_3$ in Example 3.3. For each binary variable $x_i$, the value of $|\bar{x}_i \oplus f|$ is given by:

$$|\bar{x}_1 \oplus f| = 9, \qquad |\bar{x}_2 \oplus f| = 9, \qquad |\bar{x}_3 \oplus f| = 13, \qquad |\bar{x}_4 \oplus f| = 11.$$

The value of each $R_i$ corresponding to $x_i$ is as follows:

$$R_1 = \frac{1}{8}, \qquad R_2 = \frac{1}{8}, \qquad R_3 = \frac{5}{8}, \qquad R_4 = \frac{3}{8}.$$

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)

Table 3.1: Minimization of APL for individual BDDs

| Name | In | Out | (a) Min_Nodes | | | (b) Min_APL | | | (c) Liu [31] | | (d) sifting | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | Nodes | APL | Time | Nodes | APL | Time | Nodes | APL | Nodes | APL | Time |
| 5xp1 | 7 | 10 | 66 | 34.13 | 0.01 | 81 | 31.28 | 0.01 | 91 | 31.31 | 79 | 31.28 | 0.01 |
| alu4 | 14 | 8 | 448 | 41.75 | 22.76 | 547 | 39.69 | 28.71 | 899 | 47.54 | 516 | 39.97 | 0.01 |
| b12 | 15 | 9 | 64 | 23.86 | 0.03 | 68 | 21.84 | 0.01 | 81 | 22.22 | 71 | 21.88 | 0.01 |
| con1 | 7 | 2 | 14 | 6.06 | 0.01 | 16 | 5.94 | 0.01 | 16 | 6.06 | 16 | 5.94 | 0.01 |
| cordic | 23 | 2 | 73 | 13.74 | 416.57 | 89 | 9.43 | 1006.08 | 259 | 11.82 | 88 | 9.47 | 0.01 |
| sao2 | 10 | 4 | 99 | 10.90 | 0.26 | 116 | 10.59 | 0.06 | 128 | 10.71 | 121 | 10.59 | 0.01 |
| vg2 | 25 | 8 | 202 | 31.00 | 6431.83 | 222 | 29.91 | 376.78 | 230 | 30.37 | 204 | 30.16 | 0.01 |
| misex1 | 8 | 7 | 54 | 23.22 | 0.01 | 57 | 21.97 | 0.02 | 68 | 22.16 | 64 | 21.97 | 0.01 |
| cm150a | 21 | 1 | 32 | 3.50 | 1106.23 | 32 | 3.50 | 1510.58 | 33 | 3.50 | 32 | 3.50 | 0.01 |
| cm151a | 12 | 2 | 32 | 6.00 | 0.38 | 32 | 6.00 | 0.28 | 36 | 6.50 | 32 | 6.00 | 0.01 |
| cm162a | 14 | 5 | 41 | 11.76 | 0.06 | 52 | 11.70 | 0.05 | 59 | 11.70 | 48 | 11.71 | 0.01 |
| cm163a | 16 | 5 | 35 | 11.70 | 0.01 | 38 | 11.70 | 0.01 | 42 | 11.70 | 36 | 11.70 | 0.01 |
| cm85a | 11 | 3 | 38 | 7.72 | 0.05 | 38 | 7.72 | 0.01 | 47 | 8.28 | 38 | 7.72 | 0.01 |
| mux | 21 | 1 | 32 | 3.50 | 1098.72 | 32 | 3.50 | 1410.57 | 33 | 3.50 | 32 | 3.50 | 0.01 |
| z4ml | 7 | 4 | 28 | 18.25 | 0.01 | 30 | 16.38 | 0.02 | 32 | 17.13 | 28 | 16.38 | 0.01 |
| f51m | 8 | 8 | 51 | 28.08 | 0.01 | 65 | 27.33 | 0.02 | 76 | 27.45 | 64 | 27.45 | 0.01 |
| pcle | 19 | 9 | 79 | 22.50 | 0.11 | 84 | 22.50 | 0.03 | 89 | 22.50 | 79 | 22.50 | 0.01 |
| Average of ratios | | | 1.00 | 1.00 | 1.00 | 1.12 | 0.95 | 0.93 | 1.40 | 0.99 | 1.10 | 0.95 | 0.40 |

## 3.6 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 $P(x_i = 0) = P(x_i = 1) = 0.5$ for binary logic functions.

Table 3.1 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 [31]; and (d) the heuristic APL minimization algorithm presented in this chapter. 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,

Table 3.2: Minimization of APL for shared BDDs for larger functions

| Name | In | Out | classical sifting | | Coef. | Without Walsh spectrum | | | With Walsh spectrum | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | 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 |
| C880 | 60 | 26 | 4122 | 140.42 | 0.01 | 41701 | 123.85 | 4.48 | 91767 | 122.22 | 52.12 |
| C1908 | 33 | 25 | 5532 | 254.65 | 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 | 209.15 | 0.10 | 25162 | 208.44 | 7.44 | 56898 | 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 | 65 | 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 |
| 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.

in seconds. Unfortunately, the CPU time of the algorithm in [31] is unavailable. Columns "(a) Min_Nodes", "(b) Min_APL", "(c) Liu [31]", and "(d) sifting" show the exact nodes minimization algorithm in [16], the exact APL minimization algorithm in Section 3.5.3, the heuristic APL minimization in [31], and the heuristic APL minimization in Section 3.5.4, respectively. Initial variable order for "(d) sifting" was obtained using Walsh spectrum described in Section 3.5.5. The BDDs in this table use complemented edges [6, 40]. Table 3.1 includes the same benchmark functions as the experiment in [31] 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 assignment of values to don't cares, as well as the variable order. To make our results compatible with the results in [31], 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 3.1 are different from those of the shared BDD (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 [31]", and "(d) sifting" of this row contains the relative values to the results of "(a) Min_Nodes".

The heuristic method in [31] obtained BDDs with the exact minimum APLs in 5 out of 17 benchmark functions. However, for *alu4*, *cm151a*, and *cm85a*, the algorithm in [31] obtained BDDs with much larger APLs than the exact minimum APLs. On the other hand, our heuristic method in Section 3.5.4 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 [31]". 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 3.5.3 found the minimum APL for functions with 25 inputs (*vg2*) within a reasonable computation time.

Table 3.2 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 3.2, the column "classical sifting" shows the number of nodes and APL for BDDs obtained by the sifting algorithm [59] 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 3.5.5. The column "Coef. 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 3.1 and 3.2 show that the proposed heuristic minimization minimizes the APL in short computation time. For small benchmark functions in Table 3.1, the heuristic minimization could obtain BDDs with near-minimum APLs. For large benchmark functions in Table 3.2, the heuristic algorithm reduces APLs to 84% on the average.

## 3.7 Conclusion and Comments

In this chapter, we have proposed an exact and a heuristic APL minimization algorithm for BDDs and MDDs by variable ordering. 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 the 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) For many benchmark functions, APL minimization by variable ordering increases the number of nodes.

# Chapter 4

# Area-Time Complexities of QRMDD($k$)s

This chapter shows the relations between the values of $k$ and the number of nodes, memory size, path length, and area-time complexity [8, 76] for QRMDD($k$), and derives the optimum values of $k$ for each application.

## 4.1 Introduction

Since modern computer systems have the memory hierarchical structure, suitable DDs for the memory hierarchy can shorten the runtimes of applications using DDs [34, 35, 78]. QRBDDs and QRMDDs are suitable for the memory hierarchy [52], parallel process [23, 51], and design of LUT cascades [65]. However, in general, QRBDDs and QRMDDs require more nodes than corresponding ROBDDs and ROMDDs to represent logic functions. Hence, the minimizations of QRBDDs and QRMDDs are very important. In many cases, the minimizations of DDs use the variable reordering [14, 17, 18, 19, 24, 39, 59, 74]. In the minimization of MDDs, a partition of binary variables [20, 62] is important, as well as the variable ordering.

To represent a binary logic function using an MDD, binary variables are partitioned into groups. The papers [20, 62] present the optimization algorithm of partition of input binary variables into groups of binary variables. However, the size of groups (i.e. the number of binary variables in a group) is fixed in these algorithms. In this chapter, we assume that the size of groups, that is the value of $k$ for QRMDD($k$)s, can be changed, and we find the optimum sizes of groups experimentally by showing the relations of the values of $k$ and the numbers of nodes, the memory sizes, and the path length. To show these relations, we assume that the order of binary variable is fixed. Our statistical results are useful for minimizations of MDDs, software synthesis [2], and logic simulation[1, 22, 34, 35].

The rest of this chapter is organized as follows: Section 4.2 defines MDD($k$)s, QRMDD($k$)s, computation model for MDDs, and a method to represent multiple-output functions. Section 4.3 considers the number of nodes in QRMDD($k$)s for general functions, benchmark functions, and randomly generated functions. Section 4.4 introduces the measure called *area-time complexity* [8, 76] to find the optimum value of $k$ for QRMDD($k$)s, and derives the optimum values of $k$ by experiments.

## 4.2 Definitions

This section provides definitions used in this chapter.

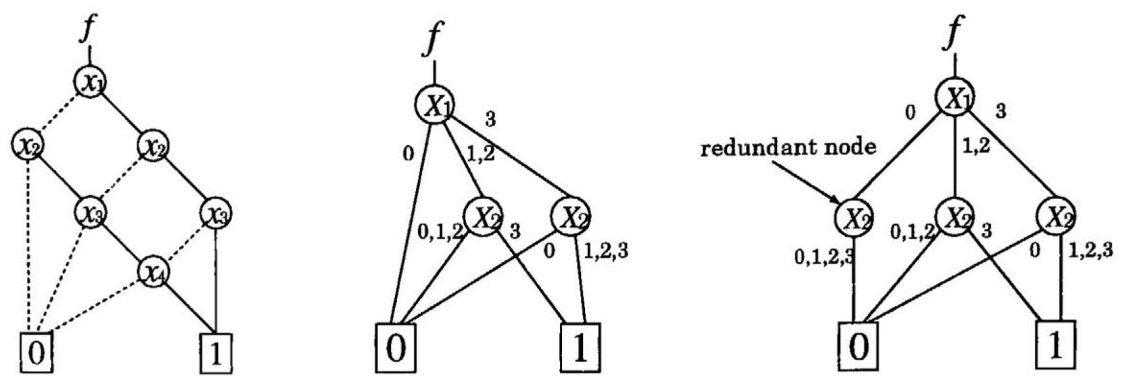
**Definition 4.1** When $X = (x_1, x_2, \dots, x_n)$ is partitioned into $(X_1, X_2, \dots, X_u)$, where $|X_i| = k$ $(i = 1, 2, \dots, u)$, an ROMDD representing a multi-valued input two-valued output logic function $f(X_1, X_2, \dots, X_u)$ is called an **MDD($k$)**. Similarly, a QRMDD representing $f(X_1, X_2, \dots, X_u)$ is called a **QRMDD($k$)**. An MDD($k$) and a QRMDD($k$) represent a mapping $f : R^u \rightarrow B$, where $R = \{0, 1, \dots, 2^k - 1\}$ and $B = \{0, 1\}$. In an MDD($k$) and a QRMDD($k$), non-terminal nodes have $2^k$ outgoing edges.

For $n$-variable logic functions $f$, if $n < ku$ (i.e. $n$ is indivisible by $k$), we use additional redundant binary variables, which are called **dummy variables**, to construct MDD($k$). The set of binary variables with dummy variables is denoted by $\{X'\} = \{x_1, x_2, \dots, x_n, x_{n+1}, \dots, x_{n+t}\}$, where $|X'| = n + t$, and $t$ denotes the number of dummy variables. Note that $f$ is independent of $x_{n+1}, x_{n+2}, \dots$ and $x_{n+t}$.

The path length of an arbitrary path in a QRMDD($k$) is equal to the number of super variables. Thus, APL of a QRMDD($k$) is also equal to the number of supper variables. An MDD($k$) has no redundant nodes, while a QRMDD($k$) usually has redundant nodes. Therefore, we have the following relation between the number of nodes in an MDD($k$) and its corresponding QRMDD($k$):

$$nodes(\text{MDD}(k)) \leq nodes(\text{QRMDD}(k)).$$

**Example 4.1** Consider the logic function $f = x_1x_2x_3 \vee x_2x_3x_4 \vee x_3x_4x_1 \vee x_4x_1x_2$ in Example 3.1. The BDD, the MDD(2), and the QRMDD(2) for $f$ are shown in Fig. 4.1(a), (b), and (c), respectively. In Fig. 4.1(a), the solid lines and the broken lines denote 1-edges and 0-edges, respectively. In Fig. 4.1(b) and (c), the input variables $X = (x_1, x_2, x_3, x_4)$ are partitioned into $(X_1, X_2)$, where $X_1 = (x_1, x_2)$ and $X_2 = (x_3, x_4)$. We have $nodes(\text{ROBDD}) = 6$, $nodes(\text{ROMDD}(2)) = 3$, and $nodes(\text{QRMDD}(2)) = 4$. (End of Example)



(a) BDD. (b) MDD(2). (c) QRMDD(2).

Figure 4.1: BDD, MDD(2), and QRMDD(2).

In this chapter, we use an encoded characteristic function for non-zero output (ECFN) [64, 66] to represent multiple-output logic functions $F = (f_0, f_1, \ldots, f_{m-1})$. An ECFN uses $u = \lceil \log_2 m \rceil$ auxiliary variables to represent the outputs, and represents a mapping:

$$ECFN : B^{n+u} \rightarrow B,$$

where $n$ is the number of binary variables and $B = \{0,1\}$.

**Definition 4.2** The **density** for an $n$-variable logic function $f$ is defined as

$$\frac{|f|}{2^n} \times 100,$$

where $|f|$ denotes the number of $\vec{a}$ such that $f(\vec{a}) = 1$.

The density for a multiple-output function $F$ is the density for an ECFN representing $F$.

## 4.3 Number of Nodes in QRMDD($k$)

In this section, we first obtain an upper bound on the number of nodes in a QRMDD($k$). Then, we obtain the numbers of nodes in QRMDD($k$)s for benchmark functions, and show that an interesting property holds for many benchmark functions. Finally, we obtain the numbers of nodes in QRMDD($k$)s for randomly generated functions, and show that they have quite different property from the benchmark functions.

Table 4.1: Upper bounds on the number of nodes in QRMDD($k$).

| $n$ | $k$ | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| 10 | 275 | 101 | 77 | 33 | 33 |
| 11 | 531 | 345 | 89 | 273 | 37 |
| 12 | 787 | 357 | 329 | 273 | 49 |
| 13 | 1299 | 601 | 589 | 277 | 289 |
| 14 | 2323 | 1381 | 601 | 289 | 1057 |
| 15 | 4371 | 1625 | 841 | 529 | 1057 |
| 16 | 8467 | 5477 | 4685 | 4369 | 1061 |
| 17 | 16659 | 5721 | 4697 | 4373 | 1073 |
| 18 | 33043 | 21861 | 4937 | 4385 | 1313 |
| 19 | 65811 | 22105 | 37453 | 4625 | 33825 |
| 20 | 131347 | 87397 | 37465 | 69905 | 33825 |

### 4.3.1 Number of Nodes for General Functions

**Theorem 4.1** An arbitrary $n$-variable logic function can be represented by a QRBDD with at most

$$2^{n-r}-1+\sum_{i=1}^{r}2^{2^i}$$

non-terminal nodes, where $r$ is the largest integer that satisfies relation $n-r \geq 2^r$ [29].

**Proof** See Appendix.

**Theorem 4.2** An arbitrary $n$-variable logic function can be represented by a QRMDD($k$) with at most

$$\frac{2^{sk}-1}{2^k-1}+\sum_{i=1}^{u-s}2^{2^{(ki-t)}}$$

non-terminal nodes, where $u$ is the number of super variables, $t$ is the number of dummy variables, and $s$ is the smallest integer that satisfies relation

$$s \geq \frac{n-r}{k}.$$

**Proof** See Appendix.

Table 4.1 shows the upper bounds on the number of nodes in QRMDD($k$)s for $n$-variable logic functions. We can see that the upper bounds are non-monotone functions of $k$.

### 4.3.2 Number of Nodes for Benchmark Functions

We used 157 benchmark functions [9, 63, 81] shown in Table 4.2, where $n$ and $m$ denote the number of input and output variables, respectively. In this table, the benchmark functions under *sequential* originally represented sequential circuits. We removed flip-flops (FFs) from these sequential circuits to make them combinational. Such functions are renamed by appending a subscript '$c$' to the original names. In this chapter, encodings for ECFNs and binary variable orders of BDDs are obtained by the heuristic algorithm in [66]. In the following experiments, we use these variable orders, and we consider only the partition of binary variables. For each benchmark function, we counted the number of nodes in the corresponding QRMDD($k$)s for various $k$. In Table 4.3, *avg* denotes the arithmetic average of the relative numbers of nodes, where the number of nodes in QRBDD is set to 1.00, and *stdv* denotes the standard deviation.

**Definition 4.3** The relation '$\simeq$' is defined as follows:

$$a \simeq b \Leftrightarrow \eta < 0.1,$$

where $a$ and $b$ are positive integers, and the **normalized difference** $\eta$ is given by:

$$\eta = \frac{|a-b|}{\min(a,b)}.$$

If $a \simeq b$, then $a$ and $b$ are **nearly equal.**

For 133 functions in Table 4.2, the following property holds.

**Property 4.1**

$$nodes(\text{QRMDD}(k)) \simeq \frac{1}{k} nodes(\text{QRBDD})$$

For the remaining 24 functions, $\eta \geq 0.1$ holds. Table 4.4 lists these 24 functions. In Table 4.4, "# nodes", "dens.", and "cate." denote the numbers of nodes in QRBDDs, the densities, and the categories of functions described below, respectively. Fig. 4.2 shows the relation between the normalized difference $\eta$ and the densities for benchmark functions. The symbols $+, \times, \odot$, and $\triangle$ correspond to the values for $k = 2, 3, 4$, and 5, respectively. For each function, we assume that Property 4.1 holds when all the symbols are below the border line of $\eta = 0.1$ (i.e., $\eta < 0.1$ holds for $k = 2 \sim 5$). From Fig. 4.2, we categorized 24 functions in Table 4.4 into three sets.

1. The densities of functions are between 40% and 60%, and the number of nodes for QRB-DDs are large relative to the number of inputs.

2. The functions have iterative properties (i.e., adder and comparator).

3. The numbers of nodes, inputs, and outputs are small. Property 4.1 does not hold for $k = 4$ or 5.

Table 4.2: List of benchmark functions.

| Name | n | m | Name | n | m | Name | n | m |
|---|---|---|---|---|---|---|---|---|
| 3adr6 | 18 | 12 | i5 | 133 | 66 | signet | 39 | 8 |
| C432 | 36 | 7 | i6 | 138 | 67 | soar | 83 | 94 |
| C499 | 41 | 32 | i7 | 199 | 67 | spla | 16 | 46 |
| C880 | 60 | 26 | i8 | 133 | 81 | sqr16 | 16 | 32 |
| C1355 | 41 | 32 | i9 | 88 | 63 | t1 | 21 | 23 |
| C1908 | 33 | 25 | i10 | 257 | 224 | t2 | 17 | 16 |
| C2670 | 233 | 140 | ibm | 48 | 17 | table5 | 17 | 15 |
| C3540 | 50 | 22 | in1 | 16 | 17 | tcon | 17 | 16 |
| C5315 | 178 | 123 | in2 | 19 | 10 | term1 | 34 | 10 |
| C7552 | 207 | 108 | in3 | 35 | 29 | ti | 47 | 72 |
| accpla | 50 | 69 | in4 | 32 | 20 | too_large | 38 | 3 |
| adr8 | 16 | 9 | in5 | 24 | 14 | ts10 | 22 | 16 |
| adr9 | 18 | 10 | in6 | 33 | 23 | ttt2 | 24 | 21 |
| al2 | 16 | 47 | in7 | 26 | 10 | unreg | 36 | 16 |
| alcom | 15 | 38 | inc16 | 16 | 17 | vda | 17 | 39 |
| apex1 | 45 | 45 | inc17 | 17 | 18 | vg2 | 25 | 8 |
| apex2 | 39 | 3 | inc18 | 18 | 19 | vtx1 | 27 | 6 |
| apex3 | 54 | 50 | jbp | 36 | 57 | wgt17 | 17 | 5 |
| apex5 | 117 | 88 | k2 | 45 | 45 | wgt18 | 18 | 5 |
| apex6 | 135 | 99 | lal | 26 | 19 | x1 | 51 | 35 |
| apex7 | 49 | 37 | log16 | 16 | 16 | x3 | 135 | 99 |
| b2 | 16 | 17 | log17 | 17 | 17 | x4 | 94 | 71 |
| b3 | 32 | 20 | log18 | 18 | 18 | x1dn | 27 | 6 |
| b4 | 33 | 23 | mainpla | 27 | 54 | x2dn | 82 | 56 |
| b9 | 41 | 21 | mark1 | 20 | 31 | x6dn | 39 | 5 |
| bc0 | 26 | 11 | misex2 | 25 | 18 | x7dn | 66 | 15 |
| bca | 26 | 46 | misg | 56 | 23 | x9dn | 27 | 7 |
| bcb | 26 | 39 | mish | 94 | 43 | xparc | 41 | 73 |
| bcc | 26 | 45 | misj | 35 | 14 | sequential | | |
| bcd | 26 | 38 | mlp8 | 16 | 16 | s208$_c$ | 18 | 9 |
| c8 | 28 | 18 | mlp9 | 18 | 18 | s298$_c$ | 17 | 20 |
| cc | 21 | 20 | mlp10 | 20 | 20 | s344$_c$ | 24 | 26 |
| chkn | 29 | 7 | mux | 21 | 1 | s349$_c$ | 24 | 26 |
| cht | 47 | 36 | my_adder | 33 | 17 | s382$_c$ | 24 | 27 |
| cm150a | 21 | 1 | nrm8 | 16 | 9 | s400$_c$ | 24 | 27 |
| comp | 32 | 3 | nrm9 | 18 | 10 | s420$_c$ | 34 | 17 |
| cordic | 23 | 2 | opa | 17 | 69 | s444$_c$ | 24 | 27 |
| count | 35 | 16 | pair | 173 | 137 | s510$_c$ | 25 | 13 |
| cps | 24 | 109 | pcle | 19 | 9 | s526$_c$ | 24 | 27 |
| dalu | 75 | 16 | pcler8 | 27 | 17 | s641$_c$ | 54 | 43 |
| des | 256 | 245 | pdc | 16 | 40 | s713$_c$ | 54 | 42 |
| dk48 | 15 | 17 | pm1 | 16 | 13 | s820$_c$ | 23 | 24 |
| duke2 | 22 | 29 | rckl | 32 | 7 | s832$_c$ | 23 | 24 |
| e64 | 65 | 65 | rdm16 | 16 | 16 | s838$_c$ | 66 | 33 |
| ex4 | 128 | 28 | rdm17 | 17 | 17 | s1196$_c$ | 32 | 32 |
| example2 | 85 | 66 | rdm18 | 18 | 18 | s1423$_c$ | 91 | 79 |
| exep | 30 | 63 | rot | 135 | 107 | s5378$_c$ | 214 | 228 |
| frg1 | 28 | 3 | rot16 | 16 | 9 | s9234$_c$ | 247 | 250 |
| frg2 | 143 | 139 | rot17 | 17 | 9 | s13207$_c$ | 700 | 790 |
| i1 | 25 | 16 | rot18 | 18 | 10 | s15850$_c$ | 611 | 684 |
| i2 | 201 | 1 | sct | 19 | 15 | s38417$_c$ | 1664 | 1742 |
| i3 | 132 | 6 | seq | 41 | 35 | s38584$_c$ | 1464 | 1730 |
| i4 | 192 | 6 | shift | 19 | 16 | | | |

Table 4.3: Relation of nodes in QRMDD($k$) and $k$ for benchmark functions.

| | $k$ | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| *avg* | 1.000 | 0.498 | 0.333 | 0.248 | 0.202 |
| *stdv* | 0.000 | 0.013 | 0.009 | 0.016 | 0.016 |

Table 4.4: Benchmark functions with $\eta \geq 0.1$.

| Name | # in | # out | # nodes | dens. | cate. |
|---|---|---|---|---|---|
| C499 | 41 | 32 | 24476 | 50.0 | 1 |
| C1355 | 41 | 32 | 30156 | 50.0 | 1 |
| C1908 | 33 | 25 | 9292 | 45.8 | 1 |
| adr8 | 16 | 9 | 153 | 50.0 | 2 |
| adr9 | 18 | 10 | 180 | 50.0 | 2 |
| comp | 32 | 3 | 114 | 37.5 | 2 |
| inc17 | 17 | 18 | 236 | 48.4 | 3 |
| log16 | 16 | 16 | 11216 | 59.9 | 1 |
| log17 | 17 | 17 | 23054 | 55.1 | 1 |
| log18 | 18 | 18 | 31458 | 55.2 | 1 |
| mlp8 | 16 | 16 | 10112 | 41.5 | 1 |
| mlp9 | 18 | 18 | 28332 | 37.5 | 1 |
| mlp10 | 20 | 20 | 82077 | 38.5 | 1 |
| my_adder | 33 | 17 | 450 | 50.0 | 2 |
| nrm8 | 16 | 9 | 8689 | 49.1 | 1 |
| nrm9 | 18 | 10 | 23152 | 49.0 | 1 |
| pcle | 19 | 9 | 221 | 29.3 | 3 |
| rot16 | 16 | 9 | 1021 | 60.8 | 3 |
| rot17 | 17 | 9 | 1429 | 49.3 | 3 |
| sqr16 | 16 | 32 | 18366 | 42.9 | 1 |
| tcon | 17 | 16 | 183 | 50.0 | 3 |
| vg2 | 25 | 8 | 217 | 22.9 | 3 |
| vtx1 | 27 | 6 | 326 | 12.5 | 3 |
| x1dn | 27 | 6 | 332 | 12.5 | 3 |

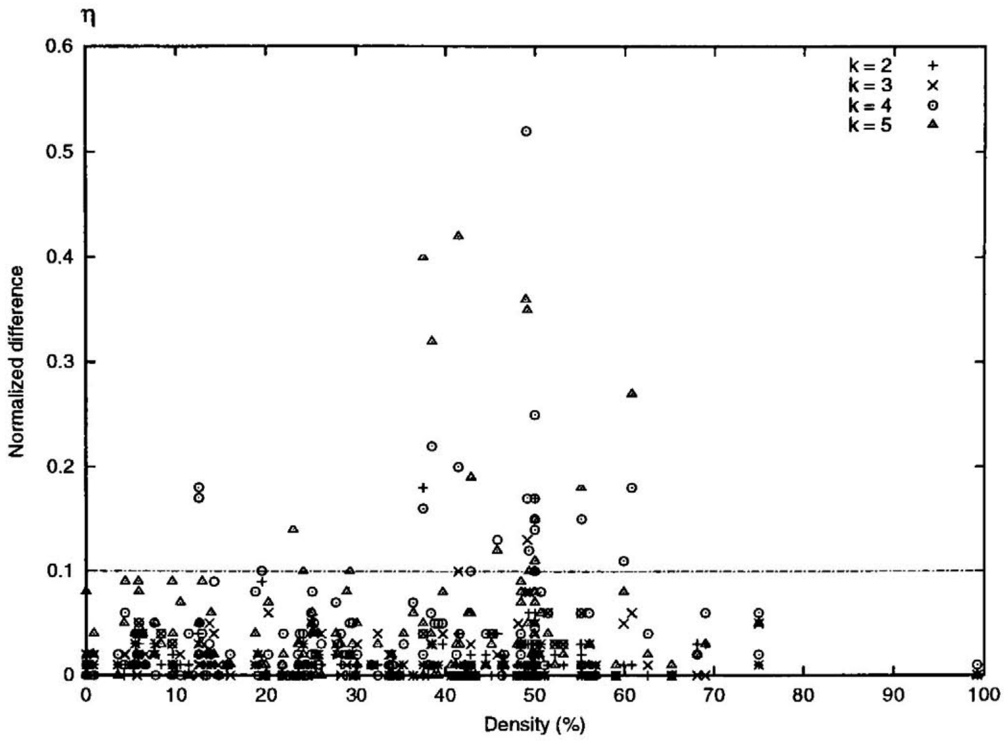


Figure 4.2: Relation between the normalized difference η and density for benchmark functions.

### 4.3.3 Number of Nodes for Randomly Generated Functions

For $d = 1, 2, \ldots, 99$, we randomly generated one 25-variable function with density $d$ to obtain 99 functions. Fig. 4.3 shows the relation between the normalized difference η and the densities for randomly generated functions. In this case, no randomly generated functions of 25 variables satisfied Property 4.1. This fact shows that randomly generated functions have quite different property from the benchmark functions in Table 4.2.

For many benchmark functions, the numbers of nodes in QRMDD($k$)s decrease as $k$ increase. However, for randomly generated functions, the number of nodes is a non-monotone function of $k$. For example, for many randomly generated functions of 25 variables, the numbers of nodes in QRMDD(5)s were larger than those in QRMDD(3)s.

For $n = 10, 11, \ldots, 20$, we also randomly generated ten $n$-variable functions with density 50%. Table 4.5 shows the average numbers of nodes in QRMDD($k$)s for randomly generated functions. The deviations were within ±2% of the averages. From Table 4.1 and Table 4.5, we can see that the numbers of nodes in QRMDD($k$)s for randomly generated functions with density 50% are nearly equal to the upper bounds.

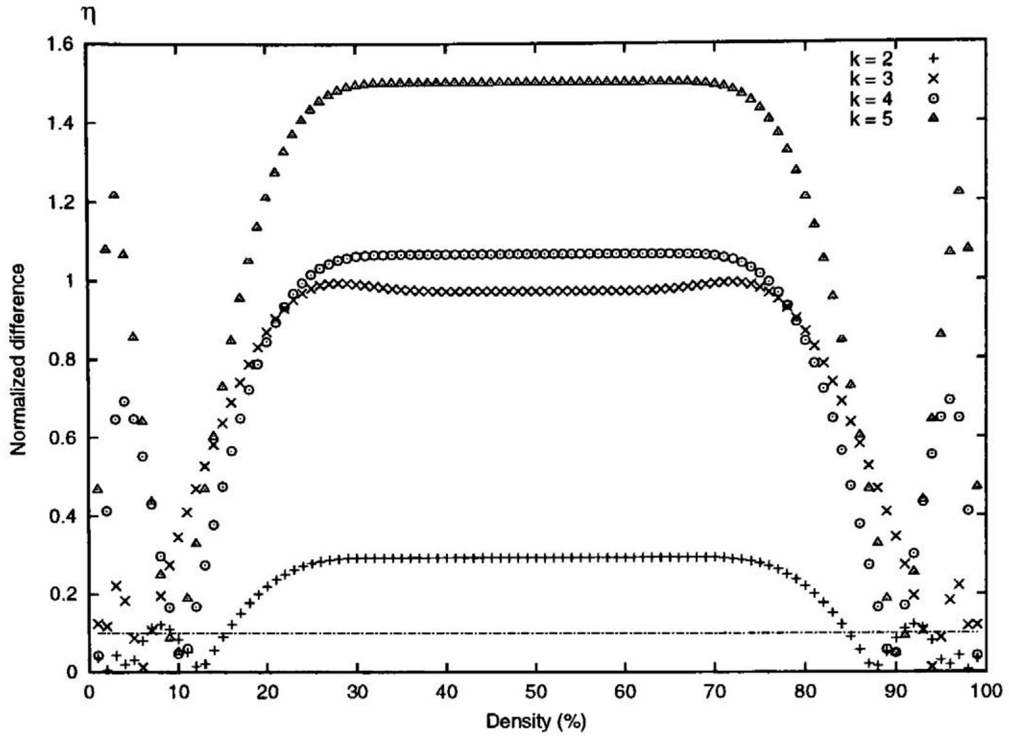


Figure 4.3: Relation between the normalized difference η and density for randomly generated functions.

## 4.4 Area-Time Complexity of QRMDD($k$)

### 4.4.1 Memory Size for QRMDD($k$)

**Definition 4.4** The **memory size for a QRMDD($k$)** is the number of bits needed to store the QRMDD($k$) in memory.

In memory, a non-terminal node in an MDD($k$) requires an index and a set of pointers that refer the succeeding nodes. However, in a QRMDD($k$), each non-terminal node has no index because $X_1, X_2, \ldots, X_u$ are evaluated always in this order, and the index of the super variable to evaluate can be obtained by a counter, where the super variable order is $X_1, X_2, \ldots, X_u$.

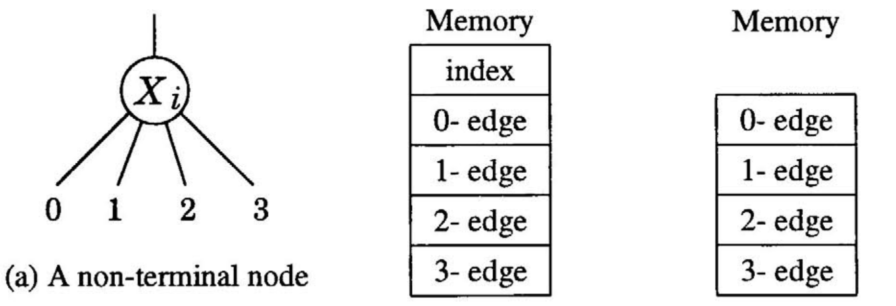
**Example 4.2** Fig. 4.4 illustrates data structures of a non-terminal node in an MDD(2) and a QRMDD(2). (End of Example)

Because each non-terminal node in a QRMDD($k$) has $2^k$ outgoing edges, we need

$$2^k nodes(\text{QRMDD}(k))$$

Table 4.5: Number of nodes in QRMDD($k$) for randomly generated functions.

| $n$ | $k$ | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| 10 | 247.4 | 101.0 | 77.0 | 33.0 | 33.0 |
| 11 | 437.1 | 251.2 | 89.0 | 179.2 | 37.0 |
| 12 | 754.0 | 356.5 | 296.5 | 272.5 | 49.0 |
| 13 | 1292.8 | 596.6 | 587.2 | 277.0 | 284.6 |
| 14 | 2316.0 | 1374.1 | 601.0 | 289.0 | 1050.1 |
| 15 | 4341.1 | 1625.0 | 841.0 | 529.0 | 1057.0 |
| 16 | 8336.5 | 5346.5 | 4554.5 | 4238.5 | 1061.0 |
| 17 | 16165.3 | 5721.0 | 4697.0 | 4373.0 | 1073.0 |
| 18 | 31155.9 | 19973.9 | 4937.0 | 4385.0 | 1313.0 |
| 19 | 58836.4 | 22105.0 | 30478.4 | 4625.0 | 26850.4 |
| 20 | 107220.3 | 63270.3 | 37465.0 | 45778.3 | 33825.0 |



(a) A non-terminal node in MDDs. (b) MDD(2). (c) QRMDD(2).

Figure 4.4: Data structure of a non-terminal node in DDs.

words to store all nodes in a QRMDD($k$). Since each node in a memory requires a unique address, each pointer requires

$$\lceil \log_2(nodes(\mathrm{QRMDD}(k))) \rceil$$

bits to specify the address. Therefore, the memory size for a QRMDD($k$) is

$$2^k nodes(\mathrm{QRMDD}(k)) \lceil \log_2(nodes(\mathrm{QRMDD}(k))) \rceil .$$

As shown in Section 4.3.2, for many benchmark functions, $nodes$(QRMDD($k$)) can be reduced with increasing $k$. On the other hand, the memory sizes for QRMDD($k$)s increase with

$2^k$. This fact shows that in QRMDD($k$), there exists optimum value of $k$ that minimizes the memory size.

### 4.4.2 Area-Time Complexity of QRMDD($k$)s

Because a QRMDD($k$) evaluates $k$ binary variables at a time, the path length of a QRMDD($k$) is $\frac{1}{k}$ of the corresponding QRBDD. On the other hand, the memory size for a QRMDD($k$) increases with $2^k$. In this section, we consider the area-time complexity [8, 76] for QRMDD($k$) and obtain the $k$ that minimizes the area-time complexity.

**Definition 4.5** The **area-time complexity** is the measure of computational cost considering both area and time. It is defined by

$$AT = (area) \times (time), \quad AT^2 = (area) \times (time)^2.$$

In this chapter, the area $A$ corresponds to the necessary memory size for QRMDD($k$), and the time $T$ corresponds to the number of memory accesses to evaluate logic function (i.e. path length of QRMDD($k$)).

The measure $AT$ is used when both the memory size and the path length are equally important. The measure $AT^2$ is used when the path length is more important than the memory size. For example, $AT$ can be used for software synthesis, while $AT^2$ can be used for logic simulators. In the software synthesis for embedded systems [2, 25, 27, 45], compact and fast program codes are required because of the memory limitations and the time limitations for systems. Thus, in the software synthesis using DDs, the optimization of DDs considering both the memory size and the number of memory accesses is important. In logic simulators [1, 22, 34, 35], fast evaluation of logic functions is more important to reduce the design verification time. Thus, in logic simulators, minimizing the number of memory accesses using a reasonable amount of memory is important.

### 4.4.3 Experimental Results

For each benchmark function in Table 4.2, we obtained three measures $A$, $AT$, and $AT^2$. Table 4.6, Table 4.7, and Table 4.8 show the relations of $k$ and $A$, $AT$, and $AT^2$, respectively. In these tables, *avg* denotes the arithmetic average, and *stdv* denotes the standard deviation for benchmark functions.

For each benchmark function in Table 4.2, $A$ takes its minimum when $k = 2$; $AT$ takes its minimum when $k = 3$ or $k = 4$; and $AT^2$ takes its minimum when $k = 4 \sim 6$.

Table 4.6: Relation of $k$ and $A$ for QRMDD($k$) for benchmark functions.

| | $k$ | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| *avg* | 1.00 | 0.90 | 1.14 | 1.61 | 2.54 |
| *stdv* | 0.000 | 0.035 | 0.079 | 0.144 | 0.292 |

Table 4.7: Relation of $k$ and $AT$ for QRMDD($k$) for benchmark functions.

| | $k$ | | | | |
|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 |
| *avg* | 1.00 | 0.46 | 0.39 | 0.42 | 0.54 |
| *stdv* | 0.000 | 0.019 | 0.030 | 0.039 | 0.070 |

### 4.4.4 Analysis for the Functions that Satisfy Property 4.1

In Section 4.4.3, for QRMDD($k$)s, we found the values of $k$ that make $A$, $AT$, and $AT^2$ minimum, experimentally. In this section, we assume that Property 4.1 holds, and will find the values $k$ that make $A$, $AT$, and $AT^2$ minimum, analytically. Let $A$ and $T$ be the memory size for a QRMDD($k$) and the number of memory accesses necessary to evaluate a QRMDD($k$), respectively. Then, we have the following:

$$A = 2^k nodes(\text{QRMDD}(k)) \lceil \log_2(nodes(\text{QRMDD}(k))) \rceil,$$

$$T = \lceil \frac{n}{k} \rceil.$$

Let $nodes(\text{QRMDD}(1)) = N$ and assume that Property 4.1 holds. Then we have:

$$A \simeq \frac{2^k}{k} N \lceil \log_2(\frac{N}{k}) \rceil,$$

$$AT \simeq \frac{2^k n}{k^2} N \lceil \log_2(\frac{N}{k}) \rceil,$$

$$AT^2 \simeq \frac{2^k n^2}{k^3} N \lceil \log_2(\frac{N}{k}) \rceil.$$

Note that $N$ is usually greater than 200, while $k$ is usually at most 7. Thus, we can use the following approximation:

$$\lceil \log_2(N) - \log_2(k) \rceil \simeq \lceil \log_2(N) \rceil.$$

Table 4.8: Relation of $k$ and $AT^2$ for QRMDD($k$) for benchmark functions.

| | $k$ | | | | | | |
|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| *avg* | 1.000 | 0.232 | 0.133 | 0.110 | 0.114 | 0.128 | 0.167 |
| *stdv* | 0.000 | 0.011 | 0.012 | 0.012 | 0.019 | 0.023 | 0.046 |

Therefore, $A, AT$, and $AT^2$ can be simplified to

$$A \simeq \frac{2^k}{k}C_0, \quad AT \simeq \frac{2^k}{k^2}C_1, \quad \text{and} \quad AT^2 \simeq \frac{2^k}{k^3}C_2,$$

respectively, where the constants $C_0$, $C_1$ and $C_2$ are independent of $k$. From the above formulas, we can see that $A, AT$, and $AT^2$ take their minimum when $k = 2$, $k = 3$, and $k = 4$, respectively.

## 4.5 Conclusion and Comments

In this chapter, we considered representations of binary logic functions using QRMDD($k$)s. Experimental results showed that: 1) For many benchmark functions, the numbers of nodes in QRMDD($k$)s are nearly equal to $\frac{1}{k}$ of the corresponding QRBDDs. On the other hand, for randomly generated functions, the number of nodes is a non-monotone function of $k$. 2) For many benchmark functions, the memory sizes and the area-time complexities for QRMDD($k$)s take their minimum when $k = 2$ and $k = 3 \sim 6$, respectively.

In commercial LUT-based FPGAs, the numbers of inputs $k$ for LUT cells are usually between 4 and 6 [10]. The studies in [28, 58] show that when $k = 4 \sim 6$, the architectures of FPGAs are optimum. The cost of $k$-LUT cell increases with $k$, while the level of network reduces with $k$. Thus, in logic synthesis with FPGAs, we can do a similar discussion. However, the optimum value of $k$ for FPGAs depends on interconnection delay, logic synthesis tools, and process technology as well as the cost of $k$-LUT cell and the level of networks [79]. It is interesting that in both cases, the optimum values of $k$ are $4 \sim 6$ even if they have different cost functions.

# Chapter 5

# Heterogeneous MDDs and Their Optimization Algorithms

This chapter proposes the representations of binary logic functions using heterogeneous MDDs and the optimization algorithms for heterogeneous MDDs that consider both orderings and partitions of binary variables.

## 5.1 Introduction

As shown in Chapter 4, when MDDs are used to represent binary logic functions, we can use an additional optimization approach, which is a partition of binary variables, as well as the variable ordering. To represent a binary logic function using an MDD, binary variables are partitioned into groups. In an MDD($k$), the groups have the same number of binary variables. On the other hand, in a heterogeneous MDD proposed in this chapter, the groups can have different numbers of binary variables. Thus, heterogeneous MDDs allow more flexible partition of binary variables than MDD($k$)s, and in heterogeneous MDDs, both orderings and partitions of binary variables can be optimized to minimize the memory sizes or APLs.

As shown in Chapters 3 and 4, APL minimization approaches using variable reordering and QRMDD($k$)s often increases the memory sizes of DDs. In fact, Table 3.2 shows that for benchmark function $C880$, APL minimization by variable ordering increases the number of nodes in the BDD by 10 times of original one. In QRMDD($k$)s, although path length can be reduced by increasing the value of $k$, it increases the memory size. However, in heterogeneous MDDs proposed in this chapter, APLs can be reduced without increasing the memory size by considering both orderings and partitions of binary variables.

The rest of this chapter is organized as follows: Section 5.2 defines heterogeneous MDDs.

Section 5.3 shows the number of different heterogeneous MDDs. Section 5.4 proposes memory size minimization algorithms for heterogeneous MDDs. Section 5.5 proposes APL minimization algorithms for heterogeneous MDDs. Section 5.6 compares memory sizes and APLs of heterogeneous MDDs for many benchmark functions.

## 5.2 Definitions

This section provides definitions used in this chapter.

**Definition 5.1** When $X = (x_1, x_2, \ldots, x_n)$ is partitioned into $(X_1, X_2, \ldots, X_u)$, an ROMDD representing a multi-valued input two-valued output function $f(X_1, X_2, \ldots, X_u)$ is called a **heterogeneous MDD**. Specially, when $k = |X_1| = |X_2| = \ldots = |X_u|$, an ROMDD for $f(X_1, X_2, \ldots, X_u)$ is called an **MDD($k$)**. A heterogeneous MDD represents a mapping $f : R_1 \times R_2 \times \ldots \times R_u \to B$, while an MDD($k$) represents a mapping $f : R^u \to B$, where $R_i = \{0, 1, \ldots, 2^{k_i} - 1\}$, $R = \{0, 1, \ldots, 2^k - 1\}$, and $B = \{0, 1\}$. In a heterogeneous MDD, non-terminal nodes representing a super variable $X_i$ have $2^{k_i}$ outgoing edges, where $k_i$ denotes the number of binary variables in $X_i$. Similarly, in an MDD($k$), non-terminal nodes have $2^k$ outgoing edges.

For $n$-variable logic functions $f$, if $n < ku$ (i.e. $n$ is indivisible by $k$), we use additional redundant binary variables to construct MDD($k$). The set of binary variables with dummy variables is denoted by $\{X'\} = \{x_1, x_2, \ldots, x_n, x_{n+1}, \ldots, x_{ku}\}$, where $|X'| = ku$. Note that $f$ is independent of $x_{n+1}, x_{n+2}, \ldots$ and $x_{ku}$.

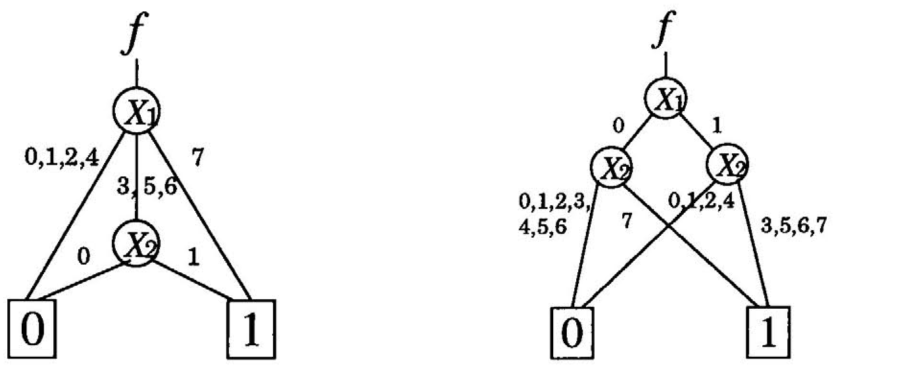
**Example 5.1** Consider the logic function $f = x_1x_2x_3 \vee x_2x_3x_4 \vee x_3x_4x_1 \vee x_4x_1x_2$ in Example 3.1. Fig. 5.1 shows the heterogeneous MDDs for $f$. In Fig. 5.1(a), the binary variables $X = (x_1, x_2, x_3, x_4)$ are partitioned into $(X_1, X_2)$, where $X_1 = (x_1, x_2, x_3)$ and $X_2 = (x_4)$. In Fig. 5.1(b), $X_1 = (x_1)$ and $X_2 = (x_2, x_3, x_4)$. (End of Example)

**Definition 5.2** The **width of a DD with respect to** $x_i$[1], denoted by $width(\mathrm{DD}, i)$, is the number of nodes in the DD corresponding to the variable $x_i$. The number of nodes in the DD is given by

$$nodes(\mathrm{DD}) = \sum_{i=1}^{n} width(\mathrm{DD}, i),$$

where $n$ denotes the number of variables.

[1]Note that this definition differs from that of "width of BDDs" in [41].



(a) Minimum heterogeneous MDD (b) Maximum heterogeneous MDD

Figure 5.1: Heterogeneous MDDs

**Definition 5.3** The **memory size of a DD**, denoted by $Mem(\text{DD})$, is the number of words needed to store all non-terminal nodes in the DD into a memory, where we assume that a word is large enough to store a variable index or an edge pointer.[2]

In memory, each non-terminal node in a DD requires an index and a set of pointers that refer the succeeding nodes. Since each non-terminal node in a BDD has two pointers, the memory size of a BDD is given by

$$Mem(\text{BDD}) = (2+1) \times nodes(\text{BDD}). \tag{5.1}$$

Similarly, since each non-terminal node in an MDD($k$) has $2^k$ pointers, the memory size of an MDD($k$) is given by

$$Mem(\text{MDD}(k)) = (2^k+1) \times nodes(\text{MDD}(k)).$$

In a heterogeneous MDD, each super variable can take different domain. Therefore, the memory size of heterogeneous MDD is calculated by summation for every super variables:

$$Mem(\text{heterogeneous MDD}) = \sum_{i=1}^{u} (2^{k_i}+1) \times width(\text{heterogeneous MDD}, i),$$

where $u$ and $k_i$ denote the number of super variables and the number of binary variables in a super variable $X_i$, respectively.

[2]Note that this definition slightly differs from Definition 4.4 in Chapter 4.

**Example 5.2** The memory sizes of BDD, MDD(2), and heterogeneous MDDs are as follows: for the BDD in Fig. 4.1(a), it is 18; for the MDD(2) in Fig. 4.1(b), it is 15; for the heterogeneous MDD in Fig. 5.1(a), it is 12; and for the heterogeneous MDD in Fig. 5.1(b), it is 21. (End of Example)

**Definition 5.4** Given a binary logic function $f$ and the order of binary variables, the **fixed-order minimum heterogeneous MDD** for the logic function $f$ is the heterogeneous MDD with the minimum memory size among the fixed-order partitions of the variables.

**Definition 5.5** Given a binary logic function $f$, the **minimum heterogeneous MDD** for the logic function $f$ is the heterogeneous MDD with the minimum memory size among the non-fixed-order partitions of the variables.

In this chapter, we use SDDs to represent multiple-output logic functions $F = (f_0, f_1, \dots, f_{m-1})$ [40]. APL of an SDD is the sum of the APLs of individual DDs for each logic function $f_i$.

## 5.3 Number of Heterogeneous MDDs

This section shows the number of different heterogeneous MDDs to estimate complexity of optimization for heterogeneous MDDs.

**Lemma 5.1** Let $N_{fix}(n)$ be the number of different fixed-order partitions of $X$. Then,

$$N_{fix}(n) = 2^{n-1}.$$

**Proof** See Appendix.

Therefore, when we fix the order of the binary variables $X = (x_1, x_2, \dots, x_n)$ and consider only partitions of binary variables for an optimization, the number of different heterogeneous MDDs to consider is $2^{n-1}$.

**Theorem 5.1** Let $N_{non\text{-}fix}(n)$ be the number of different non-fixed-order partitions of $X = (x_1, x_2, \dots, x_n)$. Then,

$$N_{non\text{-}fix}(n) = \sum_{r=1}^{n} \sum_{i=0}^{r} {}_rC_i (r-i)^n (-1)^i.$$

**Proof** See Appendix.

Table 5.1: Number of different DDs for $n$-variable logic function.

| | ROBDD | Heterogeneous MDD | | FBDD |
|---|---|---|---|---|
| $n$ | $n!$ | $N_{fix}(n)$ | $N_{non\text{-}fix}(n)$ | $S_n$ |
| 1 | 1 | 1 | 1 | 1 |
| 2 | 2 | 2 | 3 | 2 |
| 3 | 6 | 4 | 13 | 12 |
| 4 | 24 | 8 | 75 | 576 |
| 5 | 120 | 16 | 541 | 1658880 |
| 6 | 720 | 32 | 4683 | 16511297126400 |
| 7 | 5040 | 64 | 47293 | 1908360529573854283038720000 |
| 8 | 40230 | 128 | 545835 | – |
| 9 | 362880 | 256 | 7087261 | – |
| 10 | 3628800 | 512 | 102247563 | – |
| 11 | 39916800 | 1024 | 1622632573 | – |
| 12 | 479001600 | 2048 | 28091567595 | – |

Therefore, when we consider both orderings and partitions of the binary variables for an optimization of heterogeneous MDDs, the number of different heterogeneous MDDs for an $n$-variable logic function is given by $N_{non\text{-}fix}(n)$.

Table 5.1 compares the numbers of different ROBDDs, heterogeneous MDDs, and FBDDs for $n$-variable logic functions, where the number of different ROBDDs is equal to the number of different permutations of variables, that is $n!$, and the number of different FBDDs $S_n$ is given by [74]

$$S_n = nS_{n-1}^2 = \prod_{k=1}^{n} k^{2^{n-k}}.$$

When we fix the order of the binary variables and consider only partitions of binary variables for an optimization of heterogeneous MDDs, the number of heterogeneous MDDs to consider is smaller than that of ROBDDs. On the other hand, when we consider both orderings and partitions of the binary variables for an optimization of heterogeneous MDDs, the number of heterogeneous MDDs to consider is larger than that of ROBDDs. The number of different FBDDs is much larger than those of ROBDDs and heterogeneous MDDs.

When we assume a naive optimization method that finds an optimum solution by enumerating all possible ones, we have the followings:

1. When we fix the order of the binary variables and consider only partitions of binary

variables, an optimization of heterogeneous MDDs is easier than that of ROBDDs;

2. When we consider both orderings and partitions of the binary variables, an optimization of heterogeneous MDDs is more difficult than that of ROBDDs;

3. Optimizations of heterogeneous MDDs and ROBDDs are much easier than that of FB-DDs.

## 5.4 Memory Size Minimization Algorithms

Since memory size of a heterogeneous MDD depends on the partition of binary variables, as well as the order of binary variables, memory size of a heterogeneous MDD can be minimized by considering both orderings and partitions of binary variables.

**Example 5.3** Fig. 5.1(a) shows the minimum heterogeneous MDD for the function $f$, while Fig. 5.1(b) shows the maximum heterogeneous MDD for the function $f$. (End of Example)

In this section, we formulate the memory size minimization problem of heterogeneous MDDs considering both orderings and partitions of binary variables, and we present an exact minimization algorithm to solve it and a heuristic minimization algorithm.

We formulate the memory size minimization problem of heterogeneous MDDs considering both orderings and partitions of binary variables as follows:

**Problem 5.1** Given a binary logic function $f(X)$, find an order and a partition of $X$ that produces the minimum heterogeneous MDD for $f$.

### 5.4.1 Bounds on Memory Size of Heterogeneous MDDs

In this section, we derive upper and lower bounds on memory size of heterogeneous MDDs. Such bounds result in a reduction of the computation time in the algorithm, as discussed later.

**Theorem 5.2** In a fixed-order minimum heterogeneous MDD, the following relation holds for any super variable $X_i = (x_j, x_{j+1}, \ldots, x_{j+k_i-1})$:

$$(2^{k_i}+1)\,width(\text{heterogeneous MDD}, i) \leq 3 \times \sum_{t=0}^{k_i-1} width(\text{BDD}, j+t),$$

where the heterogeneous MDD and the BDD represent the same logic function, the variable order is fixed.

**Proof** See Appendix.

**Theorem 5.3** Consider a BDD and a heterogeneous MDD for an $n$-variable logic function that is not a constant function. When an order of binary variables is fixed, for the number of nodes in the BDD and the memory size of heterogeneous MDD obtained by considering only the fixed-order partitions, the following relation holds:

$$Mem(\text{heterogeneous MDD}) \geq nodes(\text{BDD}) + 2.$$

**Proof** See Appendix.

**Theorem 5.4** An arbitrary $n$-variable logic function can be represented by a heterogeneous MDD with at most the following memory size:

$$2^{n-r} + 3 \cdot 2^{2^r} - 5,$$

where $r$ is the largest integer satisfying the relation

$$n - r \geq 2^r + \log_2 3.$$

**Proof** See Appendix.

**Property 5.1** Consider a binary logic function $f(X)$. Let $Mem_{min}(f)$ be the memory size of a fixed-order minimum heterogeneous MDD for $f$. When $f$ is decomposed into $f = g(h(X_1), X_2)$, let $Mem_{min}(g)$ and $Mem_{min}(h)$ be the memory sizes of fixed-order minimum heterogeneous MDDs for $g$ and $h$, respectively. For many benchmark functions, the following two relations hold:

$$\begin{aligned} Mem_{min}(f) &> Mem_{min}(g) \\ Mem_{min}(f) &> Mem_{min}(h) \end{aligned}$$

### 5.4.2 Partition Algorithm for Memory Size Minimization

To solve Problem 5.1 efficiently, we use a partition algorithm that considers only the fixed-order partition of binary variables. This section presents the partition algorithm for memory size minimization.

Fig. 5.2 shows a pseudo-code for the partition algorithm. This algorithm uses dynamic programing. All sub-solutions are stored in the table. For simplicity, we assume that the variable order is $x_1, x_2, \ldots, x_n$.

**Algorithm 5.1**

```
 1: minimize_memory (BDD) {
 2:   table[$n$] = $(2+1)width(\text{BDD}, n)$ ;
 3:   for($i = n-1; i \geq 1; i--$) {
 4:     $min\_mem$ = (memory size of BDD) ;
 5:     for($l = 0; l \leq n-i; l++$) {
 6:       $k$ = **branch[$i$][$l$]** ;
 7:       $mdd\_mem = (2^k+1)width(\text{heterogeneous MDD}, j)$ ;
 8:       if ($mdd\_mem >$ **upper bound**)
 9:         break ;
10:       next index $i' = i+k$ ;
11:       $mdd\_mem$ += table[$i'$] ;
12:       if ($min\_mem > mdd\_mem$) {
13:         $min\_mem = mdd\_mem$ ;
14:         **register the partition $k$** ;
15:       }
16:     }
17:     table[$i$] = $min\_mem$ ;
18:   }
19:   return table[1] ;
20: }
```

Figure 5.2: Partition algorithm for memory size minimization.

This algorithm finds an optimum fixed-order partition. **table[$i$]** in Fig. 5.2 stores the fixed-order minimum memory size for sub-graph from $x_i$ to $x_n$. In the 6th line, **branch[$i$][$l$]** stores an integer $k$ that makes the following ratio the $l$-th smallest,

$$ratio = \frac{(2^k+1)width(\text{heterogeneous MDD}, j)}{3 \times \sum_{t=0}^{k-1} width(\text{BDD}, i+t)},$$

where $j$ is the index of corresponding super variable $X_j$. And, the 8th line uses **upper bound**, which is obtained by Theorem 5.2. The $j$ in the 7th line denotes the index of corresponding super variable $X_j$.

Let $n$ and $N$ be the numbers of binary variables and nodes for the BDD, respectively. Algorithm 5.1 examines at most $\frac{n^2}{2}$ candidates, and calculates the following value per the examina-

**Algorithm 5.2**

```
 1: exhaustive_search_memory (BDD) {
 2:   min_memory = minimize_memory (BDD) ;
 3:   for (all permutations of binary variables) {
 4:     Change the variable order for BDD ;
 5:     if (min_memory < nodes(BDD) + 2)
 6:       continue ;
 7:     current_memory = minimize_memory (BDD) ;
 8:     if (current_memory < min_memory) {
 9:       min_memory = current_memory ;
10:       Record the variable order for the BDD ;
11:       Record the partition of binary variables ;
12:     }
13:   }
14: }
```

Figure 5.3: Exact memory size minimization algorithm

tion:

$$(2^k + 1)width(\text{heterogeneous MDD}, j).$$

The time complexity to calculate it is $O(N)$. Therefore, the time complexity for Algorithm 5.1 is $O(n^2N)$. The space complexity for Algorithm 5.1 is $O(N)$.

### 5.4.3 Exact Memory Size Minimization Algorithm

When an order of binary variables is fixed, the memory size of a heterogeneous MDD depends on only the partition of binary variables. Thus, we use the following strategy for memory size minimization:

1. Change the order of binary variables; and

2. Fix the variable order, and change the partition of the binary variables.

Fig. 5.3 shows a pseudo-code to solve Problem 5.1. It uses a BDD for the given logic function as the internal representation. In the 2nd and 7th lines in Fig. 5.3, Algorithm 5.1 is used to find an optimum fixed-order partition that produces the fixed-order minimum heterogeneous

MDD. In the 5th line, Theorem 5.3 is used to reduce the computation time. This algorithm finds the minimum heterogeneous MDD by exhaustive search.

### 5.4.4 Heuristic Memory Size Minimization Algorithm

Although Algorithm 5.2 can find the minimum heterogeneous MDD, enumerating all possible permutations of binary variables is impractical when the number of binary variables is large, as shown in Table 5.1. Thus, this section proposes a heuristic minimization for heterogeneous MDDs using the sifting algorithm [59] and partition algorithm (Algorithm 5.1). The sifting algorithm repeatedly performs the following basic steps:

1. Change the variable order.

2. Compute a cost.

Most sifting algorithms use the number of nodes in DD as the cost. In memory size minimization, however, we use the memory size of heterogeneous MDD as the cost.

Fig. 5.4 shows a pseudo-code for 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 the 10th line in Fig. 5.4, Property 5.1 is used to find useful siftings of $x_i$. The $L_{mem}$ in the 9th line denotes the memory size of fixed-order minimum heterogeneous MDD for logic function $g$ or $h$ obtained by functional decomposition $f(X) = g(h(X_1), X_2)$. When $x_i$ moves down to the bottom of the BDD, we use $h$ to compute $L_{mem}$, where $X_1$ contains the binary variables which are above the level of $x_i$ in the variable order, and $X_2$ contains the remaining ones. If $cost \leq L_{mem}$, we stop the sifting of $x_i$ to the bottom because sifting of $x_i$ further down to the bottom seldom reduces the memory size due to Property 5.1. Similarly, when $x_i$ moves up to the top of the BDD, we use $g$ to compute $L_{mem}$, where $X_2$ contains the binary variables which are below the level of $x_i$ in the variable order, and $X_1$ contains the remaining ones. This lower bound for the memory size is similar to the one introduced for the number of nodes during the classical sifting [14].

## 5.5 APL Minimization Algorithms

Since APL of a heterogeneous MDD also depends on the partition of binary variables, as well as the order of binary variables, APL of a heterogeneous MDD can be minimized by considering both orderings and partitions of binary variables.

**Algorithm 5.3**

```
sifting_memory (BDD) {
    $cost$ = minimize_memory (BDD) ;
    do {
        for ($\forall x_i \in X$) {
            $best\_p$ = current position of $x_i$ ;
            for (all position $p$) {
                Move $x_i$ to position $p$ ;
                $memory$ = minimize_memory (BDD) ;
                Compute $L_{mem}$ ;
                if ($cost \leq L_{mem}$)
                    break ;
                if ($memory < cost$) {
                    $cost = memory$ ;
                    $best\_p = p$ ;
                    Record the partition of binary variables ;
                }
            }
            Move $x_i$ to $best\_p$ ;
        }
    } while ($cost$ is reduced) ;
}
```

Figure 5.4: Heuristic memory size minimization algorithm

**Example 5.4** The APLs of BDD, MDD(2), and heterogeneous MDDs are as follows: for the BDD in Fig. 4.1(a), it is 3.125; for the MDD(2) in Fig. 4.1(b), it is 1.75; for the heterogeneous MDD in Fig. 5.1(a), it is 1.375; and for the heterogeneous MDD in Fig. 5.1(b), it is 2.0. Note that $P(x_i = 0) = P(x_i = 1) = 0.5$. (End of Example)

In this section, we formulate the APL minimization problem of heterogeneous MDDs considering both orderings and partitions of binary variables, and we present an exact minimization algorithm to solve it and a heuristic minimization algorithm.

For any $n$-variable logic function $f(X)$, the trivial partition of $X$, where $X = X_1$ and $|X_1| = n$, produces a heterogeneous MDD with the smallest APL (i.e., $APL = 1.0$), independently of the variable ordering. However, since the memory size of the heterogeneous MDD for the trivial partition is nearly $2^n$, such a heterogeneous MDD is impractical in most cases. Therefore, we seek an order and a partition of $X$ that minimizes the APL within a given memory size limitation. We formulate the APL minimization problem considering both orderings and partitions of binary variables as follows:

**Problem 5.2** Given a binary logic function $f(X)$ and a memory size limitation $L$, find an order and a partition of $X$ that produces the heterogeneous MDD with the minimum APL and with memory size equal to or smaller than $L$.

### 5.5.1 Partition Algorithm for APL Minimization

To solve Problem 5.2 efficiently, we use a partition algorithm that considers only the fixed-order partition of binary variables. This section presents the partition algorithm for APL minimization.

Fig. 5.5 shows a pseudo-code for the partition algorithm for APL minimization. This algorithm uses a branch-and-bound method and a cache to reduce computation time. The sub-solutions are stored in the cache, but only a subset of sub-solutions is kept in it because the number of sub-solutions is too large in many cases. In other words, this algorithm is similar to the dynamic programing, except for that the cache is overwritten. In the case of cache miss, the sub-solution is searched again. Since this algorithm is recursive procedure, the top level for BDD (i.e. $level = 1$) and the memory size limitation $L$ are required as the initial arguments.

This algorithm produces an optimum fixed-order partition by calculating the APLs for different partitions of $X$. The calculation of the APL uses Theorem 3.1. To compute the node traversing probability $prob(\text{heterogeneous MDD}, v)$ of the 17th line, we used the computation method in Section 3.3. The 13th line uses **lower bounds** on the memory size obtained by Algorithm 5.1 to reduce computation time.

Let $n$, $N$, and $C$ be the number of binary variables, the number of nodes in the BDD, and the cache size, respectively. Algorithm 5.4 examines at most $2^{n-1}$ candidates by exhaustive search. The time complexities for the calculations of **lower bounds** and the value of $prob(\text{heterogeneous MDD}, v)$ are $O(n^2N)$ and $O(N)$, respectively. Note that these values are calculated before the exhaustive search and stored in tables. Therefore, the time complexity for Algorithm 5.4 is $O(2^n + n^2N)$. The space complexity for Algorithm 5.4 is $O(N+C) = O(N)$, where $C$ is considered as a constant value.

**Algorithm 5.4**

```
minimize_APL ($level$, mem_size $l$, BDD) {
    if ($level > n$)
        return 0 ;
    check the cache ;
    if (cache.level == $level$ && cache.mem == $l$) {
        register the partition cache.k ;
        return cache.APL ;
    }
    $min\_APL$ = (APL for BDD) ;
    for ($k = n - level + 1; k \geq 1; k - -$) {
        $memory = (2^k + 1)\ width$(heterogeneous MDD, $j$) ;
        next level $level' = level + k$ ;
        if ($(l - memory) <$ lower_bound[$level'$])
            continue ;
        $current\_APL = 0$ ;
        for (all nodes $v$ representing $X_j$)
            $current\_APL$ += $prob$(heterogeneous MDD, $v$) ;
        $current\_APL$ += minimize_APL ($level'$, $l - memory$, BDD) ;
        if ($current\_APL < min\_APL$) {
            register the partition $k$ ;
            $min\_APL = current\_APL$ ;
        }
    }
    store (overwrite) to the cache ;
    return $min\_APL$ ;
}
```

Figure 5.5: Partition algorithm for APL minimization.

## 5.5.2 Exact APL Minimization Algorithm

When an order of binary variables is fixed, the APL of a heterogeneous MDD depends on only the partition of binary variables. Thus, we use the same strategy as the memory size minimization.

**Algorithm 5.5**

```
 1: exhaustive_search_APL (BDD, memory size limitation $L$) {
 2:   $min\_APL$ = **minimize_APL** (1, $L$, BDD) ;
 3:   for (all permutations of binary variables) {
 4:     Change the variable order for BDD ;
 5:     if ($L < nodes$(BDD) $+ 2$)
 6:       continue ;
 7:     $memory$ = minimize_memory (BDD) ;
 8:     if ($L < memory$)
 9:       continue ;
10:     $APL$ = **minimize_APL** (1, $L$, BDD) ;
11:     if ($APL < min\_APL$) {
12:       $min\_APL = APL$ ;
13:       Record the variable order for the BDD ;
14:       Record the partition of binary variables ;
15:     }
16:   }
17: }
```

Figure 5.6: Exact APL minimization algorithm

Fig. 5.6 shows a pseudo-code to solve Problem 5.2. In the 2nd and 10th lines in Fig. 5.6, Algorithm 5.4 is used to find an optimum fixed-order partition that minimizes the APL of heterogeneous MDD within a memory size limitation $L$. Since it is recursive procedure, the top level for BDD (i.e. $level = 1$) is required as the initial argument. This algorithm finds an optimum solution for Problem 5.2 by exhaustive search.

### 5.5.3 Heuristic APL Minimization Algorithm

As well as the memory size minimization, Algorithm 5.5 is time-consuming for functions with many inputs. Thus, this section proposes a heuristic APL minimization method for heterogeneous MDDs using a sifting algorithm and partition algorithm (Algorithm 5.4).

Fig. 5.7 shows a pseudo-code for the heuristic APL minimization algorithm. In this algorithm, the APL of a heterogeneous MDD is used as the cost.

**Algorithm 5.6**

```
sifting_APL (BDD, $L$, #sifting rounds $R$) {
  $cost$ = minimize_APL (1, $L$, BDD) ;
  for ($r = 0$; $r < R$; $r$++) {
    for ($\forall x_i \in X$) {
      $best\_p$ = current position of $x_i$ ;
      for (all positions $p$) {
        Move $x_i$ to position $p$ ;
        $memory$ = minimize_memory (BDD) ;
        Compute $L_{mem}$ ;
        if ($L \leq L_{mem}$)
          break ;
        if ($L < memory$)
          continue ;
        $APL$ = minimize_APL (1, $L$, BDD) ;
        if ($APL < cost$) {
          $cost = APL$ ;
          $best\_p = p$ ;
          Record the partition of binary variables ;
        }
      }
      Move $x_i$ to $best\_p$ ;
    }
  }
}
```

Figure 5.7: Heuristic APL minimization algorithm

## 5.6 Experimental Results

To show the compactness of heterogeneous MDD and the efficiency of optimization algorithms, we compare heterogeneous MDDs with the different types of DDs using benchmark functions. Experiments were conducted in the following environment:

- CPU: Pentium4 Xeon 2.8GHz

Table 5.2: Memory sizes of OBDDs, FBDDs, and heterogeneous MDDs for all 4-variable logic functions

| Group | OBDD | | | FBDD | | | Heterogeneous MDD | | |
|---|---|---|---|---|---|---|---|---|---|
| No. | Mem | #class | #function | Mem | #class | #function | Mem | #class | #function |
| 0 | 0 | 1 | 2 | 0 | 1 | 2 | 0 | 1 | 2 |
| 1 | 3 | 1 | 8 | 3 | 1 | 8 | 3 | 1 | 8 |
| 2 | 6 | 1 | 48 | 6 | 1 | 48 | 5 | 1 | 48 |
| 3 | 9 | 4 | 364 | 9 | 4 | 364 | 5 | 1 | 12 |
| | | | | | | | 8 | 3 | 352 |
| 4 | 12 | 14 | 3168 | 12 | 14 | 3168 | 8 | 3 | 320 |
| | | | | | | | 9 | 1 | 96 |
| | | | | | | | 10 | 6 | 1216 |
| | | | | | | | 11 | 4 | 1536 |
| 5 | 15 | 38 | 12440 | 15 | 38 | 12440 | 9 | 3 | 104 |
| | | | | | | | 10 | 7 | 1056 |
| | | | | | | | 11 | 13 | 4400 |
| | | | | | | | 12 | 12 | 6528 |
| | | | | | | | 14 | 3 | 352 |
| 6 | 18 | 70 | 22488 | 18 | 70 | 22488 | 10 | 3 | 168 |
| | | | | | | | 12 | 41 | 12064 |
| | | | | | | | 14 | 13 | 4928 |
| | | | | | | | 15 | 13 | 5328 |
| 7 | 21 | 68 | 20346 | 18 | 3 | 1536 | 12 | 11 | 3520 |
| | | | | 21 | 65 | 18810 | 15 | 57 | 16826 |
| 8 | 24 | 25 | 6672 | 21 | 10 | 4032 | 15 | 25 | 6672 |
| | | | | 24 | 15 | 2640 | | | |
| Avg. | 1.00 | – | – | 0.99 | – | – | 0.72 | – | – |

- 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 $P(x_i = 0) = P(x_i = 1) = 0.5$ for binary logic functions.

### 5.6.1 Comparison with FBDDs

In this section, we compare heterogeneous MDDs with FBDDs to show the compactness of heterogeneous MDDs. FBDDs allow the different variable orders along each path, and are well known as more compact DDs than OBDDs.

We implemented Algorithm 5.2 and compared the minimum heterogeneous MDDs with the minimum OBDDs and the minimum FBDDs for all 4 and 5-variable logic functions. To compare them, we classified all the logic functions into NPN-equivalence classes [43, 63]. For the 4-variable case, $65,536$ functions are classified into 222 NPN-equivalence classes, and for the 5-variable case, $4,294,967,296$ functions are classified into $616,126$ NPN-equivalence classes. Table 5.2 compares minimum DD sizes for the 4-variable case. In Table 5.2, 222 NPN-representative functions are grouped into 9 rows according to the memory size of the minimum OBDD. The column "Mem" denotes the memory size of each DD. The columns "#class" and "#function" in Table 5.2 denote the number of NPN-equivalence classes and the number of functions included in the classes, respectively. The bottom row "Avg." denotes the arithmetic average of the relative memory sizes for all functions, where the memory size of OBDD is set to 1.00. In this experiment, no complemented edges [6, 40] are used in OBDDs, FBDDs, or heterogeneous MDDs.

For the 4-variable case, FBDDs are smaller than OBDDs for $5,568$ functions, 8.5% of all functions, while heterogeneous MDDs are smaller than OBDDs and FBDDs for all functions except for 10 degenerate functions (0, 1, $x_i$, and $\bar{x}_i$ where $i = 1,2,3,4$). For these 10 functions, the memory sizes of OBDDs, FBDDs, and heterogeneous MDDs are equal. On average over all functions, minimum FBDDs require 99% of the memory size of minimum OBDDs, while minimum heterogeneous MDDs require 72% of the memory size for minimum OBDDs.

For the 5-variable case, FBDDs are smaller than OBDDs for $1,938,548,576$ functions, 45% of all functions, while heterogeneous MDDs are smaller than OBDDs for $4,294,967,284$ functions, 99% of all functions. Also, heterogeneous MDDs are smaller than FBDDs for $4,294,921,204$ functions, 99% of all functions, and for the others, heterogeneous MDDs are equal in size to FBDDs. There was no function whose FBDD is smaller than the heterogeneous MDD. On average over all functions, minimum FBDDs require 96% of the memory size for minimum OBDDs, while minimum heterogeneous MDDs require 67% of the memory size for minimum OBDDs.

Algorithm 5.2 could obtain exact minimum heterogeneous MDDs for the functions with up to 12 inputs within a reasonable computation time, while the exact FBDD minimization [18] can find the minimum one for the functions with up to 8 inputs.

Table 5.3: Memory sizes of OBDDs, FBDDs, and heterogeneous MDDs for MCNC benchmark functions

| | | | Memory Size | | |
|---|---|---|---|---|---|
| Name | In | Out | OBDD | FBDD | MDD |
| C432 | 36 | 7 | 3189 | 3171 | 2824 |
| C499 | 41 | 32 | 77595 | 77595 | 59739 |
| C880 | 60 | 26 | 12156 | 8394 | 11812 |
| C1908 | 33 | 25 | 16575 | 15141 | 13493 |
| C2670 | 233 | 64 | 5319 | 3186 | 4649 |
| C3540 | 50 | 22 | 71481 | 62997 | 65029 |
| C5315 | 178 | 123 | 5154 | 4434 | 4582 |
| C7552 | 207 | 107 | 6633 | 4782 | 6119 |
| alu4 | 14 | 8 | 1047 | 900 | 855 |
| apex1 | 45 | 45 | 3735 | 3531 | 3016 |
| apex6 | 135 | 99 | 1491 | 1365 | 1414 |
| cps | 24 | 102 | 2910 | 2706 | 2533 |
| dalu | 75 | 16 | 2064 | 1947 | 1548 |
| des | 256 | 245 | 8832 | 8706 | 7288 |
| frg2 | 143 | 139 | 2886 | 2760 | 2671 |
| i3 | 132 | 6 | 396 | 396 | 330 |
| i8 | 133 | 81 | 3825 | 3570 | 3662 |
| i10 | 257 | 224 | 61977 | 56439 | 55766 |
| k2 | 45 | 45 | 3735 | 3408 | 3018 |
| too_large | 38 | 3 | 954 | 858 | 857 |
| vda | 17 | 39 | 1431 | 1401 | 1088 |
| Average of ratios | | | 1.00 | 0.90 | 0.86 |

Table 5.3 compares heterogeneous MDDs with OBDDs and FBDDs for selected MCNC benchmark functions. The OBDDs are obtained by the best known variable orders [72], and the numbers of nodes for FBDDs are taken from [18, 19]. The memory sizes of OBDDs and FBDDs are calculated by the formula (5.1) in Section 5.2. The columns "In" and "Out" in Table 5.3 denote the number of inputs and outputs for each benchmark function, respectively. Column "MDD" denotes the heterogeneous MDDs obtained by Algorithm 5.3, where the OBDDs [72] are used as initial solutions. The DDs in this table may not be the exact minimum since the algorithms are heuristic methods. The bottom row "Average of ratios" denotes the arithmetic average of the relative memory size, where the memory size of OBDD is set to 1.00. In this

Table 5.4: Memory sizes and APLs of ROBDDs and heterogeneous MDDs for $n$-variable logic functions

| | Memory size | | | | APL | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | BDD | | Heterogeneous MDD | | BDD | | Heterogeneous MDD | | |
| $n$ | MinNodes | MinAPL$_B$ | MinMem | MinAPL$_M$ | MinNodes | MinAPL$_B$ | MinMem | MinAPL$_M$ | #samples |
| 4 | 1.00 | 1.07 | 0.72 | 0.86 | 1.00 | 0.99 | 0.52 | 0.37 | $2^{16}$ |
| 5 | 1.00 | 1.07 | 0.67 | 0.91 | 1.00 | 0.98 | 0.39 | 0.27 | $2^{32}$ |
| 6 | 1.00 | 1.08 | 0.68 | 0.80 | 1.00 | 0.97 | 0.45 | 0.32 | 1,000 |
| 7 | 1.00 | 1.08 | 0.64 | 0.79 | 1.00 | 0.97 | 0.40 | 0.27 | 1,000 |
| 8 | 1.00 | 1.08 | 0.58 | 0.81 | 1.00 | 0.97 | 0.33 | 0.22 | 1,000 |
| 9 | 1.00 | 1.07 | 0.55 | 0.83 | 1.00 | 0.98 | 0.29 | 0.19 | 1,000 |
| 10 | 1.00 | 1.06 | 0.54 | 0.84 | 1.00 | 0.98 | 0.26 | 0.17 | 1,000 |

experiment, OBDDs, FBDDs, and heterogeneous MDDs use complemented edges.

Heterogeneous MDDs require smaller memory size than FBDDs for 14 out of 21 benchmark functions in Table 5.3. Especially, for *C499, dalu,* and *vda,* heterogeneous MDDs require at most 80% of the memory sizes for the FBDDs.

## 5.6.2 Comparison with ROBDDs

Table 5.4 compares the memory sizes and the APLs of BDDs and heterogeneous MDDs for $n$-variable logic functions. The BDDs and heterogeneous MDDs are optimized using four different algorithms: (1) exact nodes minimization algorithm for a BDD considering only the orderings (column "MinNodes"); (2) exact APL minimization algorithm for a BDD (Algorithm 3.1) considering only the orderings of binary variables (column "MinAPL$_B$"); (3) exact memory size minimization algorithm for a heterogeneous MDD (Algorithm 5.2) considering both orderings and partitions of binary variables (column "MinMem"); and (4) exact APL minimization algorithm for a heterogeneous MDD (Algorithm 5.5) considering both orderings and partitions of binary variables (column "MinAPL$_M$"). The memory size limitations $L$ for Algorithm 5.5 are set to the memory sizes of the BDDs in "MinNodes". The values in this table are the normalized averages of $n$-variable logic functions, where the memory sizes and APLs of "MinNodes" are set to 1.00. Columns "MinAPL$_B$", "MinMem", and "MinAPL$_M$" show the relative values of the memory sizes and APLs to "MinNodes". Columns "#samples" denotes the number of sample functions used for each $n$-variable function. Note that the BDDs and heterogeneous MDDs in this table do not use complemented edges.

For 4 and 5-variable logic functions, we calculated the exact averages over all functions. We did this by recognizing that the minimum memory size and APL for a function in one NPN-equivalence class [43, 63] are identical to the minimum memory sizes and APLs for other

Table 5.5: Memory sizes and APLs of ROBDDs and heterogeneous MDDs for MCNC benchmark functions

| | | | Memory size | | | | APL | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | BDD | | Heterogeneous MDD | | BDD | | Heterogeneous MDD | |
| Name | In | Out | MinNodes | MinAPL$_B$ | MinMem | MinAPL$_M$ | MinNodes | MinAPL$_B$ | MinMem | MinAPL$_M$ |
| C432 | 36 | 7 | 3189 | 3243 | 2824 | 3179 | 86.58 | 86.24 | 55.74 | 45.45 |
| C499 | 41 | 32 | 77595 | 96315 | 59739 | 77589 | 813.64 | 641.16 | 381.14 | 192.52 |
| C880 | 60 | 26 | 12156 | 54810 | 11812 | 12154 | 135.79 | 121.03 | 125.73 | 99.13 |
| C1908 | 33 | 25 | 16575 | 56328 | 13493 | 16564 | 254.35 | 183.61 | 145.81 | 92.09 |
| C2670 | 233 | 64 | 5319 | 8286 | 4649 | 5319 | 214.05 | 202.08 | 167.90 | 133.78 |
| C3540 | 50 | 22 | 71481 | 74292 | 65029 | 71480 | 209.15 | 208.06 | 141.10 | 91.78 |
| C5315 | 178 | 123 | 5154 | 5460 | 4582 | 5153 | 462.05 | 446.26 | 373.23 | 304.38 |
| C7552 | 207 | 107 | 6633 | 6585 | 6119 | 6633 | 484.03 | 469.54 | 424.85 | 314.03 |
| alu4 | 14 | 8 | 1047 | 1080 | 855 | 1019 | 40.81 | 40.70 | 24.41 | 19.59 |
| apex1 | 45 | 45 | 3735 | 4254 | 3016 | 3728 | 180.59 | 177.69 | 87.35 | 67.63 |
| apex6 | 135 | 99 | 1491 | 1887 | 1414 | 1490 | 291.54 | 230.91 | 260.66 | 231.06 |
| cps | 24 | 102 | 2910 | 4656 | 2533 | 2906 | 290.25 | 235.39 | 187.90 | 151.81 |
| dalu | 75 | 16 | 2064 | 2970 | 1548 | 2064 | 102.67 | 78.81 | 39.40 | 28.09 |
| des | 256 | 245 | 8832 | 9177 | 7288 | 8831 | 1210.00 | 1080.38 | 910.63 | 687.50 |
| frg2 | 143 | 139 | 2886 | 5070 | 2671 | 2884 | 624.69 | 322.17 | 499.27 | 348.60 |
| i3 | 132 | 6 | 396 | 396 | 330 | 396 | 26.76 | 26.76 | 17.84 | 12.61 |
| i8 | 133 | 81 | 3825 | 6954 | 3662 | 3825 | 302.54 | 270.82 | 229.12 | 207.54 |
| i10 | 257 | 224 | 61977 | 685215 | 55766 | 61974 | 1084.96 | 776.10 | 887.62 | 614.53 |
| k2 | 45 | 45 | 3735 | 4254 | 3018 | 3728 | 180.52 | 177.69 | 87.32 | 67.61 |
| too_large | 38 | 3 | 954 | 2361 | 857 | 954 | 13.16 | 11.52 | 8.47 | 6.24 |
| vda | 17 | 39 | 1431 | 1515 | 1088 | 1424 | 176.34 | 171.54 | 81.72 | 69.54 |
| Average of ratios | | | 1.00 | 2.03 | 0.86 | 1.00 | 1.00 | 0.88 | 0.67 | 0.51 |

functions in the same class. Thus, it is sufficient to consider only one function from each class and form a sum weighted by the size of each class. For larger $n$, there are too many NPN-equivalence classes. For $6 \leq n \leq 10$, we generated 1,000 pseudo-random $n$-variable logic functions with different number of minterms, and calculated the normalized averages for them.

For BDDs, APLs can be reduced up to 97% of BDDs with the minimum nodes, but the memory sizes increases to 108%. On the other hand, for heterogeneous MDDs, the APLs can be reduced up to 17% of BDDs with the minimum nodes without increasing memory sizes, and both the memory sizes and APLs can be reduced up to 54% and 26% of minimum BDDs, respectively. Table 5.4 shows that the relative values of memory sizes and APLs for heterogeneous MDDs decreases as the number of binary variables $n$ increases. Algorithm 5.5 finds exact minimum APLs of heterogeneous MDDs for the functions with up to 11 variables within a reasonable computation time.

Table 5.5 compares memory sizes and APLs of BDDs and heterogeneous MDDs for same MCNC benchmark functions as Table 5.3. Columns labeled "MinNodes" denote the BDDs obtained by the best known variable orders [72]. These are used as the initial BDDs for the algorithms in this experiment. Columns "MinAPL$_B$" denote the BDDs obtained by Algorithm 3.2. Columns "MinMem" denote the heterogeneous MDDs obtained by Algorithm 5.3. And, columns "MinAPL$_M$" denote the heterogeneous MDDs obtained by Algorithm 5.6. The memory size limitations $L$ for Algorithm 5.6 are set to the memory sizes of the BDD in "MinNodes". In Algorithm 3.2 and Algorithm 5.6, the number of rounds of sifting is set to two. Note that the BDDs and heterogeneous MDDs in this table use complemented edges. The memory sizes and APLs in this table may not be exact minimum since the algorithms are heuristic methods. The row labeled *Average of ratios* represents the normalized averages of memory size and APL, where the memory size and the APL of "MinNodes" are set to 1.00.

Algorithm 3.2 that considers only variable orderings reduced APLs to 88% of "MinNodes", on average, but increased the memory sizes by twice. Especially, for *C880, C1908, i10,* and *too_large*, Algorithm 3.2 increased the memory sizes significantly. On the other hand, by considering both orderings and partitions of binary variables, Algorithm 5.3 reduced both memory sizes and APLs to 86% and 67% of "MinNodes", respectively. Algorithm 5.6 reduced APLs to 51% of "MinNodes" without increasing the memory size.

### 5.6.3 Comparison of Computation Time for Algorithms

Table 5.6 compares the computation times for Algorithm 3.2, Algorithm 5.3, and Algorithm 5.6. The values in Table 5.6 show the CPU times needed to obtain the BDDs and heterogeneous MDDs in Table 5.5, in seconds.

Although Algorithm 5.3 considers both orderings and partitions of binary variables for memory size minimization, its computation time is as short as that of Algorithm 3.2 that considers only variable orderings for APL minimization. Algorithm 5.6 requires longer computation time than other two algorithms, since Algorithm 5.6 considers memory size to keep a memory size limitation, as well as APL.

### 5.6.4 Comparison with MDD($k$)s

Similarly, we compared heterogeneous MDDs with MDD($k$)s.

Table 5.7 and Table 5.8 compare the memory sizes and APLs of BDDs, heterogeneous MDDs, and MDD($k$)s for $n$-variable logic functions, respectively. In these tables, MDD($k$)s have the exact fewest nodes. The values in these tables are the normalized averages of $n$-variable logic functions, where the memory sizes and APLs of BDD with the fewest nodes (column "BDD") are set to 1.00. Columns "MinMem", "MinAPL$_M$", "MDD(2)s", "MDD(3)", "MDD(4)", and "MDD(5)" show the relative values of the memory sizes and APLs to "BDD".

Table 5.6: CPU times [sec] for memory size and APL minimization algorithms

| Name | MinAPL$_B$ | MinMem | MinAPL$_M$ |
|---|---|---|---|
| C432 | 0.23 | 0.23 | 1.04 |
| C499 | 10.76 | 5.12 | 698.31 |
| C880 | 4.54 | 1.23 | 22.09 |
| C1908 | 1.44 | 0.67 | 27.38 |
| C2670 | 2.21 | 1.99 | 1957.51 |
| C3540 | 12.74 | 36.96 | 523.45 |
| C5315 | 0.43 | 1.31 | 3663.57 |
| C7552 | 1.35 | 4.76 | 2258.88 |
| alu4 | 0.02 | 0.02 | 0.05 |
| apex1 | 0.11 | 0.29 | 36.07 |
| apex6 | 0.05 | 0.33 | 79.47 |
| cps | 0.09 | 0.12 | 0.80 |
| dalu | 0.15 | 0.25 | 132.41 |
| des | 0.91 | 3.59 | 60144 |
| frg2 | 0.29 | 0.89 | 218.46 |
| i3 | 0.01 | 0.23 | 95.69 |
| i8 | 0.31 | 0.59 | 30.15 |
| i10 | 160.91 | 69.27 | 71464 |
| k2 | 0.11 | 0.29 | 33.99 |
| too_large | 0.07 | 0.07 | 0.31 |
| vda | 0.02 | 0.01 | 0.15 |

From Table 5.7 and Table 5.8, we can see that for $n$-variable logic functions, heterogeneous MDDs obtained by Algorithm 5.5 have the APLs as small as MDD(5)s. The memory sizes of MDD(5)s are twice the memory sizes of BDDs. On the other hand, heterogeneous MDDs have smaller memory sizes than the BDDs.

Table 5.9 and Table 5.10 compare the memory sizes and APLs of BDDs, heterogeneous MDDs, and MDD($k$)s for MCNC benchmark functions, respectively. MDD($k$)s in these tables are obtained by the minimization algorithm in [62]. BDDs and heterogeneous MDDs are the same as those in Table 5.5.

Table 5.9 and Table 5.10 show that in heterogeneous MDDs, APLs can be reduced to a half of the BDDs without increasing memory sizes. On the other hand, in MDD($k$)s, to reduce the APLs to a half of the BDDs, we need to increase the memory sizes to 488% of the BDDs. The APLs of heterogeneous MDDs obtained by memory size minimization algorithm

Table 5.7: Memory sizes of BDDs, heterogeneous MDDs, and MDD($k$)s for $n$-variable logic functions

| $n$ | BDD | Heterogeneous MDD | | MDD(2) | MDD(3) | MDD(4) | MDD(5) | #samples |
|---|---|---|---|---|---|---|---|---|
| | | MinMem | MinAPL$_M$ | | | | | |
| 4 | 1.00 | 0.72 | 0.86 | 0.96 | 1.22 | 0.94 | 1.83 | $2^{16}$ |
| 5 | 1.00 | 0.67 | 0.91 | 0.93 | 1.24 | 1.47 | 0.99 | $2^{32}$ |
| 6 | 1.00 | 0.68 | 0.80 | 0.92 | 1.10 | 1.50 | 2.02 | 1,000 |
| 7 | 1.00 | 0.64 | 0.79 | 0.90 | 0.98 | 1.45 | 1.87 | 1,000 |
| 8 | 1.00 | 0.58 | 0.81 | 0.85 | 0.98 | 1.49 | 1.79 | 1,000 |
| 9 | 1.00 | 0.55 | 0.83 | 0.83 | 1.24 | 1.07 | 1.87 | 1,000 |
| 10 | 1.00 | 0.54 | 0.84 | 0.81 | 0.82 | 0.96 | 2.01 | 1,000 |

Table 5.8: APLs of BDDs, heterogeneous MDDs, and MDD($k$)s for $n$-variable logic functions

| $n$ | BDD | Heterogeneous MDD | | MDD(2) | MDD(3) | MDD(4) | MDD(5) | #samples |
|---|---|---|---|---|---|---|---|---|
| | | MinMem | MinAPL$_M$ | | | | | |
| 4 | 1.00 | 0.52 | 0.37 | 0.59 | 0.47 | 0.34 | 0.34 | $2^{16}$ |
| 5 | 1.00 | 0.39 | 0.27 | 0.60 | 0.47 | 0.36 | 0.25 | $2^{32}$ |
| 6 | 1.00 | 0.45 | 0.32 | 0.59 | 0.43 | 0.43 | 0.33 | 1,000 |
| 7 | 1.00 | 0.40 | 0.27 | 0.58 | 0.42 | 0.36 | 0.36 | 1,000 |
| 8 | 1.00 | 0.33 | 0.22 | 0.55 | 0.41 | 0.30 | 0.31 | 1,000 |
| 9 | 1.00 | 0.29 | 0.19 | 0.59 | 0.38 | 0.32 | 0.26 | 1,000 |
| 10 | 1.00 | 0.26 | 0.17 | 0.53 | 0.43 | 0.31 | 0.23 | 1,000 |

(Algorithm 5.3) are as small as the APLs of MDD(3)s.

Finally, Table 5.11 compares the area-time complexities [8, 76] of BDDs, heterogeneous MDDs, and MDD($k$)s for MCNC benchmark functions. In this section, we used $AT$, where the area $A$ corresponds to the memory size and the time $T$ corresponds to APL.

Table 5.11 shows that for these benchmark functions, area-time complexities of heterogeneous MDDs are a half of the BDDs, and are much smaller than MDD($k$)s.

Table 5.9: Memory sizes of BDDs, heterogeneous MDDs, and MDD($k$)s for MCNC benchmark functions

| Name | In | Out | BDD | Heterogeneous MDD | | MDD(2) | MDD(3) | MDD(4) | MDD(5) |
|---|---|---|---|---|---|---|---|---|---|
| | | | | MinMem | MinAPL$_M$ | | | | |
| C432 | 36 | 7 | 3189 | 2824 | 3179 | 3075 | 4833 | 5508 | 12441 |
| C499 | 41 | 32 | 77595 | 59739 | 77589 | 62660 | 76248 | 100810 | 174669 |
| C880 | 60 | 26 | 12156 | 11812 | 12154 | 15610 | 21933 | 32742 | 53526 |
| C1908 | 33 | 25 | 16575 | 13493 | 16564 | 16415 | 18720 | 30039 | 36135 |
| C2670 | 233 | 64 | 5319 | 4649 | 5319 | 7600 | 12483 | 19584 | 36102 |
| C3540 | 50 | 22 | 71481 | 65029 | 71480 | 84315 | 127809 | 194650 | 307197 |
| C5315 | 178 | 123 | 5154 | 4582 | 5153 | 6725 | 9981 | 17000 | 30789 |
| C7552 | 207 | 107 | 6633 | 6119 | 6633 | 8615 | 13338 | 21301 | 36828 |
| alu4 | 14 | 8 | 1047 | 855 | 1019 | 1290 | 1431 | 2295 | 3927 |
| apex1 | 45 | 45 | 3735 | 3016 | 3728 | 4575 | 6138 | 8840 | 15411 |
| apex6 | 135 | 99 | 1491 | 1414 | 1490 | 2190 | 3924 | 7123 | 12210 |
| cps | 24 | 102 | 2910 | 2533 | 2906 | 3000 | 4482 | 7786 | 11748 |
| dalu | 75 | 16 | 2064 | 1548 | 2064 | 2610 | 3690 | 6749 | 10263 |
| des | 256 | 245 | 8832 | 7288 | 8831 | 9630 | 16299 | 21488 | 41712 |
| frg2 | 143 | 139 | 2886 | 2671 | 2884 | 4395 | 7191 | 12818 | 21879 |
| i3 | 132 | 6 | 396 | 330 | 396 | 340 | 603 | 646 | 1452 |
| i8 | 133 | 81 | 3825 | 3662 | 3825 | 6035 | 9855 | 17884 | 33924 |
| i10 | 257 | 224 | 61977 | 55766 | 61974 | 85535 | 124065 | 234260 | 380655 |
| k2 | 45 | 45 | 3735 | 3018 | 3728 | 4570 | 6165 | 8823 | 15345 |
| too_large | 38 | 3 | 954 | 857 | 954 | 1090 | 1521 | 2465 | 3696 |
| vda | 17 | 39 | 1431 | 1088 | 1424 | 1375 | 2286 | 2788 | 4290 |
| Average of ratios | | | 1.00 | 0.86 | 1.00 | 1.20 | 1.80 | 2.84 | 4.88 |

## 5.7 Conclusion and Comments

This chapter proposed the representations of binary logic functions using heterogeneous MDDs and the optimization algorithms for heterogeneous MDDs that consider both orderings and partitions of binary variables. Our experimental results show that: 1) Heterogeneous MDDs represent logic functions more compactly than ROBDDs and Free BDDs. Especially, for all 4-variable and 5-variable logic functions, the minimum heterogeneous MDDs require 72% and 67% of the memory sizes for the minimum ROBDDs, on average, respectively. Algorithm 5.2 can find exact minimum heterogeneous MDDs for the functions with up to 12 inputs in a reasonable computation time, and Algorithm 5.3 can reduce heterogeneous MDDs as fast as the sifting algorithm (Algorithm 3.2). 2) In heterogeneous MDDs, APLs can be reduced by a half of corresponding BDDs, on average, without increasing the memory size. And, both memory sizes and APLs can be reduced to 86% and 67% of BDDs, respectively. Algorithm 5.5 con-

Table 5.10: APLs of BDDs, heterogeneous MDDs, and MDD($k$)s for MCNC benchmark functions

| Name | In | Out | BDD | Heterogeneous MDD | | MDD(2) | MDD(3) | MDD(4) | MDD(5) |
|---|---|---|---|---|---|---|---|---|---|
| | | | | MinMem | MinAPL$_M$ | | | | |
| C432 | 36 | 7 | 86.58 | 55.74 | 45.45 | 59.84 | 48.58 | 40.92 | 35.52 |
| C499 | 41 | 32 | 813.64 | 381.14 | 192.52 | 422.28 | 282.65 | 225.88 | 189.23 |
| C880 | 60 | 26 | 135.79 | 125.73 | 99.13 | 115.52 | 104.51 | 95.67 | 86.27 |
| C1908 | 33 | 25 | 254.35 | 145.81 | 92.09 | 168.38 | 118.34 | 103.37 | 86.56 |
| C2670 | 233 | 64 | 214.05 | 167.90 | 133.78 | 189.35 | 179.05 | 154.69 | 152.89 |
| C3540 | 50 | 22 | 209.15 | 141.10 | 91.78 | 160.52 | 132.50 | 109.62 | 93.03 |
| C5315 | 178 | 123 | 462.05 | 373.23 | 304.38 | 400.05 | 378.00 | 342.61 | 339.44 |
| C7552 | 207 | 107 | 484.03 | 424.85 | 314.03 | 418.23 | 380.40 | 336.50 | 309.67 |
| alu4 | 14 | 8 | 40.81 | 24.41 | 19.59 | 31.57 | 21.66 | 19.85 | 15.58 |
| apex1 | 45 | 45 | 180.59 | 87.35 | 67.63 | 154.81 | 124.86 | 94.26 | 95.72 |
| apex6 | 135 | 99 | 291.54 | 260.66 | 231.06 | 268.39 | 271.51 | 263.92 | 247.19 |
| cps | 24 | 102 | 290.25 | 187.90 | 151.81 | 203.95 | 211.43 | 192.44 | 149.73 |
| dalu | 75 | 16 | 102.67 | 39.40 | 28.09 | 70.55 | 51.92 | 51.58 | 41.09 |
| des | 256 | 245 | 1210.00 | 910.63 | 687.50 | 931.14 | 838.53 | 749.61 | 729.31 |
| frg2 | 143 | 139 | 624.69 | 499.27 | 348.60 | 584.58 | 531.48 | 512.31 | 501.66 |
| i3 | 132 | 6 | 26.76 | 17.84 | 12.61 | 18.84 | 15.03 | 13.10 | 11.97 |
| i8 | 133 | 81 | 302.54 | 229.12 | 207.54 | 292.75 | 248.84 | 243.43 | 238.52 |
| i10 | 257 | 224 | 1084.96 | 887.62 | 614.53 | 950.62 | 821.89 | 903.42 | 788.22 |
| k2 | 45 | 45 | 180.52 | 87.32 | 67.61 | 155.30 | 125.76 | 95.23 | 96.78 |
| too_large | 38 | 3 | 13.16 | 8.47 | 6.24 | 9.00 | 7.47 | 6.39 | 5.62 |
| vda | 17 | 39 | 176.34 | 81.72 | 69.54 | 111.91 | 110.19 | 76.03 | 74.33 |
| Average of ratios | | | 1.00 | 0.67 | 0.51 | 0.78 | 0.68 | 0.60 | 0.55 |

sidering both partitions and orderings of binary variables finds heterogeneous MDDs with the minimum APLs for functions with up to 11 variables within a reasonable time. 3) In MDD($k$)s, to reduce the APLs to a half of the BDDs, we need to increase the memory sizes to 488% of the BDDs. Area-time complexities of heterogeneous MDDs are a half of the BDDs, and are much smaller than MDD($k$)s.

Table 5.11: Area-time complexities of BDDs, heterogeneous MDDs, and MDD($k$)s for MCNC benchmark functions

| Name | In | Out | BDD | Heterogeneous MDD | | MDD(2) | MDD(3) | MDD(4) | MDD(5) |
|---|---|---|---|---|---|---|---|---|---|
| | | | | MinMem | MinAPL$_M$ | | | | |
| C432 | 36 | 7 | 276118 | 157399 | 144476 | 184002 | 234806 | 225380 | 441951 |
| C499 | 41 | 32 | 63134444 | 22768960 | 14937095 | 26460143 | 21551378 | 22771246 | 33053379 |
| C880 | 60 | 26 | 1650677 | 1485138 | 1204830 | 1803197 | 2292213 | 3132374 | 4617949 |
| C1908 | 33 | 25 | 4215784 | 1967478 | 1525388 | 2764020 | 2215349 | 3105084 | 3127715 |
| C2670 | 233 | 64 | 1138518 | 780551 | 711573 | 1439035 | 2235054 | 3029509 | 5519726 |
| C3540 | 50 | 22 | 14950022 | 9175809 | 6560680 | 13534607 | 16934893 | 21338299 | 28577026 |
| C5315 | 178 | 123 | 2381424 | 1710121 | 1568471 | 2690305 | 3772817 | 5824415 | 10451014 |
| C7552 | 207 | 107 | 3210593 | 2599668 | 2082937 | 3603038 | 5073795 | 7167682 | 11404484 |
| alu4 | 14 | 8 | 42730 | 20867 | 19967 | 40725 | 30992 | 45567 | 61168 |
| apex1 | 45 | 45 | 674496 | 263461 | 252108 | 708266 | 766385 | 833270 | 1475135 |
| apex6 | 135 | 99 | 434681 | 368575 | 344280 | 587764 | 1065407 | 1879894 | 3018209 |
| cps | 24 | 102 | 844642 | 475959 | 441173 | 611840 | 947620 | 1498369 | 1758999 |
| dalu | 75 | 16 | 211905 | 60990 | 57976 | 184138 | 191568 | 348115 | 421705 |
| des | 256 | 245 | 10686720 | 6636635 | 6071313 | 8966884 | 13667221 | 16107606 | 30421083 |
| frg2 | 143 | 139 | 1802850 | 1333541 | 1005363 | 2569241 | 3821905 | 6566790 | 10975884 |
| i3 | 132 | 6 | 10597 | 5887 | 4994 | 6405 | 9062 | 8465 | 17384 |
| i8 | 133 | 81 | 1157231 | 839045 | 793848 | 1766746 | 2452355 | 4353531 | 8091520 |
| i10 | 257 | 224 | 67242627 | 49499023 | 38084982 | 81311523 | 101968393 | 211634736 | 300040993 |
| k2 | 45 | 45 | 674228 | 263525 | 252046 | 709723 | 775331 | 840241 | 1485118 |
| too_large | 38 | 3 | 12556 | 7258 | 5957 | 9805 | 11362 | 15745 | 20762 |
| vda | 17 | 39 | 252348 | 88910 | 99025 | 153883 | 251900 | 211979 | 318856 |
| Average of ratios | | | 1.00 | 0.59 | 0.51 | 0.96 | 1.27 | 1.85 | 2.96 |

# Chapter 6

# Conclusion

In this thesis, we discussed on the optimization of DDs that minimize the memory size, average path length (APL), or both of them.

In Chapter 3, we proposed APL minimization algorithms for DDs considering only variable orderings. The APL minimization algorithms proposed in Chapter 3 yielded an improvement over an existing algorithm in both APL and runtime. However, the APL minimization algorithms considering only variable orderings often increase the number of nodes, since a variable order that minimizes the APL is often different from the variable order that minimizes the number of nodes.

Next, we used MDDs to reduce the memory sizes and APLs furthermore. MDDs are usually used to represent multi-valued logic functions. However, we used MDDs to represent binary logic functions. When MDDs are used to represent binary logic functions, we can use an additional optimization approach, which is a partition of binary variables. To represent binary logic functions using MDDs, we partition the binary variables into groups, and we treat each group as a multi-valued variable. In Chapter 4, we showed the relations between the values of $k$ and the number of nodes, memory size, path length, and area-time complexity for QRMDD($k$), and derived the optimum values of $k$ for each application. For many benchmark functions, the numbers of nodes and path lengths for QRMDD($k$)s were inversely proportional to the value of $k$. Therefore, the numbers of nodes for QRMDD($k$)s can be reduced with increasing the value of $k$. However, the memory size of each node in QRMDD($k$) increases with $2^k$. By experiments, we showed that the memory sizes for QRMDD($k$)s take their minimum when $k = 2$. To obtain the optimum values of $k$ considering both memory size and path length, we introduced the area-time complexity. By experiments, we showed that when both the memory size and path length are equally important, the optimum value of $k$ is 3 or 4. On the other hand, when the path length is more important than the memory size, the optimum value of $k$ is 4, 5 or 6.

In MDD($k$)s representing binary logic functions, the binary variables are partitioned into the groups with $k$ binary variables. On the other hand, in heterogeneous MDDs, the binary variables can be partitioned into the groups with different numbers of binary variables. Therefore, the memory sizes and APLs of heterogeneous MDDs depend on the partition of binary variables, as well as the order of binary variables. In Chapter 5, we proposed the memory size and APL minimization algorithms for heterogeneous MDDs that consider both orderings and partitions of binary variables. By considering both orderings and partitions of binary variables, heterogeneous MDDs can represent logic functions with smaller memory sizes than FBDDs and smaller APLs than OBDDs, and the APLs of heterogeneous MDDs can be reduced by a half of BDDs without increasing memory size. Heterogeneous MDDs have smaller area-time complexities than MDD($k$)s, since heterogeneous MDDs allow more flexible partition of binary variables than MDD($k$)s.

# Acknowledgements

Many people supported me in various ways during the time I spent at the Kyushu Institute of Technology in Japan. Without the help and encouragement, my aim to graduate might have faded away in an early stage of the Ph.D. program, and this thesis would not exist. I sincerely express gratitude to all of them, especially to the following people.

My adviser, Professor Tsutomu Sasao. I owe all of my accomplishments to Professor Sasao. Professor Sasao provided me all the technical skills that I currently have; how to set up the research goal, how to formulate and attack a problem, how to write a paper, and how to organize a technical presentation. I admired Professor Sasao so much that I simply mimicked his style. It was always fun when I felt that my technical abilities improved, and most pleasant when I realized that once a research has been accomplished, it is often the case that the result can be stated in a simple statement. This has been an useful check to see whether I have fully reached the end of the trail. I will treasure what I learned from Professor Sasao for the rest of my life.

It is my great pleasure to thank Professor Kyoki Imamura, Professor Takaichi Yoshida, and Professor Seiji Kajihara who kindly agreed to be members of both the qualification and the thesis committees.

I am sincerely grateful to Professor Yukihiro Iguchi of Meiji University in Japan who gave me a lot of technical skills and cooperations. He motivated my researches on decision diagrams (DDs), and introduced me to Professor Sasao and Kyushu Institute of Technology when I was his student. Without his support, my researches would not accomplish. I would like to thank Professor Iguchi sincerely for his support and introducing me to Professor Sasao.

I am grateful to Professor Jon T. Butler of Naval Postgraduate School in USA who gave me a great deal of encouragement. Many times he also spent his valuable time for reviewing my papers, even though his name was not included in the list of authors. I thankfully acknowledge his continuous support and encouragement over the years.

Dr. Alan Mishchenko of UC Berkeley in USA encouraged me to set up my researches on average path length (APL) of DDs, and greatly contributed to my paper on APL of DDs. I wish to thank him for his cooperation and encouragement.

I would like to thank Professor Randal E. Bryant, Professor Rolf Drechsler, Professor D. Michael Miller, Professor Shuzo Yajima, Professor Nagisa Ishiura, Professor Fabio Somenzi, Professor Yusuke Matsunaga, Dr. Wolfgang Guenther, and many other people who originally started the research works on DDs. I referred their papers many times and sometimes had helpful discussion with them. Especially, the BDD package of Fabio Somenzi of University of Colorado at Boulder in USA was very helpful during my experiments.

I am indebted to Dr. Wolfgang Guenther of Infineon Technologies in Germany who provided me his data for free binary decision diagrams (FBDDs).

I owe the precious memory and experience to those who stayed at the Kyushu Institute of Technology as visitors or students, as well as those who worked with Professor Sasao or often visited the Professor Sasao's laboratory. They are Professor Jon T. Butler, Professor Marek A. Perkowski, Dr. Alan Mishchenko, Professor D. Michael Miller, Professor Bogdan J. Falkowski, Dr. Marc Riedel, Professor Yukihiro Iguchi, Professor Seiji Kajihara, Mr. Munehiro Matsuura, Dr. Atsumu Iseno, and many other people.

All research works of my thesis were partly supported by the Grand 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.

# References

[1] P. Ashar and S. Malik, "Fast functional simulation using branching programs," *International Conference on Computer-Aided Design (ICCAD'95)*, pp. 408–412, Nov. 1995.

[2] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli, E. M. Sentovich, and K. Suzuki, "Synthesis of software programs for embedded control applications," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, Vol. 18, No. 6, pp. 834–849, June 1999.

[3] B. Becker and R. Drechsler, "Efficient graph based representation of multi-valued functions with an application to genetic algorithms," *24th International Symposium on Multiple Valued Logic*, pp. 40–45, May 1994.

[4] V. Bertacco, S. Minato, P. Verplaetse, L. Benini, and G. De Micheli, "Decision diagrams and pass transistor logic synthesis," *International Workshop on Logic and Synthesis*, s. 3-3, Lake Tahoe, May 1997.

[5] V. Bertacco and M. Damiani, "The disjunctive decomposition of logic functions," *International Conference on Computer-Aided Design (ICCAD'97)*, pp. 78–82, San Jose, Nov. 1997.

[6] K. Brace, R. Rudell, and R. E. Bryant, "Efficient implementation of a BDD package," *Design Automation Conference*, pp. 40–45, June 1990.

[7] R. K. Brayton, "The future of logic synthesis and verification," in S. Hassoun and T. Sasao (eds.) *Logic Synthesis and Verification*, Kluwer Academic Publisher, 2001.

[8] R. P. Brent and H. T. Kung, "The area-time complexity of binary multiplication," *Journal of the ACM*, Vol. 28, No. 3, pp. 521–534, July 1981.

[9] F. Brglez and H. Fujiwara, "Neutral netlist of ten combinational benchmark circuits and a target translator in FORTRAN," *Special session on ATPG and fault simulation, Proc. IEEE Int. Symp. Circuits and Systems*, pp. 663–698, June 1985.

[10] S. Brown, R. Francis, J. Rose, and Z. Vranesic, *Field-Programmable Gate Arrays*, Kluwer Academic Publishers 1992.

[11] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comput.*, Vol. C-35, No. 8, pp. 677–691, Aug. 1986.

[12] J. T. Butler and T. Sasao, "On the average path length in decision diagrams of multiple-valued functions," *33rd International Symposium on Multiple-Valued Logic*, pp. 383–390, Tokyo, Japan, May 2003.

[13] M. L. Dertouzos, *Threshold Logic: A Synthesis Approach*, Mass. Inst. Tech., Cambridge, Res. Monograph 32. Cambridge, Mass.: M. I. Press, 1965.

[14] R. Drechsler, W. Günther, and F. Somenzi, "Using lower bounds during dynamic BDD minimization," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, Vol. 20 No. 1, pp. 51–57, Jan. 2001.

[15] R. Drechsler and M. Thornton, "Fast and efficient equivalence checking based on NAND-BDDs," *IFIP VLSI'01*, pp. 401–405, Montpellier, 2001.

[16] R. Ebendt, W. Guenther, and R. Drechsler, "Combination of lower bounds in exact BDD minimization," *Design, Automation and Test in Europe conference and exhibition (DATE'03)*, pp. 758–763, Munich, Germany, Mar. 2003.

[17] M. Fujita, Y. Matsunaga, and T. Kakuda, "On variable ordering of binary decision diagrams for the application of multi-level logic synthesis," *EDAC*, pp. 50–54, Mar. 1991.

[18] W. Guenther and R. Drechsler, "Minimization of free BDDs," *Asia and South Pacific Design Automation Conference (ASP-DAC'99)*, pp. 323–326, Wanchai, Hong Kong, Jan. 1999.

[19] W. Guenther, "Minimization of free BDDs using evolutionary techniques," *International Workshop on Logic Synthesis 2000 (IWLS-2000)*, pp. 167–172, Loguna Cliffs Marriott, Dana Point, CA, May 2000.

[20] Hafiz Md. Hasan Babu and T. Sasao, "Heuristics to minimize multiple-valued decision diagrams," *IEICE Trans. on fundamentals*, Vol. E83-A, No. 12, pp. 2498–2504, Dec. 2000.

[21] S. L. Hurst, D. M. Miller, and J. C. Muzio, *Spectral techniques in digital logic*, Academic Press., London, 1985.

[22] Y. Iguchi, T. Sasao, M. Matsuura, and A. Iseno "A hardware simulation engine based on decision diagrams," *Asia and South Pacific Design Automation Conference (ASP-DAC'2000)*, pp. 73–76, Yokohama, Japan, Jan. 2000.

[23] Y. Iguchi, T. Sasao, and M. Matsuura, "Implementation of multiple-output functions using PQMDDs," *30th International Symposium on Multiple-Valued Logic*, pp. 199–205, May 2000.

[24] N. Ishiura, H. Sawada, and S. Yajima, "Minimization of binary decision diagrams based on exchanges of variables," *International Conference on Computer-Aided Design (IC-CAD'91)*, pp. 472–475, Nov. 1991.

[25] Y. Jiang and R. K. Brayton, "Software synthesis from synchronous specifications using logic simulation techniques," *Design Automation Conference*, pp. 319–324, New Orleans, LA, U.S.A, June 2002.

[26] T. Kam, T. Villa, R. K. Brayton, and A. L. Sagiovanni-Vincentelli, "Multi-valued decision diagrams: Theory and applications," *Multiple-Valued Logic: An International Journal*, Vol. 4, No. 1-2, pp. 9–62, 1998.

[27] C. Kim, L. Lavagno, and A. S-Vincentelli, "Free MDD-based software optimization techniques for embedded systems," *Design, Automation and Test in Europe (DATE2000)*, Paris, pp. 14–19, March 2000.

[28] Kouloheris and A. El Gamal, "FPGA area versus cell granularity — lookup tables and PLA cells," *Proc. ACM First Int. Workshop on Field Programmable Gate Arrays*, pp. 9–14, Feb. 1992.

[29] H.-T. Liaw and C.-S. Lin. "On the OBDD-representation of general Boolean function," *IEEE Transactions on Computers*, Vol. 4, No. 6, pp. 661–664, June 1992.

[30] C. L. Liu, *Introduction to Combinatorial Mathematics*, McGraw-Hill, Inc., 1968.

[31] Y. Y. Liu, K. H. Wang, T. T. Hwang, and C. L. Liu, "Binary decision diagrams with minimum expected path length," *Design, Automation and Test in Europe (DATE2001)*, pp. 708–712, Mar. 2001.

[32] Y. Matsunaga, "An exact and efficient algorithm for disjunctive decomposition," *the 7th workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI 1998)*, pp. 44–50, Sendai, Japan, Oct. 1998.

[33] M. Matsuura, and T. Sasao, "Representation of incompletely specified switching functions using pseudo-Kroneker decision diagrams," *International Workshop on Applications of the Reed Muller Expansion in Circuit Design (Reed-Muller 2001)*, pp. 27–33, Starkville, U.S.A, Aug. 2001.

[34] P. C. McGeer, K. L. McMillan, A. Saldanha, A. L. Sangiovanni-Vincentelli, and P. Scaglia, "Fast discrete function evaluation using decision diagrams," *International Workshop on Logic and Synthesis*, pp. 6_1–6_9, Lake Thahoe, May 1995.

[35] P. C. McGeer, K. L. McMillan, A. Saldanha, A. L. Sangiovanni-Vincentelli, and P. Scaglia, "Fast discrete function evaluation using decision diagrams," *International Conference on Computer-Aided Design (ICCAD'95)*, pp. 402–407, Nov. 1995.

[36] C. Meinel and T. Theobald, *Algorithms and Data Structures in VLSI Design*, Springer, 1998.

[37] D. M. Miller, "Multiple-valued logic design tools," *23rd International Symposium on Multiple Valued Logic*, pp. 2–11, May 1993.

[38] D. M. Miller and R. Drechsler, "Implementing a multiple-valued decision diagram package," *28th International Symposium on Multiple-Valued Logic*, pp. 52–57, May 1998.

[39] D. M. Miller and R. Drechsler, "Augmented sifting of multiple-valued decision diagrams," *33rd International Symposium on Multiple-Valued Logic*, pp. 375–382, Tokyo, Japan, May 2003.

[40] S. Minato, N. Ishiura, and S. Yajima, "Shared binary decision diagram with attributed edges for efficient Boolean function manipulation," *Design Automation Conference*, pp. 52–57, June 1990.

[41] S. Minato, "Minimum-width method of variable ordering for binary decision diagrams," *IEICE Trans. on fundamentals*, Vol. E75-A, No. 3, pp. 392–399, Mar. 1992.

[42] A. Mishchenko, B. Steinbach, and M. Perkowski, "An algorithm for bi-decomposition of logic functions," *Design Automation Conference*, pp. 103–108, June 2001.

[43] S. Muroga, *Logic Design and Switching Theory*, Wiley-Interscience Publication, 1979.

[44] S. Nagayama, T. Sasao, Y. Iguchi, and M. Matsuura, "Representations of logic functions using QRMDDs," *32nd International Symposium on Multiple-Valued Logic*, pp. 261–267, Boston, Massachusetts, U.S.A, May 2002.

[45] S. Nagayama and T. Sasao, "Code generation for embedded systems using heterogeneous MDDs," *the 12th workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI 2003)*, pp. 258–264, Hiroshima, Japan, April 2003.

[46] S. Nagayama and T. Sasao, "Compact representations of logic functions using heterogeneous MDDs," *33rd International Symposium on Multiple-Valued Logic*, pp. 247–252, Tokyo, Japan, May 2003.

[47] S. Nagayama, A. Mishchenko, T. Sasao, and J. T. Butler, "Minimization of average path length in BDDs by variable reordering," *International Workshop on Logic and Synthesis*, pp. 207–213, Laguna Beach, California, U.S.A, May 2003.

[48] S. Nagayama and T. Sasao, "Compact representations of logic functions using heterogeneous MDDs," *IEICE Trans. on fundamentals*, Vol. E86-A, No. 12, pp. 3168–3175, Dec. 2003.

[49] S. Nagayama and T. Sasao, "Minimization of memory size for heterogeneous MDDs," *Asia and South Pacific Design Automation Conference (ASP-DAC'2004)*, pp. 872–875, Yokohama, Japan, Jan. 2004.

[50] S. Nagayama and T. Sasao, "On the minimization of average path lengths for heterogeneous MDDs," *34th International Symposium on Multiple-Valued Logic*, pp. 216–222, Toronto, Canada, May 2004.

[51] H. Ochi, N. Ishiura and S. Yajima, "Breadth-first manipulation of SBDD of Boolean function for vector processing," *Design Automation Conference*, pp. 413–416, 1991.

[52] H. Ochi, K. Yasuoka and S. Yajima, "Breadth-first manipulation of very large binary-decision diagrams," *International Conference on Computer-Aided Design (ICCAD'93)*, pp. 48–55, Nov. 1993.

[53] S. Panda, F. Somenzi, and B. F. Plessier, "Symmetry detection and dynamic variable ordering of decision diagrams," *International Conference on Computer-Aided Design (ICCAD'94)*, pp. 628–631, San Jose, CA, Nov, 1994.

[54] D. V. Popel and R. Drechsler, "Efficient minimization of multiple-valued decision diagrams for incompletely specified functions," *33rd International Symposium on Multiple-Valued Logic*, pp. 241–246, Tokyo, Japan, May 2003.

[55] A. Prakash, R. Kotla, T. Mandal, and A. Aziz, "A high-performance architecture and BDD-based synthesis methodology for packet classification," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, Vol. 22, No. 6, pp. 698–709, June 2003.

[56] S. Reda and A. Salem, "Combinational equivalence checking using boolean satisfiability and binary decision diagrams," *Design, Automation and Test in Europe conference (DATE'01)*, pp. 122–126, 2001.

[57] S. Reda, A. Orailoglu, and R. Drechsler, "On the relation between SAT and BDDs for equivalence checking," *International Symposium on Quality Electronics Design*, pp. 394–399, Mar. 2002.

[58] J. Rose, R. J. Francis, D. Lewis, and P. Chow, "Architecture of field-programmable gate arrays: the effect of logic block functionality on area efficiency," *IEEE Journal of Solid-State Circuits*, Vol. 25, pp. 1217–1225, Oct. 1990.

[59] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," *International Conference on Computer-Aided Design (ICCAD'93)*, pp. 42–47, Nov. 1993.

[60] T. Sasao, "FPGA design by generalized functional decomposition," (Sasao ed.) *Logic Synthesis and Optimization*, Kluwer Academic Publishers, 1993.

[61] T. Sasao and M. Fujita (eds.), *Representations of Discrete Functions*, Kluwer Academic Publishers 1996.

[62] T. Sasao and J. T. Butler, "A method to represent multiple-output switching functions by using multi-valued decision diagrams," *26th International Symposium on Multiple-Valued Logic*, pp. 248–254, Santiago de Compostela, Spain, May 1996.

[63] T. Sasao, *Switching Theory for Logic Synthesis*, Kluwer Academic Publishers, 1999.

[64] T. Sasao, "Compact SOP representations for multiple-output functions: An encoding method using multiple-valued logic," *31th International Symposium on Multiple-Valued Logic*, pp. 207–211, Warsaw, Poland, May 2001.

[65] T. Sasao, M. Matsuura, and Y. Iguchi, "Cascade realization of multiple-output function and its application to reconfigurable hardware," *International Workshop on Logic and Synthesis*, pp. 225–230, Lake Tahoe, June 2001.

[66] T. Sasao, M. Matsuura, Y. Iguchi, and S. Nagayama, "Compact BDD representations for multiple-output functions and their application," *IFIP VLSI-SOC'01*, pp. 406–411, Montpellier, France, Dec. 2001.

[67] T. Sasao, Y.Iguchi, and M. Matsuura, "Comparison of decision diagrams for multiple-output logic functions," *International Workshop on Logic and Synthesis*, pp. 379–384, New Orleans, Louisiana, June 2002.

[68] T. Sasao, J. T. Butler, and M. Matsuura, "Average path length as a paradigm for the fast evaluation of functions represented by binary decision diagrams," *International Symposium on New Paradigm VLSI Computing*, pp. 31–36, Sendai, Japan, Dec. 2002.

[69] R. S. Shelar and S. S. Sapatnekar, "Efficient layout synthesis algorithm for pass transistor logic circuits," *Asia and South Pacific Design Automation Conference (ASP-DAC'2002)*, pp. 87–92, Bangalore, India, Jan. 2002.

[70] R. S. Shelar and S. S. Sapatnekar, "Efficient layout synthesis algorithm for pass transistor logic circuits," *International Workshop on Logic and Synthesis*, pp. 209-214, New Orleans, Louisiana, June 2002.

[71] T. R. Shiple, R. Hojati, A. L. Sangiovannni-Vincentelli, and R. K. Brayton, "Heuristic minimization of BDDs using don't cares," *Design Automation Conference*, pp. 225-231, June 1994.

[72] F. Somenzi, "CUDD: CU Decision Diagram Package Release 2.3.1," University of Colorado at Boulder, 2001.

[73] A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton, "Algorithms for discrete function manipulation," *International Conference on Computer-Aided Design (ICCAD'90)*, pp. 92–95, Nov. 1990.

[74] K. Takagi, H. Hatakeda, S. Kimura, and K. Watanabe, "Exact minimization of Free BDDs and its application to pass-transistor logic optimization," *IEICE Trans. on fundamentals*, Vol. E82-A, No. 11, pp. 2407–2413, Nov. 1999.

[75] A. Thayse, M. Davio, and J.-P. Deschamps, "Optimization of multiple-valued decision algorithms," *8th International Symposium on Multiple-Valued Logic*, pp. 171–177, Rosemont, IL., May 1978.

[76] C. D. Thompson, "Area-Time complexity for VLSI," *11th Annual ACM Symposium on Theory of Computing*, pp. 81–88, May 1979.

[77] M. Thornton, D. M. Miller, and R. Drechsler, "Transformations amongst the Walsh, Haar, arithmetic and Reed-Muller spectral domains," *Intl. Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, pp. 215–225, August 2001.

[78] I. Wegener, *Branching Programs and Binary Decision Diagrams: Theory and Applications*, SIAM, 2000.

[79] A. Yan, R. Cheng, and S. J. E. Wilton, "On the sensitivity of FPGA architectural conclusions to the experimental assumptions, tools, and techniques," *the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 147–156, Monterey, CA, Feb. 2002.

[80] C. Yang and M. Ciesielski, "BDS: A BDD-based logic optimization system," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, Vol. 21, No. 7, pp. 866–876, July 2002.

[81] S. Yang, *Logic synthesis and optimization benchmark user guide version 3.0*, MCNC, Jan. 1991.

# Appendix

## A. Proofs of Theorems in Chapter 3

**Lemma 3.1** [67] 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** 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 $v$. ∎

**Theorem 3.1** [67] 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** We prove only the first statement; the proof for the second statement is similar. From Definition 3.2, we have

$$ETP(e) = \sum_{p \in SP(e)} PP(p), \tag{3.1}$$

where $SP(e)$ is a set of paths including the edge $e$. We prove the following

$$APL = \sum_{i=1}^{N_e} ETP(e_i), \tag{3.2}$$

where $N_e$ denotes the number of edges in a DD. From formula (3.1), formula (3.2) can be transformed as follows:

$$APL \;=\; \sum_{i=1}^{N_e} ETP(e_i)$$

$$= \sum_{i=1}^{N_e} \sum_{p \in SP(e_i)} PP(p) \tag{3.3}$$

From Definition 2.17, we have

$$\begin{aligned} APL &= \sum_{i=1}^{N} PP(p_i) \times l_i \\ &= \sum_{i=1}^{N} \sum_{j=1}^{l_i} PP(p_i) \end{aligned} \tag{3.4}$$

Although formula (3.3) and formula (3.4) use different computational approaches, they obviously compute the same value. ∎

**Lemma 3.2** Suppose an SDD represents a multiple-output logic function $F$. Then,

$$ETP(Cut(i)) = m_U,$$

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

**Proof** An SDD for $F = (f_0, f_1, \ldots, f_{m-1})$ is traversed from a root node to a terminal node $m$ 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 $ETP(Cut(i)) = m_U$. ∎

**Lemma 3.3** Let

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

Then, for every permutation of $X_{upper}$,

$$ETP(Cut'(i)) = c_i,$$

where $c_i \leq m_U$.

**Proof** From Lemma 3.1, 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 3.2. ∎

**Theorem 3.2** Consider an SDD 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 $X_{lower}$ and any permutation of $X_{upper}$,

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

**Proof** 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. ∎

**Theorem 3.3** Consider an SDD 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 $X_{upper}$ is fixed,

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

**Proof** Let $L$ be the sum of the node traversing probabilities of the non-terminal nodes below or in level $i+1$. From Theorem 3.1, we have

$$APL = U + L.$$

Then, from Theorem 3.2, for any permutation of $X_{lower}$,

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

∎

**Theorem 3.4** 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** 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 3.1, $L$ is obtained by the sum of $ETP(Cut'(i+1))$ and $ETP(E_{lower})$, where

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

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

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

$$ETP(E_{lower}) = \sum_{e \in E_{lower}} ETP(e).$$

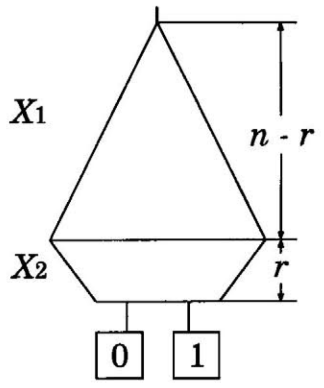


Figure B.1: Partition of BDD.

By Lemma 3.3, $ETP(Cut'(i+1))$ 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. ∎

## B. Proofs of Theorems in Chapter 4

**Definition B.1** Suppose that a QRBDD for an $n$-variable logic function is partitioned into two parts as shown in Fig B.1. It is partitioned into **the upper part** which has the variables $X_1 = (x_1, x_2, \ldots, x_{n-r})$, and **the lower part** which has the variables $X_2 = (x_{n-r+1}, \ldots, x_n)$. In this case, the BDD represents the logic function as follows:

$$f(X_1, X_2) = \bigvee_{\vec{a}_i \in B^{n-r}} X_1^{\vec{a}_i} f(\vec{a}_i, X_2),$$

where

$$X_1^{\vec{a}_i} = \begin{cases} 1 & (X_1 = \vec{a}_i) \\ 0 & (otherwise). \end{cases}$$

The upper part realizes $X_1^{\vec{a}_i}$, and the lower part realizes $f(\vec{a}_i, X_2)$.

**Theorem 4.1** An arbitrary $n$-variable logic function can be represented by a QRBDD with at most

$$2^{n-r} - 1 + \sum_{i=1}^{r} 2^{2^i}$$

non-terminal nodes, where $r$ is the largest integer that satisfies relation $n - r \geq 2^r$ [29].

**Proof** When the upper part of the QRBDD (see Fig. B.1) has $2^{n-r}-1$ nodes (i.e., a complete binary tree), it is the maximum. Because $f(\vec{a}_i, X_2)$ is an $r$-variable logic function, the number of different $f(\vec{a}_i, X_2)$ is $2^{2^r}$. When $2^{2^i}$ logic functions are realized for each level $i$ $(i = 1, 2, \ldots, r)$ from the terminal node to the $r$, the lower part is the maximum. Therefore, the number of non-terminal nodes in a QRBDD is at most

$$2^{n-r} - 1 + \sum_{i=1}^{r} 2^{2^i}.$$

This upper bound becomes the tightest when $r$ is the maximum integer satisfying $n - r \geq 2^r$ [29].

∎

**Theorem 4.2** An arbitrary $n$-variable logic function can be represented by a QRMDD($k$) with at most

$$\frac{2^{sk}-1}{2^k-1} + \sum_{i=1}^{u-s} 2^{2^{(ki-t)}}$$

non-terminal nodes, where $u$ is the number of super variables, $t$ is the number of dummy variables, and $s$ is the smallest integer that satisfies relation

$$s \geq \frac{n-r}{k}.$$

**Proof** Since each node in a QRMDD($k$) has $2^k$ outgoing edges, the upper part of QRMDD($k$) is maximum when it is equivalent to a complete $2^k$-valued tree. Therefore, the upper part has at most

$$\frac{2^{sk}-1}{2^k-1}$$

nodes, where $s$ denotes the number of super variables in upper part. The lower part is maximum when all $i$-variable functions are realized for each level $i$ $(i = 1, 2, \ldots, u-s)$, which have $2^k$-valued inputs and binary outputs. Note that $X_u$ may include dummy variables. Therefore, the number of non-terminal nodes in a QRMDD($k$) is at most

$$\frac{2^{sk}-1}{2^k-1} + \sum_{i=1}^{u-s} 2^{2^{(ki-t)}}.$$

∎

## C. Proofs of Theorems in Chapter 5

**Lemma 5.1** Let $N_{fix}(n)$ be the number of different fixed-order partitions of $X$. Then,

$$N_{fix}(n) = 2^{n-1}.$$

**Proof** $X = (x_1, x_2, \ldots, x_n)$ has $n-1$ **partition points**, the positions that can be partitioned. At each partition point, we can choose whether to partition at this point or not. Thus, $2^{n-1}$ different partitions exist. ∎

The following lemma is used for proof of Theorem 5.1.

**Lemma C.1** [30] The number of different distributions of $n$ objects into $r$ distinct cells is calculated by the following formula, where each cell has at least one object and order of objects within a cell is not important.

$$a(n,r) = \sum_{i=0}^{r} {}_rC_i(r-i)^n(-1)^i$$

**Example C.1** The number of different distributions of 5 objects into 2 distinct cells is

$$a(5,2) = 2^5 - 2 = 30.$$

(End of Example)

**Theorem 5.1** Let $N_{non\text{-}fix}(n)$ be the number of different non-fixed-order partitions of $X = (x_1, x_2, \ldots, x_n)$. Then,

$$N_{non\text{-}fix}(n) = \sum_{r=1}^{n} \sum_{i=0}^{r} {}_rC_i(r-i)^n(-1)^i.$$

**Proof** From Lemma C.1, the number of different non-fixed-order partitions of $n$ binary variables into $r$ super variables is calculated by the following:

$$a(n,r) = \sum_{i=0}^{r} {}_rC_i(r-i)^n(-1)^i.$$

Since $N_{non\text{-}fix}(n)$ is the summation of $a(n,r)$ for $r = 1, 2, \ldots, n$, we have the theorem. ∎

**Theorem 5.2** In a fixed-order minimum heterogeneous MDD, the following relation holds for any super variable $X_i = (x_j, x_{j+1}, \ldots, x_{j+k_i-1})$:

$$(2^{k_i}+1)width(\text{heterogeneous MDD}, i) \leq 3 \times \sum_{t=0}^{k_i-1} width(\text{BDD}, j+t),$$

where the heterogeneous MDD and the BDD represent the same logic function, the variable order is fixed.

**Proof** In a fixed-order minimum heterogeneous MDD, partition $X_i$ into $(X_{i_0}, X_{i_1}, \ldots, X_{i_{k_i-1}})$, where $X_{i_0} = (x_j)$, $X_{i_1} = (x_{j+1})$,..., and $X_{i_{k_i-1}} = (x_{j+k_i-1})$. The memory size of the heterogeneous MDD with respect to $X_{i_t}$ $(t = 0, 1, \ldots, k_i - 1)$ becomes

$$\sum_{t=0}^{k_i-1} (2+1) width(\text{heterogeneous MDD}, i_t) = 3 \times \sum_{t=0}^{k_i-1} width(\text{BDD}, j+t).$$

Note that each node in the BDD requires three words (see the formula (5.1)). If the theorem does not hold, then the original heterogeneous MDD was not fixed-order minimum, which is contradiction. ∎

**Theorem 5.3** Consider a BDD and a heterogeneous MDD for an $n$-variable logic function that is not a constant function. When an order of binary variables is fixed, for the number of nodes in the BDD and the memory size of heterogeneous MDD obtained by considering only the fixed-order partitions, the following relation holds:

$$Mem(\text{heterogeneous MDD}) \geq nodes(\text{BDD}) + 2.$$

**Proof** Consider a partition of $X$: $X = (X_1, X_2, \ldots, X_u)$. For arbitrary super variable $X_i = (x_j, x_{j+1}, \ldots, x_{j+k_i-1})$, the following relation holds:

$$width(\text{heterogeneous MDD}, i) \geq width(\text{BDD}, j).$$

From this, we have:

$$(2^{k_i} + 1) width(\text{heterogeneous MDD}, i) \geq (2^{k_i} + 1) width(\text{BDD}, j).$$

Also, in a BDD, the following relation holds:

$$(2^{k_i} - 1) width(\text{BDD}, j) \geq \sum_{t=0}^{k_i-1} width(\text{BDD}, j+t).$$

Then, we have:

$$(2^{k_i} + 1) width(\text{heterogeneous MDD}, i) \geq \sum_{t=0}^{k_i-1} width(\text{BDD}, j+t) + 2 width(\text{BDD}, j),$$

and

$$\sum_{i=1}^{u} (2^{k_i} + 1) width(\text{heterogeneous MDD}, i) \geq \sum_{i=1}^{u} \sum_{t=0}^{k_i-1} width(\text{BDD}, j+t) + 2 \sum_{i=1}^{u} width(\text{BDD}, j).$$

Since $width(\text{BDD}, j) \geq 1$, we have:

$$\sum_{i=1}^{u} (2^{k_i} + 1) width(\text{heterogeneous MDD}, i) \geq \sum_{i=1}^{n} width(\text{BDD}, i) + 2u = nodes(\text{BDD} : f) + 2u.$$

Since $u \geq 1$, we have the theorem. ∎

The following lemma is used for proof of Theorem 5.4.

**Lemma C.2** Suppose a BDD for an $n$-variable logic function is partitioned into two parts as shown in Fig. B.1, and let $X_{upper} = (x_1, x_2, \ldots, x_{n-r})$ and $X_{lower} = (x_{n-r+1}, \ldots, x_n)$. When the variable order $(x_1, x_2, \ldots, x_n)$ is fixed, and the widths of the BDD for the upper part $X_{upper}$ are given by

$$width(\text{BDD}, j) = 2^{j-1} \quad (j = 1, 2, \ldots, n-r),$$

the partition of $X_{upper}$ that produces the fixed-order minimum heterogeneous MDD is a trivial partition into single group (i.e., $X_{upper} = X_1$ and $|X_1| = n - r$), and the memory size of the fixed-order minimum heterogeneous MDD for the upper part is given by $2^{n-r} + 1$.

**Proof** Consider a partition of $X_{upper} = (X_1, X_2, \ldots, X_s)$, where $s \geq 1$. The memory size of a heterogeneous MDD obtained by this partition is

$$A = \sum_{i=1}^{s} (2^{k_i} + 1) width(\text{heterogeneous MDD}, i).$$

When $width(\text{BDD}, j) = 2^{j-1}$ $(j = 1, 2, \ldots, n-r)$, the BDD forms a **complete binary decision tree**. Therefore, we have the following:

$$A = (2^{k_1} + 1) \times 1 + (2^{k_2} + 1) \times 2^{k_1} + (2^{k_3} + 1) \times 2^{k_1 + k_2} + \ldots + (2^{k_s} + 1) \times 2^{k_1 + k_2 + \ldots + k_{s-1}}.$$

And, we have:

$$A = 2^{k_1 + k_2 + \ldots + k_s} + 2^{k_1 + k_2 + \ldots + k_{s-1}} + B,$$

where

$$B = \sum_{i=1}^{s-1} (2^{k_i} + 1) width(\text{heterogeneous MDD}, i).$$

Since $\sum_{i=1}^{s} k_i = n - r$, we have

$$A = 2^{n-r} + 2^{k_1 + k_2 + \ldots + k_{s-1}} + B.$$

From the relation $2^{k_1 + k_2 + \ldots + k_{s-1}} + B \geq 1$, $A$ takes its minimum when $s = 1$. ∎

**Theorem 5.4** An arbitrary $n$-variable logic function can be represented by a heterogeneous MDD with at most the following memory size:

$$2^{n-r} + 3 \cdot 2^{2^r} - 5,$$

where $r$ is the largest integer satisfying the relation

$$n - r \geq 2^r + \log_2 3.$$

**Proof** An arbitrary $n$-variable logic function can be represented by an ROBDD with at most

$$2^{n-r} - 1 + 2^{2^r} - 2$$

non-terminal nodes [29], where the numbers of nodes in the upper part and the lower part are $2^{n-r} - 1$ and $2^{2^r} - 2$, respectively. From Lemma C.2, the memory size of the fixed-order minimum heterogeneous MDD for the upper part is

$$2^{n-r} + 1.$$

Also, from Theorem 5.2, the memory size of the fixed-order minimum heterogeneous MDD for the lower part is at most

$$3 \times (2^{2^r} - 2).$$

Therefore, the memory size of this heterogeneous MDD is

$$(2^{n-r} + 1) + \{3 \times (2^{2^r} - 2)\} = 2^{n-r} + 3 \cdot 2^{2^r} - 5.$$

This formula has its minimum value when $r$ is the largest integer that satisfies the relation

$$n - r \geq 2^r + \log_2 3.$$

That is, the memory size of the heterogeneous MDD is minimum when $r$ satisfies this condition.
∎