On Efficient Ordered Binary Decision Diagram Minimization Heuristics Based on Two-Level Logic

By

Chun Gu

Supervised By

Prof. Yu-Liang Wu

Submitted to the Division of Computer Science and Engineering

For the Degree of Master of Philosophy

At

The Chinese University of Hong Kong

Shatin, N. T. Hong Kong

March, 1999

香港中文大學
圖書館系統
14 JUL 1999
UNIVERSITY
LIBRARY SYSTEM
THE CHINESE UNIVERSITY OF HONG KONG

VITA

Mr. Gu Chun graduated from Wuhan University, one of the 36 key Universities in China, with a BA degree in Software of Computer Science in 1996. From January, 1997 to now, he is a graduate student in the Department of Computer Science at the Chinese University of Hong Kong. His research interest is the CAD for VLSI.

Publications:

Chun Gu, Yu-Liang Wu, Hongbing Fan, "An Improved Pattern Processing Based Heuristic of OBDD Variable Ordering", Third International Symposium on Operations Research and Its Applications.

Chun Gu, Yu-Liang Wu, Ling-Kit Mak, Po-Kei Cheuk, "Performance-Driven Post-Placement Synthesis for 2-D Regular Segmented FPGAs", The 3rd International Conference on ASIC.

基于两层逻辑的二叉排序决定图的有效最小化

顾淳

为香港中文大学哲学硕士学位提交

摘要

二叉排序决定图（OBDD) 是布尔网络的一种有效表示。它在逻辑综合，测试和验证这些领域中都得到了广泛的应用。尽管OBDD有一些非常好的性质，它的复杂性严重依赖于其变量顺序。而且我们知道变量排序问题是一个co-NP 完全问题。大多数现有的OBDD排序的启发式算法都是基于对多层布尔网络的不同游历方法。这些方法不能被直接应用于两层逻辑和未完全指定的逻辑表达式。这里我们将介绍一些新的基于对逻辑子表达式共享的启发式算法。接着我们将讨论一种有着最优共享的逻辑表达式并提出一种对这种逻辑表达式能得到最优结果的算法。通过对大量的实验样本的测试结果表明：我们的结果同有着广泛应用的，在UC Berkeley的SIS 程序包中实现的，基于布尔网络的扇入启发式算法相比获得了超过30%的优化。

On Efficient Ordered Binary Decision Diagram Minimization Heuristics Based on Two-Level Logic

Submitted by

Gu Chun

For the degree of Master of Philosophy

At the Chinese University of Hong Kong

Abstract

Ordered Binary Decision Diagrams(OBDD) are efficient representation of Boolean networks and are widely applied in the areas of logic synthesis, test and verification. Though the OBDDs have some very good features, the complexity depends heavily on the variable order. However the variable ordering problem is known to be a co-NP complete problem. Most of the existing OBDD ordering heuristics are based on various traversal methods on multi-level Boolean network. Such methods can not be applied directly to functions described in two-level form or to incompletely specified functions. Here we will introduce several new heuristics based on sharing of the logic sub-functions. Then we will discuss the functions that have an optimal sharing and suggest that one of the algorithms can get the optimal solution of this kind of functions. The experiment results show that our results achieve an average reduction of over 30% compared with the widely used network based fan-in heuristic implemented in the SIS package of UC Berkeley on a large number of benchmarks.

Acknowledgments

First of all, I would like to give my deepest gratitude to my supervisor, Prof. Yu-Liang Wu, for his tremendous support, encouragement and guidance throughout my graduate study. He is so enlightening, critical and always ready to help me in all aspects of the study. He gives me so much instructions that not only benefits my study but also my daily life. He is so warm and patient to give his hands when I met problems no matter it is from the study or the life. I am also indebted to Prof. Hongbing Fan for his kindly help. Throughout the eight months that I get along with Prof. Fan, he teaches me how to do the research and his enlightenment and research method has greatly influenced me. Without his help, it is impossible for me to grasp so much knowledge and fulfill the program smoothly. It is my honor to have Prof. C. K. Wong and Prof. K. H. Wong to serve as my defense committee member and I highly appreciate their valuable knowledge and comments on my research work.

I would like to thank my classmates who have helped me throughout the two years study. In the course of study, my classmates helped me to grasp the UNIX system, helped me with the C programming and all other things. Let me say thanks to Mr. Zhang Xuejie, Mr. Pan Jiaofeng, Mr. Li Xuequn, Mr. Guo Ping, Ms. Yao Jian, Ms. Li Haiying, Ms. Li Yuanyuan.

Special thanks to Dr. Shao Bing who helps me as if he is my older brother and it is him who make me accustomed to the life in Hong Kong and CUHK.

I should also give my thank to Dr. Coudert Olivier who has helped me to do a lot of experiments to test our ideas.

And I should say thanks to my partner in my TA work, Roy Ngan and Steve Chong. They help me so much so that I can concentrate on the research work.

Without the education from the early stage of my life, I can not get to my

current position. I will never forget that my family use up all their savings to support me to study abroad. I am forever indebted to my family for their love and support.

Contents

Chapter 1

Introduction

One of the major concerns in logic manipulation is to find an efficient representation of Boolean functions. Among the various representations of Boolean functions, Binary Decision Diagrams(BDDs) [1], and with a constraint of fixed variable ordering, Ordered Binary Decision Diagrams(OBDDs), are the most conspicuous ones and have drawn people's attention in the last decade. Since 1986, the year when R. Bryant first proposed the OBDD representation of Boolean functions, discovered the basic properties of the OBDD, and defined the basic operations on the OBDD, the OBDDs have found widely applications in all areas of the CAD like logic synthesis, formal design verification, and test pattern generation. And there are also many other variations of the OBDD which are especially suitable for a certain purpose. However the normal OBDD never loses its importance and there are many researchers working on this subject.

The OBDD representation has the advantages that:

- most commonly encountered functions have a reasonable representation;
- the time complexity of any single operation on the OBDD is bounded by the product of the graph sizes for the functions being operated on;
- the representation of the OBDD is a canonical form if the variable order is given.

One of the major problems related to the OBDD representation of Boolean functions is the OBDD size minimization problem. That is to find a variable order of a given Boolean function such that the OBDD size is minimum. This is because that the complexity of the operations to generate and manipulate the OBDDs is polynomial with respect to the size of the OBDD. However, for a certain Boolean function, the OBDD size of a Boolean function is determined by the variable order of the OBDD. And the size of an OBDD can be very sensitive to the variable order. For example, for an n bit adder the BDD sizes may range from $O(n)$ to $O(2^{n/2})$ depending on the chosen order. The OBDD sizes of some Boolean functions grow exponentially regardless of the variable order (e.g. integer multiplier [6]). Some classes of Boolean functions have linear OBDD sizes in some variable orders, but the size will be exponential in some other orders. For example, the OBDD size of function $f(x_1, \ldots, x_{2n}) = (x_1+x_2)(x_3+x_4)\ldots(x_{2n-1}+x_{2n})$ is $2n$ in variable order $x_1, x_2, \ldots, x_{2n}$, but its OBDD size in variable order $x_1, x_3, \ldots, x_{2n-1}, x_2, x_4, \ldots, x_{2n}$ is greater than 2^n. To find an optimal OBDD variable order that minimizes the OBDD size is a co-NP-hard problem. Otherwise if we can find the optimal variable order for any OBDD in polynomial time, we will have a tractable algorithm to give an optimal representation for a Boolean function and we know that it is impossible because the optimal Boolean function representation problem is intractable[14]. Therefore what we can do is to find a heuristic ordering algorithm.

Many ordering heuristics have already been developed. Most of the existing OBDD variable ordering heuristics are based on various traversal methods on multi-level Boolean network. [7], [12], [13], [24], [16], [18]. Those methods can not be applied directly to functions described in the two-level form. Another drawbacks of these algorithms are that some of them can not be applied to the incomplete specified functions. And seldom have these algorithms explored the logic relationship between the OBDD and logic functions and have not utilized

the benefit brought out by the OBDD.

It is noticed that the key factor which influences the size of OBDDs is the sharing of nodes, the more sharing the smaller the OBDD size may be. An optimum OBDD must fully take the advantage of the sharing of nodes. We consider two sharing strategies. One is the maximal cube sharing, which gives a sound explanation for the effectiveness to our previous dynamic shortest cube first algorithm. Another is sub-function sharing. Using this new strategy, we propose a revised dynamic shortest cube first algorithm.

The comparison of experimental results is commonly used in evaluating the effectiveness of ordering heuristics. The experimental results show a great improvement on average compared with the widely used FIH algorithm.

In addition to the experimental comparison view on ordering heuristics, we propose a theoretical way to evaluate an ordering heuristic. We say an ordering heuristic H is optimal for a class of Boolean functions C, if it can find an optimal variable order for each Boolean function in C, where an optimal variable ordering is an ordering with minimum size of OBDD of all variable orderings. An ordering heuristic is considered to be theoretically better than another one if it is optimal for a larger class of Boolean functions with the same computation complexity. The difficulty of finding an optimal variable order is due to the hardness in recognizing optimal OBDDs. So that we consider classes of Boolean functions whose optimal OBDDs can be easily recognized. The first class of Boolean functions considered is the thin OBDDs, in which the number of non-terminal nodes is equal to the number of supporting variables. The structure and construction of thin OBDDs are investigated. It is proved that the class of the thin Boolean functions always has an essential prime cover; and an algorithm for essential points retrieval is given. The second class consists of Boolean functions that can be expressed by a factored form with each variable appearing exactly once. We refer to such Boolean functions as thin factored functions. It is proved that the thin factored functions

belong to the class of thin Boolean functions and they can be recognized in polynomial time with respect to the number of terms in prime cube cover. We use these two classes of Boolean functions as the testing classes. The DSCF is proved to be optimal for a subclass of thin Boolean functions which can be expressed by disjoint form, that is the sum of product (SOP) form with no common variables for each different cubes. The revised DSCF is proved to be optimal for thin factored functions.

To improve our work, we have also tried some other methods to gain improvements. On the base of sharing and pattern recognition, I have proposed another algorithm based on pattern recognition, match, and construction. This algorithm has also gain a fairly good result compared with DSCF algorithm and is greatly improved compared with the FIH algorithm.

The rest of this paper is organized as follows. In chapter 2, we describe some of the main concepts and definitions used in this paper. Chapter 3 discusses some previous work on OBDD. Chapter 4 gives the idea on the two level logic functions and OBDD. Chapter 5 presents the dynamic shortest cube first algorithm. Chapter 6 discusses the properties of thin Boolean functions and thin factored functions and gives the revised dynamic shortest cube first algorithm. The theoretical analysis of these two algorithms is also given in this chapter. In Chapter 7 we will discuss another variable ordering algorithm. And finally, the conclusion is given in Chapter 8.

Chapter 2

Definitions

In Chapter 1, we gave a motivation of the OBDD minimization problem in logic synthesis. In this chapter we describe the basic terminologies related to our work.

We first review some logic synthesis terminologies which we frequently use. Let $f = f(x_1, x_2, ..., x_n)$ be a Boolean function on variable set $S = \{x_1, x_2, .., x_n\}$. The literal set of S is the set consisting of all variables and their complements, namely, $x_1, x_1', x_2, x_2', ..., x_n, x_n'$. The restriction $f|_{x_i=b}$ of f with respect to $x_i = b$ is defined as $f|_{x_i=b}(x_1, ..., x_{i-1}, x_{i+1}, \ldots, x_n) =$
$f(x_1, \ldots, x_{i-1}, b, x_{i+1}, \ldots, x_n)$, where $b = 0$ or 1. $f|_{x_i=1}$ and $f|_{x_i=0}$ is simply denoted by $f|_{x_i}$ and $f|_{x_i'}$, where we use x_i' to denote the complement $\overline{x}_i$. More restrictions can be added to a Boolean function f, $f|_{x_{i_1}=b_1,...,x_{i_k}=b_k}$. The restriction of f is also called a cofactor (or sub-function) of f. A variable x_i is a dependent variable of f if $f|_{x_i} \neq f|_{x_i'}$. The dependent set of f, denoted by $I(f)$, is the set of all dependent variables of f.

A *minterm* of a function is a vertex of the Boolean function.

A *product term* (or *product*) is a formula of one of the following forms:

- 1;
- a non-constant literal;
- a conjunction of non-constant literals where no letter appears more than once.

A *sum term* (or *sum*) is a formula of one of the following forms:

- 0;
- a non-constant literal;
- a disjunction of non-constant literals where no letter appears more than once.

For example, x_1x_2' is a product term, $x_1 + x_2$ is a sum term and x_1' is both. On the other hand, x_1x_1' and x_1x_1 are neither product terms nor sum terms.

A *sum of product formula* is one of the following:

- 0;
- a product term;
- a disjoint of product terms.

Likewise, a *product of sums formula* is one of the following:

- 1;
- a sum term;
- a conjunction of sum terms.

For instance, $f = x_1x_2' + x_2'x_3 + x_1x_3'$ is a sum of product formula for $f[15]$. An *implicant* of a function f is a product term p that is included in the function f. For instance, xy' is a implicant of $xy' + yz$. A *prime implicant* of f is an implicant of f that is not included in any other implicant of f. If a prime implicant includes at least one minterm that is not covered by any other prime implicant, then that prime implicant is called *essential prime implicant.*

A *factored form logic function* can be defined as following:

- 0 and 1 are in factored form.

- Any variable v and its complement v' are in factored form.

- For any factored form logic function f_i, f_j, $(f_i + f_j)$ and $(f_i \cdot f_j)$ are in factored form.

- All the factored from logic function can be generated by applying the former 3 rules recursively.

A BDD of Boolean function f is a rooted directed diagram with node set of non-terminal nodes and two terminal nodes. The two terminal vertices in the diagram are 0 and 1 respectively. Each non-terminal node has an primary input variable of the Boolean function and two outgoing edges, called 1-edge and 0-edge. These two edges point to two nodes: a high son $high(v)$ and a low son $low(v)$ respectively. The two terminal nodes, denoted by 1 and 0, have their attributes as the Boolean value of 1 and 0, respectively. Each non-terminal node is in a path from the root to the terminals, and the input variables of nodes on this path are in the ordering of S. Each node v represents a Boolean function, denoted by f_v, which can be decomposed as $f_v = x_i f_{high(v)} + x_i' f_{low(v)}$, where x_i is the input variable of node v. The root of the graph represents the entire function f. The term OBDD size is referred to the number of nodes of an OBDD.

An Ordered BDD(OBDD) is a BDD whose input variables appear in a fixed order in any path of the diagram and no variable will appear more than once in a path. A Reduced Ordered BDD(ROBDD) is given after the following two rules are applied to an OBDD:

- Eliminate all the nodes whose two edges point to the same node.

- Share all the equivalent sub-diagram.

Since we pay most of our attention to the ROBDD in the class of OBDD, without further specification, we will refer to ROBDDs as OBDDs.

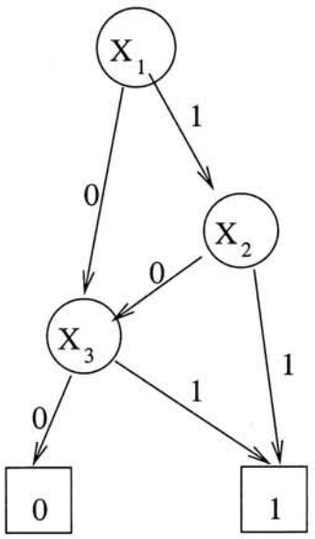


Figure 2.1: The OBDD of $x_1x_2 + x_1x_3$

Since an OBDD is uniquely determined by the function and variable order, the OBDD size of a given Boolean function depends on its variable. Different variable orders result in different OBDD sizes. There are $n!$ different variable orderings, here n is the number of dependent variables of the given Boolean functions. Therefore there must exist an ordering such that the OBDD size in this ordering is minimum among all variable orderings. Such an ordering is called an optimal ordering of the function. The OBDD size minimization problem is to find an optimal ordering for any given Boolean functions. This problem is known as a co-NP-hard problem.

An OBDD is said to be *thin* if for each variable x_i, there is only one node with input variable x_i as attribute. A Boolean function is said to be thin if there is an ordering of its dependent variables such that the corresponding OBDD is thin. If a thin OBDD contains a path from the root to terminal 1 which goes through all non-terminal nodes, then it is called a connected thin OBDD. A *connected thin Boolean function* is a Boolean function which has a connected thin OBDD representation. For instance, $f_1 = x_1 + x_2 + x_3$ and $f_2 = x_1x_2x_3$ are both connected thin Boolean functions, $f_3 = x_1x_3 + x_1'x_2$ is a thin Boolean function but not a connected thin function, and $f_4 = x_1x_2 + x_1x_3 + x_2x_3$ is not

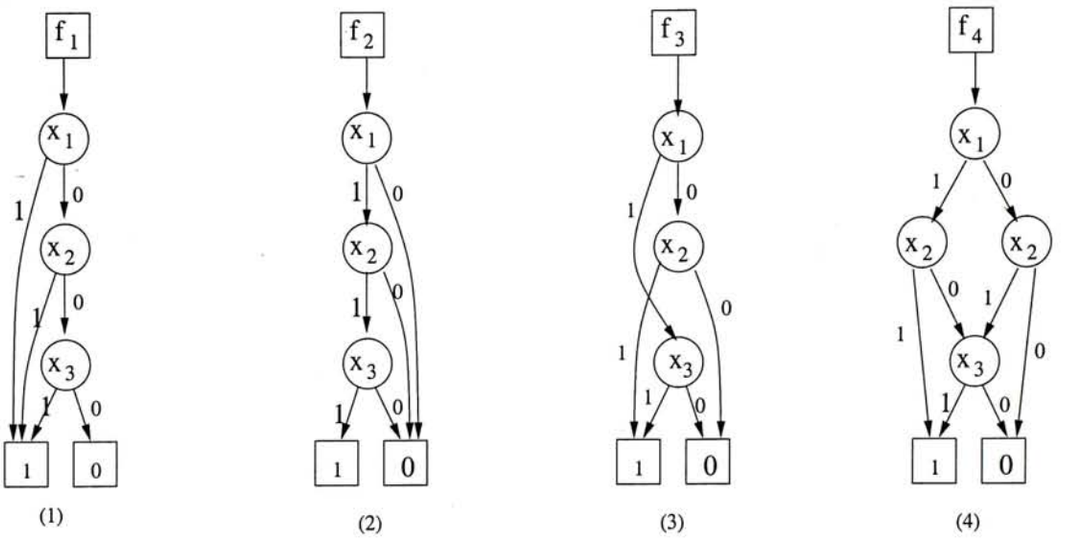


Figure 2.2: (1),(2) f_1, f_2 are thin connected, (3) f_3 is thin but not connected, (4) f_4 is not thin Boolean function.

a thin Boolean function. Figure 2.2 shows the optimal OBDDs of the above four Boolean functions.

We will study a special class of Boolean functions defined by factored from. A factored form on variable set S is said to be a thin factored form if each variable appears exactly once in the form. We refer to a Boolean function which can be expressed in thin factored form as a thin factored function. For example, $(x_1'x_2 + x_3)x_4 + x_5'$ is a thin factored form, but $(x_1 + x_2)(x_1' + x_3)$ is not a thin factored form. Clearly, a Boolean function which can be expressed by a sum of disjoint cube cover [25] is a thin factored function, where a disjoint cube cover is a cover in which no two cubes share a common variable. The expansion of a factored form is a sum of product generated by distribution operations and combining reductions.

Chapter 3

Some Previous Work on OBDD

3.1 The Work of Bryant

The research work on OBDD can be traced back to 1950's[19][1]. However, it is the originative work of R. E. Bryant[5] that really draws the attention of the researchers in last decade. From then on, there are thousands of papers published on the topic that relevant to this topic.

In [5], the author has first given a brief introduction on the logic function representation problem. He gave the main drawbacks of the logic representation techniques at that time and on the contrary, the advantages of the representation of OBDDs. However, the OBDDs have their own set of undesirable characteristics: the size of an OBDD is highly dependent to its variable order. In this paper, Bryant gave a very important character of OBDDs: For any Boolean function, there is only one OBDD representing it. The proof which is based on induction theory has given us the inspiration.

Besides the theory of OBDD, the author also gave the data structure of the OBDDs which is easy to be implemented. Some operations on OBDDs are also defined which include: reduction, apply, restriction, composition, satisfy. The reduction algorithm transforms an arbitrary function graph into a reduced graph denoting the same function. The procedure apply provides the basic method for creating the representation of a function according to the operators in a Boolean

expression or logic gate network. The restriction algorithm transforms the graph representing a function f into one representing the function $f|_{x_i} = b$ for specified values of i and b. The composition algorithm constructs the graph for the function obtained by composing two functions. Since an elegant data structure of the OBDDs and the algorithms of the operation on them are applied, all the operations are in a polynomial complexity.

However, there still exist some logic functions which are impossible to have a polynomial complexity representation by the OBDDs. Bryant proved that for the logic function of multiplier, it is impossible to have a polynomial complexity representation by the OBDDs.

Although Bryant's work has given the basic property of the OBDD together with the operations. A very important problem has not been solved, i. e. the variable ordering problem.

3.2 Some Variations of the OBDD

Besides the normal OBDD, there are also some other kinds of OBDD. One of the variation of OBDD is the *Indexed Binary Decision Diagram* (IBDD)[17]. In an IBDD, the variables are allowed to repeat systematically along a path. The graph can be viewed as been divided into various layers, and for each layer the graph behaves like an OBDD. So if IBDD has only one layer, then it will be isomorphic to the OBDD. Such representations can be proved to be extremely space efficient. People have found that the multiplier which do not have a polynomial complexity representation in the OBDD form will have it in the IBDD form.

However, unlike OBDDs, IBDDs are not a canonical representation. For the same function, there are multiple representations even under identical orders. And this has brought great inconvenience in the application of IBDDs because more sophisticated algorithms are required to compare two IBDDs for equivalence.

If we place a further restriction on IBDD that all k-layers of graphs must have the same variable order, then the IBDD will become a kOBDD. It is easy to see that the IBDD will have a more compacted representation compared with the OBDD and inferior to the IBDD. And the time taken to check the satisfiability of such an IBDD of size G is bounded by a polynomial of $O(|G|^{2k-1})$.

Another representation for functions which maps Boolean vectors to integer values has also been developed. This representation is called Binary Moment Diagram(BMD). Instead of using the standard Boolean-Shannon expansion, they describe a function f in terms of a variable x using Davio expansion.

A variation of BMD of a function is the MTBDD that results from applying the inverse Reed-Muller transformation to the given function.

Zero-suppressed BDD(ZBDD) targets for the representation of sets of combinations[21]. The ZBDD is gained by applying the following two reduction rules:

- Eliminate all the nodes whose 1-edge points to the 0-terminal node and use the subgraph of the 0-edge.
- Share all equivalent subgraphs in the same way as for ordinary BDDs.

Figure 3.1 shows how the ZBDDs represent the sets of combinations. A feature of ZBDDs is that the form is independent of the number of inputs as long as the sets of combinations are the same. It is very efficient when we manipulate very sparse combinations.

From above description, we can see that most of the variation of the OBDD have just loosen some of the restrictions of the OBDDs. However, it carries out two effects:

- It may lead to a more compacted representation which can represent some of the functions which can not be represented by the normal OBDD.
- It may lose the canonicity which is the most important property of OBDDs.

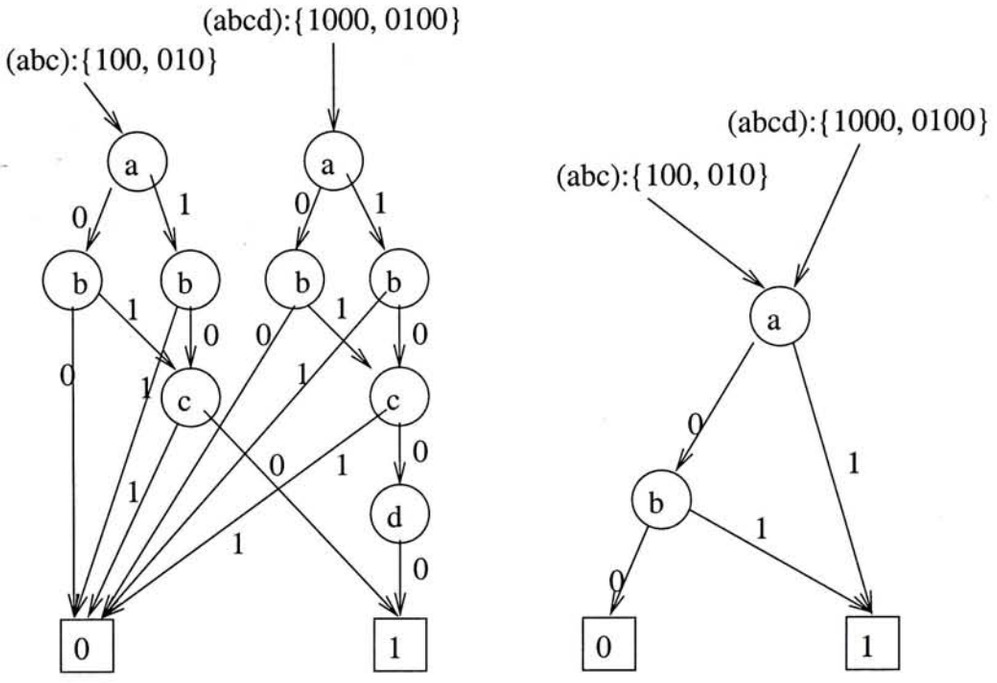


Figure 3.1: The OBDD and ZBDD of the function $f(abc) = ab'c' + a'bc'$ and $f(abcd) = ab'c'd' + a'bc'd'$

Most of the different variations of OBDD are just the trade-off between the time complexity and space complexity.

3.3 Previous Work on Variable Ordering of OBDD

3.3.1 The FIH Heuristic

Although there are many variations of OBDDs. The normal OBDD has never lost its importance because of its canonicity. However, the variable ordering problem remains unsolved and we know that it is a co-NP-complete problem.

The first very important variable ordering algorithm is given by Sharad Malik etc. in 1988[20]. The famous logic synthesis tool, SIS, has applied this algorithm to give the variable order for the construction of the OBDDs. Until nowadays, people always compare their variable ordering result with this algorithm.

The algorithm is based on the observation that the multi-level Boolean network can be treated as the combination of two-level Boolean network with the inputs as the primary inputs of the original network and some intermediate nodes.

The intermediate nodes are the outputs of some Boolean network of last level and the inputs of some Boolean network of the next level. For each node in a logic network, the inputs of it and the operation of that node are the information which are encoded into it. From the above observation, we can see that there is a similarity in the functions of intermediate nodes in a multi-level network and the vertices in a BDD. Both of them encode information about variables that is needed in subsequent levels.

For the convenience of description on the algorithm, some notations are introduced. The *transitive fanin DAG*(TFI DAG), η_n of a node n in the network η, consists of n, nodes that are transitive fanins of n and the edges between these nodes. The level of a primary output node is 0. For any other node, the level is given by:

$$level(n_i) = \max_j(level(n_j)) + 1,$$

n_j is a fanout of n_j

For a set of nodes S which does not fan out to any level less than l and a node n_i fans out only to levels less than l. Placing n_i in the order after all the nodes in S will be a good choice. So the ordering of the variables is just a process that orders the variables for each TFI DAG and concatenates these orders. The idea of the *level* heuristic was extended in visiting the TFI DAGs in order of decreasing depth. And the depth of a TFI DAG is the maximum level of any of its nodes. This heuristic is termed the *fanin* heuristic(FIH).

3.3.2 The Dynamic Variable Ordering

Though the FIH algorithm has given a significant improvement over random variable ordering, people are not satisfied with its performance: it is impossible to form the OBDDs of some large circuits.

In the process of the FIH algorithm, the position of each variable is determined when it has been picked out once. For a Boolean function, after the FIH algorithm

is applied, the variable order will be determined, the OBDD is generated and won't be changed after then. Even after some operations have been applied to the OBDDs, the variable order of the original OBDD is still maintained. So we say that the algorithm is not dynamic.

The dynamic variable ordering algorithms are quite different. The position of a variable is not fixed before the ordering is finished. To gain improvement, some methods such as the genetic algorithm and simulated annealing are also applied to adjust the variable order. Sometimes these algorithms are applied until there is no further improvement.

In the paper of [24], Rudell suggested a dynamic variable ordering algorithm. After the variable order is given randomly or by another heuristic, the variables are swapped with the neighbor variables. It has been found that swapping the order of two adjacent variables in an OBDD affects only the DAG nodes at the two levels; all other nodes remain unchanged. This property brings convenience in the implementation of the dynamic variable ordering. In this paper, Rudell has given two algorithms. One is the Window Permutation Algorithm which exhaustively search all $k!$ permutations of the k adjacent variables. Another one is the Sifting Algorithm which try to find the optimum position for each variable when it assumes all other variables remaining fixed. It is reported that the Sifting Algorithm has a better performance than the Window Permutation Algorithm.

The dynamic variable ordering algorithm suggested by Rudell allow the OBDD package determine and maintain the variable order by itself. The variable order is changed automatically by the OBDD package, transparently to the user as operations are performed. In the implementation of Rudell, he does the dynamic variable swapping in the garbage collection stage.

It is reported that if the maximum DAG size is set to 100,000 nodes, it is not possible to form the OBDD's for 11 of the 35 largest multiple-level circuits by the FIH algorithm. However, the Sift Algorithm only fails two of them. But

the drawback of dynamic variable ordering is that the runtime for the OBDD operations increases significantly.

3.3.3 The Interleaving method

The interleaving method mainly targets to the multiple output circuits. In the variable ordering algorithms before, people mainly discussed on the single output logic functions. But we know that normally, the logic functions are multiple output ones. The previous algorithm solved this matter by order the variables of some function first and then sort the variables of other functions. The new variables are appended to the previous variable list. Such algorithms may not be so suitable for the multiple output functions.

In [13], a variable ordering algorithm which is based on the FIH algorithm is suggested to attack the multiple output functions. In this algorithm, all nodes in the circuit are ordered and each node is basically ordered next to the previously ordered gate. When a gate which is visited from an output with higher priority is revisited, the insertion point of the newly ordered node for ordering is modified to be next to the revisited gate.

However, the interleaving method also has its own drawback. In some cases, it is unreasonable to insert some variables before other ones. From the experiment results we can see that the interleaving algorithm and the FIH algorithm get the same results in most of the benchmarks. And both of them lead to smaller OBDD in some benchmarks.

Chapter 4

Two Level Logic Function and OBDD

Most of the previous work on the variable ordering problem is focused on the Boolean network. But the suggested algorithms have the disadvantages that they can not be applied to the incompletely specified logic functions. And most of them have not deeply explored the logic relationship between the OBDD and the logic function. We find that it is easier to study the two-level logic function. So most of our work is based on two-level logic functions.

We give our heuristics based on the study of the two-level logic functions. For the variable ordering problem, people have the following principles after the observation of many OBDDs:

1. Sharing.

Sharing is the most important factor that influences the size of OBDD. A good OBDD must fully take the advantage of the sharing of nodes. However we can not say that an OBDD with the largest number of sharing nodes will definitely lead to a small OBDD because it depends on the two subgraph of the OBDD also. If the variable order make the number of both shared nodes and the non-shared nodes great, the variable order may not be very successful.

2. The variables that affect the whole function greatly must be put in the top of the OBDD.

Here the most important variable means the variable that influences the representation of the Shannon expansion most. This is because that the most important variables may determine the full representation of the OBDD. For example, in the function $f = ab + c$, c is the most important variable because if $c = 1$, then we do not need to know the value of any other variable. In this case, there is no nonterminal node fan out from the 1-edge of the node with variable c. But for any other variable as a or b, there is no such advantage, not any assignment of the variable a or b alone can determine the value of the function. Normally, such variables have the largest number of appearance in the cubes of the logic function or appear in the shortest cubes. The shortest cubes are the cubes which cover the largest number of minterms and the variables in the shortest cubes should exist in the largest number of minterms. The choosing of such variables can obviously deduce the complexity of the sub-functions and their OBDD representations. For example, for a 3-8 selector which has three control inputs and eight data in puts, we can see that the variables who stand for the control inputs must be more important than the variables that stand for the data inputs because the control inputs will determine the whole function that the selector performs. And we will see that for such a function, the OBDD with the control variables on top is much simpler than that with input variables on top.

3. The variables which have closer relationship must be near in the ordering of the variables.

A group of variables which have the closer relationship means a group of variables which belong to the same sum term or product term. For the Boolean function and a certain assignment of the variables, such a sum term or product term may be enough to determine if the function is 0 or 1. So it is important to group the variables in the same sum term or product term to be near in the order and a path will lead to the two terminal nodes 0 or 1 in an earlier stage and no further offspring is needed. Otherwise each path will generate 2 offspring

until very late and thus a very big OBDD will be built.

From the above discussion, we know that these principles have the problems of their own:

1. The above principles are ambiguous. For a certain function, it is very hard to say which variable is the most important one. For example, a variable which has the largest number of appearances may not be the variable which appears in the shortest cube.

2. Sometimes the principles are mutually conflicting to each other. For example, for the function of $f = ac + ad + bc + bd$, principle 3 asks for a variable order that place a and b close to c and d respectively. But there is no way to implement it without violating the rules.

3. People now are still not quite clear about how to make the OBDD share more. Unlike the Binary Decision Tree, the OBDD has a compact representation so the representation of the same sub-function may be quite different. So it is very hard to say how two functions can share some logic sub-function. To make the two functions share more, it is essential to find the sub-function which appears in both two functions. The complexity will depend on the complexity of determining if two functions are unique. For a Binary Decision Tree but not a ROBDD, things are already not easy because the size of the Binary Decision Tree will be too large. However, compared with the OBDD, the Binary Decision Tree are a bit easier to study because the same cube will have the same representation, i.e. a path lead to 1 with all literals appear in that path just as appear in the cube. For an OBDD, another problem is that there exists sharing between the sub-diagrams so that even for the same cube, there is no unique representation.

4. That principles are not ready to be applied to the multi-output functions.

Here we will treat the ordering of the OBDD as a sequence of choosing the variables one by one. Suppose the number of function f is $n(f)$. Then we have

the following equation:

$$n(f) = n(f_x) + n(f'_x) - n(f_x \cap f'_x) \tag{4.1}$$

where $n(f_x)$ is the number of nodes in function f_x, $n(f'_x)$ is the number of nodes in function f'_x, $n(f_x \cap f'_x)$ is the number of nodes in the sharing of sub-diagrams of f_x and f'_x. So for a function and the set of variable V, we have the following rules to choose the variables and generate a good OBDD.

1. f_x should be simple, i. e. $n(f_x)$ should be as small as possible.

2. f'_x should be simple and $n(f_x)$ should be as small as possible.

3. The sharing of the OBDD representation of f_x and f'_x should be as large as possible.

4. The picking of one variable should make it easy to find a new variable that is the suitable for both of the two functions f_x and f'_x.

Our principles have the advantages that they are easy to follow. Though we can not know the variable order of the sub-function before the variable of the current function is picked out, and thus we can not know the size of the sub-function before any variable is picked out. The complexity of the subfunction can still be estimated if an idea of variable ordering is given. However, it is easy to define the distance between two cube and therefore the distance between two functions. In [4], the distance of two cubes are defined as the number of variable with complement appearance in the two cubes.

Chapter 5

DSCF Algorithm

For each logic function $f(x_1...x_n)$, the Shannon expansion of it is $f = xf_x + x'f_{x'}$ which can be treated as the partition of its minterms into a $(n-1)$-dimensional Boolean space by the variable x. To follow the principles, we can have the following strategy:

Try to make the partition so that all the minterms are in one half-space. The consideration of this strategy is that: since most of the minterms are in one half-space, the minterm m, which is in the half-space with less number of minterms will have a great chance that it can be absorbed by the implicant of the half-space with more minterms. Suppose $m = xc$, where x is the partition variable and c is the implicant whose restriction on x is m. It will be quite possible that there is an implicant of $m' = x'a$, where x' is the complement of x and a is the implicant with $c \subseteq a$. Then

$$xc + x'a = \begin{cases} c + x'a, & \text{if} c \subset a, \\ a = c, & \text{if} c = a, \end{cases}$$

If minterms in the half-space with less number of minterms are not covered in the other half-space, since the number of them is not large, there is a possibility that it will be in a very simple form. And a very important advantage of this strategy is that since we try to make the partition with most of the minterms in one half-space, we may assume that the number of minterms in the half-space with less number of minterms is very small. So after the first variable is chosen,

we can pay all our attention to the half-space with more minterms and need not consider the one with less minterms and by this way, the fourth principle is also obeyed. However, this strategy has the problem of its own. For two level logic minimization, it is very difficult to calculate the number of minterms in the half-space because the cubes may intersect with each other and we can not have a polynomial algorithm to check the overlap. So an approximation of this strategy is to make sure that the largest cube is in one half-space and no any of its minterms are in the other half-space. Thus we have the following algorithm:[25]

DSCF algorithm.

1. Express the function in the optimal pattern cover form.

2. Sort the patterns according to their lengths.

3. Pick the next variable to be added to the ordered sequence from the set of the shortest patterns. In case of a tie, apply one of the following strategies:

v1: The variable which appears in the largest number of unprocessed yet patterns, i. e. the most globally binate variable of shortest patterns is picked, if tie, just pick the first one.

v2: The variable which appears in the largest number of these shortest patterns is picked, if tie, check the next longer pattern list until the tie is broken.

4. After a variable has been picked, delete this variable (including its complement) from all patterns.

5. Back to 2 until all variables are picked.

The algorithm has a very low complexity. Suppose the number of cubes is m and the number of variables is n. To choose the shortest cube will have the complexity of $O(m)$. The complexity of choosing the variable that has the largest number

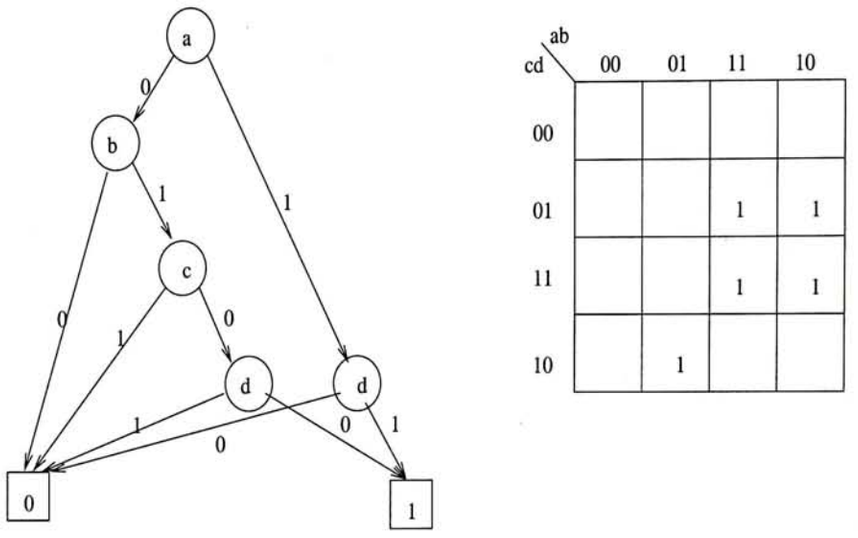


Figure 5.1: The OBDD and K-map of $ad + a'bcd'$

of appearance will have the complexity of $O(mn)$. Since there are n variables, so the total complexity is $O(mn^2)$.

But we notice that several approximation take place in this algorithm and this make it inferior in several cases.

For example, we assume that the minterms in the half-space with less minterms have the chance to be covered by the half space with more minterms and if it can not be covered, it may have a very simple representation. But it is not usually the case especially in a logic function with very few number of variables. We can illustrate it in the Figure 5. For the function $f = ad + a'bcd'$, From the figure we can see that though the minor half-space has only one minterm. It takes 2/3 of the total number of nodes in the OBDD.

Secondly, if the minterms are not so easily to be partitioned into two half with great difference of the number of minterms, this algorithm will lead to a very bad result. Let's look at the function of $f = a \oplus b \oplus c \oplus d$, the minterms of them are deployed averagely in the whole Boolean space and thus the result of the OBDD given by us is still poor. However no other order can achieve a better OBDD here.

Thirdly, even for some logic function whose minterms are not deployed aver-

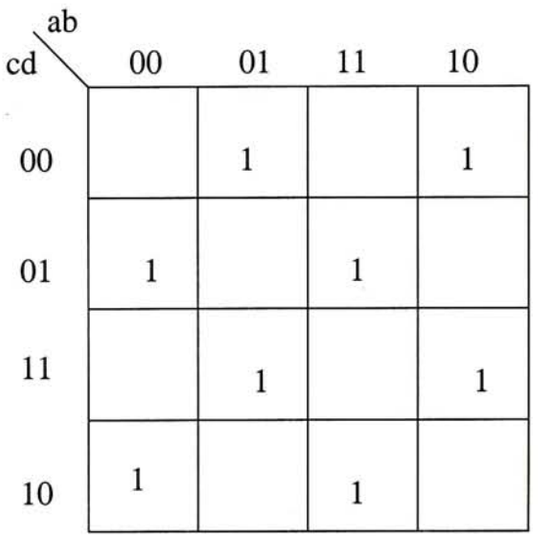


Figure 5.2: The K-map of $a \oplus b \oplus c \oplus d$

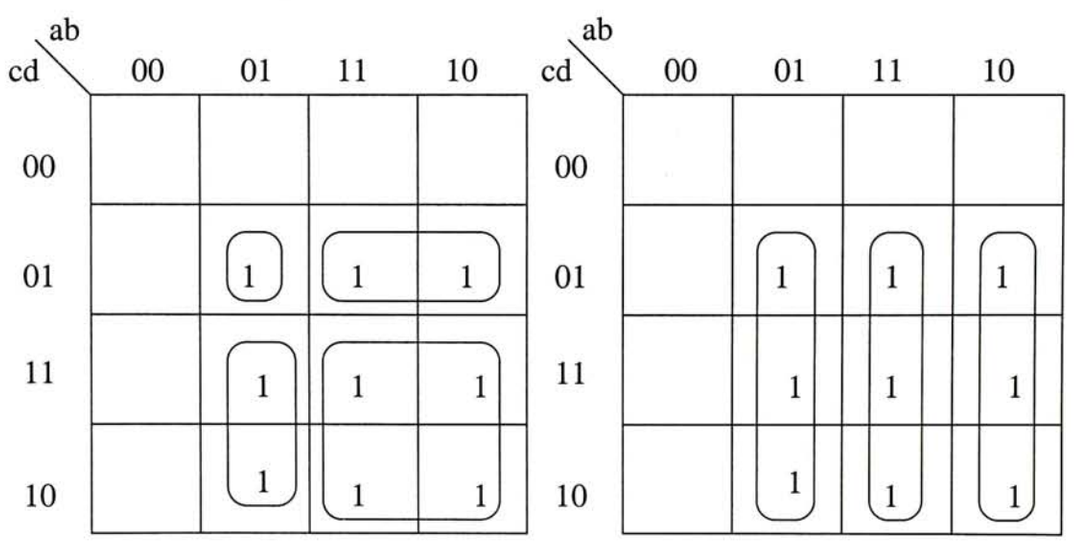


Figure 5.3: (1)The K-map of $(a+b)(c+d)$ after the variables a, c are picked out, (2) The K-map of $(a+b)(c+d)$ after the variables a, b are picked out.

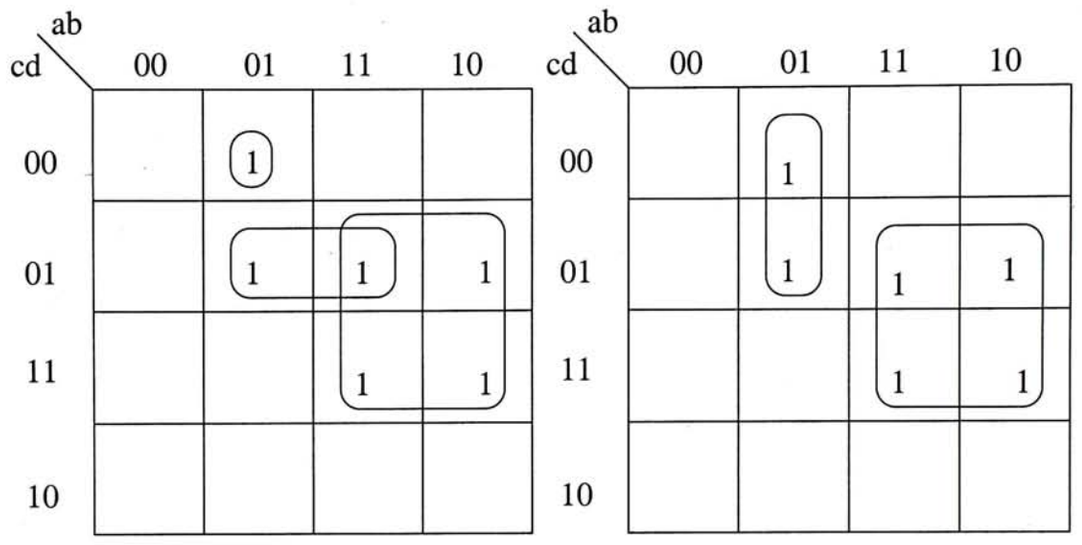


Figure 5.4: (1)K-map of $ad + a'bc'$ if the first variable is d, (2)K-map of $ad + a'bc'$ if the first variable is a.

agely in the Boolean space but is difficult to be partitioned into good shape, the algorithm also can not get good result. The worst case of this kind is the function $f = (a + b)(c + d)$. From the K-map we can see it. In Figure 5 we can see that the first two variables to be picked out should be a and b, but our algorithm can only get the variables of a and c.

Another problem is that the algorithm is not possible to control the integration of the cubes. The reason is that the algorithm is lack of the information to control the integration of the minterms. For example, for the function $f = ad + a'bc'$, the optimal variable order is a, d, b, c. From the K-map, it is easy to see that though both the distances between the two partition of minterms is 1 and the cube of f_a is more integrated than that of f_d. But because the algorithm is lack of such information, it may only get the result of d, a, b, c.

Another view point of this algorithm is to treat the OBDD as a recognizer of certain patterns. Here the pattern is defined as a finite set composed of the literals with an attribute 0 or 1 assigned to it. Note that the concept of pattern is similar to that of the implicant. The recognizer is a 3-tuple(G, S, A) where G is a directed acyclic graph, S is a set of legal input patterns on the alphabet A.

G has the following properties:

a) each non-terminal node has a variable assigned to it and is able to recognize some binary property of it, for examples polarity;

b) each leaf node has either an attribute 1(accept) or an attribute 0 (reject);

c) each non-terminal node has two out-going edges, one of which has a value 1 for a true value of the node variable and the other has a value 0 for a complemented value;

d) for each pattern a in S there exists a directed path from a root of G to a leaf t of G such that if the edge attributes are interpreted as polarity modifiers of it's parent nodes then the pattern determined by that path has a non empty intersection with the pattern a.

A pattern a is accepted by the recognizer if there exists at least one path in G originating at root and terminating at 1 leaf node and which has a non empty intersection with the pattern a. A string is rejected if there exists at least one path from a root in G which terminates at a 0 leaf node and has a non-empty intersection with a. So an accepted pattern is an implicant of the function whose representation the OBDD is, otherwise is an implicant of the complement function.

It is easy to see that for any OBDD of a function f and any essential prime p of f, there exists at least one 1-path implying p in which all literals of p are present. And for any function f, every cube of it can be recognized by any OBDD of it. So to find an optimal variable ordering minimizing the size of an OBDD is similar to the problem of finding an optimal input ordering for a pattern recognizer which would lead to an implementation with minimum number of internal nodes. We will use the name recognizer and OBDD interchangeably.

We define a spatial entropy of a pattern as the number of minterms that can imply it which is the maximal possible logical entropy of any OBDD w.r. t this pattern. The shorter a pattern is the higher its spatial entropy is. We

suspect that a pattern of higher spatial entropy has a larger effect in enlarging the logical entropy of its recognizing OBDD. Therefore the shorter patterns should be recognized with higher priority. Another observation is that grouping the supporting variables of a pattern in the ordering has a direct effect in reducing the corresponding OBDD size. However, we know that this observation can only apply for some instances. So in the algorithm, we should try to pick the variable in the same cube of the last variable already been picked out. From the algorithm, we know that it is really the case. Suppose the variable that has already been picked out is not the sole variable in the cube, than from the algorithm we know that cube is the shortest cube before the variable is picked out. And after the variable is picked out, all the same variable is deleted from the cubes and after it, all the length of the cubes should be decreased by one or zero. The length of the shortest cube will be definitely decreased by 1 and then it will still be the shortest cube. So the next variable to be chosen must be within the variables inside the shortest cube and this process goes on until all variables in the same cube is picked out.

Another interpretation of this heuristic is that we can consider each pattern as a certain job to be processed. A known queuing principle is that shorter jobs should be completed first to minimize the average jobs queuing time. Therefore scheduling shorter patterns to be "recognized" earlier would reduce the average "recognizing length" of all patterns and reduce the number of OBDD nodes needed to encode this process.

To further study the Boolean functions and their OBDDs, we have also studied the relationship between the OBDD and essential prime implicants. We try to improve the performance of the DSCF algorithm by applying it on the EPI's cover. However, the result can not make people satisfied and the details are listed in the appendix.

Chapter 6

Thin Boolean Function

Thin OBDDs and thin Boolean functions have many good properties. First of all, a thin OBDD is an optimal OBDD. This is because each dependent variable of a Boolean function must appear as an input variable of some node in the OBDD, so that for each variable there is at least one node attributed to this variable. The thin OBDD has just one node for each dependent variable, therefore it is an OBDD with minimum number of nodes. [26]

6.1 The Structure and Properties of thin Boolean functions

In this section, we first study the structure and construction of thin OBDDs, then give an estimation of non-isomorphic thin OBDDs. Secondly, we study the cube cover properties of thin Boolean functions, and thirdly, the properties of thin factored functions.

6.1.1 The construction of Thin OBDDs

For convenience, we regard non-terminal nodes of an OBDD with the same input variable attribute x_i as at level i, and assume that the levels $1, 2, \ldots, n$ are numbered from bottom to top. Therefore, a thin OBDD is an OBDD with each level containing only one node. When considering thin OBDDs, we simply denote a

node with input variable x_i by x_i, and (x_i) the represented Boolean function at node x_i.

Given a thin OBDD of n non-terminal nodes, we delete the root, which results in a digraph containing at least one node without incoming arcs. We then delete one of such nodes and the resulting digraph will again contain at least one node without incoming arcs. Repeating this process n times, we will finally obtain the terminal node 1 and 0. The original OBDD can be constructed backward. From this construction method, we obtain an algorithm for generating all non-isomorphic thin OBDDs with n variables.

Algorithm 1 . Generate thin OBDD

0. Let 1 and 0 be the starting two terminal nodes, and $G_0 = (\{1, 0\}, \emptyset)$.

1. Add node x_1 to G_0. Let x_1 point to two different nodes; label one of the new arcs 1 and the other 0. The edge-labeled digraph obtained is denoted by G_1.

2. If $k = n$, stop. Output $G = G_k$. Otherwise goto step 3.

3. Add x_{k+1} to G_k. If $k \leq n/2$, or $k > n/2$ and the number of 2-nodes (the node with no incoming arc) of G_k is less than $n - k$, then let x_{k+1} point to any two different nodes of G_k. If $k > n/2$ and the number of 2-nodes is equal to $n - k$, then let at least one edge of x_{k+1} point to a 2-nodes of G_k. Otherwise, let x_{k+1} point to two different 2-nodes of G_k. Label one of the new arcs 1 and the other 0. Let the new edge-labeled digraph obtained be G_{k+1}. Go to step 2.

Figure 6.1 shows an example of the construction of a thin OBDD with six variables. From Algorithm 1, an recursive formula for the number of non-isomorphic thin OBDDs can be obtained. We consider the edge-labeled digraphs generated in the middle of Algorithm 1 containing m non-terminal nodes, of which h are

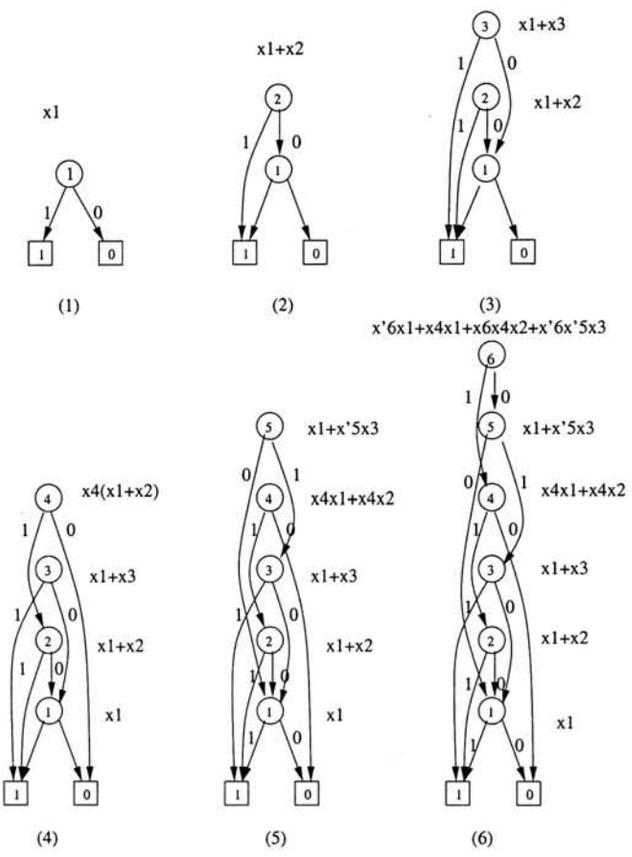


Figure 6.1: (1)-(6): the construction step of a thin OBDD.

2-nodes. Such a digraph is called an (m,h)-digraph. Let $T(m,h)$ denote the number of all non-isomorphic (m,h)-digraphs. Since an (m,h)-digraph can be obtained (1) from an $(m-1,h-1)$-digraph by adding a 2-node pointing to two non-2-nodes, (2) from an $(m-1,h)$-digraph by adding a 2-node pointing to a 2-node and a non-2-node, or (3) from an $(m-1,h+1)$-digraph by adding a 2-node pointing to two 2-nodes, the following recursive equation holds:

$$\begin{aligned} T(m,h) &= (m-h+2)(m-h+1)T(m-1,h-1) \\ &\quad +2h(m-h+1)T(m-1,h)+h(h+1)T(m-1,h+1) \quad (6.1) \end{aligned}$$

The initial values are $T(m,0)=0$, $T(1,1)=2$, and $T(m,h)=0$ when $m<h$.

Using the iteration equation 6.1, we can compute the number of non-isomorphic thin OBDDs of n non-terminal nodes by

$$T(n,1)=2nT(n-1,1)+2T(n-1,2) \quad (6.2)$$

For example, $T(2,1)=2\times 2T(1,1)+2T(1,2)=4T(1,1)=8$, $T(3,1)=6T(2,1)+2T(2,2)=6\times 8+2\times 2T(1,1)=48+8=56$, $T(4,1)=8T(3,1)+2T(3,2)=464+2\times(6T(3,1)+8T(2,2))=464+2\times(6\times 56+8\times 4)=1200$. Then we can get an estimation of $T(n,1)$.

Theorem 1

$$2^n n! \le T(n,1) \le 2^{n^2}$$

Proof. We first prove $T(m,h)\le 2^{m^2}$ by induction on m. It is clearly true when $m=2$. Assume that it is true when $m<k$. Then for $m=k\ge 3$,

$$\begin{aligned} T(m,h) &= (m-h+2)(m-h+1)T(m-1,h-1)+2h(m-h+1)T(m-1,h) \\ &\quad +h(h+1)T(m-1,h+1) \\ &\le (m-h+2)(m-h+1)2^{(m-1)^2}+2h(m-h+1)2^{(m-1)^2} \\ &\quad +h(h+1)2^{(m-1)^2} \\ &= ((m-h+2)(m-h+1)+2h(m-h+1)+h(h+1))2^{(m-1)^2} \end{aligned}$$

$$
\begin{aligned}
&= (m^2+3m+2)2^{(m-1)^2} \\
&< (3m^2)2^{(m-1)^2} \\
&= 3m^2 2^{(m-1)^2} = 2^{(m-1)^2+\log_2 3m^2} \\
&= 2^{(m-1)^2+log_2 3+2\log_2 m} \\
&< 2^{m^2+(2\log_2 m-2m+2.55)} \\
&= 2^{m^2}2^{2\log_2 m-2m+2.55} \\
&< 2^{m^2}
\end{aligned}
$$

Next we prove $T(n,1) \leq 2^{n^2}$ by induction on n. It is clearly true when $n = 2, 3$. Assume that it is true when $n < k$, and prove the truth for $n = k > 3$. Since $T(m,h) \leq 2^{m^2}$, then $T(m,2) \leq 2^{m^2}$. By (6.2), we have

$$T(n,1) = 2nT(n-1,1) + 2T(n-1,2) \leq 2^{(n-1)^2+2} \leq 2^{n^2}$$

On the other hand, $T(n,1) = 2nT(n-1,1) + 2T(n-1,2) \geq 2nT(n-1,1) \geq \ldots \geq 2^n n!$. ∎

From Theorem 1, we know that only a small amount of Boolean functions which are thin Boolean functions. To construct all non-isomorphic connected thin OBDDs, we only need to change the step 3 of Algorithm 1 to:

Step 3. Add x_{k+1} to G_k. Let x_{k+1} point to one 2-node and another arbitrary node. Label one of the new arcs 1 and the other 0. Let the new edge-labeled directed graph obtained be G_{k+1}. Go to step 2.

It is clear that the number of non-isomorphic connected thin OBDDs with n non-terminal nodes is equal to $2^n n!$.

6.1.2 Properties of Thin Boolean Functions

Thin Boolean functions are defined by the characterization of OBDD. We will give some properties of cube cover form for thin Boolean functions. These properties provide conditions for optimal variable ordering heuristics of Boolean functions.

Theorem 2 *For any support variable x_i of a Boolean function $f(x_1, x_2, ..., x_n)$ which has an EPI cover, the cofactors, f_{x_i}, f'_{x_i}, still have an EPI cover.*

Proof.

First, we can construct the cube p' of f_{x_i} as following:

For any p of f,

if x_i appears in p, then we have $p' = p_{x_i}, p_{x_i}$ is the cube with x_i deleted;

if x'_i appears in p, we discard p;

if x_i do not appears in p, we just have $p' = p$.

Obviously, the p's of f_{x_i} are still prime implicants of the Boolean function $f(x_1, x_2, ..., x_{i-1}, x_{i+1}, ..., x_n)$ over the Boolean space $B(x_1, x_2, ..., x_{i-1}, x_{i+1}, ..., x_n)$. Since the p' is in f_{x_i}, $x'p'$ is not a prime implicant of $f(x_1, x_2, ..., x_n)$. Suppose the p' is not a prime implicant of $f(x_1, x_2, ..., x_{i-1}, x_{i+1}, ..., x_n)$.

if there exists a minterm $m \in p'$ and $m \notin f$, then $x_i m$ and m will not be a prime implicant of $f(x_1, x_2, ..., x_{i-1})$;

if there exists another prime implicant m' covers the m in the function $f(x_1, x_2, ..., x_{i-1}, x_{i+1}, ..., x_n)$ over the Boolean space $B(x_1, x_2, ..., x_{i-1}, x_{i+1}, ..., x_n)$, then $x_i m'$ and m' will also cover $x_i m$ and m. And neither of them can be the prime implicant of $f(x_1, x_2, ..., x_{i-1})$;

It is also easy to see that the set of p's is a proper cover of f_{x_i}.

For an essential point m of a prime p' of f_{x_i}, x_i must appear in p. Consider m_{x_i}, and m_{x_i} must be a minterm of f_{x_i}. Suppose there exists another $p_2 \neq p$ cover m_{x_i}, then in f, $x_i p_2$ or p_2 must cover m, so m is not an essential point. The assumption is not true and there is no other prime covers m_{x_i} and m_{x_i} is also an essential point. So we can say that any p of f_{x_i} is still an EPI and f_{x_i} still has an EPI cover. For the same reason, f'_{x_i} also has an EPI cover. The proof is completed.

∎

Lemma 1 *The cofactors of a thin Boolean function are also thin Boolean functions.*

Proof. Suppose that an OBDD of function f is given and x_i is a variable of f. Then the OBDD of $f|_{x_i}$ can be obtained from the OBDD of f by following three operations: a) delete all edges with attribute 0 and out of nodes with input variable x_i, b) conduct all edges that points to the node with the input variable x_i to its high son , c) do the reduction. Clearly, above operations will not increase the number of nodes in each level. Therefore when the original OBDD is a thin OBDD, then the OBDD obtained from above operations is still a thin OBDD. This implies that the cofactors of thin Boolean functions are also thin Boolean functions. ∎

Lemma 2 *A Boolean function is a thin Boolean function if and only if its complement is a thin Boolean function.*

Proof. The OBDD of f', the complement of f, can be obtained from the OBDD of f by exchanging the terminal nodes 1 and 0. Therefore, if f has a thin OBDD representation, then f' has a thin OBDD representation and f' is a thin Boolean function. ∎

Lemma 3 *The sum, product and composition of thin Boolean functions on disjoint sets of variables are still thin Boolean functions.*

Proof. Let D_1 and D_2 be the thin OBDDs of variable disjoint thin Boolean functions f_1 and f_2, respectively.

The OBDD of $f_1 + f_2$ can be obtained in the following way. Change the arcs pointing to node 1 of D_1 into arcs pointing to node 1 of D_2, and arcs pointing to 0 of D_1 into arcs pointing to the root of D_2. Clearly, the OBDD obtained is a thin OBDD and the root of D_1 represents the function $f_1 + f_2$, so that $f_1 + f_2$ is a thin Boolean function. (Figure 6.2 shows an example of the sum of two variable disjoint thin OBDDs.)

By the above result and Lemma 2, we know that $f_1 f_2 = (f_1' + f_2')'$ is a thin Boolean function.

Let x_i be a variable of f_1. Replace x_i by f_2, we get a composition of f_1 and f_2, denoted by $f|_{x_i=f_2}$. We show that $f|_{x_i=f_2}$ is also a thin Boolean function. Since D_1 is a thin OBDD of f_1, D_1 has a node v with input variable x_i as attribute. Replace node v by D_2 with all incoming arcs of x_i pointing to the root of D_2, all arcs pointing to 1 of D_2 now pointing to the high son of v, and all arcs pointing to 0 of D_2 now pointing to the low son of v. The OBDD obtained is a thin OBDD with root representing $f|_{x_i=f_2}$. ∎

From the proof of Lemma 3, we obtain following theorem.

Theorem 3 ([25]) *For a function f which is expressed by a sum of disjoint cubes cover, any OBDD ordering with variables of the same cube grouped continuously is an optimal variable ordering.*

Theorem 4 *Each thin Boolean function has an essential prime cover.*

Proof. Let G be a thin OBDD of thin Boolean function $f(x_1, x_2, \ldots, x_n)$ with the optimal variable order sequence $[x_n, x_{n-1}, \ldots, x_1]$. Then each node in G

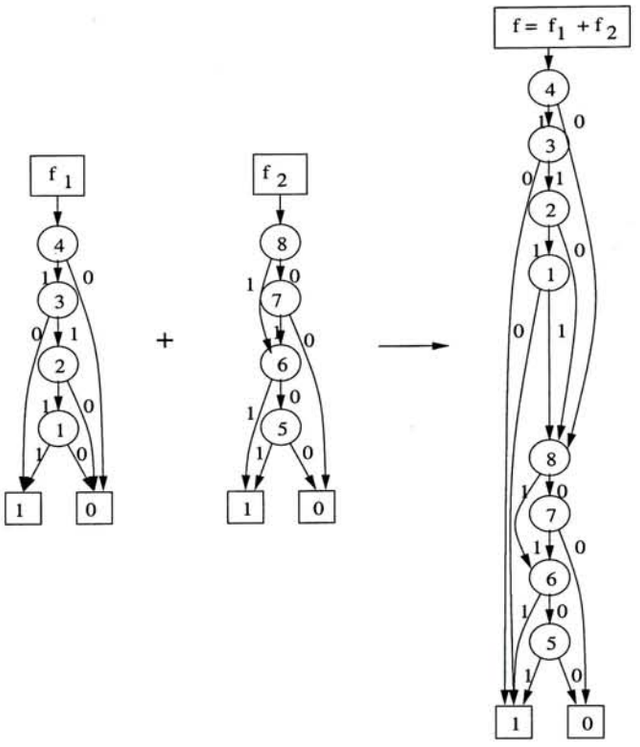


Figure 6.2: The composition of two variable disjoint thin Boolean functions.

, x_i, corresponds to a Boolean function which is also a thin Boolean function according to Lemma 1. Use the method given in [23] to construct a prime cover for $(x_i), i = 1, 2, \ldots, n$ from bottom to top. We will prove that the prime cover is also an essential prime cover by induction on i.

Suppose that we have constructed prime covers $C(x_i)$ of $(x_i), i = 1, \ldots, k-1$. Let $x_h = high(x_k), x_l = low(x_k)$. We can construct the prime cover $C(x_i)$ for (x_k) by the following algorithm.

Algorithm 2 : Compute a prime cover by OBDD

0. Let $C = \emptyset, C_h = C(x_h), C_l = C(x_l)$.

1. If $C_h = \emptyset, C_l = \emptyset$, stop. Output $C(x_k) = C$. Otherwise, if $C_h \neq \emptyset$, go to step 2; otherwise $C_l \neq \emptyset$, go to step 2.

2. Choose any cube $c_i^h \in C_h$. If c_i^h is contained in some cube of $C(x_l)$, then let $C = C \cup \{c_i^h\}, C_h = C_h \setminus \{c_i^h\}$. Otherwise let $C = C \cup \{x_k c_i^h\}, C_h = C_h \setminus \{c_i^h\}$

. Go to step 1.

3. Choose any cube $c_j^l \in C_l$. If c_j^l is contained in some cube of $C(x_h)$, then let $C = C \cup \{c_j^l\}, C_l = C_l \setminus \{c_j^l\}$. Otherwise let $C = C \cup \{x_k' c_j^l\}, C_l = C_l \setminus \{c_j^l\}$. Go to step 1.

By the theorem in [23], we know that the output $C(x_k)$ of Algorithm 2 is a prime cover of (x_k). Let $C'(x_i)$ denote the set of primes of (x_i) which are not in $C(x_i)$, and $P(x_i)$ the set of all primes of (x_i). Then $P(x_i) = C(x_i) \cup C'(x_i)$. Clearly, $C'(1) = \emptyset, C'(0) = \emptyset$.

Property 1 *There is at least one cube in $C(x_i)$ containing literal x_i. If the maximum index of literals of cube $c \in C(x_i)$ is j, then c is also a cube in $C(x_j)$.*

We prove the first statement by induction on i. It is clearly true when $i = 0, 1$. Suppose that it is true for $i < k$, we will prove it is true for $i = k$. Let $x_h = high(x_k), x_l = low(x_k)$, and without loss of generality, assume that $h > l$. Then by the induction hypothesis, there exists a cube $c \in C(x_h)$ such that c contains literal x_h. No cube of $C(x_l)$ contains literal x_h, so that c is not contained in any cube of $C(x_l)$, $x_k c$ is a cube in $C(x_k)$ by Algorithm 2.

Suppose that the maximum index of literals of cube $c \in C(x_i)$ is j. Then $j \leq i$. It is clearly true when $j = i$. Suppose $j < i$, since c does not contain literal x_i, c must be a cube of either $C(high(x_i))$ or $C(low(x_i))$. Since c can not be a cube in $C(x_m), m < j$, continue the above process, we will have $c \in C(x_j)$. For simplicity, we say that two cubes are non-containment cubes if either of them contains the other.

Property 2 *If $c_j^i \in C(x_i)$ and $c_{j'}^{i'} \in C(x_{i'})$ are non-containment cubes, then no cube $c \in \cup_{i=1}^n C(x_i)$ can be contained in both c_j^i and $c_{j'}^{i'}$.*

By contradiction, suppose that $c \in \cup_{i=1}^n C(x_i)$ is contained in both c_j^i and $c_{j'}^{i'}$. By Property 1, we may assume that i and i' are the maximum indices of literals of c_j^i and $c_{j'}^{i'}$, respectively. The literal set of c must contain both literal sets of

c_j^i and $c_{j'}^{i'}$ properly. Let i'' be the maximum index of literals in c, then c is a cube of $C(x_{i''})$ by Property 1. Since c is contained in both c_j^i and $c_{j'}^{i'}$ and that $C(x_i)$ and $C(x_i')$ are prime covers, c can not be in $C(x_i)$ and $C(x_{i'})$. Otherwise there will be mutual containment. Therefore $i'' > i, i'' > i'$. By the construction of $C(x_{i''})$, we know that the cube c' formed by deleting the literal $x_{i''}$ is also a cube in $\cup_{i=1}^{k} C(x_i)$. Clearly, the literal set of c' contains the literal sets of both c_j^i and $c_{j'}^{i'}$. Continuing this process, we will finally get a cube $c'' \in \cup_{i=1}^{n} C(x_i)$ with maximum index j of literals that equal to $max\{i, i'\}$ and is contained in both c_j^i and $c_{j'}^{i'}$. On the other hand, j should satisfy $j > i, j > i'$ by the previous discussion, which leads to a contradiction.

Property 3 *Every cube in $C'(x_k)$ must be contained in at least two non-containment cubes of $\cup_{i=1}^{k-1} C(x_i)$.*

By induction on k. It is clearly true when $k = 1, 2$. Suppose that it is true for $k-1$. Let $p \in C'(x_k)$. If x_k is a literal of p, then the cube p' obtained by deleting x_k from p must be a prime cube of $C'(x_h)$ or $C'(x_l)$, otherwise $p \in C(x_k)$. And p' is contained in at least two different cubes of $\cup_{i=1}^{h-1} C(x_i)$ or $\cup_{i=1}^{l-1} C(x_i)$ by the induction hypothesis, where $x_h = high(x_k), x_l = low(x_k)$. Now suppose that p does not contain literal x_k. Since $(x_k) = x_k(x_h) + x_k'(x_l)$, p must be an implicant of both (x_h) and (x_l) . Let p_1 and p_2 be primes of (x_h) and (x_l) containing p, respectively. Since $x_k \notin p$ and p is prime, $p = p_1 \cap p_2$. If $p_1 \in C'(x_h)$ or $p_2 \in C'(x_l)$, then the statement is true by the induction hypothesis, otherwise p_1, p_2 are cubes of $C(x_h)$ and $C(x_l)$, respectively. If p contains p_2 properly, then $p = p_2$, $x_k p$ would be a prime cube in $C(x_k)$, a contradiction to that p is a prime of (x_k). If $p_1 = p_2$, then $p = p_1 = p_2 \in C(x_k)$, a contradiction to $p \in C'(x_k)$. It is also a contradiction if p_1 is contained in p_2. Therefore, p_1 and p_2 are non-containment cubes. The proof of Property 3 is complete.

Next, we construct an essential point $e_k(c)$ for each prime cube c in $C(x_k)$ with respect to Boolean function (x_k). The construction begins from bottom to

top. At each step m, we construct a point $e_m(c)$ over variable set $\{x_1, x_2, \ldots, x_m\}$ for each cube $c \in \cup_{i=1}^{m} C(x_i)$ satisfying the following two conditions:

(1) For any cube $c_j^i \in C(x_i)$ and cube $c_l^h \in C(x_h)$, $i \leq m, h \leq m$. $e_m(c_j^i)$ is contained in c_l^h if and only if c_j^i is contained in c_l^h.

(2) If $c_j^i \in C(x_i)$, then $e_m(c_j^i)$ is not in any prime of (x_i) except c_j^i.

When $m = 1$. $C(x_1) = \{x_1\}$ or $\{x_1'\}$. If $C(x_1) = \{x_1\}$, let $e_1(x_1) = \{x_1 = 1\}$, $e_1(1) = \{x_1 = 0\}$. If $C(x_1) = \{x_1'\}$, let $e_1(x_1') = \{x_1 = 0\}$, $e_1(1) = \{x_1 = 1\}$. Clearly, $e_1(..)$ satisfies condition (1) and (2).

When $m = 2$. If $high(x_2) = x_1$,$low(x_2) = 0$ and $C(x_1) = \{x_1\}$, then $C(x_2) = \{x_2x_1\}$. Let $e_2(x_2x_1) = e_1(x_1) \cup \{x_2 = 1\}$, $e_2(x_1) = e_1(x_1) \cup \{x_2 = 0\}$, $e_2(1) = e_1(1) \cup \{x_2 = 0\}$. If $high(x_2) = 1$, $low(x_2) = x_1$ and $C(x_1) = \{x_1\}$, then $C(x_2) = \{x_2, x_1\}$. Let $e_2(x_2) = e_1(1) \cup \{x_2 = 1\}$, $e_2(1) = e_1(1) \cup \{x_2 = 0\}$ and $e_2(x_1) = e_1(x_1) \cup \{x_2 = 0\}$. If $high(x_2) = 1$ and $low(x_2) = 0$, then $C(x_2) = \{x_2\}$. Let $e_2(x_2) = e_1(1) \cup \{x_2 = 1\}$, $e_2(1) = e_1(1) \cup \{x_2 = 0\}$, $e_2(x_1) = e_1(x_1) \cup \{x_2 = 0\}$. Other cases can be defined similarly. It can be verified that $e_2(..)$ satisfies conditions (1) and (2).

Suppose that we have constructed a point $e_{k-1}(c)$ for each cube $c \in \cup_{i=1}^{k-1} C(x_i)$, which satisfies the condition (1) and (2). It remains to construct the points $e_k(c)$ for each cube $c \in \cup_{i=1}^{k} C(x_i)$, and to prove that $e_k(c)$ satisfies condition (1) and (2).

By the algorithm for $C(x_k)$, the cubes in $C(x_k)$ can be partitioned into five groups:

$$\begin{aligned} C(x_k) \;=\; & \{c_{i_1}^h | i_1 \in I_1\} \cup \{x_k c_{i_2}^h | i_2 \in I_2\} \\ & \cup \{c_{i_3}^h | i_3 \in I_3\} \cup \{c_{i_4}^l | i_4 \in I_4\} \\ & \cup \{x_k' c_{i_5}^l | i_5 \in I_5\}. \end{aligned}$$

Here $C(x_h) = \{c_i^h | i \in I_h\}, C(x_l) = \{c_i^l | i \in I_l\}, I_h = I_1 \cup I_2 \cup I_3, I_l = I_1 \cup I_4 \cup I_5$, where $\{c_{i_1}^h | i_1 \in I_1\} = \{c_{i_1}^l | i_1 \in I_1\}$; cubes in $\{c_{i_2}^h | i_2 \in I_2\}$ are not contained in

any cube of $C(x_l)$; cubes in $\{c^h_{i_3} | i_3 \in I_3\}$ are contained in some cube of $C(x_l)$ properly; cubes in $\{c^l_{i_4} | i_4 \in I_4\}$ are contained in some cube of $C(x_h)$ properly; and cubes of $\{c^l_{i_5} | i_5 \in I_5\}$ are not contained in any cube of $C(x_h)$.

Next we construct a point $e_k(c)$ for each cube c of $C(x_k)$:

For $x_k c^h_{i_2} \in \{x_k c^h_{i_2} | i_2 \in I_2\}$, let $e_k(x_k c^h_{i_2}) = e_{k-1}(c^h_{i_2}) \cup \{x_k = 1\}$, $e_k(c^h_{i_2}) = e_{k-1}(c^h_{i_2}) \cup \{x_k = 0\}$ and for all cubes $c \in \cup_{i=1}^{k} C(x_k)$ contained in $c^h_{i_2}$, let $e_k(c) = e_{k-1}(c) \cup \{x_k = 0\}$.

For $x'_k c^h_{i_5} \in \{x'_k c^h_{i_5} | i_5 \in I_5\}$, let $e_k(x'_k c^h_{i_5}) = e_{k-1}(c^h_{i_5}) \cup \{x_k = 0\}$, $e_k(c^h_{i_5}) = e_{k-1}(c^h_{i_5}) \cup \{x_k = 1\}$ and for all other cubes $c \in \cup_{i=1}^{k} C(x_i)$ contained in $c^h_{i_5}$, let $e_k(c) = e_{k-1}(c) \cup \{x_k = 1\}$.

Finally, for all the other cubes c of $\cup_{i=1}^{k} C(x_i)$ which are not updated to $e_k(c)$, let $e_k(c) = e_{k-1} \cup \{x_k = 0\}$.

We continue to prove that $e_k(..)$ satisfies the condition (1) and (2). First, the definition for $e_k(c), c \in \cup_{i=1}^{k} C(x_i)$ is well-defined. Because no two non-containment cubes contain the same cube in $\cup_{i=1}^{k} C(x_i)$, there is no cube $c \in \cup_{i=1}^{k} C(x_i)$ such that x_k in $e_k(c)$ is assigned both the value 1 and 0. This implies that $e_k(c)$ is well-defined. Let c, c' be any two cubes of $\cup_{i=1}^{k} C(x_i)$. By the definition of $e_k(..)$, $e_k(c)$ and $e_k(c')$ are points of c and c', so that if c contains c', then $e_k(c')$ is in c. Conversely, we need to show that $e_k(c)$ is not in c' and $e_k(c')$ is not in c when c and c' are non-containment cubes. If $c, c' \in \cup_{i=1}^{k-1} C(x_i)$, then by the induction hypothesis, $e_{k-1}(c)$ is not in c' and $e_{k-1}(c')$ is not in c. So does $e_k(c)$ and $e_k(c')$ by the definition of $e_k(c)$ and $e_k(c')$. Suppose that $c \in C(x_k) \setminus (\cup_{i=1}^{k-1} C(x_i))$, then c contains the literal x_k. Let $c = x_k c_1$. Then c_1 is a cube in $\cup_{i=1}^{k-1} C(x_i)$. If $c' \in \cup_{i=1}^{k-1} C(x_i)$, it is clearly true by the definition of $e_k(c)$ and $e_k(c')$ whether c_1 is equal to c' or not. Suppose $c' \in C(x_k) \setminus (\cup_{i=1}^{k-1} C(x_i))$. If $c' = x'_k c_2$, the statement is clearly true. Otherwise, $c' = x_k c_2$, $c_2 \in \cup_{i=1}^{k-1} C(x_i)$. Since c and c' are non-containment cubes, so are c_1 and c_2. $e_{k-1}(c_1)$ is not in c_2 and $e_{k-1}(c_2)$ is not in c_1 by the induction hypothesis. Therefore, $e_k(c) = e_{k-1}(c_1) \cup \{x_k = 1\}$ is

not in c' and $e_k(c') = e_{k-1}(c_2) \cup \{x_k = 1\}$ is not in c. The proof of condition (1) is complete.

For any cube $c \in C(x_k)$, $e_k(c)$ is not in other cubes of $C(x_k)$ by (1). Let p be a prime cube in $C'(x_k)$. By Property 2, p is contained in at least two non-containment cubes, say p_1, p_2, of $\cup_{i=1}^{k} C(x_i)$, and c is not contained in both p_1 and p_2 by Property 3. Hence $e_k(c)$ is not in at least one of p_1 and p_2, and therefore $e_k(c)$ is not in p. This proves (2).

By (2) we know that $C(x_k)$ is an essential prime cover. Therefore $C(x_n)$ is an essential prime cover by induction. The proof is complete. ∎

Following example explains the computation of essential point for a thin Boolean function.

Example We compute the essential prime cover for the thin Boolean function given by the final OBDD of Fig.2. The prime cover of this function is shown in Fig.4. We now compute the essential point for each cube in the prime cover. For convenience, we denote $\{x_1 = b_1, \ldots, x_k = b_k\}$ by $(b_1, \ldots, b_k)$, and use matrix E_i to represent function e_k. When $m = 1$,

$e_1(x_1) = \{x_1 = 1\} = (1), e_1(1) = \{x_1 = 0\} = (0)$.

When $m = 2$,

$e_2(x_2) = e_1(1) \cup \{x_2 = 1\} = (0, 1)$,

$e_2(1) = e_1(1) \cup \{x_2 = 0\} = (1, 0)$,

$e_2(x_1) = e_1(x_1) \cup \{x_2 = 0\} = (1, 0)$.

When $m = 3$,

$$E_3 = \begin{pmatrix} e_3 & x_1 & x_2 & x_3 \\ x_3' & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ x_2 & 0 & 1 & 0 \\ x_1 & 1 & 0 & 1 \end{pmatrix}$$

When $m = 4$,

$$E_4 = \begin{pmatrix} e_4 & x_1 & x_2 & x_3 & x_4 \\ x_4x_2 & 0 & 1 & 0 & 1 \\ x_2 & 0 & 1 & 0 & 0 \\ x_4x_1 & 1 & 0 & 1 & 1 \\ x_1 & 1 & 0 & 1 & 0 \\ x_3' & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

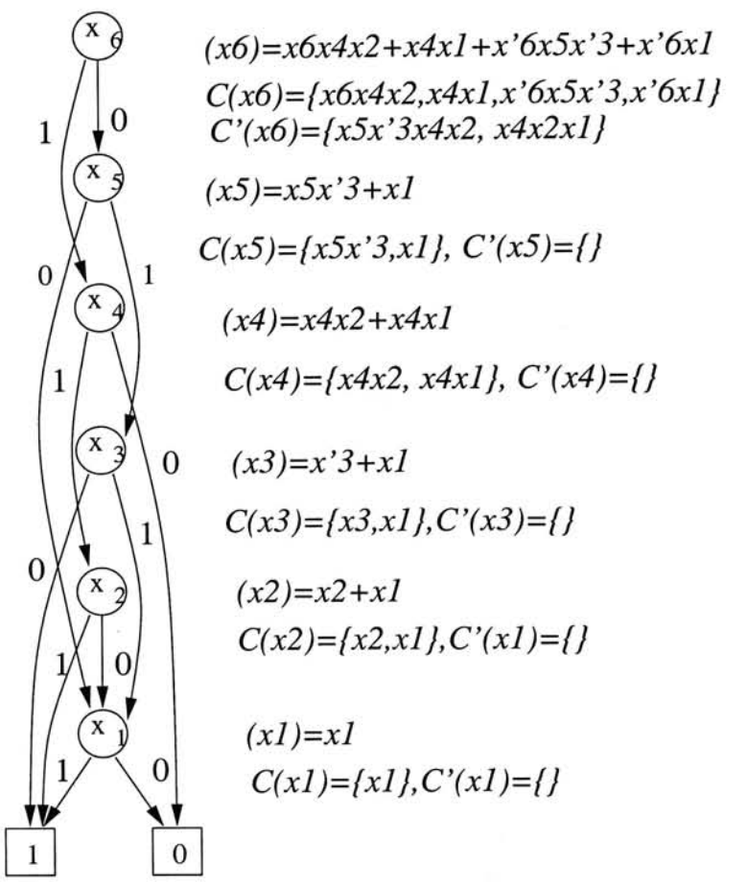
When $m = 5$,

$$E_5 = \begin{pmatrix} e_5 & x_1 & x_2 & x_3 & x_4 & x_5 \\ x_5x_3' & 1 & 0 & 0 & 0 & 1 \\ x_3' & 1 & 0 & 0 & 0 & 0 \\ x_4x_2 & 0 & 1 & 0 & 0 & 1 \\ x_2 & 0 & 1 & 0 & 0 & 0 \\ x_4x_1 & 1 & 0 & 1 & 1 & 0 \\ x_1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

When $m = 6$,

$$E_6 = \begin{pmatrix} e_5 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 \\ x_6x_4x_2 & 0 & 1 & 1 & 0 & 1 & 1 \\ x_4x_2 & 0 & 1 & 0 & 0 & 1 & 0 \\ x_6'x_5x_3' & 1 & 0 & 0 & 0 & 1 & 0 \\ x_5x_3' & 1 & 0 & 0 & 0 & 1 & 1 \\ x_6'x_1 & 1 & 0 & 1 & 0 & 0 & 0 \\ x_1 & 1 & 0 & 1 & 0 & 0 & 1 \\ x_4x_1 & 1 & 0 & 1 & 1 & 0 & 1 \\ x_4x_2 & 0 & 1 & 0 & 0 & 0 & 0 \\ x_3' & 1 & 0 & 0 & 0 & 0 & 0 \\ x_2 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

It can be seen that $e_6(x_6x_4x_2)$, $e_6(x_4x_1)$, $e_6(x_6'x_5x_3')$, $e_6(x_6'x_1)$ are in prime cubes of $x_6x_4x_2$, x_4x_1 , $x_6'x_5x_3'$, $x_6'x_1$, respectively, but they are not in other prime cubes of $C(x_6)$ and $C'(x_6)$. So $C(x_6)$ is an essential prime cover of (x_6). By Theorem 4, we know that, if a Boolean function does not contain an essential prime cover, then it is not a thin Boolean function. However, result of Theorem 4



The computation for the prime cover of a thin Boolean function by its thin OBDD.

Figure 6.3: The computation for the prime cover of a thin Boolean function by its thin OBDD.

provides a necessary condition for a Boolean function to be a thin Boolean function. It is known that unate Boolean functions also has this property because all prime cubes of unate Boolean functions are essential prime cubes. But there are unate Boolean functions which are not thin Boolean functions. For example, $f(x_1, x_2, x_3) = x_1x_2 + x_2x_3 + x_3x_1$ is a unate Boolean function, but it is not a thin Boolean function, see Figure 2.2-(4). It is hard to give a cube cover characterization of thin Boolean functions. We conjectured that the verification problem that whether or not a given cube cover form Boolean function is a thin Boolean function is a NP-complete problem.

Theorem 5 *For any node x_k of thin OBDD, if $high(x_k)$ and $low(x_k)$ share some common variables, then the set of minimal cubes of $(high(x_k))$ over the common variables is equal to that of $(low(x_k))$.*

Proof. The common variables are in both the paths from node $high(x_k)$ to 1 and from $low(x_k)$ to 1, so that they share a common cofactor on the common variables. The set of minimal cubes extended over the common variables is just the set of essential primes of the Boolean function (x_i), where x_i is a node with highest level of all common variables. Therefore the set of minimal cubes over the common variables of $(high(x_k))$ and that of $(low(x_k))$ are equal. ∎

The result of Theorem 5 is a necessary condition for an optimal variable order of a thin Boolean function. It means that at each level of a thin OBDD, the two sons of the function of the node have a maximum sharing of sub-functions. We will use this property to construct an ordering heuristic algorithm in Section 6.

6.1.3 Thin Factored Functions

Thin factored Boolean functions have much better properties. It can be recognized in polynomial bound of times with respect to the number of terms in cube cover.

Theorem 6 *Let F be a thin factored form on variable set S, and f be the thin factored function expressed by F. Then the following statements hold:*

(i) f is a connected thin Boolean function.

(ii) The algebraic expansion of F is an essential prime cover of f.

(iii) The algebraic expansion of F is the set of all prime cubes of f.

Proof. By induction on the number of variables. f can always be decomposed into either $f = f_1 + f_2$ or $f = f_1 f_2$, where f_1 and f_2 are two thin factored functions with disjoint variable sets. Then by the induction hypothesis, Lemma 3 and its proof, we know that f is a connected thin Boolean function. The second statement follows from Theorem 4 and Algorithm 2. As the terms of expansion of the factored form consist of an essential prime cover by Theorem 4.

Since each variable appears just one time in the thin factored form, so that if a variable x_i appears in a term of expansion, then x_i' can not appear in any term of the expansion; and vice versa. Therefore the function is an unate function, this implies that the set of essential prime cubes is the set of all prime cubes. ∎

Next, we give another character of thin factored functions expressed in cube cover. Let f be a Boolean function expressed in cube cover. We define four variable reduction operations on f:

R_1 If x_i is a cube, then delete x_i.

R_2 If x_i appears in every cube, then delete x_i.

R_3 If x_i and x_j appear always together, then delete x_i.

R_4 If x_i and x_j satisfy that $x_i c$ is in the cube cover if and only if $x_j c$ is in the prime cube, then delete x_j.

Theorem 7 *A Boolean function expressed in prime cover form is a thin factored function if and only if it can be reduced to null by a series of variable reduction operations.*

Proof. Suppose that f is a thin factored function expressed in prime cover. Let F be the thin factored form of f. We prove by induction on the number of variables. If F contains a product of two variables, say x_ix_j, then x_ix_j will always appear together in prime cubes of f by Theorem 6. Deleting x_j will give a thin factored function with fewer variables; so it can be reduced to null by the induction hypothesis. Suppose that F does not contain products of two variables. If there are factors, then there must be two variables x_i and x_j which are in the same bracket, now x_ic is a prime cube of f if and only if x_jc is a prime cube of f. Then x_j can be deleted by R_4. Deleting x_j, we obtain a factored form of fewer variables, and the statement is true by induction hypothesis. Finally, there are no products in the factored form. Then the prime cubes of f all consists of single literals. Now, it can be reduced to null by R_1.

Conversely, if f can be reduced to null by variable reduction operations, then we construct f backward. If x_i is deleted by R_1, then $f_{k-1} + x_i$ is the factored form of the Boolean function at the $(n-k)$-th step. If x_i is deleted by R_2, then x_if_{k-1} is the factored form of the Boolean function at the $(n-k)$-th step. If x_i is deleted by R_3, then change x_j to x_ix_j. If x_i is deleted by R_4, then change x_j to $x_i + x_j$. By this way, we will finally get the factored form of function f. ∎

Since the checking for conditions of each variable reduction operation can be done in polynomial time in terms of the number of prime cubes in the cover, the reductions can be done in polynomial time in terms of this number too. Note that the number of essential prime cubes of a thin factored function can be exponential with respect to the number of variables. For example, $f(x_1, \ldots, x_{2n}) = (x_1 + x_2)(x_3 + x_4)\ldots(x_{2n-1} + x_{2n})$ is a thin factored function, but it has 2^n essential primes. Since DSCF always choose next variable from the shortest cube, therefore, for a Boolean function expressed in disjoint cube covers, the variables in same cube will be chosen continuously by DSCF. The by Theorem 3, we have

Theorem 8 *DSCF algorithm will always find optimal OBDD ordering for Boolean*

functions expressed in disjoint cube covers.

This means that DSCF is optimal algorithm for disjoint cube cover Boolean functions. But for general thin factored functions, the DSCF is not an optimal algorithm. For example, $f(x_1, \ldots, x_n) = (x_1 + x_2)(x_3 + x_4) \ldots (x_{2n-1} + x_{2n})$ is a thin factored function. If the input of this function is in form of prime cube cover, then DSCF algorithm will find an variable ordering: $[x_1, x_3, \ldots, x_{2n-1}, x_2, x_4, \ldots, x_{2n}]$. The OBDD size of f in this ordering is 2^{n+1} because the OBDD size of its complement $x'_1x'_2 + \ldots + x'_{2n-1}x'_{2n}$ is 2^{n+1} in this variable order [5]. To overcome this shortcoming, we make an amendment to the DSCF algorithm based on the necessary conditions of thin Boolean functions.

6.2 The Revised DSCF Algorithm

Revised DSCF:

Step 0. Input Boolean function $f = f(x_1, \ldots, x_n)$ in the form of cube cover. Let $S = [x_{i_1}, \ldots, x_{i_n}]$ be any initial ordering sequence of variables. Set $j = 0$.

Step 1. Sort the patterns of f according to their lengths. Pick the next variable x_i and move it to the $(j + 1)$-th position of S. In case of tie, apply one of the following procedures:

v1: The variable which appears in the largest number of unprocessed yet patterns. i.e., the most globally binate variable of the shortest patterns is picked. In case of tie, just pick the first one.

v2: The variable which appears in the largest number of these shortest patterns is picked. In case of tie, check the next longest list until the tie is broken.(DSCF method)

$j = j + 1$;

Step 2. for j=1 to n-1

if j-th variable of S is binary

find the next variable by the DSCF method;

else Compute $S' = (I(f|_{x_i}) \cup I(f|_{x_i'})) \setminus (I(f|_{x_i}) \cap I(f|_{x_i'}))$;

Adjust the ordering of the k-th variables. $k = j+1, ..., n$ such that each variable in S' is located before all variables not in S' and each pair of variables that are both in S' or not in S' keep the order in S.

Let S be the new ordering sequence and x_i be the first unfixed variable. Set $j = j + 1$;

step 3. Delete x_i and its complement from all cubes containing x_i followed.

The complexity of this algorithm is $O(nm^2 + n^2m)$, where n is the number of variables and m is the number of input cubes.

Next, we show that the revised DSCF algorithm is optimal for thin factored functions. Let f be a thin factored function expressed in prime cover. Let F be the thin factored form of f. By Theorem 6, we know that f is a unate function, so that each variable of f is a unate variable and the algorithm will only go to the "else" branch in step 2. If x_i is a single term in the thin factored form, then $S' = (I(f_{x_i}) \cup I(f_{x_i'}) \setminus (I(f_{x_i}) \cap I(f_{x_i'}))$ is the set of variables which is in a sum term with x_i; otherwise, it is the set of variables which is in a product term with x_i. Here $t_1, \ldots, t_l$ are the sum terms with x_i if $(x_i + t_1 + \ldots + t_l)$ is a maximal sum part of F, and product terms with x_i if $x_i t_1 \ldots t_l$ is a maximal product part of F. The operation to f with respect to x_i is just the function obtained from assigning x_i to 0 if x_i is a single sum term, or 1 otherwise. Therefore, when x_i is a single term, the next variable picked by the algorithm must be a variable in a sum term with x_i; otherwise, the next variable is a variable in the product term of x_i. Such an ordering is an optimal variable ordering. We can construct a thin

OBDD with respect to the ordering backward. Following theorem follows.

Theorem 9 *The revised DSCF algorithm will always find optimal variable ordering for thin factored functions.*

Example: Consider function $f(x_1, \ldots, x_8) = x_4x_1x_2 + x_4x_3 + x_5x_7 + x_6x_7 + x_5x_8 + x_6x_8$. By the algorithm, first set $S = [x_3, x_1, x_2, x_4, x_5, x_7, x_6, x_8]$. First $j = 0$, by v1 of step 2, choose x_4, move it to the first place of S, we get $S = [x_4, x_3, x_1, x_2, x_5, x_7, x_6, x_8]$. Set $j = 1$. Go to step 1. Since x_4, the first variable of S, is a unate variable, go to step 3. $S' = (I(f_{x_4}) \cup I(f_{x_4'})) \setminus (I(f_{x_4}) \cap I(f_{x_4'})) = \{x_1, x_2, x_3\}$. Reorder S; we get $S = [x_4, x_3, x_1, x_2, x_5, x_7, x_6, x_8]$. Set $j = 2$. Go to step 4. Delete x_4 from f; we get $f = x_3 + x_1x_2 + x_5x_7 + x_6x_7 + x_5x_8 + x_6x_8$. Go to step 1. The second variable of S is x_3. It is a unate variable. Go to step 3. $S' = (I(f_{x_3}) \cup I(f_{x_3'})) \setminus (I(f_{x_3}) \cap I(f_{x_3'})) = \{x_1, x_2, x_5, x_7, x_6, x_8\}$. Update S; we get $S = [x_4, x_3, x_1, x_2, x_5, x_7, x_6, x_8]$. Set $j = 3$. Delete x_3 from f; we get $f = x_1x_2 + x_5x_7 + x_6x_7 + x_5x_8 + x_6x_8$. The next variable is x_1, and $S' = \{x_2\}$. S is updated to $S = [x_4, x_3, x_1, x_2, x_5, x_7, x_6, x_8]$, and f is updated to $f = x_2 + x_5x_7 + x_6x_7 + x_5x_8 + x_6x_8$. Next variable is x_2. $S' = \{x_5, x_7, x_6, x_8\}$, $S = [x_4, x_3, x_1, x_2, x_5, x_7, x_8, x_6]$, and $f = x_5x_7 + x_6x_7 + x_5x_8 + x_6x_8$. Next variable is x_5. $S' = \{x_6\}, andS = [x_4, x_3, x_1, x_2, x_5, x_6, x_7, x_8]$. Continuing this process, we will finally obtain the ordering sequence $S = [x_4, x_3, x_1, x_2, x_5, x_6, x_7, x_8]$. The thin OBDD by this ordering is show in Figure 6.4.

6.3 Experimental Results

The experiment results show that our algorithms can really obtain some good result though the algorithm has some drawbacks. In our experiments, in cases of multiple PO circuits, the OBDD size is first determined by processing patterns of all POs together. Then the cubes of the first (and/or second) PO(s) with the highest fan-in OBDD size are processed to get the "Dominant Variable Ordering".

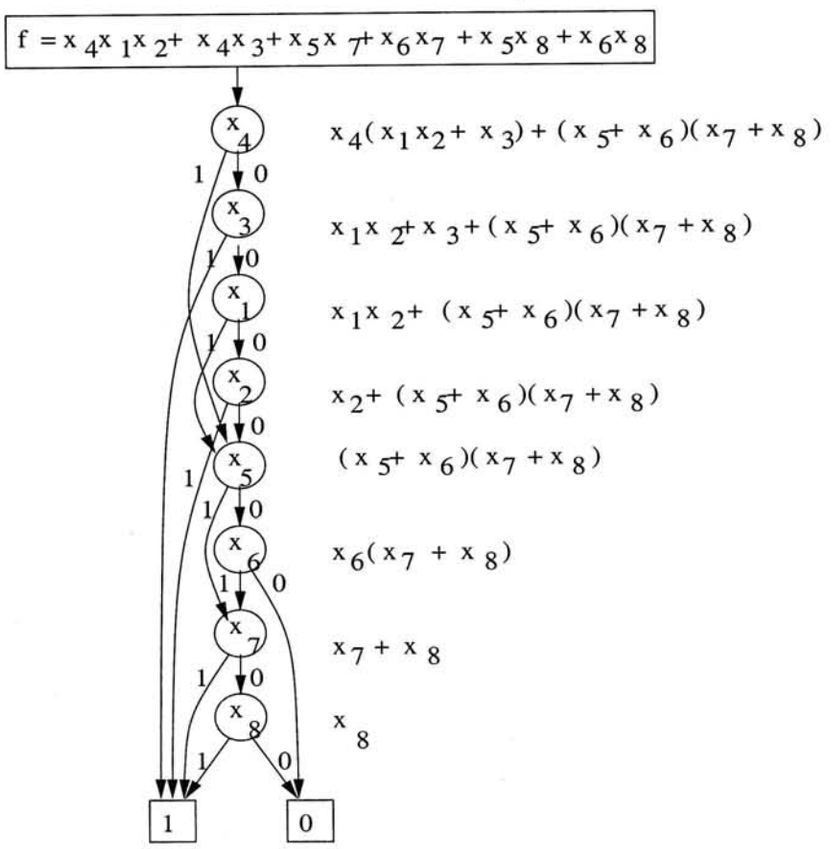


Figure 6.4: The construction of the thin OBDD of a thin factored function in the ordering given by the revised DSCF.

This ordering is applied to the entire circuit for which the size of corresponding OBDD is shown in parentheses. We pick the better result. Table 1 lists the results obtained by the MBVF, a two-level input form heuristic as compared to ours. We note that our heuristic stably achieves an average improvement of over 600%. Table 2 shows comparison between our heuristic and lexicographic ordering. Table 3 shows the results of FIH and our heuristic. The DSCF heuristic produces an average reduction of over 30% when tested extensively on 71 MCNC benchmarks. Among these 71 tested cases, only 6 resulted in an OBDD a little larger than the better one produced by the two strategies of FIH, and among them only one over 10% larger. In table 3 we show only those examples which differ at least by 10% from FIH. The best result among FIH and DSCF is shown in bold face.

Circuit name	FIH of SIS			DSCF			Our size reduction
	fixed PO	non-fix PO	CPU	v1	v2	CPU	
alu4	565	1124	1.67	536(415)	377(339)	0.92	33%
alu2	216	216	0.82	229(178)	230(162)	0.02	25%
alupla	2265	2265	0.23	1380	1629	2.75	39%
apex2	6963	6963	81.28	978	1155	2.42	86%
apex7	672	578	0.43	364	332	2.25	43%
c8	216	169	0.38	115(104)	115(104)	0.00	38%
clip	117	117	1.28	101	75	0.05	36%
count	248	248	0.18	201	201	0.50	19%
cm152a	25	25	0.05	16	16	0.00	36%
cm163a	48	48	0.10	37	36	0.03	25%
cm82a	16	16	0.03	12	12	0.02	25%
cps	1871	1871	26.98	1588	1745	0.068	15%
dalu	3973	3973	4.85	4132(1588)	4083(1497)	3.02	62%
duke2	662	565	1.02	542(492)	566(520)	0.03	13%
decod	38	38	0.07	32	32	0.00	16%
ex4	659	659	7.45	564	629	5.40	14%
e64	1760	1760	2.82	2146	2146	0.37	-22%
f51m	71	71	0.32	73(47)	74(42)	0.02	41%
frg1	309	309	0.83	194	172	0.18	44%
la1	112	113	0.28	99	92	0.15	18%
misex3	1609	1609	1.52	1241(855)	586(582)	0.15	18%
sao2	142	142	0.38	91	91	0.03	36%
t481	75	75	23.38	54	58	0.30	28%
tcon	41	41	0.08	27	27	0.02	34%
term1	369	989	0.92	219	140	0.57	62%
vda	1255	1255	2.17	525	525	0.50	58%
vg2	390	223	0.50	281(122)	281(122)	0.02	45%
z4m1	31	31	0.15	17	17	0.02	45%

Chapter 7

A Pattern Merging Algorithm

For an OBDD, each 1-path represents a disjointed implicant, i.e. there is no minterm contained in different implicants that are represented by different 1-paths. For a logic function, the number of its EPIs is the lower bound of the number of 1-path of its OBDD.

Our algorithm is still based on the pattern recognition notion. Since each 1-path of the OBDD is a disjointed implicant of the logic function, in our algorithm, we will first make the cubes disjointed to each other. Our method is to sort the implicants by length first. Suppose for a pair of implicants $x_{i_1}x_{i_2}...x_{i_n}$ and $x_{j_1}x_{j_2}...x_{j_m}, (n \leq m)$, there is no variable with different phase in these two cubes, i.e. x_i and $\overline{x_i}$ do not exist in these two cubes respectively. Then the negation of literal x_j will be added to cube $x_{j_1}x_{j_2}...x_{j_m}$ where x_j is the first variable appearing in cube $x_{i_1}x_{i_2}...x_{i_n}$ but not in cube $x_{j_1}x_{j_2}...x_{j_m}$. For example, to disjoint the cubes x_1x_2 and $x_2x_3x_4$, we will modify the $x_2x_3x_4$ into $\overline{x_1}x_2x_3x_4$.

Be aware that the function represented by the newly produced disjointed implicants may not be the same as the original one. But all the original essential points are still kept.

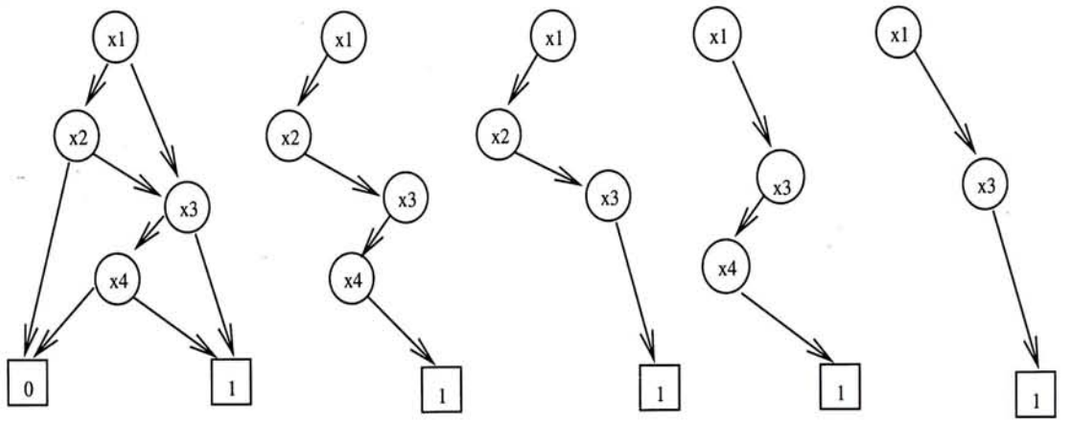


Figure 7.1: The four patterns of the function $x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4$

7.1 Merging of Patterns

The OBDD has the advantage that all the equivalent subgraphs are shared. Thus a logic function may have a simpler representation if there is a variable order that can make more nodes shared. By this way, the representation is greatly simplified and the space and time cost are reduced. So sometimes the OBDDs can represent the Boolean logic function which may be intractable in other representations.

In our algorithm, we also try to eliminate nodes as many as possible by merging the recognized patterns. The OBDDs can be treated as the merging of recognized patterns. To eliminate extra nodes, we must merge the pairs of recognized patterns close to each other so that new nodes will be created as few as possible. For example, suppose the logic function is $x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4$. If the variable order is x_1, x_2, x_3, x_4, the OBDD and the patterns are shown in Fig. 2. But if the variable order is x_1, x_3, x_2, x_4, the OBDD will have six nodes and become more complicated. The OBDD and the patterns are shown in Fig. 3.

There are several points to be noticed:

- The patterns are a partition of the onset vertices in the Boolean space. It is easy to see from the definition of the OBDD.

- There exists more than one partition of the onset vertices.

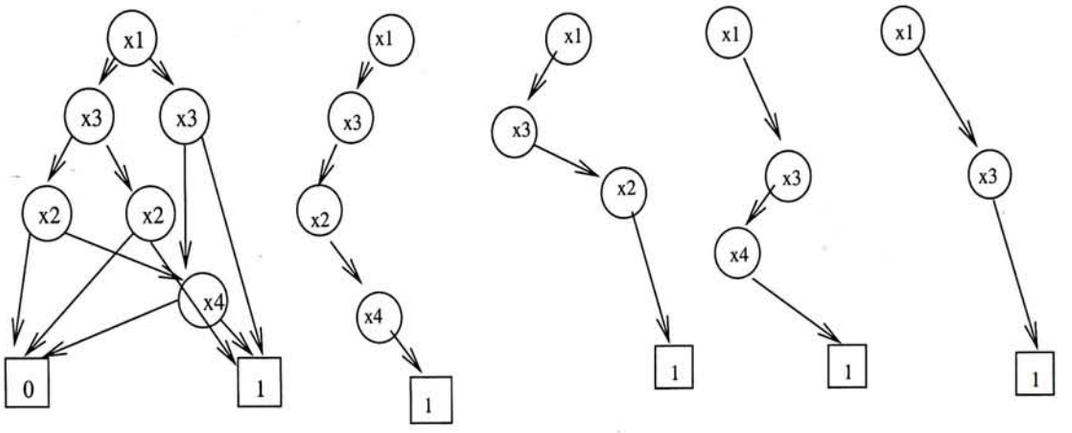


Figure 7.2: The four patterns of the function $x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4$

- Different merging of the same patterns may lead to different OBDDs. We can see that in Fig. 2 and Fig. 3, although the partition of the onset vertices are the same, the merging is different and it leads to a different OBDD.

In our algorithm, we will always try to find the closest recognized pattern to merge with the cube. Here the closest pattern is the pattern with the largest number of same literals as the cube. Another terminology is *unmerged length*, which is the number of different literals between the cube and its closest recognized pattern.

The last problem is the insertion method. In previous papers, we always append new variables to the variables that have already been picked out. But sometimes this method can not give the optimal solution.

We have observed that for an OBDD, it is essential to keep its width small because the height of OBDD is at most the number of support variables. To reduce the width of the OBDD, it is better to (1) make the two outedges point to the nodes in a level as low as possible and (2) make one child to be the offspring of another child. Consider variable x_1 and x_2, if x_1 appears in one of the f_{x_2} and f'_{x_2} while x_2 appears in both f_{x_1} and f'_{x_1}, we will think that to keep the x_2 in a higher order than that of x_1 will keep the size of OBDD small. We can see

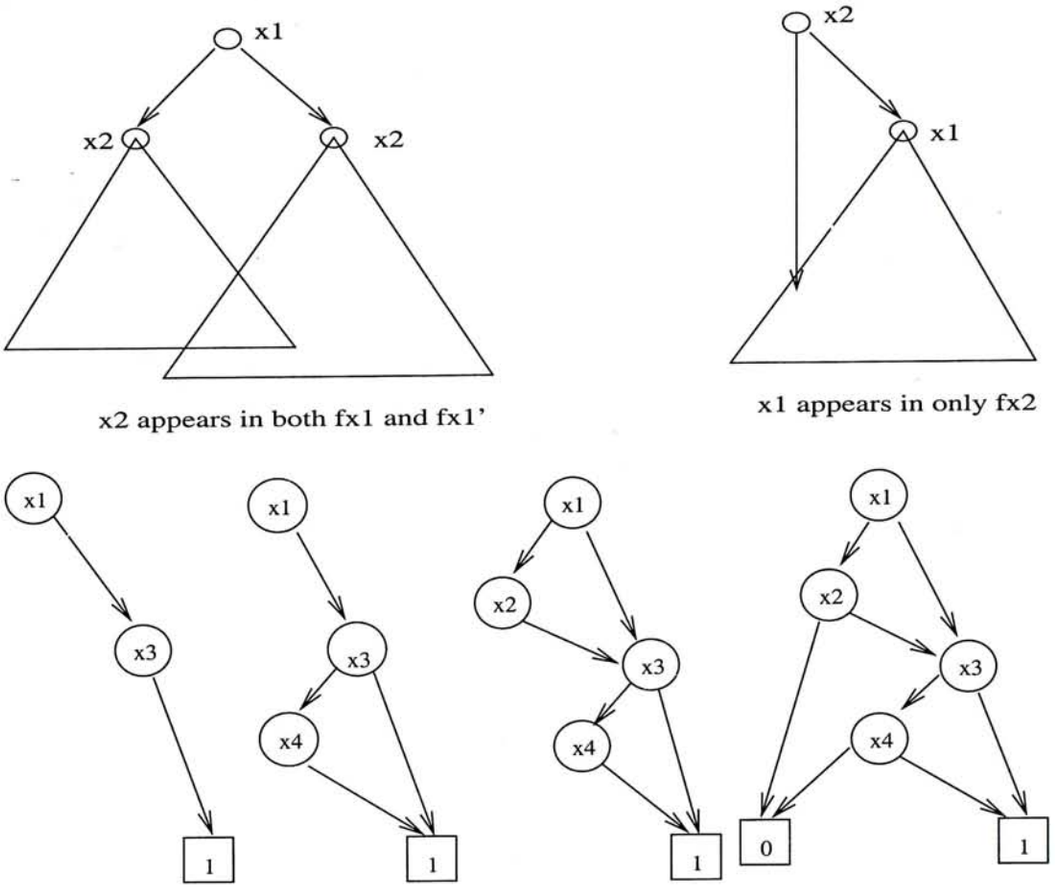


Figure 7.3: (1)the pattern of x_1x_3, (2)the merging of patterns x_1x_3 and $x_1x_3x_4$, (3) the merging of patterns x_1x_3, $x_1x_3x_4$ and $x_1x_2x_4$, (4) the final OBDD

this example in Fig. 4. So in our algorithm, normally, the new variable x will be appended to the variables that have already been picked. But in the case that for a variable y that is already picked out, if y only appears in either f_x or f'_x but not both and x appears in both f_y or f'_y, x will be inserted before y.

7.2 The Algorithm

Our algorithm can be stated as follows:

1. Express the function in the SOP form.
2. Sort the cubes according to their lengths.
3. Disjoint the cubes.
4. Resort the implicants according to their lengths. Now the length of a cube is just the unmerged length.

5. Pick the next variable to be added from the set of the cubes with shortest unmerged length. In case of a tie, pick the better one in the original logic function.

6. Check if all the variables in a cube is picked out, i.e. a new pattern is recognized. If it is, insert the new pattern into the recognized pattern set.

7. Compare the new pattern with all other patterns in the recognized pattern set, choose the pattern that is closest to the new one and merge them. According to the merging method, determine the position of the new variable.

8. Maintain the remained cubes. We can do this by comparing the new pattern with the remained cubes. If the cube is closer to the new pattern than the original one, the unmerged length of the cube is updated.

9. Back to 5 until all variables are picked.

Example:

$F = (x_1 + x_2)(x_3 + x_4)$

step 1: The function can be expressed as $x_1x_3 + x_1x_4 + x_2x_3 + x_2x_4$.

step 2: The cubes are sorted by the length. Since all the cubes have the same length. The order of the cubes is not changed.

step 3: The cubes are disjointed. First x_1x_4, x_2x_3, x_2x_4 are modified into $x_1x_3'x_4, x_1'x_2x_3$ and $x_1'x_2x_4$ respectively because they are not disjointed to x_1x_3. Then since $x_1'x_2x_3$ are not disjointed to $x_1'x_2x_4$, literal x_3' is added to $x_1'x_2x_4$. Now the cubes are: $x_1x_3, x_1x_3'x_4, x_1'x_2x_3$ and $x_1'x_2x_3'x_4$.

step 4: Since the cubes' lengths are already in the ascending order, the order is not changed after reordering.

step 5: Now that the shortest cube is x_1x_3, variable x_1 is chosen.

step 6, 7, 8 are skipped because no pattern is recognized.

step 5: x_3 is picked out.

step 6: Here a pattern x_1x_3 is recognized.

step 7: Since x_1x_3 is the first recognized pattern. No merging is needed.

step 8: We will need to update the unmerged length of unprocessed cubes. The

unmerged lengths of the unprocessed cubes $x_1x_3'x_4$, $x_1'x_2x_3$ and $x_1'x_2x_3'x_4$ are 2, 2 and 4 respectively.

step 9: Go to step 5.

step 5: $x_1'x_2x_3$ and $x_1x_3'x_4$ have the same unmerged length. Suppose we choose $x_1'x_2x_3$ as the next cube, then x_2 is picked out.

step 6: pattern $x_1'x_2x_3$ is also recognized.

step 7: Merge the new pattern $x_1'x_2x_3$ with x_1x_3. According to the cubes, we can see that: for the cubes $x_1x_3, x_1x_3'x_4, x_1'x_2x_3$ and $x_1'x_2x_3'x_4$, x_3 only appears in f_{x_2}, but x_2 appears in both f_{x_3} and f_{x_3}'. So to merge the patterns, we should insert the variable x_2 before the x_3.

step 8: The unmerged length of cube $x_1x_3'x_4$ is untouched because the number of different literals with new recognized pattern $x_1'x_2x_3$ is 3, larger than its unmerged length. But the unmerged length of cube $x_1'x_2x_3'x_4$ is changed to 2.

step 9: Go to step 5.

step 5: x_4 is chosen.

step 6: patterns $x_1x_3'x_4$ and $x_1'x_2x_3'x_4$ are recognized.

step 7: x_4 is inserted after variable x_3.

step 8: No cube is remained unprocessed.

step 9: algorithm is finished when all variables are picked.

The complexity of the algorithm is calculated as following: Suppose m is the number of cubes of the function, n is the number of the variables in the Boolean function.

In step 2, the sorting algorithm has a complexity of $O(n \ln n)$. In step 3, the disjunction algorithm has the complexity of (m^2n). In step 4, the complexity is $O(n \ln n)$.

Step 5 to 9 is a loop with at most n times. Picking the best variable has the complexity of $O(n)$. Finding the suitable patterns to merge with has a total

complexity of $O(m^2n)$. The insertion has the complexity of $O(n)$. When each time a pattern is recognized, we should also modify all the lengths of the remained cubes. This part has a total complexity of $O(m^2n)$. So the complexity of the algorithm should be $O(m^2n)$.

7.3 Experiments and Conclusion

The result of the experiment is shown in the Table 1.

Table 7.1: OBDD size from FIH, DSCF and Pattern Merge

circuit name	FIH of SIS	DSCF	Pattern Merge	Improvement over DSCF
alu4	565	536	420	21.6%
alu2	216	229	183	20.1%
alupla	2265	1380	4194	-204%
apex2	6963	1644	2187	-33.0%
apex7	672	364	667	-83.2%
c8	216	115	136	-18.3%
cm152a	25	16	16	0
cm163a	48	37	36	2.7%
cm82a	16	12	12	0
dalu	3973	4132	1924	53.4%
duke2	662	542	590	-8.9%
decod	38	32	46	-43.8%
f51m	71	73	73	0
frg1	309	194	171	11.9%
lal	112	99	104	-5.1%
misex3	1609	1241	599	51.7%
sao2	142	91	104	-14.3%
t481	75	54	56	-3.7%
tcon	41	27	41	-51.9%
term1	369	219	185	15.5%
vda	1255	525	545	-3.8%
vg2	390	281	252	10.3%
z4ml	31	17	17	0

From the table, we can see that our algorithm do have some better results for some circuits. But for many circuits, it performs worse than the DSCF algorithm.

For almost all circuits, it performs better than the FIH.

The reason that the algorithm does not perform better than the DSCF algorithm may be that the benchmark circuits we used are mostly multiple output functions and we have only considered the situation of single output functions. Our algorithm may not perform well with the multiple output functions.

Chapter 8

Conclusions

In this paper, we studied the OBDD minimization problem from two-level form representation of Boolean functions. We studied two special classes of Boolean functions, thin Boolean functions and thin factored Boolean functions. These two simple classes of Boolean functions have many good properties. Thin Boolean functions have an essential prime cube cover and their optimal OBDD can be recognized. Thin factored Boolean functions are thin Boolean functions, they can be recognized from prime cube cover representation and By taking OBDD as a pattern recognizer, we proposed a dynamic shortest cube first ordering heuristic (DSCF), which can find optimal variable order for Boolean functions expressed by disjoint cube cover. By the observation of the sharing properties in thin Boolean functions, we developed a revised DSCF ordering heuristic. This revised DSCF algorithm can find optimal variable order for thin factored functions. We implemented the algorithms and get a improvement on the Benchmark problems.

Bibliography

[1] S.B. Akers, "Binary Decision Diagrams", IEEE Trans. Comput., Vol, C-27, pp.509-516. June 1978.

[2] B. Bollig, M. Sauerhoff, D. Sieling, and I. Wegener, "Read k Times Ordered Binary Decision Diagrams-Efficient Algorithms in Presence of Null Chains", Technical Report 474, Univ. Of Dortmund, 1993.

[3] Karl S. Brace, Richard L. Rudell, Randal E. Bryant, "Efficient Implementation of a BDD Package", 27th ACM/IEEE Design Automation Conference.

[4] Robert K. Brayton, Gary D. Hachtel, Curtis T. McMullen and Alberto L. Sangiovanni-Vincentelli, "Logic minimization Algorithms for VLSI Synthesis", Kluwer Academic Publishers.

[5] R.Bryant, "Graph-Based Algorithm for Boolean Function Manipulation", IEEE Trans. Computers, vol.C-35, no.8.pp.677-691, Aug.1986.

[6] R.Bryant, "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication", IEEE Trans. Computers, vol.40,no.2,205-213, Feb.1991.

[7] K.M.Butler, D.E.Ross, R. Kapur, and M.R.Mercer, "Heuristics to Computer Variable Orderings for Efficient Manipulation of Ordered Binary Decision Diagrams". 28th ACM/IEEE Design Automation Conference. pp.417-420,1991.

[8] O. Coudert, J. C. Madre: Implicit and Incremental Computation of Primes and Essential Primes of Boolean functions. Proc. of 29th Design Automation Conference, 1992.

[9] O. Coudert, J. C. Madre, H. Fraisse: A New Viewpoint on Two Level Logic Minimization.

[10] O. Coudert: On Solving Binate Covering Problems.

[11] S. J. Friedman, K. J. Supowit, "Finding the optimal variable ordering for binary decision diagrams", IEEE Trans. Computers, Vol. 39, No. 5, pp. 710-713, may 1990.

[12] M.Fujita, Y.Matsunaga, and T. Kakuda, "On Variable Ordering of Binary Decision Diagrams for the Application of Multi-level Logic Synthesis", Proc. EDAC, pp. 50-54, Feb. 1991.

[13] M.Fujita, Y.Matsunaga, and T. Kakuda, "Interleaving Based Variable Ordering Methods for Ordered Binary Decision Diagrams", Proc. ICCAD, pp. 38-41, 1993.

[14] Michael R. Garey and David S. Johnson, "Computers and Intractability"

[15] Gary D.Hachtel and Fabio Somenzi, **Logic Synthesis and Verification Algorithms**, Kluwer Academic Publishers, Boston, 1996.

[16] N.Ishiura, H.Sawada, and S.Yajima, "Minimization of Binary Decision Diagrams Based on Exchanges of Variables", Proceedings of the IEEE International Conference on Computer-Aided Design, pp.472-475, 1991.

[17] J.Jain, J.Bitner, M.S.Abadir, J.A. Abraham, and D.F.Fussell, "Indexed BDDs: Algorithmic Advances in Techniques to Represent and Verify Boolean Functions", IEEE Trans. Computers, vol.46, no.11. pp.1230-1245.Nov.1997.

[18] S.-W.Jeong, B.Plessirer, G.D.Hachtel, and F.Somenzi, "Variable Ordering for Binary Decision Diagrams", EDAC 1992.

[19] C. Y. Lee, "Representation of switching circuits by binary-decision programs", Bell. Syst. Tech. J., vol. 38, pp.985-999, July 1959.

[20] Sharad Malik, Albert R. Wang, Robert K. Brayton, Alberto Sangiovanni-Vincentelli, "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment", ICCAD 1988.

[21] Shin-ichi Minato, "Binary Decision Diagrams and Applications for VLSI CAD".

[22] S. Minato, N. Ishiura, S. Yajima, "Shared Binary Decision Diagram with attributed Edges for efficient Boolean function manipulation, Proc, 27th ACM/IEEE Design Automation Conf., pp.52-57, June 1990.

[23] Ricardo Pezzuol Jacobi and Anne-Marie Trullemans, "Generation Prime and Irredundant Covers for Binary Decision Diagrams", 1992 IEEE.

[24] Richard Rudell, "Dynamic Variable Ordering for Ordered Binary Decision Diagrams", Proc. ICCAD, pp. 42-47, 1993.

[25] Y. L. Wu, and Malgorzata Marek-Sadowska, "Efficient Ordered Binary Decision Diagrams Minimization Based on Heuristics of Cover Pattern Processing," Proceedings of European Design Automation Conference, 1993. Paris

[26] Y. L. Wu, Hong-Bing Fan, and C. K. Wong, "On Thin Boolean Functions and Related Optimal OBDD Ordering," International Conference on Computer Design (ICCD). pp. 216-218, Oct. 1998. Austin.

[27] W.Wang, M.Abd-el-Barr, and C.McCrosky, "An Algorithm for Total Symmetric OBDD Detection", IEEE Trans. Computers, vol.46. no.6,731-733,July 1997.

Appendix

When we analyze the Boolean function, the concept of essential prime implicant is of great importance. It is very nature to think that it will also has a important role in the minimization of OBDD. This has been verified in a few points:

- In the OBDD of any Boolean function, all the variables in an EPI will appear in a 1-path[25]. That is because that in any EPI there exists at least one essential point(minterm) in it while this essential point will be contained by only one 1-path. So the implicant that the 1-path represent is contained by the EPI and the variables appear in the EPI definitely appear in that 1-path.

- In the thin Boolean function, we find out that all the primes of its two level representation are essential.

This matters indicate that to study the EPIs is important for the minimization of OBDDs. This observation inspires us to try to find out the relationship between the EPIs of an Boolean functions and its OBDD. However, we find that it is not easy to find a direct connection between the EPIs and a normal OBDD. An idea is to retrieve the EPIs of a Boolean function from its two level representation, and then apply the DSCF algorithm on these implicants to get the result.

The idea is supported by the observation before because that all the variables in the same EPI will definitely appear in one 1-path, from before we know that it is better to have all variables in the same cube close to each other in the variable order. And the browsing of the EPIs first will ensure that.

In our experiments, the EPIs are retrieved from the two level logic functions by using Dr. Olivier Coudert's program. After that the DSCF algorithm is applied on them to get the result.

The algorithm is given as following:

1. Pick out the function which has the largest number of cubes.
2. Pick out the essential primes of this function.
3. Pick out the variable by using the DSCF algorithm to parse the essential primes cover of the function.
4. Go back to 1 until all variables are picked out.

But as the following experiment result indicates, the result is not satisfactory.

Table 1: OBDD size from DSCF and EPI

Circuit name	FIH of SIS	DSCF	Parsing EPIs	Improvement over DSCF
alu4	565	377	478	-26.8%
alu2	216	230	164	28.7%
alupla	2265	1629	2628	-61,3%
apex2	6963	1155	1203	-4.2%
apex7	672	332	1650	-397%
c8	216	115	115	0%
cm163a	48	36	36	0%
cm82a	16	12	12	0%
count	248	201	200	0.5&
dalu	3973	4083	1420	65.2%
duke2	662	566	501	11.5 %
e64	1760	2146	2146	0%
frg1	309	172	371	-116%
misex3	1609	586	1500	-156%
term1	369	140	523	-274%
vda	1255	525	523	0%
vg2	390	281	334	-18.9%

It is hard to say why this method can not bring out a satisfying result because we are still lack of the information of the relationship between the EPIs and the normal OBDD. But one of the possible reason is that: the set of EPIs is sparser than the prime cover which the EPIs are retrived from. From the discussion before we know that when we pick out a variable, the Boolean space is partitioned and

the DSCF algorithm is trying to partition the Boolean space so that as many as possible minterms are within one sub-space. But after the EPIs are retrieved, only part of the minterms are kept and the partition based on the minterms in the EPIs may not lead to global optimal.

CUHK Libraries
003704425