ABSTRACT

An abstract of the dissertation of Craig Matthew Files for the Doctor of Philosophy in Electrical and Computer Engineering presented June 9, 2000.

Title: A New Functional Decomposition Method As Applied to Machine Learning and VLSI Layout

In the 1953 Ashenhurst created a multi-level logic minimization tool, called functional decomposition. Even though functional decomposition was first created to reduce the complexity of switching networks, it has been found to have many other applications, including pattern recognition, algorithm creation, data compression, machine learning and logic minimization. The difficulty is that many of the steps are exponential in complexity, limiting the size of functions which can be decomposed in a reasonable amount of time. This dissertation explores these complexity problems and presents a new functional decomposition methodology that reduces several of them.

The most important issue related to analysis, testing, and implementation of discrete functions used for logic synthesis and functional decomposition is representation. Because size complexity can hinder the runtimes of any algorithms that use them as their data structures, this dissertation uses decision diagrams for representation. There are two main reasons for using decision diagrams, ability to represent many functions with less than exponential size, and the canonical functional properties that exist when the variables in a diagram are ordered. Consequently, this dissertation explores the implementation of two types of decision diagrams and compares the two. Also presented are several new algorithms based on decision diagrams.

The functional decomposition methodology presented here is applied to VLSI circuit layout and to machine learning problems. The results show that functional decomposition is a proper methodology for solving these two very different problems.

A NEW FUNCTIONAL DECOMPOSITION METHOD AS APPLIED TO MACHINE LEARNING AND VLSI LAYOUT

by

CRAIG MATTHEW FILES

A dissertation submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY
in
ELECTRICAL AND COMPUTER ENGINEERING

Portland State University
2000

Acknowledgements

I am grateful to many people for their help while I have worked on this dissertation. First, I would like to thank my parents for their unending support and encouragement.

This dissertation would not have been possible without the mentoring, encouragement and inspiration from Dr. Marek Perkowski, my advisor at Portland State University. I feel that I have matured as a student and as a person because of the impact that Dr. Perkowski has had on my life. Through his guidance, I have accomplished many things that I never thought I would be capable of.

The presentation of this dissertation was greatly improved due to the input given by Dr. Sarah Mocas, Dr. Alan Mischenko, Ugur Kalay and my committee members: Dr. Joseph Albert, Dr. Michael Driscoll, Dr. Douglas Hall and Dr. Xiaoyu Song. I would also like to thank Dr. Auvergne and Dr. Robert, University of Montpellier II, Montpellier France, and Dr. Jozwiak, Eindhoven Technical University, Eindhoven Netherlands, for the meaningful conversations and allowing me to work at their respective universities while I was on sabbatical.

Finally, I would like to thank my godmother, Doris Lay, for her wisdom and guidance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Throughout history people have attempted to understand the “unknown” The catalyst of understanding things that were once ”unknown” is that at one point in history someone made a “discovery”. This “discovery” was then used as a basis to understand related subjects. This is the methodology of discovery where hypothesis and existing theories are used to “discover” new theories. This assumption is true in all types of learning; without understanding the basic elements of a concept, it is not possible to fully understand the concept. An analogy of learning new concepts without understanding basic concepts would be trying to teach a school child how to create/understand nuclear fusion, while the child does not know physics or chemistry.

The foundation of learning is our ability to evaluate/comprehend/understand a small part of a problem, and based on that understanding, possibly solve the entire problem. This process, called decomposition, is the process of partitioning a problem into smaller and less complex problems. The outcome is a set of sub-problems that can each be solved in a reasonable amount of time.

The notion of decomposition is used to most engineering problems. A way to solve a problem is to reduce the complexity of the problem by decomposing the problem into smaller problems. In the context of logic circuits, this is the process of decomposing a function into smaller and less complex functions. In fact, the concept of functional decomposition was first presented by Ashenhurst in 1953 as a Boolean multi-level logic minimization method [9].

The reason that decomposition has not been incorporated into most industrial logic synthesis packages is due to the complexity issues that are involved with functional decomposition [17]. Instead most industry logic synthesis packages use algebraic factorization and binary decision diagram methods. However, there has been some recent interest in functional decomposition for the synthesis of field programmable gate arrays [78, 107, 108, 109, 149, 152] and programmable logic arrays [87, 139, 140]. Also note that functional decomposition has recently been formulated for multi-valued functions [49, 86, 87, 122], real-valued functions [132] and for creating testable switching networks [149].

Decomposition has been applied to many aspects of Boolean and multi-valued logic synthesis. Several authors have expressed an interest in applying multi-valued decomposition to the field of machine learning [49, 86, 87, 122]. Each of these papers developed different decomposition methodologies applied to different types of data. The motivation in most machine learning papers that use functional decomposition is always the same: given some measure of complexity, find the solution with the minimum complexity.

The concept of decomposition in machine learning has been known since the beginnings of artificial intelligence. In 1963, Lendaris used a decomposition method applied to machine learning [80]. In 1967, Samuel used a decomposition method as an evaluation mechanism for a checker playing program [138]. Given a set of training examples, Samuel's approach learned by incrementally adjusting the examples describing the intermediate concepts. This use of decomposition performed significantly better than previous attempts of using a linear evaluation polynomial [137]. The results of Samuel in [137, 138] are fairly important because they form a foundation for machine learning by using artificial intelligence and establish the learning concepts in game playing. The first successful machine learning programs based on logic synthesis were reported in [92].

1.1 Dissertation Methodology and Scope

By developing the concepts from logic minimization, computational complexity, and machine learning, this dissertation explores functional decomposition and its use in logic synthesis, Very Large Scale Integrated (VLSI) circuit layout and machine learning. While all of these topics are disjoint, the goal of this dissertation is to show that they all are based on the reduction of complexity, which can be obtained through the use of functional decomposition.

Introduced in this dissertation is a new decomposition method that reduces several of the algorithmic complexity issues found in standard disjoint and non-disjoint Curtis-based decompositions [32]. The main hypothesis in this dissertation is that by reducing the algorithmic complexities in decomposition, the functions generated by the decomposer are still better than existing logic synthesis and machine learning methods. Results presented in this dissertation show that this new decomposition method based on multi-valued decomposition techniques does perform well for logic synthesis, circuit layout, and machine learning. To enhance this new decomposition method, different complexity measures are used which can be applied to produce a set of net-lists that are technology dependent. Normally, this is not the case because most decomposers are used as preprocessors and are generally technology independent.

To create a decomposition package that is both efficient and effective (in terms of space and time) for a variety of tasks, a data structure must be created that is fast and can represent very large functions. The data structure that is used in this dissertation is a binary/multi-valued decision diagram. Introduced here is a methodology to use a decision diagram package to represent relations. That is, functions that have different outputs for the same input can be represented by using a decision diagram package. Relations can be used in machine learning, and have been used in the encoding process during functional decomposition [122]. New decision diagram based decomposition algorithms are also discussed. The new algorithms are shown to be much better than the well known decomposition algorithms created by Lai, Pedram and Vrudhula [78].

The evaluation method of the new decomposition method presented in this dissertation is a non-optimal evaluation. That is, it is not possible to mathematically prove the "quality" of the decomposition method. Thus, to evaluate the decomposition method, comparisons to other well known logic synthesis and machine learning methods are used. The logic synthesis and machine learning methods used for comparison in this dissertation have set a basis for comparison in much of the literature. Conclusions about the new decomposition method can be derived by determining under which classes of functions the decomposition method performs "better" than the existing methods.

The selection of benchmark functions used to evaluate the new decomposition method is a set of standard benchmark functions that are used extensively in the literature. These benchmarks are Boolean/multi-valued input and output, completely/incompletely specified functions with application to logic synthesis, machine learning, data mining and knowledge discovery in databases. By giving a large set of benchmark functions, the evaluation of the new decomposition method can find characteristics in which the new decomposition performs "well" compared to existing methods. Appendix A gives more details on each of the benchmark functions and classes of benchmark functions used in this dissertation.

This dissertation explores the topic of applying functional decomposition to machine learning and to VLSI circuit layout. After the decomposition method has decomposed a function, the resulting net-list of logic blocks can be used for either machine learning or VLSI circuit layout. The flow chart of the proposed decomposition method is shown in Figure 1.1. The top three blocks are the same for all decompositions proposed in this dissertation. A benchmark function is passed to a Multi-valued Decision Diagram (MDD) package. The MDD representation of the function can then be decomposed by the functional decomposer. The functional decomposer is the new decomposition method that is proposed in this dissertation. The output of the decomposer can be used as the input to the VLSI layout synthesis and VLSI layout routing and placement tools to create a VLSI layout, or the output can be used as a learned concept of machine learning. The flow chart also illustrates the testing methodology by comparing the output of existing machine learning and logic synthesis methods to the new decomposition output for the same set of inputs. This is shown in Figure 1.1 with the dotted lines. The process of learning using functional decomposition is

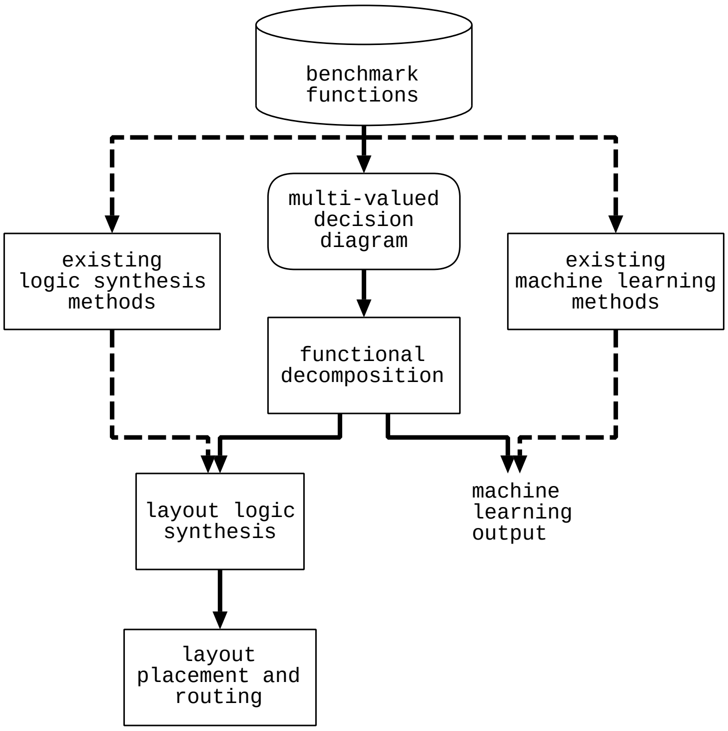


Figure 1.1: Flow chart of the proposed decomposition method.

a process of forcing "unknown" functional output values to known values. This is a process embedded in functional decomposition as new sub-functions are created. As far as the author knows, there have not been any papers written on applying functional decomposition to the layout of VLSI circuits. Functional decomposition, here, is used as the primary logic synthesis method and is used to create a net-list of functional logic blocks. Layout logic synthesis is then used to convert the net-list of blocks to a layout structure. Another tool is then used to design the physical layout.

1.2 Dissertation Overview

Chapter 2 defines basic notations that are used throughout this dissertation. These notations develop a fundamental background for logic synthesis and machine learning. The notations cover the concept of canonical forms in both Boolean and multi-valued functions.

Chapter 3 introduces the concept of a decision diagram and gives practical illustrations of decision diagrams in design and optimization problems. Binary Decision Diagrams (BDDs) are described in full detail with their extensions to MDDs. The concept of creating a MDD by using a BDD package is presented [70]. This concept is extended to represent functions that have relational outputs. A comparison of a MDD package and a MDD based on a BDD concludes the chapter.

Chapter 4 presents the fundamentals of functional decomposition as they were defined by Ashenhurst [9] and Curtis [32]. The theory of functional decomposition is quite old and can be used in many logic synthesis problems. But, the complexity of finding a decomposition is quite large, and therefore, functional decomposition has not been widely used. This chapter explains the algorithmic complexities in functional decomposition and evaluation heuristics that can be used to reduce the complexity. Decomposition based on BDDs is presented as defined by Lai, Pedram and Vrudhula [77]. Decomposition based on BDDs is extended to MDDs, as presented by the author in [49].

Chapter 5 presents a new decomposition method that reduces the complexity of functional decomposition by reducing search spaces in the exponential search problems inherent in functional decomposition. Motivation, analysis, theory and examples for this new decomposition method are presented.

Chapter 6 develops the functional decomposition algorithms that are used through the remainder of this dissertation. Two new MDD based decomposition algorithms are also presented. The advantage of these new algorithms is that they do not reorder the decision diagrams as the well known Lai, Pedram and Vrudhula algorithm does [76]. Comparisons of the new algorithms are presented.

Chapter 7 presents a new approach based on combining functional decomposition and quasi-optimal logic block design for complex gates used in VLSI layout. This chapter gives an overview of the previous research and then shows the use of functional decomposition in library free technology VLSI layout. Also presented is a section on applying functional decomposition to lattice diagrams.

Chapter 8 presents the concept of using logic synthesis for machine learning. Decomposition is shown as the logic synthesis method of choice for applications in machine learning. In some ways, the machine learning problems are the same as the logic synthesis problems. However, there are some distinct aspects in machine learning (e.g., noise, missing attributes, etc.) that are not found in logic synthesis problems. These distinct problems are presented in this chapter and several solutions are proposed.

Chapter 9 is a comparison of the methods proposed in this dissertation to existing methods used previously in machine learning and layout driven logic synthesis.

The conclusion of this dissertation is presented in Chapter 10. This chapter summarizes the work by reviewing the results discussed in Chapter 9, highlights possibilities for future research, and gives a list of all original achievements of the author.

Chapter 2

Basic Notations

This chapter defines some basic notations that are used throughout this dissertation. Notations that are specific to a single chapter are included within that chapter.

2.1 Basic Boolean Notations

A **vertex** is a combination of values of input variables. A Boolean function f is a function which maps all vertices of n-input variables to the output of f, formally, f: $\{0,1\}^n \rightarrow \{0,1\}$.

A variable, negated or not, is called a **literal** (e.g., x_1, $\overline{x_3}$). A variable available in affirmative form is called an **affirmative variable.**

Let a function f be a single-output Boolean function with the set of input variables $X = \{x_0, x_1, \ldots, x_{n-1}\}$. The number of variables in X is denoted by $|X|$. The **discriminants** of f (corresponding to the 2^n configurations of the input variables) are defined as:

$$\begin{aligned} f_0 &= f(0,0,\ldots,0) \\ f_1 &= f(0,0,\ldots,1) \\ &\vdots \\ f_{2^n-1} &= f(1,1,\ldots,1). \end{aligned}$$

Let $p_i(x_0, x_1, \ldots, x_{n-1})$ be a set of functions where each function is an elementary product (called **minterms**) consisting of all input literals:

$$\begin{aligned} p_0(x_0, x_1, \ldots, x_{n-1}) &= \overline{x_0}\ \overline{x_1} \ldots \overline{x_{n-1}} \\ p_1(x_0, x_1, \ldots, x_{n-1}) &= \overline{x_0}\ \overline{x_1} \ldots x_{n-1} \\ &\vdots \\ p_{2^n-1}(x_0, x_1, \ldots, x_{n-1}) &= x_0\ x_1 \ldots x_{n-1}. \end{aligned}$$

The product, $p_i(x_0, x_1, \ldots, x_{n-1})f_i$, is p_i given $f_i = 1$ and zero otherwise. This results in the unique algebraic representation of $f(x_0, x_1, \ldots, x_{n-1})$ using the following expansion:

$$\begin{aligned} f(x_0, x_1, \ldots, x_{n-1}) &= p_0(x_0, x_1, \ldots, x_{n-1})f_0 + p_1(x_0, x_1, \ldots, x_{n-1})f_1 \\ &\quad + \cdots + p_{2^n-1}(x_0, x_1, \ldots, x_{n-1})f_{2^n-1} \\ &= \sum_{i=0}^{2^n-1} p_i(x_0, x_1, \ldots, x_{n-1})f_i. \end{aligned} \tag{2.1}$$

Definition 2.1. *Equation 2.1 is called the* **canonical form expansion** *of* $f(x_0, x_1, \ldots, x_{n-1})$.

Equation 2.1 can be rewritten in the expanded form:

$$\begin{aligned} f(x_0, x_1, \ldots, x_{n-1}) &= (\overline{x_0}, \overline{x_1}, \ldots, \overline{x_{n-1}})f_0 + (\overline{x_0}, \overline{x_1}, \ldots, x_{n-1})f_1 \\ &+ \cdots + (x_0, x_1, \ldots, x_{n-1})f_{2^n-1}. \end{aligned} \tag{2.2}$$

By factoring Equation 2.2 by x_0, the following is found:

$$\begin{aligned} f(x_0, x_1, \ldots, x_{n-1}) &= \overline{x_0}((\overline{x_1}, \ldots, \overline{x_{n-1}})f_0 + \cdots + (x_1, \ldots, x_{n-1})f_{2^{n-1}-1}) \\ &+ x_0((\overline{x_1}, \ldots, \overline{x_{n-1}})f_{2^{n-1}} + \cdots + (x_1, \ldots, x_{n-1})f_{2^n-1}) \\ &= \overline{x_0}f(0, x_1, \ldots, x_{n-1}) + x_0 f(1, x_1, \ldots, x_{n-1}). \\ \text{Let } f_{x_0}(x_1, \ldots, x_{n-1}) &= f(1, x_1, \ldots, x_{n-1}) \\ \text{and } f_{\overline{x_0}}(x_1, \ldots, x_{n-1}) &= f(0, x_1, \ldots, x_{n-1}) \\ f(x_0, x_1, \ldots, x_{n-1}) &= \overline{x_0}f_{\overline{x_0}}(x_1, \ldots, x_{n-1}) + x_0 f_{x_0}(x_1, \ldots, x_{n-1}) \qquad (2.3) \\ &= \sum_{i=0}^{1} p_i(x_0) f_{p_i(x_0)}(x_1, \ldots, x_{n-1}). \end{aligned}$$

Definition 2.2. *Equation 2.3 is known as the* **Shannon expansion** *of* x_0 *on* $f(x_0, x_1, \ldots, x_{n-1})$ [146].

Definition 2.3. *The functions* $f_{x_0}(x_1, \ldots, x_{n-1})$ *and* $f_{\overline{x_0}}(x_1, \ldots, x_{n-1})$ *are the positive and negative* **cofactors** *of* f *with respect to* x_0.

A function $f(x_0, x_1, \ldots, x_{n-1})$ is **symmetric** (or **totally symmetric**) iff f is invariant under any permutation of its variables. A function $f(x_0, x_1, \ldots, x_{n-1})$ is **partially symmetric** in variables x_i and x_j, where $\{x_i, x_j\} \subset X$, iff $f(x_0, \ldots, x_i, x_j, \ldots, x_{n-1}) = f(x_0, \ldots, x_j, x_i, \ldots, x_{n-1})$.

A function f is **positive unate** in x_i if literal x_i appears in f and literal $\overline{x_i}$ does not appear in f. f is **negative unate** in x_i if literal $\overline{x_i}$ appears in f and literal x_i does not appear in f. A function f is **unate** if f is positive unate in all variables.

An **output don't care** designates that for a given input combination the output of the function is unspecified.

A **cube** $C = c_0 \times c_1 \times \ldots \times c_{n-1}$ can be written as a product of literals. In the case where a variable x_i in the cube can have either value 0 or 1, the literal c_i is omitted from the expression. This is called an **input don't care** because $x_i + \overline{x_i} = 1$, and therefore, x_i has no effect on output of the cube. Also note that in some instances instead of removing c_i from the expression, the symbol "-" is used to denote a variable that can have either value, 0 or 1.

A value of variable x_i is denoted by a_i. A vertex of a cube (as well as a cell of a Karnaugh map) is denoted by $a^k = (a_0^k, \ldots, a_i^k, \ldots, a_{n-1}^k)$, $k = 0, \ldots, 2^{n-1}$, where a_i^k is a value of variable x_i for a vertex a^k. An example of the vertex of a four-variable function is shown in Figure 2.1 with function $f(x_0, x_1, x_2, x_3)$. A set of one-cubes (true minterms) of a Boolean function f is denoted by $ON(f)$, a set of zero-cubes (false minterms) by $OFF(f)$ and a set of output don't cares by $DC(f)$. A function f covers cell a^k, when $a^k \sqsubseteq ON(f)$.

x_2x_3 \ x_0x_1	00	01	11	10
00	a^0	a^1	a^3	a^2
01	a^4	a^5	a^7	a^6
11	a^{12}	a^{13}	a^{15}	a^{14}
10	a^8	a^9	a^{11}	a^{10}

Figure 2.1: Vertices in a four-variable function.

2.2 Basic Multi-Valued Notations

Let a function f be a single-output multi-valued function with the set of multi-valued input variables $X = \{x_0, x_1, \ldots, x_{n-1}\}$.

A multi-valued variable x_i can take values from the set $Q_i = \{q_0, q_1, \ldots, q_{|Q_i|-1}\}$. For reference, Q_i denotes the set of values that the variable x_i can have. Each symbolic value q_k can be associated with a unique integer k, and because only integer values are considered, then $Q_i = \{0, 1, \ldots, |Q_i| - 1\}$.

A vertex for multi-valued functions is a combination of values of input variables, i.e., a point in the space $Q_0 \times Q_1 \times \ldots \times Q_{n-1}$. A multi-valued function f is a function which maps vertices in $Q_0 \times Q_1 \times \ldots \times Q_{n-1}$ to Q_f, formally, $f : Q_0 \times Q_1 \times \ldots \times Q_{n-1} \rightarrow Q_f$.

A **multi-valued literal** $x_i^{q_i}$, where $q_i \subseteq Q_i$, is defined as a literal x_i that can take all values in q_i.

A cube $c = c_0 \times c_1 \times \ldots \times c_{n-1}$ can be written as a product of multi-valued literals in the form:

$$x_0^{q_0} x_1^{q_1} \cdots x_{n-1}^{q_{n-1}}.$$

Note that if $q_i = Q_i$ and because $x_i^{Q_i} = 1$, then $x_i^{Q_i}$ is an input don't care and can be omitted from the expression of the cube. Also note that $x_i^{Q_i}$ can also be written as the symbol "-" to denote an input don't care.

The discriminants of f (corresponding to the $Q_0 \times Q_1 \times \ldots \times Q_{n-1}$ configurations of the input variables) are defined as:

$$\begin{aligned}
f_0 &= f(0, 0, \ldots, 0) \\
f_1 &= f(0, 0, \ldots, 1) \\
&\vdots \\
f_{|Q_{n-1}|-1} &= f(0, 0, \ldots, |Q_{n-1}| - 1) \\
&\vdots \\
f_{|f|-1} &= f(|Q_0| - 1, |Q_1| - 1, \ldots, |Q_{n-1}| - 1).
\end{aligned}$$

Let $p_i(x_0, x_1, \ldots, x_{n-1})$ be a set of functions where each function is an elementary product consisting of all input literals:

$$\begin{aligned}
p_0 &= x_0^0 x_1^0 \ldots x_{n-1}^0 \\
p_1 &= x_0^0 x_1^0 \ldots x_{n-1}^1 \\
&\vdots \\
p_{|Q_{n-1}|-1} &= x_0^0 x_1^0 \ldots x_{n-1}^{|Q_{n-1}|-1} \\
&\vdots \\
p_{|f|-1} &= x_0^{|Q_0|-1} x_1^{|Q_1|-1} \ldots x_{n-1}^{|Q_{n-1}|-1}.
\end{aligned}$$

By rewriting Equation 2.3 to fit the notations for multi-valued functions, the following is found:

$$\begin{aligned}
f(x_0, x_1, \ldots, x_{n-1}) &= x_0^0 f_{x_0^0}(x_1, \ldots, x_{n-1}) + x_0^1 f_{x_0^1}(x_1, \ldots, x_{n-1}) + \ldots \\
&\quad x_0^{|Q_0|-1} f_{x_0^{|Q_0|-1}}(x_1, \ldots, x_{n-1}) \qquad (2.4) \\
&= \sum_{i=0}^{|Q_0|-1} p_i(x_0) f_{p_i(x_0)}(x_1, \ldots, x_{n-1}).
\end{aligned}$$

Definition 2.4. *Equation 2.4 is known as the* **multi-valued Shannon expansion** *on $f(x_0, x_1, \ldots, x_{n-1})$ with respect to x_0.*

2.3 Graphs

A **graph** is a pair $G = (V, E)$, where E is a set of *edges* and V is a set of *vertices* (nodes). A **directed graph** is a graph where each edge, $e \in E$ and $e = (u, v)$, denotes a direction from the vertex u to the vertex v, given that $u, v \in V$. An **undirected graph** is a graph where each edge, $e = \{u, v\}$, does not denote a direction u to v.

A **path** is a sequence of connected edges. A **cycle** is a closed path (the first and last vertices in the path sequence are the same). A **directed acyclic graph** is defined as a directed graph $G = (V, E)$ that has no cycles.

An **incompatibility graph** is defined as a graph where every edge denotes an incompatibility between two vertices. An example of an incompatibility graph can be found in the *graph coloring problem*. The **graph coloring problem** is a problem of assigning colors to nodes in a graph in such a way that every pair of vertices that share a common edge have different colors [29, 58, 127]. Thus, the edges in the graph denote an incompatibility or non-shared-color-ability between two vertices. The **minimum graph coloring** is the smallest possible coloring of the graph, i.e., the coloring with the minimum number of colors. Example 2.1 illustrates the notion of graph coloring.

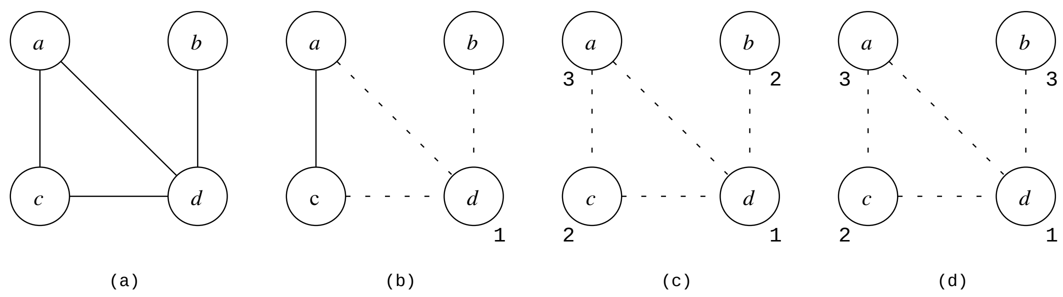


Figure 2.2: Graph coloring example.

Example 2.1. *Figure 2.2a shows an incompatibility graph; in Figure 2.2b vertex d is colored '1'. Edges in the graph are "dotted" to signify that vertices connected* *to d cannot have color '1'. Figure 2.2c gives one possible complete coloring of the graph by coloring vertices b and c with color '2', which implies that a is colored with '3'. Figure 2.2d gives another possible coloring where vertices a and b have color '3'. The minimum graph coloring in this example is three colors.*

A **subgraph** of a graph $G = (V, E)$ is another graph $G' = (V', E')$, where $V' \subseteq V$ and $E' \subseteq E$. If $G = (V, E)$ is undirected, a **maximal completely connected subgraph** or **clique** $G' = (V', E')$ is a subgraph of G, where $\forall u \in V'$, $\forall v \in V'$, and $\{u, v\} \in E'$. A clique can overlap with any other clique and each vertex in the clique must be a complete graph (all vertices have edges to all other vertices). A **maximal clique** is the largest clique (complete sub-graph) that exists in the given graph G. Example 2.2 illustrates the notion of maximal clique.

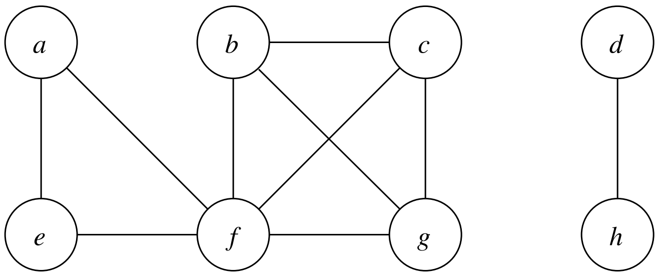


Figure 2.3: Maximal clique example.

Example 2.2. *Given the graph in Figure 2.3, there are several cliques in the graph:* $\{d, h\}$, $\{a, e, f\}$ *and* $\{b, c, f, g\}$. *The maximal clique is* $\{b, c, f, g\}$.

2.4 Basic Machine Learning Notations

The goal in machine learning is to find a description of a *concept* given a set of *examples*. A **concept** is defined as the simplified description in certain representations of a *function*. A **function** in this case, may be fairly simple (e.g., addition or subtraction), complex (e.g., predicting winning moves in chess), or abstract (e.g., a description of a car, a description of the universe).

Formally, an **example** is a pair $(x, f(x))$, where x is the input (**attribute**) and $f(x)$ is the output of the function applied on x. **Inductive learning** [93, 125] creates a general description of past examples and then applies this description to new data. This is a task that, given a set of examples of f, returns a function g that approximates f. The function g is called the **hypothesis** function.

In the case that all inputs and outputs of a function are fully explored, the function is said to be **completely specified**. Otherwise, the function is said to be **incompletely specified**, and typically in machine learning, because the number of examples of a function is much smaller than the possible number of examples, these functions are said to be **strongly incompletely specified** functions.

In logic synthesis, an output don't care designates that for a given input combination the output value of the function is unspecified, and, thus, can be any value. This is helpful in logic synthesis because the output don't cares can be used to reduce the logic when constructing the function. This is much like the situation in machine learning except that the output values of the function that are not known are called **output don't knows**. In machine learning, the goal is to determine the value of each don't know. In fact, to determine the performance of a machine learning algorithm the concept of *error* is used. **Error** is defined as forcing a don't know to an incorrect value in the resulting network. To evaluate a machine learning algorithm two functions are used, a test function (the original function) and a training function which is a subset of the examples that make up the test function. The machine learning algorithm is run on the training function and the amount of error is found by comparing the resulting network to the test function. The goal is to approximate f by g with small complexity (Occam's Razor [12, 15]) and with as little error as possible. Thus, a don't know has the same properties as a don't care, but forcing a don't know to a value must be done, in such a way, that the amount of error of the resulting network is reduced.

2.5 Algorithmic Complexity

This section gives the notations used in this dissertation related to **asymptotic** runtimes of an algorithm. Asymptotic runtimes of an algorithm reflect the rate of growth on the amount of time used by the algorithm in terms of the length of the input. The notations used here describe the asymptotic runtimes of an algorithm as defined in terms of functions whose domains are the set of natural numbers.

The notation used throughout this dissertation is that of **asymptotic upper bound**, or O-notation.

Definition 2.5. *A function $f(n)$ is in the set $O(g(n))$ iff there exists positive constants c and n_0, such that*

$$f(n) \leq c \cdot g(n), \forall n \geq n_0.$$

From Definition 2.5, the value of the function $f(n)$ is less than or equal to $c \cdot g(n)$ for all values of n larger or equal to n_0. Consequently, O-notation is used to refer to the upper bound complexity/runtime of a function/algorithm.

The following is a set of definitions used to define NP-complete. This is a very simplistic definition of NP and NP-complete languages; see Hopcroft and Ullman [63] for a more detailed presentation of NP.

A **complexity class P** is the set of concrete decision problems that are solvable by a deterministic algorithm (e.g., a Turing machine) in polynomial time. A function $f : \{0,1\}^n \rightarrow \{0,1\}^{nc}$, where c is a constant, is **polynomial-time computable** if there exists a polynomial-time algorithm A that, given any input $x \in \{0,1\}^*$, produces the output $f(x)$. An alphabet α is a finite set of symbols. A language L over α is any set of strings made up of symbols from α.

Definition 2.6. *Given strings $x, y \in \{0,1\}^*$, a language $L \subseteq \{0,1\}^*$ is* **verified** *by an algorithm A if there exists at least one string y that proves that string $x \in L$.*

The **complexity class NP** is the class of languages that can be verified by a deterministic polynomial-time algorithm, so that the length of the "proof" string y is polynomial in the length of x. That is, algorithm A verifies language L in polynomial time. NP stands for nondeterministic polynomial time.

Definition 2.7. *A language $L \subseteq \{0,1\}^*$ is* **NP-complete** *if*

1. *$L \in NP$, and*

2. *$L' \leq_p L$ ($L' \leq_p L$ denotes a polynomial-time computable function f, such that, given $x \in L'$, implies $f(x) \in L$) for every $L' \in NP$.*

If a language L satisfies item 2, but not necessarily item 1, then L is **NP-hard**.

A problem (language) is NP-hard if an algorithm for solving it can be translated into an algorithm for solving any known NP (nondeterministic polynomial time) problems (languages). A problem (language) is NP-complete if it is in NP (solvable in nondeterministic polynomial-time computable) and NP-Hard.

An example of the difference between NP-complete and NP-hard is graph coloring. The problem of coloring a graph with $k > 2$ colors is known to be NP-complete. In comparison, there is a deterministic polynomial-time algorithm for determining if a graph can be colored with two colors. Further, the problem of finding a best coloring (minimal number of colors) is only known to be NP-hard. (It is NP-hard because it cannot determine if the solution is the minimal solution in polynomial-time.) Given this, deterministic algorithms for solving anything greater than two-coloring usually use exponential time at best.

Chapter 3

Decision Diagrams

One of the most important issues related to analysis, testing, and implementation of discrete functions is representation. This is very important because the complexity (time and space) of operations depend on the type of data structures used. General representations used to describe a discrete function include truth tables, equations, cube arrays, and tabular representations. The problem with these descriptions is that their sizes grow exponentially with the number of input variables. Because of their size complexity, the general representations listed above can hinder the runtimes of any algorithms that use them as their data structures.

The notion of a Binary Decision Diagram (BDD) was introduced by Lee and Akers for representing Boolean functions [6, 79]. Fortune [55] realized the full potential of a BDD by ordering the input variables in a BDD to make it a canonical representation. Bryant [21] implemented these restrictions, creating an Ordered Binary Decision Diagram (OBDD). As a result, testing of functional properties, such as satisfiability or equivalence, can be done with OBDDs.

An advantage of an Ordered Decision Diagram (ODD) is that problems from many domains can be expressed entirely in terms of an ODD. This advantage makes it unnecessary to construct the problem space in terms of any other structure, e.g., truth tables, state-transition graphs, or search trees. Another advantage is that by using the same restrictions found in [55], any type of ODD has a canonical form, including Multi-valued Decision Diagrams (MDDs). The fact that ODDs are a compact canonical representation of functions makes them a very good representation for many applications, including functional decomposition. It has been shown by Lai, Pedram and Vrudhula [77] that OBDDs can be used to decompose very large functions that are beyond the capacity of decomposers using other representations.

This chapter presents the basic notations for BDDs and MDDs and presents a new way to represent relations using a decision diagram package. The chapter concludes by comparing two different MDD representations on strongly unspecified multi-valued functions.

3.1 Binary Decision Diagrams

It is well known that an n-input Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$ can be represented by an OBDD [21], i.e., a directed acyclic graph where a Shannon expansion is carried out in each vertex.

Definition 3.1. *A* **Decision Diagram** (DD) *over a set of variables X and a non-empty terminal set T is a connected, directed acyclic graph $G = (V, E)$ with the following properties:*

- *a vertex $v_i \in V$ is either a non-terminal or a terminal vertex,*
- *each* **non-terminal vertex** *v_i represents a variable $x_i \in X$, where v_i has exactly $|Q_i|$* **successors** *in V, where $|Q_i|$ is the number of cofactors of x_i,*
- *each* **terminal vertex** *v_i has no successors and is labeled with a value $t_i \in T$ where $T = Q_f$, where Q_f is the set of output values of a function f,*
- *a* **root vertex** *is the top vertex in G, i.e., no vertex in the graph has a root vertex as a successor.*

Definition 3.2. *Types of decision diagrams are defined with the following properties:*

1. *a decision diagram G is* **complete** *(also known as a* **decision tree***), if each variable is encountered exactly once on each path in G from the root vertex to a terminal vertex,*

a	b	Y
0	0	0
0	1	1
1	0	1
1	1	1

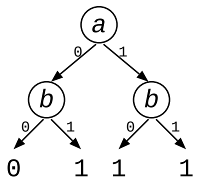


Figure 3.1: Truth table and corresponding binary decision tree.

2. *a decision diagram G is* **free**, *if each variable is encountered no more than once on each path in G from the root vertex to a terminal vertex,*

3. *a decision diagram G is* **ordered**, *if it is free and each variable is encountered in the same order on each path in G from the root vertex to a terminal vertex.*

In general, OBDDs have graphs that are much smaller (in terms of number of vertices) than the worst case of the exponential size of a binary decision tree. Integer multiplication seems to be one of the few practical functions that require an exponential size OBDD for any variable ordering. This occurs because exclusive-or functions are widely used in the multiplication routines. But, there has been ample empirical evidence indicating that many Boolean functions encountered in real applications can be represented as BDDs with a size much smaller than exponential [22]. In the remainder of this dissertation, all DDs considered are ODDs.

3.2 Multi-Valued Decision Diagrams

Obviously, BDDs can be extended to represent a function $f : \{0,1\}^n \rightarrow \{0,..,Q_{f-1}\}$ (the functional output of f is Q_f-valued) and the resulting graphs are denoted as Multi-Terminal BDDs (MTBDDs). The operations on MTBDDs can be carried out as efficiently as in the case of two terminals [30].

It is straightforward to extend MTBDDs to MDDs [148], representing functions in the form $f : \{0, .., Q_{f-1}\}^n \rightarrow \{0, .., Q_{f-1}\}$. For this, each internal vertex has Q_f outgoing edges (instead of two edges in a BDD). In [148], it was shown that the efficient operations known for BDDs can also be carried out on MDDs using a *case operator* (multi-valued Shannon expansion operator, discussed in Subsection 3.5.2).

To expand the definition of a MDD, each variable x_i has value Q_i, such that f may be denoted as $f : Q_0 \times Q_1 \times \ldots \times Q_{n-1} \rightarrow Q_f$. The extension or addition of don't cares to a MDD is fairly straightforward by representing the function as $f : Q_0 \times Q_1 \times \ldots \times Q_{n-1} \rightarrow Q_f + 1$. The (Q_f)-th value is used to represent an output don't care of a function.

There are several reasons for using MDDs:

1. most machine learning and data mining problems are multi-valued or real-valued, not Boolean [94],

2. in binary decomposition, multi-valued values can be used during the encoding process to avoid placing extra don't cares in the function (see Chapter 5). It is possible to make the entire decomposition process multi-valued,

3. unrelated to the decomposition approach presented in this dissertation, state machines can be represented by MDDs [18, 69, 148].

Publications on MDDs include (in chronological order): Srinivasan, Kan, Malik and Brayton [148]; Miller [98]; Becker and Drechsler [13]; Drechsler [38]; Miller and Drechsler [99]; Kam, Villa, Brayton, and Sangiovanni-Vincentelli [70].

3.3 Representation of Relations

A relation $\Re$ over a set of multi-valued variables with output values in the domain $Q_\Re$ is a mapping of the input variable domains into the domain of subsets of the output values. Formally, $\Re(x_0, x_1, \ldots, x_{n-1}) : Q_0 \times Q_1 \times \ldots \times Q_{n-1} \rightarrow Q'_\Re$ where $Q'_\Re \subseteq Q_\Re$. In the instance that $Q'_\Re = Q_\Re$, the output of the relation is an output don't care (because the given input maps to all possible output values of the relation).

The importance of representing relations is that many benchmark functions used in machine learning have different output values given the same input value. To solve these types of machine learning problems it is necessary to have a data structure that can represent relations. Relations have also been used in the encoding process during decomposition [122]. As far as the author knows, the data structures used for relations in the literature are cubes and other structures, not DDs. The following is a new representation of relations based on decision diagrams:

Given a Q_f-valued function $f(x_0, x_1, \ldots, x_{n-1})$ to represent f in a DD the set of non-empty terminal vertices T must represent all values in Q_f. To represent a relation $\Re$, the set T must represent all values of $Q_\Re$ plus all possible configurations (subsets) of $Q_\Re$. For instance, given $Q_\Re = \{0, 1, 2\}$ then to represent all possible configurations of $Q_\Re$, T must be the set $\{0, 1, 2, 01, 02, 12, 012\}$. Because the size of T is $2^{|Q_\Re|} - 1$ for $|Q_\Re| > 3$ constructing a DD with all possible relations as terminal vertices is not practical.

Given a $Q_\Re$-valued relation $\Re$ over the set of variables X, each of the $Q_\Re$ values can be encoded and represented by a BDD. To encode $Q_\Re$, a set of variables TQ must have $\lceil log_2 |Q_\Re| \rceil$ elements, where each $tq_i \in TQ$ is a variable not in set X. This results in a BDD that can represent a relation given the following definition:

Definition 3.3. *A* **relation DD** ***(RDD)*** *over a set of variables X, a set of encoding variables and a non-empty terminal set T is a connected, directed acyclic graph $G = (V, E)$ with the following properties:*

- *a vertex $v_i \in V$ is either a non-terminal or a terminal vertex,*
- *if a non-terminal vertex v_i represents a variable $x_i \in X$, then v_i has exactly $|Q_i|$ successors in V, given that $|Q_i|$ is the number of cofactors of x_i,*
- *if a non-terminal vertex v_i represents a variable $tq_i \in TQ$, then v_i has*

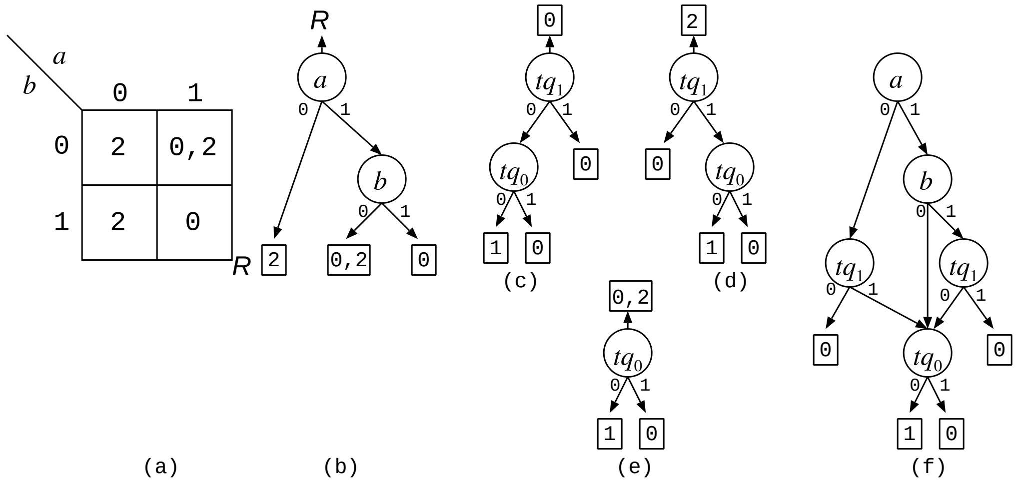


Figure 3.2: Missing attributes using relations (a) two values for the minterm $a\overline{b}$ (b) DD of the function (c) output value '0' represented as a RDD (d) RDD of output value '2' (e) RDD of {0,2} (f) RDD of function.

exactly two successors in V,

- *each terminal vertex v_i has no successors and is labeled with a value $t_i \in T$ where $T = \{0, 1\}$.*

Example 3.1. *Given the relation $\Re$ in Figure 3.2a, the associated DD with relation terminal vertices is shown in Figure 3.2b. The output values of the function are shown in the square boxes. Given that $TQ = \{tq_1, tq_0\}$ (T has two variables to represent the three output values of the function), and tq_0 is the least significant bit, then terminal value '0' is encoded as $\{0, 0\}$. The RDD representation of output value '0' is shown in Figure 3.2c, where the only path to the terminal vertex-1 is through both 0-successors of tq_0 and tq_1. Terminal value '2' is encoded as $\{0, 1\}$, and is shown in Figure 3.2d. To construct the relation {0,2}, the RDDs representing '0' and '2' are ORed together to get the RDD as shown in Figure 3.2e. The RDD of the relation $\Re$ is shown in Figure 3.2c.*

3.4 Building a MDD Package

This section discusses the implementation issues of creating a MDD. In general, a MDD is developed by creating vertices in the graph that contain all the information about the variable that the vertex represents and the edges to each of its successors. There are two basic ways to create a MDD package: The first MDD defines each MDD vertex as a vertex that contains the information about the multi-valued variable that it represents and a vector of the successors of the vertex. This representation is called a Multi-valued Vector Decision Diagram (MVDD). The second MDD representation uses a BDD package for the underlying data structure of a MDD (MBDD) [70]. A MBDD is created by encoding the successors of each MDD vertex to a BDD structure. An advantage of a MBDD package is that it uses an existing BDD package. In addition, it was reported in [148] that MBDDs have faster runtimes than MVDDs. There are several publications on implementing a MBDD package [18, 69] with the constraint that only functions with Boolean outputs are represented. The rest of this chapter discusses the implementation issues of creating both types of MDDs and compares these two approaches.

Any number of values can be referenced or encoded into a binary string. So for any given multi-valued vertex, a small binary diagram can be used to represent that vertex. Figure 3.3a shows a MVDD vertex of a four-valued variable and Figure 3.3b shows the corresponding MBDD vertex. The successors $\{0, 1, 2, 3\}$ of the MVDD vertex are encoded as {00, 01, 10, 11} in the MBDD representation, where variables a_0 and a_1 are used to represent the encoding of the variable a. The encoding used throughout this dissertation is a natural code to binary conversion.

A disadvantage of a MVDD package compared to a MBDD package is that it is difficult to find redundant vertices in the graph as the number of successors of each vertex gets larger. Because the number of redundant nodes is smaller for a MVDD package, the number of vertices in the graph can be much larger than the number of vertices in the graph produced by the MBDD package [49]. In

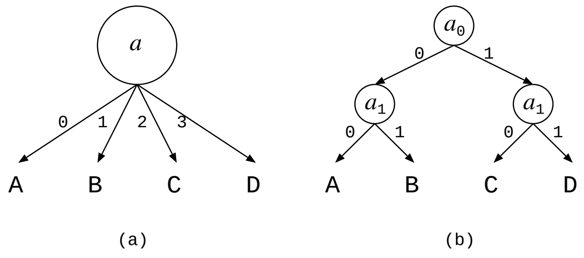


Figure 3.3: (a) Four-valued vertex (b) corresponding BDD.

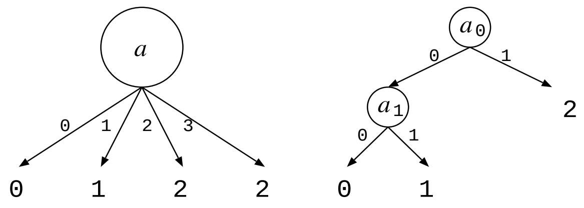


Figure 3.4: Basic comparison of MVDD and MBDD.

the case that a multi-valued vertex is represented by a small BDD, it is possible that the vertices within the BDD can be redundant, and thus, shared.

Example 3.2. *In Figure 3.4, the MVDD vertex and the MBDD actually have the same number of pointers, i.e. they both have four pointers. Notice that in Figure 3.3a,b the MVDD vertex has four pointers and the MBDD has six pointers (two pointers from each of the vertices shown). For a four-valued variable, the MBDD can only be* $6 \div 4$ *or 150% larger than the MVDD representation of a function (this is used later in the chapter to show inconsistencies in already published results).*

However, statistically, as the number of cofactors for a given variable grows, the probability of finding redundant MVDD vertices decreases. Because each vertex in a MBDD has two successors, the possibility of finding redundant vertices stays the same.

Observation 3.1. *Given two vertices in a MBDD, the first vertex has successors* $\{a, b\}$*, the second vertex has successors* $\{c, d\}$*. The probability that one of the vertices is redundant is* $p(a = c) * p(b = d)$*. If* $\{a, b, c, d\} \in \{0, 1\}$*, then* $p(a = c) = \frac{1}{2}$*,* $p(b = d) = \frac{1}{2}$*, and* $p(a = c) * p(b = d) = 0.25$*.*

Observation 3.2. *Given two vertices in a MVDD, if the vertices have* $|Q|$ *successors and* $m = T$ *terminal vertices, then the probability of combining the vertices is* $(\frac{1}{m})^{|Q|}$*.*

From Observation 3.2, the probability of combining two vertices is significantly small as n increases. For example, given $|Q|$=8 and m=2, the probability of combining two vertices is 0.004, a very small probability compared to 0.25 when $|Q|$=2 and m=2. Also note that as the MDD increases in size, m is no longer the number of terminal vertices, but the number of possible vertices that exist below the current vertex in the decision diagram. So for vertices near the top of the MDD as compared to the bottom of the MDD, m is much larger.

From these observations a MBDD can give a smaller representation of a function than a MVDD because of the probabilities of combining vertices.

Example 3.3. *Figure 3.5 illustrates a case when a MBDD can have a smaller representation than a MVDD because of the probabilities of combining vertices. The figure has three MVDD vertices that do not have the same successors and are not redundant. The MBDD representation of the three MVDD vertices has a redundant vertex, which is equivalent to the right-most MVDD vertex. In this figure, the MVDD has twelve edges, while the MBDD only has six edges.*

3.5 Implementation Issues

This section explores some of the implementation details of building a DD, for a more comprehensive analysis and more details see the paper by Bryant [22].

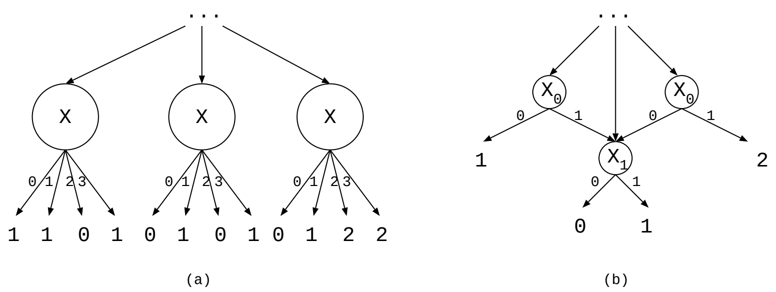


Figure 3.5: Combining vertices (a) MVDD (b) MBDD.

3.5.1 Basic Vertex

The most basic element in a DD is a vertex, which may be declared as follows:

```
struct vertex {
  variable id
  reference count
  mark
  successors
}
```

The *variable id* is an integer value that is unique to the variable that the vertex represents. The *variable id* is an element in an ordered set of integers from zero to the number of variables in the decision diagram. The *variable id* corresponds to the location of the variable in the diagram, where the variable at the top of the diagram has *variable id*: zero and each successive *variable id* is increased by one. The *reference count* is the number of times this vertex is used as a root vertex (top vertex of function declared by the user). The *reference count* is used by *garbage collection* routines to remove unused vertices from the package by analyzing which vertices are in use and which vertices can be deleted. *mark* is used for traversal routines. By setting a *mark*, it implies that this vertex has already been traversed by the traversal routine and does not need to be traversed again. The last item, *successors* denotes the outgoing edges from the vertex. For a BDD or a MBDD the *successors* are the 0- and 1-cofactors of a variable. For a MVDD the *successors* are a vector array that represents all cofactors of a variable.

The most basic procedure in a MDD package creates and returns a vertex:

$$CREATE\_VERTEX(v, G_0, G_1, \ldots, G_{n-1}).$$

This procedure creates a vertex with *variable id* v and with *successors* $\{G_0, G_1, \ldots, G_{n-1}\}$. This is done by using a *hash table* called the **Unique Table**, in which every vertex in the MDD is stored and no two vertices representing the same variable have the same cofactors. If a vertex already exists in the Unique Table then CREATE_VERTEX returns a pointer to that vertex, otherwise CREATE_VERTEX creates a new vertex, puts it in the Unique Table, and returns the pointer to that vertex.

The amount of time that it takes to run the recursive procedures is greatly reduced by using a *Computed Table*. A **Computed Table** is a hash table that stores a vector of input parameters and the returned value of the procedure. Any data computed earlier can be put into the hash table, thus the name Computed Table. After the first time the procedure is called with a set of input parameters, the procedure does not have to calculate the return value, the corresponding value from the computed table is used instead.

3.5.2 CASE Operator

One of the most important operations of a DD is the ITE (if-then-else) or CASE (used in MVDD or MBDD) operator. The CASE operator is discussed here because it is an extension of the ITE operator. The case procedure:

$$CASE(F, G_0, G_1, \ldots, G_{n-1})$$

returns a new function where all terminal vertices v_i are replaced with G_i. The output of this procedure is represented by the following:

$$CASE(F, G_0, G_1, \ldots, G_{n-1}) = \sum_{i=0}^{n-1} (F = i) \ return \ G_i.$$

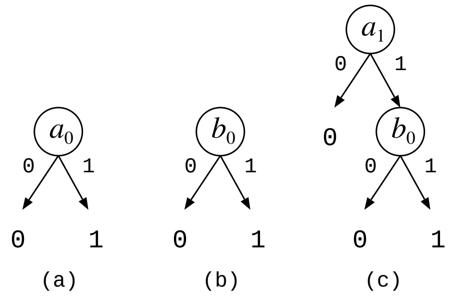


Figure 3.6: CASE example of Boolean AND (a) a-vertex (b) b-vertex (c) $a \cdot b$.

Example 3.4. *To illustrate the CASE operation, the Boolean AND operation ($a \cdot b$) of the variables a and b is used. The F input parameter is a vertex representing the positive literal of variable of a, shown in Figure 3.6a as the vertex a_0. Because this is a Boolean operation, there are two G parameters: G_0 and G_1. The G_0 input parameter is a terminal vertex-0, and G_1 is a vertex representing the positive literal of variable b, shown in Figure 3.6b as the vertex b_0. While running the CASE procedure, the 0-successor of a_0 is not changed (0 replaced with 0) and the 1-successor of a_0 is changed from a terminal vertex-1 to the b_0 vertex. A new vertex, a_1 is created that represents the function $a \cdot b$ (a terminal vertex-1 is only reached by traversing the positive cofactors of a and b), see Figure 3.6c.*

For the algorithms presented in this chapter, given a vertex v, and a variable y that has $|Q_y|$ cofactors, the following notation is used in the algorithms presented in this chapter:

$$\text{let } v^{y_i} = \begin{cases} \text{the } i^{th} \text{ successor of } v & \text{when } v \text{ represents variable } y,\ 0 \le i < |Q_y|, \\ v & \text{otherwise.} \end{cases}$$

The basic pseudo code for the CASE operation is shown in Algorithm 3.1.

Algorithm 3.1 CASE operation pseudo code.

CASE (vertices: $F, G_0, G_1, \ldots, G_{n-1}$)
  if F is a terminal vertex with value i
    return G_i
  if Computed Table has entry $(F, G_0, G_1, \ldots, G_{n-1})$
    return value from Computed Table
  let $z \in \{F, G_0, G_1, \ldots, G_{n-1}\}$ be the vertex with the smallest variable id
  let y be the variable represented by z
  let $|Q_y|$ be the number of cofactors of the variable y
  for $i = 0$ to $(|Q_y| - 1)$
    $H_i = \text{CASE}(F^{y_i}, G_0^{y_i}, G_1^{y_i}, \ldots, G_{n-1}^{y_i})$
  $r = \text{CREATE\_VERTEX}(y, H_0, \ldots, H_{|Q_y|-1})$
  place r and $(F, G_0, G_1, \ldots, G_{n-1})$ in Computed Table
  return r.

3.5.3 Variable Swap

Another major operation of any DD package is the ability to change the variable ordering of two adjacent variables. This is important because the size of a DD is dependent on the variable ordering, and by changing the variable ordering the diagram size can be greatly reduced. This concept of changing the ordering of adjacent variables to reduce the size of a DD is called *sifting* [136]. In some functional decomposition algorithms based on DDs, the high quality variable reordering procedures are very important (see Section 4.5 for more details).

In a BDD package, two variables can be exchanged by reordering the variables by using Shannon expansion. For example, given the following function with variable ordering x_0, x_1, and x_2:

$$\begin{aligned} f(x_0, x_1, x_2) &= x_0(x_1 f(1,1,x_2) \vee \overline{x_1} f(1,0,x_2)) \vee \\ &\quad \overline{x_0}(x_1 f(0,1,x_2) \vee \overline{x_1} f(0,0,x_2)) \end{aligned}$$

Now after reordering of x_0 and x_1 the function becomes:

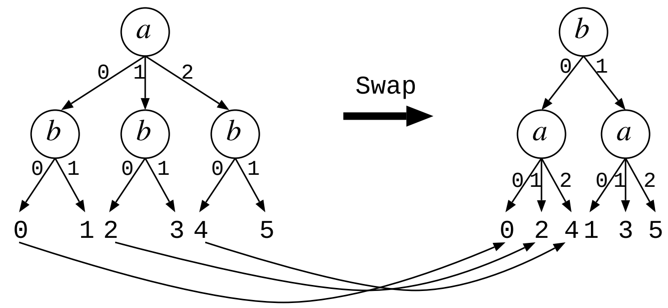


Figure 3.7: Variable swapping.

$$f(x_1, x_0, x_2) = x_1(x_0 f(1,1,x_2) \vee \overline{x_0} f(0,1,x_2)) \vee \\ \overline{x_1}(x_0 f(1,0,x_2) \vee \overline{x_0} f(0,0,x_2))$$

Notice that before and after reordering the variables in the function, the first term $x_1 x_0 f(1,1,x_2)$, and the last term $\overline{x_1}\ \overline{x_0} f(0,0,x_2)$, in both equations are the same. Because the middle two terms are the only terms that changed, the variable reordering is performed by creating new vertices that represent the two middle terms.

In a MDD package, the reordering is done by using multi-valued Shannon expansion and is executed in a very similar way as in a BDD package. In the case of a MBDD, all the variables that represent a single MDD variable must be swapped with all the variables that represent the adjacent MDD variable.

Example 3.5. *Figure 3.7 illustrates variable reordering of a three-valued variable and a two-valued variable in a MDD package. The figure shows that all three 0-successors of variable b (before the swap) become the successors of the left-most vertex a after the swap.*

Variable swapping is a time-intensive problem because all vertices in the DD package that represent the two variables being swapped must go through the transformation presented above. In the case of swapping two m-valued MDD variables in a MBDD, there are $\lceil log_2 m \rceil^2$ variable swaps. Note that although MBDDs have more redundancies, and thus, less vertices for a given variable, it does not imply that the swapping of variables in the MBDD package is slower than the swapping of variables in the MDD package (see results in Section 3.6).

3.5.4 Diagram Traversal

The objective of diagram traversal is to traverse (only once) all vertices that define a given function(s) and perform some *action* on each vertex. An **action** is defined as any procedural or algorithmic evaluation of the current vertex. An example of an action is counting the number of vertices in a given function; during the traversal, the action performed on each vertex is to increment a counter by one. The most general form of traversal is a *depth first* traversal of a diagram. Given the root vertex of a function, traverse all vertices under that root vertex, starting with the left-most path from the root vertex to a terminal vertex. Perform an action on each vertex and do not traverse vertices that have already been traversed. Algorithm 3.2 presents the pseudo code for traversal.

Algorithm 3.2 Traversal pseudo code.

```
TRAVERSAL(vertex v)
  if v has already been traversed
    return
  perform action
  let c be the number of successors of v
  let v^i denote the i-th successor of v
  for i = 0 to (c − 1)
    TRAVERSAL(v^i)
  return.
```

3.5.5 Don't Care Algorithms

This subsection presents topics that, as far as author knows, have not been published before. These topics are presented because of their importance for using a MDD package for functional decomposition.

Tautology is the determination of functional equivalence. Because a DD is canonical, two functions are tautologically equivalent if the root vertices of both functions are equal. The problem of checking if two functions are tautologically equivalent has a $O(1)$ complexity. This is a great advantage when DDs are used in logic synthesis for completely specified functions. For unspecified functions, tautology is still used, but it is important to see if two functional graphs can be combined by forcing don't care values to care values in order to verify the tautology. Algorithm 3.3 is the pseudo code for finding the don't care tautology.

Algorithm 3.3 don't care tautology pseudo code.

```
X_TAUTOLOGY(vertices : v0, v1)
  if v0 and v1 are equal
    return TRUE
  if v0 and v1 are terminal vertices then
    if v0 is not a don't care and v1 is not a don't care then
      return FALSE
    else return TRUE
  if Computed Table has entry (v0, v1)
    return value from Computed Table
  let z ∈ {v0, v1} be the vertex with the smallest variable id
  let y be the variable represented by z
  let |Qy| be the number of cofactors of the variable denoted by y
  for i = 0 to (|Qy| − 1)
    if X_TAUTOLOGY(v0^yi, v1^yi) is FALSE then
      place (v0, v1) and FALSE in Computed Table
      return FALSE
  place (v0, v1) and TRUE in Computed Table
  return TRUE.
```

After determining that two graphs have a don't care tautology, an algorithm is needed to combine the two graphs. Algorithm 3.4 is the pseudo code to combine two graphs.

Decomposition algorithms (presented in the next two chapters) require many tautological checks. In the case that the function to decompose is incompletely specified, Algorithms 3.3 and 3.4 are frequently used.

Algorithm 3.4 COMBINE pseudo code.

X_COMBINE(vertices: v_0, v_1)
  if v_0 and v_1 are equal
    return v_0
  if v_0 and v_1 are terminal vertices then
    if v_0 is not a don't care return v_0
    else return v_1
  if Computed Table has entry (v_0, v_1)
    return value from Computed Table
  let $z \in \{v_0, v_1\}$ be the vertex with the smallest variable id
  let y be the variable represented by vertex z
  let $|Q_y|$ be the number of cofactors of the variable denoted by y
  for $i = 0$ to $(|Q_y| - 1)$
    H_i = X_COMBINE($v_0^{y_i}, v_1^{y_i}$)
  r = CREATE_VERTEX($y, H_0, \ldots, H_{|Q_y|-1}$)
  place (v_0, v_1) and r in Computed Table
  return r

3.6 Decision Diagram Experimental Results

A goal of this dissertation is to derive a "good" mechanism for the decomposition of Boolean and multi-valued functions. The mechanism should be able to represent completely specified Boolean functions for layout and general logic synthesis, and should also represent highly unspecified multi-valued functions for machine learning. This section presents some comparisons of a MVDD package and a MBDD package. In the hope to fully understand issues related to efficient processing of decision diagrams the author has written three different DD packages: BDD, MBDD, and MVDD. The experimental results presented here are a comparison of these three packages. Because the benchmark functions used for comparison are not relations, the MBDD experimental results are equivalent to the experimental results of a RDD.

Table 3.1 is a comparison between the MVDD package developed by Miller [99] and the MBDD package. Each of the benchmark functions is a binary function that Miller encoded to four-valued vertices for his MVDD package (two binary input variables are made into one four-valued variable). The results show

Function	in	out	MVDD nodes	BDD nodes	Pointer%
9sym	9	1	18	25	69%
alu2	10	8	48	134	140%
alu4	14	8	510	1197	117%
bw	5	28	66	108	82%
duke2	22	29	711	973	68%
mdiv7	8	10	78	183	117%
misex1	8	7	39	41	53%
misex2	25	18	128	136	53%
misex3	14	14	351	1301	185%
postal	8	1	11	25	113%
rd53	5	3	14	17	61%
rd73	7	3	20	31	78%
rd84	8	4	22	42	95%
sao2	10	4	60	155	129%
74181	14	8	431	858	100%
vg2	25	8	685	1044	76%
total			3192	6447	100%

Table 3.1: Comparison of Miller's MVDD (results taken from [99]) and the MBDD package created by the author.

the number of vertices needed to represent the given function. The column labeled Pointer% is the number of BDD vertices times two, divided by the number of MVDD vertices times four (this column was not in the table presented in [99]).

Table 3.1 shows that for four-valued MVDD vertices, the BDD has about the same memory requirements as the MVDD. Some of the results Miller reported in Table 3.1 are surprising. In particular, for the benchmark *misex3*, the table shows that the MBDD is larger than the MVDD package by 185%. From Example 3.2, for a four-valued variable, the MBDD can only be 150% larger than the MVDD on functions with four-valued inputs. Pointer percentage is not a great estimator of memory requirements used by each DD package, but is used here because it shows the erroneous values in the table. Table 3.1 is shown in this dissertation because of Miller's influence in multi-valued logic, and that this table is the only published table, that the author is aware of, that presents results of a MDD on truly multi-valued functions. The table is also presented to show that Miller's results are inconsistent, and thus, cannot be used. Because there are no other published results of MDDs on multi-valued functions, the results presented here are comparisons of DD packages developed at Portland State University.

Table 3.2 compares the memory sizes of a MVDD package to a MBDD packages on multi-valued benchmark functions. Each of the benchmark functions has multi-valued inputs and most of them have multi-valued outputs. To compare memory sizes, each BDD vertex has a memory size defined by the following: It takes one (32 bit) integer to store the variable id, reference count and the mark variables, one integer for each successor, and one integer to hold the pointer of each vertex (held in the Unique Table). The memory size of each BDD vertex is four. The only difference in a MVDD vertex and a BDD vertex memory size is the number of successors. The size of a MVDD vertex is three plus the number of successors. In Table 3.2, the second column shows the number of multi-valued inputs, the third: the number of BDD variables needed to encode the multi-valued inputs, the fourth: the number of BDD vertices, the fifth: the number of multi-valued vertices, the sixth: the size of the BDD, the seventh: the size of the MVDD, and the eighth: the percentage of the BDD size divided by the MVDD size. The benchmarks are taken from three different benchmark sets; UCI repository of machine learning databases [25], multi-valued logic benchmark functions [47], and a set of multi-valued functions where the origin and the type of the functions are unknown [47]. (See Appendix A for more details on each set of benchmark functions.)

Table 3.2 shows that the MBDD is roughly the same size as the MVDD, given all the functions. Notice that the benchmark *letter* skews the overall results of the table because of its size. In the case that the benchmark *letter* is not used in the evaluation, the overall size for the MVDD is smaller (overall) than the MBDD. But, the table shows that the MBDD is slightly smaller for the larger functions, and is not much larger than the MVDD on small functions (130% larger in the worst case).

Table 3.3 compares the traversal speeds of the MBDD and the MVDD packages over the same set of benchmarks. The table shows the runtime (in seconds) that it takes to perform a traversal. The three columns are labeled as follows: BDD: the runtime traversing all BDD vertices in the MBDD, MBDD: the runtime traversing all equivalent MDD vertices in the MBDD, MVDD: the runtime to traverse all MVDD vertices in the MVDD package. The table shows that the BDD and MVDD traversal runtimes are about the same, and are much faster than the MBDD traversal routine. In fact, the traversal speeds of the MVDD and BDD are faster than the MBDD, in all cases. For large benchmarks, such as *letter*, the MBDD runtimes are an order of magnitude higher than the MVDD runtimes.

Table 3.4 compares the time to change the order of the variables so that the bottom variable of a function is moved to the top of the DD. Evaluation of all benchmark functions shows that the runtimes for the MBDD are much faster than the MVDD for truly multi-valued functions. The overall runtimes show that the MBDD runtimes are an order of magnitude slower than the MVDD, but the overall results are skewed because of the large benchmark *letter*. Without this benchmark, the two packages are comparable on the average. This result is different than the experimental results of Miller [99] where he reported that his MVDD package is typically two or three times slower than an optimized BDD package. The only difference in comparisons is that Miller used a set of binary functions for his benchmark testing. The functions were changed so that two binary input variables were encoded to one four-valued variable. A possible explanation of the speed difference is that the functions were made from binary functions and because they were small-valued; the experimental results were skewed towards the BDD package.

Table 3.5 compares the Computed Table sizes and the runtimes of the X_TAUTOLOGY algorithm. To make the comparison, the X_TAUTOLOGY algorithm checks the tautology for all successors of the root vertex of each benchmark function. The Computed Tables are broken up into three columns which are size, hits, attempts. Size: the number of items stored in the Computed Table. Attempts: the number of times the Computed Table was called. Hits: the number of times that the computed value had the value that was requested. In almost all the functions, the MBDD Computed Table size is larger than the MVDD Computed Table size. The number of Computed Table hits varies from function to function. In the case of the large benchmark functions, the number of MBDD Computed Table hits is smaller for most functions than the number of MVDD Computed Table hits. The number of attempts is generally smaller for the MBDD. The number of Computed Table attempts gives the number of times the X_TAUTOLOGY procedure is recursively called. The number of recursive calls is a relative measurement of the runtime of the X_TAUTOLOGY algorithm. As shown in the table, the X_TAUTOLOGY runtime is almost always faster for the MBDD. In fact, in the three cases that the MVDD is faster, the difference is not significant.

3.7 Decision Diagram Conclusions

This chapter presented the implementation issues of creating a MDD package. In terms of small valued functions, the MVDD package and the MBDD package are about the same in terms of memory requirements. In contrast, the MBDD package has less memory requirements the the MVDD package on very large multi-valued functions (such as the benchmark *letter*). Variable reordering was shown to be an intensive process for MBDDs. Because the presented results do not clearly define which package is the best, given a particular application of the MDD, one should decide which package works the best for the application. If traversal is used more often than other algorithms then the MVDD is probably the best representation. If variable swapping is extensively used then the MBDD package should be preferred over the MVDD package. It was also shown that the size of the MVDD versus the MBDD is quite unpredictable. If memory is of great concern it may be useful to try both packages on a function to see which package results in the smaller representation. Chapter 9 presents more comparisons of the MBDD and MVDD packages when applied to functional decomposition.

An attempt was made to create a table that is similar to Table 3.5 to compute the Computed Table values while running the CASE function and reading benchmark functions. To produce comparable results, garbage collection was turned off. Remember that garbage collection (Subsection 3.5.1) is used to remove any vertices that are no longer needed to represent the given functions in the DD. The functional representation of each benchmark function is a set of cubes that represent the function. To input each benchmark function into a MDD, each cube must be logically ORed with the current DD representation of the function. Because the number of cubes is quite large (e.g., there are 20,000 cubes in the benchmark *letter*), and the DD representation of the function is changed each time a new cube is added, many different DD representations are created. When attempting to evaluate the CASE function, the memory requirements exceeded 120 MBytes, therefore comparisons could not be made.

This chapter described the details of creating two types of MDD packages. One advantage of using a MDD is that it can hold strongly unspecified multi-valued functions used in machine learning. A second advantage is that MDDs are a strong canonical form from which many logic synthesis algorithms may be derived. Given the advantages of a MDD, it is a practical data structure for functional decomposition applications. A new extension to MDDs was also presented that allows the representation of relations. The use of a RDD package allows for decomposition of many machine learning benchmark functions and has possible application to the encoding problem in functional decomposition. The next chapter extends the discussion of decision diagrams by presenting a section on using decision diagrams as a data structure for functional decomposition. Chapter 9 presents further comparisons of the MDD packages when applied to functional decomposition.

Function	in	MBDD in	MBDD nodes	MVDD nodes	MBDD size	MVDD size	Size %
audiology	69	80	7039	6668	28156	34021	82%
breastc	9	36	4093	1119	16372	14547	112%
bridges1	9	16	359	195	1140	1137	100%
bridges2	10	18	503	262	1576	1537	102%
chess1	6	16	7820	3091	31280	33981	92%
chess2	36	37	8802	8446	34900	42538	82%
connect-4	42	84	82639	40724	273252	244344	111%
flag	28	57	6651	3557	26284	25854	101%
house-votes	16	16	407	407	1628	2035	80%
letter	16	64	318883	77004	1275532	1463076	87%
lung-cancer	56	112	2953	1472	11812	10304	114%
programm	12	24	33317	16419	115496	104737	110%
sensory	11	19	1853	1074	6992	6541	106%
sleep	9	31	933	238	3328	3143	105%
sponge	44	86	3472	1745	13888	11987	115%
tic-tac-toe	9	18	779	338	2400	2028	118%
trains	32	51	314	193	1256	1247	100%
alet	18	72	21967	5316	79500	69108	115%
d4	14	29	486	219	1872	1543	121%
d7	24	61	1123	416	4284	3647	117%
d8	32	80	1527	588	5800	4869	119%
d9	34	84	1616	629	6156	5162	119%
d10	37	89	1720	688	6572	5554	118%
geo	11	32	3163	831	11556	8879	130%
let	18	72	21910	5304	79296	68952	115%
u1	60	153	22552	9839	90208	73631	122%
u1_4	60	91	329	237	1316	1319	99%
u1_5	60	98	437	295	1748	1701	102%
u1_10	60	129	1106	571	4424	3773	117%
u2	60	144	21344	10085	85376	71369	119%
u3	60	151	22363	9831	89452	71898	124%
u4	60	144	21492	9989	85968	70693	121%
u5	60	143	21779	10064	87116	71157	122%
total			645,731	227,854	2,485,936	2,536,120	98%
w/o letter			326,848	150,850	1,210,404	1,073,044	113%

Table 3.2: MVDD and MBDD size comparisons.

Function	in	MBDD nodes	MVDD nodes	BDD (sec)	MBDD (sec)	MVDD (sec)
audiology	69	7039	6668	1.00	4.00	2.00
breastc	9	4093	1119	0.01	3.00	0.01
bridges1	9	359	212	0.01	0.01	0.01
bridges2	10	503	288	0.01	0.01	0.01
chess1	6	7820	3091	1.00	7.00	0.01
chess2	36	8802	8446	2.01	7.00	2.00
connect-4	42	82639	40724	10.0	59.0	9.00
flag	28	6651	3557	0.01	4.01	1.00
house-votes	16	407	407	0.01	0.00	0.01
letter	16	318883	77004	47.0	312.0	21.0
lung-cancer	56	2953	1472	0.01	2.00	0.01
programm	12	33317	16420	4.01	23.0	3.00
sensory	11	1853	1074	0.01	1.00	0.01
sleep	9	933	238	0.01	0.01	0.01
sponge	44	3472	1745	0.01	2.01	0.01
tic-tac-toe	9	779	338	0.01	0.01	0.01
trains	32	314	193	0.01	0.01	0.01
alet	18	21967	5316	2.00	19.0	1.00
d4	14	486	219	0.01	0.01	0.01
d7	24	1123	416	0.01	0.01	0.01
d8	32	1527	588	0.01	1.00	0.01
d9	34	1616	629	0.01	1.00	0.01
d10	37	1720	688	0.01	2.00	0.01
geo	11	3163	831	0.01	2.00	0.01
let	18	21910	5304	2.01	19.0	1.00
u1	60	22552	9839	3.00	17.0	2.00
u1_4	60	329	237	0.01	0.01	0.01
u1_5	60	437	295	0.01	0.01	1.00
u1_10	60	1106	571	0.01	0.01	0.01
u2	60	21344	10085	3.00	17.0	5.00
u3	60	22363	9831	3.00	17.0	2.00
u4	60	21492	9989	3.00	17.0	2.00
u5	60	21779	10064	3.00	17.0	2.00
total		645,731	227,898	84.23	553.1	54.19
total w/o letter		326,848	144,226	36.23	237.1	31.19

Table 3.3: Comparison of MVDD and MBDD traversal speeds.

Function	in	MBDD nodes	MVDD nodes	MBDD (sec)	MVDD (sec)
audiology	69	7039	6668	5.0016	5.0021
breastc	9	4093	1119	1.0037	0.0067
bridges1	9	359	212	0.0010	0.0006
bridges2	10	503	288	0.0013	0.0008
chess1	6	7820	3091	2.0014	2.0033
chess2	36	8802	8446	3.0081	4.0053
connect-4	42	82639	40724	40.007	52.010
flag	28	6651	3557	2.0089	1.0063
house-votes	16	407	407	0.0013	0.0014
letter	16	318883	77004	158.01	5564.0
lung-cancer	56	2953	1472	3.0004	1.0060
programm	12	33317	16420	8.0007	22.004
sensory	11	1853	1074	0.0038	0.0033
sleep	9	933	238	0.0030	0.0011
sponge	44	3472	1745	2.0036	1.0033
tic-tac-toe	9	779	338	0.0022	0.0013
alet	18	21967	5316	8.0070	8.0041
d4	14	486	219	0.0020	0.0009
d7	24	1123	416	0.0088	0.0032
d8	32	1527	588	1.0040	0.0053
d9	34	1616	629	1.0053	0.0059
d10	37	1720	688	1.0074	0.0069
geo	11	3163	831	0.0067	0.0036
let	18	21910	5304	8.0038	7.0095
u1	60	22552	9839	15.007	9.0036
u1_4	60	329	237	1.0054	1.0006
u1_5	60	437	295	1.0071	1.0008
u1_10	60	1106	571	2.0060	1.0024
u2	60	21344	10085	13.010	8.0098
u3	60	22363	9831	14.009	9.0020
u4	60	21492	9989	14.002	8.0074
u5	60	21779	10064	14.010	8.0097
total				317.16	5712.1
total w/o letter				154.15	143.13

Table 3.4: Comparison of MVDD and MBDD swap speeds.

filename	MVDD Computed Table size	hits	atts	MBDD Computed Table size	hits	atts	MVDD (sec)	MBDD (sec)
audiology	68	0	69	1111	45	1958	0.0000	0.0006
breastc	1813	1922	11205	5180	1142	8301	2.0071	0.0034
bridges1	144	27	284	146	16	213	0.0003	0.0000
bridges2	188	49	394	251	54	385	0.0004	0.0001
chess1	33	1	57	96	4	111	0.0001	0.0000
chess2	140	9	201	624	18	954	0.0002	0.0004
connect-4	54607	26236	154560	91453	23853	176245	16.009	6.0001
flag	490	115	1657	939	67	1396	0.0037	0.0006
flare1	279	98	551	320	55	469	0.0007	0.0001
flare2	237	72	416	313	47	420	0.0005	0.0001
hayes	237	72	416	313	47	420	0.0005	0.0001
house-votes	14	0	18	9	0	11	0.0000	0.0000
letter	30455	11159	195240	67337	4025	97290	125.01	46.009
lung-cancer	12	2	18	13	2	18	0.0000	0.0000
sleep	201	757	1304	501	227	796	0.0037	0.0002
sponge	66	12	133	105	6	148	0.0002	0.0000
tic-tac-toe	631	801	1656	1100	696	1961	0.0018	0.0006
alet	2431	1587	14525	6896	881	11137	4.0062	1.0076
d4	78	8	134	101	5	124	0.0002	0.0000
d7	19	0	29	26	0	31	0.0000	0.0000
d8	59	8	125	94	2	128	0.0002	0.0000
d9	66	7	119	87	4	106	0.0002	0.0000
d10	78	8	134	101	5	124	0.0002	0.0000
geo	211	15	613	391	5	509	0.0020	0.0002
let	2608	1754	16295	7448	1011	12235	5.0007	2.0052
u1	454	101	1073	762	61	1009	0.0025	0.0004
u1_4	14	6	28	20	3	28	0.0000	0.0000
u1_5	18	6	32	25	2	32	0.0000	0.0000
u1_10	29	15	73	46	7	62	0.0001	0.0000
u2	645	135	1371	1036	73	1376	0.0024	0.0005
u3	403	60	877	639	33	868	0.0018	0.0003
u4	587	157	1326	961	88	1344	0.0023	0.0005
u5	576	131	1207	989	72	1338	0.0025	0.0005
total	113365	58945	446813	189476	32557	321612	152.06	55.031

Table 3.5: Comparison of MVDD and MBDD computed tables.

Chapter 4

Functional Decomposition

This chapter provides a review of the fundamentals of functional decomposition as defined by Ashenhurst and Curtis. The end of this chapter explores the concept of functional decomposition based on decision diagrams.

While many papers were written about the topic of functional decomposition [9, 74, 124, 144], there was no comprehensive approach until Ashenhurst defined the basic properties and presented a unified theory of functional decomposition in [10, 11]. Curtis used the theorems of Ashenhurst to develop a generalized form of decomposition in [32, 33]. Note that all decomposition forms developed by Ashenhurst can be treated as special cases of Curtis' decompositions. But, there have been many other proposed types of functional decomposition since the advent of Curtis' decomposition. However, there are only three distinct types of decompositions in the literature that make use of the concept of partitioning input variables. These three methods are Curtis' decomposition [33], Steinbach's decomposition [149], and Perkowski's decomposition [117, 120]. Although these three methods are distinct, they are still based on the fundamentals of Ashenhurst's and Curtis' decomposition (such as the concept of variable partitioning and finding certain patterns in rows and columns in a decomposition table). In fact, the formulation of Steinbach decomposition was actually proposed by Ashenhurst and developed by Bochmann [16], but Steinbach came up with the theory for finding partitions for this type of decomposition. The Perkowski decomposition [117, 120] combines multi-level multiplexor design with variable partitioning/column minimization (the total number of non-trivial inputs to multiplexors should be minimized) techniques typical for Curtis' decomposition. The decompositions of Ashenhurst [10], Luba [87], Lai and Pedram [76, 77, 78] and many others are just special cases or different realizations of the Curtis' decomposition. Consequently, the fundamentals of Ashenhurst's and Curtis' decompositions provide essential insight into a wide range of functional decomposition techniques.

4.1 General Decomposition

The decomposition of a function can be an expression of the function in terms of a composition of other functions. For example, if

$$f(x_0, x_1, x_2, x_3) = F(\Phi(x_0, x_1), x_2, x_3)$$

then the term on the right is a decomposed function that is equivalent in behavior to the original function f. The above example is called *simple* because there is only one composite function Φ in the term to the right. This can be extended into *complex* decompositions, in which more than one composite function is used. Examples of complex structures are the following:

1. $f(A, B) = F(\Phi(A), \Upsilon(B))$,
2. $f(A, B, C) = F(\Phi(A), \Upsilon(B), C)$,
3. $f(A, B, C) = F(\Phi(\Upsilon(A), B), C)$.

In general, an n-input, single-output Boolean function, f: $\{0,1\}^n \rightarrow \{0,1\}$, has the set of input variables $X = \{x_0, x_1, \ldots, x_{n-1}\}$. The number of variables in set X is denoted by $|X|$.

Definition 4.1. *Let $A \subset X$ and $B \subset X$, where $A \neq \emptyset$ and $B \neq \emptyset$. A* **partition**, *denoted as $A|B$, exists if $A \cap B = \emptyset$ and $A \cup B = X$.*

Definition 4.2. *A function $f(x_0, x_1, \ldots, x_{n-1})$ has an* **Ashenhurst's simple disjunctive decomposition** [10], *iff f can be represented by $f = F(\Phi(B), A)$.*

$$
\begin{array}{cc|cccc|}
 & & & B & & \\
 & & 0 & 1 & \cdots & 2^{|B|}-1 \\
\cline{3-6}
 & 0 & f_0 & f_1 & \cdots & f_{2^{|B|}-1} \\
A & 1 & f_{2^{|B|}} & f_{2^{|B|}+1} & \cdots & f_{2^{|B|+1}-1} \\
 & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & 2^{|A|}-1 & f_{(2^{|A|}-1)\times 2^{|B|}} & f_{(2^{|A|}-1)\times 2^{|B|}+1} & \cdots & f_{2^{|A|\times|B|}-1} \\
\cline{3-6}
\end{array}
$$

Figure 4.1: Partition matrix for binary functions.

This is known as partitioning the input variables into the **bound set** *B and the* **free set** *A. The variables in B and A are known as the* **bound variables** *and* **free variables**, *respectively.*

Definition 4.3. *For the partition $A|B$ on X, a* **partition matrix** *representation of $f(X)$ is defined as a rectangular array of the 2^n functional values of f, arranged in $2^{|A|}$ rows and $2^{|B|}$ columns.*

A general partition $A|B$ is shown in Figure 4.1, the rows correspond to the $2^{|A|}$ configurations of A in order, and the columns correspond to the $2^{|B|}$ configurations of B in order.

A partition matrix is defined as a canonical form by factoring the variables in B from Expression 2.1, much like Expression 2.4 to find the following:

$$
\begin{aligned}
f(x_0, x_1, \ldots, x_{n-1}) &= \sum_{i=0}^{2^{|A|}-1} p_i(A) f_{p_i(A)}(A, B) \\
&= \sum_{i=0}^{2^{|B|}-1} p_i(B) f_{p_i(B)}(A, B). \qquad (4.1)
\end{aligned}
$$

Definition 4.4. *The number of distinct column vectors in a partition matrix is called the* **column multiplicity** *of the partition and is denoted by ν. The number of distinct row vectors in a partition matrix is called the* **row multiplicity** *of the partition and is denoted by μ.*

Definition 4.5. *Two columns in a partition matrix are* **equal** *if for every row, the output value of both columns are equal, i.e., the columns are non-distinct).*

Theorem 4.1. *A function* $f(x_0, x_1, \ldots, x_{n-1})$ *has an* **Ashenhurst's simple disjunctive decomposition***, denoted by* $f = F(\Phi(B), A)$ *iff the partition matrix* $A|B$ *has column multiplicity* $\nu = 2$*.*

Proof. The proof is broken into two parts: the first part proves that if the column multiplicity $\nu = 2$, then $f = F(\Phi(B), A)$; the second part proves that if $f = F(\Phi(B), A)$, then $\nu = 2$.

1. If the partition matrix has a column multiplicity $\nu = 2$, then there are only two distinct columns in the partition. Given that the first column is labeled Φ and the second column is labeled $\overline{\Phi}$, the two columns are denoted as $F(\Phi, A)$ and $F(\overline{\Phi}, A)$, which gives the form $f = F(\Phi(B), A)$.

2. Assuming that $f(x_0, x_1, \ldots, x_{n-1}) = F(\Phi(B), A)$ then by using Shannon expansion, f can be expressed as:

$$f(x_0, x_1, \ldots, x_{n-1}) = \sum_{i=0}^{2^{|A|}-1} p_i(A) f_{p_i(A)}(B) \tag{4.2}$$

$$= \sum_{i=0}^{2^{|A|}-1} p_i(A) F_{p_i(A)}(\Phi(B)). \tag{4.3}$$

   Because this expansion is unique, f_i can only be $\Phi(B)$, $\overline{\Phi(B)}$, 0, or 1.

□

The block diagram of decomposition using the notations from Definition 4.2 is shown in Figure 4.2. Obviously, Theorem 4.1 can be expanded for partitions that have larger column multiplicities.

Definition 4.6. *A Boolean function* $f(x_0, x_1, \ldots, x_{n-1})$ *has a* **Curtis' disjunctive decomposition** [31]*, denoted by the composite function:*

$$f = F(\Phi_0(B), \Phi_1(B), \ldots, \Phi_{k-1}(B), A)$$

iff the partition matrix $A|B$ *has column multiplicity* $\nu \leq 2^k$*, where* $k < |B|$*.*

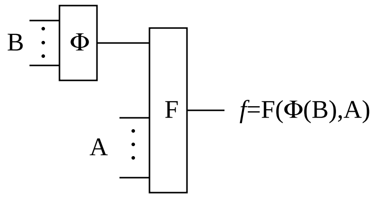


Figure 4.2: Basic functional decomposition block diagram with free set A and bound set B.

B={b,c,d}

A={a}	a \ bcd	000	001	010	011	100	101	110	111
	0	1	0	0	0	0	0	1	1
	1	0	0	1	1	1	0	0	0
column labels		0	1	2	2	2	1	0	0

Figure 4.3: Partition matrix $\{a\}|\{b,c,d\}$ with column labels identifying unique columns.

Example 4.1. *Consider the Boolean function* $f(a,b,c,d) = \sum(0,6,7,10,11,12)$. *The partition* $\{a\}|\{b, c,\ d\}$ *is shown in Figure 4.3. There are three unique columns in the partition matrix, i.e, the column multiplicity* $\nu(a|bcd)$*=3. Because* $\nu = 3$*, and the number of* Φ *composite functions is* $k = \lceil log_2\nu\rceil = \lceil log_2 3\rceil = 2$*,* f *can be written as* $F(\Phi_0(b,c,d), \Phi_1(b,c,d), a)$.

Definition 4.6 defines simple disjunctive decomposition and is the basis for more complex decomposition structures, i.e., more than one function Φ. However, it has been shown by Shannon [146] that the fraction of all functions of n-variables possessing non-trivial simple disjunctive decompositions approaches zero as n approaches infinity. To allow for more complex decompositions, Ashenhurst used the concepts of his disjunctive decomposition theory to develop a non-disjunctive decomposition theory.

Definition 4.7. *Let $A \subset X$ and $B \subset X$, $C \subset X$, where $A \neq \emptyset$, $B \neq \emptyset$, and $C \neq \emptyset$. A* **non-disjunctive partition**, *denoted as $A|B:C$, exists if $A \cap B = \emptyset$, $A \cap C = \emptyset$, $B \cap C = \emptyset$, and $A \cup B \cup C = X$.*

Definition 4.8. *A function $f(x_0, x_1, \ldots, x_{n-1})$ has an* **Ashenhurst's simple non-disjunctive decomposition** [10], *iff f can be represented by $F(\Phi(B,C), A, C)$. This is known as partitioning the input variables into the* **bound set** *B, the* **free set** *A, and the* **shared set** *C. The variables in B, A and C are known as the* **bound variables**, **free variables** *and* **shared variables**, *respectively.*

Definition 4.9. *For the partition $A|B:C$ on X, the partition representation of $f(X)$ is defined as $2^{|C|}$ rectangular arrays* (**partition submatrices**), *each with $2^{|A|}$ rows and $2^{|B|}$ columns.*

Theorem 4.2. *A function f has an Ashenhurst's simple non-disjunctive decomposition, denoted by $f = F(\Phi(B,C), A, C)$ iff each of its $2^{|C|}$ partition submatrices has column multiplicity $\nu_i = 2$, where $0 \leq i < 2^{|C|}$, and ν_i denotes the column multiplicity of submatrix i.*

Proof. If each partition submatrix has a column multiplicity $\nu = 2$, then it holds that $f_i(A,B) = F_i(\Phi_i(B), A)$ for all i. Hence,

$$
\begin{array}{lrcl}
 & f(A,B,C) & = & \displaystyle\sum_{i=0}^{2^{|C|}-1} f_i(A,B)p_i(C) \\
 & & = & \displaystyle\sum_{i=0}^{2^{|C|}-1} F_i(\Phi(B), A)p_i(C) \\
\text{Note :} & \Phi_i(B)p_i(C) & = & \Phi(B,C)p_i(C) \\
\text{and} & F_i(\Phi_i(B), A)p_i(C) & = & F_i(\Phi(B,C), A, C)p_i(C) \\
\text{giving} & f(A,B,C) & = & \displaystyle\sum_{i=0}^{2^{|C|}-1} F_i(\Phi(B,C), A, C)p_i(C) \\
 & & = & F_i(\Phi(B,C), A, C)\displaystyle\sum_{i=0}^{2^{|C|}-1} p_i(C) \\
 & & = & F_i(\Phi(B,C), A, C).
\end{array}
$$

□

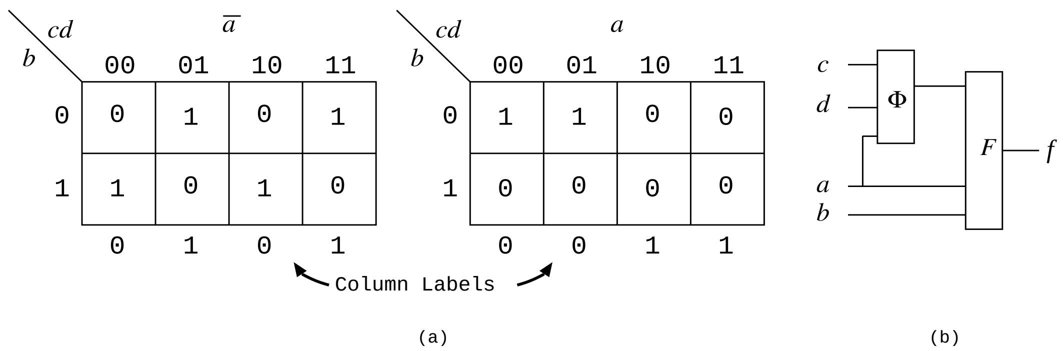


Figure 4.4: Non-disjunctive example (a) non-disjunctive partition $\{b\}|\{c,d\}$: $\{a\}$ (b) decomposed blocks.

Example 4.2. *Consider the Boolean function $f(a,b,c,d)$. The partition $\{b\}|$ $\{c,d\}$: $\{a\}$ is shown in Figure 4.4a. Based on Ashenhurst's definitions, the a and $\overline{a}$ partition submatrices both have column multiplicity ν=2, so the partition has a column multiplicity of two. From the partition, a decomposition can be found in the following form: $f = F(\Phi(a,c,d),\ a,b)$. A column label is used to define a given column for a given submatrix, and the same column label may be used in different submatrices to label different columns. Figure 4.4b shows the block diagram of f after the non-disjunctive decomposition is completed.*

The principle idea of Theorem 4.2 can be expanded for a set of larger column multiplicities.

Definition 4.10. *A Boolean function f has a* **Curtis' non-disjunctive decomposition** [31]*, denoted by the composite function:*

$$f = F(\Phi_0(B,C), \Phi_1(B,C), \ldots, \Phi_{k-1}(B,C), A, C)$$

iff each of the $2^{|C|}$ partition submatrices has column multiplicity $\nu_i \leq 2^k$, where $k < |B|$, and $0 \leq i < 2^{|C|}$.

This ends the discussion of basic functional decomposition, as defined by Ashenhurst and Curtis. The remainder of this chapter extends functional decomposition to multi-valued functions.

4.2 Multi-Valued Functional Decomposition

This section presents the basic principles of the decomposition of multi-valued functions. Note that with a good functional representation, the decomposition of multi-valued functions is just an extension of the decomposition of Boolean functions. Also note the early work in the field of multi-valued functional decomposition [3, 4, 87, 96, 114, 120]. This dissertation addresses the issue of multi-valued decomposition, including functions that are incompletely specified.

A multi-valued function over multi-valued arguments x_i, $0 \leq i < n$, denoted by $f(x_0, x_1, \ldots, x_{n-1})$, takes output values from a finite set of values. A multi-valued variable x_i can take values from the set $Q_i = \{q_0, q_1, \ldots, q_{|Q_i|-1}\}$. For reference, Q_i denotes the set of values that variable x_i can have. Each symbolic value q_k can be associated with a unique integer k, and because only integer values are considered, then $Q_i = \{0, 1, \ldots, |Q_i| - 1\}$.

A multi-valued function f is a function which maps vertices in $Q_0 \times Q_1 \times \ldots \times Q_{n-1}$ to Q_f, formally, $f : Q_0 \times Q_1 \times \ldots \times Q_{n-1} \rightarrow Q_f$.

Definition 4.11. *For the partition $A|B$ over a set of input variables X, a* **multi-valued partition matrix** *representation of $f(X)$ is defined as a rectangular array of the functional values of f. Given that $A = \{x_j, \ldots, x_k\}$ and $B = \{x_l, \ldots, x_m\}$, then the partition matrix is arranged in $|Q_A| = |Q_j| \times \cdots \times |Q_k|$ rows and $|Q_B| = |Q_l| \times \cdots \times |Q_m|$ columns.*

A general multi-valued partition $A|B$ is shown in Figure 4.5, the rows correspond to the $|Q_A|$ configurations of A in order, and the columns correspond to the $|Q_B|$ configurations of B in order.

A multi-valued partition matrix is defined as a canonical form by factoring the variables in B from Expression 2.1, much like Expression 2.5 to find the

$$
\begin{array}{cc|cccc|}
 & & & B & & \\
 & & 0 & 1 & \cdots & |Q_B|-1 \\
\cline{3-6}
 & 0 & f_0 & f_1 & \cdots & f_{|Q_B|-1} \\
A & 1 & f_{|Q_B|} & f_{|Q_B|+1} & \cdots & f_{(2\times|Q_B|)-1} \\
 & \vdots & \vdots & \vdots & \vdots & \vdots \\
 & |Q_A|-1 & f_{(|Q_A|-1)\times|Q_B|} & f_{((|Q_A|-1)\times|Q_B|)+1} & \cdots & f_{(|Q_A|\times|Q_B|)-1} \\
\cline{3-6}
\end{array}
$$

Figure 4.5: Partition matrix for multi-valued functions.

following:

$$
\begin{aligned}
f(x_0, x_1, \ldots, x_{n-1}) &= \sum_{i=0}^{Q_A-1} p_i(A) f_{p_i(A)}(A, B) \\
&= \sum_{i=0}^{Q_B-1} p_i(B) f_{p_i(B)}(A, B). \qquad (4.4)
\end{aligned}
$$

The definitions of simple disjunctive decompositions can easily be applied to functions that are incompletely specified. An incompletely specified function can be represented in a partition matrix where the column multiplicity is found by finding columns that are *compatible.*

Definition 4.12. *Two columns in a partition matrix are* **compatible** *if for every row, the output values of both columns are equal, or if at least one of the output values is a don't care.*

Example 4.3. *Given the partition $\{a\}|\{b, c\}$ of f in Figure 4.6, the columns "00" and "11" are equal, and columns "01" and "10" are compatible. By setting the don't care in column "01" to 3, columns "01" and "10" are now equal and the column multiplicity ν is two.*

a \ bc	00	01	10	11	
0	0	1	1	0	
1	2	-	3	2	
	0	1	1	0	← column labels

Figure 4.6: Partition matrix $\{a\}|\{b, c\}$.

4.3 Decomposition Algorithmic Complexities

One of the reasons that functional decomposition has not yet become a mainstream algorithm for logic minimization tools has to do with the decomposition procedures that have large computational complexity. In fact, little has been written about the practical implementations of decomposition for logic synthesis or machine learning from the 1950s until the late 1980s. However, the advent of faster computing has allowed decomposition to be used in many applications. For example, decomposition has become quite popular because of its application to logic synthesis for Field Programmable Gate Arrays (FPGAs) [78, 107, 108, 109, 149, 152].

Functional decomposition is an iterative process that can be divided into the following three categories: partition selection, column minimization and encoding. In almost all publications on functional decomposition, the methodology for finding a decomposition of a function is as follows: first, find the column multiplicity of a subset of all possible partitions of a function. Second, use a figure of merit based on the number of bound, shared and free variables, and the column multiplicity for each partition to determine which partition should result in the "best" decomposition. Third, using the "best" partition, decompose the function into the form $F(\Phi(A), B)$. In most cases, the creation of the composite functions F and Φ is done by encoding the unique columns in the partition. Fourth, iteratively decompose the composite functions F and Φ until the resulting composite functions cannot be decomposed further. The following subsections illustrate the algorithmic complexities of the decomposition process.

4.3.1 Partition Selection

Partition selection is the problem of finding the "best" partition. In the case of a simple disjunctive decomposition the number of non-trivial partitions is determined by the following:

$$\sum_{i=2}^{n-1} \binom{n}{i} = 2^n - n - 2 = O(2^n),$$

where n is the number of input variables, and i signifies the number of variables in the bound set. (Notice that the number of bound variables is at least two and less than the number of variables.)

In determining the number of partitions for a non-disjunctive decomposition there must be at least one shared variable and at least one variable in the free set and two variables in the bound set. Consequently, the number of non-trivial, non-disjunctive partitions is given by the following:

$$\sum_{i=3}^{n-1} \binom{n}{i} \times (2^i - i - 2) = O(3^n).$$

Also, the function being evaluated must have at least four input variables to have a non-trivial, non-disjunctive decomposition. Due to the large number of non-disjunctive partitions, there have only been a few attempts [149, 152].

In general, when the problem space is exponential in size, researchers look for heuristic solutions. In the case of FPGAs, most decomposition programs look for partitions with only four or five bound variables (e.g., a general size for computed logic blocks in FPGAs is four or five inputs for each block), which reduces the search space significantly. Note that after finding a partition and determining the composite functions, the new composite functions can be further decomposed. That is, decomposition is an iterative process in which each composite function that is created by decomposition may also be decomposed.

4.3.2 Column Minimization

Column minimization is the process of finding the smallest column multiplicity for a given partition. It is assumed by most authors that solving for the smallest column multiplicity will produce the smallest resulting net-list after decomposing a given function. An exception to this is found in [121], where the authors do not use the minimal column multiplicity. Their experimental results show that the decomposition generally finds the smallest possible net-lists (but this is not true for all cases).

In minimizing the column multiplicity, one must determine which columns are compatible (Definition 4.12). To determine if two columns are compatible for every row the output value of both columns must be equal or at least one of the output values must be a don't care. For this reason, the determination of column compatibility (equality) between two columns has complexity $O(|Q_A|)$, where $|Q_A|$ denotes the number of rows for a given column. Section 4.5 presents a method that is $O(1)$ for finding column equality and $O(|Q_A|)$ in the worst case for checking column compatibility for two columns.

After finding column compatibilities, column minimization is used to determine the smallest column multiplicity. When a function is completely specified, this is a task of determining equalities. But when a function is incompletely specified, many different column compatibilities may exists. For instance, given that columns a and b, and columns a and c, are compatible, but columns b and c are not compatible, in determining column multiplicity, should columns a and b become equal by setting don't cares or should columns a and c?

In determining the smallest column multiplicity, graph coloring algorithms may be used to solve this problem. In the case of machine learning problems, the functions can be 99.9% unspecified. In terms of column minimization with a highly unspecified function, most columns may be columns of don't cares and many column compatibilities may exist. For highly unspecified functions, it is possible to have graph coloring problems where the compatibility graph is almost a complete graph.

In the past, the column minimization problem has been reduced to one of the theoretically equivalent optimization problems: graph coloring, graph maximum independent set partitioning, graph maximum clique partitioning, or graph maximum clique covering [120] (see Section 2.3 for more information on graph coloring and maximal clique). All of these problems are known to be NP-hard.

4.3.3 Partition Encoding

Partition encoding is the problem of encoding each column label in a disjunctive partition matrix. Given that a partition has a column multiplicity ν, then there are ν unique labels for the partition. Encoding the symbolic labels to binary functions creates $\lceil log_2\nu \rceil$ Φ composite functions. Given that $P(n,r) = \frac{n!}{n-r!}$, there are $P(2^{(\lceil log_2\nu \rceil)}, \nu)$ different binary encodings for a given partition with column multiplicity ν. But, the number of non-redundant, non-negative Boolean encodings is

$$\frac{(2^{(\lceil log_2\nu \rceil)} - 1)!}{(2^{(\lceil log_2\nu \rceil)} - \nu)!(\lceil log_2\nu \rceil)!}.$$

For example, if the column multiplicity $\nu = 9$, then there are 4.15×10^9 possible Boolean encodings. However, the number of non-redundant, non-negative encodings is 1.08×10^7.

In terms of multi-valued functional decomposition, the multi-valued equivalent of Ashenhurst's simple decomposition can be created by making the number of output values of the composite function Φ equal to the column multiplicity ν of the partition. This creates a single composite function Φ that is ν-valued. Obviously, creating large ν-valued Φ composite functions might not always be applicable for some particular decomposition applications. To restrict the number of output values for the composite function Φ, a partition is declared non-decomposable if ν is too large.

Example 4.4. *Consider the function $f(a,b,c)$ shown in Figure 4.7, where a and c are three-valued variables, and b is a two-valued variable. The three types of encoding are shown directly below the partition. Three types of encoding are shown*

c \ ab	00	01	10	11	20	21	
0	1	1	1	0	1	0	
1	0	1	0	0	2	0	
2	2	2	2	1	2	1	
	0	1	0	2	3	3	} ν–valued encoding
	0	1	0	2	0	2	} 3–valued encoding
	0	0	0	0	1	0	
	0	1	0	0	1	0	} 2–valued encoding
	0	0	0	1	1	1	

Figure 4.7: Multi-valued encoding example.

because it is unclear which of the encodings should be used. Should the output composite functions be based on the value of one of the inputs, or should only one composite function be created by making the composite function ν-valued? This is an open question that is very complex, and specific to each problem domain.

There has been very little written about the encoding problem for functional decomposition (Boolean or multi-valued), but a good overview of the topic can be found in [118].

4.3.4 Insertion of Don't Cares

During the decomposition process, it is possible that don't cares will be added to either of the composite functions. Insertion of don't cares may add to the complexity of finding decompositions for the composite functions. Of course, there are possible advantages of inserting don't cares, given that the presence of don't cares may create more column compatibilities [23].

Don't cares are added to the composite functions as a by-product of column minimization. For instance, given a Boolean decomposition, if the column mul-

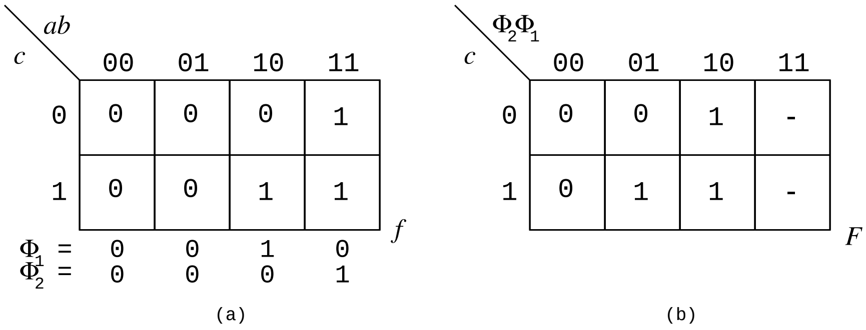


Figure 4.8: Don't care insertion example.

tiplicity is not a power of two, then don't cares are added to the partition to complete the construction of the composite functions. This is illustrated in the following example.

Example 4.5. *Given the function $f(a, b, c)$ in Figure 4.8a, the partition has a column multiplicity ν=3, one possible encoding is shown just below the partition. The decomposition produces $f = F(\Phi_1(a, b), \Phi_2(a, b), c)$, where F is shown in Figure 4.8b. Because of the encoding in Figure 4.8a, F is not specified when $\Phi_1 = 1$ and $\Phi_2 = 1$, thus, for that column in the partition $\{c\}|\{\Phi_2, \Phi_1\}$, all output values are don't cares.*

4.4 Figure of Merit and Heuristics

In order to reduce the algorithmic complexities in functional decomposition, several heuristics and figures of merit have been proposed. Curtis pointed out that to evaluate each partition a cost function must be used to determine the "effectiveness" of the partition. At the time of writing his book and his papers, the technology that existed placed constraints on the timing and area of certain Boolean gates. With current VLSI technology, constraints are placed on some devices, e.g., an XOR gate takes up more area than a corresponding AND gate or

OR gate. With FPGAs that use lookup table architectures, there is no difference between gates with equal numbers of inputs.

An appropriate figure of merit for binary representations that do not require a distinction between the different gate types is called Decomposed Function Cardinality (DFC). This idea was first proposed by Abu-Mostafa in 1988 [2], and its links to decomposition are discussed by Ross *et al.* in [132].

Definition 4.13. *A Boolean function $f(x_0, x_1, \ldots, x_{n-1})$ with m-outputs, has* **cardinality** $2^n * m$. **DFC** *is defined as the sum of the function cardinality of each decomposed functional block. A* **non-decomposable** *function is defined as a function that has a DFC $\geq 2^n * m$.*

DFC is a good representation of the complexity or "pattern-ness" of a function, because a function is a set of ordered pairs and, as with any set, has a definite property in the number of elements or cardinality. Thus, for a function that has multiple decompositions, DFC may be used to find the minimum combined component cardinality of all the decompositions of that function. Because of the simplistic representation of DFC, it is also a good figure of merit for evaluating function implementations in most current technologies.

A problem with using DFC in decomposition occurs when a function is found to be non-decomposable (according to Definition 4.13), because the decomposition process stops. This approach would be insufficient in the case of decomposing functions to realize circuit layouts. As a consequence, some other evaluation criterion must be added to DFC or a totally different figure of merit must be used for layout-driven functional decomposition. The overall requirement is that the figure of merit that is used to control the decomposition process must lead to small decompositions without having an exponential complexity. Because the decomposition process in this dissertation is used for several different outputs (e.g., VLSI layout, machine learning), different figures of merit are applied. Figures of merit are discussed again in the associated chapters on machine learning and VLSI layout.

4.5 Decision Diagram Based Decomposition

Because partition matrices are exponential in the number of functional inputs, many researchers have looked for different data structures to represent partition matrices, including decision diagrams. This section discusses decision diagram based-functional decomposition [49, 51, 76, 77, 78] (see Definition 3.1 for the basic definition of decision diagrams).

Definition 4.14. *A* **decision diagram partition matrix** *of a function can be represented by partitioning the input variables in the decision diagram, such that all of the bound variables are above (at the top of the graph) all of the free variables. Note, the order of variables within the bound set or within the free set has no effect on the partition found.*

Definition 4.15. *The* **cut_level** *designates the boundary between the bound variables and the free variables in the decision diagram.*

The method for detecting decompositions using the decision diagram canonical form is called the *cut_level* algorithm [78]. The algorithm states that the number of distinct columns in a decision diagram partition matrix is the number of vertices below the cut_level that have an edge that crosses the cut_level. This statement is true because of the canonical representation of the decision diagram. If a function is completely specified, then the column multiplicity is just the number of distinct columns found in the decision diagram.

Example 4.6. *Given a Boolean function $f(a, b, c, d, e)$, the partition $\{a, b\}|\{c, d, e\}$ is shown in Figure 4.9a. Because the column multiplicity is three and the number of Φ composite functions is $\lceil log_2 3\rceil = 2$, f can be written as $F(\Phi_0(c, d, e), \Phi_1(c, d, e), a, b)$.*

The corresponding decision diagram of f is shown in Figure 4.9b. The bound set variables are above the cut_level and all the free set variables are below the cut_level. The column multiplicity is the number of vertices below the cut_level that have an edge that crosses the cut_level, i.e., ν=3. Figure 4.9c shows the functional blocks that result from the decomposition.

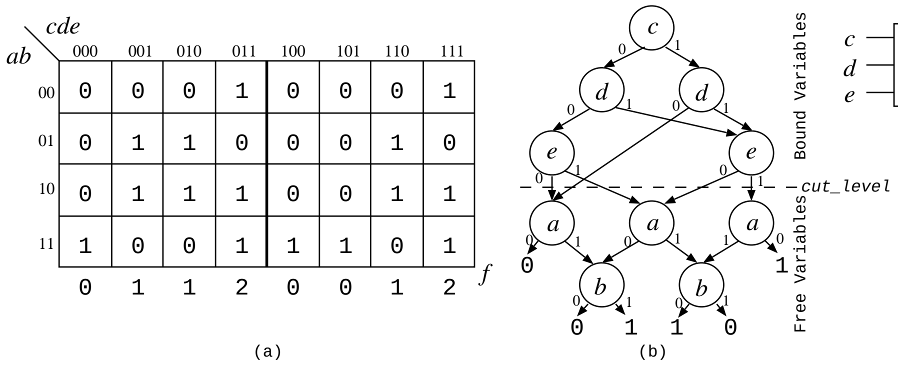


Figure 4.9: Binary Curtis' decomposition example (a) partition matrix $\{a,b\}|$ $\{c,d,e\}$ (b) corresponding BDD (c) decomposed blocks.

As proposed by the author in [49], the *cut_level* algorithm can easily be extended to multi-valued functions. The following example illustrates the *cut_level* algorithm for multi-valued functions.

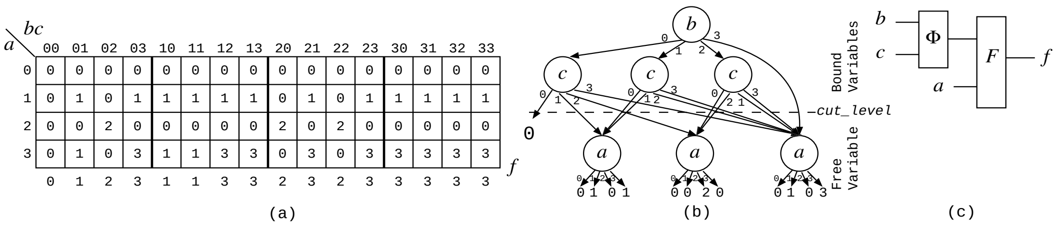


Figure 4.10: Multi-valued decomposition example (a) partition matrix $\{a\}|\{b,c\}$ (b) corresponding MDD (c) decomposed blocks.

Example 4.7. *Consider the four-valued function in Figure 4.10, where the partition* $\{a\}|\{b,c\}$ *of the function* $f(a,b,c)$ *is shown in Figure 4.10a. Determining the column multiplicity using the decision diagram shown in Figure 4.10b is the same as in Example 4.6. The number of distinct columns is four, which implies the decomposition* $f = F(\Phi(b,c), a)$*, where* Φ *is a four-valued function. Figure 4.10c illustrates the decomposed functional blocks.*

Another possible decomposition could be of the form: $f = F(\Phi_0(b,c), \Phi_1(b,c),$

a), where Φ_0 *and* Φ_1 *are three-valued or two-valued functions. In the case that* Φ_0 *and* Φ_1 *are three-valued functions, and given that the column multiplicity is four, there are* $\frac{9!}{(9-4)!} = 3024$ *different encoding possibilities.*

The *cut_level* algorithm, although popularly used, does have its problems. To evaluate partitions, the variables must be reordered within the decision diagram package. The process of variable reordering can actually lead to diagrams that have exponential size [22], which can result in increased runtimes.

4.6 Decomposition Summary

This chapter presented Ashenhurst's and Curtis' decompositions. The reason that decomposition has not become mainstream is because of the NP-hard problems inherent in the decomposition process. But, because of the robustness of functional decomposition (ability to find minimal structures for many different applications), it is still an area of great interest. Also presented was a BDD based decomposition method called the *cut_level* algorithm [78]. The author has extended the *cut_level* algorithm to MDDs, as presented in [49]. This allows the decomposition of multi-valued and machine learning functions.

The rest of this dissertation explores the application of decomposition to VLSI layout and machine learning. To aid in the decomposition of these applications, the next chapter presents a new decomposition method. This method is based on Ashenhurst's and Curtis' decomposition but reduces the algorithmic complexity problems by evaluating partitions with only small bound sets.

Chapter 5

A New Decomposition Method

This chapter presents a new decomposition method based on attempts at solving problems that existed when trying to find an efficient implementation of standard Curtis' decomposition for binary and multi-valued functions with an arbitrary number of don't cares.

The new decomposition strategy is to reduce the algorithmic complexities of functional decomposition partition selection. This chapter presents the methodology of only evaluating small bound sets. The objective is to only evaluate bound sets up to three values, which reduces the partition selection complexity from $O(3^n)$ to $O(n^3)$. An advantage of small bound sets is that the number of columns in a partition is also reduced. By reducing the number of columns in a partition, column multiplicity can be found by using exact methods. Another advantage of small bound sets is that the number of possible binary encodings is also reduced.

A new encoding strategy is also presented in this chapter, where only one composite function is created that is ν-valued. In the case of decomposing Boolean functions, at the completion of the decomposition, post-decomposition encoding is performed to obtain a net-list of Boolean functions.

It is the author's hypothesis that a decomposition methodology that is based solely on small bound sets and shared (repeated) variables will produce the same quality decompositions as decompositions based on medium or large bound sets on most classes of functions. This hypothesis is demonstrated by the decomposition of various classes of benchmarks as presented in Chapter 6. By showing that a decomposition based on small bound sets generates the minimal solutions for these benchmarks, the hypothesis should apply to many other benchmarks. Goals of this chapter:

1. create a new decomposition methodology that extends Curtis' Boolean decomposition by creating a new definition of a decomposable function. The new definition allows the decomposition of many small bound set partitions that cannot be decomposed by using Curtis' definitions (Section 5.1),

2. introduce a multi-valued labeling scheme that is used instead of Boolean encoding. Multi-valued labeling, as far as the author knows, has never been used in the decomposition of Boolean functions for logic synthesis. The idea of multi-valued labeling is to always produce an Ashenhurst's decomposition (only one composite function Φ) by making Φ a multi-valued function that is ν-valued. After decomposing the function, the multi-valued composite functions are encoded with binary values to produce Boolean functions (Section 5.2),

3. illustrate the need to reduce the number of all partition evaluations to a set of partitions with only small bound sets (Section 5.3),

4. demonstrate that evaluating small bound set partitions still results in high quality decompositions (Section 5.3),

5. demonstrate why small bound set partitions must be used when decomposing highly unspecified or machine learning functions (Section 5.4),

6. define existing and new decomposition types that can easily be found using small bound set partitions. While not explored here, each of these new decomposition types can be extended to any size bound set partitions (Section 5.5).

The algorithms for this new decomposition method, based on small bound sets and post-decomposition encoding, are presented in Chapter 6.

5.1 Extension to Curtis' Decomposition

Given an n-input, single-output Boolean function, f: $\{0,1\}^n \to \{0,1\}$, f has the set of input variables $X = \{x_0, x_1, \ldots, x_{n-1}\}$. Let $A \subset X$ and $B \subset X$ where $A \neq \emptyset$, $B \neq \emptyset$, $A \cap B = \emptyset$ and $A \cup B = X$.

From Definition 4.6, Curtis defined a disjunctive decomposition of a Boolean function as a function in the form:

$$f = F(\Phi_0(B), \Phi_1(B), \ldots, \Phi_{k-1}(B), A),$$

iff the partition matrix $A|B$ has column multiplicity $\nu \leq 2^k$, where $k < |B|$.

Curtis' definition on a decomposable partition forces the composite function F to have less inputs than the original function f. This implies that F is a simpler function than f, and the cardinality of F is smaller than f.

His definition insures that a recursive decomposition algorithm will always reduce the complexity of a function.

It should be noted that Curtis' definition does not allow for certain decompositions, most generally when the bound set of a partition is small. These decompositions are important in formulating the small bound set decomposition types presented at the end of this chapter. The following example illustrates the case when a partition defined by Curtis as non-decomposable, leads to a decomposition.

Example 5.1. *Consider the Boolean function shown in Figure 5.1a. $f(a,b,c)$ is a Curtis non-decomposable function, as shown in Figure 5.1b where the column multiplicity ν=3. Because ν=3, there must be $k = \lceil log_2\nu \rceil = 2$, Φ functions. By Curtis' definition of a decomposable partition (k must be smaller than the number of bound variables) and k=2, which implies that this partition is non-decomposable. Assuming that this function is decomposable, i.e. $k \leq |B|$, the partition is encoded into Φ_1 and Φ_2 as shown below the partition $\{c\}|\{a,b\}$. The resulting decomposition is $f = F_0(\Phi_1(a,b), \Phi_2(a,b), c)$, as shown in the block diagram in Figure 5.1c. Because F and f both have three inputs, the two functions have the same cardinality.*

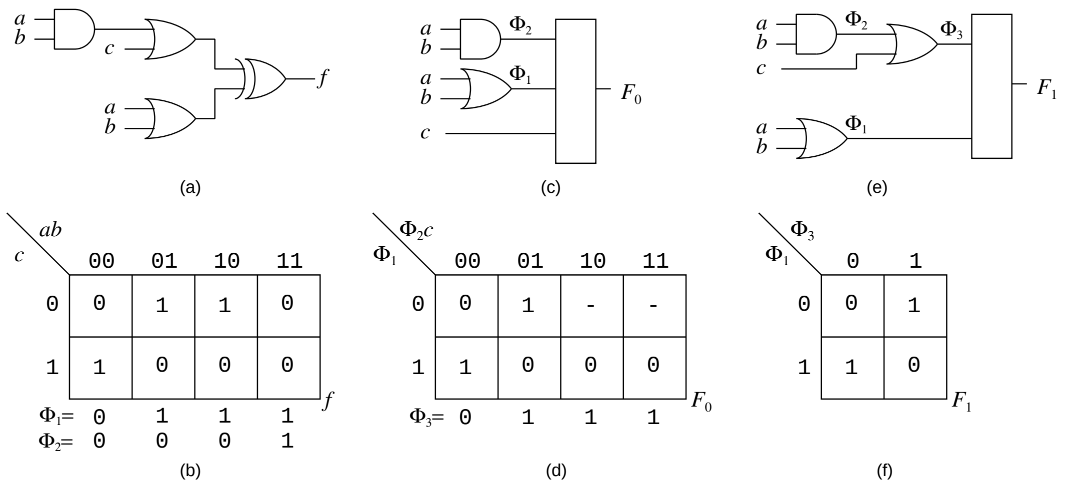


Figure 5.1: Example of decomposing a Curtis non-decomposable function.

Figure 5.1d shows the partition $\{\Phi_1\}|\{\Phi_2, c\}$ for F_0; because the partition has a column multiplicity ν=2, it is Curtis' decomposable. The resulting decomposition is $f = F_1(\Phi_3(\Phi_2(a,b),c),\Phi_1(a,b))$ as shown in the block diagram (Figure 5.1e). The partition $\{\Phi_1\}|\{\Phi_3\}$ for F_1 is shown in Figure 5.1f, where $F_1 = \Phi_1 \oplus \Phi_3$. The decomposition finds f=$((a \cdot b)+c) \oplus (a+b)$, which is the original minimal expression of f (Figure 5.1a).

Example 5.1 illustrates that a small bound set partition that is defined as non-decomposable by Curtis still results in a good decomposition. The following definition extend Curtis' definition of decomposability to include partitions of the type shown in the example.

Definition 5.1. *An* **extended disjunctive decomposition** *of a Boolean function is a function in the form, $f = F(\Phi(B), A)$, iff the partition matrix $A|B$ has column multiplicity $\nu < 2^k$, where $k \leq |B|$.*

Definition 5.1 sets the conditions for a new decomposition method that allows a function to be decomposed if the column multiplicity ν of a partition is less than the total number of columns in the partition. The only time a function cannot be decomposed for a given partition is if ν is equal to the number of columns in the partition.

Definition 5.2. *An* **extended multi-valued disjunctive decomposition** *of a multi-valued function is a function in the form:*

$$f = F(\Phi_0(B), \Phi_1(B), \ldots, \Phi_{k-1}(B), A)$$

iff the partition matrix $A|B$ has column multiplicity $\nu < |Q_B|$.

Through the remainder of this dissertation, a function is considered decomposable as defined by Definition 5.1 or Definition 5.2. These two definitions allow many different decompositions that cannot be found with Curtis' decomposition. Because the remainder of this chapter presents decompositions with small bound sets, Definition 5.1 and Definition 5.2 are used to increase the number of decomposable functions with small bound sets.

5.2 Multi-Valued Partition Labeling

This section presents a methodology of using multi-valued partition *labeling* instead of Boolean encoding. After column multiplicity has been found by using graph coloring, each compatible column grouping is given a color. **Labeling** is defined as assigning each color a unique integer value, unlike encoding where each color is assigned a unique binary coded value. Multi-valued labeling results in an Ashenhurst's decomposition with a single-output, ν-valued Φ composite function, where ν is the column multiplicity.

The method of decomposition in [154] uses multi-valued labeling instead of Boolean encoding for machine learning functions. The difference in this dissertation is that the multi-valued labeling is also used for the decomposition of Boolean functions. In the case of Boolean logic synthesis, after the decomposition process is completed the multi-valued composite functions are encoded to Boolean values. Note that the number of post-decomposition Boolean encodings is the same as the number of possible encodings while decomposing the function.

But, by using multi-valued labeling, the decomposition process is not dependent on the encoding process. The end result is a decomposition based solely on partition selection and column compatibility and post-decomposition encoding to produce Boolean functions.

In comparison to Boolean encoding, where several possible Boolean encodings are possible, there is only one multi-valued labeling. Each Boolean encoding results in a different decomposition, which implies that all Boolean encodings must be used to find the smallest decomposition.

The number of non-redundant, non-negative Boolean encodings is

$$\frac{(2^{(\lceil log_2 \nu \rceil)} - 1)!}{(2^{(\lceil log_2 \nu \rceil)} - \nu)!(\lceil log_2 \nu \rceil)!}.$$

Therefore, for each bound set, testing all encodings is impractical. The problem is that encoding is used to create the composite functions that result from the decomposition. Consequently, choosing a bad encoding can create composite functions that cannot be decomposed by the iterative decomposition algorithm.

Subsection 4.3.4 illustrated that in some cases (based on encoding and column minimization) the decomposition process may add don't cares to the resulting composite functions. In some papers, the addition of don't cares is used to help in the decomposition process [24]. In this dissertation, the intention is to experimentally verify that the addition of don't cares is unnecessary and should not be used. In machine learning, the function is already 99.9% unknown, so why add more unknown values? In the case of logic synthesis, or completely specified functions, the addition of don't cares may lead the decomposition algorithm to decompositions that are not minimal. By adding don't cares, the number of column compatibilities increases, which increases the number of different graph colorings. This leads to algorithms that cannot find exact solutions, but must use heuristic searches. The advantage of multi-valued labeling is that don't cares are never added to the composite functions.

Mathematically, this labeling technique has a great advantage over Boolean encoding in that the bound variables are not separated into separate functions but are defined by one function. The following definition defines multi-valued labeling mathematically, using relations.

Definition 5.3. *Given sets S and T, a* **relation** ρ *on* $S \times T$ *is a subset of* $S \times T$.

From this definition, given a set S, a relation ρ on set S is a subset of $S \times S$. For example, let $S = \{0, 1\}$, and ρ be given by $x \rho y \leftrightarrow x \cap y$, then $(1, 1) \in \rho$.

Definition 5.4. *Given a Boolean function* $f(x_0, x_1, \ldots, x_{n-1})$ *and bound set* $B = \{x_i, x_j\}$, *then the function* $f = F(\rho(B), A)$ *implies that* ρ *is an* **encoded relation** *on* B *such that* $\rho = \{\overline{x_i}\ \overline{x_j}, \overline{x_i}x_j, x_i\overline{x_j}, x_ix_j\}$.

ρ fully represents the variable pair x_ix_j, because it is a set of all possible configurations of x_ix_j, i.e., $\rho = \{\overline{x_i}\ \overline{x_j}, \overline{x_i}x_j, x_i\overline{x_j}, x_ix_j\}$. Each element in ρ is referred as $\rho[k]$, where k is the k-th element in the set. For example, from the above definition, $\overline{x_i}\ \overline{x_j}$ is the $\rho[0]$ element in the list, and x_ix_j is the $\rho[3]$ element. The relation ρ is reduced by removing any redundant elements. If two elements in ρ are equal or can be made equal by setting don't cares then one of the elements is thought of as redundant. This, of course, is the same as finding the column multiplicity of the partition.

The following example compares multi-valued labeling and encoding on a simple Boolean function.

Example 5.2. *Given the Boolean function $f = (ab) + c$ (Figure 5.2a), the best decomposition is obviously the partition with bound set $B = \{a, b\}$. The partition $\{c\}|\{a, b\}$ has column multiplicity $\nu = 2$. The encoding of Φ is shown below the partition. The decomposition $f = F(\Phi(a, b), c)$ occurs when $\Phi = ab$ (Φ is true when a and b are both true), $\overline{\Phi} = \overline{ab}$ (Φ is false when either a or b is false). Figure 5.2b shows the composite function F, where $F = \Phi + c$. The decomposition results in the initial Boolean function $f = (ab) + c$.*

In terms of relations, the multi-valued labeling is the same as the encoding of the partition (shown below the partition in Figure 5.2a). The decomposition $f = G(\rho(a, b), c)$ occurs when $\rho = \{\overline{ab}, ab\}$, where $\overline{ab}$=($\overline{a}\ \overline{b}$ + $\overline{a}b$ + $a\overline{b}$). Figure 5.2c

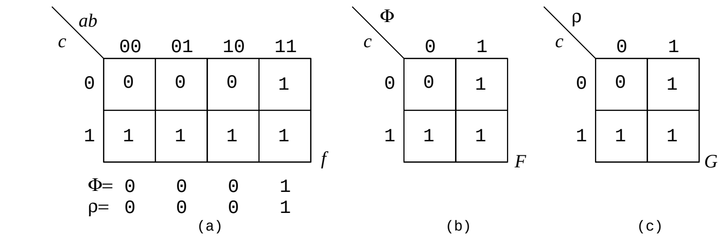


Figure 5.2: Decomposition of $f = (ab) + c$ with partition $\{c\}|\{a,b\}$.

shows that $G = \rho[1] + c$*, which implies that* $G = ab + c$*. The decomposition in terms of labeling also finds the original function* $f = (ab) + c$*.*

Example 5.2 shows that multi-valued labeling can be used on simple functions to represent the resulting composite function ρ. In essence, the labeling carries information about the bound set variables to the next level of the decomposition.

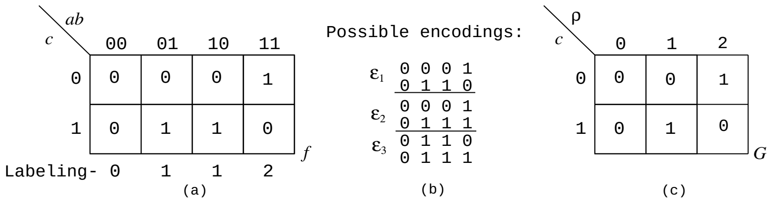


Figure 5.3: Decomposition of $f = (a \oplus b)c + \overline{c}ab$ with partition $\{c\}|\{a,b\}$.

Example 5.3. *Given the Boolean function* $f = (a \oplus b)c + \overline{c}ab$ *(Figure 5.3a), the partition* $\{c\}|\{a,b\}$ *has column multiplicity* ν*=3. If the partition is Boolean encoded, then there are three distinct (non-redundant) encodings, as shown in Figure 5.3b. To decompose the function further it is necessary to find the correct encoding. In this example, if the encoding* $\{0001\}$ $(a \cdot b)$ *and the encoding* $\{0110\}$ $(a + b)$ *are not chosen, then the resulting decomposition will not be the original function. From this example, it is obvious that all possible encodings must be checked to guaranty that the minimal decomposition can be found.*

In the case that multi-valued labeling is used, $f = G(\rho(a,b),c)$, *where* $\rho = \{\overline{a}\ \overline{b},\ a \oplus b,\ ab\}$. *Obviously,* $a \oplus b = (\overline{a}b + a\overline{b})$ *is used. Figure 5.3c, shows that* $G = c\rho[1] + \overline{c}\rho[2]$, *which gives* $f = c(a \oplus b) + ab\overline{c} = (a+b)c + ab$.

Example 5.3 illustrates that all encodings must be checked to insure that the decomposition results in the original minimal expression of the function. But, most published decomposers use heuristics to reduce the number of encodings that are used. In fact, generally only one encoding is used.

Example 5.3 also shows that multi-valued labeling finds the original minimal function. If a partition is labeled using ρ, ρ stores all the information about each compatible grouping of columns in the partition. The following theorem is used to illustrate the affect of multi-valued labeling on the decomposition process.

Theorem 5.1. *If* $f(A,B,C)$ *has Ashenhurst's decompositions of the form* $f = H(\Omega(A,B),C)$ *and the form* $f = G(\Upsilon(A),B,C)$, *then* f *has an Ashenhurst's decomposition in the form:*

$$f = F(\Phi(\Upsilon(A),B),C).$$

Proof. The expansion of $f(A,B,C)$ about C can be expressed as follows:

$$\begin{aligned} f(A,B,C) &= \sum_{i=0}^{2^{|C|}-1} H_i(\Omega(A,B))p_i(C) \\ &= \sum_{i=0}^{2^{|C|}-1} G_i(\Upsilon(A),B)p_i(C) \\ &= \sum_{i=0}^{2^{|C|}-1} F_i(\Phi(\Upsilon(A),B))p_i(C). \end{aligned}$$

Because of the uniqueness of the expansions, $H_i(\Omega(A,B)) = G_i(\Phi(A),B) = F_i(\Phi(\Upsilon(A),B))$. □

Theorem 5.1 (shown graphically in Figure 5.4) holds for multi-valued labeling and is used in the next section for small bound sets.

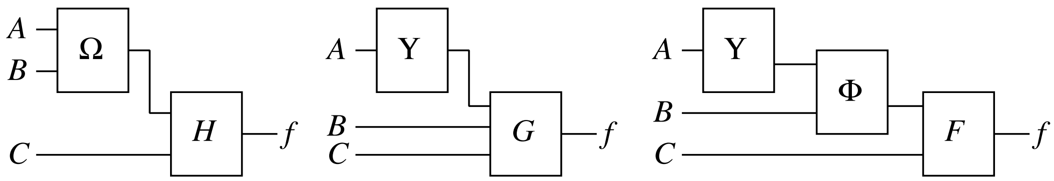


Figure 5.4: Block diagrams illustrating Theorem 5.1.

This section has completed the analysis of a multi-valued labeling scheme that is used to create Ashenhurst's decompositions. By using Ashenhurst's decomposition, the decomposition iterative process is not constrained by the number of different encodings that are possible when using Curtis' decomposition. That is, the decomposition process is not dependent on the encoding that was chosen; therefore, the resulting decomposed function of a multi-valued labeling-based decomposition should produce the same quality results as an encoding-based decomposition. The concept of multi-valued labeling is experimentally verified in Chapters 6 and 9. For Boolean encoding-based decomposition to guarantee the same results as multi-valued labeling-based decomposition, all possible Boolean encodings will have to be checked. This is an impractical solution because of the complexities involved in decomposition. The rest of this chapter reduces the complexity of decomposition further by evaluating partitions with only small bound sets.

5.3 Small Bound Set Partitioning

While there are three different aspects to functional decomposition (partitioning, encoding, and column multiplicity) partitioning is the starting point and probably the most important component. This was shown in the previous chapter, where the number of possible disjunctive partitions is $O(2^n)$, and the number of possible non-disjunctive partitions is $O(3^n)$.

While many papers discuss the issue of setting the number of bound set variables to a fixed number for field programmable gate array or programmable logic array decomposition [78, 87, 107, 108, 109, 139, 140, 149, 152], there are only a few publications [44, 45, 40, 41, 50, 59, 141, 142, 143] that discuss which partitions should be evaluated and used for functional decomposition.

Several algorithms at Portland State University have been developed on the topic of *medium* sized bound sets (half the variables are in the bound set, the other half in the free set) [23, 24]. The authors advocate the advantages of medium sized bound sets in decomposition. A difficulty is that the number of medium bound sets to search is very large.

To show that small bound set decompositions result in "good" decompositions, the theorems by Ashenhurst and Curtis are used.

Theorem 5.1 (Section 5.2) states that it does not matter which composite function is found first, Ω or Υ; the decomposition $F(\Phi(\Upsilon(A), B), C)$ is still found. The theorem is shown graphically in Figure 5.4. This theorem is important because it shows that if a small bound set decomposition exists, then it does not matter which decomposition was found first; the final decomposition is still the same. Thus, small bound set decompositions can be used in the decomposition process to fully decompose a function. Theorem 5.1 can be expanded to a Curtis' decomposition by using the following theorem:

Theorem 5.2. *If $f(A,B,C)$ is expressible in the form $f = H(\Omega_0(A,B), \Omega_1(A, B), \ldots, \Omega_k(A,B), C)$ and also in the form $f = G(\Upsilon_0(A), \Upsilon_1(A), \ldots, \Upsilon_k(A), B, C)$ then f has the form:*

$$f = F(\Phi_0(\Upsilon_0(A), B), \Phi_1(\Upsilon_1(A), B), \ldots, \Phi_k(\Upsilon_k(A), B), C).$$

Proof. The proof is equivalent to the proof for Theorem 5.1. □

It is possible that $\Upsilon_i(A) = A$ or that $\Phi_j(\Upsilon_j(A), B) = \Upsilon_j(A)$. Theorem 5.2 shows that finding composite functions with small bound sets still holds for Curtis' decompositions.

There are cases, however, when $f(A,B,C)$ is not expressible in the form

$f = G(\Upsilon(A), B, C)$, but is expressible in the form:

$$f = F(\Phi_0(\Upsilon(A), B), \Phi_1(A, B), \ldots, \Phi_k(A, B), C). \tag{5.1}$$

The encoding process used in decomposing $f(A, B, C)$ into $F(\Phi_0(A, B)$, $\Phi_1(A, B), \ldots, \Phi_k(A, B), C)$ produces a function Φ_0 that can be decomposed into $\Phi_0(\Upsilon(A), B)$. Notice that the decomposed form of f repeats the variable A in each Φ function. If $f(A, B, C)$ is expressible in the form shown in Equation 5.1, then f must be expressible in the form: $f = G(\Upsilon(A), A, B, C)$. The problem is finding $G(\Upsilon(A), A, B, C)$ to be a "good" decomposition result. There can be many decompositions in the form $f = G(\Upsilon(A), A, B, C)$, each with the same cost. But, the same argument exists while decomposing a function with large or medium bound sets, i.e., how to determine which decompositions will result in the minimal solution.

There are functions that cannot be decomposed using small bound sets. An example of one such function is $f = F_0(x_0, x_1, x_2, x_3) \oplus F_1(x_4, x_5, x_6, x_7)$, where F_0 and F_1 are non-decomposable functions. The only way that a decomposition of this function will be found is if all four-variable bound sets are checked. Of course, F_0 and F_1 could represent functions with more variables than four. In essence, all partitions must be evaluated to decompose functions of this form, which generally is impractical.

The following is an example of not using small bound sets in the decomposition process:

Example 5.4. *Given the partition $\{d, e, f\}|\{a, b, c\}$ of f in Figure 5.5; the multi-valued labeling is shown below the partition. Two composite functions: $(a + b)$ and $(a \cdot b) + c$ are needed for the decomposition to result in the original function. But, it is impossible to find these two composite functions given this partition (See, "Impossible to get this encoding" in the figure). The only way that the composite functions $(a + b)$ and $(a \cdot b) + c$ can be found is if the columns labeled "1" are separated into two groups "1" and "3" as shown as "Alternative column grouping" in the figure. The alternative column grouping is the only possible*

def \ abc	000	001	010	011	100	101	110	111	
000	0	0	0	1	0	1	1	1	
001	0	0	0	1	0	1	1	1	
010	0	0	0	1	0	1	1	1	
011	1	0	1	0	1	0	0	0	
100	0	0	1	0	1	0	0	0	
101	0	0	1	0	1	0	0	0	
110	0	0	1	0	1	0	0	0	
111	1	1	0	1	0	1	1	1	f
	0	1	2	1	2	1	1	1	- Column labels
	0	0	1	1	1	1	1	1	=>(a+b) } Impossible to get this encoding
	0	1	0	1	0	1	1	1	=>(ab)+c } Impossible to get this encoding
	0	1	2	3	2	3	3	3	- Alternative column grouping

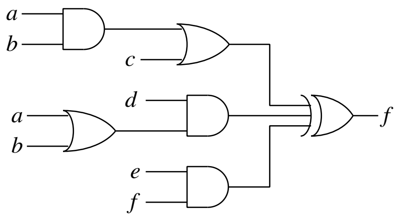


Figure 5.5: Three-input bound set example.

way to find the original composite functions. But, in attempting to separate equal columns into different groups, many iterations are usually necessary. While it is possible to find the minimal solution by using bound sets larger than two variables, the only way that it can be done is if groups of equal columns are separated. Separating the equal columns into different groups, though, is not a viable practice in decomposition because of the complexity involved.

Example 5.4 shows the importance of evaluating two-variable bound sets. When two-variable variable bound sets are not used, it is possible that the minimal decomposition is not found.

5.4 Variable Partitioning for Strongly Unspecified Functions

This dissertation approaches the decomposition process both for logic synthesis and for machine learning. Machine learning functions are typically highly unspecified, which means that the functions have a high percentage of don't knows. (The concept of a don't know in machine learning is covered in Section 2.4.) By don't know the value is not known, and caution must be used in setting a don't know to a value. Setting a don't know to an incorrect value increases the error in the hypothesis function.

Machine learning problems typically have many variables and are highly unspecified. Because most logic synthesis problems are highly specified, this creates a problem for researchers in logic synthesis that apply their algorithms to machine learning. Also, the formulation for decomposing machine learning functions must be based on the concept of the don't know and the complexity issues found in trying to decompose functions with many variables. With the concept of a don't know in mind, this section presents a methodology for selecting partitions for machine learning based decompositions.

For example, given a Q-valued function with 100 inputs (all inputs are also Q-valued), if the bound set has two-variables, then there are Q^2 columns and Q^{98} rows in the partition table. Whereas, if the bound set has 50-variables, then there are Q^{50} columns and Q^{50} rows in the partition table. From Subsection 4.3.2, the column multiplicity problem is a known NP-hard problem. Because of the time complexity of checking for column compatibility, the process of evaluating partitions with two bound variables takes much less time than that of evaluating partitions with 50 bound variables.

Observation 5.1. *Given a function that is completely specified, two columns are* **equal** *if for every row, the output value of the first column is equal to the output value of the second column in the same row (Definition 4.5). Obviously, if the function is Q-valued, completely specified and each value is equally likely,* *then for every row the probability that both column values are equal is* $\frac{1}{Q}$.

From Observation 5.1, for a column with $|Q_A|$ rows, the probability of finding a pair of equal columns is $\left(\frac{1}{Q}\right)^{|Q_A|} = \frac{1}{Q^{|Q_A|}}$. Obviously, the more rows in the columns, the less likely the two columns will be equal. Note, Observation 5.1 is for completely specified functions, but does not hold for machine learning functions which are strongly unspecified (typically less than 1% of a machine learning function is specified).

Observation 5.2. *Assume the probability of a don't know is* $P(don't\ know)$ $= 0.99$*, then the probability of having any other value in the function is* $P(care)$ $= \frac{1-0.99}{Q} = \frac{0.01}{Q}$*, given that the function is* Q*-valued and each care value is equally likely. The only way to have an incompatibility between two columns occurs when, for any given row, the output values of both columns are not equal and neither output value is a don't know. The probability of having an incompatibility between two columns (for any row) is* $P(incomp) = 0.01(\frac{Q-1}{Q})$.

From Observation 5.2, the probability that two columns are compatible for a given row is $1 - P(incomp)$. Consequently, the probability that two columns with $|Q_A|$ rows being compatible is $(1 - P(incomp))^{|Q_A|}$. For example, given a 100-input, Q-valued function, the probability of finding a compatible pair of columns with two bound variables, $(1 - P(incomp))^{Q^{98}}$, given that $|Q_A| = Q^{98}$, is much smaller than the probability of finding a compatible pair of columns with 50 bound variables, $(1 - P(incomp))^{Q^{50}}$, given that $|Q_A| = Q^{50}$. That is, it is highly unlikely that two columns will be found compatible in the case of two-variable bound sets. The only way that they will be compatible is if there is some pattern (symmetry, or other relationship between variables) in the data that allows the combination of the two columns.

If medium or large bound sets are used in the decomposition of machine learning functions, the probability of finding compatible columns is much greater than for small bound sets. Typically, when checking for column compatibility, graph coloring algorithms are used. When evaluating partitions with medium or large bound sets, many columns exist with many column compatibilities which

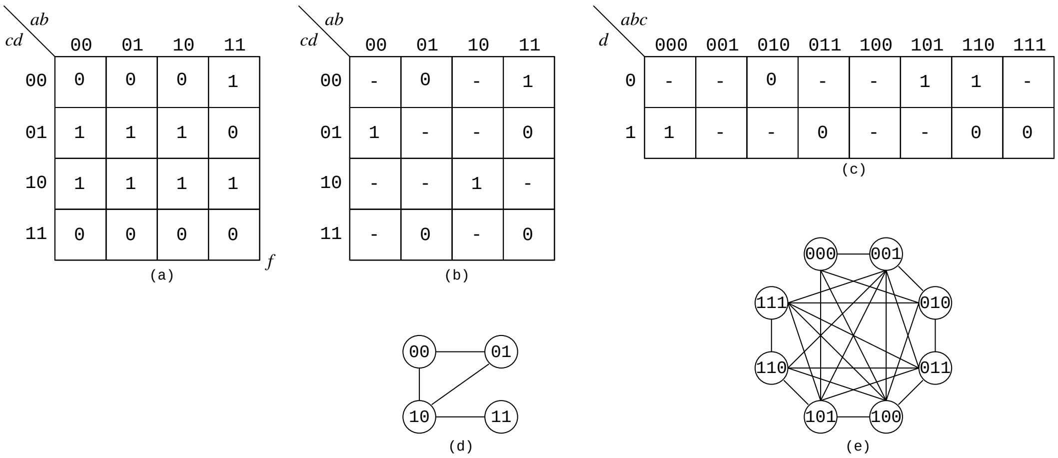


Figure 5.6: Example of partitioning incompletely specified functions.

produces a dense compatibility graph that has many possible colorings, making graph coloring a random process. Often columns are combined that do not come from any existing pattern of tautological cofactors in the function being examined. Combining these columns leads to don't knows being forced to random values, in essence, creating a hypothesis function with a greater error, Example 5.5 illustrates this point.

Example 5.5. *Consider the Boolean function* $f = ((a \cdot b) + c) \oplus d$ *shown in Figure 5.6a.* f *can be decomposed by using the partition* $\{c, d\}|\{a, b\}$ *or the partition* $\{d\}|\{a, b, c\}$ *(both partitions have* $\nu = 2$*).* f *is made incompletely specified by randomly setting output values to don't knows as shown in the partition* $\{c, d\}|\{a, b\}$ *and* $\{d\}|\{a, b, c\}$ *in Figure 5.6b,c. The compatibility graphs for both partitions are shown below their respective partitions in Figure 5.6d,e. The compatibility graph for the partition* $\{c, d\}|\{a, b\}$ *only has four edges, while the compatibility graph for the partition* $\{d\}|\{a, b, c\}$ *has 23 edges. The number of possible colorings for the partition* $\{d\}|\{a, b, c\}$ *is much greater than the number of possible colorings for the partition* $\{c, d\}|\{a, b\}$*.*

Partition $\{c, d\}|\{a, b\}$ *has two column compatibility solutions that result in a column multiplicity* $\nu = 2$*. The first coloring is* $\{00, 01\}$*,* $\{10, 11\}$*, and the second* *coloring is* $\{00, 01, 10\}$, $\{11\}$. *If the second coloring is used, the decomposition results in the original function, while the first coloring will not lead to such a decomposition.*

Partition $\{d\}|\{a, b, c\}$ *has many column compatibility solutions, but there is only one solution that results in the original function. If the wrong solution is used then the decomposition will produce a hypothesis function with error. From the complexity stand point, when the number of bound variables grows, so does the number of possible graph colorings. As the number of colorings grows, it becomes impossible to try all possible colorings. In contrast, if the partition* $\{c, d\}|\{a, b\}$ *is used, then there are only two colorings and both of them may be used in the decomposition to find the original function.*

Also note that when coloring the partition $\{d\}|\{a, b, c\}$, *it is possible that other colorings (not the coloring that finds the original function) will produce a solution that is less complex than the original function. This is a situation in machine learning when finding the function with the lowest complexity does not result in the best solution (this is covered in more detail in Chapter 8).*

From Example 5.5 it is easy to see that as the number of bound variables increases, the number of column compatibilities increases dramatically. In essence, this makes the selection of columns a random process, resulting in more error in the hypothesis function. So, when using functional decomposition to decompose strongly unspecified functions, small bound sets should be used to reduce both the complexity (number of graph colorings, column compatibilities) and the resulting error.

5.5 Small Bound Set Decomposition Types

In the interest of creating a decomposer that is efficient (in terms of time) and effective for completely specified and highly unspecified functions, this section shows two different types of decompositions with small bound sets. The decomposition methodology presented here is applicable to completely/incompletely specified Boolean/multi-valued functions. Note that other small bound set decomposition types exist, but are not applicable when using the multi-valued labeling scheme presented in this chapter. The two decomposition types presented in this section:

1. *single-variable free sets* (row multiplicity is used, so that the complexity of each partition is equivalent to finding the column multiplicity of a single-variable bound set),
2. two-variable bound set partitions,

The goal is to explain these two small bound set decomposition types, each of which can lead to the minimal decomposition of a function. In Chapter 6, a decomposition program based on the decomposition types presented in this section is discussed.

These two decomposition types have been published before by Sawada [142, 143]. But, because Sawada did not redefine the concept of decomposable partitions (as proposed earlier in this chapter), there are cases when Sawada cannot decompose functions presented in this section.

5.5.1 Single-Variable Free Sets

The simplest set of partitions to check is the set with only one variable in the free set. The evaluation of each free variable is done by checking the cofactors of each variable. That is, only the row multiplicity is checked, not column multiplicity with only one free variable. By evaluating cofactors of a given variable (rows in a partition) the following Boolean single-variable free set decompositions are possible:

1. if $f_{\overline{x}} = 0$, f has decomposition $f = x \cdot f_x$,
2. if $f_x = 0$, f has decomposition $f = \overline{x} \cdot f_{\overline{x}}$,
3. if $f_{\overline{x}} = 1$, f has decomposition $f = \overline{x} + f_x$,

4. if $f_x = 1$, f has decomposition $f = x + f_{\overline{x}}$,

5. if $f_x = f_{\overline{x}}$, f has decomposition $f = f_x$ (x is a vacuous variable),

6. if $f_{\overline{x}} = \overline{f_x}$, f has decomposition $f = x \oplus f_{\overline{x}}$ or $f = \overline{x} \oplus f_x$.

The block diagram for these types of decompositions is shown in Figure 5.7.

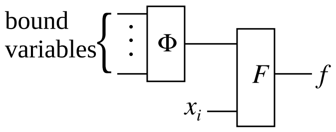


Figure 5.7: Block diagram of a single-variable free set decomposition where x_i is the only free variable.

This method of decomposition quickly reduces the dimension of the problem for the next stages of decomposition. That is, if a single-variable free set decomposition is possible, then the number of input variables to the next stage of decomposition is reduced by one. If a decomposition of this type exists, the time to find a solution is greatly reduced.

Example 5.6. *Consider the Boolean function $f(a, b, c)$, the partition $\{a\}|\{b, c\}$ is shown in Figure 5.8a. Because $f_{\overline{a}} = \overline{f_a}$, f has the decomposition $f = a \oplus f_{\overline{a}}$, as shown in Figure 5.8b or has the decomposition $f = \overline{a} \oplus f_a$, as shown in Figure 5.8c.*

There are only n-partitions of this type for an n-input function. Therefore, the algorithmic complexity of single-variable free set decomposition is $O(n)$. This decomposition quickly reduces the complexity of the function and the next level decompositions.

When decomposing an incompletely specified function it is possible that several variables have a single-variable decomposition (because of the don't cares). Because different orders of decompositions result in different solutions, selecting which partition should be decomposed first is problematic.

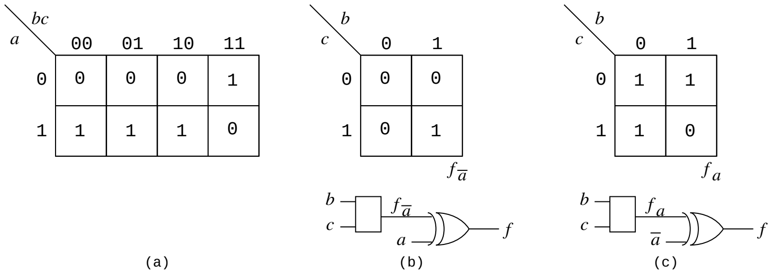


Figure 5.8: An example of decomposing a function with a single variable in the free set.

In most of the pseudo-code listings presented in this chapter, the procedure PARTITION is used. PARTITION is a procedure that is passed a set of bound variables (and possibly shared variables) and returns the respective columns for that bound set. The pseudo code for single-variable free set decomposition is shown in Algorithm 5.1.

Algorithm 5.1 Single variable free set decomposition pseudo code.

for each input variable x_i in function f
  let set $rows$ = PARTITION(x_i)
  for each row in set $rows$
    if all output values in row are a constant value
      create decomposition with x_i as free variable
  if f is a Boolean function
    for each row pair $\{row_j, row_k\} \in rows$
      if $(row_j \oplus row_k)$ is a row with all row values = 1
        create decomposition with x_i as free variable.

For multi-valued functions, a decomposition exists if any cofactor of a given variable is a constant output value (much like the top four cases above). The condition of $f_{\overline{x}} = \overline{f_x}$, cannot be formed under multi-valued functions unless a logic system is given in which a negation operation is defined (such as Galois Field Logic or Post Logic). It is assumed in this dissertation that no such logic system is assumed for machine learning problems, and conditions of this type will not be considered. But, the problem of applying this decomposition to a multi-valued variable is that the variable must be encoded into two new multi-valued variables. For machine learning problems this is not applicable because a multi-valued variable should not be encoded into two new variables (Subsection 8.3.3). But, for the multi-variable labeling technique presented in this dissertation, encoding the multi-valued variable into two new multi-valued variables can be done.

5.5.2 Two Variable Bound Sets

Another decomposition type is the set of partitions with two variables in the bound set. As it turns out, bound sets with two variables can detect several different *symmetric* decomposition types. Each of these symmetric decomposition types is a viable decomposition when used with Definition 5.1. The concept of symmetry (defined in Section 2.1) is useful because it shows the existence of patterns in the function. If a function is partially symmetric in variables $\{x_i, x_j\}$, then some pattern exists between the two variables. In logic synthesis, the pattern can be a basic Boolean operator between the two variables.

Symmetry is not only important for decomposition but also has application to Boolean matching, cell library design, and testing [150, 151]. Much research has been done to detect symmetries in logic functions [34, 39, 42, 64, 90, 97, 106, 147, 150, 151]. Simply the easiest way to evaluate symmetries is to evaluate the cofactors of a pair of variables. For any pair of variables $\{x_i, x_j\}$ in a Boolean function, there are four cofactors: $f_{\overline{x_i}\,\overline{x_j}}$, $f_{\overline{x_i}x_j}$, $f_{x_i\overline{x_j}}$, and $f_{x_ix_j}$. Obviously, the four cofactors of $\{x_i, x_j\}$ are the four columns in the partition with x_i and x_j in the bound set [106].

In all of the research on detecting symmetries in logic synthesis, the same set of definitions is given (with some variance in the naming):

Definition 5.5. *Given a function $f(x_0, \ldots, x_{n-1})$:*

1. *if $f_{x_i\overline{x_j}} = f_{\overline{x_i}x_j}$, f has a* **nonequivalent symmetry** *in $\{x_i, x_j\}$, or is symmetric in $\{x_i, x_j\}$,*

2. *if* $f_{x_i x_j} = f_{\overline{x_i}\,\overline{x_j}}$*, f has an* **equivalent symmetry** *in* $\{x_i, x_j\}$*, or is symmetric in* $\{\overline{x_i}, x_j\}$*,*

3. *if* $f_{x_i \overline{x_j}} = f_{\overline{x_i} x_j}$ *and* $f_{x_i x_j} = f_{\overline{x_i}\,\overline{x_j}}$*, f has a* **multiform symmetry** *in* $\{x_i, x_j\}$*,*

4. *if* $f_{x_i x_j} = f_{x_i \overline{x_j}}$ *or* $f_{\overline{x_i} x_j} = f_{\overline{x_i}\,\overline{x_j}}$*, f has a* **single-variable symmetry** *in* $\{x_j\}$*,*

In general, the block diagram for a two-variable bound set decomposition is shown in Figure 5.9.

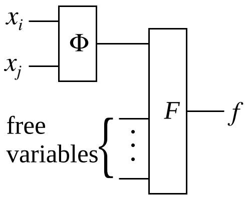


Figure 5.9: Block diagram of a two-variable bound set decomposition.

Example 5.7. *Figure 5.10 shows some basic types of symmetries using two-variable bound set partitions. Figure 5.10a shows a partition with a nonequivalent symmetry. Figure 5.10b shows a partition with an equivalent symmetry. Figure 5.10c shows a partition with a multiform symmetry. Multiform symmetry always results in the bound variables being decomposed as an XOR function, as shown in Figure 5.10d. Figure 5.10e shows a partition with a single-variable symmetry.*

The following defines a special set of decompositions when the function has a single-variable symmetry. $X - \{x_i, x_j\}$ is used to denote all variables in set X, excluding x_i and x_j.

1. if $f_{\overline{x_i}\,\overline{x_j}} = f_{\overline{x_i} x_j}$, f has decomposition $f = F(\Phi(x_i, x_j), X - \{x_i, x_j\})$, where Φ is either $x_i \cdot x_j$, $\overline{x_i} + x_j$, $\overline{x_i \cdot x_j}$, or $\overline{\overline{x_i} + x_j}$,

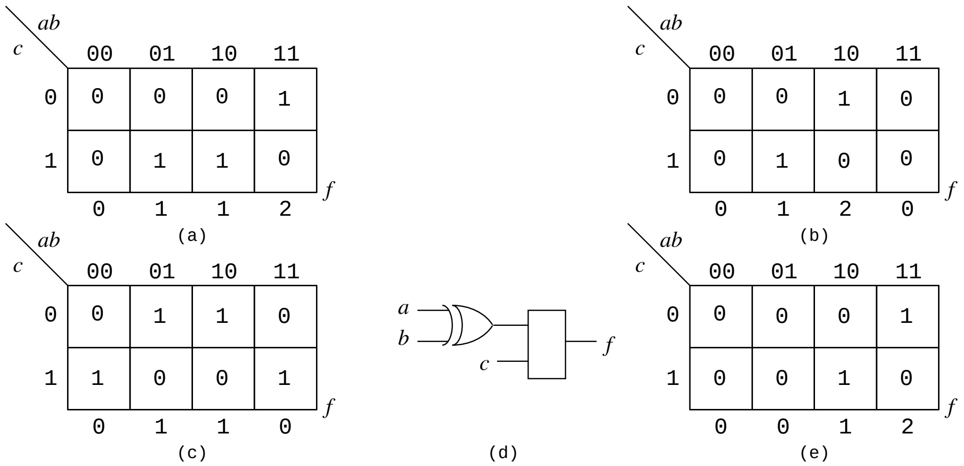


Figure 5.10: Symmetries in two-variable bound set partitions (a) nonequivalent symmetry (b) equivalent symmetry (c) multiform symmetry (d) multiform decomposition (e) single-variable symmetry.

2. if $f_{x_i\overline{x_j}} = f_{x_ix_j}$, f has decomposition $f = F(\Phi(x_i, x_j), X - \{x_i, x_j\})$, where Φ is either $x_i + x_j$, $x_i \cdot \overline{x_j}$, $\overline{x_i + x_j}$, or $\overline{x_i \cdot \overline{x_j}}$.

While the above decompositions have four different Φ functions based on encoding, the negated forms of Φ are redundant and are not used. The following example illustrates single-variable symmetry by using non-disjunctive partitions.

Example 5.8. *Figure 5.11a shows a partition $\{c\}|\{a,b\}$ of function f which has a single-variable symmetry, $f_{\overline{a}\,\overline{b}} = f_{\overline{a}b}$. The corresponding non-disjunctive partition $\{c\}|\{b\} : \{a\}$ is shown in Figure 5.11b. The non-disjunctive partition has a column multiplicity $\nu = 2$ (as defined in Section 4.1), and the two possible encodings are shown below the partition. The encoding $\{0001\}$ produces the function $a \cdot b$, and the encoding $\{1101\}$ produces the function $a + \overline{b}$.*

As shown in Example 5.8, a partition with $f_{\overline{x_i}\,\overline{x_j}} = f_{\overline{x_i}x_j}$, has two different encodings $(x_i \cdot x_j,\ \overline{x_i} + x_j)$. The block diagram of a single-variable symmetry is shown in Figure 5.12. The resulting composite function F has the same cardinality as the original function f. The following is a larger example of single-variable symmetry.

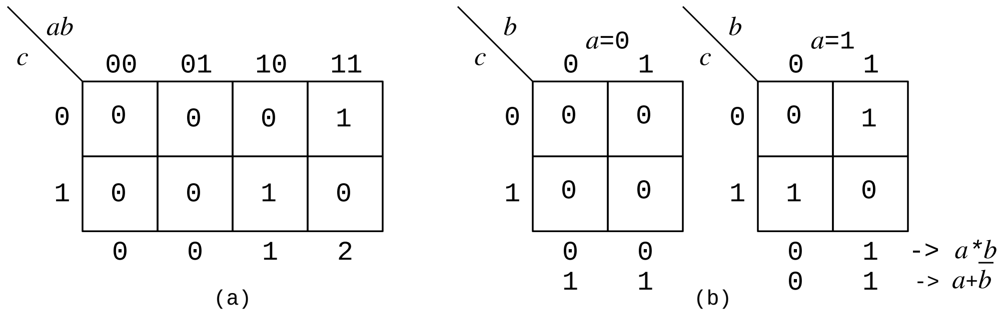


Figure 5.11: Single-variable symmetry example.

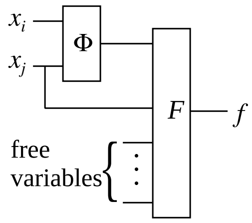


Figure 5.12: Block diagram of a single-variable symmetry.

Example 5.9. *Given the partition in Figure 5.13a of the function* $f = ((a \cdot b) + c) \oplus (a \cdot d)$. *The partition shows that* $f_{\overline{a}\,\overline{b}} = f_{\overline{a}b}$, *which implies a single-variable symmetry exists when* $a = 0$. *The function can be decomposed using the non-disjunctive partitions shown in Figure 5.13b. The decomposition results in two composite functions* $\Phi_1 = (a \cdot b)$ *and* $F(\Phi_1, a, c, d)$. *The partition* $\{c, \Phi_1\}|\{a, d\}$ *for* F *is shown in Figure 5.13c. The column multiplicity is two, which implies that the function can be decomposed into a new composite function* $\Phi_2 = (a \cdot d)$. *The row multiplicity is two, so a second composite function is possible,* $\Phi_3 = (c + \Phi_1)$. *The resulting decomposition is,* $f = \Phi_2 \oplus \Phi_3 = (a \cdot d) \oplus (c + \Phi_1) = (a \cdot d) \oplus (c + (a \cdot b))$.

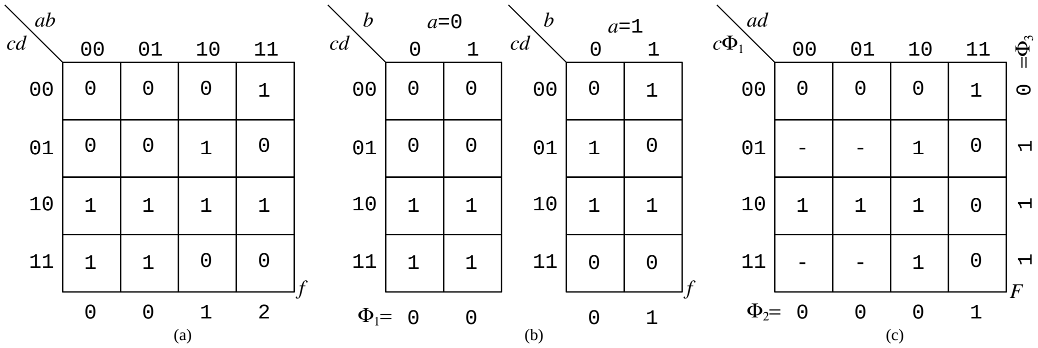


Figure 5.13: A larger single-variable symmetry example.

5.6 New Decomposition Method Summary

The new decomposition method presented in this chapter allows for decompositions that cannot be found when using standard Curtis' decomposition. A problem with the classical Curtis' decomposition is that Curtis' decomposition places constraints on the decomposition process to reduce the search space of complex decompositions. The main constraint is that a composite function cannot have the same number of inputs and outputs. As this chapter has pointed out, there are many small bound set partitions where the composite functions have the same number of inputs and outputs.

Also introduced in this chapter was the concept of multi-valued labeling of compatible columns, which is used to reduce the complexity of decomposition. That is, the decomposition process does not rely on the encoding of a partition. At the end of the decomposition, the multi-valued composite functions can be encoded to binary functions. The post-decomposition encoding is only necessary when the input function is a Boolean logic synthesis function. For machine learning functions, post-decomposition encoding is not needed.

The decomposition strategy proposed in this chapter is to only evaluate small bound set partitions. From Subsection 4.3.1, for an n-input function there are 2^n possible disjoint partitions or $O(2^n)$. If the size of the bound set is restricted to two variables, then there are $\binom{n}{2} = \frac{n\times(n-2)}{2}$ possible partitions or $O(n^2)$. For medium sized bound sets there are $\binom{n}{n/2} \approx \frac{2^n}{10}$ possible partitions or $O(2^n)$. For instance, given that there are 100 input variables, then there are approximately 1×10^{29}, 50-variable bound sets. The search space is much too large to go through all partitions with 50 bound set variables. Given 100 input variables there are $\binom{100}{2} = 4950$, two-variable bound sets. In fact, the number of all two-variable bound set partitions is $\binom{n}{2}$, i.e the algorithmic complexity is $O(n^2)$ and the complexity for all three-variable bounds sets is $O(n^3)$. Therefore, based on complexity, it is possible to evaluate all the bound sets with three or less variables.

Two small bound set decomposition types were presented in this chapter. Both of these decomposition types can be used to decompose a given function.

The next chapter presents decomposition algorithms that use the small bound set decompositions presented in this chapter. It also presents new MDD based decomposition algorithms which have runtimes that are much smaller than the Lai algorithm presented in Chapter 3. The decomposition strategy presented in the next chapter is based on evaluating only small bound set and a post-decomposition encoding algorithm. The small bound set decomposition types introduced in this chapter are used by this comprehensive decomposition strategy.

Chapter 6

Decomposition Algorithms

This chapter presents algorithms for decomposing Boolean, multi-valued, completely specified, unspecified, and highly unspecified functions. Because of the complex nature of the data/functions being represented a MDD package is used for the basic data structure of all the algorithms presented here.

Section 6.1 presents a set of algorithms used for partition selection as presented by the author in [52]. In Chapter 4, the partition selection algorithm called the *cut_level* algorithm was presented. To check a partition, the *cut_level* algorithm orders all the bound variables at the top of the decision diagram. The problem is that variable reordering in a decision diagram can lead to diagrams that have exponential size. Because of this exponential size, using the *cut_level* method can be very costly in terms of time. This chapter presents new algorithms that do not reorder the decision diagram to find and evaluate partitions. The new algorithms presented here are especially efficient when evaluating partitions with small bound sets. These new algorithms complement the functional decomposition method presented in the previous chapter, where small bound set decomposition is emphasized. Results of these new algorithms are compared to existing algorithms of Lai and Pedram.

Section 6.2 presents a set of algorithms that are based on the concept of minimal support. Minimal support is a method of finding the minimal set of input variables that defines the given function. This is a process of removing vacuous variables or variables that can be made vacuous by setting don't cares from a functional representation. Minimal support is used as a preprocessor to decomposition because it quickly reduces the complexity of decomposition by reducing the number of variables.

The post-decomposition encoding algorithm is presented in Section 6.3. The algorithm is based on the multi-valued labeling scheme and the small bound set decomposition methodology presented in the previous chapter. An advantage of the post-decomposition encoding algorithm presented here is that many encoding possibilities are evaluated for each decomposed block, while published decomposers generally only pick one encoding for each partition (decomposed block) during the decomposition process.

Also presented in this chapter is the development of an algorithm based on the small bound set decomposition types presented in the previous chapter. The algorithm ties all the concepts of the previous chapter with post-decomposition encoding, to produce a new decomposition algorithm. Experimental results on this decomposition algorithm are shown at the end of this chapter.

6.1 MDD Based Algorithms for Small Bound Set Decomposition

This section presents algorithms for MDD based decomposition. The *cut_level* algorithm presented in Chapter 4 evaluates partitions by moving variables up and down within the decision diagram. Given that variable reordering can cause large timing and size constraints, researchers looked for a method that would require very little reordering. The only algorithm that has been published, as far as the author knows, is the LPV evaluation algorithm proposed by Lai, Pedram and Vrudhula [78]. An advantage of the LPV algorithm is that the inclusion or exclusion of bound variables is done without reordering the variables within the BDD package.

This section is organized so that the LPV algorithm is presented in the first subsection. At the end of the first subsection the LPV algorithm is extended for multi-valued functions. The second subsection presents new MDD based algorithms without reordering. The last subsection compares the new algorithms with the algorithms of Lai and Pedram.

6.1.1 Lai and Pedram Algorithm without Reordering

The LPV algorithm determines the number of distinct columns by using *bit vectors* to represent each of the columns. A **bit vector** is defined as a set of Boolean values corresponding to the functional output values of a function in order. Recall that a column in a partition matrix is defined as the set of output values given a polarity (value) of each bound variable. Because the output values found within a column are a subset of all the functional outputs, each column may be represented by a bit vector. By encoding a bit vector to an integer value, the determination of column equality is done by comparing the integer-encoded value of each column. If the integer-encoded values of two columns are equal, then the two columns are equal. Because the integers can be very large (the size of each bit vector is exponential in the number of free variables), an edge-valued BDD is used to represent each of the integer-encoded values.

The LPV algorithm is based on a recursive algorithm that finds the integer-encoded values by traversing a BDD representation of a function. The input parameter to the recursive algorithm is the root vertex in the functional diagram. The recursive algorithm returns a vector of integer values. Each integer represents a column in the partition. The first integer-encoded in the vector represents the left-most column in the partition matrix, and the last integer-encoded represents the right-most column. The LPV pseudo code is shown in Algorithm 6.1.

Example 6.1. *Given is the partition matrix $\{a\}|\{b,c\}$ of $f(a,b,c)$ shown in Figure 6.1a. For explanation only, a decision tree is shown in Figure 6.1b, instead of a decision diagram. The LPV algorithm traverses the tree from the root vertex $a0$ to $b0$ to $c0$ (the path to $c0$ was through the 0-successors of $a0$ and $b0$). At the vertex $c0$, calling LPV on the 0- and 1-successors of $c0$ (both are terminal zero vertices) returns the values 0 and 0, respectively. Because the*

Algorithm 6.1 LPV pseudo code.

LPV(v: vertex)
  if v is a terminal vertex
    return terminal vertex value as an integer vector with one element
  let $\underline{X}$ and $\underline{Y}$ be integer vectors
  $\underline{X}$=LPV(0-successor of v)
  $\underline{Y}$=LPV(1-successor of v)
  let A be the set of free variables $\{x_j, \ldots, x_k\}$
  let x be the variable represented by v
  if x is a variable in the free set
    let I be a vector of integers $[0, 1, \ldots, |A| - 1]$
      the elements in I are a one-to-one mapping to each variable in A
    let $I(x)$ be the element in I that is mapped of x
    let $N = 2^{2^{I(v)}}$
    return integer vector $(N \times \underline{Y}) + \underline{X}$
  else x is a variable in the bound set
    let $\underline{Z}$ be an integer vector with size $|\underline{X}| + |\underline{Y}|$
      the first $|\underline{X}|$ elements in $\underline{Z}$ are equal to $\underline{X}$
      the last $|\underline{Y}|$ elements in $\underline{Z}$ are equal to $\underline{Y}$
    return integer vector $\underline{Z}$

vertex $c0$ *represents the bound variable c, a new integer vector with elements* $[0, 0]$ *is created. The traversal is continued until all vertices in the diagram have been traversed. Calling LPV on the 0- and 1-successors of vertex* $a0$ *returns the integer vectors* $[0, 0, 0, 1]$ *and* $[0, 0, 1, 1]$*, respectively. Because variable a is a free variable and the integer mapped to variable a is 0, the algorithm calculates* $[0, 0, 0, 1] + 2^{2^0} \times [0, 0, 1, 1] = [0, 0, 2, 3]$ *(also shown in Figure 6.1c). The column multiplicity is three, found by the number of different integer-encoded values in the vector* $[0, 0, 2, 3]$*.*

An advantage of the LPV algorithm is that the variables are not reordered in the decision diagram to check each bound set. A constraint on the LPV algorithm is that it encodes all output values in a given column, where the number of output values is exponential with the number of free variables. But, because of the decision diagram representation, the computation of output values is generally greatly reduced. According to Lai and Pedram, when comparing the

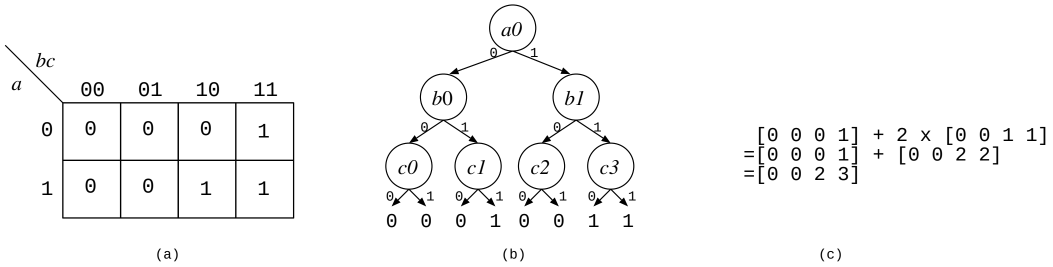


Figure 6.1: LPV example (a) partition $\{a\}|\{b,c\}$ (b) decision tree (c) LPV encoding.

LPV algorithm to that of the SIS [145] Roth-Karp decomposition algorithm [135] on finding the first decomposable form of many MCNC benchmark functions [91], the LPV algorithm is 17.2 times faster. In the case of finding all decomposable forms, the LPV algorithm is 35.4 times faster.

The following is an extension of the LPV algorithm to incompletely specified functions (note, this is not presented in [78]). Instead of using one integer-encoded value to represent a column, a pair of integer-encoded values are used. The pair of integer-encoded are the minimum and maximum values of a given column bit vector, denoted as [min, max]. The minimum and maximum integer-encoded values are obtained by forcing all don't cares in the bit vector to 0 and 1, respectively. For instance, given the cube of a column as $00-10-$, where a dash denotes a don't care value, the minimum and maximum values are [4,13]. The bit vector representations of 4 and 13 are 000100 and 001101. If a column does not have don't cares then min=max.

The pair [min, max] can be used to compute the corresponding cube C that represents the output values of a column. Two columns are compatible if their corresponding cubes have a non-empty intersection. Converting the integer values to cubes and checking if the two cubes intersect are linear procedures, but the cubes are exponential in the number of free variables. Because of the edge-valued BDD representation of the integer-encoded values, computation of the cubes is generally less than exponential.

To compare the new algorithm EVAL (presented in the next subsection) with the LPV algorithm, the LPV algorithm has been extended by the author for multi-valued functions. This extension is exactly the same as in the LPV algorithm for incompletely specified functions except that it uses powers of Q_f instead of powers of two, given that the functional output takes values from the set Q_f.

6.1.2 New Decomposition Algorithms Based on MDDs

The decomposition strategy proposed in this dissertation is the same as the strategy of Lai, Pedram and Vrudhula algorithms, where one algorithm (EVAL) is used to evaluate the column multiplicity of a partition and a second algorithm (PARTITION) is used to decompose the function. Two algorithms are used because the algorithm to determine column multiplicity is much faster than the algorithm for decomposing a function given a partition. Both algorithms were developed for large Boolean/multi-valued, completely/incompletely specified functions to check small bound sets.

The first algorithm, called EVAL determines column incompatibilities while traversing a functional graph by using an incompatibility graph to store the incompatibilities between columns [121]. While traversing a functional graph, the EVAL algorithm extracts sets of partition matrix rows. Each set of rows is a subset of all rows in a partition matrix and pairs of columns can be checked for incompatibilities over the subset of rows. When the EVAL algorithm completes its traversal, the columns have been checked for incompatibilities over all rows of a partition matrix. Incompatibility of two columns is determined by simultaneously traversing the graphs that represent the columns over a given set of rows. If the traversals reach terminal vertices that have different values and neither vertex is a don't care, then the columns are incompatible as defined in Definition 4.12.

Before running the EVAL algorithm, an incompatibility graph is created with no edges, and each vertex in the graph is associated with a column in the partition matrix. If two columns are found to be incompatible over a given set of partition matrix rows, an edge is placed in the incompatibility graph between the two vertices that represent the two columns. If two columns are found to be incompatible during the traversal, the columns are not checked for incompatibility over any remaining set of rows. In the worst case (two columns are compatible), columns are checked for incompatibilities over all possible row values. In the best case, the columns are found to be incompatible for the first row in the partition matrix, and only one row is evaluated.

The EVAL algorithm extracts a set of rows from the decision diagram by using a *depth-first* algorithm that simultaneously traverses sub-graphs of the graph representation of f. The number of sub-graphs traversed equals the number of columns, $|Q_B|$, in the partition matrix. The algorithm starts by traversing the functional diagram of f, until a vertex that represents a bound variable, x_i, is reached. The traversal is continued by simultaneously traversing each of the $|Q_i|$ successors of the vertex. When the traversals reach vertices that represent a second bound variable, x_j, the traversal is continued by simultaneously traversing each of the $|Q_j|$ successors of each vertex. At this point, the number of sub-graphs being traversed simultaneously is $|Q_i| \times |Q_j|$. The traversal is continued until the number of sub-graph traversals is $|Q_B|$. Each of the $|Q_B|$ sub-graphs represents a column over a set of rows, and therefore, pairs of sub-graphs are checked for incompatibilities. The traversal routine then continues to find the next set of rows. The EVAL pseudo code is shown in Algorithm 6.2.

Example 6.2. *Given the Boolean function $f(a, b, c, d)$, the partition $\{c, d\}|\{a, b\}$ is shown in Figure 6.2a, the corresponding decision tree in Figure 6.2b is used to explain the EVAL algorithm. First, an incompatibility graph is created with four vertices and no edges, the vertices represent each of the columns $\{00, 01, 10, 11\}$ in the partition matrix and the edges between vertices represent incompatibilities between the columns.*

The EVAL algorithm is started by traversing the decision tree representation of f. The root vertex, c0, does not represent a bound variable, so EVAL is called

Algorithm 6.2 EVAL pseudo code.

EVAL(vector of MDD vertices: $\underline{S}$)
  let x be the variable represented by the set of vertices in $\underline{S}$
  let $|Q_x|$ be the number of cofactors of variable x
  let $\underline{S}_i$ be a vector of vertices with the i-th successor of each vertex in $\underline{S}$
  let $[\underline{S}_j, \underline{S}_k]$ be a vector of vertices where $\underline{S}_j$ and $\underline{S}_k$ are subsets
  if x is a bound variable
    call EVAL($[\underline{S}_0, \underline{S}_1, \ldots, \underline{S}_{|Q_x|-1}]$)
  else x is a free variable
    if $|S| = |Q_B|$
      $\underline{S}$ denotes all the columns in the partition matrix
      for all pairs $\{s_i, s_j\} \in \underline{S}$
        if there is no edge between s_i and s_j in the incompatibility graph
          if elements $\{s_i, s_j\}$ are incompatible columns
            add an edge in the incompatibility graph between s_i and s_j
    else
      for $i = 0$ to $|Q_x| - 1$
        call EVAL($\underline{S}_i$)

with the 0-successor of $c0 : [d0]$. *d is not in the bound set, so EVAL is called with the 0-successor of* $d0 : [a0]$. *Variable a is in the bound set, so EVAL is called with both successors of* $a0 : [b0, b1]$. *Variable b is in the bound set, so EVAL is called with all four successors of* $[b0, b1] : [1, 0, 0, -]$ *(shown in the figure as S'). The number of vertices in vector S is the same as the number of columns in the partition, so the vertices in vector S are checked for incompatibility. Edges are placed in the incompatibility graph between the pairs of columns* $\{00, 01\}$ *and* $\{00, 10\}$ *because the first element in S is incompatible with the second and third elements in S. The algorithm then moves back to the vertex $d0$, and then traverses to the 1-successor of* $d0 : [a1]$. *Because variables a and b are bound variables, EVAL is iteratively called with both successors of* $a1 : [b2, b3]$, *and then all four successors of* $[b2, b3] : S = [-, 1, 1, 0]$. *Edges are placed between the pairs of columns* $\{01, 11\}$ *and* $\{10, 11\}$ *in the incompatibility graph (shown in Figure 6.2c). By traversing the remaining vertices in the diagram, the vectors* $[0, -, 1, 0]$ *and* $[-, 0, 0, 1]$ *are checked for incompatibility. Because no additional*

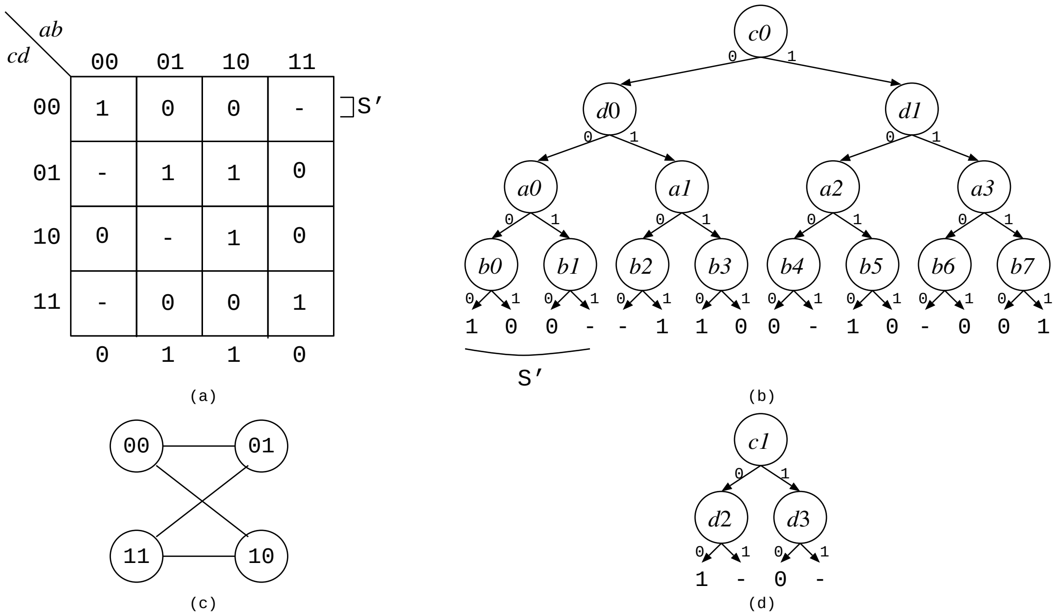


Figure 6.2: EVAL and PARTITION algorithm examples (a) partition matrix $\{c, d\}|\{a, b\}$ (b) decision tree (c) EVAL incompatibility graph (d) 00 column returned by PARTITION.

incompatibilities exist, the incompatibility graph is not changed. After EVAL traverses all vertices in the diagram, graph coloring of the incompatibility graph determines that the column multiplicity of this partition is two. Notice that ν=2 *is also true for the partition matrix shown in Figure 6.2a.*

To decompose a function given a partition, the PARTITION algorithm is used. The PARTITION algorithm returns decision diagrams that represent each of the columns in the partition matrix. Algorithm 6.3 shows the recursive procedure that returns the left-most column in the partition matrix, by only traversing the 0-successor of each bound variable. The full version of the PARTITION algorithm simultaneously calculates all columns in the partition matrix.

Example 6.3. *Given the decision tree in Figure 6.2b, and bound set* $\{a, b\}$*, the PARTITION algorithm starts at* $c0$*, traverses to vertex* $d0$*, then down to vertex* $a0$ *(traversing the 0-successor of each vertex). Vertex* $a0$ *represents a bound*

Algorithm 6.3 PARTITION pseudo code to find the left-most column.

PARTITION(v : vertex)
  if v is a terminal vertex
    return v
  let x be the variable represented by v
  let $|Q_x|$ be the number of cofactors of variable x
  let v_i be the i-th successor of the vertex v, $0 \leq i < |Q_x|$
  if x is a bound variable
    return PARTITION(v_0)
  else x is a free variable
    return a new vertex with successors [PARTITION(v_0), PARTITION(v_1),…,
      PARTITION($v_{|Q_x|-1}$)]

variable so traverse the 0-successor of $a0 : b0$*. The 0-successor of* $b0$ *is a terminal vertex-1, so a 1 is returned. PARTITION moves back to* $d0$ *and then traverses the 1-successor of* $d0$ *down to the 0-successor of* $b2$ *: don't care terminal vertex. PARTITION returns '-' and moves back to* $d0$*, creating a new vertex,* $d2$ *vertex with successors* $[1, -]$*. Calling PARTITION on the 1-successor of* $c0$ *returns a new vertex* $d3$ *with successors* $[0, -]$*. Because* $c0$ *is a free variable, a new vertex,* $c1$ *is created with successors* $[d2, d3]$*. The new diagram that represents the* 00 *column in the partition matrix is shown in Figure 6.2d.*

Because PARTITION and EVAL do not perform any variable swapping, the algorithms are fairly fast. The EVAL and PARTITION algorithms are limited to small bound sets because the number of columns is exponential with respect to the number of variables in the bound set. The restriction of small bound sets is used in many decomposition algorithms, including FPGA synthesis [76] and decomposition for machine learning decomposition [50, 155]. Plus, the decomposition scheme presented in this dissertation is based on small bound sets. The LPV algorithm does not have the same restriction, but recall that finding the column multiplicity of an incompletely specified function is a NP-hard problem with respect to the number of columns in the partition. To reduce the complexities of finding the column multiplicity of a partition, the bound set size should be restricted, which in essence also restricts the bound set sizes used by the LPV algorithm.

6.1.3 MDD Based Decomposition Experimental Results

Tables 6.1 and 6.2 compare the runtimes of the different algorithms presented in this chapter on different benchmark sets using a MDD package. The benchmark sets used are the MCNC benchmarks [91], and the multi-valued benchmarks [47]. The benchmarks are separated in the tables for readability and to show the difference of timing between each benchmark type. In the tables, the MCNC benchmarks are at the top, and the multi-valued benchmarks are on the bottom. The MCNC benchmark functions are multi-input and multi-output, completely specified functions. The multi-valued benchmarks are all single-output, incompletely specified, multi-valued input and multi-valued output functions.

The units for time are in seconds. Any element in the table that begins with ">" designates that the program did not complete in the allotted time. A "-" denotes that the number of bound variables was larger than the number of input variables and a decomposition could not be performed. The MDD package used in the decomposition of these functions was developed at Portland State University.

Table 6.1 compares the EVAL and the LPV algorithms. To make the comparison, the author implemented a multi-valued version of the LPV algorithm. The EVAL and LPV algorithms were run using variable bound set sizes of two, four, six, and eight. For each bound set size, the algorithms were run on 100 random partitions, and the time to evaluate all 100 partitions is displayed in the table. The set of random partitions was the same for both algorithms. The comparison of these algorithms is the runtime to find 100 partitions; this does not include the time to determine the column multiplicity of each partition. In all cases of two-variable bound sets, the EVAL algorithm is much faster than the LPV algorithm. As the size of the bound sets increases, the LPV algorithm starts to perform better than the EVAL algorithm. This is caused by the num-

	EVAL				LPV			
filename	2	4	6	8	2	4	6	8
9sym	0.1	1.0	10.0	94.0	3.0	3.0	8.0	49.0
alu2	0.1	3.0	33.0	285.0	10.0	13.0	25.0	124.0
b12	1.0	3.0	21.0	288.0	713.0	285.0	180.0	499.0
frg1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
sao	1.0	3.0	17.0	159.0	9.0	10.0	15.0	35.0
spla	5.0	20.0	104.0	>1000	>1000	>1000	1106	>1000
t481	0.1	3.0	46.0	514.0	>1000	>1000	877.0	>1000
table3	4.0	28.0	173.0	882.0	384.0	237.0	212.0	431.0
pal2	0.1	0.1	2.0	-	1.0	1.0	3.0	-
c1b	0.1	0.1	1.0	-	1.0	1.0	2.0	-
c2b	1.0	7.0	153	>1000	>1000	>1000	>1000	>1000
c3b	2.0	10.0	279	>1000	>1000	>1000	>1000	>1000
d1	0.1	1.0	9.0	-	>1000	>1000	>1000	-
d2	3.0	24.0	>1000	>1000	>1000	>1000	>1000	>1000
d3	4.0	31.0	>1000	>1000	>1000	>1000	>1000	>1000

Table 6.1: Time comparisons of the EVAL and LPV algorithms.

ber of simultaneous traversals in the EVAL algorithm. When comparing the EVAL and LPV algorithms on the multi-valued benchmark functions, the run-times of the LPV algorithm were found to be very high. The reason for the large amount of time is that each of the functional inputs are multi-valued, which greatly increases the number of possible output combinations for the function. In fact, the LPV algorithm does not complete in the specified time for most of the multi-valued benchmark functions, while the EVAL completes for all two- and four-variable bound sets.

Table 6.1 shows that the two algorithms are almost equivalent for the binary, completely specified functions, but for multi-valued functions the EVAL algorithm performs much better than the LPV algorithm. The algorithmic complexity of the LPV algorithm encoding depends on the number of row output values for a given column; specifically, the complexity is exponential in the number of free variables. Once the LPV algorithm is completed, each integer-encoded value must be decoded to determine column compatibilities, which is based on

filename	init. size	PARTITION time	PARTITION size	*cut_level* time	*cut_level* size	$\frac{cut\_level}{PARTITION}$ size%
9sym	41	0.1	70	0.1	70	100.0
alu2	264	2.0	517	2.0	456	88.3
b12	100	3.0	161	6.0	115	71.5
frg1	230	25.0	415	441.0	19411	4677.4
sao	164	1.0	311	1.0	305	98.1
spla	630	15.0	1637	22.0	1331	81.4
t481	47	3.0	134	7.0	154	115.0
table3	951	15.0	2075	24.0	2297	110.7
pal2	26	0.1	42	0.1	34	81.0
c1b	21	0.1	32	0.1	32	100.0
c2b	100	2.0	182	2.0	182	100.0
c3b	133	4.0	245	7.0	245	100.0
d1	31	0.1	51	0.1	51	100.0
d2	152	2.0	291	2.0	287	98.7
d3	217	8.0	419	8.0	415	99.1

Table 6.2: Time comparisons of the PARTITION and *cut_level* algorithms.

the number of free variables for the decoding and the number of bound variables for checking for column compatibility. The time complexity of the EVAL algorithm is exponential in the number of bound variables. Consequently, for small bound sets, the EVAL algorithm should perform well, even for very large multi-valued functions. The reason that EVAL did not complete in the specified time with eight-variable bound sets for most of the benchmarks, is that the number of columns in each partition is greater than 20 thousand.

Table 6.2 compares the PARTITION and *cut_level* algorithms. To compare the two algorithms, all two-input bound set partitions of the functions were evaluated. The table shows the initial size (number of vertices in the MDD) before starting the decomposition process. PARTITION and *cut_level* were compared by showing the time to evaluate all two-input bound set partitions and the worst-case size (number of vertices). The two algorithms were also compared on the worst-case number of vertices in the MDD when evaluating a partition. The column "size%" gives the size percentage of the two algorithms. Table 6.2 shows that the PARTITION algorithm is always faster than the *cut_level* algorithm. When comparing the decision diagram sizes created by the two algorithms, the sizes are generally the same, except for the benchmark function *frg1*. This function fits in the category of reordering that results in a decision diagram of exponential size. While the PARTITION and *cut_level* algorithms usually result in the same size decision diagrams and the same amount of time to complete the task, there are situations where the *cut_level* algorithm results in decision diagrams that are much larger than the PARTITION algorithm.

6.2 Minimal Support

The minimal support problem consists of finding a minimal set of input variables for a given function. The constraint is that the minimal set of input variables must preserve the functionality of the original function. That is, all information that is required to compute the original function must be preserved.

There are actually three different classes of variables as defined in [61]. A variable is either *essential*, *inessential* or *vacuous*. **Vacuous** variables are defined as variables that do not have an impact on the output. **Essential** variables are defined as variables that have an impact on the output. **Inessential** variables are defined as variables that by setting don't cares to values, the variable either becomes essential or vacuous. Because of the structure of a MDD, if a variable is vacuous it is no longer included in the diagram, so vacuous variables are automatically removed from the MDD representation of the function. For functions represented by cubes, finding vacuous variables is not trivial, and is a NP-hard problem.

Theorem 6.1. *Given is a function with inessential variables x_i, x_j, x_k. If the pairs of variables: $\{x_i, x_j\}$, $\{x_i, x_k\}$, and $\{x_j, x_k\}$ can be made vacuous by setting don't cares, then it does not imply that all variables in the set $\{x_i, x_j, x_k\}$ can be made vacuous.*

Proof. The theorem is proved by the following example:

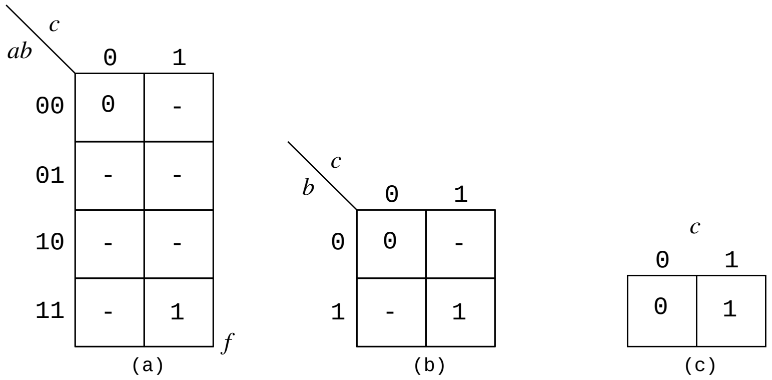


Figure 6.3: Example of removing inessential variables (a) original function (b) variable a removed (c) variable b removed, variable c is no longer inessential.

Example 6.4. *Given the function shown in Figure 6.3a, variables* $\{a, b, c\}$ *are all inessential variables. It is possible to find three different functions by making any single variable vacuous (by setting don't cares), but it is impossible to make all three variables* $\{a, b, c\}$ *vacuous. This is shown by Figure 6.3, where variable* a *is first made vacuous by setting don't cares, which gives the function shown in Figure 6.3b. The variable* b *is then made vacuous which gives the function in Figure 6.3c, where the variable* c *is an essential variable and cannot be made vacuous.*

$\square$

To determine which variables can be made vacuous, a compatibility graph algorithm is used. Each node in the graph represents an inessential variable, and each vertex between two nodes designates that both variables can be forced vacuous at the same time. From Theorem 6.1, a conventional graph cannot be created (a graph where it is assumed that if $\{x_i, x_j\}$, $\{x_i, x_k\}$, and $\{x_j, x_k\}$ can be made vacuous by setting don't cares, then all variables in the set $\{x_i, x_j, x_k\}$ can be made vacuous). For this reason, conventional maximum clique and graph coloring algorithms cannot be used, and new algorithms must be created. This

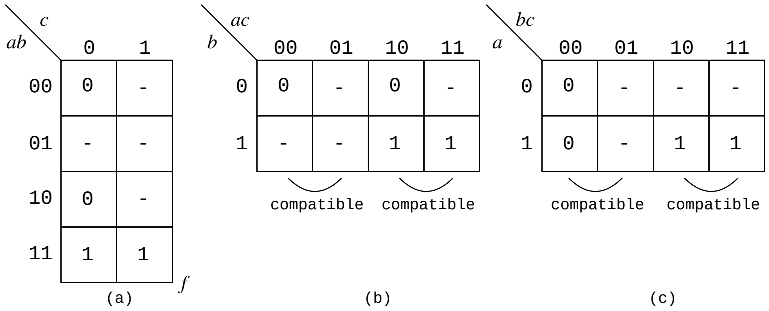


Figure 6.4: Decomposition with an inessential variable (a) variable c is inessential (b) partition $\{b\}|\{a, c\}$ (c) partition $\{a\}|\{b, c\}$.

section explores the existing algorithms and new algorithms for solving the minimal support problem.

The reason that the minimal support problem is being explored in this dissertation is because of its relevance to functional decomposition. By finding the minimal support of a function, the complexities of decomposing the function are reduced. Example 6.5 illustrates the difficulty in decomposing strongly unspecified functions.

Example 6.5. *Given the function shown in Figure 6.4, variable c is an* inessential *variable. Because variable c is an* inessential *variable, any partition with variable c in the bound set will always have column multiplicity* $\nu \leq |Q_B|/|Q_c|$, *where* $|Q_c|$ *is the number of values for the variable c and* $|Q_B|$ *is the number of columns in the partition. This is shown in Figures 6.4b,c, where the columns that are labeled compatible below the partitions are compatible because variable c is an inessential variable.*

The example shows that if inessential variables are used in the decomposition process, there are many compatible columns, not only the columns that are made compatible by the inessential variable. By not removing inessential variables the complexity of the decomposition is greatly increased.

term symbols	Inputs x_0	x_1	x_2	x_3	x_4	x_5	Outputs y_0	y_1	y_2	y_3
0	0	-	0	0	0	1	0	1	1	0
1	0	0	0	1	0	-	1	1	-	0
2	1	-	0	0	1	-	1	1	1	0
3	1	1	1	1	1	-	0	-	0	1
4	-	0	1	-	0	0	-	0	-	0

Figure 6.5: Incompletely specified multi-output Boolean function [67].

6.2.1 Previous Minimal Support Algorithms

For incompletely specified Boolean functions, an elegant approach to minimal support was proposed by Lin [82]. Lin's approach to minimal support uses a bottom-up traversal of BDDs to implicitly enumerate all feasible supports. Unfortunately, this approach cannot be generalized to multi-valued functions.

To the best of the author's knowledge, most of the work published on the minimal support problem for multi-valued functions has been done by Jozwiak [67, 68]. Jozwiak also claims to have the best algorithms for solving the minimal support problem. This subsection presents a set of definitions and examples found in [67]. Jozwiak uses these definitions, based on cube functional representations, to determine the minimal support of a given function.

Definition 6.1. *A* **Set System** (SS) *on a set S is a collection of nonempty and distinct subsets $B_0, B_1, \ldots, B_{k-1}$ of S, such that:*

$$\bigcup_i B_i = S \text{ and } B_i \not\subset B_j \text{ for } i \neq j.$$

Definition 6.2. *An* **Information Set** (IS) *on two elements $\{s_i, s_j\} \in S$ where $s_i \neq s_j$ is defined as $IS = \{\{s_i, s_j\}|\ s_i$ is distinguished from s_j by the modeled information$\}$.*

Example 6.6. *Given the multi-output Boolean function in Figure 6.5, the following SSs and ISs can be found for each of the outputs. Note, each of the* *numbers represented in the following sets correspond to each of the term symbols in Figure 6.5.*

$$
\begin{array}{ll}
SS(y_0) = \{\{0,3,4\};\{1,2,4\}\} & IS(y_0) = \{0|1, 0|2, 1|3, 2|3\} \\
SS(y_1) = \{\{3,4\};\{0,1,2,3\}\} & IS(y_1) = \{0|4, 1|4, 2|4\} \\
SS(y_2) = \{\{1,3,4\};\{0,1,2,4\}\} & IS(y_2) = \{0|3, 2|3\} \\
SS(y_3) = \{\{0,1,2,4\};\{3\}\} & IS(y_3) = \{0|3, 1|3, 2|3, 3|4\}
\end{array}
$$

The minimal support of a function is found by analyzing the relationships between the information required for computing the output values and the information provided by a particular input variable. The concept is to find the variable that gives the most information in the determination of the functional output.

Definition 6.3. Common Information (CI) *on two sets SS_1 and SS_2 is defined as $CI(SS_1, SS_2) = IS(SS_1) \cap IS(SS_2)$.*

Example 6.7. *Given the multi-output Boolean function in Figure 6.5, the following common information can be found, given that $Y = \{y_0, y_1, y_2, y_3\}$.*

$$
\begin{array}{ll}
CI(Y, x_0) = \{0|2, 0|3, 1|3\} & |CI(Y, x_0)| = 3 \\
CI(Y, x_1) = \{0|3, 1|3, 3|4\} & |CI(Y, x_1)| = 3 \\
CI(Y, x_2) = \{0|3, 0|4, 1|3, 1|4, 2|3, 2|4\} & |CI(Y, x_2)| = 6 \\
CI(Y, x_3) = \{0|1, 0|3, 2|3\} & |CI(Y, x_3)| = 3 \\
CI(Y, x_4) = \{0|2, 0|3, 1|3, 2|4, 3|4\} & |CI(Y, x_4)| = 5 \\
CI(Y, x_5) = \{0|4\} & |CI(Y, x_4)| = 1
\end{array}
$$

Given that $|CI(Y, x_2)| > |CI(Y, x_4)| > |CI(Y, x_0)| = |CI(Y, x_1)| = |CI(Y, x_3)| > |CI(Y, x_5)|$, then x_2 gives more information required to compute Y than x_4, x_4 more than x_0, etc. In this example, x_2 does not cover the output Y of the function completely, and therefore at least one more variable is needed to completely define the function.

Each CI value is a weighted measure that can be used to determine the minimal support of a function. Note, that using CI is not an exact measure because a cube representation of a function is not a canonical representation of a function. Therefore, many different cube representations can be used to represent the same function, and thus, for each cube representation, CI values can be different.

In both of his papers, Jozwiak uses the CI weighted measures to help initialize a Genetic Algorithm (GA). The type and implementation of the GA is not discussed in either paper, and the benchmark functions that are used are not public domain; therefore it is impossible to compare to his results. Plus, the benchmark functions that he uses have a very small number of cubes, which vastly improves his results. The next subsection presents an implementation of the Jozwiak algorithm using a MDD and a depth-first maximal clique algorithm.

6.2.2 New Minimal Support Algorithms

The proposed solution to the minimal support problem is to use two things, a decision diagram package and a maximal clique algorithm. Using a decision diagram implementation of a function has the advantage over a cube representation, in that vacuous variables and essential variables can be determined very easily. If a variable is vacuous in the decision diagram, then there are no vertices in the diagram to represent that variable. If a variable is essential, then it is not possible to make the variable vacuous by setting don't cares in the function. That is, cofactors of the variable are not compatible, which is easily checked in a decision diagram.

The maximal clique algorithm, called Top_Down is used to find the maximal number of vacuous variables, i.e., the number of inessential variables forced to be vacuous. The Top_Down algorithm is a depth-first search algorithm that uses a heuristic algorithm to determine the order in which each inessential variable is made vacuous. Several different heuristic algorithms were tested and the results are shown in Subsection 6.2.3. The algorithm iteratively forces inessential variables to be vacuous, returning the "best" function with the fewest essential variables (the most vacuous variables).

Algorithm 6.4 Top_Down minimal support maximal clique pseudo code

Top_Down(function f)
  determine all inessential variables in f
  if no inessential variables exist
    if number of vacuous variables is greater than previous best
      best = current function
    return
  else
    order inessential variables using a heuristic from best to worst
    for each inessential variable e in ordered heuristic list
      F= f with variable e forced vacuous
      Top_Down(F).

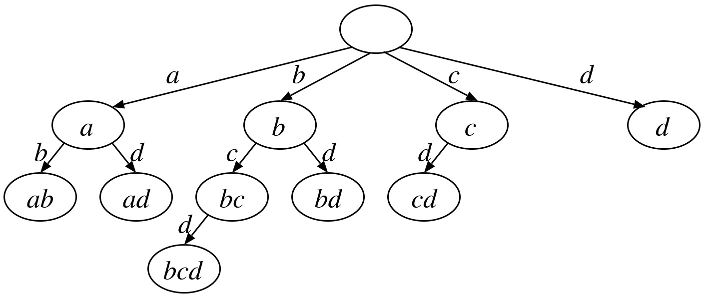


Figure 6.6: Depth-first search algorithm example.

Example 6.8. *Figure 6.6 shows a graph that is used to illustrate the depth-first graph traversal used in the Top_Down algorithm. In the graph, starting at the top vertex, variable a is forced vacuous as shown on the edge labeled a to reach the vertex representing a new function where variable a is vacuous. Variable b is then forced vacuous to create a new function where variables a and b are vacuous. No other variables can be made vacuous, so the traversal backtracks to the vertex-a, where the variable d is made vacuous, which creates the new function where variables a and d are vacuous.*

The traversal is continued until all possible combinations of forcing inessential variables to be vacuous have been explored. The graph shows that the best solution *is found when variables b, c, and d are all forced vacuous.*

For comparisons, the Jozwiak algorithm has been implemented using a MDD package. The decision diagram package can be used to determine Common Information CI by representing a cube as a path in the diagram from the root vertex to a terminal vertex. To determine the CI for the output(s) of the function all paths from the root to the terminal vertex are explored. The application of this in the decision diagram is a single traversal of all the vertices in the diagram and is very fast.

Algorithm 6.5 Jozwiak minimal support maximal clique algorithm

```
Jozwiak(function f, current essential variables v)
  if current essential variables completely represent f
    if number of essential variables is less than previous best
      best = current function with current essential variables
    return
  else
    order inessential variables using a heuristic from best to worst
    for each inessential variable e in ordered heuristic list
      F= f with variable e forced essential
      Jozwiak(F).
```

6.2.3 Minimal Support Experimental Results

Results of the Top_Down algorithm and the Jozwiak algorithms are shown in Table 6.3. The first column gives the name of the benchmark function; the column labeled "in" is the number of inputs for the function, "sol. size" is the minimum number of essential variables to represent the given function. Two categories are given in the table: Top_Down and Jozwiak that represent the respective algorithms. Under both categories "random", "fifo" and "CI" columns are given which state the time to run different heuristics for sorting the variables for the given algorithm. "random" is a random sort heuristic, "fifo" is a sort heuristic that sorts the inessential variables in the same order as the initial variable order given the benchmark function; "CI" uses Common Information to sort the variables. Under the Jozwiak category the column "pure CI" is added to represent a pure Jozwiak heuristic that cannot determine whether a variable is essential, inessential or vacuous. This should represent, more accurately, how fast the Jozwiak algorithm would run using cube representation of the functions, not the MDD representation.

The units for time are in seconds. Any element in the table that begins with ">" designates that the program was aborted after not completing in the allotted time. A "?" denotes that the smallest solution size is not known and that the number given is the smallest returned value found when running the algorithms.

Table 6.3 shows that in most cases the Top_Down algorithm is faster than the Jozwiak algorithms. This includes the benchmark *mushroom* where the Jozwiak algorithms do not find a solution in the completion time while the Top_Down algorithm does. There are cases, though, when the Jozwiak algorithms are faster, including the benchmarks *d8, d9* and *d10*. Surprisingly, in many instances the CI heuristic used for either algorithm is slower than the random heuristic used. Also, notice the time difference of the CI and the pure CI heuristics while using the Jozwiak method. This illustrates the importance of evaluating the state of each variable of inessential, essential, or vacuous. Table 6.3 shows that in most cases, the Top_Down algorithm using the CI heuristic is generally the best method of solving the minimal support problem.

6.3 Post-Decomposition Encoding

The decomposition methodology presented in this dissertation is based on the new concept of multi-valued labeling instead of using Boolean encoding. In the case of decomposing Boolean functions for logic synthesis, the multi-valued composite functions generated during the decomposition must be encoded to Boolean functions. This section presents the new concept of post-decomposition encoding and a heuristic algorithm for it.

Basically the concept of post-decomposition encoding is the following: given

filename	in	sol.	Top Down random	Top Down fifo	Top Down CI	Jozwiak random	Jozwiak fifo	Jozwiak CI	Jozwiak pure CI
audiology	69	15?	>2000	>2000	>2000	>2000	>2000	>2000	>2000
balance	4	4	0.1	0.1	0.1	0.8	0.8	0.8	0.6
breastc	9	5	92.5	58.9	94.6	92.6	89.7	106.0	234.9
bridges1	9	9	0.4	0.1	0.1	0.7	0.7	0.7	67.1
bridges2	10	10	0.7	0.1	0.1	1.0	1.0	1.0	193.1
car	6	6	0.1	0.1	0.1	0.2	0.2	0.2	3.2
chess1	6	6	0.1	0.1	0.1	9.0	9.0	9.1	156.6
chess2	36	29	56.4	49.4	41.7	76.3	76.0	75.5	>2000
cloud	6	6	0.1	0.1	0.1	0.4	0.4	0.4	9.8
connect-4	42	34?	>2000	>2000	>2000	>2000	>2000	>2000	>2000
employ1	9	9	0.2	0.1	0.1	0.2	0.2	0.2	29.6
employ2	7	7	0.1	0.1	0.1	0.1	0.1	0.1	7.7
flag	28	7?	>2000	>2000	>2000	>2000	>2000	>2000	>2000
flare1	10	10	0.7	0.1	0.1	1.4	1.4	1.4	271.4
flare2	10	9	0.1	0.9	0.9	2.4	2.9	2.6	324.2
house-votes	16	16	0.2	0.1	0.1	1.8	1.9	1.8	>2000
letter	16	15?	>2000	>2000	>2000	>2000	>2000	>2000	>2000
lung-cancer	56	4?	>2000	>2000	>2000	>2000	>2000	>2000	>2000
monks1tr	6	3	0.1	0.8	0.9	0.1	0.1	0.1	2.7
monks2tr	6	6	0.2	0.1	0.1	0.3	0.3	0.3	5.7
monks3tr	6	4	0.1	0.4	0.5	0.2	0.2	0.2	3.4
mushroom	22	4	1277.7	544.0	698.0	>2000	>2000	>2000	>2000
post-op	8	8	0.4	0.1	0.1	0.4	0.4	0.4	23.4
programm	12	12	4.3	0.7	0.7	100.3	100.5	97.4	>2000
sensory	11	5	30.9	25.0	35.2	391.0	375.6	600.4	1321.5
shuttlem	6	6	0.1	0.1	0.1	0.5	0.5	0.5	1.4
sleep	9	5	17.2	9.3	7.6	9.6	6.1	11.4	168.1
sponge	44	3	>2000	>2000	>2000	1736.2	>2000	>2000	>2000
tic-tac-toe	9	8	1.3	0.4	0.4	212.1	213.3	250.1	250.2
trains	32	1	14.4	15.5	15.3	23.4	6.2	0.3	0.3
zoo	16	5	10.5	10.2	14.9	36.6	44.4	26.5	1001.7
alet	18	17	24.0	13.7	9.6	8.9	8.9	9.3	>2000
c2a	11	2	1.1	1.4	1.8	3.6	9.2	3.9	3.7
c2b	11	3	4.2	2.9	6.2	6.8	6.5	7.1	7.9
c3a	14	2	5.2	4.3	4.9	7.8	14.9	9.9	9.8
c3b	14	3	12.1	8.5	19.7	13.5	14.4	15.7	15.8
c4a	14	2	5.0	4.2	7.7	12.4	12.7	11.3	11.2
c4b	14	3	12.9	9.6	19.6	13.8	14.0	16.1	16.2
c5a	13	2	4.7	3.8	5.5	4.9	4.7	2.5	2.4
c5b	13	2	4.4	2.0	5.8	1.9	3.6	2.5	2.4
c6a	13	2	6.3	4.5	7.5	3.8	2.1	2.6	2.7
c6b	13	2	5.4	4.9	7.2	2.4	3.3	2.5	2.4
d2	11	4	1.3	1.2	1.4	1.3	1.4	1.5	34.7
d3	14	4	17.8	15.7	24.7	95.1	125.9	97.9	98.4
d4	14	3	19.9	13.2	22.4	29.0	28.3	36.5	36.3
d5	13	2	4.4	4.3	9.1	9.6	18.2	3.9	3.7
d6	13	2	12.2	9.3	14.9	10.7	24.0	5.2	4.6
d7	24	2	184.8	88.4	119.4	129.9	82.2	17.3	17.5
d8	32	2	1276.7	271.4	352.8	125.2	130.1	31.4	31.2
d9	34	2	372.9	343.8	436.1	283.6	199.6	35.2	35.9
d10	37	2	617.1	477.1	616.4	329.6	310.6	41.1	41.2
geo	11	6	57.5	35.3	75.3	156.2	174.4	157.2	1505.2

Table 6.3: Top_Down algorithm comparison with Jozwiak's algorithm.

a net-list of decomposed blocks, each with multi-valued inputs and/or output, encode the multi-valued signals to Boolean signals. The problem, again, is that the number of encodings is very large. In fact, the number of different binary encodings for a given multi-valued signal is the same as the number of encodings for a partition with a given column multiplicity ν (Subsection 4.3.3). For ease of notation, let $Q \uparrow = 2^{\lceil log_2 Q \rceil}$ and $Q \downarrow = 2^{\lceil log_2 Q \rceil - 1}$. Given that a multi-valued signal is Q-valued, and $P(n,r) = \frac{n!}{n-r!}$, there are $P(Q \uparrow, Q)$ different binary encodings for the multi-valued signal. But, the number of non-redundant, non-negative Boolean encodings is

$$\frac{(Q \uparrow -1)!}{(Q \uparrow -Q)!(\lceil log_2 Q \rceil)!}.$$

Table 6.4 illustrates the number of non-redundant, non-negative Boolean encodings for $Q < 10$, where $C(n,r) = \frac{n!}{(n-r)!r!}$. As a consequence, for each multi-valued signal, testing all encodings is impractical. But, the advantage of using post-decomposition encoding is that many of the encodings may be tried. This is a luxury that generally cannot be used in general decomposition algorithms because of the complexity problems associated with decomposition, i.e., partitioning, column multiplicity, etc. In fact, almost all published decomposer algorithms select only one encoding for each partition. Because encoding creates the composite functions generated during the decomposition, a bad encoding choice can result in composite functions that cannot be decomposed further.

Even though the number of possible Boolean encodings is very large, the number of single-bit encodings of a multi-valued output is greatly reduced. By single-bit encoding, it is implied that only one bit (signal) out of the $\lceil log_2 Q \rceil$ signals is used. The number of possible single-bit encodings for a Q-valued output function is given by:

$$\sum_{i=Q-Q\downarrow}^{Q\downarrow} C(Q,i).$$

Q	$P(2^{(\lceil log_2 Q \rceil)}, Q)$	$C(2^{(\lceil log_2 Q \rceil)}, Q)$	non-redundant encodings
2	2	2	1
3	24	4	3
4	24	1	3
5	6720	56	140
6	20160	28	420
7	40320	8	840
8	40320	1	840
9	4.15e9	11440	10,810,800

Table 6.4: Number of possible Boolean encodings for a Q-valued function.

The number of non-negative encodings is given by:

$$\frac{1}{2} \times \sum_{i=Q-Q\downarrow}^{Q\downarrow} C(Q, i),$$

which is always less than 2^{Q-1}, given that $\sum_{i=0}^{Q} C(Q,i) = 2^Q$. Table 6.5 illustrates the number of non-negative single-bit encodings that can be obtained by encoding a Q-valued function.

Given that during the decomposition process the largest output value of a function is generally small, it is possible to use all the non-negative encodings to generate all corresponding Boolean functions. (Testing of benchmark functions found that most output values were less than 15, but some functions exist that have more than 20 output values.) Algorithm 6.6 presents the heuristic algorithm that is used for the cases when the number of output values is less than 20. While not explored here, heuristics can be used so that not all non-negative encodings are generated. Example 6.9 uses Algorithm 6.6 to find the post-decomposition encoding of an example decomposed net-list.

Example 6.9. *Given the function $f = ((a \cdot b) + c) \oplus (d \cdot b)$, a possible decomposition of the function using multi-valued labeling is shown in Figure 6.7a. The corresponding output values for each of the composite functions is shown in Figures 6.7b,c,d.*

Q	reduced equations	non-negative functions	2^{Q-1}
3	$C(3,1)$	3	4
4	$C(4,2)/2$	3	4
5	$C(5,1)+C(5,2)$	15	16
6	$C(6,2)+C(6,3)/2$	25	32
7	$C(7,3)$	35	64
8	$C(8,4)/2$	35	128
9	$C(9,1)+C(9,2)+C(9,3)+C(9,4)$	255	256
10	$C(10,2)+C(10,3)+C(10,4)+C(10,5)/2$	501	512
11	$C(11,3)+C(11,4)+C(11,5)$	870	1024
12	$C(12,4)+C(12,5)+C(12,6)/2$	1749	2048

Table 6.5: Number of possible Boolean functions for encoding a Q-valued function.

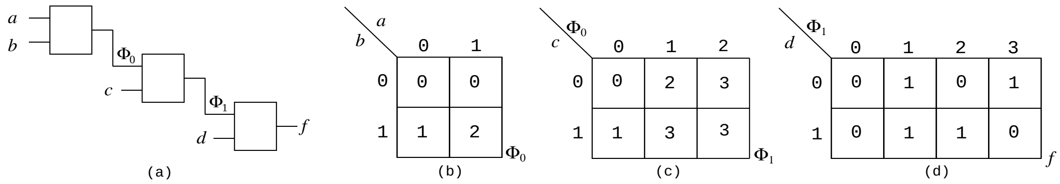


Figure 6.7: Example input to post-decomposition encoder.

Algorithm 6.6 starts by creating all possible encodings of the composite function Φ_0. Each of the possible encodings is shown in Figure 6.8a. The encoding used to create the functions ϕ_0, ϕ_1 and ϕ_2 is illustrated in Figure 6.8b. To create each function ϕ, each of the output values of Φ_0 are encoded. For instance, ϕ_0 was created by encoding the 0-output values with 1, and all other output values with 0.

Because Φ_0 is three-valued, two ($\lceil log_2 3 \rceil$) encodings must be used. Using DFC (Section 4.4) as the figure of merit, ϕ_0 has DFC of two, and ϕ_1 and ϕ_2 have a DFC of four. Either the encoding using ϕ_0, ϕ_1 or the encoding using ϕ_0, ϕ_2 can be used. Algorithm 6.6 uses both encodings, but for illustration, the encoding with ϕ_0 and ϕ_2 is used.

Algorithm 6.6 Post-decomposition encoding algorithm.

Encoder

Order the decomposed blocks from primary inputs to primary outputs.
For each decomposed block with multi-valued output:

1) Create all possible non-negative Boolean functions that can result by encoding the multi-valued output.
2) Using a figure of merit, sort the Boolean functions from least to most complex (e.g., smallest DFC to largest DFC)
3) Replace the current functional multi-valued output block with the least complex set of Boolean functions that encode the multi-valued output (e.g., set of functions with smallest total DFC).
   It is possible that more than one set of Boolean functions has the smallest complexity. In this case, the first 10 sets are tried.
4) Encode all functional block inputs that are the output of the current functional block.

The algorithm continues by carrying the new encoding to the functional blocks that have Φ_0 as an input, in this example this is the function Φ_1. The new function Φ_1 and all encoded functions of Φ_1 are shown in Figure 6.8c, with the encodings used illustrated in Figure 6.8d. ϕ_4 and ϕ_3 have the smallest combined DFC of the possible encodings, so they are used to encode Φ_1. The encoding is carried to the function f, where $f = ((a \cdot b) + c) \oplus (d \cdot b)$, is the original function. (Note, the expression $f = \overline{\phi_4} \oplus \overline{\phi_3} d$ is found by using single-variable free set decomposition, with ϕ_4 as the free set variable.)

6.4 Small Bound Set Decomposition

This section presents the development of a new decomposition algorithm called Small Bound Set Decomposition (SBSD) which is based on the small bound set decomposition types in Chapter 5. The section is broken into two parts, algorithm development and experimental results. Algorithm development presents the algorithms used to construct the SBSD decomposer. Experimental results are presented that show the ability of SBSD to find the minimal decomposed structures of benchmark functions.

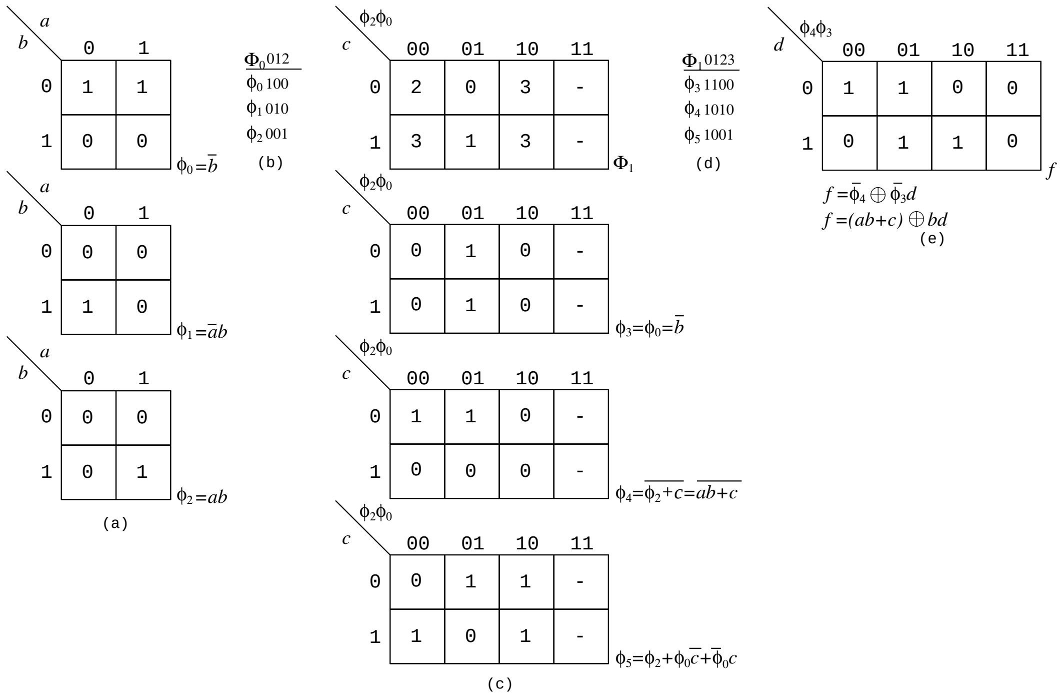


Figure 6.8: Example of encoding the decomposed blocks.

6.4.1 Small Bound Set Algorithm Development

While many different decomposers have been published, in general, each of these decomposers uses the same basic algorithms. Algorithm 6.7 illustrates the basic structure of a decomposer.

The implementation of SBSD has the same structure as the Algorithm 6.7 pseudo code. The heuristics that SBSD uses during the decomposition process are for finding partitions with small bound sets and using multi-valued labeling of the resulting composite functions. The small bound set partitions that are used by SBSD are the small bound set decomposition types: single-variable free set decomposition, and two-variable bound set decomposition.

Figure 6.9 shows the general block diagram of the SBSD decomposer with technology-dependent devices, e.g. layout, and machine learning. Once the decomposer has decomposed the function, it outputs a net-list of decomposed

Algorithm 6.7 Basic decomposition pseudo code.

```
general decomposer (function f)
   if f has less than 3 input variables
      function cannot be decomposed further
      evaluate current decomposition
      return
   evaluate partitions of f
   determine best partition through heuristics
      heuristics based on column multiplicity and number of bound variables
   encode/label partition
   create composite function(s) Φ
   create composite function F
   for each Φ_i in Φ
      call general decomposer(Φ_i)
   call general decomposer(F).
```

blocks that are used by the post-decomposition encoder. The encoder encodes the decomposed blocks and outputs the encoded-blocks for technology-dependent analysis. Technology-dependent analysis checks the blocks against a figure of merit to analyze the decomposition for a given technology. For machine learning problems, the network size of blocks is checked using DFC. For layout problems the net-list of blocks is given to layout software to determine the layout structure (quality is determined by the structure and size of the layout). This may require creating a new network of blocks, where the decomposition can be placed in the technology-device (e.g., some FPGAs require four-input, single-output blocks, so the decomposition must be in the form of four-input, single-output blocks).

While the block diagram shows a technology-independent block of decomposition and a technology-mapping block, these blocks are not independent. To create near-optimal results, these blocks are used in conjunction with each other. The most critical is the post-decomposition encoding of the blocks, where the blocks are technology-dependent and mapped to a given technology. The decomposition process may be refined for a given technology, and decompositions that fit the technology will be easier to find. This methodology is a series of checks-and-balances between the technology-independent and the technology-dependent

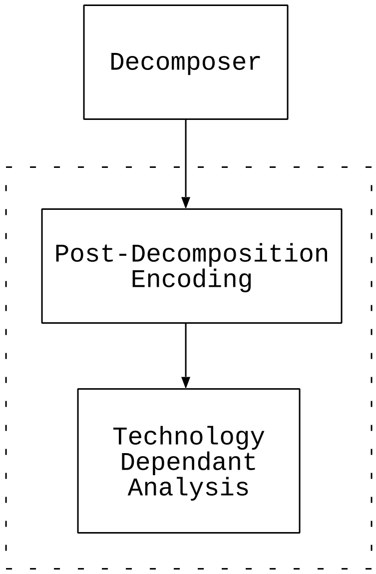


Figure 6.9: New decomposition block diagram.

approaches.

Therefore, not only does SBSD use the small bound set decomposition types presented above; it is also based on algorithms to help reduce complexity of the decomposition and find the best technology-dependent decomposition. These algorithms (pseudo code and implementations in SBSD are shown at the end of this subsection) are

1. minimal support,
2. three-variable symmetry,
3. post-decomposition encoding and technology-dependent analysis.

Minimal support removal is used to reduce the complexity of the function by removing inessential variables that have no effect on the output of the function. Minimal support is actually a pre-processor algorithm that is called before running the decomposer. In Chapter 5, two-variable bound set decompositions are discussed using the concept of symmetry. It is possible to extend symmetric pairs of variables using the following relations:

1. if f has symmetry $\{x_i, x_j\}$ and $\{x_j, x_k\}$, then f is symmetric in $\{x_i, x_j, x_k\}$,

2. if f has symmetry $\{x_i, \overline{x_j}\}$ and $\{x_j, x_k\}$, then f is symmetric in $\{x_i, \overline{x_j}, \overline{x_k}\}$,

3. if f has symmetry $\{x_i, \overline{x_j}\}$ and $\{x_j, \overline{x_k}\}$, then f is symmetric in $\{x_i, \overline{x_j}, x_k\}$.

These relations allow the decomposition of functional blocks with two or three inputs. The least complex functional blocks have two or three inputs, so it is reasonable to decompose the function into two- or three-input blocks. To reduce the complexity of a highly symmetric function, the symmetric variables are grouped in blocks of three. Another reason to look for three-variable symmetries is that the column multiplicity of a three-variable symmetrical bound set may be less than the column multiplicity of each of the two-variable symmetry bound sets. This is illustrated in Example 6.10.

cd \ ab	00	01	10	11
00	0	0	1	0
01	1	1	0	1
10	1	1	1	1
11	1	1	1	1
	0	1	2	0

(a)

cb \ ad	00	01	10	11
00	0	0	1	0
01	1	1	0	1
10	1	1	1	1
11	1	1	1	1
	0	1	2	0

(b)

ca \ bd	00	01	10	11
00	0	0	1	0
01	1	1	0	1
10	1	1	1	1
11	1	1	1	1
	0	1	2	0

(c)

c \ abd	000	001	010	011	100	101	110	111
0	0	1	0	1	1	0	0	1
1	1	1	1	1	1	1	1	1
	0	1	0	1	1	0	0	1

(d)

Figure 6.10: Three variable symmetry example.

Example 6.10. *Figure 6.10a,b,c shows that the variable pairs:* $\{a, b\}$,$\{a, c\}$ *and* $\{b, c\}$ *are all symmetric pairs. Thus, the variables* $\{a, b, c\}$ *must also be symmetric. The partition* $\{c\}|\{a, b, d\}$ *is shown in Figure 6.10d. The partition* $\{c\}|\{a, b, d\}$ *has column multiplicity* $\nu = 2$*, where the two-variable bound sets each have* $\nu = 3$*.*

Algorithm 6.8 is the pseudo code for the three-variable symmetry test.

Algorithm 6.8 Three variable symmetry test pseudo code.

```
for each input variable pair (x_i, x_j) in function f
   let set columns_1 = PARTITION(x_i, x_j)
   if column multiplicity ν_1 < number of columns_1
      variable pair is symmetric
      for each input variable x_k in f, x_k ≠ x_i, x_k ≠ x_j
         let set columns_2 = PARTITION(x_i, x_k)
         let set columns_3 = PARTITION(x_j, x_k)
         if column multiplicity ν_2 < number of columns_2
            and column multiplicity ν_3 < number of columns_3
            symmetry may exist, try partition with bound set B = {x_i, x_j, x_k}
            let set columns_4 = PARTITION(x_i, x_j, x_k)
            if symmetry exists
               create decomposition.
```

To complete the decomposition process, post-decomposition encoding and technology-dependent analysis is needed. The post-decomposition encoding algorithm is presented in the previous section. The pseudo code for SBSD is shown in Algorithm 6.9. The algorithm is recursive because each of the partitioning algorithms call SBSD with the composite functions found during the decomposition process.

After SBSD is finished, the best decomposition or the set of best decompositions are returned to the user. (The user may see significant differences in the returned set of decompositions that are not noticeable using the given figure of merit. The user is the final judge of the decomposition process.)

Algorithmic complexity of SBSD based on the number of partitions evaluated is lowered bounded by evaluating $2n - 1$ partitions (n is the number of

Algorithm 6.9 SBSD pseudo code.

```
SBSD(function f) if number of input variables to function f < 3
      post-encoding and technology-dependent analysis()
      return
   single-variable free set decomposition()
   three-variable symmetry decomposition()
   if (three-variable symmetry decomposition() does not exist)
      two-variable bound set decomposition()
      if (two-variable bound set decomposition() does not exist)
         three-variable bound set decomposition()
         if (three-variable bound set decomposition() does not exist)
            No decomposition possible
            return
```

input variables). This occurs if every decomposition is a single-variable free set decomposition. That is, the first level decomposition evaluates all n variables in the free set, and at each remaining $n-1$ decomposition levels only one partition is evaluated. In the worst case, the number of partitions evaluate through the entire decomposition of the function is $O(n^3)$. This is given by evaluating all three-variable bound set partitions on the first level, given as $n(n-1)(n-2)/2$. The number of three-variable bound set partitions evaluated on the second level is $(n-3)(n-4)/2$, on the third level is $(n-5)(n-6)/2$, etc. Because the first level number of partitions has complexity $O(n^3)$ and the addition of all other levels is not on the order of n^3, the resulting worst case complexity of SBSD is $O(n^3)$.

6.4.2 Small Bound Set Decomposition Results

Table 6.6 shows SBSD with post-decomposition encoding on several eight-input, single-output functions from the Wright Patterson benchmark set. SBSD is compared to FLASH [134], using the figure of merit DFC (Decomposed Function Cardinality) discussed in Section 4.4. This set of benchmark functions represents several different functional families. Included in these families are the

KDD (Knowledge Discovery in Databases) functions, Boolean logic functions from circuit design, machine learning, and random functions.

FLASH is a program written by the Wright Patterson Pattern Theory group. The reason for comparing to FLASH is that FLASH was run extensively on this set of benchmarks, over many years. The decomposition strategy of FLASH was to try all possible decompositions for a given function. Any time a smaller DFC was found for a function, the results in the table were updated. FLASH's results come from running under many different conditions, over several years, and saving the best solutions. Consequently, it was thought that the results of FLASH were minimal, and thus, would provide an excellent comparison to future decomposers. Because of this, it was quite a surprise when SBSD actually outperformed FLASH on several benchmark functions. To confirm that the results of SBSD are not a mistake, all SBSD decompositions have been checked for functional equivalence with the original function using the "verify" option in SIS [145]. In the cases that FLASH has a smaller DFC than SBSD, the difference in DFC is generally quite minimal. The benchmark *rnd_m10* is the only instance where FLASH significantly performs better than SBSD. Because of this function and the performance of SBSD on all the randomly generated functions, a possible conclusion is that SBSD will not perform well on future randomly generated functions. This, of course, is not treated as a weakness in this dissertation because the randomly generated functions are not used in machine learning or logic synthesis problems.

In a couple of the instances when SBSD has smaller DFC than FLASH, the difference is quite large. The advantage of SBSD over FLASH is that SBSD is the post-decomposition encoder and the reduced set of partitions that were evaluated. While FLASH searched all bound sets to obtain the results in the table, FLASH was generally restricted to only one encoding choice that was pseudo-random. Also, FLASH would run for several days on a given benchmark function, while the runtimes of SBSD varied from just a couple of seconds to less than four minutes.

6.4.3 Conclusions of Minimum Bound Set Decomposition

Two new algorithms, EVAL and PARTITION, were created that do not reorder the decision diagram and, thus, do not cause an exponential increase in size of the MDD by reordering. Comparisons were run against the well-known algorithms presented by Lai, Pedram and Vrudhula. In most cases, the EVAL algorithm is faster than the LPV algorithm, especially on small bound sets. In fact, the EVAL algorithm was able to complete in the specified amount of time on the large multi-valued functions while the LPV algorithm could not. EVAL is also much faster than running the PARTITION or *cut_level* algorithms. A restriction of the EVAL and PARTITION algorithms is that they are only practical for partitions with small bound sets. This restriction is actually used in many decomposition strategies, including FPGA decomposition and decomposition for machine learning [50, 155]. Machine learning functions are typically large multi-valued, incompletely specified functions where heuristics are used to reduce the search space by only evaluating small bound sets. The EVAL and PARTITION algorithms are the MDD decomposition algorithms used throughout the remainder of this dissertation.

The minimal support problem was also examined and the previous work by Jozwiak was presented. A new minimal support algorithm named Top_Down was presented and compared to the work by Jozwiak. The results show that for most benchmark functions the Top_Down algorithm is much faster at finding a solution than the algorithm created by Jozwiak.

The last two sections of this chapter presented the SBSD algorithm and the post-decomposition encoding algorithm that are used in the remainder of this dissertation. The algorithmic complexities of both algorithms are much smaller than the complexities of most decomposers in the literature. While SBSD and the post-decomposition encoding algorithm do not always find the minimal DFC of a function, the offset of results is generally quite small. It is shown in later chapters that SBSD is effective and efficient for both layout and machine learning problems.

The remaining chapters of this dissertation explore the application of decomposition to VLSI layout and machine learning. To aid in the decomposition of these applications the SBSD decomposition method is used.

Function	in	FLASH	SBSD
add0	8	28	28
add2	6	20	20
and_or_chain8	8	28	28
ch22f0	6	20	20
ch30f0	6	**32**	40
ch47f0	6	60	**56**
ch52f4	8	180	**156**
ch70f3	8	**40**	44
ch74f1	8	**72**	84
ch83f2	8	**116**	120
ch8f0	6	**32**	40
4_ones	8	76	76
greater_than	8	28	28
interval1	8	128	**88**
interval2	8	92	**76**
kdd2	5	16	16
kdd3	5	12	12
kdd5	8	**32**	48
kdd6	8	12	12
kdd7	8	28	28
kdd9	8	20	20
kdd10	6	20	20
majority_gate	8	**64**	76
monkish1	4	12	12
monkish2	8	60	60
monkish3	5	20	20
mux8	6	**24**	32
or_and_chain8	8	28	28
pal	8	28	28
parity	8	28	28
rnd_m1	8	28	28
rnd_m10	8	**80**	108
rnd_m25	8	**172**	180
rnd_m5	8	**64**	72
rnd_m50	8	**224**	256
substr1	8	72	72
substr2	8	60	60
subtraction1	8	**64**	68

Table 6.6: SBSD comparison to FLASH on Wright Lab benchmark functions.

Chapter 7

Decomposition For Layout

There are many problems in creating and developing an Integrated Circuit (IC) chip, including logic synthesis. Logic synthesis is used in the layout of an IC chip to minimize the overall size of the chip by reducing the number of logic gates being created. Two layout strategies are presented in this chapter. The first is Metal Oxide Silicon (MOS) or Complementary MOS (CMOS); CMOS is the technology mostly used in Very Large Scale Integration (VLSI) [111]. The second layout strategy is for lattice diagrams [5, 39]. Both of these layout types need good logic synthesis preprocessors to produce good layouts.

Introduced here is the integration of three logic synthesis methods for VLSI layout: functional decomposition, gate compaction and negative gates. The functional decomposition algorithm presented in the previous chapter is used as the primary logic minimization. (Note: the author is unaware of any publications that use functional decomposition for layout.) Gate compaction is used to functionally combine several two-input gates into a single *complex gate*. The concept of a complex gate is defined later in this chapter as a logic gate that can implement many different logic functions. It is possible that the decomposer may output non-decomposable blocks that have more than two inputs. In this case, negative gate logic synthesis is used to create complex gates when a logic block is not decomposable.

Figure 7.1 illustrates the integration of functional decomposition and the other logic synthesis algorithms used for layout. After presenting the layout types, this chapter presents the proposed logic synthesis methods that are used as preprocessors for layout. The proposed logic synthesis method incorporates the functional decomposition algorithms presented in the preceding chapters.

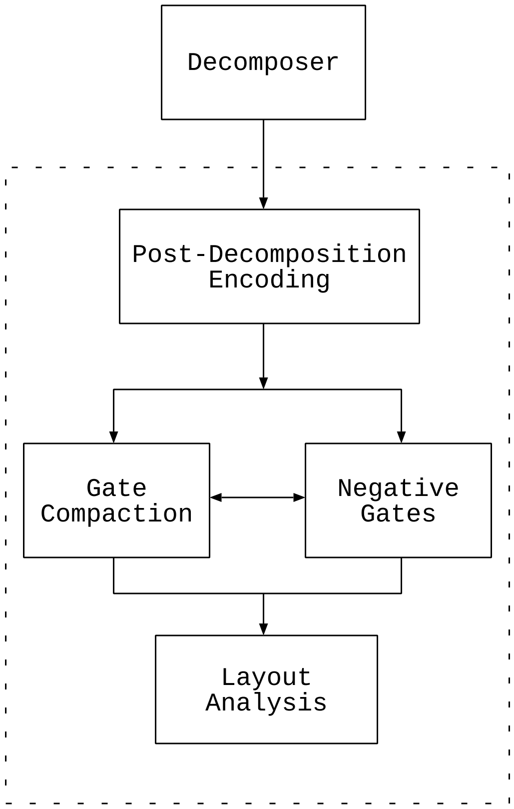


Figure 7.1: Layout decomposition block diagram.

7.1 MOS and CMOS Layout

CMOS implementation of functions has become increasingly important to VLSI strategies over the past two decades. The basic principles of the MOS Field-Effect Transistor (MOSFET) were proposed by Lilienfeld in 1925 [81], and a similar structure closely resembling a modern MOS transistor was proposed by Heil in 1935 [60]. Since the introduction of the first commercial IC in 1961, the size and timing of digital ICs has increased very rapidly [153]. Commercial IC chips in the 1960s had about 100 transistors; after many improvements in fabrication technology, a VLSI circuit may now have over 10 million transistors [89]. The design cost and time to implement a design increases as the number of the transistors grow. To aid in the designing of ICs, computer-aided design tools were created to manage design complexity and to improve design productivity.

While [89, 111, 153] and many other publications go into great details about the construction of VLSI components, this section only discusses the basic components of MOS and CMOS circuits. There are two types of MOSFETs, n-type and p-type. The difference in the two types of MOSFETs is the way that they are created by using doping agents in silicon. A n-type MOSFET (or n-MOS) is created by placing two n-type diffusion regions separated by a p-region on a substrate [111]. A p-MOS is created by placing two p-type diffusion regions separated by a n-region on a substrate. The difference in these two MOSFETs, in terms of logic, is that n-MOS is positive logic, in that logic values (0 and 1) are represented by low and high voltages, respectively; p-MOS is negative logic, where logic values (0 and 1) are represented by high and low voltages, respectively. If the input to a n-MOS transistor is positive, then the gate is turned on, respectively if the input to a p-MOS transistor is negative, then the gate is turned on. Figure 7.2 shows the symbols that represent n-MOS and p-MOS transistors.

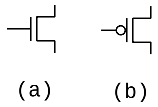


Figure 7.2: MOSFET types (a) n-MOS (b) p-MOS.

MOS and CMOS **gates** (p-type or n-type MOSFETs) can be placed together to create a **cell** (an ordered layout of transistors) in such a way as to create basic (or complex) Boolean functions. Given that n-MOS can represent positive logic, then it is possible to combine n-MOS transistors in series or in parallel to

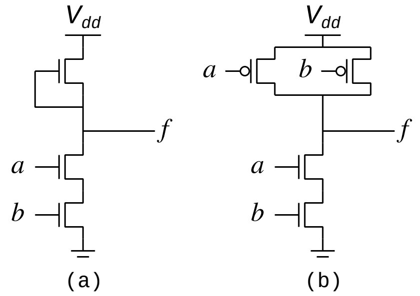


Figure 7.3: Implementing $f = \overline{a \cdot b}$ (a) MOS (a) CMOS.

create more complex cells to represent logic gates. n-MOS OR operations are performed by parallel connected transistors and AND operations are performed by series-connected transistors. Because p-MOS transistors are the dual of n-MOS, p-MOS OR operations are performed by series connected transistors, and AND operations are performed by parallel connected transistors.

In creating logic solely based on a p-MOS network or a n-MOS network, n-MOS is most generally chosen because of some inherent performance advantages over p-MOS. An example of a MOS cell created only with n-MOS gates is shown in Figure 7.3a, where the function $f = \overline{a \cdot b}$ is implemented. If inputs a and b are both logic high values, the bottom two gates are turned on, which forces the output f to be a logic low, and turns off the top gate. If either of the variables a or b is a logic low, then the voltage at the output is not pulled to GROUND, which allows the top gate to be turned on, producing a logic high on the output.

It is possible to create a cell based on both p-MOS and n-MOS; this is called CMOS. Because p-MOS is the dual of n-MOS, p-MOS transistors can be used to pull logic values of a function to a high, whereas, n-MOS transistors can be used to pull logic values of a function to a low. An example of a CMOS cell is shown in Figure 7.3b, where the function $f = \overline{a \cdot b}$ is implemented. If a and b are both logic high values, both n-MOS gates are turned on, and both p-MOS gates are turned off, which forces the output of f to be 0. If either of the variables a or b is a logic low, then one of the p-MOS gates is turned on, which pulls the output value high. One of the n-MOS gates in this situation is turned off, so there is no path from the output to GROUND and the output produces a logic high.

In general, both CMOS and MOS cells represent negated functions, e.g., NAND, NOR, etc. The reason that MOS represents negated functions is the physical characteristics of n-MOS and p-MOS transistors. n-MOS (positive logic) is always used to pull functional outputs low, while p-MOS (negated logic) is always used to pull functional outputs high. This is illustrated in Figure 7.3b where the n-MOS transistors are placed between GROUND and the functional output (pull-down) and the p-MOS transistors are placed between V_{dd} and the functional output (pull-up). Logic synthesis methods based on negative gates can be applied to either MOS or CMOS technology. The disadvantages of CMOS compared to n-MOS have been low speed (p-MOS is slower than n-MOS) and large chip area (almost twice the size). But, MOS requires much more power than CMOS; as integration sizes increase, dissipation of heat is a big problem (by-product of power). Also, many technology advances have been created to reduce the timing and size of CMOS cells, and CMOS has been chosen for most commercial applications.

To construct CMOS cells for basic logic functions, such as NAND and NOR gates, designs may be created by specialists and stored in libraries. Each of these cells is called a *standard cell* and can be used to implement any particular design. This has great applications because the standard cells are regular in height, and only vary in width, depending on the complexity. That is, to create an IC, it is necessary to route V_{dd} and GROUND to every cell in the IC. If all cells have the same height, then V_{dd} may run parallel to GROUND, separated by the standard cells, across the entire IC. Another advantage of standard cells is that net-lists found during logic synthesis are directly mapped to standard cells in the layout.

The standard cell approach is accepted as a good solution for the layout synthesis problem. The algorithms used in this approach are well known, and the cells are pre-characterized, allowing a fast performance prediction and faster production times. However, this approach is dependent on design rules (each new technology requires a new set of standard cells to be created). Technology mapping is limited by the cells found in the library and sizing possibilities are restricted to the templates of each cell. As the standard cell libraries are limited not only by the choice of logical cells, but also in the choice of the cells performance, the designers cannot realize all the improvements that can be obtained at the higher synthesis levels. One such improvement is an **AndOrInverter** (AOI), which is a standard cell that represents small sum-of-product expressions and inversions. The natural evolution of the standard cell method is for a layout design methodology that allows a flexible technology mapping with runtime cell generation. This includes not only standard logic gates and AOI but many different random logic functions. These gates are called **static CMOS Complex Gates** (SCCG). The advantages of using SCCG are

1. reduction in area saving (smaller number of transistors and smaller number of SCCGs[1]),
2. small delay (smaller number of logic levels),
3. small power consumption (smaller number of switching nodes: each switching node is a SCCG).

Library Free Technology Mapping (LFTM) is a technique involving the extensive use of complex gates, i.e., a mapping that is not based on standard logic functions from a cell library. Generic runtime cell generation allows layout synthesis from a structural description, using any kind of CMOS gate.

Three methodologies have been proposed [1, 14, 36] which try to exploit the great number of different logic gates that can be implemented in CMOS technology for a given limitation in the number of serial transistors. This concept is different from the AOI definition presented in [71], where an AOI is a two-level logic gate. The number of different SCCGs for a given limitation in the number of serial n-MOS and p-MOS transistors was calculated in [36] and is shown in Table 7.1. The set of the SCCGs with a maximum of three-serial n-MOS

[1] each SCCG requires routing channels which increase size.

		Number of Serial PMOS Transistors				
		1	2	3	4	5
Number of Serial NMOS Transistors	1	1	2	3	4	5
	2	2	7	18	42	90
	3	3	18	87	396	1677
	4	4	42	396	3503	28435
	5	5	90	1677	28435	125803

Table 7.1: Number of SCCGs with limited serial transistors [36].

transistors and four or less p-MOS transistors is denoted by SCCG(3,4). From Table 7.1, the cardinality of SCCG(3,4) is $|\text{SCCG}(3,4)| = 396$. Due to CMOS implementation constraints, a maximum of four-serial p-MOS transistors and five-serial n-MOS transistors are allowed, resulting in 28,435 different SCCGs. This is a great advantage over standard cell libraries where it is almost impossible to create 28,435 different standard cells. In fact, as CMOS technology advances, the number of SCCGs should also increase, which makes LFTM more practical. The concept of LFTM was a topic during a panel discussion during the 1999 Design Automation Conference. The conclusion of the panel was the need for more LFTM generation tools, especially because industry is increasingly using LFTM.

Table 7.2 compares the LFTM program TROPIC to a standard cell library layout tool [102, 103, 104]. Two benchmark functions *adder* and *alu* were used from the ISCAS85 benchmarks, with transistor widths $w = 2\mu$m and load capacitance 706fF to find timing results (both values are components of transistors). TROPIC was used to create AOI cells. The results show that by creating more complex structures than those found in the standard cell libraries, the number of transistors and gates is greatly decreased. A reduced number of transistors results in minimizing the other parameters in the table. These two functions show the effectiveness of using AOI (SCCG) and LFTM in layout synthesis.

It is important to understand that the creation of SCCGs is a logic synthesis problem, not a layout synthesis problem. TROPIC is a layout synthesis tool that is used to place the transistors on an IC. Another tool must be used to generate

	simple gates	AOI gates
Circuit: ADDER		
number of transistors	40	28
number of gates	10	4
rise time (ns)	4.0	3.3
fall time (ns)	3.3	2.0
power (mW)	1.4	1.35
$\sum$ parasitic cap. (pF)	1.06	0.5
Area (mm 2)	0.0185	0.0115
Circuit: ALU		
number of transistors	432	262
number of gates	101	39
rise time (ns)	12.4	10.0
fall time (ns)	6.2	5.9
power (mW)	9.3	6.7
$\sum$ parasitic cap. (pF)	18.4	8.6
Area (mm 2)	0.3025	0.1834

Table 7.2: Example differences between standard cell and complex gate implementations (taken from [104]).

and minimize the number of SCCGs. The next section presents previous logic synthesis methods and the proposed logic synthesis method for creating SCCGs.

7.2 Previous Synthesis Methods for Layout

The LFTM approach is the extensive use of SCCGs instead of using a precharacterized library. The logic synthesis method proposed in [36] is based on pattern matching and the methods proposed in [1, 14] use greedy structural approaches based on local limitation of serial transistors while constructing a SCCG structure. Example 7.1 illustrates the need for a different logic synthesis method for CMOS layout.

Example 7.1. *Figure 7.4 shows the Karnaugh maps of the function f and $\overline{f}$. If sum-of-product logic synthesis is used, the essential prime implicants are circled in the Karnaugh maps. To create the respective SCCGs for each function*

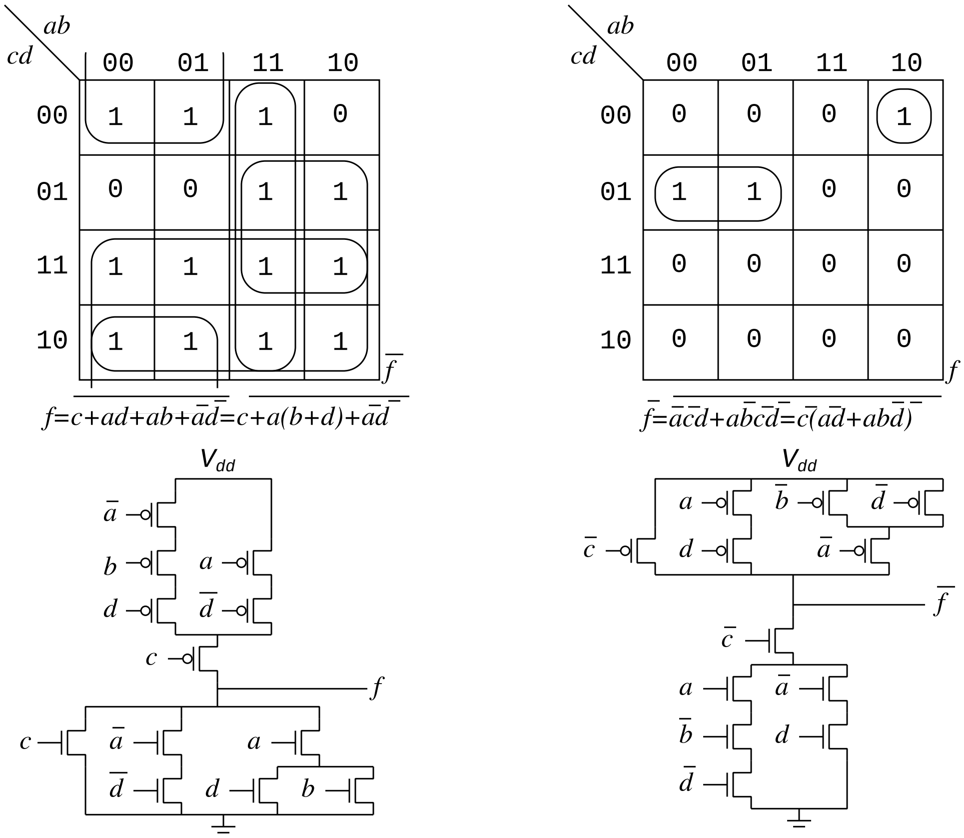
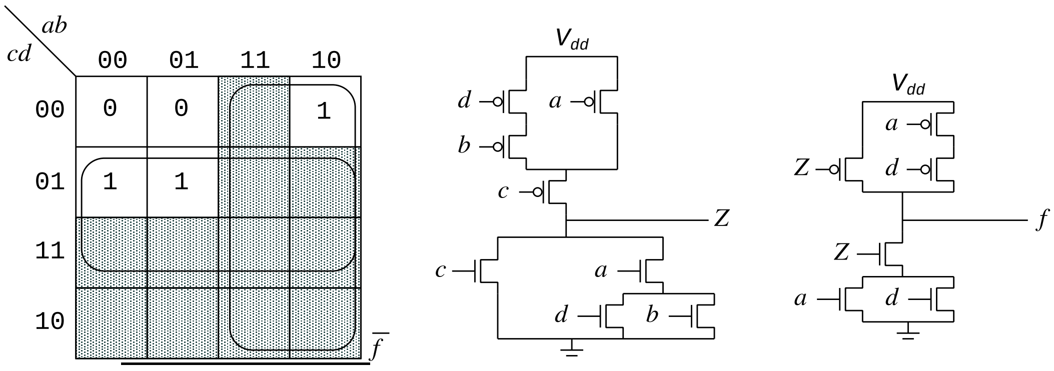


Figure 7.4: SCCGs for example function.

(shown below the Karnaugh maps), the sum-of-products representation for both functions are used. The functional output of f is a logic high when the p-MOS transistors pull the output to V_{dd}. The expression $f = \overline{\overline{f}}$ is used because CMOS is a complementary logic. To find the n-MOS representation of f, the expression $c + a(d + b) + \overline{a}\ \overline{d}$ is used. Because p-MOS is the dual of n-MOS each OR and AND is replaced with AND and OR, respectively giving the expression $c(a + db)(\overline{a} + \ \overline{d}) = c(a\overline{d} + \overline{a}db)$. The SCCG that represents f has twelve transistors, but needs inputs a and d in negated form. Each negation requires one SCCG with two transistors, which means that f can be represented with three SCCGs and sixteen transistors. f can also be found using $\overline{f}$ and the expression $\overline{c(\overline{a}d + a\overline{b}d)}$, and then negating the output, which takes six SCCGs and 22



$$f = \overline{(a+d)\overline{(c+ad+ab)}}$$
$$f = \overline{(a+d)Z}$$
$$Z = \overline{(c+ad+ab)}$$

Figure 7.5: Using dissected pairs on example function.

transistors to represent the function.

Figure 7.5 uses the concept of dissected pairs (presented later in this chapter) for the logic synthesis. The dissected pair-based algorithm forces all variables to be in the positive form. While this theory is very intensive, in essence, the dissected pair method finds the largest unate cubes that cover a minterm, and then subtracts a unate cube that covers an associated maxterm. All minterms of f can be covered with the unate term $(a + d)$, while subtracting out the term $Z = (c + ad + ab)$, shown in the figure as the cells in gray. The result of the dissected pair algorithm is two SCCGs and 14 transistors, a smaller result than using SOP minimization.

LFTM relies on the extensive use of an implicit generator of SCCGs instead of using a pre-characterized library of explicit cells. It is necessary to discuss the enumeration of logically different SCCGs and the consequences of their use. The following subsections present existing technology mapping approaches (pattern and Boolean matching), the greedy structural approaches for LFTM and finally some logic synthesis methods for target structures (negative logic and TANT network synthesis).

7.2.1 Pattern Matching Approaches

The difficulty of applying pattern matching techniques to LFTM is that the number of patterns is greater than the number of SCCGs. DAGON [71] is based on pattern matching on a Directed Acyclic Graph (DAG) reduced to a forest of trees. In order to perform the mapping on a 165 gate library, DAGON uses seven hundred cell patterns as well as a few thousand unique local optimization patterns [71]. The LFTM in MIS [36] is based on recognizing patterns in a tree, and the depth of the search is limited in order to reduce the number of patterns. The greater number of SCCGs treated in [36] is 396 = |SCCG(3,4)|, but 839 different patterns are needed to represent these 396 SCCGs. The number of patterns is not linear with the library size, and 1171 different patterns are needed to represent a subset of SCCG(4,4) containing only 123 SCCGs out of 3503 [36].

7.2.2 Boolean Matching Approaches

Boolean matching [88] is done by comparing the logic functions of subsets of the subject description with the logic functions of library elements. The technique from [88] uses the canonicity of ROBDDs [21]. Even if there is only one logic function for each element, the number of pin assignments to match a cell is exponential with the number of inputs. In [88], the libraries are limited to gates with eleven or less inputs and in practice, only gates with a maximum of six inputs are used. Because SCCG(3,3) contains SCCGs with nine inputs and SCCG(4,4) contains SCCGs with up to 16 inputs, Boolean matching is not appropriate to perform LFTM.

7.2.3 Greedy Structural Approaches

The method proposed in [1] is based on a tree representation of the circuit (as in [36, 71]). However, there is no need to recognize patterns because the use of inversions is limited to the root and leaf nodes. This allows an easy calculation of the number of serial transistors (the search depth is not limited; the constraint is in the structure of the tree). The approach is based on decomposing trees into sub-trees by locally considering the number of serial transistors in the sub-trees. The algorithm starts from the deepest leaf node of the circuit's tree and selects the largest sub-tree that satisfies the serial transistor constraint; this sub-tree is then substituted with a leaf node. The process is repeated until the remaining tree satisfies the serial transistor constraints. The method proposed in [14] is based on a graph representation and has the same limitations as [1] in exploiting internal inversions. The advantage of this method is that the output is at the transistor level, while in [1, 36] the output of the mapping is at the logic level, and an extra step is needed to give the transistor topology to the leaf cell generator. The main problem of the greedy structural approaches is that only local serial transistor limitation is taken into account while constructing a SCCG structure. In other words, they solve the LFTM problem as the task of finding a set of SCCGs that represent a circuit, obeying the following two conditions: each SCCG has less than a given number of serial transistors, and the size of each SCCG is maximized so that the SCCG is as large as possible (defined by the maximum number of serial transistors). By creating SCCGs as large as possible, the total number of SCCGs in the layout is minimized in the hopes of reducing timing and power consumption.

7.2.4 Negative Logic Network Synthesis

Another approach to perform LFTM is the synthesis of a negative gate network directly from a functional specification (ON, DC and OFF sets, truth table, etc). The methodology presented in [110] minimizes the number of (negative) cells as a primary objective, and the number of interconnections as a secondary objective is presented. However, this method may produce complex cells that will not work properly because of reasonable gate-delay time as a result of an excessive number of serial transistors. In this case, it is necessary to decompose these cells into a set of reasonably simple cells, which does not produce the minimal number of cells.

7.2.5 TANT Network Synthesis

A topic closely related to synthesis of negative gates is the synthesis of three-level NAND networks, also known as TANT networks, as introduced in [73]. It was shown in [26] that the set of negated positive kernels of prime implicants from the cover of $\overline{f}$ is sufficient as a set of tail factors in the TANT expression for the function f. The above work assures a minimal unate function as an estimation of minimal third-level, and does not take interest in third-level minimization. In [73] the condition for the set of tail loops was given to be a P_f set of the TANT network. P_f is denoted as a set of kernels that are sufficient to create the second level of a complex gate network for function f. The restriction of this method is gate model and at most three levels of gates.

7.3 Logic Synthesis Methods for Static CMOS Complex Gates

The approach presented in this chapter is to use the decomposition method presented in Chapter 6 and a technology-dependent multi-level minimizer. The technology-dependent minimizer takes a net-list of decomposed blocks and synthesizes each block separately, so that they fit the parameters of a SCCG. This usually requires many complex gates for a block. The synthesis is done by using the concept of dissected pairs [46] (presented in Section 7.5). The most important point of this approach is that the techniques are developed for a layout generator (mapping each SCCG onto silicon). Therefore, it can be used in high-performance applications that are usually optimized "by hand" These techniques emphasize finding solutions of a very "high quality", with the assumptions that increases in the design time, and/or limits to the sizes of the synthesized networks are worthy if they achieve nearly exact minimum solutions. The idea is to first decompose the function into small blocks and then to use synthesis (a synthesis technique named COMPLEX) on each block or a small combination of blocks.

The interest here is in general methods for designing two-level and multi-level, multi-output complex gate circuits for both completely and incompletely specified Boolean functions. The remainder of this chapter presents basic concepts and notions, as well as one complete synthesis method for single-output functions realized by using two levels of SCCG (multi-output synthesis is not explored because the decomposition results in single-output blocks). The given algorithms minimize the number of gates, and find the quasi-minimum numbers of transistors in both the first- and second-level gates of each block. Methods for design with complex gates have been presented in [27, 65, 66, 83, 84, 85, 110, 112, 113, 115, 116]. Most of these methods do not attempt to minimize a number of literals, transistors or other parameters.

The LFTM problem is formulated as the task of finding a set of SCCGs that represents a circuit and obeys the following two conditions:

1. each SCCG has less than a given number of serial transistors,

2. the overall number of transistors is as small as possible (i.e., the number of transistors is used as the main component of the cost function).

It is assumed that a multi-level decomposition program is used to create a net-list of logic gates and logical blocks (SBSD from Chapter 6). To compare the realizations found during the logic synthesis, a weighted-cost function of several parameters is used. This means that set-covering attempts to find a covering that minimizes a user-defined weighted-cost function which is a sum of partial-cost functions multiplied by user-assigned parameters. This way the following advantages are achieved:

1. a complex cost function is minimized, which reflects the complexity of gates applied,

2. the user is able to tune the solutions by selecting the values of numerical-weight coefficients,

3. certain parameters can also be used as "restricting conditions", which directs the design and limits the search space,

4. the process may be useful to find optimal solutions automatically by using feedback from the TROPIC layout generator back to the preprocessing logic minimization system SBSD+COMPLEX.

In library-based design, it is common to minimize the depth of a function's realization (the number of levels of logic is minimized for speed); the number and the size of the complex gates are minimized for area or power, and the number of transistors is minimized for testability.

In order to evaluate a complex gate net-list, the first order evaluation is done using the parameters shown below (assuming they are reasonable high-level estimators). As a result of a discussion of circuit-design specialists who created TROPIC, the parameters to be calculated for a multi-input, multi-output module were selected to be the following:

P1 *The total number of transistors.* It is assumed that the overall number of transistors in the network is the first order estimator for both area (assuming that the transistor area is constant by intervals) and power (the number of transistors is a relative good estimator for active area capacitances [101]) of the final circuit.

P2 *The total number SCCGs.* This parameter is also related to both area and power, because it is an estimator of the number of switching nodes.

P3 *Maximum number of serial transistors in all gates.* Each SCCG has its number of n-MOS and p-MOS serial transistors limited to a given number. Too many serial transistors may cause noise margin and speed problems due to increasing the threshold voltage due to body effect [101]. For current technologies, four-serial p-MOS and five-serial n-MOS are within reasonable margins [101]. As technologies change, these constraints can be changed and the logic synthesis must be based on variable constraint values. (This parameter is used as a constraint rather than component of the weighting function.)

P4 *Maximum fanout of all gates.* The user must have a means to control the maximum fanout of each SCCG. (This parameter is used as a constraint rather than a component of the weighting function.)

P5 *Number of serial SCCGs in the module.* This parameter is related to the depth of the net-list and its speed. The logic depth inside each SCCG (and throughout the net-list) is not significant.

P6 *Number of transistors in the longest path in the module.* In addition to the SCCG depth of the net-list, the speed of the SCCGs is also important. It is also assumed that the number of serial transistors inside each SCCG can be used as the first-order speed estimator.

P7 *The total number of connection points* with more than two adjacent transistors, other than V_{dd}, GROUND and the output.

P8 *The total ratio factor. The normalized ratio* of a single gate is the ratio of the number of serial to parallel transistors, normalized to a number not smaller than 1. The arithmetic product of the normalized ratios for all gates is called the total ratio factor.

P9 *The total number of external connections of gates.*

The goal is to control these parameters during the logic synthesis phase in order to experimentally verify the resulting layouts. (But, because of lack of cooperation by the programmer of TROPIC this has not been completed.) In enumerative-search procedures, such as set covering, many solutions have the same simple cost, and a random one may be selected. The reasoning here is to perform multi-criteria optimization at a slightly increased calculation cost. The author's hypothesis is that the covering methods using all these parameters will be sufficient to create nearly exact solutions.

The logic synthesis is broken into two parts (and are described in the following sections in full detail):

1. combining logical blocks using Terminal Suppressed BDDs,

2. dissected pairs (negative gates).

7.4 Terminal Suppressed BDDs

Terminal suppressed BDDs (TBDDs) are briefly presented in [128, 129]. TBDDs are appropriate for CMOS technology mapping because a TBDD will always match a SCCG, and the transistor topology of the p-MOS and n-MOS transistor networks can be obtained by graph isomorphism with 0- and 1-successor edges, respectively.

Definition 7.1. *A* **TBDD** *is defined as a BDD that has the following four properties:*

1. *only the 0-successor of a vertex is connected to the 1-terminal vertex,*

2. *only the 1-successor of a vertex is connected to the 0-terminal vertex,*

3. *there is a path that passes through all non-terminal vertices (vertex precedence condition).*

In brief, if a BDD obeys these three conditions, then it is possible to associate CMOS transistors directly in a one-to-one association with TBDD edges to obtain SCCGs.

Each node in a TBDD represents the gate input of a transistor, and the edges represent the connections to the source or drain of a transistor. For all paths through 0-successor edges, nodes are directly mapped to p-MOS transistors, and the edges give the wiring of the sources and drains of the p-MOS transistors. Likewise, all paths through 1-successor edges are mapped to n-MOS transistors and their connections. An implicit edge to the root vertex gives the connections to V_{dd} for the p-MOS transistors and the connections to the output of the gate for the n-MOS transistors. Edges connected to the 1-terminal vertex and the 0-terminal vertex give the p-MOS connections to the output of the gate and the n-MOS connections to GROUND, respectively. Example 7.2 illustrates the association of TBDDs to SCCGs.

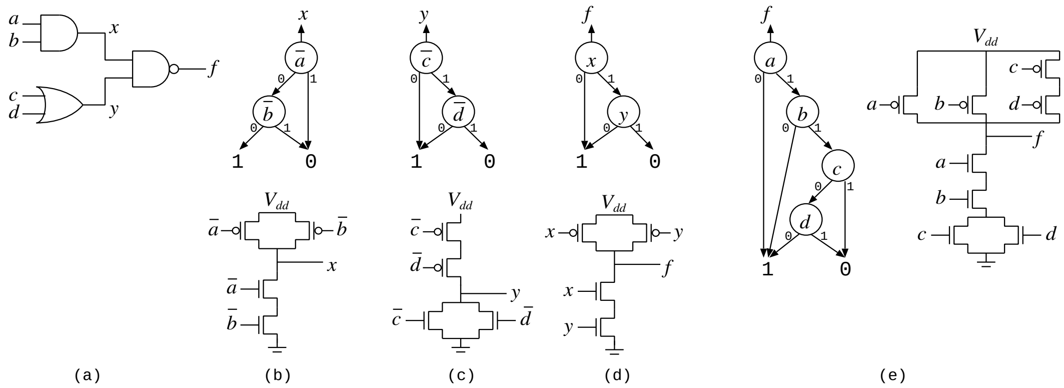


Figure 7.6: TBDD example (a) original function f (b) $x = \overline{\overline{a} + \overline{b}} = a \cdot b$ (c) $y = \overline{\overline{c} \cdot \overline{d}} = c + d$ (d) $f = \overline{x \cdot y}$ (e) $f = \overline{(a \cdot b) \cdot (c + d)}$.

Example 7.2. *The stages of technology assignment with TBDDs using a virtual library of SCCGs is shown in Figure 7.6. The input is a Boolean net-list that is broken into simple logic gates (shown in Figures 7.6b,c,d); each of the logic gates is represented by a TBDD and the corresponding CMOS layout (shown below each TBDD). If only standard cells are used to implement this function, then there are three SCCGs and twelve transistors plus eight transistors to create the negations for a, b, c and d. The three TBDDs are combined, and the resulting CMOS layout has one SCCG and eight transistors, as shown in Figure 7.6e.*

Because TBDDs have a one-to-one correlation to SCCGs, they can be used in the logic synthesis of SCCGs. Figure 7.6 shows the association of the TBDDs to SCCGs. Given that each of the TBDDs can be combined to form larger structures with the use of a TBDD *compose*-operator, the small TBDDs can be composed to create the larger TBDD in the figure. The objective of using TBDDs in logic synthesis is to symbolically create and combine SCCGs to reduce the number of SCCGs that represent a function. Generally, by reducing the number of SCCGs, the number of transistors in the network and the power consumption of the network are reduced. For this dissertation, TBDDs take small logic blocks to create larger, more complex SCCG structures. Whenever the logic blocks are not small (non-decomposable blocks), dissected pairs are used for the construction of SCCGs. The next section presents the methodology of dissected pairs.

7.5 Dissected Pairs

In order to minimize the area for inverters (as the number of inverters increase, the layout area increases), with respect to optimize timing, fan-out constraints, or because of the restrictions imposed by certain technologies, it is sometimes reasonable to assume that a subset of input variables is available in an affirmative form and some other subset in a complemented form only. It is also assumed that the logic synthesis methods for layout are based on logic blocks created during the decomposition process. These blocks, in general, are completely specified functions, and it is assumed that the input variables are in affirmative form only. The goal of this section is to present a logic synthesis methodology that reduces the number of inverters. The methodology, called dissected pairs is based on negated logic functions that are easily generated by CMOS or MOS. Which means that each SCCG produces a negated function.

7.5.1 Definition of Dissected Pairs

Let f be a single-output, completely specified function (recall the notations given in Section 2.1).

Definition 7.2. *A pair of cells* $a^k = (a^k_0, \ldots, a^k_j, \ldots, a^k_{n-1}) \sqsubseteq OFF(f)$, $a^l = (a^l_0, \ldots, a^l_j, \ldots, a^l_{n-1}) \sqsubseteq ON(f)$ *is an* **adjacent dissected pair** *when there exists*

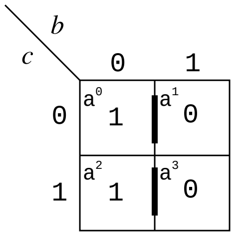


Figure 7.7: Small example of dissected pairs.

exactly one $j \in \{0, \ldots, n-1\}$ *so that the following conditions are simultaneously satisfied:*

$$a_j{}^k = 1, a_j{}^l = 0$$
$$(\forall i \in \{0, \ldots, n-1\}, i \neq j)[a_i{}^k = a_j{}^l].$$

Property 7.1. *Let* a^k/a^l *be an adjacent dissected pair, where* $a^k = (a_0^k, \ldots, a_i^k, \ldots, a_{n-1}^k)$, $a^l = (a_0^l, \ldots, a_i^l, \ldots, a_{n-1}^l)$. *This pair can be represented by the cube* $S = (S_0^{kl}, S_1^{kl}, \ldots, S_{n-1}^{kl})$, *defined as follows, where* $i = 0, \ldots, n-1$.

$$S_i^{kl} = \begin{cases} 0 & \text{when } a_i^k = a_i^l = 0, \\ 1 & \text{when } a_i^k = a_i^l = 1, \\ - & \text{when } a_i^k = 1, a_i^l = 0. \end{cases}$$

Example 7.3. *There are two adjacent dissected pairs for the function* $f(b, c)$ *as shown in Figure 7.7, 11/10* (a^3/a^2), *and 01/00* (a^1/a^0). *To illustrate the dissected pairs in the figure, a bold line is drawn between each adjacent dissected pair. The two adjacent dissected pairs can be represented by the respective cubes* $S^{32} = 1-$ *and* $S^{10} = 0-$.

A **full kernel** $FK(C)$ of a product C of variables (a vertex, a cube, a cell of a Karnaugh map) is denoted as a product of Available Literals (AL) which belong to this product. For instance, if $AL = \{x_1, x_2, \overline{x_3}, x_4\}$, then a full kernel of a product term: $x_1\overline{x_2}\ \overline{x_3}$ is $x_1\overline{x_3}$. The **positive kernel** $PK(C)$ of cube C is the product of all non-negated variables from this cube. For instance, $PK(x_1x_2\overline{x_3}x_4)$=$x_1x_2x_4$. The **negative kernel** $NK(C)$ of cube C is the product of all negated variables from this cube. For instance $NK(\overline{x_1}x_2\overline{x_3}x_4) = \overline{x_1}\ \overline{x_3}$. An **available kernel** $AK(C)$ of cube C is an arbitrary cube that fulfills the relation $FK(C) \sqsubseteq AK(C) \sqsubseteq PK(C)$. For instance, if $C = x_1x_2\overline{x_3}\ \overline{x_4}$, $FK(x_1x_2\overline{x_3}\ \overline{x_4}) = x_1x_2\overline{x_3}\ \overline{x_4}$ and $PK(x_1x_2\overline{x_3}\ \overline{x_4}) = x_1x_2$, then the available kernels are : $x_1x_2\overline{x_3}\ \overline{x_4}$, $x_1x_2\overline{x_3}$, $x_1x_2\overline{x_4}$ and x_1x_2.

7.5.2 Indispensable Dissected Pairs

Definition 7.3. *The pair a^s/a^t is said to* **dominate** *the pair a^k/a^l if each kernel dissecting the pair a^s/a^t also dissects the pair a^k/a^l.*

Definition 7.4. *A dissected pair a^k/a^l is called a* **redundant dissected pair** *if there exists some dissected pair a^s/a^t, so that a^s/a^t dominates a^k/a^l.*

Definition 7.5. *The dissected pair a^k/a^l is an* **indispensable dissected pair** *iff there exists no pair a^s/a^t that dominates a^k/a^l.*

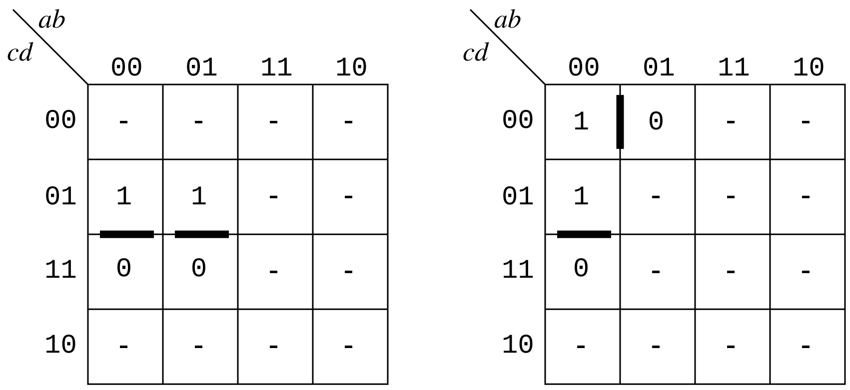
Lemma 7.1. *A dissected pair a^k/a^l is a redundant dissected pair if there exists a dissected pair a^s/a^t such that:*

$$\begin{aligned} PK(a^s) &\sqsubseteq PK(a^k) \\ NK(a^l) &\sqsubseteq NK(a^t). \end{aligned}$$

Lemma 7.2. *A dissected pair S^{kl} is a redundant dissected pair if there exist a dissected pair S^{st} such that:*

$$\begin{aligned} PK(S^{st}) &\sqsubseteq PK(S^{kl}) \\ NK(S^{kl}) &\sqsubseteq NK(S^{st}). \end{aligned}$$

A method of generating indispensable dissected pairs results from Lemmas 7.1 and 7.2. If a^s/a^t is a dissected pair, then the dissected pairs are not



(a) (b)

Figure 7.8: Dissected pairs example (a) Indispensable dissected pairs (b) non-dominating dissected pairs.

generated for any pair with a false minterm a^k and a true minterm a^l so that the conditions of this lemma are satisfied.

Example 7.4. *Consider the function $f(a,b,c,d)$ shown in Figure 7.8a, where the dissected pairs 0011/0001(00-1) and 0111/0101(01-1) are shown by a thick line in the Karnaugh map. From Lemma 7.2, $PK(\mathit{00\text{-}1}) = PK(\overline{a}\,\overline{b}cd) = \{cd\} \sqsubseteq PK(\mathit{01\text{-}1}) = PK(\overline{a}bcd) = \{bcd\}$ and $NK(\mathit{01\text{-}1}) = NK(\overline{a}b\overline{c}d) = \{ac\} \sqsubseteq NK(\mathit{00\text{-}1}) = NK(\overline{a}\,\overline{b}\,\overline{c}d) = \{abc\}$. It holds that 00-1 dominates 01-1, which implies that 00-1 is indispensable.*

Lemma 7.3. *A dissected pair a^k/a^l, where $a^k = (a_0^k, \ldots, a_i^k, \ldots, a_{n-1}^k)$, $a^l = (a_0^l, \ldots, a_i^l, \ldots, a_{n-1}^l)$, is redundant if the result of the operation $DOM(S^{st}, S^{kl})$ is a unit cube 1_n. Operation* **DOM** *is defined as follows:*

$$DOM(S^{st}, S^{kl}) = \begin{cases} \text{true} & (\forall i)[dom(S_i^{st}, S_i^{kl}) = X], \\ \text{false} & \text{all other cases,} \end{cases}$$

where **dom** *is the operator specified in Figure 7.9.*

The above properties can be used by a searching algorithm to find all dissected and indispensable dissected pairs. Example 7.5 illustrates the *DOM*

$S^{st} \backslash S^{kl}$	0	1	-
0	X	X	X
1	0	X	X
-	0	0	X

Figure 7.9: Operator $dom(S_i^{st}, S_i^{kl})$.

operator.

Example 7.5. *From Figure 7.8a, the two dissected pairs with variable ordering abcd: 00-1 and 01-1 are given and used by the operator DOM in the following:*

$$DOM(\mathit{00\text{-}1{,}01\text{-}1}) = dom(\{\mathit{00\text{-}1}\}, \{\mathit{01\text{-}1}\}) = \mathit{1111}.$$

Because the output of DOM is a unit cube 1_4*, 00-1 dominates 01-1. Figure 7.8b shows two dissected pairs 0-00, 00-1.*

$$DOM(\mathit{0\text{-}00{,}00\text{-}1}) = dom(\{\mathit{0\text{-}00}\}, \{\mathit{00\text{-}1}\}) = \mathit{1011}.$$

Because the output of DOM is **not** *a unit cube* 1_4*, 0-00 does not dominate 00-1.*

Now that the concept of dissected pairs has been developed, it is necessary to explain how to apply them to SCCGs. From Example 7.1, the use of dissected pairs created two SCCGs. The two SCCGs are known as the first and second levels of the synthesis. The second level is known as the negated set of cubes (from Example 7.1: Z). The first level is known as the positive set of cubes. The minimization of the first and second levels is given in the following sections.

7.6 Minimization of the Second Level

The experimental results from the papers for TANT networks discussed in Subsection 7.2.5 leads to the problem of minimizing the second level in two level complex gate network synthesis. To illustrate the problems Karnaugh maps are used, while the program is implemented with MDDs.

Theorem 7.1. *The set of positive kernels (PK) is the set P_f of the completely specified function f iff for each adjacent dissected pair of cells a^k/a^l there exists some $T \in PK$ so that T dissects this pair.*

Let T_k be the positive kernel of the minterm of $\overline{f}$ corresponding to a^k, where a^k/a^l is an indispensable dissected pair of $\overline{f}$. Let T_r be the positive kernel which satisfies the conditions:

a) $V(T_r) \subseteq V(T_k)$, where V is the vertex,

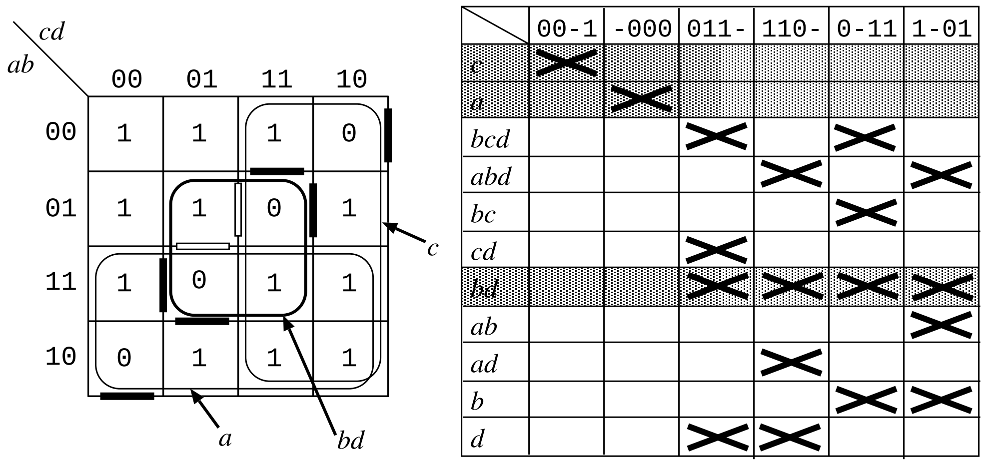
b) there exists at least one $x_i \in V(T_r)$ such that $a_i^l = 1, a_i^k = 0$.

Definition 7.6. *The set of all positive kernels, T_r, which satisfy the above conditions is called the K_l^k set for the pair a^k/a^l. The sum of all K_l^k sets for all indispensable dissected pairs a^k/a^l of f is called the K_f set.*

Theorem 7.2. $P_{f_{min}} \subseteq K_f$.

Theorem 7.3. *Each available kernel $T_r \in K_l^k$ where a^k/a^l is a redundant pair, belongs either to some K_t^s, where a^s/a^t is an indispensable pair or T_r dissects only redundant pairs.*

The procedure for obtaining $P_{f_{min}}$ is based on Theorems 7.1, 7.2, and 7.3, which are used in the construction of the table presented in Figure 7.10b. Theorem 7.1 is used to find the sets $P_{f_{min},i}$ (there may be more than one). By introducing this table, the problem of finding $P_{f_{min}}$ is reduced to the classical covering problem. This table uses positive kernels from K_f as rows and dissected pairs as columns. According to Theorem 7.3, only the indispensable dissected pairs are placed in the headings of the table's columns. An X is placed at the intersection of a row and a column if the corresponding kernel dissects the corresponding pair of cells. The table can be solved by the various covering algorithms outlined in [19, 20, 37, 56]. After solving the table, the set $P_{f_{min}}$ (one or more)



(a) (b)

Figure 7.10: Minimizing the second level.

is obtained. The positive kernels in $P_{f_{min}}$ serves to create minimal number of second level gates.

Example 7.6. *Consider the function $f(a,b,c,d)$ shown in Figure 7.10a. The dissected pairs (marked in Figure 7.10a with block between the cells) are 00-0, -000, 011-, 0-11, 01-1, 110-, -101, and 1-01. A pair a^k/a^l is dominated by the pair a^s/a^t when these pairs are dissected with respect to the same x_j and $a_i^k \geq a_i^s$ where $0 \leq i < n$. The indispensable dissected pairs, 00-0, -000, 0-11, 011-, 110-, 1-01 are marked by dark blocks in Figure 7.10a. The pairs -101 and 01-1 are dominated by the pairs -000 and 00-0, respectively.*

After solving the table presented in Figure 7.10b, the set $P_{f_{min}} = \{a, c, bd\}$ is obtained. In hand synthesis, the Karnaugh map can be used to obtain $P_{f_{min}}$ without solving the table in Figure 7.10b. After marking the Karnaugh map with all indispensable dissected pairs, the goal is to choose from K_f the minimal set of covers which dissect all marked pairs. For instance, the covers a, c and bd dissect all indispensable dissected pairs, marked in the Karnaugh map (Figure 7.10a).

7.7 Minimization of the First Level

The algorithm for the minimization of the first level is for completeness and explanation only. It follows the algorithm from [28] for minimization of the second level of the TANT network and is not efficient because it requires finding of all prime implicants of the function. The algorithm to find a set of kernels of the first level is presented in Algorithm 7.1 and illustrated in Example 7.7.

Algorithm 7.1 Minimization of the first level.

---

1. Find the complement $\overline{f}$ of function f.
2. Find set $PI(\overline{f})$ of all prime implicants of function $\overline{f}$.
3. Group prime implicants from $PI(\overline{f})$ into groups KI according to their kernels K_i. All prime implicants with the same kernel K_i are included to the same group KI.
4. For each group KI, create a sum of prime implicants from the group. These sums are called maximum implicants.
5. Solve the covering problem with minterms as columns and maximum implicants as rows of the covering table. The result is a set $SOLUTION$ with a minimum number of maximum implicants that cover all minterms.
6. Return set K of the kernels from $SOLUTION$.

Example 7.7. *The prime implicants of the complemented function* $\overline{f}(a, b, c, d)$ *are* $PI(\overline{f}) = \{\overline{a}\,\overline{c}d,\ \overline{a}\,\overline{b}d,\ c\overline{b},\ c\overline{d},\ b\overline{d},\ \overline{a}b\overline{c}\}$. *The kernels are* b, c *and* d. *The maximal implicants are*

1^{st}) $\overline{a}\,\overline{c}d + \overline{a}\,\overline{b}d = d\overline{a}(\overline{c} + \overline{b}) = d\overline{(a)}\,\overline{(cb)}$,

2^{nd}) $c\overline{b} + c\overline{d} = c\overline{(bd)}$,

3^{rd}) $b\overline{d} + \overline{a}b\overline{c} = b(\overline{d} + \overline{a}\,\overline{c}) = b(\overline{a} + \overline{d})(\overline{d} + \overline{c}) = b\overline{(ad)}\,\overline{(cd)}$.

The covering table is shown in Figure 7.11. The minimum solution to this table is the entire set of maximum implicants. The corresponding set of kernels is $\{b, c, d\}$.

maximal implicants \ minterms	1	2	3	4	5	6	10	11	12	14
$d(\overline{a})(\overline{bc})$	X		X		X					
$c(\overline{bd})$		X	X			X	X	X		X
$b(\overline{ad})(\overline{cd})$				X	X	X			X	X

Figure 7.11: Minimizing the first level.

7.8 Minimization Algorithms

The algorithm to minimize a single-output function from complex gates is presented in Algorithm 7.2 and illustrated in Example 7.8.

Algorithm 7.2 Algorithm for minimizing a two-level network with complex gates.

1. Find a complement $\overline{f}$ of a given function f. $COVER := \emptyset$.
2. Find set $DP(\overline{f})$ of dissected pairs of a function $\overline{f}$.
3. Find set EG of all elementary groups dissecting any pair from $DP(\overline{f})$.
4. Find set CG of all composed groups. A composed group is a sum of groups from any such subset EG' of set EG, that any group in EG' is not included in any other group from EG'. For each group in CG find the respective dissected pairs.
5. Create a covering table with groups from CG as rows and dissected pairs from $DP(\overline{f})$ as columns.
6. Find a minimum solution to this covering problem, i.e., find a subset CG' of set CG, that has a minimum number of groups and covers all columns.
7. Find set K, kernels of $\overline{f}$, with Algorithm 7.1.

Example 7.8. *Given the complemented function $\overline{f}$ shown in Figure 7.12a. The dissected pairs are illustrated in the figure with lines between cells. Indispensable pairs are illustrated with dark lines between the cells. The elementary dissecting*

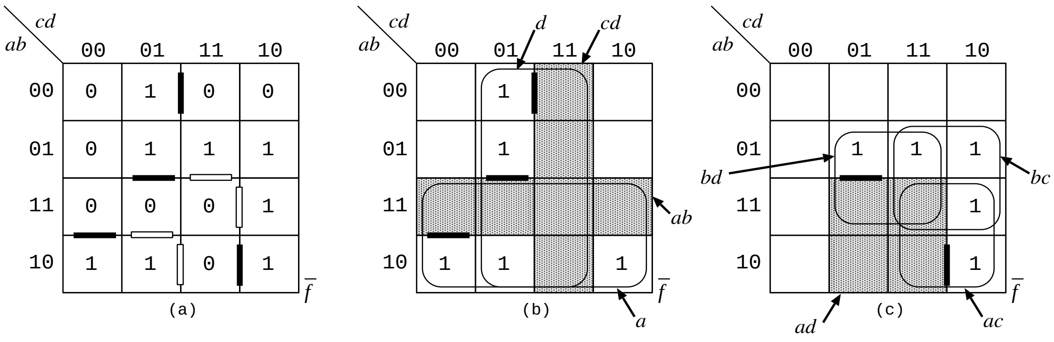


Figure 7.12: Stages of solving the function from Example 7.8.

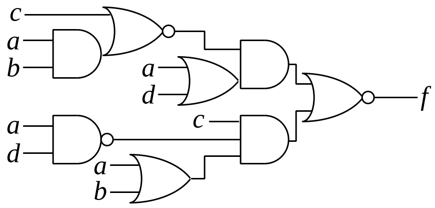


Figure 7.13: Final SCCG network.

groups are: $EG = \{ab, cd, ad, abd, acd, a, b, c, d\}$. *The set of composed dissecting groups for all dissected pairs is* $CG = \{ab + cd, ab + bd, \ldots, cd\}$. *After solving the covering problem with* CG *and the dissected pairs the minimum second level cover* $CG' = \{ab + cd\}$. *The prime implicants are:* $a\bar{b}\ \bar{c}, a\bar{b}\ \bar{d}, \bar{a}\ \bar{c}d, \bar{b}\ \bar{c}c, \bar{a}bc$, $\bar{a}bd, bc\bar{d}$, *and* $ac\bar{d}$. *The set of simple kernels found from the prime implicants is* $\{a, d, bc, bd, ac\}$.

Algorithm 7.3 presents the approximate method to find a covering with complex gates, and is illustrated in Example 7.9.

Example 7.9. *Given the function from the previous example. The minimum set of composed dissecting groups for all indispensable pairs is* $\{ab + cd\}$. *The dissected pairs not covered by this group are -111 and 111-.*

Algorithm 7.3 Approximate method to find a covering with complex gates.

1. Find the complement $\overline{f}$ of the given function f.
2. Find set $DP(\overline{f})$ of dissected pairs of a function $\overline{f}$.
3. From set $DP(\overline{f})$ find set $IDP(\overline{f})$ of indispensable dissected pairs of $\overline{f}$.
4. Find set EG of elementary groups dissecting all pairs from $IDP(\overline{f})$.
5. Find set CG of composed groups dissecting all pairs from $IDP(\overline{f})$ and covering the smallest number of dissected pairs.
6. Use Algorithm 7.1 to find the set of kernels.
7. Realize each kernel from step 6, dissecting as many pairs from $IDP(\overline{f})$ as possible. If the kernel cannot be realized, skip it. Remove the dissected pairs from $IDP(\overline{f})$ and $DP(\overline{f})$.
8. When set IDP is empty repeat step 7 (as in Algorithm 7.2) for the remaining kernels and not dissected pairs from $DP(\overline{f})$.

The set of simple kernels found from the prime implicants is $\{a, d, bc\}$. A complex implicant for kernel a can be realized with a group $\overline{ab + cd}$ (see Figure 7.12b). This implicant is $a\overline{(ab + cd)}$. Similarly a complex implicant for d is created, $d\overline{(ab + cd)}$. The set of remaining kernels is now: bc. The set of remaining dissected pairs is -111, 111-. This unequivocally determines a complex implicant $bc\overline{(ad)}$ (Figure 7.12c). The solution is $\overline{f} = (a+d)\overline{(ab + c)}+c(a+b)\overline{(ad)}$. The diagram of the corresponding complex gate network is presented in Figure 7.13.

7.9 Complex Gate Minimization

In this section, a complex gate minimization algorithm is presented for single-output, completely specified functions. The algorithm is based on a Boolean operation on two implicants. The following is the definitions used to create the Boolean operation:

An implicant of a function can be rewritten in terms of its positive kernel

PK and negative kernel NK. Given that $pk_i \in PK$ and $nk_i \in NK$, then an implicant ci (known as a **complex implicant**) can be written as:

$$\begin{aligned}
ci &= (pk_o \cdot pk_1 \cdot \ldots \cdot pk_{|PK|-1}) \cdot \overline{(nk_o \cdot nk_1 \cdot \ldots \cdot nk_{|NK|-1})} \\
&= (pk_o \cdot pk_1 \cdot \ldots \cdot pk_{|PK|-1}) \cdot \overline{(\overline{nk_o} + \overline{nk_1} + \ldots + \overline{nk_{|NK|-1}})} \\
\text{let} \quad pk &= (pk_o \cdot pk_1 \cdot \ldots \cdot pk_{|PK|-1}) \\
\text{and} \quad nk &= (\overline{nk_o} + \overline{nk_1} + \ldots + \overline{nk_{|NK|-1}}) \\
ci &= pk \cdot \overline{nk}.
\end{aligned}$$

For reference, $ci^i = pk^i \cdot \overline{nk^i}$. Given two complex implicants ci^j and ci^k, if $pk^j = pk^k$, then the complex implicants can be factorized by pk, giving the following implicant:

$$\begin{aligned}
ci^l &= pk \cdot (\overline{nk^j} + \overline{nk^k}) \\
&= pk \cdot \overline{(nk^j \cdot nk^k)}.
\end{aligned}$$

Lemma 7.4. *A* **secondary complex implicant**, *ci^l, is created by using the following Boolean operation of two complex implicants ci^j and ci^k:*

$$ci^l = (pk^j + pk^k) \cdot \overline{(nk^j \cdot nk^k)}.$$

There are cases when a secondary complex implicant cannot be created. This occurs when there does not exist a negative term that can cover all the OFF-minterms and none of the ON-minterms in both of the implicants.

Definition 7.7. *A secondary complex implicant, ci^l, exists if the following expression is true, given two complex implicants (or secondary complex implicants) ci^j and ci^k:*

$$(pk^j + pk^k) \cdot \overline{(nk^j \cdot nk^k)} = pk^j \cdot \overline{nk^j} + pk^k \cdot \overline{nk^k}.$$

There are, however, cases when

$$(pk^j + pk^k) \cdot \overline{(nk^j \cdot nk^k)} \neq pk^j \cdot \overline{nk^j} + pk^k \cdot \overline{nk^k},$$

but the functionality of the function is not changed. This occurs when ci^l covers minterms not covered by ci^j or ci^k. Creating secondary complex implicants is an iterative process and continues until no secondary complex implicants can be created.

The usage of secondary complex implicants is given in Algorithm 7.4 and Example 7.10 illustrates the algorithm.

Algorithm 7.4 Complex Gate Minimization Algorithm.

1. Find the prime implicant covers of $\overline{f}$, $PI(\overline{f})$, using a prime covering table.
2. Create complex implicants, $ci \in CI(\overline{f})$ from the set of implicants in $PI(\overline{f})$.
3. Find factorized forms of elements in $CI(\overline{f})$.
4. Create a secondary complex implicants and add them to set $CI(\overline{f})$.
5. Create a complex implicant covering table composed of the pk elements of each complex implicant as the columns in the table, and each $ci^i \in CI(\overline{f})$ as the rows in the table.
6. Solve the complex implicant covering table by evaluating the cost of each solution as a weighted function of the parameters given in Section 7.3. Select a subset of rows that covers all columns and minimizes the weighted cost function, of the form $\sum_{i=1}^{9} \alpha_i P_i$.
7. End. (Possibly repeat all steps using f instead of $\overline{f}$, because a reduced set of sufficient implicants will lead to a better solution.)

	SIS	COMPLEX
transistors	20	18
number of nets	12	11
number of rows	3	2
Width/heigth (μm)	6.70 x 37.80	6.70 x 25.20
Area (mm^2)	0.00025	0.00017
Density (xtor/mm^2)	78970.23	106609.80

Table 7.3: Comparison of SIS and COMPLEX for Example 7.10.

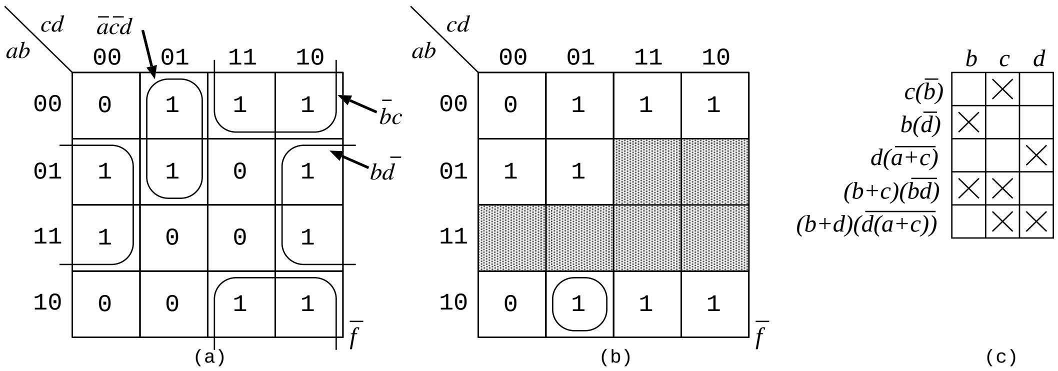


Figure 7.14: Using COMPLEX to solve the function in Example 7.10.

Example 7.10. *Given the function $\overline{f}(a, b, c, d)$ shown in Figure 7.14a.*

1. *Find all the essential implicants of $\overline{f}$. The minterms 1011 and 1100 produce the respective essential primes, $\overline{b}c$ and $b\overline{d}$, as shown in Figure 7.14a.*

2. *From the remaining two minterms the cover $\overline{a}\ \overline{c}d$ is selected.*

3. *The kernels for $\overline{f}$ are $\{c, b, d\}$.*

4. *There are no implicants with the same kernel, so nothing needs to be done here.*

5. *Initially, the set CI contains the elements: $(c)\overline{(b)}$, $(b)\overline{(d)}$, $(d)\overline{(a + c)}$ which are placed in the complex implicant covering table shown in Figure 7.14c.*

6. *By combining the elements in CI, the secondary complex implicants are found as $(b + c)\overline{(bd)}$ and $(b + d)\overline{(d(a + c))}$. Notice that the kernels c and d could not be combined. In this case, if c and d are combined, the resulting secondary complex implicant would be $(c + d)\overline{(b(a + c))}$ and is shown in Figure 7.14b. This does not cover the same set of minterms as the initial complex implicants.*

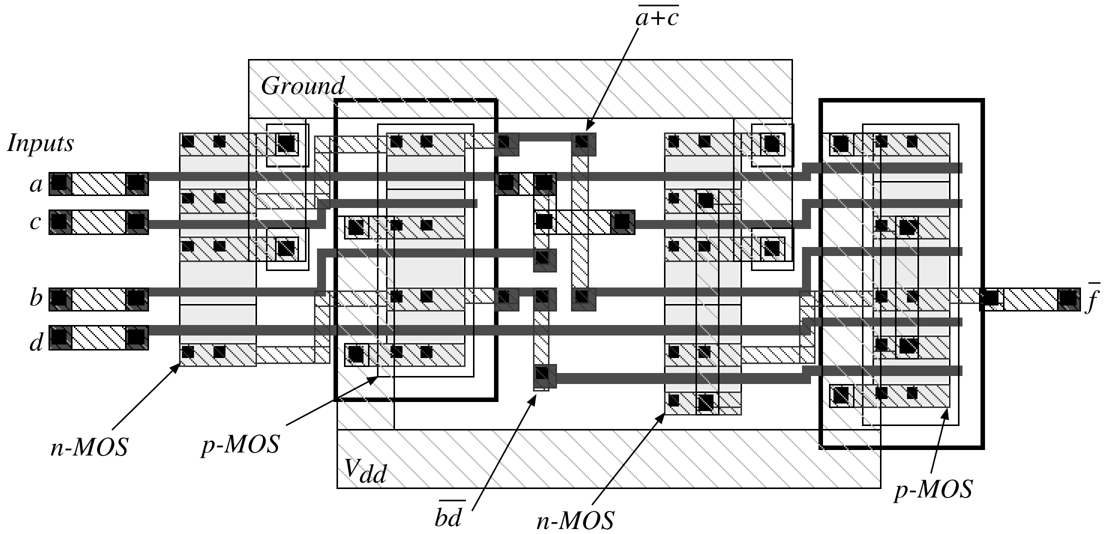


Figure 7.15: VLSI layout of $\overline{f} = d\overline{(a+c)} + (b+c)\overline{(bd)}$.

7. *The complex implicant covering table is shown in Figure 7.14c. The resulting minimized network is given by the function* $\overline{f} = (b+c)\overline{(bd)} + (d)\overline{(a+c)}$.

The VLSI layout generated from TROPIC for $\overline{f}$ *is shown in Figure 7.15. For comparison, SIS found the following expression:*

$$\overline{f} = \overline{(b+c+d)\cdot(\overline{d} + \overline{(a\overline{c})}\cdot\overline{(bc)})}.$$

Using TBDDs to compress this expression into SCCGs, the following is found:

$$Z = d \cdot (bc + a\overline{c})$$
$$\overline{f} = \overline{\overline{Z}\cdot(b+c+d)}.$$

Table 7.3 presents the results of mapping the expressions from COMPLEX and SIS to silicon. While this example is fairly small it illustrates that using COMPLEX as the logic synthesis method for TROPIC results in smaller area.

7.10 Results of Dissected Pairs for SCCGs

The author presented a dissected pair algorithm similar to the one presented in the previous section in [53, 54]. After running COMPLEX on a number of benchmark functions, the following benchmarks were chosen because of their complexity. That is, functions that are a fairly good test of COMPLEX. Most of the functions are eight-variable functions, the only exception is the *parity* function of four variables. Also, no decomposition logic synthesis was done before running COMPLEX. Such functions, like *parity* (*parity* is a symmetric function), will never be given to COMPLEX because they are easily decomposable. The reason that the eight-variable parity benchmark function is not used is that COMPLEX generates a huge covering table, 128 kernels by 1000 rows. Even though many of the solutions are exactly the same, COMPLEX is not able to manipulate this size of covering table. Note, that the methodology of this algorithm is two-level synthesis, and the goal is to use COMPLEX only on decomposed functions. This is the reason for evaluating the COMPLEX program on small functions and does not cause much concern that COMPLEX is not able to minimize the eight-variable *parity* function.

The experimental results were generated by running COMPLEX on a benchmark function and passing the best result of the function to the TROPIC program. For comparison, SIS, using a multi-level minimization method, was also used as an input to the TROPIC system. The experimental results are shown in Table 7.4. In the table, the columns under each program are the parameters P1 (number of transistors), P2 (number of SCCGs), and P5 (number of serial SCCGs) which are described in Section 7.3. Delay is the estimated timing delay, in nano-seconds, of the resulting function after layout. The other parameters given in Section 7.3 are not shown because the information found within each of these parameters does not give values that can be used for comparison. The logic net-lists generated by COMPLEX and SIS were converted to transistor net-lists with the number of serial transistors set to a maximum of four-serial transistors [128, 129]. The transistor net-lists were then used as an input to TROPIC.

function	SIS multi-level				COMPLEX			
	P1	P2	P5	Delay	P1	P2	P5	Delay
ch22f0	40	9	5	1.88	40	3	3	2.14
ch47f0	94	17	9	4.58	78	6	4	2.65
or_and_chain	28	6	7	2.05	22	4	5	1.75
substr1	54	10	6	2.06	46	6	5	2.04
parity(4 var)	52	10	5	1.90	66	7	5	2.33
ch30f0	66	12	7	3.62	58	5	5	2.63
ch74f1	120	20	10	4.66	82	8	5	3.07
modulus2	96	18	8	3.10	76	10	5	2.70
rnd_m10	148	27	9	3.25	160	21	7	3.68
pal	160	28	7	2.84	320	36	10	6.06

Table 7.4: Comparison of SIS and COMPLEX without using SBSD as a pre-processor.

The experimental results show that COMPLEX performed very well compared to the multi-level minimization method SIS. The only function that COMPLEX did not perform well on was the *pal* function which is a completely symmetric function, much like parity. This is expected because SIS should always beat COMPLEX on these types of functions. Also, for the function *rndm10*, COMPLEX returned a larger transistor count, but the number of SCCGs was smaller.

Further comparisons of COMPLEX are shown in Chapter 9, where the entire logic synthesis package is compared to SIS. That is, the experimental results of the SBSD, TBDD and COMPLEX algorithms is presented.

7.11 Use of Small Bound Set Decomposition for Lattice Diagrams

Traditionally, implementation of circuits has been broken into disjoint tasks of logic synthesis and physical design. First, a synthesizer estimates circuit area and speed independently of target technology. A layout engine then maps synthesis results to hardware. This approach works well with circuits designed with rules of older manufacturing processes. For such circuits and processes, a good area metric is the total number of devices used, and a good speed metric is the length of the longest chain of devices. Area estimates based entirely on device count are used because wires are very small compared to device size for older technologies and because wires can pass over regions of die already used to implement transistors. Speed estimates based entirely on device count are also fairly accurate because interconnect resistance/capacitance delay typically contributes negligible amounts to total speed compared to device propagation delay.

As circuit size, connectivity increase and gate dimensions approach 0.1 micron, problems of logic synthesis and physical design become more closely interrelated. For VLSI and deep submicron processes, synthesis processes are required which optimize decomposition with respect to implementation [5] resource topology. Synthesis results should map directly to hardware with minimal interconnect delays.

One way to control the complexity of such layout-driven synthesis is to use regular hardware architectures that are tuned to specific types of logic decomposition. An early example of this is Akers' rectangular logic array [5] which is a two-dimensional, planar grid of small cells. A benefit of this structure is that each cell connects to its four neighbors, thereby limiting the distance most signals are required to travel. However, because Akers designed his synthesis method for worstcase problems, it produces very large and sparse arrays for many common circuits.

Akers' approach could benefit from symmetry information of functions to be processed. Symmetry indicates parts of a function that are used more than once. Methods which detect symmetry can save area by implementing symmetrical parts only one time in hardware. Target architectures optimized for this type of sub-function sharing can also minimize interconnect delays.

A **lattice** is an array of hardware logic cells which is interconnected by a regular pattern of wires. There are two kinds of interconnects among cells:

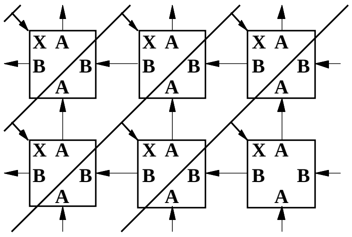


Figure 7.16: Lattice interconnect pattern.

global and local. The global interconnects distribute common signals to multiple cells, and the local interconnects transmit information between two physically neighboring cells. Examples of local connection types include inverted and non-inverted wires. Cells of a given lattice can only implement logic from a single, finite library of possible functions. Similarly, a given lattice has a library of possible wire functions and signal capacities. Each wire implements a subset of wire functionality from the same library and transmits a number of signals equal or less than the wire capacity for a given lattice. For this dissertation, a lattice is defined as a two-dimensional grid of cells. Each cell connects to its immediate four neighbors as well as all cells on its positive diagonal (see Figure 7.16). All cells contain the same logic elements and can represent different simple structures, e.g., Shannon decomposition with a simple multiplexor. More complex cells may be created which can give more functionality to each cell and may reduce to overall size of the lattice.

A **Lattice Decision Diagram** (LDD) is a Decision Diagram (DD) with respect to a lattice L given the following:

1. if the logic of any vertex in a DD can be implemented by any cell in L,

2. any edge in a DD can be implemented by a wire in L,

3. the pattern of edges connecting vertices in a DD can be implemented by interconnected resources of L in a regular pattern.

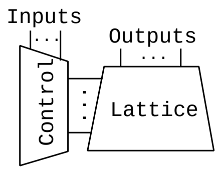


Figure 7.17: Decomposition-based synthesis for lattice diagrams block diagram.

It should be noted that the decomposition methodologies presented in the previous chapter could be used to create lattice diagrams [39]. The lattice diagram synthesis approach constructs, for combinational functions, regular lattices with only local connections and input buses. The advantage of lattice diagrams is that they are directly mappable to hardware without additional layout steps. To construct small lattices, the synthesis approach decomposes functions symmetrically using Shannon expansion and by repeating variables when necessary. In general the construction of lattice diagrams is based on the functional decomposition of a function to a regular structure of simpler functions using symmetries much like all of the decomposition methods presented in Chapter 5.

The problem with DD variable reordering is that it does not take advantage of common sub-functions that can easily be found using decomposition. That is, given a function that has several sub-functions at level X that are identical, force the sub-functions onto the same logic cell by "creating better variables". This creates new variables based on the information found in the function. Figure 7.17 illustrates the block diagram of decomposition-based synthesis for lattice diagrams. The idea is to decompose a function, so that simple logic blocks are found during the decomposition and used as control variables to the lattice diagrams. The figure shows the control logic on the left of the diagram and the lattice diagram on the right. The following example illustrates the concept of using decomposition for lattice diagram creation.

Example 7.11. *Figure 7.18 shows control logic that is used to control the data flow through the lattice diagram. Decomposition is used to determine logic blocks*

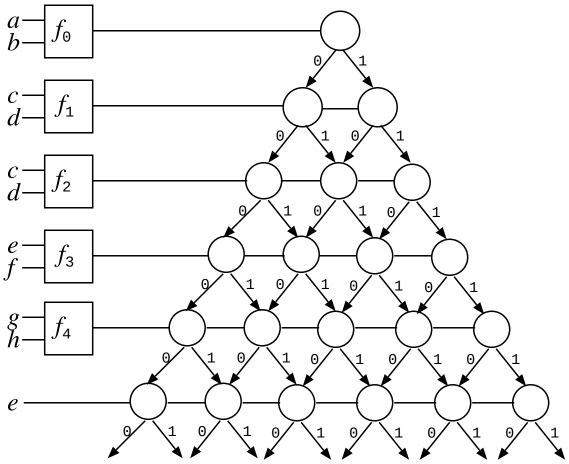


Figure 7.18: Example of decomposition based synthesis for lattice diagrams.

in the function that can be used to control the data in the lattice diagram. For example, decomposition finds that variables a and b are a symmetrical pair and result in a two-input, one-output logic block. Several other logic blocks are shown in the figure to illustrate the preferred layout. Notice that a logic block can just be a single variable, and it is possible that the same input be used for multiple logic blocks.

Here a set of variables in the initial function f are computed in an external block outside of the lattice diagram. This is not without a cost; the computation outside of the lattice diagram may introduce latency, delay, or bus-routing problems. But the advantage is that large sections of the lattice diagram may be deleted, which largely reduces the size of the lattice diagram. Again, the cost of this method is based on the particular technology that the function is being implemented in.

The redundancy of variables in this method is that variables may be used external to the lattice diagram and inside of the lattice diagram.

Example 7.12. *Figure 7.19a shows a BDD for a given function with input variables x, y, and z. The labels A, B, C and D denote sub-functions of the*

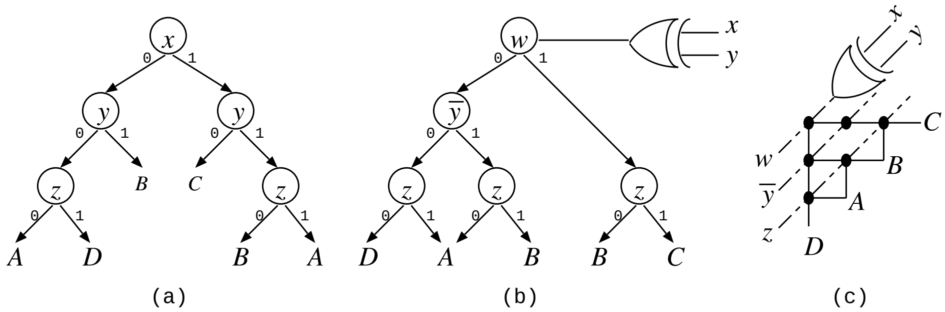


Figure 7.19: Lattice example (a) original BDD (b) creating new variable $w = x \oplus y$ (c) new lattice diagram.

function f and are used to denote sub-functions that are equal.

After analyzing the BDD, the idea is to combine sub-functions in the easiest possible method. There are two sub-functions labeled A, and two sub-functions labeled B. To combine them, force the A vertices together by creating a new variable $w = x \oplus y$. This creates the BDD shown in Figure 7.19b which maps directly to the lattice diagram in Figure 7.19c. The solution uses only one shared variable, x, and the size of the linear diagram is reduced by six cells with only the addition of the external cell that finds $x \oplus y$.

Example 7.12 can be used with SBSD algorithm presented in the previous chapter and another example of using SBSD is illustrated in Example 7.13.

Example 7.13. *Consider the function in Figure 7.20a. The BDD shown in Figure 7.20b is not a LDD because it cannot be mapped directly onto a lattice. The BDD has too many vertices for variable d (variable d is the fourth variable, only four vertices are allowed). SBSD is used to determine that a and b can be placed in the bound set, with column multiplicity three. Encoding is used to find $\Phi_1 = a \cdot b$ and $\Phi_2 = a + b$. Using Φ_1 and Φ_2, the LDD can be created as shown in Figure 7.20c. Not only does this produce a LDD, but it also frees one vertex on the variable c level, and two vertices on the variable d level.*

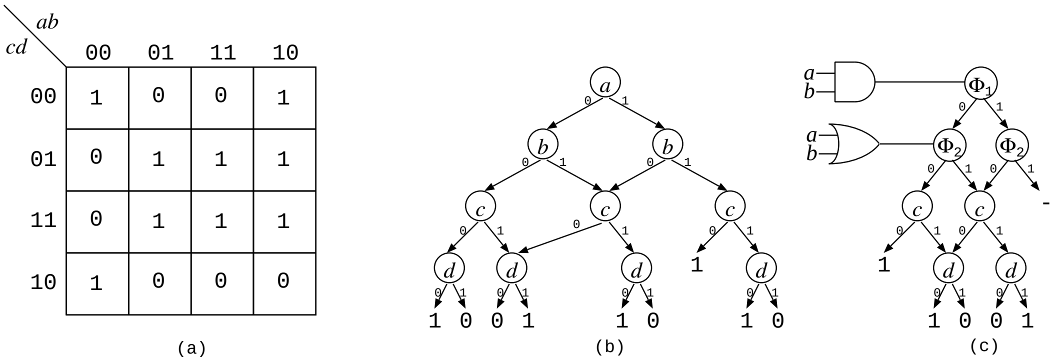


Figure 7.20: Converting a BDD into a LDD by introducing functional inputs of the variables (a) Karnaugh map of function (b) BDD (c) LDD after creating new variables.

7.12 Decomposition for Layout Conclusions

A technique to minimize two-level networks from complex gates has been discussed. A new factorization operation was introduced that finds good secondary complex implicants without creating all possibilities and leads to "high quality" results. However, contrary to the method presented in [53, 54] the author was not able to prove that the exact minimum of the first or second level is achieved. One method of improving this approach would be to repeat the algorithm for any set of prime kernels from a minimal prime cover. Further experimental results of this logic synthesis method are presented in Chapter 9.

Also presented was a simple discussion of lattice diagrams. A methodology to apply functional decomposition to lattice diagrams is proposed. While this methodology appears to be very promising, no experimental results are explored in this dissertation and the idea is left for possible future work.

Chapter 8

Machine Learning

The central idea in machine learning is to gather information from a given data set. This can be a very difficult task because practical databases are usually very large. To address this difficulty, the assumption of Occam's Razor (simplest is best) and *explicit domain knowledge* are used to reduce the search space [12, 15]. Machine learning is not an exact methodology or formal theory of learning, but is an attempt at learning based on heuristics and probabilities. Machine learning is becoming more and more important as computers provide relatively inexpensive means to collect and store data. Traditional methods such as standard statistical methods are insufficient to fully explore all possibilities or learn a given data set. Instead, knowledge discovery in databases and data mining techniques are analytical learning tools developed in domains of statistics, pattern recognition, and artificial intelligence.

Amazingly, while the logic synthesis community has worked on reducing the complexity of Boolean functions, until recently the machine learning community has not used this work for their own set of problems. Concepts that have been known in the logic synthesis community for some time, have had to be re-learned by the machine learning community. Recently several papers have been written on rule generation systems that are based on circuit design techniques [8, 62]. The bridge between the two communities, called *pattern theory* [130], is an approach to finding algorithms for both the logic synthesis community and the machine learning community. Pattern theory attempts to analyze the behavior of these algorithms using COmputational Learning Theory (COLT) [7]. COLT ascertains how much information is needed for the algorithm to meet some probabilistic criteria of correctness. The findings so far show that pattern theory algorithms, based on logic synthesis, are very promising in finding solutions for machine learning [133]. However, the computational complexity of these algorithms still remains an important problem.

8.1 Machine Learning Introduction

Within the structure of this dissertation, the term **machine learning** is defined as a process by which a machine gains the capability to solve a problem by examining *examples* or data. Machine learning, defined here, is a process called *inductive learning* [93, 125] which uses empirical evidence, in the form of examples, to derive *rules* for the given data. These systems use *automated inductive inference* (in the machine learning paradigm) to give rules or meaning to the data (known as *classification*). **Rules** are combinations operations on variables, such as the average function, addition, subtraction, etc. Finding rules on variables is a process of discovering patterns and relationships that exist between variable combinations [130]. In other words, learning is done by using all the known outputs of a given function to help in the determination of values for all the don't knows in the output of the given function. By treating examples as cares and considering functions with many don't knows, a machine learning problem can be directly converted to a logic synthesis problem.

Definition 8.1. *The components of a function $y = f(x_0,\ x_1, \ldots, x_{n-1})$, where the set $\{x_0,\ x_1, \ldots, x_{n-1}\}$ is defined as the* **input attributes**, *minterms are defined as* **examples**, *and the output y is the* **target concept** *to be learned.*

Definition 8.2. *During the decomposition of machine learning functions, $y = F(x_0, x_1, \ldots, x_{n-1})$ any x_i is known as either an input attribute or an* **intermediate concept** *(e.g., a Φ composite function in Curtis' decomposition).*

The setting of don't knows to values is done by creating a network of multi-valued input and multi-valued output blocks by decomposing the original function into a hierarchical network of multi-leveled blocks (intermediate concepts). A machine learning algorithm is evaluated on its learning effectiveness by how it reduces the *error* of the resulting network. **Error** is how well the algorithm in question sets don't know terms to care terms. A common method for evaluating a machine learning algorithm is to select a *training set*, which is a random sampling of the original known values from the *test* function. The result of learning the *training set* is then compared to the original test function. If the expression has a high error rate then it does not approximate the *test* function well and is not a useful way to describe the function.

An example of a machine learning problem is teaching a computer to learn the concept of a tree. The user can specify data, stating that a tree has roots in the ground, has branches, and is made of wood. In most situations the more information given should lead to a better concept. Occasionally though, more information (e.g., samples, variables) may actually remove some items from the group. For instance, adding the concept that a tree has leaves would exclude all conifer trees, or deciduous trees in the winter time. By adding information it is possible that some elements may not be defined in the set. Without concise information about a subject, items that do not belong in the concept may be included. For example, given the initial description of a tree, a telephone pole is a tree.

Given that induction is a method of extrapolating samples of a function, the extrapolating process is very complex unless some reasonable simplification assumptions are used. For a Boolean function over n-attributes, the function has a truth table with 2^n rows. Any truth table with 2^n rows can represent 2^{2^n} different functions. Because of the large number of possible functions, it is difficult to find a hypothesis function g that approximates f given a small set of examples of f.

To allow reasonable results in the extrapolation process for finding g, assumptions must be used. The assumption that is generally made is the one of low complexity, as in **Occam's Razor** (also known as Ockham's Razor). In Occam's Razor the most likely hypothesis is the simplest one that is consistent with all observations. This assumption has been applied and theoretically analyzed in several papers, including [12, 15]. Plus, Occam-based learning theory has been extended to actual implementations. One such implementation is Abductory Inference Mechanism (AIM[1]) [100] which creates networks of polynomials. The resulting "expert" is a network with the least complexity based on the size of a given polynomial network and how well the polynomial network maps to the given data.

Another implementation, C4.5 [126] is based on decision trees, where the resulting network is determined by comparing the sizes of decision trees and selecting the least complex or the smallest decision tree. The decision tree implementation used in C4.5 can be thought of as a network of multi-valued multiplexors, where the least complex solution is the smallest network from these gates. Note that both of these implementations can be applied to data with noise, continuous variables, and conflicting data. In Section 8.3, aspects of learning and their corresponding types of data are explained in more detail, but even though functional decomposition was initially created for discrete Boolean values; the functional decomposition has been used in solving problems with all of these types of data [119, 35, 72, 132, 154].

One drawback of both C4.5 and AIM is their use of a narrow complexity measure to determine the effectiveness of a solution. For instance, neither of these programs finds the parity function to have low complexity, even though parity is highly patterned, and can easily be computed [130]. One solution for determining complexity was found by [130] to convert the learning problem to pattern theory, where small complexity is defined as the existence of a pattern in the data. Given that complexity is based on combinational methods, there are many possible measures of complexity that can be used. Each of these complexities is based on the number of basis functions (gates/sub-blocks) necessary to represent a function. The complexity measure selected by Ross *et al.* is called **Decomposed Function Cardinality** (DFC) [2]. Recall that DFC (Sec-

[1] Trademark of AbTech Corporation

tion 4.4) is the sum of the cardinalities of the intermediate concepts/composite functions in a combinational representation, when that sum has been minimized as defined in Section 4.4. The cardinality of an n-input m-output binary function is $2^n * m$. For Boolean gates, a two-input AND-gate has the same cardinality as a two-input XOR-gate.

So why use functional decomposition for machine learning? Functional decomposition has the advantage that it is not based on any given set of operators or gates (in contrast to other logic synthesis methods, C4.5, AIM, neural networks, fuzzy logic, etc.). This is especially notable in the case of multi-valued logic, where the number of useful elementary operators increases quickly with the radix (number of values) of the logic. Decomposition is not constrained by a technology of realizing particular types of logic functions or a preselected theory of synthesis. Decomposition has wide applications in both logic synthesis and machine learning. After decomposition, the sub-blocks are small, so a highly sophisticated and/or specialized logic synthesis/rule derivation methods can be used to optimize them [105]. Since the publications by Miller [96], Epstein, Liu, Muzio [43], it has been determined that functional decomposition used in machine learning can be used to discover optimal representations of a function. In essence, functional decomposition can be used to find a hypothesis function that is low in complexity.

This chapter discusses the implementation issues of using functional decomposition (or any logic minimizer) in machine learning problems. The chapter also presents several machine learning concepts that cannot be easily mapped to logic synthesis methodologies, and possible solutions to these concepts are given.

8.2 Previous Results of Using Logic Synthesis for Machine Learning

Given that machine learning has been around since the 1950s, why would the multi-valued logic synthesis community want to get involved? This section may not fully answer this question, but it does develop a comparison between the two methods. Among these existing approaches, the logic synthesis approach does much better than previous machine learning approaches because machine learning problems often have multi-valued outputs and inputs. This is the reason for choosing multi-valued functional decomposition; it does not put constraints on the operations available in multi-valued systems or it can give appropriate operations to achieve an optimal decomposed structure.

The first example of how well Boolean logic synthesis compares to machine learning concepts is found in [57] which compares C4.5, functional decomposition (using DFC as the measure of complexity) and Espresso. Espresso is a logic minimization tool distributed by the University of California at Berkeley [75] (an industry standard tool for circuit design of accepted quality). The data found in [57] is shown in Table 8.1. Each of the two-valued functions in Table 8.1, using operations AND, OR, EXOR and COMPLEMENT, are assumed to have eight-input variables. But for some of the functions tested, there are several variables that are vacuous. If these functions were completely specified, determining whether the variables are vacuous is a trivial issue. When the functions are highly unspecified it becomes very difficult to determine which variables are vacuous and which variables actually have an effect on the output of the function. In Table 8.1 each method was given a random set of data to train on, ranging from 25 to 250 out of the 256 (2^8) possible cases. Once each method was trained, the result was compared to the original test function, the number of differences in the 256 output values were recorded as errors. This is under the section "Average Error", where the values can range from 0 to 100, with 0 error being the best. The data present under the "Number of Samples", show the number of random samples (examples) necessary before the learning method obtained a concept where the error in the result is 0%. By looking at the averages at the bottom of the tables, notice that decomposition outperforms C4.5 and Espresso in both average error and number of samples.

Perkowski *et al.* [123] applied the concept of machine learning to a Boolean logic synthesis tool that minimizes EXOR of products expressions and showed

Original Function	Known DFC	Average Error			Number of Samples		
		C4.5	Decomp.	Espresso	C4.5	Decomp.	Espresso
kdd1	2	0	0	0	8	7	9
kdd2	8	0.32	0	0.96	31	25	40
kdd3	8	6.35	0	5.64	83	25	51
kdd4	12	2.48	3.72	2.64	74	67	76
kdd5	12	1.28	2.72	3.52	61	76	54
kdd6	16	2.76	2.4	12.86	97	126	113
kdd7	20	17.52	8.18	17.16	200	60	181
kdd8	20	13.79	6.55	16.54	224	104	205
kdd9	28	20.69	10.53	5.69	256	126	51
kdd10	36	10.52	11.11	8.44	249	251	229
Average		7.57	4.52	7.35	128.3	86.7	100.9

Table 8.1: A machine learning approach versus several logic synthesis approaches [57].

that this type of synthesis performs as well as C4.5 on functions with a small and medium percentage of don't knows but did not do very well with functions with a large number of don't knows. Their conclusion was that EXOR synthesis modified for strongly unspecified functions is a viable method that could complement existing machine learning techniques.

Another result of decomposition being applied to machine learning is shown by Zupan [154] wherein many results of a described multi-valued decomposition tool, HINT is compared to C4.5 and other machine learning techniques such as neural networks. In most cases, HINT learned significantly better than C4.5. Interestingly, HINT was compared with other machine learning programs on data with continuous data and data that contains noise. The continuous data was discretized using the method introduced by Kononenko *et al.* [72].

The experimental results from previous research show that decomposition and logic synthesis methods can be used in the area of machine learning. In fact, this section has presented previous experimental results where logic synthesis tools have outperformed the machine learning program C4.5.

8.3 Machine Learning Concepts

This section gives an overview of reducing a machine learning problem to multi-valued functional decomposition. While there are many aspects that are the same between the two methodologies, this section also shows the differences between the two. Also presented are some difficult issues that the machine learning community has not fully analyzed and implemented.

8.3.1 Machine Learning Examples

An **example** is described by the input attributes and the value of the output of the function (target concept). A complete set of examples is called the **training set**. A training set does not imply that the set fully explores all possible inputs and outputs of a function. For example, in the case of a cancer database, doctors may take data about a person's age, sex, height, weight, x-rays, etc., and whether the person has cancer or not. The number of input data values (variables) that are taken can be enormous while the number of patients that are evaluated in the database can be quite small. For instance, if a doctor asks 20 questions about sex, height, etc., but only evaluates 500 patients, then the number of examples is quite small compared to the number of possible outcomes given the number of questions. In this example, if each question was a yes/no question, then there are 2^{20} $\approx$1-million combinations given different values of the inputs, while only 500 patients were evaluated. The training set is only a small representation of the possible values that exist, each example is only a small piece or an "example" of the actual function.

Mapping from machine learning to logic synthesis is done by mapping the examples to care terms. But, the number of don't knows in the function that will be evaluated with logic synthesis algorithms is quite large. In fact, it is thought that most machine learning problems have more than 99.9% output don't knows. Consequently, to use a logic synthesis algorithm on machine learning problems, it must be efficient for functions with a significantly large number of don't knows.

Table 8.2 shows the percentage of known output values for a set of machine learning benchmark functions. On some small functions, the functions are completely specified. In general though, most of the functions shown in the table have a very small percentage of known output values. In fact, the benchmark function *prison* is only $1.27 \times 10^{-68}\%$ known ($7.87 \times 10^{67}\%$ unknown). This leads to a percentage of unknown values that is much larger than that of the 99.9% unknown values discussed above.

8.3.2 Missing Attributes

The possibility exists that data collected could have *missing attributes* in the example. A missing attribute is represented by an input don't know to denote the missing data. To associate the concept of a missing attribute to that of logic synthesis, an input don't know is akin to an input don't care. In logic synthesis, an input A is a don't care when, given a set of all other inputs, all possible values of A do not result in a change in the output of the function.

Definition 8.3. *An* **input don't care** *is defined as covering all possible values for a given variable.*

In machine learning, the possibility exists that data collected could have a missing attribute in the data.

Definition 8.4. *An* **input don't know** *is defined as representing one (unknown) data value, not all possible values for a given variable.*

For example, a questionnaire asks for a person's age. It is possible that the person does not want to state his/her age; in general, instead of guessing at an approximate age, a don't know is placed in the category. In terms of logic synthesis, a don't care implies that the person is all the ages between 0 and 150. While a don't know implies that a person has only one age between 0 and 150.

One possible method of determining missing attributes is to find two input cubes C_1 and C_2 that intersect and have the same output value. Thus, $C_1 \cap C_2 \neq$

Function	number inputs	number examples	number possible	percentage known values
audiology	69	200	1.12e+23	1.78e-19%
breastc	9	699	1.00e+9	6.99e-05%
bridges1	9	108	27648	0.39%
bridges2	10	108	82944	0.13%
car	6	1728	1728	100%
chess1	6	28056	65536	42.81%
chess2	36	3198	1.03e+11	3.10e-06%
cloud	6	108	209952	0.05%
connect-4	42	67557	1.09e+20	6.17e-14%
employ1	9	9600	9600	100%
employ2	7	18000	18000	100%
flag	28	194	1.14e+16	1.71e-12%
flare1	10	323	48384	0.67%
flare2	10	1065	48384	2.20%
house-votes	16	435	65536	0.66%
letter	16	20000	1.85e+19	1.08e-13%
lung-cancer	56	32	5.19e+33	6.16e-31%
monks1te	6	432	432	100%
monks1tr	6	124	432	28.70%
monks2te	6	432	432	100%
monks2tr	6	169	432	39.12%
monks3te	6	432	432	100%
monks3tr	6	122	432	28.24%
mushroom	22	8124	1.95e+14	4.16e-09%
nursery	8	12960	12960	100%
post-op	8	90	3888	2.31%
prison1	80	258	2.03e+72	1.27e-68%
programm	12	20000	2.99e+7	0.67%
sensory	11	576	221184	0.26%
shuttlem	6	256	256	100%
sleep	9	62	1.17e+8	5.26e-05%
sponge	44	76	1.79e+23	4.25e-20%
tic-tac-toe	9	958	19683	4.86%
trains	32	11	2.44e+14	4.52e-12%
zoo	16	101	294912	0.03%
total	658	197968	2.03e+72	9.75e-66%

Table 8.2: Number of don't knows in example benchmark functions.

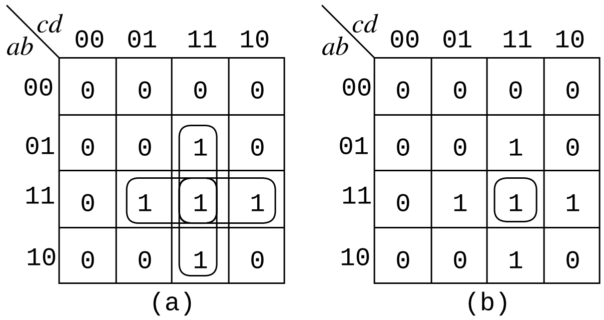


Figure 8.1: Don't knows in the input variables.

$\emptyset$ and the minimal solution for the missing attributes for these two cubes is $C_1 \cap C_2$. This is illustrated in the next example.

Example 8.1. *Given the set of input cubes (a dash indicates a don't know, or a don't care):* $-bcd$, $a-cd$, $ab-d$, *and* $abc-$. *Figure 8.1a shows the logic synthesis reduction method, where all the minterms are placed in essential prime implicants. The result is a minimum circuit with four product terms and one sum term. Figure 8.1b shows the machine learning reduction. The cube:* $-bcd$, *is thought to have a don't know such that* $-bcd$ *specifies that either the minterm* $abcd$ *exists or the minterm* $\overline{a}bcd$ *exists. Because the intersection of all the input cubes is* $abcd$, *the minimal solution is* $abcd$. *(Note: the minimal solution does not imply that it is the best solution).*

Another method of determining missing attributes is to use probabilities. Probabilities are used to determine the value of a missing attribute by determining how probable a value is. A baseline probability is defined as any value occurring more than X-times. If this occurs, then the value is accepted and a don't know can represent all values in the range when the baseline probability is zero or it is possible that no values are in the range. This methodology can be performed by using a preprocessing algorithm that determines (probabilistically) what the don't know values should be changed too. The following example illustrates the concept of using probabilities for determining the value of a don't

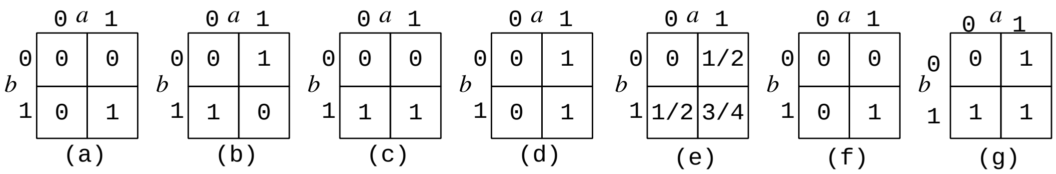


Figure 8.2: Using probability to find a function from cubes $a-$ and $-b$.

know.

Example 8.2. *Figures 8.2a through 8.2d show the possible combinations when given the cubes: $a-$ and $-b$. Because there are four combinations, then the probability of finding any of the Karnaugh maps in Figures 8.2a-d, is $\frac{1}{4}$. Figure 8.2e shows the probabilities of finding an output value with a value of one. By assuming the probability baseline in the interval [0,1], every minterm is calculated to be true or false. Setting the probability baseline to be greater than 0.5 causes that only the minterm ab is true, resulting in the Figure 8.2f. If the probability baseline is greater than 0.4, the function is shown in Figure 8.2g.*

One can appreciate that this topic is of great importance to the machine learning community and the author is unaware of any published papers that fully explore the above considerations. In [122], the authors use relations as an attempt at this problem, but the relations are only used to identify overlapping cubes. Consequently, relations do not distinguish between don't cares and don't knows. But, relations can represent machine learning functions, where different output values exist for the same set of input values. Example 8.3 illustrates the concept of a relation (this is also covered in Section 3.3, where the concept of relation decision diagrams are discussed).

Example 8.3. *Given the cubes: $a-$ and $-\overline{b}$, the relation recognizes that the two cubes overlap at $a\overline{b}$. For instance, if $a-$ has the output value 0, and $-\overline{b}$ has the output value 2 then the cube $a\overline{b}$ has the output value of 0 or 2. The use of relations only preserves the information that the output of the minterm $a\overline{b}$ has*

$b \backslash a$	0	1
0	2	0,2
1	2	0

f

Figure 8.3: Missing attributes using relations; two values for the minterm $a\overline{b}$.

two different values. This is shown in Figure 8.3.

For logic synthesis methods to be used on machine learning problems, missing attributes/input don't knows need to be represented in the data structure. A possible representation is to create a new cofactor for each variable that contains don't know information. This new representation was proposed by the author in [48].

Theorem 8.1. *Given a Q_f-valued function $f(x_0, x_1, \ldots, x_{n-1})$, each variable x_i has value Q_i, so that f may be denoted as $f : Q_0 \times Q_1 \times \ldots \times Q_{n-1} \to Q_f$. To represent an input don't know, each variable x_i has value $Q_i + 1$, where the $(Q_i + 1)^{th}$ value represents a don't know. The function f may now be denoted as $f : (Q_0 + 1) \times (Q_1 + 1) \times \ldots \times (Q_{n-1} + 1) \to Q_f$. This is called the* **Don't Know** *(DK) representation.*

The Theorem is proved by the following:

Proof. $f : Q_0 \times Q_1 \times \ldots \times Q_{n-1} \to Q_f \subset f : (Q_0+1) \times (Q_1+1) \times \ldots \times (Q_{n-1}+1) \to Q_f$, thus, the original function is still represented by the DK representation of the function.

□

For the partition $A|B$, a **DK partition matrix** is defined as the $(Q_0 + 1) \times (Q_1 + 1) \times \ldots \times (Q_{n-1} + 1)$ functional values of f, arranged in $|Q_A|$ rows, and

$|Q_B|$ columns. An example of a DK partition matrix is shown in Figure 8.4a, where the partition $\{a\}|\{b\}$ is shown. In the original function, a and b are two valued variables, but to represent input don't knows the added cofactor '-' is used.

After creating the DK representation of f, and forcing the input don't knows in the function to values, it is necessary to transform the DK representation of f back to f's original functional representation.

Theorem 8.2. *Given that f is denoted as $f : (Q_0+1) \times (Q_1+1) \times \ldots \times (Q_{n-1}+1) \rightarrow Q_f$ then it is possible to transform f back to $f : Q_0 \times Q_1 \times \ldots \times Q_{n-1} \rightarrow Q_f$, iff each $Q_i + 1$ value for variable x_i is not defined.*

Proof. If each $Q_i + 1$ value is not defined, then all input don't knows in the function are set to a given value. The addition of the $Q_i + 1$ value of each variable is not needed and can be removed. This has no effect on f because the $Q_i + 1$ value was not originally defined in the original function.

□

Example 8.4. *Given the cubes: $a-$ and $-b$, the Karnaugh map for $f(a, b)$ is shown in Figure 8.4a. The '-' for an input represents a don't know. The cube $a-$ represents either minterm $a\overline{b}$ or ab, this is shown in the Karnaugh map by circling all the values in column with $a = 1$. The cube $b-$ is shown in the Karnaugh map by circling all values in the row when $b = 1$. Because the cubes $a-$ and $-b$ intersect on the Karnaugh map at ab, one possible solution is to set the don't knows of the cubes $a-$ and $-b$ to result in the minterm ab, as shown in Figure 8.4b (the input don't knows have been removed from the Karnaugh map shown in Figure 8.4a). In Figure 8.4b, f is denoted as $f : 2 \times 2 \rightarrow \{0, 1\}$.*

Given the cubes $\overline{a}b$ and $--$, the Karnaugh map for f is shown in Figure 8.4c. The dotted line is used in the Karnaugh map to illustrate that the cube $--$ represents all values in f. Because, $\overline{a}b \cap -- = \overline{a}b$, a possible solution is $\overline{a}b$ as shown in Figure 8.4d.

Given the cubes $a-$, $-b$ and $\overline{a}b$, the Karnaugh map for f is shown in Figure 8.4e. Two possible solutions are shown in Figures 8.4f,g.

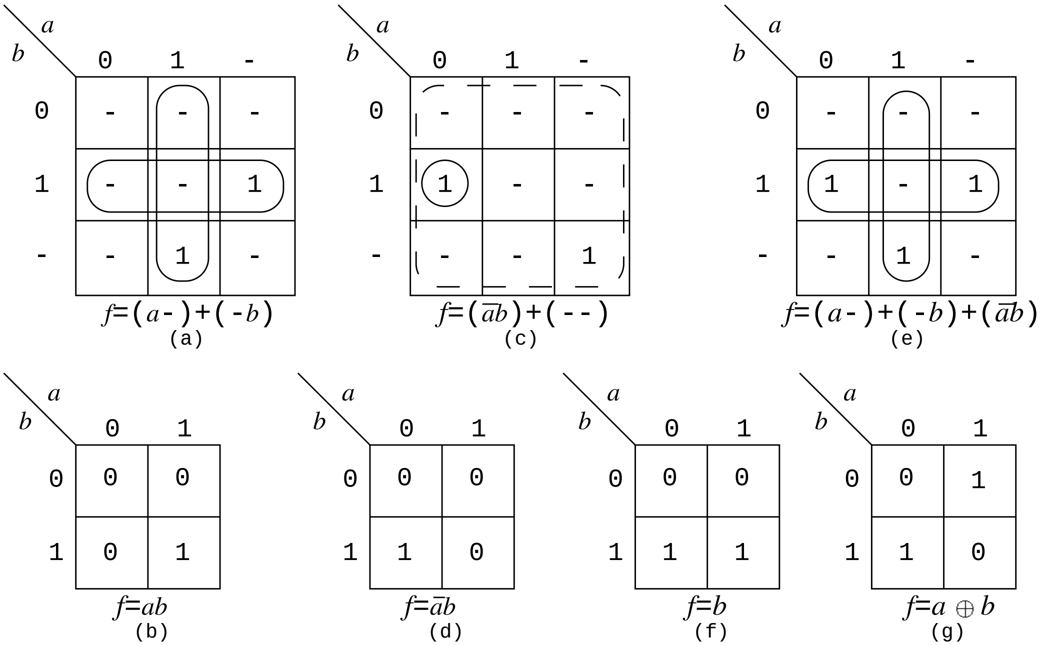


Figure 8.4: New representation to represent missing attributes.

It is also possible to extend the DK representation to MDDs, where each non-terminal node in the decision diagram has an extra cofactor. The extra cofactor defines the input don't know of each variable in the function. This is called the **DK MDD** representation. Example 8.5 illustrates the DK MDD representation.

Example 8.5. *Given the cubes shown in Figure 8.5a, the BDD representation of these cubes using don't cares is shown in Figure 8.5b. Note that anytime an output value is not given by one of the input cubes, the output is thought to be a don't care. Using the DK MDD representation of holding don't knows is shown in Figure 8.5c. The diagram in this case is much different than that of Figure 8.5b. The best choice is to map the don't know values onto care values (not the don't care output values). Two mappings are possible by doing this. The first is by taking the don't know* a_0b_- *and mapping it onto the known value* a_0b_1*. The second is by mapping* a_1b_- *to* a_1b_0*. These two mappings produce the DK*

MDD structure shown in Figure 8.5d.

Figure 8.5d illustrates the case of a don't know whose value cannot be mapped to a care term. In fact, a_b_0 cannot be mapped onto a_1b_0 because the output values are different. But, by a process of elimination, the only output value that the don't know (a_b_0) can be mapped to is a_0b_0. This produces the BDD (not reduced) structure shown in Figure 8.5e.

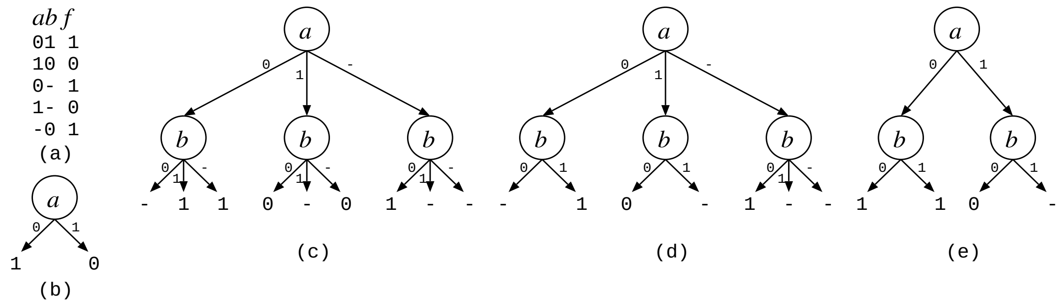


Figure 8.5: MDDs with input don't knows.

Functional decomposition may also be done using this representation of don't knows. This decomposition is called the **Don't Know Decomposition** (DKD).

Given a n-input, single-output, multi-valued function f, don't knows are represented in f by adding an extra cofactor to each variable in f. This creates a DKD partition with two types of columns and rows: a non-don't know, which does not represent a don't know for an input variable; a don't know, which does represent a don't know for a given input variable.

Definition 8.5. *Two columns in a DKD partition are* **compatible** *if:*

1. *for every non-don't know row, the output value of both columns are equal, or at least one of the output values is a don't know,*

2. *any associated don't knows represented by either column must intersect with any of the non-don't know represented row output value-sets of the other column.*

The column multiplicity of a partition matrix is identified when compatibilities between non-don't know represented columns are present. Columns that represent don't knows are not used for finding column multiplicities, they are only used for representation purposes.

b \ a	0	1	2	-
0	0	-	1	1
1	0	0	-	-
-	-	1	0	-
	0	1	1	

Figure 8.6: Don't know compatible columns.

Example 8.6. *Definition 8.5 is illustrated graphically in Figure 8.6. Column "0" and column "1" are compatible for every non-don't know row, but the don't know represented in column "1" does not intersect with any of the values in column "0". This implies that column "0" and column "1" are not compatible. Column "0" and column "2" are also not compatible because the columns are not compatible in the top row. Column "1" and column "2" are compatible because for every non-don't know row the columns are compatible and the don't know represented in both columns, intersect with values in the other column. There is no reason to evaluate column "-" because it represents the don't know column, and does not effect the column multiplicity.*

Example 8.7. *Figure 8.7 illustrates Curtis' decomposition with don't cares (Figure 8.7b) and DKD (Figure 8.7c) with bound variables a and b and given the input cubes in Figure 8.7a. The decomposition process forces the don't know values shown in Figure 8.7c to create the partition matrix in Figure 8.7d. Of course,*

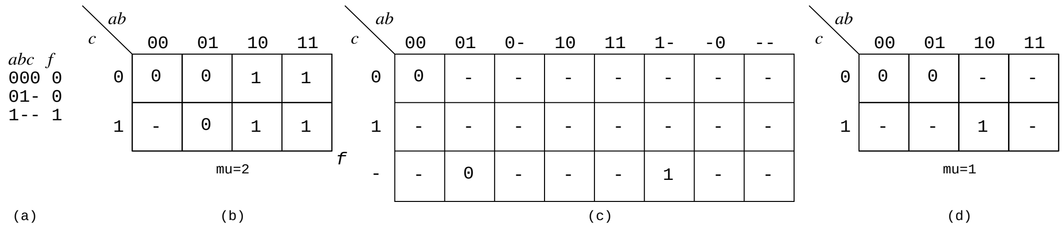


Figure 8.7: Decomposition with input don't knows (a) input cubes (b) Curtis' partition $\{c\}|\{a,b\}$ (c) DKD partition $\{c\}|\{a,b\}$ (d) after don't knows forced to values.

many different decompositions are possible, but it is important to show that the decomposition process actually leads to the setting of the don't knows.

From the example it is important to notice that using Curtis' partitions with don't cares has a larger column multiplicity ν than that of the DKD partitions. Using DKD partition matrices will result in the same or smaller decompositions than using Curtis' decomposition partitions with don't cares.

8.3.3 Retaining MVL Synthesis Techniques

Because Boolean logic synthesis techniques have been practiced since the 1950s, they are well known and fairly well understood compared to multi-valued logic synthesis. That does not imply that they are the future of computing. It also does not imply that machine learning problems should be encoded with Boolean expressions before analysis. The encoding of multi-valued expressions to Boolean expressions may destroy data either by adding or removing components inherent in the multi-valued expressions of the function or data.

Given a four-valued variable a, i.e., a represents the values $\{0, 1, 2, 3\}$. If Boolean encoding is used, two new variables are created, a_0 and a_1, derived from the following encoding:

$$\begin{array}{cc} a & \underline{0123} \\ a_0 & 0011 \\ a_1 & 0101. \end{array}$$

Given two four-valued variables, b and c, that are encoded as shown above into four new variables: b_0, b_1, c_0 and c_1. From the encoded values, say that there is a correlation of the $\{0,1\}$ attributes of b with the $\{2,3\}$ attributes of c. Because variables b and c have been encoded, the only way that the correlation between the two-variable attributes will be found is if all four encoded variables are used. Also, if b_0 and c_0 are found to have a correlation, what is actually learned from this? Because b_0 and c_0 are encoded variables they are subsets of the original function but do not contain enough information about b and c to be useful alone.

Boolean encoding can also add values that are not in the original functional representation. Given that a is a three-valued variable then a can be encoded as follows:

$$\begin{array}{cc} a & \underline{012} \\ a_0 & 001 \\ a_1 & 010. \end{array}$$

There is no instance when $a_0 = 1$ and $a_1 = 1$. In the previous chapters where encoding is used in decomposition, the case that $a_0 = 1$ and $a_1 = 1$ is a don't know/don't care case. This adds don't cares to the representation of the function which makes the function more complex.

One important aspect of machine learning that is not highly publicized is that an expert system will usually perform best by the way the data is given to it by a user. This implies that if the user has some understanding of the data being inputted into the expert system, the user may control the way the machine learning algorithms are applied to the data. When the data is encoded to Boolean expressions, the user may loose all intuition of what the data imply and how variables interact with each other.

In machine learning it is sometimes important that the algorithms find hypothesis functions are not a *black-box* concept, but knowledge-oriented, concept-learning systems [94]. It is important that the system communicates the form of the hypothesis function in an operationally effective symbolic form. This implies that the hypothesis function is expressible in a form that a human may understand and use to extract information about the function. The class of functions that include artificial neural networks and statistical methods (including AIM), create black-box representations of the hypothesis function and are not understandable. But, the result of functional decomposition methods is in the class of hypothesis functions that are highly understandable. This is another reason to use functional decomposition on machine learning problems. But, if the data has been encoded to Boolean values, it is possible that data in a form that is obvious to a human, has been changed drastically (during the encoding) and the hypothesis function may mean nothing to a human trying to evaluate it.

Each of the above discussions illustrates why it is important to leave machine learning problems in their original form, and why it is important to use multi-valued logic synthesis techniques, instead of Boolean techniques.

8.3.4 Noise in the Data

Another issue in machine learning that is different from logic synthesis is the concept of noise.

Definition 8.6. Noise *is defined as faulty values in either the attributes or the output in an example of the function.*

Noise in machine learning may originate from many different causes. For example, noise could occur in writing down some data, writing the data down in the wrong place or writing down the wrong information, etc. This can cause problems in both the machine learning algorithms and the logic synthesis algorithms. How do you detect that the example is actually incorrect and not the correct value?

There are two types of noise, noise that is in the input, and noise that is generated on the output. The distinction between input and output noise is that input noise causes the output to be associated with the wrong set of attributes, while output noise occurs when the output value is incorrect for a given attribute. Both types of noise cause different problems in determining that noise is actually in the data set. But the removal of noise is the same for both types of noise. The following is the author's solution for the removal of noise:

It is assumed that the amount of error in a data set is a percentage of the total number of data points in the set. By understanding the data, a user may be able to give the percentage of error that may exist. This percentage may then be used to determine compatible elements. Two sets are compatible if the percentage of incompatible elements (compared to compatible elements) in the sets is less than the error percentage supplied by the user. For instance, given two sets with 1000 elements in each and both sets are equal except for one pairing, then should this pairing be thought of as noise? If the user states that the amount of error in the data is 0%, then the sets are not compatible. If the percentage of allowed error is greater than $\frac{1}{1000} \times 100\%$, then the sets are compatible. The use of an allowed error percentage should be variable because there may not be any noise on some data sets, while other data sets may have a large amount of error. Of course, low thresholds may allow sets to be combined, showing that the noise exists, when in fact there are no incorrect values.

Another problem that may be encountered is what happens when the algorithm finds noisy data and what should the noisy values become? If given two values (assuming that one of the values is noise, but it is not known which value is noisy), should the resulting value become one of the values or an average of the two values, a don't know, etc. Noise is notably a very deep theoretical issue in machine learning, and needs to be explored further.

8.3.5 Continuous Data

Machine learning problems also contain continuous data or data in the form of real numbers, not natural numbers. This type of data, so far, has shown itself to be very difficult for multi-valued functional decomposition. There have been several attempts at this, though. In [72, 154], the authors map real-valued variables onto natural numbers by discretization. In [35, 132], the data is actually left in the form of real values which causes problems in the decomposition process when trying to combine two values. A way around this is to combine values that are some δ distance apart from another. For example, 2.46 and 2.55 could be combined if $\delta = 0.1$ ($|2.46 - 2.55| = 0.09 < 0.1$). Combining the two values may be done by using one of the following: the average of the two values, one of the two values, a don't know, etc. This seems to be a topic that needs further research. Given that existing tools (AIM, C4.5, neural networks) can be applied to continuous data, a possible solution is to use them as preprocessors for functional decomposition. This approach is applicable because machine learning tools are based on experience gained in years of experimentation on machine learning problems.

8.3.6 Minimal Support and Machine Learning

In creating a decomposer used for machine learning, the complexity is reduced significantly when inessential variables are removed (the minimal support problem). It is the author's opinion that the concept of minimal support is the most important concept in using logic synthesis for machine learning. If this concept is not used, the number of possible decompositions becomes astronomical because of the percentage of don't knows. Plus, it is necessary to find several minimal support solutions, even though the runtime of the algorithm increases significantly as the number of solutions requested is increased. The decomposition algorithm can evaluate which of the cliques is the best solution for the given problem. Again, evaluation is based on the smallest complexity, based on Occam's Razor and DFC.

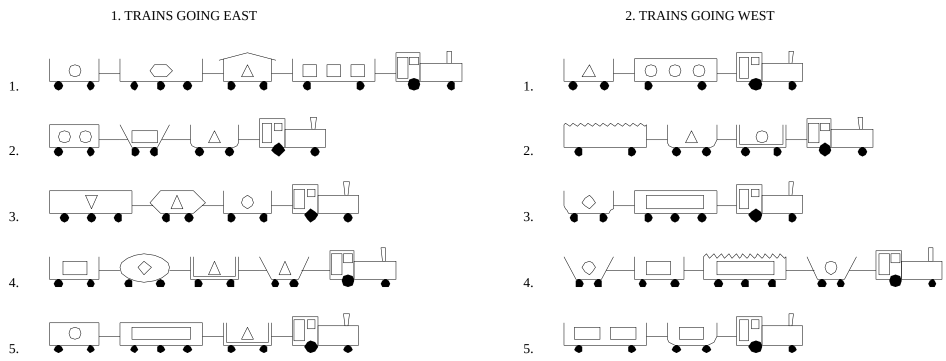


Figure 8.8: The trains problem [95].

A famous problem from the machine learning community is the trains problem, a set of ten examples and 32 variables, where each variable has up to ten attributes [95]. The trains problem is shown graphically in Figure 8.8. The goal of the problem is to find the minimal expression that defines the direction of a train (east-bound or west-bound). The best rules derived at the time the train problem was created were

1. if a train has a short, closed car, then it is an east-bound train,

2. if a train has two cars, or has a car with a jagged roof, then it is a west-bound train.

The above solutions to the train problem seem to be low in complexity, but they are not. To determine if a train has a short, closed car (the first solution above), four variables, each with ten attributes are needed. That is, the initial problem defines each car individually by its shape in the train. To determine if a train has a short, closed car, all cars in the train are checked to see if any of them have the attributes of a short, closed car. The second solution (above) requires five variables. Again, four of the variables are used to determine if a train has a car with a jagged roof. The fifth variable is used to determine the number of cars on the train.

The best rules/solutions that were given for the trains problem have a large complexity because of the complexity of the initial description of the train. It was not published how the above solutions were obtained, but the above analysis shows that the solutions are still complex in terms of the 32-variable description.

The following shows that by using a minimal support algorithm used for the removal of inessential variables (defined in Section 6.2), rules to define the direction of a train can be found with less complexity. The minimal support algorithm removes sets of inessential variables and returns the n-smallest sets of essential variables that support the function. n is a number requested by the user (this is included in the software that was written for this dissertation). For the train problem, the minimal support algorithm returned the five smallest sets of essential variables which are illustrated in Figure 8.9. The figure breaks up each of the five solutions and distinguishes what type of input results in an east-bound, west-bound or unknown output. For instance, Solution One is based on the shape of the third car in the train. Figure 8.9 shows the train cars that give an output result of an east, west or unknown direction train. Note, in Solution One, it is unlikely that the third car could be an engine, but it is still a possibility and must be included in the analysis.

The first of the five solutions is the only solution that has only one variable. The problem with Solution One, as well with Solutions Three and Four, is that there is a large set of instances when the direction is still unknown. In reducing the complexity of the hypothesis function, it is also necessary to find solutions that are completely specified. Solutions Two and Five are the only solutions that do not have unknown instances. Both of these solutions are the smallest of the set of solutions, based not only on the number of variables that represent the hypothesis function, but the number of attributes that each variable holds. Because all the other solutions contain variables that represent ten attributes, they are more complex than Solution Five, where the variables only represent two attributes. This shows that the smallest set of essential variables returned from the minimal support algorithm is not always the best solution in terms of the minimal complexity.

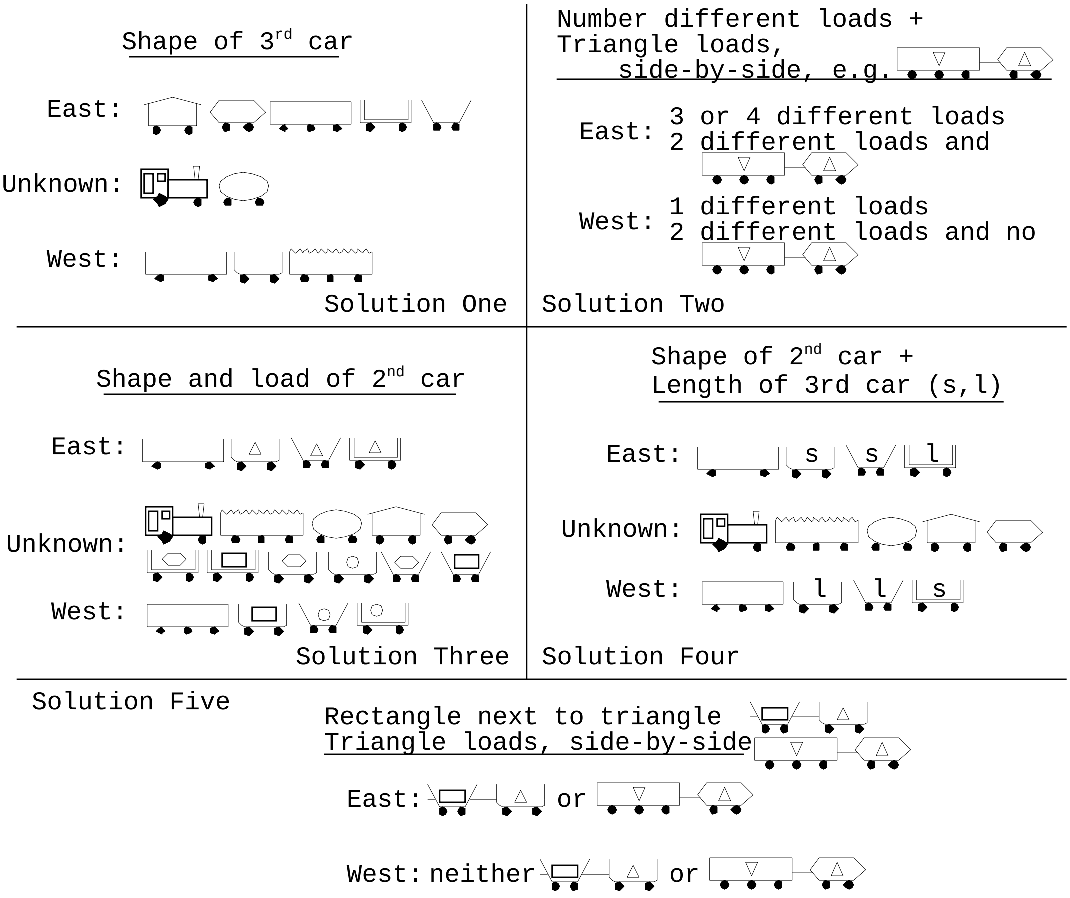


Figure 8.9: The results of the train problem by removing inessential variables.

8.4 Machine Learning Conclusions

This chapter presented the basis for using multi-valued logic synthesis in the field of machine learning. While the problems inherent in machine learning are not exactly the same as the problems found in multi-valued logic synthesis, there still exists a strong correlation between the two areas. The important aspect of this is that bridging the gap between the two areas could prove beneficial to both logic synthesis and machine learning. There are many theories to explore, especially in the context of manipulating machine learning data.

Several machine learning concepts were also presented. Each of the concepts has been described by machine learning writers for several years with no solutions. Implementing these concepts for logic synthesis is also a very difficult process. But, as shown in this section, possible solutions may be useful in the goal of finding algorithms that really "learn".

While some of the data pose certain difficulties for the logic minimization techniques, they are also difficult for the applications in machine learning. There are many areas that need to be explored on the boundary between multi-valued logic synthesis and machine learning. By exploring this boundary, it is possible that both logic synthesis and machine learning algorithms will benefit from the research.

The reason that multi-valued functional decomposition was the main focus of this chapter is due to its generalized behavior. Functional decomposition, based on heuristics, can be used to find a set of rules, or it can be used as a machine learning methodology. Again, because of the complexities inherent in functional decomposition, it is possible that other multi-valued logic synthesis techniques may demonstrate better efficiency. Because there has been little done in the comparison of multi-valued logic synthesis tools, multi-valued decomposition, and machine learning approaches, there is much more space for new research.

While there are many issues to work on in this new field of multi-valued functional decomposition for machine learning, results [51, 50] (presented in Chapter 9) are good enough to compete with the existing "top" machine learning programs.

As the size of databases grows, it is obvious that the need for machine learning is going to be necessary to process all the data. This is a tough problem, but by using multi-valued functional decomposition, hopefully, the problems can still be solved. It is difficult to predict the future, but the possibility exists that machine learning based tools may be used in circuit design as well.

Chapter 9

Experimental Results

This chapter presents the final set of experimental results. In most of the previous chapters, experimental results were presented on the algorithms and ideas presented in that chapter. This chapter presents experimental results over algorithms and ideas that are covered in more than one chapter.

The chapter is divided into the following sections. Section 9.1 compares the MDD packages, MVDD and MBDD when running the decomposition algorithms EVAL, PARTITION and *cut_level*. Section 9.2 presents a comparison of SBSD to SIS and presents the experimental results of SBSD, TBDDs and COMPLEX for VLSI layout. The last section, Section 9.3, presents the experimental results of functional decomposition applied to machine learning.

9.1 MBDD and MVDD Comparison for Functional Decomposition

Tables 9.1, 9.2 and 9.3 compare the runtimes and decision diagram sizes of the MBDD and MVDD decision diagram packages on the functional decomposition algorithms. The benchmark sets used are the Wright Patterson benchmarks [131], the MCNC benchmarks [91], and the multi-valued benchmarks [47]. The benchmarks are separated in the tables for readability and to show the difference of timing between each benchmark type. In the first two tables, the Wright Patterson benchmarks are at the top; the MCNC benchmarks are in the middle, and the multi-valued benchmarks are on the bottom. The last table, Table 9.3, the benchmarks are the multi-valued benchmark set, and is broken into the machine learning benchmarks, on the top, and the multi-valued logic benchmarks, on the bottom. The Wright Patterson benchmark functions are all 12-input, single-output, and completely specified functions. The MCNC benchmark functions are multi-input and multi-output, completely specified functions. The multi-valued benchmarks are all single-output, incompletely specified, multi-valued input and multi-valued output functions.

The units for time are in seconds. Any element in the table that begins with ">" designates that the program did not complete in the allotted time. A "-" denotes that the number of bound variables was larger than the number of input variables and a decomposition could not be performed. Again, both MDD packages were developed in the research phase of this dissertation.

Table 9.1 compares the MVDD and MBDD packages while running the EVAL algorithm (Subsection 6.1.2). The packages were run using variable bound set sizes of two, four, six and eight. For each bound set size, the algorithms were run on 50 random partitions, and the time to evaluate all 50 partitions is displayed in the table. The set of random partitions was the same for both packages. The comparison of these algorithms is the runtime to find 50 partitions; this does not include the time to determine the column multiplicity of each partition. Surprisingly, there are only a few instances where one of the packages performs significantly faster than the other (for instance the benchmark *c5b* with six bound set variables, the MVDD is 10% faster than the MBDD package). But, because SBSD only uses bound sets with two and three variables, the comparison does not conclusively state which package should be used for running the EVAL algorithm.

Table 9.2 compares the MVDD and MBDD packages on the PARTITION and *cut_level* algorithms. To compare the two algorithms, all two-variable bound set partitions of the functions were evaluated. The table shows the initial size (number of vertices of each the MDD) before starting the decomposition process. PARTITION and *cut_level* were compared by showing the time to evaluate all two-input bound set partitions and the worst-case size (number of vertices). Notice that the number of vertices is the same for the top two sections in the

filename	MBDD Multiplicity				MVDD Multiplicity			
	2	4	6	8	2	4	6	8
add0	0.1	1.0	7.0	91.0	0.1	1.0	7.0	88.0
4_ones	0.1	0.1	5.0	64.0	0.1	0.1	5.0	62.0
greater_than	0.1	1.0	8.0	84.0	0.1	1.0	8.0	86.0
interval1	0.1	0.1	0.1	2.0	0.1	0.1	0.1	2.0
interval2	0.1	0.1	2.0	13.0	0.1	0.1	2.0	13.0
majority_gate	0.1	1.0	5.0	74.0	0.1	0.1	5.0	72.0
pal	0.1	1.0	5.0	31.0	0.1	1.0	5.0	31.0
parity	0.1	0.1	3.0	46.0	0.1	0.1	3.0	44.0
remainder2	0.1	2.0	12.0	106.0	0.1	2.0	12.0	107.0
sim12	0.1	0.1	7.0	87.0	0.1	0.1	7.0	87.0
substr1	0.1	1.0	9.0	85.0	0.1	1.0	8.0	85.0
subtraction1	0.1	2.0	10.0	102.0	0.1	1.0	10.0	104.0
subtraction3	0.1	2.0	12.0	107.0	0.1	1.0	12.0	109.0
9sym	0.1	0.1	5.0	48.0	0.1	0.1	4.0	42.0
9symml	0.1	0.1	4.0	42.0	0.1	0.1	4.0	43.0
alu2	0.1	2.0	15.0	127.0	0.1	2.0	14.0	126.0
b12	0.1	2.0	10.0	134.0	0.1	1.0	9.0	134.0
clif	0.1	2.0	12.0	120.0	0.1	1.0	12.0	120.0
con1	0.1	0.1	5.0	0.1	0.1	0.1	5.0	0.1
inc	0.1	1.0	8.0	0.1	0.1	1.0	8.0	0.1
misex1	0.1	1.0	7.0	0.1	0.1	0.1	7.0	0.1
sao	0.1	2.0	9.0	71.0	0.1	2.0	8.0	70.0
sao2	0.1	2.0	8.0	71.0	0.1	1.0	8.0	72.0
spla	3.0	12.0	53.0	505.0	3.0	11.0	50.0	507.0
t481	0.1	1.0	23.0	245.0	0.1	1.0	24.0	242.0
table3	2.0	13.0	90.0	450.0	2.0	10.0	82.0	449.0
pal2	0.1	0.1	-	-	0.1	0.1	-	-
c1b	0.1	0.1	-	-	0.1	0.1	-	-
c2b	1.0	5.0	82.0	>2000	0.1	4.0	82.0	>2000
c3b	2.0	6.0	137.0	>2000	1.0	5.0	138.0	>2000
c4b	1.0	6.0	208.0	>2000	1.0	5.0	183.0	>2000
c5b	1.0	8.0	332.0	>2000	1.0	7.0	299.0	>2000
d1	0.1	0.1	-	-	0.1	0.1	-	-
d2	2.0	13.0	523.0	>2000	2.0	13.0	532.0	>2000
d3	3.0	18.0	914.0	>2000	2.0	17.0	922.0	>2000
d4	4.0	21.0	1759.0	>2000	2.0	19.0	1773.0	>2000

Table 9.1: MVDD and MBDD comparison on the EVAL algorithm.

table because these benchmark functions are all binary functions. Remember that the number of vertices is not indicative of the memory used by each package. Consequently, comparisons of the number of vertices cannot be made directly. But, the percentage of vertices initial vertices versus the number of vertices created while running the PARTITION algorithm can be made. In most cases, the percentage of MVDD nodes created during the call to PARTITION is larger than the call to the MBDD PARTITION algorithm. Of course, this is expected from Observation 3.2, where it was determined that the size of MVDD grows faster than the MBDD. The table also shows that in almost all cases, the MVDD is faster than the MBDD. In fact, for the benchmark *table3* the MVDD is 56% faster than the MBDD.

Table 9.2 compares the MVDD and MBDD packages on the multi-valued benchmark set. All tree decomposition algorithms are used in the comparison: EVAL, PARTITION, and *cut_level*. The decomposition algorithms were run to calculate all two-variable bound sets. In all cases, except for the benchmark *sleep*, the MVDD is faster than the MBDD. The surprise in the table is the benchmark *alet* where the *cut_level* algorithm has a faster runtime than the other two algorithms. This, is a very unexpected result, and as shown by the table, is not the norm.

While the results of this section show that the MVDD package has a slight performance advantage over the MBDD package, the MBDD package has been chosen as the data structure for the decomposer of this dissertation. The main reason for this decision is that the MBDD package can represent relations as a RDD (as presented in Section 3.3). This is a great factor because of the number of machine learning benchmark functions that have multiple output values for a given set of attributes (inputs).

9.2 SBSD Comparison with SIS

This section compares the SBSD decomposition algorithm with SIS [145]. In this section, to make a complete comparison, SIS and SBSD are compared based

	MBDD					MVDD				
	init.	Partition		*cut_level*		init.	Partition		*cut_level*	
filename	size	time	size	time	size	size	time	size	time	size
add0	189	4.0	327	3.0	189	189	2.0	327	2.0	189
4_ones	55	1.0	95	2.0	55	55	1.0	95	2.0	55
greater_than	195	4.0	333	3.0	195	195	2.0	333	2.0	195
interval1	34	1.0	53	2.0	34	34	0.1	53	2.0	34
interval2	52	1.0	115	2.0	75	52	1.0	115	2.0	75
majority_gate	53	1.0	91	2.0	53	53	1.0	91	2.0	53
pal	200	4.0	356	3.0	200	200	2.0	356	2.0	200
parity	34	1.0	53	2.0	34	34	1.0	53	2.0	34
remainder2	311	6.0	606	4.0	328	311	3.0	606	3.0	328
sim12	23	1.0	55	2.0	51	23	0.1	55	2.0	51
substr1	48	1.0	141	2.0	113	48	1.0	141	2.0	113
subtraction1	246	5.0	423	3.0	246	246	2.0	423	3.0	246
subtraction3	200	4.0	436	3.0	253	200	2.0	436	3.0	253
9sym	41	0.1	70	0.1	41	41	0.1	70	0.1	41
9symml	41	0.1	70	0.1	41	41	0.1	70	0.1	41
alu2	264	4.0	517	2.0	291	264	1.0	517	2.0	291
b12	100	5.0	161	12.0	115	100	5.0	161	10.0	115
clif	259	3.0	456	1.0	300	259	1.0	456	1.0	300
con1	22	0.1	36	0.1	32	22	0.1	36	0.1	32
inc	84	0.1	166	0.1	114	84	0.1	166	0.1	114
misex1	51	0.1	91	0.1	80	51	0.1	91	0.1	80
sao	164	2.0	311	1.0	190	164	1.0	311	1.0	190
sao2	164	2.0	311	1.0	190	164	1.0	311	1.0	190
t481	47	3.0	134	8.0	154	47	2.0	134	7.0	154
table3	951	33.0	2075	19.0	2297	951	14.0	2075	27.0	2297
pal2	26	0.1	42	0.1	26	26	0.1	42	0.1	26
c1b	21	0.1	32	0.1	21	21	0.1	32	0.1	21
c2b	179	3.0	324	4.0	185	100	1.0	182	2.0	106
c3b	247	7.0	452	11.0	254	133	3.0	245	5.0	139
c4b	254	7.0	466	11.0	258	134	3.0	247	5.0	138
c5b	245	6.0	445	9.0	250	121	3.0	231	4.0	128
d1	45	0.1	75	0.1	46	31	0.1	51	0.1	32
d2	316	5.0	609	5.0	363	152	2.0	291	2.0	188
d3	464	13.0	893	15.0	509	217	6.0	419	7.0	251
d4	509	14.0	966	15.0	519	233	7.0	447	8.0	251

Table 9.2: MVDD and MBDD comparison on the PARTITION and *cut_level* algorithms.

filename	MVDD EVAL time	MVDD Partition time	MVDD *cut_level* time	MBDD EVAL time	MBDD Partition time	MBDD *cut_level* time
audiology	>2000	>2000	>2000	>2000	>2000	>2000
breastc	142.0	92.0	28.0	147.0	100.0	42.0
chess1	47.0	11.0	10.0	43.0	20.0	20.0
chess2	644.0	768.0	>2000	780.0	1808.0	1929.0
cloud	10.0	2.0	0.1	13.0	4.0	2.0
connect-4	>2000	>2000	>2000	>2000	>2000	>2000
flag	451.0	586.0	402.0	696.0	952.0	680.0
house-votes	1.0	6.0	11.0	1.0	11.0	11.0
letter	>2000	>2000	>2000	>2000	>2000	>2000
lung-cancer	523.0	936.0	>2000	824.0	1765.0	>2000
mushroom	89.0	88.0	51.0	136.0	151.0	129.0
sensory	37.0	15.0	15.0	41.0	27.0	18.0
sleep	66.0	9.0	3.0	78.0	16.0	9.0
sponge	362.0	618.0	1003.0	558.0	1122.0	1798.0
tic-tac-toe	8.0	3.0	3.0	10.0	6.0	5.0
trains	28.0	39.0	133.0	47.0	86.0	250.0
zoo	4.0	5.0	11.0	5.0	10.0	13.0
alet	>2000	>2000	1603.0	>2000	>2000	1105.0
c6a	4.0	4.0	4.0	6.0	8.0	14.0
d4	6.0	7.0	7.0	8.0	13.0	16.0
d5	6.0	6.0	5.0	7.0	11.0	13.0
d6	15.0	18.0	6.0	24.0	21.0	20.0
d7	49.0	80.0	64.0	71.0	131.0	167.0
d8	101.0	189.0	193.0	154.0	312.0	485.0
d9	117.0	224.0	243.0	179.0	369.0	607.0
d10	143.0	282.0	338.0	218.0	464.0	890.0
u1	>2000	>2000	>2000	>2000	>2000	>2000
u2	>2000	>2000	>2000	>2000	>2000	>2000
u3	>2000	>2000	>2000	>2000	>2000	>2000
u1_2	24.0	118.0	1390.0	33.0	161.0	1436.0
u1_3	41.0	147.0	1435.0	62.0	227.0	>2000
u1_4	59.0	174.0	1483.0	90.0	286.0	>2000
u1_5	78.0	206.0	1556.0	121.0	350.0	>2000
u1_10	212.0	428.0	1736.0	348.0	771.0	>2000

Table 9.3: MVDD and MBDD comparison of the EVAL, PARTITION and *cut_level* algorithms for finding all two-variable bound sets.

on DFC and the number of two-input gates. The problem with comparing SIS with SBSD is that of the functional output. SBSD cannot be forced to create two-input (composite functions) blocks, because some blocks are not decomposable. And, it is not a fair analysis to compare SIS using DFC. The problem is that SIS will find a functional representation of a non-decomposable block, while SBSD will not.

Even though DFC does not favor SIS, DFC can still be used to show the complexity of the resulting functions from both SIS and SBSD. The benchmark functions used in this analysis are the Wright Patterson benchmark set with up to twelve input variables and a single-output. The DFC comparisons of SIS and SBSD are presented in Tables 9.4 and 9.5. SIS can be used as a multi-level decomposer using different scripts, or methods to decompose a function. The different scripts that are included in the SIS package are Algebraic, Boolean, Delay, MCNC, MSU, OCT, and Rugged. While running the scripts on the test functions, the MSU and OCT scripts had the same results, so the OCT column was deleted from the tables.

The results of Tables 9.4 and 9.5 show that while DFC does not favor the output functions of SIS, generally there is one SIS script that has about the same DFC of SBSD. In fact, for the benchmark *mux8*, SIS (script Algebraic) has a smaller DFC than SBSD. But there are benchmark functions like *4_ones, interval1* and *interval2*, where the SBSD has a much smaller DFC value than SIS. Again, the complexity of the resulting SIS functions must be very high to get DFC values that are so much larger than SBSD.

To construct a comparison based on the number of two-input gates, two things must be done. First, the libraries of SIS must be restricted to use only two-input gates. This removes the scripts MSU, OCT and MCNC, because these scripts are based on specified libraries and cannot be restricted to two-input gates. Second, the non-decomposable blocks created by SBSD must be reduced to two-input gates, which is done using SIS. The comparisons of SBSD and SIS on two-input gates is shown in Tables 9.6 and 9.7. The result show that in most instances SBSD has a smaller gate count than SIS. But, as shown in Chapter 6,

Function	Algebraic	Boolean	Delay	MCNC	MSU	Rugged	SBSD
add0	180	256	70	56	60	84	28
add2	44	24	78	44	40	32	20
and_or_chain8	28	256	36	28	28	256	28
ch22f0	20	64	64	52	132	64	20
ch30f0	88	144	106	84	128	256	40
ch47f0	88	208	164	116	144	132	56
ch52f4	256	256	256	212	236	256	156
ch70f3	144	256	78	68	68	256	44
ch74f1	256	256	162	128	168	256	84
ch83f2	256	256	256	220	256	256	120
ch8f0	84	64	78	64	40	64	40
4_ones	256	256	256	212	244	256	76
greater_than	180	256	64	64	64	84	28
interval1	256	256	256	212	256	256	88
interval2	256	256	256	188	232	256	76
kdd2	20	32	26	16	20	32	16
kdd3	12	16	12	12	12	16	12
kdd5	56	256	76	48	48	256	48
kdd6	16	16	16	16	16	16	12
kdd7	256	256	36	36	256	256	28
kdd9	20	24	48	32	20	24	20
kdd10	24	24	48	20	20	20	20
majority_gate	256	256	256	188	164	256	76
monkish1	16	16	28	24	24	16	12
monkish2	100	256	164	160	180	256	60
monkish3	68	32	36	28	28	32	20
mux8	**24**	64	54	48	48	64	32
or_and_chain8	28	72	34	28	28	256	28
pal	28	48	92	32	32	72	28
rnd_m1	256	256	44	36	36	256	28
rnd_m10	256	256	208	200	256	256	108
rnd_m25	256	256	256	256	256	256	180
rnd_m5	256	256	228	120	148	256	72
rnd_m50	256	256	256	256	256	256	256
substr1	256	256	128	72	72	256	72
substr2	256	256	90	72	72	256	60
subtraction1	256	256	256	148	140	256	68

Table 9.4: DFC comparison of SIS and SBSD on eight-variable functions.

Function	Algebraic	Boolean	Delay	MCNC	MSU	Rugged	SBSD
add0	220	296	296	84	100	100	44
add2	84	40	40	92	88	288	36
add4	44	24	24	44	40	32	22
4_ones	NF	NF	1024	452	448	1024	172
greater_than	220	296	296	104	104	100	44
interval1	1024	1024	1024	104	104	1024	88
interval2	1024	1024	1024	296	376	1024	180
majority_gate	NF	NF	1024	464	716	1024	172
pal	48	96	96	48	48	272	44
parity	NF	NF	76	48	48	84	44
sim12	NF	NF	192	56	56	192	44
substr1	736	1024	1024	152	240	1024	168
subtraction1	1024	1024	1024	292	292	1024	120
subtraction3	1024	1024	1024	248	252	1024	112

Table 9.5: DFC comparison of SIS and SBSD on twelve-variable functions.

SBSD has problems with the randomly generated benchmark functions. (The instances when SBSD has a larger gate count than SIS are in bold.) A surprise in the results is that SIS is not able to find the minimal representation of the benchmark function *parity* (for either 8- or 12-inputs). The *parity* benchmark is a set of XOR gates, used to determine the parity of the inputs, which would seem to make it a fairly simple function to find the minimal result.

Table 9.8 presents the results of logic synthesis applied to VLSI layout. To construct the table, SBSD, COMPLEX and TBDDs are used to create SCCGs (Static CMOS Complex Gates). To obtain results while using SIS as a preprocessor, TBDDs are used to force the blocks into SCCGs. TBDDs must be used to make a fair comparison because SIS and SBSD + COMPLEX do not produce minimized SCCGs. Thus, TBDDs are used to convert the resulting functional outputs to SCCGs. The table is broken into the following columns: Cells - the number of SCCGs generated, Signals - number of input signals to all cells (including primary inputs), Trans - number of transistors, Nets - number of silicon connection points, Width - width of the realized layout, Height - height of the realized layout, Area - area of the realized layout. The height, width, and area

filename	Algebraic	Boolean	Delay	Rugged	SBSD
add0	21	21	21	21	21
add2	21	17	23	20	17
add4	9	9	12	9	9
4_ones	NA	NA	144	95	79
greater_than	23	23	21	21	21
interval1	23	23	23	23	23
interval2	55	55	64	61	54
majority_gate	NA	NA	135	98	80
pal	11	11	11	11	11
parity	NA	NA	19	21	12
sim12	11	11	11	11	11
substr1	32	33	33	33	**58**
subtraction1	113	87	177	68	44
subtraction3	79	55	109	58	43

Table 9.6: Number of gates comparison of SIS and SBSD on twelve-variable functions.

values are all generated by TROPIC for .25 micron.

Table 9.8 shows that in most cases the number of SCCGs generated by SBSD + COMPLEX is smaller than SIS. The only instance where SIS is better than SBSD + COMPLEX is on the benchmark *substr1*. It is not clear why SIS does so much better on this benchmark than SBSD + COMPLEX, but it might be due to the fact that SIS results in a smaller two-input gate count than SBSD on this function. Overall, the results shown that SBSD + COMPLEX result in smaller layout sizes than using SIS for the logic synthesis phase of VLSI layout.

9.3 Machine Learning Results

This section compares SBSD, C4.5, a program from Wright State University (WSU) and a functional decomposer written at the University of Ljubljana (HINT). PERL scripts were written to do conversions of the file formats for each of the Machine learning programs.

The training sets are made from randomly selecting care terms from a given

Function	Algebraic	Boolean	Delay	Rugged	SBSD
add0	15	15	17	13	13
add2	9	9	12	11	9
and_or_chain8	7	7	7	7	7
ch22f0	11	11	13	11	7
ch30f0	18	19	18	18	15
ch47f0	24	24	27	24	19
ch52f4	52	52	59	47	**55**
ch70f3	16	16	14	14	14
ch74f1	29	30	29	27	**37**
ch83f2	51	53	55	49	45
ch8f0	12	9	13	9	**13**
4_ones	46	46	83	45	38
greater_than	15	15	13	13	13
interval1	45	58	57	45	23
interval2	43	55	76	39	30
kdd2	4	4	4	4	4
kdd3	3	3	3	3	3
kdd5	9	11	11	11	10
kdd6	3	3	3	3	3
kdd7	7	7	7	7	7
kdd9	5	5	10	5	5
kdd10	5	5	5	5	5
majority_gate	54	51	67	35	33
monkish1	5	5	3	5	3
mux8	9	9	9	9	10
or_and_chain8	7	7	7	7	7
pal	7	7	7	7	7
parity	13	13	11	13	7
rnd_m1	7	7	7	7	7
rnd_m10	37	38	42	42	**41**
rnd_m25	62	69	82	78	**82**
rnd_m5	23	23	25	23	**30**
substr1	15	15	15	15	**20**
substr2	15	15	15	15	13
subtraction1	38	52	83	26	25

Table 9.7: Number of gates comparison of SIS and SBSD on eight-variable functions.

filename	Cells	Signals	Trans	Nets	Width (μm)	Height (μm)	Area (mm^2)
SBSD/Complex							
add0	4	18	50	28	23	39	0.00093
add2	4	18	50	28	23	39	0.00093
add4	4	18	50	28	23	39	0.00093
4_ones	**49**	**63**	292	93	60	64	0.00390
greater_than	14	28	70	38	22	56	0.00127
interval1	**10**	**24**	**66**	25	**17**	58	0.00102
interval2	**28**	**42**	**178**	**64**	**38**	65	0.00250
majority_gate	**52**	**66**	**268**	**93**	**57**	**57**	**0.00325**
pal	11	25	74	32	24	**54**	0.00130
parity	**11**	**25**	**74**	**32**	**24**	54	**0.00130**
sim12	2	**16**	26	21	17	24	0.00042
substr1	36	**50**	186	74	41	**57**	**0.00235**
subtraction1	27	**43**	**138**	**56**	**32**	**54**	**0.00174**
subtraction3	25	**43**	**132**	**55**	**33**	**54**	**0.00183**
SIS							
add0	4	18	50	28	23	39	0.00093
add2	4	18	50	28	23	39	0.00093
add4	4	18	50	28	23	39	0.00093
4_ones	55	69	**268**	**92**	**58**	**56**	**0.00330**
greater_than	14	28	70	38	22	**56**	0.00127
interval1	14	28	74	25	18	**50**	**0.00093**
interval2	40	54	196	74	42	56	**0.00241**
majority_gate	58	72	316	102	67	63	0.00429
pal	11	25	74	32	24	54	0.00130
parity	22	36	110	47	25	**52**	0.00133
sim12	2	16	26	21	17	24	0.00042
substr1	**14**	28	94	**47**	**36**	65	0.00240
subtraction1	42	56	220	87	53	66	0.00354
subtraction3	41	55	198	77	51	65	0.00336

Table 9.8: Complex gate analysis comparison with SIS and SBSD as logic synthesis preprocessors.

function. The selection is done in steps of 8 in the range [8,248]. At each step, 20 different data sets were found by randomly selecting P care terms, where P is equal to the current step. The reason for selecting 20 different data sets is to get a good distribution so the output could be fully analyzed.

The test functions that were used come from the Wright Patterson's Pattern Theory Group set of benchmarks [131]. There are several reasons for using these functions: the functions are completely specified which makes the comparison of error exact, the smallest DFC of the functions are known, so analysis of the decomposers heuristics can be performed, and many training samples may be run. Of course, this is just a measure to see how well a function can represent data which is highly unspecified. But, notice that the minimum number of random examples evaluated is 8, implying 248 don't knows, which means that the function is 96.875% incompletely specified.

The experimental results are shown in the Figures 9.1, 9.2 and 9.3. For each function the following are shown in the figures: the *error* of the training output to the original KDD function, the DFC that was found on the training function, and the time that it took for the algorithm to find a solution. The title of each graph designates what is actually being displayed on the $Y - axis$ (*error*, DFC, Time), while the $X - axis$ is the number of training samples. For each function, there is some error when the number of training samples is quite small, but as the number of training samples gets larger, the amount of error goes to zero. Also note that the algorithm usually finds the correct function once it finds the DFC of the original function. That is, as the number of training samples grows the algorithm is able to find the original function, and the DFC of the original function. The last thing to note is the time to a solution; it is quite interesting that the time to find a solution was much longer for small training samples. The reason for this is the number of possibilities that exist with a large number of *don't knows*.

Each of the Figures 9.1, 9.2 and 9.3 was selected out of the 54 possible functions in the benchmark set. These three functions, in general, show the execution of the SBSD decomposer. The first two functions are small examples

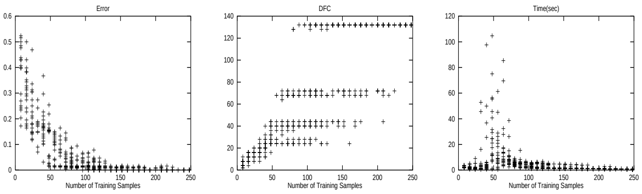


Figure 9.1: Finding the error, DFC, and time of the decomposer on the benchmark *kdd5*.

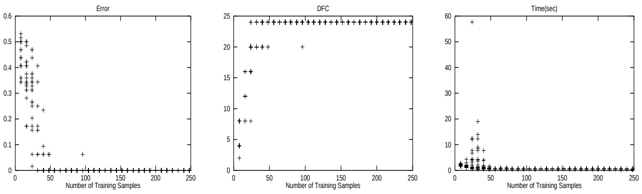


Figure 9.2: Finding the error, DFC, and time of the decomposer on the benchmark *kdd10*.

of Knowledge Discovery in Databases, while the third is a character recognition function.

A comparison of the programs SBSD, HINT, C4.5, and WSU, on the benchmark function *kdd*7 function is shown in Figure 9.5. Each program is compared on three categories, error rate, DFC and run-time. Note, that the run-time for C4.5 and HINT could not be calculated correctly. As can be seen from the figure, the error rates for SBSD, WSU, and HINT are all about the same, with HINT being the best.

Figure 9.4 shows the error rate found for the programs when averaged over all 54 Wright Patterson Functions. The figure shows that SBSD is the best overall in terms of error. But, it does have some difficulty when the number of samples is small. Again, notice that HINT and WSU both perform better than C4.5 overall.

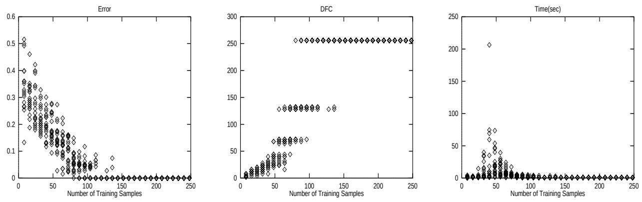


Figure 9.3: Finding the error, DFC, and time of the decomposer on the benchmark *ch52f4*.

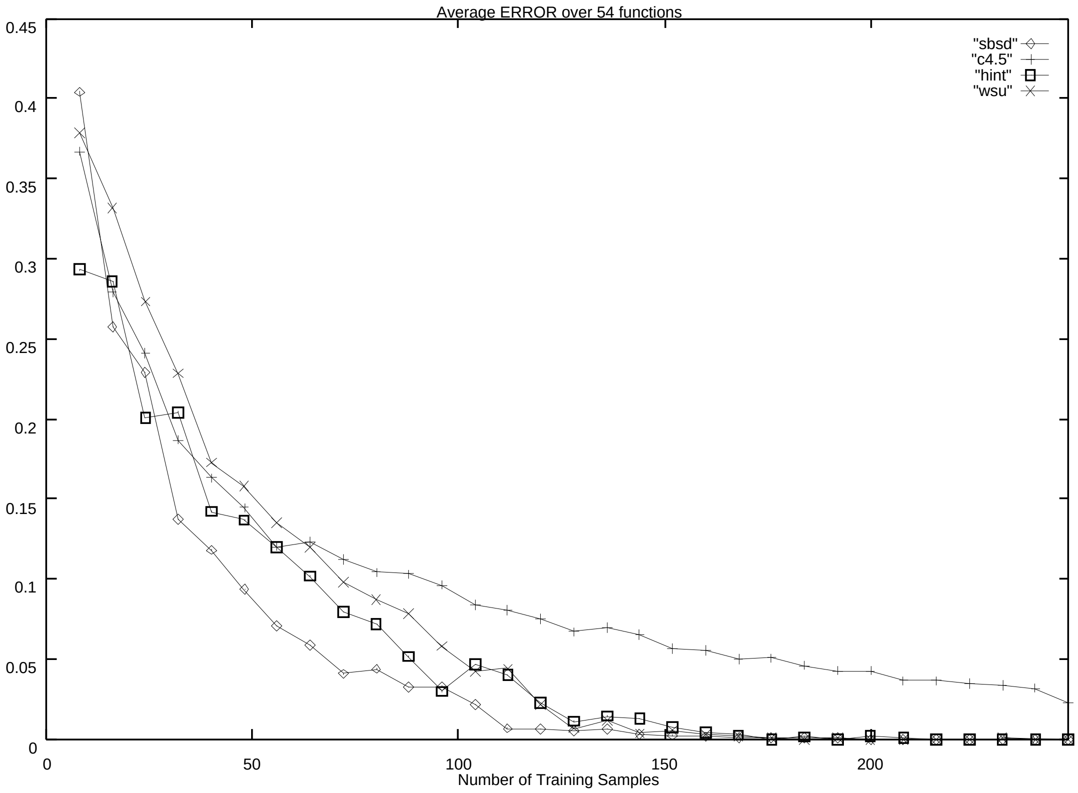


Figure 9.4: The average error over all 54 benchmark functions.

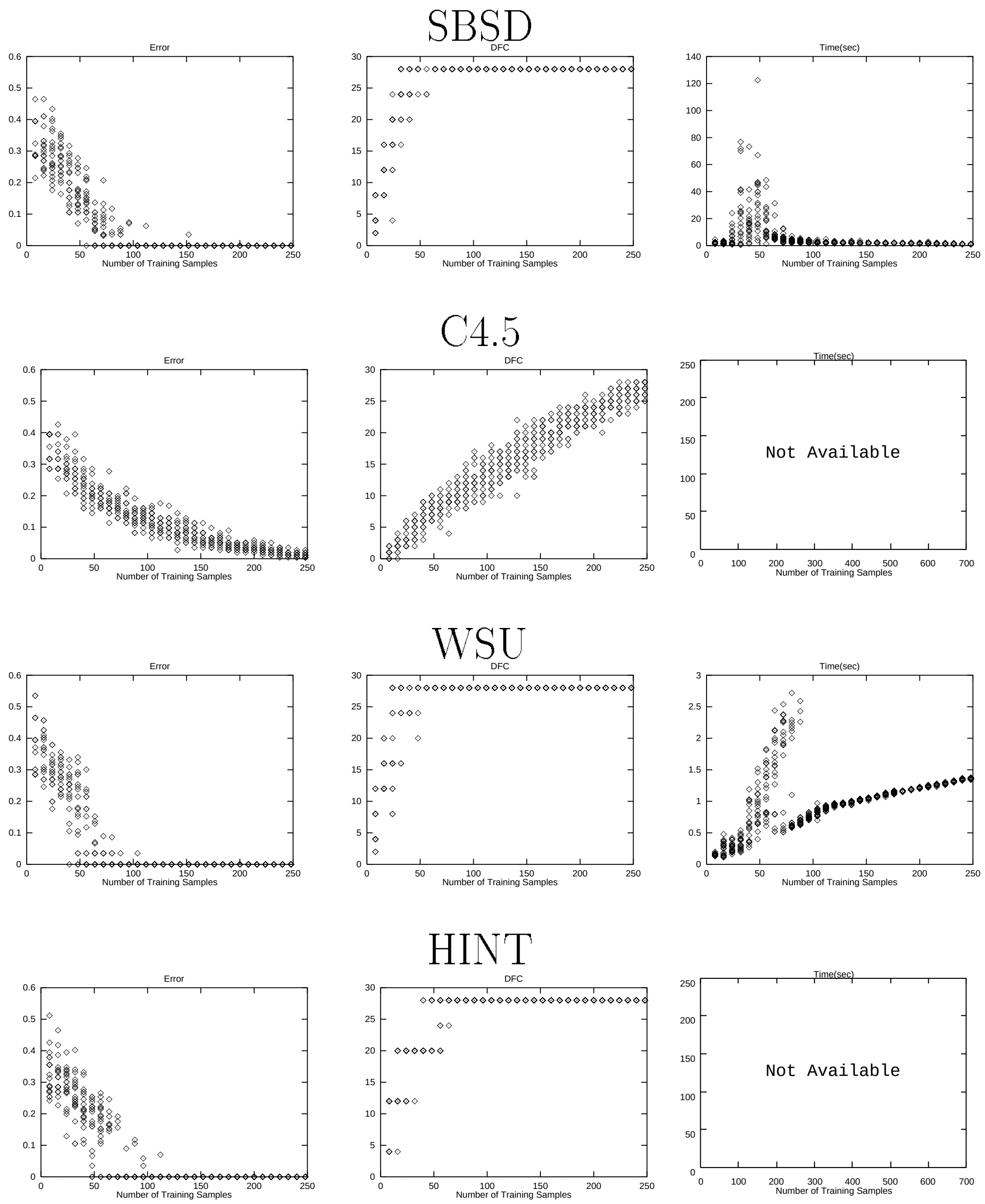


Figure 9.5: Finding the error, DFC, and time of each program on the function KDD7 (Time for C4.5 and HINT are not given) and DFC for C4.5 is actually the number of vertices in the C4.5 decision tree.

Chapter 10

Conclusion

This dissertation proposed the use of functional decomposition for VLSI layout synthesis and for machine learning. A new functional decomposition methodology was proposed which is based on using small bound set partitions and the use of multi-valued labeling instead of using encoding during the decomposition process. The new decomposition methodology reduces the algorithmic complexities of functional decomposition partition selection. The maximum size of each bound set is restricted to three variables, which reduces the partition selection complexity from $O(3^n)$ to $O(n^3)$. An advantage of using only small bound sets is that the number of columns in a partition is also reduced. By reducing the number of columns in a partition, determining column multiplicity (a known NP-hard problem) can be determined using exact methods.

In the past, it has been shown that functional decomposition can be used as a machine learning problem solver. But, because most logic synthesis problems are highly specified, applying logic synthesis, i.e., functional decomposition to machine learning problems can be a very daunting task. The proposed functional decomposition methodology applied to machine learning uses only small bound sets. By using probabilities, it was shown that by reducing the size of the bound set sizes, a functional decomposition algorithm is more likely to create a hypothesis function with little error than by using partitions with medium or large bound set sizes.

While machine learning problems have been solved using functional decomposition in the past, this is the first time that functional decomposition has been proposed as a synthesis method used for VLSI layout. The proposed synthesis method uses functional decomposition as the primary synthesis method. Once a function has been decomposed, small logical blocks are composed into larger SCCG gates by using TBDDs, large non-decomposable blocks are synthesized using dissected pairs. As far as the author knows, this is the first time that dissected pairs has been used for the synthesis of VLSI layout.

Also discussed in this dissertation is the use a MDD as the data structure for functional decomposition. Advantages of using a MDD as a data structure include: the ability to represent very large functions, a strong canonical form from which many logic synthesis algorithms may be derived, and problems from many domains can be expressed entirely in terms of a MDD. This dissertation has shown that MDDs can represent very large functions, including the benchmark *letter* which has 16 inputs, each with 16 values, and the output is 26-valued. Decomposition algorithms based on MDDs were also derived which can improve the speed performance of functional decomposers.

10.1 Experimental Result Conclusions

While many concepts have been proposed in this dissertation, the concepts have been experimentally evaluated throughout the dissertation.

Comparisons were run on two different types of MDDs. While the experimental results were inconclusive as to which package has the best performance, the MBDD package is the data structure used in the remainder of the dissertation. The MBDD package is used because it can be based on any BDD package (many exists) and because the MBDD package can easily represent relations.

Two minimal support algorithms were also presented here. Comparisons of these two algorithms show that much work is still needed before these methods can be used for finding the minimal support of very large multi-valued functions. The two algorithms were compared to the algorithms created by Jozwiak. This is not quite a fair comparison because Jozwiak methods are based on cubes and not decision diagrams, which were used to make the comparisons. But, because

Jozwiak has not released his software and has not made his benchmark functions available, this was the only way to make a comparison to his research. Overall, the two minimal support algorithms presented here form a basis for future work.

Functional decomposition concepts and algorithms were also presented in this dissertation. The algorithms EVAL and PARTITION were compared to the LPV and *cut_level* algorithms that were created by Lai, Pedram and Vrudhula [78]. The EVAL and PARTITION algorithms were shown to have faster runtimes than the Lai *et al.* algorithms, and they do not lead to diagrams that have exponential sizes.

A decomposition methodology of using post-decomposition encoding and small bound sets was also discussed. The new methodology, called SBSD, was compared to FLASH [134] and to SIS [145]. Results show that for most functions SBSD finds the same DFC as FLASH, and in some instances finds smaller values than FLASH. The exception were randomly generated functions, where FLASH performed better than SBSD. But, because randomly generated functions are not used for logic synthesis or machine learning, this has no effect on the application of SBSD. When SBSD was compared to SIS, it was found that generally SBSD has a smaller DFC than SIS and also has a smaller number of two-input gates. From this, it is possible that SBSD will have the same performance on many different benchmarks.

A logic synthesis method was also presented based on negative-gate logic. The algorithm, called COMPLEX, was compared to SIS for generating SCCGs (static CMOS complex gates). The experimental results show that this approach can find a smaller number of SCCGs and transistors (when compared to SIS). Further experimental results confirmed this when SBSD was used as a preprocessor for COMPLEX.

SBSD was also compared to C4.5 [126], HINT [154] and WSU on machine learning problems. The overall experimental results show that SBSD results in smaller error than the other algorithms, once the number of samples is increased. Overall, this a very promising result that can be used for future decomposers in their many possible applications.

10.2 Contributions

This dissertation contributes to the related research areas as follows:

- an analysis of decision diagram algorithms for multi-valued functions,
- a methodology for Boolean logic synthesis,
- a methodology for encoding in logic synthesis,
- a methodology for multi-valued logic synthesis,
- a set of algorithms for logic synthesis and functional decomposition,
- a decomposition methodology applied to circuit synthesis,
- a synthesis method applied to library free technology mapping,
- a concept learning paradigm in machine learning,
- a method to construct new features that are not predefined in constructive induction,
- a method to find decision models from data in decision support,
- a supervised or unsupervised model that decomposes data into manageable subproblems that characterize the data in knowledge discovery and data mining,

The **original** contributions of this dissertation include:

- an analysis and comparison of MBDDs and MVDDs on truly multi-valued functions,
- redefining Curtis' definitions of decomposable partitions to allow for composed blocks with the same number of inputs as outputs,
- for the first time multi-valued labeling is used instead of using Boolean encoding for the decomposition of Boolean circuits. The use of multi-valued labeling has the following advantages:

  - the decomposition process is not constrained to the Boolean encoding that was found,
  - Curtis' decompositions are Ashenhurst's decompositions,
  - the post-decomposition encoding used to retain Boolean values for circuit synthesis can go through many possible encoding possibilities, while existing decomposers are generally restricted to just one Boolean encoding,

- decomposition based on small bound set partitions, which gives the following advantages:
  - the number of partitions evaluated is reduced to $O(3^n)$ from $O(n^3)$,
  - small bound sets are more likely to result in hypothesis functions with small error,
  - determining column multiplicity can be completed using exact methods,
  - the number of possible encodings (post-decomposition encodings) possibilities is greatly reduced,

- using a new definition of a decomposable partition, finding new decomposition types with two variables in the bound set,
- analysis and contribution of minimal support algorithms,
- application of decomposition to VLSI layout,
- use of dissected pairs in library free technology mapping,
- application of decomposition to lattice diagrams,
- multi-valued decision diagram structure with relations,
- decomposition partitions and multi-valued decision diagrams that can represent functions with input don't knows.

10.3 Future Work

The goal of this research was to present a functional decomposition methodology that could be applied to machine learning and VLSI layout synthesis. The methodology was based on research of finding algorithms that could reduce the complexity of the partition selection problem. The result of the research was a functional decomposition methodology that is based on small bound sets. The experimental results revealed that a decomposition methodology solely based on small bound sets is not practical for all benchmark functions. But, the small bound set decomposition types proposed in this dissertation could be applied to bound sets of any size. Future work in this area for logic synthesis would be to use these decomposition types for larger bound set sizes. Of course, heuristics will have to be used to help find partitions that can lead to a good decomposition.

Several new methodologies were presented in this dissertation, but experimental results were not conducted. Future work would be to experientially verify each of the following methodologies and their usefulness to their own paradigms:

- small bound set decompositions applied to lattice diagrams,
- decomposition partitions and multi-valued decision diagrams based on don't knows

Another focus would be the construction of a decomposer that is capable of performing all Curtis, Steinbach and Perkowski decomposition types using many different heuristic methods that have been published. This would allow comparisons that could illustrate the instances when each heuristic/decomposition type should be used in the decomposition process.

Plus, the algorithm SBSD was only implemented for single-output functions. In the future, this should be extended to implement multi-output functions. Part of the reason that this was not implemented in this dissertation was because it is unclear how to represent the multi-valued outputs. Should the outputs be converted to a multi-valued output? This would fit the framework for which

SBSD has been created, but the author is not sure if the concept would work efficiently.

Last, evaluation of the results presented in this dissertation to determine the cause of inconsistent outputs for some benchmark functions. This evaluation could lead to further experimentation and possibly a new decomposition/machine learning/layout synthesis tool that would have better performance comparing with all existing tools.

Bibliography

[1] P. Abouzeid, R. Leveugle, G. Saucier, and R. Jamier, "Logic synthesis for automatic layout," *Proc. of EURO-ASIC*, pp. 146–151, 1992.

[2] Y. Abu-Mostafa, *Complexity in Information Theory.* New York: Springer-Verlag, 1988.

[3] S. Abugharbieh and S. Lee, "A fast algorithm for the disjunctive decomposition of m-valued functions, part i: The decomposition algorithm," *International Symposium on Multi-Valued Logic*, pp. 118–125, 1993.

[4] S. Abugharbieh and S. Lee, "A fast algorithm for the disjunctive decomposition of m-valued functions, part ii: Time complexity analysis," *International Symposium on Multi-Valued Logic*, pp. 126–131, 1993.

[5] S. Akers, "A rectangular logic array," *IEEE Transactions on Computers*, vol. C-21, pp. 848–857, 1972.

[6] S. Akers, "Binary decision diagrams," *IEEE Transactions on Computers*, vol. 6, pp. 509–516, June 1978.

[7] M. Anthony and N. Biggs, *Computational Learning Theory.* New York, NY: Cambridge University Press, 1992.

[8] C. Apte, S. J. Hong, S. Prasad, and B. Rosen, "Ramp: Rules abstraction for modeling prediction," *Technical report*, 1995.

[9] R. L. Ashenhurst, "The decomposition of switching functions," *Bell Laboratories' Report*, vol. 1, pp. II–1–II–37, 1953.

[10] R. L. Ashenhurst, "The decomposition of switching functions," *Bell Laboratories' Report*, vol. 16, pp. III–1–III–72, 1956.

[11] R. L. Ashenhurst, "The decomposition of switching functions," *International Symposium on the Theory of Switching Functions*, pp. 74–116, 1959.

[12] A. R. Barron and R. L. Barron, "Statistical learning networks: A unifying view," *Symposium on the Interface: Statistics and Computing Science*, p. 12, 1988.

[13] B. Becker and R. Drechsler, "Efficient graph based representation of multi-valued functions with an application to genetic algorithms," *International Symposium on Multi-Valued Logic*, pp. 65–72, 1994.

[14] M. Berkelaar and J. Jess, "Technology mapping for standard-cell generators," *Proc. of ICCAD*, pp. 470–473, 1988.

[15] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth, "Occam's razor," *Information Processing Letters*, pp. 377–80, 1987.

[16] D. Bochmann, "Zerlegung von blockschaltungen," *Messen Steuern Regeln*, vol. 11, no. 3, 1968.

[17] E. Boros, V. Gurvich, P. Hammer, T. Ibaraki, and A. Kogan, "Decomposition of partially defined Boolean functions," *DIMACS Technical Report*, vol. 9, 1994.

[18] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa, "VIS: A system for verification and synthesis," in *8th International Conference on Computed Aided Verification* (R. Alur and T. Henzinger, eds.), (New Brunswick, NJ), pp. 428–432, 1996.

[19] R. Brayton, P. McGeer, J. Sanghavi, and A. Sangiovanni-Vincentelli, "A new exact minimizer for two-level logic synthesis," in *Logic Synthesis and Optimization* (T. Sasao, ed.), pp. 1–31, Kluwer Academic, 1993.

[20] M. Breuer, ed., *Design Automation of Digital Systems*, vol. 1. Englewood Cliffs, New Jersey: Prentice Hall, 1972.

[21] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Transactions on Computer*, vol. 8, pp. 677–691, August 1986.

[22] R. E. Bryant, "Symbolic Boolean manipulation with ordered binary-decision diagrams," *ACM Computing Surveys*, vol. 24, no. 3, pp. 293–318, 1992.

[23] M. Burns, "New approaches to column compatibility checking and column-based input/output encoding for Curtis decomposition of completely or incompletely specified switching functions," Master's thesis, Portland State University, 1997.

[24] M. Burns, M. Perkowski, and L. Jozwiak, "An efficient approach to decomposition of multi-output Boolean functions with large sets of bound variables," *Proceedings of Euromicro Conference*, pp. 18–29, 1998.

[25] E. K. C. Blake and C. Merz, "UCI repository of machine learning databases." http://www.ics.uci.edu/~mlearn/MLRepository.html, 1998.

[26] K. Chakrabarti, A. Choudhury, and M. Basu, "Complementary function approach to the synthesis of three-level NAND network," *IEEE Transactions on Computers*, vol. C-19, pp. 509–514, 1970.

[27] K. Chen and M. Fujita, "Network optimization using don't cares and Boolean relations," in *Logic Synthesis and Optimization* (T. Sasao, ed.), Kluwer Academic Publishers, 1993.

[28] A. Choudhury, K. Chakrabarti, and D. Sharma, "Studies on the problem of three-level NAND synthesis," *International Journal on Control*, vol. 6, no. 6, pp. 547–572, 1967.

[29] N. Christofides, *Graph Theory.* New York, NY: Academic Press, 1975.

[30] E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao, "Multi-terminal binary decision diagrams: An efficient data structure for matrix representation," *International Workshop on Logic Synthesis*, pp. P6a:1–15, 1993.

[31] H. Curtis, "Generalized tree circuit," *ACM*, pp. 484–496, 1963.

[32] H. Curtis, "Generalized tree circuit-the basic building block of an extended decomposition theory," *ACM*, vol. 10, pp. 562–581, 1963.

[33] H. A. Curtis, *A new approach to the design of switching circuits.* Princeton, NJ: Van Nostrand, 1962.

[34] S. Das and C. Sheng, "On detecting total or partial symmetry of switching functions," *IEEE Transactions on Computers*, vol. C-20, no. 3, pp. 352–355, 1971.

[35] J. Demsar, B. Zupan, M. Bohanec, and I. Bratko, "Constructive induction by decomposition of real functions," in *Proc. of ECML-97* (G. W. M. Someren, ed.), pp. 94–107, 1997.

[36] E. Detjens, G. Gannot, R. Rudell, A. Sangiovanni-Vinccentelli, and A.Wang, "Technology mapping in MIS," *Proc. of ICCAD*, pp. 116–119, 1987.

[37] D. Dietmeyer, *Logic Design of Digital Systems.* Boston, MA: Allyn & Bacon, 1971.

[38] R. Drechsler, "Verification of multi-valued logic networks," *International Symposium on Multi-Valued Logic*, pp. 10–15, 1996.

[39] B. Drucker, C. Files, and M. Perkowski, "Polarized pseudo-kronecker symmetry with an application to the synthesis of lattice decision diagrams," *Proc. ICCIMA*, pp. 745–755, 1998.

[40] E. Dubrova, *Boolean and Multiple-Valued Functions in Combinational Logic Synthesis*. PhD thesis, University of Victoria, Victoria British Colombia, Canada, 1997.

[41] E. Dubrova and Stengel, "Finding composition trees for multiple-valued functions," *International Symposium on Multi-Valued Logic*, pp. 19–26, 1997.

[42] C. Edwards and S. Hurst, "A digital synthesis procedure under function symmetries and mapping methods," *IEEE Transactions on Computers*, vol. c-27, no. 11, pp. 985–997, 1978.

[43] G. Epstein, *Multiple-valued Logic Design: An Introduction*. Bristol: Institute of Physics Publishing, 1993.

[44] C. Files, "Using a search heuristic in an np-complete problem in ashenhurst-curtis decomposition," tech. rep., Air Force Office of Scientific Research, Wright Laboratory, Wright Air Force Base, Dayton Ohio, 1994.

[45] C. Files, "A new approach to partition selection in Ashenhurst-Curtis functional decomposition," Master's thesis, University of Idaho, 1995.

[46] C. Files, "Report on the realization of dissected pairs," tech. rep., LIRMM, University Montpellier II, Montpellier France, 1996.

[47] C. Files, "POrtland Logic Optimization (POLO) group Boolean and multi-valued benchmark functions." http://www.ee.pdx.edu/polo/functions/, 1999.

[48] C. Files, "New functional representation for the decomposition of machine learning problems," *Third Oregon Symposium on Logic, Design and Learning*, 2000.

[49] C. Files, R. Drechsler, and M. Perkowski, "Functional decomposition of MVL functions using a multi-valued decision diagram," *International Symposium on Multi-Valued Logic*, pp. 27–32, 1997.

[50] C. Files and M. Perkowski, "An error reducing approach to machine learning using multi-valued functional decomposition," *International Symposium on Multi-Valued Logic*, pp. 167–172, 1998.

[51] C. Files and M. Perkowski, "Multi-valued functional decomposition as a machine learning method," *International Symposium on Multi-Valued Logic*, pp. 173–178, 1998.

[52] C. Files and M. Perkowski, "New multi-valued functional decomposition algorithms based on MDDs," *IEEE Transactions on Computer Aided Design*, September 2000.

[53] C. Files, A. Reis, M. Perkowski, M. Robert, and D. Auvergne, "Exact minimization of networks with complex gates using terminal suppressed binary decision diagrams and dissected pairs," *Workshop on Post-Binary Ultra-Large Scale Integration*, 1996.

[54] C. Files, A. Reis, M. Perkowski, M. Robert, and D. Auvergne, "Minimization of networks with complex gates," *Workshop Boolesche Probleme*, 1996.

[55] S. Fortune, J. Hopcroft, and E. Schmidt, "The complexity of equivalence and containment for free single variable schemes.," in *Lecture notes in computer science* (G. Goos, J. Hartmanis, Ausiello, and Boehm, eds.), (Berlin, Germany), pp. 227–240, Springer-Verlag, 1978.

[56] A. Friedman and P. R. Menon, *Theory and Design of Switching Circuits*. Woodland Hills: Computer Science Press, Inc., 1975.

[57] J. A. Goldman and M. L. Axtell, "On using logic synthesis for supervised classification learning," *Tools with AI conference*, pp. 198–205, 1995.

[58] M. Gondran and M. Minoux, *Graphs and Algorithms*. New York, NY: John Wiley & Sons, 1984.

[59] S. Grygiel, *Decomposition of relations as a new approach to constructive induction in machine learning and data mining*. PhD thesis, Portland State University, Portland Oregon, USA, 1999.

[60] O. Heil, "British patent 439457," 1935.

[61] S. Hight, "Complex disjunctive decomposition of incompletely specified Boolean functions," *Transactions on Computers*, pp. 103–110, 1973.

[62] S. J. Hong, "RMINI: A heuristic algorithm for generating minimal rules from examples," *Pacific Rim International Conference on Systematic Methods of Scientific Discovery*, 1994.

[63] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[64] S. L. Hurst, "Detection of symmetries in combinatorial functions by spectral means," *IEE Journal on Elect. Circuits and Systems*, vol. 1, pp. 173–80, 1977.

[65] T. Ibaraki, "Gate interconnection minimization of switching networks using negative gates," *IEEE Transactions on Computers*, vol. C-20, pp. 698–706, 1971.

[66] T. Ibaraki and S. Muroga, "Synthesis of a network with minimum number of negative gates," *IEEE Transactions on Computers*, vol. C-20, pp. 49–58, 1971.

[67] L. Jozwiak, "Information relationships and measures in application to logic design," *International Symposium on Multi-Valued Logic*, pp. 228–235, 1999.

[68] L. Jozwiak and N. Ederveen, "Genetic algorithm for input support minimization," *International Conference on Computational Intelligence and Multimedia Applications*, pp. 765–772, 1998.

[69] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, *Synthesis of Finite State Machines: Functional Optimization.* Norwell, Massachusetts: Kluwer Academic Publishers, 1997.

[70] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, "Multi-valued decision diagrams: Theory and applications," *Multi-Valued Logic Journal*, vol. 4, pp. 9–62, 1998.

[71] K. Keutzer, "DAGON: Technology binding and local optimization by DAG matching," *Proc. of Design Automation Conference*, pp. 341–347, 1987.

[72] I. Kononenko, E. Simec, and M. R. Sikonja, "Experiments in automatic learning of medical diagnostic rules," tech. rep., Jozef Stefan Institute, Ljubljana, 1996.

[73] Z. Kulpa, "Synthesis of quasi-minimal logic circuits of many variables with use of NAND and NOR gates," Master's thesis, Institute of Automatic Control, Warsaw Technical University, 1970.

[74] A. V. Kuznetsov, "O bespovorotnykh kontaktnykh skhemakh i bespovorotnykh superpozitsiyakh funktsii algebry logiki," *Akademiya Nauk SSSR*, pp. 185–225, 1958.

[75] E. R. Laboratory, *OCTOOLS-5.2 Users guide.* University of California at Berkeley, May 1993.

[76] Y. T. Lai, K. R. Pan, and M. Pedram, "OBDD-based functional decomposition: Algorithms and implementation," *IEEE Transactions on Computer Aided Design*, vol. 15, no. 8, pp. 977–990, 1996.

[77] Y. T. Lai, M. Pedram, and S. B. K. Vrudhula, "BDD-based logic decomposition," tech. rep., Dept. of EE. Systems, University of Southern California, 1992.

[78] Y. T. Lai, M. Pedram, and S. B. K. Vrudhula, "BDD based decomposition of logic functions with application to FPGA synthesis," *Design Automation Conference*, pp. 642–647, 1993.

[79] C. Y. Lee, "Binary decision programs," *Bell System Technical Journal*, vol. 4, pp. 985–999, July 1959.

[80] G. G. Lendaris and G. Stanley, "On the structure-dependent properties of adaptive logic networks," tech. rep., GM Defense Research Laboratories, Santa Barbara, California, July 1963.

[81] J. Lilienfeld, "Current control - u.s. patent 1745175," 1925.

[82] B. Lin, "Efficient symbolic support manipulation," *Proc. of ICCD*, pp. 513–516, 1993.

[83] T. Liu, "Synthesis algorithms for 2-level MOS networks," *IEEE Transactions on Computers*, vol. C-24, pp. 72–79, 1975.

[84] T. Liu, "Synthesis of feed-forward MOS networks with cells of similar complexities," *IEEE Transactions on Computers*, vol. C-26, pp. 826–831, 1977.

[85] T. Liu, "Synthesis of multilevel feed-forward MOS networks," *IEEE Transactions on Computers*, vol. C-26, pp. 581–588, 1977.

[86] T. Luba, "Decomposition of multiple-valued functions," *International Symposium on Multi-Valued Logic*, pp. 256–261, 1995.

[87] T. Luba and R. Lasocki, "Decomposition of multiple-valued Boolean functions," *Applied Math and Computer Science*, vol. 4, no. 1, pp. 125–138, 1994.

[88] F. Mailhot and G. DeMicheli, "Algorithms for technology mapping based on binary decision diagrams and on Boolean operations," *IEEE Transactions on Computer Aided Design*, vol. 12, no. 5, pp. 599–620, 1993.

[89] R. Maziasz and J. Hayes, *Layout minimization of CMOS cells.* Boston, Massachusetts: Kluwer Academic Publishers, 1992.

[90] E. McCluskey, "Detection of group invariance of total symmetry of a boolean function," *Bell System Technical Journal*, vol. 35, pp. 1445–1453, 1956.

[91] MCNC, "Benchmark functions." ftp://mcnc.mcnc.org/, 1991.

[92] R. Michalski, "Discovering classification rules using variable-valued logic system vl1," in *Third International Conference on Artificial Intelligence*, (Stanford University), pp. 162–172, 1973.

[93] R. Michalski, "A theory and methodology for inductive learning," in *Machine Learning: An Artificial Intelligence Approach* (R. Michalski, J. Carbonell, and T. Mitchell, eds.), pp. 83–134, 1983.

[94] R. Michalski, I. Bratko, and M. Kubat, *Machine Learning and Data Mining.* New York, NY: Wiley, 1998.

[95] D. Michie, S. Muggleton, D. Page, and A. Srinivasan, "To the international computing community: A new east-west challenge," 1994. Oxford University Computing Laboratory, Oxford, UK.

[96] D. Miller, *Decomposition in many valued design.* PhD thesis, University of Manotoba, May 1976.

[97] D. Miller, "Spectral symmetry tests," *International Symposium on Multi-Valued Logic*, pp. 130–134, 1981.

[98] D. Miller, "Multiple-valued logic design tools," *International Symposium on Multi-Valued Logic*, pp. 2–11, 1993.

[99] D. Miller and R. Drechsler, "Implementing a multiple-valued decision diagram package," *International Symposium on Multi-Valued Logic*, pp. 52–57, 1998.

[100] G. J. Montgomery and K. C. Drake, "Abductive networks," *SPIE, Applications of Neural Networks Conference*, 1990.

[101] F. Moraes, *Synthese topologique de macro-cellules en technologie CMOS.* PhD thesis, LIRMM, University Montpellier II, Montpellier, France, 1994.

[102] F. Moraes, N. Azemard, M. Robert, and D. Auvergne, "Flexible macro-cell layout generator," *Proc. of ACM/SIGDA Physical Design Workshop*, pp. 105–116, 1993.

[103] F. Moraes, M. Robert, D. Auvergne, and R. Reis, "An efficient layout synthesis tool for CMOS random logic circuits," *Proc. of IX SUBMICRO*, pp. 48–57, 1994.

[104] F. Moraes, M. Robert, D. Auvergne, and R. Reis, "Towards optimal use of CMOS complex gates in automatic layout synthesis," *Proc. of X SUBMICRO*, pp. 11–20, 1995.

[105] S. Muggleton and L. D. Raedt, "Inductive logic programming: Theory and methods," *Journal of Logic Programming*, vol. 16, no. 20, pp. 629–679, 1989.

[106] A. Mukhopadhyay, "Detection of total or partial symmetry of a switching function with the use of decomposition charts," *IEEE Transactions of Electical Computers*, vol. ec-12, no. 5, pp. 553–557, 1963.

[107] R. Murgai and R. Brayton, "Using encoding for functional decomposition with application to look up table architectures," *2nd International ACM/SIGDA Workshop on FPGAs*, 1994.

[108] R. Murgai, Y. Hishizaki, N. Shenoy, R. Brayton, and A. Sangiovanni-Vincentelli, "Logic synthesis for programmable gate arrays," *Design Automation Conference*, pp. 620–625, 1990.

[109] R. Murgai, N. Shenoy, R. Brayton, and A. Sangiovanni-Vincentelli, "Improved logic synthesis algorithm for table look up architectures," *IEEE International Conference on Computer Aided Design*, pp. 564–567, 1991.

[110] S. Muroga, "Section 4.2: Logic design of MOS networks," *VLSI System Design: When and How to Design Very-Large-Scale Integrated Circuits*, pp. 144–54, 1982.

[111] S. Muroga, *VLSI System Design: When and How to Design Very-Large-Scale Integrated Circuits.* New York, New York: John Wiley and Sons, 1982.

[112] S. Muroga, "Logic synthesizers, the transduction method and its extension," in *Logic Synthesis and Optimization* (T. Sasao, ed.), Kluwer Academic Publishers, 1993.

[113] S. Muroga and H. C. Lai, "Minimization of logic networks under generalized cost function," *IEEE Transactions on Computers*, vol. C-25, pp. 893–906, 1976.

[114] J. C. Muzio and T. C. Wesselkamper, *Multiple-valued switching theory.* Boston, MA: Adam Hilger, 1986.

[115] K. Nakamura, "Synthesis of gate-minimum multi-output two-level negative gate networks," *IEEE Transactions on Computers*, vol. C-28, no. 10, pp. 768–772, 1979.

[116] K. Nakamura, N. Tokura, and T. Kasami, "Minimal negative gate networks," *IEEE Transactions on Computers*, vol. C-21, pp. 5–11, 1972.

[117] M. Perkowski and J. Brown, "A unified approach to designs with multiplexers and to the decomposition of Boolean functions," *Proceedings ASEE Annual Conference*, pp. 1610–1619, 1988.

[118] M. Perkowski, M. Burns, R. Almeria, and N. Iliev, "Approaches to the input-output encoding problem in Boolean decomposition," tech. rep., Portland State University, Portland Oregon, 1995.

[119] M. Perkowski, C. Files, and L. Jozwiak, "Minimization of sum of continuous-input and/or implicant expressions for data mining applications," *Workshop Boolesche Probleme*, 1998.

[120] M. Perkowski, T. Luba, S. Grygiel, P. Burkey, M. Burns, N. Iliev, M. Kolsteren, R. Lisanke, R. Malvi, Z. Wang, H. Wu, F. Yang, S. Zhou, and J. Zhang, "Unified approach to functional decompositions of switching functions," tech. rep., Portland State University, Portland Oregon, 1995.

[121] M. Perkowski, R. Malvi, S. Grygiel, M. Burns, and A. Mishchenko, "Graph coloring algorithms for fast evaluation of Curtis decompositions," *Design Automation Conference*, pp. 225–230, 1999.

[122] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Z. Wang, and J. S. Zhang, "Decomposition of multiple-valued relations," *International Symposium on Multi-Valued Logic*, pp. 13–18, 1997.

[123] M. Perkowski, T. Ross, D. Gadd, J. Goldman, and N. Song, "Application of ESOP minimization in machine learning and knowledge discovery," *International Workshop on Applications of the Reed-Muller Expansion in Circuit Design*, pp. 102–109, 1995.

[124] G. N. Povarov, "O funktsional 'noi razdelimosti bulevykh funktsii," *Doklady Akademii Nauk SSSR*, vol. 94, pp. 801–803, 1954.

[125] J. Quinlan, "Learning efficient classification procedures and their application to chess end games," in *Machine Learning 1: an Artificial Intelligence Approach* (R. Michalski, J. Carbonell, and T. Mitchell, eds.), (Redwood City, CA), Morgan Kaufmann, 1983.

[126] J. R. Quinlan, *C4.5: Programs for Machine Learning.* Palo Alto, California: Morgan Kaufmann, 1993.

[127] R. Read, *Graph Theory and Computing.* New York, NY: Academic Press, 1972.

[128] A. Reis, M. Robert, D. Auvergne, and R. Reis, "Associating CMOS transistors with BDD arcs for technology mapping," *IEE Electronics Letters*, vol. 31, no. 14, pp. 1118–1120, 1995.

[129] A. Reis, M. Robert, D. Auvergne, and R. Reis, "Extensive using of CMOS complex gates with terminal suppressed BDDs," *Journal of the Brazilian Computer Society*, vol. 2, no. 2, pp. 63–75, 1995.

[130] T. Ross, M. Axtell, M. Noviskey, and D. Gadd, "Pattern theory paradigm for system design," *Midwest Symposium on Circuits and Systems*, pp. 721–724, 1993.

[131] T. Ross, J. Goldman, D. Gadd, M. Noviskey, and M. Axtell, "Pattern theory benchmark functions." http://www.mbvlab.wpafb.af.mil/t̃ross/pt/, 1993.

[132] T. Ross, J. Goldman, D. Gadd, M. Noviskey, and M. Axtell, "On the decomposition of real-valued functions," *Third International Workshop on Post-Binary ULSI Systems*, 1994.

[133] T. Ross, M. Noviskey, D. Gadd, and J. Goldman, "Pattern theoretic feature extraction and constructive induction," *ML-COLT Workshop on Constructive Induction and Change of Representation*, 1994.

[134] T. Ross, M. Noviskey, R. Taylor, and D. Gadd, "The learning and synthesis hotbed," Tech. Rep. WL/AART-2, Wright Laboratory, WPAFB, OH, 1994.

[135] J. P. Roth and R. M. Karp, "Minimization over Boolean graphs," *IBM Journal of Research and Development*, pp. 227–38, 1962.

[136] R. Rudell, "Dynamic variable ordering for ordered binary decision diagrams," *Proc. IEEE/ACM ICCAD*, pp. 43–47, 1993.

[137] A. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal Res. Development*, vol. 3, pp. 221–229, 1959.

[138] A. Samuel, "Some studies in machine learning using the game of checkers II: recent progress," *IBM Journal Res. Development*, vol. 11, pp. 601–617, 1967.

[139] T. Sasao, "Multiple-valued decomposition of generalized Boolean functions and the complexity of programmable logic arrays," *Complexity of Programmable Logic Arrays*, vol. 30, no. 9, pp. 635–643, 1981.

[140] T. Sasao, "Functional decomposition of PLAs," *Proceedings of the International Workshop on Synthesis Logic*, pp. 59–65, 1987.

[141] T. Sasao, "Totally undecomposable functions: Applications to efficient multi-valued decompositions," *International Symposium on Multi-Valued Logic*, 1999.

[142] H. Sawada, S. Yamashita, and A. Nagoya, "Restricted simple disjunctive decompositions based on grouping symmetric variables," *Proceedings of Great Lakes Symposium on VLSI*, pp. 39–44, 1997.

[143] H. Sawada, S. Yamashita, and A. Nagoya, "Restructuring logic representations with easily detectable simple disjunctive decompositions," *Proceedings of Design, Automation and Test in Europe*, pp. 755–759, 1998.

[144] W. L. Semon, "Characteristic numbers and their use in the decomposition of switching functions," *Association of Computer Machinery*, pp. 273–280, 1952.

[145] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephan, R. Brayton, and A. Sangiovanni-Vincentelli, "SIS: a system for sequential circuit synthesis," tech. rep., Dept. of Electrical Engineering and Computer Science, University of California, Berkeley, 1992.

[146] C. Shannon, "The synthesis of two-terminal switching circuits," *Bell System Technical Journal*, vol. 28, pp. 59–98, Jan 1949.

[147] C. Sheng, "Detection of totally symmetric boolean functions," *IEEE Transactions on Computers*, vol. C-20, no. 3, pp. 352–355, 1965.

[148] A. Srinivasan, T. Kam, S. Malik, and R. Brayton, "Algorithms for discrete function manipulation," *International Conference on CAD*, pp. 92–95, 1990.

[149] B. Steinbach and M. Stockert, "Design of fully testable circuits by functional decomposition & implicit test pattern generation," *IEEE VLSI Test Symposium Proceedings*, pp. 22–27, 1994.

[150] C. Tsai and M. Marek-Sadowska, "Detecting symmetric variables in Boolean functions using generalized Reed-Muller forms," *Proceedings ISCAS*, pp. 287–290, 1994.

[151] C. Tsai and M. Marek-Sadowska, "Generalized Reed-Muller forms as a tool to detect symmetries," *IEEE Transactions on Computers*, vol. 45, no. 1, pp. 33–40, 1996.

[152] W. Wan and M. Perkowski, "A new approach to the decomposition of incompletely specified multi-output functions based on graph coloring and local transformations and its application to FPGA mapping," *IEEE European Design Automation Conference*, pp. 230–235, 1992.

[153] N. Weste and K. Eshraghian, *Principles of CMOS VLSI Design*. Reading, Massachusetts: Addison-Wesley, 1993.

[154] B. Zupan, *Machine Learning Based on Functional Decomposition*. PhD thesis, University of Ljubljana, Slovenia, 1997.

[155] B. Zupan, M. Bohanec, I. Bratko, and J. Demsar, "Machine learning by function decomposition," in *Proc. of ICML-97* (D. Fisher, ed.), pp. 421–429, 1997.

Appendix A

Benchmark Review

A.1 Wright Lab Benchmark Functions

Wright Lab Pattern Theory Group Benchmark, eight-variable input function set: this set of functions is intended to be representative of a wide variety of functions for testing machine learning systems. All functions are of the form: $f : \{0,1\}^8 \rightarrow \{0,1\}$

Filename	Description
4_ones	$f(x) = 1$ iff the string x has four ones, symmetric function
and_or_chain	$\cdot(+(\cdot(+(\cdot(+(\cdot(x_1,x_2),x_3),x_4),x_5),x_6),x_7),x_8)$
add0	most significant bit of a four-bit adder
add2	third bit of a four-bit adder
add4	least significant bit of a four-bit adder
character	(General Description) chXfY means character X from font Y of the Borland font set
ch8f0	kind of a flat plus sign
ch15f0	an Aztec looking design
ch22f0	horizontal bar
ch30f0	solid isosceles triangle
ch47f0	slash
ch176f0	every other column of a checker board
ch177f0	checker board
ch74f1	triplex J
ch83f2	small S (thin strokes)
ch70f3	sans serif F
ch52f4	gothic 4

continued on next page

continued from previous page

Filename	Description
greater_than	$f(x_1, x_2) = 1$ iff $x_1 > x_2$
interval1	interval acceptor, accepts strings with three or fewer intervals (i.e. substrings of all zeros or all ones)
interval2	interval acceptor, accepts strings with four or fewer intervals two sub-string detectors
kdd1	$(x_1 x_3) + \overline{x_2}$
kdd2	$(x_1 \overline{x_2} x_3)(x_4 + \overline{x_6})$
kdd3	$\overline{(x_1 + x_2)} + (\overline{x_1} x_4 x_6)$
kdd4	$\overline{x_4}$
kdd5	$(x_1 x_2 \overline{x_4}) + (x_3 \overline{x_5} x_7 x_8) + (x_1 x_2 x_5 x_6 x_8) + (\overline{x_3}\ \overline{x_5})$
kdd6	$x_2 + x_4 + x_6 + x_8$
kdd7	$(x_1 x_2) + (x_3 x_4) + (x_5 x_6) + (x_7 x_8)$
kdd8	$(x_1 \overline{x_2}) \oplus (x_1 x_5)$
kdd9	$(x_2 \oplus x_4)(\overline{x_1} \oplus (x_5 x_7 x_8))$
kdd10	$(x_1 \Rightarrow x_4) \oplus (\overline{(x_7 x_8)})(x_2 + x_3)$
majority_gate	$f(x) = 1$ iff x has more 1's than 0's
monkish	(General Description) All monkish problems have 6 attributes, where x_4, x_5 and x_6, x_7 represent 4-valued attributes, all other variables represent 2-valued attributes
monkish1	$(x_1 == x_2) \cdot ([x_6, x_7] == 0)$
monkish2	exactly two of the six attributes have the first value
monkish3	$(([x_6, x_7] == 2) \cdot ([x_4, x5] == 0)) + (([x_6, x_7]! = 3) \cdot (x_2! = 1))$
mux8	two-address bit, four-data bit multiplexor with two vacuous variables (x_0 and x_1)
or_and_chain	$+(\cdot(+(\cdot(+(\cdot(+(x_1, x_2), x_3), x_4), x_5), x_6), x_7), x_8)$
pal	palindrome acceptor; created by randomly generating 128 bits then mirror image to create the outputs of an eight-variable function
pal_output	palindrome output; created by randomly generating 128 bits then mirror image to create the outputs of an eight-variable function

continued on next page

continued from previous page	
Filename	Description
pal_dbl_out	palindrome output; created by randomly generating 64 bits then mirror image four times to create the outputs of an eight-variable function
parity	$x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus x_5 \oplus x_6 \oplus x_7 \oplus x_8$
primes8	output is 1 iff the input string, as a decimal number, is prime. 0 and 1 are not considered prime
rnd1	Randomly generated function, created by FLASH with seed 1
rnd2	Randomly generated function, created by FLASH with seed 2
rnd3	Randomly generated function, created by FLASH with seed 3
rnd_m1	One randomly generated minority element, created by FLASH
rnd_m5	Five randomly generated minority elements, created by FLASH
rnd_m10	Ten randomly generated minority elements, created by FLASH
rnd_m25	Twenty five randomly generated minority elements, created by FLASH
rnd_m50	Fifty randomly generated minority elements, created by FLASH
rndvv36	Randomly generated Taylor set, with vacuous variables x_3 and x_6
substr1	accepts strings with the sub-string "101"
substr2	accepts strings with the sub string "1100"
subtraction1	output bit 1 of the absolute value of a four-bit difference (0 is the MSB)
subtraction2	output bit 3 of the absolute value of a four-bit difference (0 is the MSB)

Table A.1: Description of Wright Lab benchmark functions.

A.2 Machine Learning Functions

Filename	in	out	max. Q_i	Q_f	MDD nodes	cubes	% Known
	Description						
audiology	69	1	6	24	6668	200	$1.78e^{-19}$%
	Determine one of 26 hearing problems						
balance	4	1	5	5	31	625	100%
	Balance scale weight & distance						
balloon1	4	1	2	2	2	20	125%
	Inflate balloon if age=adult or act=stretch						
balloon2	4	1	2	2	2	20	125%
	Inflate balloon if age=adult or act=stretch						
balloon3	4	1	2	2	2	20	125%
	Inflate balloon if color=yellow or size=small						
balloon4	4	1	2	2	4	16	100%
	Inflate balloon if color=yellow or size=small						
breastc	9	1	10	2	1119	699	$6.99e^{-05}$%
	Breast cancer data (Michalski has used this)						
bridges1	9	1	4	7	212	108	0.39%
	Determine bridge type						
bridges2	10	1	4	7	288	108	0.13%
	Determine bridge type						
car	6	1	4	4	69	1728	100%
	Determine car cost given many car attributes						
chess1	6	1	8	18	3091	28056	42.81%
	Chess endgame for king and rook versus king						
chess2	36	1	2	2	8446	3198	$3.10e^{-06}$%
	Chess endgame for king and rook versus king and pawn						
cloud	6	1	9	2	174	108	0.05%
	Determine if clouds should be seeded						
connect-4	42	1	3	3	40724	67557	$6.17e^{-14}$%
	Connect-4 game opening move results in win/loss/draw						
employ1	9	1	5	4	49	9600	100%
employ2	7	1	5	5	37	18000	100%
flag	28	1	14	6	3557	194	$1.71e^{-12}$%
	Determine continent of a given flag						
flare1	10	3	7	5	324	323	0.67%
	Solar Flare - Determine common/moderate/severe flares						

continued on next page

continued from previous page

Filename	in	out	max. Q_i	Q_f	MDD nodes	cubes	% Known
	Description						
flare2	10	3	7	5	406	1065	2.20%
	Solar Flare - Determine common/moderate/severe flares						
hayes	4	1	5	3	41	132	35.2%
	Classification of people by four variables						
house-votes	16	1	2	2	407	435	0.66%
	'84 house voting to determine if democrat or republican						
iris	4	1	12	3	41	151	6.29%
	Determine classification of an iris						
irish	4	1	12	3	23	432	100%
	Determine school given test scores, sex, etc.						
lensesmv	4	1	3	3	8	24	100%
	Determine if person has hard/soft or no contacts						
letter	16	1	16	26	77004	20000	$1.08e^{-13}$%
	Patterns to determine a alphabetic letter						
lung-cancer	56	1	4	3	1472	32	$6.16e^{-31}$%
monks1te	6	1	4	2	5	432	100%
	Logic problem test function						
monks1tr	6	1	4	2	77	124	28.70%
	Logic problem training function						
monks2te	6	1	4	2	9	432	100%
	Logic problem test function						
monks2tr	6	1	4	2	88	169	39.12%
	Logic problem training function						
monks3te	6	1	4	2	4	432	100%
	Logic problem test function (5% error added)						
monks3tr	6	1	4	2	78	122	28.24%
	Logic problem training function (5% error added)						
mushroom	22	1	12	2	588	8124	$4.16e^{-09}$%
	Determine if mushroom is edible given different attributes						
nursery	8	1	5	5	41	12960	100%
	Determine by status if parents recommend a nursery						
post-op	8	1	4	3	126	90	2.31%

continued on next page

continued from previous page

Filename	in	out	max. Q_i	Q_f	MDD nodes	cubes	% Known
	Description						
	Determine if patient is discharged (yes/no/unknown)						
prison	80	55	70	104	246350	14137	$1.27e^{-68}$%
programm	12	1	4	5	16420	20000	0.67%
sensory	11	1	6	11	1074	576	0.26%
ships	4	1	5	4	30	34	17.0%
	Number of damages to a ship given different attributes						
shuttlem	6	1	4	2	10	256	100%
	Determine if shuttle launch will occur						
sleep	9	1	21	5	238	62	$5.26e^{-05}$%
	Amount of sleep for different animal species						
sponge	44	1	9	3	1745	76	$4.25e^{-20}$%
	Determine type of sponge given different attributes						
tic-tac-toe	9	3	3	2	338	958	4.86%
	Determine winning move for tic-tac-toe						
trains	32	1	10	2	193	11	$4.52e^{-12}$%
	Determine direction of train						
zoo	16	1	9	7	224	101	0.03%
	Determine animal type for different attributes						
mm3	5	1	5	5	27	243	100%
mm4	5	1	5	5	39	1024	100%
mm5	5	1	5	5	94	3125	100%
pal2	6	1	5	5	21	64	100%
pal3	6	1	5	5	54	729	100%
alet	18	1	5	5	5316	1410	$1.41e^{-13}$%
c1a	6	1	5	5	15	3	2.08%
c1b	6	1	5	5	15	3	4.68%
c2a	11	1	5	5	77	10	$2.26e^{-03}$%
c2b	11	1	5	5	89	10	$3.21e^{-03}$%
c3a	14	1	5	5	108	10	$8.37e^{-05}$%
c3b	14	1	5	5	119	10	$5.02e^{-05}$%
c4a	14	1	5	5	110	10	$7.06e^{-05}$%
c4b	14	1	5	5	120	10	$3.53e^{-05}$%
c5a	13	1	5	5	105	10	$1.43e^{-04}$%
c5b	13	1	5	5	108	10	$7.53e^{-05}$%

continued on next page

continued from previous page							
Filename	in	out	max. Q_i	Q_f	MDD nodes	cubes	% Known
	Description						
c6a	13	1	5	5	116	10	$3.77e^{-06}\%$
c6b	13	1	5	5	102	10	$2.19e^{-06}\%$
d1	6	1	5	5	25	6	1.85%
d2	11	1	5	5	141	20	$1.62e^{-03}\%$
d3	14	1	5	5	203	20	$2.03e^{-05}\%$
d4	14	1	5	5	219	20	$1.52e^{-05}\%$
d5	13	1	5	5	196	20	$2.32e^{-05}\%$
d6	13	1	5	5	204	20	$1.58e^{-07}\%$
d7	24	1	5	5	416	20	$5.14e^{-14}\%$
d8	32	1	5	5	588	20	$4.02e^{-19}\%$
d9	34	1	5	5	629	20	$2.51e^{-20}\%$
d10	37	1	5	5	688	20	$7.85e^{-22}\%$
geo	11	1	5	5	831	432	$5.22e^{-03}\%$
let	18	1	5	5	5304	1410	$1.41e^{-13}\%$
sim61	60	1	5	5	178	3	$1.54e^{-20}\%$
u1	60	1	5	5	9839	192	$1.82e^{-35}\%$
u1_10	60	1	5	5	571	10	$1.17e^{-30}\%$
u1_2	60	1	5	5	119	2	$1.73e^{-16}\%$
u1_3	60	1	5	5	178	3	$1.54e^{-20}\%$
u1_4	60	1	5	5	237	4	$3.81e^{-22}\%$
u1_5	60	1	5	5	295	5	$6.62e^{-24}\%$
u2	60	1	5	5	10085	192	$2.50e^{-32}\%$
u3	60	1	5	5	9831	192	$4.64e^{-34}\%$
u4	60	1	5	5	9989	192	$2.50e^{-32}\%$
u5	60	1	5	5	10064	192	$3.76e^{-32}\%$

Table A.2: Description of Machine Learning benchmark functions.