Minterm Based Search Algorithms for Two-Level Minimization of Discrete Functions

by

Michael James Whitney
B.Sc., University of Alberta, 1986
M.Sc., University of Victoria, 1988

A dissertation submitted in partial fulfillment
of the requirements for the degree of
DOCTOR OF PHILOSOPHY
in the Department of Computer Science

We accept this dissertation as conforming
to the required standard

Dr. J. C. Muzio, Supervisor (Department of Computer Science)

Dr. D. M. Miller, Departmental Member (Department of Computer Science)

Dr. H. A. Müller, Departmental Member (Department of Computer Science)

Dr. P. F. Driessen, Outside Member (Dept. of Electrical & Computer Engineering)

Dr. R. C. Aitken, External Examiner (Hewlett Packard Co., Santa Clara, CA, USA)

Name Michael J. Kearns

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Computer Science — SUBJECT TERM

0984 — SUBJECT CODE

UMI

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS
Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION
General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS
Language
- General 0679
- Ancient 0289
- Linguistics 0290
- Modern 0291

Literature
- General 0401
- Classical 0294
- Comparative 0295
- Medieval 0297
- Modern 0298
- African 0316
- American 0591
- Asian 0305
- Canadian (English) 0352
- Canadian (French) 0355
- English 0593
- Germanic 0311
- Latin American 0312
- Middle Eastern 0315
- Romance 0313
- Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY
Philosophy 0422
Religion
- General 0318
- Biblical Studies 0321
- Clergy 0319
- History of 0320
- Philosophy of 0322

Theology 0469

SOCIAL SCIENCES
American Studies 0323
Anthropology
- Archaeology 0324
- Cultural 0326
- Physical 0327

Business Administration
- General 0310
- Accounting 0272
- Banking 0770
- Management 0454
- Marketing 0338

Canadian Studies 0385
Economics
- General 0501
- Agricultural 0503
- Commerce Business 0505
- Finance 0508
- History 0509
- Labor 0510
- Theory 0511

Folklore 0358
Geography 0366
Gerontology 0351
History
- General 0578
- Ancient 0579
- Medieval 0581
- Modern 0582
- Black 0328
- African 0331
- Asia, Australia and Oceania 0332
- Canadian 0334
- European 0335
- Latin American 0336
- Middle Eastern 0333
- United States 0337

History of Science 0585
Law 0398
Political Science
- General 0615
- International Law and Relations 0616
- Public Administration 0617

Recreation 0814
Social Work 0452
Sociology
- General 0626
- Criminology and Penology 0627
- Demography 0938
- Ethnic and Racial Studies 0631
- Individual and Family Studies 0628
- Industrial and Labor Relations 0629
- Public and Social Welfare 0630
- Social Structure and Development 0700
- Theory and Methods 0344

Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES
Agriculture
- General 0473
- Agronomy 0285
- Animal Culture and Nutrition 0475
- Animal Pathology 0476
- Food Science and Technology 0359
- Forestry and Wildlife 0478
- Plant Culture 0479
- Plant Pathology 0480
- Plant Physiology 0817
- Range Management 0777
- Wood Technology 0746

Biology
- General 0306
- Anatomy 0287
- Biostatistics 0308
- Botany 0309
- Cell 0379
- Ecology 0329
- Entomology 0353
- Genetics 0369
- Limnology 0793
- Microbiology 0410
- Molecular 0307
- Neuroscience 0317
- Oceanography 0416
- Physiology 0433
- Radiation 0821
- Veterinary Science 0778
- Zoology 0472

Biophysics
- General 0786
- Medical 0760

EARTH SCIENCES
Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES
Environmental Sciences 0768
Health Sciences
- General 0566
- Audiology 0300
- Chemotherapy 0992
- Dentistry 0567
- Education 0350
- Hospital Management 0769
- Human Development 0758
- Immunology 0982
- Medicine and Surgery 0564
- Mental Health 0347
- Nursing 0569
- Nutrition 0570
- Obstetrics and Gynecology 0380
- Occupational Health and Therapy 0354
- Ophthalmology 0381
- Pathology 0571
- Pharmacology 0419
- Pharmacy 0572
- Physical Therapy 0382
- Public Health 0573
- Radiology 0574
- Recreation 0575
- Speech Pathology 0460
- Toxicology 0383

Home Economics 0386

PHYSICAL SCIENCES
Pure Sciences
Chemistry
- General 0485
- Agricultural 0749
- Analytical 0486
- Biochemistry 0487
- Inorganic 0488
- Nuclear 0738
- Organic 0490
- Pharmaceutical 0491
- Physical 0494
- Polymer 0495
- Radiation 0754

Mathematics 0405
Physics
- General 0605
- Acoustics 0986
- Astronomy and Astrophysics 0606
- Atmospheric Science 0608
- Atomic 0748
- Electronics and Electricity 0607
- Elementary Particles and High Energy 0798
- Fluid and Plasma 0759
- Molecular 0609
- Nuclear 0610
- Optics 0752
- Radiation 0756
- Solid State 0611

Statistics 0463

Applied Sciences
Applied Mechanics 0346
Computer Science 0984

Engineering
- General 0537
- Aerospace 0538
- Agricultural 0539
- Automotive 0540
- Biomedical 0541
- Chemical 0542
- Civil 0543
- Electronics and Electrical 0544
- Heat and Thermodynamics 0348
- Hydraulic 0545
- Industrial 0546
- Marine 0547
- Materials Science 0794
- Mechanical 0548
- Metallurgy 0743
- Mining 0551
- Nuclear 0552
- Packaging 0549
- Petroleum 0765
- Sanitary and Municipal 0554
- System Science 0790

Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY
General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

Supervisor: Jon C. Muzio

Abstract

Techniques for the heuristic and exact two-level minimization of Boolean and multi-valued functions are presented. The work is based on a previously existing algorithmic framework for two-level minimization known as directed search. This method is capable of selecting covering prime implicants without generating all of them. Directed search differs from most other minimization methods in that implicant cubes are generated from minterms, not from other cubes.

Heretofore, the directed search algorithm and published variants have not been capable of minimizing PLA's of "industrial" size. The algorithms in this work significantly ameliorate this situation. In particular, original and efficient techniques are proposed for prime implicant generation, computation of dominance relations, elimination of redundant minterms, storage and retrieval of cubes and minterms, and isolation and reduction of cycles.

The algorithms are embodied in a working minimizer called MDSA. In the absence of cycles, MDSA provides provably optimum cube covers. Empirical comparison with other minimizers show the new algorithms to be very competitive, even superior. For mid-sized non-cyclic PLA's, MDSA is nearly always faster, and usually faster for PLA's containing cycles, than the best known heuristic competitor. The number of cubes found for cyclic PLA's is also better (lower), on average. MDSA can also be set to provide provably minimum solutions for cyclic functions. In this case, MDSA again outperforms competitive minimizers in a similar mode of operation. Both heuristic and exact versions of MDSA are restricted to PLA's with 32 or fewer inputs, and 32 or fewer outputs.

Examiners:

Dr. J. C. Muzio, Supervisor (Department of Computer Science)

Dr. D. M. Miller, Departmental Member (Department of Computer Science)

Dr. H. A. Müller, Departmental Member (Department of Computer Science)

Dr. P. F. Driessen, Outside Member (Dept. of Electrical & Computer Engineering)

Dr. R. C. Aitken, External Examiner (Hewlett Packard Co., Santa Clara, CA, USA)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Two Level Logic Design

Logic design is the process of finding a realization for a given logic function. It has immediate relevance to the manufacture of digital integrated circuits, where the use of chip "real estate" remains a significant issue. Design can be performed at any of a number of abstraction levels; one such level uses *logical gates* as primitive building blocks. The objective of logic *minimization* is to find a realization for a given logic function, or set of functions, that requires the smallest possible number of devices. The subject of this dissertation is the minimization of gate level realizations of logic functions, where the topology of the realization is restricted to only two levels.

In general, a given logic function can be implemented in many different ways. A *two-level* implementation is one in which every path from an input signal to the function output passes through at most two gates. A *sum-of-products* form is an algebraic expression, and it corresponds to a two-level design where a signal always passes through first an `AND` gate, then an `OR` gate. Conceptually, this is exactly the

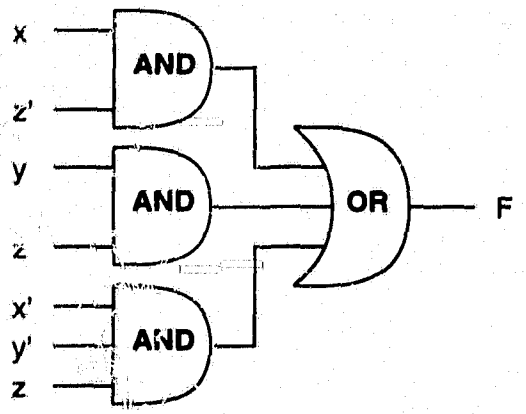


Figure 1.1: Simple Two-level Circuit

design realized by a device called a *programmable logic array*, or PLA. The size of a PLA is directly related to the number of products, or *cubes*, in the corresponding sum-of-products expression. A simple sum-of-products gate-level circuit is illustrated in Figure 1.1.

Ease of automatic generation and layout of PLA's make them a useful and popular device to be included in computer chips. Although many functions can be realized in smaller area through multi-level, or "random" logic, even recent computer chips, comprising many hundreds of thousands of transistors, still incorporate PLA's for some control functions. Two-level representations are also used as functional descriptors in random logic design algorithms [BBH+88, MB90]. The size of the representation impacts the efficiency of such design algorithms.

Two-level logic design is much easier than multi-level design. Still, there are enough degrees of freedom for the problem to be far from trivial. In fact, the minimization of two-level representations is one of the most studied questions in computer

science, as well as in formal logic and philosophy[1]. Some of the notable early pioneers to address the problem from a computational or algorithmic point of view are Shannon, Karnaugh, Quine, and McCluskey. Research has continued through the decades, with a surprising quantity of renewed interest and published results in the last few years (*e.g.* [CM92, CM93, LCM92, BMS92]).

1.2 The Minimization Problem

A sum-of-products expression is of the form $S = p_1 + p_2 + ... + p_t$. Each product p_i is composed of from one to n literals, each of which corresponds to one input signal in positive or negated form. For example, the sum-of-products expression for the circuit of Figure 1.1 is

$$F = xz' + yz + x'y'z \ .$$

The set of product terms $\{p_1, p_2, ..., p_t\}$ is called a *cover* for S. The *minimization* problem for sum-of-products expressions is simply to minimize t; the number of cubes in the cover[2]. A minimization for a given problem is *exact* if there does not exist an alternative cover for it with smaller t. The three term expression for the circuit of Figure 1.1 is exact because there does not exist another sum-of-products expression for the same function with fewer terms. An exact solution is not necessarily unique; another expression for the example is $F = x'z + xy + yz'$.

The problem of finding sum-of-products expressions for single Boolean functions is of limited value in practice. PLA's realize many Boolean functions simultaneously. These functions may be thought of collectively as a single output function with one

[1] In this domain the sum-of-products representation is known as *disjunctive normal form.*

[2] Alternative interpretations for "minimization" exist, but are not of interest here.

multi-valued variable [Sas78]. Similarly, the other input variables need not be limited to Boolean (two-valued) variables. Multi-valued variables are useful in PLA-based designs of finite automata [BHMSV85], and in the design of PLA's augmented with additional input encoding circuitry [RSV87]. The use of higher order signals and devices in logic circuits is limited at present, but interest in such devices is growing.

As is common in many areas of computer science, there is a trade-off between the size t of a cover, and the resources required to compute the cover. Programs that strive to minimize t must seek a compromise. An unfortunate effect of this is exact minimizers that sometimes take more resources than necessary for comparatively easy problems, and heuristic minimizers that return poor covers for problems that would have been easy to solve exactly.

1.3 Motivation

Two-level minimization is known to belong to the class of NP-complete problems [BHMSV85]. Since this leaves little chance of finding optimum solutions in the worst case, it is sometimes argued that heuristics *must* be used, and a "good" rather than optimum solution sought, if the input PLA (or set of functions) has more than some threshold number of inputs and outputs. This argument is frequently heard elsewhere in computer science. However, it is unknown until minimization proceeds, just how difficult an input problem might be. It is needless and disadvantageous to assume intractability from the outset. A better approach is to isolate difficult subproblems, then solve them separately using heuristics only if necessary. Conditions indicating difficulty are left to arise during the process of solving the problem, rather than being based *a priori* on simple input parameters.

Some two-level minimization algorithms work directly with existing covers, per-

forming iterative simplifications on these covers while their sizes can still be reduced. These minimizers are not exact, though they are able to provide near-minimum covers in many scenarios. Most exact minimizers are variations on a common theme, whereby a very large set of cubes is generated, and then a suitable cover extracted. This approach has the drawback of requiring the generation of potentially many cubes that are subsequently found to be irrelevant to the covering problem.

A third approach bases the minimization algorithm on individual "points" in the function to be minimized. Such an approach is said to be *minterm-based* [GB89], and has the advantage of being tailorable to both heuristic and exact minimization. The research reported here grew out of experiments with a minimization algorithm known as *directed search* [McK74, RNMP77]. The algorithm shows much promise, because it has the property that the task at hand is kept "simplified" as minimization proceeds. However, not all available simplifying techniques are incorporated, and the algorithm is incapable of minimizing anything except very small single-output "toy" problems.

1.4 Approach

The aim of this research is to determine if two-level minimization based on minterm search is viable for problems of "industrial" size (*e.g.*, PLA specifications with fourteen or more input variables). Both heuristic and exact minimization algorithm variants are considered. The chosen approach is dissection of the original directed search algorithm (from [McK74]) into its constituent parts, followed by improvement or replacement of each part. In this scenario, the original algorithm is interpreted as a high-level "template" into which lower level algorithms are inserted. Also, the template is altered in several key ways to permit the inclusion of additional techniques, and data structures are provided to make basic operations more efficient.

The efficacy of the methods, techniques, and algorithms reported here is demonstrated both empirically, and by formal argument. The data come from numerous experiments performed on a set of benchmark PLA's, taken from benchmark sets used extensively in the literature. Thus, comparisons between the current work and other published work is possible. Concrete comparisons are made with the heuristic and exact versions of *espresso* [BHMSV85, RSV87], considered the state-of-the-art in two-level minimization technology, and readily available in most VLSI research settings.

1.5 Dissertation Overview

The remainder of this dissertation is organized as follows. Chapters 2 and 3 provide the necessary background material. Fundamentals of logic design are reviewed in Chapter 2, with particular emphasis on concepts involved in sum-of-products expressions and their minimization. The chapter also includes a short summary of past and current research in two-level minimization, and discusses some of the working programs implementing these ideas. Both heuristic and exact minimization techniques are included. The two minimizers used for comparison purposes throughout this work, namely *espresso* and *espresso-exact*, are introduced.

Chapter 3 provides the precise background material for the present work. The directed search algorithm is reviewed, as well as related research. The terminology and nomenclature associated with the algorithm, and used throughout the sequel, are introduced. A sharp lower bound on the number of required algorithm iterations is derived, thereby providing an approximation to how well the algorithm may be expected to perform. The chapter concludes by developing a methodology for the current work, based on perceived and potential shortcomings of directed search.

Chapter 4 introduces the prototype program developed for testing the ideas reported in this research. The main data structures and their accessing methods are discussed. The chapter also considers one of the main tenets of directed search: that of "direction", *i.e.* specifying how the search for a minimum solution proceeds based on heuristic choices. It is revealed how "essential probabilities" are derived from the calculations involved in making the choices, thereby increasing the power of the heuristics.

The directed search algorithm depends on up-to-date information concerning dominance relations among cubes, which in turn depends on the concept of cube cells. Chapter 5 shows how certain conditions are exploited to avoid dominance tests altogether, by conceptually classifying cubes into four categories. Algorithms for rapidly calculating the cells of cubes when updating is unavoidable, are presented.

Chapter 6 introduces a simple tree-based data structure used to store minterm and cube sets. It is shown how the times for searches for single members and larger subsets are reduced. It is also shown how these cube trees can be created and maintained as partial orders of bounded depth, called "cube heaps". Maintaining sets as heaps makes selection of key members immediate, and simplifies the calculation of certain heuristics. The cost for insertion and deletion of individual members is shown to be related to the length of their representation.

A key aspect of all minimizers is how and when the relevant cubes are discovered, and how few irrelevant ones are discovered. Chapter 7 deals with the problem of prime implicant generation. The original directed search algorithm uses a depth-first tree search, augmented with complex rules for avoiding redundant computations. An alternative algorithm is introduced, which uses a more flexible approach permitting implicant generation to be guided by past discoveries. The new technique is compared and contrasted against the old one.

The original directed search algorithm, and all subsequent work to the present, does not incorporate dominance relations among minterms. Chapter 8 presents efficient techniques for doing so, based on the product of cube sets. The relevance of cube cell intersections is demonstrated. A comparison between versions of directed search with and without minterm dominance is provided, indicating the importance of the technique.

Chapter 9 discusses the phenomenon of cycles, which correspond to the nondeterministically solvable set covering problem of graph theory. It is shown how cycles are solved both heuristically and exactly, utilizing the cube heap data structure to ensure memory usage linear in the number of primitive objects involved in the cycle. An improved branch-and-bound search model for finding exact minimum solutions is presented. The model incorporates efficient use of lower and upper bounds, partitioning, and backtracking capabilities.

Chapter 10 introduces a technique for storing all data structures as sets within the minimizer, thereby avoiding the requirement for exponential storage. The new method is based on cube heaps, and requires far less storage in the best and average case than previous directed search algorithms, for larger problem instances. The required changes to existing directed search minimizers are discussed. The version of the minimizer incorporating the introduced storage techniques, is compared against the previous version and other minimizers.

In Chapter 11, the key results of this work are summarized, along with their significance in a broader context. Strengths and weaknesses of the proposed algorithms are emphasized, and potential avenues of continuing research are outlined. Finally, Appendix A is a small glossary of the terms and phrases used in this work, and Appendix B provides the meanings of abbreviations used.

Chapter 2

Background

2.1 Introduction

This chapter provides general background material required to understand, compare, and contrast the algorithms and techniques introduced in subsequent chapters. The next section provides a review of basic concepts from Boolean and multi-valued algebra. An investigation into the fundamental principles of two-level circuit design follows. Section 2.4 introduces the bit-string notations used throughout this work for representing minimization objects, and how basic operations are performed on these using simple logical bit-wise functions. Some important two-level minimization approaches are briefly summarized in the final two sections, including both heuristic and exact methods. Attention is focussed primarily on "the competition": minimizers that have received significant use in practice.

2.2 Boolean and Multi-valued Algebras

The definitions and fundamental minimization rules given below may be found in many logic design texts (*e.g.* [Giv70, Die88, McC86, Rot85]). Here, attention is restricted to those definitions and results of immediate relevance to the present work.

A *Boolean function* $f(\mathbf{X})$ maps n input variables $\mathbf{X} = \{\mathbf{x_0}, \mathbf{x_1}, \mathbf{x_{n-1}}\}$ to a single output f, where all the $x_i \in \{0, 1\}$, and each point in the domain ot f is in $\{0, 1, X\}$. Such a function may be specified by a *truth-table*, in which the function value for each possible assignment to the input variables is enumerated.

A *vertex* is a point in the domain of a discrete function f. In the case of Boolean functions, a vertex corresponds to a unique assignment of 0's and 1's to the n input variables. This assignment provides a convenient mechanism for specifying the vertex, *i.e.* as a string of n 0's and 1's. The function space of f contains 2^n vertices. A vertex assuming the value 1 is called a *true minterm*, or simply *minterm* (short for *minimal term*).

A vertex assuming the value X is called a *"don't-care"*, which means that it is irrelevant to the function whether that vertex assumes the value 1 or 0. A function containing one or more "don't-cares" is said to be *incompletely specified.* Otherwise, it is *fully specified.*

The number of variables that are different in two Boolean vertices is the *Hamming distance* between them. For example, the vertices `0100110` and `1101010` are at Hamming distance three. A vertex is *adjacent* to another vertex *iff* they differ in only one variable, *i.e.* they are at Hamming distance one.

The set of vertices for which f equals 0 is called the *OFF-set.* Similarly, the set of 1 vertices is the *ON-set*, and the "don't-care" vertices comprise the *DC-set.*

abcd	f	abcd	f
0000	0	1000	0
0001	0	1001	1
0010	0	1010	0
0011	1	1011	0
0100	1	1100	0
0101	x	1101	1
0110	0	1110	1
0111	1	1111	1

Figure 2.1: Truth-table for a Boolean Function

Figure 2.1 illustrates the above concepts. This is a truth-table for an incompletely specified four variable Boolean function $f(a, b, c, d)$. Corresponding to each of $2^4 = 16$ distinct assignments to the four variables, given in lexicographic order, is the output value for the function f. Of these sixteen vertices, the ON-set contains seven, the OFF-set contains eight, and the DC-set contains one. The single "don't-care" vertex, 0101, is permitted to assume the value 0 or 1, when a fully specified expression, or circuit, is found for f.

Algebraic expressions are constructed using variables, the binary operators AND and OR, and the unary operator NOT. The AND operation is also called *product*, and is indicated in an expression by the concatenation of subexpressions. The OR operation is called *sum*, and is indicated by the + symbol.

By convention, product is of higher priority than sum. The NOT operation is also called *complement*, and has the highest priority. A *literal* is an appearance of a variable v or its complement v'. A *product term*, or simply *term*, is the product of one or more literals.

A simple though lengthy expression for the function in Figure 2.1 corresponds to

an enumeration of its ON-set,

$$f = a'b'cd + a'bc'd' + a'bcd + ab'c'd + abc'd' + abcd' + abcd \tag{2.1}$$

In this case, the "don't-care" vertex is moved to the OFF-set. Another expression, logically equivalent to the above, is

$$f = a'(b'cd + bc'd' + bcd) + ab'c'd + ab(c'd' + cd' + cd) \tag{2.2}$$

where a' is "factored out" of the first three terms, and ab is factored out of the last three.

A *multi-valued* variable x_i may take on any of r_i different discrete values, where r_i is the *radix* of the variable, and is fixed. A multi-valued literal is of the form x_i^R, where R is a subset of $\{0, 1, 2, ..., r_i - 1\}$. Thus, there are $2^{r_i} - 2$ distinct non-trivial multi-valued literals associated with x_i. A multi-valued input, single binary-output function maps its multi-valued input variables to $\{0, 1, X\}$. A *multiple-output* binary function maps n input variables to m separate output functions. Because the algebraic notation for multi-valued variables is so unwieldy, further discussion is postponed to Section 2.4.

2.3 Basics of Two-level Minimization

A *sum-of-products* expression, or SOP, is of the form $f(X) = p_1 + p_2 + p_3 + ... + p_t$ where each p_i is a product term (*i.e.* a conjunction of literals). Whether or not the variables are multi-valued, each term evaluates to 1 or 0, and thus $f(X)$ is a two-valued, or *decisive*, function. Equation 2.1, above, is an example of a simple sum-of-products expression for the function of Figure 2.1.

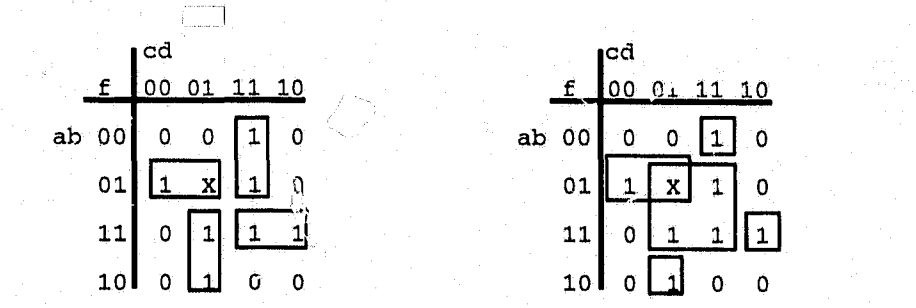


Figure 2.2: Four-variable Karnaugh Maps

An *implicant* of a function f is a term none of whose vertices are zero in f. A *prime implicant* I is an implicant that is maximal, *i.e.* no larger implicant contains I. Implicants of a function f are also called *cubes*, because they correspond to a hypercube in the function space of f. A cover is prime if all cubes included in the cover are themselves prime. The *dimension* of a cube is the number of variables *not* involved in its corresponding term. Thus, a cube of zero dimension is equivalent to a vertex.

One way to represent a Boolean function is with a *Karnaugh map*. This representation is basically tabular, but entries are arranged so that adjacencies are more easily recognized. Two four-variable Karnaugh maps for the function of Figure 2.1 are illustrated in Figure 2.2. Each side of the figure represents a possible cover for f. Cubes of the covers are drawn on the map as rectangles. The first cover corresponds to equation

$$f = a'bc' + a'cd + ac'd + abc \tag{2.3}$$

and the second to equation

$$f = a'bc' + a'b'cd + abcd' + ab'c'd + bd \tag{2.4}$$

A desirable quality for a cover is *irredundancy*, which means that the correctness of the cover cannot be maintained if any one cube is removed from it. Such a cover is by definition *minimal*[1]. (In fact, in this context minimal is equivalent to irredundant.) A *minimum*, or *exact*, cover of f is as small as any cover for f could possibly be; there do not exist any other covers with fewer cubes. A minimal cover need not be minimum, but a minimum cover must be minimal. The cover illustrated in Figure 2.2(a) is exact. Both covers are irredundant, but the one in Figure 2.2(b) is not prime.

An *essential minterm* is one covered by only one prime implicant, which is then called an *essential prime implicant*. The minterms on the perimeter of Figure 2.2(a) { 0100, 0011, 1110, 1001 } are all essential, as are their covering cubes. However, the remaining three minterms of the ON-set, namely { 0111, 1101, 1111 }, are non-essential because they are each covered by two prime implicants.

A cube C_1 is said to *dominate* another cube C_2 *iff* all the minterms in C_2 are also contained in C_1. Any non-prime implicant is dominated by a prime containing it. However, it is also possible for one prime cube to dominate another, when the second contains one or more "don't-care" vertices. The two primes shown in Figure 2.3(a) illustrate this relation.

A minterm M_1 is said to *dominate* another minterm M_2 *iff* if M_1 is covered by a superset of the primes that cover M_2. In Figure 2.3(b), minterm 1001 dominates 1101. Only the two outlined cubes cover the latter, both of which also cover the former. The relation is not symmetric: note that minterm 1001 is covered by a third prime, shown with dotted outline. (This last cube is more difficult to draw, because it straddles the boundary corresponding to $b = 0$, and a is both 0 and 1.)

A *pseudo-essential minterm* is one that is not essential, yet is covered by a prime

[1] A set is *minimal* with respect to a particular characteristic, if that characteristic is no longer true upon removal of any one member of the set.

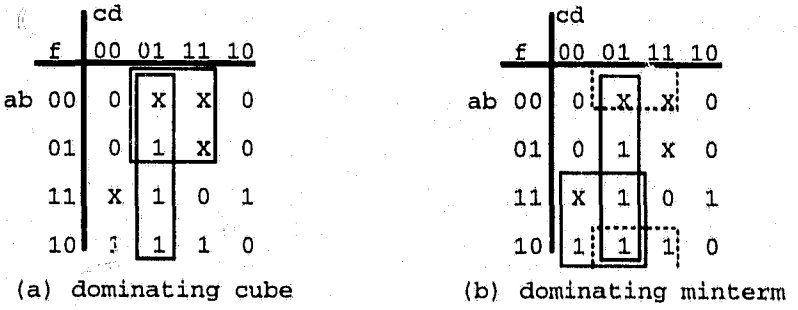


Figure 2.3: Dominance

implicant that dominates all other prime implicants covering the minterm. Minterm 0101 in Figure 2.3 is pseudo-essential, because one of its two covering primes dominates the other.

Exact two-level minimization requires the identification of all essential and pseudo-essential cubes of a given input problem. It is a basic result that only prime implicants need be considered [Giv70]. If all the essential and pseudo-essential cubes are not sufficient to cover the entire ON-set, then the remaining part of the problem is referred to as a *cycle*. Cycles consist of the remainder of the ON-set, plus all undominated prime implicants covering at least one of those minterms. There is no deterministic method for solving cycles; the problem is NP-complete [GJ79].

2.4 Representation of Terms

This section shows how the objects of two-level minimization are represented in the remainder of this work. Two-valued variables may assume the value 0, 1, or X. A cube of n binary variables is then represented by a string of n catenated symbols taken from

this alphabet, thereby avoiding the more awkward representation involving subsets of literals and their complements. For example, let the number of variables n equal four. Then, the string `0010` represents a vertex, and the string `X0X0` represents a cube (of dimension two).

A bit-vector representation is also used for multi-valued cubes and minterms. For each variable x_i of radix r_i, r_i bits are required. By convention, the rightmost of these bits corresponds to r_i assuming the value 0; the leftmost bit corresponds to the value $r_i - 1$. For example, let $f(x_2, x_1, x_0)$ be a multi-valued input decisive function, where the radices of the variables are $r_2 = 4$, $r_1 = 2$, and $r_0 = 3$. Then, the zero minterm is written `0000,00,000` (commas are used to help distinguish the substrings for adjacent variables). The minterm corresponding to $x_2 = 2$, $x_1 = 1$, and $x_0 = 2$ is written `0100,10,100`.

Given the bit-vector representation for cubes and minterms, it is possible to define the primitive operations required for them, as bit-wise operations and relations. In this way, the complication arising from the use of obscure notations and alphabets is avoided.

A cube C contains a minterm M *iff* there does not exist a set bit in M's bit-vector that is cleared in C's. The rule extends to cube containment. For example, `0110,010,11` contains minterm `0010,010,01` and cube `0100,010,11`.

The intersection cube I, or *product*, of cubes C and D is given by the bit-wise AND of their bit-vectors. If the resulting bit-vector for I contains at least one substring of zeros corresponding to a variable, then the intersection is empty (I is not a valid cube). For example, `0110,010,11` intersects with `0011,011,11`, giving the cube `0010,010,11`. However, `0110,010,11` does not intersect with `1000,010,01`, because there is no commonality in the leftmost variable.

In this work, attention is chiefly directed to the case of n input variables, and m output variables. As Sasao shows, minimization of such a multi-function is equivalent to the minimization of one single function of n Boolean variables, and one m-valued variable [Sas78]. To increase readability, in the sequel cubes and minterms are written using a combination of the two alphabets. For example, the cube `0X10,011` in this notation indicates that $n = 4$ and $m = 3$. Thus, there are n two-valued variables, and the multi-valued variable is three-valued. The same cube in the strictly multi-valued notation is written `01,11,10,01,011`.

2.5 Heuristic Minimizers

Most heuristic two-level minimizers may be classified into one of two broad categories. The approach of the first category is to iteratively improve an existing cover until some halting criteria are met. The approach taken by minimizers of the second category is to simplify an exact (or pseudo-exact) algorithm, by making guesses when there are (apparently) no pseudo-essential cubes available. This section introduces the most significant minimizer of only the first category, since minimizers of the second category are really just special cases of exact minimizers.

2.5.1 Espresso

The *espresso* minimizer [BHMSV85, Rud86, RSV85, RSV87, BHMSV84] is largely based on an iterative improvement strategy first embodied in MINI [HCO74]. Currently, *espresso* is perhaps the most used two-level minimizer. The program is reasonably efficient, and usually outputs near-minimum covers. A complementation algorithm is used for obtaining the OFF-set, which in turn is used for prime genera-

tion. Complementation and prime "expansion" utilize fast recursive operators based on *unateness* and *tautology* principles [BCH+82]. Recursion is based on a unateness measure of variables, and heuristics are used to find these splitting variables. The main iterations are quite similar to those used by MINI, but *espresso* also recognizes and deletes essential cubes in the first iteration. The program then continues to iterate, generating new covers by breaking cubes up into subcubes and then attempting new re-combinations, until no improvement over previous iterations is realized.

Unless all the cubes are essential, *espresso* is incapable of knowing how close to the exact minimum an output cover is. However, it is an important point that espresso never outputs more cubes than it receives as input. Hence, if a minimum cover is input, a minimum cover is output. Moreover, as a general rule the minimization proceeds faster when the input is closer to the minimum, because *espresso* recognizes sooner that it cannot improve on the previous iteration. The approach to multi-valued minimization is to consider outputs separately in the low-level operators, then to merge these low-level results across all functions.

2.6 Exact Minimizers

2.6.1 Quine-McCluskey and Prime Implicant Tables

The traditional approach to exact two-level minimization is the following three-phase procedure:

1. Generate all prime implicants.
2. Construct and simplify a prime implicant table.
3. Solve cycles.

Most exact minimizers use some variant of this basic plan. Quine and McCluskey were among the first to formalize the method [McC56]. In particular, they introduced the famous *Quine-McCluskey* procedure (Q-M) for generating all prime implicants from the minterms. However, the technique is impractical since not only may there be too many prime implicants to construct a table, but memory requirements may also explode while isolating them. This is because sub-primes are all stored at each iteration of the algorithm. Other researchers have subsequently improved on the basic approach, but the necessity to generate all PI's remains.

When all PI's are available, a *prime implicant table* is created, where rows correspond to cubes, and columns correspond to minterms. Entries of the table indicate containment of minterms in cubes. The PI table is a representation for a problem from graph theory called *minimum cover* [GJ79]. In general, the problem's parts are *sets* and *elements*, which correspond to cubes and minterms, respectively. Techniques exist for simplifying the table, namely recognition of essential elements, and dominating elements and sets [Giv70]. When no simplifications are applicable, the table is *cyclic*. In this case, the table represents the *cyclic core* of the input function, which is the portion known to be NP-complete [GJ79].

2.6.2 McBOOLE

The McBOOLE program is reported in [DAR85, DAR86]. This exact minimizer includes a graph-based method for generating prime implicants, which avoids some of the overhead incurred by Q-M. A graph is also used for storing the primes, and it provides additional information which assists in the location of dominated primes. The effects of cube selections are calculated and applied locally. Branching and partitioning techniques are used to help solve cycles. However, the cycles recognized

by McBOOLE are not true cycles, because the algorithm does not include minterm dominance techniques. Also, it requires a huge PI table initially, just like Q-M.

2.6.3 Espresso-exact

Espresso-exact [RSV86, RSV87] is built on the low-level operators of *espresso*, and even uses some of the latter's algorithms in the initial phase of isolating and removing essential cubes. Then, the minimizer creates a PI table where essential cubes, and their contained minterms, are omitted. Nertheless, many PI's may remain in this table, because the pseudo-essential cubes are still present. Repeated application of the standard table simplification techniques reduce the PI table to a truly cyclic one. Good heuristics are used for the branch-and-bound phase; these are discussed further in Chapter 9. A related heuristic algorithm is realized by making only one path through the search space. In some circumstances the heuristic solution is provably minimum because of lower bound calculations.

Chapter 3

The Directed Search Algorithm

3.1 Introduction

This chapter reviews the directed search algorithm, which is the basis for the present research. In addition to the original algorithm, there have been three published related variations. Each is briefly described. All versions of directed search are based on an iterative strategy resulting from the repeated selection of certain minterms, and in some variations, groups of minterms. A lower bound on the required number of iterations is derived. The strengths and weaknesses of directed search are identified, and a detailed methodology for the present research, based on these characteristics, concludes the chapter.

3.2 The Directed Search Algorithm

The original directed search algorithm (DSA), described by McKinney [McK74], is intended for the design of single-output two-level binary circuits. The basic idea of

DSA is to generate only the prime implicants that cover selected minterms. Each minterm selection constitutes the first part of one main iteration of the algorithm. Implicants found to be pseudo-essential for their generating minterm are immediately selected for the cover, and already-generated implicants are rechecked for pseudo-essentiality created through cube dominance. Not all minterms need be selected, nor all prime implicants generated, as long as essential and pseudo-essential implicants are found early in the process. This is the key to efficiency, and is in direct contrast to other exact algorithms which must first generate *all* prime implicants (*e.g.* McBOOLE [DAR86]), or a very large proportion of them (*e.g. espresso-exact* [RSV87]).

DSA requires a list of minterms as input, and gradually builds a list of *potentially cyclic prime implicants*, or PCPI, whose essentiality properties are unknown. In each main iteration of the algorithm, a *prime implicant set* PIS is generated by *expanding* a selected uncovered minterm into all previously undiscovered prime implicants that cover that minterm. Primes found to be dominated by other primes are removed. When essentiality (or pseudo-essentiality through dominance relations) of primes is proven, they are removed from PCPI and added to the cover. The algorithm proceeds as follows:

1. Choose an uncovered, unexpanded minterm MT. If there are none left, the solution has been obtained.

2. Find the set PIS of all undominated prime implicants covering MT by expanding MT. Add any new implicants to PCPI.

3. If there is only one prime implicant P covering MT, select it for the cover, as it must be either essential or pseudo-essential. Mark all minterms contained in P as covered (or "don't-care"), and go to step 5.

4. If there is more than one prime implicant covering MT, add MT to the set of expanded minterms EMTS, and select another MT from the set of all unexpanded minterms covered by at least one cube in PCPI. Recursively try again, *i.e.* go to step 2. If there is no such "recursive" MT, go to step 7.

5. Find the subset of all cubes in PCPI that contain at least one minterm just covered by the selected cube P. For each such affected cube, find out if it is dominated by any other cube in PCPI. Remove from PCPI all cubes dominated in this way.

6. If there are no cubes left in PCPI, go to step 1. If there is now an expanded minterm covered by only one cube in PCPI, go to step 3. Otherwise, go to step 4.

7. A cycle exists, consisting of all cubes in PCPI, which collectively cover the current set of expanded minterms. The cycle must be resolved either exactly, (afterwards, go to step 1), or heuristically (choose the "best" prime implicant P for the cover, then go to step 5).

A minterm selected for expansion always comes from one of two classes:

- those minterms not yet considered (the *new minterms*, or NMT's), or
- those minterms already covered by at least one implicant in PCPI (the *recursive minterms*, or RMT's).

New minterms are not selected unless there are no recursive minterms available. The selected minterms, each of whose set of covering prime implicants have been found, are called *expanded minterms*, or EMT's. When a minterm is covered by a selected cube, it becomes a "don't-care" minterm for the remainder of the minimization. Creation

of these new "don't-cares" may create new dominance relations among cubes in the PCPI. In these cases, the dominated cubes are removed, thereby introducing the possibility of new pseudo-essential implicants.

The effectiveness of DSA primarily depends on the early determination of essential and pseudo-essential cubes. This in turn depends on steps 1 and 4, which both involve the heuristic selection of minterms for expansion. The prime implicant set generation method involves the use of *required adjacency directions*, or RAD's, to guide the expansion in a depth-first search for new prime implicants [McK74, DM88]. A RAD of a minterm is defined to be a non-zero vertex at a Hamming distance of one from that minterm. Thus, in an n variable Boolean function, a minterm may have up to n RAD's. It is suggested that these RAD's be used to order the minterms, so that those with fewer RAD's are chosen for expansion first [RNMP77]. Also, it is observed that when choosing a recursive minterm, it is best to choose one covered by only one cube in PCPI, when possible. Together, these two *ordering heuristics* help to raise the probability that an essential or pseudo-essential cube is found in step 3.

Figure 3.1 illustrates a DSA minimization of a partially specified four variable Boolean function `f(a,b,c,d)`, in Karnaugh map form. Cubes discovered in minterm expansions are shown outlined, and cubes found to be dominated have dotted outlines. The first selected NMT chosen for expansion is `0000` in this example. Expansion results in the discovery of three prime cubes {`XX00 X00X 0XXX`} and six RMT's {`0001 0010 0011 1000 1001 1100`}, in Figure 3.1(a). Since more than one cube covers `0000`, the three cubes are put in the PCPI. In Figure 3.1(b), selection and expansion of RMT `0011` leads to the discovery of two new prime cubes (`0XXX` has already been generated), and two new RMT's. In Figure 3.1(c), only one cube is found to cover RMT `0010`; thus this cube is immediately selected, and its covered minterms changed to "don't-cares". In Figure 3.1(d), RMT `1100` is expanded, yielding a new cube `X1X0`,

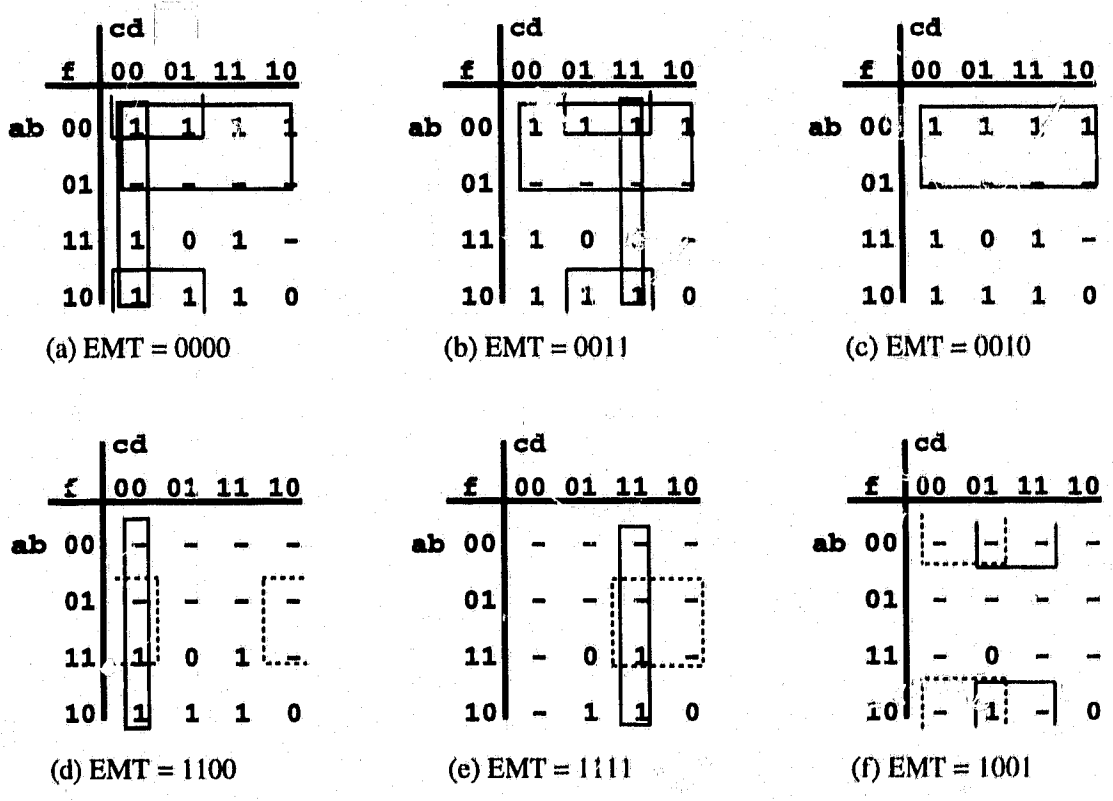


Figure 3.1: DSA Minimization of a Four Variable Function

but it turns out to be dominated by XX00, so the new cube is removed. Then, only XX00 covers EMT 1100, so the implicant is pseudo-essential, and therefore selected. A similar thing happens with minterm 1111 in Figure 3.1(e). Finally, the algorithm halts with the selection of cube X0X1 to cover EMT 1001, in Figure 3.1(f).

RAD's are compatible vertices adjacent to a minterm. For example, in Figure 3.1, minterm 1111 has three RAD's: 0111, 1110, and 1011. In Figure 3.1(b), RMT 0011 is a much better choice for expansion than RMT 0001, because the latter RMT is already known to be covered by more than cube in the PCPI, and will therefore certainly not be found to be essential.

3.3 A Lower Bound on Iterations

Let t be the number of cubes in the minimum cover of a Boolean function F, and let i be the number of DSA iterations required to find the cover. It is an interesting aspect of DSA that for non-cyclic functions[1], if i minterms are chosen for expansion in an appropriate order, only $i = t$ iterations are required. Certainly, *at least* t iterations are needed in any DSA minimization of F, because each selected cube corresponds to an EMT found to be pseudo-essential, and the number of EMT's in EMTS can never be greater than i. A situation in which $i = t$ is realized by ensuring that each iteration involves the generation of a PIS of cardinality one covering the selected new minterm. In this best-case scenario, no recursive minterms are ever generated. The required iteration order is attainable for *any* non-cyclic F, by taking the covering cubes found in any DSA minimization of F, in the order in which they were discovered, and using their corresponding pseudo-essential minterms, in the same order, for the hypothetical

[1]Non-cyclic in the sense defined by the DSA algorithm, *i.e.* a function whose minimization does not reach step 7 in the algorithm given on page 22.

DSA minimization run. The lower bound of t for i, is therefore "sharp".

Thus, the effectiveness of minterm selection ordering heuristics can be measured by comparing the number of DSA iterations required to the number of cubes eventually found for the cover. In addition, the maximum and average size of the generated prime implicant sets indicate the complexity of the iterations. The number of generated RMT's provides another measure of minimization quality, since in the best case, none need be generated. (In the best-case scenario, it is assumed that the DSA minimizer is smart enough to avoid RMT generation when a new covering cube P is found to be pseudo-essential for its generating minterm. All minterms covered by P are instead moved directly to the "don't-care" class. Actually, this straightforward optimization is not mentioned in McKinney's work, though it is subsequently noted by Dueck [DM88].)

A similar lower bound can be obtained for cyclic functions. In this case, $i_{best} = t' + e$, where t' is the number of pseudo-essential cubes in the cover, and e is the number of EMT's in the DSA cycle. The term t' is a sharp lower bound for the number of iterations required to find the deterministic part of the solution, using the same argument as above. The term e is also sharp, by constructing a scenario in which the EMT's eventually found to be in the cyclic EMTS, are all selected for expansion *after* the t' pseudo-essential primes are discovered in the first t' iterations. No extra iterations could possibly be needed, since no minterms are ever changed to "don't-care" while the cycle is in the process of being isolated.

Note that the number h of heuristic cube selections required to "break" the cycle (step 7 of the algorithm on page 22) is not included in the calculation of i_{best}. This number h is best considered separately from the iterations required to output pseudo-essentials and isolate the cycle, because h is a function of which cycle-breaking heuristic is used, and the complexity of the cycle. Moreover, the handling of cycles

is really independent of the rest of DSA, relying on different heuristics and solution techniques. The number h is typically much smaller than e above, and is sometimes called the *nesting depth* of the cycle [DAR86].

3.4 Directed Search Variations

The application of DSA to multiple-output problems is first considered by Serra, who suggests heuristic approaches for both the ordering of minterms and the generation of prime implicants [Ser84a, Ser84b]. The heuristics guarantee faster execution, but not minimum results, even for non-cyclic functions. The algorithm treats minterms corresponding to identical input variable assignments but differing functions, as a kind of multi-minterm. This restricts the class of prime implicants that are generated, and some essential and pseudo-essential primes are left undiscovered.

Dueck broadened the application of DSA to encompass multiple-valued two-level minimization [DM88, Due88]. The multiple-output Boolean input problem is considered simply a special case of the multi-valued problem. Thus, Dueck's algorithm is far more general than the original DSA. The main addition to McKinney's algorithm, besides extending it to multiple-valued variables, is the addition of a different ordering heuristic, used to choose both new minterms, and recursive minterms, for expansion. (Note that several such ordering heuristics are suggested by Serra [Ser84b].) Dueck's ordering heuristic consists of two positive integers, together called the *expandable adjacency vector weight*, or EAVW. The first indicates the number of (multiple-valued) variables that have at least one compatible adjacency to the corresponding minterm. The second number is the sum over all actual values that each (multiple-valued) variable can assume and still be compatible. The first number has higher priority when selecting minterms for expansion.

A remarkable point about Dueck's algorithm is that it is a relatively straightforward extension of McKinney's original DSA (aside from the minterm ordering techniques, which are absent in McKinney's original description). In [DM88], the definitions of minterms, cubes, and prime implicants are all extended to multi-valued logic. In the multiple-output case, a minterm is specified not only by the phases of all input variables, but also by the function in which it occurs. In this way, the m output functions are considered an m-valued variable: a transformation proven valid by Sasao [Sas78].

Another version of DSA is reported in [PR88]. Like Serra's work, the algorithm, called EDSA for *extended DSA*, is intended for multiple-output minimization of Boolean functions. The multiple-outputs are treated as special cases; "minterms" (actually, an assignment to the input variables) are augmented with *tags* indicating the functions to which the assignment applies. The primary ordering heuristic is based on small numbers of one bits in tags. Ties are broken by considering counts similar to the RAD's of McKinney. Ordering is not based on the number of cubes covering a minterm, apparently.

All minterms are first ordered, then stored in the *parameter table.* The order changes as minimization proceeds, requiring adjustments to this table. Trees of primes are generated for several minterms simultaneously, corresponding to one assignment to each of the n inputs, and all m outputs. Like DSA, many complicated rules exist for pruning the expansion tree, based on containment relations among branches. Like Serra's DSA2, minimum output is not guaranteed, even for non-cyclic functions.

3.5 Strengths and Limitations of DSA

The primary strength of DSA and variants is that a cover is extracted without generating all the prime implicants. Moreover, the McKinney and Dueck variants are capable of exactly solving a large class of functions. When exact solutions cannot be generated, the algorithms output as many cubes as they are capable of determining as essential or pseudo-essential. The cyclic parts are isolated so that heuristic selections of covering cubes are made with as much relevant information, and as little irrelevant information, as possible.

A main drawback for DSA and all variants is their requirement for a list of true minterms as input. The method for generating prime implicants is quite awkward, requiring many checks amongst depth-first search tree branches to ensure that prime implicants are not generated more than once [Due88]. Also, it is possible for dominated cubes to be regenerated, then re-removed, in the same minimization. The calculation of RAD's for each minterm, then sorting all the minterms based on these values, requires much time and memory just for getting started on a minimization. And even with the RAD information, the number of iterations can stray far from the lower bound because of cycles containing many minterms.

Both McKinney's and Dueck's directed search algorithms can be described as the processing of *partial* prime implicant tables, where table reductions are applied as the table is constructed. (This interpretation is explained in more detail in Chapter 8.) Both algorithms are also claimed to be exact in the absence of cycles. However, this is not strictly true. Like some other "quasi-exact" algorithms such as McBOOLE [DAR86], no provision is made for the removal of redundant minterms. Thus, isolated cyclic parts are not as small as they could be. It is even possible for DSA to report non-cyclic functions as cyclic, erroneously. Also, where real or imagined cycles are

found to exist, the algorithm is no longer exact.

3.6 Methodology of this Research

Having identified the weaknesses of DSA, the approach chosen in this research is to take each one, find a better way to do it, insert the new techniques back into the DSA algorithmic "framework", then see if it is now a viable competitor to other minimizers. Just about every aspect of DSA is open to improvement. They include:

- Minterm selection
- Cube dominance relations
- Search methods in data structures
- Prime implicant generation
- Minterm dominance relations
- Solution of cycles
- Memory requirements

In fact, the above list is somewhat contrived — it roughly corresponds to the main subjects of the next seven chapters. However, the items do go a long way toward addressing the drawbacks of DSA, so that their solution results in an improved minimizer comparing favourably with existing ones, both heuristic and exact. The remaining chapters describe these solutions, and their efficacy.

Chapter 4

A New Implementation of Directed Search

4.1 Introduction

This chapter describes the unenhanced version of directed search, used as an initial "programming template" for algorithms and techniques reported in subsequent chapters. Experiments and results for rules and heuristics used to select minterms are also presented here, since they are basic to the philosophy of "directed search". Both the minimizer and the ideas it incorporates are henceforth known as MDSA, for *modified directed search algorithm* (or even *Mike's directed search algorithm*). The minimizer is written in the C programming language.

4.2 Data Structures

A key property of DSA is that it is *minterm based* rather than *cube based.* As such, the algorithm requires some representation for the entire ON-set — one that permits the easy extraction of individual members. A simple list of minterms is used in [McK74]. Because the number of minterms is of the same order as the entire output space of a function, and because directly addressable entries are much easier to retrieve, a map-based representation is chosen for MDSA.

The research is directed primarily to the multi-output Boolean function case, where input PLA's have n Boolean input variables, and m output functions. Each point in the space of a function assumes one of the three values in {0,1,X} . Thus, at least two bits are required to represent each vertex. The strategy used in MDSA is to reserve two 32-bit words for each assignment to the n input variables. Then, every pair of bits in the two words, one bit from each word, corresponds to an output line. Thus, m is limited to 32. However, bit operations on all m functions can be carried out in parallel.

In addition to the truth value of each vertex, MDSA requires the *cover count* for true vertices, indicating how many cubes in the PCPI cover each. These cover counts are central to the algorithm, and therefore require fast accessing times. Again, an exhaustive array of size exponential in the number of input variables n, appears to be the easiest solution. The array requires $m2^n$ easily addressable entries. Eight bits per entry permit only $2^8 - 1 = 255$ covering cubes, which is insufficient for many PLA's otherwise minimizable. Sixteen bits per entry allow cover counts as high as $2^{16} - 1 = 65535$, which is easily enough for all minimizations reported here. The largest benchmark PLA actually minimized in the present research is `in2`, with

$n = 19$ and $m = 10$. Thus, the map requires

$$(32 * 2)/8 * 2^{19} = 4194304$$

bytes, and the counts array requires

$$2 * 10 * 2^{19} = 10485760$$

for a grand total of 14680064 bytes (or 14 Mbytes). This taxes the available computing resources to near the limit.

The cube sets PCPI and PIS, and the minterm sets RMTS and EMTS, are stored as linked lists of record structures. Improved data structures are presented in Chapter 6.

4.3 Ordering Heuristics for Minterms

The selection of minterms for expansion controls the number of iterations that a directed search minimization eventually requires. Thus, it is extremely important to make sensible selections. In MDSA, the ordering heuristic is initially approximated by a single positive integer, called the *adjacency degree* of the minterm. It is equal to the number of non-zero adjacencies at Hamming distance one from the minterm, plus the number of functions in which the same input variable assignment is compatible. Empirically, this number works better on average than Dueck's EAVW ordered pair of numbers [DM88], though the difference is not great.

4.4 Improved Ordering using Covering Cubes

The directed search framework actually offers more information about the probability of finding essential cubes, than the adjacency degree of selected minterms alone. In

MDSA, the initial creation of an RMT happens when a newly generated prime cube is found to cover a previously uncovered NMT. Thus, at the time of RMT creation, a covering prime implicant is available. It is convenient to define a cube's *degree* as the number of unspecified variables in the cube, plus the number of functions in which that cube is compatible. The similarity in definition between the degree of cubes and the adjacency degree of minterms is no accident. When the degree of a cube is equal to the degree of any one of its contained minterms, that cube must be both prime and essential for that minterm.

Using similar reasoning, the probability of finding that a cube is *not* essential for a minterm is proportional to the degree of the minterm, minus the degree of the cube. Thus, it is possible to define a new ordering heuristic for RMT's, equal to this non-negative difference, called the *order* of that minterm. RMT's are first generated when a cube is found to cover a minterm that has a cover count of zero. By retaining the degree of the cube as new RMT's are found, these minterms' orders may be found immediately, and stored as part of their record. Unlike degree, which is a static quality, the order of a minterm varies, depending on which cube of that minterm's PIS happens to be generated first.

4.5 Sorting Minterms

Presorting the minterms by ordering heuristic requires yet another array of exponential size, where *each entry* must be large enough to address a multi-function minterm. However, presorting only benefits the selection of NMT's, and experiments show that this happens infrequently compared to RMT selection. Furthermore, the rapid selection of smaller cubes arising from choosing NMT's of small degree does little to speed up the overall minimization. In MDSA the degrees of RMT's are calculated only at

the time of their initial generation. Thus, the calculation of degree for *every* minterm is avoided, since only a fraction of all minterms initially in NMT ever make it to the RMT set. Most are first covered by selected cubes, or removed using the techniques of a subsequent subsection.

4.6 Potential Adjacencies

Adjacency calculations for minterms are not cheap. In the data structure chosen to represent multiple-output functions in MDSA, n probes are required — one for each input variable. (Adjacencies in the m output lines are computed in parallel.) The directly-addressable data structure permits constant-time probing, but so many minterms require the adjacency calculations that any effort expended in their simplification is worthwhile.

In MDSA, RMT's are detected by covering cubes. Thus, when calculating the order of such minterms, only those adjacencies lying *outside* the cube need be tested for compatibility. These outside adjacencies are called *potential adjacencies* since they are the only ones that can possibly be incompatible. All adjacencies within the cube are implicitly known already. This provides a rationale for adding cubes of each PIS to the PCPI, in descending order of cube degree.

Potential adjacencies also apply to the m output functions. These adjacencies in the m-valued variable are computed in parallel. One probe of the map entry corresponding to the minterm's n input variables, suffices. With the m-valued variable removed, if the degree of the covering prime cube is d, then d adjacencies are implied. An additional $n - d$ probes are required to find the remaining adjacencies.

4.7 Known Essential and Inessential Minterms

If a new RMT is found to have zero potential adjacencies, all of its adjacencies are contained in the prime cube responsible for the recursive minterm's discovery. The minterm is therefore known to be essential, as is its covering prime cube. Thus, it is possible to immediately select the covering cube for the cover. MDSA does this, immediately turning all covered minterms to "don't-cares" in the map data structure, and entering the search for dominated cubes (described in Chapter 5) early. Adjacency calculations and other overhead involved with the rest of the newly-covered minterms, is avoided.

Similarly, if a given minterm has one or more adjacencies that are not to "don't-care" minterms, then the minterm is known *inessential*, and in fact non-pseudo-essential. MDSA assigns such minterms a lower priority for selection, by increasing their ordering heuristic by $n + m$. This ensures that all RMT's not belonging to this class, are of higher priority, since minterm orders higher than $m + n$ are not possible. Note that minterms classified as inessential may subsequently become pseudo-essential in the future of the same minimization, as more of their adjacent minterms become "don't-care" through cube selection. However, it is far too expensive to keep this information up-to-date, given the typically large size of the RMTS.

4.8 Results

Table 4.1 shows the effectiveness of several ordering strategies for minterm selection. The table also introduces the benchmark set of PLA's used throughout the research to gather empirical data. Non-cyclic PLA's occupy the topmost sections of all tables in this section. These PLA's are all completely solvable (coverable)

	PLA			none		Dueck		degree		order	
name	n	m	c	iter	time	iter	time	iter	time	iter	time
alu1	12	8	19	25	0.7	32	1.0	27	0.9	30	0.8
alu2	10	8	68	162	0.8	107	0.6	97	0.5	101	0.4
alu3	10	8	65	377	1.4	119	0.5	114	0.4	111	0.3
add4	8	5	75	116	0.1	84	0.1	84	0.1	86	0.1
add6	12	7	355	724	6.5	418	2.4	401	2.2	395	2.1
add7	14	8	735	1679	78.0	735	5.0	753	5.0	805	4.7
in1	16	17	104	188	10.8	148	12.1	149	11.8	148	7.7
dk48	15	17	22	35	64.0	31	3.5	32	3.4	32	2.4
clip	9	5	117	681	1.30	320	0.5	317	0.5	325	0.5
*dist	8	5	121	263	0.3	244	0.2	244	0.2	240	0.2
*mult4	8	8	126	549	0.6	525	0.6	525	0.6	526	0.6
*in0	15	11	107	211	3.9	182	4.8	182	4.3	181	3.3
*sym9	9	1	88	489	3.7	502	4.2	502	4.3	502	3.8
*apex4	9	19	430	1636	3.9	1511	4.1	1477	4.4	1480	3.0
*sev	8	10	209	1024	2.3	1001	2.5	1000	2.6	1002	2.7
*tial	14	8	577	4496	88.0	1711	20.0	1692	19.2	1681	13.6
*sym10	10	1	211	868	20.1	834	17.5	834	17.5	834	20.6
*misex3	14	14	708	13015	177	11969	246	11851	244	12101	248

Table 4.1: Comparison of Ordering Strategies

through the repeated selection of essential and pseudo-essential primes. The cyclic PLA's, occupying the lower sections of tables and preceded by the "*" symbol, contain non-empty subproblems that are not solvable through deterministic selection of pseudo-essential cubes (though in many of them, these subproblems are of near-trivial size). The PLA `clip` is an interesting special case. It is "borderline" cyclic, because DSA (and the MDSA prototype introduced in this chapter) consider it cyclic, though in fact it is not. This phenomenon, and methods for addressing it, are discussed at length in Chapter 8.

The leftmost section of Table 4.1 gives the name, number of inputs n, number of outputs m, and the number of cubes c in the cover found by MDSA, for each PLA. Each of the remaining four sections gives the number of main iterations (*i.e.* minterm selection/expansions) and total time required, in the minimization of each PLA. Experiments are all carried out on a SUN SPARC1 computer, and times are measured in seconds. Four separate experiments are reported. In the first ("none" section), minterms are ordered according to cover count, but no adjacency information. Thus,

PLA				DSA	no essentials			essentials		
name	n	m	c	bound	iterations	minterms	time	iterations	minterms	time
alu1	12	8	19	19	30	11313	0.8	20	10167	0.7
alu2	10	8	68	68	101	4844	0.4	85	4584	0.4
alu3	10	8	65	65	209	5099	0.6	201	2795	0.5
add4	8	5	75	75	86	283	0.1	84	265	0.0
add6	12	7	355	355	395	9524	2.1	390	7364	2.0
add7	14	8	735	735	805	27432	4.7	800	25944	5.0
in1	16	17	104	101	148	238312	7.7	124	152400	5.7
dk48	15	17	22	22	31	57407	2.6	25	41003	2.0
clip	9	5	117	234	325	1262	0.5	315	1192	0.4
*dist	8	5	121	183	240	572	0.2	226	524	0.2
*mult4	8	8	126	490	526	615	0.6	516	603	0.6
*in0	15	11	107	110	181	66107	3.3	142	29857	2.5
*sym9	9	1	88	420	502	420	3.8	502	420	3.8
*apex4	9	19	430	1242	1480	2727	3.2	1424	2168	3.2
*sev	8	10	209	927	1002	1220	2.7	985	1187	2.6
*tial	14	8	577	1217	1681	59218	14.4	1461	45103	11.1
*sym10	10	1	211	792	834	792	20.6	834	792	17.5
*misex3	14	14	708	13963	14334	23002	270	11853	21126	244
*misex3c	14	14	204	14874	17143	23192	15860	17098	21659	15727
*in2	19	10	138	61929				64516	283263	16400
*misex3c	14	14	204		8865	23192	3890	8828	21659	2810

Table 4.2: Early Recognition of Essential Cubes

minterms covered by only one member of PCPI are selected first, but it does not matter which one. In the second section ("Dueck"), the ordering heuristic described in [DM88] is used to order within the same cover count. In the third ("degree"), minterm degree is used, and in the fourth ("order"), minterm order is used.

The data of Table 4.1 indicates that there is significant improvement in the number of iterations for many PLA's when adjacency information is used in conjunction with cover counts. The ratio is sometimes two-to-one or more, as in the cases of **add7** and **tial**. On the other hand, the difference in the number of iterations between the three rightmost sections, corresponding to three measures of adjacency, is negligible. However, the times are generally better in the "order" section. This is because fewer adjacency calculations are needed, due to the use of potential adjacencies. The improvement is as high as one-third for several PLA's, including **dk48** and **apex4**.

The effectiveness of the recognition of essential cubes in MDSA is shown by the

data in Table 4.2. Two more PLA's are included now, `misex3c` and `in2`, since the use of the essential cube technique enables their solution in "reasonable" time (in this section, "reasonable" means less than five hours). The two rightmost main sections of the table, "no essentials" and "essentials", correspond to experiments in which MDSA is used without essential recognition, then with essential recognition. The sections include columns for the number of minterm records generated in each minimization run, since this is another characteristic improved by the technique.

Table 4.2 also includes a column "DSA bound" giving the low bound on the number of DSA iterations required for each PLA. In Chapter 3 it is proven that this low bound is sharp, and is equal to the number of essential and pseudo-essential cubes in a minimum cover, plus the number of minterms involved in the subject PLA's cycle. This formula simplifies to the number of cubes in a minimum cover, for non-cyclic PLA's. The same bound applies to the MDSA prototype without essential recognition, and therefore provides a means for measuring the efficacy of various directed search techniques. In most cases, the number of iterations is well within a factor of two of the bound, with `alu3` being the most notable exception.

The number of iterations required in an MDSA minimization generally drops when using essential recognition, as do the number of minterms generated. The effect is minor for many PLA's, but significant for others, including `in0`, `in1`, and `in2`. These PLA's are all quite large, and evidently contain many essential primes. The PLA `in2` is not even solvable by MDSA without essential cube recognition, in five hours or less. Note that in the case of `misex3`, the number of iterations required is actually *less* than the "low bound". This is because it is possible that several essential minterms, and therefore several essential cubes, are discovered in one iteration. The low bound only applies to the version of MDSA without the essential recognition optimization technique.

PLA				MDSA			espresso			esp-exact		
name	n	m	p	cubes	mem	time	cubes	mem	time	cubes	mem	time
alu1	12	8	19	19	460	0.8	19	332	*0.1	19	536	12.4
alu2	10	8	68	68	356	*0.4	68	412	1.5	68	760	17.6
alu3	10	8	66	*65	352	*0.5	66	408	1.1	65	832	19.4
add4	8	5	75	75	296	*0.0	75	412	0.8	75	552	2.3
add6	12	7	355	355	528	*2.0	355	448	16.5	355	1452	577
add7	14	8	735	735	776	*4.5	735	576	71.0	735	5264	7000
in1	16	17	105	104	336	*5.7	104	460	10.3	104	624	40.0
dk48	15	17	22	22	1612	1.9	22	440	*0.1	22	500	95.0
clip	9	5	167	*117	340	*0.4	120	440	6.4	117	712	13.2
*dist	8	5	121	121	344	*0.2	121	504	4.2	*120	492	4.0
*mult4	8	8	225	*124	336	*0.6	128	452	7.6	*121	1196	65.0
*in0	15	11	107	107	1440	*2.4	107	444	4.0	107	484	13.5
*sym9	9	1	84	85	408	4.2	*84	488	*1.1	*84	6164	155
*apex4	9	19	438	*430	456	*3.0	436	524	102	*427	848	93.0
*sev	8	10	206	209	376	*2.6	*204	528	11.3			
*tial	14	8	586	*578	1088	*11.0	581	568	91.0	*575	1720	758
*sym10	10	1	1024	210	540	*22.3	210	512	24.7	210	27000	1280
*misex3	14	14	1848	703	1256	264	*690	704	*177			
*misex3c	14	14	305*	204	1344	2810	*197	580	*55.2			
*in2	19	10	137	138	17000	16400	*134	528	*3.5	*134	592	*12.2

Table 4.3: MDSA Prototype versus Espresso and Espresso-exact

The PLA `misex3c` is a very difficult one for MDSA to solve. The final row of results in Table 4.2 is a repeat one for this PLA, where the maximum number of cubes permitted to be in the PCPI is restricted to 5000. At this point, MDSA assumes that the PCPI is cyclic, and accordingly chooses a covering cube heuristically. Without this restriction, the lowbound of 14874 is far too large for a DSA minimization to be carried out in reasonable time. Even with the limit imposed on PCPI, the times are not good, though much improved. The number 5000 is rather arbitrarily chosen, and this limiting factor only comes into play for `misex3c`, among all the benchmarks. Hereafter, until Chapter 8, results for `misex3c` reflect use of the limit.

Table 4.3 compares the prototype version of MDSA, complete with minterm ordering and recognition of essential cubes, with *espresso* [BHMSV85] and *espresso-exact* [RSV87]. These latter programs are considered state-of-the-art minimizers, and enjoy widespread use in academe and industry. In addition, they both undergo periodic im-

provement. Therefore, they provide a good base measure for minimization methods. In the table, results are given for cubes output, memory use (measured in kilobytes), and times (in seconds). For MDSA and *espresso* sections, asterisk characters ("*") indicate where measurements are superior to corresponding measurements by the other minimizer, for time and cubes. Where similar asterisks appear in the *espresso-exact* section, they indicate better results compared to MDSA.

In the first section, "PLA", of Table 4.3, the column p gives the number of cubes given in the input specification of the PLA's. The entry for `misex3c`, 305, is different than the others in that 108 "don't-care" cubes are included. The other 197 cubes belong to the ON-set. The p values are very significant, because it is not possible for *espresso* or *espresso-exact* to return covers that are worse (larger) than these values. On the other hand, MDSA simply turns all cubes into the appropriate map entries, then discards them.

Considering first the top half of Table 4.3, MDSA never fares worse than *espresso* for number of cubes output. This is to be expected, as MDSA *must* obtain the minimum solution for non-cyclic functions. MDSA finds minimum solutions for `alu3` and `clip`, where *espresso* does not. Naturally, cubes data is identical for MDSA and *espresso-exact*. MDSA is always faster than *espresso*, except for `alu1` and `dk48`. But all the times for minimization of non-cyclic PLA's by MDSA, are better than *espresso-exact*. This is notable, because MDSA, like *espresso-exact*, is able to guarantee that the output is indeed minimum. However, it must be noted that MDSA is capable of minimizing only those PLA's that will fit in a map-based representation.

Turning now to the lower half of Table 4.3, MDSA is no longer so far ahead of its competitors. The PLA's are all cyclic, which means that MDSA can no longer guarantee minimum cube covers. In fact, it fails to find the minimum cover for nine of the eleven PLA's. (*Espresso-exact* is not capable of solving `sev`, `misex3` or

misex3c, but *espresso* returns better covers than MDSA for these PLA's. It follows that the MDSA results are not minimum.) In three instances, MDSA is better than *espresso*, compared to five in which *espresso* is better. Apparently, MDSA requires some improvement before being competitive with cyclic PLA's.

The times results for MDSA minimizations compare better than the cube results. MDSA is faster than *espresso* for seven of the eleven cyclic benchmarks. This is especially true for apex4 and tial, both PLA's with extremely large covers. Evidently, this characteristic affects *espresso* adversely. The PLA misex3 also has many cubes in the output cover. The reason why MDSA is not ahead of *espresso* for this problem is revealed by the iterations figures in Table 4.2; an exceptional number of iterations is required. *Espresso* is far ahead of MDSA for misex3c and in2. The time ratios are substantial: about 50 and 4500 respectively. Certainly, the number of iterations that misex3c requires is one limiting factor, and in2 is an extremely large PLA, stretching memory use to near the limit.

4.9 Summary

This chapter introduces the minimizer MDSA. It is based on the multi-valued minimizer DSA-MV [DM88], but can handle larger PLA's with up to 20 Boolean input variables and 32 Boolean output functions. The minimizer is written in the C programming language, and uses exhaustive arrays to hold vertex values, and covering counts. Linked lists are used for the other main data structures. The requirement for memory space exponential in the number of input variables, limits the problems amenable to solution.

Minterm adjacency information aids the selection of minterms for expansion. However, different strategies for arriving at an ordering number have little effect. The

use of covering cube dimensions and potential adjacencies permits faster adjacency count calculations. A side-effect of these calculations is early recognition of essential minterms and their corresponding cubes, plus the identification of minterms that are provably inessential.

The MDSA prototype compares favourably with *espresso* for the non-cyclic PLA's in the benchmark set. However, performance is not competitive with larger PLA's, and cyclic PLA's requiring many minterm expansions. Thus, it appears that methods for controlling either the number of iterations, or the amount of computation per iteration, are needed.

Chapter 5

Efficient Cube Cell Strategies

5.1 Introduction

Some minimizers are capable of recognizing essential cubes quite easily, as a side-effect of calculations required. For example, Chapter 4 shows how essential minterms are recognized in MDSA. However, DSA and MDSA differ from many other minimizers because the former are capable of recognizing *pseudo-essential* cubes as well. This is due to the introduction of *cube dominance* to these minimization algorithms, in particular ensuring that these relations do not exist among cubes in the potential prime implicant set. Cube dominance depends, in turn, on the concept of cube *cells*. This chapter describes how cells are calculated, maintained and used in MDSA.

In MDSA, the idea of *minterm cells* is borrowed from McKinney's work [McK74]. (There, they are called *trueform cells*.) The minterm cell (or just *cell*) of a cube is the smallest subcube containing all the cube's minterms. With the notion of cells, dominance is readily defined: a cube A dominates a cube B if and only if the cell of B is a subcube of the cell (or equivalently, cube) of A.

5.2 Cells and Cell Classifications

The two cube sets involved in MDSA minimizations are the PIS and PCPI. The former are generated for each iteration, and the latter grows and shrinks throughout a minimization. Every cube (which is a prime implicant) maintains a corresponding cell as part of its record. Storing the cell information does not require another complete cube record, since the cell is always a subcube of its parent prime implicant. In MDSA, a cube is given by a bit-vector describing any contained minterm, and another bit-vector indicating "don't-care" variables. Since any minterm will do, one contained in the cell is used. Thus, one additional bit-vector indicating "don't-care" variables in the cell cube is all that is required.

Cells of cubes are initially generated with the prime cubes themselves, at the time of prime creation in the prime implicant generator. Thereafter, cells may require adjustment when selected cubes are removed from the PCPI. They always either remain the same size, or grow smaller, as their contained true minterms change to "don't-cares". A dominance relation between two cubes can exist only if they *intersect* (*i.e.* their product is non-empty). Thus, it is important to know when these intersections might arise.

In MDSA, the cells of cubes, rather than the cubes themselves, are used for intersection calculations. The number of required cell adjustments is reduced, because the chance of intersection is reduced. This is significant, because cells must be regenerated any time an intersection between a PCPI cube and a selected cube is non-empty. If an intersection (product) of cubes is non-empty, but the intersection (product) of their corresponding cells *is* empty, then the intersection of cubes must be entirely "don't-care". In such cases, none of the involved cubes' cells require any adjustment.

In MDSA, when a cube C is selected for the cover, each cube $P \in$ PCPI implicitly

belongs to one or more of the following five categories with respect to C:

- UNAFFECTED: The cells of P and C do not intersect. This does *not* imply that their cubes do not intersect, but if they do, that intersection does not contain any minterms.
- INTERSECTED: The cells P and C intersect. The cube P is also called a *candidate* cube.
- AFFECTED: The intersection of P and C contains at least one minterm.
- CONTRACTED: The cell of the intersected cube P grows smaller, after recalculation following the removal of the selected cube C from the ON-SET.
- DOMINATED: The contracted cell is a subcube of some other PCPI cube Q's cell. Thus, Q dominates P.

The UNAFFECTED and INTERSECTED sets are mutually disjoint, and together comprise the entire PCPI. The set of intersected cells contains the set of affected cells, which in turn contains the set of contracted cells, in turn containing the set of dominated cells. The classification scheme is used to separate out the cells which require adjustment, and then possibly dominance tests, from the other cells, thereby minimizing these expensive calculations.

When a cube C is selected for the cover, its contained minterms are all set "don't-care" in the map. Then, the PCPI is scanned to separate the unaffected cubes' cells from the intersected ones. Every prime cube P with an intersected cell might possibly be contracted. A necessary condition is that P is affected, but MDSA tests immediately for the contracted condition by recalculating P's cell, as described in Section 5.3 below. If P's cell is indeed contracted, it might also be dominated by

some other member of PCPI. A search for a dominator is performed for each such prime with a contracted cell.

5.3 Fast Cell Confirmation

A naive way to find the cell of a cube is to initialize it with a null cell, and then examine each vertex contained in the cube, using it to update the cell when the vertex happens to be a minterm rather than a "don't-care". A straightforward improvement to this approach is examination of every vertex in the *cell* of the cube, rather than the entire cube itself. This is possible because cells decrease in dimension monotonically throughout a DSA minimization. Every time a cell grows smaller, it is a proper subcube of the former cell for the same cube. When a prime implicant is first generated, its cell is initialized to be the implicant itself.

In MDSA, an efficient method is used for updating affected cells. Instead of probing *all* vertices, the probing starts at one vertex of the cell, then moves to the one "diametrically opposite" (*i.e.* at maximum Hamming distance) in the same cell. The next check (which is simply a probe of the truth table) moves back to the first vertex adjusted by one adjacency, *etc.* For example, the checking order for implicant `XXXOX` with cell `1XXOX` is as follows:

10000, 11101, 10001, 11100, 10100, 11001, 10101, 11000

Variables v_3, v_2, and v_0 are called *counting variables*; they are the ones that change in the probing order. The number of counting variables is equal to the degree of the cell. Variables v_4 and v_1 are non-counting; they must remain the same for all probes, or else the probe would be outside of the cell.

The probing sequence is an interleaved lexicographic ordering, one incrementing and the other decrementing, with each number at Hamming distance k and $k-1$ from its immediate neighbours, where k is the number of counting variables. The idea is to gain a chance to break out of the cell generator early, because it most often happens that the cell of a cube is not changed. Thus, when a cell must be found, the procedure can halt when the new cell grows as large as the old. A minimum of two minterms need to be checked with this method, for any size cube. The upper bound is still the number of vertices contained in the former cell, however.

5.4 Cube Differences and Multi-Cells

A cell requires re-calculation when it is found to intersect with a selected cube. By definition, no minterms exist in the intersection of the two cubes, because all vertices of selected cubes become "don't-care". Thus, it is wasteful to re-test this intersection, especially when the intersection accounts for a large proportion of the intersected cell.

It is convenient to identify two special cases of intersection: (1) the intersection is *equal* to the intersected cell (it can never be larger), and (2) the intersection covers exactly *half* of the intersected cell. (In the latter case, the covered subcube of the intersected cell contains precisely one more specified variable than the intersected cell.) The two cases are the only ones in which the remaining uncovered part of the intersected cell is known to be a cube. Thus, the techniques of the previous section can be applied to this cube to find its cell. (Actually, the first case is trivial because the cube whose cell must be found is the empty cube. Thus, its cell is the empty cell.)

The more general case is when the intersection covers one quarter, one eighth, one sixteenth, *etc.* of the intersected cell. The *cell difference* is defined to be that part of

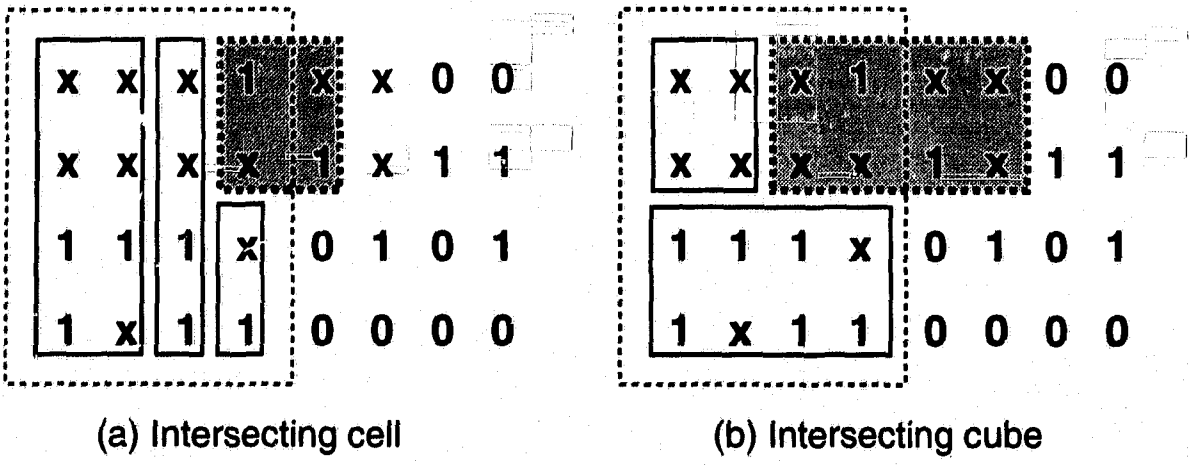


Figure 5.1: Multi-cube Cell Differences

the intersected cell not contained in the intersection. This definition is similar to the "sharp" operator used in other minimization algorithms (*e.g.* McBOOLE [DAR86]). It is still better to not retest the intersection, but it gets more complicated to find a probing order for the odd-shaped difference. Thus, it is convenient to have a cube-based representation for the cell difference. Then, the technique of the previous section is used to find the cell of each cube.

The cubes remaining in a cell after an intersection part has been removed is called a *multi-cube*. Its most convenient form is as follows: given a cell c of dimension i, with a subcube s of dimension $j \leq i$, the multi-cell difference D consists of k cubes of degrees 0 through $k-1$, where $k = j - i$. This set of cubes is disjoint, and together with the intersection cube, entirely covers the original intersected cell.

Figure 5.1 illustrates the concepts of cell intersection and multi-cube cell differences. The Karnaugh map represents a five variable subspace of a Boolean function. The intersected cell of degree four is shown by the dotted line. In (a), it is intersected

by the cell of a selected cube of degree two, shown shaded. The intersection part is of degree one. The multi-cube cell difference is then given by the $4 - 1 = 3$ cubes of degrees 3, 2, and 1. The cell of each subcube is calculated, and the logical sum of all these calculated, giving the new updated cell of degree three, occupying the lower half of the original. (The two minterms in the intersecting selected cell become "don't-cares".)

The method of finding the new cell is as follows. First, the cell difference is found. Then, the cell of each cube in the cell difference is obtained, using the fast cell technique of the previous section. Finally the new cell is the smallest cube encompassing all of these newly-found cells. Note that the lower bound for the number of probes required is now $2d$, where d is the difference in degrees between a cell and its intersected subcube.

Now that there is a method for testing only the non-intersected part of an intersected cell, it is possible to use the cube rather than the cell of the intersecting (selected) cube, in the calculation of the difference. In Figure 5.1(b), the entire selected cube rather than just its cell, is shown shaded. In this example, the difference is actually smaller. Thus, in MDSA the cells of cubes are used to see if an intersection exists, but when one does, the actual parent prime cube of the selected cell is used in the calculation of the intersection.

5.5 Results

Table 5.1 summarizes the effects of the cell calculation algorithms on the set of benchmark PLA's. As usual, the first main section gives PLA name, input variables n, output lines m, and number of cubes in the specification p. The remaining three sections give performance results for MDSA with no clever cell calculation techniques

PLA				no improvement		fast cells		cell difference	
name	n	m	p	calculations	probes	calculations	probes	calculations	probes
alu1	12	8	19	661	88176	661	48504	650	44490
alu2	10	8	68	671	61626	671	22310	671	19093
alu3	10	8	66	742	28365	742	17448	733	15745
add4	8	5	75	305	2135	305	1416	305	1068
add6	12	7	355	7364	151487	7364	81797	7364	63841
add7	14	8	735	15654	567842	15654	400383	15654	299801
in1	16	17	105	497	245276	497	185659	493	172033
dk48	15	17	22	44	229504	44	57464	44	16506
clip	9	5	167	1928	24340	1928	12452	1928	10973
*dist	8	5	121	864	5627	864	3345	864	3089
*mult4	8	8	225	1418	6155	1418	4698	1418	4394
*in0	15	11	107	609	169296	609	95027	604	82651
*sym9	9	1	84	7797	61864	7797	55055	7797	24107
*apex4	9	19	438	4481	13481	4481	12036	4459	11914
*sev	8	10	206	3438	21283	3438	14330	3438	13813
*tial	14	8	586	11531	1467898	11531	480246	11481	448473
*sym10	10	1	1024	20882	165830	20882	77624	20882	67472
*misex3	14	14	1848	44679	2073941	44679	669485	44677	631980
*misex3c	14	14	305*	105444	23398562	105444	14845891	105444	14754799
*in2	19	10	137	755	3034218	755	1129502	754	929934

Table 5.1: Comparison of Cell Calculation Techniques

("no improvement"), MDSA augmented with fast cell calculation techniques, and then further augmented with cell difference techniques. In each section, the number of cell calculations and truth-table probes throughout entire minimizations are given.

The most dramatic improvement comes from the use of the fast cell calculation algorithm. The number of required probes drops by an average of about 50% for the benchmarks, reaching as high as 75% for **dk48**. Unsurprisingly, the PLA's showing the most marked improvement are the larger ones (in terms of number of input variables), because they contain larger primes, with larger cells. The improvement is generally less dramatic when adding the use of cell difference techniques, though **dk48** is again much improved, as are **sym9** and **add7**. In the experiments, cell differences are used only when the maximum number of cells in one difference set is two. The overhead for finding the difference outweighs the benefits realized, for larger numbers.

No time figures are given in Table 5.1. However, the time required to calculate

and recalculate cells is exactly proportional to the number of probes performed, since probing a minterm is a constant time operation. Overall minimization times improve from 0% to about 50% in the case of **dk48**, averaging about 20%. The techniques do not affect memory requirements.

5.6 Summary

The identification of dominance relations among cubes, using the identical relation of cell containment, is central to directed search algorithms. Because of their frequency, it is crucial in the MDSA minimizer to keep the overall number of cell calculations down to a minimum, and to accomplish these calculations quickly when they are inevitable. This chapter shows how both objectives are met. A classification scheme ensures that cells are recalculated only for those primes that are eligible. When cell calculations do occur, an efficient probe ordering increases the probability of early termination. Disjoint cell differences ensure that redundant probing of larger known "don't-care" areas are avoided.

Chapter 6

Tree Based Sets of Cubes and Minterms

6.1 Introduction

Like most other minimizers, execution of MDSA involves the maintenance of several cube sets, which dynamically change from one main algorithm iteration to the next. The changes stem from the selection of minterms, and the dominance relations that exist and arise between cube set members. Newly-generated cubes are added to the *prime implicant set*, or PIS, of their generating minterm. Primes may then migrate to the *potentially cyclic prime implicants*, or PCPI. Dominated cubes must be identified and removed from both sets, as well as selected cubes. Also, the cells of existing set members must be kept current.

The directed search algorithm also requires the maintenance of several minterm sets. These include the set of *expanded minterms* EMTS, the set of *recursive minterms* RMTS, and the set of *new minterms* NMTS. Like the cube sets, the three minterm

sets also change dynamically from iteration to iteration. This additional requirement for maintaining minterm sets, sets DSA (and MDSA) apart from heuristic algorithms like *espresso* [BHMSV85]. However, the minterm sets are required for the ability to solve more problems exactly.

One of the main goals of directed search algorithms is to keep the internally maintained sets as small as possible, thereby preventing the time required to search through them from becoming prohibitive. However, some problem instances cause these sets to become quite large. This is especially likely for minimization problems containing large and complex cycles. For example, the EMTS and PCPI sets must grow at least as large as any contained cycle. The naive implementation for cube and minterm sets is simply a linked list. When lists are thousands of elements in length, search time becomes a dominant factor in overall minimization time.

This chapter shows how a more intelligent internal representation of the cube and minterm sets, within the minimization program, beneficially affects the program efficiency. In particular, a tree-based organization leads to a substantial improvement over simple lists, for many primitive and more complex set search operations. The improved search behaviour comes at a cost of increased set maintenance time. The new data structures are called *cube trees*. It is also possible to maintain these trees as *heaps*, thereby improving search performance when weighting is an issue.

6.2 Cube Trees

For present purposes, the most appropriate definition of a cube is a string of n characters, each from the alphabet $\{0, 1, X\}$. Each position in the string corresponds to a two-valued variable, which must take the value 0, 1, or both 0 and 1 simultaneously, indicated by the X character. The cube represents a set of vertices in the space of an

n-variable Boolean function. The rightmost position in a string representing a cube corresponds to variable v_0, and the leftmost position corresponds to variable v_{n-1}. If v_i is either 0 or 1 in a cube, it is called a *specified* variable for that cube. Otherwise, it is *unspecified*, or "don't-care".

In the MDSA minimizer, the cubes of interest are almost always prime implicants. However, the cells of these cubes are also of great interest. (The *cell* of an implicant is the smallest subcube containing all true vertices also contained in the implicant, as defined in Chapter 5.) Since most searches in MDSA involve minterms rather than "don't-care" vertices, cells provide more concise information on where these minterms lie. To put it another way, the set difference between a prime implicant and its cell consists only of "don't-care" minterms, which are not of interest. These comments are clarified when the search operations required on cube trees are defined.

Sets of cells can be organized into trees based on the values of cells' variables, and the variable positions. A *cube tree* is defined to have the following structure. Each tree node corresponds to a cell stored in the tree. Each level of a tree corresponds to a particular variable. A tree edge corresponds to a particular *value* of a particular variable. Since the relevant alphabet is $\{0, 1, X\}$, the degree of the tree is three[1]. The top (root node) of the tree is at level zero, and the (up to) three edges leading to the next level correspond to the "first" variable (v_0; the rightmost variable). In general, edges leading from level i to level $i+1$ correspond to variable v_i. This property makes cube trees quite different from other types of trees, which can be altered and rotated at will.

Figure 6.1 shows two cube trees, each representing (indeed, containing) the same set of nine cells. A particular set of cells can, in general, be represented by many different cube trees. The only required property of a node is that its cell's variables

[1]The discussion is temporarily confined to the Boolean case, for increased clarity.

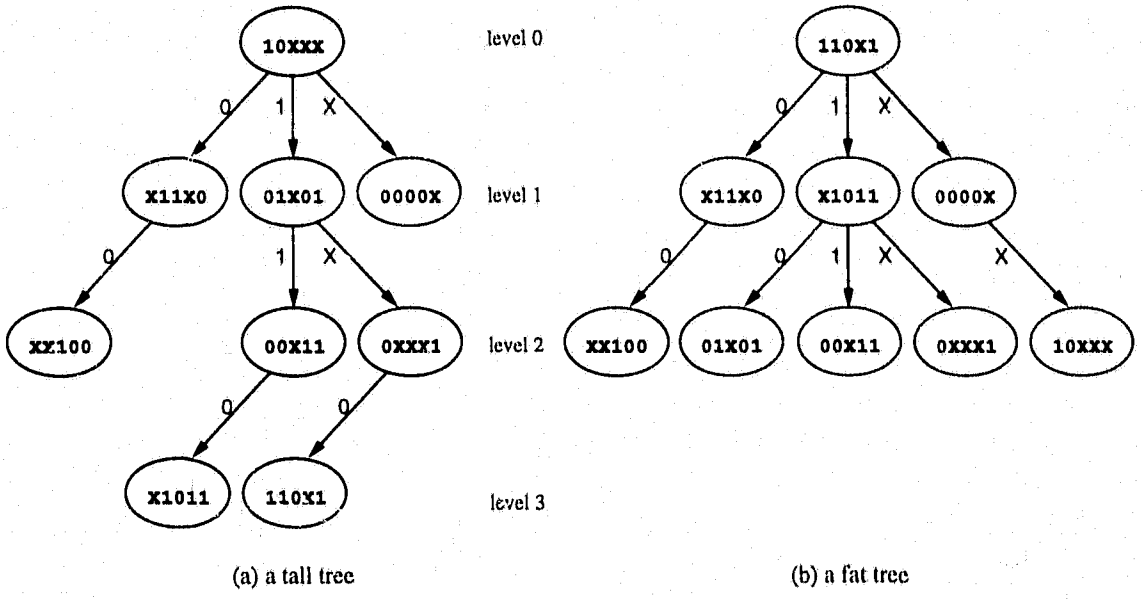


Figure 6.1: Two Cell Trees for the Same Set

use the correct edges in the path leading to that node from the root. Thus, any cell may occupy the root node (no edges lead into the root). A cell occupying level i is reachable from the root via a path of i edges, corresponding to the values of the rightmost i variables of the cell. For example, the leftmost cell of level 3 in the left tree of Figure 6.1 ends in the string `011`, describing the (reverse) path from the cell back to the root. The same node occupies level 2 in the right tree, reflecting only the two rightmost variable values `11`.

The relationship between tree level and variables imposes a strict maximum depth of n on a cube tree. If a search is guided down only one path, that search can be performed in $O(n)$ time. However, most MDSA operations require more complicated searches, often branching to two or three of the subtrees emanating from a node. In Figure 6.1, the right cube tree (b) is more "balanced", in a sense. However,

the branching is also more "dense" than in (a), and search times may be adversely affected. Different trees for the same set of cells (dependent on the same variables) arise from different orderings of insertions and deletions of members, as shown in the next section.

The branching pattern of a tree also depends on how the variables are permuted. For example, if the root level corresponds to variable v_{n-1}, and down to v_0 at the bottom-most level, the structure will be quite different, and search times will be affected. In Figure 6.1, the cell `110X1` in the rightmost position of level 3, would occupy the node to its immediate left had the same set been embedded in the same tree, but with reversed variable ordering. This is required because `110X1` is the only cell *beginning* with the string `110`. Given any particular permutation of variables, a cell must lie somewhere on a distinct path of length n through a hypothetical "complete" tree (filled with all possible cubes). The exact location along this path depends on the order in which cubes are inserted and deleted from the set.

6.3 Primitive Tree Maintenance Operations

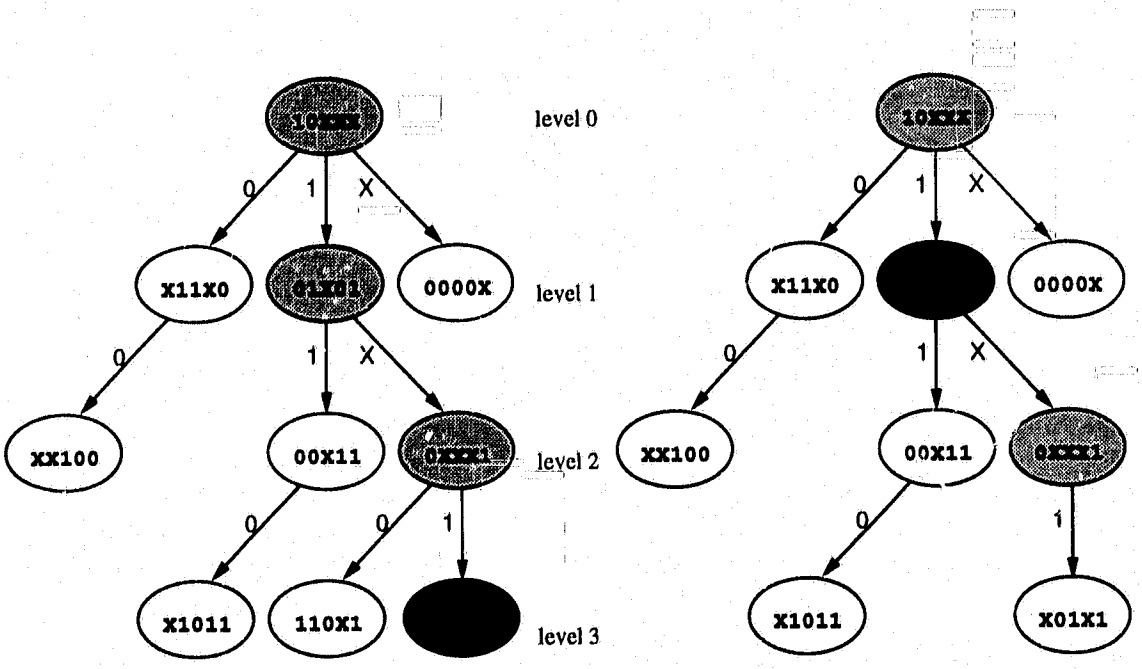
An unordered list-based set allows insertion of members in constant time, plus removal of members in constant time once they are found. Insertion and removal operations are more complicated in cube trees, because the tree properties must be maintained to ensure correct and fast searching time. Trees become useful data structures when the time saved by having more efficient searching, is greater than that required for these primitive maintenance operations. This occurs at relatively small set sizes, because the primitive operations have only an $O(lg\ k)$ expected time complexity, where k is the size of the set. The expected time depends on maintaining the tree in a somewhat balanced state, which is possible using the techniques described below.

6.3.1 Inserting a cell

When a cell is inserted into a cube tree, it cannot be put anywhere that would cause another node to move to an incorrect level, or else the branches from the bumped node may no longer correspond to the appropriate variable. Thus, the easiest way to accomplish the insertion of a new member is to restrict this operation to occur only at the leaves. If the tree is empty, the inserted cell becomes the root node. If the root is already occupied, the branch leaving the root corresponding to the value in position 0 of the cell is taken. If that branch is null, then the branch is created, terminating in the newly inserted cell at level 1. If already occupied, the search for an empty branch continues recursively, down through the tree, taking the appropriate branch determined by the current level and the corresponding value in the cell, until the new cell can be inserted as a leaf.

Figure 6.2(a) shows the insertion of a new node with cell `X01X1`, to the cube tree from Figure 6.1(a). In this example, three existing nodes must be visited; they are shown shaded. The new node, shown more darkly shaded, terminates the chain.

Clearly, the number of nodes that must be bypassed in an insertion operation is bounded above by n: the depth of the tree. It may be desirable to maintain the tree such that the number of bypassed nodes is minimized in the average case. However, search times may be adversely affected. One way to reproduce any given legal cube tree is to insert cells in the order given by a preorder traversal of that tree. In fact, any topological ordering of a cube tree gives one possible insertion sequence. Inorder and postorder reproductions will end up as completely different trees.



(a) Insertion of X01X1

(b) Removal of 01X01

Figure 6.2: Primitive Operations on Cell Trees

6.3.2 Removing a Cell

For this operation, it is assumed that the node requiring removal has already been located (perhaps via the search techniques of the next section). In order to maintain the cube tree property, the location left vacant by removing the node must be filled with some node allowed to be there. Furthermore, this must be done without affecting the positions of nodes below the vacant location. This is accomplished by filling the vacancy with any leaf node beneath it, since all nodes belonging to a subtree rooted at a given node satisfy the cube tree property of that node. Like insertion of a node, this operation is bounded above by n.

If the processing of a cube tree is organized such that nodes to be removed are near leaves, the time required to search for a leaf is minimized. A postorder traversal has this desirable property. Also, in MDSA it is best to look down the "don't-care" branch to find a leaf, because this reduces the overall depth of "don't-care" accessible nodes. This in turn reduces expected search time, since most searches require examination of the "don't-care" branch, then only one or the other of the remaining two branches. In Figure 6.1(a), removal of cell `01X01` at the centre of level 1 would result in that node's replacement with cell `110X1`, formerly at level 3. Then, `0XXX1` would become a leaf.

Figure 6.2(b) shows the removal of cell `01X01` from the tree in Figure 6.2(a). The vacated position is shown darkly shaded. When `01X01` is removed, the search for a replacement proceeds first to `0XXX1`, shown lightly shaded, down the "don't-care" branch. This node is not a leaf, but since it does not have a "don't-care" child, the first child, `110X1` down the 0 branch, is visited. This last node is a leaf, so it is brought up to the vacated position.

6.3.3 Bump Insertion

There is an alternative way to insert cells into a cube tree, when changes to the set structure larger than effects to leaves are permissible. The standard insertion operation requires the traversal of a chain of maximum length n, before the insertion is finally performed at a leaf. It is sometimes possible to *redirect* this chain as the traversal proceeds, by substituting the cube tree node currently being visited, with the cell to be inserted. The upper bound is not changed, because the bound on any path through the tree is n.

It is always possible to perform the substitution, because (1) the cell to be inserted is allowed to occupy the current spot, and (2) there must be a location for the substituted cell, further down the subtree rooted at that spot. Both claims follow from the definition of cube trees, in the previous section. Whether the traversed chain is actually redirected depends on the values of the cells' variable for that level of the tree. When the values are different, a new direction is taken. The introduced technique is called *bump insertion*, because of the substitution that takes place as an empty leaf position is searched for.

Bump insertion could be used for keeping cube trees balanced, but in MDSA it is more beneficial to use the technique to reduce the number of set members accessible down "don't-care" branches. The rationale is the same as the one for removing nodes down "don't-care" branches whenever possible, briefly discussed in the previous subsection. The potential improvements to search performance are made more lucid in subsequent sections, and greatly outweigh the slightly higher insertion times required by the generally taller resulting cube trees. (Maximum depth is still bounded by n.)

Two circumstances must be true for a "bump", or substitution, to occur during bump insertion:

1. the cell to be inserted has a "don't-care" for the variable corresponding to the current level, and

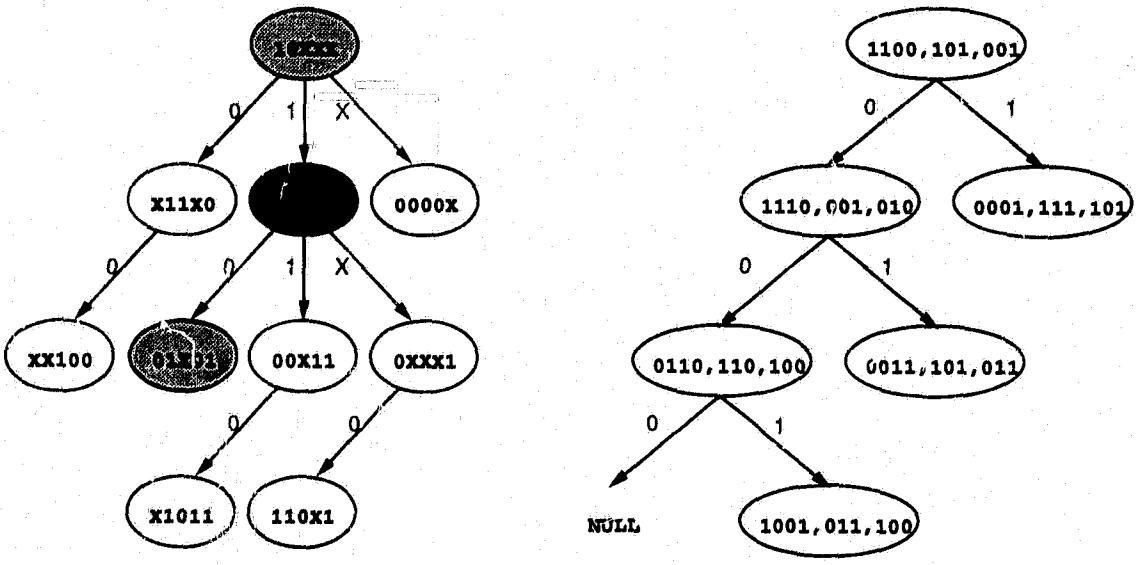
2. the visited cell in the tree does *not* have a "don't-care" for the same variable

In these conditions, a substitution of the two cells results in the next traversed edge to be down a specified branch, rather than a "don't-care" one. Thus, an insertion to a "don't-care" subset of the cube tree is avoided.

A bump insertion is illustrated in Figure 6.3(a), where the cell `0X1X1` is bump inserted to the tree of Figure 6.1(a). The visited and affected nodes are shown shaded in Figure 6.3(a). The traversal begins at the root, `10XXX`. No substitution is performed here, because `0X1X1` is destined to take the 1 branch, whereas `10XXX` would take the less desirable X ("don't-care") branch. At the next visited node in the chain, `01X01` at level 1, a substitution does take place, because the node to be inserted `0X1X1` would take the X branch, but `01X01` does not. The operation terminates with the substituted node occupying the new leaf position down the 0 branch.

6.4 Multi-valued Extensions

In the preceding discussion, attention is restricted to Boolean-valued cubes to simplify the explanation. The extension to multi-valued variables is relatively straightforward, though the effectiveness of the tree is diminished as the radix of each variable increases. In particular, the notion of a special "don't-care" branch is abandoned, because it does not extend efficiently to the multi-valued case.



(a) Bump insertion of 0X1X1

(b) Multi–valued cell tree

Figure 6.3: Bump Insertion, and Cell Tree for (4,3,3)-valued Cells

6.4.1 Multiple-output PLA's

An m-output two-valued two-level minimization problem may be considered a single output problem of n Boolean variables, and one m-valued variable [Sas81]. Since there is only one multi-valued variable, it is left as the last variable to be examined in any search. Thus, it is possible to keep the "don't-care" branches and three-way branching associated with the binary variables. The fact that all nodes correspond to cells of distinct prime implicants allows the last m-valued "variable" to not require any subsequent searching, after the previous n two-valued variables have been processed. In other words, the cell of a prime implicant cannot be any deeper than level n, since two cells that both satisfy the cube tree property for a particular node at level n, cannot both be prime.

6.4.2 Multiple Valued Variables

A general multi-valued cube can be represented by a single bit-string, with r bits required for variable v_i of radix r. Because the concept of "don't-care" does not apply readily to multi-valued variables (there are $2^r - r - 1$ distinct "don't-care" conditions for a variable of radix r), the trees used for storing such cubes are best left strictly binary. Their depth is bounded by the *sum* of all radices of all variables. The bound is proportionally identical to the three-way trees for cubes of binary variables. The latter are simply restricted to *half* the sum of radices for depth, by introducing the special "don't-care" branch. For each variable v_i of radix r, there are r corresponding levels in the multi-valued cube tree.

Figure 6.3(b) shows a binary cube tree storing six cubes, each of three multi-valued variables. The radix of v_0 is three, that of v_1 is three, and that of v_2 is four. Note the branch labeled "NULL". This branch would *always* be NULL, because it

corresponds to the last (leftmost) bit position for variable v_0, and corresponds to the 0 value for that bit position, which is not possible after it is known that the previous two positions are already 0. This phenomenon does not cause any ill effects while searching. It simply indicates that the name space is slightly too large. (There are only $2^r - 1$ possible values for the part of a cube corresponding to a variable of radix r.) The depth at which a cube might be found is still bounded by the number of bits in its representation.

6.4.3 Primitive Operations on Multi-valued Cube Trees

The simple insertion of nodes to multi-valued cube trees is basically unchanged: nodes are inserted at the end of the appropriate search chain. However, the removal and bump insertion operations must be redefined, because multi-valued cube trees do not have explicit "don't-care" branches. The meaning of a 1 in a bit vector representing a multi-valued cube or cell, is the dependence of that cube or cell on some particular value of some variable. The more the number of 1's in the substring corresponding to a particular variable, the less the cell depends on that variable. Thus, removals favour 1 branches and bump insertions favour 0 branches.

6.5 Primitive Search Operations

Cube trees are useful because search operations often require the examination of only one or two of the three branches leaving every node. Thus, whole subtrees, corresponding to large fractions of the entire set, are eliminated from consideration with each node visit. Such restriction of the search space is not possible with list-based set representations. In MDSA, there are four primitive search operations that

are used, namely

- contained minterm
- intersecting cell
- contained cell
- equal cell

The traversal order chosen for a search can be postorder, preorder, or inorder. Sometimes, search time is reduced by choosing paths more likely to lead to an expected find, first. These effects depend on the particular application.

Most searches require traversing the "don't-care" branches, which explains why cells are used instead of their subsuming cubes in the sets. Every "don't-care" variable in a cell is also in its containing cube, but the converse is not true. The undesirability of "don't-care" branches also explains why node removals explore the "don't-care" branches first, and why bump insertion is designed to favour 0 and 1 branches.

6.5.1 Contained Minterm

In this search, all the cells that contain a particular minterm are sought. After examination of a node, the search proceeds down two of the three emanating branches. The "don't-care" branch is always taken, because cells that have a "don't-care" in that variable location are candidates. Also, the branch corresponding to the value of the variable in the position of the minterm corresponding to the current tree level, is taken.

6.5.2 Intersecting Cell

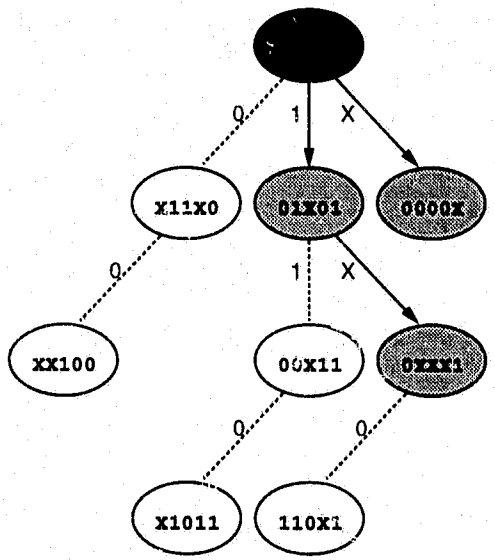
A cell intersects with another *iff* there is some non-empty cube contained by both of them. In searching for cells intersecting a given cell, the number of branches that must be taken depends on the value in the current variable in the cell. If it is "don't-care", all three branches must be taken. If defined 0, then the two branches 0 and "don't-care" are taken, and similarly for defined 1. Again, note that "don't-care" branches are taken, regardless.

6.5.3 Contained Cell

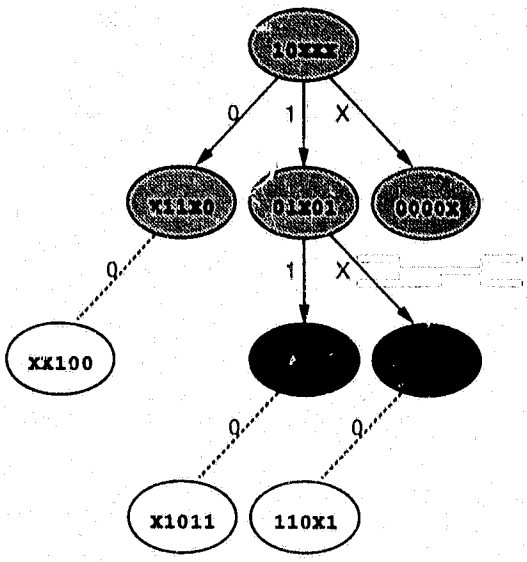
In MDSA, this test is used in the final stages of the determination of dominated cubes. Given a suspect *affected* cell, a dominator must be found for it, if one exists. In this case, the search halts when the very first dominator is located. The search proceeds down only the "don't-care" branch if the value of the current variable in the affected cell is "don't-care". Otherwise, the search proceeds down the "don't-care" branch and one other, as in the case of intersecting cells, described above.

6.5.4 Equal Cell

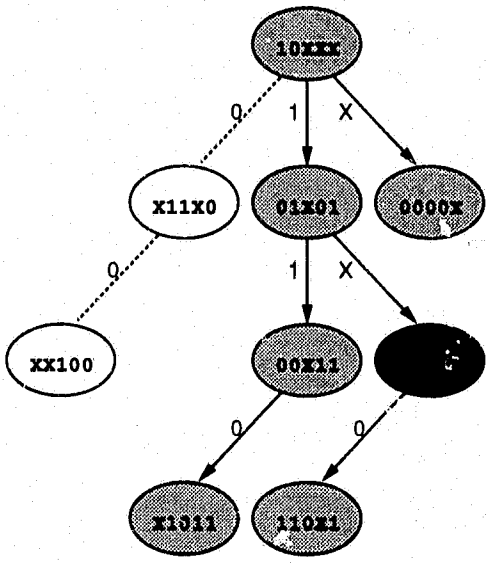
This type of search is used in the prime implicant generator, described in Chapter 7. It is used to see if a particular cell already exists in a cube tree. Unlike the previous three primitive search types, this one requires the examination of only one path through the tree, because of the basic cube tree property that stipulates that a cell must lie on the path determined by the values of its n variables. In circumstances where there is danger of inserting a duplicate cell to a cube tree, this search becomes part of the primitive cell insertion maintenance operation.



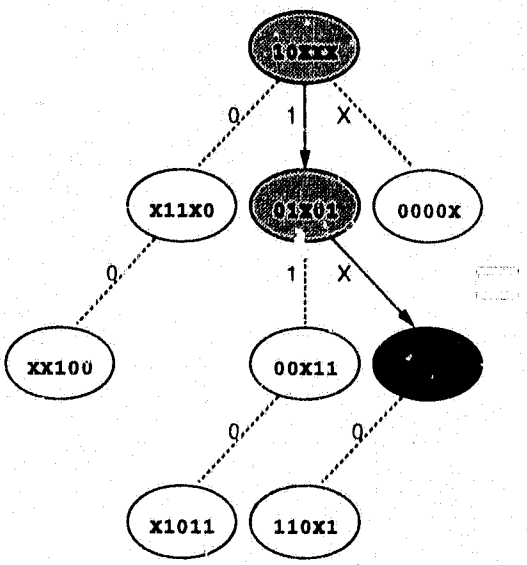
(a) Contained minterm: 10101



(b) Intersecting cell: 0011X



(c) Contained cell: 0X011



(d) Equal cell: 0XXX1

Figure 6.4: Four Search Subtrees

Figure 6.4 shows the parts of the tree from Figure 6.1(a) that would have to be searched, given the four primitive searches and example search items. The nodes belonging to the searched subtrees are shaded, with the nodes actually satisfying the search criteria more darkly shaded. In Figure 6.4(a), if the search was required to find precisely one cell containing minterm `10101`, then the search could halt right away. Similarly, if the cell `0X011` in Figure 6.4(c) is an affected cell, for which a dominator must be found (if one exists), the search could halt as soon as it reaches node `0XXX1`.

Note that in the first three search operations, the subtrees lying down "don't-care" branches exiting each visited node, must also be visited. This is why it is desirable to make the "don't-care" branches null whenever possible. Like tree shape, the number of nodes visited in a search depends on variable ordering and node insertion order.

6.6 Minterm Trees

Naturally enough, the minterm sets that MDSA maintains can also be structured as trees. If, as before, attention is initially confined to the Boolean variables case, the branching factor is two. In this case, minterm trees are strictly binary, and their depth is bounded by n: the number of variables. Like cells with multi-valued variables, branching is determined by the 0's and 1's in the minterms' bit vectors. Minterm trees may be considered as special cases of cube trees, where all the cubes have zero degree.

Though all MDSA minterm sets may potentially be stored as minterm trees, only the set of expanded minterms EMTS is converted to this representation. This is because only EMTS is of "controlled" size; the others (NMTS and RMTS) could be almost arbitrarily huge, and alternative techniques for representing them are considered in Chapter 10. Detailed discussion of maintaining EMTS as a minterm tree is

postponed to Section 6.9, because it is convenient to maintain the set as a partial order.

6.7 Applications of Cube Trees in MDSA

This section identifies the specific areas in which cube trees are useful in the MDSA minimizer. There are three main categories of interest:

- prime implicant generation,
- location of pseudo-essential cubes, and
- the determination of dominated cubes.

In MDSA, the cell sets that use the trees are the PPIS (potential prime implicant set), the PIS (prime implicant set), and the PCPI (set of potentially cyclic prime implicants). The first two sets are generated and disposed of with each main algorithm iteration. There is only one PCPI, which grows and shrinks throughout a minimization.

Within the three categories, seven distinct operations are identified:

1. Maintaining the implicant frontier
2. Avoiding regeneration of dominated cubes
3. Locating previously generated cubes for PIS
4. Locating previously generated cubes for selection
5. Dominance in the prime implicant set (PIS)
6. Dominance induced by selected cubes

7. Dominance induced by intersection cells

The first three operations are involved in the generation of prime implicants. The fourth operation is involved in the search for pseudo-essential implicants, and the last three operations are all involved with the detection and removal of dominated cubes.

6.7.1 Maintaining the Implicant Frontier

The prime implicant generation algorithm of Chapter 7 involves the continual updating of the *potential prime implicant set*, or PPIS, as new covering cubes are generated. Each iteration of the generation algorithm involves the identification and splitting of every member of PPIS which does not satisfy the "don't-care constraints" imposed by the latest prime cube generated or found. With the PPIS maintained as a cube tree, these "unsatisfied" cubes are located more directly. The required search is based on the cell containment traversal. Gradually, the implicants comprising this "implicant frontier" become prime, and are added to the PIS. The process is described in detail in Chapter 7.

6.7.2 Avoiding Regeneration of Dominated Cubes

When a cube is found to be dominated (because its cell is contained by some other cube's cell), it remains dominated for the remainder of the minimization. In many instances, such cubes are regenerated by the prime implicant generator in subsequent iterations. This regeneration of redundant cubes is avoided by priming the prime implicant generation engine with known dominated cubes covering the selected minterm, as well as the known undominated prime cubes already in existence, in PCPI.

A tree of removed dominated cubes is maintained. When a minterm is selected for expansion, this cube tree, called DOM, is searched using a minterm containment traversal to find cubes with which to "prime" the prime implicant generator. Thus, it is possible to avoid the regeneration and re-removal of these cubes. The search through DOM proceeds using cells, but the priming of the generator uses the corresponding cubes. The technique is described in detail in Chapter 7.

6.7.3 Locating Covering Cubes

Each main iteration of MDSA invoives the selection of a minterm, whose complete covering set of implicants is then found. If this minterm is "recursive", then one or more of its covering cubes already exist in the potential prime implicant set, or PCPI. All such cubes must be found before the generation of new cubes proceeds, so that various redundant operations within the generator are avoided. A minterm containment traversal of the PCPI cube tree is used. Note the similarity between this operation and the previous one, above. DOM is searched in the previous operation; PCPI is searched here.

6.7.4 Locating Previously Generated Cubes for Selection

When dominated cubes are removed from PCPI, one or more other cubes of PCPI may become pseudo-essential. The existence of pseudo-essentiality is determined by searching the set of expanded minterms EMTS. When an EMT with cover count of one is discovered, the minterm containment search is performed on PCPI to locate the unique corresponding cube. The search terminates as soon as the one pseudo-essential prime, corresponding to the pseudo-essential minterm, is located.

6.7.5 Dominance in the Prime Implicant Set

The prime implicant generator does not contain any strategies for avoiding the generation of dominated cubes, except for using previously-discovered dominated cubes to prime it (described above). Thus, after initial PIS generation, the affected and contracted cells must be determined, and a search for a dominator of each must be performed. A cell intersection traversal is required to find candidates. For each candidate determined to be actually contracted, a cell containment search on the PIS is used, and the search halts upon discovery of the first dominator, if there is one. The dominated cube is removed from PIS, and transferred to the set of rejected dominated cubes DOM, to help prime the prime implicant generator in future iterations.

6.7.6 Dominance Induced by Selected Cubes

When a cube is selected for the cover, all minterms covered by its cell (and thus, by definition, all minterms contained by that cube) become "don't-care". This may cause some cubes' cells to become contracted, and some of these cells may then be dominated. In this case, affected cells are discovered by performing the intersecting cell search on PCPI, revealing the candidates. The cell of each candidate must be re-calculated. If it grows smaller, it is officially contracted, then it is (probably) in the wrong location in the PCPI tree, so it is removed. The remaining required operations are similar to those described above.

6.7.7 Dominance Induced by Intersection Cells

A similar situation to the above two is when a set of minterms become "don't-care" because they are found to be *dominating*. In MDSA, the intersection cell of PIS cells

is used to identify and eliminate these minterms. A process similar to those described in the above two subsections is used. Minterm dominance techniques are discussed at length in Chapter 8.

6.8 Dominance Traversals of Cube Trees

The three types of dominance search described in the previous section all involve the same basic operations:

- find intersected (candidate) cells,
- determine if each intersected cell is contracted,
- if so, see if the contracted cell is dominated

Given a cell that is newly removed because it is dominated, one main traversal of the cube tree of interest suffices to identify all intersected and contracted cells. A postorder traversal is used. When a candidate is identified, the traversal is suspended, and the cell of the candidate is recalculated. If found to be contracted, it is removed from the tree. Then, a cell containment traversal of the tree is performed (any traversal order is OK). If no dominator is found, the contracted cell is re-inserted in the tree, and the postorder search for candidate cubes resumes.

Postorder traversal is required so that each cell is tested for candidacy, affectedness, and contractedness only once. If the cell of a candidate contracts, then one or more "don't-care" variables in that cell change to specified variables. If the cell is then found to be still undominated, then its new location when re-inserted in the cube tree *must be in a section already searched for candidates*, so the structure of the cube tree being traversed is not corrupted. If the rightmost changed variable belongs

to a level no higher than that previously occupied by the cell, it will be re-inserted as a leaf which is a direct descendant of its former location. This explains the postorder processing. Also, the "don't-care" branch of a node is processed *last*, because when the rightmost changed variable belongs to a higher level (smaller i in v_i), the cell will be re-inserted down a subtree belonging to a specified variable branch. All such subtrees would have been previously searched for affected cells.

The dominance (cell containment) tests do not require the cells of potential dominators to be up-to-date. This is because the dominance relation between two cubes is the same among all subcubes of those cubes that fully contain the cells of those cubes. In other words, a contracted cell is contained by another contracted cell *iff* the cube of the former cell is dominated by the cube of the latter. Thus, there is no danger in searching for a dominator of a contracted cell, before all contracted cubes are found. Note that this traversal technique precludes the use of bump insertion, because changes to the structure of the PCPI tree would cause errors.

6.9 Cube Heaps

Cube and minterm trees have a property that permits their maintenance as partially ordered sets, simultaneously with their ordering based on variables' values. Recall that the root of any subtree of a given cube tree can legally be any node contained in that subtree. Thus, it is possible for the root of a given subtree to be, say, the heaviest cube of that subtree. The same argument holds inductively throughout the entire cube tree. Thus, a cube or minterm tree can be maintained as a *partial order* [Knu73].

Partially ordered trees are called *heaps* when they are also balanced. Cube trees are not necessarily balanced, but their depth is bounded, and this provides similar

restrictions on the complexity of basic operations. Thus, cube trees arranged as partial orders are called *cube heaps*. These data structures are used most extensively in Chapter 9, but are introduced here because of their relationship to cube trees. Figure 6.5(a) is an example of a cube heap. Hereafter, the word "heap" implies "cube heap".

Three primitive operations are required for properly maintaining heaps. They are:

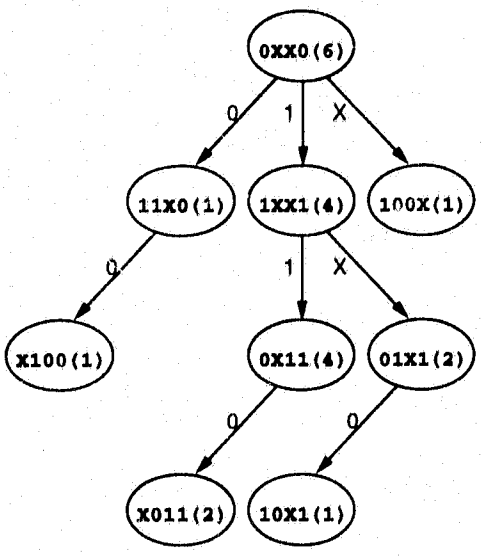
- Insertion
- Removal
- Adjustment

For the descriptions which follow below, it is assumed that the required heap order is that a node is no lighter than any member of its subtree. All three operations are bounded by the depth of the tree, as before in the case of cube trees.

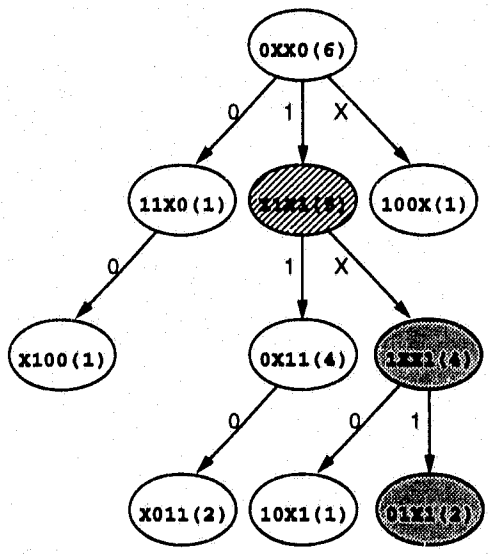
6.9.1 Insertion to a Heap

A technique similar to that of bump insertion, introduced in Section 6.3.3, permits the insertion of a new record to a cube heap. The operation proceeds like simple insertion, down the appropriate path of the heap, until a node is reached that is lighter than the node to be inserted. It would violate the heap property for the new node to become a descendant of this light node. Instead, the node to be inserted is substituted for the visited node, and the latter becomes the node to be inserted. The technique is then carried out recursively with the new node to be inserted, on the current subtree now rooted by the original node to be inserted.

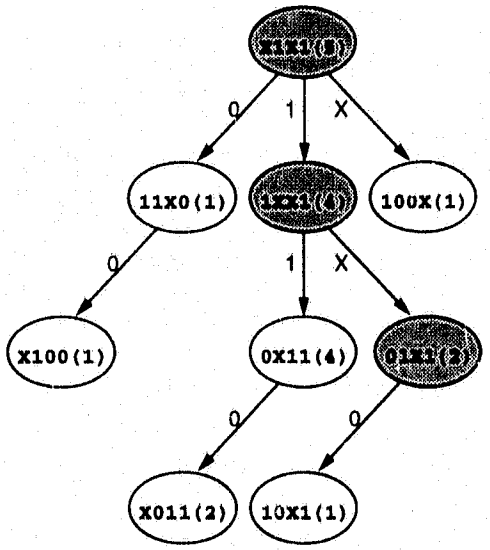
In Figure 6.5(b), the cell `X1X1` with weight 5 is inserted in the heap of Figure 6.5(a). Nodes affected by the insertion are shown shaded. `X1X1` bumps `1XX1` out of its



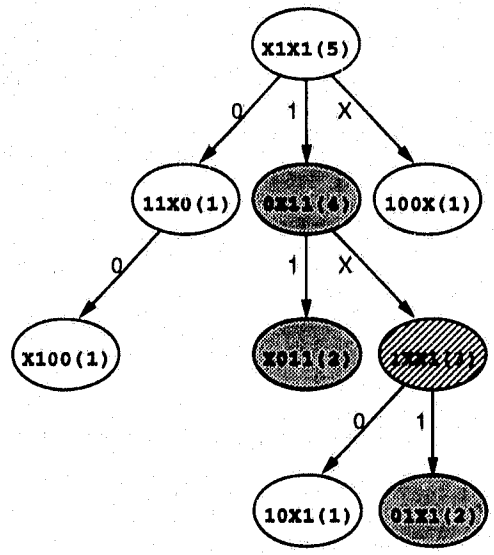
(a) A cell heap



(b) Insertion of cell X1X1(5)



(c) Removal of root 0XX0(6)



(d) Adjustment of 1XX1 to 3

Figure 6.5: Heap Operations

position, because the weight of the latter is too small. 1XX1 is then recursively heap-inserted to the subtree now rooted with X1X1, causing 01X1 to be bumped.

6.9.2 Removal from a Heap

Removal from simple cube trees requires the replacement of the node to be removed with any leaf that is a descendant of that node. The situation is quite different with heaps, because both the heap ordering and the cube tree properties must be maintained. Removal of a record requires choosing the appropriate child to fill the vacancy, rather than defaulting to a particular branch. The cube tree property makes any child legal, but the heap property requires choosing the heaviest. The appropriate technique is a recursive one, whereby the same heap removal primitive is applied to the newly vacated position occupied by the heaviest child. When there are ties between heaviest weights, nodes down "don't-care" branches have higher priority for moving up, for the same reasons as those discussed for simple removal.

Figure 6.5(c) shows what the example heap looks like after the removal of the root cell 0XX0 of Figure 6.5(b). Again, nodes affected by the operation are shown shaded. The heaviest child of 0XX0 is X1X1, so the latter takes the former's place. This leaves a vacancy, so the same process is applied recursively at this location. This time, the two orphaned children 0X11 and 1XX1 have equal weight. 1XX1 is chosen to move up the tree to occupy its former parent's location, because it lies down a "don't-care" branch. Finally, 01X1 also moves up one level.

6.9.3 Heap Adjustments

Correct maintenance of heaps requires some additional bookkeeping, because for some heap sets the ordering weights are *dynamic* quantities. For example, a useful heap

ordering for cube trees is the weight of cells, and these are liable to change (grow lighter, actually) as minimization proceeds. Similarly, the covering counts of members of EMTS grow smaller, as dominated cubes are recognized and removed.

When an adjustment to a heap member's weight occurs, it might no longer be in its correct location. The connections in the heap data structure actually used in MDSA, only go towards leaves. Comparisons between the adjusted member and its children are possible, but not between the adjusted member and its parent. Thus, postorder traversals are used when adjustments are necessary, so that changes to heap structure are limited to those subtrees that have been traversed already.

The adjustment itself is quite a simple operation, requiring only a few comparisons, and use of the heap-removal and heap-insertion primitives. If the currently visited node is potentially out of order with respect to its subtree, it is compared with its heaviest child. Since the traversal is postorder, an inductive argument guarantees that the subtree rooted at this heavy child is a heap. If the current node is lighter than the heavy child, all that is required to maintain the heap property is a heap-removal of the child from its subheap, substitution of this former child for its former parent, and a heap-insertion of the substituted parent to the new subheap rooted at the former child.

Figure 6.5(d) shows the effects of an adjustment to `1XX1` in Figure 6.5(c), whose weight drops from 4 to 3. It is found to be out of heap order by comparing its weight to that of its children. The heap-remove primitive is used, causing upward movement of `0X11` and `X011`. Then, the heap-insert primitive is used on `1XX1` and the subheap now rooted with `0X11`, eventually causing `01X1` to be bumped.

6.10 Detection of Pseudo-Essential Minterms

Ordering all the minterm and cube sets of MDSA is potentially beneficial. Chapter 9 shows how the PCPI is best stored as a heap for heuristic and exact resolution of cycles. Chapter 10 shows how it is useful to maintain the PIS sets as heaps, as well as the new disjoint cube sets introduced there. In this section attention is confined to ordering the set of expanded minterms EMTS as a heap, because the heapifying of the remaining sets is more appropriate in subsequent contexts.

Heretofore, the last operation of an MDSA iteration is to examine the set of expanded minterms to see if any have become pseudo-essential, due to the removal of one or more dominated cubes. If a pseudo-essential minterm (*i.e.* an expanded minterm with a cover count of 1) indeed exists. then its covering cube is located, covered minterms removed, and if any new dominateds arise, process repeats.

This part of the algorithm is improved by maintaining the EMTS as a heap, ordered by smallest cover count. If one or more pseudo-essential minterms exist, one must occupy the root position. This minterm is then simply heap-removed from EMTS. The set is maintained as a heap by heap-inserting new EMT's as they are discovered, traversing the heap with each removed dominated cube, and traversing with each selected cube so that contained minterms can be heap-removed. Both traversals are postorder, to maintain the proper structure.

6.11 Results

In this section, experiments with benchmark PLA's are described, comparing times required by the old list-based version of MDSA against the time required by the new tree-based version. Various counters are used to see how search time is reduced, and

PLA				INITIALIZE			PSEUDOS			AFFECTED	
name	n	m	p	num	list	tree	num	list	tree	list	tree
alu1	12	8	19	17	216	100	0	0	0	9432	3445
alu2	10	8	68	61	302	296	10	54	50	2091	1356
add6	12	7	355	253	34323	5122	92	17178	1807	480799	59232
add7	14	8	735	475	8226	3158	5	15	28	123859	41418
dk48	15	17	22	12	23	24	2	2	3	29	27
in1	16	17	104	74	106	140	6	6	12	926	939
*dist	8	5	120	208	3076	2076	55	2221	572	18682	3584
*in0	15	11	107	92	428	720	15	140	101	3219	1898
*in2	19	10	134	174	820	937	30	178	132	3704	1907
*sym9	9	1	96	11760	425931	46520	22	397	170	647949	51635
*apex4	9	19	428	1343	106994	29986	252	48056	6447	499413	43191
*sev	8	10	203	1782	133597	20085	138	16157	2265	479349	34902
*tial	14	8	576	1012	27629	22151	151	9812	3006	370558	58692
*sym10	10	1	221	27290	2178829	129513	108	5421	1778	4220010	195781
*misex3	14	14	686	33031	4424165	281416	360	12899	8026	12.67M	0.68M
*misex3c	14	14	202	95596	7955631	926777	105	15462	3377	44.6M	13.0M

Table 6.1: Search Statistics for Lists vs. Trees

how set maintenance time is increased, for the relevant data structures. The results are presented in Tables 6.1, 6.2, and 6.3. In each table, the first section (four columns) gives invariant data of each benchmark: name, number of inputs n, number of outputs m, and the number of cubes found by MDSA for the cover p.

All experiments are performed using the "full order" option of MDSA, specifying that heuristic-based searches for expandable minterms and cycle-breaking cubes must look at the "values" (represented by various bit-strings) of the data, in order to break ties in heuristic value. For example, when an RMT is to be chosen, and there exist several covered by only one cube, and having an order of one (one unexplored adjacency), then all such RMT's must be compared. Similarly, if a cycle-breaking heuristic applies equally to two or more cubes, the bit-strings representing the cubes are compared to select the winner. In this way, the output of MDSA with and without various tree options, is the same for those data not immediately concerned with the cube sets. It should be noted that this "normalization" affects performance adversely.

Table 6.1 shows how the organization into trees improves the search for cubes.

	PLA			INSERTIONS		REMOVALS	
name	n	m	p	calls	tests	calls	tests
alu1	12	8	22	488	2737	439	822
alu2	10	8	68	644	1568	601	839
add6	12	7	355	6257	32512	5999	9407
add7	14	8	735	16298	88677	15573	18268
dk48	15	17	22	72	56	59	78
in1	16	17	104	729	1139	656	816
*dist	8	5	120	1358	3522	1311	1825
*in0	15	11	107	850	2588	800	1269
*in2	19	10	134	1148	2971	1067	1721
*sym9	9	1	96	27002	147080	27002	34964
*apex4	9	19	428	7357	26905	7242	9869
*sev	8	10	203	7272	28316	7261	10950
*tial	14	8	576	9077	34956	8362	13966
*sym10	10	1	221	63651	393834	63631	90476
*misex3	14	14	686	108576	722943	108482	177347
*misex3c	14	14	203	689576	5273127	689510	948075

Table 6.2: Maintenance Overhead of Trees

It gives the number of searches required, and the total number of cubes and cells examined, for both the list-based PCPI (or PIS) set and the tree-based PCPI (or PIS) set. Three search categories are identified. The "INITIALIZE" section refers to the search for already-existing cubes in PCPI covering selected recursive minterms. The "PSEUDOS" section is for the search through PCPI required when an EMT with cover count of one arises after deletion of dominated cubes. In such cases, the single covering cube must be found, and added to the cover. Finally, the "AFFECTED" section gives the number of cell containment tests required to locate dominated cubes. In this case, the numbers represent the sum total of all tests required by searching through both the PCPI set and the PIS. Note that all the figures in Table 6.1 reflect minimizations run *without* using the DOM tree to avoid re-generation of dominated cubes. The figures for affected cubes are significantly lower with the DOM cube tree.

Table 6.2 gives data indicating the cost of maintaining the cube sets (both PCPI, and the PIS set generated at each MDSA iteration). The first section gives the number of cubes inserted into trees, and the overall number of tree records requiring traversal

	PLA			DOM Tree		TIMES			
name	n	m	p	size	rejects	before	doms	trees	all
alu1	12	8	22	198	0	0.34	0.34	0.36	0.36
alu2	10	8	68	205	31	0.19	0.21	0.20	0.23
add6	12	7	355	3055	203	1.40	1.44	1.24	1.30
add7	14	8	735	12475	0	3.57	3.90	3.64	4.11
dk48	15	17	22	10	0	1.05	1.03	1.08	1.07
in1	16	17	104	314	45	2.88	2.85	2.90	2.82
*dist	8	5	120	265	107	0.15	0.14	0.15	0.15
*in0	15	11	107	236	57	1.24	1.19	1.24	1.21
*in2	19	10	134	182	76	13.90	13.56	13.95	13.70
*sym9	9	1	96	1584	0	2.58	2.61	2.05	2.10
*apex4	9	19	428	1647	613	1.71	1.84	1.23	1.21
*sev	8	10	203	733	205	1.17	1.15	0.85	0.85
*tial	14	8	576	2267	645	6.32	6.49	6.43	6.50
*sym10	10	1	221	3979	335	15.00	15.10	6.55	6.53
*misex3	14	14	686	5878	16293	64.70	64.30	40.10	37.70
*misex3c	14	14	203	51217	354974	531.00	313.00	520.00	261.00

Table 6.3: Dominated Cube Trees and Tree Performance

for such operation. The second section gives similar results for removing tree records. Note that the average number of nodes requiring examination for record removal is consistently between 1 and 2. This is very good; one is the minimum required in each instance, because the record to be removed must be tested for the leaf property, which is counted as one test. On the other hand, insertion to cube trees is quite expensive for some examples, averaging as high as seven. (this number also can be considered the "average" depth of the cube trees.) Note that for the list-based version of MDSA, insertions and removals to cube sets are very fast - constant time, in fact, with no procedure calls required.

Table 6.3 gives the results of the domination tree experiment, and the minimization times for the MDSA variants. The first section, "DOM Tree", gives the eventual size of the dominated cube set (the size monotonically increases throughout a minimization), and the number of dominated cubes that would have been regenerated, had the DOM tree not been used. Bump insertion is used for building the DOM tree. The TIMES section of Table 6.3 gives times for MDSA without any trees, with the DOM tree but

no other cube trees, with PCPI and PIS cube trees but no DOM tree, and finally with all three sets in cube trees (DOM, PCPI and PIS).

The general effect of using the cube trees is that some examples are much improved, while none are adversely affected to any significant degree. The PLA `misex3` is an example of one whose MDSA minimization improves with the use of the PPIS and PIS sets, but shows hardly any improvement at all with DOM. On the other hand, the minimization of `misex3c` is much improved with the DOM tree alone, with comparatively little improvement due to PIS and PCPI trees, alone. Using all three trees yields still better improvement, because use of the DOM tree decreases the size of each PIS tree, whose maintenance tends to dominate the maintenance of PCPI because so many distinct prime implicant sets must be generated.

6.12 Summary

This chapter shows how the use of cube trees and minterm trees enhances the efficiency of the MDSA minimizer. Generally, trees require fewer primitive operations in searches, compared to lists, because whole subtrees are eliminated when branches are pruned. There is a cost in inserting and removing members from cube trees, and this overhead tends to overide the searching benefits for PLA examples of low MDSA minimization complexity. But when the cube sets are both large and frequently searched, minimization performance is much improved. The cost of primitive operations on cube trees is bounded by the depth of the tree, in turn bounded by the number of input variables n.

The similarity between cube trees and other tree-like data structures already proposed and used for minimization (such as *binary decision diagrams*, or BDD's) is not explored here. It is possible that other minimizers also could benefit from the

use of cube trees, and it is also possible that MDSA could employ BDD's. Another unknown is the mathematical analysis of cube trees. Upper bounds on search depth are easily derived, but it is more difficult to establish bounds on depth and breadth of search simultaneously. The problem is compounded by the dependence of trees on the order of variables in cubes, and the order in which set members are inserted and removed. It is likely that methods used for setting variable order for BDD's would have application to cube trees.

Handling of the DOM set might be improved in several ways. Note that *all* prime implicants found to be dominated move to DOM, and stay there for the remainder of the minimization. This is unnecessary for those prime implicants whose cells completely disappear, either because their removal caused the immediate selection of a pseudo-essential cube, or a combination of selected cubes eventually covers the removed cube. Whether the increased overhead in keeping DOM to a minimum in size is worth the decrease in search time is unknown. Also, if a cube in DOM covers only expanded minterms and no recursive minterms, then it will never again be required in prime generation, and therefore could be removed from DOM.

The cube tree traversals are all implemented as simple recursive functions. This keeps the specification of the traversal algorithms simple, but a price is paid in efficiency. A more efficient implementation using an explicit stack of visited nodes could substantially reduce the level of overhead.

Chapter 7

Prime Implicant Generation

7.1 Introduction

Generation of prime implicants is a key aspect of almost all two-level minimization algorithms. It often dominates overall minimization time. In many exact minimizers, all prime implicants are generated before any essential and pseudo-essential ones are identified. In DSA and MDSA, the task is that of finding a complete covering set of prime implicants, for a given minterm. Ideally, this should be accomplished without re-generating primes already in existence, and without generating any new ones more than once for the same minterm.

This chapter presents an improved approach for finding prime implicant sets; one that exploits the more specific nature of directed search requirements. The introduced algorithm, named HYPER, finds implicants by combining elements of depth-first search with breadth-first search. The algorithm allows a more straightforward technique for incorporating already-known prime implicants, and is much less sensitive to variable ordering than the existing depth-first search alternative.

7.2 Hypercube Search Spaces

Prime implicants are maximal compatible hypercubes of dimension k, where $0 \leq k \leq n$, and n is the number of input variables (attention is restricted to the single-output problem for the moment). Compatibility implies that an implicant covers no zero vertices. The existence or non-existence of implicants is a characteristic of the function to be minimized; therefore implicants may be thought of as "discovered" or "found", as well as "generated". An implicant implies the compatibility of all cubes that are subcubes of it. Similarly, an incompatible cube implies the incompatibility of all cubes that are supercubes of it.

The *search space* for finding the prime implicant set covering a given minterm (called the *base* minterm), is itself a hypercube of dimension n. Vertices of the search space correspond to (hyper)cubes which cover the base minterm, though only a fraction of these cubes are actually implicants. Adjacency in the search space is equivalent to the *immediate subcube* relation. This corresponds to a difference of Hamming distance one in the vectors indicating the "don't-care" variables of cubes. It is convenient to think of the search hypercube as directed, where an edge connects vertex A to vertex B *iff* cube A is an immediate subcube of cube B (*i.e.* cube A covers exactly half the minterms as cube B, or equivalently, cube B includes all the "don't-care" variables as cube A, plus one additional one). In order to avoid confusion among function vertices and search space vertices, the latter are henceforth referred to as *nodes*.

The directed hypercube search space graph is acyclic. It has the property that for any two connected nodes, all their connecting paths have the same length, equal to the difference in the number of contained "don't-cares" in their corresponding cubes. For each of 2^n potential minterms in an n variable function, there exists a distinct

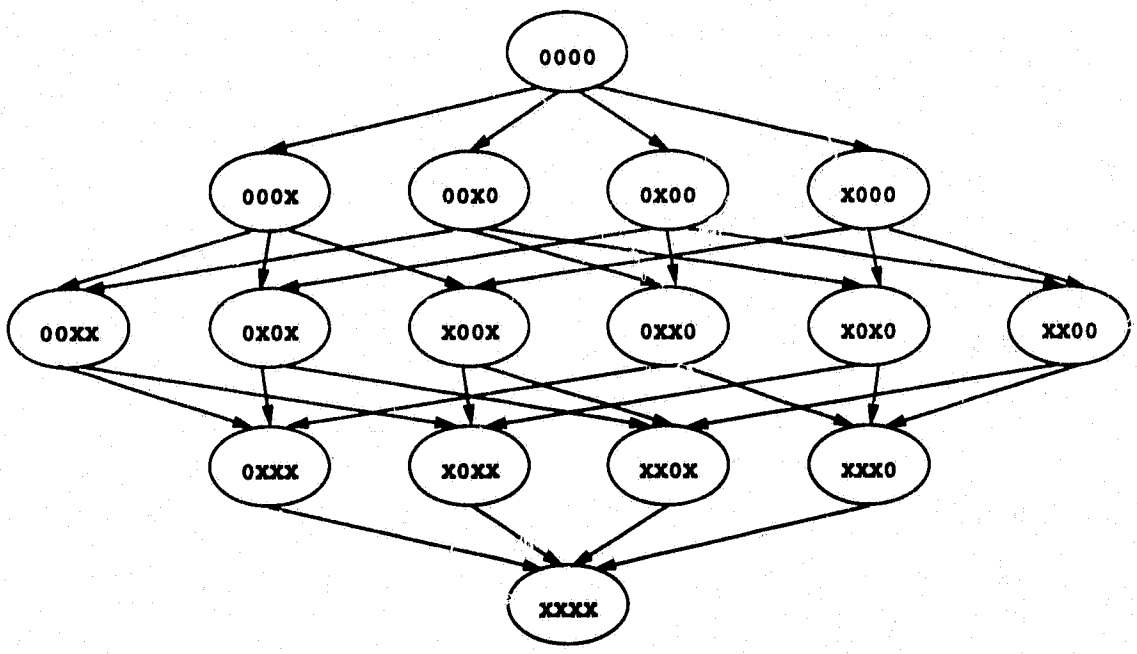


Figure 7.1: A Hypercube of Potential Implicants

directed hypercube search space originating at the minterm itself (a cube of dimension 0), and terminating at the complete function space. In fact, all such search spaces for a given function terminate at this same cube of dimension n.

Figure 7.1 shows the hypercube search space H for cubes covering vertex 0000, in a four variable function ($n = 4$). Vertex 0000 is a *source* node; no edges lead into it, and it therefore must be first in any topological ordering of H. Similarly, node XXXX is a *sink*; all incident edges lead into it. Note that the subgraph created by considering any node as a source, and all paths leading from it, is itself a hypercube with the same sink XXXX. When the search begins, all 2^n nodes of H represent *potential implicants*. As compatibility is established, the potential implicants become *true* implicants. Compatibility of implicants is established by using a primitive to traverse

edges, henceforth referred to as `TestCube`. An implicant is known to be prime if all its descendants in the search hypercube are incompatible.

Given a function and a base minterm, the problem of finding all the prime implicants in H, may be considered a *traversal* of H. For functions of non-trivial size, H is huge; it is imperative that any traversal algorithm eliminate incompatible subgraphs of H as soon as possible.

The technique used in DSA to search through hypercubes is a kind of modified depth-first search [McK74]. The method remains essentially unchanged in subsequent work [PR88, Ser84b, DM88]. Since DFS trees do not precisely embed in directed hypercubes, it is difficult to avoid performing redundant computation. Some techniques are provided for pruning some branches and discarding others, effectively eliminating large parts of the search space [McK74]. However, most of the redundancy lies in the re-examination of implicants already known to be compatible. In fact, it is possible to generate the same prime implicants more than once, effectively arriving at the same vertex of the search hypercube via distinct paths from the base minterm [DM88]. Also, incorporating information about previously generated prime implicants is awkward at best. The most serious limitation of DFS strategies are their sensitivity to variable permutations, or ordering.

7.3 The HYPER Algorithm

The problem is to find the complete set of prime implicants covering a given minterm. Without loss of generality, the base minterm is defined to consist of n zeros, *i.e.* the minterm corresponds to an assignment of zero to all its variables. For example, if $n = 4$, then the base minterm (source node of the search hypercube) is `0000`, and the complete Boolean space (corresponding to the sink node) is `XXXX`, indicating four

"don't-cares".

If an implicant A is connected by a directed edge in the search hypercube to B, B is a *child* of A and A is a *parent* of B. Alternatively,

$$parent(A, B) \equiv immediate\_subcube(B, A).$$

For any given node of the search tree, the number of parents plus the number of children is always n (a fundamental property of undirected hypercubes). *Ancestors* and *descendants* are also defined, in the obvious way - transitive closures on the *parent* and *child* relations, respectively. The child of an implicant is *legitimate* if it is also compatible; this also implies that the parent is non-prime. An illegitimate child implies the illegitimacy of all its descendants.

HYPER is based on the concept of *"don't-care" constraints*. Assume that some prime implicant P covering the base minterm, is known. Then, any *other* prime implicant covering the base minterm must include a "don't-care" (DC) variable in some place that is not a "don't-care" in P. This is easily seen by a simple argument: if some second implicant $I \neq P$ does in fact only have DC's in places where P has DC's, this implicant must be a subcube of P. Then, I cannot be prime, by definition. HYPER ensures that such redundant implicants are never generated by maintaining a set of source nodes that satisfy DC constraints. In addition, the algorithm ensures that these sources are *mutually irredundant*, and that each prime implicant that will eventually be discovered has at least one subcube (source) in this set.

For example, assume that the prime implicant `0XX0`, known to cover the base minterm `0000`, has been discovered. The prime provides two "don't-care" constraints, involving the leftmost and rightmost variables. Any other PI covering `0000` *must* include either `X000` or `000X` as a subcube. The primitive for developing PI's is based on *expansion* of implicants, whereby "don't-cares" are inserted one at a time as com-

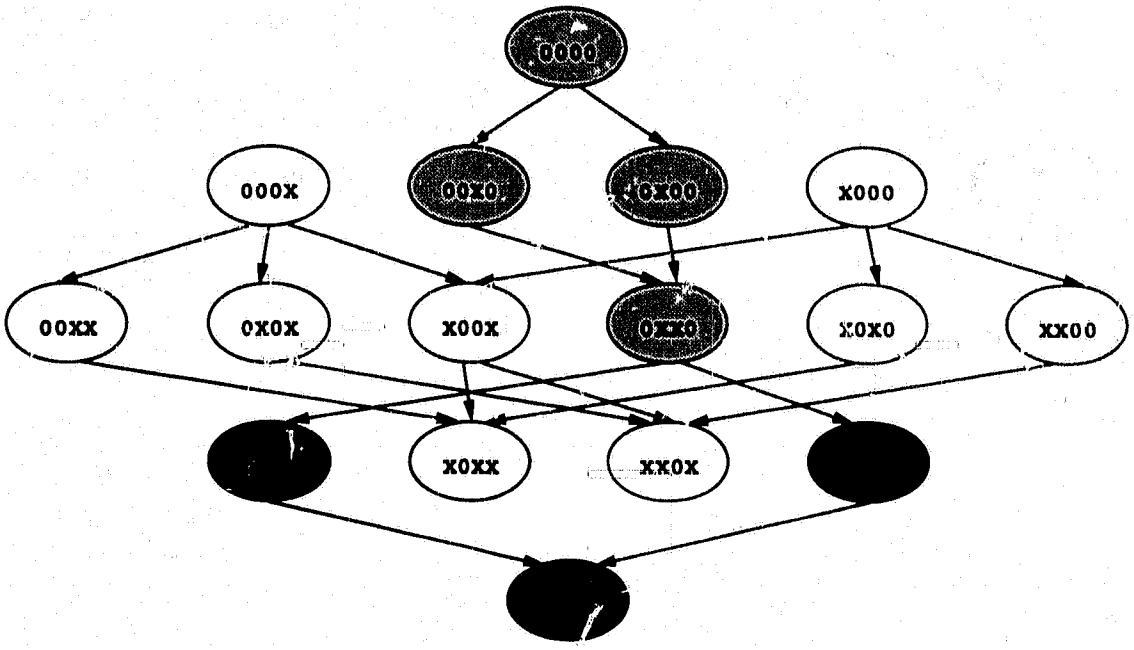


Figure 7.2: Removal of Subhypercubes

patibility is established by the `TestCube` primitive. Then in the example, *all* other PI's are *reachable*, using expansion, from the implicants `X000` and `000X`. If PI `X0XX` exists, it is possible to reach it via four distinct paths, namely

```
[X000,X00X,X0XX]
[000X,X00X,X0XX]
[X000,X0X0,X0XX]
[000X,00XX,X0XX]
```

Figure 7.2 is similar to Figure 7.1, except that the subgraph with sink node at `0XX0`, and the subgraph with the source node at the same place, are disconnected from the rest of the hypercube by removal of incident edges. The removed subgraphs are themselves hypercubes. Using DC constraints imposed by `0XX0` is like removing

the upper, lightly shaded subhypercube from the search space, leaving a graph with not one but *two* sources, namely `X000` and `000X`. All additional prime implicants must be within this remaining graph. It is worthwhile noting that the discovery of `0XX0` as a prime implicant indicates that the subgraph containing the more darkly shaded nodes `XXX0`, `0XXX` and `XXXX` could also be removed, as they are all known to be incompatible. However, this cannot be done with DC constraints alone.

HYPER starts with the base minterm as the sole member of the *potential prime implicant set*, or PPIS. There follows an iterative process in which a prime implicant P is found by expansion of a *potential prime implicant* (PPI) and added to the set of known prime implicants PIS. Then PPIS is modified to reflect the DC constraints imposed by P, thereby maintaining PPIS as a set of current sources. This iterative double-step is repeated until PPIS is empty, meaning that all its members have been expanded into prime implicants, or removed according to the process described shortly. Then, the correct and complete prime implicant set is in PIS.

The expansion of a PPI to a PI is a depth-first plunge through the search hypercube, using `TestCube` once per edge traversal. The subsequent adjustment to members of PPIS resembles a breadth-first expansion. However, it is not exactly so, because PPIS can contain implicants of several different dimensions, *i.e.* occupying different levels of the original search hypercube. After each iteration, the PPIS constitutes a new *frontier* in the search space, with some of its members occupying new positions closer to the sink vertex.

The `TestCube` primitive is used to attempt edge traversals from legitimate parent to child. The discovery of a prime implicant P of degree k by expansion from a member I in PPIS of degree i, involves $k - i$ successful `TestCube` calls. Then, proof that P is indeed prime involves zero or more failure `TestCube` calls. When new members of PPIS are generated due to new DC constraints imposed by P, each

requires a successful `TestCube` call. There are also zero or more failed `TestCube` calls for this phase of the algorithm.

The set PPIS is equivalent to the set of nodes in the search graph that have indegree zero, after the subhypercubes ending in the already-found prime implicants have been removed. In fact, PPIS alone *implies* the remaining subgraph of the original search hypercube that contains undiscovered prime implicants. Accordingly, before each depth-first expansion of a PPI, the following characteristics of PPIS must be ensured:

1. each implicant is compatible (i.e. a valid implicant)
2. no implicant is a subcube of any PI already in PIS
3. no implicant is a subcube of any other implicant in PPIS
4. all remaining PI's are reachable by expansion of at least one implicant

The relation "subcube" subsumes equality unless it is specified "proper subcube"; similarly, expansion of implicants may not include any real "expansion", if that implicant happens to be prime already.

After an expansion, the members of PPIS are updated to reflect the new DC constraints imposed by the new PI, P. An implicant that is not a subcube of P is not affected at all; it is known that it still satisfies the rules for PPIS. However, if an implicant *is* a subcube of P, then that implicant I no longer satisfies PPIS criterion 2. It is convenient to define a partition of PPIS imposed by P, consisting of the set of *satisfied* vertices S, and the set of *unsatisfied* vertices U. In order to ensure that PI's reachable from an implicant $I \in U$ are all still reachable, I is replaced by all its children that satisfy the above four criteria. All legitimate (compatible) children are included to ensure that all possible expansion paths in the

directed hypercube, originating at I, are still reachable. In other words, the implicant is expanded along each of its unused "required adjacency directions" (RAD's) that correspond to variables that are *not* DC in PI. The union of all legitimate children generated in this way is the set N.

For example, let the base minterm be 0000, and its RAD vector be 1111. This means that 0001, 0010, 1011, and 1000 are all compatible vertices (either minterms, or "don't-cares"). The PPIS is initially {0000}. The RAD vector, used in Chapter 4 for ordering minterms, gives a preliminary reduction in the size of the search hypercube, reducing its dimension from n to the number of adjacencies. Without loss of generality, it is assumed that all n adjacencies initially exist. Then, let P_1 = XX00 be obtained by expansion (the prime implicant found by an expansion of a particular node may depend on the order in which non-DC variables are tested). Then, 0000 gets taken out of PPIS and replaced with its legitimate children 00X0 and 000X. For the second iteration, let 00X0 be chosen for expansion, yielding P_2 = 0XX0. (XXX0 cannot be reached in this expansion, because that would imply that the first PI is not prime.) Then, 00X0 gets replaced by 00XX and X0X0. 000X is unaffected, so it remains in PPIS. Note that the current PPIS set, namely {00XX X0X0 000X}, now violates criterion 3. This phenomenon is discussed in the next section.

It is possible to identify a special case when "expansion" of an implicant I results in a prime implicant $P = I$. In this case, no successful **TestCube** calls occur, and the potential prime implicant I turns out to be prime after all. The significance to HYPER is that no partitioning of PPIS is required at this iteration, because it is known by definition that $S = PPIS$ and $U = \phi$. No change to PPIS is required, except for the removal of I.

The execution time of HYPER is related to the number of **TestCube** primitive calls required, and overhead in maintaining the PPIS (especially the partitioning into

U and S). Thus, time is related to the size and number of prime implicants eventually found. If this number is large HYPER suffers, just as other PIS generation techniques must suffer. The adjustment of each $I \in PPIS$ for each $P \in PIS$ generated, indicates a factor of c^2 in the upper bound for order of complexity, where c is the maximum number of implicants in PPIS. After every iteration (each generation of a prime implicant), the union of the sets PIS and PPIS could be the final prime implicant set, but this is unknown until proven so by the expansion and adjustment, of implicants in PPIS. However, this supports the argument that c is closely related to the eventual number of prime implicants found.

An important strength of HYPER is its easy incorporation of known prime implicants. Whether PI's are obtained through expansion, or by some form of previous knowledge, their effect on PPIS is identical, *viz* remove their corresponding subhypercubes. Thus, an initial phase of the algorithm is to take the known PI's covering the base minterm, and simply adjust PPIS with each, one PI at a time, effectively temporarily avoiding the expansion part of the iterations. This is in strong contrast to the depth-first search of DSA, which must incorporate rules concerning already-expanded minterms in implicants, to avoid redundant generation. A related point is that expansion of implicants and adjustment of PPIS are completely separated in HYPER. In other words, HYPER specifies only how the PPIS is adjusted given a new prime implicant, rather than how that implicant is generated.

A second strength of HYPER is its comparative non-dependence on the ordering of variables. In depth-first search, variable order affects the order in which PI's are discovered, thereby affecting the amount of hypercube that must be examined [McK74]. However, in HYPER the variable ordering still affects the order in which the PI's are discovered, but this order has little affect on the remainder of the algorithm. The creation of child nodes is completely independent of ordering; the same children

exist, up to permutation of input variables.

7.4 Multi-function and Multi-valued Extensions

HYPER extends easily to the multi-function and the multi-valued variations. In the previous discussions, each of n two-valued variables in implicants takes one of the values {0,1,X}. As shown in Chapter 2, this may be considered a shorthand for {01,10,11}, meaning the zero-space, the one-space, and both spaces simultaneously, respectively, of a two-valued variable. In general, an m-valued variable can be represented by an m-vector of 0's and 1's. Any particular adjacency in a variable can refer to any of its m different planes. Instead of converting both 1's and 0's to DC's as implicants are expanded, as in the Boolean case discussed heretofore, 0's are simply changed to 1's in the m-vector.

An adjacency, and by extension an edge in the search hyperspace, is defined by any 0 in the concatenation of all vectors for all variables, that can be turned into a 1. For example, the problem of finding implicants for a four Boolean input variable and three output multi-function minterm 0110 in function 0, would use the hyperspace rooted at the multi-vector (01,10,10,01,001). A single expansion step involves finding some legitimate child with a similar multi-vector, except that one of the 0's in the parent is a 1 in the child. A "complete" expansion is accomplished when no more 0's can be turned into 1's, thereby proving that the implicant is maximal, *i.e.* prime. The dimension of the implicant is defined as the number of 1's in the multi-vector, minus the number of groupings (variables). This reduces to the old definition of degree for the Boolean case.

In MDSA, a distinction is made between the one m-valued variable and the n binary variables, for reasons of efficiency. Expansion of the binary variables starts with

the rightmost potential adjacency, and works to the left. On the other hand, adjacencies in the m-valued variable are always done in parallel because of the particular data structures and accessing methods used in the minimizer. Thus, the choice of whether expansion of some $I \in PPIS$ should occur first in the binary variables, or in the m-valued variable, can sometimes have a profound effect. However, the improvement discussed in the next section (failed adjacencies) eliminates this phenomenon.

7.5 Failed Adjacencies

A very significant improvement in performance is obtained by associating distinct RAD vectors with each implicant in PPIS. The straightforward approach is to use all adjacencies to the base minterm, throughout the PIS generation. However, depending on the origin of an implicant, some of these RAD's can be removed from consideration. Moreover, all the legitimate descendants of an implicant inherit the same reduced set of RAD's as their ancestor. Consider an implicant $I \in PPIS$ that does not satisfy criterion 2. It must be replaced by its legitimate children, by testing adjacency in one variable at a time. These variables are indicated by a RAD vector associated with I. Some of these attempted single step expansions may fail. Since all descendants of I are supercubes of I, *any* RAD found to fail at this stage *must also* fail in all I's descendants. Therefore, the children of I need not inherit the failed RAD's.

Use of failed adjacencies limits the dimension of each search hypercube, rooted at each implicant in PPIS, to a minimum. With HYPER, it is in fact not necessary to calculate the single-minterm adjacencies (Hamming distance one) beforehand, as is done in DSA [McK74]. Instead, the source node could be assigned RAD's corresponding to all its variables, since it will be split up properly at the first iteration, with the removal of each failed adjacency. The situation is inferior in DFS as described in

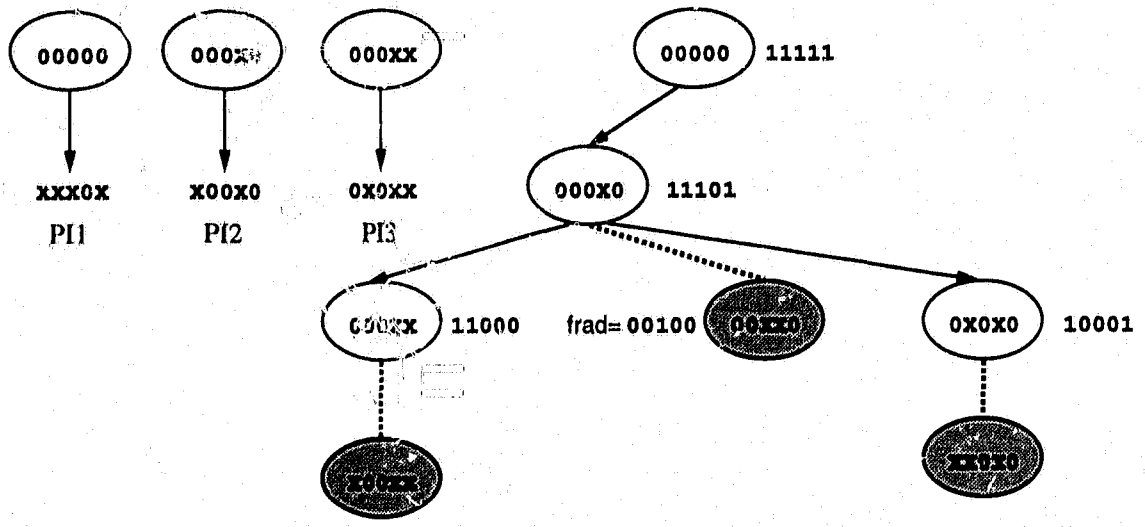


Figure 7.3: Failed Adjacencies

[McK74], since all of the same HD=1 adjacencies are used throughout the traversal.

An example of the use of failed adjacencies is given in Figure 7.3. In this instance, the complete set of prime implicants covering minterm 0000 is {XXX0X,X00X0,0X0XX}. The nodes (cubes) in Figure 7.3 are augmented with their adjacency vectors, given to their immediate right. Nodes found to be illegitimate are shaded. The first source node, comprising the entire PPIS, is 00000. It is found to be adjacent in all five variables, so its RAD vector is 11111. The HYPER algorithm does not impose any particular order of expansion, so let the first prime implicant discovered be PI1=XXX0X[1]. Then, node 00000 is unsatisfied, and is replaced with one child 000X0 with RAD 11101. (The one missing RAD corresponds to the variable in which expansion just occurred.) Next, PI2 is discovered by expansion of this child, and then 000X0 must be

[1]In MDSA, cube expansion happens to be right to left, so the first PI discovered would actually be 0X0XX.

split into potentially three children. However, one is found to be illegitimate because of a failed RAD. Its legitimate siblings inherit this failed RAD. Thus, when 000XX is expanded to find the third prime implicant, there are only two rather than three different adjacencies to check. Discovery of PI3 leads to the subsequent discovery of the last two illegitimate children, terminating the search.

The concept of failed adjacencies does not apply so easily to DFS. When a single expansion occurs successfully in DFS, no failed adjacencies are detected. If an attempted expansion fails, other descendants of that node will never test that adjacency again anyway. The failure cannot be applied to ancestors of the current node, which will be expanded in other directions sometime in the future, because the current node is of larger degree. In other words, the failure may be due to an incompatibility involving only the variable corresponding to the current node's incoming edge. Thus, the failed adjacency *could* exist in the node's ancestors, for all the algorithm knows.

7.6 Elimination of Redundancy

The satisfaction of criterion 3, repeated below, is not guaranteed in the discussion of Section 7.3.

- no implicant is a subcube of any other implicant in PPIS

As a matter of fact, it is possible that criterion 3 is violated whenever the PPIS is adjusted by a prime implicant P. Let the PPIS before an adjustment consist of *old* implicants, and let the legitimate children of old implicants that do not satisfy the non-subcube criterion (criterion 2), be *new* implicants. Then, the old implicants are partitioned into U and S, and legitimate children of U become the set of new implicants $N \cup R$. The next PPIS, after the adjustment imposed by P, is then $S \cup N$,

and it satisfies the four criteria of HYPER PPIS sets. The set R of redundant children must first be found, preferably before they are born.

Any redundancy introduced into an otherwise legal PPIS must involve a new implicant. Five potential conflicts may be identified (the first three cases refer to an old implicant other than the new implicant's parent):

1. an old implicant is a subcube of a new implicant
2. an old implicant is equal to a new implicant
3. an old implicant is a supercube of a new implicant
4. an new implicant is a subcube of a new implicant
5. an new implicant is equal to a new implicant

Claim: Given that the four HYPER criteria hold prior to discovery of a new prime implicant, redundancy can only occur in case (1) above, *i.e.* $r \in R$ is a proper supercube of some $I \in U$.

Proof: The example of Figure 7.4 suffices to show that case (1) is indeed possible (see below). Thus, it is only required to disprove cases (2) through (5). Cases (2) and (3) are trivially impossible, because they both imply that the parent of the new implicant would have to be a subcube of the old implicant, which is not permitted by definition (criterion 3). In cases (4) and (5), the two new implicants would have to have distinct parents, both in U. The children differ from their respective parents by one more DC variable each. If this variable was the same for both children, the PPIS prior to adjustment would violate criterion 3. If the expansion variable is different for each child, they cannot be related, because neither would be born in the first place if they had a DC in the other's variable of expansion.

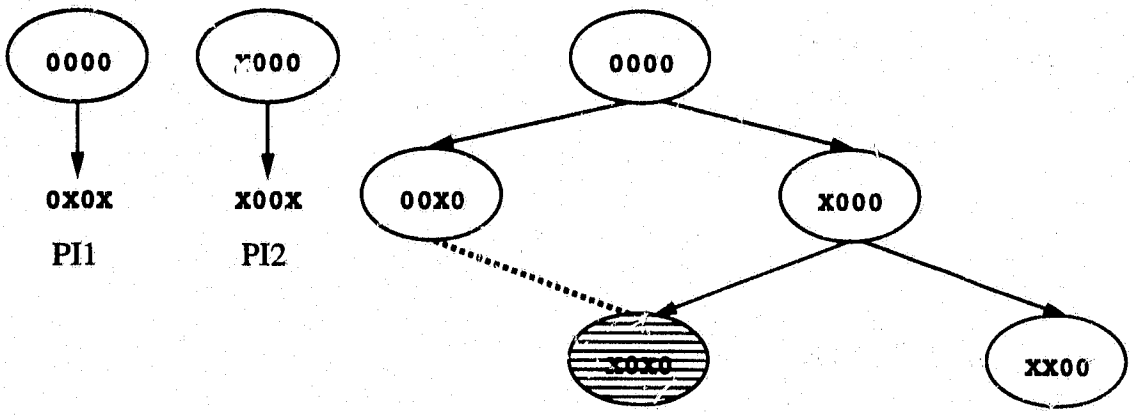


Figure 7.4: Removal of a Redundant Child

Figure 7.4 illustrates redundancy in potential prime implicant sets. This time the first prime implicant discovered is 0X0X, resulting in the two children 00X0 and X000 for parent node 0000. Assume X000 is next chosen for expansion, resulting in the second prime X00X. Of the two siblings currently comprising PPIS, one satisfies the DC constraints of PI2, and one does not. The latter would apparently be split into its two legitimate children, but one child (shaded in Figure 7.4 with horizontal lines) turns out to be a subcube of the satisfied node. It is therefore redundant, and therefore should not be generated.

HYPER avoids the removal of redundant children from set N by ensuring that they are not conceived in the first place. Upon expansion of implicant $I \in PPIS$ to prime implicant p, and assuming that at least one expansion occurred, PPIS is first partitioned into the sets S and U. Then, the available adjacencies for procreation of each element of U into legitimate children are diminished be comparing each member of U against each member of S of equal or less degree. This is accomplished by associating a new RAD vector with each element u of U. This vector is initially

computed by taking u's RAD's, and obtaining the bit-wise AND of these RAD's with a vector indicating the non-DC's of p. The new vector effectively gives the children of u that potentially would be members of N. Only after the elimination of these RAD's by comparison against S, are these children actually born.

7.7 Results

The HYPER algorithm is designed specifically to compete with DFS for prime implicant set generation. Therefore, this section is concerned primarily with aspects of the PIS generation process alone, rather than overall two-level minimization. Results are divided into three main categories:

- effect on single prime implicant sets
- behaviour under variable permutation
- results with "real-world" examples

All tests use the MDSA program, which can be flagged in the command line to perform either DFS or HYPER for PIS generation. For each of the above three test classifications, identical examples were run with DFS, then HYPER, and their respective execution times recorded (in conjunction with other pertinent data).

7.7.1 Single Prime Implicant Sets

In the absense of any other information, MDSA chooses the lexicographically "lowest" minterm for initial expansion. Thus, the easiest way to use MDSA to test the DFS and HYPER algorithms on single prime implicant sets is to provide as input a set of

			DFS			HYPER		
file	cubes	max	succ	fail	time	succ	fail	time
14c2	91	13	183	1456		104	1092	
14c4	1001	715	1004	2002	1.38	1364	10010	.35
14c6	3003	1716	3008	3432	10.40	5004	24024	3.56
14c8	3003	1287	3010	2002	9.87	6434	18018	7.63
14c10	1001	286	1010	364	1.79	3002	4004	5.37
14c12	91	13	102	14	.26	454	182	1.08
16c4	1820	1365	1823	4368	5.24	2379	21840	1.06
16c12	1820	455	1831	560	12.50	6187	7280	55.70
16-20%	335	589	552	7043	1.40	1935	3830	.61
16-40%	283	677	518	9865	4.94	3323	4977	2.19
18-20%	563	1064	956	19644	6.22	4016	13249	2.23
18-60%	96	231	231	3811	18.40	1409	1242	6.00

Table 7.1: DFS vs HYPER for Single PI Sets

implicants, each of which covers minterm $000\cdots0$. Two generator programs, which produce lists of n variable cubes belonging to the ON-SET of a single-output Boolean function, were written. The first generator takes two parameters, n and k, and produces all n-cubes containing k DC variables, with the remaining variables set to 0. Thus, the output is a list of $\binom{n}{k}$ prime implicants covering vertex $000\cdots0$. The second generator takes three parameters, n, p and q, where p is a probability that any given variable in a cube is a "don't-care" variable (rather than 0), and q is the number of cubes generated. In this case, the number of prime implicants found is always less than or equal to q, with a more random distribution of "don't-cares" in each implicant.

Both cube generators are used in the experiment whose results are given in Table 7.1. The topmost eight rows are for PLA's with n equal to 14 or 16 variables, and from k equal to from 2 to 12 DC's. The remaining rows are for PLA's with 16 or 18 variables, and a given percentage probability that any one cube position is DC.

The parameter q is not listed, nor is it relevant, because cube lists are converted by MDSA into truth table format regardless of input format. The second column of the table gives the number of prime implicants in the PIS covering the zero minterm; this is always $\binom{n}{k}$ for the first generator, and some number between 1 and q for the second. The third column gives the maximum size that a HYPER PPIS set expands to while finding the PIS.

The first two columns of each of the DFS and HYPER sections of Table 7.1 give the number of calls to the `TestCube` primitive required by each algorithm. This primitive is used both in implicant expansion, and in the birth of legitimate children in HYPER from the set of unsatisfied implicant parents U. Successful calls means that one DC was successfully added to an implicant, and failure calls means that the implicant was proven prime. It is important to make the distinction between these two cases, because failure calls are the most wildly fluctuating numbers. Finally, there are the "time" columns which give the time, in seconds, required to generate the PI sets. All experiments reported in this section are performed on a SUN4 SPARC1 computer.

The main weakness of HYPER is its apparent $O(c^2)$ behaviour, which becomes noticeable with large implicant sets. This is even more of a problem when the cubes involved get very large. In particular, examination of the results for `14c2` vs `14c12`, `14c4` vs `14c10`, and `16c4` vs `16c12`, reveals this problem. Within each pair, the PIS size is the same, but there is discrepancy in the ratios of DFS time to HYPER time. However, the PLA's generated by the $\binom{n}{k}$ generator are peculiar in their homogeneity: all cubes are the same size and variable permutation is meaningless.

If attention is diverted to the lower four rows of Table 7.1, it is evident that HYPER is much further ahead (by about a factor of three, in fact). However, these implicant sets are generally smaller than those in the upper part of the table. The

distribution of cubes in the sets is strange indeed; it would be informative to include data concerning the distribution of "don't-cares" for each example. The average number of DC's per cube is close to the probability p specified in the command line, but the variance is quite high.

7.7.2 Permutation of variables

HYPER performs best with small prime implicant sets, consisting of very large cubes. This is the very situation where DFS is weak. Both DFS and HYPER have a dependence on the order of variables given in the input PLA specification. This order has no theoretical relevance to the minimum solution, nor does it have any effect on the prime implicant set of a given minterm. Both algorithms expand cubes by attempting to fill in DC's from the right to the left, with each expansion requiring one `TestCube` primitive. The imposed order of expansion has relatively little effect on HYPER, but is critical to DFS.

Below is a nineteen variable example that exhibits this behaviour. It is given in the *espresso* PLA format, and is based on an actual phenomenon observed in the MDSA minimization of benchmark PLA `in2`.

```
.i 19
.o 1
0------------------ 1
--0000000000000---- 1
-0----------------- 1
.e
```

The effects of regenerating these three cubes using both algorithms, and permuting the variable positions, is shown in Table 7.2. The first row reflects the order of

	DFS				HYPER			
perm	total calls	success probes	failure probes	time	total calls	success probes	failure probes	time
1	131.0K	393K	130M	188.00	69	524K	263K	1.03
2	65.5K	459K	43M	64.40	69	524K	131K	.87
3	32.8K	492K	14.5M	21.50	69	524K	65.7K	.80
4	16.4K	508K	4.82M	7.10	69	524K	32.9K	.74
5	8.2K	516K	1.61M	3.10	69	524K	16.4K	.73
18	53	524K	52	0.77	93	524K	112	.78

Table 7.2: DFS vs HYPER when Permuting Variables

variables as given in the three cube *espresso* style specification above. Each subsequent row corresponds to a rotation of the rightmost variable to the leftmost position. The last row corresponds to the permutation given below, which is the one most advantageous to DFS.

```
.i 19
.o 1
-----------------0- 1
----0000000000000-- 1
------------------0 1
.e
```

The first column of each main vertical section of Table 7.2 gives the number of `TestCube` calls required for each experiment. It should be noted that the successful calls required is always less than sixty for these examples, thus the first column is almost all failure calls. The next two columns of each table section gives the total number of truth table probes required by all the `TestCube` calls. Because MDSA uses a truth table data structure, the expansion of an implicant by one DC is exponentially proportional to the number of DC's already existing in the implicant. For example, if the implicant `00XXX0X` is to be expanded to `00XXXXX`, the implicant `00XXX1X` is tested

for compatibility via one `TestCube` call, requiring $2^4 = 16$ probes. This is relevant for several reasons. Firstly, the amount of time required for a `TestCube` call in MDSA is far from constant throughout a minimization. With a set-based approach, the time would be much closer to a constant. Secondly, when an implicant of degree d is expanded to one of degree $d+1$, if information from known compatible implicants in PIS or PPIS were used, less than 2^d probes would be required. Neither DFS nor HYPER exploit this possibility. Finally, failure `TestCube` calls often require much less than 2^d probes; the call is terminated upon probing the first minterm whose value is 0. There is, therefore, a random luck factor dependent on the order in which minterms are probed.

The DFS times in Table 7.2 show a clear degradation of performance as the specified variables (*i.e.* variables that are not DC) move to the left. This is borne out by the "total calls" and "failure probes" columns. On the other hand, HYPER is minimally affected by these permutations. As a matter of fact, the number of calls remains constant (except for the last column, which is different only because of the different order in which the three prime implicants are discovered). The testing of many of the benchmark PLA's supports this observation. MDSA has a command line flag which specifies that the input variables should be randomly shuffled. Also, MDSA can read from the input PLA file a "`.j`" line which contains a permutation of the integers 1 to n. The n variables are then shuffled according to this permutation, before minimization commences. Certain PLA's (especially `dk48`) show wildly varying PI set generation times when their input variables are shuffled, using MDSA with DFS. With HYPER, generation time remains comparatively constant.

7.7.3 Benchmark PLA's

PLA	n	m	c	DFS	HYP	TOT
alu1	12	8	19	.34	.10	.50
dk48	15	17	22	.29	.28	1.30
in1	16	17	105	1.00	.42	3.80
*in0	15	11	107	.83	.26	1.70
*in2	19	10	134	11.70	2.30	18.30
*misex3	14	14	687	25.90	9.50	83.80
*misex3c	14	14	203	682.00	476.00	719.00
*sym10	10	1	212	5.10	3.10	17.60
*sym9	9	1	84	1.70	1.30	4.20
*tial	14	8	576	3.10	1.20	9.40
add6	12	7	355	.46	.53	1.90
add7	14	8	735	2.20	2.80	5.40
*apex4	9	19	428	.10	.40	2.40
*sev	8	10	200	.14	.28	1.62

Table 7.3: Comparison of HYPER and DFS on Benchmark PLA's

Finally, how well HYPER does over entire minimizations is examined. Table 7.3 includes only those PLA's that (a) have less than 20 input lines (variables), and (b) have PIS generation times of $100ms$ or greater (the latter because small times are unmeasurable). Some of these PLA's involve the generation of hundreds or even thousands of PIS's, in a single minimization. The table is split horizontally into two parts; the lower part contains those examples where DFS outperforms HYPER. The first four columns are descriptors of the PLA's themselves: name, input lines n, output lines m, and number of cubes in best known minimization c. The next two columns give the total over all PI set generations for each minimization, for each PI set generation algorithm. (Ordering of minterm choice in MDSA is independent of PI set generation.) The last column indicates the total minimization time for the run performed with HYPER, including the time required by HYPER[2]. Thus, these figures

[2]The figures reflect the use of optimizations discussed in Chapter 8.

give an idea of how significant PI set generation time is to the overall minimization. With HYPER, generation time runs from about 10% to 60% of overall minimization time, averaging around 20%.

It is important to note that the two PLA's exhibiting the greatest ratio of HYPER time to DFS time, *i.e.* **sev** and **apex4**, are both relatively small PLA's where the PIS generation time is quite low anyway. The significance of this is that the longer time required by HYPER may be attributed to the overhead of maintaining the potential PI set, which becomes a dominating factor when the expansion of cubes takes very little time. Also, **apex4** has $m = 19$ output lines - more than any other test PLA. Whether or not this is relevant to DFS vs HYPER comparisons is unclear.

7.8 Summary

Depth-first search as a prime implicant generation strategy is subject to great increases in complexity, due to circumstances other than implicant sizes and number found. This is primarily due to searches through largely redundant areas of the hypercube search space which cannot be removed by pruning or discarding branches, since these areas could include undiscovered prime implicants. The new HYPER algorithm avoids this issue by keeping only those parts of the search space that could potentially produce new prime implicants. Thus, HYPER is affected only superficially by different permutations of the same input variables. The price is increased overhead for updating the search space data structure.

The discovery of new prime implicants works in tandem with, yet is separated from, the maintenance of the search space. This separation of concerns permits easy incorporation of previously discovered prime implicants. Test examples with large implicants and small prime implicant sets exhibit especially substantial speed-up with

HYPER. The worst performance of HYPER with respect to DFS apparently occurs when either (a) PI set generation times are small, or (b) cubes are large *and* PI sets are large. In the first case, the performance of the generation part of the MDSA minimizer has little relevance, and the second case is rare because of the difficulty in constructing examples that have *both* large implicants and large implicant sets. (The largest PI sets contain implicants of degree averaging about $\frac{n}{2}$.) On average, HYPER takes about 33% to 50% as long as DFS for PI set generation, for the available benchmark PLA examples.

There remain at least three main areas in which any future work should be directed:

- retesting of known incompatible children
- redundant probing
- choice of potential prime implicant for expansion

The first point refers to the fact that when a prime implicant is found, HYPER is unable to remove its descendants. Some of these descendants, which must all be illegitimate, are reachable from implicants in the PPIS. In other words, HYPER ensures that implicants themselves are not revisited, but cannot extend the same guarantee to illegitimate children of prime implicants. This is likely the prime reason for the rapid growth in number of failure `TestCube` calls, as PI sets become large. One way of solving this problem amounts to introducing the opposite of DC constraints, say "specification constraints", that would say that certain variables in an implicant must remain specified. The idea of failed adjacencies helps here, but is not complete. It does not seem possible to encode specification constraints in a single vector, except in some special cases where an implicant in PPIS is separated by a prime implicant in PIS by Hamming distance one.

Another way of handling the retesting problem is to partially remove the distinction between the PIS and PPIS. When prime implicants are discovered, they are moved by HYPER into PIS, so that they are not involved in any subsequent searches over PPIS. However, PIS implicants are, by definition, always in the set of satisfied implicants, no matter what the current expanded PI is. Now, if these prime implicants were to remain in PPIS as implicants with zero available adjacencies for expansion, then some of the failure `TestCube` calls from splitting unsatisfied implicants in U will not occur, when the potential children are aborted because they would be illegitimate children of satisfied implicants in PIS. Letting $S' = S \cup PIS$, it is possible to trade more implicant comparisons between members of U and S', for fewer failure `TestCube` calls.

The second problem area concerns the fact that the truly unknown portion of a potential implicant of degree d is often smaller than degree $d-1$. In these cases, use of `TestCube` would require fewer map probes. This is also an unresearched problem area in DFS, and again one with great potential benefit in map-based DSA algorithms.

The last problem area concerns the choice of PPIS for expansion into a PI, at each main iteration of the algorithm. HYPER picks the one of lowest degree (and first in the list among those) by default, but can be flagged to pick the largest. This option does slightly better on average, though is not significant. Other possibilities include choosing $I \in PPIS$ with fewest available expansion RAD's, or I with potentially the least likelihood of producing a large set N. Of course, it is also possible that it is actually desirable to choose nodes that could potentially undergo many expansions, or those that increase the size of PPIS most. None of this is known.

HYPER is still slower than DFS in certain situations. A hybrid algorithm, using both DFS and HYPER, may be appropriate, so that an explosion in the size of PPIS could trigger a more depth-first oriented approach for the generation of the remaining

prime implicants. For example, when the PPIS reaches a certain size, DFS can be performed from its elements, and PPIS updated according to all the prime implicants discovered from each element. Thus, every main iteration of the HYPER algorithm would involve updating PPIS according to a *set* of prime implicants, rather than a single one. Other possibilities include basing searches on breadth-first expansions, or finding a way to incorporate failed RAD's in DFS.

Chapter 8

Minterm Dominance

8.1 Introduction

The inclusion of *minterm dominance* techniques in the MDSA minimizer is the subject of this chapter. Minterm dominance does not correspond to one particular directed search algorithm step. Instead, the introduced techniques reduce the complexity of other steps, whilst simultaneously increasing the "correctness" of the minimized output. The benefits arise from the identification and removal of certain minterms, which are *not* covered by any cubes selected thus far, but are redundant nonetheless. Minterm dominance is by no means a new idea [Giv70], but its incorporation in directed search algorithms certainly is.

8.2 Partial Prime Implicant Tables

Some early exact two-level minimization methods involve the use of *prime implicant tables*, where table rows correspond to prime implicants, and columns correspond to

minterms [Giv70][1]. Tables are meant to solve the second phase — selection of a covering subset of prime implicants — of the classical two-phase approach to minimization. A table entry corresponding to cube C and minterm M is 1, or *marked*, if and only if the cube C (row) covers (*i.e.* contains) the minterm M (column).

Essential minterms correspond to columns with only one 1; the intersecting row then corresponds to an essential cube. Dominated cubes are recognized by rows not having a 1 in any column position other than those occupied by some of the 1's in another (dominating) row. Dominated rows can be removed without affecting minimality. When they are, intersecting columns each lose a mark, thereby potentially creating one or more pseudo-essential minterms, in turn leading to the identification of new pseudo-essential cubes. Columns corresponding to minterms covered by essential and pseudo-essential cubes, are removed along with the rows corresponding to the cubes.

The operation of the directed search algorithm, including both DSA and MDSA variants, actually imitates some aspects of this process. A snapshot of the state of DSA at a particular iteration is given by an incomplete, or *partial* prime implicant table. The columns correspond to expanded and recursive minterms in the EMTS and RMTS sets, and the rows to potentially cyclic cubes in the PCPI. Conceptually, the "table" may be divided into two vertical sections. One section contains those columns that are known to be *complete*, *i.e.* all marks, corresponding to cubes covering these minterms, are in existence. This first section corresponds to the set of expanded minterms. The other section contains the columns that are not known to be complete, corresponding to the recursive minterms.

Every time a cube is added to the PCPI, it is like adding a new, complete row

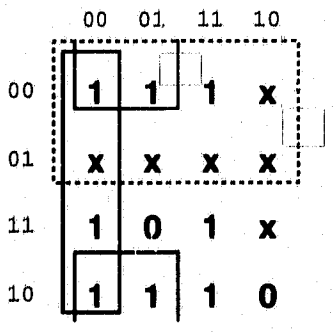
[1]The interpretation of rows and columns is interchanged in some of the literature, *e.g.* [RSV87].

to the bottom of the partial prime implicant table. Addition of new columns to the second vertical section corresponds to newly discovered recursive minterms, moved from the set of new minterms because they are covered by one or more newly generated cubes. However, these columns are *not* complete; the complete set of prime implicants covering a recursive minterm is unknown until the minterm is expanded. Every directed search iteration involves either the start of a new partial table (selection of an NMT), or movement of one RMT from the right of the table, to become an EMT at the left of the table, with its column completed by the addition of new cubes (rows).

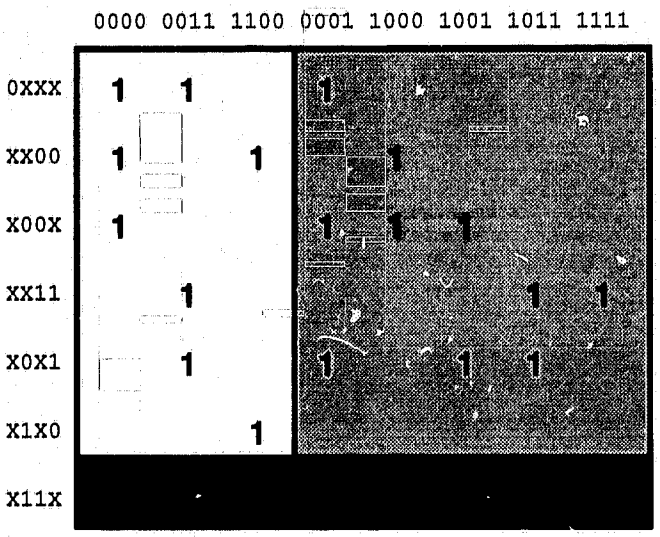
Figure 8.1(b) shows the partial prime implicant table corresponding to the first three expansions in a directed search minimization of the function given in Figure 8.1(a). (The cubes illustrated in Figure 8.1(a) may be ignored for the moment.) In this example, it is assumed that new minterms are selected for expansion in the order `0000`, `0011`, and `1100`. The columns to the left of the vertical dividing line in the table correspond to the EMT's; those to the right are RMT's. The incomplete portions of the prime implicant table are shown lightly shaded, and the as yet undiscovered portion is shown darkly shaded.

Expansion of the third minterm `1100` results in the discovery of one more new prime implicant `X1X0`. However, the latter's row is dominated by the second row: cube `XX00`. Thus, the new cube is subsequently removed from the table, which in turn leads to the discovery and selection of cube `XX00` as sole covering cube for newly pseudo-essential minterm `1100`. All the resulting simplification occurs without complete knowledge of the shaded portions of the table in Figure 8.1(b).

Classical methods of prime implicant table reduction also include the removal of *dominating columns*. When column A dominates column B, the corresponding situation is that all cubes that cover minterm B also cover minterm A. That is, when



(a) Four variable map



(a) Prime implicant table

Figure 8.1: A Partial Prime Implicant Table

a cube to cover B is found - and this must happen eventually - it is known that A will be covered also. The dominating column/minterm may be removed without affecting the correctness of the minimization, thereby reducing the complexity of the table.

In Figure 8.1(b), the first column (minterm 0000) dominates the fifth (minterm 1000). However, the latter minterm is still just an RMT; it is not known whether or not there are more undiscovered cubes that cover it. Had 1000 been expanded earlier, it would have been possible to discover that its column is indeed dominated by that of 0000, thereby indicating that the latter minterm is redundant. The cubes involved in these minterm dominance relations between the two minterms are drawn in Figure 8.1(a). The two common cubes are shown outlined, and the one extra cube covering 0000 is shown dotted.

Previous versions of the directed search algorithm do not incorporate minterm domination in any sense. Yet the elimination of these redundant minterms is potentially extremely useful. Firstly, the number of RMT's, as candidates for expansion, is reduced. Secondly, cube dominance relations arise more frequently, because the cells of cubes are in general smaller. All that is required to fit minterm domination in with other MDSA operations is a way of casting the technique in the language of the MDSA data structures and primitive operations. In a sense, the cells in MDSA *implicitly* encode the 1 entries in rows of a virtual partial prime implicant table. Cell containment relations among cells imply dominance relations among rows. What is needed is another implicit encoding for dominance relations among minterms.

8.3 Prime Implicant Set Intersections

It is not difficult to devise a form of minterm domination for MDSA. When the prime implicant set of a selected minterm is first generated, the *intersection* (or product)

of these cubes' cells is calculated. Note that this adds only incidental complexity to the generation of primes, since their cells must each be calculated anyway. The cell of intersection always contains the selected minterm, and in the trivial case, no other minterms. However, if the intersection cell is non-trivial, all other minterms covered by it are dominating, and can therefore be changed to "don't-care".

Returning again to the example of Figure 8.1(a), the two outlined cubes cover minterms 0000 and 1000. In the example directed search minimization, 0000 is expanded first, and the third covering cube 0XXX also belongs to the set of covering prime implicants for that minterm. If in an alternative directed search minimization, 1000 is expanded first, then only the two prime implicants are in the prime implicant set. The intersection of these two cubes' cells is X000. Thus, minterm 0000 is known to be dominating, and therefore redundant.

The intersection cell, minus its generating minterm, is treated just as a selected pseudo-essential prime. Affected cubes in PCPI are found, and new dominance relations among cubes uncovered. Thus, each main iteration of MDSA may potentially remove one or more minterms, even if no pseudo-essential cubes are found. Intuitively, the probability of finding a non-trivial intersection cell is inversely proportional to the number of cubes in a minterm's prime implicant set. The greater the number of cubes, the smaller the intersection. Thus, it remains desirable to choose minterms likely to have few covering cubes.

Figure 8.2 shows the directed search minimization of a simple four-variable function, where prime implicant set intersections are used to reduce the number of minterms. A three cube minimum solution is found in four iterations, illustrated in Figure 8.2(a) through Figure 8.2(d). Each iteration results in the discovery of one redundant minterm (marked by + signs in the figure), due to prime implicant set intersection cells of dimension one (shown shaded). When the first pseudo-essential minterm is

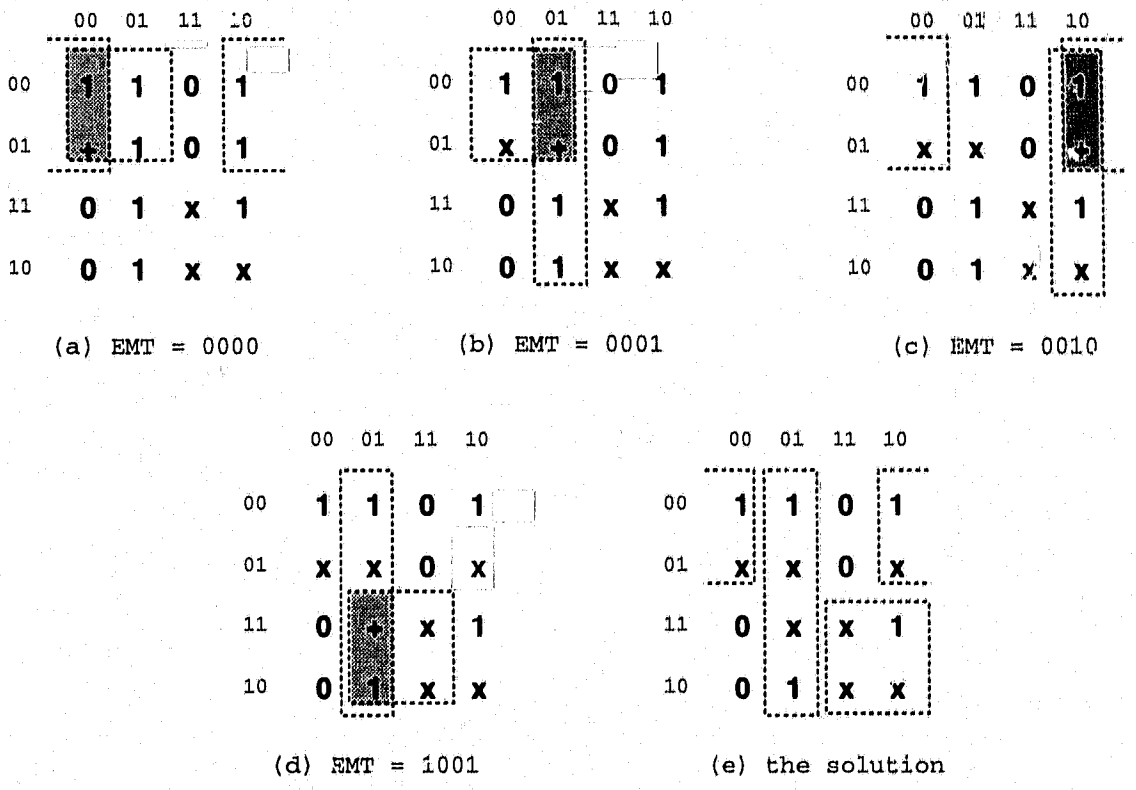


Figure 8.2: Redundant Minterms in Prime Implicant Set Intersections

finally discovered in Figure 8.2(d), the set of recursive minterms RMTS contains only one minterm, 1110. A chain of pseudo-essential discoveries in the fourth iteration result in the solution depicted in Figure 8.2(e).

8.4 True Cycles and Pseudo-Cycles

The chief benefit of removing dominating minterms contained in PIS intersections is that fewer minterms make it to the RMT and EMT sets. Another benefit is more subtle, but important with respect to minimality. McKinney and Dueck both describe cycles in terms of the DSA algorithm itself, *i.e.* a cycle exists when the RMTS is empty, but the EMTS and PCPI are not [McK74, DM88]. However, these "cycles" are often larger than those that would be obtained after the removal of dominating minterms. A more accurate definition of a cycle is the existence of a non-empty EMTS or PCPI, *after* full prime implicant table reduction has taken place.

In fact, there exist DSA "cycles" that do not even contain a non-empty irreducible cyclic core. Here, such phenomena are defined to be *pseudo-cycles*. A good example of a pseudo-cyclic function is found in [Ser84b], and is repeated here in Figure 8.3(a). The complete prime implicant table for this five variable single output function contains no cube dominance relations, and each minterm is covered by at least two primes. Therefore the table is apparently "cyclic" in the DSA sense. However, there do exist some minterm dominance relations. The original function does not include any "don't-care" vertices, but in MDSA some are changed to that category using PIS intersection (marked as + signs in the figure).

Other directed search algorithms report seven cubes, using the heavy-cube heuristic to break the DSA "cycle" after expansion of all twenty minterms [Ser84b, DM88]. The minimum solution actually consists of only six cubes, all of which are pseudo-

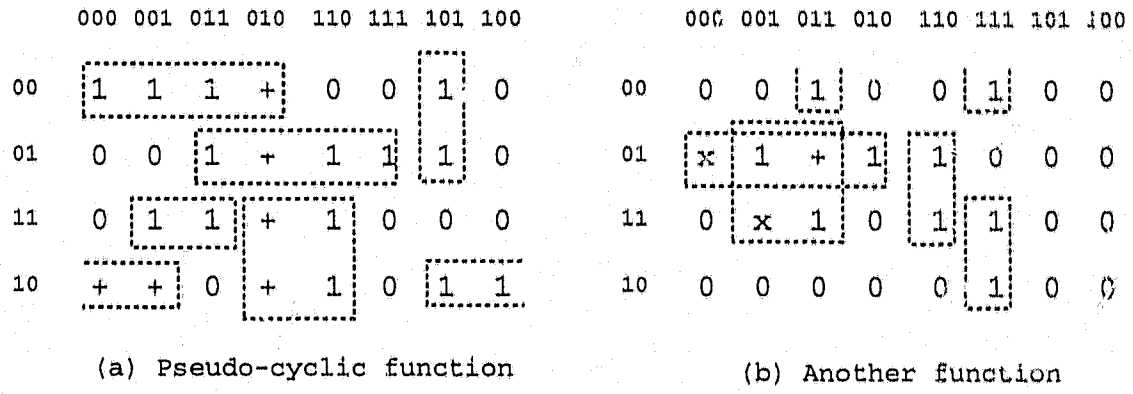


Figure 8.3: Two Pseudo-cyclic Five-variable Functions

essential, which is evident only after minterm dominance relations are exploited. The solution is illustrated in the figure. MDSA augmented with the calculation of prime implicant set intersections, finds the six cube solution in only twelve iterations, detecting no cycles. Of the total sixteen prime implicants, the improved MDSA generates fifteen.

Another five variable pseudo-cyclic function is shown in Figure 8.3(b). This one differs from the previous example in that solutions reported by all variants of the directed search algorithm, each comprise five cubes. Thus, no penalty in size of cube cover is incurred by being unable to recognize the pseudo-cycle. The function is included here to demonstrate that the lower bound for the number of iterations that MDSA requires, is not always attainable when using minterm domination. The three pseudo-essential cubes, shown unshaded, are discovered only after minterm `01011` is removed, because it is contained by the prime implicant intersection cell of minterm `01001`. This accounts for a minimum of four expansions already; one for the discovery of the sole dominating minterm, and one each for the three resulting pseudo-essential cubes. However, two more expansions are still required: one each for

01010 and 11011. Neither shaded cube 010XX nor X10X1, which are both needed for the minimum solution, can be determined to be psuedo-essential for their common minterm 01001, because of the presence of these other two. Thus, the lower bound is no longer sharp, in contrast to that derived in Chapter 3 for DSA.

8.5 Joining Cube and Minterm Dominance

Removal of prime implicant set intersection cells may or may not affect cubes in the PCPI, but all cubes of the PIS are affected. By definition, the intersection cube intersects with every member of this latter set, so in fact all the PIS cubes are in the *affected* category when the cell of intersection is removed. Thus, their cells are possibly smaller than originally calculated. Whenever one or more dominating minterms are removed, it is necessary to traverse both the PCPI set, and the current PIS (which at this point of execution, are disjoint). Each cube of PIS, and some members of PCPI, must have its cell recalculated, to see if it is a member of the *contracted* category. Contracted cells might possibly be dominated.

In order to ensure that no dominated cubes could possibly exist after a main iteration is finished, the determination of affected, contracted, and dominated cells is itself iterated until no new changes occur. Whenever a PIS cube is found to be contracted, the search for a dominator is confined to the other members of PIS, since members of PCPI do not contain the expanded minterm, and are therefore ineligible as dominators. When all dominated members of PIS are removed, the intersection cell of PIS is potentially affected. If the cell has changed in dimension, it must have grown bigger; it belongs to the *enlarged* category[2]. This category for affected intersection

[2]A more appropriate name for the category might be *expanded*, but this term already applies to the class of expanded minterms.

cells is a dual of the contracted category for affected prime implicant cells.

The efficiency of the recalculation of intersections is enhanced by calculating not only the intersection of cells, but also the intersection of the prime implicants themselves. The *intersection cell* I_{min} of a set P is defined to be the intersection of all the cells of cubes of P. Similarly, the *intersection cube* I_{max} of P is the intersection of all the cubes of P, where each cube is prime. Any minterms found to be in an enlarged cell must also be contained in the corresponding cube. If any such previously undiscovered dominating minterms actually exist, the intersection cube must also be enlarged, because the previous intersection cell did not contain any. Thus, if I_{min} enlarges, but I_{max} does not, no new dominating minterms exist, and therefore no change occurs. The same is true when a newly-calculated I_{min} is no larger than the previous I_{max}.

These concepts are illustrated in Figure 8.4. The two five-variable Karnaugh maps are identical, and show the covering prime implicant set for minterm 01010 (row 01, column 010). At the time of 01010's expansion, three covering prime cubes are found, with intersection cube 0101X and intersection cell 01010 (the degenerate case). The three covering prime cubes are shown on the left, and the corresponding covering cells are shown on the right. No searching for dominating minterms to remove is necessary, because the cell difference 0101X − 01010 = 01011, must be "don't-care".

Now, assume that minterm 01110 is covered by some other cube. Then, the cube 01X1X, with cell 01X10, becomes dominated, and is removed. This causes the intersection cube and cell to expand to the shaded areas of the figure. However, since the original intersection cube I_{max} is 0101X, it is already known that no new dominating minterms could exist. Moreover, the intersection cube I_{max} is now 0X01X (shown shaded in Figure 8.4(a)), so it is known that no dominating minterms exist in that space, also.

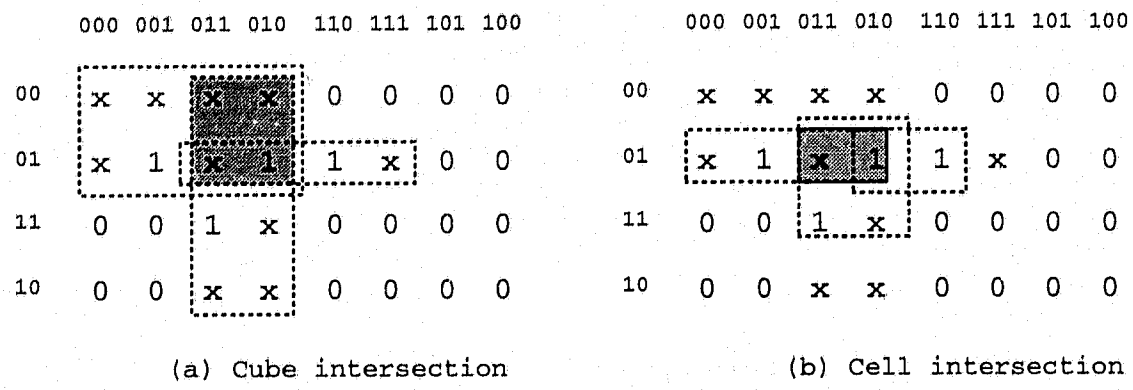


Figure 8.4: Expansion of Intersection Cubes and Cells

Another way efficiency is improved is by using shortcuts when recalculating intersections, much in the same way as shortcuts are used in cell calculations (cf. Chapter 5). In particular, if while recalculating an intersection cell, the cell becomes as small as the previously calculated one, then the recalculation can halt; no change has occurred. This technique has little relevance when the intersection is first calculated, but can save time for subsequent intersection recalculations for the same minterm. The tree holding the PIS need be traversed only until the incrementally calculated intersection cell contracts to the size of the previously calculated one.

8.6 Comprehensive Reduction

Not all dominating minterms are detected by the techniques described above. While it is true that the idea of prime implicant set intersection fully covers all cases of minterm domination, the problem with the approach so far is that the intersection is found only at the time of minterm expansion (although several recalculations during the same main iteration may be necessary before all involved dominance relations have

been uncovered). When dominated cubes covering already-expanded minterms are removed in subsequent iterations, the intersection cells of these minterms (members of EMTS) may grow larger, incorporating more dominating minterms. It is possible to report the existence of a true cycle only after it has been ascertained that the intersections of *all* the expanded minterms prime implicant sets have *not* grown.

It is convenient to distinguish between *primary dominance*, which permits the removal of *some* dominating minterms, and *secondary dominance*, which accounts for the other dominating minterms, not detectable in the initial cubes of intersection. Primary dominance refers to the techniques used to remove minterms at the time of prime implicant set generation, described in the previous two sections. Hence, this form of dominance is not a static quality of a function; it depends on the order in which prime implicant sets are generated. If a minterm is expanded late in a minimization, it could have a smaller initial set, and could therefore have a larger intersection cell, than would have been found, had the minterm been expanded earlier.

Secondary minterm dominance makes sure that prime implicant set intersection cells contain *only* their generating minterm, at all times. The technique is accomplished by noting when the number of cubes covering an expanded minterm decreases, because of the removal of a dominated cube which covers it. Such expanded minterms are classified as *affected*. They are easily recognized when their cover counts are adjusted. At such time, their intersection cells are recalculated by traversing the PCPI whilst simultaneously calculating the new intersection cell. The minterms themselves are classified as *enlarged*, if their intersection cells are likewise enlarged.

Records of expanded minterms require new fields describing the state of their associated intersection cell I_{min} and intersection cube I_{max}, so that it can be determined when cells are enlarged, potentially covering new dominating minterms. The easiest way to do this is to simply associate a cube record with each expanded minterm.

However, it is possible to do slightly better. A cube is uniquely determined by a contained minterm, and a vector indicating the unspecified ("don't-care") variables, and this is how they are represented in MDSA. The expanded minterm records need only the vectors giving the unspecified variables for the cube and the cell; the records already contain the minterm.

8.7 Results

The most important benefit from the elimination of dominating minterms is the reduction in the number of RMT's that need be expanded. An immediate affect of this is fewer expected iterations of the directed search algorithm during a minimization. Also, fewer RMT's are generated in the first place, which saves much time because of the relatively expensive calculation of minterm order accompanying the discovery of each RMT (Chapter 4). In the case of cyclic functions, the existence of cycles is usually proven with far fewer minterm expansions.

The results of experiments with some of the larger PLA benchmarks, both non-cyclic and cyclic, are given in Tables 8.1 and 8.2. The cyclic PLA's are indicated by an asterisk next to their name. A new policy is used for indicating PLA size; instead of the number of cubes in the benchmarks specifications themselves, the absolute minimum cover size is given (*min*). Where this number is not known, the best known cover size (found by MDSA) is indicated, with an asterisk preceding it.

In both tables, results are given for MDSA without any minterm domination (columns headed by "none"), with primary minterm domination (columns headed by "prim"), and with comprehensive (both primary and secondary) minterm domination (columns headed by "comp"). Table 8.1 shows the effects of the new techniques on internal data structures, including the maximum size that the PCPI reaches, and also

PLA				PCPI (maximum)			PCPI (cycle)			EMTS (cycle)		
name	n	m	min	none	prim	comp	none	prim	comp	none	prim	comp
add6	12	7	355	1320	1148	1148	0	0	0	0	0	0
dk48	15	17	22	6	6	6	0	0	0	0	0	0
in1	16	17	104	7	7	7	0	0	0	0	0	0
*mult4	8	8	121	438	354	354	438	318	313	463	315	302
*sev	8	10	*195	833	691	690	833	684	683	901	478	473
*apex4	9	19	427	1043	895	878	992	318	180	1038	308	172
*prom2	9	21	*287	2501	1921	1819	2501	1917	1813	2854	1623	1524
*tial	14	8	575	318	317	316	129	71	71	442	116	116
*in0	15	11	107	51	46	46	6	6	6	24	6	6
*in2	19	10	134	97	36	36	97	32	32	61824	36	36
*misex3	14	14	60	537	431	425	537	399	389	3492	615	561
*misex3	14	14	*380	4979	3068	3067	4979	2336	2234	10272	2664	2352
*misex3c	14	14	39	14697	10494	10488	6931	5099	697	11316	5560	837
*misex3c	14	14	60	537	434	425	537	400	389	3492	625	561

Table 8.1: Effect of Dominating Minterms Removal on Data Structures

the sizes of the PCPI and EMTS sets when a (pseudo) cycle is detected. Two entries are given for `misex3` and `misex3c`, since each contain two non-trivial cycles. In these cases, the number *min* is the number of cubes in the minimum known cover for the true cyclic cores.

There are significant drops in most of the data structure dimensions, when minterm domination is incorporated. The biggest improvement comes from the inclusion of primary minterm dominance; the improvement realized through adding secondary minterm dominance is not so dramatic. Note that the cycle PCPI and EMTS sizes are dimensions of the true "cyclic cores" of the PLA's, when comprehensive reduction is used, because by definition dominance relations among neither cubes nor minterms could exist.

A convincing metric for illustrating the power of minterm removal within directed search is the number of main iterations required. The results are particularly striking for the larger PLA's, especially `in2`. For the cyclic PLA's in Table 8.2, better results (minimal cube output) when removing dominating minterms are due to the existence of pseudo-cycles. The cycles solved by the improved versions of MDSA are smaller,

PLA				CUBES			ITERATIONS			TIME		
name	n	m	min	none	prim	comp	none	prim	comp	none	prim	comp
add6	12	7	355	355	355	355	390	393	393	1.90	1.90	2.20
dk48	15	17	22	22	22	22	25	25	25	2.00	1.67	1.68
in1	16	17	104	104	104	104	124	124	124	5.10	4.10	4.10
*mult4	8	8	121	123	123	122	515	393	381	0.86	0.60	0.67
*sev	8	10	*195	209	205	203	989	542	537	2.50	1.30	1.39
*apex4	9	19	427	438	428	428	1427	966	862	2.80	2.00	1.80
*prom2	9	21	*287	382	344	326	3029	1810	1711	12.00	6.30	6.80
*tial	14	8	575	577	576	576	1459	962	957	8.00	7.50	7.60
*in0	15	11	107	107	107	107	141	123	123	1.90	1.80	1.90
*in2	19	10	134	138	134	134	64093	214	195	3100.00	21.90	20.10
*misex3	14	14	*653	702	685	675	14251	3797	3631	151.00	39.50	42.20
*misex3c	14	14	190	203	201	194	17095	6845	6739	2643.00	1423.00	1670.00

Table 8.2: Empirical Effect of Removing Minterms in MDSA

and the probability of choosing a cube that belongs to an actual minimum cover is therefore higher. Unsurprisingly, the times required for minimization are also much improved, especially for the last three PLA's.

A final table is included for comparing the number of iterations now required, against those required before using minterm dominance, for all (apparently) cyclic benchmark PLA's introduced in Chapter 4. The same set of cyclic PLA's from Table 4.2 (page 39) is repeated in Table 8.3. These include one, `clip`, that is actually only pseudo-cyclic. The table has two main vertical sections of results: one for experiments without use of minterm domination, and one for experiments with comprehensive minterm domination. The minimizations are performed without the use of essential cube recognition, and the figures for the number of iterations performed by MDSA do not include the number of times heuristic cube selections are made. This is so that the number of iterations required may be compared directly against the theoretical lower bounds.

The columns labeled "pseudo" give the number of essential and pseudo-essential cubes in the minimum covers, detectable by the respective versions of MDSA. The "cycle" columns give the number of minterms involved in the PLAs' cycles (which

PLA				WITHOUT MINTERM DOMINATION				WITH COMPLETE DOMINATION			
name	n	m	min	pseudo	cycle	total	MDSA	pseudo	cycle	total	MDSA
clip	9	5	117	86	148	234	322	117	0	117	219
*dist	8	5	120	87	96	183	235	100	41	141	204
*mult4	8	8	121	27	463	490	511	34	302	336	382
*in0	15	11	107	104	6	110	179	104	6	110	160
*sym9	9	1	84	0	420	420	420	0	420	420	420
*apex4	9	19	427	204	1038	1242	1460	355	172	527	909
*sev	8	10	*195	26	901	927	953	32	473	505	551
*tial	14	8	575	499	718	1217	1656	520	212	732	1163
*sym10	10	1	210	0	792	792	792	0	792	792	792
*misex3	14	14	*653	199	13764	13963	14125	211	2913	3124	3549
*misex3c	14	14	190	66	14808	14874	17122	91	1398	1489	6679
*in2	19	10	134	105	61824	61929	*64516	117	42	159	227

Table 8.3: Iterations and Bounds for (Pseudo) Cyclic PLA's

are guaranteed to be *true* only in the rightmost section). The "total" columns give the sums of the previous two columns, and therefore provide lower bounds on the number of MDSA iterations required to output pseudo-essential cubes, and isolate cycles. The "MDSA" columns indicate the number of iterations actually required during minimization to perform these tasks (the entry 64516 for `in2` includes the use of essential cube recognition; no data for `in2` without this optimization is available).

The lower bounds in the "totals" columns are much improved for most of the benchmarks. The corresponding drop in the number of MDSA iterations actually required is most dramatic for `misex3`, `in2`, and `misex3c`. However, the latter PLA still proves to be difficult for MDSA; the number of required iterations is over four times the lower bound. At least, `misex3c` is now MDSA solvable (*i.e.* its truly cyclic cores are isolated) in less than one half hour, without imposing a limit on the size of PCPI.

8.8 Summary

Directed search minimization without minterm dominance is inefficient for many input problems, because such a large proportion of minterms must be processed as either RMT's or EMT's. Moreover, the number of cubes output for the cover can be too large, when unremoved redundant minterms cause reported cycles to be larger than they really are. This chapter shows how minterm dominance is incorporated into the directed search algorithm. The techniques fit in naturally, and exhibit a duality with the techniques for the identification of cube dominance relations. In the MDSA minimizer, the concept of prime implicant set intersection cells embodies that of minterm dominance, which nicely complements the use of prime implicant cells to embody cube dominance.

It is also possible to encompass the idea of pseudo-essential cubes with minterm domination; pseudo-essential cubes correspond to an intersection calculated for a prime implicant set of cardinality one. In these cases, not only are the minterms in the intersection cell removed, but also the minterm itself. This idea is elaborated on in Chapter 9.

Chapter 9

Cycles

9.1 Introduction

Empirical data thus far shows that MDSA is very effective in minimizing non-cyclic PLA's, or PLA's with very small cycles, when the PLA size parameters are within the specified limits. The solutions found are guaranteed to be minimum, and they are found quickly. For these types of PLA's, MDSA compares very well indeed with competing minimizers. However, many PLA's contain complex cycles, and it is necessary to be able to effectively minimize them also. Cycles detrimentally affect MDSA (and most other minimizers) in two ways. The first is the problem of *recognition*: isolating the cycles that exist in a given problem instance. The second problem is that of actual *solution*, and this is the subject of the present chapter.

The problem of finding minimum solutions for cycles is equivalent to the graph-theoretic *minimum cover* problem, which is known to be NP-complete [GJ79]. In fact, the cycle problem is really a separate issue from the other techniques and algorithms presented in this work, which are tailored to cycle *avoidance* and/or *isolation*.

However, the data structures used in these other endeavors are also quite useful for solving cycles. This chapter presents both heuristic and exact solution techniques, using the cube and minterm trees and heaps introduced in Chapter 6. This permits linear rather than quadratic "storage" of the "prime implicant table", implicitly stored in the PCPI and EMTS sets. With p cubes in the PCPI and m minterms in the EMTS, an explicit prime implicant table requires $O(mp)$ storage, but with trees and traversals, only $O(m + p)$ is required.

This chapter covers the following topics. First, the cycle phenomenon is more carefully defined, both in general and in the specific context of an MDSA minimization. Several "greedy" heuristic methods for solving cycles are presented and compared, plus a technique for ensuring irredundancy in the resultant near-minimal solutions. An improved backtracking model for finding exact solutions is introduced, and it is shown how its use in MDSA requires only linear storage. The model requires a new method for finding cube cells, techniques for selecting cubes, plus lower bound functions and partitioning algorithms, all of which are explained. Empirical performance on the benchmark set is reported, for both heuristic and exact algorithms. Finally, the contributions are reviewed, and outstanding issues discussed.

9.2 Cycles in MDSA

In general, a prime implicant table is *cyclic* when the table has two or more rows (corresponding to prime implicants) and two or more columns (corresponding to minterms), such that there does not exist a column with only one mark (*i.e.* there are no pseudo-essential minterms) [Die88]. However, because MDSA is so effective in eliminating all dominated cubes and dominating minterms (Chapter 8), "true" cycles are of more interest. This means that the table is not only cyclic, but also *fully re-*

f	bc 00	01	11	10
a 00	0	-	1	-
01	-	1	-	1

(a) Karnaugh Map

P	minterms 011	101	110
cubes XX1	1	1	
X1X	1		1
1XX		1	1

(b) Prime Implicant Table

Figure 9.1: An Extremely Simple Cycle

duced, which means that there does not exist either a dominated row, or a dominated column [Giv70, RSV87][1]. Figure 9.1 shows the map and prime implicant table of the simplest true cycle possible. Unfortunately, the cycles encountered in practice are very often larger.

Table 9.1 summarizes the true cycle dimensions of the cyclic benchmark PLA's. The definition of true cycles does not include the phenomenon of *partition*, which means that the cyclic part of a problem instance can be broken up into two or more subproblems. Usually, if an input problem contains several distinct partitions, MDSA discovers them separately. However, this is not guaranteed. Table 9.1 gives the total number of unseparable partitions, and the sizes of the two largest. Partitions are discussed further in a subsequent section.

Some exact minimizers require explicit prime implicant tables for performing table reductions and extracting pseudo-essential cubes, as well as solving cycles (*e.g.* [RSV86]). In MDSA, potential cycles are represented by the potentially cyclic prime

[1] A fully reduced prime implicant table must have a minimum of three rows and three columns.

PLA				PARTITIONS	LARGEST			NEXT LARGEST		
name	n	m	min	number	minterms	cubes	minimum	minterms	cubes	minimum
dist	8	5	120	1	41	45	20			
f51m	8	8	76	2	9	8	4	6	6	3
mult4	8	8	121	1	302	313	87			
sqr6	6	12	47	1	135	150	41			
5xp1	7	10	63	1	129	119	37			
in0	15	11	107	1	6	6	3			
in2	19	10	134	2	36	32	14	6	6	3
sym9	9	1	84	1	420	1680	84			
sym10	10	1	210	1	792	4200	210			
apex4	9	19	427	1	172	180	72			
sev	8	10	*195	1	473	683	*163			
tial	14	8	575	9	24	14	6	20	15	6
misex3	14	14	*653	3	2352	2234	*380	549	377	56
misex3c	14	14	190	3	837	697	39	549	377	56

Table 9.1: Cycle Characteristics of Benchmarks

implicant set (PCPI), and the set of expanded minterms (EMTS). True cycles are identified when all minterms covered by members of PCPI have been expanded. Thus, cycles are recognized when the set of recursive minterms (RMTS) is empty, but PCPI and EMTS are not. (The PCPI set might be more accurately called the cyclic prime implicant set, at this point.) Chapter 8 explains how the cycle existence requirement is lax in the original DSA: all that is required is that there do not exist any dominated cubes in the PCPI. The irreducibility requirement is precise in MDSA, where dominating minterms are also excluded. The important point is that in MDSA, when the solving of a cycle commences, no irrelevant cubes or minterms are present. (The exception is when a user-imposed limit on the size of PCPI or EMTS is reached. This does not apply to *any* of the benchmark PLA's, though the MDSA minimization of `misex3c` does result in a maximum PCPI size of over 10000 cubes.)

Whether or not redundancy exists in some "cycle" recognized by the minimizer, has no bearing on how the "cycles" are treated. Whatever methods DSA or MDSA or any other minimizer uses for simplifying or reducing the cycle, the same reduction methods are used after a heuristically chosen simplification, whereby an assumption is

made that either a cube belongs to the solution, or it does not. In DSA and MDSA, the effect of this selection or rejection on the PCPI and EMTS is calculated, and the appropriate simplifications made (elimination of dominated cubes in DSA, and elimination of both dominated cubes and dominating minterms in MDSA). Such a move is said to *break* the current cycle.

As stated before, an important strength of MDSA is that its implicit representation of a cyclic prime implicant table, in the form of PCPI and EMTS, requires only linear storage. The information needed to solve the minimum cover problem is *encoded* in the bit string representations of cubes and minterms, and the trees and heaps of Chapter 6 provide an accessing mechanism. Given the PCPI and EMTS, it always possible to determine which cubes cover which minterms, and (conversely) which minterms are covered by which cubes. In this regard, two-level minimization differs from the more general minimum cover problem [GJ79].

9.3 Dumping Cycles

One approach for solving cycles is to simply leave the problem to another program MDSA is good for finding all the pseudo-essential cubes of a cover, and isolating the "true" cycles in one or more pairs of PCPI and EMTS sets. If these data structures from a given problem instance can be written out in a suitable format, then a program tailored to the specific problem of cycle resolution might be employed. As a simple example, a PLA can be first "minimized" by MDSA, yielding a list of covering cubes which the program finds to be pseudo-essential, plus the cycles of the PLA in some form. Then, *espresso* could be used to solve the cycles. The main strength of MDSA, that of isolating fully reduced cycles, is used to full advantage.

The strategy is as depicted in Figure 9.2. The cycle isolater is **MIN_0**, and the

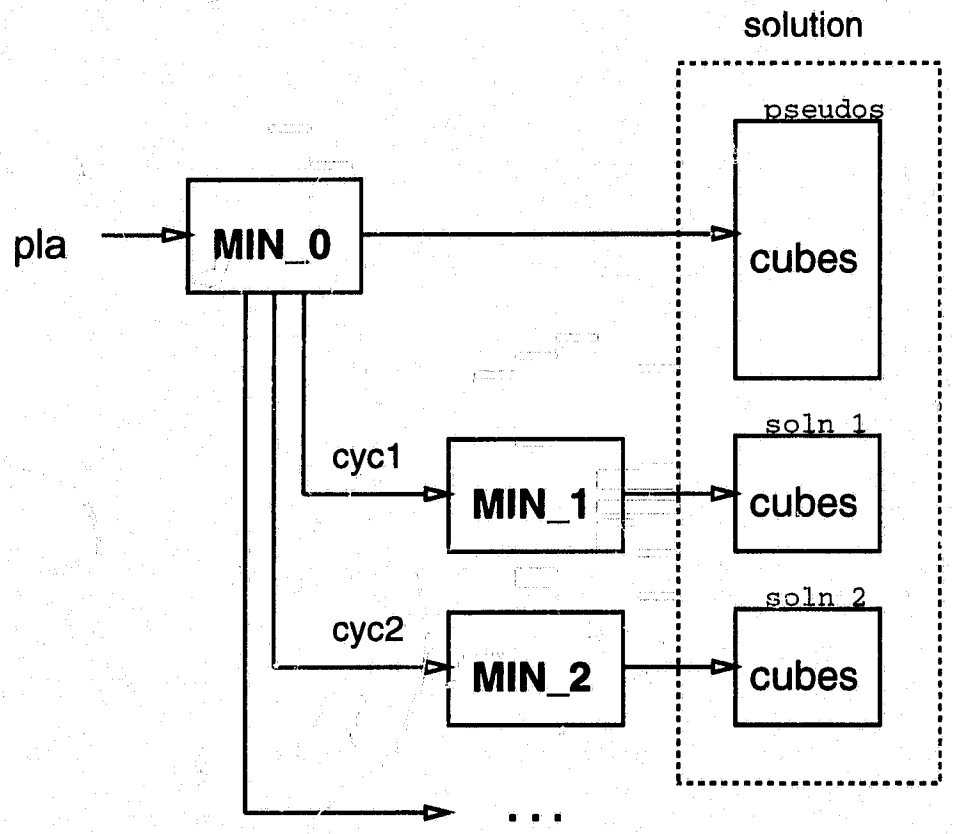


Figure 9.2: Parallel Minimizers

other minimizers solve the cycles output by this program (whether MDSA or not). Note that the other minimizers need not be PLA minimizers. As previously stated, minimum cover is a well-known graph-theoretic problem, and research developed in that area can be used for finding cycle solutions. All that is required is the appropriate translation from *cubes* and *minterms* in the domain of discrete functions, to the more general concepts of *sets* and *elements*. Moreover, the other minimizers can work independently, and therefore potentially in parallel, as **MIN_0** discovers and outputs new cycles.

Of course, this approach is worthwhile only if the total time spent by MDSA in its own non-cyclic minimization, *plus* the time spent in outputting the cycle information, *plus* the time spent by the cycle-breaking minimizer (say, *espresso*) in breaking these output cycles, is *less* than the time it takes either MDSA or *espresso* to perform the whole minimization. Alternatively, the strategy may be justified if the total number of cubes found is lower.

The most straightforward way to implement the two program strategy is to convert the essential and pseudo-essential cubes found by MDSA to "don't-care" cubes, and to append this set to the original *espresso*-style specification of the problem instance. In the *espresso* format, vertices covered by these new cubes are considered to be "don't-care", even if previously specified as true minterms. Thus, the new specification is precisely the cyclic part of the PLA. Unfortunately, this appending of "don't-care" cubes does little to help *espresso*. The increase in the size of the DC-set does not affect the number of truly essential cubes, which *espresso* depends on for initial simplification [BHMSV85]. However, the strategy *does* work with MDSA, and is used in some of the experiments on cycles in this chapter, where the cycles would otherwise take too long to isolate. (This phenomenon is observed in only `misex3` and `misex3c`.) Thus, MDSA itself is used as **MIN_1**, as well as **MIN_0**. A better candidate has not been found.

9.4 Heuristic Resolution of Cycles

This section describes a class of heuristic algorithms for quickly finding sub-optimal covers for cycles, and the data structures required to support the algorithms in MDSA. It is important to distinguish between heuristic choices made for selecting minterms for expansion, and finding a heuristic solution for a cycle. In the former case, the

heuristic is used to (hopefully) improve the speed of finding a solution, but does not affect the *correctness* of the solution. However, in this section the word "heuristic" refers to methods that not only seek to increase speed, but also can no longer guarantee optimality.

In the benchmark results presented heretofore, a simple heuristic rule is used to break cycles. When a cycle is encountered, the entire PCPI is traversed, and the cube that covers the most minterms (all of which, by definition, must be in EMTS) is selected for the cover. This "heavy cube" heuristic works reasonably well in practice, as evidenced by the previous results. It also has the advantage that it causes rapid simplification of the cyclic core, since a maximum of minterms are immediately removed from EMTS upon selection of the heuristically chosen cube.

Choosing the heaviest cube and then proceeding with the consequences is an example of a *greedy algorithm.* No backtracking is performed. An assumption is made, and the results are accepted. The data structures and cell-based techniques used in MDSA permit several possible approaches for selecting the heuristic cube:

1. Choose a cube at random.
2. Choose the "heaviest" cube, *i.e.* the one covering the largest number of expanded minterms.
3. Choose the minterm covered by the fewest number of cubes. Among those cubes, choose the heaviest.
4. Choose the cube intersecting with the largest number of other cubes.
5. Choose the cube affecting the largest number of other cubes' cells.
6. Choose the cube whose selection would result in the largest number of dominated cubes.

No matter how a heuristic cube is chosen, the same directed search process is used as with other selected cubes: find all intersected, affected, and dominated cubes, bring PCPI up-to-date with respect to dominance relations, remove dominated cubes, and (in MDSA) remove dominating minterms. Then, there might exist a pseudo-essential cube, and normal directed search iterations start again. However, it is more common for the complex cycles to be *nested.* In this case, many heuristic choices are required before the cycle is completely broken.

When selecting an heuristic cube, it is undesirable to scan the entire PCPI to find the most suitable one. The nesting nature of the more complex and larger cycles, which require repeated heuristic selection/rejection, makes this point particularly significant. For some heuristic criteria, it is easy to maintain the heuristic as an integer associated with each cube. This is true of the "heavy" cube criterion, which depends on the *count* (or *weight*) fields of cubes' records, already required in other aspects of MDSA. The heap implementation of cube trees (Chapter 6) is thus a natural choice for the PCPI. Selection of heuristic cubes is simply a matter of removing the top of the heap.

There is no appreciable overhead incurred by heap-inserting new primes into PCPI, as compared to simple tree-inserting, but the weights of cubes do change while an MDSA minimization progresses. If an attempt is made to maintain the PCPI as a heap throughout a minimization, weight changes require heap adjustments to occur frequently. This results in extra overhead, even if no cycles are ever found. Moreover, fast cell calculation algorithms preclude the ability to determine the weights of cubes. The approach used in MDSA is to initially use the *cell size* as a heap ordering criterion. Then, when a cycle is detected, the weights of all cubes in PCPI are found, and any required heap adjustments performed at that time, before the first heuristic cube selection is made.

PLA				RANDOM CUBE		HEAVY CUBE		LIGHT MINTERM		RUDELL WEIGHT	
name	n	m	min	cubes	red	cubes	red	cubes	red	cubes	red
dist	8	5	120	120	0	120	0	120	0	120	0
f51m	8	8	76	76	0	76	0	76	0	76	0
mult4	8	8	121	130	1	123	0	125	0	123	0
sqr6	6	12	47	50	1	48	1	49	0	48	0
5xp1	7	10	63	69	2	63	0	67	0	63	0
in0	15	11	107	107	0	107	0	107	0	107	0
in2	19	10	134	135	0	134	0	134	0	134	0
sym9	9	1	84	93	3	88	0	84	0	90	0
sym10	10	1	210	216	3	211	0	210	0	211	0
ex1010	10	10	*267	390	29	274	0	330	4	278	2
apex4	9	19	427	430	1	427	0	428	0	427	0
sev	8	10	*195	210	4	201	0	209	1	202	0
tial	14	8	575	593	8	576	0	576	0	576	0
misex3	14	14	*653	737	27	675	3	676	1	668	1
misex3c	14	14	190	210	15	194	0	195	0	193	0

Table 9.2: Comparison of Cycle Breaking Heuristics

Table 9.2 summarizes the results of experiments comparing some of the greedy heuristic selection options. The random selection option is included to show the importance of using heuristics, especially for PLA's with larger cycles. The remaining three heuristics all make use of maintaining the PCPI heap by "weight", with heaviest cube at the top of PCPI. The table includes the "Rudell weight" heuristic, where the weight of a cube is defined to be the sum of the reciprocals of its contained minterms' cover counts [RSV87]. This weighting strategy is discussed further in the context of exact cycle resolution (Section 9.8). The results for the intersected/affected/dominated cubes heuristics are not included. In general, they do not compare favourably with the others.

The heuristic method that chooses the lightest minterm, then the heaviest cube covering that minterm, requires more time overhead than the others. The set of minterms EMTS is already stored as a minterm heap, with the lightest minterm M with weight k on top, for detection of pseudo-essential minterms (Chapter 6). No extra overhead is incurred by inspecting this minterm M. But, in general several of the k covering cubes in the PCPI heap must be inspected before the heaviest is found.

A branch is bounded in this search when the first covering cube down that branch is found, since all cubes below must be lighter. Thus, not all k cubes must be found. The heuristic does not appear to be a good one in general, but it is interesting that it works so well with the `sym9` and `sym10` PLA's. (Multiple experiments with random juggling of the input variables and their phases confirm this effect.)

Table 9.2 also includes columns labeled "red", giving the number of redundant cubes found in the experiments. This is described in more detail in the following section.

9.5 Ensuring Irredundancy

When heuristic choices are used for solving a cycle, there is a chance that the resulting cycle cover is *redundant*. A redundant cover is one that contains at least one redundant cube, which in turn is a cube that can be removed from the cover without affecting the cover's correctness. A redundant cover is not even *minimal*, let alone minimum. A cube can be removed without affecting correctness *iff* every minterm contained in that cube is covered by at least one of the remaining cover cubes. In the context of cycles within MDSA, contained minterms are precisely the ones in EMTS at the time of cyclic core recognition, and do not include any minterms that became "don't-care" previously during the same minimization, before the cycle was reduced and isolated.

Prime irredundant covers are not necessarily minimum, but they are certainly closer to that state than near-minimal covers. If the cubes that must be selected to cover minterms in a cycle are all retained in a separate set, and the set of minterms involved in the cycle is retained, the cube tree data structure of Chapter 6 can be used to see if any cubes are redundant. As cubes are selected during heuristic minimization of cycles, they are stored in a separate tree CPI. After an heuristic cover is obtained,

```
procedure REDUNDANT;
begin

    clear count fields of EMTS;
    for each cube C in CPI,
        for each minterm M in EMTS, contained by C,
            increment count field of M;

    for each cube C in CPI,
        Redundant = TRUE;
        for each minterm M in EMTS, contained by C,
            if count of M = 1,
                Redundant = FALSE;
                break;

        if Redundant = TRUE,
            remove C from CPI;
            for each minterm M in EMTS, contained by C,
                decrement count field of M;
end
```

Figure 9.3: Algorithm for Finding Redundant Cubes

and the original set of minterms contained in the cycle recovered, the algorithm of Figure 9.3 is used to find redundancies. The complexity of the algorithm depends on the complexity of containment traversal of minterm trees. If the average time required for this traversal is d, and there are c cubes in CPI, then the overall complexity is $O(dc)$.

The results of applying the redundancy algorithm to heuristic cycle solutions are included in Table 9.2, in the "red" columns. Thus, the number of cubes in the covers resulting from applying the redundancy algorithm is the difference between

the "cubes" and "red" columns. Note that redundancies do not often arise, except when particularly bad cube selection heuristics are used (the "RANDOM CUBE" results), or when the cycles involved are huge (for example PLA `misex3`). Despite the infrequency with which redundant cubes are found, the algorithm for removing them is worthwhile to execute since it runs so quickly in comparison to the other cycle processing algorithms. In fact, the running time of the algorithm averages about 1% of the time required to solve a cycle heuristically, and never has been observed to be greater than 3%.

9.6 An Improved Exact Resolution Model

This section discusses the problem of exact cycle resolution, and a model for incorporating it in the MDSA minimizer. Success in this endeavor cannot always be expected, because of the complexity of the problem [GJ79]. However, it is worthwhile to explore conditions in which exact (or *minimum*) solutions are computable in "reasonable" time, and how often these conditions arise. It turns out that many of the cyclic benchmarks are solvable in a matter of seconds or minutes. Also, for many of these the exact solution is superior to solutions found by heuristic methods. The minimizer incorporating the new model and associated algorithms is dubbed *MDSA-exact*.

A general technique used to search the solution space of problems of exponential complexity is *branch-and-bound*, whereby the solution space is spanned by a (usually binary) tree. A node of the tree corresponds to a state in the process of computing a solution. Branches emanating from nodes correspond to different moves that would affect the solution from a given state. Leaves of the search tree correspond to solutions. A branch is *bounded* when a calculation shows that it is not possible to find

a better solution down that branch than one already obtained. Because the search space expands exponentially, it is generally worthwhile to spend some extra effort in these bounding calculations.

Both *espresso-exact* and *McBOOLE* use the branch-and-bound model for solving cycles. In both minimizers, the spanning binary tree uses the *selection* and *rejection* of prime implicants for its branching criterion [RSV86, DAR86]. *Espresso-exact* also incorporates an algorithm for computing a lower bound on the number of cubes required in a solution, used in bounding the search. The pseudo-code for this algorithm is provided in [RSV87]. It serves as a starting point for the work in this section, and is thus repeated in a slightly modified form in Figure 9.4. (Partitioning is excluded for now; it is discussed separately in Section 9.10.) The algorithm has the following weaknesses:

1. **Use of an explicit prime implicant table.** After prime implicants are generated, a complete matrix is constructed that contains all the minterms and all the implicants, *i.e.* a prime implicant table. This standard two-phase approach fails with even some non-cyclic PLA's, when the number of minterms and/or prime implicants is too large [RSV87]. (McBOOLE generates all primes; *espresso-exact* removes essential primes and their minterms before table construction.)

2. **Restoration of previous states.** The procedure call stack is used to recall previous states, as backtracking is performed. In the given algorithm, this is done by passing the entire PI table as a parameter, in addition to two sets of prime cubes. This approach becomes expensive when the nesting level of cycles grows.

3. **Correctness of prime implicant table reduction.** The exit condition of the `repeat` loop depends on the existence of induced pseudo-essential cubes.

```
function MIN_COVER( T, soln, best, low_bound, level ) : cubeset;
    matrix T;
    cubeset soln, best;
    integer low_bound, level;

    matrix T1;
    cubeset P, left, right;
    integer bound;
begin
    /* fully reduce T */
    repeat
        T = Remove_dominating_minterms( T );
        T = Remove_dominated_cubes( T );
        P = Find_pseudoessentials( T );
        soln = soln union P;
        T = Remove_minterms( T,P );
    until P==emptyset;

    bound = compute_low_bound( T );
    if (level==0) low_bound = bound;

    if (card(best)<=card(soln)+bound) return best;
    if (T is empty) return soln;

    /* recur */
    B = Choose_branching_cube( T );

    T1 = Remove_minterms( T,B );
    left = MIN_COVER( T1, soln union B, best, low_bound, level+1);
    if (card(left)<card(best)) best = left;
    if (card(best)==low_bound) return best;

    T1 = Remove_cube( T,B );
    right = MIN_COVER( T1, soln, best, low_bound, level+1);

    if (card(right)<card(best)) best = right;
    return best;
end
```

Figure 9.4: Suboptimal Algorithm for Solving Cycles

If no pseudos are found, the table is assumed to be fully reduced. This is not necessarily true, and negatively impacts the efficiency of the next recursive call.

4. **Efficiency of prime implicant table reduction.** The same table reduction strategy (the `repeat` loop of Figure 9.4) is used after both selection of cubes, and rejection of cubes. The order in which cubes and minterms and pseudo-essentials are removed is fixed, and the three processes do not inter-communicate.

5. **Use of the lower bound.** The lower bound is not used as effectively as it could. (This is discussed at length in a subsequent section.) Also, its computation requires memory quadratic in the number of minterms. adding to the memory burden from point (4).

The branch-and-bound model is also used for exact cycle resolution in MDSA-exact. The tree data structures for PCPI and EMTS lend themselves well to the model. In fact, with these data structures it is not necessary to create an explicit prime implicant table. Figure 9.5 is a simplified representation of the call graph implemented by the cycle solving procedures of MDSA-exact. It is illustrated as a portion of a solution space spanning tree, to show that the branches for cube selection and rejection differ. For clarity, the calculation and use of lower and upper bounds, and partitioning, are omitted from the figure.

The model of Figure 9.5 is essentially a composite of the MDSA implicit prime implicant table reduction methods, used after selection of pseudo-essential cubes, and when dominated cubes are rejected from PCPI, as discussed in Chapter 8. The main change is the provision of backtracking capabilities. The model consists of three recursive procedures (**S/R**, **DC** and **DM**), that communicate via minterm sets and cube sets. The sets are passed as parameters, but are never changed by the callee. The strategy employs the following improvements over the algorithm of Figure 9.3:

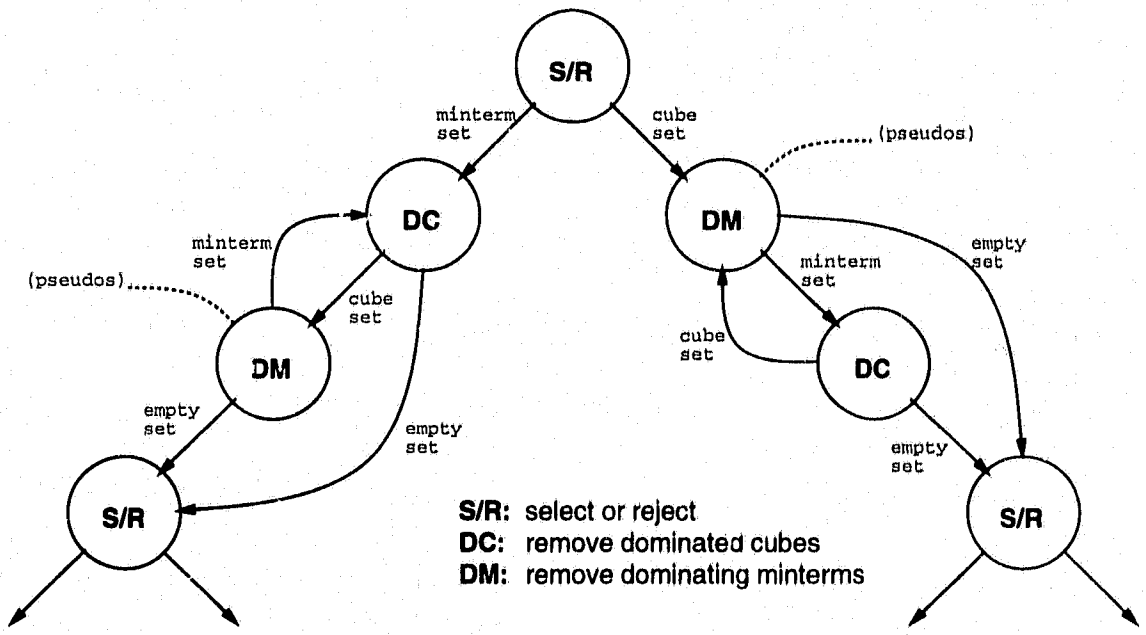


Figure 9.5: Improved Branch-and-Bound Model

1. An explicit prime implicant table is not required. Instead it is implicitly represented by the PCPI and EMTS data structures. Moreover, this implicit table is guaranteed to be fully reduced before branching commences.

2. The branch-and-bound algorithm is implemented as a set of mutually recursive procedures, each performing their own backtracking duties on the relevant minterm sets and cube sets. Multiple copies of the PI table in various states are not required in this scenario.

3. The loop that performs reductions on the PI table is independent of whether or not pseudo-essential cubes are found. The loop containing **DC** and **DM** is simply exited as soon as there is *no change* in either the "rows" or the "columns" of the table.

4. The order in which **DC** (remove dominated cubes) and **DM** (remove dominating minterms) are applied depends on whether a cube is selected or rejected. Moreover, the search for affected and dominated cubes and minterms in each of these procedures is *localized* by using the input `set` parameter.

5. The lower bounds are used to full advantage, and bounding of branches is permitted to occur at points other than the main branch point **S/R**.

Points (1) through (4) are discussed in greater detail below. Point (5) warrants its own section, and is therefore considered separately.

The three mutually recursive procedures all work on the global data structures PCPI and EMTS. As depth in the search tree increases, elements of these structures are extracted and become elements of removed subsets (`minterm set` or `cube set`). These are stored as local variables of the procedures, and are also passed as parameters to the next procedure in the call sequence. The important point is that each cube or

minterm record exists in only one place: either the PCPI or EMTS, or an extracted set. Thus the amount of memory being used remains nearly constant. Slightly extra memory is incurred by branching depth and is from some minor scalar status variables, and pointers to the minterm set and cube set parameters.

S/R: Selection and Rejection

The main recursive procedure is **S/R**. It handles the heuristic selection and removal of a branching cube C from PCPI, then recursively attempts the selection and rejection of C from the (implicit) prime implicant table. Selection requires the removal of minterms covered by C from EMTS (there must be at least two, by definition of cycle). These minterms are passed as a set parameter to **DC**, on the left branch. After **DC** returns, the `minterm set` is restored to EMTS. Cube C is then passed as a parameter to **DM** on the right branch. (The cardinality of this `cube set` is thus one.) Finally, cube C is restored to PCPI, and control passes back up the tree to the caller.

DC: Removal and Replacement of Dominated Cubes

Procedure **DC** is used to adjust cells and find dominated cubes, when one or more minterms are removed by the other procedures. Its sole parameter is a `minterm set`. (As shown in Chapter 8, dominated cubes cannot possibly arise in PCPI unless one or more minterms have been removed.) The PCPI tree is traversed with each member M of `minterm set` in turn. Containing cubes have their cells re-calculated to see if they are affected. Another traversal of PCPI is performed for each affected cube, to see if it is dominated. If one or more dominated cubes are found, then **DM** is called, otherwise a cycle exists again, and **S/R** is called. After the recursion returns, the "prime implicant table" is restored by returning the dominated cubes of `cube set` to PCPI.

DM: Removal and Replacement of Dominating Minterms

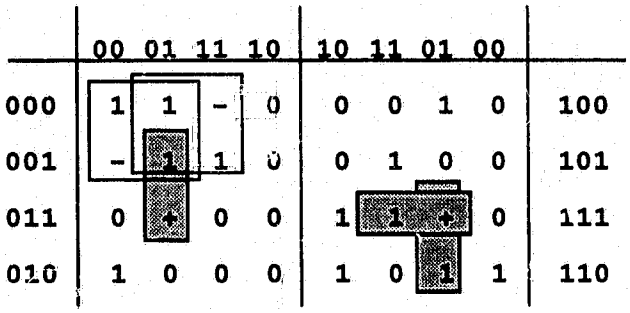
Again, from Chapter 8 it is known that dominating minterms cannot possibly arise in EMTS, unless one or more cubes are removed from PCPI. Thus, procedure **DM** is passed a non-empty cube `set`, and finds a set of resulting dominating minterms. For each cube, EMTS is traversed to find affected minterms, and the cover counts and intersection cubes of all such minterms M are recalculated. If the intersection cube of a minterm expands, then the new part may cover some new dominating minterms in EMTS, which are duly extracted. After the call to **DC** (or **S/R** when no new dominating minterms are found), the minterms are restored to EMTS.

Induced Pseudo-essential Primes

The detection of new pseudo-essential cubes happens in **DM**. If the new cover count of an affected minterm M is one, then M's intersection cube is equivalent to the one remaining cube C in PCPI that covers M. Thus, M is now an induced pseudo-essential minterm, and is added to the dominating set along with the other minterms in its intersection cube. A side effect is the removal of the induced pseudo-essential covering cube C from PCPI, to the solution set.

The incorporation of pseudo-essentials with dominating minterms corrects an error in the algorithm of Figure 9.3. A pseudo-essential cube need not immediately arise when dominating minterms are found. For example, consider the five variable cyclic function in Figure 9.6(a). Assume the cube `X1101` (reading row, then column) is selected as the branching cube. Its two contained minterms, marked as '+' in the figure, become "don't-care". Three dominated cubes arise (shown shaded), and are removed from consideration by both algorithms. However, no induced pseudo-essential cubes yet exist, and the algorithm in Figure 9.3 exits its reduction loop.

In the example, there is one new dominating minterm `00001`, because of the af-



(a) Karnaugh map

```
S/R : X1101
 DC : 0X101 111X1 11X01
 DM : 00001
 DC : X0001
 DM : 10001 11001
      (pseudo=1X001)
 DC : 1100X
S/R :  ...
```

(b) Behaviour of algorithm

Figure 9.6: A Five Variable Cyclic Function

fected minterm `00101`. The latter's covering count shrinks from three to two cubes (shown by the unshaded square boxes), with a corresponding expansion in its intersection cube. Removal of newly-discovered dominating minterm `00001` results in affected cube `00XX1` shrinking to `001X1`, and one more dominated cube `X0001`. Finally, a new pseudo-essential cube `1X001` is discovered, because the covering count of minterm `10001` is reduced from two to one. One more dominated cube is found and removed before branching starts again. The entire reduction process is summarized in Figure 9.6(b).

Contraction and Expansion of Primes' Cells

In procedure **DC**, returning extracted dominated cubes to PCPI for backtracking also involves restoring their original cells. In fact, all cubes affected by the removal of the input `minterm set` must have their original cells restored. These affected cells remain in the PCPI during execution of **DC**, so they must be located down predictable branches of the PCPI tree, so that they can be found when cell expansion occurs at the backtracking phase. Thus, the cubes of PCPI must be stored not by their cell strings, but by their cube strings.

For example, consider a member of PCPI whose cell is `01XX` upon entry to **DC**. There, it is found to be affected, and becomes `01X0`, because of a minterm `0101` passed in as part of the `minterm set` parameter. The new cell is not dominated however, so it must remain in PCPI. When control returns to **DC** after the recursive call, a traversal of PCPI with minterm `0101` is performed to find all affected cells that must be re-expanded. But cell `01X0` would not be found (unless it was the root), because it does not contain `0101`. By storing the cell by its containing cube, this problem is avoided. Note that storing the cell upon discovery that it is affected is in general unworkable, because a cell may shrink many times before it becomes dominated or pseudo-essential.

9.7 Cell Calculation with Minterm Trees

The ability to backtrack while solving cycles exactly precludes the use of the truth table for determining the cells of cubes. The number of minterms contained in a cell containing k vertices may be anywhere between 2 and k, and they may form any of an exponential number of truth table "patterns". Recovering a change to a cell would require storing a whole section of the truth table corresponding to the change. Since the cell of any given cube may change a near-arbitrary number of times down a given search tree branch, this approach is unworkable.

During an MDSA minimization, the minterms indicated by set bits in the truth table representation are the union of the EMTS, NMTS, and RMTS sets. When a cycle is detected, the RMTS set is empty, and members of NMTS have no relevance to the current cycle. Thus, all of the information required to calculate the cell of a PCPI cube C lies in the EMTS set. That is, all the minterms contained by the cell of C, and no more, are in EMTS. All that is required is a new algorithm for calculating

cells, directly from EMTS.

The basic machinery required for the new cell calculation algorithm is in the tree traversal techniques of Chapter 6. Cells must be recalculated in procedure **DC** of Figure 9.5, due to the extraction from EMTS of a `minterm set`, passed to **DC** as a parameter. All that is required to find the new cell of a candidate cube C (*i.e.* one whose present cell contains at least one or more members of `minterm set`) is to perform a containment traversal of EMTS with the current cell of C. The new cell must be contained by the old, thus all relevant minterms are found in the traversal.

At the end of **DC**, all cells that were affected at the beginning, including those removed because they were found to be dominated, must have their original cells restored. Rather than perform another traversal of EMTS for each such affected cell, PCPI is traversed with each member of the extracted `minterm set`, and containing cubes are incrementally re-expanded. Only then is `minterm set` restored to EMTS.

The weight of a cube is calculated concurrently with its cell. The basic definition of cube weight is simply the number of minterms it contains, and this is obviously trivial to find. Other definitions of cube "weight" are also easy to calculate. In particular, the "Rudell weight" of a cube is the sum of the reciprocals of the minterms' cover counts. Thus, Rudell weight is accumulated in much the same way, as contained minterms of EMTS are encountered during the containment traversal of EMTS.

9.8 Cube Selection

The choice of branching cube has a profound effect on the time required to exactly solve a given problem instance. It is desirable for the removal of the branching cube from the PI table to affect the table as much as possible, so that many table

simplifications (and many pseudo-essential cubes) occur before the next selection. This keeps the nesting depth low. Any heuristic used for greedy selection can also be used here, for exact solution.

The effects of cube rejection are as important as cube selection. In the case of the former, only the cube is removed from the PI table, instead of the cube *and* all its contained minterms. Thus, the effects of rejection are a subset of the effects of selection. In light of this, it is desirable to concentrate on the effects of cube rejection, to minimize the tendency of the search tree to skew to the left.

Many experiments have been performed in this research, comparing various selection heuristics. These include the six given in Section 9.4, and several more based on cube weight definitions, in turn based on characteristics of the PI table. It turns out that the best one, by a large margin, is the selection heuristic defined in [RSV87], here called *Rudell weight*. This is defined to be the sum of the reciprocals of the contained minterms' weights. The strength of Rudell weight is that rejection of a cube with high Rudell weight is likely to lead to more affected minterms [RSV87].

Selection of cube by heavy cube weight is easy if the PCPI tree is maintained as a heap throughout an exact minimization. Selection is performed by simply removing the top of the heap, and performing the required heap adjustments. However, it is also desirable to incorporate information gleaned from low bound calculations, as described below.

9.9 Lower Bounds

An improved upper bound on the number of cubes in the actual minimum solution is provided every time a new best solution is found. However, the more interesting

cyclic benchmarks contain cycles where the minimum solution has dozens or hundreds of cubes. Thus, the upper bound alone is not sufficient to prune the search space. Low bound calculations are sometimes extremely useful for bounding branches early. Effective low bounds must be calculated comparatively rapidly, and provide the ability to bound branches much higher in the search tree than possible with upper bounds.

Like the cube selection heuristic, it is difficult to find a better low bound than that defined in [RSV87]. This is a greedy algorithm for finding a *maximal independent set* of subsets of minterms. The algorithm (called MIS) works by iteratively choosing a minterm M, then finding the set of all minterms covered by cubes that cover M. Iterations continue until all minterms have been chosen. Then, the number of iterations required is a lower bound on the number of cubes in the minimum solution, because each selected minterm M must be covered by at least one of the cubes associated with it.

The maximal independent set algorithm as described in [RSV87] uses an *independency graph* for heuristically choosing the key minterms. The idea is to choose minterms that are "independent" of the most other minterms. The algorithm is slightly modified for use in MDSA. A minterm M is said to *depend* on minterm N *iff* there exists one or more cubes covering both M and N. A *dependency graph* may now be defined, where nodes correspond to minterms, and edges correspond to dependency relations. Because dependency is a symmetric relation, the edges are undirected. A dependency relation between two minterms remains as long as both minterms remain in EMTS.

The dependency graph of a cycle is modified in two ways during exact minimization. The first way reflects the changes in the graph due to minterms being extracted and returned from EMTS, as the search tree is traversed. The second way is during an MIS calculation, as minterms are extracted along with all incident edges. These

changes must be recognized, because the heuristic used to choose key minterms is based on low degree.

The dependency graph is not stored explicitly by MDSA, since that would require quadratic storage. Instead, the required operations on it are performed using traversals of the EMTS and PCPI trees. Heuristic selection of minterms with low dependency degree is accomplished by maintaining the EMTS as a heap. (EMTS does not have to be maintained by cover count anymore, because the recognition of pseudo-essential cubes does not require it whilst in cycle mode.)

The algorithm for finding a lower bound is given in Figure 9.7. Note that two new scalar fields per minterm record are required to store dependency counts. Also note that every lowbound calculation involves the destruction and recreation of both the PCPI set and the EMTS set. Complexity is related to the total number of minterms and cubes: each is removed once. Also, dependency counts of minterms undergo adjustment. This is equivalent to the elimination of edges in the dependency graph. Thus, the number of objects affected during a MIS calculation is $O(n^2)$. Additional overhead is incurred by the time required to search for these objects.

The calculation of maximal independent set helps in the selection of a branching cube. In particular, if there is any hope of actually attaining the lower bound calculated in a given state, the cube chosen for selection must be a member of the set of cubes involved in the calculation [RSV87]. This is exploited by MDSA in the following way. Each cube record has an `MIScount` field, initially zero. Cubes are ordered in the PCPI heap by low MIScount, then Rudell weight. Cubes *not* extracted from PCPI during an MIS calculation have their `MIScount` incremented. Then, cubes most often involved in MIS calculations are most likely to be selected.

```
procedure MIS;
begin
    E = emptyset;
    P = emptyset;
    lowbound = 0;

    while (EMTS not empty)

        lowbound = lowbound + 1;
        extract M from top of EMTS heap;
        restore M's original dependency count;
        heap add M to set E;
        extract set SC of all cubes in PCPI covering M;

        for each C in SC,
            heap add C to set P;
            for each minterm N in EMTS covered by C,
                restore N's original cover count;
                heap add N to set E;

                for each cube D in PCPI covering N,
                    for each minterm P covered by D,
                        decrement P's dependency count;
    EMTS = E;
    PCPI = P;
    return lowbound;
end
```

Figure 9.7: Algorithm for Calculating Lower Bound

9.10 Partition

A true cycle is *partitionable*, or *separable*, if its prime implicant table can be divided into two non-empty subtables, such that the minterms and cubes of one do not share any containment relationships with the minterms and cubes of the other [RSV87]. If a cycle cannot be further partitioned, it is *non-separable*. Partitions can sometimes exist before MDSA exact solution of a cycle commences, but this is rare. However, partitions often do arise during the branch-and-bound search for a minimum solution. It is important to recognize these induced partitions, because the total complexity of exactly solving two partitions is always less than the complexity of the unpartitioned original.

The algorithm used in MDSA for extracting non-separable partitions is given in Figure 9.8. It works by iteratively extracting covered minterms, then covering cubes, then minterms again *etcetera* until no more of either can be extracted from the original PCPI and EMTS. Upon completion, a non-separable non-empty partition resides in PCPI and EMTS. If the original cycle is indeed partitionable, then the sets P and E are non-empty, and represent another cycle which could be further partitionable.

The inclusion of partitioning in MDSA is illustrated in Figure 9.9. The original procedure **S/R** in Figure 9.5 is replaced by the new procedures **PT** and **S/R**. All calls to procedure **S/R** in the original model become calls to **PT** in the present one. The algorithm provided in [RSV87] attempts a partition upon entry to the main recursive procedure, which leads to repeated attempts to partition non-separable parts. This problem is avoided in the given model for MDSA.

```
procedure PARTITION;
    minterm M; minterm set E, MM;
    cube C; cube set P, PM;

begin
    E = EMTS;  EMTS = emptyset;
    P = PCPI;  PCPI = emptyset;

    remove minterm M from top of EMTS heap;
    MM = { M };
    while (MM not empty)

        extract cube set PM from P covering minterms in MM;
        for each minterm M in MM,
            heap add M to EMTS;

        extract minterm set MM from E covered by cubes in PM;
        for each cube C in PM,
            heap add C to PCPI;

    return (E,P);
end
```

Figure 9.8: Algorithm for Extracting Partitions

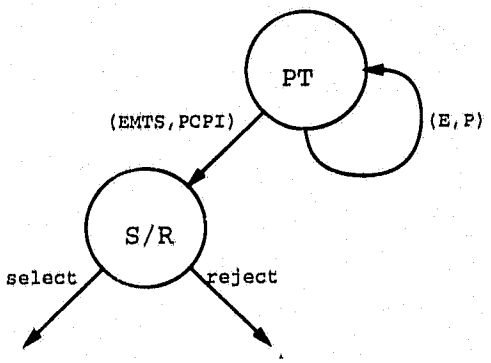


Figure 9.9: Partitioning Model

9.11 Results

This section compares characteristics of the new exact minimizer, *MDSA-exact*, with the best available exact minimizer, *espresso-exact* [RSV87][2]. (The *McBOOLE* minimizer ([DAR86]) is not competitive. A more recent, apparently powerful exact minimizer ([BMS92, BMSSV93]) is not available.) Experiments are confined to those fourteen PLA's of the benchmark set that contain true cycles. The results of the experiments are presented in Tables 9.3 and 9.4. Table 9.3 gives characteristics of the data structures and search requirements of the two programs, and Table 9.4 gives corresponding timing and memory use results. Blank and bracketed entries indicate that accurate numbers could not be extracted because of failure of the minimizer.

The first three columns of each five-column main section of Table 9.3 give dimensions of the prime implicant tables stored within the minimizers upon isolation of (potential) cycles. When a PLA contains several smaller partitions along with one

[2]The exact version of *espresso* is available by specifying the **-Dexact** option.

PLA				espresso-exact					MDSA-exact				
name	n	m	p	cubes	mints	entries	nodes	soln	cubes	mints	entries	nodes	soln
dist	8	5	120	376	313	1554	10	97	45	41	100	3	20
f51m	8	8	76	544	425	3323	6	63	14	15	30	2	7
mult4	8	8	121	594	530	3113	1619	109	313	302	1283	1342	89
sqr6	6	12	47	202	188	995	330	44	150	135	555	49	41
5xp1	7	10	63	381	388	2719	27	55	119	129	418	6	37
in0	15	11	107	323	189	1226	3	47	6	6	12	1	3
in2	19	10	134	327	376	2061	17	49	32	36	81	4	14
sym9	9	1	84	1680	420	13440	84	84	1680	420	13440	118	84
sym10	10	1	210	4200	792	33600	103	210	4200	792	33600	105	210
ex1010	10	10	*267	25203					11568	1466	63965		(268)
apex4	9	19	427	2214	1418	7330	38	368	180	172	431	7	108
sev	8	10	*195	931				(179)	683	473	2285		(163)
tial	14	8	575	5981	4468	43855	55	355	113	212	424	20	55
misex3	14	14	*653	7107	1095	61583	1.0M	60	389	561	7769	3.5M	60
misex3	14	14	*653	5126				(382)	2234	2352	20662		(380)
misex3c	14	14	190						697	837	15850	1093	39
misex3c	14	14	190	7107	1095	61583	1.0M	60	389	561	7769	3.5M	60

Table 9.3: Internal Cycle Characteristics

main partition, their dimensions are all summed together. The exceptions are `misex` and `misex3c`, which each contain two very large non-separable partitions. For the purposes of the current experiment, the partitions are first isolated using the techniques of Section 9.3. This is necessary to extract meaningful results, because neither program can minimize the non-partitioned `misex3`, and only MDSA-exact can minimize the non-partitioned `misex3c`. Note that there is apparently one cycle shared by both PLA's. Neither minimizer could find the minimum solution for `sev` or `ex1010`.

The first two columns, *cubes* and *mints*, give the number of rows and columns in the equivalent tabular representation of the covering problem. It is unknown precisely how tables are represented within *espresso-exact*; they are stored as cube and minterm heaps within MDSA-exact. The *entries* column gives the number of cube/minterm containment relations within the cycle. In MDSA-exact, this number equals the sum of all minterm cover counts, also equal to the sum of all cube weights. The *entries* numbers are equivalent to numbers of prime implicant table "marks", or "checks" [Man91]. The blank entries for the table dimensions of `ex1010`, `sev`, and the larger

cycle of `misex3c`, are because of an inability to extract these dimensions from the debug output of *espresso-exact*. That minimizer is unable to construct the PI table for one of the `misex3c` cycles because of memory problems.

For almost all benchmark PLA's, the prime implicant table dimensions are smaller in MDSA-exact. This is because *espresso-exact* removes essential cubes only, before construction of the table. No pseudo-essentials are found until the table is set up (although some dominated prime implicants are removed). In contrast, the tables are fully reduced in MDSA, before branching begins. The only two PLA's where the table dimensions are the same size in both minimizers are `sym9` and `sym10`. Both these PLA's contain zero essential prime implicants. It is reassuring that both minimizers report the same table dimensions for these PLA's.

The last two columns of each main section of Table 9.3 give characteristics of the branch-and-bound search used to extract minimum covers from the tables. The *nodes* columns give the number of cube selection/rejection search space nodes visited while determining the minimum solution. The number of rejection branches that are traversed is always less than or equal to *nodes*, because low bound calculations may cause rejection branches to be bound. The *soln* columns give the number of cubes eventually found to be in a minimum solution of the corresponding covering problem. These latter numbers differ for the two minimizers because *espresso-exact*'s "cycles" are not fully reduced. For example, the PLA `mult4` contains 12 essential cubes, which is why *espresso-exact* finds the minimum solution for the "cyclic" part to have 109 cubes. MDSA-exact finds 20 more pseudo-essential cubes, then reports the minimum solution of the true cycle to contain 89 cubes. Both minimizers correctly report a total minimum of 121 cubes.

If the reduction algorithms of *espresso-exact* work correctly (*i.e.* full reduction between heuristic selection/rejection), the solution of "potential cycles" begins with full

reduction (and partition) down to the size of the corresponding table in MDSA-exact, at the cost of zero search space nodes. This hypothesis is suggested by the *espresso-exact* minimization of non-cyclic PLA's, which require the construction of non-trivial prime implicant tables, that are subsequently solved without any branching[3]. In other words, a table is created, then reduced to nothing by the repeated selection of pseudo-essential cubes, and removal of dominated cubes and dominating minterms. It is confirmed on all non-cyclic PLA benchmarks that *espresso-exact* does indeed never require heuristic selection for them.

The bracketed entries in the *soln* columns give the best results before early termination of the running minimizers, on the three unsolvable cycles of `ex1010`, `sev`, and `misex3`. MDSA-exact has an unfair advantage for these numbers, because they were extracted by running the minimizer with a restriction on maximum branching depth (a technique suggested in [DAR86]). Also, the figure 179 for *espresso-exact*'s minimization of `sev` is actually within one cube of MDSA-exact's 163, because the 179 figure includes 15 pseudo-essential cubes. All these cycles are extra-ordinary because of the depth at which the initial solutions are found: 170 for `ex1010`, 34 for `sev`, and 113 for `misex3`. *Espresso-exact* is unable to find even one solution for `ex1010`, even after running for over twenty hours.

Both minimizers find the exact solutions for the difficult benchmarks `sym9` and `sym10` with a minimum of branching. In fact, the 84 nodes visited by *espresso-exact* for `sym9` are in one long chain, because the first low bound calculation returns 84 at the root of the search tree, and an 84 cube solution is found right away, at a depth of 84 in the tree. Thus, all rejection branches are immediately bounded. A

[3]It would be informative to include some of these non-cyclic PLA's in the results, to show the penalty experienced by *espresso-exact* in creating prime implicant tables where none are really necessary.

similar phenomenon occurs with `sym10`, this time in both minimizers. There is no way these PLA's could be solved without the powerful low bound calculations and selection heuristics, because of the exceptional nesting depths. The selection heuristic greatly benefits from the technique of rejecting cubes not involved in the maximal independent set low bound calculation. This is why MDSA-exact is able to find the minimum solutions with a minimum of branching, but the heuristic version is far off the minimum (see Table 9.2).

The *nodes* numbers are generally better with MDSA-exact. This could be due to (1) a mistaken interpretation of the "nodes" numbers in the debug output of *espresso-exact*, or (2) *espresso-exact* does not correctly fully simplify the prime implicant table between heuristic selections. It is unlikely that the phenomenon is due to better low bound calculations or cube selection heuristics in MDSA-exact, because these are borrowed from [RSV87], with the exception that "tie-breakers" for heuristic selection are not employed by the latter minimizer.

The significant exception to better *nodes* numbers for MDSA-exact is the extremely difficult example of the second cycle of `misex3c`. The best solution of 60 cubes is the third solution found by *espresso-exact*, at a depth of only 15. The maximum depth reached in the search is 59. However, the 60 cube minimum is the fifth solution found by MDSA-exact, and it is found at a depth of 22[4]. The first four solutions are found at depths 14, 15, 15, and 22. There is evidence to suggest that MDSA-exact does not choose the best cube for heuristic selection/rejection. It is unclear why this is so.

Timings and memory usage for *espresso-exact* and MDSA-exact are given in Table 9.4. Timings are split across three columns. The first, *other*, refers to the time

[4] Unfortunately, the output of MDSA-exact does not include a figure for maximum depth reached.)

	PLA			TIME (seconds) espresso-exact			MDSA-exact			MEMORY (kbytes) es-ex	MDSA-ex
name	n	m	min	other	cycle	total	other	cycle	total		
dist	8	5	120	2.33	0.16	2.49	0.18	0.03	0.21	496	328
f51m	8	8	76	2.98	0.28	3.26	0.08	0.02	0.10	548	320
mult4	8	8	121	4.40	35.30	39.70	0.40	128.20	129.60	1204	420
sqr6	6	12	47	0.84	4.04	4.88	0.10	2.80	2.90	652	356
5xp1	7	10	63	1.92	0.39	2.31	0.20	0.37	0.57	544	360
in0	15	11	107	8.13	0.09	8.22	1.14	0	1.14	488	1344
in2	19	10	134	7.17	0.20	7.37	12.80	0	12.80	512	13412
sym9	9	1	84	20.00	78.60	98.60	1.80	43.50	45.30	7552	460
sym10	10	1	210	109.00	663.00	772.00	5.00	258.00	263.00	27000	600
ex1010	10	10	*267	9400			24.00			*60000	1640
apex4	9	19	427	56.20	2.70	58.90	1.23	0.65	1.88	780	468
sev	8	10	*195	10.90			0.90			*3000	408
tial	14	8	575	440.60	19.80	460.40	5.04	0.26	5.30	2004	1048
misex3	14	14	*653	165714	244661	410375	15.10	1040851	1040866	8000	1500
misex3	14	14	*653	327.00			18.30			*18000	1276
misex3c	14	14	190	90000			109.00	2779.00	2888.00	*32000	2536
misex3c	14	14	190	165714	244661	410375	15.10	1040851	1040866	8000	1500

Table 9.4: Time and Memory Usage for Exact Cycle Solutions

taken to perform operations other than finding the minimum solution for the prime implicant table. The *cycle* figures give the times taken to find these minimum solutions. Thus, *total* time is *other* time plus *cycle* time.

In general, there is a correspondence between the prime implicant table dimensions given in Table 9.3, and the resource requirements given in Table 9.4. Again, MDSA-exact appears to be superior, even when comparing only the *cycle* figures. (It is already known that MDSA-exact is better at isolating the core.) The two exceptions (at least for timings) are `mult4` and the second cycle of `misex3c`. In the case of `mult4`, the figures indicate that MDSA-exact takes about five times as long as *espresso-exact*, per node visited. This is the price paid for avoiding the construction of an explicit minterm dependency graph for low bound calculations; the time taken by MDSA-exact is dominated by calculation of dependency counts and low bounds. The time figures for `misex3c` tell more than the obvious result that *espresso-exact* is faster. MDSA-exact takes 4.2 times the time (1040851s vs. 244661s) to search 3.5 times

the nodes. Thus, the advantage that *espresso-exact* has for time-per-node values, apparently diminishes for larger cycles.

Memory figures are averages returned by the UNIX utility *time*. In general, MDSA-exact uses far less of this resource. For *espresso-exact*, memory requirements depend both on sizes of the cycles, and also the size and complexity of low bound calculations. (The four Entries preceded by asterisks indicate estimates, because those cycles cannot be solved by *espresso-exact*.) In such cases, the figures are obtained using the *ps* program.) MDSA-exact has no such dependency; it requires only a linear amount of storage for the cube and minterm records. In MDSA-exact, memory requirements are still dominated by the truth-table, which explains the large figure for `in2`.

9.12 Summary

This chapter shows how true cycles are processed by the MDSA minimizer, in both heuristic and exact modes of operation. In both modes, minimization is made more efficient by maintaining the set of prime implicants as a heap. Exact minimization also requires structuring of the minterms in a heap, so that low bound calculations using maximal independent sets of minterms are performed efficiently.

The "Rudell weight" of a cube is apparently the best heuristic to use for cube selection, in both heuristic and exact modes. The heuristic MDSA mode is equivalent to the exact mode, where branching is restricted to zero, and low bound calculations are suppressed. This heuristic minimizer almost invariably returns better results than *espresso*, but suffers for PLA's like `misex3c`, where much work is required to isolate the cycles. An algorithm for extracting irrelevant covering cubes ensures that heuristic solutions are not only prime, but also irredundant.

The improved branch-and-bound model for exact minimization corrects several deficiencies in that presented elsewhere [RSV87]. In particular, the computation required to change portions of the implicit prime implicant table are *localized*, by passing changes as parameters and performing tree traversals. The detection of induced pseudo-essential cubes is handled in a more natural way; they are simply prime implicant sets of cardinality one. Low-bound calculations are used to maximum advantage.

A notable improvement of MDSA-exact over other exact minimizers is its requirement for only linear memory, both for the prime implicant table information, and for the calculation of lower bounds. Also, since in MDSA exact minimization commences with fully reduced cycles, more PLA's can be minimized than competitors which require larger initial data structures. Reduced memory requirements come at the cost of decreased speed, but empirical data shows this cost to grow smaller as input problems grow larger. This is due to the behaviour of cube trees and heaps; the representation tends to pay off more as they grow in size.

Chapter 10

Set Based Directed Search

10.1 Introduction

The one remaining significant limitation of the MDSA minimizer is its inability to handle a problem with more than a "medium" number of inputs n and outputs m. With current machines, n is limited to about 20, and the data structures used in the implementation limit m to 32. These restrictions are due in part to the reliance of MDSA on the full truth-table of a PLA. In addition to this array of size $m2^n$, MDSA uses an array of similar size to keep track of the cube cover counts of each minterm. Removing the counts array would enable the minimization of only slightly larger PLA's. What is needed is a method of storing the original multi-function, and its subsequent images (as "don't-cares" are introduced), in less than $O(m2^n)$ storage, while simultaneously maintaining the correct cover counts for the relevant minterms.

The worst-case memory requirement of just about any practical scheme is still $O(m2^n)$. For example, the XOR function of n variables requires a list of 2^n cubes. Thus, whether or not it is even possible to use less storage, in the worst case, is not

the concern here. Instead, it is sensible to take the approach of many other cube-based minimizers — the sum-of-products form is used not only for input and output, but also for internal representations. There is some logic to this approach — only those functions with a relatively compact sum-of-products form would be interesting subjects for two-level minimization. If the input problem, given as one or more lists of cubes, is not unreasonably large, then hopefully the internal representation will likewise not explode.

For minimizers that are based on direct cube manipulations, the avoidance of truth tables and reliance on cube set representations is natural. Avoiding truth tables is more of a challenge with a minterm-based algorithm like directed search, since so many of the basic elements involved in the algorithm are minterms and their properties (*e.g.* in MDSA, adjacencies and cover counts). However, all facets of the directed search algorithm, including the extensions introduced in this research, can be adapted to work with cube sets, rather than directly indexed arrays of exponential size. This is the subject of the present chapter.

10.2 Data Requirements of MDSA

A cube-based specification of a multiple-output discrete function consists of one or more lists of cubes. There are various syntactic rules to interpret the meanings of cubes, but in any case the representation specifies the zero-space, one-space, and don't-care-space of each function in a PLA. These subspaces are often called the *ON-set*, *OFF-set*, and *DC-set* [BHMSV85]. Only two subspaces need be specified, in which case the third is implicit.

In MDSA, prior to the actual minimization of a PLA, the set of new minterms NMTS is equivalent to the ON-set. During minimization, the ON-set shrinks as

"don't-cares" are introduced. In MDSA this is accomplished by actually rewriting the truth-rable with "don't-care" values. Minimization is finished when the ON-set is empty and all cycles have been resolved. Thus, the ON-set is a dynamic quantity, and any data structure that replaces the truth table must support this property. The truth table of MDSA also holds zero vertices, which collectively are equivalent to the OFF-set of a function. Unlike the ON-set and DC-set, the OFF-set remains static throughout a minimization. Thus, its requirements for a replacement data structure are easier to meet.

During an MDSA minimization, the active minterm sets are (1) the new minterms (NMTS), (2) the recursive minterms (RMTS), and (3) the expanded minterms (EMTS). All three change throughout a minimization, and therefore their representations must be flexible enough to permit frequent dynamic changes. In the map-based version of MDSA, the NMTS is not explicitly stored as a minterm set. Instead, the inclusion of a minterm in this set is indicated by the minterm's coding in the truth table. In contrast, the RMTS is stored as an explicit set of minterms, though as a simple linked list. Only the EMT exists as a tree-heap (Chapter 6), because of the requirement for finding pseudo-essential cubes efficiently.

A set-based version of MDSA requires an efficient representation of all three minterm sets, with the cover counts of each minterm available. However, there is a need to avoid storing *each* minterm as a *distinct* record, or else the minimizer is still stuck with exponential storage requirements. Thus, minterms must be "coalesced" into cubes in such a way that counts are still recoverable.

10.3 Generation of Prime Implicants

In most PLA specifications geared for *espresso* minimization, the OFF-set is implicit (type `.f` or `.fd`) [BHMSV85]. That is, it is assumed to consist of all vertices of a function space not already accounted for in the input cubes, which represent cubes of 1's and "don't-cares". However, an explicit internal form for the OFF-set is required by *espresso* for "expansion" of cubes, and thus the set must be generated. More generally, if an explicit representation of one of the three sets is needed, but not given, then it must be generated by finding the complement of the union of the other two sets.

In MDSA, a simple way to generate the prime implicant set for a given minterm is to adapt the HYPER algorithm of Chapter 7. All that is required is to make the `TestCube` primitive work with the OFF-set, rather than the truth-table. Each `TestCube` call involves finding out if a cube C is compatible. The map-based MDSA does this by accessing the function value at each minterm contained in C. If one is found to be zero, the test immediately halts with failure. If instead the OFF-set is in the form of a set of cubes, then if any member cube D of zeros intersects with C, that implies that C contains one or more zero minterms, and is therefore incompatible.

The requirement for the new set-based `TestCube` primitive is therefore a mechanism for determining if an intersecting cube of the OFF-set exists. The cube tree representation of Chapter 6 is a logical choice for the OFF-set. Finding out if a member cube D intersects with C requires an intersection traversal. `TestCube` calls that eventually result in failure, terminate upon discovery of the first intersecting OFF-set cube. Successful `TestCube` calls require the entire search subtree to be examined, thereby proving that no members of the OFF-set are contained in C. This is in contrast to the original map-based version, which must examine every vertex contained

	PLA			OFF-set	MAP BASED		SET BASED		
name	n	m	min	size	calls	probes	calls	visits	hits
alu1	12	8	19	20	3344	83504	3425	36069	19543
alu2	10	8	68	48	3623	27501	4197	84489	19308
alu3	10	8	65	51	3850	22512	4403	90222	26984
add4	8	5	75	83	1211	2582	1819	35995	8418
add6	12	7	355	387	24593	104001	28294	1500778	245204
dist	8	5	120	146	2142	4329	3280	77104	16595
in2	19	10	134	123	5036	1484880	7484	214971	67697
dk48	15	17	22	49	406	196777	628	17199	9388
apex4	9	19	427	457	13283	17135	20510	639736	59162
tial	14	8	575	395	37391	384627	46987	2933635	349832
misex3	14	14	653*	355	175372	717389	203822	9567021	2004674
misex3c	14	14	190	365	10340044	99850563	10046143	609699541	117094046
vg2	25	8	110	173	-	-	5927	335326	59096
duke2	22	29	86	99	-	-	9681	418324	85771
misex2	25	18	28	45	-	-	950	26212	11201
bc0	26	11	177	412	-	-	49428	7186032	1065238
chkn	29	7	140	170	-	-	9446	675820	133843
in4	32	20	211	247	-	-	28536	2734529	479228
in5	24	14	62	145	-	-	4104	300323	47577
in7	26	10	54	66	-	-	8914	357062	121312
rckl	32	7	32	32	-	-	7296	126612	15279
x1dn	27	6	110	158	-	-	5863	422143	48470
x9dn	27	7	120	159	-	-	7110	453585	52100

Table 10.1: OFF-Set Size and TestCube Requirements

in the cube.

Table 10.1 gives some of the computational requirements of the old map-based `TestCube` primitive versus the new set-based one. The OFF-sets for the second experiment are generated using *espresso* to generate the complement of the ON and DC sets, in fast mode (the command is `espresso -Decho -epos`). Note that neither a minimal nor even irredundant representation is required. (No effort is made here to describe complementation algorithms, for which several efficient techniques already exist [Sas85, Cha87, HO72, BHMSV85].) The OFF-set sizes are given in column five. Columns six and eight give the total number of `TestCube` calls required in each experiment. More calls are required for the set-based MDSA because the prime implicant generator must determine the immediate adjacencies in each variable for expanded minterms. These are automatically available in the map-based version,

since adjacencies for expanded minterms are always available.

The time required by the map-based `TestCube` is estimated by the total number of "probes" required; that is, the number of vertices examined by the primitive in the search for a false one. The complexity of the set-based primitive is estimated by the total number of OFF cubes visited in the traversals ("visits"), and the number of these cubes that were actually relevant, because they intersected with the tested cube C ("hits"). The "visits" figures are almost directly proportional to the time required. The structure of the OFF-set tree affects the performance of the primitive, and therefore so does the input variable ordering. Unfortunately, experiments comparing the set-based primitive under different variable permutations are not completed.

Note the addition of eleven new PLA benchmarks to the table. These new PLA's are too large to be minimized by the map-based MDSA, but can be handled by the set-based version. The data of Table 10.1 suggests that the set-based version of MDSA is substantially slower than its map-based counterpart, just from prime generation, at least for most of the smaller PLA's. However, larger PLA's can now be minimized. In particular, note that in the case of `dk48` and `in2`, with 15 inputs and 19 inputs respectively, the new prime generator is just starting to require fewer operations than the old. This is because the primes that are generated in each case are quite large on average, and therefore expensive for the map-based version to calculate because so many probes per cube are required.

10.4 Minterm Sets as Disjoint Cube Trees

This section addresses the problem of keeping track of minterms and their counts. (Recall that the *count* of a minterm is the number of primes thus far discovered that contain that minterm.) In the map-based version, the counts are kept in their

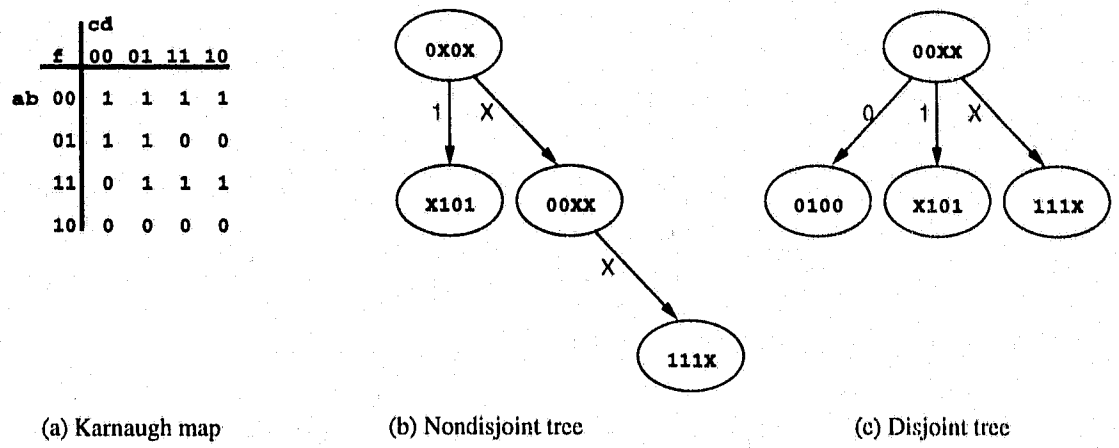
own array COUNTS of exponential size. Counts are required only for the expanded minterms (EMTS) and recursive minterms (RMTS), which means that a large part of the array remains empty, corresponding to the false vertices and the "don't-care" vertices of the function space. An obvious first step is to keep the counts with the minterms' records, whatever form these records may take.

The map-based MDSA creates a new record for each newly-discovered recursive minterm, when generated prime implicants are found to cover new minterms (see Chapter 4). Sometimes, RMTS gets very large[1]. For example, in the minimization of `in2`, many hundreds of thousands of minterm records are created in the first few iterations, because very large primes were the first to be discovered. Thus, it is not practical to keep the RMT minterms as individual records.

A set S of minterms can be represented by a set of cubes S_C, such that each minterm in S is contained by one or more cubes of S_C, and there does not exist any minterm covered by a cube of S_C that is *not* in S. If every minterm in S is covered by precisely ONE cube in S_C, the cubes are *disjoint.* In general, a minimizer like MDSA or *espresso* finds a *nondisjoint* cover for the ON-set of minterms, because cube overlap is permitted.

The cube trees of Chapter 6 may be used to hold sets of disjoint cubes. Figure 10.1 shows a nondisjoint tree and a disjoint tree covering the same set of nine minterms in the given Karnaugh map. In general, one would expect the nondisjoint tree to contain fewer cubes. Certainly, the minimum required cubes to cover a given set of minterms is no greater for nondisjoint sets compared to disjoint ones. However, the disjoint set may be very nearly as small in some cases. A disjoint tree is typically

[1]Interestingly, almost all the benchmark PLA's used in this research are sparse, *i.e.* their OFF-sets are much larger than their ON-sets. Thus, blow-ups in the number of RMT records occur when trying to minimize the inverse phase of each output function.



(a) Karnaugh map (b) Nondisjoint tree (c) Disjoint tree

Figure 10.1: Nondisjoint and Disjoint Cube Sets

"wider", because the cubes are smaller and therefore are more likely to be reached down non-DC branches.

A cube set can be used for holding the set of recursive minters RMTS, where all minterms included in a given cube record have an identical cover count. A disjoint representation is the one chosen for MDSA. Both NMTS (the set of new minterms) and RMTS are stored as one set MTS (simply, *the minterms*). The structure of the set is a cube heap, as described in Chapter 6. New minterms, *i.e.* the former members of NMTS, are now considered to be just "minterms" with a cover count of zero. The former members of RMTS are just "minterms" with a cover count greater than zero.

The heap structure is required so that appropriate minterms may be chosen for expansion. Recalling from Chapter 4 that lower cover counts have higher priority, the "lightest" records are at the top of the heap. There is one exception to this rule; the minterms with zero count reside in records at the bottom (leaves) of the heap, because they are of *lowest* priority. (Recall that directed search stipulates that all

RMT's be exhausted before NMT's are chosen.) If the heap property is maintained, it is known that a minterm suitable for expansion is contained by the cube record at the top of the heap.

10.5 Maintenance of the Minterm Set

This section shows how a minterm set in the form of a heap of disjoint cubes is correctly maintained throughout an MDSA minimization. Three main operations are required: *initialization*, *removal*, and *adjustment*.

10.5.1 Initialization

The MTS set is first constructed from the ON-set of the problem specification, or from the complement of the problem specification, as the case may be. The ON-set is typically non-disjoint, so its members cannot simply be inserted into the MTS tree. Instead, every cube C in the ON-set to be added must first be checked to see if it intersects with any member D of MTS thus far inserted. If no intersection is found, then C can be added to the MTS as is. If, on the other hand, an intersection is found, then the original cube C to be added is broken up into the *disjoint cell difference* $C - D$ of it with respect to the intersecting cube (Chapter 5). Then, the same operation is carried out recursively on each of the new cubes $C_1, C_2, ..., C_j$ comprising the difference. The preorder tree intersection traversal of Chapter 6 is used in this process.

When a cube C to be added is found to be contained by a cube D already in the MTS set, then C is simply removed; its minterms are already in the MTS set. A containing cube D, if one exists, will be found because of the structure imposed on

cube trees and the nature of the intersection traversal. If the containing cube D does indeed exist, there is no way another containing or just intersecting cube in MTS could be found first, because that would violate the assumption that the MTS set is disjoint. Addition of records is always at the leaves, because the heap operation of bump insertion is not required, since all records have an initial cover count of zero.

The cubes of minterms that are to be added to MTS would ideally be initially prime, though that is certainly not a guaranteed state for the input, unless it was preprocessed in some appropriate way. In any case, it works out much better if the cubes to be added are processed in order of decreasing size (*i.e.* the number of minterms they contain), in an attempt to keep the minterms of MTS as clustered together as possible.

MDSA first reads the input cubes of the ON-set into a cube heap H based on cube size, with largest ones toward the root (top). H is then simply a new nondisjoint representation of the ON-set. Then, an iterative procedure commences whereby the cube C at the root of H is removed, and an intersection traversal is performed on MTS with C. If C does not intersect with any members of MTS, it is added to MTS at the appropriate leaf. If C must be broken up into difference subcubes, these are all heap-inserted back to H. The process continues until no cubes remain in H. Then, MTS is a disjoint representation of the ON-set.

Figure 10.2 shows the effect of adding a cube C to an MTS set. In this example, the cube C =`XX111`, comprising the entire ON-set, is removed from the ON-set and subsequently found to intersect with `01XXX` at the root of MTS. (C also intersects with `111X1`, but this is not discovered since a preorder intersection traversal is used.) The difference `XX111`−`01XXX` is found, yielding the two new cubes `X0111` and `11111`, which are heap-inserted back to the ON-set. Then, the root `X0111` is removed from the ON-set, and added to MTS at the appropriate leaf since no intersecting cubes in

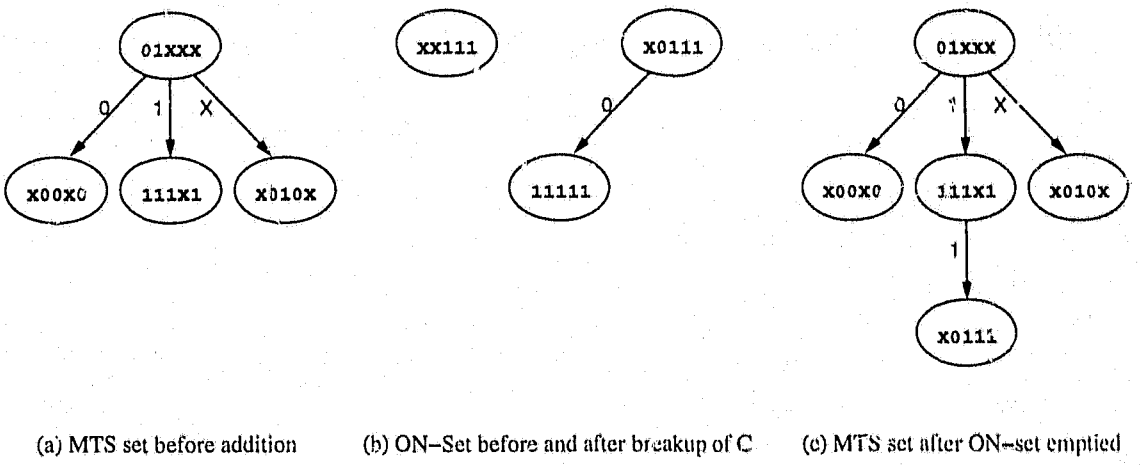


Figure 10.2: Addition of a Cube to the MTS Set

MTS are found. Finally, the last remaining member of the ON-set, cube `11111`, is found to be contained by `111X1` in MTS, so no change occurs to MTS.

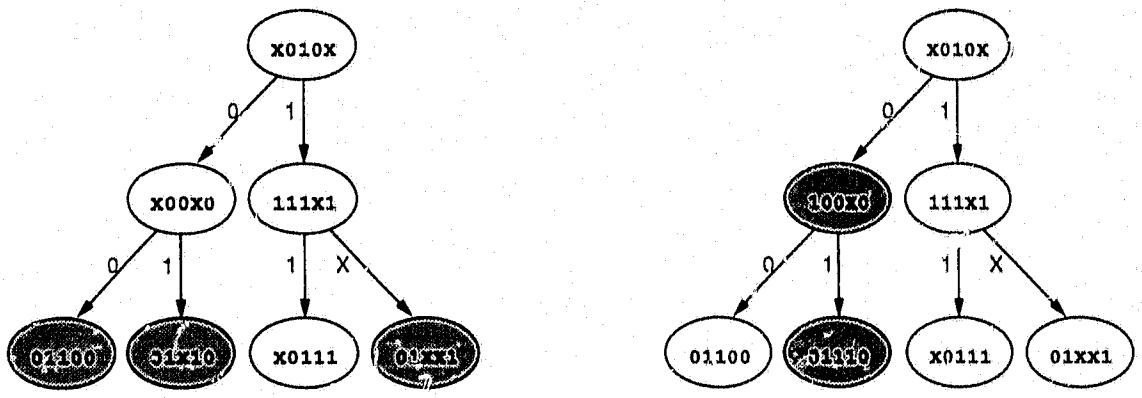
10.5.2 Removal of Minterms and Cubes

Minterms to be removed from MTS come from (1) minterms selected for expansion, (2) cubes selected for the cover, or (3) intersection cells of prime implicant sets. In case (1), the minterm-to-be-expanded M is a member of the cube C at the top of the MTS heap (because this is the mechanism for choosing M). If C is the minterm itself ($C \equiv M$), then C is simply heap-removed, with lower records bubbling up through the heap to take C's place. If the top record is a larger cube that properly contains M, then C is replaced with the difference $C - M$. One of these new cubes takes the place of C (this is permitted, because the counts of the difference cubes remain as yet uncharged, and any cube may legally occupy the root position). The remaining difference cubes $C_2, C_3, ..., C_j$ are heap-inserted to MTS.

In both cases (2) and (3) from above, the minterms to be removed are themselves represented by a cube R. An intersection traversal of the MTS set with R is performed. This time, the traversal does not halt with the first intersecting cube found. Each cube record D found to intersect with the removed cube R is removed from MTS. The disjoint cell difference $D - R$ is found, and these new cubes $D_1, D_2, ..., D_j$ are temporarily stored in a set S until the traversal of MTS with R is complete. Then, all cubes in S are heap-inserted back into MTS. No checks for overlap are required for the difference cubes of that are now members of S, since they are all subcubes of cubes that were already disjoint. A list of cubes is sufficient to store S. Because nodes in MTS could be removed during traversal, the traversal must be postorder.

The disjoint property of MTS permits a small optimization in the removal of a cube of minterms. When possible, the cube R is reduced to a smaller cube R', which in turn restricts the branches that the traversal for that cube must take. This simplification is possible when the difference $D - R$ is a single cube. When the cube R is fully contained by a cube encountered in the traversal, then the traversal can halt because no other cubes in MTS could possibly intersect with R.

Figure 10.3 shows the effect of removing first a minterm and then a cube from the example MTS tree of Figure 10.2(c). The minterm M =01000 is chosen for expansion, since it is the "first" member of the root cube of MTS. The minterm is removed from the record, which causes 01XXX to be removed and split up into the three new difference cubes 01XX1, 01X10 and 01100. Removal of 01XXX causes it to be replaced as root by X010X. (Recall from Chapter 6 that nodes down "don't-care" branches are the best candidates to replace removed nodes.) The three difference cubes are subsequently re-inserted into the MTS set, yielding the tree in Figure 10.3(a). The new additions to the set are shown as shaded nodes. (The heap property of MTS is still not of concern; in this example all counts of all records are zero.)



(a) After removal of M = 01000

(b) After removal of R = 0X0X0

Figure 10.3: Removal from the MTS Set

Now, assume that the prime implicant 0X0X0 is discovered from expansion of M, and is chosen for the cover for one reason or another. (For this example, it is not of concern whether or not 0X0X0 might possibly be pseudo-essential, or even prime.) Then, R =0X0X0, and the first intersecting cube found in a postorder traversal is D =01X10. The difference $D - R$ is 01110. Since the Hamming distance between R and 01110 is greater than one, there is more than one cube in the difference, and therefore no simplification of R takes place here. However, since there is only one cube in $D - R$, and since that cube can legally occupy D's position, it simply replaces D.

The postorder traversal backs up towards the root, and D =X00X0 is found to intersect with R. $D - R$ =100X0 is again a single cube, and also can legally replace D. This time, however, the difference $R - D$ is a single cube, so R becomes 000X0) for the remainder of the traversal. In this example, the remaining part of MTS that must still be traversed consists only of the root, which does not intersect, so MTS

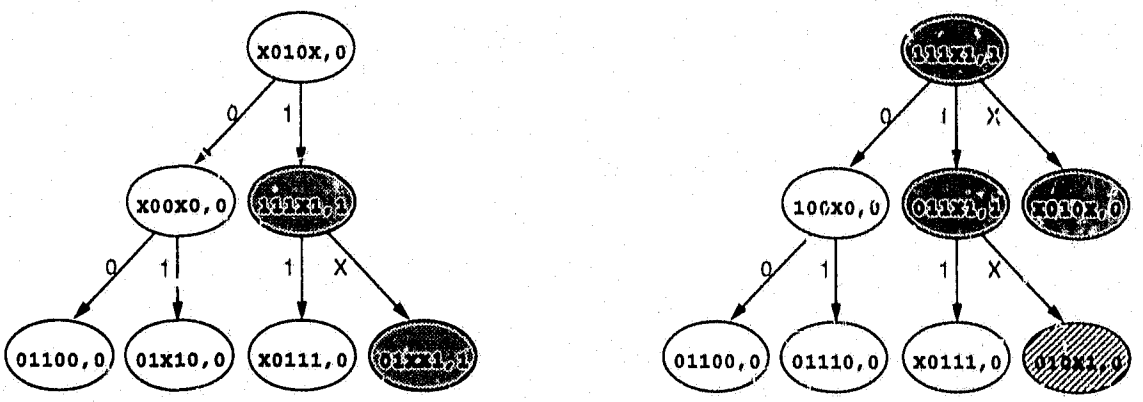
is left as the tree shown in Figure 10.3(b). The two nodes which changed to their respective single-cube differences because of intersection with R, are shown shaded.

10.5.3 Count Adjustments

The counts of minterms must be incremented when prime implicants are added to the PCPI, and decremented when primes are removed. The same procedure is used for both cases. Both operations involve yet another postorder intersection traversal of the set. When an intersection with some prime C is found with some MTS member record D, if C fully contains D, then the count of D is simply incremented (if C is being added to PCPI), or decremented (C has been found to be dominated, and is being removed from PCPI). Whether or not D is changed or even intersects with C, a heap adjustment operation on D is performed, to ensure that it is correctly positioned with respect to the subtree beneath it (Chapter 6).

If D is not fully contained, then the count adjustment applies only to that part of D that is also in C. The cube record D is replaced with the intersection of C and D, and the new record gets an adjusted count. New cube records with the original count of D are created from the cell difference $D - C$, and are added to the list S of cube records which will all be added to the heap at the end of the postorder traversal. If the intersection cube record can no longer legally occupy the position in the tree in which it was created, because of a variable change, the intersection cube must also be removed to the list S. After the intersection traversal of MTS with C, members of the set S are added back to MTS. No more intersections exist, so a simple heap-insertion is sufficient for each record.

Figure 10.4 shows the node changes and heap adjustments required by the addition of a new prime C =X11X1 to the PCPI, using the example MTS tree of Figure 10.3(b).



(a) After counts have been adjusted (b) After heap adjustment and insertion of S

Figure 10.4: Adjustment by a Prime of the MTS Set

Now, the records are augmented with the appropriate cover counts. The first intersecting record encountered in the postorder intersection traversal is D =01XX1. The intersection $D \cap P$ is 011X1, which may legally occupy D's position. This new cube gets a new cover count of one, and is checked for heap correctness against its subtree. The latter is empty, so the correctness is trivially true. The difference $D - P$ is the single cube 010X1, which retains D's original cover count of zero, and is moved to the holding set S. The difference $P - D$ is the single cube 111X1, which becomes the new P'.

The next intersecting record encountered in the traversal is 111X1. It becomes the intersection $D \cap P'$ =111X1. This time, no difference cubes exist, and P' is obliterated because the difference $P' - D$ is also empty. The situation is now as shown in Figure 10.4(a), where the affected nodes are shown shaded. The traversal is not complete however. The root need not be checked for intersection, but it does require heap adjustment since its cover count of zero is of less priority than its child's

PLA						MTS SET SIZE					
						NO HEAP		HEAP1		HEAP2	
name	n	m	min	minterms	density	start	max	start	max	start	max
alu1	12	8	19	15872	48	25	5500	19	5207	19	4518
alu2	10	8	68	5225	63	157	3185	84	2626	84	2028
alu3	10	8	65	3903	47	147	1488	67	899	67	975
add6	12	7	355	14304	49	902	1094	355	1388	355	1237
add7	14	8	735	65472	49	3801	10908	735	7768	735	7797
dist	8	5	120	591	46	182	436	147	404	137	367
in0	15	11	107	84196	23	309	1183	123	847	119	1133
in1	16	17	104	328488	29	193	833	118	626	110	453
in2	19	10	134	686336	13	297	653	194	520	183	497
sym9	9	1	84	420	82	184	382	165	382	165	382
dk48	15	17	22	130268	32	46	85	25	82	25	82
mult4	8	8	121	678	33	225	438	225	494	225	506
apex4	9	19	427	2770	28	595	1075	514	1084	492	944
sev	8	10	195*	1220	47	415	896	294	731	271	765
tial	14	8	575	62256	47	1982	20672	1252	10679	1196	9347
sym10	10	1	210	792	77	792	792	792	792	792	792
misex3	14	14	653*	23196	10	2342	8561	2068	7986	1357	7517
misex3c	14	14	190	23196	10(41)	3830	10702	2132	10544	1720	12784
vg2	25	8	110	61570752	22	1316	19737	243	1168	243	335
duke2	22	29	86	8464768	6	260	3372	97	764	95	465
misex2	25	18	38	37257216	6	29	29	29	29	29	29
bc0	26	11	177	284933120	38	870	18480	261	11960	252	7002
chkn	29	7	140	788036832	20	33537	70830	343	370	343	433
in4	32	20	211	1.33e9	15	12399	506803	391	63287	391	79182
in5	24	14	62	24912896	10	406	612	135	365	130	281
in7	26	10	54	220769280	32	128	27712	57	20223	57	13055
rckl	32	7	32	36581120	0	32	182	32	314	32	506
x1dn	27	6	110	133035072	16	1535	31580	233	250	230	269
x9dn	27	7	120	133041984	14	1452	21819	246	573	245	587

Table 10.2: Behaviour of Disjoint MTS Sets

count of one. Thus, `X010X` is heap-removed, `111X1` rises up to take its place, and `010X1` rises up to take the latter's place. `X010X` is then heap-inserted back into MTS. Finally, the sole member `010X1` of the holding set S is heap-inserted into MTS. The result is the tree shown in Figure 10.4(b), where all moved nodes are shaded.

10.5.4 Dynamic Behaviour of MTS

Table 10.2 shows the behaviour of the MTS set for several of the benchmark PLA's. The column "minterms" gives the number of minterms in each PLA; the column "density" gives the fraction (as a percentage) that these minterms occupy in

the entire function space. (An additional 31% of misex3c is "don't-care".) The first of the three rightmost main sections ("NO HEAP") corresponds to an experiment in which initialization of MTS is accomplished by inserting ON-set cubes in the order in which they appear in the input data. Figures in the second experiment ("HEAP1") are for first ordering the ON-set cubes as a heap, and then inserting them into MTS in descending order. This strategy generally provides very significant improvement, as the figures show.

In the third section ("HEAP2"), two smaller optimizations are added. Firstly, the primes that cause adjustments to MTS in one main MDSA iteration are processed in descending order of cell size. This is accomplished by building the PIS set for that iteration as a heap. (Thus, the last remaining set data structure in MDSA is heapified.) Apparently, processing the PIS primes in this order tends to decrease fragmentation in the MTS set. The second optimization is to *merge* MTS records in one simple instance. This is done by watching for adjacent cubes when a cube is added to the heap. If, while adding cube C to the MTS heap, a cube D with identical count and differing in only one variable is encountered, the two are merged into one record. For example, if cube 0X100 is to be inserted, but 1X100 is encountered while searching for an appropriate location, then the two are merged into the single cube XX100.

In each of the three main sections of Table 10.2, figures are given for the initial number of records in the MTS set, and the maximum number of records observed during the minimization. The fifth column gives the total number of minterms contained in the MTS set at the beginning of each minimization, and is therefore the same for all three experiments. Note the large initial and maximum sizes for MTS in the example of vg2. Careful construction and maintenance of MTS is especially beneficial for this PLA. The table does not include figures for when variables are

permuted; in general this causes the set sizes to vary considerably.

10.6 Adjacency Information in the Minterm Set

Unfortunately, the calculation and maintenance of adjacency information for each minterm in MTS, as done previously for each member of RMTS, is prohibitively expensive. In the map-based version, the n potential adjacencies of a given minterm are each directly probed. With the set-based version, a traversal through the OFF-set is required. Also, unlike cover counts which tend to apply to whole groups of minterms at once, adjacency counts are more likely to vary minterm to minterm. Thus, many more records would be required in the MTS tree to store this additional information because of fragmentation in the (disjoint) MTS set.

It is still possible to keep some adjacency information with the individual MTS tree records. At any point during a minimization, each MTS record has a unique "history" up to that time. All minterms contained in that cube record have been adjusted by precisely the same additions and removals of the same primes in the PCPI. Thus, there is information available of at least some of the covering cubes of those minterms. In particular, when a record is adjusted because of the addition of a prime implicant, the "don't-care" variables contained in that prime give some adjacency information.

An n-vector of proven adjacencies is kept with each record in the MTS set. The kth position in this vector indicates whether or not a "don't-care" was seen in any cube that caused adjustment of the record in the past. The total number of ones in the vector gives an estimate of how many more adjacencies the minterms in that record are likely to see. When adjustment of a record D by a prime C causes D to be replaced by the intersection of D and C, that new cube inherits all the proven

adjacencies of both cubes. The cubes in the difference $C - D$ inherit all the proven adjacencies that D originally had.

The total number of proven adjacencies in an MTS cube record provides an additional ordering relation, within that provided by the count of the record. Lower numbers of proven adjacencies have higher priority, because there is greater chance that no new implicants will be found for a minterm within that record, if one is chosen for expansion. Thus, the probability of finding a pseudo-essential cube is increased.

Inheritance of proven adjacencies from covering cubes is also used at initial construction of the MTS set, from the cubes read in the input specification. Thus, there is some adjacency information about "new minterms" as well as the "recursive" ones, which is an improvement over the map-based version, for which it was decided that adjacency calculations for all minterms is too costly.

10.7 Cell Calculation with Trees

In Chapter 9, a new cube cell generator is described, that uses the set of expanded minterms EMTS to determine the cell of a cube. Only the EMTS has to be traversed in that context, because the other minterm sets are empty in cyclic situations. However, MDSA also requires calculations of cells when some portion of the relevant minterms are in the MTS tree, in the form of cube records. In fact, unless a minimization is approaching detection of a cycle, most of the minterms in a cube's cell can be expected to be in the MTS set.

The following approach is used in the set-based version of MDSA. As discussed in Chapter 5, the new cell is "initialized" to as small a cell as can be guaranteed to be large enough to contain the actual cell. Then, a traversal of both the MTS and

PLA					MAP BASED		SET BASED				
						%	EMTS Set		MTS Set		%
name	n	m	min	calls	probes	time	visits	hits	visits	hits	time
alu1	12	8	19	1236	79303	3	0	0	427034	140949	40
alu2	10	8	68	1489	40487	2	8878	642	170454	75282	41
alu3	10	8	65	1300	28724	8	7215	375	103481	22089	35
add6	12	7	355	3918	64820	3	839	41	136809	11145	27
add7	14	8	735	22755	492390	5	41302	1859	2386346	188513	37
dist	8	5	120	1295	4763	0	12682	755	20934	3636	22
in0	15	11	107	763	98190	10	2072	121	46719	7097	9
in1	16	17	104	746	228255	7	1074	94	13755	3926	2
in2	19	10	134	1222	1390107	11	8186	815	40345	9566	1
dk48	15	17	22	44	61568	9	8	3	364	113	0
apex4	9	19	427	5400	13931	1	61652	2099	85217	13169	17
tial	14	8	575	18828	690451	6	239696	10083	3092178	560063	46
misex3	14	14	653*	75174	1204948	4	2670780	228136	2971071	645750	39
misex3c	14	14	190	330365	33682883		48750634	6101852	37307592	9822096	21
vg2	25	8	110	388	-	-	307	57	20131	1849	7
duke2	22	29	86	1032	-	-	8450	217	52652	7248	18
misex2	25	18	28	33	-	-	4	1	190	35	0
bc0	26	11	177	7952	-	-	145827	3654	3282839	323558	27
chkn	29	7	140	882	-	-	2728	280	78123	4358	10
in5	24	14	62	396	-	-	791	132	15459	1915	4
in7	26	10	54	1072	-	-	5579	355	505948	162050	22
rckl	32	7	32	367	-	-	1117	88	26865	3465	7
x1dn	27	6	110	227	-	-	37	19	14407	840	5
x9dn	27		120	325	-	-	185	55	25543	2082	5

Table 10.3: Cell Calculation Overhead

EMTS sets is required. The minterm containment search from Chapter 6 is used on the EMTS, and the cube intersection search is used with MTS. Traversals continue until the search space is exhausted, or the new cell can be proven unchanged. An obvious experiment is to switch the order in which the EMT set and MTS sets are traversed: that is, MTS before EMT. However, the order as given appears to be preferable, though the difference is slight.

Table 10.3 shows the overhead for calculating cells in an MDSA minimization, comparing the techniques from Chapter 5 using map probes at maximum Hamming distance, to the new set-based technique that traverses the MTS and EMTS trees. The number of calls to the cube calculator is the same in both experiments, and is given in the fifth column. The measure of complexity for the map-based technique

is given by the total number of "probes", or direct addressing into the truth-table, required throughout a minimization. In the case of the set-based cell calculator, complexity is measured by the number of nodes visited ("visits"), and the number of cubes found to actually intersect ("hits"), during tree traversals. The EMTS and MTS trees are counted separately. Also included are columns giving the percentage of overall minimization time required by the cell calculator.

In some examples, especially the smaller PLA's, the time taken by the set-based cell generator now dominates overall minimization time. However, in larger PLA's such as `in1`, `in2` and `dk48`, the new generator clearly outperforms the map-based one. In general, very large cells affect the map-based method adversely, and large MTS sets have the worst effect on the set-based method.

10.8 Results

Table 10.4 gives the final timings and cover sizes for the new set-based MDSA minimizer versus the map-based version. The heuristic MDSA algorithm, which "solves" cycles after isolating them by greedily choosing cubes with large Rudell weight, *without* calculating the maximal independent set lower bound function, is used in both MDSA versions. Also included are figures for *espresso*. The input for the set-based MDSA is of *espresso* type ".fr", meaning that both the ON-set and the OFF-set are provided as input. The OFF-set is provided by *espresso*, and therefore the set-based MDSA timings are adjusted by the added time requirement. Figures for the number of iterations required by both versions of MDSA are included, to show the increase due to diminished adjacency information in the choice of expanded minterms.

The times for the set-based MDSA are most significantly slower than the competition for the smaller PLA's, as can be expected. In the most extreme example,

	PLA			espresso		MDSA (map based)			MDSA (set based)		
name	n	m	min	cubes	time	iter	cubes	time	iter	cubes	time
alu1	12	8	19	19	.09	20	19	.35	25	19	3.60
alu2	10	8	68	68	.90	78	68	.19	130	68	2.60
alu3	10	8	65	66	.64	109	65	.20	122	65	1.40
add4	8	5	75	75	.53	84	[illegible]	.04	96	75	.23
add5	10	6	167	167	2.40	184	167	.22	195	167	1.30
add6	12	7	355	355	10.00	393	355	1.20	471	355	6.00
add7	14	8	735	735	44.00	800	735	3.90	925	735	61.00
apla	10	12	25	25	.20	30	25	.03	44	25	.15
dc2	8	7	39	40	.28	55	39	.05	65	39	.11
dist	8	5	120	121	2.60	193	120	.22	212	120	.75
in0	15	11	107	107	2.50	123	107	1.30	177	107	2.10
in1	16	17	104	104	6.20	124	104	2.80	172	104	1.40
in2	19	10	134	134	2.30	193	134	13.00	242	134	1.50
sym9	9	1	84	84	.67	422	88	4.20	422	94	6.20
dk48	15	17	22	22	.12	25	22	1.00	33	22	.13
mul4	8	8	121	128	4.80	382	124	.65	415	125	1.70
apex*	9	19	427	436	62.00	958	427	1.50	989	427	4.70
ex1010	10	10	267*	284	107.00	1468	274	50.00	1468	277	129.00
sev	8	10	195*	204	6.90	542	201	1.60	567	202	3.90
tial	14	8	575	581	57.00	946	576	5.60	1186	575	37.00
sym10	10	1	210	210	15.70	794	211	12.00	794	217	21.00
misex3	14	14	653*	690	111.00	3643	670	37.00	3772	635	94.00
misex3c	14	14	190	197	35.00	6812	195	1015.00	6849	195	3093.00
vg2	25	8	110	110	1.70	-	-	-	151	110	1.80
duke2	22	29	86	86	2.50	-	-	-	149	86	2.40
misex2	25	18	28	28	.21	-	-	-	31	28	.16
bc0	26	11	177	178	19.50	-	-	-	333	177	47.00
chkn	29	7	140	140	3.80	-	-	-	222	140	3.50
in5	24	14	62	62	1.00	-	-	-	96	62	1.60
in7	26	10	54	54	.68	-	-	-	96	54	13.40
rckl	32	7	32	32	.38	-	-	-	63	32	1.50
x1dn	27	6	110	110	1.10	-	-	-	146	110	2.50
x9dn	27	7	120	120	1.40	-	-		151	120	2.60

Table 10.4: Set-based MDSA Benchmark Comparisons

that of `alu1`, the set-based version spends 70% of total time in maintaining the MTS set and finding cells with it. As shown in Table 10.3, serious fragmentation of MTS occurs in this case. A similar phenomenon occurs with `alu2`. The results for `sym9` are still a disaster; the reasons why are explained in the previous chapter. The case of `misex3c` is also still deplorable.

The set-based version of MDSA is competitive with *espresso* for most of the larger PLA's. However, `bc0` and `in7` indicate that the new algorithm could use more work. In the cases of `dk48`, `in1`, and `in2`, with 15, 16, and 17 inputs respectively, the new MDSA pulls ahead of the old map-based one. Of course, the latter cannot even handle the ten large PLA's at the bottom of the table. Unfortunately, no other large benchmark PLA's are available for the experiment, because the data structures of set-based MDSA are still "hard-coded" to be limited to 32 input variables and 32 output functions.

10.9 Summary

The reduction of memory requirements for the MDSA minimizer is the subject of this chapter. The truth-table is removed by replacing its purpose in prime implicant generation by the OFF-set tree. The MTS set holds all the new minterms, and also the recursive minterms with their counts, in the form of a heap of disjoint cubes. Initialization of MTS occurs by reading the input cubes of the ON-set, then "adding" them to MTS in a certain order. Removal and adjustment of members of the ON-set is accomplished by traversals of the MTS tree. Cells of cubes are calculated by traversing the MTS tree and the EMTS tree.

The time required for a minimization by the set-based version of MDSA is dominated by the following three operations:

- the `TestCube` primitive
- maintenance of the MTS set
- determination of cube cells

The last two points are inter-related in that fragmentation of the MTS set not only increases its own maintenance time, but also the time required to traverse it for finding cells.

The simplistic solution of rewriting the `TestCube` primitive and re-using the algorithm of Chapter 9 is almost certainly not optimal. In fact, other minimizers use quite different approaches for generating prime implicants. Perhaps one of these could be adapted to generating only those implicants covering a specified minterm.

Serious fragmentation of the MTS set is so detrimental to minimization time that it should be avoided at all costs. The counts of MTS cube records are required to make good choices for minterms to expand. However, those with counts of zero or one are of far more interest than the rest. Therefore, when fragmentation starts to occur, a reasonable approach is to cease to maintain correct counts for those minterms already covered by $k \geq 2$ or more primes. Preliminary experiments, unreported here, show that this could work, as the number of MTS records with cover counts of zero or one remains relatively small, even when fragmentation occurs.

Cell determination could be made more efficient by more carefully choosing the order in which branches of the MTS and EMTS trees are taken. In fact, situations can arise in which it can be proven that no new growth to a partial cell can occur down a particular branch. For example, if it is known that a cell can be no larger than `X1XX`, and the partial cell `X10X` is already known, then no new information could possibly be found by searching the subtree corresponding to $x_1 = 0$.

Chapter 11

Conclusion

11.1 Summary

The original directed search algorithm provides an interesting approach to two-level minimization, but the algorithm is incomplete, and incapable of working with problems of meaningful size. Subsequent work improves on the original, but the same basic problems persist. The research reported here accompanies a working minimizer incorporating a modified directed search approach, with many of the original shortcomings of DSA removed. The program, MDSA, is competitive with the best existing heuristic minimizers with regards to speed. But MDSA often gives better (smaller) covers, because they are provably minimum for a wider class of functions.

Modified directed search is an iterative process, in which the number of iterations i corresponds to the number of minterms requiring expansion. The lower bound for i is the number of essential and pseudo-essential cubes eventually found for the cover, plus the number of minterms involved in true cycles. The upper bound for i is the number of minterms in the ON-set of the function. In practice the number of iterations

required is much closer to the lower bound. This is achievable partially because of improved rules and heuristics for minterm selection, but also because minterms found to be dominating are removed as early as possible.

Throughout an MDSA minimization, dominance relations among all generated prime implicants and all expanded minterms are kept current. This in turn is accomplished through comprehensive cell classification and calculation techniques. Cells are used for representing both intersection sets, and the relevant parts of prime implicants. As minimization proceeds, the cells of prime implicants shrink, and the cells of intersection sets expand. These changes in dimension trigger searches for dominated and dominating elements.

The MDSA minimizer is primarily an exact one: methods for both finding and solving the cycles of an input PLA are provided. The main strength of the minimizer is its ability to efficiently identify the cyclic cores of functions, where these cores are in a fully reduced state. The heuristic version of MDSA is identical to the exact version, in the phase of minimization where cyclic cores are determined, and isolated from the rest of the input problem. The heuristic version differs at this point, suppressing low-bound calculations and limiting branching depth to zero. The results are still often better than *espresso*, although MDSA is apparently incapable of consistently good results for PLA's of symmetric functions, like `sym9` and `sym10`. The exact version of MDSA uses a branch-and-bound algorithm that requires only linear storage.

Many primitive directed search operations require searches on various sets. Trees of cubes and minterms are defined and used for improving the performance of searches. These sets are ordered according to the value of variables in cubes and minterms, and the time required to remove or insert a member is bounded above by the number of input variables n. Cube heaps are special cases of cube trees, and facilitate improved selections and heuristic calculations. Heaps of both minterms and prime implicants

are of paramount importance in the exact cycle solution algorithms, where efficient selection and traversal are required in order to avoid construction of graphs.

The modified directed search algorithm works best with sparse functions. This characteristic is one shared with many other minimizers, and originates from the lesser number of minterms and prime implicants that need be considered in such cases. Some protection from generating too many redundant prime implicants is available by redirecting the minterm search. This particular aspect of MDSA, called *redirected search minimization* or RSM, involves the storage of some small number of potential cycles. Unfortunately, research in RSM is too preliminary to be included here.

A truth-table representation is used for problems with fourteen or fewer input variables. A set representation employing heaps of disjoint cubes is used for larger problems. Research in this area is preliminary, and minimization times are somewhat slower for MDSA, compared to *espresso*. The increased time comes largely from the prime implicant generator, which is designed for efficient operation with tables, not sets. There is also the problem of cube set fragmentation, which not only inhibits the process of maintaining correct cover counts for minterms, but also adversely affects the determination of cubes' cells.

11.2 Further Work

A practical minimizer must be able to deal with *all* types of input problems, including dense functions, which are more likely to contain large cyclic cores. Not enough experimentation has been done with heuristic approaches, for the situation where cyclic cores of problems are too large. Surprisingly, not one single benchmark PLA used for experiments in this research has a cyclic core that is too large to generate.

Unfortunately, it is not currently possible to carry out experiments with very large examples, because the MDSA program is limited to PLA's with at most 32 inputs and 32 outputs. The fundamental cube record data structures and primitive operations must be rewritten to provide this flexibility. At that time, effective research into the set-based version of MDSA could continue.

The generation of prime implicants in the set-based version of MDSA dominates the time required to minimize some of the larger examples. A new generator is needed: perhaps one that shrinks cubes from the universe, rather than growing them from a minterm. Also, it is desirable to incorporate into the generator a method for avoiding the generation of cubes that are immediately found to be dominated. Intuitively, this is possible by integrating the initial determination of cells into the generator.

Though prime generation is limited to primes covering selected minterms, there can still be too many to handle efficiently. Implicit representations for prime implicant sets have recently been proposed [CM92], and even used in some minimization algorithms [SBM93]. These are quite similar to the intersection cubes and cells maintained by MDSA for expanded minterms. It is possible that such a representation could be used for prime implicant sets in MDSA, although it is unclear how minterm cover counts and dominance relations among cubes could be kept current.

The phenomenon exhibited by the MDSA minimization of the PLA `misex3c` has not been adequately addressed. This PLA contains two medium-sized cyclic cores, but the potential cycle leading up to one of the true cycles reaches a much larger size than that of the core. The prototype redirected search minimizer works for `misex3c`, but it is unclear whether that approach would always be effective. RSM is only partially functional with the map-based version of MDSA, and not functional at all with the set-based version.

Further research into alternative cube-based and non-cube-based minterm set rep-

resentations is needed. For example, the minterm sets employed in Chapter 10 could be permitted to contain some redundancy, thereby potentially reducing the size of the representation, but also potentially increasing maintenance time. Also, no experimentation with variable ordering on set-based minterm sets has been performed. Research into this subject could limit the fragmentation phenomenon. Current BDD-based work may well be useful in this regard.

Bibliography

[BBH+88] K. Bartlett, R. Brayton, G. Hachtel, R. Jacoby, C. Morrison, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang. Multi-level logic minimization using implicit don't-cares. *IEEE Transactions on Computer-Aided Design*, 7(6):723–740, June 1988.

[BCH+82] R.K. Brayton, J.D. Cohen, G.D. Hachtel, B.M. Tragger, and D.Y.Y. Yun. Fast recursive boolean function manipulation. In *Proc. Int. Symp. Circuits and Systems*, pages 58–62, May 1982.

[BHMSV84] R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A. Sangiovanni-Vincentelli. Espresso-ii: A new logic minimizer for programmable logic arrays. In *Proc. Custom Integrated Circuits Conf.*, pages 370–376, May 1984.

[BHMSV85] R.K. Brayton, G.D. Hachtel, C.T. McMullen, and A.L. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1985.

[BMS92] R.K. Brayton, P.C. McGeer, and J. Sanghavi. A new exact and heuristic minimizer for two-level logic synthesis. In *Proceedings of the Japanese Conference on Logic Design*, pages 7–15, 1992.

[BMSSV93] R.K. Brayton, P.C. McGeer, J.V. Sanghavi, and A.L. Sangiovanni-Vincentelli. A new exact minimizer for two-level logic synthesis. In T. Sasao, editor, *Logic Synthesis and Optimization*, pages 1–31. Kluwer Academic Publishers, 1993.

[Cha87] A.H. Chan. Using decision trees to derive the complement of a binary function with multiple-valued inputs. *IEEE Transactions on Computers*, C-36(2):212–214, February 1987.

[CM92] O. Coudert and J.C. Madre. A new implicit graph based prime and essential prime computation method. In *Proceedings of the Japanese Conference on Logic Design*, pages 124–131, 1992.

[CM93] O. Coudert and J.C. Madre. A new graph based prime computation technique. In T. Sasao, editor, *Logic Synthesis and Optimization*, pages 33–57. Kluwer Academic Publishers, 1993.

[DAR85] M.R. Dagenais, V.K. Agarwal, and N.C. Rumin. The mcboole logic minimizer. In *Proc. 22nd Design Automation Conference*, page 667, June 1985.

[DAR86] M.R. Dagenais, V.K. Agarwal, and N.C. Rumin. Mcboole: A new procedure for exact logic minimization. *IEEE Transactions on Computer-Aided Design*, CAD-5(1):229–238, Jan 1986.

[Die88] D.L. Dietmeyer. *Logic Design of Digital Systems*. Allyn and Bacon, Inc., 3rd edition, 1988.

[DM88] G.W. Dueck and D.M. Miller. Directed search minimization of multiple-valued functions. In *Proceedings of the 18th International Symposium on Multiple-Valued Logic*, pages 218–225, May 1988.

[Due88] G.W. Dueck. *Algorithms for the Minimization of Binary and Multiple-Valued Logic Functions.* PhD thesis, University of Manitoba, 1988.

[GB89] B. Gurunath and N.N. Biswas. An algorithm for multiple output minimization. *IEEE Transactions on Computer-Aided Design*, 8(9):1007–1013, September 1989.

[Giv70] D.D. Givone. *Introduction to Switching Circuit Theory.* McGraw-Hill, 1970.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company, 1979.

[HCO74] S.J. Hong, R.G. Cain, and D.L. Ostapko. Mini: A heuristic approach for logic minimization. *IBM Journal of Research and Development*, pages 443–458, September 1974.

[HO72] S.J. Hong and D.L. Ostapko. On complementation of boolean functions. *IEEE Transactions on Computers*, C-21:1022, 1972.

[Knu73] D.E. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

[LCM92] B. Lin, O. Coudert, and J.C. Madre. Symbolic prime generation for multiple-valued functions. In *Proc. 29th Design Automation Conference*, pages 40–44, June 1992.

[Man91] M.M. Mano. *Digital Design.* Prentice Hall, 2nd edition, 1991.

[MB90] P.C. McGeer and R.K. Brayton. The observability don't-care set and its approximations. In *IEEE International Conference on Computer Design*, 1990.

[McC56] E.J. McCluskey. Minimization of boolean functions. *Bell Systems Technical Journal*, 35:1417–1444, November 1956.

[McC86] E.J. McCluskey. *Logic Design Principles.* Prentice-Hall, 1986.

[McK74] M.H. McKinney. *A Directed Search Algorithm for the Canonical Minimization of Switching Functions.* PhD thesis, Texas A&M University, August 1974.

[PR88] S.R. Perkins and T. Rhyne. An algorithm for identifying and selecting the prime implicants of a multiple-output boolean function. *IEEE Transactions on Computer-Aided Design*, 7(11):1215–1218, November 1988.

[RNMP77] V.T. Rhyne, P.S. Noe, M.H. McKinney, and U.W. Pooch. A new technique for the fast minimization of switching functions. *IEEE Transactions on Computers*, C-26(8):757–764, August 1977.

[Rot85] C.H. Roth. *Fundamentals of Logic Design.* West Publishing Company, 3rd edition, 1985.

[RSV85] R. Rudell and A. Sangiovanni-Vincentelli. Espresso-mv: Algorithms for multi-valued logic minimization. In *Proc. CICC85*, pages 230–234, May 1985.

[RSV86] R. Rudell and A. Sangiovanni-Vincentelli. Exact minimization of multiple-valued functions for pla optimization. In *Proc. IEEE Int. Conf. on CAD*, pages 352–355, Nov 1986.

[RSV87] R.L. Rudell and A. Sangiovanni-Vincentelli. Multiple-valued minimization for pla optimization. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):727–750, September 1987.

[Rud86] R. Rudell. Multiple-valued logic minimization for pla synthesis. Technical Report M86/65, University of California at Berkeley, 1986.

[Sas78] T. Sasao. An application of multiple-valued logic to a design of programmable logic arrays. In *Proc. 8th International Symposium on Multiple-Valued Logic*, pages 65–72, May 1978.

[Sas81] T. Sasao. Multiple-valued decomposition of generalized boolean functions and the complexity of programmable logic arrays. *IEEE Transactions on Computers*, C-30(9):655–643, September 1981.

[Sas85] T. Sasao. An algorithm to derive the complement of a binary function with multiple-valued inputs. *IEEE Transactions on Computers*, C-34(2):131–140, February 1985.

[SBM93] G.M. Swamy, R.K. Brayton, and P. McGeer. A fully implicit quine-mccluskey procedure using bdd's. In *International Workshop on Logic Synthesis: Workshop Notes*, pages 2b1–2b16, May 1993.

[Ser84a] M. Serra. Directed search minimization of multiple-output networks. Master's thesis, University of Victoria, Apr 1984.

[Ser84b] M. Serra. Directed search minimization of multiple-output networks for pla's. In *Technical Digest of the 1984 Canadian Conference on Very Large Scale Integration*, pages 5.147–5.151, Oct 1984.

Appendix A

Glossary

adjacent cubes or minterms differing in only one variable.

affected cube a cube (always a prime implicant) whose cell contains fewer minterms, because some have changed to "don't-care" vertices.

affected minterm a minterm whose covering set of prime implicants is reduced in number.

base minterm a minterm whose covering set of prime implicants is sought.

Boolean function a function over a number of two-valued variables.

branch-and-bound a method for spanning a tree-shaped solution space.

candidate cube a cube whose cell intersects with a selected cell, thereby becoming potentially affected.

cell the smallest subcube of a cube containing all that cube's minterms.

complete column a column (corresponding to a minterm) in a partial prime implicant table, for which all intersecting rows (prime implicants) are known to be in existence.

completely specified describing a discrete function in which all output points are defined 1 or 0 (no "don't-care" vertices).

contracted cell an affected cell which contracts due to the removal of some minterms.

cover a set of implicants where each minterm of a function is contained in at least one implicant.

cube a subspace of a function corresponding to an assignment of each variable to a subset of its range of values.

cube cell see cell.

cycle a condition existing when cubes belonging to a minimum cover cannot be deterministically found.

cyclic core that part of a function for which a solution cannot be found using deterministic methods. Corresponds to a fully-reduced non-empty prime implicant table.

cyclic prime implicant set the set of prime implicants involved in a cycle.

decisive describing a discrete function in which each output point must assume one of two values.

density the property of a decisive function describing the relative frequency of minterms, compared to zero vertices.

directed search algorithm a minterm-based two-level minimization algorithm.

dominated cube a cube found to be redundant because all its covered minterms are covered by some other cube.

dominating minterm a minterm found to be redundant because it is covered by a superset of the cubes that cover some other minterm.

domination a relation used to establish the redundancy of certain cubes and minterms.

don't-care a value in the output space of a decisive function, meaning that the vertex can assume any real value.

don't-care set the set of all "don't-care" vertices of a given function.

don't-care variable an unspecified variable.

enlarged minterm a minterm whose prime implicant set intersection cube has grown, because some covering prime implicants have been removed.

essential minterm a minterm covered by only one prime implicant.

essential prime implicant a prime implicant that covers a minterm not covered by any other prime implicant.

espresso an extremely popular and effective heuristic two-level minimizer.

espresso-exact the exact version of the espresso minimizer.

exact minimization in which the extracted cube cover for a function is known to be minimum.

frontier for a given base minterm, the boundary in the prime implicant set search space occupied by the potential prime implicant set.

fully reduced cycle or fully reduced prime implicant table: one which cannot be simplified through identification of essential elements, domination among rows, or domination among columns.

fully specified see completely specified.

Hamming distance the number of differing bits in two bit-strings of equal length over (0,1).

implicant a cube that does not cover any zero vertices of a function.

incompletely specified describing a discrete function in which one or more output points are unspecified.

induced pseudo-essential prime implicant an implicant that becomes pseudo-essential only after some of its contained minterms have been removed.

intersection cell the intersection of the cells of all mutually irredundant prime cubes covering a given minterm.

intersection cube the intersection of all mutually irredundant prime cubes covering a given minterm.

Karnaugh map a tabular representation for a Boolean function, indicating adjacencies by proximity.

literal a variable assuming either its true or complemented form.

map a tabular representation of a discrete function.

minimization two-level minimization, in which a cube cover of as few cubes as possible is sought for some discrete function.

minimizer a computer program for performing minimization.

minterm a vertex assuming the value 1, in a decisive function.

minterm cell see cell.

minterm set (MTS) set of as-yet-uncovered minterms, organized in a disjoint cube heap.

new minterm (NMT) a true minterm to be covered, for which no covering prime implicant has yet been discovered.

new minterm set (NMTS) the set of all new minterms.

OFF-set the set of all zero vertices of a given function.

ON-set the set of all minterms of a given function.

partial prime implicant table a proper subset of a complete prime implicant table, where not all relations between cubes and minterms have yet been found.

potentially cyclic prime implicant set (PCPI) generated prime implicants that are not known to be either dominated, essential, or pseudo-essential.

primary minterm dominance a method for removing a subset of all redundant minterms, using the intersection of all cubes comprising a prime implicant set, at the time of that set's generation.

prime a prime implicant.

prime cover a set of cubes covering a given function, all of which are prime.

prime implicant an implicant of maximal size.

prime implicant table a tabular representation of the covering problem.

programmable logic array a structured hardware implementation of a two-level cube cover of a multiple-output binary function.

pseudo-cycle an apparently cyclic situation which actually contains no irreducible cyclic core.

pseudo-essential minterm a minterm found to be covered by only one prime implicant after dominated prime implicants are removed from consideration.

pseudo-essential prime implicant the undominated prime implicant that covers a pseudo-essential minterm.

quasi-exact describing a minimization algorithm that is purported to be exact, but that does not include any minterm dominance.

reachable prime implicant in the HYPER algorithm, an undiscovered prime that is a supercube of at least one member of the potential prime implicant set.

recursive minterm (RMT) a minterm covered by at least one cube in the potential prime implicant set.

recursive minterms set (RMTS) those minterms covered by one or more cubes in the potential prime implicant set.

required adjacency direction minterm adjacencies used to aid the DSA in depth-first generation of prime implicants, and also used to aid in the heuristic selection of minterms.

secondary minterm domination a method for removing all redundant minterms in a function by ensuring that no dominating ones could possibly exist.

sparse function a function with relatively few minterms, compared to the number of zero vertices.

specified variable in a product term (or cube), a variable that must assume either the value 0, or 1.

sum-of-products an algebraic expression corresponding to a conjunction of product terms, or cubes.

truth table a tabular representation of a discrete function.

unspecified variable a variable whose value is irrelevant to a given product term. Also called a "don't-care" variable.

vertex a point in a discrete function space.

weight, cube the number of minterms covered by a cube.

weight, minterm the number of cubes covering a minterm.

Appendix B

Abbreviations

BDD binary decision diagram.

BFS breadth first search.

CCT cyclic cube tree.

CMT cyclic minterm tree.

CPC complete potential cycle.

CPI cyclic prime implicant set.

DC "don't-care".

DCC "don't-care" constraint.

DC-set "don't-care" set.

DFS depth first search.

DOM set of dominated cubes.

DSA McKinney's directed search algorithm.

DSA-MV Dueck's multi-valued directed search algorithm.

DSA2 Serra's directed search algorithm.

EDSA Perkin's extended directed search algorithm.

EMT expanded minterm.

EMTS expanded minterm set.

FAD failed adjacency.

HYPER Mike's prime implicant generation algorithm.

MDSA Mike's (modified) directed search algorithm.

MTS minterm set.

NMT new minterm.

NMTS new minterm set.

OBDD ordered binary decision diagram.

OCI one can imagine.

Q-M Quine-McCluskey algorithm.

PAD potential adjacency.

PC potential cycle

PCPI potentially cyclic prime implicant set.

PI prime implicant.

PIG prime implicant generator.

PIS prime implicant set.

PIT prime implicant table.

PLA programmable logic array.

PPI potential prime implicant.

PPIS potential prime implicant set.

PPIT partial prime implicant table.

RAD required adjacency direction.

RMT recursive minterm.

RMTS recursive minterm set.

SOP sum of products.

VLSI very large scale integration.