

136
35

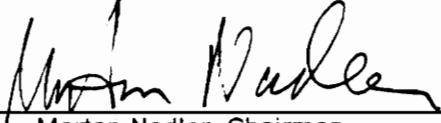
An Algorithm for Multi-output Boolean Logic Minimization

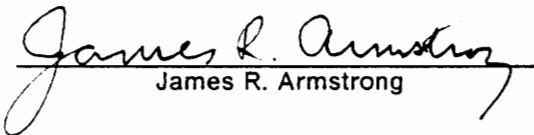
by

Rohit H. Vora

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of
Master of Science
in
Electrical Engineering

APPROVED:


Morton Nadler, Chairman


James R. Armstrong


Festus G. Gray

June, 1987

Blacksburg, Virginia

21

LD
5655
V855
1977
V672
0.2

An Algorithm for Multi-output Boolean Logic Minimization

by

Rohit H. Vora

Morton Nadler, Chairman

Electrical Engineering

(ABSTRACT)

A new algorithm is presented for a guaranteed absolute minimal solution to the problem of Boolean Logic Minimization in its most generalized form of multi-output function with arbitrary cost criterion.

The proposed algorithm is shown to be tighter than the Quine-McCluskey method in its ability to eliminate redundant prime implicants, making it possible to simplify the cyclic tables. In its final form, the proposed algorithm is truly concurrent in generation of prime implicants and construction of minimal forms. A convenient and efficient technique is used for identifying existing prime implicants. Branch-and-bound method is employed to restrict the search tree to a cost cut-off value depending on the definition of cost function specified.

A most generalized statement of the algorithm is formulated for manual as well as computer implementation and its application is illustrated with an example. A program written in Pascal, for classical diode-gate cost function as well as PLA-area cost function, is developed and tested for an efficient computer implementation. Finally, various advantages of the proposed approach are pointed out by comparing it with the classical approach of Quine-McCluskey method.

Acknowledgements

I am deeply indebted to my advisor Dr. Nadler for his unvarying support throughout my graduate studies. This work would not have been possible without his active interest at all stages of this research, and above all, his intuitive feeling about this research problem which kept me going in the right direction.

I would also like to thank Dr. Armstrong and Dr. Gray for serving in my committee and taking time to carefully review this work.

I am also grateful to all my friends and colleagues who, directly or indirectly, helped in preparation of my thesis. Finally, I would like to thank my parents for making my studies in the United States possible.

Table of Contents

Minimization Problem - A Survey	1
1.1 Introduction to the Problem	1
1.2 Overview of Different Approaches	3
1.3 A Survey of Relevant Literature	5
1.3.1 Classical Minimization Methods	5
1.3.2 Prime Implicant Generation Techniques	6
1.3.3 Minimization Algorithms Using Concurrency	7
1.3.4 Application to PLA Minimization	8
1.3.5 Complexity of the Boolean Minimization Problem	9
Preliminary Concepts	11
2.1 Representation of a Boolean Function	11
2.2 K-cell Concept	13
2.2.1 Basic Definitions	13
2.2.2 The K-cell	18
2.3 Some Important Results	25
2.4 Generation of k-cells	27

Proposed Algorithm	31
3.1 Problem Definition	31
3.2 Algorithm Statement	32
3.3 Terminology	34
3.4 Justification	38
3.4.1 Rule 1: Selection Rule	38
3.4.2 Rule 2: Elimination Rule	44
3.4.3 Rule 3: Branching Rule	46
3.5 Illustration	47
3.5.1 Gate-input count cost function	47
Computer Implementation	65
4.1 Computer Algorithm	65
4.1.1 K-cell Generation	66
4.1.2 K-cell Selection	71
4.2 Program Specifications	74
4.2.1 Input-Output Requirements	74
4.2.2 Program Features	75
4.3 Program Logic	77
4.3.1 K-cell Generation	77
4.3.2 K-cell Selection	78
4.4 Revised Algorithm	87
4.5 Sample Run	96
4.5.1 Sample Run A	97
4.5.2 Sample Run B	100
Discussions and Results	111
5.1 Comparison with Quine-McCluskey's Algorithm	111

5.1.1 K-cell Generation	111
5.1.2 K-cell Selection	114
5.2 Solutions of Selected Problems	119
5.2.1 Problem 1	119
5.2.2 Problem 2	121
5.2.3 Problem 3	122
5.2.4 Problem 4	123
5.2.5 Problem 5	125
5.2.6 Problem 6	126
5.2.7 Problem 7	127
5.2.8 Problem 8	129
5.3 Scope for Further Work	130
Bibliography	131
Program Listing	135
Vita	165

List of Illustrations

Figure 1. Marquand chart and Karnaugh map representations of a Boolean function. . . 12

Figure 2. Regular pattern on a Marquand chart to locate all neighboring states. 17

Figure 3. An example illustrating the k-cell concept. 19

Figure 4. Application of the proposed algorithm for an example using the logical matrix. 48

Figure 5. An example of the possibility of non-maximal k-cells being generated. 69

Figure 6. Flowcharts showing program structure and major procedures. 81

Figure 7. An example illustrating the improvement in efficiency of the revised algorithm. 90

Figure 8. Flowchart of the revised algorithm combining k-cell generation and selection. 95

Figure 9. An example to demonstrate the improvement over the QM method. 117

List of Tables

Table 1. States of boolean functions and their indices.	16
Table 2. Generic k-cells and their indices.	21
Table 3. Definite k-cell and their indices.	23
Table 4. Relationship among state, generic k-cell and definite k-cell indices.	24
Table 5. Sequence of generic k-cells generation.	30
Table 6. QM's cyclic table for an example to demonstrate the looseness of QM rules.	115

Chapter 1

Minimization Problem - A Survey

1.1 Introduction to the Problem

Since the advent of the digital age, the field of digital logic design has been of a considerable interest. A substantial amount of theoretical research has gone into development of the basic principles of switching theory and design methodologies for combinational as well as sequential logic. In fact, the field of switching theory may be considered as a well established field in the sense that the fundamental theories behind the digital switching concept are well understood by now.

Logic design techniques have been the target of much of the application based research in recent years. The rapid advancements in digital design methodologies owe a great deal to the simultaneous development of boolean algebra. The digital design problems can be transformed into boolean algebra problems and treated as such. Thus the purely theoretical

problem of boolean minimization found a very important application in digital circuit minimization.

Traditionally, the digital logic has been implemented using discrete or integrated diode-gate circuits. The philosophy of minimization concept was based entirely on the implementation of combinational logic with a minimum of hardware. But since the introduction of VLSI and Programmable Logic Arrays (PLA's), there is a renewed interest in the general problem of boolean minimization with a broader outlook. The cost minimization is device dependent and so the definition of cost-factor may be different for different devices. For instance, in a VLSI layout for a combinational logic circuit, the minimization effort would be for the area occupied by the layout. For a PLA, on the other hand, the cost factor may be one of many constraints on the particular PLA, such as number of output-functions, number of minterms or even size of minterms which can be accommodated. For a VLSI implementation of a PLA, it is the area as determined by number of minterms, number of input variables as well as number of output functions. The variations in the definition of cost function may result in a situation where a minimal solution for a particular device may be far from minimal in another device; and hence the need exists for a more generalized minimization algorithm independent of the way cost is defined.

Algorithms which produce minimal or near minimal forms of a given boolean system have been known for some time. The quest is for algorithms which are more efficient or which guarantee an absolutely minimal solution [16]. In the class of problems such as information coding and circuit synthesis, not only the proof of existence of a finite algorithm is required, but it is required actually to construct it [58]. Even when an algorithm guarantees, at least theoretically, a solution after a finite number of steps, in practice, the solution is often impossible because of its cumbersome nature or an impractical amount of time required for termination of the algorithm for a given problem. The general problem of boolean minimization is NP-hard, although the non-existence of a polynomial-time algorithm has not been proved [30].

We shall review some of the approaches to this problem of logic minimization in the next section. When we compare the efficiency of these different algorithms, several factors such as the size of the problem and particular configuration of the given problem affect the outcome. There has been a considerable effort to specify the best lower bound and upper bound for the complexity growth depending on the size of the problem [15,10,21].

1.2 Overview of Different Approaches

A number of methods have been suggested in the literature for switching function minimization. The following list [43], broadly categorizes the major approaches used for the general problem of boolean logic minimization.

1. *Map methods:*

This is essentially a manual method using various grids to represent the switching functions, such as Karnaugh maps [23] and Marquand charts [27]. Karnaugh maps are known to be difficult for number of variables greater than six, while Marquand charts have been successfully used for problems of twelve variables [50,51,35].

2. *Consensus methods:*

This was originally introduced by Quine [40,41,42] and was later generalized by Tison [55]. This method is based on finding a new prime implicant (PI) by consensus operation among two or more PI's. The same principle is used for finding the minimal sums of a boolean function.

3. *Conjunctive/disjunctive manipulations:*

These are prime implicant generation techniques based on the laws of conjunctive/disjunctive logic, originated by Nelson [37] and extended by Slagle [46]. Variations of this techniques are treated by Das [13], Bredeson [5] and others. This method was recently extended by use of factoring method presented by Hulme and Worrell [19].

4. *Topological methods:*

First introduced by Roth [44], this approach uses n-space representation of n-variable boolean functions for algorithmic operations.

5. *Quine-McCluskey tabular method:*

This is the most widely accepted method for computer implementation due to its step by step procedural nature. In addition to McCluskey [28,39,29], it is discussed in depth by Biswas [3] and many others.

6. *Cellular containment method:*

This is basically a PI generation method. Described by Nagle [36], this technique identifies higher order PI's first, as opposed to Quine-McCluskey method, and then identifies lower order PI's which are not contained in any higher order PI's.

7. *Concurrent methods of generation and selection of PI's:*

The concept was proposed by Svoboda [50] for a manual implementation. Sureshchander [49] used concurrency to the extent of identifying all essential PI's by consensus operation, suitable for computer implementation. The concurrency approach has the major advantage of construction of a minimal solution simultaneously with PI generation. Rhyne [43] also gives an algorithm based on directed-search which employs concurrency to some extent.

1.3 A Survey of Relevant Literature

1.3.1 Classical Minimization Methods

The famous Quine-McCluskey algorithm has come to be known as the classical approach to the boolean minimization problem. The basic method was suggested by Quine and then it was later adapted by McCluskey for a more complete presentation. This method consists of two basic stages. The first stage constructs all prime-implicants (possible coverages) for a given function by extensive comparison of all non-zero fundamental minterms in the function. This requires a careful ordering of the minterms before the comparison process is started. Each comparison consists of checking if two given minterms can be replaced in the expression of that function by a single term, consistent with the laws of boolean algebra.

For the case of multi-output boolean function, this process is repeated for all the individual functions as well as the auxiliary functions. Auxiliary functions are the intersection functions of all possible combinations [29]. It is clear that this kind of computation can get very tedious, especially for a multi-output function with large number of output functions.

The second stage consists of extracting the minimal prime implicants based on the concept of dominance and cost function of each prime implicant. This again requires construction of a table or a matrix, on which the rules of dominance are based. Handling of such a table efficiently, especially the cyclic tables, has been known to be a difficult proposition. The listing of all prime implicants at each stage of its construction, as well as the storage required for the dominance table, especially in case of branching for a cyclic table, demands large memory availability.

In search for a more efficient algorithm, several authors have suggested some improved algorithms based on the Quine-McCluskey approach. Vandling [56] used an extension of notation and technique for minimization of single-output switching networks to multi-output switching networks composed of unilateral devices such as diodes. Bartee [2] used essentially the same technique as the classical method, except that it is more suitably adapted for computer implementation. Miller [33] also gave a transformation method for multi-output minimization which transforms the problem into a single-output problem for application of QM technique. Luccio [26] used a technique of column combination and extension of dominance rules for successive reduction of PI tables. Bubenik [8] introduced weighting of PI's based on its cost and size for determination of irredundant set of PI's. Curtis [11] introduced another tabular method called adjacency table method for deriving minimal sums, where a single table records adjacency properties of all minterms. Ishikawa [22] introduced an algorithm which uses a more efficient list structure for the storage of prime-implicant tables. It also makes use of some convenient branching heuristics. More recently, Cutler and Muroga [12] made use of the concept of generalized function and its decomposition to solve the cyclic tables by applying branch-and-bound technique to solve a Petrick function [38] obtained from the cyclic table.

1.3.2 Prime Implicant Generation Techniques

Most of the techniques available for switching function minimization consist of two separate modules of PI generation and PI selection. There are several techniques available to generate all possible PI's, including the one suggested by Quine-McCluskey. Morreale [34] makes use of partitioned list algorithm for PI determination as an improvement over Quine's exhaustive comparison method. Svoboda [53,54] has a procedure for ordering of PI's using triadic numeric representation of boolean functions. Slagle, Chang and Lee [48] use conjunctive/disjunctive forms of the boolean function in an algorithm based on frequency or-

dering. Other techniques include those of Das [13], Bredeson [5] and Hulme [19] based on symbolic logic manipulation and that of Hwa [20].

1.3.3 Minimization Algorithms Using Concurrency

Breitbart and Vairavan [6] showed that the algorithms that produce an irredundant sum of products for a switching function by first generating all PI's and then extracting the minimal PI's can be substantially wasteful. They specifically show that the ratio of the number of PI's to the number of useful PI's (which are eventually selected in a minimal form) grows exponentially in the number of variables.

Svoboda [50] came up with a different approach to the solution of single-output boolean minimization, which does not require listing and tabularization of all the prime implicants. He re-introduced the Marquand chart and, among others, successfully demonstrated its effectiveness for manual computation for problems of large number of variables [51,24,35,31]. This procedure involves identifying PI's for a minterm to be covered and one of these PI's is selected immediately in the minimal form, if possible. The method introduced the concurrent construction of the minimal forms and the generation of PI's. DeVries and Svoboda [14] also introduced the concept of boolean functions of variables x_j transformed into a single (mosaic) function $f(x,y)$ by using additional variables y_i , related to the structure of the resulting network. The mosaic function [52] $f(x,y)$, when minimized for a cost function dependent on x_i & y_i , models the multi-output minimal network with the smallest number of gates requiring minimum number of inputs.

The concept of concurrent process of generation and selection of PI's was further advanced by Sureshchander [49] by treating the problem as a whole instead of decomposing it into two separate parts. Rhyne, Noe, McKinney and Pooch [43] introduced directed-search algorithm,

which also looks for concurrency. Caruso [9] takes advantage of the numerical equivalent of completely specified function vertices as pointers to generate each graph with the help of a pointer table.

More recently, heuristic approaches have been successfully used to make minimization process faster. The most notable among these are MINI [18], PRESTO [7], MIN370 [46] and EXPRESSO [4]. It is interesting to note that all of these heuristic algorithms employ the technique based on the so called procedure of *expansion* and *elimination* of implicants.

1.3.4 Application to PLA Minimization

More recently, a number of algorithms have been presented with PLA implementation in mind. Kobylarz and Al-najjar [25] examined the cost function for PLA's and accordingly suggested some modifications of Quine-McCluskey's algorithm. Arevalo and Bredeson [1] as well as Roth [45] present algorithms giving near-minimal solutions for a more efficient computer implementation. Young and Muroga [57] have made use of an entirely new approach using theory of symmetric minimal covering problem. It is claimed that this approach can be applied not only to any symmetric (totally or partially) single or multi-output functions of any complexity, but also to any switching functions which are not symmetric, if the corresponding minimal covering problems have symmetric permutations. Sasao [47] introduced another method for input variable assignment and output phase optimization of the PLA's, with or without a decoder.

1.3.5 Complexity of the Boolean Minimization Problem

With various algorithms available for solutions to the boolean minimization problem, a considerable effort has been made to arrive at a generalized criterion for comparing different algorithms. Although it is known that certain types of algorithms can be compared for complexity growth depending on a common factor, it has been conjectured that different *types* of algorithms will not be comparable [58]. For instance, the efficiency of any boolean minimization algorithm will depend on a number of factors such as number PI's and essential PI's generated for a boolean function of given number of variables and given number of minterms. It will also depend on the number of comparisons involved in selection and elimination of PI's, memory requirements, additional computations such as ordering of PI's and tabularization, as done in several methods.

"To help choose among the various possible variants of these methods, in relation to the values of parameters of a given problem," Mileto and Putzolo [32] have obtained formulas which give average values for the quantities of statistical interest such as total number of implicants, number of prime implicants and number of essential prime implicants depending on number of variables and number of minterms. This also gives us an indication of the average computation time required for a problem with given parameters.

Several lower and upper bounds have been suggested for the maximum number of PI's that can be generated for any problems as functions of different parameters. If g_{\max} be the maximum number of PI's that can be generated, Chandra and Markowsky [10] gave lower and upper bounds as $3^{k/3} \leq g(k)_{\max} \leq 2^k$, where k = number of conjuncts for a function in its disjunctive normal form. As a function of n , the number of variables, Dunham and Fridshal [26] gave the lower bound of,

$$g(n)_{\max} \geq \left(\lfloor n/3 \rfloor \lfloor (n+1)/3 \rfloor \lfloor (n+2)/3 \rfloor \right)$$

while Harrison [16] observed an upper bound of $g(n)_{\max} \leq 3^n - 2^n$. Chandra [10] improved on the upper bound giving,

$$g(n)_{\max} \leq \binom{n}{\lfloor (n+1)/3 \rfloor} 2^{\lfloor (2n+1)/3 \rfloor}$$

Igarashi [21] showed a better lower bound as a recursive function of the variable n . More recently, McMullen and Shearer [30] presented arguments to give an upper bound of $g(m)_{\max} = 2^m - 1$, where m is number of product terms in the expression of the boolean function.

The proposed algorithm presented in the following pages is based on Svoboda's approach to the single-output minimization. It is an effort to provide a rigorous and most generalized solution to the problem of multi-output minimization with variable definitions of cost functions. Its manual as well as computer implementations are discussed. The basic approach of PI identification is similar to the *cellular containment* approach mentioned above, while the selection of PI is based on the *concurrency* concept.

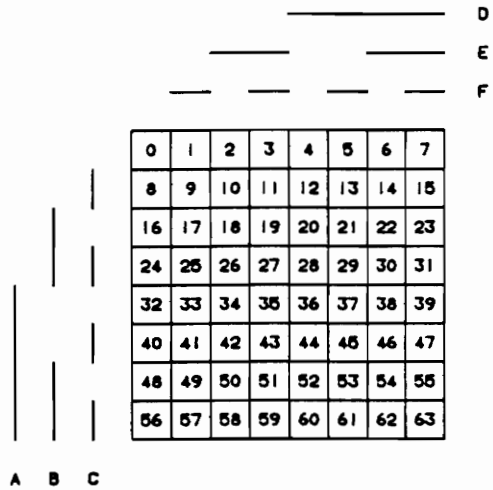
Chapter 2

Preliminary Concepts

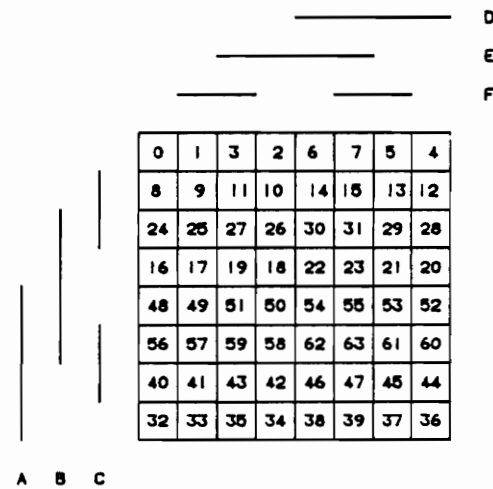
2.1 *Representation of a Boolean Function*

Various techniques have been employed for representing a boolean function to make visualization of inter-relationships among the minterms easier. These include Marquand chart and Karnaugh map, both of which are shown in Figure 1 along with the decimal index representation (defined below) for each square in the grid.

Although Karnaugh maps have been proved to be a powerful tool for boolean functions of up to four variables, their main advantage of easy visualization of formation of minterms by grouping is lost for functions of more than four variables. The classical method of Marquand chart, however, does not depend on number of variables, which allows it to retain the same form even for large number of variables. Another important feature of Marquand chart, especially for the proposed minimization algorithm, is the regularity in identifying a minterm on the chart for any number of variables. The key to this property is, as can be seen in Figure 1, the



(a) Marquand chart of 6-variable with decimal indices.



(b) Karnaugh map of 6-variable with decimal indices.

Figure 1. Marquand chart and Karnaugh map representations of a Boolean function.

fact that each square can be represented by regular pattern of decimal indices, going sequentially from left to right and top to bottom. This property can be exploited to conveniently represent a minterm as a group of squares in a regular pattern, however large is the size of the problem.

We shall employ the Marquand chart representation throughout this work and it will be called the **Logical Matrix**, recognizing its matrix properties. It can be easily shown that operations on boolean functions become easier to perform using the logical matrix, e.g., complementation, addition, multiplication, extension to higher variables, transformation of variables, function rotation, expansion about a subset of variables [35]. Also, the regularity in identifying minterms in the logical matrix for any number of variables, becomes even more important in the case of multi-output minimization. A multi-output function is represented as a logical matrix for each individual function, all of them placed side-by-side.

2.2 *K-cell Concept*

2.2.1 Basic Definitions

1. A state of a boolean function

Any boolean function F of n variables can be represented in its canonical form as a sum of products:

$$F = f(x_1, x_2, \dots, x_n) = \sum_{i=0}^{2^n - 1} a_i m_i$$

where x_1, x_2, \dots, x_n are the n variables, m_i are all possible 2^n fundamental minterms (or fundamental products)¹ for n variables, and

$$a_i = 1 \text{ if } F = 1 \text{ when } m_i = 1$$

$$a_i = 0 \text{ if } F = 0 \text{ when } m_i = 1$$

In the above representation of F , each of the 2^n product terms are called *states* of the function F , i.e.,

$$S_i = \{m_i\}, i = 0 \text{ to } 2^n - 1$$

In its truth table representation (for completely specified F), each of the 2^n rows corresponds to a state of F . For each row of the truth table the output F can be 0, 1 or, in case F is incompletely specified, don't care. Thus we say that each state of a boolean function F can be either a 1-state, 0-state or d-state.

This gives us a convenient way to represent F in terms of its non-zero states as:

$$F = m(m_1, m_2, \dots, m_p) + d(d_1, d_2, \dots, d_q)$$

where m is the set of p 1-states and d is the set of q d-states. The rest of the $(2^n) - (p + q)$ states obviously constitute the set of 0-states.

2. Index of a state

Each individual state can be represented by substituting the values 1 in place of each true variable and 0 in place of each complemented variable in the corresponding fundamental minterm of that state. Such an ordered string of 1's and 0's is called the binary *index* of the given state. Note that the order here depends on the order of variables in the fundamental minterm; the left-most one being the most significant variable while the right-most being the least significant variable.

¹ Fundamental minterms (or products) of an n variable boolean system are the products of all n variables, each variable may be true or complemented.

Thus, what we have here is simply a numeric index to represent each state, corresponding to each square in the logical matrix. The binary index can be easily converted into an octal, decimal or hexadecimal index. This is illustrated in Table 1 for the case of four variables A,B,C,D. The corresponding representation with decimal index on the logical matrix and the Karnaugh map is shown in Figure 1.

3. Neighbors of a state

We define the *neighbor states* of a given state as that set of n states, n being the number of variables, such that there is a single bit complementation between index of each state belonging to this set and the index of the given state. The given state and any one of its n neighbor states are called neighboring states.

In Karnaugh map representation, adjacent squares are neighboring states. In the logical matrix representation, all the neighbors of a given state can be located from the regular pattern shown in Figure 2 [35]. Figure 2(a) shows these patterns for a 4-variable grid, while Figure 2(b) shows the extension of these same patterns for an arrangement of 16 such grids forming an 8-variable grid. It is easy to see that the same pattern holds for extension of logical matrix to any number of variables.

4. Generic Variables & Definite Variables

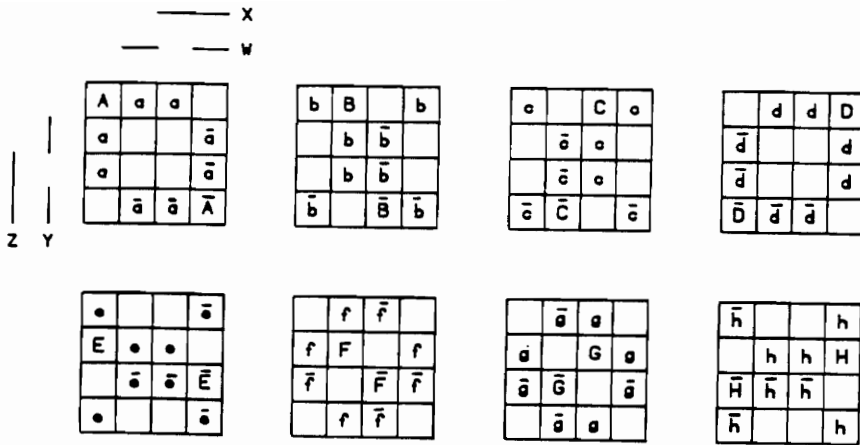
A boolean function $F(X)$ is said to be a function of variable X , where X can be true as well as complemented. Thus we denote F as $F(\hat{X})$, \hat{X} can take the value X or x ,² since X and x are not independent of each other [35].

We define the variable \hat{X} as the *generic variable* and the variables X and x as the *definite variables*. This greatly simplifies the representation of a variable in a minterm as having

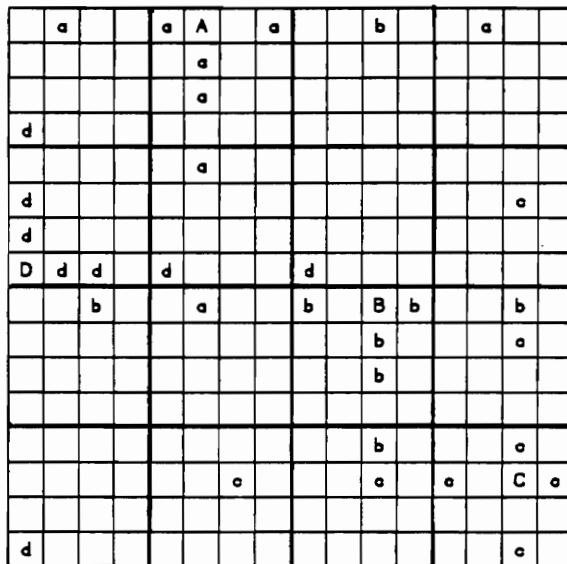
² We shall use the corresponding lowercase letter to denote complement of the variable, when the uppercase letter denotes the true variable.

Table 1. States of boolean functions and their indices.

Minterm or State	Binary Index	Octal Index	Decimal Index	Hex Index
abcd	0000	0	0	0
abcD	0001	1	1	1
abCd	0010	2	2	2
abCD	0011	3	3	3
aBcd	0100	4	4	4
aBcD	0101	5	5	5
aBCd	0110	6	6	6
aBCD	0111	7	7	7
Abcd	1000	10	8	8
AbcD	1001	11	9	9
AbCd	1010	12	10	A
AbCD	1011	13	11	B
ABcd	1100	14	12	C
ABcD	1101	15	13	D
ABCd	1110	16	14	E
ABCD	1111	17	15	F



(a) Location of neighbor states in a 4-variable Marquand chart.



(b) Extending the pattern to an 8-variable Marquand chart.

Figure 2. Regular pattern on a Marquand chart to locate all neighboring states.

the value of 1 or 0 or none (not present). The concept is used later in the definitions of generic k-cells and definite k-cells.

2.2.2 The K-cell

We define a *k-cell* as a configuration by a group of 2^k states within the logical matrix of n variables, in which each state in the group has exactly k neighbors within the group.

In other words, a *k-cell* is a group of 2^k states, in which the index³ of any state differs from that of exactly k other states by one bit. It can be shown that for any *k-cell*, all the 2^k states have the same binary indices except for exactly k -bits, i.e., $(n - k)$ bits of the n -bit index of each state are the same [16]. This can be verified by the fact that number of ways of keeping $(n - k)$ bits constant and varying the remaining k -bits in all possible ways is obviously 2^k , as each of the k -bits can either be 1 or 0. From this definition we can say that two states can be covered by a single 1-cell if their indices differ in just one bit; four states can be covered by a single 2-cell if their indices are same except for two bits; eight states can be covered by a single 3-cell if their indexes are same except for three bits and so on. Also, a state with no neighbour can be covered by a 0-cell, which covers that state only.

To illustrate the *k-cell* concept, and its use for minimization, consider the example shown in Figure 3. It can be seen that states $ABcd$ & $ABcD$ can be covered by a 1-cell $ABcd + ABcD = ABc$. The state $abCD$ can only be covered by a single 0-cell $abCD$, as this state has no neighbours. Four states $abcd$, $aBcd$, $Abcd$, & $ABcd$ can be covered by a single 2-cell cd .

An interesting inference from the above illustration is that a *k-cell* can be represented by $(n - k)$ variables out of n . This means that in representing a *k-cell*, k variables are not *present* in

³ Unless specified otherwise, the term "index" would mean binary index by default.

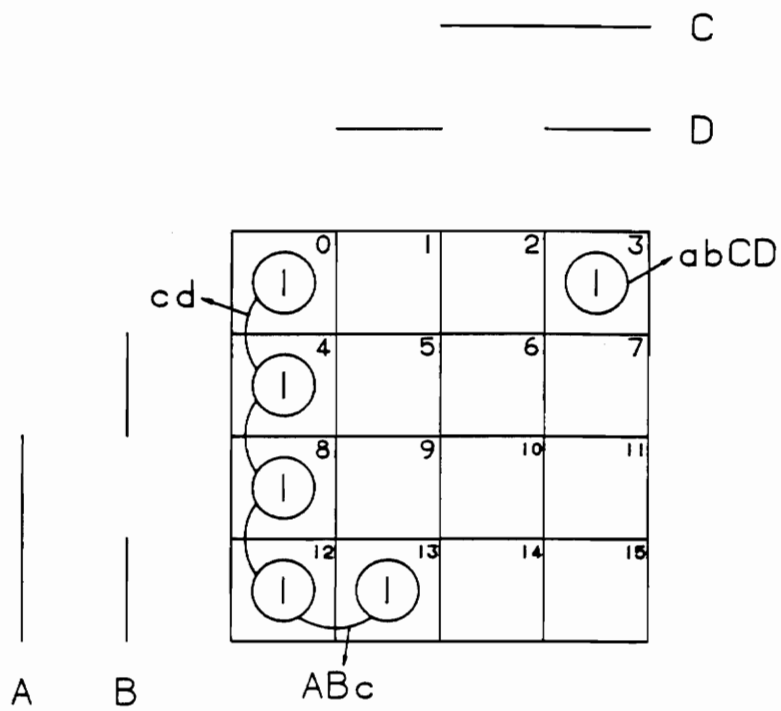


Figure 3. An example illustrating the k-cell concept.

the expression of that k-cell, i.e., these k variables are not required to be explicitly specified as 1 or 0 for this k-cell to be constructed. This leads to the concept of generic k-cells and definite k-cells.

Generic K-cell

Using the idea of generic variables, as introduced earlier, we can look at any given k-cell as logical products of a set of $(n - k)$ generic variables $\{\hat{X}_i\}$, $i = 0$ to $(n - k - 1)$, and each \hat{X}_i taking a value of X_i or x_i for the given k-cell. In other words, in the expression of any particular k-cell, a given variable is 1 (true), 0 (complemented) or not present.

The group of k-cells in which a particular set of $(n - k)$ generic variables is present and the rest k are not present is called a *generic k-cell*.

A generic k-cell may be represented by an index of n-bits, each bit corresponding to the presence or the absence of a variable. We introduce the following convention for each bit of the **generic k-cell index**:

- 1 : if the corresponding variable is present.
- 0 : if the corresponding variable is not present.

Table 2 illustrates the concept of generic k-cells for $n = 3$.

Note that, while 1 is the generic n-cell with no variables present and covering all the states in the entire grid, $\hat{A}\hat{B}\hat{C}$ is a generic 0-cell covering any one, and only one, state in the grid. The concept of generic k-cells helps divide the set of all k-cells in an n-variable system into several sets of k-cells, each group corresponding to a generic k-cell.

Table 2. Generic k-cells and their indices.

Generic k-cell	Order	Generic cell index
$\hat{A}\hat{B}\hat{C}$	0-cell	111
$\hat{A}\hat{B}, \hat{A}\hat{C}, \hat{B}\hat{C}$	1-cell	110,101,011
$\hat{A}, \hat{B}, \hat{C}$	2-cell	100,010,001
I	n-cell	000

Definite K-cell

A definite k-cell, as opposed to a generic k-cell, consists of a definite variable in place of each generic variable in a generic k-cell. In other words, a group of definite k-cells can be derived from each generic k-cell by simply substituting for each generic variable \hat{X}_i present with either X_i or x_i . Hence, a group of definite k-cells is derived from each generic k-cell by substitution.

Again, to represent each definite k-cell by its **definite k-cell index**, of n-bits, we follow the following convention:

- 1 : if the corresponding variable is 1 (true).
- 0 : if the corresponding variable is 0 (complemented)
or the corresponding variable is not present.

Table 3 illustrates this with some examples.

A k-cell can be completely represented by the pair of generic cell index (g-index) and the definite cell index (k-index). Note that the g-index or the k-index alone cannot uniquely specify a k-cell. Table 4 for the example in Figure 3 shows the relationship among g-index, k-index and state index (s-index).

Given the (g-k) index pair, it is easy to obtain the k-cell expression. It calls for substitution only for those generic variables \hat{X}_i which have corresponding g-index bit = 1, with the definite variables X_i and x_i for corresponding k-index bit = 1 and 0, respectively. For example, for the (g-k) pair 1110/1100, the k-cell is ABc

Another very important relation observed above is:

Table 3. Definite k-cell and their indices.

Definite k-cell	Order	Definite cell index
ABc	0-cell	110
Ab	1-cell	100
a	2-cell	000
1	n-cell	000

Table 4. Relationship among state, generic k-cell and definite k-cell indices.

States	s-index	Gen. k-cell	g-index	Def. k-cell	k-index	(g-k) pair
abCD	0011	$\hat{A}\hat{B}\hat{C}\hat{D}$	1111	abCD	0011	1111/0011
ABcd	1100	$\hat{A}\hat{B}\hat{C}$	1110	ABc	1100	1110/1100
ABcD	1101					
abcd	0000	$\hat{C}\hat{D}$	0011	cd	0000	0011/0000
aBcd	0100					
Abcd	1000					
ABcd	1100					

$$(s\text{-index}) \text{ .AND. } (g\text{-index}) = (k\text{-index})$$

where .AND. here is bit by bit word-AND operation. This is an extremely useful relation to obtain a definite k-cell from a given generic k-cell and a given state. Also, given the k-cell as (g-k) index pair, we can use the above relation to check if a given state belongs to that k-cell. For the above example,

- Given the state ABcd and the g-cell $\hat{A}\hat{B}\hat{C}$, we can form the k-cell⁴ ABc from the word-AND operation:

$$(1100) \text{ .AND. } (1110) = (1100)$$

- Also, now that we have the (g-k) pair 1110/1100 for the k-cell ABc, we can say that the state ABcD belongs to this k-cell as the following relation holds:

$$(1101) \text{ .AND. } (1110) = (1100)$$

In the following section, we look at some quantitative results on the number of g-cells and k-cells possible for an n-variable matrix.

2.3 Some Important Results

1. For a given k, $0 \leq k \leq n$, number of possible g-cells is given by:⁵

$$G(k)_{\max} = \binom{n}{k}$$

⁴ The term g-cell will be used in short for generic k-cell and k-cell will be used for definite k-cell, whenever there is no scope of confusion.

⁵ $\binom{x}{y}$ denotes combination of y entities out of x.

This is obvious, since each of these g-cells has only $(n - k)$ out of n variables present, i.e., exactly k variables missing. Thus $G(k)_{\max}$ is the number of ways to select k missing variables from n available.

2. Total number of possible g-cells for all values of k (0 to n) for a given n is given by:

$$G_{\max} = \sum_{k=0}^n \binom{n}{k} = 2^n$$

This last expression comes straight from the Binomial theorem.

3. For a given k , $0 \leq k \leq n$, the number of possible k -cells is given by:

$$K(k)_{\max} = 2^{(n-k)} \times \binom{n}{k}$$

This comes from the fact that for each of the g-cells for a given k we can have $2^{(n-k)}$ k -cells, as each of the $(n - k)$ variables present in the g-cell can either be 0 or 1.

4. Total number of possible k -cells for all values of k (0 to n) for a n is given by:

$$K_{\max} = \sum_{k=0}^n 2^{(n-k)} \times \binom{n}{k} = 3^n$$

The last expression is easy to derive:

$$K_{\max} = \sum_{k=0}^n 2^{(n-k)} \times \binom{n}{k} = \sum_{k=0}^n 1^n \times 2^{(n-k)} \times \binom{n}{(n-k)} = (1 + 2)^n = 3^n$$

The last step again is from Binomial theorem.

5. For a given k , we can divide all the k -cells into sub-groups with equal number of 1's in their k -indices. The idea is to obtain the distribution over the k -cells of number of 1's in each k -cell.

We know that for a given k , there are $\binom{n}{k}$ generic cells, and each of these has $(n - k)$ generic variables. Since each of these $(n - k)$ g-variables can be either 1 or 0, and if $N(j)$ is the number of k -cells with j 1's for the given k -cell of given k , we have,

$$N(0) = \binom{n-k}{0}, N(1) = \binom{n-k}{1}, N(2) = \binom{n-k}{2}, \text{ and so on.}$$

And so, for the general case, the number of k -cells with j number of 1's for a given generic k -cell is:

$$N(j) = \binom{n-k}{j}, 0 \leq j \leq (n - k)$$

Thus the number of k -cells with j number of 1's for all generic k -cells for a given k is:

$$N(j)_{\max} = \binom{n}{k} \times \binom{n-k}{j}, 0 \leq j \leq (n - k)$$

The above expression gives the maximum number of k -cells possible with exactly j 1's at each level of k .

2.4 Generation of k -cells

The inter-relationship among the three indices, viz., (s-index) .AND. (g-index) = (k-index), can be used to find the k -cell from a given g -cell and a state. Thus if we can generate all possible g -cells for all values of k , we can obtain the definite k -cells covering any of the non-zero states of a boolean function, using the above relation. This way, we generate only the k -cells covering at least one non-zero state of the given function and avoid generating the rest. We shall see in Chapter 4 that we do not have to generate g -cells for all values of k . This will further cut down on the number of k -cells to be generated to cover all non-zero states of the function.

It should be noted here that the conventional methods use an altogether different approach for doing the same thing. In these methods, starting from the lowest possible order of k-cells, i.e., 0-cells, higher order k-cells are computed by extensive comparison of the indices of the states. Although the comparisons for 1-bit difference in indices are carried out only among the adjacent groups of states, all states in a group having same number of 1's in their s-indices, the number of operations shoots up exponentially with the number of variables. This approach involves a lot of redundant operations due to the generation of redundant lower-order k-cells in the process, which cannot co-exist with the higher order k-cells. This will be discussed further in the following chapters.

We present here a simple procedure to generate all possible g-cells for a given value of k. Recall that we have to select (n - k) variables out of the n available to form any such g-cell. The following routine will select all possible combinations of (n - k) variables systematically.

Let the n bit-positions of the g-cell index bit-string be denoted by:

$$G = b_n b_{n-1}, \dots, b_2 b_1$$

where G is the binary number representing the g-cell index.

The idea here is to move (n - k) 1's on an n-bit string in such a way that they cover all possible combinations. We start with all 1's on the right most (n - k) bits of G and end up with all 1's on the leftmost (n - k) bits of G.

Start

1. Set bit-positions $b_{n-k}, \dots, b_2 b_1 = 1, \dots, 11$. Thus we start with all (n - k) 1's at lowermost (n - k) bits out of n. Let the 1's on the positions b_{n-k}, \dots, b_1 be called $1_{n-k}, \dots, 1_1$.
Go to step 4.
2. Check if $B = n - k$.
If so, go to step 4.

3. Move 1_g left by 1 bit, say on position b_j . Place the 1's $1_{B+1}, \dots, 1_{n-k}$ in positions on the left of b_j , i.e., positions $b_{j+1}, \dots, b_{j+(n-k-B)}$.
4. Set g-cell index $R = G$ and store R .
5. Set counter $B = n - k$.
6. Let current position of 1_g be b_j . Check if $j < B + k$, i.e., if 1_g can move any further to the left.
If so, go to step 2.
7. Set $B = B - 1$ and Check if $B = 0$.
If not, go to step 6.

Stop

We shall use this routine as a part of the first stage (k-cell generation) of the minimization program in chapter 4.

For an example, a sequence of generation of g-cells is listed in Table 5, for the case of $n = 6$ and $k = 2$.

Once we have all possible k-cells that could be used to cover a state in the given boolean function, we need to select only those k-cells out of these, which cover all the states minimally. The criteria for minimality and for selection of k-cells are discussed in the next chapter.

Table 5. Sequence of generic k-cells generation.

Binary G	Decimal G
0 0 1 1 1 1	15
0 1 0 1 1 1	23
1 0 0 1 1 1	39
0 1 1 0 1 1	27
1 0 1 0 1 1	43
1 1 0 0 1 1	51
0 1 1 1 0 1	29
1 0 1 1 0 1	45
1 1 0 1 0 1	53
1 1 1 0 0 1	57
0 1 1 1 1 0	30
1 0 1 1 1 0	46
1 1 0 1 1 0	54
1 1 1 0 1 0	58
1 1 1 1 0 0	60

Chapter 3

Proposed Algorithm

3.1 *Problem Definition*

Given a multi-output n -variable function $(F(n))$ consisting of a set of functions (F_1, F_2, \dots, F_x) , each specified with a set of 1-states $m(m_1, m_2, \dots, m_p)$ and a set of don't-care states $d(d_1, d_2, \dots, d_q)$; find the multi-output minimal form(s) of function F , implementing it in two-level logic with minimum possible cost.

3.2 Algorithm Statement

Rule 1: Selection Rule

A multi-output maximal k-cell (K1) covering a reference state (S1) in a function (F1) may be included in the minimal expression of F1, if K1 is the only k-cell⁶ covering S1 in F1.

Rule 2: Elimination Rule

If K1 is not the only k-cell covering S1 in F1, an alternate k-cell (K2) covering S1 in F1 can be eliminated

1. from *all* the functions where it exists, provided:
 - a. K2 does not cover a free state not covered by K1 in F1;
AND
 - b. K2 is not of a lower cost than K1;
AND
 - c. in any other function (F2) where *corresponding reference state* (S1') exists, condition 1a above is satisfied by *corresponding k-cells* K2' and K1' in F2.

OR

2. from only the function F1, provided:
 - a. K1' has already been selected in any other function F2;
AND
 - b. condition 1a above is satisfied by K2 and K1 in F1.

⁶ Henceforth, the term 'k-cell' will be used interchangeably with 'multi-output maximal k-cell', unless specified otherwise.

Rule 3: Branching Rule

If no k-cell can be selected after application of Rule 1 & 2 to all possible reference states, a reference state with least number of alternate k-cell coverage⁷ is selected, each of these alternate k-cells being selected one at a time and for each of these selection branches, Rule 1 & 2 are applied as before until all states are covered.

The minimal form or forms are the branches with the least cost.

⁷ Alternate k-cell coverage includes only the k-cells which cannot be eliminated against each other.

3.3 Terminology

1. *Free and bound states:*

A free state is a 1-state of the boolean function which has not yet been covered by any of the k-cells selected in the minimal form being constructed.

A bound state is either a 1-state of the boolean function which has been covered by a k-cell selected in the minimal form being constructed or it is a d-state (don't care) of the boolean function.

2. *Multi-output maximal k-cell:*

A maximal k-cell (or a prime implicant) for a single-output function is a k-cell existing in the function which is not a subcell of any other k-cell also existing in the function.

If all the free states belonging to a k-cell are covered by another k-cell, the former is called the dominated k-cell while the latter is called the dominating k-cell.

A multi-output maximal k-cell (or a multi-output prime implicant) for a multi-output function is either a maximal k-cell of any single function or a maximal subcell for the multi-output function.

A maximal subcell for a multi-output function is a subcell of a single function maximal k-cell, provided that it is the highest order k-cell shared in another function. In other words it is the maximally shared k-cell among two or more functions in a multi-output function.

Note that a multi-output k-cell dominates another if it does so in all the functions.

3. *Reference state:*

A reference state is a free state belonging to any function, provided that the maximal k-cells covering it do not include (or dominate) those covering another free state belonging to the same function.

If k-cells covering a free state *in a given function* dominate those covering another free state *in that same function*, the former is called the dominating free state while the latter is called the dominated free state. Note that a free state can dominate another free state only if both belong to the same function and all k-cells covering the latter also cover the former.

Note that if the above condition is not satisfied, then the dominating free state cannot be taken as a reference state due to the fact that it can be considered as effectively bound. This is obvious because such a state will always be covered whenever the free state it dominates is covered by any k-cell covering it in the given function. In other words, a dominating free state cannot be a reference state.

4. *Cost of a k-cell:*

The definition of cost function depends on the particular device in mind for the multi-output logic implementation. The application of Rule 2 in checking condition 1b is independent of the cost function specified. We define below two different cost functions.

a. *Gate-input count cost function:*

This is the cost function defined for two level multi-output implementation using logic gates in AND-OR (or NAND-NAND) configuration. Here, the cost of a k-cell denotes

the number of input lines required in the AND-OR implementation of that term. Each term will require an input line to the AND gate corresponding to each literal present in the term. In addition to this, the output of this AND gate will form one input line to the OR gate. The total of these AND-costs and OR-costs is called the cost of the k-cell.

There are two special cases of cost definition for such an implementation.

- For an $(n - 1)$ k-cell with only one literal in the corresponding term, obviously we don't need an AND gate as no other literal is present. In this case the literal can be directly given to the OR gate for the particular function where the k-cell belongs. Hence, there is no AND-cost for a k-cell with only one literal and so the cost of such a k-cell is just the OR-cost ($= 1$).
- For an individual output function with only one k-cell covering all the free states of that function, i.e., only one term in its expression, there is no OR-cost. This is obvious as no other term is present for any other input of the OR gate. Hence, there is no OR-cost for an only k-cell selected in any function and the cost of such a k-cell is just the AND-cost ($=$ number of literals).

Note that after a k-cell has already been selected for a function, the cost of that k-cell in any other function is now just the OR-cost, as AND-cost is eliminated by sharing the same AND gate. Thus, for any k-cell already selected before, $\text{cost} = 1$.

b. *PLA area cost function:*

When a multi-output switching function is implemented in a PLA, the cost is the total area occupied by all the wires. The wires are one each horizontally for each AND term, and one each vertically for each input literal and each output function. Thus, the cost-area can be defined as,

$$\text{cost-area} = \text{product terms} \times (\text{input literals} + \text{output functions})$$

It should be noted that for a PLA with fixed availability of both true and complemented input wires, presence of a true or complemented variable also implies the presence of the other. In a VLSI implementation of a logic function using PLA structure, the true and complemented wires are independently provided. We shall use the latter case as the PLA cost definition for its generality.

5. *Corresponding reference state:*

The corresponding reference state is the state in a different function than the reference state function, but having the same state index as the reference state.

This corresponding state can be free, bound or may not exist at all in a given function. Note again that all dominating free states, as defined above, are considered as bound in all respects.

6. *Corresponding k-cell:*

The corresponding k-cell is a k-cell in a different function than the reference state function, but whose indices are same as the k-cell under consideration in the reference state function.

Again, the corresponding k-cell may or may not exist in another function. If it exists in another function, the k-cell under consideration is said to be shared by the two functions. If a k-cell is shared in two functions, in each covering a free state, this k-cell is said to be interfering between the two functions.

3.4 Justification

3.4.1 Rule 1: Selection Rule

The selection rule for K1, when combined with Rule 2 for elimination of each alternate k-cell K2, can be rewritten as follows:

A *multi-output maximal k-cell* (K1) covering a *reference state* (S1) in a function (F1) may be included in the minimal expression of F1, if for *any* alternate k-cell (K2) covering S1 in F1:

1. Condition 1(a,b&c) of Rule 2 satisfied;
OR
2. Condition 2(a&b) of Rule 2 satisfied.

Necessity

The necessity for each part of both the conditions can be shown by proving that an alternate k-cell K2 covering S1 cannot be eliminated (and so K1 cannot be selected) when that part of the condition is not satisfied, even when other parts are satisfied.

Condition 1

1. Part a:

Suppose condition 1a is not satisfied, while 1b & 1c are satisfied.

This means that K2 covers an extra free state not covered by K1 in F1, but not in any other function. Also, K2 is not of a lower cost.

Now, if K2 happens to be the only k-cell covering that extra free state, it will definitely be selected in F1 to cover that free state; thus covering S1 too. This renders k-cell K1 redundant for covering S1 in F1.

Hence, K2 cannot be eliminated as an alternate coverage for S1 and K1 cannot be selected at this point.

2. Part b:

Suppose condition 1b is not satisfied, while 1a & 1c are satisfied.

This means that K2 does not cover an extra free state not covered by K1 in any function. Also, K2 is of a lower cost.

If K1 is not selected in any other functions, then K2 becomes the k-cell of lower cost than K1 to cover S1.

Thus, K2 cannot be eliminated as K1 cannot be selected to cover S1.

3. Part c:

Suppose condition 1c is not satisfied, while 1a & 1b are satisfied.

This means that K2 does not cover an extra free state not covered by K1, but does cover an extra free state in another function. Also, K2 is not of a lower cost.

If K2 happens to be essential to cover the extra free state in another function, cost of K2 in F1 becomes lowest possible, definitely lower than K1.

So, K2 may be a lower cost coverage for S1 and K1 cannot be selected now.

Condition 2

1. Part a:

Suppose condition 2a is not satisfied and 2b is satisfied.

As K1 is not selected in another function, and nothing is known about costs of K1 & K2, K2 can be of lower cost to cover S1, even though K1 covers all free states covered by K2.

This is same as condition 1b not satisfied, and so K2 cannot be eliminated now.

2. Part b:

Suppose condition 2b is not satisfied but 2c is satisfied, then cost of K1 is lowest in F1.

But if K2 is essential for the extra free state it covers in F1, but not covered by K1, S1 will be covered by K2 in any case.

This is again same as condition 1a not satisfied and thus K2 cannot be eliminated against K1.

Sufficiency

The sufficiency of both conditions can be shown if the K-cell K1 can be proved to be minimal when each part of a condition is satisfied.

Condition 1

Given:

For a given k-cell K_1 covering a reference state S_1 in function F_1 , an alternate k-cell K_2 covering S_1 in F_1 can be eliminated if

1. K_2 does not cover a free state S_2 not covered by K_1 in F_1 ;
2. K_2 is not of a lower cost than K_1 ;
3. in any other function F_2 with corresponding reference state S_1' , corresponding k-cell K_2' does not cover a free state S_3 not covered by corresponding k-cell k_1' .

To show that:

K_1 can be included in the minimal expression of F_1 ; i.e.,

1. K_1 is the lowest cost k-cell covering S_1 in F_1 ;
2. K_1 is not redundant in F_1 .

Proof:

It is easy to show that K_1 is a lowest cost k-cell covering S_1 , as it is obvious from condition 2 above that no alternate k-cell K_2 exists in F_1 with a cost lower than that of K_1 .

Hence, K_1 should be selected to cover S_1 rather than any alternate k-cell K_2 if K_1 can be shown to be non-redundant in F_1 .

Note that K_1 cannot be made redundant by K_2 in F_1 as, from condition 1 above, K_2 does not cover an extra free state S_2 not covered by K_1 . If it did, this extra free state covered by K_2 might make selection of K_2 essential.

Thus, the only way in which K1 can be redundant is if K2' is essential in F2 so that, due to the selection of K2' in F2, cost of K2 = 1 in F1 which cannot be greater than the cost of K1.

Hence, to show that K1 is not redundant in F1, we now have to show that,
there cannot exist a k-cell K2 in F1 such that K2' in F2 is essential. ... A

Suppose there does exist such a k-cell K2 such that K2' in F2 is essential.

This means that,

K2' covers a free state S3 such that K2' is the only k-cell covering S3 in F2. ... B

But, from condition 3 above, K2' does not cover a free state not covered by K1' in F2. Thus, obviously, every free state covered by K2' is covered by K1'; and hence S3 covered by K2' is also covered by K1'.

It can be seen that K2' cannot be selected instead of the alternate k-cell K1' to cover reference state S3, as, from given conditions 2 & 3 above, K1' may cover a free state not covered by K2' or K2' may be of a higher cost.

It follows that,

S3 is covered by K2' as well as K1' and thus K2' is not the only k-cell covering S3. ... C

Statements B and C are clearly contradictory, which shows that the above assumption is not possible.

Thus we can say that validity of statement A has been shown.

Hence, we can infer that K1 is indeed not redundant, given the conditions 1,2 & 3 above.

We have showed that K1 can be included in the minimal expression of F1 as K1 is the lowest cost k-cell covering S1 *and* it is not redundant in F1.

Condition 2

Given:

For a given k-cell K1 covering a reference state S1 in function F1, for any alternate k-cell K2 covering S1 in F1:

1. corresponding k-cell K1' in F2 has already been selected;
2. K2 does not cover a free state S2 not covered by K1 in F1.

To show that:

K1 can be included in the minimal expression for F1, i.e.,

1. K1 is the lowest cost k-cell covering S1 in F1;
2. K1 is not redundant in F1.

Proof:

Again, it is obvious that K1 is of lowest cost as cost of K1 = 1 in F1 due to the selection of K1' in F2.

Hence, K1 can be selected immediately in F1 if it can be shown to be non-redundant in F1.

K1 can be rendered redundant if an alternate k-cell K2 is essential in F1.

This is possible only if K2 covers a free state where K2 is the only k-cell covering S1. In other words, if K2 covers a free state which is not covered by K1. But this contradicts the given condition 2 above.

Thus, it is not possible to have K_2 such that it is essential in F_1 . Hence, K_1 cannot be rendered redundant by an alternate k-cell K_2 covering S_1 in F_1 .

The other way in which K_1 can be redundant in F_1 is if an alternate k-cell K_2 exists in F_1 such that K_2' in F_3 is essential. But since, from condition 1 above, K_1' in F_2 has been already selected, cost of K_1 in $F_1 = 1$.

Thus, even if a K_2 exists in F_1 such that K_2' in F_3 is essential, cost of K_2 in $F_1 = 1$, which is not less than that of K_1 in F_1 . Hence, to cover S_1 , K_1 is still the lowest cost k-cell in F_1 .

Hence, since S_1 has to be covered by a k-cell, it might as well be covered by K_1 , as it can never cost more than any other k-cell.

Thus, K_1 is not redundant in F_1 , given conditions 1 and 2 above.

We have showed that K_1 can be included in the minimal expression of F_1 as K_1 is the lowest cost k-cell covering S_1 and it is not redundant in F_1 .

3.4.2 Rule 2: Elimination Rule

The necessity has already been shown under the Selection Rule proofs. For sufficiency, we have to show that for any k-cells K_1 and K_2 , if condition 1 or 2 satisfied, we can eliminate K_2 from any further consideration as an alternate k-cell in all the functions, irrespective of whether K_1 is the minimal coverage or not.

Condition 1

Given:

For given k-cells K1 & K2 in F1, conditions 1(a,b&c) satisfied at a given reference state S1.

To show that:

K2 can be eliminated from all the functions where it exists, i.e., K2 is redundant in all the functions.

Proof:

For K2 to be non-redundant in any function, either it should be of a lower cost than K1 or it can possibly be essential in F1 or another function.

From condition 1b, we know that it is not of a lower cost than K1.

From condition 1a & 1c, we can say that all free states covered by K2 are also covered by K1 in F1 and any other function; so K1 cannot be essential in any function.

Hence, K2 is redundant in all the functions as it is neither of a lower cost nor covers an extra free state not covered by K1.

Therefore, K2 can be eliminated against K1.

Condition 2

Given:

For given k-cells K1 & K2 in F1, conditions 2(a&b) satisfied at a given reference state S1.

To show that:

K2 can be eliminated from F1, i.e., K2 is redundant in F1.

Proof:

Since K1 is already selected in another function, cost of K1 in F1 = 1, which cannot be higher than that of K2 (previously selected or not).

Hence, to eliminate K2 from F1, we just have to make sure that K2 is not essential in F1. From condition 1b, we know that K2 cannot be essential, since all free states covered by K2 are also covered by K1 in F1.

Thus, K2 is redundant in F1 and so can be eliminated.

3.4.3 Rule 3: Branching Rule

Justification of Rule 3 is rather obvious. We select a reference state with least possible alternate coverage so that we have least possible branches in most cases. Ending up with least possible branches cannot be ensured always because a branching with higher alternate coverage might give no further branches, while a branching with lower alternate coverage might give further branching.

In other words, we choose a depth-first search rather than a width-first search. In its computer implementation, we shall introduce some convenient cut-off criterion to terminate the branch as soon as its cost exceeds the cut-off limit. This effectively limits the maximum depth of any search branch.

3.5 Illustration

3.5.1 Gate-input count cost function

The following example illustrates the conditions for k-cell selection. Here, the multi-output logical matrix discussed in Chapter 2 is used to compute the solution manually. For each k-cell selection, Figure 4(a-v) shows the graphical representation on the logical matrix.

Problem:

$$\begin{aligned} F(4) &= (F1, F2, F3) \\ F1 &= m(0,4,5,9) + d(8) \\ F2 &= m(4,7,8,9,13) + d(0,5) \\ F3 &= m(7,13,15) \end{aligned}$$

Solution:

Figure 4(a) shows the problem transformed to the logical matrix.

We now scan each free state (S) for possible selection of it as a reference state (S1) (non-dominating free state) and the possible selection of a k-cell (K1) covering it in that function (F1) using Rules 1 & 2.

Note that, from the definition, all dominating free states are considered bound and therefore cannot be considered as reference states. For a free state S in F1 to be a reference state (non-dominating) S1, all other free states covered by k-cells covering S in F1 should in turn be covered by a k-cell not covering S in F1.

Rule 1-2:

1. *Free state:* $S = [1,0]^8$

k-cells covering S: $acd(0,4)^9, bcd(0,8)$

For S to be a reference state, all other free states covered by acd and bcd , viz. state $[1,4]$ (state $[1,8]$ is bound), should be covered by a k-cell in F1 other than acd and bcd . As the free state $[1,4]$ has an alternate k-cell in $aBc(4,5)$, it cannot be dominated by $[1,0]$. Therefore, S is a reference state.

Reference state: $S1 = [1,0]$

- For $K1 = acd$, $K2 = bcd$ violates condition 1c, as $K2'$ in F2 covers a free state $[2,8]$ not covered by $K1'$ in F2.
- For $K1 = bcd$, $K2 = acd$ violates condition 1a, as $K2$ in F1 covers a free state $[1,4]$ not covered by $K1$ in F1.

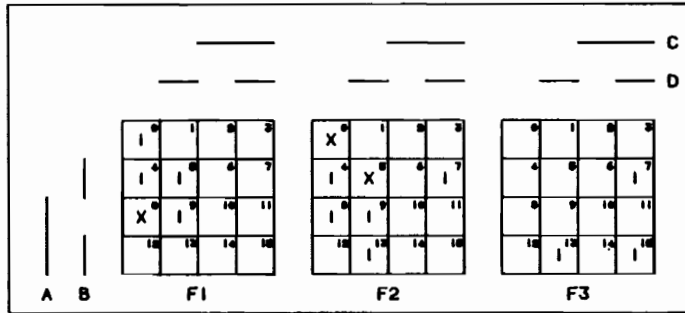
k-cell eliminated: None

k-cell selected: None

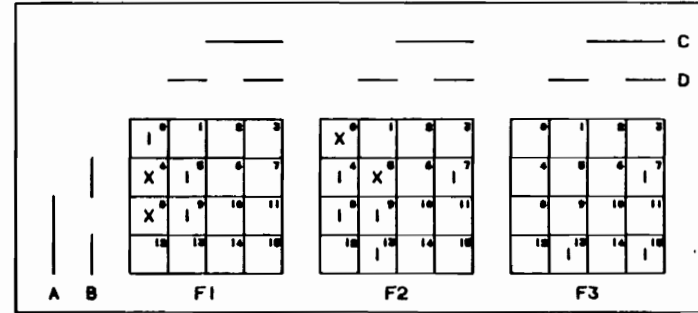
States covered: None

⁸ The notation $[f,s]$ will be used to denote a state which belongs to function f and whose index is s .

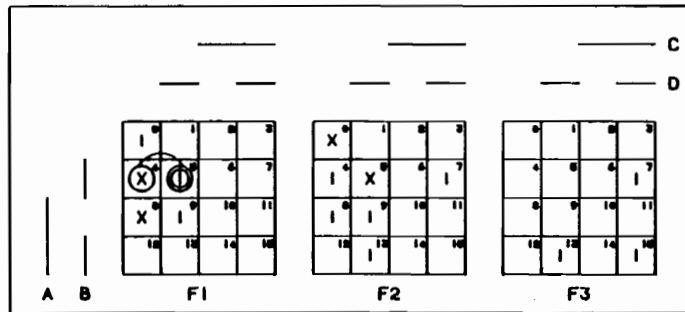
⁹ The parentheses immediately following the k-cell consists of the list of states covered by that k-cell.



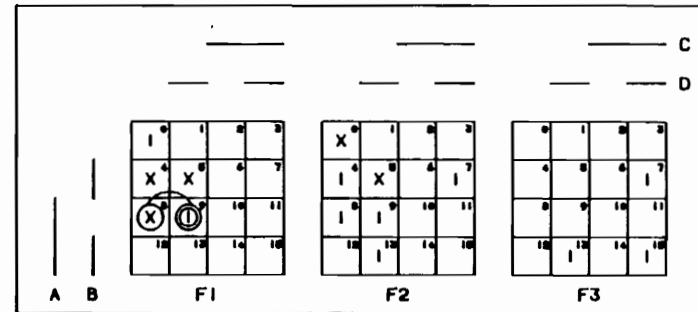
(a) The problem represented on the logical matrix.



(b) Dominating state [1,4] is bound.



(c) K-cell aBc selected for ref. state [1,5].



(d) K-cell Abc selected for ref. state [1,9].

Figure 4. Application of the proposed algorithm for an example using the logical matrix.

Cost:¹⁰ None

2. *Free state:* $S = [1,4]$

k-cells covering S: $acd(0,4), aBc(4,5)$

The free state [1,0] has an alternate k-cell in $bcd(0,8)$, but the free state [1,5] has no other coverage other than $aBc(4,5)$. Therefore, [1,4] dominates [1,5] and thus $S = [1,4]$ is not a reference state.

Since [1,4] is a dominating state, it can be effectively considered as bound (Figure 4(b)). This is due to the fact that, as discussed previously, [1,4] will be covered automatically when [1,5] is covered.

Reference state: Not possible

k-cell eliminated: None

k-cell selected: None

States covered: [1,4]

Cost: None

3. *Free state:* $S = [1,5]$

k-cells covering S: $aBc(4,5)$

As the state [1,4] has been bound, aBc covers no other free states. Therefore, [1,5] cannot dominate any free state. Therefore, S is a reference state.

Reference state: $S_1 = [1,5]$

Since $K_1 = aBc$ is the only k-cell covering [1,5], Rule 1 is satisfied. Hence, k-cell aBc is selected in (Figure 4(c)).

It can be seen that aBc cannot be the only k-cell selected in F_1 , as not all free states in F_1 are covered by aBc . This should always be checked when the k-cell is selected in a function is the first one in that function. If the k-cell does cover all free states in that function, no OR-cost will be incurred by selection of such a k-cell.

k-cell eliminated: None

K-cell selected: aBc in F_1 .

States covered: [1,5]

*Cost:*¹¹ $3 + 1$

4. *Free state:* $S = [1,9]$

k-cells covering S: $Abc(8,9)$

As Abc covers no other free states, [1,9] cannot dominate any free state. Therefore, S is a reference state.

¹⁰ This will indicate the total (AND + OR)-cost of the minimal form at this point.

¹¹ The first figure is the OR-cost, while the second one is the AND-cost.

Reference state: $S1 = [1,9]$

Since $K1 = abc$ is the only k-cell covering $[1,9]$, Rule 1 is satisfied. Hence, k-cell abc is selected in $F1$ (Figure 4(d)).

k-cell eliminated: None

K-cell selected: abc in $F1$.

States covered: $[1,9]$

Cost: $6 + 2$

5. *Free state:* $S = [2,4]$

k-cells covering S: $aBc(4,5)$, $acd(0,4)$

As the states $[2,5]$ and $[2,0]$ are both bound, aBc covers no other free states. Therefore, $[2,4]$ cannot dominate any free state. Therefore, S is a reference state.

Reference state: $S1 = [2,4]$

- For $K1 = aBc$, $K2 = acd$, condition 2(a&b) satisfied, since aBc has been selected in $F1$ and acd does not cover a free state not covered by aBc . Therefore, acd is eliminated from $F2$ only.

Since $K1 = aBc$ is now the only k-cell covering $[2,4]$, Rule 1 is satisfied. Hence, k-cell aBc is selected in $F2$ (Figure 4(e)).

Note again that aBc does not cover all free states in $F2$, and so it cannot be the only k-cell.

k-cell eliminated: acd in $F2$.

K-cell selected: aBc in $F2$.

States covered: $[2,4]$

Cost: $6 + 3$

6. *Free state:* $S = [2,7]$

k-cells covering S: $aBD(5,7)$, $aBCD(7)$.

No other free states covered by k-cells covering $[2,7]$, as $[2,5]$ is bound. Therefore, S is a reference state.

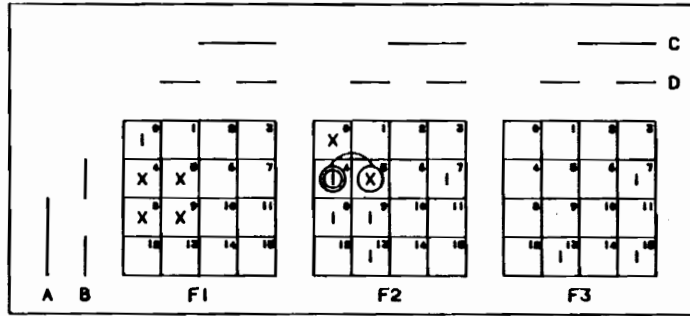
Reference state: $S1 = [2,7]$

- For $K1 = aBD$, $K2 = aBCD$ violates condition 1c, as $K2'$ in $F3$ covers a free state $[3,7]$ not covered by $K1'$ in $F3$ ($K1'$ does not exist in $F3$). Hence, $aBCD$ cannot be eliminated.
- For $K1 = aBCD$, $K2 = aBD$ violates condition 1b, as aBD is of a lower cost than $aBCD$. Hence, aBD cannot be eliminated.

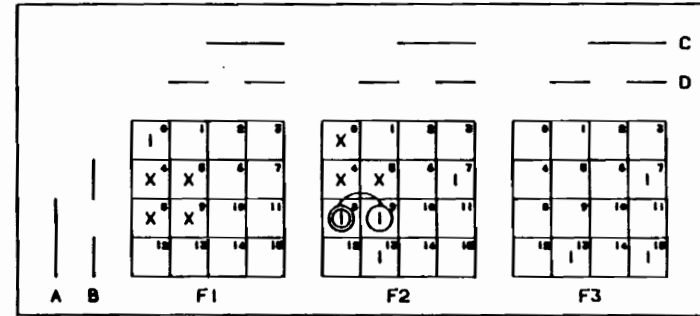
k-cell eliminated: None

k-cell selected: None

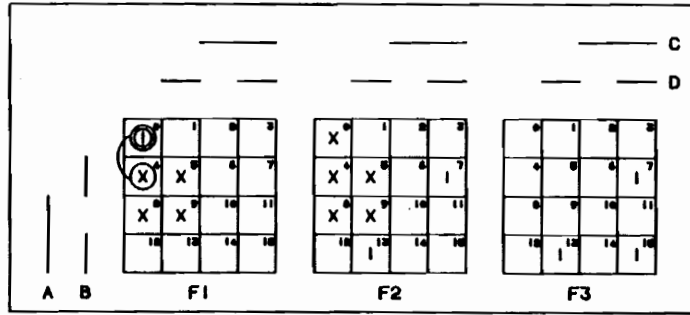
States covered: None



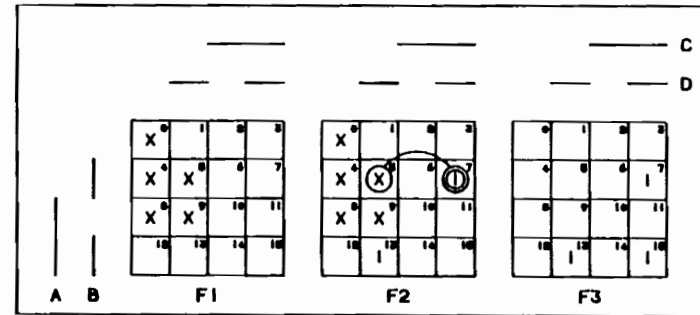
(e) K-cell aBc selected for ref. state [2,4].



(f) K-cell Abc selected for ref. state [2,8].



(g) K-cell acd selected for ref. state [1,0].



(h) K-cell aBD selected for bra. state [2,7] for Branch 1.

Figure 4. Application of the proposed algorithm for an example using the logical matrix.

Cost: $6 + 3$

7. Free state: $S = [2,8]$

k-cells covering S: $Abc(8,9)$, $bcd(0,8)$

As the state $[2,9]$ has an alternate *k-cell* in *k-cell* $AcD(9,13)$, $[2,8]$ cannot dominate $[2,9]$. Therefore, S is a reference state.

Reference state: $S1 = [2,8]$

- For $K1 = Abc$, $K2 = bcd$, condition 2(a&b) satisfied, since Abc has been selected in $F1$ and bcd does not cover a free state not covered by Abc . Therefore, bcd is eliminated from $F2$ only.

Since $K1 = Abc$ is now the only *k-cell* covering $[2,8]$, Rule 1 is satisfied. Hence, *k-cell* Abc is selected in $F2$ (Figure 4(f)).

k-cell eliminated: bcd in $F2$.

K-cell selected: Abc in $F2$.

States covered: $[2,8]$, $[2,9]$

Cost: $6 + 4$

8. Free state: $S = [2,13]$

k-cells covering S: $AcD(9,13)$, $BcD(5,13)$, $ABcD(13)$.

No other free states covered by *k-cells* covering $[2,13]$. Therefore, S is a reference state.

Reference state: $S1 = [2,13]$

- For $K1 = AcD$, $K2 = BcD$ satisfies condition 1(a,b&c), since, BcD is of same cost as AcD and does not cover any extra free state not covered by AcD . Therefore, BcD is eliminated from all functions where it exists, i.e., $F2$.

Note that, if $K1 = BcD$, $K2 = AcD$ also satisfies condition 1(a,b&c), and so anyone of the *k-cells* BcD and AcD can be eliminated.

- For $K1 = AcD$, $K2 = ABcD$ violates condition 1c, as $K2'$ in $F3$ covers a free state $[3,13]$ not covered by $K1'$ in $F3$ ($K1'$ does not exist in $F3$). Hence, $ABcD$ cannot be eliminated.
- For $K1 = ABcD$, $K2 = AcD$ violates condition 1b, as AcD is of a lower cost than $ABcD$. Hence, AcD cannot be eliminated.

k-cell eliminated: BcD in $F2$

k-cell selected: None

States covered: None

Cost: $6 + 4$

9. Free state: $S = [3,7]$

k-cells covering S: $BCD(7,15)$, $aBCD(7)$.

The only other free state [3,15] has an alternate k-cell in ABD(13,15). Therefore, S is a reference state.

Reference state: S1 = [3,7]

- For K1 = BCD, K2 = aBCD violates condition 1c, as K2' in F2 covers a free state [2,7] not covered by K1' in F2 (K1' does not exist in F2). Hence, aBCD cannot be eliminated.
- For K1 = aBCD, K2 = BCD violates condition 1b, as BCD is of a lower cost than aBCD. Hence, AcD cannot be eliminated.

k-cell eliminated: None

k-cell selected: None

States covered: None

Cost: 6 + 4

10. *Free state:* S = [3,13]

k-cells covering S: ABD(13,15), ABcD(13).

The only other free state [3,15] has an alternate k-cell in BCD(7,15). Therefore, S is a reference state.

Reference state: S1 = [3,13]

- For K1 = ABD, K2 = ABcD violates condition 1c, as K2' in F2 covers a free state [2,13] not covered by K1' in F2 (K1' does not exist in F2). Hence, ABcD cannot be eliminated.
- For K1 = ABcD, K2 = ABD violates condition 1b, as ABD is of a lower cost than ABcD. Hence, ABD cannot be eliminated.

k-cell eliminated: None

k-cell selected: None

States covered: None

Cost: 6 + 4

11. *Free state:* S = [3,15]

k-cells covering S: ABD(7,15), BCD(13,15).

Other free state [3,15] has an alternate k-cell in aBCD(7) and [3,13] has an alternate k-cell in ABcD(13). Therefore, S is a reference state.

Reference state: S1 = [3,15]

- For K1 = ABD, K2 = BCD violates condition 1a, as BCD covers a free state [3,13] not covered by ABD. Hence, BCD cannot be eliminated.
- For K1 = BCD, K2 = ABD violates condition 1a, as ABD covers a free state [3,7] not covered by BCD. Hence, ABD cannot be eliminated.

k-cell eliminated: None

k-cell selected: None

States covered: None

Cost: 6 + 4

After checking out all the states, we list the states which are not yet covered:

[1,0], [2,7], [2,13], [3,7], [3,13], [3,15]

Since all the states are not covered, we scan each of the above states again to check if any of them can be covered now. This is necessary because a k-cell which could not be selected previously to cover a given state might now be selected due to some other states getting bound during selection of other k-cells.

1. *Free state:* S = [1,0]

k-cells covering S: acd(0,4), bcd(0,8)

Note that none of these were eliminated from F1, though they were eliminated from F2. After a k-cell is eliminated from a function, it will no longer be considered as a possible coverage for any free states in that function.

As no other free states are covered, S is still a reference state. Note that this has to be checked again because a state which was previously a free state, may no longer be one due to elimination of some k-cells during covering of some other states.

Reference state: S1 = [1,0]

- For K1 = acd, K2 = bcd satisfies condition 1(a,b&c), as now, since [2,8] has been bound, K2' does not cover any free state not covered by K1' in F1. Therefore, acd can now be selected for covering [1,0] in F1 (Figure 4(g)).

Note that if K1 = bcd, K2 = acd are considered, bcd also satisfies condition 1(a,b&c) as [2,0] is also bound. Hence in this case, whichever k-cell is selected out of acd & bcd, we get the minimal solution.

k-cell eliminated: bcd in F1

k-cell selected: acd in F1

States covered: [1,0]

Cost: 9 + 5

We can similarly check the rest of the not yet covered free states and find that none of them can be covered still. We can apply the branching rule only after scanning all free states for possible elimination of any more k-cells and still cannot eliminate any. This being the case here, we apply the branching rule, Rule 3.

Rule 3:

Reference state for branching: [2,7]

We could have selected any of the uncovered states since all of them have two alternate coverage.

Alternate k-cells possible: aBD(5,7), aBCD(7)

Branch 1: (Figure 4(h)).

k-cell selected: aBD in F2

States covered: [2,7]

Cost: 12 + 6

We now apply Rule 1 & 2 again for the rest of the uncovered states.

Rule 1-2:

We see that [3,13] can still not be covered as none of the two k-cells ABD and ABcD can still be eliminated. Therefore, we go to the next state.

1. *Free state:* S = [3,7]

k-cells covering S: BCD(7,15), aBCD(7).

S is still a reference state.

Reference state: S1 = [3,7]

- For K1 = BCD, K2 = aBCD satisfies condition 1(a,b&c), as K2' in F2 no longer covers a free state not covered by K1' in F2, as state [2,7] is now bound. Hence, k-cell aBCD can now be eliminated.

BCD being the only k-cell, it is selected in F3 (Figure 4(i)). Note again that BCD cannot be the only k-cell in F3 as [3,13] is not covered by BCD.

k-cell eliminated: aBCD in F3

k-cell selected: BCD in F3

States covered: [3,7], [3,15]

Cost: 15 + 7

Again, we see that for both uncovered states [3,13] and [2,13], none of the k-cells can be selected. It can be seen that, in both cases, the lower cost k-cell does not satisfy condition 1c due to the presence of a corresponding free state in another function.

Hence we branch again as before.

Rule 3: (Figure 4(j)).

Reference state for branching: [2,13]

Again, state [3,13] would be equally correct as both have two alternate coverage.

Alternate k-cells possible: AcD(9,13), ABcD(13)

k-cell selected: AcD in F2

States covered: [2,13]

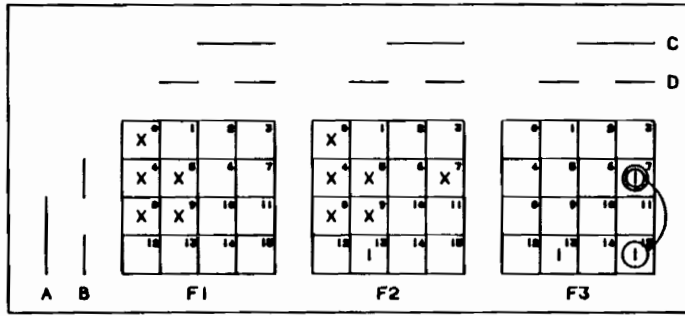
Cost: 18 + 8

Applying Rule 1 & 2 again for the remaining free state [3,13]:

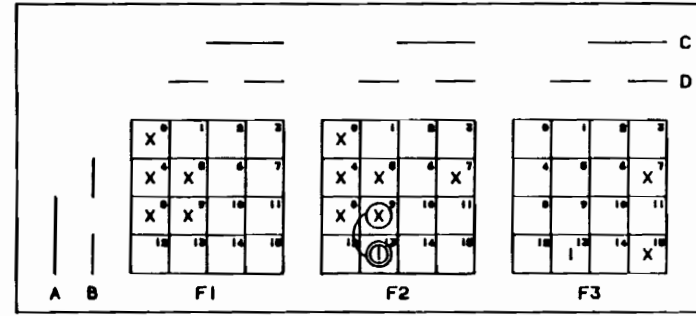
Rule 1-2:

1. *Free state:* S = [3,13]

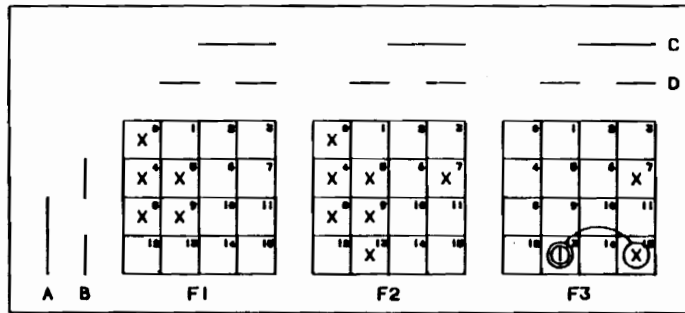
k-cells covering S: ABD(13,15), ABcD(13)



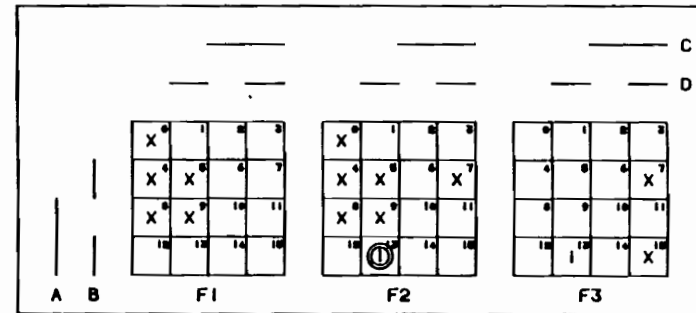
(i) K-cell BCD selected for ref. state [3,7].



(j) K-cell AcD selected for bra. state [2,13].



(k) K-cell ABD selected for ref. state [3,13].



(l) K-cell ABcD selected for bra. state [2,13] for Branch 2.

Figure 4. Application of the proposed algorithm for an example using the logical matrix.

S is a reference state.

Reference state: $S_1 = [3,13]$

- For $K_1 = ABD$, $K_2 = ABcD$ condition 1(a,b&c) satisfied, as state $[2,13]$ is now bound. Thus, k-cell $ABcD$ can be eliminated.

ABD being the only k-cell, by Rule 1, ABD is selected in F_3 (Figure 4(k)).

k-cell eliminated: $ABcD$

k-cell selected: ABD

states covered: $[3,13]$

Cost: $21 + 9$

We can check now that all states are covered and so branch 1 terminates here.

Cost of branch 1: 30

Branch 2: (Figure 4(l)).

Reference state for branching: $[2,13]$

k-cell selected: $ABcD$ in F_2

States covered: $[2,13]$

Cost: $19 + 8$

Applying Rule 1 & 2 again for the remaining free state $[3,13]$:

Rule 1-2:

1. *Free state:* $S = [3,13]$

k-cells covering S: $ABD(13,15)$, $ABcD(13)$

S is a reference state.

Reference state: $S_1 = [3,13]$

- For $K_1 = ABcD$, $K_2 = ABD$ condition 2(a&b) satisfied, as $ABcD$ is selected in F_2 and ABD does not cover an extra free state. So, ABD is eliminated.

$ABcD$ being the only k-cell, by Rule 1, $ABcD$ is selected in F_3 (Figure 4(m)). Again, $ABcD$ cannot be the only k-cell in F_3 .

k-cell eliminated: ABD

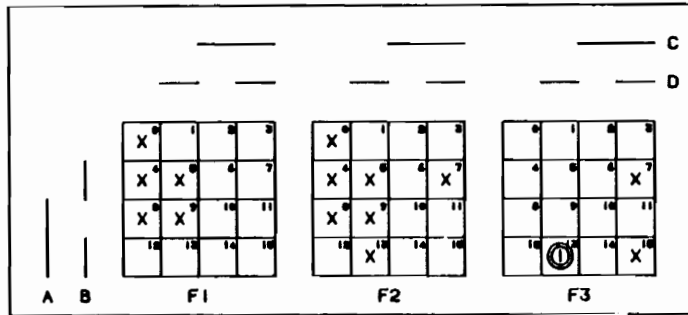
k-cell selected: $ABcD$

states covered: $[3,13]$

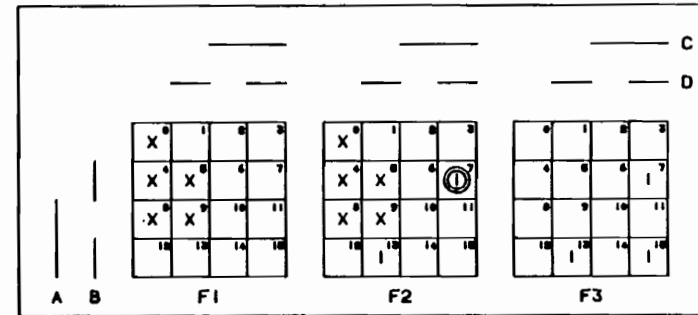
Cost: $19 + 9$

All states are covered and so branch 2 terminates here.

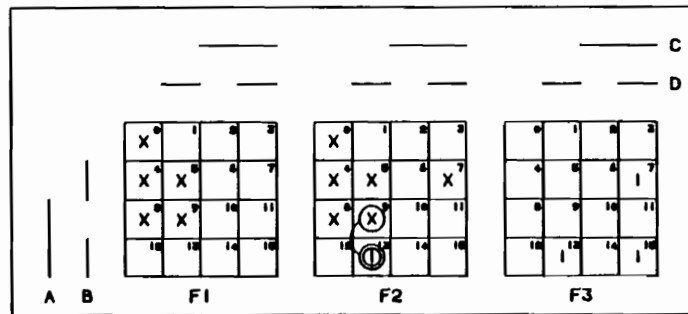
Cost of branch 2: 28



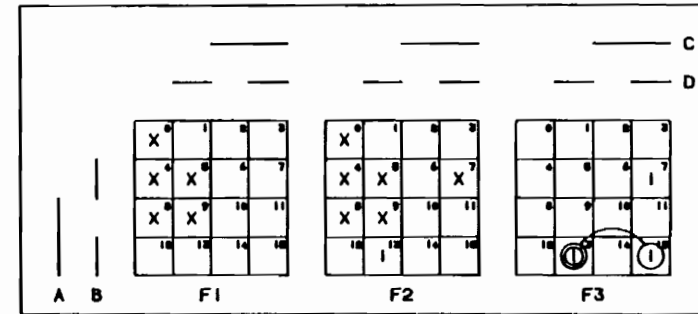
(m) K-cell ABCD selected for ref. state [3,13].



(n) K-cell aBCD selected for bra. state [2,7] for Branch 3.



(o) K-cell AcD selected for bra. state [2,13].



(p) K-cell ABD selected for ref. state [3,13].

Figure 4. Application of the proposed algorithm for an example using the logical matrix.

Branch 3: (Figure 4(n)).

Reference state for branching: [2,7]

k-cell selected: aBCD in F2

States covered: [2,7]

Cost: 13 + 6

Again, applying Rule 1 & 2 for the rest of the uncovered states.

Rule 1-2:

We see that [3,13] cannot be covered in this case also, and so we go to the next state.

For state [3,7], in this case, for $K1 = aBCD$, $K2 = BCD$ satisfies condition 2a but not 2b, as aBCD is selected already in F2 (2a) but state [3,15] is free and is covered by BCD (2b). Hence, k-cell BCD cannot be eliminated. Similarly, as aBCD is of lower cost it cannot be eliminated either.

Again, we see that for both other uncovered states [3,13] and [2,13], none of the k-cells can be selected for the same reason as before.

Hence we branch again.

Rule 3: (Figure 4(o)).

Reference state for branching: [2,13]

Alternate k-cells possible: AcD(9,13), ABcD(13)

k-cell selected: AcD in F2

States covered: [2,13]

Cost: 16 + 7

Applying Rule 1 & 2 again for the remaining free states [3,13]: [3,15] and [3,7];

Rule 1-2:

1. *Free state:* S = [3,13]

k-cells covering S: ABD(13,15), ABcD(13)

S is a reference state.

Reference state: S1 = [3,13]

- For $K1 = ABD$, $K2 = ABcD$ condition 1(a,b&c) satisfied, as state [2,13] is now bound. Thus, k-cell ABcD can be eliminated.

ABD being the only k-cell, ABD is selected in F3 (Figure 4(p)). Also, ABD cannot be the only k-cell in F3.

k-cell eliminated: ABcD in F3

k-cell selected: ABD in F3

states covered: [3,13], [3,15]

Cost: $19 + 8$

2. *Free state:* $S = [3,7]$

k-cells covering S: BCD(7,15), aBCD(7).

S is still a reference state.

Reference state: $S1 = [3,7]$

- For $K1 = aBCD$, $K2 = BCD$ satisfies condition 2(a&b), as $[3,15]$ is now bound. Hence, k-cell BcD can now be eliminated.

aBCD being the only k-cell, it is selected in F3 (Figure 4(q)).

k-cell eliminated: BCD in F3

k-cell selected: aBCD in F3

States covered: $[3,7]$

Cost: $19 + 9$

All states are covered again and so branch 3 terminates here.

Cost of branch 3: 28

Branch 4: (Figure 4(r)).

Reference state for branching: $[2,13]$

k-cell selected: ABcD in F2

States covered: $[2,13]$

Cost: $17 + 7$

Applying Rule 1 & 2 again for the remaining free states, we find that no more k-cells can be selected. For $[3,7]$, aBCD(7) does not satisfy condition 2b as BCD(7,15) covers a free state $[3,15]$ and BCD does not satisfy condition 1b as aBCD is of a lower cost. For $[3,13]$, aBCD(13) cannot be selected as ABD(13,15) violates condition 2b and ABD cannot be selected as aBCD violates condition 1b. Similarly for $[3,15]$, both ABD(7,15) and BCD(13,15) violates condition 1a as each covers a free state not covered by the other.

Hence we branch again.

Rule 3: (Figure 4(s)).

Reference state for branching: $[3,7]$

Alternate k-cells possible: BCD(7,15), aBCD(7)

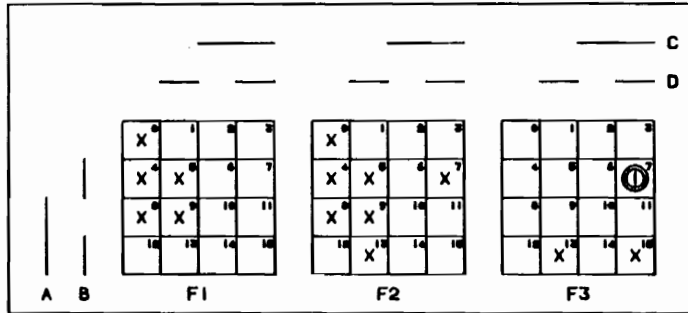
k-cell selected: BCD in F3

States covered: $[3,7], [3,15]$

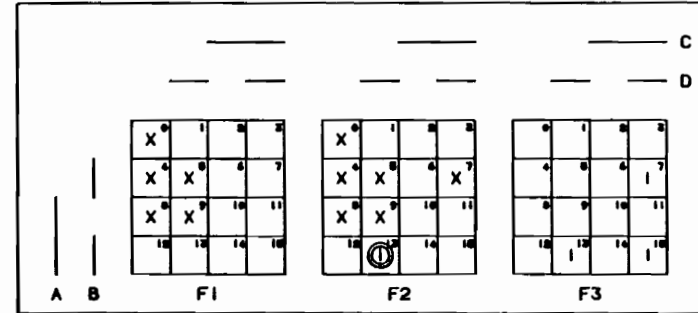
Cost: $20 + 8$

Here too, BCD cannot be the only k-cell in F3, so OR-cost is included.

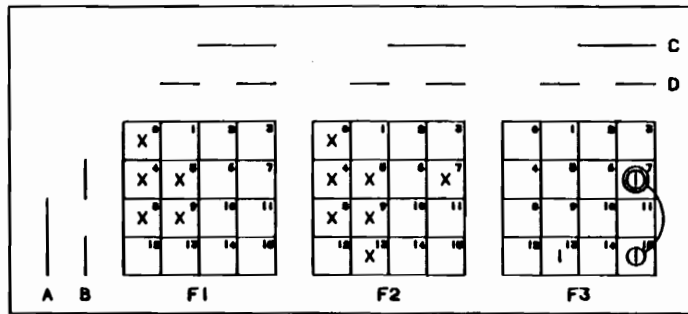
Applying Rule 1 & 2 again for the remaining free state $[3,13]$;



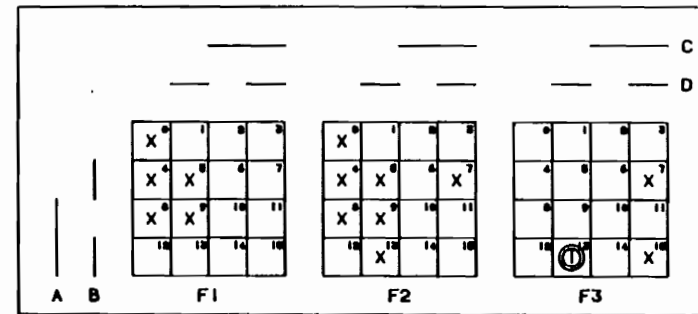
(q) K-cell aBCD selected for ref. state [3,7].



(r) K-cell ABCD selected for bra. state [2,13] for Branch 4.



(s) K-cell BCD selected for bra. state [3,7].



(t) K-cell ABCD selected for ref. state [3,13].

Figure 4. Application of the proposed algorithm for an example using the logical matrix.

Rule 1-2:

1. *Free state:* $S = [3,13]$

k-cells covering S: ABD(13,15), ABcD(13)

S is a reference state.

Reference state: $S1 = [3,13]$

- For $K1 = ABcD$, $K2 = ABD$ condition 2(a&b) satisfied as [3,15] is bound, and so ABD is eliminated.

ABcD being the only k-cell, ABcD is selected in F3 (Figure 4(t)).

k-cell eliminated: ABD in F3

k-cell selected: ABcD in F3

states covered: [3,13]

Cost: $20 + 9$

All states are covered and so branch 4 terminates here.

Cost of branch 4: 29

Branch 5: (Figure 4(u)).

Reference state for branching: [3,7]

k-cell selected: aBCD in F3

States covered: [3,7]

Cost: $17 + 8$

Again we check that aBCD cannot be the only k-cell in F3. Recall that this has to be checked in each branch when first k-cell selected for each function.

Applying Rule 1 & 2 again for the remaining free states [3,13] and [3,15];

Rule 1-2:

1. *Free state:* $S = [3,15]$

k-cells covering S: ABD(13,15), BCD(7,15)

S is a reference state.

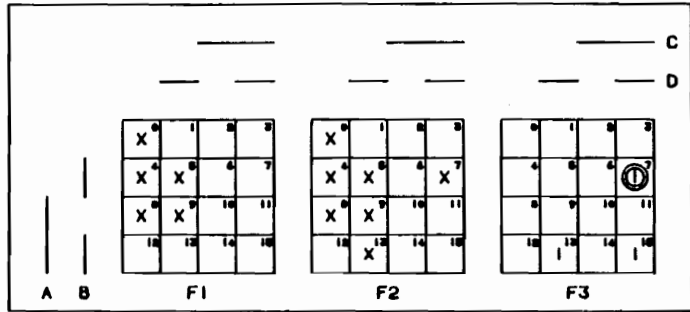
Reference state: $S1 = [3,15]$

- For $K1 = ABD$, $K2 = BCD$ condition 1(a,b&c) satisfied as [3,7] is bound, and so BCD is eliminated.

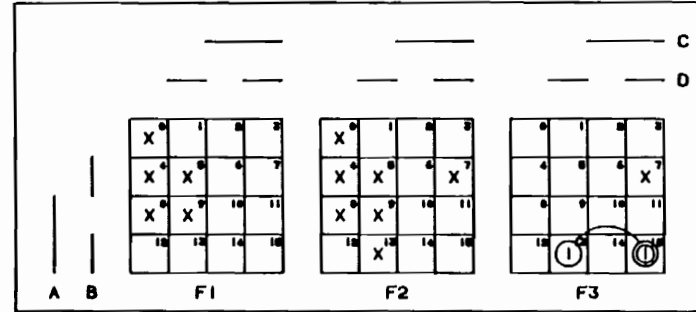
ABD being the only k-cell, ABD is selected in F3 (Figure 4(v)).

k-cell eliminated: BCD in F3

k-cell selected: ABD in F3



(u) K-cell aBCD selected for bra. state [3,7] for Branch 5.



(v) K-cell ABD selected for ref. state [3,15].

Figure 4. Application of the proposed algorithm for an example using the logical matrix.

states covered: [3,13], [3,15]

Cost: 20 + 9

All states are covered and so branch 4 terminates here.

Cost of branch 5: 29

Hence, finally comparing the costs of all five branches, cost of branches 2 & 3 are minimum (=28).

Hence, the minimal forms are:

1. $F1 = aBc + Abc + acd$
 $F2 = aBc + Abc + aBD + ABcD$
 $F3 = BCD + ABcD$
2. $F1 = aBc + Abc + acd$
 $F2 = aBc + Abc + aBCD + AcD$
 $F3 = ABD + aBCD$

The above example gives a fair idea of manual computation of multi-output minimization using the logical matrix. In the next chapter, we describe the computer implementation of this algorithm.

Chapter 4

Computer Implementation

4.1 Computer Algorithm

For the purpose of computer implementation of the proposed algorithm, we need to introduce some extensions. These are not modifications, but merely some convenience features introduced in the way the algorithm is applied.

When applying the algorithm rules in manual computations, we can easily notice that the process of obtaining maximal k-cells and the process of selecting them for the minimal form are essentially concurrent. This is because *we do not form all possible maximal k-cells covering all free states in each function* before we start selecting them. Instead, we *identify* maximal k-cells covering the given reference state only (in all the functions where it exists) and then look for possible selection of one of those k-cells to cover that reference state. The reason behind this approach is that as the free states are covered by selection of k-cells, we do

not have to form all possible maximal k-cells covering these bound states. Thus, this approach saves us computation required for formation of k-cells covering some of the free states.

One way to obtain all possible maximal k-cells covering a state is to *construct* the k-cells by comparing indices of each reference state with those of all the other free as well as bound states, as done in the conventional approach. Another way is to *generate* the k-cells for the given number of variables and the given set of states. The main difference between these two approaches is that the former one constructs all lower order k-cells on its way to construct the maximal higher-order k-cells, while the latter starts with the highest order k-cell possible in the given function and then goes on to generate only the maximal lower order k-cells. Also, in the former method, the k-cells are constructed from the set of states of the given function, while in the latter method, k-cells of a given order are generated first and then checked for its existence in the set of states of the given function. There are some obvious advantages of the generation approach over the conventional one, which we shall discuss in the next chapter. It is easier to appreciate this method for the purpose of computer implementation once the k-cell generation algorithm described below is investigated.

4.1.1 K-cell Generation

This constitutes the first stage of the minimization program, which generates all possible maximal k-cells covering each free state in each function. The following are the basic ideas behind this method.

- In the process of k-cell generation, it is clear that we need to generate only those k-cells that cover at least one free state. This reduces the scope of k-cell generation from total number of k-cells possible in an n-variable boolean function to a definite subset depending on the given function to be minimized. Also, we generate k-cells simultaneously for

all the functions, i.e., each generated k-cell is checked for all the functions. It can be seen that this would be most efficient for a problem with a lot of functions and a lot of possible maximal k-cells. Hence, the more complex a problem, the more efficient the k-cell generation becomes, for a given number of variables.

- We generate higher order k-cells first because if we start covering free states by higher order maximal k-cells, we would have to generate a smaller number of lower order k-cells. Note that we generate k-cells of a lower order only if all free states are not covered by the higher order k-cells generated so far. This means that we generate *all* k-cells of a given order and then, before going on to a next lower order, we check for each free state being covered by at least one k-cell generated so far. If each of them are covered, we stop any more k-cell generation. This prevents any non-maximal k-cells (subcells of an existing higher order k-cell) being generated to cover a free state which has been covered by a higher order k-cell.
- For the case of a multi-output function, we have to generate the *maximal subcells* also. Recall that these are the subcells of single output maximal k-cells of each functions if they are maximally shared in two or more functions. To accommodate this extension, we modify the definition of a free state being *covered* by a k-cell generated.

Maximally-covered states:

A free state is said to be maximally-covered by a given k-cell generated if that k-cell exists in all the functions where the given free state exists.

The idea is to keep generating lower order k-cells only till a k-cell is generated which exists in all the functions where a not-yet-covered free state exists, i.e., when a k-cell is generated which is maximally shared in **all** the functions. This will ensure generation of all maximal subcells which are maximally shared in any combinations of functions.

It can be shown that some non-maximal subcells may also be generated. The reason for this is that we stop generation of k-cells for a free state only when a k-cell generated is shared maximally in all functions with this free state. Because of this, we may end up generating a k-cell which is not shared in all of the functions and is also not maximally shared in the functions where it exists. An example of such a case is shown in Figure 5.

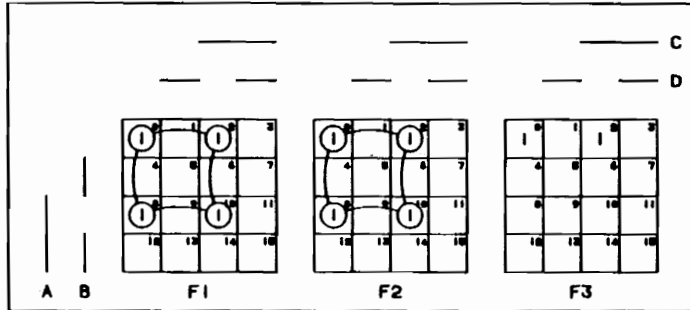
Figure 5(a) & (b) show the only two maximal k-cells possible for this multi-output function. Note that we stop generating any more k-cells after the k-cell abd is generated, which exists in all three functions. *But since we have to generate all k-cells of the same order as abd before we can stop*, two non-maximal k-cells bcd & bCd, as shown in Figure 5(c) & (d) are also generated.

So what we have here is a compromise between the two options. The first one is that of taking all possible intersection combinations of functions F_1, F_2, \dots, F_n and obtaining maximal k-cells for these $2^n - 1$ combinations¹², as done in the conventional method. The second option is to generate only the maximal k-cells for all the functions at a time and generate subcells of these maximal k-cells *only until* a k-cell of a given order is generated which is shared maximally in *all* the functions. We choose the second option for the obvious advantage of saving a lot of computation as well as for the fact that the overhead computation of generating a few non-maximal subcells is, in most cases, negligibly small compared to repeating the process $2^n - 1$ times.

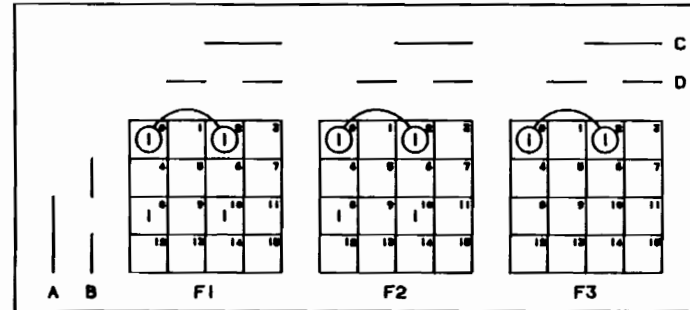
- It is obvious that we do not need to start with the highest order k-cells possible, i.e., of the order of $\log_2 n$ in the n-variable problem given, as for such a k-cell to exist, number of non-zero states required would be 2^n . We can determine the value of k (or the order of k-cell) for the highest order k-cell possible from the following relation:

$$2^{(k_m)} \leq T \leq 2^{(k_m + 1)}$$

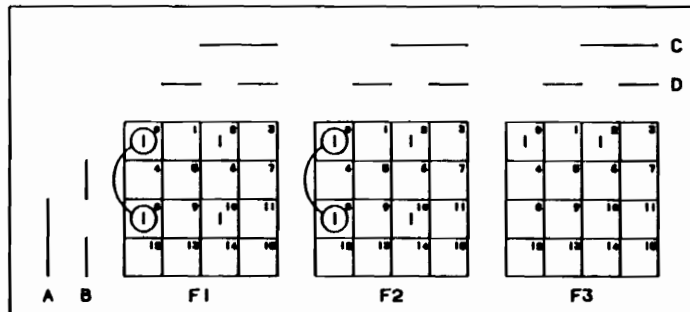
¹² Number of ways 1,2,.. or n functions can be selected for intersection equals $2^n - 1$.



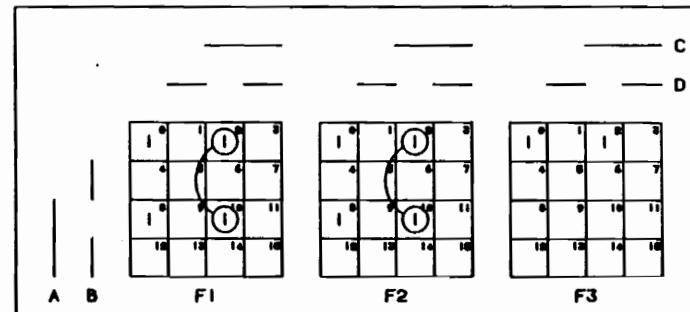
(a) Maximal k-cell bd of order 2 generated in F1 and F2.



(b) Maximal k-cell abd of order 1 generated in F1, F2 and F3.



(c) Non-maximal k-cell bcd of order 1 generated in F1 and F2.



(d) Non-maximal k-cell bCd of order 1 generated in F1 and F2.

Figure 5. An example of the possibility of non-maximal k-cells being generated.

where K_m is the maximum possible value of k and T is the total number of non-zero states in the given boolean function.

This simply means that if the number of non-zero states in the given function is not large enough to form a k -cell, such a k -cell cannot exist in the given function. Instead, we start with the highest order k_m -cell which can be accommodated within the total number of non-zero states T . The value of k_m can be easily computed from the relation¹³:

$$k_m = INT(\log_2 T)$$

The above relation prevents unnecessary higher order k -cells from being generated.

Note that in a multi-output function, since k -cells are generated for all the functions simultaneously, T in the above relation would be T_{max} , T_{max} being the maximum number of non-zero states in any function.

- As discussed in Chapter 2, we have the word-AND relation

$$k\text{-cell index} = g\text{-cell index} \text{ .AND. state index}$$

So we generate a generic cell (starting with the one of order k_m) and then form all possible k -cells by computing the corresponding k -cell index using the above relation for *only those states which are not yet maximally-covered*. We go on to the next lower order generic k -cell only if a free state is still not maximally-covered by any k -cells generated so far. Note that if we have to go to a next lower order to cover a set of free states, we have to generate all possible k -cells of that order for that set of free states only. After that, we again check if still any more free states are left to be covered, and if so, go to the next lower order of generic cells.

¹³ $INT(x)$ gives the truncated integer number for real x .

We shall discuss the actual logic of k-cell generation, based on the above points, later in this Chapter.

4.1.2 K-cell Selection

The manual and the computer computation of selection of k-cells from possible alternate covers is exactly the same as far as the application of rules 1,2 & 3 is concerned. Hence, the output of the k-cell generation module gives us possible coverages for each free state and we eliminate and select the k-cells out of these alternate covers by applying the rules 1 & 2. The only extension is while branching and it is discussed below.

Modified Branching Rule:

According to the branching rule (Rule 3) stated in Chapter 3, we continue to search all branches of the minimization tree until all free states are covered by selected k-cells in that branch. This means, as in the illustration in Chapter 3, that we may have to branch further in the search and then again trace all such offshoots of the original branch. In an NP-hard problem such as the general boolean minimization, in large and complex problems, we may end up with an astronomical number of branches and as a result computation time shoots up exponentially.

If we can specify a cut-off value for any branch in the tree, we may abort the search of a particular branch as soon as it exceeds this cut-off value. Since we are essentially looking for the branches with minimum cost, the cut-off value is obviously in terms of the cost of the branch at the instant of cut-off. If we are looking for a minimal implementation in a given programmable logic device with its device constraints or with any other hardware constraints in mind, the cut-off value is of course the constraint that is exceeded first. These constraints could be, for instance, maximum number of minterms (maximum number of k-cell selections per-

mitted) or the maximum size of any individual minterm (lowest order k-cell selection permitted) for a device such as a PLA. In an AND-OR or NAND-NAND implementation, we have the corresponding constraints of maximum number of gates available and the number of inputs in each gate.

But when we talk about the most general multi-output minimization problem requiring *all* possible minimal solutions, we have to trace at least one branch to termination, before we can obtain a cut-off value. Now, if this branch happens to be far from minimal and adjacent branches would most probably be nearer to this branch because of same roots, we might end up tracing a lot of branches unnecessarily before we can get to a reasonably close cut-off value.

To circumvent this uncertainty problem of the cut-off value, we suggest the *sum of single-output minimal cost* to be taken as the initial cut-off value. The single output minimal forms are the minimal forms of each function taken individually. Since sum of the costs of single-output minimal forms cannot be less than their multi-output implementation (due to more shared k-cells in the latter), this sum gives a reasonably close cut-off value to start with. As computation of single output minimum forms is usually much less complex, time taken by this computation is just a fraction of the total computation time. This may not be true for the manual computation, but we can hardly handle problems involving too many branches manually.

As mentioned earlier, this approach particularly becomes very efficient for the problems of very large size and complexity. Due to its reasonable closeness to multi-output solution most of the time, this initial cut-off value avoids a lot of branching further down in the search.

We can now state the modified branching rule, which incorporates the branch-and-bound method based on the initial cut-off value of single-output minimum cost sum.

Modified Rule 3:

If no k-cell can be selected after application of Rule 1 & 2 to all possible reference states, a reference state with least number of alternate k-cell coverage is selected, each of these alternate k-cells being selected one at a time and for each of these selection branches, Rule 1 & 2 are applied as before until all states are covered or *the cost of that branch exceeds the sum of single-output minimal form costs*.

The minimal form or forms are the completely searched branches with the least cost.

The branching rule and the subsequent tree searching is implemented using recursive co-routines. One of these applies Rule 1 & 2 to all free states until all are covered or no more can be covered. In the latter case it calls the other routine which selects each possible alternate k-cell one at a time and then in turn calls the first routine each time. Thus, these routines keep calling each other recursively until all states are covered in all branches. To now introduce the above modification, we keep track of the cost of the current branch and update it whenever a k-cell is selected. After each such update, we check if the cost exceeds the single-output cut-off value, and if it does, we terminate the search for this branch immediately and start from its most recent root into the next adjacent branch. This way, we need not store information about all the branches that are abandoned but store only the current branch (the branch being traced now) and the branches with minimum cost at this point. Thus, at the end, we would be left with only the branches which are minimal.

It can be easily appreciated that this technique saves a substantial amount of redundant computation, especially in a large and complex problem. We discuss the detailed logic of the k-cell selection and branching later in this chapter.

4.2 Program Specifications

The program objective is to accept a multi-output boolean function as input and compute its multi-output minimal cost expression(s), based on the proposed algorithm. We specify below the requirements for input and output as well as the program capabilities.

4.2.1 Input-Output Requirements

Input Format:

The program accepts a boolean function as a set of non-zero states. Thus, any multi-output boolean function can be accepted as a set of 1-states and d-states (don't cares) for each function. The program asks for the following basic input data:

1. Number of functions (x).
2. Number of variables (n).
3. Set of 1-states for each function (m_1, m_2, \dots, m_p).
4. Set of d-states for each function (d_1, d_2, \dots, d_q).

It should be noted that another form the function may be available in is the truth table. This can be easily converted into the required compact form, as each row of the truth table represents a state.

Output Format:

The primary output required is of course all possible minimal forms of the given multi-output function. These are given as sets of prime-implicants for each function, represented in terms of the input variables. In addition, it gives the costs of both single-output and multi-output

minimal forms for comparison. As an indication to the size of the branchings, it also gives the total number of terminal nodes of the minimization tree. Note that in this case terminal node may be either due to a completed search or due to an aborted search as discussed in the previous section.

4.2.2 Program Features

The following are the main features of the multi-output minimization program.

1. The program gives the most general solution of multi-output minimization problem. It accepts a multi-output boolean function with any number of functions of any number of variables,¹⁴ and computes *all* possible minimal forms for its two level logic implementation, based on the proposed algorithm.
2. The first module of the program computes all possible multi-output k-cell coverages for each free state in each function, by the method of k-cell generation as opposed to the method of forming of k-cells by extensive comparison. As discussed earlier, we do not generate redundant higher order and lower order k-cells, but generate only those k-cells which cover a free state *and* whose orders lie within the non-redundant range.
3. The second module then selects minimal k-cells out of the alternate coverages for each free state, by repeated application of rules 1, 2 & 3 until all free states are covered by at least one selected k-cell. Co-recursion¹⁵ is used to implement this repeated application of selection/elimination and branching; routine A for rules 1-2 and routine B for rule 3. The program essentially ends when these routines come out of the recursive loop.

¹⁴ With, of course, the computer system limitations on available memory and maximum computation time.

¹⁵ This is the case of A calling B and B calling A recursively.

4. As mentioned earlier, definition of cost-function is dependent on the device implementation. The proposed algorithm is independent of the cost-function specification for multi-output minimization. The program includes two cost-functions for which it computed the minimal forms of a given multi-output boolean function, viz., the gate-input count cost function and the PLA area cost function, both defined in Chapter 3.

The program output gives minimal forms for both of these cost definitions.

5. The program has the capability to accommodate any cut-off value specified for the purpose of trimming down the minimization tree. A branch is not searched further once its cost exceeds the the cut-off value. As mentioned earlier, this cut-off value could be implementation device dependent, or for the general problem, be the sum of single-output minimal costs.

This single-output minimal cost sum can be easily computed by calling routines A & B above for each individual function, treated each time as a multi-output function of only one function. Such a program structure avoids any extra code for computing this cut-off value.

6. Finally, the program has an added facility to compute minimal forms for all 2^x possible combinations of multi-output function $F(F_1, F_2, \dots, F_x)$, each combination corresponding to a subset of F being inverted¹⁶. The reason for such an option is that in most programmable logic devices and implementations, both non-inverted and inverted outputs are available for each individual function F_1, F_2, \dots, F_x . Hence, for the absolute minimal implementation of F , we can select a combination (out of possible 2^x), which gives *the minimal over all 2^x minimal form costs*.

¹⁶ The inverted form of function F_1 , denoted as F_1' , has 1-states and 0-states of F_1 interchanged, d-states being the same.

To avoid computation of minimal forms for all 2^x combinations, we can use a heuristic to compute single-output minimal forms for all $2x$ (non-inverted + inverted) functions ($F_1, F_1', F_2, F_2', \dots, F_x, F_x'$) and then compute the multi-output minimal forms only for the combination with least single-output-cost sum, i.e., the combination $\{F_{i_{\min}}\}$, $i = 1$ to x , where $F_{i_{\min}}$ is F_i or F_i' , whichever has the lower single-output cost.

It should be noted that, though it can be expected to obtain the minimum of all minimal forms in most cases, this is still a heuristic and it is possible that a combination with least single-output-cost sum may result in a multi-output minimal cost greater than that of another combination.

4.3 Program Logic

4.3.1 K-cell Generation

We have the following procedure for this first stage of the proposed algorithm.

Start

1. Set $Q = INT(\log_2 T)$, where Q = order of the k-cell & T = max number of non-zero states in a function.
2. Compute a generic cell¹⁷ of order Q and set R = generic k-cell index.

¹⁷ Refer to Chapter 2 for the details of generic k-cell generation routine.

3. Get a free state with index S belonging to any function which is not yet *maximally-covered* and compute the definite k-cell of index K from the word-AND relation: $K = R \text{ .AND. } S$
4. Check if this k-cell has been generated previously for another state.
If already generated, go to step 6.
5. Test if this k-cell generated exists in a function, i.e., if a function exists with all 2^Q states covered by this k-cell.
Store this k-cell if it exists in a function.
6. Take the next not-maximally-covered free state existing in a function and go to step 3 (until all free states are checked for this k-cell of indices [R,K]).
7. Go to step 2, i.e., compute the next generic k-cell of order Q (until all generic k-cells of order Q have been generated).
8. Make bound all the free states maximally-covered by any k-cells of order Q (so that no more k-cells of order $< Q$ will be generated for these maximally-covered free states).
9. Set $Q = Q - 1$, i.e., lower the order of generic cells to be generated for free states not yet maximally-covered, and go to step 2 (until all free states are maximally-covered).

Stop

4.3.2 K-cell Selection

For the purpose of computer implementation, we have put the application of rules 1,2 & 3 in the form of the following procedure.

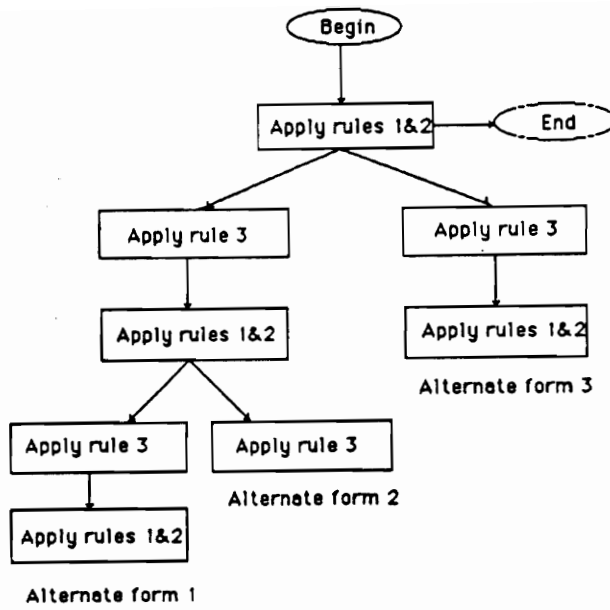
Start

1. Select a free state belonging to any function and check if it is a reference state, i.e., check that it does not dominate any other free states.
2. If it is not a reference state, make the state bound and go to step 7 (next free state).
3. Apply rule 1, i.e., check if there is more than one k-cell covering this reference state in its function.
If more than one k-cell, go to step 5.
4. (If no alternate k-cells)
 - a. select the only k-cell K1 covering the reference state in that function, i.e., store the selected k-cell and make bound all the free states covered by the k-cell in that function.
 - b. Check if after selection of K1, cost of the minimal form exceeds the cut-off value.
If it does, go to step 10 (check if any more branches).
else, go to step 7 (next free state).
5. Apply rule 2 to see if any alternate k-cell can be eliminated.
 - a. Select an alternate k-cell K2.
 - b. Check for conditions 1 & 2 for first K2 as the alternate k-cell and then K1 as the alternate k-cell.
 - c. If condition 1 or 2 satisfied, i.e., if either of K1 or K2 can be eliminated against the other, eliminate that k-cell as an alternate cover for the reference state.

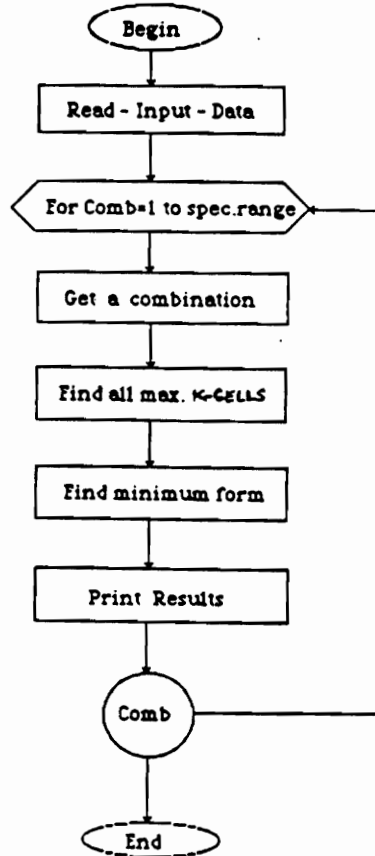
- d. Go to step a (until no more alternate k-cells available).
6. Go to step 3 (until no more k-cells can be eliminated).
7. Go to step 1 (until no more free states available).
8. Check if any k-cells selected during the last scan of all the free states.
If so, go to step 1 (until no k-cells selected in a complete scan of all free states).
9. Check if any free states left, still to be covered.
If so, go to step 11 (i.e., branch).
10. (if all states covered)
Check if any more k-cells, covering the branching state, still to be selected (i.e., any more branches left to be searched).
If so, go to step 12 (next branch), else go to step 13 (search ends).
11. Apply Rule 3, i.e., select a reference state with minimum number of alternate coverage.
12. Select a k-cell covering the branching reference state.
go to step 1 (until all k-cells covering the branching state are selected one by one).
13. Select only the branches with the minimum cost as the minimal forms. The minimum cost is of course the branch searched completely most recently, as all others would have been aborted.

Stop

The flowcharts showing the program structure and major procedures are given in Figure 6.

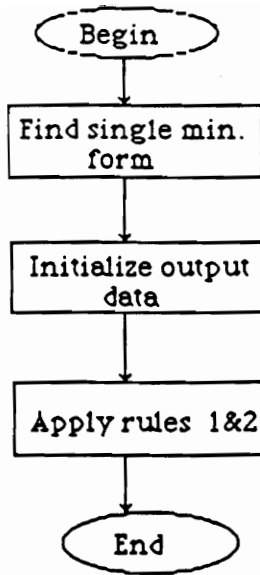


(a) Recursive tree structure of the program.

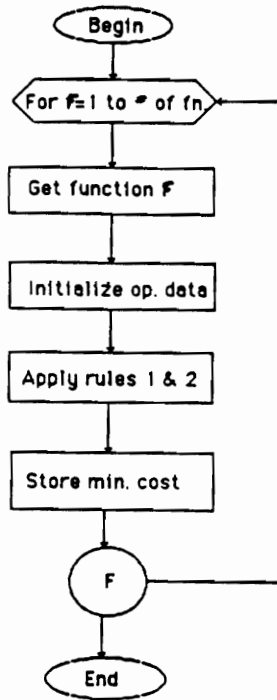


(b) Program Multimin

Figure 6. Flowcharts showing program structure and major procedures.

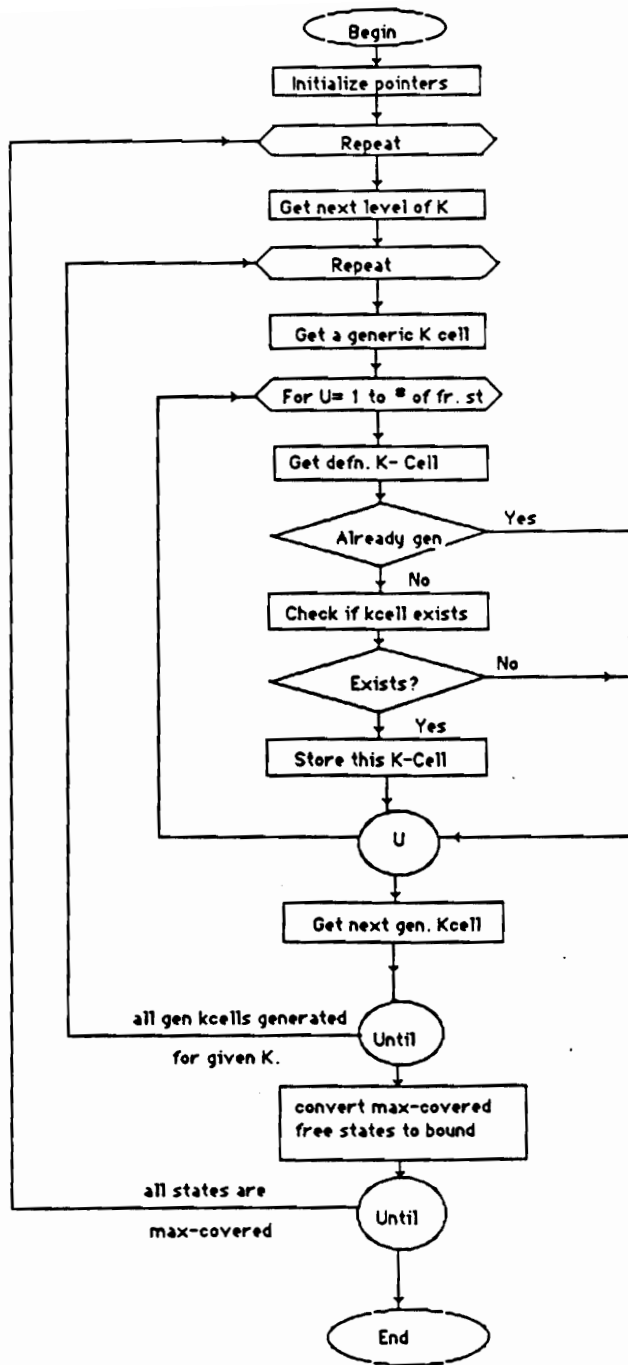


(c) Procedure Find-Minimum-Form



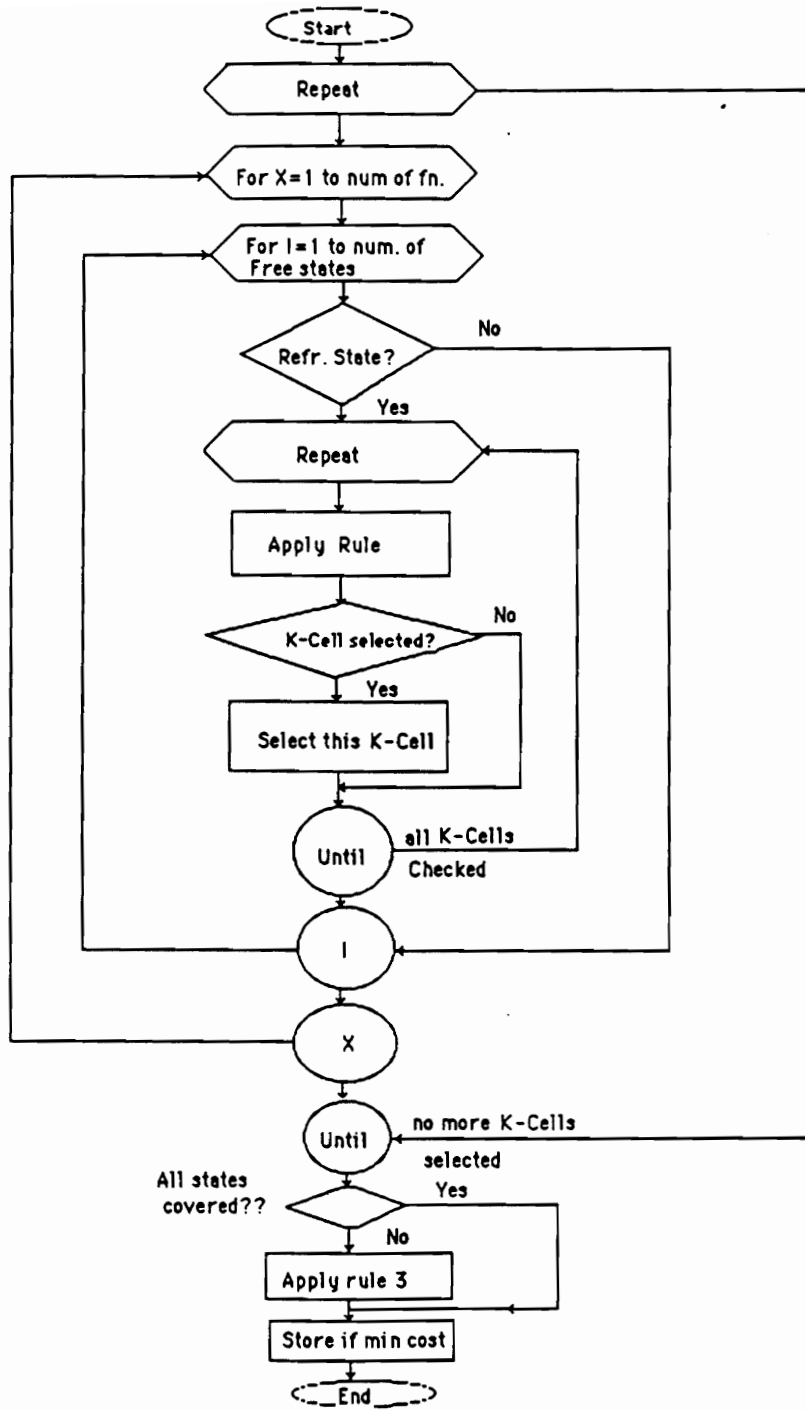
(d) Procedure Find-Single-Min-Form

Figure 6. Flowcharts showing program structure and major procedures.



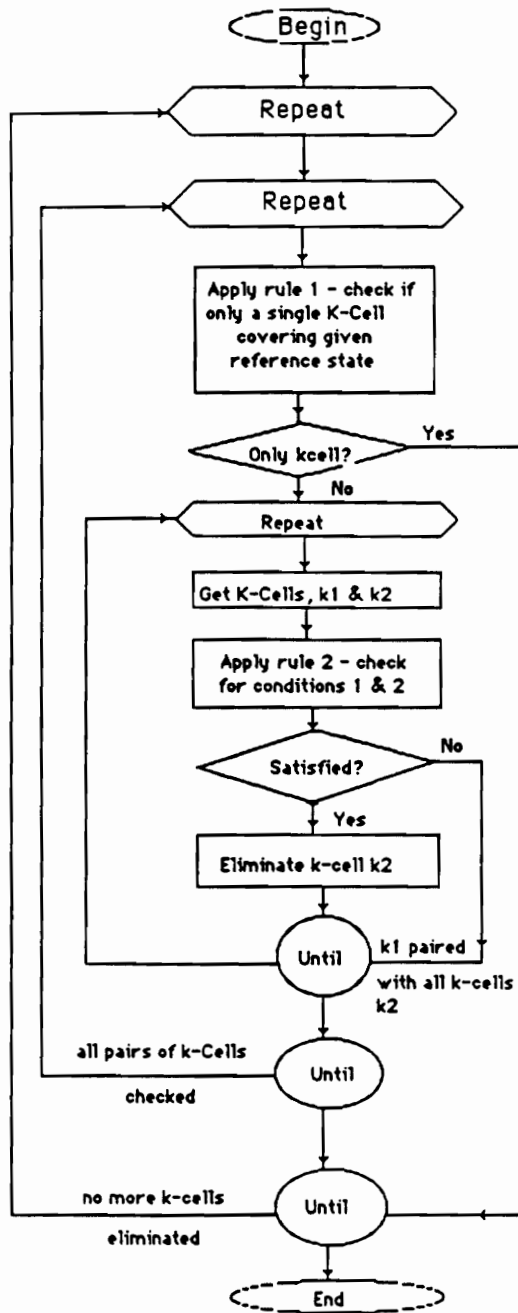
(e) Procedure Find-all-maximal-k-cells

Figure 6. Flowcharts showing program structure and major procedures.



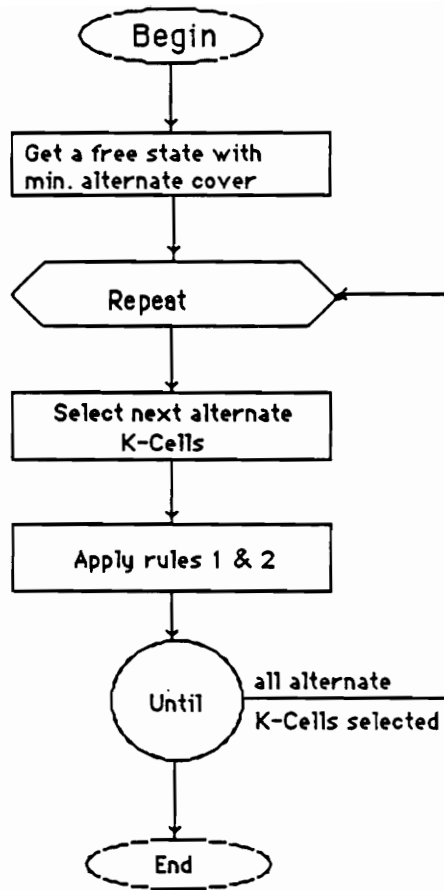
(f) Procedure Applyrules1-2

Figure 6. Flowcharts showing program structure and major procedures.



(h) Procedure Applyrule

Figure 6. Flowcharts showing program structure and major procedures.



(g) Procedure Applyrule3

Figure 6. Flowcharts showing program structure and major procedures.

4.4 Revised Algorithm

In the k-cell generation procedure described in the previous section, by starting with the highest order k-cell possible, we eliminate generation of non-maximal lower order k-cells by making bound all the free states maximally-covered after every level of k-cell generation. Thus in this algorithm, the stages of k-cell generation and k-cell selection are not completely independent, as in the conventional methods. In fact, the process of converting a set of maximally-covered free states bound is equivalent to declaring that all possible maximal k-cells for these free states have been generated and so no more lower order k-cells should be generated for these states. In other words, we could apply rules 1 & 2 to these free states to see if a k-cell could be selected in the minimal form of the function. Hence, we have the possibility of constructing a minimal form simultaneously with the process of k-cell generation.

So we see that although we have some concurrency in the two stages of the algorithm presented above, this concurrency is limited only to the conversion to bound status for the free states maximally-covered after each level of k-cell generation, as opposed to actually obtaining the minimal form. The reason for the presence of concurrency is that after all potentially selectable k-cells of the given level are generated, we make all maximally-covered free states bound; and so even though we don't know which of the k-cells are going to be selected eventually, we know which of the free states are definitely going to be covered by these selected k-cells. So we go ahead and do not bother about the maximally-covered free states anymore, *until* we actually start selection of k-cells.

Let us now look at the possible advantages of making the algorithm truly concurrent in the sense that the construction of minimal forms and k-cell generation is carried out concurrently. Since maximal-covering includes only those free states for which a k-cell is selected in *all* the functions where the given free state exists, *we cannot yet make those free states bound which*

do not satisfy the above condition, but may actually be a part of a selectable k-cell. Thus, if we apply the selection and elimination rules after each level of generic k-cell generation, we can now convert to don't cares not only the maximally-covered free states, but also all the free states covered by any k-cells that could be selected at this point.

An important point here is that we can apply rules 1 & 2 only for those free states that are maximally-covered (i.e., those for which a k-cell is generated in all the functions with this free state). This is obvious since only maximally-covered states are guaranteed to have all possible maximal k-cells generated after each level of k-cell generation. It can be seen that this is an improvement over the computer algorithm which selects k-cells only after generation of all maximal k-cells, as now we avoid generation of redundant k-cells for even more free states than was possible previously. *Note that application of the basic algorithm proposed in Chapter 3 is not affected in any way by this revised procedure for computer implementation.*

It is easy to see that this revised computer algorithm is a **truly concurrent** minimization algorithm, as is the case with the manual implementation of the proposed algorithm. This means that we actually construct the minimal solution as we generate the prime implicants. So, after each level of generic k-cells generation, we *temporarily* cover those free states which are maximally-covered by k-cells generated so far and *permanently* cover (make them don't cares) those free states which are covered by the k-cells actually selected in the minimal form. Thus, the k-cells or the prime implicants are generated only until all the free states are covered in either of the two ways. After this, what remains is the application of Rules 1, 2 & 3 as before to permanently cover all those free states which were only temporarily covered by the generated k-cells not yet included in the minimal form.

Hence, we look for selection and elimination during the stage of k-cell generation, but do not branch until all possible k-cells are generated (i.e., all free states are at least temporarily covered). We branch only after the completion of all k-cell generation *and* if all free states

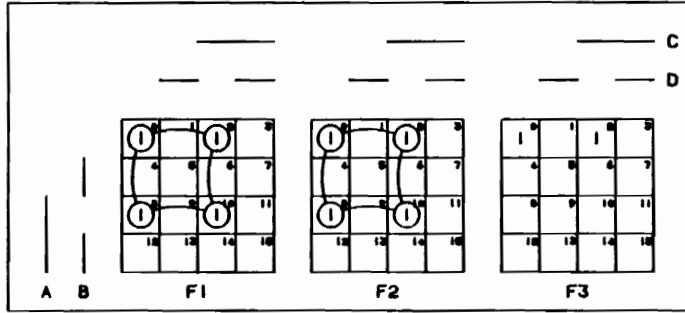
have not been permanently covered by k-cells selected with application of Rules 1 & 2 after each level of k-cell generation.

To illustrate this concept of concurrency and to show the obvious improvement in the revised algorithm, consider the example given in Figure 7 which is the same example with which we illustrated the possibility of non-maximal k-cell generation earlier in the chapter (Figure 5).

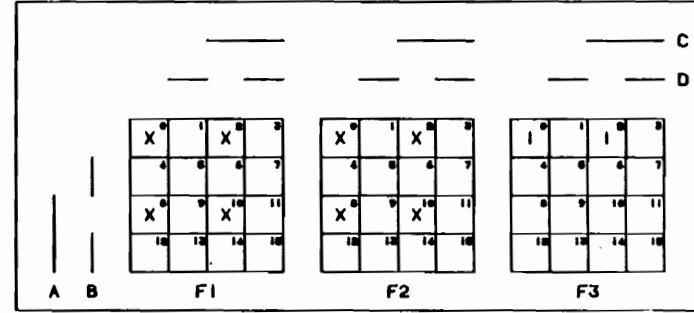
Figure 7(a) shows the first level (of order 2) of k-cell generation. At this point, we can temporarily cover only the maximally-covered free states 8 & 10 in both F1 & F2, as the k-cell bd is generated in both the functions where these free states exist. Had it not been for the revised algorithm, we would still be generating all next level (of order 1) k-cells, viz., abd, bcd & bCd, out of which bcd & bCd are non-maximal (Figure 5). But with the revised concurrent algorithm, when we apply rule 1 to states [1,8] (or [1,10]) and [2,8] (or [2,10]), we can see that the k-cell bd, being the only k-cell, can be selected in both F1 & F2.

Note that we can apply rule 1 & 2 only to the above four temporarily covered states and not to the remaining states 0 & 2 in F1, F2 & F3. Thus, immediately we can make all states covered by bd as bound or don't care as shown in the Figure 7(b). Note that this will automatically make states 0 & 2 in F1 & F2 bound, which we could not have accomplished without the application of rule 1 at this point. The next level of k-cell generation will generate only the k-cell abd in F3, as it is the only possible k-cell in F3 and it is now generated in all the functions where *free* states 0 & 2 exist, which is just F3, since states 0 & 2 in F1 & F2 are no longer free. It is clear that in large problems such an approach would eliminate a lot of redundant k-cell generation, as well as a lot of redundant applications of rules 2 for eliminating these redundant k-cells.

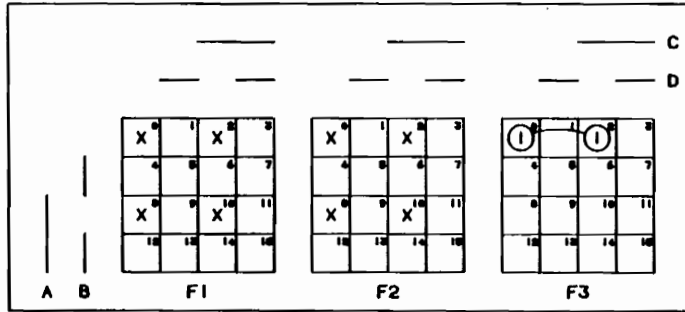
We present the concurrency logic of the revised algorithm described above in the form of the following procedure, which essentially combines the two separate procedures for k-cell generation and k-cell selection given in the previous section.



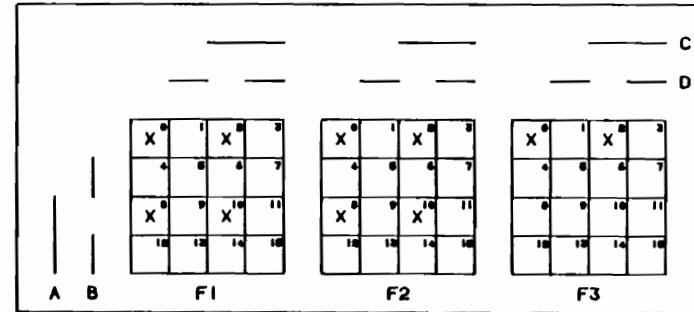
(a) K-cell bd generated in F1 and F2.



(b) Free states covered by bd are now bound.



(c) K-cell abd generated in F3 only.



(d) K-cell generation stopped as all free states bound.

Figure 7. An example illustrating the improvement in efficiency of the revised algorithm.

Start

1. Set $Q = INT(\log_2 T)$, where Q = order of the k-cell & T = max number of non-zero states in a function.
2. Compute a generic cell¹⁸ of order Q and set R = generic k-cell index.
3. Get a free state with index S belonging to any function which is not yet *covered* and compute the definite k-cell of index K from the word-AND relation: $K = R .AND. S$
4. Check if this k-cell has been generated previously for another state.
If already generated, go to step 6.
5. Test if this k-cell generated exists in a function, i.e., if a function exists with all 2^Q states covered by this k-cell.
Store this k-cell if it exists in a function.
6. Take the next not-covered free state existing in a function and go to step 3 (until all free states are checked for this k-cell of indices $[R,K]$).
7. Go to step 2, i.e., compute the next generic k-cell of order Q (until all generic k-cells of order Q have been generated).
8. Make bound (i.e., temporarily cover) all the free states covered, as defined earlier in this chapter, by any k-cells of order Q (so that no more k-cells of order $< Q$ will be generated for these covered free states).

¹⁸ Refer to Chapter 2 for the details of generic k-cell generation routine.

9. Select a temporarily bound free state (and not a free state not yet temporarily bound) belonging to any function and check if it is a reference state, i.e., check that it does not dominate any other free states.
10. If it is not a reference state, make the state permanently bound (convert to don't care) and go to step 15 (next free state).
11. Apply rule 1, i.e., check if there is more than one k-cell covering this reference state in its function.
If more than one k-cell, go to step 13.
12. (If no alternate k-cells)
 - a. select the only k-cell K1 covering the reference state in that function, i.e., store the selected k-cell and make permanently bound all the free states covered by the k-cell in that function.
 - b. Check if after selection of K1, cost of the minimal form exceeds the cut-off value.
If it does, go to step 20 (check if any more branches).
else, go to step 15 (next free state).
13. Apply rule 2 to see if any alternate k-cell can be eliminated.
 - a. Select an alternate k-cell K2.
 - b. Check for conditions 1 & 2 for first K2 as the alternate k-cell and then K1 as the alternate k-cell.
 - c. If condition 1 or 2 satisfied, i.e., if any one of K1 or K2 can be eliminated against the other, eliminate that k-cell as an alternate cover for the reference state.

- d. Go to step a (until no more alternate k-cells available).
14. Go to step 11 (until no more k-cells can be eliminated).
 15. Go to step 9 (until no more free states available).
 16. Check if any k-cells selected during the last scan of all the free states.
If so, go to step 9 (until no k-cells selected in a complete scan of all free states).
 17. Check if any free states left, still to be permanently bound.
If not, go to step 20.
 18. Set $Q = Q - 1$, i.e., lower the order of generic cells to be generated for not-yet-covered free states, and go to step 2 (until all free states are bound, permanently or temporarily).
 19. Go to step 21.
 20. (if all states covered)
Check if any more k-cells, covering the branching state, still to be selected (i.e., any more branches left to be searched).
If so, go to step 12 (next branch), else go to step 23 (search ends).
 21. Apply Rule 3, i.e., select a reference state with minimum number of alternate coverage.
 22. Select a k-cell covering the branching reference state.
go to step 1 (until all k-cells covering the branching state are selected one by one).
 23. Select only the branches with the minimum cost as the minimal forms. The minimum cost is of course the branch searched completely most recently, as all others would have been aborted.

Stop

The flowchart for the revised algorithm program structure is given in Figure 8.

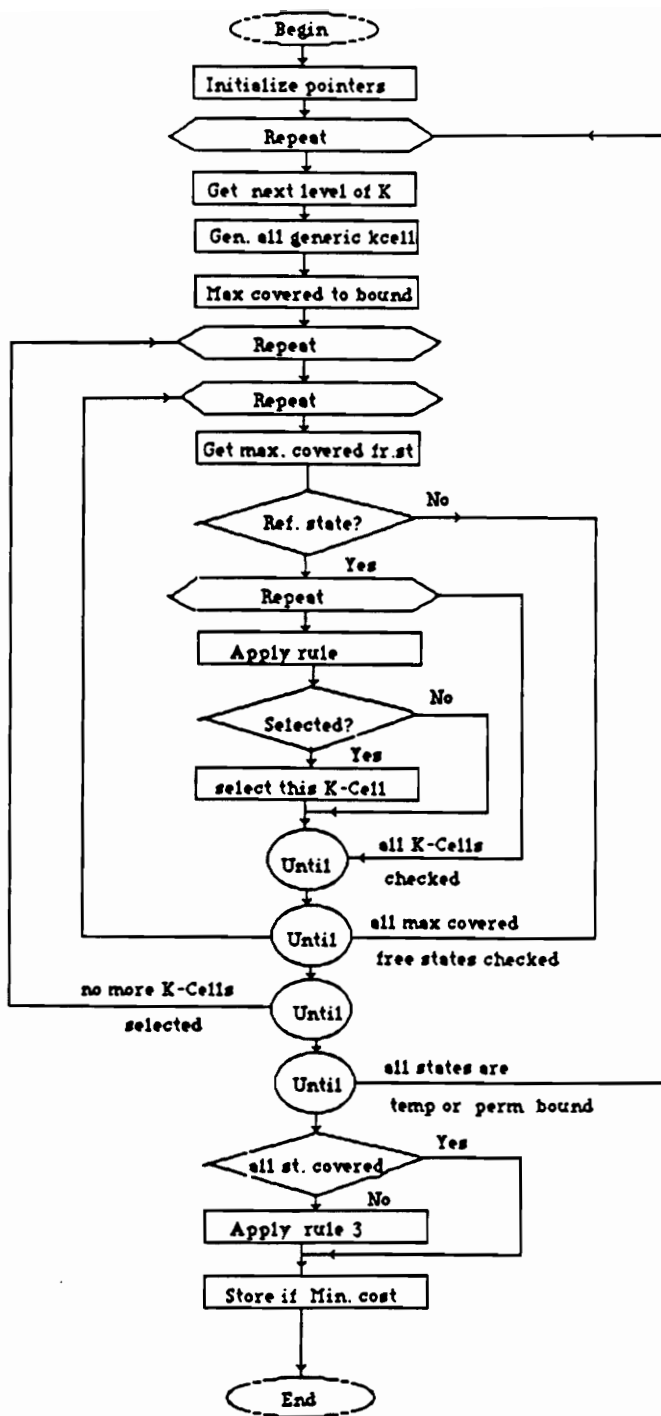


Figure 8. Flowchart of the revised algorithm combining k-cell generation and selection.

4.5 Sample Run

A sample run of the multi-output boolean minimization program is included here. The problem selected for the run is the same as the one illustrated for manual computation using the logical matrix in Chapter 3.

Sample run problem:

$$F(4) = (F1, F2, F3)$$

$$F1 = m(0,4,5,9) + d(8)$$

$$F2 = m(4,7,8,9) + d(0,5)$$

$$F3 = m(7,13,15)$$

This allows us a means of comparison of the manual and the computer implementation of the algorithm. To facilitate this, some additional information is also included in the program output (Sample Run A). Note that the program output gives minimal forms for both the gate-cost and the PLA-cost.

As an additional exercise, runs of this same problem for all eight combinations of non-inverted/inverted forms of the three functions are also included (Sample run B). For the gate-input cost function, it is interesting to note that the minimum cost of minimal forms of all combinations (=24) is for the case of all three functions inverted, where the single-output cost sum is also the minimum (=26). The same relationship is true for The PLA area cost function, where the combination of the first two functions inverted and the third one not inverted gives minimum cost of multi-output minimal forms of all combinations (=60), the single-output cost sum also being minimum (=90) for this combination.

4.5.1 Sample Run A

K-cell Generation

ALL POSSIBLE K-CELLS FOR EACH STATE :

STATE	TYPE	K-CELLS NOS	
FUNCTION 1			
0	1	1	4
4	1	4	8
5	1	8	
9	1	9	
8	d	1	9

FUNCTION 2				
4	1	4	8	
7	1	6	10	
8	1	1	9	
9	1	5	9	
13	1	2	5	11
0	d	1	4	
5	d	2	6	8

FUNCTION 3			
7	1	3	10
13	1	7	11
15	1	3	7

DECIMAL G-CELL AND K-CELL INDICES FOR EACH K-CELL :

K-CELL NO	G-CELL INDEX	K-CELL INDEX	K-CELL
1	7	0	bcd
2	7	5	BcD
3	7	7	BCD
4	11	0	acd
5	11	9	AcD
6	13	5	aBD
7	13	13	ABD
8	14	4	aBc
9	14	8	Abc
10	15	7	aBCD
11	15	13	ABcD

K-cell Selection

(a) Gate-input count cost function:

SINGLE-OUTPUT MINIMUM COST FOR EACH FUNCTION:

FUNCTION 1 : 12

FUNCTION 2 : 16

FUNCTION 3 : 8

SINGLE-OUTPUT MINIMUM COST SUM = 36

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 5

MULTI-OUTPUT MINIMUM COST = 28

MINIMAL FORM/S WITH MINIMUM COST :

FORM 2

FUNCTION 1 : $bcd + aBc + Abc$

FUNCTION 2 : $aBD + aBc + Abc + ABcD$

FUNCTION 3 : $BCD + ABcD$

FORM 3

FUNCTION 1 $bcd + aBc + Abc$

FUNCTION 2 $BcD + aBc + Abc + aBCD$

FUNCTION 3 $ABD + aBCD$

(b) PLA area cost function:

SINGLE-OUTPUT MINIMUM COST FOR EACH FUNCTION:

FUNCTION 1 33

FUNCTION 2 44

FUNCTION 3 22

SINGLE-OUTPUT MINIMUM COST SUM = 99

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 3

MULTI-OUTPUT MINIMUM COST = 66

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

FUNCTION 1 $bcd + aBc + Abc$

FUNCTION 2 $aBD + aBc + Abc + ABcD$

FUNCTION 3 $BCD + ABcD$

FORM 2

FUNCTION 1 $bcd + aBc + Abc$

FUNCTION 2 $aBc + Abc + aBCD + ABcD$

FUNCTION 3 $BCD + ABcD$

FORM 3

FUNCTION 1 $bcd + aBc + Abc$

FUNCTION 2 $aBc + Abc + aBCD + ABcD$

FUNCTION 3 $ABD + aBCD$

4.5.2 Sample Run B

SOLUTION FOR THE FOLLOWING COMBINATION :

FUNCTION 1 : NON-INVERTED

FUNCTION 2 : NON-INVERTED

FUNCTION 3 : NON-INVERTED

(a) Gate-input count cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 36

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 5

MULTI-OUTPUT MINIMUM COST = 28

MINIMAL FORM/S WITH MINIMUM COST :

FORM 2

FUNCTION 1 $bcd + aBc + Abc$

FUNCTION 2 $aBD + aBc + Abc + ABcD$

FUNCTION 3 $BCD + ABcD$

FORM 3

FUNCTION 1 $bcd + aBc + Abc$

FUNCTION 2 $BcD + aBc + Abc + aBCD$

FUNCTION 3 $ABD + aBCD$

(b) PLA area cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 99

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 3

MULTI-OUTPUT MINIMUM COST = 66

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

FUNCTION 1 $bcd + aBc + Abc$

FUNCTION 2 $aBD + aBc + Abc + ABcD$

FUNCTION 3 $BCD + ABcD$

FORM 2

FUNCTION 1 $bcd + aBc + Abc$

FUNCTION 2 $aBc + Abc + aBCD + ABcD$

FUNCTION 3 $BCD + ABcD$

FORM 3

FUNCTION 1 $bcd + aBc + Abc$

FUNCTION 2 $aBc + Abc + aBCD + ABcD$

FUNCTION 3 $ABD + aBCD$

SOLUTION FOR THE FOLLOWING COMBINATION :

FUNCTION 1 : INVERTED

FUNCTION 2 : NON-INVERTED

FUNCTION 3 : NON-INVERTED

(a) Gate-input count cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 32

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 5

MULTI-OUTPUT MINIMUM COST = 30

MINIMAL FORM/S WITH MINIMUM COST :

FORM 2

FUNCTION 1 $C + AB + abD$

FUNCTION 2 $acd + aBD + Abc + ABcD$

FUNCTION 3 $BCD + ABcD$

FORM 3

FUNCTION 1 $C + AB + abD$

FUNCTION 2 $BcD + acd + Abc + aBCD$

FUNCTION 3 $ABD + aBCD$

(b) PLA area cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 99

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 2

MULTI-OUTPUT MINIMUM COST = 80

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

FUNCTION 1 $C + AB + abD$

FUNCTION 2 $aBc + Abc + aBCD + ABcD$

FUNCTION 3 $BCD + ABcD$

FORM 2

FUNCTION 1 $C + AB + abD$

FUNCTION 2 $aBc + Abc + aBCD + ABcD$

FUNCTION 3 $ABD + aBCD$

SOLUTION FOR THE FOLLOWING COMBINATION :

FUNCTION 1 : NON-INVERTED

FUNCTION 2 : INVERTED

FUNCTION 3 : NON-INVERTED

(a) Gate-input count cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 33

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 1

MULTI-OUTPUT MINIMUM COST = 33

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

FUNCTION 1 $acd + aBc + Abc$

FUNCTION 2 $Cd + AC + ab + ABd$

FUNCTION 3 $BCD + ABD$

(a) PLA area cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 99

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 1

MULTI-OUTPUT MINIMUM COST = 99

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

FUNCTION 1 $acd + aBc + Abc$

FUNCTION 2 $Cd + AC + ab + ABd$

FUNCTION 3 $BCD + ABD$

SOLUTION FOR THE FOLLOWING COMBINATION :

FUNCTION 1 : INVERTED

FUNCTION 2 : INVERTED

FUNCTION 3 : NON-INVERTED

(a) Gate-input count cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 29

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 2

MULTI-OUTPUT MINIMUM COST = 26

MINIMAL FORM/S WITH MINIMUM COST :

FORM 2

FUNCTION 1 $C + abD + ABd + ABD$

FUNCTION 2 $Cd + AC + abD + ABd$

FUNCTION 3 $BCD + ABD$

(b) PLA area cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 90

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 6

MULTI-OUTPUT MINIMUM COST = 60

MINIMAL FORM/S WITH MINIMUM COST :

FORM 5

FUNCTION 1 $Cd + AC + BCD + abD + ABd + ABD$

FUNCTION 2 $Cd + AC + abD + ABd$

FUNCTION 3 $BCD + ABD$

SOLUTION FOR THE FOLLOWING COMBINATION :

FUNCTION 1 : NON-INVERTED

FUNCTION 2 : NON-INVERTED

FUNCTION 3 : INVERTED

(a) Gate-input count cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 33

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 1

MULTI-OUTPUT MINIMUM COST = 25

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

FUNCTION 1 $bcd + aBc + Abc$

FUNCTION 2 $BcD + aBD + aBc + Abc$

FUNCTION 3 $d + b + aBc$

(b) PLA area cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 100

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 1

MULTI-OUTPUT MINIMUM COST = 70

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

FUNCTION 1 $bcd + aBc + Abc$

FUNCTION 2 $BcD + aBD + aBc + Abc$

FUNCTION 3 $d + b + aBc$

SOLUTION FOR THE FOLLOWING COMBINATION :

FUNCTION 1 : INVERTED

FUNCTION 2 : NON-INVERTED

FUNCTION 3 : INVERTED

(a) Gate-input count cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 29

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 2

MULTI-OUTPUT MINIMUM COST = 27

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

FUNCTION 1 $C + AB + abD$

FUNCTION 2 $AcD + aBD + aBc + Abc$

FUNCTION 3 $d + b + aBc$

(b) PLA area cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 110

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 16

MULTI-OUTPUT MINIMUM COST = 99

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

FUNCTION 1 $C + AB + abD$

FUNCTION 2 $AcD + aBD + aBc + Abc$

FUNCTION 3 $d + b + aBc$

FORM 2

FUNCTION 1 $C + Cd + Ad + abD + ABcD$

FUNCTION 2 $aBc + Abc + aBCD + ABcD$

FUNCTION 3 $b + Cd + Ad + aBc$

FORM 3

FUNCTION 1 $C + Cd + Ad + abD + aBCD + ABcD$

FUNCTION 2 $aBc + Abc + aBCD + ABcD$

FUNCTION 3 $b + Cd + Ad + aBc$

FORM 4

FUNCTION 1 $Cd + bC + AB + abD + aBCD$

FUNCTION 2 $AcD + aBc + Abc + aBCD$

FUNCTION 3 $d + bC + abD + aBc + Abc$

SOLUTION FOR THE FOLLOWING COMBINATION :

FUNCTION 1 : NON-INVERTED

FUNCTION 2 : INVERTED

FUNCTION 3 : INVERTED

(b) Gate-input count cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 30

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 1

MULTI-OUTPUT MINIMUM COST = 28

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

FUNCTION 1 $bcd + aBc + Abc$

FUNCTION 2 $Cd + AC + ab + ABd$

FUNCTION 3 $d + b + aBc$

(b) PLA area cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 100

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 8

MULTI-OUTPUT MINIMUM COST = 80

MINIMAL FORM/S WITH MINIMUM COST :

FORM 2

FUNCTION 1 $bcd + aBc + Abc$

FUNCTION 2 $Cd + AC + ab + ABd$

FUNCTION 3 $b + Cd + ABd + aBc$

FORM 3

FUNCTION 1 $acd + aBc + Abc$

FUNCTION 2 $Cd + AC + ab + ABd$

FUNCTION 3 $b + Cd + ABd + aBc$

SOLUTION FOR THE FOLLOWING COMBINATION :

FUNCTION 1 : INVERTED

FUNCTION 2 : INVERTED

FUNCTION 3 : INVERTED

(a) Gate-input count cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 26

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 1

MULTI-OUTPUT MINIMUM COST = 24

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

FUNCTION 1 $C + AB + abD$

FUNCTION 2 $Cd + AC + abD + ABd$

FUNCTION 3 $d + b + ac$

(b) PLA area cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 110

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 2

MULTI-OUTPUT MINIMUM COST = 88

MINIMAL FORM/S WITH MINIMUM COST :

FORM 2

FUNCTION 1 $C + AB + abD$

FUNCTION 2 $Cd + AC + abD + ABd$

FUNCTION 3 $b + Cd + ac + ABd$

Chapter 5

Discussions and Results

5.1 Comparison with Quine-McCluskey's Algorithm

The proposed algorithm is based on a completely different approach from that of the classical approach of Quine-McCluskey's algorithm.¹⁹ We shall specify the differences for both k-cell generation and k-cell selection stages.

5.1.1 K-cell Generation

1. Unlike the QM method, *we do not generate all possible prime implicants* for the given function. QM method starts with a set of 0-cells (or individual states) and then by exhaustive comparison it constructs all possible higher order k-cells. Thus, effectively, all

¹⁹ Henceforth referred to as QM method.

possible k-cells (maximal as well as non-maximal) are generated in the process. Instead, we start from the highest order k-cell that could be accommodated within the given number of states and go towards lower order k-cells. This approach essentially eliminates the generation of any non-maximal subcells, as we generate k-cells covering only those states which are not maximally-covered by a higher order k-cell.

In fact, we make all the free states maximally-covered by the set of k-cells generated so far as bound after every level of k-cell generation, to prevent generation of any more k-cells for these bound states. It can be seen that a greater number of bound states would call for a smaller number of definite k-cells to be generated for a smaller number of free states present. Thus the presence of don't cares or bound states in a function and later making free states as bound at each level of k-cell generation, as described above, eliminates a lot of redundant computation. In QM method, of course, the don't cares are treated as any other states while generating the prime implicants, so presence of don't cares does not affect the computation process.

2. We introduced a truly concurrent procedure of k-cell generation and minimal form construction in the revised algorithm. This approach is in complete contrast to the most of the techniques of boolean minimization available, including the QM method, using two independent procedures for k-cell generation and k-cell selection. It has been shown [6] that the process of k-cell selection is of greater complexity and the deduction of minimal covering from the knowledge of all the PI's is practically impossible for functions having more than ten variables. The problem becomes all the more complex for the case of multi-output minimization.

Hence, we choose to construct the minimal forms concurrently with k-cell generation, where the information available at different levels of k-cell generation is used to select the k-cells, if possible; thus eliminating redundancy of unnecessary k-cell generation for those free states which we know are definitely going to be covered by an already generated

k-cell. Note that the proposed method of k-cell generation is closely related to the implementation of concurrency; the proposed algorithm for k-cell selection presented in Chapter 3 is completely independent of any implementation.

3. In case of the multi-output functions, QM method repeats the process of exhaustive comparison $2^n - 1$ times, i.e., for all possible intersection functions. Here instead, for any number of functions, the generation of k-cells is done simultaneously. This is made possible since we convert a generic k-cell into a definite k-cell for a free state belonging to any number of functions only once, for a given system of n variables. We do not have to repeat this process because of the practice of continuing to generate k-cells for a free state until the generated k-cell is selected in all the functions where that free state exists. This is taken care of by the definition of *maximally-covered free state* during the k-cell generation process, as discussed in Chapter 4.
4. An important computation we avoid here is that of ordering of the implicants as done in QM method. For a large problem this could become substantially comparable to the computation required to actually generate k-cells by comparison of these ordered sets of implicants. This is because of the fact that we generate a definite k-cell from a given generic k-cell for each free state available, and so ordering cannot affect the process.
5. This is not a tabular method of minimization, in the sense that we do not first tabulate the implicants to generate prime implicants and then tabulate the prime implicants against the free states to eliminate redundant prime implicants. As QM method has to store all intermediate k-cells in the process of prime implicants generation, it imposes large memory space requirements. In contrast, in the proposed algorithm, it is required to store the prime-implicants only, i.e., their indices, along with only the non-zero states of the function.

Note that the basic operation in QM method is comparison of two states for the 1-bit difference or of two columns or rows for dominance. The basic operation in the proposed algorithm, on the other hand, is the word-AND operation of two n-bit numbers, which is an instruction by itself in most machine level instruction sets. It is felt that an efficient assembly program of this algorithm would greatly enhance the speed of computation.

6. This approach is more efficient in the case of large problems. Since we generate all possible generic k-cells for a given level, the greater the number of these that exist in a function, the more efficient the process becomes. Thus for a given n variable problem, when the number of states is such that the number of possible maximal k-cells is greater, the greater is the possibility of a generated k-cell to exist. This is also helped if there are more functions, as with the same generation process the k-cells are generated for all the functions simultaneously. This is completely opposite in case of QM method, where it becomes more and more difficult to handle the problem, as the number of PI goes up.

5.1.2 K-cell Selection

1. **The most important and fundamental feature of this algorithm is that it is tighter than the QM method, i.e., with this algorithm we can eliminate a k-cell against another when the QM method cannot.** Note that this is in spite of the fact that *both* algorithms guarantee all possible minimal solutions of any given problem. This is essentially due to the concept of sharing of prime implicants in multi-output function implementation. We shall demonstrate this by the example in Table 6.

Table 6 shows the multi-output function $F = (F_1, F_2)$ represented on the Quine-McCluskey's table. The state [1,5] can be covered right away by the only k-cell covering it, viz., aBc, from Rule 1 of the proposed algorithm or essential prime implicant

Table 6. QM's cyclic table for an example to demonstrate the looseness of QM rules.

K-cells	F1 = m(0,4,5) + d(8)			F2 = m(4,8) + d(0,5)		Cost
	0	4 [✓]	5 [✓]	4	8 [✓]	
bcd(0,8)	X			-----X---		1
acd(0,4)	X	X		X		4
aBc(4,5)	-----X---		X---	X		1

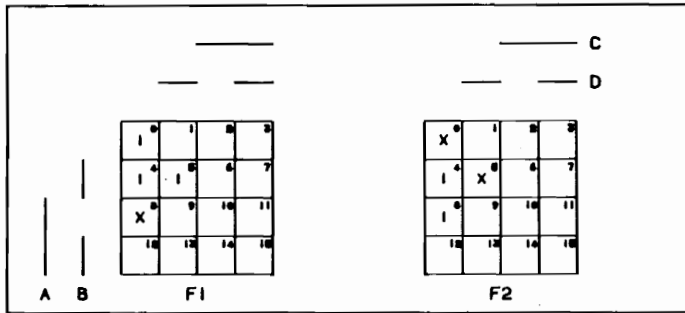
rule of QM method. Similarly, the state [2,8] can be covered by the only k-cell bcd. These two steps are shown in Table 6 by cancelling the row aBc in F1 and the row Abc in F2.

Inspect the QM table at this stage. Free state [1,0] is covered by k-cells acd & bcd, neither of which can be eliminated against the other from QM dominance rule, since acd dominates bcd but cost of bcd (=1) is less than that of acd (=4). Similarly, the state [2,4] is covered by acd & aBc, acd dominating aBc while aBc is of lower cost than that of acd. This is the case of cyclic table in QM method and so it calls for branching.

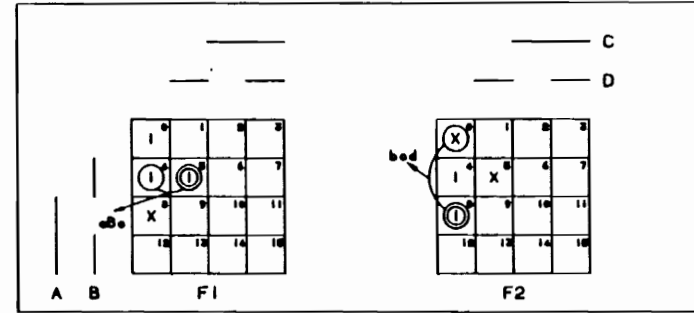
Now let us look at the problem when applying the proposed algorithm (Figure 9(a)). The first two steps of selection of k-cells aBc in F1 and Abc in F2, from Rule 1, are shown in Figure 9(b). For [1,0] as the reference state, k-cell bcd satisfies condition 2 (a&b) of Rule 2, as bcd is selected already in F2 and acd does not cover a free state not covered by bcd in F1 (Figure 9(c)). This allows us to select the k-cell bcd to cover the state [1,0]. The state [2,4] can now obviously be covered by aBc, as acd does not dominate aBc any more Figure 9(d). Note that the same arguments hold if we choose to select aBc first to cover [2,8] (by condition 2) and then select bcd to cover [1,0].

It should be recalled that the rationale behind this liberty to eliminate acd in F1 is that, to cover [1,0], the cost of any other k-cell than bcd would be equal (=1) or higher (condition 2 (a)). Also, bcd cannot be made redundant in F1 by acd unless acd is essential in F1 due to another free state covered by it in F1, which of course is not true (condition 2 (b)). This ensures that acd cannot be used in F1 to cover [1,0] as long as bcd is available.

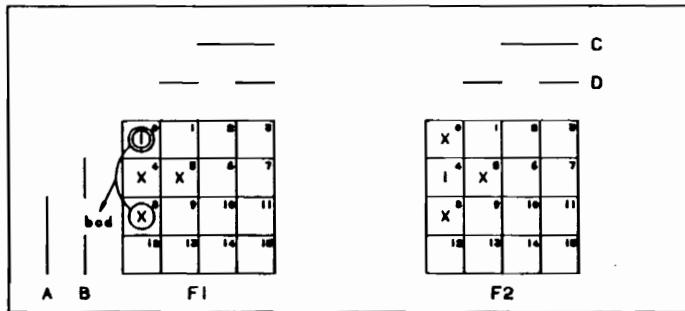
Thus, condition 2 in Rule 2 is an important clause which distinguishes this algorithm from that of QM method, as far as capability to eliminate a k-cell against another goes. It is not difficult to see that in complex problems, this feature could cut down substantially on branching. It provides a tool to further reduce the cyclic tables for multi-output minimization before we resort to branching.



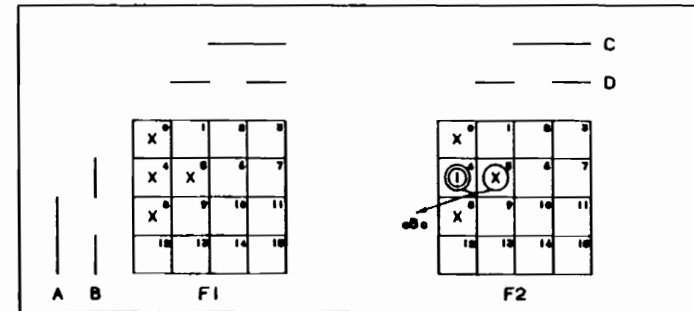
(a) The example shown on the logical matrix.



(b) K-cells aBc for state [1,5] and bcd for state [2,8] selected.



(c) K-cell bcd selected in F1 for state [1,0].



(d) K-cell aBc selected in F2 for state [2,4].

Figure 9. An example to demonstrate the improvement over the QM method.

2. Due to its tabular nature, in the case of cyclic tables, branching requires the QM tables to be copied for each branch. For multiple branchings, this may result in duplicating the tables many times before the branch is terminated. In the proposed algorithm, while branching, only the output, i.e., the solution information, need be copied for each branch, while the input information of non-zero states and prime implicants can be stored globally. A simple word-AND operation between the state and the cell indices would give the information that is stored in a QM table column, i.e., whether a given state belongs to a given k-cell.

3. In its manual implementation, with the use of the logical matrix, the proposed algorithm does not require listing of all prime implicants. This is mainly made possible by the fact that here the approach is to spot all maximal k-cells covering a given state and see if one of them could be selected to cover the given state. So the manual procedure is truly concurrent, in the sense that k-cell construction and selection is done simultaneously. Also, we do not have to form all possible intersection functions to find maximal subcells of individual function maximal k-cells. This is again taken care of by the definition of maximal subcells, discussed in Chapter 3. These features make it possible to use the proposed method for manual computation of multi-output minimal forms for problems with number of variables as large as the logical matrix could handle.

5.2 Solutions of Selected Problems

Problems 1 to 7 are taken from [15], while problem 8 is from [26].

5.2.1 Problem 1

$$F1 = m(2,4,10,11,12,13)$$

$$F2 = m(4,5,10,11,13)$$

$$F3 = m(1,2,3,10,11,12)$$

(a) Gate-input count cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 40

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 12

MULTI-OUTPUT MINIMUM COST = 34

MINIMAL FORM/S WITH MINIMUM COST :

FORM 4

$$\text{FUNCTION 1} \quad bCd + Bcd + AbC + ABcD$$

$$\text{FUNCTION 2} \quad aBc + AbC + ABcD$$

$$\text{FUNCTION 3} \quad bCd + abD + AbC + ABcd$$

FORM 5

$$\text{FUNCTION 1} \quad bCd + AbC + ABc + aBcd$$

$$\text{FUNCTION 2} \quad BcD + AbC + aBcd$$

$$\text{FUNCTION 3} \quad bCd + abD + AbC + ABcd$$

(b) PLA area cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 110

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 8

MULTI-OUTPUT MINIMUM COST = 77

MINIMAL FORM/S WITH MINIMUM COST :

FORM 2

FUNCTION 1 $bCd + Bcd + AbC + ABcD$

FUNCTION 2 $aBc + AbC + ABcD$

FUNCTION 3 $bCd + abD + AbC + ABcd$

FORM 3

FUNCTION 1 $bCd + AbC + ABc + aBcd$

FUNCTION 2 $BcD + AbC + aBcd$

FUNCTION 3 $bCd + abD + AbC + ABcd$

FORM 4

FUNCTION 1 $bCd + AbC + aBcd + ABcd + ABcD$

FUNCTION 2 $aBc + AbC + ABcD$

FUNCTION 3 $bCd + abD + AbC + ABcd$

FORM 5

FUNCTION 1 $bCd + AbC + aBcd + ABcd + ABcD$

FUNCTION 2 $BcD + AbC + aBcd$

FUNCTION 3 $bCd + abD + AbC + ABcd$

5.2.2 Problem 2

$$F1 = m(0,1,2,3,6,7,20,21,26,27,28)$$

$$F2 = m(0,1,6,7,14,15,16,17,19,20,24,27)$$

$$F3 = m(0,1,2,3,8,9,16,20,26,28,30)$$

(a) Gate-input count cost function:

$$\text{SINGLE-OUTPUT MINIMUM COST SUM} = 69$$

MULTIPLE OUTPUT MINIMIZATION :

$$\text{NO OF TERMINAL NODES IN THE SEARCH TREE} = 1$$

$$\text{MULTI-OUTPUT MINIMUM COST} = 58$$

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

$$\text{FUNCTION 1} \quad abD + abc + ACde + AbCd + ABcD$$

$$\text{FUNCTION 2} \quad bcd + aCD + Acde + AcDE + Abde$$

$$\text{FUNCTION 3} \quad acd + abc + ACde + Abde + ABDe$$

(b) PLA area cost function:

$$\text{SINGLE-OUTPUT MINIMUM COST SUM} = 195$$

MULTIPLE OUTPUT MINIMIZATION :

$$\text{NO OF TERMINAL NODES IN THE SEARCH TREE} = 1$$

$$\text{MULTI-OUTPUT MINIMUM COST} = 156$$

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

$$\text{FUNCTION 1} \quad abD + abc + ACde + AbCd + ABcD$$

$$\text{FUNCTION 2} \quad bcd + aCD + Acde + AcDE + Abde$$

$$\text{FUNCTION 3} \quad acd + abc + ACde + Abde + ABDe$$

5.2.3 Problem 3

F1 = m(0,1,2,8,9,10,13,16,17,18,19,24,25)

F2 = m(0,1,3,5,7,9,13,16,17,22,23,30,31)

F3 = m(2,3,8,9,10,11,13,15,16,17,18,19,22,23)

(a) Gate-input count cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 52

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 1

MULTI-OUTPUT MINIMUM COST = 46

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

FUNCTION 1 $cd + ace + Abc + aBdE$

FUNCTION 2 $abE + bcd + ACD + aBdE$

FUNCTION 3 $aBE + acD + AbD + aBc + Abc$

(b) PLA area cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 169

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 1

MULTI-OUTPUT MINIMUM COST = 143

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

FUNCTION 1 $cd + Abc + acDe + aBdE$

FUNCTION 2 $abE + bcd + ACD + aBdE$

FUNCTION 3 $aBE + acD + AbD + aBc + Abc$

5.2.4 Problem 4

$$F1 = m(0,2,9,10) + d(1,8,13)$$

$$F2 = m(1,3,5,13) + d(0,7,9)$$

$$F3 = m(2,8,10,11,13) + d(3,9,15)$$

(a) Gate-input count cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 21

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 8

MULTI-OUTPUT MINIMUM COST = 18

MINIMAL FORM/S WITH MINIMUM COST :

FORM 4

$$\text{FUNCTION 1} \quad bd + AcD$$

$$\text{FUNCTION 2} \quad aD + AcD$$

$$\text{FUNCTION 3} \quad bC + Ab + AcD$$

(b) PLA area cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 70

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 4

MULTI-OUTPUT MINIMUM COST = 50

MINIMAL FORM/S WITH MINIMUM COST :

FORM 2

$$\text{FUNCTION 1} \quad bd + AcD$$

$$\text{FUNCTION 2} \quad aD + AcD$$

$$\text{FUNCTION 3} \quad bC + Ab + AcD$$

FORM 3

FUNCTION 1 $bc + bCd$

FUNCTION 2 $aD + AcD$

FUNCTION 3 $Ab + bCd + AcD$

5.2.5 Problem 5

$$F1 = m(2,3,6,10) + d(8)$$

$$F2 = m(2,10,12,14) + d(6,8)$$

$$F3 = m(2,8,10,12) + d(0,14)$$

(a) Gate-input count cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 24

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 1

MULTI-OUTPUT MINIMUM COST = 18

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

$$\text{FUNCTION 1 } \quad bCd + aCd + abC$$

$$\text{FUNCTION 2 } \quad Ad + bCd$$

$$\text{FUNCTION 3 } \quad Ad + bCd$$

(b) PLA area cost function

SINGLE-OUTPUT MINIMUM COST SUM = 56

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 1

MULTI-OUTPUT MINIMUM COST = 32

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

$$\text{FUNCTION 1 } \quad bCd + aCd + abC$$

$$\text{FUNCTION 2 } \quad Ad + bCd$$

$$\text{FUNCTION 3 } \quad Ad + bCd$$

5.2.6 Problem 6

$$F1 = m(0,5,7,14,15) + d(1,6,9)$$

$$F2 = m(13,14,15) + d(1,6,9)$$

$$F3 = m(0,1,5,7,9) + d(13,14)$$

(a) Gate-input count cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 27

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 1

MULTI-OUTPUT MINIMUM COST = 19

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

FUNCTION 1 $aBD + abc + ABC$

FUNCTION 2 $ABD + ABC$

FUNCTION 3 $aBD + abc$

(b) PLA area cost function:

SINGLE-OUTPUT MINIMUM COST SUM = 70

MULTIPLE OUTPUT MINIMIZATION :

NO OF TERMINAL NODES IN THE SEARCH TREE = 1

MULTI-OUTPUT MINIMUM COST = 40

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

FUNCTION 1 $aBD + abc + ABC$

FUNCTION 2 $AcD + ABC$

FUNCTION 3 $aBD + abc$

5.2.7 Problem 7

$$F1 = m(2,8,10,12,18,26,28,30) + d(0,14,22,24)$$

$$F2 = m(2,3,6,10,18,24,26,27,29) + d(8,19,25,31)$$

$$F3 = m(1,3,5,13,1,6,18,25,26) + d(0,7,9,17,24,29)$$

(a) Gate-input count cost function:

$$\text{SINGLE-OUTPUT MINIMUM COST SUM} = 36$$

MULTIPLE OUTPUT MINIMIZATION :

$$\text{NO OF TERMINAL NODES IN THE SEARCH TREE} = 10$$

$$\text{MULTI-OUTPUT MINIMUM COST} = 36$$

MINIMAL FORM/S WITH MINIMUM COST :

FORM 1

$$\text{FUNCTION 1} \quad Be + cDe$$

$$\text{FUNCTION 2} \quad Bce + ABE + bcD + abDe$$

$$\text{FUNCTION 3} \quad BdE + Ace + abE$$

(b) PLA area cost function:

$$\text{SINGLE-OUTPUT MINIMUM COST SUM} = 108$$

MULTIPLE OUTPUT MINIMIZATION :

$$\text{NO OF TERMINAL NODES IN THE SEARCH TREE} = 10$$

$$\text{MULTI-OUTPUT MINIMUM COST} = 96$$

MINIMAL FORM/S WITH MINIMUM COST :

FORM 4

$$\text{FUNCTION 1} \quad Be + cDe$$

$$\text{FUNCTION 2} \quad cDe + ABc + abDe + ABdE + abcDE$$

$$\text{FUNCTION 3} \quad adE + Ace + ABdE + abcDE$$

FORM 5

FUNCTION 1 $Be + cDe$

FUNCTION 2 $cDe + ABE + abDe + ABcd + abcDE$

FUNCTION 3 $adE + Ace + ABcd + abcDE$

5.2.8 Problem 8

$$F1 = m(2,3,5,7,8,9,10,11,13,15)$$

$$F2 = m(2,3,5,6,7,10,11,14,15)$$

$$F3 = m(6,7,8,9,13,14,15)$$

(a) Gate-input count cost function:

$$\text{SINGLE-OUTPUT MINIMUM COST SUM} = 25$$

MULTIPLE OUTPUT MINIMIZATION :

$$\text{NO OF TERMINAL NODES IN THE SEARCH TREE} = 2$$

$$\text{MULTI-OUTPUT MINIMUM COST} = 22$$

MINIMAL FORM/S WITH MINIMUM COST :

FORM 2

$$\text{FUNCTION 1} \quad bC + aBD + ABD + Abc$$

$$\text{FUNCTION 2} \quad C + aBD$$

$$\text{FUNCTION 3} \quad BC + ABD + Abc$$

(b) PLA area cost function:

$$\text{SINGLE-OUTPUT MINIMUM COST SUM} = 80$$

MULTIPLE OUTPUT MINIMIZATION :

$$\text{NO OF TERMINAL NODES IN THE SEARCH TREE} = 4$$

$$\text{MULTI-OUTPUT MINIMUM COST} = 50$$

MINIMAL FORM/S WITH MINIMUM COST :

FORM 4

$$\text{FUNCTION 1} \quad bC + aBD + ABD + Abc$$

$$\text{FUNCTION 2} \quad bC + BC + aBD$$

$$\text{FUNCTION 3} \quad BC + ABD + Abc$$

5.3 Scope for Further Work

1. Efficiently implementing the revised algorithm with optimal memory space and computation time trade-offs.
2. Introducing some heuristics for near-minimal minimization based on the revised algorithm and comparing the performance with currently available minimization programs such as EXPRESSO-II.
3. Investigating and comparing complexity growth of the revised algorithm and specifying some quantitative results.
4. Adapting the minimization program for direct implementation on various PLD's with different cost functions.

Bibliography

1. Z. Arevalo and J. G. Bredeson, "A method to simplify a Boolean function into a near-minimal sum-of-products for PLA's," *IEEE Trans. Comput.*, vol C-27, no. 11, Nov. 1978, pp. 1028-1039.
2. T.C. Bartee, "Computer Design of Multiple-output logical networks," *IRE Trans. Electron. Comput.*, vol. EC-10, Mar. 1961, pp. 21-30.
3. N.N. Biswas, "Minimization of Boolean functions," *Bell Syst. Tech. J.*, vol C-20, Aug. 1971, pp. 925-929.
4. R.K. Brayton, G.D. Hachtel, C.T. McMullen and A.L. Sangiovanni-Vincentelli, "Logic Minimization Algorithms for VLSI Synthesis," Kluwer Academic Publishers, 1984.
5. J.G. Bredeson and P.T. Hulena, "Generation of prime implicants by direct multiplication," *IEEE Trans. Comput.*, vol C-20, Apr. 1971, pp. 475-476.
6. Y. Breitbart and K. Vairavan, "The computational complexity of a class of minimization algorithms for switching functions," *IEEE Trans. Comput.*, vol C-28, no 12, Dec. 1979, pp. 941-943.
7. D.W. Brown, "A State-Machine Synthesizer - SMS," *Proc. 18th Des. Auto. Conf.*, pp. 301-304, Nashville, June 1981.
8. V. Bubenik, "Weighting method for the determination of the irredundant set of prime implicants," *IEEE Trans. Comput.*, vol C-21, pp. 1449-1451, Dec. 1972.
9. G. Caruso, "A local selection algorithm for switching function minimization," *IEEE Trans. Comput.*, vol C-33, no 1, Jan. 1984, pp. 91-97.
10. A.K. Chandra and G. Markowsky, "On the number of prime implicants," *Discrete Math.*, no. 24, 1978, pp. 7-11.
11. H.A. Curtis, "Adjacency table method of deriving minimal sums," *IEEE Trans. Comput.*, C-26, Nov. 1977, pp. 1136-1141.

12. R.B. Cutler and S. Muroga, "Derivation of minimal sums for completely specified functions," vol C-36, no 3, Mar. 1987.
13. S.R. Das, "Comments on 'A new algorithm for generating prime implicants'," IEEE Trans. Comput., vol C-20, Dec. 1971, pp. 1614-1615.
14. R. C. DeVries and A. Svoboda, "Multiple-output minimization with mosaics of Boolean functions," IEEE Trans. on Electronic Computers, vol C-24, Aug. 1975, pp. 777-784.
15. B. Dunham and R. Fridshal, "The problem of simplifying logic expressions," J. Symbolic Logic, no. 24, 1959, pp. 17.
16. M.A. Harrison, "Introduction to switching and automata theory," New York: McGraw-Hill, 1965.
17. F.J. Hill and G.R. Peterson, "Switching Theory and Logical Design," New York: Wiley, 1974.
18. S.J. Hong, R.G. Cain and D.L. Ostapko, "MINI: A heuristic approach for logic minimization," IBM J. of Res. and Dev., vol. 18, pp. 443-458, Sept. 1974.
19. B.L. Hulme and R.B. Worrell, "A prime implicant algorithm with factoring," IEEE Trans. Comput., vol. C-24, Nov. 1975, pp. 1129-1131.
20. H.R. Hwa, "A method for generating prime implicants of a boolean expression," IEEE Trans. Comput., vol C-23, June 1974, pp. 634-641.
21. Y. Igarashi, "An improved lower bound on the maximum number of prime implicants," Trans. IECE, Japan, vol E-62, June 1979, pp. 389-394.
22. K. Ishikawa, T. Sasao and H. Terada, "A minimization algorithm for logic expressions and its bounds of application," Syst., Comput., Contr., vol. 13, no. 3, Silver Spring, MD: Scripta Publishing Co., 1982, pp. 425-450.
23. M. Karnaugh, "The map method for synthesis of combinational logic circuits," AIEE Trans., (Commun. and Electronics), vol 72, 1953, pp. 593-599.
24. G.J. Klir and M.A. Martin, "New considerations in teaching switching theory," IEEE Trans. on Educ. vol E-12, no. 4, Dec. 1969, pp. 257-261.
25. T. Kobylarz and A. Al-Najjar, "An examination of the cost function for Programmable Logic Arrays," IEEE Trans. Comput. vol C-28, no. 8, Aug. 1979, pp. 586-590.
26. F. Luccio, "A method for the selection of prime implicants," IEEE Trans. Electron. Comput., vol EC-15, no 2, Apr 1966, pp. 205-212.
27. A. Marquand, "On logical diagrams for n terms," Phil. Mag. 12, 1881, pp. 266-270.
28. E. J. McCluskey, "Minimization of Boolean functions," Bell Syst. Tech. J. no. 35, 1965, pp. 1417-1444.
29. E.J. McCluskey, "Introduction to the Theory of Switching Circuits," New York: McGraw-Hill, 1965.

30. C. McMullen and J. Shearer, "Prime implicants, minimal covers, and complexity of logic simplification," *IEEE Trans. Comput.*, vol C-35, no. 8, Aug. 1986, pp. 761-762.
31. L.P. McNamee, "Pioneering approaches to automatic circuit design," *IEEE Trans. on Educ.*, vol. E-12, no. 4, Dec. 1969, pp. 256.
32. F. Mileto and G. Pulzov, "Average quantities appearing in Boolean function minimization," *IEEE Trans. Electron. Comput.*, vol EC-13, 1964, pp. 87-92.
33. R.E. Miller, "Switching Theory: vol 1," New York: Wiley, 1965.
34. E. Morreale, "Partitioned list algorithms for prime implicant determination from canonical forms," *IEEE Trans. Electron. Comput.*, vol EC-16, no 5, Oct 1967, pp. 611-620.
35. M. Nadler, "Topics in Engineering Logic," New York: MacMillan, 1962, ch. 1,3.
36. H.T. Nagle, Jr., B.D. Carroll, and J.D. Irwin, "An Introduction to Computer Logic," Eaglewood Cliffs, NJ: Prentice Hall, 1975, ch. 4.
37. R.J. Nelson, "Simplest normal truth functions," *J. Symbolic Logic*, vol. 20, June 1955, pp. 105-108.
38. S.R. Petrick, "Minimal synthesis of switching networks," U.S. Gov't Research Rept., vol 36, July 1961, pp. 40A.
39. I.B. Pyne and E.J. McCluskey, "The reduction of redundancy in solving prime implicant tables," *IRE Trans. Electron. Comput.*, vol EC-11, Aug. 1962, pp. 473-482.
40. W.V. Quine, "The problem of simplifying truth functions," *Am. Math. Monthly*, vol. 59, Oct. 1952, pp. 521-531.
41. W.V. Quine, "A way to simplify truth functions," *Am. Math. Monthly*, vol. 62, Oct. 1955, pp. 627-631.
42. W.V. Quine, "On cores and prime implicants of truth function," *Am. Math. Monthly*, vol. 66, Nov. 1959, pp. 755-760.
43. V.T. Rhyne, P. Noe, M.H. McKinney, and U.W. Pooch, "A new technique for the fast minimization of switching functions," *IEEE Trans. Comput.*, vol C-26, Aug. 1977, pp. 757-763.
44. J.P. Roth, "Algebraic topological method for synthesis of switching systems I," *Trans. Amer. Math. Soc.*, vol 88, July 1958, pp. 301-326.
45. J.P. Roth, "Programmed logic array minimization," *IEEE Trans. Comput.*, vol. C-27, Feb. 1980, pp. 174-176.
46. J.P. Roth, "Computer Logic Testing and Verification," Computer Science Press, 1980.
47. T. Sasao, "Input variable assignment and output phase optimization of PLA's," *IEEE Trans. Comput.*, vol C-33, no 10, Oct. 1984, pp. 879-894.
48. J.R. Slagle, C.L. Chang, and R.C.T. Lee, "A new algorithm for generating prime implicants," *IEEE Trans. Comput.*, vol C-19, Apr. 1970, pp. 304-310.

49. Sureshchander, "Minimization of switching functions-A fast technique," *IEEE Trans. Comput.*, vol C-24, July 1975, pp. 753-756.
50. A. Svoboda, "Some applications of contact grids," *Proc. of an Int. Symp. on the Theory of Switching*, Havard U. Press, Cambridge, Mass. 1959, pp. 293-305.
51. A. Svoboda, "Logical instruments for teaching logical design," *IEEE Trans. on Educ.* vol E-12, no. 4, Dec. 1969, pp. 262-273.
52. A. Svoboda, "The concept of term exclusiveness and its effect on the theory of Boolean functions," *J. of ACM*, vol. 22, no. 3, July 1975, pp. 425-440.
53. A. Svoboda, "Ordering of Implicants," *IEEE Trans. Electron. Comput.*, vol EC-16, Feb. 1967, pp. 100-105.
54. A. Svoboda, "Advanced Logical Circuit Design Techniques," New York: Garland STMP Press, 1979, pp. 130-157.
55. P. Tison, "Generalization of consensus theory and application to the minimization of boolean functions," *IEEE Trans. Electron. Comput.*, vol EC-16, Aug. 1967, pp. 446-456.
56. G.C. Vandling, "The simplification of multiple-output switching networks composed of unilateral devices," *IRE Trans. Electron. Comput.*, vol. EC-9, Dec. 1960, pp. 477-486.
57. M.H. Young, "Symmetric minimal covering problem and minimal PLA's with symmetric variables," *IEEE Trans. on Comput.*, vol. C-34, no. 6, June 1985, pp. 523-541.
58. S.Y. Yablonski, "Algorithmic difficulties in sythesizing minimal switching circuits," *Problems of Cybernetics*, London: Pergamon, 1962, pp. 401-557.

Appendix A Program Listing

```
PROGRAM MULTIMIN (INPUT,OUTPUT);
```

```
(* This program accepts all non-zero states of a multi-output boolean
function in the form of their decimal indices and computes all possible
absolute minimal form/s for diode-gate cost function as well as PLA area
cost function. *)
```

```
CONST
```

```

MAXFUNCTIONS      = 4;  (* max # of individual functions *)
MAXALTFORMS       = 32; (* max # of alternate minimal forms *)
MAXSTATES         = 32; (* max # of states in any function *)
MAXVARIABLES      = 6;  (* max # of variables *)
MAXKCELLS         = 128; (* max # of k-cells (PI's) generated *)
MAXALTCELLS       = 64; (* max # of alternate covers for a state *)
MAXVARIABLESPUS2  = 8;  (* # of variables + 2, for pointers *)
NOCOSTFN          = 2;  (* # of diff. cost functions specified *)

```

```
TYPE
```

```

STATEARRAY        = ARRAY[1..MAXFUNCTIONS,0..MAXSTATES]OF INTEGER;
KCELLARRAY        = ARRAY[1..MAXFUNCTIONS,0..MAXKCELLS]OF INTEGER;
BOOLKCELLARRAY    = ARRAY[1..MAXFUNCTIONS,1..MAXKCELLS]OF BOOLEAN;
STATEKCELLARRAY   = ARRAY[1..MAXFUNCTIONS,1..MAXSTATES,
                          1..MAXALTCELLS]OF INTEGER;
STATEALTARRAY     = ARRAY[1..MAXFUNCTIONS,1..MAXSTATES,
                          1..MAXALTFORMS]OF INTEGER;
ALTFORMSARRAY     = ARRAY[1..MAXFUNCTIONS,1..MAXALTFORMS]OF INTEGER;
NOOPRNARRAY       = ARRAY[1..MAXFUNCTIONS,0..MAXVARIABLES,
                          1..MAXALTFORMS]OF INTEGER;
ALREADYSELARRAY   = ARRAY[1..MAXFUNCTIONS,1..MAXKCELLS,
                          1..MAXALTFORMS]OF BOOLEAN;
MININDICESARRAY   = ARRAY[1..MAXFUNCTIONS,1..MAXKCELLS,
                          1..MAXALTFORMS]OF INTEGER;
VARSYMBOLARRAY    = ARRAY[1..MAXVARIABLES]OF CHAR;
BINARYARRAY       = ARRAY[1..MAXVARIABLES]OF BOOLEAN;
FUNCTIONARRAY      = ARRAY[1..MAXFUNCTIONS]OF INTEGER;
FLAGARRAY         = ARRAY[1..MAXFUNCTIONS]OF BOOLEAN;
TOTOPRNCOUNTARRAY = ARRAY[1..MAXALTFORMS]OF INTEGER;
INDEXARRAY        = ARRAY[0..MAXSTATES]OF INTEGER;
RKARRAY           = ARRAY[0..MAXKCELLS]OF INTEGER;

```

```
VAR
```

```

V,V1,V2           : STATEARRAY;
VI,V0,V1I,V1O    : STATEARRAY;
VS                : STATEARRAY;
VSS               : INDEXARRAY;
S1S,S2S          : RKARRAY;

```



```

S1,S2          : KCELLARRAY;
SS             : ALREADYSELARRAY;
V3            : STATEKCELLARRAY;
V4            : STATEALTARRAY;
S5            : ALTFORMSARRAY;
S3,S4         : MININDICESARRAY;
S5TOT         : TOTOPRNCOUNTARRAY;
T,TX,TMTX     : FUNCTIONARRAY;
TI,TXI,TMTXI  : FUNCTIONARRAY;
OT,TXO,TMTXO  : FUNCTIONARRAY;
N             : INTEGER;
S6            : NOOPRNARRAY;
C1,C2         : VARSYMBOLARRAY;
BINO1,BINO2   : BINARYARRAY;
VARC1,VARC2   : BINARYARRAY;
BIWORDAND     : BINARYARRAY;
WORDAND       : INTEGER;
ONECOUNT1    : INTEGER;
ONECOUNT2    : INTEGER;
MINNOOFOPRN   : INTEGER;
NOALTFORMS    : INTEGER;
NOOFFUNCTIONS : INTEGER;
NOMAXKCELLS   : INTEGER;
MAXSTATENO    : INTEGER;
MINSINGLEFN    : FUNCTIONARRAY;
COMBFLAG      : FLAGARRAY;
ITRN          : INTEGER;
OPTION        : INTEGER;
COMBRANGE     : INTEGER;
NOFN          : INTEGER;
SINGLESOLN    : INTEGER;
TERMIN,COSTIN : INTEGER;
FORMENDFLAG   : BOOLEAN;

```

```
FUNCTION F(POWER:INTEGER):INTEGER;
```

```
(* Computes specified power of 2 *)
```

```

VAR
  J           : INTEGER;
  FTEMP       : INTEGER;

```

```

BEGIN
  FTEMP := 1;
  FOR J := 1 TO POWER DO
    FTEMP := (FTEMP * 2);
  F := FTEMP;
END;

```

```
PROCEDURE DOWORDAND(NUM1,NUM2:INTEGER);
```

```
(* Performs bit-by-bit word-AND operation between two numbers *)
```

```

VAR
  X1,Y1       : INTEGER;
  I           : INTEGER;

```

```

BEGIN
  WORDAND := 0;
  ONECOUNT1 := 0;
  ONECOUNT2 := 0;
  X1 := NUM1;
  Y1 := NUM2;
  FOR I := 1 TO N DO
    BEGIN
      IF ((X1 MOD 2) = 1) THEN
        BEGIN

```

```

        BINO1[I] := TRUE;
        ONECOUNT1 := ONECOUNT1 + 1;
    END
ELSE
    BINO1[I] := FALSE;
    IF ((Y1 MOD 2) = 1) THEN
        BEGIN
            BINO2[I] := TRUE;
            ONECOUNT2 := ONECOUNT2 + 1;
        END
    ELSE
        BINO2[I] := FALSE;
        X1 := X1 DIV 2;
        Y1 := Y1 DIV 2;
        BIWORDAND[I] := (BINO1[I]) AND (BINO2[I]);
        IF (BIWORDAND[I]) THEN
            WORDAND := WORDAND + F(I-1);
        END;
    END;
END;

PROCEDURE DECODEINDICES(IND1,IND2:INTEGER);

(* Decodes g-index and k-index to express k-cell in terms of the
   variable characters *)

VAR
    Z,Z1          : INTEGER;

BEGIN
    DOWORDAND(IND1,IND2);
    FOR Z := 1 TO N DO
        BEGIN
            Z1 := (N + 1) - Z;
            IF (BINO1[Z1]) THEN
                BEGIN
                    IF (BINO2[Z1]) THEN
                        WRITE (C1[Z]);
                    ELSE
                        WRITE (C2[Z]);
                    END;
                END;
            END;
        END;
    END;

PROCEDURE READINPUTDATA;

(Reads the input data for the multi-output function to be minimized
 and stores true as well as inverted forms of each function *)

VAR
    I,X,J          : INTEGER;
    SN,SI          : INTEGER;
    YESFLAG        : BOOLEAN;
    VMAX           : INTEGER;
    YESNO          : CHAR;

BEGIN
    WRITELN ('GIVE NO OF FUNCTIONS');
    READLN (NOFN);
    NOOFFUNCTIONS := NOFN;
    WRITELN ('GIVE NO OF VARIABLES');
    READLN (N);
    MAXSTATENO := F(N) - 1;
    WRITELN('GIVE SYMBOLS REPRESENTING THE VARIABLES');
    FOR I := 1 TO N DO
        READLN (C1[I]);
        WRITELN('GIVE SYMBOLS REPRESENTING COMPLEMENTS OF VARIABLES');
        FOR I := 1 TO N DO

```

```

READLN (C2[I]);
FOR X := 1 TO NOOFFUNCTIONS DO
  BEGIN
    WRITELN ('GIVE TOTAL NO OF STATES FOR FUNCTION',X:3);
    READLN (OT[X]);
    WRITELN ('GIVE NO OF UNDEFINED STATES FOR FUNCTION',X:3);
    READLN(TXO[X]);
    WRITELN('GIVE DECIMAL INDICES OF STATES FOR FUNCTION',X:3);
    WRITELN ('FIRST GIVE ALL FREE STATES INDICES FOR FN',X:3);
    TMTXO[X] := OT[X] - TXO[X];
    FOR I := 1 TO TMTXO[X] DO
      BEGIN
        READLN (VO[X,I]);
        VIO[X,I] := 0;
      END;
    IF (TXO[X] <> 0) THEN
      WRITELN ('NOW GIVE ALL UNDEFINED STATES INDICES FOR FN',X:3);
      FOR I := (OT[X] - TXO[X] + 1) TO OT[X] DO
        BEGIN
          READLN (VO[X,I]);
          VIO[X,I] := -1;
        END;
      END;
    WRITELN ('INVERTED FUNCTION MINIMIZATION ? Y/N');
    READLN (YESNO);
    IF (YESNO = 'Y') THEN
      COMBRANGE := 8
    ELSE
      COMBRANGE := 1;
    FOR X := 1 TO NOOFFUNCTIONS DO
      BEGIN
        TII[X] := F(N) - TMTXO[X];
        TXI[X] := TXO[X];
        TMTXI[X] := TII[X] - TXI[X];
        SI := 0;
        I := 1;
        FOR SN := 1 TO (TMTXO[X] + 1) DO
          BEGIN
            VMAX := VO[X,SN];
            IF (SN = TMTXO[X] + 1) THEN
              VMAX := F(N);
            WHILE (SI < VMAX) DO
              BEGIN
                YESFLAG := FALSE;
                FOR J := (TMTXO[X] + 1) TO OT[X] DO
                  BEGIN
                    IF (SI = VO[X,J]) THEN
                      YESFLAG := TRUE;
                  END;
                IF (NOT(YESFLAG)) THEN
                  BEGIN
                    VI[X,I] := SI;
                    VII[X,I] := 0;
                    S1 := S1 + 1;
                    I := I + 1;
                  END
                ELSE
                  SI := SI + 1;
                END;
              SI := SI + 1;
            END;
          END;
        FOR SN := (TMTXO[X] + 1) TO OT[X] DO
          BEGIN
            VI[X,I] := VO[X,SN];
            VII[X,I] := -1;
            I := I + 1;
          END;
        END;
      END;
    END;
  END;

```

```

END;
END;

PROCEDURE GETACOMBINATION;

(* Gets a combination of intersection functions (true or inverted)
   for a given combination iteration *)

VAR
  FN,SN      : INTEGER;
  BICODE     : INTEGER;

BEGIN
  BICODE := ITRN - 1;
  DOWORDAND(BICODE,BICODE);
  FOR FN := 1 TO NOOFFUNCTIONS DO
    BEGIN
      IF (NOT(BINO1[FN])) THEN
        BEGIN
          COMBFLAG[FN] := TRUE;
          T[FN] := OT[FN];
          TX[FN] := TXO[FN];
          TMTX[FN] := TMTXO[FN];
          FOR SN := 1 TO T[FN] DO
            BEGIN
              V[FN,SN] := VO[FN,SN];
              VI[FN,SN] := VIO[FN,SN];
            END;
          END;
        ELSE
          BEGIN
            COMBFLAG[FN] := FALSE;
            T[FN] := TI[FN];
            TX[FN] := TXI[FN];
            TMTX[FN] := TMTXI[FN];
            FOR SN := 1 TO T[FN] DO
              BEGIN
                V[FN,SN] := VI[FN,SN];
                VI[FN,SN] := V1I[FN,SN];
              END;
            END;
          END;
        END;
      END;
    END;
  END;

PROCEDURE FINDALLMAXIMALKCELLS;

(* Finds all maximal k-cells for the multi-output function *)

TYPE
  ANDARRAY      = ARRAY[1..MAXVARIABLESP2]OF INTEGER;

VAR
  P2,L          : ANDARRAY;
  K1,I,J1S,TXX,J1,
  P,Q,I1M,B,KPS,U,
  S,R,K,J,X     : INTEGER;
  TEST,ALLBOUND : BOOLEAN;
  TESTFLAG      : FLAGARRAY;

PROCEDURE GETAGENERICCELL(J1,B,I1M:INTEGER;VAR R:INTEGER;
  VAR L:ANDARRAY;P2:ANDARRAY);

(* Computes g-cell index from the current pointers *)

VAR
  I1            : INTEGER;

```

```

BEGIN
  IF (B <> J1) THEN
    BEGIN
      FOR I1 := B TO I1M DO
        L[I1 + 1] := L[I1] + 1;
      END;
      R := 0;
      FOR I1 := 1 TO N DO
        R := R + P2[L[I1]];
      END;
    END;

  PROCEDURE GETNEXTGENERICCELL(VAR B:INTEGER;J1,Q:INTEGER;
    VAR L:ANDARRAY);

  (* Updates pointers for the next generic k-cell to be generated *)

  VAR
    NEXTGFLAG      : BOOLEAN;

  BEGIN
    NEXTGFLAG := FALSE;
    B := J1;
    REPEAT
      L[B] := L[B] + 1;
      IF (L[B] <= (Q + B + 1)) THEN
        NEXTGFLAG := TRUE
      ELSE
        B := B - 1;
      UNTIL ((B = 0) OR (NEXTGFLAG));
    END;

  PROCEDURE TESTIFALREADYGENERATED(VAR TEST:BOOLEAN;VAR K:INTEGER;
    K1,KPS,U,R:INTEGER);

  (* Checks if the definite k-cell is generated already *)

  VAR
    KP              : INTEGER;

  BEGIN
    TEST := FALSE;
    DOWORDAND(R,U);
    K := WORDAND;
    IF (S2S[K1] = R) THEN
      BEGIN
        KP := KPS+1;
        REPEAT
          IF (K = S1S[KP]) THEN
            TEST := TRUE
          ELSE
            KP := KP + 1;
          UNTIL ((KP > K1) OR (TEST));
        END;
      END;
    END;

  PROCEDURE TESTIFMAXIMAL(VAR TESTFLAG:FLAGARRAY;
    K,P,R:INTEGER;VAR TEST:BOOLEAN);

  (* Checks for the maximal existence of the generated k-cell *)

  VAR
    I,X,S          : INTEGER;

  BEGIN
    TEST := FALSE;

```

```

FOR X := 1 TO NOOFFUNCTIONS DO
  BEGIN
    S := -1;
    TESTFLAG[X] := FALSE;
    I := 0;
    REPEAT
      IF (VS[X,I] <> 0) THEN
        BEGIN
          DOWORDAND(R,I);
          IF (K = WORDAND) THEN
            BEGIN
              S := S + 1;
              IF (S = P) THEN
                BEGIN
                  TEST := TRUE;
                  TESTFLAG[X] := TRUE;
                END
              ELSE
                I := I + 1;
            END
          ELSE
            I := I + 1;
        END
      ELSE
        I := I + 1;
    UNTIL ((I > MAXSTATENO) OR (TESTFLAG[X]));
  END;
END;

PROCEDURE UPDATEMATRICES(VAR K1:INTEGER;P,K,R:INTEGER;
  TESTFLAG:FLAGARRAY);

(* Updates the matrices storing the indices of the k-cell *)

VAR
  J2,X,XX,S      : INTEGER;
  STATEBOUND     : BOOLEAN;

BEGIN
  K1 := K1 + 1;
  S1S[K1] := K;
  S2S[K1] := R;
  FOR X := 1 TO NOOFFUNCTIONS DO
    BEGIN
      IF (TESTFLAG[X]) THEN
        BEGIN
          S := P;
          S1[X,K1] := K;
          S2[X,K1] := R;
          J2 := 1;
          REPEAT
            DOWORDAND (R,V[X,J2]);
            IF (K = WORDAND) THEN
              BEGIN
                S := S - 1;
                IF (V1[X,J2] = 0) THEN (* OR VSS[V[X,J2] = 1 *)
                  BEGIN
                    STATEBOUND := TRUE;
                    FOR XX := 1 TO NOOFFUNCTIONS DO
                      BEGIN
                        IF ((VS[XX,V[X,J2]] <> 0)
                          AND (NOT(TESTFLAG[XX]))) THEN
                          STATEBOUND := FALSE;
                      END;
                    IF (STATEBOUND) THEN
                      BEGIN
                        VSS[V[X,J2]] := 2;
                      END;
                  END;
                END;
              END;
            ELSE
              J2 := J2 + 1;
            END;
          UNTIL (J2 > MAXSTATENO);
        END;
      ELSE
        J2 := J2 + 1;
      END;
    END;
  END;

```

```

                V1[X,J2] := 2;
                END;
            END;
            V2[X,J2] := V2[X,J2] + 1;
            V3[X,J2,V2[X,J2]] := K1;
            IF (S >= 0) THEN
                J2 := J2 + 1;
            END
        ELSE
            J2 := J2 + 1;
            UNTIL (S < 0);
        END
    ELSE
        BEGIN
            S1[X,K1] := -1;
            S2[X,K1] := -1;
        END;
    END;
END;

PROCEDURE CONVERTFREETOBOUND(VAR ALLBOUND:BOOLEAN);

(* Converts maximally-covered free states to bound *)

VAR
    J2,X          : INTEGER;

BEGIN
    ALLBOUND := TRUE;
    FOR X := 1 TO NOOFFUNCTIONS DO
        BEGIN
            FOR J2 := 1 TO TMTX[X] DO
                BEGIN
                    IF (V1[X,J2] = 2) THEN (* OR VSS[V[X,J2]] = 2 *)
                        BEGIN
                            V1[X,J2] := -2;
                            VSS[V[X,J2]] := -2;
                        END;
                    IF (V1[X,J2] = 0) THEN (* OR VSS[V[X,J2]] = 1 *)
                        ALLBOUND := FALSE;
                    END;
                END;
            END;
        END;
    END;

BEGIN (* FINDALLMAXKCELLS *)
    FOR X := 1 TO NOOFFUNCTIONS DO
        BEGIN
            FOR I := 0 TO MAXSTATENO DO
                BEGIN
                    VS[X,I] := 0;
                    VSS[I] := 0;
                END;
            END;
        END;
    FOR X := 1 TO NOOFFUNCTIONS DO
        BEGIN
            FOR I := 1 TO T[X] DO
                BEGIN
                    IF (I <= TMTX[X]) THEN
                        BEGIN
                            VS[X,V[X,I]] := 1;
                            VSS[V[X,I]] := 1;
                        END
                    ELSE
                        BEGIN
                            VS[X,V[X,I]] := -1;
                            IF (VSS[V[X,I]] = 0) THEN

```

```

        VSS[V[X,I]] := -1;
    END;
END;
END;
TXX := 0;
FOR X := 1 TO NOOFFUNCTIONS DO
    BEGIN
        IF (T[X] > TXX) THEN
            TXX := T[X];
            S1[X,0] := -1;
            S2[X,0] := -1;
            S1S[0] := -1;
            S2S[0] := -1;
            FOR I := 1 TO MAXSTATES DO
                BEGIN
                    V2[X,I] := 0;
                    FOR J := 1 TO MAXALTCELLS DO
                        V3[X,I,J] := 0;
                    END;
                END;
            END;
            K1 := 0;
            P2[1] := 0;
            P2[2] := 1;
            FOR I := 2 TO (N + 1) DO
                BEGIN
                    P2[I+1] := P2[I] * 2;
                    L[I] := 1;
                END;
            J1S := N - TRUNC(LN(TXX)/LN(2));
            J1 := J1S - 1;
            REPEAT
                J1 := J1 + 1;
                Q := N - J1;
                P := F(Q) - 1;
                L[1] := 2;
                I1M := J1 - 1;
                B := 1;
                REPEAT
                    GETAGENERICCELL(J1,B,I1M,R,L,P2);
                    KPS := K1;
                    FOR U := 0 TO MAXSTATENO DO
                        BEGIN
                            IF (VSS[U] > 0) THEN
                                BEGIN
                                    TESTIFALREADYGENERATED(TEST,K,K1,KPS,U,R);
                                    IF (NOT(TEST)) THEN
                                        BEGIN
                                            TESTIFMAXIMAL(TESTFLAG,K,P,R,TEST);
                                            IF (TEST) THEN
                                                UPDATEMATRICES(K1,P,K,R,TESTFLAG);
                                            END;
                                        END;
                                    END;
                                END;
                            END;
                        GETNEXTGENERICCELL(B,J1,Q,L);
                        UNTIL (B = 0);
                        CONVERTFREETOBOUND(ALLBOUND);
                        UNTIL (J1=N) OR (ALLBOUND);
                        NOMAXKCELLS := K1;
                    END;
                END;
            REPEAT
                PROCEDURE FINDMINIMUMFORM;
                (* Computes all possible absolute minimal forms from the maximal
                k-cells generated *)
                VAR

```



```

J                : INTEGER;
NOSELECTED      : INTEGER;
TIMES           : FUNCTIONARRAY;
C1VAR,C2VAR     : BINARYARRAY;
MINTERM,INCOST : INTEGER;

PROCEDURE INITIALIZEOUTPUTDATA(VAR J,NOSELECTED:INTEGER;
                               VAR TIMES:FUNCTIONARRAY;
                               VAR C1VAR,C2VAR:BINARYARRAY;
                               VAR MINTERM,INCOST:INTEGER);

(* Initializes output matrices for each iteration of combination *)

VAR
  I1,J1,X        : INTEGER;

BEGIN
  FOR J1 := 1 TO MAXALTFORMS DO
    BEGIN
      S5TOT[J1] := 0;
      FOR X := 1 TO NOOFFUNCTIONS DO
        BEGIN
          S5[X,J1] := 0;
          FOR I1 := 0 TO N DO
            S6[X,I1,J1] := 0;
          END;
        END;
      FOR X := 1 TO MAXFUNCTIONS DO
        BEGIN
          TIMES[X] := 0;
          FOR I1 := 1 TO MAXSTATES DO
            BEGIN
              FOR J1 := 1 TO MAXALTFORMS DO
                V4[X,I1,J1] := 0;
              END;
              FOR I1 := 1 TO NOMAXKCELLS DO
                BEGIN
                  FOR J1 := 1 TO MAXALTFORMS DO
                    BEGIN
                      S3[X,I1,J1] := 0;
                      S4[X,I1,J1] := 0;
                      SS[X,I1,J1] := FALSE;
                    END;
                  END;
                END;
              END;
            END;
          FOR I1 := 1 TO N DO
            BEGIN
              C1VAR[I1] := FALSE;
              C2VAR[I1] := FALSE;
            END;
          NOALTFORMS := 1;
          NOSELECTED := 0;
          MINTERM := 0;
          INCOST := 0;
          J := 1;
        END;
      END;
    END;
  END;

PROCEDURE APPLYRULE2(VAR J:INTEGER;
                    V1,V2:STATEARRAY;V3:STATEKCELLARRAY;
                    NOSELECTED:INTEGER;TIMES:FUNCTIONARRAY;
                    C1VAR,C2VAR:BINARYARRAY;
                    MINTERM,INCOST:INTEGER);FORWARD;

(* Forward declaration for co-recursion with procedure APPLYRULE1 *)

PROCEDURE FINDCONDITION(VAR COND1,COND2A,COND2B,COND3:BOOLEAN;

```

```

X,CELL1,CELL2:INTEGER;
V1,V2:STATEARRAY;V3:STATEKCELLARRAY;
VAR BOUNDFLAG:BOOLEAN;J:INTEGER;
TIMES:FUNCTIONARRAY;
C1VAR,C2VAR:BINARYARRAY;
MINTERM,INCOST:INTEGER);

(* Find conditions for elimination rule (Cond 1a,b) for the given
k-cells K1 & K2 *)

PROCEDURE FINDCONDGATE (VAR COND1,COND2A,COND2B,COND3:BOOLEAN;
X,CELL1,CELL2:INTEGER;
V1,V2:STATEARRAY;V3:STATEKCELLARRAY;
VAR BOUNDFLAG:BOOLEAN;J:INTEGER;
TIMES:FUNCTIONARRAY;
C1VAR,C2VAR:BINARYARRAY;
MINTERM,INCOST:INTEGER);

(* Find conditions for diode-gate cost function *)

VAR
I
: INTEGER;
CELL1TERMS
: INTEGER;
CELL2TERMS
: INTEGER;
R1,K1,R2,K2
: INTEGER;
COST1,COST2
: INTEGER;
CELL1FLAG,CELL2FLAG
: BOOLEAN;
ALLFLAG1,ALLFLAG2
: FLAGARRAY;

BEGIN
COND1 := FALSE;
COND2A := FALSE;
COND2B := FALSE;
COND3 := FALSE;
R1 := S2[X,CELL1];
K1 := S1[X,CELL1];
R2 := S2[X,CELL2];
K2 := S1[X,CELL2];
ALLFLAG1[X] := TRUE;
ALLFLAG2[X] := TRUE;
I := 0;
REPEAT
I := I + 1;
IF (V1[X,I] = -2) THEN
BEGIN
DOWORDAND(R1,V[X,I]);
IF (WORDAND <> K1) THEN
ALLFLAG1[X] := FALSE;
DOWORDAND(R2,V[X,I]);
IF (WORDAND <> K2) THEN
ALLFLAG2[X] := FALSE;
END;
UNTIL ((I = TMTX[X]) OR
((NOT(ALLFLAG1[X])) AND (NOT(ALLFLAG2[X]))));
DOWORDAND(R1,R2);
CELL1TERMS := ONECOUNT1;
CELL2TERMS := ONECOUNT2;
IF (SS[X,CELL1,J]) THEN
COST1 := 0
ELSE
COST1 := CELL1TERMS;
IF ((CELL1TERMS > 1) AND
((TIMES[X] > 0) OR (NOT(ALLFLAG1[X])))) THEN
COST1 := COST1 + 1;
IF (SS[X,CELL2,J]) THEN
COST2 := 0

```

```

ELSE
  COST2 := CELL2TERMS;
  IF ((CELL2TERMS > 1) AND
      ((TIMES[X] > 0) OR (NOT(ALLFLAG2[X])))) THEN
    COST2 := COST2 + 1;
  IF (COST2 < COST1) THEN
    COND2A := TRUE;
  IF (COST1 < COST2) THEN
    COND2B := TRUE;
  BOUNDFLAG := FALSE;
  FOR I := 1 TO TMTX[X] DO
    BEGIN
      CELL1FLAG := FALSE;
      CELL2FLAG := FALSE;
      DOWORDAND(R1,V[X,I]);
      IF (WORDAND = K1) THEN
        CELL1FLAG := TRUE;
      DOWORDAND(R2,V[X,I]);
      IF (WORDAND = K2) THEN
        CELL2FLAG := TRUE;
      IF ((CELL1FLAG) AND (V1[X,I] = -2)) THEN
        BOUNDFLAG := TRUE;
      IF ((CELL2FLAG) AND (NOT(CELL1FLAG))
          AND (V1[X,I] = -2)) THEN
        COND1 := TRUE;
      IF ((CELL1FLAG) AND (NOT(CELL2FLAG))
          AND (V1[X,I] = -2)) THEN
        COND3 := TRUE;
    END;
  END;

PROCEDURE FINDCONDPLA (VAR COND1,COND2A,COND2B,COND3:BOOLEAN;
  X,CELL1,CELL2:INTEGER;
  V1,V2:STATEARRAY;V3:STATEKCELLARRAY;
  VAR BOUNDFLAG:BOOLEAN;J:INTEGER;
  TIMES:FUNCTIONARRAY;
  C1VAR,C2VAR:BINARARRAY;
  MINTERM,INCOST:INTEGER);

(* Find conditions for PLA area cost function *)

VAR
  I,I1      : INTEGER;
  CELL1TERMS : INTEGER;
  CELL2TERMS : INTEGER;
  R1,K1,R2,K2 : INTEGER;
  COST1,COST2 : INTEGER;
  CELL1FLAG,CELL2FLAG : BOOLEAN;
  ALLFLAG1,ALLFLAG2 : FLAGARRAY;

BEGIN
  COND1 := FALSE;
  COND2A := FALSE;
  COND2B := FALSE;
  COND3 := FALSE;
  R1 := S2[X,CELL1];
  K1 := S1[X,CELL1];
  R2 := S2[X,CELL2];
  K2 := S1[X,CELL2];
  CELL1TERMS := 0;
  CELL2TERMS := 0;
  DOWORDAND(R1,K1);
  FOR I := 1 TO N DO
    BEGIN
      I1 := (N + 1) - I;
      IF (BINO1[I1]) THEN

```

```

        BEGIN
          IF (BINO2[I1]) THEN
            BEGIN
              IF (NOT(C1VAR[I])) THEN
                CELL1TERMS := CELL1TERMS + 1;
              END
            ELSE
              BEGIN
                IF (NOT(C2VAR[I])) THEN
                  CELL1TERMS := CELL1TERMS + 1;
                END;
              END;
            END;
          END;
        DOWORDAND(R2,K2);
        FOR I := 1 TO N DO
          BEGIN
            I1 := (N + 1) - I;
            IF (BINO1[I1]) THEN
              BEGIN
                IF (BINO2[I1]) THEN
                  BEGIN
                    IF (NOT(C1VAR[I])) THEN
                      CELL2TERMS := CELL2TERMS + 1;
                    END
                  ELSE
                    BEGIN
                      IF (NOT(C2VAR[I])) THEN
                        CELL2TERMS := CELL2TERMS + 1;
                      END;
                    END;
                  END;
                END;
              END;
            IF (SS[X,CELL1,J]) THEN
              COST1 := 0
            ELSE
              COST1 := ((MINTERM + 1) * ((INCOST + CELL1TERMS) + NOFN))
                - ((MINTERM) * (INCOST + NOFN));
            IF (SS[X,CELL2,J]) THEN
              COST2 := 0
            ELSE
              COST2 := ((MINTERM + 1) * ((INCOST + CELL2TERMS) + NOFN))
                - ((MINTERM) * (INCOST + NOFN));
            IF (COST2 < COST1) THEN
              COND2A := TRUE;
            IF (COST1 < COST2) THEN
              COND2B := TRUE;
            BOUNDFLAG := FALSE;
            FOR I := 1 TO TMTX[X] DO
              BEGIN
                CELL1FLAG := FALSE;
                CELL2FLAG := FALSE;
                DOWORDAND(R1,V[X,I]);
                IF (WORDAND = K1) THEN
                  CELL1FLAG := TRUE;
                DOWORDAND(R2,V[X,I]);
                IF (WORDAND = K2) THEN
                  CELL2FLAG := TRUE;
                IF ((CELL1FLAG) AND (V1[X,I] = -2)) THEN
                  BOUNDFLAG := TRUE;
                IF ((CELL2FLAG) AND (NOT(CELL1FLAG))
                    AND (V1[X,I] = -2)) THEN
                  COND1 := TRUE;
                IF ((CELL1FLAG) AND (NOT(CELL2FLAG))
                    AND (V1[X,I] = -2)) THEN
                  COND3 := TRUE;
                END;
              END;
            END;
          END;
        END;

```

```

BEGIN (* FINDCONDITION *)
  IF (OPTION = 1) THEN
    FINDCONDGATE(COND1,COND2A,COND2B,COND3,
                 X,CELL1,CELL2,
                 V1,V2,V3,BOUNDFLAG,J,TIMES,
                 C1VAR,C2VAR,MINTERM,INCOST)
  ELSE
    FINDCONDPLA(COND1,COND2A,COND2B,COND3,
                 X,CELL1,CELL2,
                 V1,V2,V3,BOUNDFLAG,J,TIMES,
                 C1VAR,C2VAR,MINTERM,INCOST);
  END;

PROCEDURE FINDNEWCONDITION(VAR COND4A,COND4B,COND5A,COND5B:BOOLEAN;
                            X,I,CELL1,CELL2:INTEGER;
                            V1,V2:STATEARRAY;V3:STATEKCELLARRAY;
                            J:INTEGER;TIMES:FUNCTIONARRAY;
                            C1VAR,C2VAR:BINARAYARRAY;
                            MINTERM,INCOST:INTEGER);

(* Find condition of the elimination rule related to functions
   other than the reference functions (Cond 1c) *)

VAR
  XX,I1                : INTEGER;
  STATETEMP            : INTEGER;
  TEMPK1,TEMPR1       : INTEGER;
  TEMPK2,TEMPR2       : INTEGER;
  CELLTEMP1,CELLTEMP2 : INTEGER;
  FLAG,BOUNDFLAG      : BOOLEAN;
  COND1,COND3         : BOOLEAN;
  COND2A,COND2B       : BOOLEAN;

PROCEDURE FINDSTATENO(VAR I1:INTEGER;STATETEMP,XX:INTEGER;
                     V1:STATEARRAY;VAR FLAG:BOOLEAN);

(* Find corresponding state in the given function *)

VAR
  I2          : INTEGER;

BEGIN
  I2 := 0;
  FLAG := FALSE;
  REPEAT
    I2 := I2 + 1;
    IF ((V[XX,I2] = STATETEMP)) THEN
      BEGIN
        FLAG := TRUE;
        I1 := I2;
      END;
    UNTIL ((FLAG) OR (I2 = T[XX]));
  END;

PROCEDURE FINDKCELLNO(VAR CELLTEMP:INTEGER;XX,I1:INTEGER;
                     VAR FLAG:BOOLEAN;TEMPK,TEMPR:INTEGER;
                     V3:STATEKCELLARRAY);

(* Find corresponding k-cell in the given function *)

VAR
  J1          : INTEGER;
  KCELLNO     : INTEGER;

BEGIN

```

```

J1 := 0;
REPEAT
  J1 := J1 + 1;
  IF (V3[XX,I1,J1] > 0) THEN
    BEGIN
      KCELLNO := V3[XX,I1,J1];
      IF ((S1[XX,KCELLNO] = TEMPK) AND
          (S2[XX,KCELLNO] = TEMPR)) THEN
        CELLTEMP := KCELLNO
      ELSE
        FLAG := FALSE;
    END;
  UNTIL ((V3[XX,I1,J1] = 0) OR (J1 = MAXALTCELLS));
END;

BEGIN (* FINDNEWCONDITION *)
  STATETEMP := V[X,I];
  COND4A := FALSE;
  COND4B := FALSE;
  TEMPK1 := S1[X,CELL1];
  TEMPR1 := S2[X,CELL1];
  TEMPK2 := S1[X,CELL2];
  TEMPR2 := S2[X,CELL2];
  FOR XX := 1 TO NOOFFUNCTIONS DO
    BEGIN
      IF (XX <> X) THEN
        BEGIN
          FLAG := FALSE;
          I1 := 0;
          FINDSTATENO(I1,STATETEMP,XX,V1,FLAG);
          IF (FLAG) THEN
            BEGIN
              CELLTEMP1 := 0;
              CELLTEMP2 := 0;
              FINDKCELLNO(CELLTEMP1,XX,I1,FLAG,
                          TEMPK1,TEMPR1,V3);
              FINDKCELLNO(CELLTEMP2,XX,I1,FLAG,
                          TEMPK2,TEMPR2,V3);
              IF ((CELLTEMP1 <> 0) AND (CELLTEMP2 <> 0)) THEN
                BEGIN
                  FINDCONDITION(COND1,COND2A,COND2B,COND3,
                                XX,CELLTEMP1,CELLTEMP2,
                                V1,V2,V3,BOUNDFLAG,J,TIMES,
                                C1VAR,C2VAR,MINTERM,INCOST);

                  IF (COND1) THEN
                    COND4A := TRUE;
                  IF (COND3) THEN
                    COND4B := TRUE;
                END
              ELSE
                BEGIN
                  IF((CELLTEMP1 = 0) AND (CELLTEMP2 <> 0))THEN
                    BEGIN
                      CELLTEMP1 := CELLTEMP2;
                      FINDCONDITION(COND1,COND2A,COND2B,COND3,
                                    XX,CELLTEMP1,CELLTEMP2,
                                    V1,V2,V3,BOUNDFLAG,J,TIMES,
                                    C1VAR,C2VAR,MINTERM,INCOST);

                      IF (BOUNDFLAG) THEN
                        COND4A := TRUE;
                    END
                  ELSE
                    BEGIN
                      IF((CELLTEMP1 <> 0) AND (CELLTEMP2 = 0))THEN
                        BEGIN
                          CELLTEMP2 := CELLTEMP1;
                          FINDCONDITION(COND1,COND2A,

```

```

COND2B,COND3,
XX,CELLTEMP1,CELLTEMP2,
V1,V2,V3,BOUNDFLAG,J,TIMES,
C1VAR,C2VAR,MINTERM,INCOST);
IF (BOUNDFLAG) THEN
  COND4B := TRUE;
END;
END;
END;
END;
END;
END;
END;

```

```

PROCEDURE SELECTTHISKCELL(J,CELL1,X:INTEGER;VAR V1:STATEARRAY;
  SINGLESELFLAG:BOOLEAN;
  VAR TIMES:FUNCTIONARRAY;
  VAR C1VAR,C2VAR:BINARARRAY;
  VAR MINTERM,INCOST:INTEGER);

```

(* Select the given k-cell in the given function for the minimal form *)

```

PROCEDURE SELECTGATEKCELL(J,CELL1,X:INTEGER;VAR V1:STATEARRAY;
  SINGLESELFLAG:BOOLEAN;
  VAR TIMES:FUNCTIONARRAY;
  VAR C1VAR,C2VAR:BINARARRAY;
  VAR MINTERM,INCOST:INTEGER);

```

(* Select the given k-cell in case of diode-gate cost function *)

```

VAR
  II,XX,JJ      : INTEGER;
  RTEMP,KTEMP   : INTEGER;
  K1TEMP        : INTEGER;
  FUNCTIONFLAG  : BOOLEAN;
  FNSELFLAG     : BOOLEAN;
  ALLFLAG       : FLAGARRAY;

```

```

BEGIN
  RTEMP := S2[X,CELL1];
  KTEMP := S1[X,CELL1];
  TIMES[X] := TIMES[X] + 1;
  IF (NOT(SS[X,CELL1,J])) THEN
    BEGIN
      DOWORDAND (RTEMP,RTEMP);
      IF (ONECOUNT1 > 1) THEN
        BEGIN
          S5TOT[J] := S5TOT[J] + ONECOUNT1;
          S5[X,J] := S5[X,J] + (ONECOUNT1 + 1);
        END;
        S6[X,ONECOUNT1,J] := S6[X,ONECOUNT1,J] + 1;
      END;
    FOR XX := 1 TO NOOFFUNCTIONS DO
      BEGIN
        JJ := 0;
        FUNCTIONFLAG := FALSE;
        REPEAT
          JJ := JJ + 1;
          IF ((S2[XX,JJ] = RTEMP) AND (S1[XX,JJ] = KTEMP)) THEN
            BEGIN
              FUNCTIONFLAG := TRUE;
              K1TEMP := JJ;
            END;
          UNTIL ((FUNCTIONFLAG) OR (JJ = NOMAXKCELLS));
          IF (FUNCTIONFLAG) THEN

```

```

BEGIN
  IF (SINGLESELFLAG) THEN
    BEGIN
      SS[XX,K1TEMP,J] := TRUE;
    END;
  IF ((SINGLESELFLAG) AND (XX = X)) THEN
    BEGIN
      FNSSELFLAG := FALSE;
      FOR II := 1 TO TMTX[XX] DO
        BEGIN
          IF ((V1[XX,II] = -2) OR (V1[XX,II] = 2)) THEN
            BEGIN
              DOWORDAND(S2[XX,K1TEMP],V[XX,II]);
              IF (WORDAND = S1[XX,K1TEMP]) THEN
                BEGIN
                  S3[XX,K1TEMP,J] := S1[XX,K1TEMP];
                  S4[XX,K1TEMP,J] := S2[XX,K1TEMP];
                  V1[XX,II] := 1;
                  V4[XX,II,J] := K1TEMP;
                  FNSSELFLAG := TRUE;
                END;
              END;
            END;
          IF (FNSSELFLAG) THEN
            BEGIN
              ALLFLAG[X] := TRUE;
              JJ := 0;
              REPEAT
                JJ := JJ + 1;
                IF (V1[X,JJ] = -2) THEN
                  ALLFLAG[X] := FALSE;
                UNTIL ((JJ = TMTX[X]) OR (NOT(ALLFLAG[X])));
              IF ((TIMES[X] > 1) OR (NOT(ALLFLAG[X]))) THEN
                S5TOT[J] := S5TOT[J] + 1;
              END;
            END;
          END;
        END;
      IF (S5TOT[J] > MINNOOFOPRN) THEN
        FORMENDFLAG := TRUE
      ELSE
        FORMENDFLAG := FALSE;
      END;
    END;
  PROCEDURE SELECTPLAKCELL(J,CELL1,X:INTEGER;VAR V1:STATEARRAY;
    SINGLESELFLAG:BOOLEAN;
    VAR TIMES:FUNCTIONARRAY;
    VAR C1VAR,C2VAR:BINARARRAY;
    VAR MINTERM,INCOST:INTEGER);

  (* Select the given k-cell for the PLA area cost function *)

  VAR
    II,XX,JJ      : INTEGER;
    RTEMP,KTEMP   : INTEGER;
    K1TEMP        : INTEGER;
    FUNCTIONFLAG  : BOOLEAN;
    FNSSELFLAG    : BOOLEAN;
    ALLFLAG       : FLAGARRAY;

  BEGIN
    RTEMP := S2[X,CELL1];
    KTEMP := S1[X,CELL1];
    IF (NOT(SS[X,CELL1,J])) THEN
      BEGIN
        MINTERM := MINTERM + 1;
        DOWORDAND (RTEMP,KTEMP);
      END;
    END;
  END;

```



```

FOR XX := 1 TO N DO
  BEGIN
    II := (N + 1) - XX;
    IF (BINO1[II]) THEN
      BEGIN
        IF (BINO2[II]) THEN
          BEGIN
            IF (NOT(C1VAR[XX])) THEN
              BEGIN
                INCOST := INCOST + 1;
                C1VAR[XX] := TRUE;
              END;
            END;
          END;
        ELSE
          BEGIN
            IF (NOT(C2VAR[XX])) THEN
              BEGIN
                INCOST := INCOST + 1;
                C2VAR[XX] := TRUE;
              END;
            END;
          END;
        END;
      END;
    S5TOT[J] := MINTERM * (INCOSt + NOFN);
  END;
FOR XX := 1 TO NOOFFUNCTIONS DO
  BEGIN
    JJ := 0;
    FUNCTIONFLAG := FALSE;
    REPEAT
      JJ := JJ + 1;
      IF ((S2[XX,JJ] = RTEMP) AND (S1[XX,JJ] = KTEMP)) THEN
        BEGIN
          FUNCTIONFLAG := TRUE;
          K1TEMP := JJ;
        END;
      UNTIL ((FUNCTIONFLAG) OR (JJ = NOMAXKCELLS));
      IF (FUNCTIONFLAG) THEN
        BEGIN
          IF (SINGLESELFLAG) THEN
            BEGIN
              SS[XX,K1TEMP,J] := TRUE;
            END;
          IF ((SINGLESELFLAG) AND (XX = X)) THEN
            BEGIN
              FNSSELFLAG := FALSE;
              FOR II := 1 TO TMTX[XX] DO
                BEGIN
                  IF ((V1[XX,II] = -2) OR (V1[XX,II] = 2)) THEN
                    BEGIN
                      DOWORDAND(S2[XX,K1TEMP],V[XX,II]);
                      IF (WORDAND = S1[XX,K1TEMP]) THEN
                        BEGIN
                          S3[XX,K1TEMP,J] := S1[XX,K1TEMP];
                          S4[XX,K1TEMP,J] := S2[XX,K1TEMP];
                          V1[XX,II] := 1;
                          V4[XX,II,J] := K1TEMP;
                          FNSSELFLAG := TRUE;
                        END;
                      END;
                    END;
                END;
              END;
            END;
          END;
        END;
      END;
    END;
  END;
  IF (S5TOT[J] > MINNOOFOPRN) THEN
    FORMENDFLAG := TRUE
  ELSE

```

```

    FORMENDFLAG := FALSE;
END;

BEGIN (* SELECTTHISFCCELL *)
  IF (OPTION = 1) THEN
    SELECTGATEKCELL(J,CELL1,X,V1,SINGLESELFLAG,
                   TIMES,C1VAR,C2VAR,
                   MINTERM,INCOST)
  ELSE
    SELECTPLAKCELL(J,CELL1,X,V1,SINGLESELFLAG,
                  TIMES,C1VAR,C2VAR,
                  MINTERM,INCOST);
  END;
END;

PROCEDURE APPLYRULE(J:INTEGER;VAR NOSELECTED,CELL:INTEGER;
                   X,I:INTEGER;VAR SELECTFLAG,SINGLEFNFLAG:BOOLEAN;
                   VAR V1,V2:STATEARRAY;VAR V3:STATEKCELLARRAY;
                   TIMES:FUNCTIONARRAY;C1VAR,C2VAR:BINARYARRAY;
                   MINTERM,INCOST:INTEGER);

(* Apply the Selection and the Elimination rules to the given
k-cells K1 & K2 *)

VAR
COND1,COND3      : BOOLEAN;
COND2A,COND2B   : BOOLEAN;
COND4A,COND4B   : BOOLEAN;
COND5A,COND5B   : BOOLEAN;
CELL1,CELL2     : INTEGER;
J1,J2,J3        : INTEGER;
SINGLEFLAG       : BOOLEAN;
SELENDFLAG      : BOOLEAN;
BOUNDFLAG       : BOOLEAN;
CELL1CONDITION  : BOOLEAN;
CELL2CONDITION  : BOOLEAN;

PROCEDURE CHECKFORSINGLEKCELL(VAR SINGLEFLAG:BOOLEAN;
                             X,I:INTEGER;
                             V3:STATEKCELLARRAY);

(* Check if the only k-cell covering the given reference state *)

VAR
  J4, COUNT      : INTEGER;

BEGIN
  J4 := 0;
  COUNT := 0;
  REPEAT
    J4 := J4 + 1;
    IF (V3[X,I,J4] > 0) THEN
      COUNT := COUNT + 1;
    UNTIL ((J4 = MAXALTCELLS) OR (V3[X,I,J4] = 0)
           OR (COUNT > 1));

    IF (COUNT > 1) THEN
      SINGLEFLAG := FALSE;
    END;

  BEGIN (* APPLYRULE *)
    SELECTFLAG := FALSE;
    SINGLEFNFLAG := FALSE;
    REPEAT
      SELENDFLAG := TRUE;
      J1 := 0;

```



```

J := 0;
JJ := 0;
REPEAT
  J := J + 1;
  IF (V3[X,I,J] > 0) THEN
    BEGIN
      JJ := JJ + 1;
      VK[JJ] := V3[X,I,J];
    END;
UNTIL ((V3[X,I,J] = 0) OR (J = MAXALTCELLS));
J := 0;
REPEAT
  J := J + 1;
  FOR II := 1 TO TMTX[X] DO
    BEGIN
      IF ((V1[X,II] = -2) AND (II <> I)) THEN
        BEGIN
          DOWORDAND(S2[X,VK[JJ]],V[X,II]);
          IF (WORDAND = S1[X,VK[JJ]]) THEN
            VV[II] := V[X,II];
          END;
        END;
      UNTIL ((VK[JJ] = 0) OR (J = MAXKCELLS));
    II := 0;
    CANCELFLAG := FALSE;
    REPEAT
      II := II + 1;
      IF (VV[II] >= 0) THEN
        BEGIN
          JJ := 0;
          CANCELFLAG := TRUE;
          REPEAT
            JJ := JJ + 1;
            CHECKIFOTHERKCELL(VK,V3[X,II,JJ],OTHERFLAG);
            IF (OTHERFLAG) THEN
              CANCELFLAG := FALSE;
            UNTIL ((NOT(CANCELFLAG)) OR (V3[X,II,JJ] = 0)
              OR (JJ = MAXALTCELLS));
          END;
        UNTIL ((II = TMTX[X]) OR (CANCELFLAG));
        IF (CANCELFLAG) THEN
          V1[X,I] := 2;
        END;
    BEGIN (* APPLYRULE1 *)
    REPEAT
      NOSELECTED := 0;
      FOR X := 1 TO NOOFFUNCTIONS DO
        BEGIN
          FOR I := 1 TO TMTX[X] DO
            BEGIN
              BOUNDIFDOMINANT(X,I,V1,V3);
              IF (V1[X,I] = -2) THEN
                BEGIN
                  REPEAT
                    V2TEMP1 := V2[X,I];
                    APPLYRULE(J,NOSELECTED,CELL,X,I,SELECTFLAG,
                      SINGLEFNFLAG,V1,V2,V3,TIMES,C1VAR,C2VAR,
                      MINTERM,INCOST);
                    IF (SELECTFLAG) THEN
                      BEGIN
                        SELECTTHISKCELL(J,CELL,X,V1,SINGLEFNFLAG,
                          TIMES,C1VAR,C2VAR,
                          MINTERM,INCOST);
                        NOSELECTED := NOSELECTED + 1;
                      END;
                END;
            END;
          END;
        END;
      END;
    END;

```

```

                UNTIL ((SELECTFLAG) OR (FORMENDFLAG)
                    OR (V2[X,I]=V2TEMP1));
            END;
        END;
    END;
UNTIL ((NOSELECTED = 0) OR (FORMENDFLAG));
IF (NOT(FORMENDFLAG)) THEN
    BEGIN
        X := 0;
        ALLSTATESCOVERED := TRUE;
        REPEAT
            X := X + 1;
            I := 0;
            REPEAT
                I := I + 1;
                IF (V4[X,I,J] = 0) THEN
                    ALLSTATESCOVERED := FALSE;
                    UNTIL ((I = TMTX[X]) OR (NOT(ALLSTATESCOVERED)));
                    UNTIL ((X = NOOFFUNCTIONS) OR (NOT(ALLSTATESCOVERED)));
                    IF (NOT(ALLSTATESCOVERED)) THEN
                        BEGIN
                            APPLYRULE2(J,V1,V2,V3,NOSELECTED,TIMES,
                                C1VAR,C2VAR,MINTERM,INCOST);
                        END;
                    IF ((NOT(FORMENDFLAG)) AND
                        (S5TOT[J] < MINNOOFOPRN)) THEN
                        BEGIN
                            IF (OPTION = 2) THEN
                                BEGIN
                                    MINNOOFOPRN := S5TOT[J];
                                    TERMMIN := MINTERM;
                                    COSTIN := INCOST;
                                    FOR JV := 1 TO N DO
                                        BEGIN
                                            VARC1[JV] := C1VAR[JV];
                                            VARC2[JV] := C2VAR[JV];
                                        END;
                                    END
                                ELSE
                                    MINNOOFOPRN := S5TOT[J];
                                END;
                            END;
                        END;
        END;
    END;
PROCEDURE APPLYRULE2;

(* Apply the Branching rule in case no more k-cell selection
possible *)

TYPE
    TEMPARRAY      = ARRAY[1..MAXFUNCTIONS,
                        0..MAXVARIABLES]OF INTEGER;

VAR
    ENDRULE2      : BOOLEAN;
    SINGLEKCELLFLAG : BOOLEAN;
    I,I1,I2,J1,J2 : INTEGER;
    II,JJ,X2,X,XX : INTEGER;
    CELL          : INTEGER;
    V4TEMP,V1TEMP : STATEARRAY;
    S3TEMP,S4TEMP : KCELLARRAY;
    SSTEMP        : BOOLKCELLARRAY;
    S5TOTTEMP     : INTEGER;
    S5TEMP        : FUNCTIONARRAY;
    S6TEMP        : TEMPARRAY;
    MINTEMP,INTEMP : INTEGER;

```

```

TIMETEMP      : FUNCTIONARRAY;
C1TEMP,C2TEMP : BINARYARRAY;
COND4A,COND4B : BOOLEAN;
COND5A,COND5B : BOOLEAN;

PROCEDURE GETSTATETOSELECT(VAR X,I:INTEGER;VAR V1,V2:STATEARRAY;
                           VAR V3:STATEKCELLARRAY);

(* Get a reference state with minimum alternate covers for
   branching *)

VAR
  SMALLKCELL      : INTEGER;
  SMALLX,SMALLI   : INTEGER;

BEGIN
  SMALLKCELL := MAXKCELLS;
  SMALLX := 0;
  SMALLI := 0;
  X := 0;
  REPEAT
    X := X + 1;
    I := 0;
    REPEAT
      I := I + 1;
      IF ((V1[X,I] = -2) AND (V2[X,I] < SMALLKCELL)) THEN
        BEGIN
          SMALLKCELL := V2[X,I];
          SMALLX := X;
          SMALLI := I;
        END;
      UNTIL (I = TMTX[X]);
    UNTIL (X = NOOFFUNCTIONS);
    X := SMALLX;
    I := SMALLI;
  END;

BEGIN (* APPLYRULE2 *)
  ENDRULE2 := FALSE;
  X := 0;
  I := 0;
  GETSTATETOSELECT(X,I,V1,V2,V3);
  FOR II := 1 TO N DO
    BEGIN
      C1TEMP[II] := C1VAR[II];
      C2TEMP[II] := C2VAR[II];
    END;
  MINTEMP := MINTERM;
  INTEMP := INCOST;
  S5TOTTEMP := S5TOT[J];
  FOR XX := 1 TO NOOFFUNCTIONS DO
    BEGIN
      TIMETEMP[XX] := TIMES[XX];
      S5TEMP[XX] := S5[XX,J];
      FOR II := 1 TO N DO
        S6TEMP[XX,II] := S6[XX,II,J];
        FOR II := 1 TO TMTX[XX] DO
          BEGIN
            V4TEMP[XX,II] := V4[XX,II,J];
            V1TEMP[XX,II] := V1[XX,II];
          END;
        FOR JJ := 1 TO NOMAXKCELLS DO
          BEGIN
            S3TEMP[XX,JJ] := S3[XX,JJ,J];
            S4TEMP[XX,JJ] := S4[XX,JJ,J];
            S5TEMP[XX,JJ] := S5[XX,JJ,J];
          END;
        END;
      END;
    END;
  END;

```

```

END;
J1 := 1;
I1 := 0;
REPEAT
  REPEAT
    I1 := I1 + 1;
  UNTIL ((V3[X,I,I1] >= 0) OR (I1 = MAXALTCELLS));
  IF (V3[X,I,I1] <= 0) THEN
    ENDRULE2 := TRUE
  ELSE
    BEGIN
      CELL := V3[X,I,I1];
      SINGLEKCELLFLAG := TRUE;
      SELECTTHISKCELL(J,CELL,X,V1,SINGLEKCELLFLAG,
        TIMES,C1VAR,C2VAR,MINTERM,INCOST);
      IF (NOT(FORMENDFLAG)) THEN
        APPLYRULE1(J,NOSELECTED,V1,V2,V3,TIMES,C1VAR,C2VAR,
          MINTERM,INCOST);
      IF (J1 < V2[X,I]) THEN
        BEGIN
          IF (NOT(FORMENDFLAG)) THEN
            J := J + 1;
            MINTERM := MINTEMP;
            INCOST := INTEMP;
            S5TOT[J] := S5TOTTEMP;
            FOR II := 1 TO N DO
              BEGIN
                C1VAR[II] := C1TEMP[II];
                C2VAR[II] := C2TEMP[II];
              END;
            FOR X2 := 1 TO NOOFFUNCTIONS DO
              BEGIN
                TIMES[X2] := TIMETEMP[X2];
                S5[X2,J] := S5TEMP[X2];
                FOR I2 := 1 TO N DO
                  S6[X2,I2,J] := S6TEMP[X2,I2];
                FOR I2 := 1 TO TMTX[X2] DO
                  BEGIN
                    V4[X2,I2,J] := V4TEMP[X2,I2];
                    V1[X2,I2] := V1TEMP[X2,I2];
                  END;
                FOR I2 := 1 TO NOMAXKCELLS DO
                  BEGIN
                    S3[X2,I2,J] := S3TEMP[X2,I2];
                    S4[X2,I2,J] := S4TEMP[X2,I2];
                    SS[X2,I2,J] := SSTEMP[X2,I2];
                  END;
                END;
                J1 := J1 + 1;
                NOALTFORMS := NOALTFORMS + 1;
              END
            ELSE
              ENDRULE2 := TRUE;
            END;
          UNTIL (ENDRULE2);
        END;
      END;
    END;
  REPEAT
    PROCEDURE FINDSINGLEMINFORM(J,NOSELECTED:INTEGER;
      V1,V2:STATEARRAY;
      V3:STATEKCELLARRAY;
      TIMES:FUNCTIONARRAY;
      C1VAR,C2VAR:BINARAYARRAY;
      MINTERM,INCOST:INTEGER);

      (* Find single function minimal forms for all functions *)

      PROCEDURE EXCHANGEDATA(VAR V1,V2:STATEARRAY;

```



```

                                VAR V3:STATEKCELLARRAY;
                                XX : INTEGER);

(* Get the data related to the given function *)

VAR
    II,JJ,TEMP      : INTEGER;

BEGIN
    FOR II := 1 TO MAXSTATES DO
        BEGIN
            TEMP := V[XX,II];
            V[XX,II] := V[1,II];
            V[1,II] := TEMP;
            TEMP := V1[XX,II];
            V1[XX,II] := V1[1,II];
            V1[1,II] := TEMP;
            TEMP := V2[XX,II];
            V2[XX,II] := V2[1,II];
            V2[1,II] := TEMP;
            FOR JJ := 1 TO MAXALTCELLS DO
                BEGIN
                    TEMP := V3[XX,II,JJ];
                    V3[XX,II,JJ] := V3[1,II,JJ];
                    V3[1,II,JJ] := TEMP;
                END;
            FOR JJ := 1 TO MAXALTFORMS DO
                BEGIN
                    TEMP := V4[XX,II,JJ];
                    V4[XX,II,JJ] := V4[1,II,JJ];
                    V4[1,II,JJ] := TEMP;
                END;
            END;
        FOR II := 1 TO MAXKCELLS DO
            BEGIN
                TEMP := S1[XX,II];
                S1[XX,II] := S1[1,II];
                S1[1,II] := TEMP;
                TEMP := S2[XX,II];
                S2[XX,II] := S2[1,II];
                S2[1,II] := TEMP;
                FOR JJ := 1 TO MAXALTFORMS DO
                    BEGIN
                        TEMP := S3[XX,II,JJ];
                        S3[XX,II,JJ] := S3[1,II,JJ];
                        S3[1,II,JJ] := TEMP;
                        TEMP := S4[XX,II,JJ];
                        S4[XX,II,JJ] := S4[1,II,JJ];
                        S4[1,II,JJ] := TEMP;
                    END;
                END;
            TEMP := T[XX];
            T[XX] := T[1];
            T[1] := TEMP;
            TEMP := TX[XX];
            TX[XX] := TX[1];
            TX[1] := TEMP;
            TEMP := TMTX[XX];
            TMTX[XX] := TMTX[1];
            TMTX[1] := TEMP;
        END;

PROCEDURE GATESINGLEMINFORM(J,NOSELECTED:INTEGER;
                            V1,V2:STATEARRAY;
                            V3:STATEKCELLARRAY;
                            TIMES:FUNCTIONARRAY;
                            C1VAR,C2VAR:BINARARRAY;

```

```

                                MINTERM, INCOST:INTEGER);

(* Find single-function minimal forms based on doide-gate cost
function *)

VAR
  IX, NOOFFNTEMP, XX : INTEGER;
  C1TEM, C2TEM       : BINARYARRAY;
  SUMMIN, SUMIN      : INTEGER;

BEGIN
  NOOFFNTEMP := NOOFFFUNCTIONS;
  NOOFFFUNCTIONS := 1;
  SINGLESOLN := 0;
  FOR XX := 1 TO NOOFFNTEMP DO
    BEGIN
      EXCHANGEDATA(V1, V2, V3, XX);
      INITIALIZEOUTPUTDATA(J, NOSELECTED, TIMES,
                           C1VAR, C2VAR, MINTERM, INCOST);
      MINNOOFOPRN := (N + 1) * F(N);
      APPLYRULE1(J, NOSELECTED, V1, V2, V3, TIMES, C1VAR, C2VAR,
                 MINTERM, INCOST);
      MINSINGLEFN[XX] := MINNOOFOPRN;
      SINGLESOLN := SINGLESOLN + MINNOOFOPRN;
      EXCHANGEDATA(V1, V2, V3, XX);
    END;
  NOOFFFUNCTIONS := NOOFFNTEMP;
  MINNOOFOPRN := SINGLESOLN;
END;

PROCEDURE PLASINGLEMINFORM(J, NOSELECTED:INTEGER;
                          V1, V2:STATEARRAY;
                          V3:STATEKCELLARRAY;
                          TIMES:FUNCTIONARRAY;
                          C1VAR, C2VAR: BINARYARRAY;
                          MINTERM, INCOST:INTEGER);

(* Find single function minimal forms based on PLA area cost
function *)

VAR
  IX, NOOFFNTEMP, XX : INTEGER;
  C1TEM, C2TEM       : BINARYARRAY;
  SUMMIN, SUMIN      : INTEGER;

BEGIN
  FOR IX := 1 TO N DO
    BEGIN
      C1TEM[IX] := FALSE;
      C2TEM[IX] := FALSE;
    END;
  NOOFFNTEMP := NOOFFFUNCTIONS;
  NOOFFFUNCTIONS := 1;
  SINGLESOLN := 0;
  SUMMIN := 0;
  FOR XX := 1 TO NOOFFNTEMP DO
    BEGIN
      EXCHANGEDATA(V1, V2, V3, XX);
      INITIALIZEOUTPUTDATA(J, NOSELECTED, TIMES,
                           C1VAR, C2VAR, MINTERM, INCOST);
      MINNOOFOPRN := (2 * N + NOFN) * F(N);
      APPLYRULE1(J, NOSELECTED, V1, V2, V3, TIMES, C1VAR, C2VAR,
                 MINTERM, INCOST);
      FOR IX := 1 TO N DO
        BEGIN
          IF (VARC1[IX]) THEN
            C1TEM[IX] := VARC1[IX];
        END;
      END;
    END;
  END;

```

```

                IF (VARC2[IX]) THEN
                    C2TEM[IX] := VARC2[IX];
                END;
                MINSINGLEFN[XX] := TERMIN;
                SUMMIN := SUMMIN + TERMIN;
                EXCHANGEDATA(V1,V2,V3,XX);
            END;
            SUMIN := 0;
            FOR IX := 1 TO N DO
                BEGIN
                    IF (C1TEM[IX]) THEN
                        SUMIN := SUMIN + 1;
                    IF (C2TEM[IX]) THEN
                        SUMIN := SUMIN + 1;
                    END;
                FOR XX := 1 TO NOOFFNTEMP DO
                    MINSINGLEFN[XX] := MINSINGLEFN[XX] * (SUMIN + NOFN);
                SINGLESOLN := SUMMIN * (SUMIN + NOFN);
                NOOFFFUNCTIONS := NOOFFNTEMP;
                MINNOOFOPRN := SINGLESOLN;
            END;

            BEGIN (* FINDSINGLEMINFORM *)
                IF (OPTION = 1) THEN
                    GATESINGLEMINFORM(J,NOSELECTED,V1,V2,V3,
                        TIMES,C1VAR,C2VAR,MINTERM,INCOST)
                ELSE
                    PLASINGLEMINFORM(J,NOSELECTED,V1,V2,V3,
                        TIMES,C1VAR,C2VAR,MINTERM,INCOST);
            END;

            BEGIN (* FINDMINIMUMFORM *)
                FINDSINGLEMINFORM(J,NOSELECTED,V1,V2,V3,
                    TIMES,C1VAR,C2VAR,MINTERM,INCOST);
                INITIALIZEOUTPUTDATA(J,NOSELECTED,TIMES,C1VAR,C2VAR,MINTERM,INCOST);
                APPLYRULE1(J,NOSELECTED,V1,V2,V3,TIMES,C1VAR,C2VAR,MINTERM,INCOST);
            END;

            PROCEDURE PRINTRESULTS1;

            (* Print all maximal k-cells generated, if required *)

            VAR
                I,J,K,X          : INTEGER;

            BEGIN
                FOR X := 1 TO NOOFFFUNCTIONS DO
                    BEGIN
                        WRITELN;
                        WRITELN;
                        WRITELN('FUNCTION':10,X:3);
                        WRITELN;
                        WRITELN('ALL MAXIMAL K-CELLS FOR EACH STATE');
                        WRITELN;
                        WRITELN('STATE':10,'TYPE':7,'MAX. K-CELLS':25);
                        WRITELN;
                        FOR I := 1 TO T[X] DO
                            BEGIN
                                WRITE(V[X,I]:8,V1[X,I]:8);
                                IF (V2[X,I] >= 1) THEN
                                    BEGIN
                                        FOR J := 1 TO V2[X,I] DO
                                            WRITE(V3[X,I,J]:5);
                                        END;
                                    END;
                            END;
                        END;
                    END;
                END;
            END;

```

```

        WRITELN;
    END;
    WRITELN;
    WRITELN('DECIMAL GEN-CELL AND K-CELL INDICES FOR EACH K-CELL');
    WRITELN;
    WRITELN('MAX K-CELL':15,'GEN-CELL INDEX':15,'K-CELL INDEX':15);
    WRITELN;
    FOR K := 1 TO NOMAXKCELLS DO
    BEGIN
        IF (S2[X,K] <> -1) THEN
        BEGIN
            WRITE(K:11,S2[X,K]:13,S1[X,K]:16,'      ');
            DECODEINDICES(S2[X,K],S1[X,K]);
            WRITELN;
        END;
    END;
END;
END;

PROCEDURE PRINTRESULTS2;

(* Print all minimal forms and their cost *)

VAR
    I,J,K,X          : INTEGER;
    FLAG              : BOOLEAN;

BEGIN
    WRITELN;
    WRITELN;
    WRITELN ('SOLUTION FOR THE FOLLOWING COMBINATION :');
    WRITELN;
    FOR I := 1 TO NOOFFUNCTIONS DO
    BEGIN
        WRITE ('FUNCTION',I:2,' : ');
        IF (COMBFLAG[I]) THEN
            WRITELN ('NON-INVERTED')
        ELSE
            WRITELN ('INVERTED');
        WRITELN;
    END;
    WRITELN;
    WRITELN;
    WRITELN;
    WRITELN ('SINGLE-OUTPUT MINIMUM COST FOR EACH FUNCTION: ');
    WRITELN;
    FOR X := 1 TO NOOFFUNCTIONS DO
    BEGIN
        WRITELN;
        WRITELN ('FUNCTIION ',X:2,' ',MINSINGLEFN[X]:2);
    END;
    WRITELN;
    WRITELN;
    WRITELN ('SINGLE-OUTPUT MINIMUM COST SUM = ',SINGLESOLN:2);
    WRITELN;
    WRITELN;
    WRITELN ('MULTIPLE OUTPUT MINIMIZATION :');
    WRITELN;
    WRITELN;
    WRITELN('NO OF TERMINAL NODES IN THE SEARCH TREE = ',NOALTFORMS:5);
    WRITELN;
    WRITELN;
    WRITELN('MULTI-OUTPUT MINIMUM COST = ',MINNOOFOPRN:8);
    WRITELN;
    WRITELN;
    WRITELN('MINIMAL FORM/S WITH MINIMUM COST :');
    J := 1;

```

```

WHILE ((S5TOT[J] > 0) AND (J <= MAXALTFORMS)) DO
  BEGIN
    IF (S5TOT[J] = MINNOOFOPRN) THEN
      BEGIN
        WRITELN;
        WRITELN;
        WRITELN('FORM ':8,J:2,' ':5);
        WRITELN;
        FOR X := 1 TO NOOFFUNCTIONS DO
          BEGIN
            WRITE('FUNCTION':8,X:2,' ':5);
            FLAG := FALSE;
            FOR I := 1 TO NOMAXKCELLS DO
              BEGIN
                IF (S4[X,I,J] <> 0) THEN
                  BEGIN
                    IF (FLAG) THEN
                      WRITE (' + ');
                    DECODEINDICES((ABS(S4[X,I,J])),S3[X,I,J]);
                    FLAG := TRUE;
                  END;
                END;
              WRITELN;
              WRITELN;
            END;
          WRITELN;
        END;
        J := J + 1;
      END;
    END;
  BEGIN (* MULTIMIN *)
    READINPUTDATA;
    FOR ITRN := 1 TO COMBRANGE DO
      BEGIN
        GETACOMBINATION;
        FINDALLMAXIMALKCELLS;
        (* PRINTRESULTS1; *)
        FOR OPTION := 1 TO NOCOSTFN DO
          BEGIN
            FINDMINIMUMFORM;
            PRINTRESULTS2;
          END;
        END;
      END;
    END.
  
```

Vita

The author was born on February 14th, 1963 in Calcutta, India. After graduating from St. Xavier's College, Bombay, with his High School diploma, he joined College of Engineering, Poona, to take up Electronics and Telecommunication Engineering. He received his Bachelor of Engineering degree from University of Poona in 1984. He was with Wipro Information Technology for a brief period of ten months, during which he served as a Customer Support Engineer. In April 1985, he started his graduate studies in Dept. of Electrical Engineering, Virginia Polytechnic Institute and State University, with emphasis on Computer Engineering. He received his Master of Science from the same in 1987.

