

AD-A239 470



2

NAVAL POSTGRADUATE SCHOOL

Monterey, California

DTIC
ELECTE
AUG 19 1991
S D



THESIS

APPLICATION OF THE CONSTRAINED
IMPLICANTS SET CONCEPT TO THE
MINIMIZATION OF BINARY FUNCTIONS

by

Ugur Ozkan

September 1990

Thesis co-advisor:

Chyan Yang

Thesis co-advisor:

Jon T. Butler

Approved for public release; distribution is unlimited.

91 8 16 021

91-08085



Unclassified

security classification of this page

REPORT DOCUMENTATION PAGE

1a Report Security Classification Unclassified			1b Restrictive Markings		
2a Security Classification Authority			3 Distribution/Availability of Report		
2b Declassification Downgrading Schedule			Approved for public release; distribution is unlimited.		
4 Performing Organization Report Number(s)			5 Monitoring Organization Report Number(s)		
6a Name of Performing Organization Naval Postgraduate School		6b Office Symbol (if applicable) 62	7a Name of Monitoring Organization Naval Postgraduate School		
6c Address (city, state, and ZIP code) Monterey, CA 93943-5000			7b Address (city, state, and ZIP code) Monterey, CA 93943-5000		
8a Name of Funding/Sponsoring Organization		8b Office Symbol (if applicable)	9 Procurement Instrument Identification Number		
8c Address (city, state, and ZIP code)			10 Source of Funding Numbers		
			Program Element No	Project No	Task No
			Work Unit Accession No		
11 Title (include security classification) APPLICATION OF THE CONSTRAINED IMPLICANTS SET CONCEPT TO THE MINIMIZATION OF BINARY FUNCTIONS					
12 Personal Author(s) Ugur Ozkan					
13a Type of Report Master's Thesis		13b Time Covered From To		14 Date of Report (year, month, day) September 1990	15 Page Count 89
16 Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.					
17 COSATI Codes			18 Subject Terms (continue on reverse if necessary and identify by block number)		
Field	Group	Subgroup	Constrained Implicants Set Concept, Binary Minimization.		
19 Abstract (continue on reverse if necessary and identify by block number)					
Several heuristics and algorithms have been developed to find minimal sum-of-products expressions in binary logic. Most of them use prime implicants during minimization process. An efficient search strategy has been developed for finding minimal sum-of-products expressions for multiple-valued logic (MVL) functions by using the constrained implicants set concept. The search space can be considerably reduced over the only other known exact minimization technique and exhaustive search. The primary goals of this research are to: (1) examine whether the constrained implicant set concept can be efficiently used in binary logic, and; (2) develop a heuristic called the constrained implicant set heuristic (CISH). The general idea of the CISH is to select the minterm with the least implicant cover size and find the implicant with the largest minterm coverage that covers a selected minterm. In this research, the implementation of the CISH is presented, the performance analysis of the CISH is shown by comparing with other heuristics (Maximum Implicant Heuristic, Espresso II) with respect to the average number of the product terms, the average computation time, and the average memory usage.					
20 Distribution Availability of Abstract			21 Abstract Security Classification		
☒ unclassified unlimited ☐ same as report ☐ DTIC users			Unclassified		
22a Name of Responsible Individual Chyan Yang			22b Telephone (include Area code) (408) 646-2266		22c Office Symbol 62Ya

Approved for public release; distribution is unlimited.

Application of the Constrained Implicants
Set Concept to the Minimization of
Binary Functions

by

Ugur Ozkan
Lieutenant Junior Grade, Turkish Navy
B.S., Turkish Naval Academy, 1984

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING

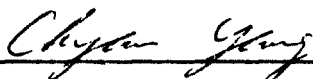
from the

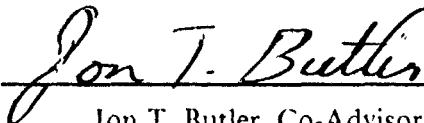
NAVAL POSTGRADUATE SCHOOL
September 1990

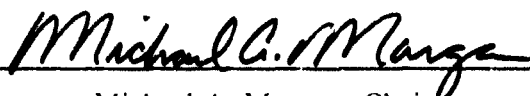
Author:


Ugur Ozkan

Approved by:


Chyan Yang, Thesis Advisor


Jon T. Butler, Co-Advisor


Michael A. Morgan, Chairman,
Department of Electrical and Computer Engineering

ABSTRACT

Several heuristics and algorithms have been developed to find minimal sum-of-products expressions in binary logic. Most of them use prime implicants during minimization process.

An efficient search strategy has been developed for finding minimal sum-of-products expressions for multiple-valued logic (MVL) functions by using the constrained implicants set concept. The search space can be considerably reduced over the only other known exact minimization technique and exhaustive search.

The primary goals of this research are to: (1) examine whether the constrained implicant set concept can be efficiently used in binary logic, and; (2) develop a heuristic called the constrained implicant set heuristic (CISH). The general idea of the CISH is to select the minterm with the least implicant cover size and find the implicant with the largest minterm coverage that covers a selected minterm.

In this research, the implementation of the CISH is presented, the performance analysis of the CISH is shown by comparing with other heuristics (Maximum Implicant Heuristic, Espresso II) with respect to the average number of the product terms, the average computation time, and the average memory usage.



Accession For	
NTIS	CRA&I ☒
DTIC	TAB ☐
Unannounced ☐	
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

THESIS DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government. The views and opinions of the author expressed herein do not necessarily state or reflect those of the United States Government and shall not be used for advertising or product endorsement purposes.

TABLE OF CONTENTS

I. INTRODUCTION	1
A. MOTIVATION	1
B. NATURE OF THE PROBLEM	2
C. THESIS OUTLINE	3
II. BACKGROUND AND DEFINITIONS	4
A. DEFINITIONS IN BINARY LOGIC	4
B. BACKGROUND IN MULTIPLE-VALUED LOGIC	6
C. DEFINITIONS USED IN CISH	12
D. MEASURES USED IN CISH	13
1. Clustering Factor	13
2. Use of Implicants	13
3. Effects of Don't Care Terms	16
III. CONSTRAINED IMPLICANT SET HEURISTIC	19
A. INTRODUCTION	19
B. MINIMIZATION ALGORITHM OF CISH	19
1. Initialization	19
2. Recursive Algorithm	20
3. Extended Search Technique (EST)	20
C. EXPLANATION OF CISH	20
D. A WALKTHROUGH EXAMPLE OF CISH	29

IV. COMPARATIVE RESULTS	35
A. PERFORMANCE COMPARISON	35
B. TIMING COMPARISON	36
C. MEMORY COMPARISON	38
V. DISCUSSIONS AND CONCLUSIONS	40
A. DISCUSSIONS	40
B. CONCLUSIONS	41
APPENDIX A. ABSOLUTE MINIMIZATION ALGORITHM	43
APPENDIX B. MAXIMUM IMPLICANT HEURISTIC	46
APPENDIX C. ESPRESSO II	49
APPENDIX D. AVERAGE NUMBER OF PRODUCT TERMS	52
APPENDIX E. AVERAGE COMPUTATION TIME	53
APPENDIX F. AVERAGE MEMORY USAGE	54
APPENDIX G. PROGRAM LISTING	55
LIST OF REFERENCES	75
INITIAL DISTRIBUTION LIST	77

LIST OF TABLES

Table 1. INFORMATION ABOUT IMPLICANTS	31
Table 2. INFORMATION ABOUT MINTERMS	31
Table 3. AVERAGE NUMBER OF PRODUCT TERMS	52
Table 4. AVERAGE COMPUTATION TIME (SEC)	53
Table 5. AVERAGE TOTAL MEMORY USAGE (KBYTE)	54

LIST OF FIGURES

Figure 1. Example Function for Creating Search Space	8
Figure 2. Search Space for Absolute Minimization	10
Figure 3. Search Space for the Constrained Implicants Set	11
Figure 4. An Example on Calculation of $ICS(\alpha)$	14
Figure 5. Use of Different Sets to Calculate $ICS(\alpha)$	15
Figure 6. Special Case in Calculation of $ICS(\alpha)$	17
Figure 7. Importance of Selection of Minterm	23
Figure 8. Application of Extended Search Technique	27
Figure 9. Constrained Implicants in Extended Search Technique	28
Figure 10. Instructive Example for CISII	30
Figure 11. Search Space and Constrained Implicant Paths	34
Figure 12. Average Number of Product Terms	36
Figure 13. Average Computation Time	37
Figure 14. Average Total Memory Usage	38

ACKNOWLEDGEMENTS

I would like to offer special thanks to Professor Chyan Yang and Professor Jon T. Butler for their patient guidance and knowledge during various developments of my thesis. I wish also to thank my wife, Nurcan and my daughter Melis Gizem for their support and understanding in these years of studying at Naval Postgraduate School.

I. INTRODUCTION

A. MOTIVATION

The primary goal of good design in binary logic is to find a realization of a given function at minimal cost. There have been several different costs in binary logic design. The number of the gates is a commonly used cost. Another cost is the number of product terms used in the sum-of-products expression of the given function. This criterion has become especially important in recent years with the introduction of PAL's (programmable array logic) and PLA's (programmable logic arrays). There has been a corresponding increase in interest in algorithms for finding the fewest number of product terms needed to realize the given function.

An efficient search strategy has been developed for finding a minimal sum-of-products expression by using the constrained implicant set concept in multiple-valued logic. The search space can be considerably reduced over the only known exact minimization techniques by using the constrained implicant set concept [Ref. 1]. A primary motivation of this thesis is to see if the constrained implicant set concept can be extended to binary logic.

A new heuristic for binary functions is described which is called the constrained implicant set heuristic (CISH). An analysis of the performance of the CISH has been done by comparing the results of the average number of sum-of-products term, computation time, and memory usage with two existing algorithms 1) Maximum Implicant Heuristic (MIH) and 2) Espresso II.

B. NATURE OF THE PROBLEM

The goal of logic minimization is to find a minimal sum-of-products expression of a binary function. This problem has received considerable attention for some time. Early methods such as Quine-McCluskey [Ref. 2], and iterated consensus [Ref. 3] begins the minimization by finding all prime implicants of the function. The finding of prime implicants is used by most heuristics minimization methods.

The interest in heuristic methods is due to the large computation times required by exact minimizations algorithms. For example, a 10 variable binary function can have as many as 5904 prime implicants while a function with 20 variables can have as many as 174,339,220 prime implicants. The relationship between the number of the prime implicants and number of the variables (n) has been shown as $3^n/n$ in the worst case. [Refs. 4, 5: p. 49]

It is possible to find an exact minimal sum-of-products expression for a logic function with a small number of variables or simple functions with larger number of variable. When the number of the variable increases or the function becomes more complex, then more computation time is needed to extract the exact minimal solution. Sometimes finding nearly minimal solution of a function in shorter computation time has more importance and advantage than finding an exact minimal representation in very long computation time [Ref. 6].

The exact minimal sum-of-products expression can be solved by enumeration. In this method, all possible solutions are tried and then the one having the fewest number of sum-of-products is chosen. This method needs very long computation time due to large number of possibilities of potential solutions. Besides, it has been shown that extracting minimal sum-of-products solutions from a complete set of prime implicants is

an NP-hard problem. The best known algorithms for such problems require exponential time [Refs. 1, 7: p. 246].

C. THESIS OUTLINE

Notation and definitions are given in Chapter II. The constrained implicant set heuristic is introduced in Chapter III. Chapter IV and Chapter V discuss the comparison results and performance with Maximum Implicant Heuristic and Espresso II.

II. BACKGROUND AND DEFINITIONS

A. DEFINITIONS IN BINARY LOGIC

In this section, we briefly summarize fundamental definitions used in binary logic [Refs. 2,3].

Definition 1:

A *literal* is a variable or the complement of a variable. Examples: $w, x, \bar{w}, \bar{x}$

Definition 2:

A *product term* is a single literal or a Boolean product of the literals. Examples: $\bar{w}y\bar{z}, \bar{x}\bar{y}\bar{z}, wz$

Definition 3:

A *minterm* is a product term where a literal of each variable appears exactly once. Examples: $\bar{w}x\bar{y}z, wxy\bar{z}$.

Definition 4:

A logic function $F(x_1, x_2, \dots, x_n)$ covers a logic function $P(x_1, x_2, \dots, x_n)$ if for every input combination such that $P=1$, then $F=1$.

Definition 5:

Let I be a product term of function f . If f is nonzero for all minterms covered by I , then I is an *implicant* of the function. Examples: $\bar{w}\bar{z}, \bar{y}z$.

Definition 6:

Let I be an implicant of the function f . I can be said *prime implicant* of the function, if it is not covered by any other implicants of the function f .

Definition 7:

A minterm is said to be *distinguished-1-cell* of f if this minterm is covered by only one implicant of the f .

Definition 8:

A prime implicant I is said to be an *essential prime implicant* if it covers a distinguished-1-cell of f .

Definition 9:

Let A^{ON} , A^{OFF} , A^{DC} be sets of assignments of values to variables of function f respect to minterm values

- A^{ON} is the set of minterms of f .
- A^{OFF} is the set of assignments of values to the variables such that f is 0.
- A^{DC} is the set of assignments of values to variables such that f is don't care.

Definition 10:

A function f can be considered to have a *cycle* if it has more than one minimal sum-of-products expression. Example: see the function in Figure 6 on page 17.

Definition 11:

Let α and β be minterms such that complementing one literal in α yields β . Minterm α and β are called *Direct Neighbors*.

B. BACKGROUND IN MULTIPLE-VALUED LOGIC

The constrained implicant set concept was originally developed from the work of finding absolute minimization in multiple-valued logic by Jon T. Butler and P. Tirumalai [Ref. 1]. Absolute minimization tries to find the absolute minimal realization of a function by doing an exhaustive search of all possible solutions. An algorithm for absolute minimization is introduced in Appendix A.

As stated before, absolute minimization needs considerable computation time. The search space is also very large in absolute minimization. This space can be made smaller by applying a limitation rule: constrain some implicants and establish the constrained implicant set to find minimal solution. The constrained implicant set concept significantly reduces search space as well computation time to get exact minimal solution in multiple-valued logic [Refs. 1,8].

Definition 12 :

$\mathbb{R}(\alpha)$ is a constrained implicant set of minterm α for function f , if

$$\mathbb{R}(\alpha) = \bigcup I$$

where I is an implicant of f . [Ref. 1]

Lemma 1 :

If $\mathbb{R}(\alpha)$ is a constrained implicant set, then every possible sum-of-products expression for f has to contain at least one implicant in $\mathbb{R}(\alpha)$ [Ref. 1].

Definition 13:

$\mathbb{R}(\phi)$ is a minimal constrained implicant set of a function if and only if

$$0 < |\mathbb{R}(\alpha)| \leq |\mathbb{R}(\beta)|$$

for all other minterms β of f , where $\mathbb{R}(\alpha)$ is a constrained implicant set [Ref. 1].

The search space can be represented as a tree where each node represents a function and each edge corresponds to implicant of the upper level function. The root node is the given function to be minimized. The functions at the next level down from the root node can be obtained by subtracting an implicant from the root node. Subtracting an implicant from a function corresponds to setting 1's or don't cares in the function covered by the implicant to don't cares.

If the root function has ξ implicants, the root function has ξ branches or subfunctions. Further, each subfunction has a maximum of $\xi-1$ descendents. It can be seen that when the root function has many implicants, the search space is large. This situation can produce large computation time needed to find the minimal solution of the function. That is, the solution is to try each possible path on the search space where the shortest path (having the fewest number of the implicant) is chosen as the minimal expression of the function [Refs. 1, 8].

By using Lemma 1, the search space can be made smaller. At least one implicant from the constrained implicant set has to be in the minimal sum-of-products expression of the function. When $\mathbb{R}(\alpha)$ is chosen as small as possible, there are fewer choices than for larger size $\mathbb{R}(\alpha)$. This decreases the computation time because there are fewer paths to be examined. [Ref. 8]

Example 1:

This example illustrates the search space of a specific function and the finding of exact minimal solution of a f in search space by using absolute minimization algorithm and constrained implicant set concept.

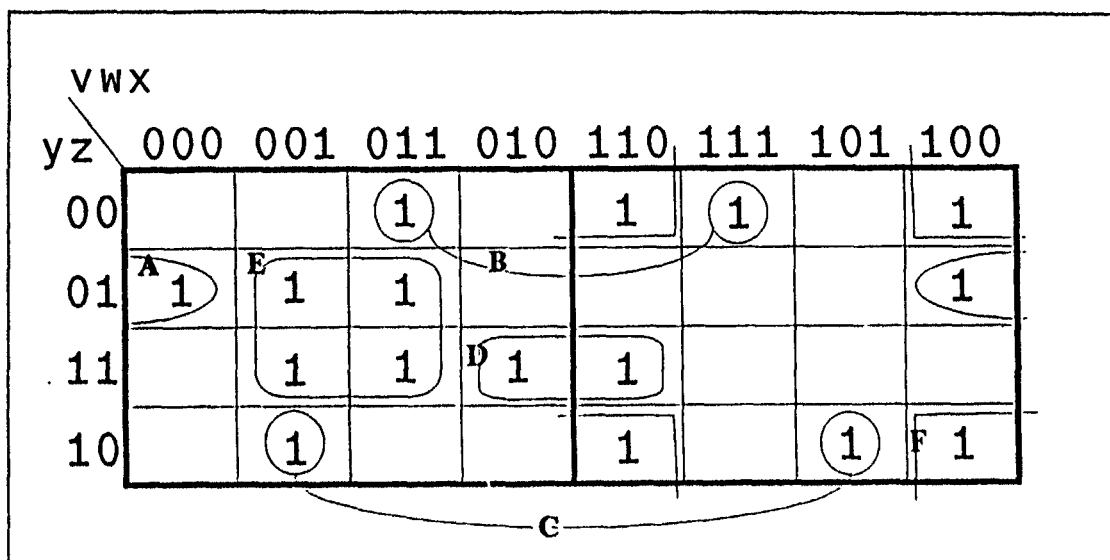


Figure 1. Example Function for Creating Search Space

A five variable function with sixteen minterms is chosen. The function and exact minimal solution is shown in Figure 1.

Let

$$f(v,w,x,y,z) = \sum m(1, 5, 6, 7, 11, 12, 13, 15, 16, 17, 18, 22, 24, 26, 27, 28)$$

The numbers enclosed in parenthesis on the right hand side correspond to minterms in the binary representation. For example, minterm $\alpha=15$ corresponds to assignment 01111. Specific prime implicants in an exact minimal solution are represented by a capital letter as seen in Figure 1.

The search spaces for the given function are illustrated in Figure 2 on page 10 and Figure 3 on page 11. Because of the difficulty of showing all possible search paths, only one search path is shown as example (that which gives the exact minimal solution). Each node function is indicated by a solid dot in search space. These node functions can

be found by subtracting an implicant of the function from the root function as described on page 7.

The node functions on the search path chosen as an example are represented by letters NF. These functions at the next level down from the root node can be obtained by subtracting the implicant (that shown by capital letter in Figure 1) from the root node. These node functions are stated at the bottom of the Figure 2 and Figure 3.

The search space to find exact minimal solution f is shown in Figure 2. The given function f has fourteen prime implicants, and thus it has fourteen branches from the root as described in absolute minimization. The search space is very large, wide and deep, because there are many node functions and subbranches in the search space of f (i.e., 2,162,160 possible branches). All possible search paths must be investigated at each individual node to find the exact minimal solution.

On the other hand, the constrained implicant set concept can be applied to f . A search space has been created for the same function as shown in Figure 3 on page 11. This search space has fewer branches (i.e., 8) and fewer node functions (i.e., 384). Definitions and rules to create a search space and finding the minimal solution is discussed in Section C and Chapter III.

As a results of the comparision of the two search spaces in Figure 2 and Figure 3, we see that although the same implicant are chosen, the absolute minimization algorithm has many more branches at each node function. Applying the constrained implicant set concept to the minimization of function reduces the number of branches and node functions to be examined. Therefore, the program needs less time to find the minimal solution.

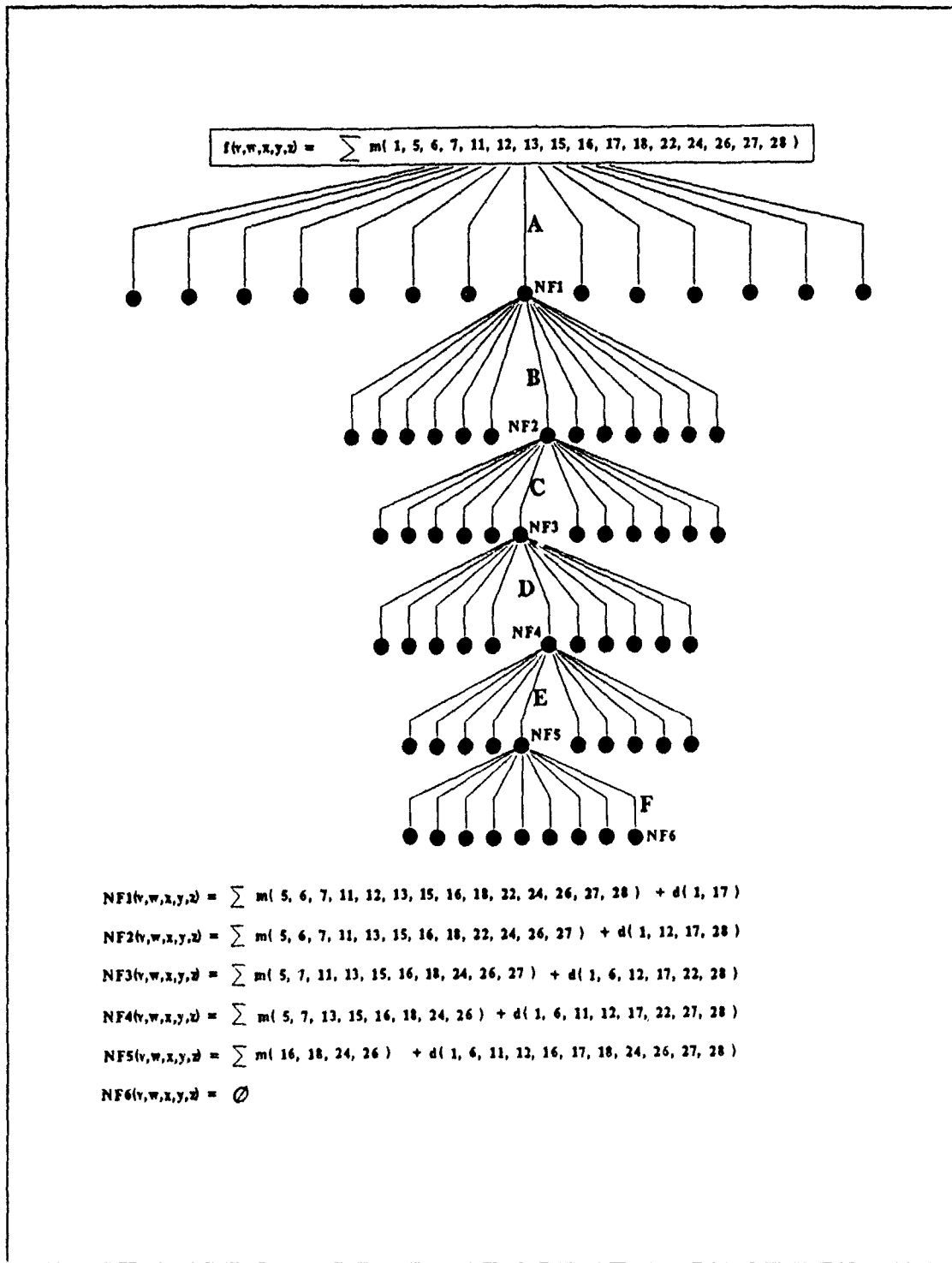


Figure 2. Search Space for Absolute Minimization

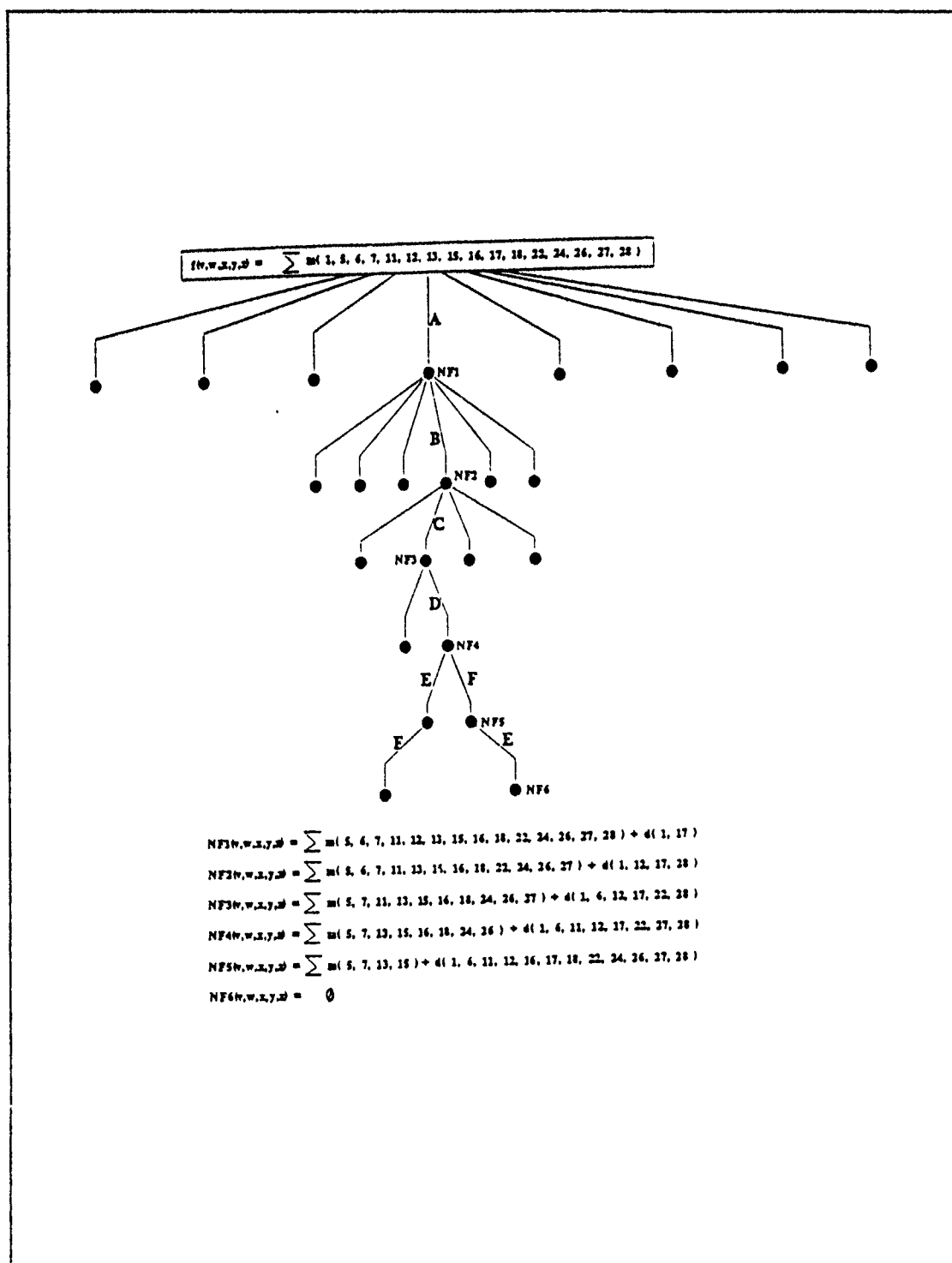


Figure 3. Search Space for the Constrained Implicants Set

C. DEFINITIONS USED IN CISH

In this section, the fundamental measures used in CISH is defined.

Definition 14:

Let $ICS(\alpha)$ denote implicant cover size of minterm α . The *implicant cover size* is the number of the implicants that cover minterm α .

When $ICS(\alpha)$ is calculated for minterm α , the minterm α is not counted as an implicant in CISH. On the other hand, there is an exception. That is, if a minterm does not have any direct neighbor, it is counted as an implicant in calculating $ICS(\alpha)$. Therefore, the minterm with no direct neighbor has same $ICS(\alpha)$ (i.e., 1) as the minterm with only one direct neighbor.

Definition 15:

Let $MC(I)$ denote the minterm coverage of implicant I . The *minterm coverage* is the number of the minterms that are covered by I .

$MC(I)$ is used to determine the importance or the cost of I during the minimization process. The cost of the implicant is introduced in the minimization as the number of minterms covered by I . The implicant with the highest $MC(I)$ is chosen in the minimization process.

In CISH, each minterm covered by a selected implicant I turns to a don't care term. Each new don't care term covered by I is subtracted from $MC(I)$. So the don't care term affects the $MC(I)$. Therefore, the $MC(I)$ changes during the minimization process in CISH.

D. MEASURES USED IN CISH

1. Clustering Factor

$ICS(\alpha)$ is a measure of the degree to which other minterms cluster around α . It shows how many minterms with which a minterm α can combine. The lower $ICS(\alpha)$ is, the fewer combinations exist.

Example 2:

To illustrate these definitions, consider the following four variable function with ten minterms. Let

$$f(w,x,y,z) = \sum m(0, 3, 4, 7, 8, 9, 10, 11, 14, 15)$$

The function f is illustrated in Figure 4 on page 14.

The $ICS(\alpha)$'s for each minterm in Figure 4 are shown in the corresponding upper left corner. For example, the minterm $\alpha = 0$ has $ICS(\alpha) = 2$. Two implicants (I_1, I_2) cover this minterm, where $I_1 = \bar{w}\bar{y}\bar{z}$ and $I_2 = \bar{x}\bar{y}\bar{z}$. In general, minterms in the center of a cluster have a higher $ICS(\alpha)$. For example, minterm $\alpha = 11$ is in the middle of a cluster of 1's and it has a high $ICS(\alpha)$ (7), while minterm $\alpha = 4$ is remote and has a low $ICS(\alpha)$ (i.e., 1).

2. Use of Implicants

As an experimental result, using all implicants rather than prime implicants of function f in calculating $ICS(\alpha)$ provides more information about minterms and implicants. We believe that only using the prime implicants set of f may miss some information about clustering of minterm α with neighboring minterms. Therefore, all implicants of given function are counted in $ICS(\alpha)$.

		wx			
		00	01	11	10
yz	00	² 1	¹ 1		³ 1
	01				³ 1
	11	³ 1	³ 1	⁶ 1	⁷ 1
	10			³ 1	⁶ 1

Figure 4. An Example on Calculation of $ICS(\alpha)$

Example 3:

This example illustrates the advantage of the using implicants versus prime implicants to calculate $ICS(\alpha)$. The same function in Example 1 is used. The minterm located at 00101 (i.e., $\alpha = 5$) is chosen as sample minterm to show changing of $ICS(\alpha)$. The $ICS(\alpha)$ of each minterm that is counted by using both prime implicants (Figure 5.a) and all implicants (Figure 5.b) are shown in page 15.

In Figure 5.a, only prime implicants are considered to calculate $ICS(\alpha)$. All minterms have same $ICS(\alpha)$ (i.e., 2). It shows that all minterms have the same clustering with their neighbors. Thus, all of them all equally likely candidates as the starting point for the minimization.

On the other hand, if all possible implicants are counted in $ICS(\sigma)$ we have the the situation shown in Figure 5.b. In that case, the $ICS(\sigma)$ of each minterm is not the

VWX		000	001	011	010	110	111	101	100
yz	00			1 ²		1 ²	1 ²		1 ²
	01	1 ²	1 ²	1 ²					1 ²
	11		1 ²	1 ²	1 ²	1 ²			
	10		1 ²			1 ²		1 ²	1 ²

(a)

VWX		000	001	011	010	110	111	101	100
yz	00			1 ²		1 ⁴	1 ²		1 ⁴
	01	1 ²	1 ⁴	1 ⁴					1 ²
	11		1 ⁴	1 ⁴	1 ²	1 ²			
	10		1 ²			1 ⁴		1 ²	1 ⁴

(b)

Figure 5. Use of Different Sets to Calculate $ICS(\alpha)$

same as in Figure 5.a. The $ICS(\alpha)$ shows more accurately the clustering and coupling strength among the minterms. Besides, the importance of the minterms to be starting point for minimization is significantly changed (i.e., $ICS(\alpha = 5) = 4$). The minimization can start any one of the eight minterms where their $ICS(\alpha) = 2$. These minterms are $\alpha = 1, 6, 11, 12, 17, 22, 27, 28$.

There is a special case that either use of all implicants or prime implicants cannot provide any advantage over other set in calculating $ICS(\alpha)$. This case occurs when the set of all implicants and is identical to the set of all prime implicants; that is, when each implicant of the function is a prime implicant.

Example 4:

This example illustrates the special case in calculation of $ICS(\alpha)$. A four variable function with eight minterms is used as an example. It is shown in Figure 6 on page 17. $ICS(\alpha)$ of each minterm in Figure 6 is shown in the corresponding upper right corner.

Let

$$f(w,x,y,z) = \sum m(0, 2, 4, 5, 10, 11, 13, 15)$$

It can be seen that the $ICS(\alpha)$ is the same for each minterm whether all implicants or prime implicants are counted in $ICS(\alpha)$. Thus, there is no advantage in using prime implicant or all implicant in calculating $ICS(\alpha)$.

3. Effects of Don't Care Terms

From the definition of $ICS(\alpha)$, all implicants should be considered in counting $ICS(\alpha)$. If it is desired to find the $ICS(\alpha)$ during the minimization process, it is necessary to consider all don't care terms as 1 terms. Changing the don't care terms to 1 terms in

		wx			
		00	01	11	10
yz	00	1 ²	1 ²		
	01		1 ²	1 ²	
	11			1 ²	1 ²
	10	1 ²			1 ²

Figure 6. Special Case in Calculation of $ICS(\alpha)$

substeps of the minimization gives the original root function (that we count $ICS(\alpha)$ at the very beginning). Therefore, $ICS(\alpha)$ of each minterm will remain unchanged through the minimization process. Thus, it is enough to calculate the $ICS(\alpha)$ of each minterm once at the beginning of the minimization. It means that the don't care terms that introduced during the minimization process don't affect the $ICS(\alpha)$ of each minterm.

$MC(I)$ is affected by don't care terms introduced during the minimization process. The importance of the implicant is inversely proportional to the number of minterms changed to don't care at each node function in the minimization. Each minterm changed to don't care and covered by I reduces the importance (increases the cost) of the I for next node function in minimization. The cost of I is determined by $MC(I)$.

The new don't care terms covered by selected implicant I are subtracted from $MC(I)$. The implicant that covers fewer minterms has higher cost (is less important) with

respect to an implicant that covers many minterms during the minimization process. So each don't care term introduced to the remaining function affects $MC(I)$ and increases its cost. Therefore, $MC(I)$ should be updated during the minimization process in CISH.

III. CONSTRAINED IMPLICANT SET HEURISTIC

A. INTRODUCTION

This heuristic discussed in this section is named after the constrained implicant set heuristic (CISH) because the original concept has been developed under the same name in multiple-valued logic minimization [Ref. 8]. The idea is to extend the constrained implicant set concept in multiple-valued logic to binary minimization.

The CISH has two computational phases: 1) constrain and select a minterm according to its implicant cover size and 2) constrain and select an implicant with respect to its minterm coverage. The selection of the minterm differs from other exact minimization algorithms due to the difference in its rules of decision.

The selection of the implicant which covers a selected minterm depends on its cost to the minimization. CISH chooses an implicant and investigates its effect on the future selection of minterms and implicants.

B. MINIMIZATION ALGORITHM OF CISH

The general steps of CISH are described (see Appendix G for the C program listing) in this Section. In the algorithm below, f denotes the function to be minimized.

All information about minterms, the implicants, and necessary sets are initialized once and updated in the recursive part of the heuristic.

1. Initialization

- Form the uncovered minterm set (UMS) from all minterms of f .
- Form the don't care set (DCS) from all don't care minterms of f (Initially, this is empty).
- Form the implicant set (IS) from all implicants of f .
- Tally the $MC(I)$ for each implicant I in IS .
- Find the $ICS(\alpha)$ of each minterm in UMS .

2. Recursive Algorithm

Apply the following steps recursively to a function f until the function consists of only don't care or 0 terms.

- Select an uncovered minterm α with the lowest $ICS(\alpha)$ from UMS . If more than one such minterm exists, the one with smaller binary representation is selected.
- Construct the constrained implicant set $CIS(\alpha)$ that includes all implicants that cover minterm α .
- If $ICS(\alpha) \neq 2$, select the implicant $I_i(\alpha)$ with the lowest $MC(I)$.
- If $ICS(\alpha) = 2$ and $MC(I) = 2$ for each of the two implicants in $CIS(\alpha)$, apply the extended search technique (EST) to select $I_i(\alpha)$.
- Put $I_i(\alpha)$ into the minimized sum-of-products set (MSP).
- Find the minterms in UMS that are covered by $I_i(\alpha)$.
- Remove these minterms from UMS , and place them into DCS .
- For each implicant in IS that covers at least one new don't care term, subtract the number of new don't care terms covered by $I_i(\alpha)$ from $MC(I)$. If the $MC(I) = 0$, remove $I_i(\alpha)$ from IS .

3. Extended Search Technique (EST)

Apply this search technique, if there is a minterm with $ICS(\alpha) = 2$ in the recursive part of the heuristic.

- Compute $ICS(\beta)$ of each direct neighbor β of α .
- Choose a β with $ICS(\beta) = ICS(\alpha) (= 2)$, if such a β exists. Select $I_i(\alpha)$ that covers β and α .
- Otherwise, select a neighbor β of α with the smaller $ICS(\beta)$. Select $I_i(\alpha)$ that covers β and α .

C. EXPLANATION OF CISH

In this section, the algorithm described in Section III.B is explained by using examples.

Initialization Part:

All information about minterms and implicants are found and prepared for the recursive phase. Mainly, three basic working sets (UMS , DCS , IS) and two basic measures ($ICS(\alpha)$, $MC(I)$) of CISH have been formed.

Firstly, the UMS is formed. This set includes the minterms that belong to A^{ON} of function f . DCS is assumed to be empty at the initialization part. DCS and UMS are complimentary sets. When UMS decreases, DCS increases. The IS initially includes all implicants of the function. IS gets smaller as a result of some implicants being removed during the minimization process.

Two basic measures of CISH are formed in initialization part. These are $MC(I)$ and $ICS(\alpha)$. All minterm's $ICS(\alpha)$ are computed. The CISH computes the $ICS(\alpha)$ the column-row order. For example, the minterm $\alpha = 1$ (i.e., binary representation 0001) is evaluated earlier than minterm $\alpha = 2$ (i.e., binary representation 0010).

$MC(I)$ is calculated by counting the number of the minterms covered by I . For example, if I covers 4 minterms, its $MC(I)$ is equal to 4. But, the $MC(I)$ of each I will change during the minimization process as mentioned in Section II.C.

Recursive Part:

Recursive part is the second computational phase of CISH. The purpose of the recursive part is to select the minterm with the lowest $ICS(\alpha)$, then constrain and select the implicant that has the lowest cost to the minimization. All of the minimization has been done in this part of heuristic. All the steps shown in Section III.B are explained by Example 6 and 7 in this section.

It is important to select the first minterm intelligently during the minimization process in CISH. The importance of the minterms is determined by $ICS(\alpha)$. As mentioned in Chapter II, $ICS(\alpha)$ is a measure of how many possible combinations a minterm

has with neighboring terms. A lower $ICS(\alpha)$ means that fewer combinations exist. The minterm with lower $ICS(\alpha)$ gains importance with respect to the minterms having higher $ICS(\alpha)$ in CISH, because the minterm with lower $ICS(\alpha)$ tends to be isolated and the minterm with a higher $ICS(\alpha)$ tends to be in the middle of the cluster of 1's.

If a function has ξ minterms, the search space of the function has ξ main branches (see Figure 11 in Section D). CISH selects the minterm with the lowest $ICS(\alpha)$ in *UMS*, then selects the main branch that includes the selected minterm. All of the minimization process in CISH is done in this main branch and its subbranches. CISH does not make a search to find the minimal expression of the function for any one of the remaining $\xi-1$ main branches. One of the properties of CISH is to find a near minimal solution by searching only one main branch of over-all search space. That's why the selection of the minterm is very important. The selection of minterm is done at every node function generated from root function by heuristic rules. The selection of minterm with the lowest $ICS(\alpha)$ reduces the search space significantly (unlike the exact minimization algorithms such as Quine-McCluskey).

Example 5:

To illustrate the effects of the selection of minterms in minimization, consider the following four variable function with eight minterms.

Let

$$f(w, x, y, z) = \sum m(0, 1, 2, 3, 4, 7, 9, 10).$$

Two possible minimal solutions for the given function are shown in Figure 7 on page 23.

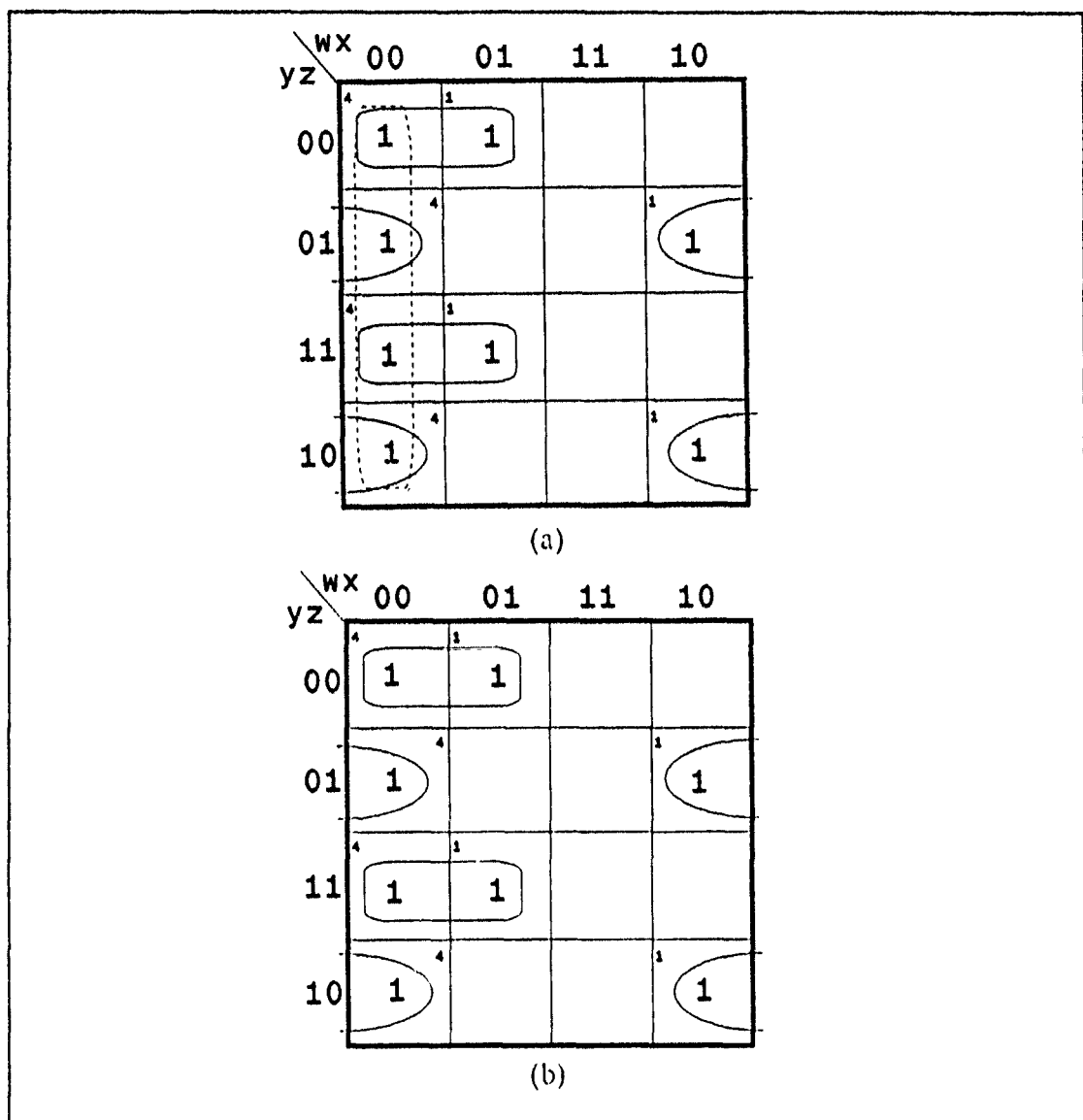


Figure 7. Importance of Selection of Minterm

If the minimization begins at any one of the minterms in the center of a cluster (i.e., $\alpha = 0, 1, 2, 3$), the prime implicant that corresponds to $\bar{w}\bar{x}$ will be included in the solution set. This prime implicant is indicated by a dashed line in Figure 7.a. Four additional implicants are necessary to cover the remaining minterms with $ICS(o) = 1$. The sum-of-

products expression of f includes five product terms. The implicant $\bar{w}\bar{x}$ is redundant in the minimization of f .

If the minimization begins with any one of the minterms such that $ICS(\alpha)=1$ (i.e., $\alpha=4, 7, 9, 10$), the minimal solution will have four prime implicants as seen in Figure 7.b. The sum-of-products expression of f for two solutions are written below:

$$f(w, x, y, z) = \bar{w}\bar{y}\bar{z} + \bar{x}\bar{y}z + \bar{w}yz + \bar{x}y\bar{z} + \bar{w}\bar{x}$$

$$f(w, x, y, z) = \bar{w}\bar{y}\bar{z} + \bar{x}\bar{y}z + \bar{w}yz + \bar{x}y\bar{z}$$

The CISH constructs the $CIS(\alpha)$ after selection of minterm from the implicants in IS . A group of implicants in IS is constrained by using $ICS(\alpha)$. All implicants in $CIS(\alpha)$ cover the minterm α .

The selection of implicants is equivalent to breaking the coupling between that implicant and its neighbors. The candidate implicant should have the lowest cost to minimization in CISH. [Ref. 9]

The coupling strength is introduced as the minimization cost. The $MC(I)$ is used as the cost of implicant to minimization. The implicant with the lowest cost has the highest $MC(I)$. The implicant with the highest $MC(I)$ covers the largest area in the function. The largest area can contain *don't care terms* as well as *1 terms*.

The chosen implicant in $CIS(\alpha)$ is to be a prime implicant due to the prime implicant theorem. The prime implicant theorem states that a minimal sum-of-product must always consist of a sum of prime implicants. [Ref. 10: p. 206]

At least one of the implicants in $CIS(\alpha)$ should be a prime implicant and it contains the fewest literals among other implicants that cover minterm α . Naturally, the prime implicants always have higher $MC(I)$ than non-prime implicants in $CIS(\alpha)$. The strategy of CISH is to always select an implicant that covers minterm α with the lowest cost.

This rule provides for that selected implicant being one of the prime implicants having the highest $MC(I)$ in $CIS(\alpha)$.

Extended Search Technique:

This technique is applied only for the special case when the $ICS(\alpha)$ of a minterm α equals two. It indicates that minterm α is covered by only two implicants and at least one of them should be chosen as a part of the solution. If the $MC(I)$ of these implicants is not equal to two, CISH selects the implicant with the highest $MC(I)$. On the other hand, if the $MC(I)$ of each implicant in $CIS(\alpha)$ equals two, it indicates that the minterm α is covered by two implicants, that none of them covers don't care terms, and their cost to the minimization process is equal. In this case, the best and the most efficient implicant should be chosen in $CIS(\alpha)$. The purpose of the *EST* is to minimize the negative impact for future minterm selection as well as implicant selection by choosing the most efficient implicant in $CIS(\alpha)$.

In *EST*, CISH finds the two direct neighbors β of minterm α . It checks the $ICS(\beta)$ of β . If a β exists such that $ICS(\beta) = ICS(\alpha) (= 2)$, it selects the implicant in $CIS(\alpha)$ that covers both β and α . If both direct neighbors have the same $ICS(\beta)$ such that $ICS(\beta) = ICS(\alpha)$, CISH selects the neighbor with smaller binary representation, then chooses the implicant that covers both β and α . If none of its direct neighbors have the same $ICS(\beta)$ as $ICS(\alpha)$, CISH then selects the β with smaller $ICS(\beta)$.

Example 6:

To illustrate the application of *EST* in CISH, consider the following a four variable function with eight minterms. Options for the minimization of f will either 1) not apply *EST* (Figure 8.a) or 2) apply *EST* (Figure 8.b) as shown on page 27.

Let

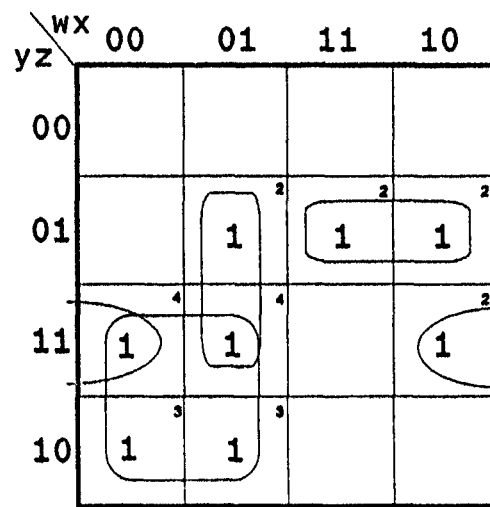
$$f(w, x, y, z) = \sum m(2, 3, 5, 6, 7, 9, 11, 13).$$

There are four minterms (i.e., $\alpha = 5, 9, 11, 13$) to be selected due to their $ICS(\alpha)$ (i.e., 2). The CISH selects the minterm $\alpha = 5$ (located at 0101) according to column-row order. The selected minterm has two implicants that cover it (01-1, -101), that is, $CIS(0101)$ has implicants $I_1(\alpha)$ and $I_2(\alpha)$ represented in binary as 01-1 and -101 respectively.

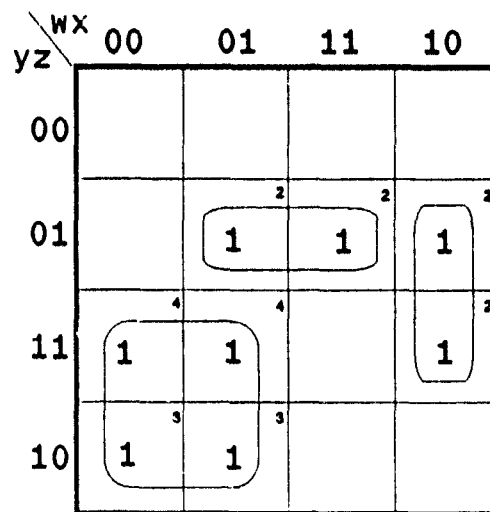
Figure 8.a shows the case when *EST* is not applied. Since two implicants ($I_1(\alpha)$, $I_2(\alpha)$) have the same cost for minimization (i.e., $MC(I) = 2$), the one of them must be selected. If $I_1(\alpha)$ is selected as the implicant to be placed in the solution set, then Figure 8.a shows the minimal sum-of-products expression of the function.

On the other hand, if *EST* is applied, after selection of the minterm $\alpha = 5$, its direct neighbors are found as $\beta_1 = 0111$ and $\beta_2 = 1111$. Since $ICS(0111) = 4$ and $ICS(1111) = 2$, *EST* selects 1111 which has lower value. In words, the implicant $I_2(\alpha)$ (that covers the α and β_2) is the best and the most efficient implicant to minimize negative impact for future selection of minterms and implicants at the next node function.

It can be seen that if *EST* is not applied, there may be a negative impact to minimization of function. The minimal solution obtained in this way has four product terms. *EST* provides a better minimal solution for the same function. The minimal solution has only three product terms as shown in Figure 8.b.



(a)



(b)

Figure 8. Application of Extended Search Technique

The process described above is illustrated in Figure 9 on page 28. The minimization (applying *EST*) is indicated by bold lines, located on the right branch from the root.

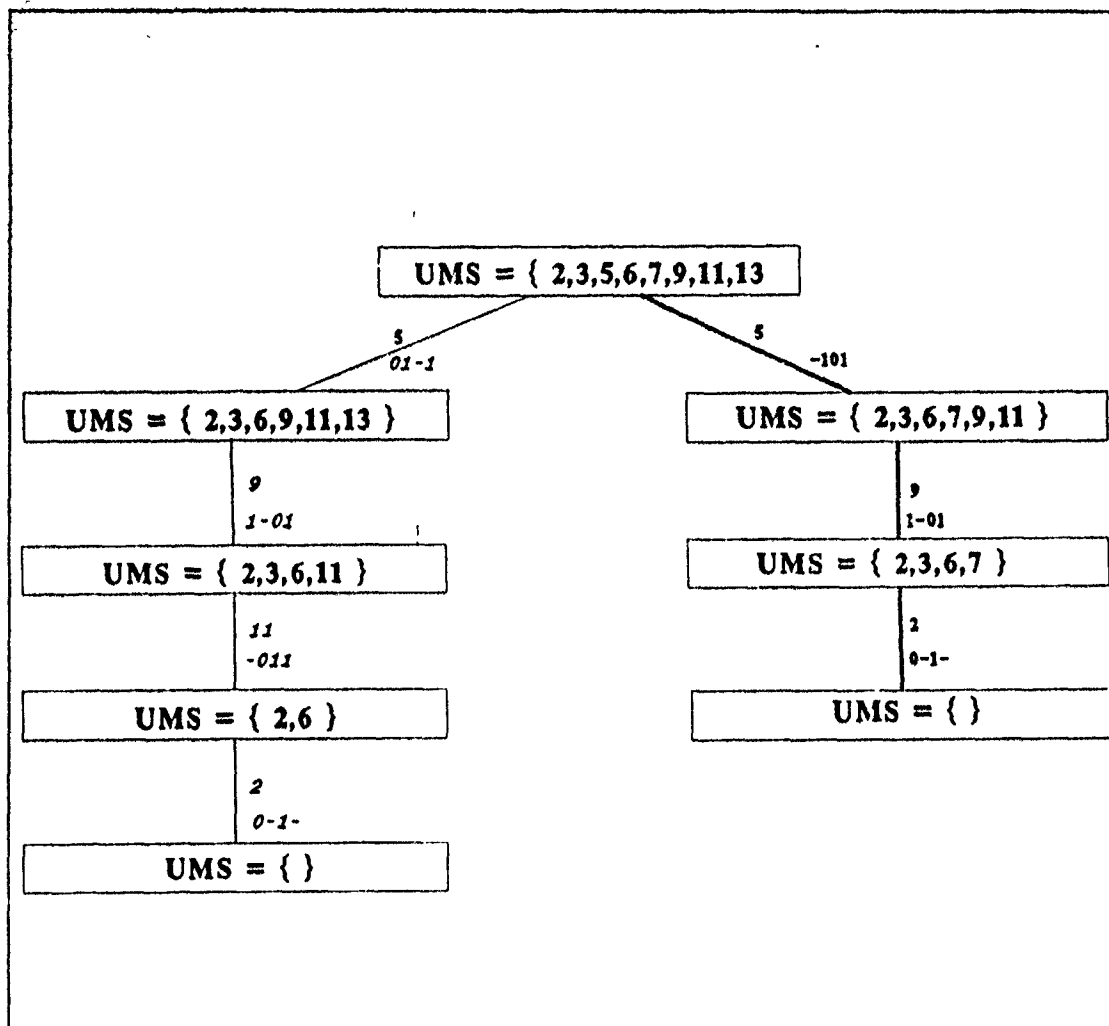


Figure 9. Constrained Implicants in Extended Search Technique

The selected minterms and implicants in binary representation is placed next to these lines. Another solution (without applying *EST*) is indicated by italic font located on left branch from the root.

The selected implicant $I_i(\alpha)$ is placed into *MSP*. *CISH* finds the minterms covered by $I_i(\alpha)$ in *UMS*. It turns these minterms to the don't care terms and puts them into *DCS*. *CISH* updates the cost of the implicant in *IS* that covers at least one of the new

don't care term. Updating of the cost of $I(\alpha)$ is done by subtracting the number of newly introduced don't care terms (covered by $I(\alpha)$) from $MC(I)$.

For any implicant I , if $MC(I)=0$, then it is removed from IS . The number of implicants in the IS gets smaller in each recursive step. The computation stops when UMS is empty.

D. A WALKTHROUGH EXAMPLE OF CISH

It is instructive to examine the application of the CISH. A four variable function with eight minterms is used as an example. The input function has been shown in Figure 10 on page 30.

Let

$$f(w, x, y, z) = \sum m(3, 4, 5, 7, 9, 13, 14, 15).$$

All information about minterms and implicants are presented in Table 1 and Table 2 on page 32. For simplicity, binary representation of each implicant is indicated by a capital letter in Table 1.

The sets of CISH constructed in the initialization part are listed below:

$$UMS = \{ 3, 4, 5, 7, 9, 13, 14, 15 \}$$

$$IS = \{ A(2), B(2), C(2), D(2), E(4), F(2), G(2), I(2), J(2) \}$$

$$DCS = \phi$$

$$CIS(\alpha) = \phi$$

There are eight main branches in the search space. These branches are shown in Figure 11 on page 34. CISH can begin minimization from any one of these four

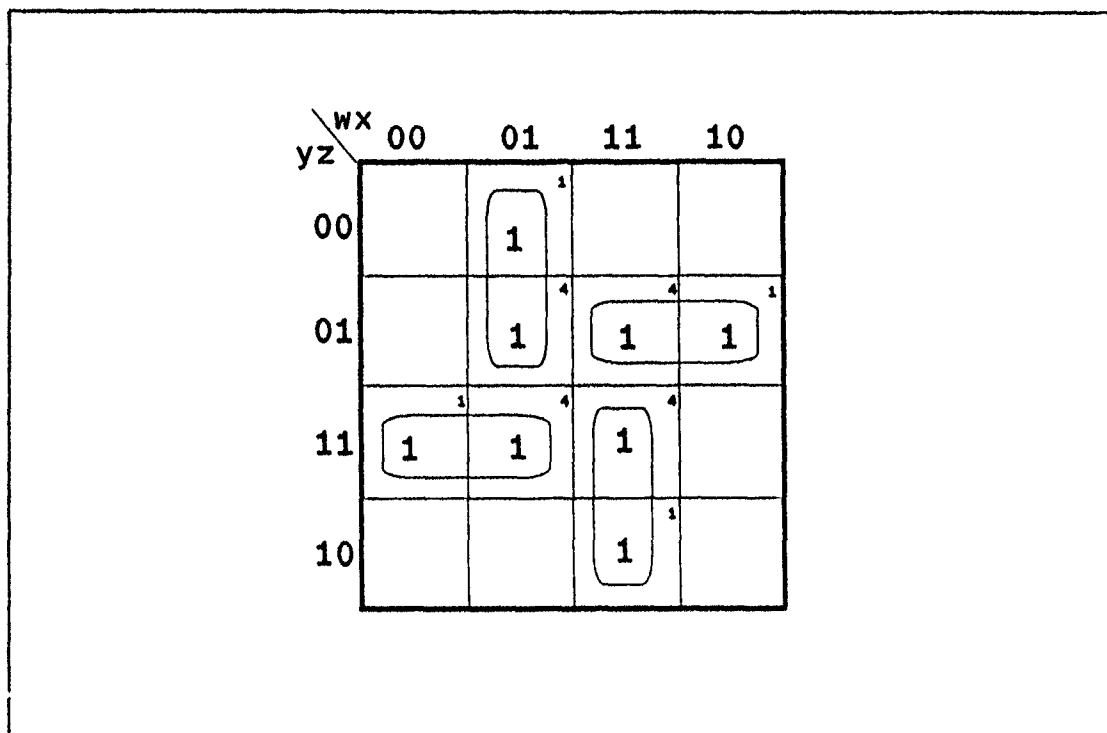


Figure 10. Instructive Example for CISH

branches ($\alpha = 3, 4, 9, 14$), because their $ICS(\alpha)$ are the same and the lowest in the UMS (i.e., $ICS(\alpha) = 1$). By using $ICS(\alpha)$, CISH reduces the search space from eight branches to four branches.

It is assumed that minterm $\alpha = 3$ is selected to begin minimization among four minterms with $ICS(\alpha) = 2$. $CIS(\alpha)$ is constructed with respect to $\alpha = 3$. This set consists of only implicant B in IS . Implicant B covers $\alpha = 3$ and $\alpha = 5$. The sets of CISH are updated with the rules of the heuristic. For example, the costs of implicant E, F, I change in IS . The cost of each implicant is indicated in parenthesis next to the implicant. The new sets of CISH and costs of the implicants are shown below and corresponding to the leftmost node at level 1 in Figure 11 on page 34.

$$UMS = \{ 4, 5, 9, 13, 14, 15 \}$$

$$IS = \{ A(2), C(2), D(2), E(3), F(1), G(2), I(1), J(2) \}$$

$$DCS = \{ 3, 7 \}$$

$$MPS = \{ B \}$$

Table 1. INFORMATION ABOUT IMPLICANTS

Implicant ($I(x)$)	Binary Representation	$MC(I)$	Minterm Covered by $I(x)$
A	010-	2	4, 5
B	0-11	2	3, 7
C	1-01	2	9, 13
D	111-	2	14, 15
E	-1-1	4	5, 7, 13, 15
F	01-1	2	5, 7
G	11-1	2	13, 15
I	-111	2	7, 15
J	-101	2	5, 13

Table 2. INFORMATION ABOUT MINTERMS

Minterm (x)	Binary Representation	$ICS(x)$	Covered by Implicant
3	0011	1	B
4	0100	1	A
5	0101	4	A, E, F, J
7	0111	4	B, E, F, I
9	1001	1	C
13	1101	4	C, E, G, J
14	1110	1	D
15	1111	4	D, E, G, I

UMS includes six minterms and CISH chooses three of them to start the minimization, since, their $ICS(\alpha)$ is smaller than others. These are $\alpha = 4, 9, 14$ and their $ICS(\alpha)$ are equal. It is assumed that $\alpha = 4$ is selected. $CIS(\alpha)$ contains only implicant *A*. *A* covers $\alpha = 4$ and 5. The new sets of CISH becomes:

$$UMS = \{ 9, 13, 14, 15 \}$$

$$IS = \{ C(2), D(2), E(2), G(2), I(1), J(1) \}$$

$$DCS = \{ 3, 4, 5, 7 \}$$

$$MPS = \{ A, B \}$$

Implicant *F* was removed from *IS* because it would cause the highest cost for all future minimization processes (i.e., $MC(I) = 0$).

There are only two minterms to begin the next selection and constraining implicant. These are $\alpha = 9, 14$ and they are equal in $ICS(\alpha)$. So we can arbitrarily select the one with smaller binary representation, i.e., minterm $\alpha = 9$ is selected. The $CIS(\alpha)$ includes only implicant *C*.

After selection of *C*, the sets are updated as shown below;

$$UMS = \{ 14, 15 \}$$

$$IS = \{ D(2), E(1), G(1), I(1) \}$$

$$DCS = \{ 3, 4, 5, 7, 9, 13 \}$$

$$MPS = \{ A, B, C \}$$

Now, only one minterm is in UMS ($\alpha = 14$). $CIS(\alpha)$ consists of only implicant D . D covers $\alpha = 14$ and $\alpha = 15$. The sets of the CISII become :

$$UMS = \phi$$

$$IS = \phi$$

$$DCS = \{ 3, 4, 5, 7, 9, 13, 14, 15 \}$$

$$MPS = \{ A, B, C, D \}$$

The minimal sum-of-products expression of f is the UMS or:

$$\begin{aligned} f(w, x, y, z) &= 010 - + 0 - 11 + 1 - 01 + 111 - \\ &= \bar{w}x\bar{y} + \bar{w}yz + w\bar{y}z + wxy \end{aligned}$$

All branches and constrained implicants are shown in Figure 11 on page 34. The constrained paths are shown with bold lines, located on the left branch from root. Another search path is shown by selection $\alpha = 5$ at the very beginning of the minimization. This path, located on the right branch from the root, gives another possible solution, without applying CISII.

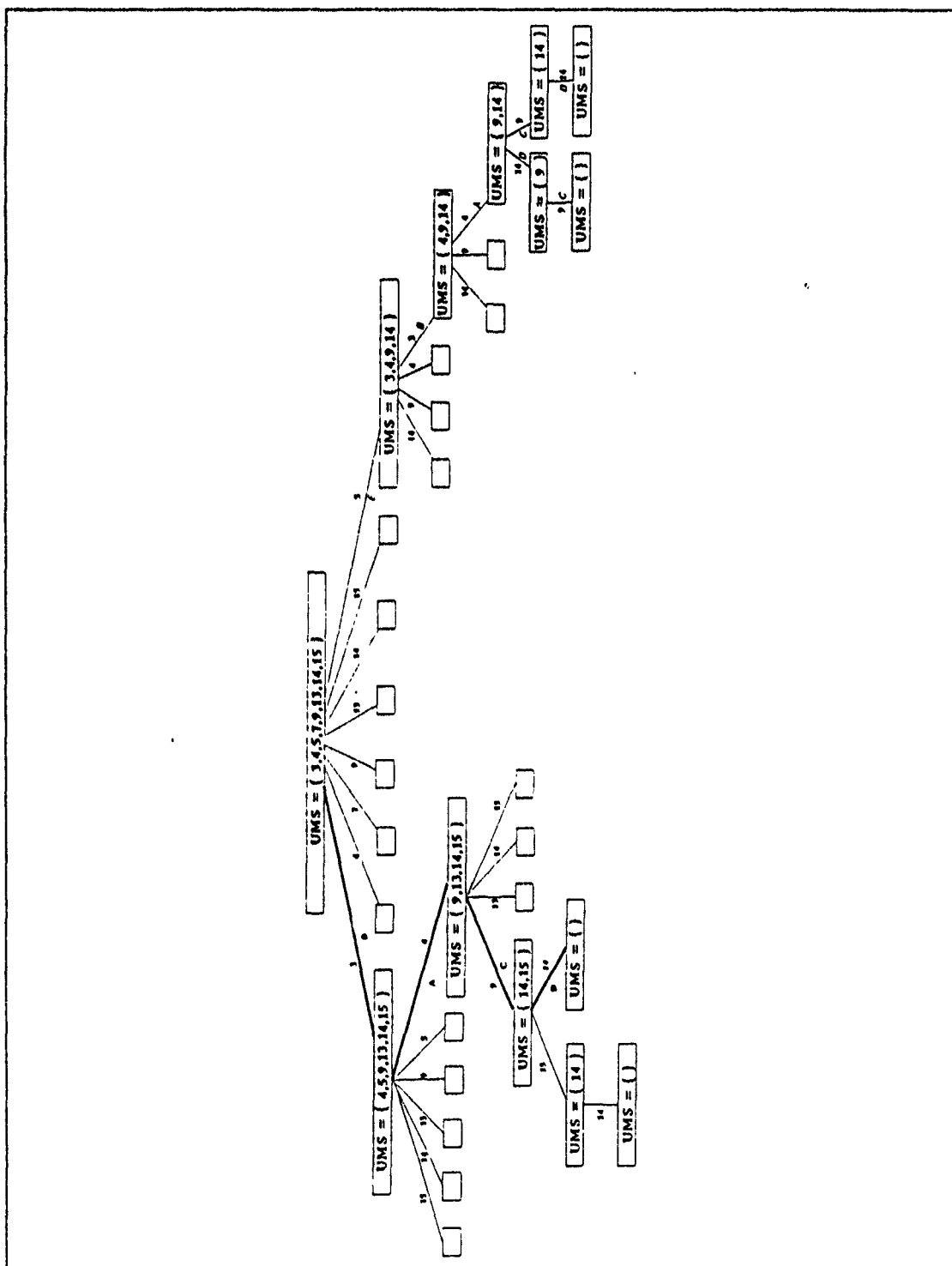


Figure 11. Search Space and Constrained Implicant Paths

IV. COMPARATIVE RESULTS

In this thesis, all test results were obtained by running sample functions on the VAX 11/785 and ISI workstations. Different number of sample functions (4100) were randomly generated. The input functions are generated for different variables with different number of minterms, (i.e., 9 variable function with 475 minterms or 7 variable function with 120 minterms). Each algorithm was applied to these sample functions, then the average number of product terms, average computation time, and average memory usage are recorded. The computation time for 9 variable functions and larger is very large. This explains why we did not simulate more than 9 variables. This thesis investigated three algorithms: 1) CISH, 2) MIII (see Appendix B), 3) Espresso II (see Appendix C).

A. PERFORMANCE COMPARISON

The performance measures are recorded and compared. These are 1) the average number of product terms, 2) the average computation time, and 3) the average memory usage.

The average number of product terms will show us the advantage of the each algorithm. For each set of sample input functions, the average number of product terms (see Appendix D) is computed. From these data points, a curve is plotted to indicate the average number of product terms as a function of the number of the variables. The plot is shown in Figure 12 on page 36.

In this exponential-growing shaped figures, it is observed that:

- The differences in the number of average products terms among these algorithms are not significant. Less than 1% of testing functions are different.
- When the number of variables gets larger, the curve grows up exponentially. This is chiefly due to the fact that both the number of minterms and implicants increase exponentially and they make the computation time longer.

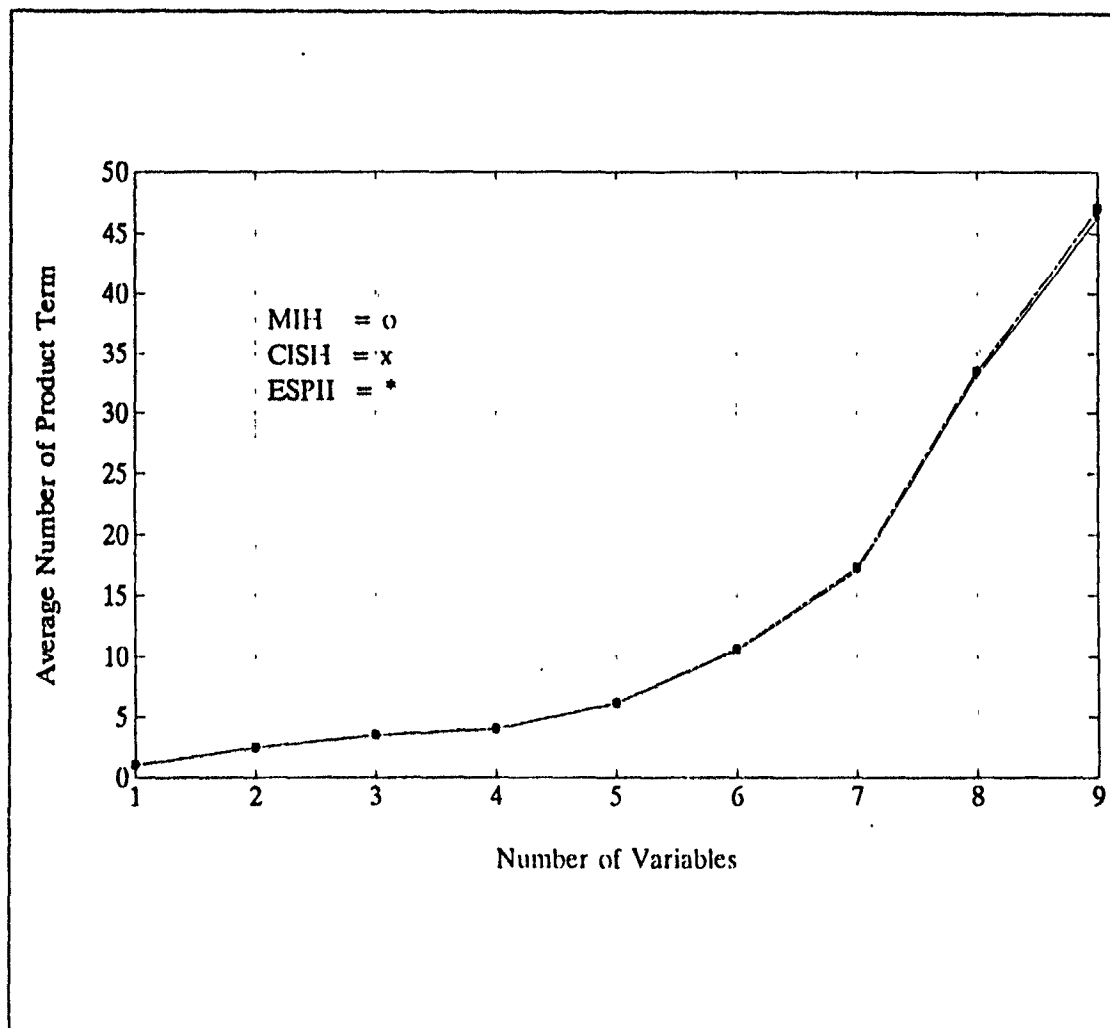


Figure 12. Average Number of Product Terms

From Figure 12, the CISI performs as well as the other two algorithms (MIH, Espresso II).

B. TIMING COMPARISON

Both VAX 11/785 and ISI workstations at NPS can measure a program's computation time in UNIX environment. In this section, timing comparison counts the average computation time for each group of input functions.

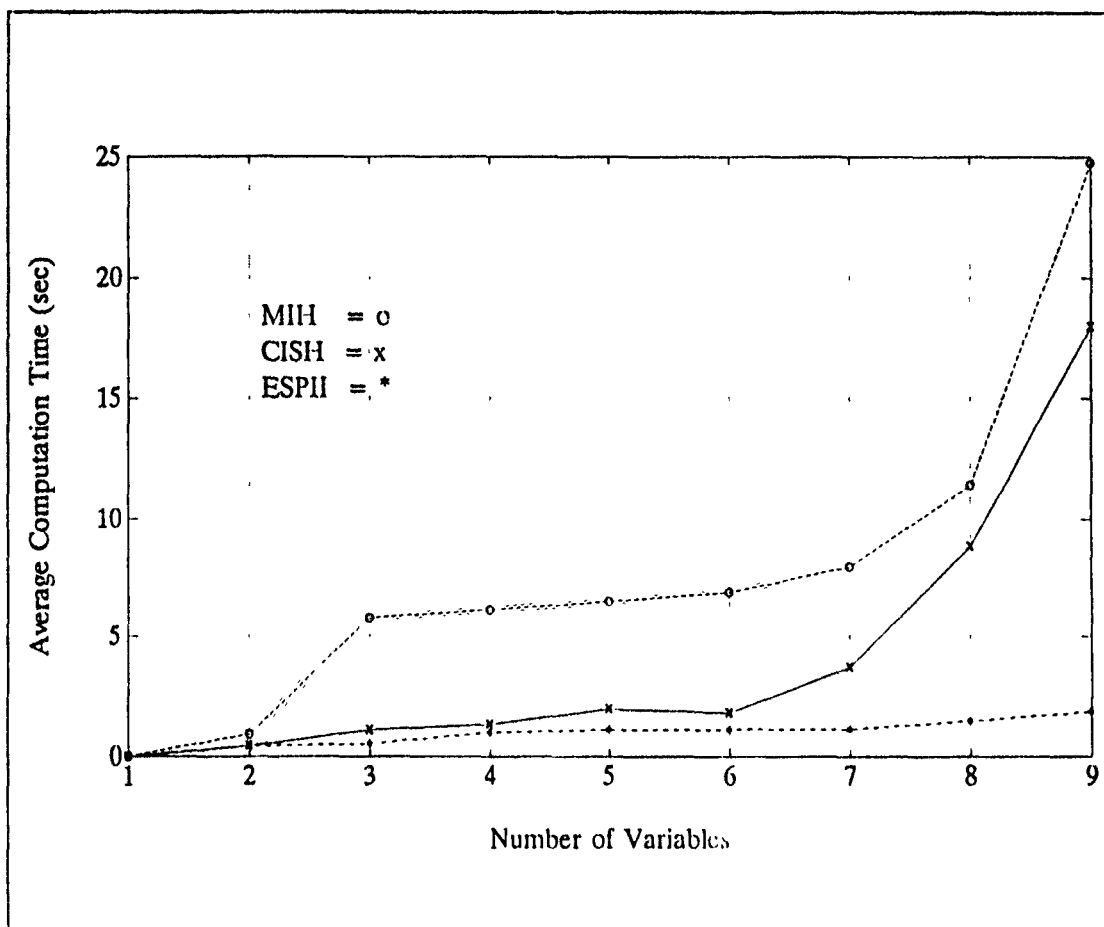


Figure 13. Average Computation Time

The decision rules for selecting the minterm α and the implicant $I(\alpha)$ in each algorithm are different. Generally, an algorithm that has complex rules to select α and $I(\alpha)$ takes longer computation time. The CISH uses more complex decision rules than MIH and Espresso II. It may appear that CISH needs more computation time than other algorithms. However, the computation time of the CISH is shorter than MIH, although it is not as fast as Espresso II. The graphical result is shown in Figure 13. Espresso II outperforms the other two heuristics. This might be due to smaller constant in computation complexity. The numerical results are shown in Appendix E.

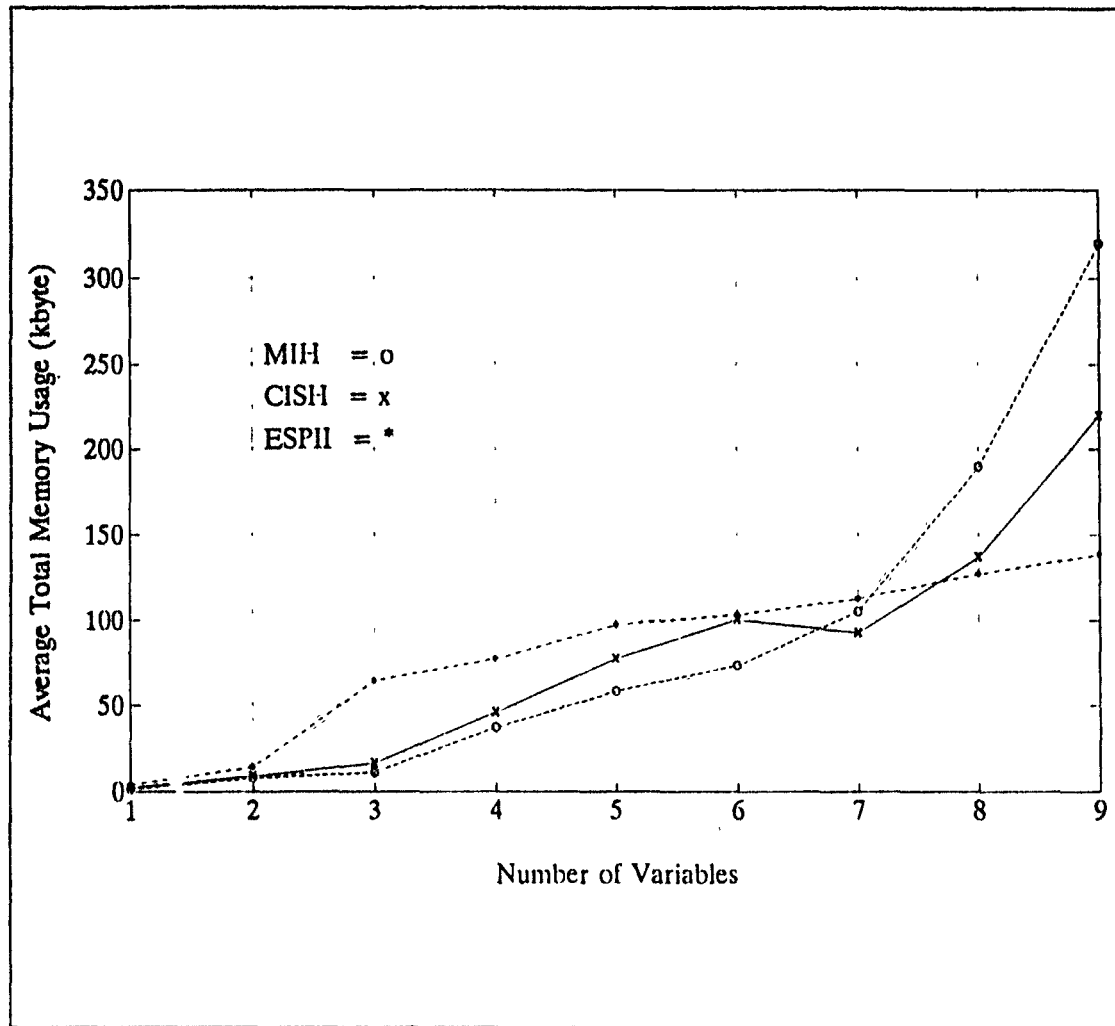


Figure 14. Average Total Memory Usage

C. MEMORY COMPARISON

The average memory usage is measured for each algorithm (see Appendix F). From these data points, a curve is plotted to indicate the average memory usage as a function of the number of the variables. The plot is shown in Figure 14 on page 38.

It is expected that a heuristic or algorithm based on tabular method needs more memory to store the information for keeping track of each term during the minimization process. The MIH is based on a tabular method, thus it needs a lot of memory space.

The CISH selects and constrains the minterm α and implicant $I(\alpha)$. As a natural consequence, CISH uses less memory than MIH due to heuristic strategy.

A sophisticated dynamic memory scheme was used in Espresso II, so that it demands less memory than the other two heuristics (MIH, CISH).

To summarize:

- There has been significant difference among the memory usage of each algorithm.
- Direct covering and tabular method used in CISH and MIH needs more room to store the information about terms than the decomposition technique used in Espresso II.
- If we use dynamic memory allocation in CISH, the memory requirement will be reduced.

V. DISCUSSIONS AND CONCLUSIONS

A. DISCUSSIONS

There is no significant difference among these algorithms with respect to number of product terms. The small difference in the average number of product terms originated from the application of *EST* to CISH. Recall that in *EST*, we compute $ICS(\alpha)$ of a given minterm, then constrain and select the implicant $I_i(\alpha)$ with respect to the relationship among direct neighbors of minterm α (i.e., one step look-ahead).

Naturally, a further look-ahead while selecting the $I_i(\alpha)$ may provide a better selection of implicants. The exponential growth of the number of all possible implicants restricts the practical use of k look-aheads for $k \neq 1$. The application of *EST* provides better solutions for CISH over MIH. On the other hand, the best solutions are provided by Espresso II.

From the computation time results, the CISH runs faster than MIH in all testing conditions. This time efficiency is a result of the decision rules employed in CISH (that takes advantages of the properties of the constrained implicant set concept). On the other hand, Espresso II runs faster than the other two heuristics as a results of the behavior of the algorithm based upon theunate paradigm (see Appendix C). Typical PLA's produce shallow recursion trees terminating quickly at unate leaves in search space. This benefit is used in Espresso II.

The memory comparison shows that MIH and CISH need more room to store the information about the minterms and the implicants to operate efficiently. On the other hand, Espresso II uses less memory by dealing primarily with matrix representation of the function and the minimization. It is believed that use of more memory makes

heuristics slower, since memory intensive programs tend to run slower than cpu intensive programs.

B. CONCLUSIONS

One of the primary goals of this research is to examine whether the constrained implicant set concept in multiple-valued logic can be efficiently used in binary minimization. It is seen that the binary minimization of a given function can be done efficiently by using the constrained implicant set concept.

In the development of the CISH and from comparative results, we have the following observations:

- The constrained implicant set concept reduces the search space significantly in the binary minimization.
- CISH does not lose run time efficiency unlike MIH because the heuristic finds the solution and stops earlier than MIH.
- Direct covering fits well with the constrained implicants set concept. On the other hand, it does not provide efficiency on the computation time as much as the decomposition technique does, such as Espresso II.
- In the cyclic case, applying *EST* provides better results than MIH. On the other hand, using only one step look-ahead in *EST* may lose the optimality. However, it is not practical to have more than one look-ahead, that requires longer computation time and larger memory space.
- By constraining implicants with the cost factor, it is possible to find near and good minimal sum-of-products expressions.
- It is possible to get a near and good minimal solution by only searching the main branches of the search space.
- The memory usage can be decreased by using the dynamic memory allocation like Espresso II. Originally CISH uses the static memory allocation.

Generally, Espresso II is more efficient than CISH. This is not surprising since our heuristic is an initial unoptimized prototype. Espresso II on the other hand is the product of considerable effort by a very large team over a long period which has concentrated on producing a production quality software package.

Directions for Future Research

It is our hope that the ideas and heuristic described in this thesis represent a plateau for the development of two-level binary minimization. The minimization problem is in principle difficult, and future developments will exploit fundamentally new ideas. Here, we briefly describe directions in which future research might be pursued.

- In CISH, function is represented in the form of the truth table. All the minimization process is done by using the data structure. On the other hand, Espresso II uses the matrix representation of function and the minimization. This provides increased speed of execution. This matrix representation can be applied CISH.
- Direct covering technique is applied to CISH. It is obtained that decomposition technique can provide faster and more accurate solutions in minimization like Espresso II. The constrained implicants set concept can be improved by using the decomposition technique instead of direct covering.
- The speed of execution can be increased by applying the concept of unate paradigm to CISH. Actually, CISH reduces the search space significantly. On the other hand, we believe that using the concept of unate paradigm reduces the search space (that already reduced by applying the constrained implicant set concept).
- The function applied to CISH includes only the 1 and 0 terms. The CISH can be improved by using the don't care terms at the beginning of the minimization. We hope that using the don't care terms at the beginning of the minimization with 1 terms helps reduce the computation time and use less memory space.
- CISH is developed using only the single output case. The heuristic can be modified to be used in the minimization of the multi-output case. We believe that CISH provides more efficient results than MIH in the multi-output cases.

APPENDIX A. ABSOLUTE MINIMIZATION ALGORITHM

This algorithm is taken from [Ref. 1].

```
algorithm absolute_minimization;  
f ← input_function;  
cur_best_soln_set ← best solution from the Pomper and Armstrong,  
                    Besslich and Dueck and Miller heuristics;  
cur_best_soln_size ← number of implicants in best solution;  
cur_partial_soln_set ←  $\phi$ ;  
cur_partial_soln_size ← 0 ;  
minimize(f);  
stop
```

```
procedure minimize(f);  
 $\phi$  ← some essential implicant set of f ;  
while (( there exists another implicant in  $\phi$ ) and ( cur_partial_soln_size + 1 <  
                                                    cur_best_soln_size ) ) do  
begin  
    I ← the next implicant in  $\phi$   
    cur_partial_soln_set ← cur_partial_soln_set  $\cup$  {I}
```

```

cur_partial_soln_size ← cur_partial_soln_size + 1;
if( for all assignments x of values to X,  $f(x) = 0$  or  $f(x) = r$  )
    then
        begin
            cur_best_soln_set ← cur_partial_soln_set;
            cur_best_soln_size ← cur_partial_soln_size
        end
    end
    [[backtrack ]]
else if ( cur_partial_soln_size + 1 < cur_best_soln_size ) then
    minimize(  $f - I$  );
    cur_partial_soln_size ← cur_partial_soln_size - 1;
    cur_partial_soln_set ← cur_partial_soln_set - {I}
end
return

```

/* The subtraction of an implicant I from a function f , as described by $f \leftarrow f - I$, takes into account the value of the input_function. Let x be assignment of values to variables X . Then, $f \leftarrow f - I$ means

```

for(all assignments x of values to X) do
    begin
        if(  $f(x) = r$  ) or ( input_function(x) =  $r$  ) or (  $f(x) \leq I(x)$  and input_function(x)
            =  $r - I$  ))
            then  $f(x) \leftarrow r$ 
        else  $f(x) \leftarrow f(x) - I(x)$ 
    end
end

```

end

*/
/

APPENDIX B. MAXIMUM IMPLICANT HEURISTIC

The Maximum Implicant Heuristic (MIH) is developed by modifying the Quine-McCluskey Algorithm (QM). MIH is based on obtaining the switching function as a near minimal sum-of-products. MIH differs from QM with respect to the two basic approaches. These are stated below:

- Incorporation of A^{ON} , A^{OFF} , A^{DC} sets of function at the beginning of the minimization
- Approaching to solve the cyclic case for a given function

The differences between two algorithms are explained by showing the main steps of QM. QM is an exact minimization algorithm. It consists of two main parts shown below [Ref. 2; p. 58]:

- Generation of the prime implicants
- Extraction of a minimum prime cover

Modifications are done in these main parts of QM.

Modification 1:

The first modification to develop MIH is done at the first part of QM. The MIH does not include the don't care terms while generating all prime implicants. It uses only the minterms that belong to A^{ON} and A^{OFF} of given function at the beginning. On the other hand, QM can consider A^{ON} , A^{OFF} , A^{DC} of function. Use of the don't care terms in the minterm list makes generation of all the prime implicants more complex. The more computation time and memory space is necessary to find prime implicants. Thus, a modification has been done to save the computation time and memory space. This modification is reflected to development of MIH as not including A^{DC} of a given function

at the beginning of minimization and dealing with only the function with A^{ON} and A^{OFF} in its minterm list.

Modification 2:

QM algorithm sets up a covering table that shows all of the prime implicants and minterms covered by prime implicants. The essential prime implicants are removed from this covering table, then a reduced covering table is formed. Secondary essential prime implicants are found and removed from last covering table. A new covering table is set up by remaining minterms and prime implicants [Refs. 2: pp. 59-64, 11: pp. 146-156].

In fact, it occurs in last reduced covering table, there is more than one possible cover for given function. It is said that function has cyclic case. This cyclic case is solved by applying the Petrick Algorithm in QM. Petrick Algorithm can be stated as producing all possible covers for the function in covering table and selection of the one of the covers requiring the smallest number of prime implicants and literals [Ref. 2: p. 64].

Producing all possible covers for the function needs more computation time and memory requirement. All possible covers in last reduced covering table must be found in QM. The reason is: QM guarantees exact minimal solution by searching all possible covers and selecting one having the fewest number of product terms.

The second modification is made at this part of QM. The reason is to avoid spending a lot of computation time to find all possible covers and using more memory place.

The modification can be stated as: 1) find the maximum implicant that covers the largest area in the last reduced covering table and 2) declare it being in the solution set. The minimization process is continued after removing the maximum prime implicant from this covering table, then establishing a new reducing table. If function still has a cyclic case, the same process is applied until no cyclic case exists.

The example about applying Petrick Algorithm in QM and breaking the cyclic case in MIH can be found in [Ref. 2: pp. 64-65].

Minimization Algorithm of MIH:

In summary, the MIH (for finding a near minimal sum-of-products expression to a given function) follows the step given below.

- Find the set of all prime implicants of the function by using minterms that belong to A^{ON} and A^{OFF} of the given function.
- Construct a covering table from all generated prime implicants.
- Identify all of the essential prime implicants and form a reduced covering table.
- Identify secondary essential prime implicants and reduce the covering table again.
- If there is a cyclic case in reduced covering table, break the cyclic case by selecting the maximum prime implicant to find a minimal cover for remaining minterms. Apply this process until no cyclic case occurs in reduced covering table.

Observations on MIH:

The most important observations about MIH are stated below:

- Most of the steps of both algorithms (MIH, QM) are identical. If a function does not have any cyclic case during its minimization process, MIH and QM can be considered as the same algorithm.
- The MIH provides a near minimal sum-of-products expression for given function as a results of not finding all possible covers to solve cyclic case in reduced covering table.

APPENDIX C. ESPRESSO II

Espresso II is a set of algorithms for logic minimization which basically follows the sequence of top-level transformation of iterated expansion-reduction pioneered by MINI [Ref. 5: p.13].

The fundamental definitions used in explanation of Espresso II are summarized below:

Definition :

A logic function f is monotone increasing (monotone decreasing) in a variable x_j if changing x_j from 0 to 1 causes all the outputs of f that change, to increase also from 0 to 1 (from 1 to 0). A function that is either monotone increasing or monotone decreasing in x_j is said to be *monotone* or *unate* in x_j .

Definition :

A function is said to be *unate function*, if it is unate in all its variables. Example:
$$f = x_1 \bar{x}_2 + \bar{x}_2 x_3$$

Theorem :

The *Shannon expansion theorem* states that a function can be expanded about any chosen variable that is, to produce an equivalent expression for the function in which the chosen variable appears once in uncomplemented form and once in complemented form.

The statement of the theorem is:

$$f(x_1, x_2, \dots, x_i, \dots, x_n) = x_i f(x_1, x_2, \dots, 1, \dots, x_n) + \bar{x}_i f(x_1, x_2, \dots, 0, \dots, x_n)$$

Algorithms of Espresso II based on a single basic strategy: a recursive divide and conquer. It basically uses decomposition technique. Decomposition is based upon Shannon Expansion. The Shannon Expansion uses the cofactors of a logic function. Since Espresso II uses the benefit of unate functions in Shannon expansion.

The algorithms of Espresso II form a logic minimization tools which achieves both robust performance and quality results. Iterative improvement produces well-minimized cover with high confidence, while unite paradigm together with special-case handling ensures reasonably efficient execution for a broad range of incoming problem [Ref. 5: pp. 161-162].

Speed of execution in Espresso II is based on the unate paradigm. Typical PLA's produce shallow recursion trees terminating quickly at unate leaves. Espresso II uses this benefit to trim the recursion tree and balance it judiciously. Besides, using matrix representation for representing logic function and minimization process in Espresso II requires less memory, and the operations to be executed faster than other forms of representation of function [Ref. 5: pp. 44-46].

The objectives of Espresso II are to minimize:

- The number of the product terms in the cover (NPT)
- The number of literals (not don't care) in the input parts of the cover (NLI)
- The number of literals in the output part (NLO)

The Espresso II minimization procedure defines a vector objective function

$$\Phi = (NPT, NLI, NLO)$$

and continues to iterate through into main minimization loop until none of the three components of Φ have been reduced since the last past through the loop [Ref. 5: pp. 54-55]

Espresso II minimization involves seven basic routines and the sequence of operations carried out by Espresso II is outlined below:

- **Complement** : Computes the complement of PLA's and the don't care set.
- **Expand** : Expand each implicant into a prime and remove covered implicant.
- **Essential-Primes** : Extract the essential primes and put them in the don't care set.
- **Irredundant Cover** : Find a minimal (optionally minimum) irredundant cover.
- **Reduce** : Reduce each implicant to a minimum essential implicant.
- **Iterate** : Expand, irredundant cover, reduce until no improvement.
- **Lastgrap** : Try reduce, expand and irredundant cover one last time using a different strategy. If succesful, continue the iteration.
- **Makesparse** : Include the essential primes back into the cover and make the PLA structure as sparse as possible.

The widely description and explanation of over-all Espresso II program can be obtained at Ref. 5.

APPENDIX D. AVERAGE NUMBER OF PRODUCT TERMS

Table 3. AVERAGE NUMBER OF PRODUCT TERMS

Number of Variables	Espresso II	MIH	CISH
1	1.000	1.000	1.000
2	2.450	2.450	2.450
3	3.480	3.480	3.480
4	4.000	4.010	4.000
5	6.100	6.118	6.112
6	10.500	10.660	10.567
7	17.000	17.268	17.256
8	33.160	33.473	33.475
9	46.320	46.992	46.990

APPENDIX E. AVERAGE COMPUTATION TIME

Table 4. AVERAGE COMPUTATION TIME (SEC)

Number of Variables	Espresso II	MIH	CISH
1	0.000	0.000	0.000
2	0.400	0.890	0.420
3	0.490	5.760	1.070
4	0.960	6.010	1.300
5	1.060	6.480	1.952
6	1.070	6.860	1.780
7	1.100	7.976	3.690
8	1.446	11.410	8.830
9	1.824	24.704	19.992

APPENDIX F. AVERAGE MEMORY USAGE

Table 5. AVERAGE TOTAL MEMORY USAGE (KBYTE)

Number of Variables	Espresso II	MIH	CISH
1	3.600	1.180	2.160
2	13.90	7.660	8.980
3	64.160	10.950	16.150
4	77.000	36.720	45.786
5	96.960	57.980	77.400
6	103.200	73.170	100.22
7	112.552	115.090	92.670
8	127.080	189.670	137.060
9	138.576	319.696	219.480

APPENDIX G. PROGRAM LISTING

```

#include <stdio.h>

#define maxint 32767
#define maxvars 10
#define maxcint 65535
#define maxcubes 1000
#define TRUE 1
#define FALSE 0
#define Function
#define max 5
typedef struct cube {
    int tt[maxvars];
    int ff[maxvars];
    int ics;
    int oldics;
    int ic;
    int selected;
    int track;
} Cube;

int numvar;
int m,n;
int aux,aux1;
int naux,naux1;
int num_minterm;
int num_product;
int countimpl;
int numcubes[maxvars];
int covered[maxvars][maxcubes];
int j, k,kl,p,rc,r,rm,rl,rk,rz;
int found;
int isonum;
int temp;
int small,high;
int num_implicant;
int sub,sub1;
int dum,dum1;

Cube cubes[maxvars+1][maxcubes+1];
Cube temporarycube;
Cube tempcube;
Cube implcubes[maxvars+1][maxcubes+1];
Cube tempi[1][max];
Cube nebor[1][max];

FILE *fpil,*fpol;

Function main (argc, argv)

```

/*Max # of variables in a product term.*/
 /*2**(maxVars)-1*/
 /*Max # of cubes to allocate per level*/
 /*Bits 1 for uncomplemented variables.*/
 /*Bits 1 for complemented variables.*/
 /* degree of clustering of minterm */
 /* degree of clustering of minterm */
 /* minterm coverage of each implicant */
 /* flag of selected implicants in SOP */
 /* to keep track of implicants,cubes */
 /* number of the actual variables */
 /* counters for loops */
 /* variables for temporary implicants */
 /* variables for temporary cubes */
 /* number of the minterms */
 /* number of the sum_of_product terms */
 /* counter for implicants cover cube */
 /* flag to indicate levels and cubes */
 /* flag to indicate covered minterms */
 /* Index into the cubes,covered array */
 /* flag to keep track of covered imp */
 /* counter for remainder minterms */
 /* dummy variable for implicant cube */
 /* variable for selection sorting */
 /* number of all implicants in funct. */
 /* temporary variables */
 /* temporary variables */
 /* Cube representation for minterms */
 /* temporary cube for manipulation */
 /* temporary cube for manipulation */
 /* cube representation for impl. */
 /* temporary implicant */
 /* temporary neighbor cubes */
 /* pointers for the files */

```

int argc;
char **argv;
{
    int i,k;
    found = FALSE;
    for ( m=0; m <= maxvars ; m++ )
    {
        numcubes[m] = 0; /* initialization for all level */
    }
    for ( m=0; m < numvar ; m++ )
    {
        for ( i=0; i < maxcubes ; i++ )
        {
            covered[m][i] = FALSE ;
            cubes[m][i].ics=0;
            cubes[m][i].oldics=0;
            cubes[m][i].ic=0;
            cubes[m][i].selected =0;
            cubes[m][i].track =0;
            for ( k=0; k < numvar ; k++ )
            {
                cubes[m][i].tt[k]=0;
                cubes[m][i].ff[k]=0;
            }
        }
    }

    /* Read the minterm from input file */

    readfile(argv[1]);

    for ( m = 0; m < numvar; m++ ) /* For all level except the last */
    {
        for(j =0; j < numcubes[m]; j++) /* For all cubes at this level */
        {
            for (k =j+1; k < numcubes[m] ; k++) /* other cubes at this level*/
            {
                if( rc=combinable(&cubes[m][j],&cubes[m][k]))
                {
                    covered[m][j] = TRUE; /* mark the cubes as covered */
                    covered[m][k] = TRUE; /* mark the cubes as covered */

                    /* Combine into an (m+1)-cube store in tempcube */

                    combine(&cubes[m][j],&cubes[m][k],&tempcube);
                    found = FALSE ; /* See if it is generated before */
                    for (p=0; p < numcubes[m+1]; p++)
                    {
                        if ( r = equalcubes(&cubes[m+1][p],&tempcube))
                        {
                            found = TRUE;
                        }
                    }
                }
            }
        }
        if(! found)
        {

```

```

        /* add this as new implicant of higher level */
        for(k1=0; k1 < numvar; k1++)
        {
            cubes[m+1][numcubes[m+1]].tt[k1] = tempcube.tt[k1];
            cubes[m+1][numcubes[m+1]].ff[k1] = tempcube.ff[k1];
        }
        numcubes[m+1] = numcubes[m+1] + 1;
    }
}
}
}
}
}

```

```

        /* Find all possible implicant of function */
        /* ----- */

for (m=0 ;m < numvar ; m++)
{
    for ( j= 0; j < numcubes[m]; j++ )
    {
        if((m == 0) && (!(covered[0][j])))
        {
            num_implicant++;
            temp=num_implicant-1;
            change(&cubes[m][j],&implcubes[0][temp]);
        }
        if((m != 0) && (covered[m][j]))
        {
            num_implicant++;
            temp=num_implicant-1;
            change(&cubes[m][j],&implcubes[0][temp]);
        }
        if((m !=0) && (!covered[m][j]))
        {
            num_implicant++;
            temp=num_implicant-1;
            change(&cubes[m][j],&implcubes[0][temp]);
        }
    }
}
}

```

```

        /* find the Implicant Cover Size (ICS) */
        /* ----- */

for( m=0; m < num_minterm; m++)
{
    for( n=0; n < num_implicant; n++)
    {
        if(rm = imp_cov_size(&cubes[0][m],&implcubes[0][n],&temporarycube))
        {
            cubes[0][m].ics++;
            cubes[0][m].oldics++;
        }
    }
}

```

```

    }
}

/* find Minterm Coverage MC(I) */
/* ----- */

for(m=0; m < num_implicant ;m++)
{
    implcubes[0][m].ic = coverage_size(&implcubes[0][m]);
}

for(i=0; i< num_minterm;i++)
{
    small=i;
    for(j=i+1; j<num_minterm; j++)
    {
        if(cubes[0][j].ics<cubes[0][small].ics){
            small=j;
        }
        swap(&cubes[0][small],&cubes[0][i],&temporarycube);
    }

    /* Application of CIS to find minimal sum expression */
    /* ----- */

    fpol=fopen("cis.o","w");

    REC: for(i=0;i<num_minterm;i++)
    {
        if(cubes[0][0].ics == maxcint && cubes[0][i].ics != maxcint)
        {
            swap(&cubes[0][i],&cubes[0][0],&temporarycube);
            break;
        }
    }
    for(i=0; i < num_implicant; i++)
    {
        high=i;
        for(j=i+1; j<num_implicant;j++)
        {
            if(implcubes[0][j].ic > implcubes[0][high].ic){
                swap(&implcubes[0][j],&implcubes[0][high],&temporarycube);
            }
        }
    }

    /* Extended Search Technique */
    /* ----- */

    if((cubes[0][0].ics == 2))
    {
        aux=0;
        aux1=0;
    }
}

```

```

sub=0;
sub1=0;
naux=0;
naux1=0;
for(m=0;m<num_implicant;m++)
{
    if(r1=cover(&cubes[0][0],&implcubes[0][m],&tempcube))
    {
        sub++;
        dum=sub - 1;
        swap(&tempi[0][dum],&implcubes[0][m],&tempcube);
        tempi[0][dum].track = m;
    }
}

/* ----- */

if(tempi[0][0].ic > tempi[0][1].ic)
{
    tempi[0][0].selected = 1;
    aux=tempi[0][0].track;
    aux1=tempi[0][1].track;
    swap(&tempi[0][0],&implcubes[0][aux],&temporarycube);
    swap(&tempi[0][1],&implcubes[0][aux1],&temporarycube);

    for(j=0;j<num_minterm;j++)
    {
        if((rz=cover(&cubes[0][j],&implcubes[0][aux],&temporarycube)) &&
            (cubes[0][j].ics != maxcint ))
        {
            cubes[0][j].ics = maxcint;
            isonum++;
            for(k=0;k<num_implicant;k++)
            {
                if((r=cover(&cubes[0][j],&implcubes[0][k],&temporarycube)) &&
                    (implcubes[0][k].selected != 1) && (implcubes[0][k].ic>0))
                {
                    --implcubes[0][k].ic;
                }
            }
        }
    }

    if(isonum == num_minterm)
    {
        goto EXIT;
    }
    goto REC;
}

/* ----- */

if(tempi[0][1].ic > tempi[0][0].ic)
{
    tempi[0][1].selected = 1;
    aux=tempi[0][1].track;
    aux1=tempi[0][0].track;
    swap(&tempi[0][1],&implcubes[0][aux],&tempcube);

```

```

swap(&tempi[0][0], &implcubes[0][aux1], &tempcube);
for(j=0; j<num_minterm; j++)
{
    if((rz=cover(&cubes[0][j], &implcubes[0][aux], &tempcube)) &&
        (cubes[0][j].ics != maxcint))
    {
        cubes[0][j].ics = maxcint;
        isonum++;
        for(k=0; k<num_implicant; k++)
        {
            if((r=cover(&cubes[0][j], &implcubes[0][k], &tempcube)) &&
                (implcubes[0][k].selected != 1) && (implcubes[0][k].ic > 0))
            {
                --implcubes[0][k].ic;
            }
        }
    }
}
if(isonum == num_minterm)
{
    goto EXIT;
}
goto REC;
}

```

/* ----- */

```

if((tempi[0][0].ic == 1) && (tempi[0][1].ic == 1))
{
    for(m=0; m<num_minterm; m++)
    {
        if(r1=combinable(&cubes[0][m], &cubes[0][0], &tempcube))
        {
            sub1++;
            dum1=sub1 - 1;
            swap(&nebor[0][dum1], &cubes[0][m], &tempcube);
            nebor[0][dum1].track = m;
        }
    }
}

```

/* ----- */

```

if((nebor[0][0].oldics == 2) && (nebor[0][1].oldics != 2))
{
    if(r1=cover(&nebor[0][0], &tempi[0][0], &tempcube))
    {
        tempi[0][0].selected = 1;
        aux=tempi[0][0].track;
        aux1=tempi[0][1].track;
        swap(&tempi[0][0], &implcubes[0][aux], &temporarycube);
        swap(&tempi[0][1], &implcubes[0][aux1], &temporarycube);
        naux=nebor[0][0].track;
        naux1=nebor[0][1].track;
        swap(&nebor[0][0], &cubes[0][naux], &temporarycube);
        swap(&nebor[0][1], &cubes[0][naux1], &temporarycube);
        for(j=0; j<num_minterm; j++)

```

```

{
    if((rz=cover(&cubes[0][j],&implcubes[0][aux],&temporarycube)) &&
        (cubes[0][j].ics != maxcint ))
    {
        cubes[0][j].ics = maxcint;
        isonum++;
        for(k=0;k<num_implicant;k++)
        {
            if((r=cover(&cubes[0][j],&implcubes[0][k],&temporarycube)) &&
                (implcubes[0][k].selected != 1) && (implcubes[0][k].ic > 0 ))
            {
                --implcubes[0][k].ic;
            }
        }
    }
}
if(isonum == num_minterm)
{
    goto EXIT;
}
goto REC;
}
else
{
    tempi[0][1].selected = 1;
    aux=tempi[0][1].track;
    aux1=tempi[0][0].track;
    swap(&tempi[0][1],&implcubes[0][aux],&tempcube);
    swap(&tempi[0][0],&implcubes[0][aux1],&tempcube);

    naux=nebor[0][0].track;
    naux1=nebor[0][1].track;
    swap(&nebor[0][0],&cubes[0][naux],&temporarycube);
    swap(&nebor[0][1],&cubes[0][naux1],&temporarycube);
    for(j=0;j<num_minterm;j++)
    {
        if((rz=cover(&cubes[0][j],&implcubes[0][aux],&tempcube)) &&
            (cubes[0][j].ics != maxcint ))
        {
            cubes[0][j].ics = maxcint;
            isonum++;
            for(k=0;k<num_implicant;k++)
            {
                if((r=cover(&cubes[0][j],&implcubes[0][k],&tempcube)) &&
                    (implcubes[0][k].selected != 1) && (implcubes[0][k].ic>0))
                {
                    --implcubes[0][k].ic;
                }
            }
        }
    }
}
if(isonum == num_minterm)
{
    goto EXIT;
}
goto REC;
}

```

```

    }
}

/* ----- */

if((nebor[0][1].oldics == 2) && (nebor[0][0].oldics != 2))
{
    if(rl=cover(&nebor[0][1],&tempi[0][1],&tempcube))
    {
        tempi[0][1].selected = 1;
        aux=tempi[0][1].track;
        aux1=tempi[0][0].track;
        swap(&tempi[0][1],&implcubes[0][aux],&tempcube);
        swap(&tempi[0][0],&implcubes[0][aux1],&tempcube);

        naux=nebor[0][0].track;
        naux1=nebor[0][1].track;
        swap(&nebor[0][0],&cubes[0][naux],&temporarycube);
        swap(&nebor[0][1],&cubes[0][naux1],&temporarycube);
        for(j=0; j<num_minterm; j++)
        {
            if((rz=cover(&cubes[0][j],&implcubes[0][aux],&tempcube)) &&
                (cubes[0][j].ics != maxcint))
            {
                cubes[0][j].ics = maxcint;
                isonum++;
                for(k=0; k<num_implicant; k++)
                {
                    if((r=cover(&cubes[0][j],&implcubes[0][k],&tempcube)) &&
                        (implcubes[0][k].selected != 1) && (implcubes[0][k].ic>0))
                    {
                        --implcubes[0][k].ic;
                    }
                }
            }
        }
        if(isonum == num_minterm)
        {
            goto EXIT;
        }
        goto REC;
    }
}
else
{
    tempi[0][0].selected = 1;
    aux=tempi[0][0].track;
    aux1=tempi[0][1].track;
    swap(&tempi[0][0],&implcubes[0][aux],&temporarycube);
    swap(&tempi[0][1],&implcubes[0][aux1],&temporarycube);

    naux=nebor[0][0].track;
    naux1=nebor[0][1].track;
    swap(&nebor[0][0],&cubes[0][naux],&temporarycube);
    swap(&nebor[0][1],&cubes[0][naux1],&temporarycube);
    for(j=0; j<num_minterm; j++)
    {

```



```

        if((rz=cover(&cubes[0][j],&implcubes[0][aux],&temporarycube)) &&
            (cubes[0][j].ics != maxcint ))
        {
            cubes[0][j].ics = maxcint;
            isonum++;
            for(k=0;k<num_implicant;k++)
            {
                if((r=cover(&cubes[0][j],&implcubes[0][k],&temporarycube)) &&
                    (implcubes[0][k].selected != 1) && (implcubes[0][k].ic>0))
                {
                    --implcubes[0][k].ic;
                }
            }
        }
    }
    if(isonum == num_minterm)
    {
        goto EXIT;
    }
    goto REC;
}
}

```

/* ----- */

```

if((nebor[0][0].oldics !=2 && nebor[0][1].oldics !=2)
    (nebor[0][0].oldics ==2 && nebor[0][1].oldics ==2))
{
    tempi[0][0].selected = 1;
    aux=tempi[0][0].track;
    aux1=tempi[0][1].track;
    swap(&tempi[0][0],&implcubes[0][aux],&temporarycube);
    swap(&tempi[0][1],&implcubes[0][aux1],&temporarycube);
    naux=nebor[0][0].track;
    naux1=nebor[0][1].track;
    swap(&nebor[0][0],&cubes[0][naux],&temporarycube);
    swap(&nebor[0][1],&cubes[0][naux1],&temporarycube);
    for(j=0;j<num_minterm;j++)
    {
        if((rz=cover(&cubes[0][j],&implcubes[0][aux],&temporarycube)) &&
            (cubes[0][j].ics != maxcint ))
        {
            cubes[0][j].ics = maxcint;
            isonum++;
            for(k=0;k<num_implicant;k++)
            {
                if((r=cover(&cubes[0][j],&implcubes[0][k],&temporarycube)) &&
                    (implcubes[0][k].selected != 1) && (implcubes[0][k].ic>0))
                {
                    --implcubes[0][k].ic;
                }
            }
        }
    }
    if(isonum == num_minterm)
    {

```

```

        goto EXIT;
    }
    goto REC;
}

/* ----- */

if((tempi[0][0].ic == 2) && (tempi[0][1].ic == 2))
{
    for(m=0; m<num_minterm; m++)
    {
        if(rl=combinable(&cubes[0][0], &cubes[0][m], &tempcube))
        {
            sub1++;
            dum1=sub1 - 1;
            swap(&nebor[0][dum1], &cubes[0][m], &tempcube);
            nebor[0][dum1].track = m;
        }
    }
    if((nebor[0][0].oldics == 2) && (nebor[0][1].oldics != 2))
    {
        if(rl=cover(&nebor[0][0], &tempi[0][0], &tempcube))
        {
            tempi[0][0].selected = 1;
            aux=tempi[0][0].track;
            aux1=tempi[0][1].track;
            swap(&tempi[0][0], &implcubes[0][aux], &temporarycube);
            swap(&tempi[0][1], &implcubes[0][aux1], &temporarycube);

            naux=nebor[0][0].track;
            naux1=nebor[0][1].track;
            swap(&nebor[0][0], &cubes[0][naux], &temporarycube);
            swap(&nebor[0][1], &cubes[0][naux1], &temporarycube);

            for(j=0; j<num_minterm; j++)
            {
                if((rz=cover(&cubes[0][j], &implcubes[0][aux], &temporarycube)) &&
                    (cubes[0][j].ics != maxcint))
                {
                    cubes[0][j].ics = maxcint;
                    isonum++;
                    for(k=0; k<num_implicant; k++)
                    {
                        if((r=cover(&cubes[0][j], &implcubes[0][k], &temporarycube)) &&
                            (implcubes[0][k].selected != 1) && (implcubes[0][k].ic>0))
                        {
                            --implcubes[0][k].ic;
                        }
                    }
                }
            }
            if(isonum == num_minterm)
            {
                goto EXIT;
            }
        }
    }
}

```

```

    }
    goto REC;
}
else
{
    tempi[0][1].selected = 1;
    aux=tempi[0][1].track;
    aux1=tempi[0][0].track;
    swap(&tempi[0][1],&implcubes[0][aux],&tempcube);
    swap(&tempi[0][0],&implcubes[0][aux1],&tempcube);

    naux=nebor[0][0].track;
    naux1=nebor[0][1].track;
    swap(&nebor[0][0],&cubes[0][naux],&temporarycube);
    swap(&nebor[0][1],&cubes[0][naux1],&temporarycube);
    for(j=0; j<num_minterm; j++)
    {
        if((rz=cover(&cubes[0][j],&implcubes[0][aux],&tempcube)) &&
            (cubes[0][j].ics != maxcint ))
        {
            cubes[0][j].ics = maxcint;
            isonum++;
            for(k=0; k<num_implicant; k++)
            {
                if((r=cover(&cubes[0][j],&implcubes[0][k],&tempcube)) &&
                    (implcubes[0][k].selected != 1) && (implcubes[0][k].ic>0))
                {
                    --implcubes[0][k].ic;
                }
            }
        }
    }
    if(isonum == num_minterm)
    {
        goto EXIT;
    }
    goto REC;
}
}

```

/* ----- */

```

if((nebor[0][1].oldics == 2) && (nebor[0][0].oldics !=2))
{
    if(rl=cover(&nebor[0][1],&tempi[0][0],&tempcube))
    {
        tempi[0][0].selected = 1;
        aux=tempi[0][0].track;
        aux1=tempi[0][1].track;
        swap(&tempi[0][0],&implcubes[0][aux],&temporarycube);
        swap(&tempi[0][1],&implcubes[0][aux1],&temporarycube);

        naux=nebor[0][0].track;
        naux1=nebor[0][1].track;
        swap(&nebor[0][0],&cubes[0][naux],&temporarycube);
        swap(&nebor[0][1],&cubes[0][naux1],&temporarycube);
    }
}

```

```

for(j=0; j<num_minterm; j++)
{
    if((rz=cover(&cubes[0][j], &implcubes[0][aux], &temporarycube)) &&
        (cubes[0][j].ics != maxcint))
    {
        cubes[0][j].ics = maxcint;
        isonum++;
        for(k=0; k<num_implicant; k++)
        {
            if((r=cover(&cubes[0][j], &implcubes[0][k], &temporarycube)) &&
                (implcubes[0][k].selected != 1) && (implcubes[0][k].ic>0))
            {
                --implcubes[0][k].ic;
            }
        }
    }
}
if(isonum == num_minterm)
{
    goto EXIT;
}
goto REC;
}
else
{
    tempi[0][1].selected = 1;
    aux=tempi[0][1].track;
    aux1=tempi[0][0].track;
    swap(&tempi[0][1], &implcubes[0][aux], &tempcube);
    swap(&tempi[0][0], &implcubes[0][aux1], &tempcube);

    naux=nebor[0][0].track;
    naux1=nebor[0][1].track;
    swap(&nebor[0][0], &cubes[0][naux], &temporarycube);
    swap(&nebor[0][1], &cubes[0][naux1], &temporarycube);
    for(j=0; j<num_minterm; j++)
    {
        if((rz=cover(&cubes[0][j], &implcubes[0][aux], &tempcube)) &&
            (cubes[0][j].ics != maxcint))
        {
            cubes[0][j].ics = maxcint;
            isonum++;
            for(k=0; k<num_implicant; k++)
            {
                if((r=cover(&cubes[0][j], &implcubes[0][k], &tempcube)) &&
                    (implcubes[0][k].selected != 1) && (implcubes[0][k].ic>0))
                {
                    --implcubes[0][k].ic;
                }
            }
        }
    }
}
if(isonum == num_minterm)
{
    goto EXIT;
}

```

```

        goto REC;
    }
}

/* ----- */

if((nebor[0][0].oldics == 2 && nebor[0][1].oldics ==2)
    (nebor[0][0].oldics != 2 && nebor[0][1].oldics !=2))
{
    tempi[0][0].selected = 1;
    aux=tempi[0][0].track;
    aux1=tempi[0][1].track;
    swap(&tempi[0][0],&implcubes[0][aux],&temporarycube);
    swap(&tempi[0][1],&implcubes[0][aux1],&temporarycube);

    naux=nebor[0][0].track;
    naux1=nebor[0][1].track;
    swap(&nebor[0][0],&cubes[0][naux],&temporarycube);
    swap(&nebor[0][1],&cubes[0][naux1],&temporarycube);
    for(j=0;j<num_minterm;j++)
    {
        if((rz=cover(&cubes[0][j],&implcubes[0][aux],&temporarycube)) &&
            (cubes[0][j].ics != maxcint ))
        {
            cubes[0][j].ics = maxcint;
            isonum++;
            for(k=0;k<num_implicant;k++)
            {
                if((r=cover(&cubes[0][j],&implcubes[0][k],&temporarycube)) &&
                    (implcubes[0][k].selected != 1) && (implcubes[0][k].ic>0))
                {
                    --implcubes[0][k].ic;
                }
            }
        }
    }
    if(isonum == num_minterm)
    {
        goto EXIT;
    }
    goto REC;
}
}
}

```

/* ----- */

```

if(cubes[0][0].ics !=2)
{
    for(m=0;m<num_implicant;m++)
    {
        if(rc=cover(&cubes[0][0],&implcubes[0][m],&temporarycube) &&
            (implcubes[0][m].selected != 1) && (implcubes[0][m].ic >0))

```

```

{
implcubes[0][m].selected = 1;
implcubes[0][m].ic = 0;
for(j=0; j<num_minterm; j++)
{
    if(rz=cover(&cubes[0][j],&implcubes[0][m],&temporarycube)
        && (cubes[0][j].ics != maxcint))
    {
        cubes[0][j].ics = maxcint;
        isonum++;
        for(k=0; k<num_implicant; k++)
        {
            if(r=cover(&cubes[0][j],&implcubes[0][k],&temporarycube)
                && (implcubes[0][k].selected != 1)&&
                (implcubes[0][k].ic > 0 ))
            {
                --implcubes[0][k].ic;
            }
        }
    }
}
if(isonum == num_minterm)
{
    goto EXIT;
}
goto REC;
}
}
}

```

```

/* ----- */
EXIT: for(m=0; m<num_implicant; m++){
    if(implcubes[0][m].selected == 1)
    {
        printcube(&implcubes[0][m]);
        num_product++;
    }
}
fclose(fpol);
printf("%d n", num_product);
}

```

```

/* ----- */

```

```

Function imp_cov_size(a1,a2,a3)
Cube *a1,*a2,*a3;
{
    int i;
    int v_ics;
    int check;
    check =0;
    v_ics =0;

```

```

/* check that this minterm can be covered by this implicant or not */
for(i=0; i < numvar ; i++) {
    a3 ->tt[i] = AND(a1->tt[i], a2->tt[i]);
    a3 ->ff[i] = AND(a1->ff[i], a2->ff[i]);
    if((a3->tt[i] == a2->tt[i]) && (a3->ff[i] == a2->ff[i]))
    {
        check++;
    }
}

/* if all bits are same as implicant than returns as v_ics =1 */
/* Otherwise it remains as 0. */

if( check == numvar)
{
    v_ics=1;
}
return(v_ics);
}

```

```

/* ----- */

```

```

Function int coverage_size(a1)
Cube *a1;
{
    int i;
    int parameter;
    int messenger;
    parameter = 0;

    for(i=0; i < numvar; i++){
        if((a1->tt[i] == 0) && ( a1->ff[i] == 0)){
            parameter++;}
        if(parameter == 0){
            messenger =1;}

        if(parameter == 1){
            messenger = 2;}

        if(parameter == 2){
            messenger =4;}

        if(parameter == 3){
            messenger =8;}

        if(parameter == 4){
            messenger = 16;}

        if(parameter == 5){
            messenger = 32;}

        if(parameter ==6){
            messenger = 64;}
    }
}

```

```

    if(parameter == 7){
        messenger = 128;}

    if(parameter == 8){
        messenger = 256;}

    if(parameter == 9){
        messenger = 512;}

    return(messenger);
}

/* ----- */

Function int swap(a1,a2,a3)
Cube *a1,*a2,*a3;
{
    int i;
    for( i=0; i < numvar ; i++)
    {
        a3->tt[i] = a1->tt[i];
        a1->tt[i] = a2->tt[i];
        a2->tt[i] = a3->tt[i];

        a3->ff[i] = a1->ff[i];
        a1->ff[i] = a2->ff[i];
        a2->ff[i] = a3->ff[i];
    }
    a3->ics = a1->ics;
    a1->ics = a2->ics;
    a2->ics = a3->ics;

    a3->oldics = a1->oldics;
    a1->oldics = a2->oldics;
    a2->oldics = a3->oldics;

    a3->ics = a1->ics;
    a1->ics = a2->ics;
    a2->ics = a3->ics;

    a3->selected = a1->selected;
    a1->selected = a2->selected;
    a2->selected = a3->selected;

    a3->track = a1->track;
    a1->track = a2->track;
    a2->track = a3->track;

}

/* ----- */

```



```

Function int cover(a1,a2,a3)
Cube *a1,*a2,*a3;
{
  int i;
  int v_cover;
  int check;
  check=0;
  v_cover=0;

  for(i=0; i < numvar; i++){
    a3->tt[i] = AND(a1->tt[i],a2->tt[i]);
    a3->ff[i] = AND(a1->ff[i],a2->ff[i]);
    if((a3->tt[i] == a2->tt[i]) && (a3->ff[i] == a2->ff[i]))
      {
        check++;
      }
  }

  if( check == numvar)
  {
    v_cover=1;
  }
  return(v_cover);
}

/* ----- */

```

```

Function int equalcubes(a1,a2)
Cube *a1,*a2;
{
  /* if EQ return 1 else return 0 */
  int i;
  int v_equal;
  v_equal =1;
  for (i = 0; i < numvar ; i++) {
    if ((a1->tt[i] != a2->tt[i] || a1->ff[i] != a2->ff[i]))
      {
        v_equal = 0;
      }
  }
  return(v_equal);
}

/* ----- */

```

```

Function int combinable(a1,a2)
Cube *a1,*a2;

```

```

{
    int i;
    int v_combinable;
    int cone;
    cone=0;          /* 1: combinable else not */
    v_combinable =0;
    for (i = 0; i < numvar ; i++) {
        if (((a1->tt[i] != a2->tt[i]))    ((a1->ff[i] != a2->ff[i])))
        {
            cone++;
        }
    }
    if (cone ==1) v_combinable=1;
    return(v_combinable);
}

```

/* ----- */

```

Function readfile(filename)
char *filename;
{
    int i,j;
    Cube *c1;          /* c1 point to c[0][0] */

    if((fpil=fopen(filename,"r")) == NULL)
    {
        printf(" nERROR - Cannot open designated read file n");
        return;
    }
    while(!feof(fpil)){

                                fscanf(fpil,"%d %d n", &num_minterm,&numvar);
                                numcubes[0] = num_minterm;
        for(i=0;i < num_minterm; i++){
            for(j=0;j < numvar; j++)
                fscanf(fpil,"%ld",&cubes[0][i].tt[j]); complement(&cubes[0][i]);
        }
    }
}

```

/* ----- */

```

Function int complement(a1)
Cube *a1;
{
    int j;
    for(j=0; j < numvar; j++)
    {
        if (a1 -> tt[j] == 1)
        {

```

```

        a1 -> ff[j] = 0;
    }
    else
        a1 -> ff[j] = 1 ;
    }
}

/* ----- */

Function combine(a1,a2,a3)
Cube *a1,*a2,*a3 ;
{
    int i;
    for (i = 0; i < maxvars ; i++) {
        a3->tt[i] = AND(a1->tt[i], a2->tt[i]) ;
        a3->ff[i] = AND(a1->ff[i], a2->ff[i]) ;
    }
}

/* ----- */

Function int AND(i1,i2)
int i1,i2;
{
    if(i1==i2) return(i1);
    else return(0);
}

/* ----- */

Function int printcube(a1)
Cube *a1 ;
{
    int j;

    for(j=0; j < numvar; j++){fprintf(fpol,"%ld", (a1->tt[j]));}
                                fprintf(fpol,"n");

    for(j=0; j<numvar; j++){fprintf(fpol,"%ld", a1->ff[j]); }
                                fprintf(fpol,"n");
                                fprintf(fpol,"n");
}

/* ----- */

```

```
Function int change(a1,a2)
Cube *a1,*a2;
{
  int i;
  for( i=0; i< numvar; i++)
  {
    a2->tt[i] = a1->tt[i];
    a2->ff[i] = a1->ff[i];
  }
}
```

LIST OF REFERENCES

1. P. Tirumalai and J. T. Butler, *Minimization Algorithms for Multiple-Valued Programmable Logic Arrays*, Forthcoming in IEEE Transactions on Computers.
2. J. K. Breeding, *Digital Design Fundamentals*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.
3. J. F. Wakerly, *Digital Design Principles and Practices*, Prentice Hall, Englewood Cliffs, New Jersey, 1990.
4. Y. Igaraski, *An Improved Lower Bound on the Maximum Number of Prime Implicants*, The Transactions on the IECE of Japan, Vol E62, June 1979, pp.389-394.
5. R. K. Brayton, Gary D. Hacht, *Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.
6. S. H. Unger, *The Essence of Logic Circuits*, Prentice Hall, Englewood Cliffs, New Jersey, 1989.
7. U. Manber, *Introduction to Algorithms*, Addison-Wesley, Menlo Park, 1989.

8. P. Tirumalai and J. T. Butler, *Analysis of Minimization Algorithms for Multiple-Valued Programmable Logic Arrays*, Proceeding of 18th International Symposium on Multiple-Valued Logic, May 1988 pp. 226-236.
9. Chyan Yang and Y. M. Wang, *A Neighborhood Decoupling Algorithm for Truncated Sum Minimization*, Proceeding of the 20th International Symposium on Multiple-Valued Logic, May 1990.
10. E. J. McCluskey, *Logic Design Principles with Emphasis on Testable Semicustom Circuits*, Prentice Hall, Englewood Cliffs, New Jersey, 1986.
11. H. Taub, *Digital Circuits and Processors*, McGraw-Hill, NY, 1982.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2. Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3. Deniz Kuvvetleri K.ligi Personel Sube Bsk.ligi Bakanliklar, Ankara / TURKEY	1
4. Kara Harp Okulu K.ligi Kutuphanesi Bakanliklar, Ankara / TURKEY	1
5. Deniz Harp Okulu K.ligi Kutuphanesi Tuzla, Istanbul / TURKEY	1
6. Hava Harp Okulu K.ligi Kutuphanesi Yesilyurt, Istanbul / TURKEY	1
7. Orta Dogu Teknik Universitesi Okul Kutuphanesi Balgat, Ankara / TURKEY	1
8. Bogazici Universitesi Okul Kutuphanesi Bebek, Istanbul / TURKEY	1
9. Ege Dokuz Eylul Universitesi Okul Kutuphanesi Cumhuriyet Meydani, Izmir / TURKEY	1
10. Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5000	1
11. Professor Chyan Yang Department of Electrical and Computer Engineering, Code EC:Ya Naval Postgraduate School Monterey, CA 93943-5000	2

- | | | |
|-----|--|---|
| 12. | Professor Jon T. Butler
Department of Electrical and Computer Engineering, Code EC/Bu
Naval Postgraduate School
Monterey, CA 93943-5004 | 2 |
| 13. | Ugur Ozkan
Haci Zihni Efendi Sok. No. 8/3
Capa, Istanbul / TURKEY | 3 |
| 14. | Kadri Hekimoglu
DeGol Caddesi No. 1/5
Tandogan, Ankara / TURKEY | 1 |