

AD-A238 018



2

NAVAL POSTGRADUATE SCHOOL Monterey, California



THESIS

DTIC
ELECTE
JUL 12 1991
S B D

THE USE OF
SEARCHING ALGORITHMS
FOR THE MINIMIZATION OF
MULTI-VALUED LOGIC FUNCTIONS

by

Alan W. Watts

June 1990

Thesis Advisor

Jon T. Butler

Approved for public release; distribution is unlimited.

81 91-04507



91 7 09 081

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) 32	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) THE USE OF SEARCHING ALGORITHMS FOR THE MINIMIZATION OF MULTI-VALUED LOGIC FUNCTIONS					
12 Personal Author(s) Alan W. Watts					
13a Type of Report Master's Thesis		13b Time Covered From To		14 Date of Report (year, month, day) June 1990	15 Page Count 47
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	Multi_Valued Logic, search, minimization, PLA, back-tracking		
19 Abstract (continue on reverse if necessary and identify by block number)					
<p>The goal of this thesis was to develop a searching algorithm for simplifying Multi-Valued Logic (MVL) functions. The algorithm was implemented as a program written in C for the UNIX operating system. The algorithm accepts an MVL function in the format required by HAMLET, an MVL computer-aided design tool, and produces a minimal or near-minimal realization. The output also conforms to that required by HAMLET to produce a layout of a programmable logic array (PLA) integrated circuit that realizes the given function.</p> <p>The advantage of the algorithm is that it allows backtracking to investigate alternate solutions, producing a greater expectation of minimal results. It stops upon finding a solution, thus producing results much faster than an exhaustive search of all possible solutions.</p>					
20 Distribution/Availability of Abstract			21 Abstract Security Classification		
<input checked="" type="checkbox"/> unclassified/unlimited <input type="checkbox"/> same as report <input type="checkbox"/> DTIC users			Unclassified		
22a Name of Responsible Individual Jon T. Butler			22b Telephone (include Area code) (408) 646-3299		22c Office Symbol EC/Bu

DD FORM 1473,84 MAR

83 APR edition may be used until exhausted
All other editions are obsolete

security classification of this page

Unclassified

Approved for public release; distribution is unlimited.

The Use of
Searching Algorithms
for the Minimization of
Multi-Valued Logic Functions

by

Alan W. Watts
Captain, United States Army
B.S., United States Military Academy, 1979

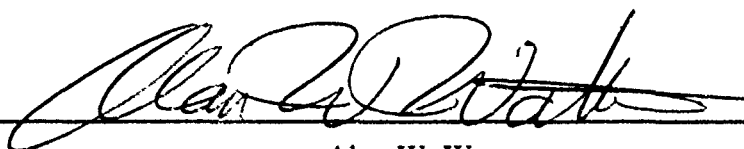
Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

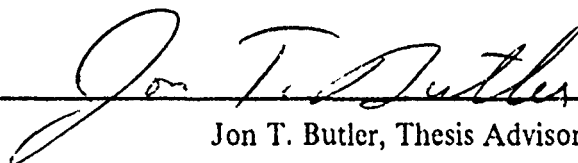
NAVAL POSTGRADUATE SCHOOL
June 1990

Author:

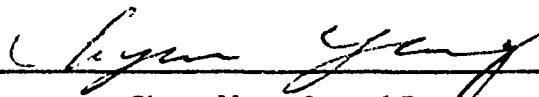


Alan W. Watts

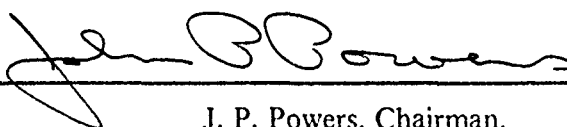
Approved by:



Jon T. Butler, Thesis Advisor



Chyan Yang, Second Reader



J. P. Powers, Chairman,

ABSTRACT

The goal of this thesis was to develop a searching algorithm for simplifying Multi-Valued Logic (MVL) functions. The algorithm was implemented as a program written in C for the UNIX operating system. The algorithm accepts an MVL function in the format required by HAMLET, an MVL computer-aided design tool, and produces a minimal or near-minimal realization. The output also conforms to that required by HAMLET to produce a layout of a programmable logic array (PLA) integrated circuit that realizes the given function.

The advantage of the algorithm is that it allows backtracking to investigate alternate solutions, producing a greater expectation of minimal results. It stops upon finding a solution, thus producing results much faster than an exhaustive search of all possible solutions.

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	



TABLE OF CONTENTS

I. INTRODUCTION	1
II. BACKGROUND	3
A. DEFINITIONS	3
1. MVL.	3
2. Sum of Products.	3
3. Minterm.	3
4. Implicant.	4
5. Don't-Cares.	4
B. PLA CIRCUIT TOOL ENVIRONMENT	4
1. HAMLET	5
2. Input Format	6
3. Output Format	6
III. DESCRIPTION OF MINIMIZATION ALGORITHM	8
A. DESCRIPTION OF THE SEARCH STRATEGY	8
1. Cost Function	10
2. Evaluation Function	11
B. UNIQUE CHARACTERISTICS OF THE PROPOSED APPROACH	12
IV. EVALUATION OF ALGORITHM PERFORMANCE	13
A. METHOD OF EVALUATION	13
B. RESULTS	13
1. RBC as Evaluation Function.	13
2. Number of Non-Zero Minterms as an Evaluation Function.	16
C. COMPARISON OF ALGORITHM PERFORMANCE	17
V. CONCLUSIONS	22
APPENDIX A. ALGORITHM DESCRIPTION	23

APPENDIX B. PROGRAM LISTING	25
LIST OF REFERENCES	36
INITIAL DISTRIBUTION LIST	38

LIST OF FIGURES

Figure 1. Four Valued MVL Function of Two Variables	5
Figure 2. Example Search Tree for MVL Function	9
Figure 3. MVL Function that Causes RBC to do Exhaustive Search	15
Figure 4. The average number of implicants versus the number of nonzero	20
Figure 5. The average elapsed time per function versus the number of	21

ACKNOWLEDGMENT

I would like to express my appreciation to Dr. J. T. Butler, my thesis advisor from the Department of Electrical and Computer Engineering of the Naval Postgraduate School. He provided invaluable encouragement and guidance throughout the research. Without him this research would not have been finished. I must also acknowledge the great assistance of CDR J. M. Yurchak from the Department of Computer Science of the Naval Postgraduate School. He provided access to computer systems capable of developing and running the research program. His computer code for HAMLET often provided the model used to write my programs. Without his assistance this research would not have been possible.

I must also express my thanks to my family for the many, many hours they left Daddy alone when he was banging on the computer.

I. INTRODUCTION

Demand for speed and performance in information processing will continue to increase. This, in turn, creates a need for low cost, fast Very Large Scale Integration (VLSI) devices. As attempts are made to extract better performance from each device, several problems surface. For example, present VLSI devices create a need for more pins to send and receive information and to control the processes within the device. Also, interconnecting devices on a chip is difficult and takes a large proportion of the chip area. If each line of a device could provide more information than a 1 or 0, then these problems would be partially solved. In addition, although we have not yet reached the physical limit of miniturization, there is a limit. A possible way to push back the limit is through Multi-Valued Logic (MVL). In MVL there are typically more logic levels than two.

As in binary, there is a need in MVL to generate sum-of-product expressions for implementation by a programmable logic array (PLA). If the number of product terms can be reduced, as in binary logic, the implementation as a PLA would be easier, smaller, and cheaper. Unfortunately, no method of finding the exact minimum solution for a MVL functions exists other than an exhaustive search. A search of all possible solutions for relatively complicated expressions is too time consuming.

The implementation of technologies to realize MVL circuits [Ref. 1,2,3] has inspired research on heuristics for simplifying MVL functions. Prominent among these are Pomper and Armstrong [Ref. 4], Besslich [Ref. 5], and Dueck and Miller [Ref. 6]. Tirumalai and Butler [Ref. 7, pp. 237-238] analyzed these heuristics and found that for only about 39% of the functions did Pomper and Armstrong find a minimal solution. Besslich and Dueck and Miller found minimal solutions for 52% of the functions. In more recent research using HAMLET [Ref. 8, p. 148] where the functions have been specified as a sum-of-products, the heuristics sometimes produced solutions with more terms than the original function. It is desirable to produce a heuristic that will give the designer a greater confidence of achieving a minimal or better solution without being prohibitively expensive.

Most current heuristics [Ref. 4,5,6] proceed in a straight-line manner from problem to solution. Once a term is chosen as part of the solution, it cannot be discarded in favor of a better choice later on. Attempts to improve existing heuristics have focused on in-

creasing the number of choices available at each step and in saving those choices in case there is a need to discard a part of the solution and start over [Ref. 8, pp. 149-150]. In short, heuristics are being formed which search for the minimal solution among several possible solutions.

The minimal solution can always be found using an exhaustive search of all possible solutions. Unfortunately, this is very expensive in terms of computer time. A middle route was chosen in this research. A search for a solution involves several possible choices, but not all. A searching heuristic is used rather than trying to modify a straight-line heuristic into a search. The method chosen is an A* (pronounced A star) search. This method is explained in detail in Chapter III.

II. BACKGROUND

A. DEFINITIONS

1. MVL.

Consider a multi-valued system in which there are r levels, $0, 1, \dots, r-1$. The quantity r is called the radix. There are operations with a behavior that is dependent on how we wish to represent the function.

2. Sum of Products.

There are several ways of representing an MVL function. The sum of products method is chosen. This method is preferred by most designers wishing to implement an MVL function as a Programmable Logic Array (PLA). It is also simple and straightforward. An expression for a sum of products MVL function of i variables and r values can be written as follows:

$$f(x_0, x_1, \dots, x_{i-1}) = \sum_{j=0}^{n-1} C_j \prod_{k=0}^{i-1} x_k(l_{jk}, u_{jk})$$

This is composed of:

- Literals $x(l, u)$

$$\begin{aligned} x(l, u) &= r - 1 \text{ if } l \leq x \leq u \\ &= 0 \text{ otherwise} \end{aligned}$$

In some notations, this is denoted ' x_l^u '.

- Products of a constant and literals. The product is implemented as a minimum function. That is, the product ab has the value $\min(a, b)$.

$$C \prod_{k=0}^{i-1} x_k(l_k, u_k) \text{ is the minimum value of all } x_k(l_k, u_k) \text{ and } C$$

- Sum. \sum represents the truncated sum operation $+$. That is, $a + b$ has a value which is the sum of a and b (with logic values viewed as integers), except when the sum exceeds $r-1$, in which case it is assigned the value $r-1$.

3. Minterm.

An MVL function can also be represented as a map such as the one in Figure 1. Figure 1 represents a MVL function of radix four with two inputs. If we designate a specific value for each of the input variables, the function assumes some value. If this

value is nonzero, it is called a minterm. For example if $x_1 = 0$ and $x_2 = 1$, the function assumes the value 3. The entry 3 is a minterm.

4. Implicant.

Several minterms can be grouped together to form a single product term. This product term is called an implicant. For example, the grouping of the four minterms:

$$x_1 = 1, x_2 = 1$$

$$x_1 = 2, x_2 = 1$$

$$x_1 = 1, x_2 = 2$$

$$x_1 = 2, x_2 = 2$$

is an implicant which can be written as $1x_1(1,2)x_2(1,2)$. Specifically, a minterm is covered by an implicant if the implicant is nonzero for the assignment of variables such that the minterm is nonzero. For example, the product term $1x_1(1,2)x_2(1,2)$ covers the four minterms listed above. Since implicants are added using a truncated sum, the value of a minterm can be determined by more than one implicant. For example, if exactly two implicants cover a minterm and both are 1, the value of the minterm is 2 (if the radix is three or greater).

5. Don't-Cares.

For a given MVL function, if we do not care what value a minterm is we call it simply a 'don't-care'. This is represented by the radix r in a map of the function. During simplification of the MVL function, the don't-care assumes any value that assists in reducing the number of product terms necessary to solve the function.

B. PLA CIRCUIT TOOL ENVIRONMENT

In order to adequately test the searching algorithm, it is necessary to perform a search on a large number of functions of different types and compare the algorithm's performance against other existing heuristics. Because of the large number of searches required and the large amount of record-keeping involved, the best approach is to implement the search as a computer program. A program allows many searches to be performed in a short period of time. If the search method is found to be superior, it may be included in a computer aided design (CAD) tool for use in circuit design or further research.

The program *astar* performs the A* search. It is designed to be compatible with HAMLET [Ref. 8]. *astar* is not currently available under HAMLET but accepts input

X1 X2		0	1	2	3
0					
1	3	1	1	3	
2		1	1		
3		2	2		

Figure 1. Four Valued MVL Function of Two Variables

files of identical format and produces output files of correct format for generation of a circuit layout.

1. HAMLET

HAMLET is a CAD tool written in C for a UNIX operating system. It is modular in design to facilitate future improvements in minimization heuristics and future MVL-PLA technologies. Its major components are:

- *mvlc* This module performs a function similar to *astar*. It takes an MVL function (or functions) of the proper format from a file and tries to find a simpler expression that describes the same function. It is capable of using Dueck and Miller [Ref. 6] or Pomper and Armstrong [Ref. 4] to find the simplest expression. It will also use both heuristics and pick the best solution (called the Gold heuristic), or perform a search variation on either Dueck and Miller or Pomper and Armstrong. *astar* takes the place of *mvlc* to simplify the given MVL function.
- *mvll* This module takes the simplified function from *mvlc* and constructs an MVL PLA circuit that implements the function. Currently HAMLET implements the circuit using current-mode CMOS technology. The layout is in MAGIC format, suitable for submission to a chip manufacturer.
- *mvla* This module analyzes the performance of a heuristic. It takes performance data from successive runs of *mvlc* and produces various statistical data. *mvla* was used to analyze *astar*.
- *mvlt* This module generates random test functions to user specifications of proper format. *mvlt* was used to generate the test functions for *astar*.

2. Input Format

The input to *astar* is a file containing one or more functions. The function(s) must be expressed as a sum-of-products in the following format:

radix of the function: number of inputs:

For each implicant-

+first coefficient*X1(lower bound,upper bound)*...

There must be a lower and upper bound for each input.

The last implicant ends with a semi-colon.

The function in Figure 1 would be expressed as:

4: 2:

+3*X1(0,0)*X2(1,1)

+1*X1(1,2)*X2(1,2)

+3*X1(3,3)*X2(1,1)

+2*X1(1,2)*X2(3,3);

3. Output Format

astar must express the simplified function in the following format in order for *mvll* to convert it into a PLA.

number of inputs (blank) number of outputs

For each expression in the solution-

coeff

For each input variable-

lower upper

...

99

The function in Figure 1 would look like the following if it were formatted for input into *mvll*

2 1

3

0 0

1 1

1

1 2

1 2

3

3 3

1 1

2

1 2

3 3

99

III. DESCRIPTION OF MINIMIZATION ALGORITHM

In order to find a simplified solution to the MVL function, a searching algorithm was used. The starting point is the MVL function that needs to be minimized. Any one of several implicants can be chosen that covers a part of the function. Each implicant is a branch in the search. Once an implicant is chosen and subtracted from the original function, a smaller, simpler function remains. This function also has several choices for a next implicant. The choices are represented by branches in a tree similar to the one found at Figure 2. The search looks for the shortest depth of the search tree that completely describes the function.

Figure 2 is a possible search tree for the function in Figure 1. The circles represent nodes. Each node is an MVL function with the top node being the original MVL function and the subnodes being smaller MVL functions. These subnodes result from subtracting an implicant from the parent node. The branches are labeled with the implicant they represent. The solution is the sequence of branches on the far left expressed by $1x_1(1,2)x_2(1,3) + 1x_1(1,2)x_2(3,3) + 3x_1(0,0) + 3x_1(3,3)x_2(1,1)$.

The criteria used to choose the next branch to be investigated constitutes a searching strategy. The strategy we will use is an A* search strategy [Ref. 9, p. 203]. The A* search is characterized by determining the cost for choosing each branch and evaluating it for closeness to a final solution. A measure of the overall worth of a branch is found by summing the value of the cost and evaluation functions. The effect is for the evaluation function to steer the solution towards what seems to be the best solution. The cost function brings us back to try alternative branches when the number of implicants in the expression becomes large.

A. DESCRIPTION OF THE SEARCH STRATEGY

A formal description of the algorithm of the computer program is found at Appendix 1. An informal description follows.

The search method will be described as if it were already in the middle of a search. The procedure is the same at the beginning as in the middle except that the starting node is the original MVL function and the agenda is empty.

The starting node is the MVL function that requires simplification. Branches and subnodes must be generated. It is possible to generate a branch and a subnode for every possible implicant of the original MVL function. Unfortunately, this would generate an

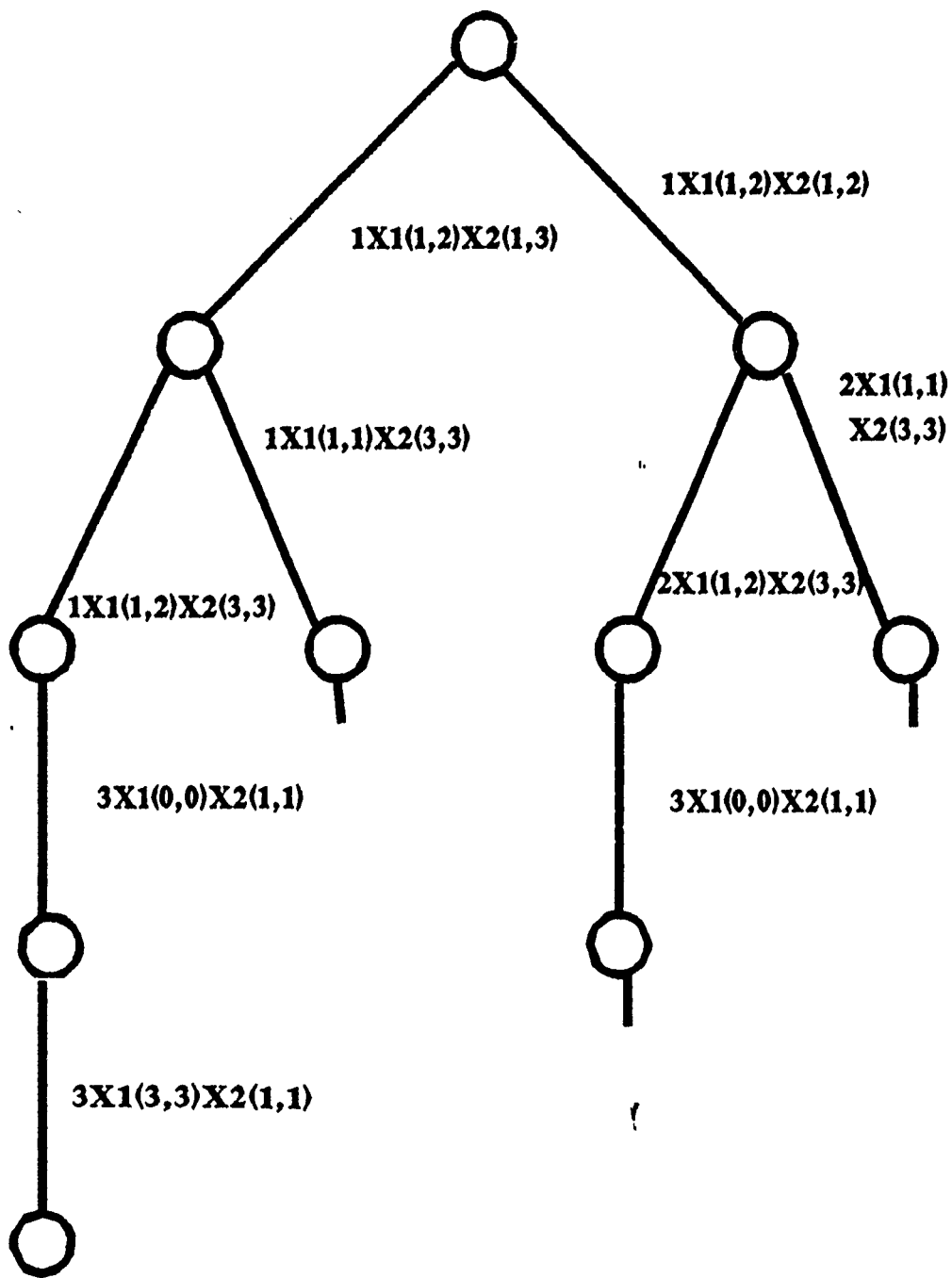


Figure 2. Example Search Tree for MVL Function

extremely large tree. Fortunately, this is not required. Since any solution must cover each minterm, any minterm may be chosen and used to generate all the implicants that cover it. One of these implicants must be in the solution. Of the minterms chosen to be covered, certain are covered by fewer implicants than others. Dueck and Miller [Ref. 6, p. 223] call such minterms "most isolated minterms (mim)". Choosing the most isolated minterm tends to reduce the search. Isolation is a measure of how many possible combinations a minterm has with neighboring terms. If several minterms are equally isolated, the first one found is chosen.

Having chosen the most isolated minterm, all the implicants that cover the minterm are generated. If the value of the minterm is $r - 1$ or a don't-care, we also generate implicants that include the minterm with values from 0 to $r - 1$. For each of these implicants, we determine the cost and the evaluation function. Each of these implicants with their cost and evaluation are put on a list of available branches called an agenda. The branch from the agenda which has the smallest sum of cost and evaluation is chosen. The agenda contains not only the branches just generated, but all branches previously generated and not yet chosen. If several branches on the agenda have equal sums, the one most recently added is chosen. This branch should have been generated further along in our search and will hopefully be closer to a solution. The effect is to push the search toward a solution and eliminate unnecessary backtracking.

Having chosen a branch (an implicant), it is subtracted from the MVL function. This leaves a smaller, simpler function to solve. The chosen branch is taken off of the agenda so we will not consider it again. We now repeat the process on the smaller MVL function, generating more branches and putting them on the agenda. We stop the search when the function is completely covered, i.e., when there are no more implicants to generate. The solution is the chosen branch (implicant) plus those branches used to solve the smaller function plus those branches chosen that brought us to this node.

1. Cost Function

The key to a successful A* search is to choose an appropriate cost and evaluation function. They guide the search to the solution. If chosen correctly, they do this quickly with few unnecessary branches. In this case, the number of implicants should be minimized. Each implicant in the expression is another column in the PLA circuit implementation [Ref. 10, p. 15, 23-24]. Therefore, the number of implicants used to that point is chosen as the initial cost function. For example, if a node (simplified function) used a sequence of three branches (implicants) to get to it, all branches from the node will have a cost of four, there being four implicants needed to reach the subnode.

Other options may be considered for the cost function. The number of implicants may be weighted in order for its magnitude to balance that of the evaluation function. In such a case, the cost would then be the number of implicants used times the weighting factor. There is also a second alternative. The cost of implementing a MVL function into circuitry may also be a function of the number of inputs required [Ref. 11]. The number of implicants determines the number of columns. The number of inputs determines the height of the columns or the number of rows [Ref. 10, pp. 23-24]. Therefore another cost function might be the the number of implicants multiplied by the number of variables.

2. Evaluation Function

The evaluation function is an estimate of how close the current expression is to the solution. As the search progresses deeper into the tree, the values of an evaluation function grow smaller (as we get closer to a solution). Therefore the cost and evaluation functions must be minimized. To do this, add the cost and evaluation functions, producing a criterion number. If chosen properly, the most advantageous branch will be the one with the smallest criterion number. In that way, both the cost and the evaluation functions are minimized.

Two evaluation functions are investigated, the number of non-zero minterms and the relative break count (RBC). The objective of our search is to completely cover the MVL function. That is, the search seeks a function containing zeros or don't-cares exclusively. Therefore a good measure of closeness to a solution is the number of minterms that are neither zero nor a don't-care. This number will grow smaller as we near a solution. For simplicity, this is called the number of non-zeros.

Another measure of closeness to a solution is called the relative break count (RBC) [Ref. 6, p. 224]. Simply stated, a break is where two adjacent minterms have different values. For example, in Figure 1 there is a break between $x_1 = 0, x_2 = 1$ and $x_1 = 1, x_2 = 1$. The number of breaks in a MVL function is the break count. Subtracting an implicant from an MVL function may result in an MVL function with a different break count. This change in the break count due to the implicant is the RBC. RBC is related to the number of breaks introduced minus the number of breaks removed. If an implicant introduces breaks, its RBC will have a positive number related to the number of breaks added. If an implicant takes out breaks, its RBC will be a negative number of magnitude related to the number of breaks removed. MVL functions with few breaks can have many minterms grouped into implicants. This produces a solution with few implicants, which is desired. If there are a lot of breaks we will be unable to group

minterms into implicants easily and our solution will probably have more implicants. Therefore, if an implicant with a large (positive) RBC is chosen, more implicants are needed to completely cover the function. While if an implicant with a small (negative) number is chosen, it will probably develop into path with a simpler solution. In this way, the RBC can be used as an evaluation function.

B. UNIQUE CHARACTERISTICS OF THE PROPOSED APPROACH

The difference between this research and other heuristics is that this research used a searching strategy to find a simplified solution rather than a straight-line approach or a straight-line approach modified to search. The other heuristics mentioned (Pomper and Armstrong, Besslich, and Dueck and Miller) are all straight-line heuristics with no branches. They use one criteria to determine the next implicant chosen. When they choose an implicant it must be used in the final solution. All alternative choices are lost. The heuristics do not allow backtracking. Some research has been done to expand these heuristics into more of a search method [Ref. 6, pp. 152-153]. This usually done by saving a fixed number of implicants at each node. This approach potentially saves too many implicants and slows down the search or saves too few implicants and misses a simpler solution.

The proposed approach is a search strategy with backtracking. It may backtrack and choose an alternate solution whenever it appears that the alternate is better. Potentially, all the implicant branches are available, if necessary, to find the simplest solution. Yet, it generates only those branches needed and available. It uses two criteria (cost and evaluation functions) to guide it to a solution.

IV. EVALUATION OF ALGORITHM PERFORMANCE

A. METHOD OF EVALUATION

In order to evaluate the performance of the A* search, its performance must be analyzed for several MVL functions and compared to some baseline. Dueck and Miller is chosen as the baseline since its performance is the best of the existing heuristics. The performance of A* in minimizing two-input MVL functions of radix 4 is analyzed and compared to the performance of Dueck and Miller on the same functions. MVL functions of six or fewer non-zero minterms are relatively simple and all heuristics perform the same. Therefore, only functions with six or more non-zero minterms were analyzed. Five hundred functions were generated for each number of non-zero minterms.

Two different versions of A* were evaluated. The first version uses the number of implicants as the cost function and RBC as the evaluation function. The second version also uses the number of implicants as the cost function, but uses the number of non-zero minterms as the evaluation function. *mvlt* from HAMLET was used to generate 500 random sample functions for each number of non-zero minterms from six to 16. Each version of A* was used to minimize the same sets of sample functions. The results were analyzed using *mvla* from HAMLET. *mvla* provided information on the average number of implicants needed in the solution, the average time to solve, the ratio of number of heuristic implicants to number of given implicants and their standard deviations. The same functions were solved using *mvlc* from HAMLET using the Dueck and Miller heuristic. This provided the baseline for comparison.

An exhaustive search for the absolute minimum solution was not performed due to the excessive computer time required. To perform an exhaustive search on one function took four hours. In this research, we solved 5500 functions for each heuristic. However, the values for average number of implicants for Dueck and Miller were very close to those found in Tirumalai and Butler [Ref. 7 p. 242]. We will therefore use their values for average number of implicants for absolute minimization.

B. RESULTS

1. RBC as Evaluation Function.

The first version of A* used the number of implicants as the cost function and the RBC as the evaluation function. Trial runs found that this version was unacceptably slow. One run of 100 functions was not completed after running 36 hours. Investigation

showed that this version acted, like a breadth-first search for functions with few zeros. In other words, the search would investigate every node in the first level and generate every node in the second level before it began to investigate nodes on the second level. This behavior resulted in an exhaustive search for a solution. This was because, in functions of few zeros, the first level implicants could eliminate most of the breaks resulting in very negative RBCs. At the second level, there were relatively few breaks left to eliminate so the RBCs of these implicants were only slightly negative. Therefore, in choosing the implicant with the smallest sum of cost and evaluation functions, the search always picked the first level implicants. That the cost was greater for the second level implicants only made the situation worse. An example of this effect can be found in Figure 3.

The initial most isolated minterm (mim) chosen for the function in Figure 3 is $1x_1(0, 0)x_2(0, 0)$. This minterm has 16 implicants that range from $1x_1(0, 3)x_2(0, 3)$ with an RBC of -24 to $1x_1(0, 0)x_2(0, 0)$ with an RBC of $+1$. The algorithm correctly chooses $1x_1(0, 3)x_2(0, 3)$. This node has an rim of $2x_1(2, 2)x_2(0, 0)$. This mim generates four implicants. This obvious choice is $2x_1(?, 3)x_2(0, 1)$. This has an RBC of -8 and a sum of cost and RBC of -6 . Unfortunately, there are five other implicants on the first level with sums lower than -6 . All these implicants must be chosen and investigated before we can get to it. Three of these implicants have branches that also have implicants with RBCs lower or equal to -6 . The algorithm always looks at the most recently added implicants first, so all these implicants must also be investigated before we chose $2x_1(2, 3)x_2(0, 1)$. The algorithm correctly chooses the minimal expression of two implicants, but only after a long search.

The problem with RBC as an evaluation function is that it has no memory. It only looks at the present node to determine its value. Previous good choices cannot influence the value of RBCs for the present node. Yet, RBC is an excellent way to determine the best implicant among several implicants at the same node. To give the evaluation function a memory, it becomes the sum of the RBC of the implicant plus the evaluation function of the current node. The current node is the node (MVL function) for which we are generating the branches (implicants). The evaluation function then becomes the sum of the RBC for the implicant plus the RBCs of all the implicants needed to get to it.

This makes the evaluation function very strong, in fact too strong. The evaluation function drives the search deeper and deeper into the tree always using the best RBC. The cost function is not strong enough to pull the search back and do some

X1 \ X2		0	1	2	3
X2	0	1	1	3	3
	1	1	1	3	3
	2	1	1	1	1
	3	1	1	1	1

Figure 3. MVL Function that Causes RBC to do Exhaustive Search

backtracking if the current solution becomes too long. The search behaves just like a Dueck and Miller heuristic. Having strengthened the evaluation function so the search will look deeper, faster, we must also strengthen the cost function so they will be in balance. As mentioned in Chapter III, an alternate cost function could be the number of implicants multiplied by the number of variables. The number of variables is the number of inputs to the function. This number does not change during the solution of

a given function, but does allow the algorithm to adjust to different functions with different numbers of inputs. This new cost function would be a relative measure of the area of the PLA, something that should be kept as small as possible. When this version of A* was evaluated, its performance was better than Dueck and Miller. That is, the average number of implicants needed to solve the function was less than Dueck and Miller. However, the improvement was statistically insignificant.

Increasing the cost function by using a weighting factor would increase backtracking. Increased backtracking should improve the performance of the algorithm, but also increase the average elapsed time needed to solve for functions. Experiments were conducted to determine a suitable weighting factor. A weighting factor of two was found to cause sufficient backtracking for improved performance, yet did not cause the search to slow to unacceptable levels. When this is used as the cost function, the performance of the search avoids the two extremes. It is neither an exhaustive search nor a simple Dueck and Miller.

This search algorithm was evaluated for the full 5500 functions. The cost function was the number of implicants multiplied by the number of variables multiplied by a weighting factor of two. The evaluation function was the RBC of the implicant plus the evaluation function of the current node. The results follow.

2. Number of Non-Zero Minterms as an Evaluation Function.

This version was initially evaluated with the number of non-zero minterms as the evaluation function and number of implicants as the cost function. Don't-cares were treated like a zeros. Results from an earlier version of the program indicated that the version performed very well, achieving minimal or near-minimal expressions in a short period of time. The solutions were as good or better than Dueck and Miller. However, comparisons on a large number of difficult functions found that Dueck and Miller was actually performing better. Analysis showed that RBC is a better judge of the best implicant to choose than just choosing the implicant that covers the most minterms. That is why Dueck and Miller performs slightly better than other heuristics. The cost function of the search was not causing enough backtracking to find the minimum solutions of the more difficult functions.

If the accuracy of the cost function in predicting unsuitable implicants could be improved, then the search would do more backtracking. As mentioned in Chapter III and above with the RBC version, one way to obtain an improvement in the cost function is to make it the number of implicants multiplied by the number of variables. Unfortunately, when this was tried it produced a search as slow as the original RBC version.

Further research indicated that a value near that of number of implicants times the number of variables was needed, but the actual value was just too slow. A cost function of the value multiplied by a weighting factor of five-sixths was used. This seemed to produce better results than Dueck and Miller without being unacceptably slow.

The final version analyzed for 5500 functions used the number of non-zero minterms as the evaluation function and the number of implicants multiplied by the number of variables multiplied by a weighting factor of five-sixths as the cost factor. Others using this version of A* for MVL design or heuristic research may wish to experiment with other weighting factors to achieve the proper balance of performance and speed.

C. COMPARISON OF ALGORITHM PERFORMANCE

Table 1 shows the Average Number of Minterms used by the indicated algorithms to cover the MVL functions. The values for Absolute Minimization come from Tirumalai and Butler [Ref. 7, p. 242]. These values are not precisely correct for this research as the random functions solved by absolute minimization are not the same ones used in this research. This can be seen for the Number of Non-Zero Minterms equal to six and seven, where the average number of implicants is greater for Absolute Minimization than for any of the heuristics. Still, values for Dueck and Miller using random functions from this research are within one tenth of the values for Dueck and Miller using Tirumalai and Butler's functions for Number of Non-Zero Minterms greater than seven. If absolute minimization was performed on the random functions generated here, the average number of implicants would likely be within one-tenth of those given.

Table 1. AVERAGE NUMBER OF IMPLICANTS

Number of Non-Zero Minterms	RBC	Number of Non-Zero Minterms	Dueck and Miller	Absolute Minimization from Ref. 7
6	4.60	4.62	4.60	4.96
7	5.35	5.36	5.35	5.52
8	5.91	5.94	5.95	5.94
9	6.41	6.48	6.48	6.40
10	6.75	6.84	6.85	6.69
11	7.05	7.18	7.17	6.91
12	7.26	7.43	7.45	7.20
13	7.32	7.63	7.64	7.16
14	7.34	7.71	7.64	7.07
15	7.24	7.72	7.58	6.94
16	6.89	7.20	7.13	6.69

From this table we can see that A* using number of non-zero minterms performed the worst, having the largest average number of implicants for all classes of functions. A* using cumulative RBC performed as well or better than Dueck and Miller. The closeness of A* using cumulative RBC to Absolute Minimum indicates that the number of functions where Absolute Minimum is achieved is greater than Dueck and Miller.

There is however a trade-off. Table 2 shows the average elapsed CPU time in seconds needed to solve a function for each class of function.

Table 2. AVERAGE ELAPSED TIME PER FUNCTION

Number of Non-Zero Minterms	RBC	Number of Non-Zero Minterms	Dueck and Miller
6	1.378	2.500	0.334
7	2.726	4.217	0.404
8	3.814	5.137	0.475
9	5.247	6.364	0.597
10	6.346	7.810	0.693
11	8.621	10.672	0.808
12	11.673	15.412	0.928
13	16.899	20.461	1.003
14	16.531	18.349	1.172
15	12.805	14.639	1.304
16	10.115	12.833	1.383

As can be seen from Table 2, A* using number of non-zero minterms is the slowest algorithm as well as the one with the worst performance. A* using cumulative RBC is somewhat faster, but much slower than Dueck and Miller. Dueck and Miller is very fast. Table 2 does not tell the whole story. For both versions of A*, the standard deviation of average elapsed time was very large. For example, for 12 non-zero minterms, the average elapsed time is 11.673 seconds and the standard deviation is 16.158 seconds. The latter figure is surprisingly large. It results because some functions give near instantaneous solutions, while others may take up to 30 seconds to solve.

Another interesting observation from Tables 1 and 2 is that the most difficult functions are those with 12, 13, and 14 non-zero minterms. This is caused by the zero minterms forcing the implicants to go around them. In functions with few or no zeros, the implicants have more freedom to be larger and cover more minterms.

Figures 4 and 5 are graphical representations of Tables 1 and 2.

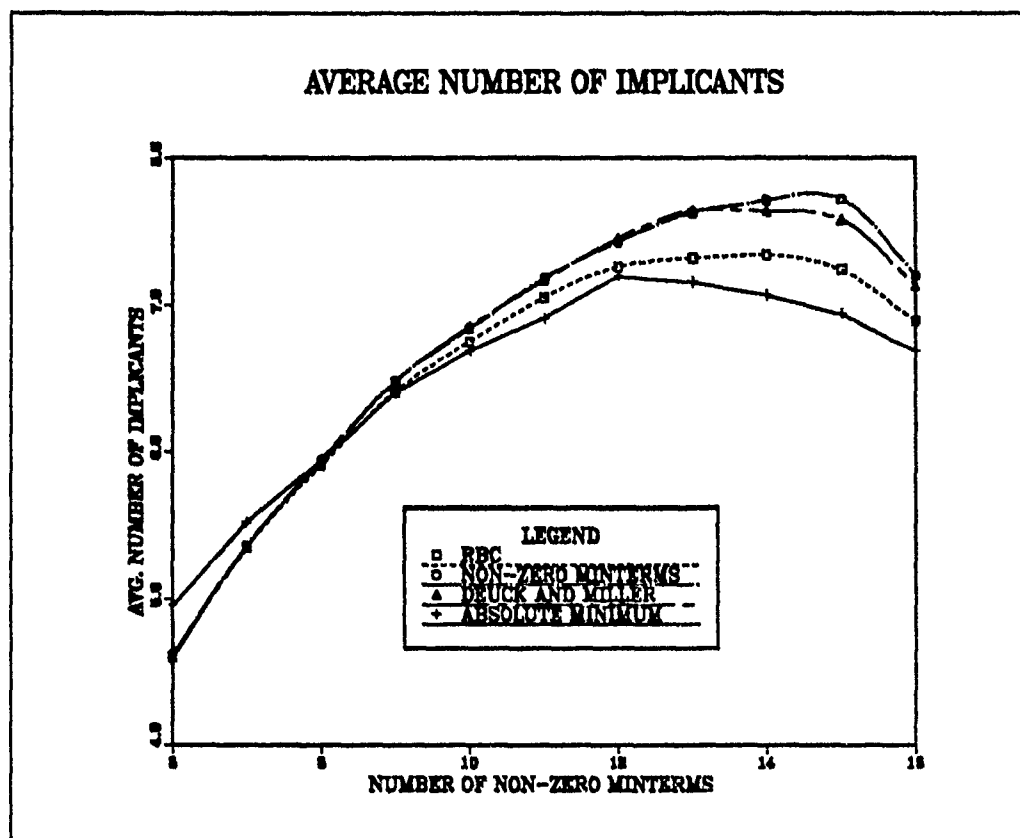


Figure 4. The average number of implicants versus the number of nonzero values for random functions

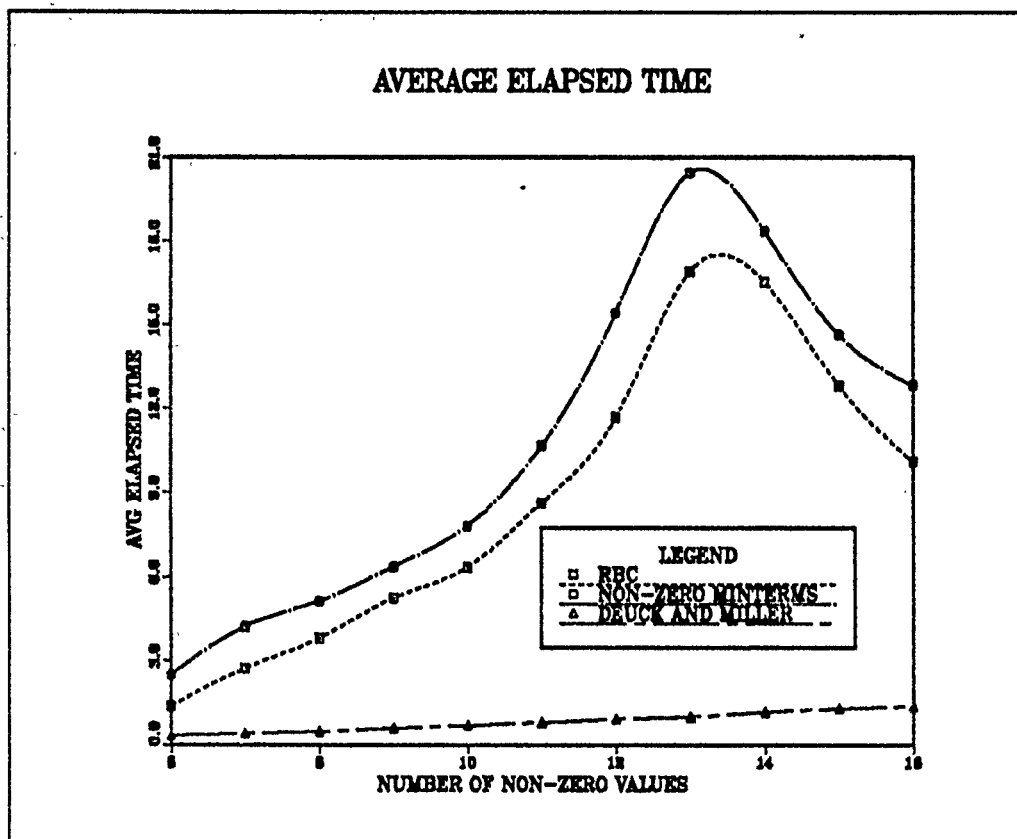


Figure 5. The average elapsed time per function versus the number of nonzero values for random functions

V. CONCLUSIONS

As with most engineering problems, there are certain trade-offs. A simplified MVL function that is minimal or nearly so can be found with A*. But it uses significant computer time to get it. If time is more important than a minimal solution, then a straight-line heuristic such as Dueck and Miller is better.

A* using the number of non-zero minterms is not effective. It takes longer to reach a solution and that solution is usually not as good as Dueck and Miller. Using the number of implicants multiplied by the number of variables is an effective cost function. It forces sufficient backtracking to minimize the number of implicants in the solution. This is true even for strong evaluation functions. Dueck and Miller and other straight-line heuristics are very fast, much faster than searching methods.

A* using cumulative RBC is an effective algorithm for finding minimal or near minimal solution for MVL functions. It provides a smaller average number of implicants needed to express the function than Dueck and Miller. It is much faster than an exhaustive search for absolute minimization. Once again RBC proves to be an excellent method to evaluate implicants to be included in a solution. In this case, RBC was modified to provide it with memory of the quality of previous implicants. An important question is how well A* does in finding absolute minimums. Unfortunately, time and resources were not available to do exhaustive searches in sufficient quantities for this research. It is a question which needs to be answered in further research. The data in Tirumalai and Butler [Ref. 7, pp. 235-241] seems to indicate that even small improvements in the average number of implicants needed to cover yields significant improvement in numbers of functions where the absolute minimal realization is achieved. If this is true, then A* using cumulative RBC may do very well in achieving absolute minimization. The additional cost to do an exhaustive search or use an alternative heuristic is not justified.

Both the version using the number of non-zero minterms and the version using cumulative RBC needed weighting factors. Use of a weighting factor gives the user control over the amount of backtracking (and thus confidence in a minimal or near-minimal solution) and the amount of time needed to solve. Further research is needed to determine the best weighting factors to use.

APPENDIX A. ALGORITHM DESCRIPTION

This Appendix provides a formal description of the of the searching algorithm as used in the program *astar*. It also describes the structure of a node as used in the algorithm.

algorithm

input: A file containing one or more MVL functions.

output: One or more simplified expressions of the functions
from the input.

begin

copy *f_original* into *f_work*

while minterms remain in *f_work* repeat

find most isolated minterm (*mim*) of *f_work*

while implicants remain for this *mim* repeat

generate an implicant

generate a new node from the implicant

place the node on the agenda

end while

current_node is the node from the agenda with the lowest sum

remove *current_node* from agenda

f_work is *f_original* minus the implicants used to get to
current_node

end while

copy implicants used to get to *current_node* into *f_final*

end

a node consists of:

- list of all implicants used to reach this implicant
- plus this implicant
- value of cost function for this implicant
- value of evaluation function for this implicant
- sum of cost and evaluation functions.

APPENDIX B. PROGRAM LISTING

```
/* Source:      astar.c
   Revision:    1.1
   Date:       27 FEB 90
   Author:     Watts
   */

/* This version of Astar uses the sum of all relative break counts
   of all implicants used to reach the current implicant, plus the
   rbc of the current implicant as the evaluation function.

   The cost function is the number of implicants multiplied by the
   number of variables.

   This version of Astar is compatible with the following options
   from HAMLET

       -a      provide output formatted for analysis by mvla
       -b      do not print the version banner
       -e      print original expression
       -i      print every implicant chosen
       -q      quiet, don't print anything
       -m      print Karnaugh map of each input expression
       -M      do not execute the algorithm, just print the
               Karnaugh map
       -s      print statistics on this algorithm
       -Sc     search for a solution until completion
       -oFile  output formatted solutions for mvll to "File"
       -xfile  output source expressions built to "File"
               instead of x.mvl
   */

#include "defs.h"

/* Structures and Global Variables needed by Astar */

typedef struct node_struct {
    int cost,
        evaluation,
        criterion;
    Expression *path;
    struct node_struct *next;
} Node;

Node *header,
    *current_node;

AStar()
/* -----
   function
       -Perform an A* search on the input expression to find a solution.
   algorithm
```

```

Start with a working copy E_work of the original function E_orig;
Initialize a final function E_final, the agenda, and current_node;
While (there are still minterms to pick) {
    Find the most isolated minterm;
    Find all implicants of the minterm;
    Generate Nodes for each implicant {
        Cost;
        Evaluation;
        Criterion;
        Path
    };
    Place Nodes in the agenda;
    Choose the Node in the agenda with the smallest criterion;
    current_node = chosen node;
    Subtract implicant of current_node from E_work;
    remove current_node from agenda;
}
globals
    E_orig
side effects
    STAT
    E_work
    E_final
called by
    main()
calls
    agenda()                from astar.c
    alloc_node()            from astar.c
    dealloc_expr()          from alloc.c of HAMLET
    dealloc_node()          from astar.c
    dup_expr()              from alloc.c of HAMLET
    free_mem()              from astar.c
    gen_bounds()            from common.c of HAMLET
    init_implicant()        from common.c of HAMLET
    mim()                   from dm.c of HAMLET
    next_implicant()        from common.c of HAMLET
    New_node()              from astar.c
    print_Node()            from astar.c
    print_source()          from main.c of HAMLET
    print_terms()          from main.c of HAMLET
    remove()                from astar.c
    search()                from astar.c
    subtract_path()         from astar.c
    valid_implicant         from dm.c of HAMLET
----- */
{
    int first = 0;
    int trace = 0;
    int i, better_found;
    double ratio;
    Coord *X;

    Node *next_node = NULL;
    Node *New_node(), *search();

    Bound *B;

```

```

    Implicant *I;

extern MVL_stats AS_stat;

Eval_value V;

/* Initialize all necessary values */
if (E_final.I != NULL)
    dealloc_expr(&E_final);
header = NULL;
current_node = NULL;

#ifdef KEEP_STATS
    STAT = &AS_stat;
#endif

/* Initialize the working expression to the original expression */
dup_expr(&E_work, &E_orig);

#ifdef ALEVEL_2
    if (opt_print_orig_expr)
        print_terms(&E_orig);
    if (opt_print_map) {
        printf("Orig map (AS): n");
        print_map(&E_work);
    }
#endif

resource_used(START);

/* While there are still minterms to pick repeat */
for (;;) {
    /* Find the most isolated minterm */
    if ((X = mim(&E_work)) == NULL)
        break;

    init_implicant(X);
    B = gen_bounds(X);

    /* While there are still implicants to pick repeat */
    while ((I = next_implicant(B)) != NULL) {

        /* If the original value of this minterm was the max value
           investigate for all possible values, not just its current
           value. */
        if (V.eval == (radix-1)) {
            for (I->coeff = X[nvar]; I->coeff < radix; (I->coeff)++) {
                if (valid_implicant(I)) {
                    next_node = New_node(I, next_node);
                    agenda(next_node);
                }
            }
        }
        /* Otherwise just generate a Node for the current value. */
        else {
            I->coeff = X[nvar];
            if (valid_implicant(I)) {

```

```

        next_node = New_node(I, next_node);
        agenda(next_node);
    }
}

/* Set the current_node to the best choice on the agenda */
if (current_node != NULL)
    dealloc_node(current_node);
current_node = search(header);
/* Remove the current node from the agenda */
remove(current_node);
/* E_work now consists of E_orig minus the path of implicants
   needed to get to current_node */
subtract_path(current_node);

#   ifdef ALEVEL_2
    if (opt_print_each_impl) {
        printf("ncurrent node: n");
        print_Node(current_node);
    }
#   endif

}
resource_used(STOP);

/* If the solution found is worse than the original
   use the original unless the search to completion
   option is chosen. */
if (current_node->path->nterm > E_orig.nterm) {
    better_found = 0;
    if (!opt_S_to_coverage)
        dup_expr(&E_final, &E_orig);
}
/* Otherwise us the solution found */
else {
    better_found = 1;
    dup_expr(&E_final, current_node->path);
}

/* Depending on the options desired, prepare the output
   and write it. */
if (xf_name[0])
    print_source(&E_final, "AStar", E_final.nterm);

#   ifdef ALEVEL_1
    ratio = ((double)E_final.nterm/((double)E_orig.nterm));

    /* Get the output needed for analysis by mvla */
    if (opt_mvla && (is_redir || !opt_be_quiet)) {
        printf("%-4d AS:  %4d/%-4d %4.2f %6d:%3.3ld n",
            expr_seq, E_final.nterm, E_orig.nterm, ratio,
            secs_used(), tsecs_used());
    }
    else if (!opt_be_quiet) {
        printf("Case: %-5d  User: %d n", expr_seq, E_orig.nterm);
        printf("Heur: AStar  Perf: ");
        if (better_found)

```

```

        printf("%d n n",E_final.nterm);
    else
        printf("no better n n");
    fflush(stdout);
}
# endif

/* Free all the memory used in the agenda */
free_mem(header);
}

Node *alloc_node(p)
Node *p;
/* -----
function
    -allocates memory for a node of the agenda
returns
    -the address of the node
called by
    New_node()
calls
    alloc_expr()
----- */

{
    Expression *alloc_expr();

    if ((p=(Node *)malloc(sizeof(Node))) == NULL)
        fatal("alloc_node(): out of memory n");
    p->path = NULL;
    p->next = NULL;
    p->path = alloc_expr();

    return(p);
}

Node *New_node(I,p)
Implicant *I;
Node *p;
/* -----
function
    -Generates a node for the implicant I
returns
    -the address of the node
called by
    -AStar()
calls
    alloc_implicant()      from alloc.c of HAMLET
    alloc_node()
    Cost()
    dup_expr()             from alloc.c of HAMLET
    Eval()
----- */

{
    int i, term;

```

```

    p = alloc_node(p);

    p->cost = Cost(I);
    p->evaluation = Eval(I);
    p->criterion = p->cost + p->evaluation;
    p->next = NULL;

    p->path->radix = E_work.radix;
    p->path->nvar = E_work.nvar;

    if (current_node == NULL) {
        p->path->nterm = 1;
        p->path->I = alloc_implicant(p->path->I, I->coeff, 1);

        for (i=0; i < nvar; i++) {
            p->path->I[0].B[i].lower = I->B[i].lower;
            p->path->I[0].B[i].upper = I->B[i].upper;
        }

    } else {
        dup_expr(p->path, current_node->path);
        term = p->path->nterm;
        p->path->nterm++;
        p->path->I = alloc_implicant(p->path->I, I->coeff, p->path->nterm);
        for (i=0; i < nvar; i++) {
            p->path->I[term].B[i].lower = I->B[i].lower;
            p->path->I[term].B[i].upper = I->B[i].upper;
        }
    }

    return(p);
}

Expression *alloc_expr()
/* -----
   function
   -allocates memory for an expression
   returns
   -the address of the expression
   called by
   alloc_node()
   ----- */

{
    Expression *p;

    if ((p = (Expression *)malloc(sizeof(Expression))) == NULL)
        fatal("alloc_node(): out of memory n");
    p->I = NULL;
    p->nvar = 0;
    p->radix = 0;

    return(p);
}

```

```

agenda(p)
Node *p;
/* -----
function
    -places a node pointed to by p on the agenda. The agenda is
    implemented as a linked list.
called by
    AStar()
----- */

{
    /* If the agenda is empty, just put the new node in it */
    if (header == NULL)
        header = p;

    /* Otherwise, put the new node at the first slot and have it
    point to the old first node. */
    else {
        p->next = header;
        header = p;
    }
}

Node *search(p)
Node *p;
/* -----
function
    -finds the best node on the agenda and sets current_node to that
    node
returns
    -the address of the best node
called by
    astar.c
----- */

{
    Node *best;

    /* Initialize the best choice to the first node on the agenda */
    best = p;

    #ifdef KEEP_STATS
    STAT->calls_pick_implicant++;
    #endif
    /* Repeat until we reach the end of the agenda */
    for (; p != NULL;) {
        /* If this node is better than the node in best
        this node becomes best. */
        if (p->criterion < best->criterion)
            best = p;
        /* Go to next node */
        p = p->next;
    }
}

```

```

    return(best);
}

dealloc_node(p)
Node *p;
/* -----
   function
   -frees memory of a node
   called by
   astar.c
   calls
   dealloc_expr()    from alloc.c of HAMLET
   ----- */

{
    dealloc_expr(p->path);
    free(p);
}

remove(p)
Node *p;
/* -----
   function
   -removes a node from the agenda
   called by
   astar.c
   ----- */

{
    Node *np;

    /* If the node is the first node, change the header */
    if (p == header)
        header = header->next;

    /* Else traverse the agenda until we find the node we wish to
       remove. Fix the pointer of the previous node to point to the
       next node. */
    else {
        np = header;
        for (; np != NULL;) {
            if (np->next == p) {
                np->next = np->next->next;
                break;
            }
            else np = np->next;
        }
    }
}

subtract_path(p)
Node *p;
/* -----
   function
   -finds the remainder of the function left to solve by taking

```



```

    E_orig and subtracting the path of the current_node.
called by
    astar.c
calls
    alloc_implicant()      from alloc.c of HAMLET
    dealloc_expr()        from alloc.c of HAMLET
    dup_expr()            from alloc.c of HAMLET
side effects
    E_work

```

```

----- */

{
    int i, j, term;

    dealloc_expr(&E_work);
    dup_expr(&E_work, &E_orig);

    for (i = 0; i < p->path->nterm; i++) {
        term = E_work.nterm;
        E_work.nterm++;
        E_work.I
            = alloc_implicant(E_work.I, -(p->path->I[i].coeff), E_work.nterm);
        for (j = 0; j < nvar; j++) {
            E_work.I[term].B[j].lower
                = p->path->I[i].B[j].lower;
            E_work.I[term].B[j].upper
                = p->path->I[i].B[j].upper;
        }
    }
}

print_Node(p)
Node *p;
/* -----
   function
   -to print out the contents of a node pointed to by p
   called by
   astar.c
   calls
   print_terms()      from main.c of HAMLET
   ----- */
{
    printf("cost = %3d evaluation = %3d criterion = %3d n",
        p->cost, p->evaluation, p->criterion);
    print_terms(p->path);
}

int Cost(I)
Implicant *I;
/* -----
   function
   -Determines the cost of an implicant I
   returns
   -the cost of the implicant I
   called by

```

```

New_node()
----- */
{
    int i;

    /* The cost is the number of implicants times the number of
       variables times a weighting factor of 2 */
    i = (E_work.nterm - E_orig.nterm + 1) * nvar * 4 / 2;
    return(i);
}

int Eval(I)
Implicant *I;
/* -----
   function
   -Evaluates the Implicant I for closeness to final solution
   returns
   -an integer that represents how much closer we are to a
     solution. The smaller the number the closer we are.
   called by
     New_node()
   calls
     compute_rbc()      from dm.c of HAMLET
   ----- */
{
    short i;
    int j, k;

    /* The evaluation function is the relative break count (rbc)
       of the current implicant plus the cumulative rbc's of the
       implicants in the path to this implicant. */
    i = compute_rbc(I);
    if (current_node != NULL) {
        j = (current_node->evaluation);
        k = i + j;
        return(k);
    }
    else
        return(i);
}

free_mem(p)
Node *p;
/* -----
   -function:
   -frees all the memory for the next function
   called by
     astar()
   ----- */
{
    Node *hold;

    /* Traverse the agenda freeing all the pointers. */
    for (; p != NULL;) {
        hold = p->next;
        free(p->path->I->B);
    }
}

```

```
free(p->path->I);  
free(p->path);  
free(p);  
p = hold;
```

```
} }
```

LIST OF REFERENCES

1. Etiemble, D., and Israel, M., "Comparison of Binary and Multivalued ICs According to VLSI Criteria ", *Computer*, Vol. 21, April 1988, pp. 28-42
2. Kameyama, M., Kawahito, S., and Higuchi, T., "A Multiplier Chip with Multiple-Valued Bi-Directional Current-Mode Circuits," *Computer*, Vol 21, April 1988, pp. 43-56
3. Kerkhoff, H. G., and Butler, J. T., "A Module Compiler for High-Radix CCD PLA's," forthcoming in *International Journal of Electronics*, Vol. 66, 1989
4. Pomper, G. and Armstrong, J. A., "Representation of Multi-Valued Functions Using the Direct Cover Method," *IEEE Trans. on Comput.*, Vol. C-30, No. 9, Sept 1981, pp. 674-679
5. Besslich, P. W., "Heuristic Minimization of MVL Functions: A Direct Cover Approach," *IEEE Trans. on Comput.*, Vol. C-35, No. 2, Feb 1986, pp. 134-144
6. Dueck, G. W. and Miller, D. M., "A Direct Cover MVL Minimization Using the Truncated Sum," *Proc. of the 17th Inter. Symp. on Multi-Valued Logic*, May 1987, pp. 221-227
7. Tirumalai, P. P., and Butler, J. T., "Analysis of Minimization Heuristics for Multiple-Valued Programmable Logic Arrays," *Proceedings of the 1988 International Symposium on Multiple-Valued Logic*, May 1988, pp. 226-236
8. Yurchak, J. M., and Butler, J. T., "HAMLET-An Expression Compiler/Optimizer for the Implementation of Heuristics to Minimize Multiple-Valued Programmable Logic Arrays," *Proc. of the 20th Inter. Symp. on Multi-Valued Logic*, May 1990, pp. 144-152

9. Rowe, Neil C., *Artificial Intelligence Through PROLOG*, Prentice-Hall, Englewood Cliffs, New Jersey, 1988, pp. 191-262
10. Ko, Yong Ha, *Design of Multiple-Valued Programmable Logic Arrays*, Master's Thesis, Naval Postgraduate School, Monterey, California, December 1988
11. Miller, D. Michael, and Muzio, Jon C., "On the Minimization of Many-Valued Functions," *Proceedings of the 9th International Symposium on Multiple-Valued Logic*, May 1979, pp. 294-299

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5004	1
4.	Prof. J. T. Butler, Code EC/Bu Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5004	10
5.	CDR J. M. Yurchak, Code 52Yu Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5000	2
6.	CPT A. W. Watts 2713 Fairbrook St. Irving, TX 75062	5
7.	LT David York Navy Underwater System Center Newport Laboratory Newport, Rhode Island 02841	1
8.	Dr. George Abraham, Code 1005 Office of Research and Technology, Applied Physics Naval Research Laboratories 4555 Overlook Ave. N. W. Washington, DC 20375	1
9.	Dr. Robert Williams Naval Air Development, Code 5005 Warminster, PA 18974-5000	1
10.	Prof. Chyan Yang, Code EC/Ya Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5004	1