1993-12

Multiple-valued programmable logic array minimization by solution space search

Wendt, Charles G.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

MULTIPLE-VALUED PROGRAMMABLE
LOGIC ARRAY MINIMIZATION BY
SOLUTION SPACE SEARCH

by

Charles G. Wendt

December, 1993

Thesis Advisor: Jon T. Butler

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*

2. REPORT DATE
December 1993

3. REPORT TYPE AND DATES COVERED
Master's Thesis

4. TITLE AND SUBTITLE
MULTIPLE-VALUED PROGRAMMABLE LOGIC ARRAY MINIMIZATION BY SOLUTION SPACE SEARCH

5. FUNDING NUMBERS

6. AUTHOR(S)
WENDT, Charles G.

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey, CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
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 US Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A minimal realization of a multiple-valued programmable logic array can only be achieved by exhaustive search. However, an exhaustive search is unrealistic even with the high speed CPU's in use today. Heuristic algorithms have been developed that provide near-minimal solutions, using significantly less CPU time. This thesis investigates a new type of heuristic that uses implicant operations (combine, reshape, and cut) to move through the solution space. The choice of move is dynamically controlled by feedback from a queue of previous moves, called a TABU queue. This new heuristic performs better than existing heuristics, in certain situations, but requires more CPU time than direct cover methods.

In addition, this heuristic provides a unique capability to fix the move acceptance probabilities associated with the basic implicant operations. Fixing move acceptance probabilities allows a study of the solution space of multiple-valued logic functions under controlled conditions. For example, the results of a preliminary study into the solution space of a four-valued, three variable special function (SF) are presented. This suggests that the search space is not homogeneous; rather it suggests that the space is segmented with restrictive access between segments. The results of such studies will be a basis for improving the performance of current and future minimization heuristics.

14. SUBJECT TERMS
MVL (multiple-valued logic) minimization; PLA (programmable logic array); HAMLET; Solution Space Search

15. NUMBER OF PAGES
60

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std 239-18

Approved for public release; distribution is unlimited.

Multiple-Valued Programmable Logic Array Minimization By Solution Space Search

by

Charles G. Wendt
Lieutenant Commander, United States Navy
B.S., United States Naval Academy

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 1993

Author: [redacted]
Charles G. Wendt

Approved by: [redacted]
Jon T. Butler, Thesis Advisor

[redacted]
David Erickson, Second Reader

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

ABSTRACT

A minimal realization of a multiple-valued programmable logic array can only be achieved by exhaustive search. However, an exhaustive search is unrealistic even with the high speed CPU's in use today. Heuristic algorithms have been developed that provide near-minimal solutions, using significantly less CPU time. This thesis investigates a new type of heuristic that uses implicant operations (combine, reshape, and cut) to move through the solution space. The choice of move is dynamically controlled by feedback from a queue of previous moves, called a TABU queue. This new heuristic performs better than existing heuristics, in certain situations, but requires more CPU time than direct cover methods.

In addition, this heuristic provides a unique capability to fix the move acceptance probabilities associated with the basic implicant operations. Fixing move acceptance probabilities allows a study of the solution space of multiple-valued logic functions under controlled conditions. For example, the results of a preliminary study into the solution space of a four-valued, three variable special function (SF) are presented. This suggests that the search space is not homogeneous; rather it suggests that the space is segmented with restrictive access between segments. The results of such studies will be a basis for improving the performance of current and future minimization heuristics.

TABLE OF CONTENTS

ACKNOWLEDGMENT

I would like to express my sincere appreciation to the United States Navy for providing this exceptional educational opportunity. Special thanks go to Dr. Butler for his indispensable assistance. I would also like to thank my wife, Cathy, and my children, Hannah and Bobby, for their extreme patience and support.

I. INTRODUCTION

A. MOTIVATION

The recent progress in very large scale integration (VLSI) technology has made the manufacture of chips with millions of integrated circuits possible. However, with this progress have come two major problems, *interconnect* and *pinout*.

In binary VLSI design, interconnect wiring takes up about 70% of the chip area. In multiple-valued logic (MVL), there are usually more than two levels of logic. Therefore, with MVL, fewer digits are needed than with binary to convey the same information. With fewer digits, less area is required for interconnect and more area is available for logic gates.

However, there is the question of implementing a multiple-valued system. Recent applications of MVL in programmable logic arrays (PLA) implemented in charge-coupled devices (CCD) [Ref. 1, 2] and current mode CMOS [Ref. 3, 4] have adequately shown the feasibility of such a system. In fact, CCD circuits with 16 logic levels have been fabricated [Ref. 14].

B. BACKGROUND

Circuit design is a complex problem. One way to bring order to this problem is with a programmable logic array or PLA. PLA's are simple, regular circuit structures that are easily reproducible in VLSI. As the name implies, PLA's are *programmable*, which makes them flexible and useful. The physical size of a PLA is determined by the size of the function to be implemented. Therefore, the more product terms (sum-of-products form) in the function, the larger the PLA needed to implement the function. Therefore, to reduce the size of the PLA, we want to reduce the number of product terms.

MVL function minimization is a combinatorial optimization problem. Combinatorial optimization often falls into the class of problems known as NP-hard. In such problems, an exact solution is not likely to be achieved. Thus, we turn to heuristic techniques for finding an optimal solution to a given problem.

Several heuristic algorithms have been developed for use with computer aided design (CAD) and logic synthesis tools for multiple-valued PLA's [Refs. 5, 6, 7, 8, 9]. The majority of these heuristic algorithms are direct cover algorithms. Direct cover heuristics operate by selecting a minterm and an implicant that covers that minterm. This process is then repeated until the expression is covered.

C. SIMULATED ANNEALING

Simulated annealing (SA) is a heuristic technique that has only recently [Ref. 10] been applied to the problem of combinatorial optimization. SA is a general purpose algorithm. SA is modeled on the annealing process used on metals or glass, by which the material is first heated to a molten, high energy state and then slowly cooled to a low energy, crystalline state.

In MVL minimization by SA (MVLSA) [Ref. 5], the number of product terms in the solution is analogous with the energy state and cost increasing moves are accepted with probability $P(\Delta E) = e^{-\Delta E/kBT}$, where k_B is the Boltzmann constant (set to 1 for

MVLSA), T is temperature, and ΔE is the increase in cost for a given move. Initially, a high temperature is selected to "melt" the solution. Then, after a period of time (i.e., fixed number of moves attempted) for the solution to stabilize, the temperature is reduced and the process repeated until the solution is "frozen." The process of reducing the temperature is the *annealing schedule*. A slow reduction in temperature is critical to attaining a global minimum, but requires more time.

The primary advantage of MVLSA is its potential for finding a minimal solution every time and its ability to avoid purely local minima, a characteristic not shared by direct cover. MVLSA has shown improvement over direct cover heuristics [Refs. 6, 7, 8, 9]. However, there are some apparent inefficiencies in MVLSA. For example, MVLSA uses a fixed number of failed attempted moves as a stopping criterion [Ref. 11]. Additionally, MVLSA can visit the same solution many times. It is this time spent (re)visiting the same state(s) that contributes to this inefficiency.

Proposed here is a new heuristic algorithm, solution space search (SSS), that uses many basic operations of MVLSA, but incorporates a TABU Queue [Ref. 12] to improve the efficiency of the algorithm through dynamic adjustment of move acceptance probabilities. A review of the MVL PLA minimization problem follows. Then, implementation of the SSS heuristic is presented.

II. A NEW HEURISTIC

A. BACKGROUND

An r-valued function, $f(x_1, x_2, \ldots x_n)$, takes on a value $\{0, 1, 2, \ldots r\text{-}1\}$, for each assignment of values to the variables. The variables are also r-valued (i.e., $x_i \in \{0, 1, \ldots r\text{-}1\}$), where the radix, r, is the number of logic values in the function. The literal function is

$$^{ai}x_i^{bi} = \begin{cases} r-1 & \textit{if } ai \le x_i \le bi \\ 0 & \textit{otherwise} \end{cases}$$

and concatenation is the *min* function (i.e., xy = min(x, y)). Multiple-valued PLA's are implemented using the truncated sum of the sum-of-products form of the function. The truncated sum $A+B$ is the arithmetic sum of A and B, with A and B viewed as integers, unless that sum exceeds r-1, in which case, the arithmetic sum is truncated to r-1. For example, the *OR* function in binary is the truncated sum. A product term or implicant is expressed as

$$c\, {}^{a1}x_1^{b1}\, {}^{a2}x_2^{b2}\, {}^{a3}x_3^{b3} \ldots {}^{an}x_n^{bn}, \qquad (1)$$

where $c \in \{1, 2, \ldots r\text{-}1\}$, is a nonzero constant. In a PLA, circuit area is needed to realize a product term. Thus, we seek the sum-of-products expression for $f(x_1, x_2, \ldots x_n)$ that has the fewest product terms.

B. ALGORITHM OVERVIEW

A flowchart for the solution space search heuristic is shown in Figure 1. This heuristic employs the same basic operations used by MVLSA (combine, reshape, and cut), but incorporates a TABU queue [Ref. 12] to control when cost increasing move probability (cut_prob) is changed. The TABU queue as implemented provides "memory" of *previous moves* vice previous states visited. This modification was necessary to accommodate the data structure used by HAMLET. The entire expression is not stored after each move because of the large amount of memory necessary to save even a few moves. Instead, only the two implicants involved in a move are saved. By using a different data structure for HAMLET, or by putting the input expression in a different format, this compromise can be avoided.

C. HEURISTIC MECHANICS

While the total number of iterations is less than the maximum number of iterations, two implicants are randomly chosen from the working expression. This step is repeated until two adjacent implicants are found. These implicants are combined, if possible.

1. Combine

For simplicity, the flowchart treats all moves resulting in a reduction in the number of implicants (i.e., a decrease in the cost function) as a combine operation.

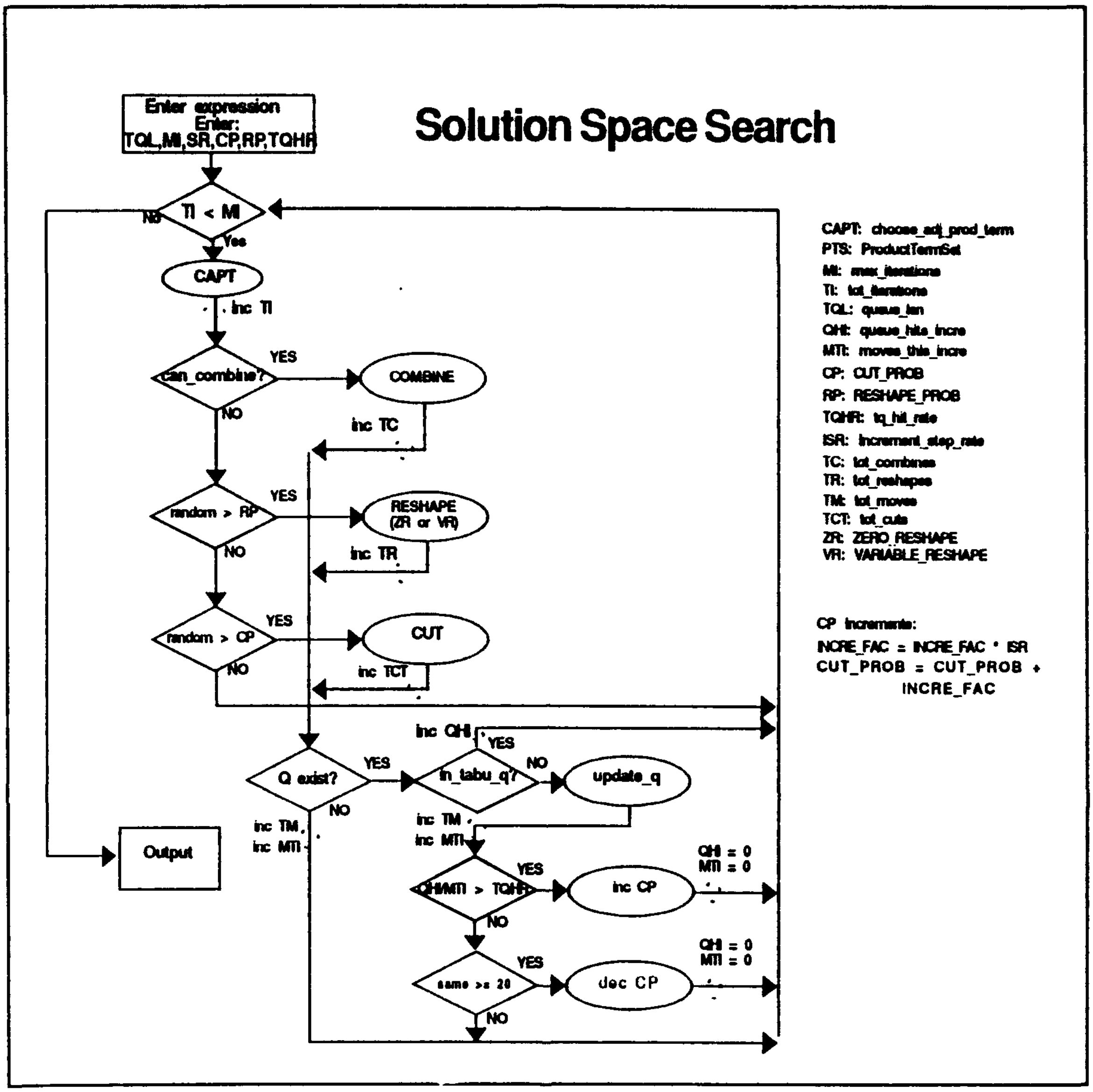

Figure 1. Solution Space Search flowchart

There are three types of moves that accomplish this; combine, bounds identical, and absorb. The combine move occurs if both of the following conditions are satisfied:

- the coefficients of the two implicants are identical

- for an n variable function, the bounds are identical in n-1 variables and the bounds *abut* in the remaining variable (i.e., for a four-valued function, ${}^{0}x_{1}{}^{1}$ and ${}^{2}x_{1}{}^{3}$)

The bounds identical move occurs if the bounds of the two implicants are the same for all variables. The absorb move occurs if one implicant is *saturated* (i.e., the coefficient $= r$-1) and the bounds of the other implicant are a subset of the bounds of the first.

If any one of these three moves can be made, then that move is made. When none of these moves is possible, a random number ω is generated and compared to the user-specified reshape_prob (reshape probability). If $\omega \geq$ reshape_prob, then a reshape move is performed.

2. Reshape

The reshape move performed is one of two possible types; zero-cost reshape or variable-cost reshape. The user selects the reshape move type at the time the heuristic is started. The selected reshape move type is then used exclusively during the program execution. As the name implies, the zero-cost reshape move produces two implicants, resulting in no net change in the number of implicants in the function (i.e., no change in the cost function). In contrast, the variable-cost reshape move may produce two or more implicants. The number of implicants produced is a function of the coefficient and bounds of the two input implicants and the number of variables in the function.

If $\omega <$ reshape_prob, a new random number η is generated and compared to the user-specified cut_prob (cut probability). If $\eta \geq$ cut_prob, a cut is performed.

3. Cut

The cut move randomly selects one of the two implicants. The selected implicant is then randomly cut in one of two ways; a coefficient cut or a bounds cut. A coefficient cut is a simple random cut of the coefficient. Note that a *saturated* coefficient cut provides the maximum number of ways of dividing the implicant because of the many ways the truncated sum can form. A bounds cut randomly selects one of the n variables and performs a cut. All variables have equal probability of being selected. If η < cut_prob, the heuristic returns to the start and begins another iteration.

4. TABU queue

After a combine, reshape, or cut is performed, the heuristic checks for the existence of the TABU queue. The user can select whether or not the TABU queue is used. Selecting zero (0) for the TABU queue length parameter overrides the control function provided by the TABU queue. When the control function is overridden, the heuristic runs are conducted with FIXED reshape and cut probabilities. Use of this feature provides a unique capability for exploring the solution space of a given function.

For user-specified TABU queue lengths other than zero, the heuristic searches the TABU queue for the implicant pair used in making the just completed move. If the implicant pair is found in the TABU queue, a counter is incremented (queue_hits_incre), the algorithm returns to the start of the loop and another iteration is performed. If the implicant pair is not found in the TABU queue, then the TABU queue is updated. In this case, the current implicant pair is placed at the beginning of the queue and the implicant pair at the end of the queue is removed (a first-in, first-out operation). At this point, a move is deemed to have been made, in which case two counters are incremented, a tot_moves (total moves) counter and a moves_this_incre (moves made this increment) counter. Moves_this_incre is used to determine when the cut probability is to be increased. Note that if the TABU queue does not exist (i.e., the user specified a zero length TABU queue), a move would be deemed made after the combine, reshape, or cut operation is completed.

5. Probability Control

Control is provided by changing the cut probability in response to (1) feedback from the TABU queue as moves are rejected and (2) when the solution shows signs of being trapped in a local minimum (i.e., number of terms in the function remains constant as moves are made). Increases in the cut probability are performed to drive the solution to a minimal state. Then, to prevent the solution from being trapped in a local minimum, decreases in cut probability are performed.

a. Increases in cut probability.

Increases in cut probability are performed by comparing the ratio, queue_hits_incre (tabu queue hits this increment) to moves_this_incre (moves made this increment), to the user-specified TABU queue hit rate (tq_hit_rate). If the former is greater than latter, the cut probability is increased (i.e., the probability of a cut occurring is decreased). Additionally, both the TABU queue hits this increment (queue_hits_incre) and moves made this increment (moves_this_incre) counters are reset to zero. The ratio of TABU queue hits this increment to moves made this increment was chosen based on the following reasoning. After a function reaches the "equilibrium" number of terms for a given reshape probability and cut probability, there will be more moves that result in a TABU queue hit (i.e., the two implicants were used in a move recently) than do not. Thus, as total moves in a given increment increase, the ratio of TABU queue hits to total moves this increment increases.

A means is needed to determine the size of the incremental increases in the cut probability. We do this by adding to the current cut probability a cut probability increment. The first increment is Ca, the next is Ca^2, etc. , where C is a constant called the increment factor (incre_fac) and a is the user-specified step_rate. The resulting current cut probability approaches 1.0 as time increases. Thus,

$$cut\_prob + C\sum_{i=1}^{\infty} a^{i} = 1.0. \tag{2}$$

Recall [Ref. 13] that

$$\sum_{n=0}^{\infty} a^{n} = \frac{1}{1-a},\ for\ a<1. \tag{3}$$

The same expression starting from $n=1$ is

$$\sum_{n=1}^{\infty} a^{n} = \frac{1}{1-a} - 1 \ or \ \frac{a}{1-a}, \tag{4}$$

and, substituting Equation (4) into Equation (2),

$$cut\_prob + \frac{Ca}{1-a} = 1.0. \tag{5}$$

Finally, we solve for *C*, the incre_fac, which yields

$$C = (1.0 - cut\_prob)\frac{(1-a)}{a}. \tag{6}$$

Each time the cut probability is to be increased, the increment factor (incre_fac) is multiplied by the user-specified step rate (step_rate). The result of the operation is the new increment factor. This new increment factor is added to the cut probability and saved for use in calculating the next cut probability increment. Each successive increment factor is smaller than the previous one. Thus, the cut probability increases by a smaller and smaller amount with each successive increment (approaching 1.0 in the limit)(i.e., no cuts performed).

b. Decreases in cut probability

As the cut probability approaches 1.0, fewer cuts occur. When very few cuts occur, the number of combinable terms is quickly exhausted. At this point, only zero cost reshape moves occur, and the total number of terms in the function remains constant (i.e., no change in the total cost). This is a local minimum.

To escape the local minimum, the cut probability must be decreased to allow cuts to occur (i.e., cost increasing moves). In the solution space search method, the cut probability is decreased when the number of terms in the expression remains constant for 20 moves. This number was chosen high enough to prevent premature resetting of the cut probability, while low enough to minimize time spent in the local minimum. The size of the decrease is a fixed percentage of the difference between the user-specified cut probability and 1.0. Thus, the cut probability is decreased to a level slightly higher than the initial cut probability, where cuts again occur and the heuristic continues to move. The process repeats as often as the conditions dictate.

III. PARAMETER OPTIMIZATION

A. PARAMETERS

There are seven parameters that determine the performance of the algorithm: cut probability, reshape probability, TABU queue length, increment step rate, TABU queue hit rate, maximum iterations, and variable cost reshape.

Cut probability (opt_SSS_cut_prob) [0.0 $< x <$ 1.0] sets the level that a random number must exceed before a cut will be performed.

Reshape probability (opt_SSS_reshape_prob) [0.0 $< x <$ 1.0] sets the level that a random number must exceed before a reshape will be performed.

TABU queue length (opt_SSS_tabuq_len) [0 $\leq x \leq$ 10,000] sets the length of the TABU queue. When set to zero, the TABU queue is bypassed and the cut probability incrementing feature is disabled, thus providing a "fixed probability" analysis capability.

Increment step rate (opt_SSS_step_rate) [0.0 $\leq x \leq$ 1.0] determines the size of the cut probability increment.

TABU queue hit rate (opt_SSS_tq_hit_rate) [0.0 $\leq x \leq$ 1.0] sets the threshold level for incrementing the cut probability. The queue hit ratio (queue_hits_this_increment / moves_this_increment) must exceed this threshold level before a cut probability increment will occur.

Maximum iterations (opt_SSS_max_iterations) set the maximum number of iterations the algorithm will perform.

Variable cost reshape (opt_SSS_method) is the flag that signals the heuristic to use the variable-cost reshape move. The variable-cost reshape move allows for the formation of multiple implicants (i.e., more than two implicants). Zero-cost reshape, the default mode, allows only two implicants to be formed (i.e., net cost of zero). All data runs performed for analysis in this thesis used the zero-cost reshape move. Further research using the variable-cost reshape move is indicated.

B. DEFAULT SETTINGS

Table 1 contains the default settings for the solution space search algorithm. The following parameter settings were used for initial testing of the algorithm and determination of default settings:

- Cut probability (0.975, 0.99, 0.995, 0.999, 0.9995)
- Reshape probability (0.50, 0.75, 0.90, 0.99)
- TABU queue length (0, 100, 500, 1000, 10000)
- Increment step rate (0.15, 0.25, 0.50, 0.75, 0.90)
- TABU queue hit rate (0.01, 0.001, 0.0005, 0.0001, 0.00001)
- Maximum iterations (1,000,000; 5,000,000; 10,000,000; 15,000,000)

No attempt was made to test every possible combination of parameter values because of the large number of such combinations. Instead, the following process was used. Three test functions were generated using *mvlt*, the test function generation module of HAMLET [Ref. 3]. These test functions were randomly generated as four-valued, five variable functions consisting of 25, 100, and 200 terms. A sensitivity analysis was conducted on each test function by choosing combinations of cut probability, reshape probability, increment step rate, and TABU queue hit rate covering the full range of parameter variability. This analysis was repeated for different TABU queue length settings, including the fixed probability setting. The results of the sensitivity analysis determined the default parameter values for the algorithm. It is important to note that the listed parameter values should only be considered a starting point, and not optimum values for every possible input function. The intent of the sensitivity analysis was to establish default settings which would yield reasonable results over the range of functions tested. Figure 2 is a example of data output produced by solution space search. The input file for this example was the aforementioned randomly generated,

TABLE 1. DEFAULT PARAMETER SETTINGS

PARAMETER	RANGE	DEFAULT SETTING
Cut probability	$0.0 < x < 1.0$	0.99
Reshape probability	$0.0 < x < 1.0$	0.50
TABU queue length	$0 \leq x \leq 10{,}000$	1000
Increment step rate	$0.0 \leq x \leq 1.0$	0.90
TABU queue hit rate	$0.0 \leq x \leq 1.0$	0.0001
Maximum iterations	---	12,000,000

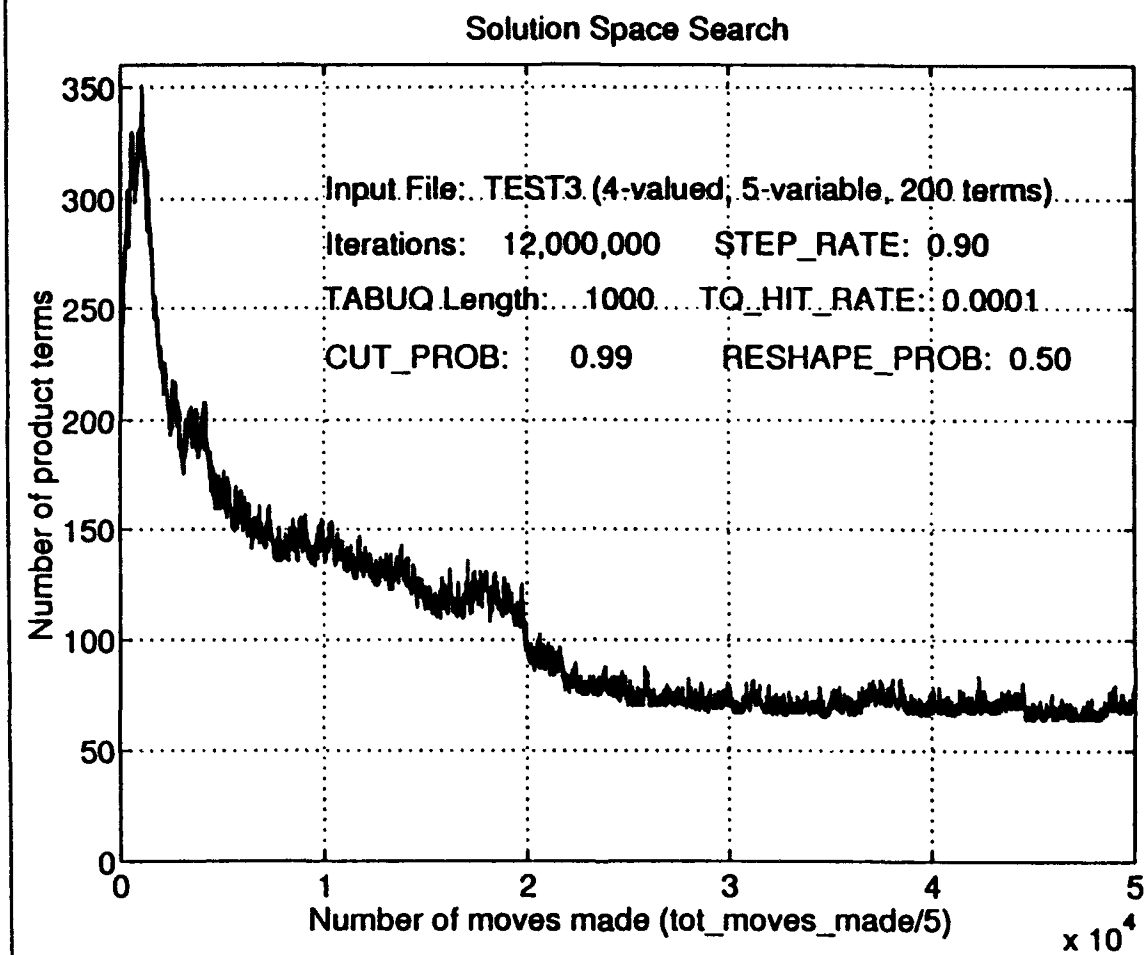

Figure 2. Sample plot of solution space search output data

four-valued, five variable, 200 term test file. For this example, default settings were used for all user-specified parameters of the solution space search algorithm.

IV. PERFORMANCE ANALYSIS

A. COMPARISON WITH OTHER MINIMIZATION HEURISTICS

To present a fair comparison of solution space search with the other minimization heuristics implemented in HAMLET [Ref. 3], nine test set ensembles of five test expressions were analyzed. All test sets were generated using the *mvlt* module of HAMLET. Each test set was created using a different random "seed" and consisted of five expressions. The test expressions were all four-valued, five variables. Solution space search used the zero cost reshape feature (the default) for these comparisons. All other heuristics were run using their **default parameter settings** and no attempt was made to "tune" any heuristic for this comparison.

A comparison of the performance of the selected heuristics is provided in Figure 3 . Solution space search produced better results than all other heuristics for the 50-, 75- and 100-term test sets. For test sets with 125-terms or greater, solution space search performed better than *Reshape* and *Cut & Combine* [Ref. 5], but not as good as the other heuristics.

CPU times for the test runs are shown in Figure 4. All test runs were performed on the same SunSPARC 10 workstation. Actual times on different operating systems will vary. However, the relative performance will be consistent and is the basis of this comparison. Solution space search required less time, on average, than *Reshape* and *Cut & Combine* but more time than the other (direct cover) heuristics. The test data shown

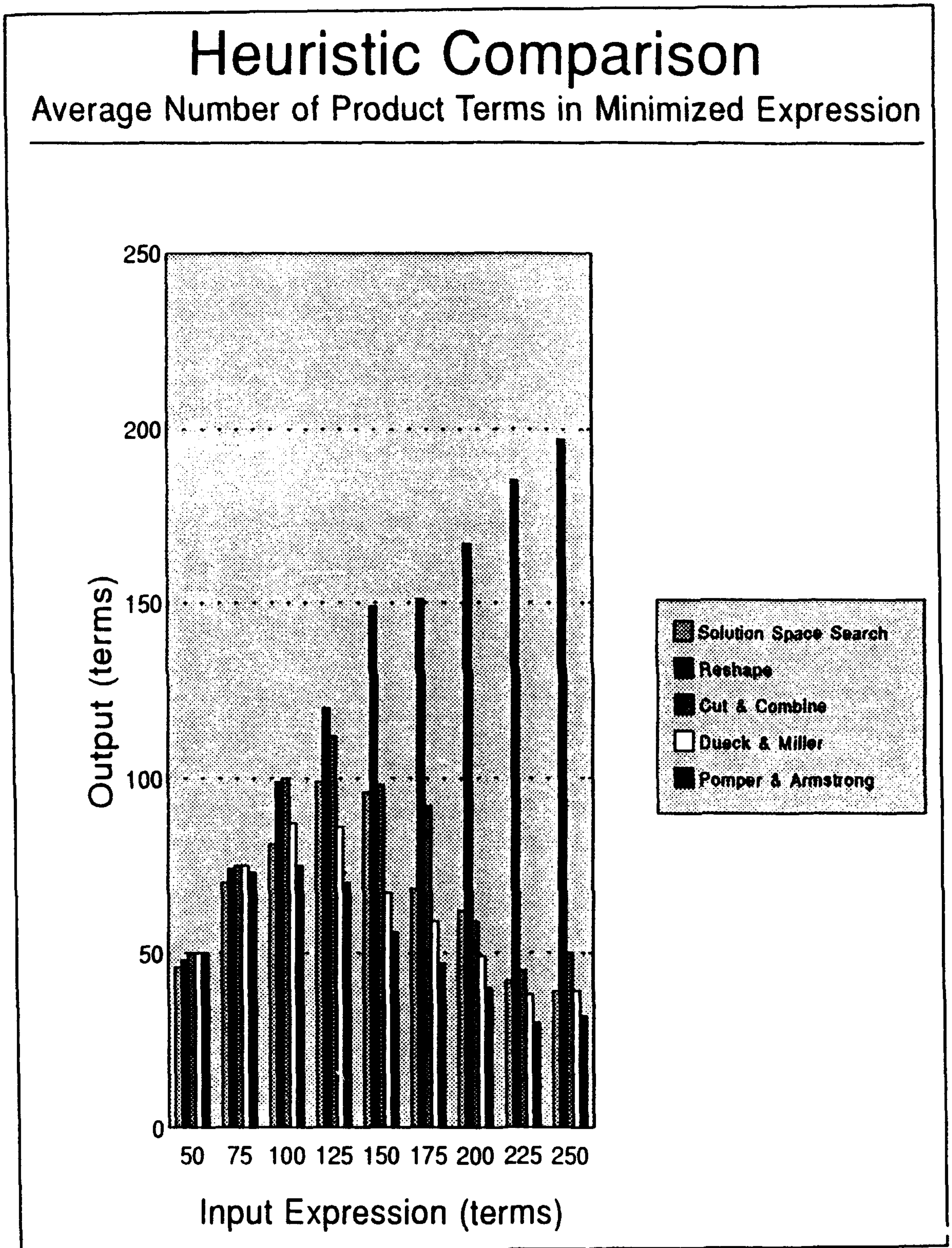

Figure 3. Heuristic comparison for test function ensembles

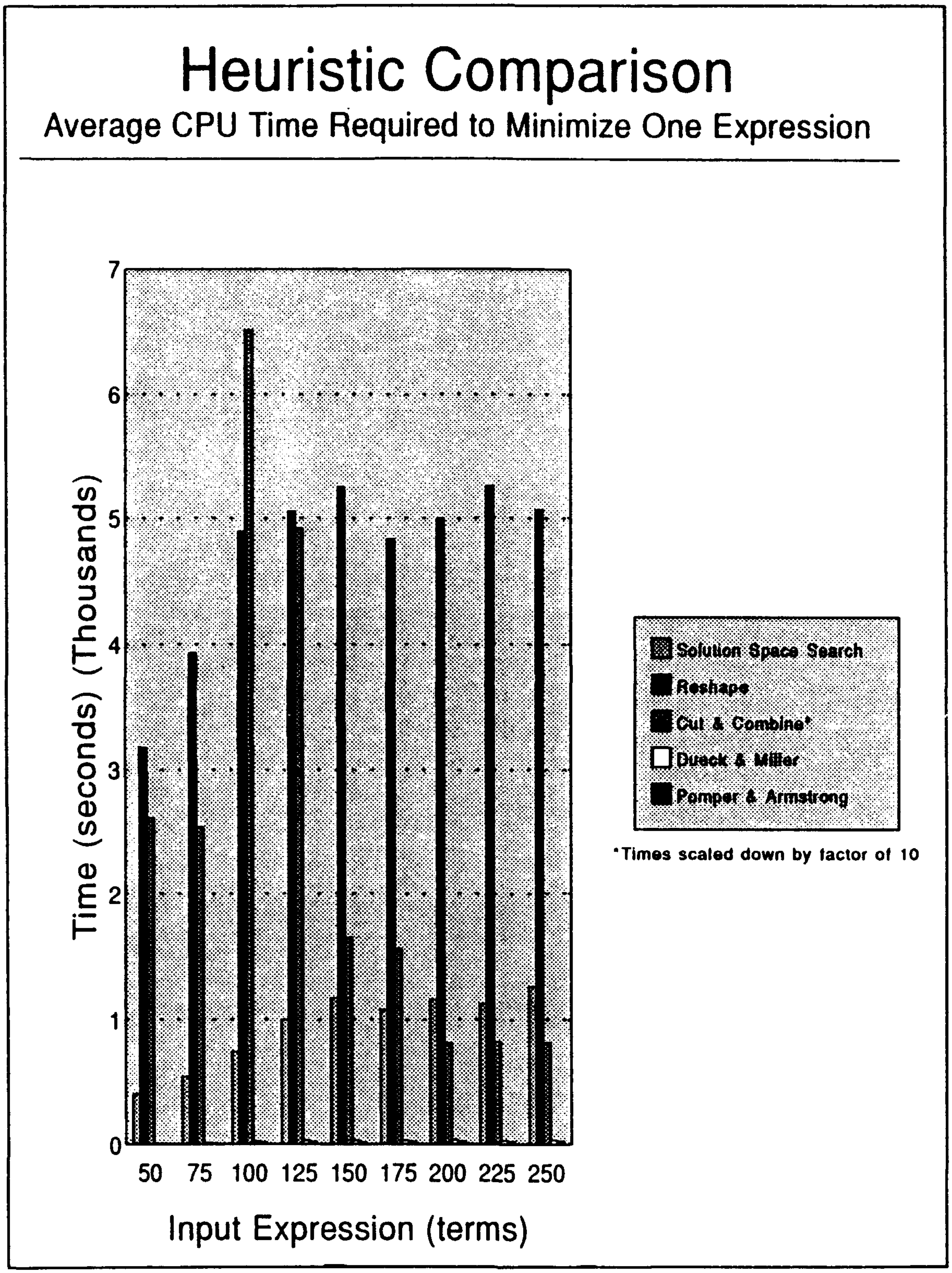

Figure 4. CPU time comparison for test function ensembles

provides an indication that the initial goal of improving on the speed of simulated annealing has been achieved.

B. SOLUTION SPACE EXPLORATION

Little is known about the solution space of MVL functions. Previous work has centered on the minimization problem directly, with no investigation into the nature of the MVL function solution space. However, we seek insights into the solution space of MVL functions to improve the performance of the heuristics. It was with this objective in mind that the fixed probability feature (i.e., setting TABUQ length to zero) of the solution space search heuristic was developed.

Time constraints precluded a full investigation into the nature of the MVL function solution space in this work. However, preliminary investigations have provided some valuable insight.

C. RESTRICTIONS TO MOVEMENT

Analysis of data from early testing of the solution space search algorithm led to an investigation into the exact nature of the moves performed in transitioning between a saturated expression and an oversummed expression. A saturated expression is one with one or more minterms having coefficients equal to three (i.e., r-1). An oversummed expression has one or more minterms whose coefficients are oversummed (i.e., sum to *greater* than r). It has been generally held that no restrictions exist to movement between saturated and oversummed expressions. To conduct this investigation, a four-valued, three variable special function (SF) was constructed as illustrated in Figure 5. The SF consists of implicants, with coefficient 1, placed along every edge. To study the onset of production of oversummed minterms, the cut probability and reshape probability were varied over their full range. To extract an oversummed minterm from a vertex, a sequence of special cuts must occur. The probability of these cuts occurring is a

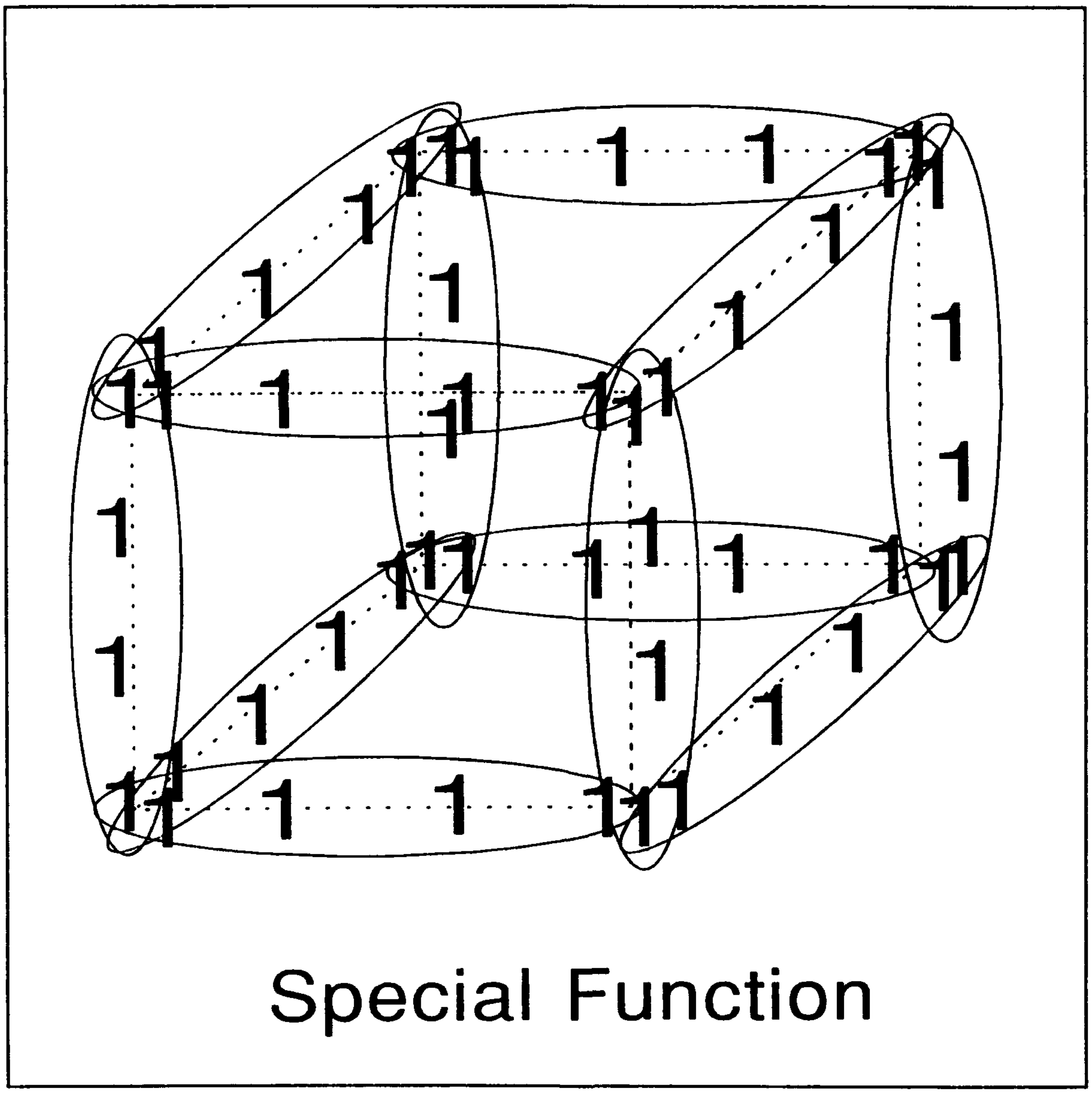

Figure 5. Special Function

relatively straightforward exercise. As the coefficients of all implicants in the SF equal 1, and a bounds cut is equally likely to occur in any variable, every product term can be cut in r-1 ways. The SF is four-valued (i.e., r=4), so there are three different cuts possible in each product term. Since the SF contains 12 product terms, there are a total of 36 (3 x 12) separate cuts possible. To extract a *specific* corner, the probability, ρ, is

$$\rho = \frac{1}{36*35*34}. \tag{6}$$

Since there are eight corners in the SF, the overall probability of extracting *a* corner is then,

$$\rho = \frac{8}{35*35*34} \approx \frac{1}{5000}. \tag{7}$$

Note that the probability is a function of r and n (number of variables) and decreases rapidly. For example, the probability for a four-valued, four-variable SF would be 1 in approximately 50,000. The significance of this finding is that this oversumming process is essential, in certain situations, to achieving a minimal solution. Thus, as r and n increase, it is less likely that a minimal solution will be achieved.

Figure 6 is a plot of the expression produced by solution space search showing the onset of saturation. Parameters used were: Maximum iterations = 100, cut probability = 0.10 (i.e, lots of cuts occurring), reshape probability = 0.90 (i.e., very few reshapes occurring), and zero TABU queue length (i.e., fixed probability mode). Figure 7 is

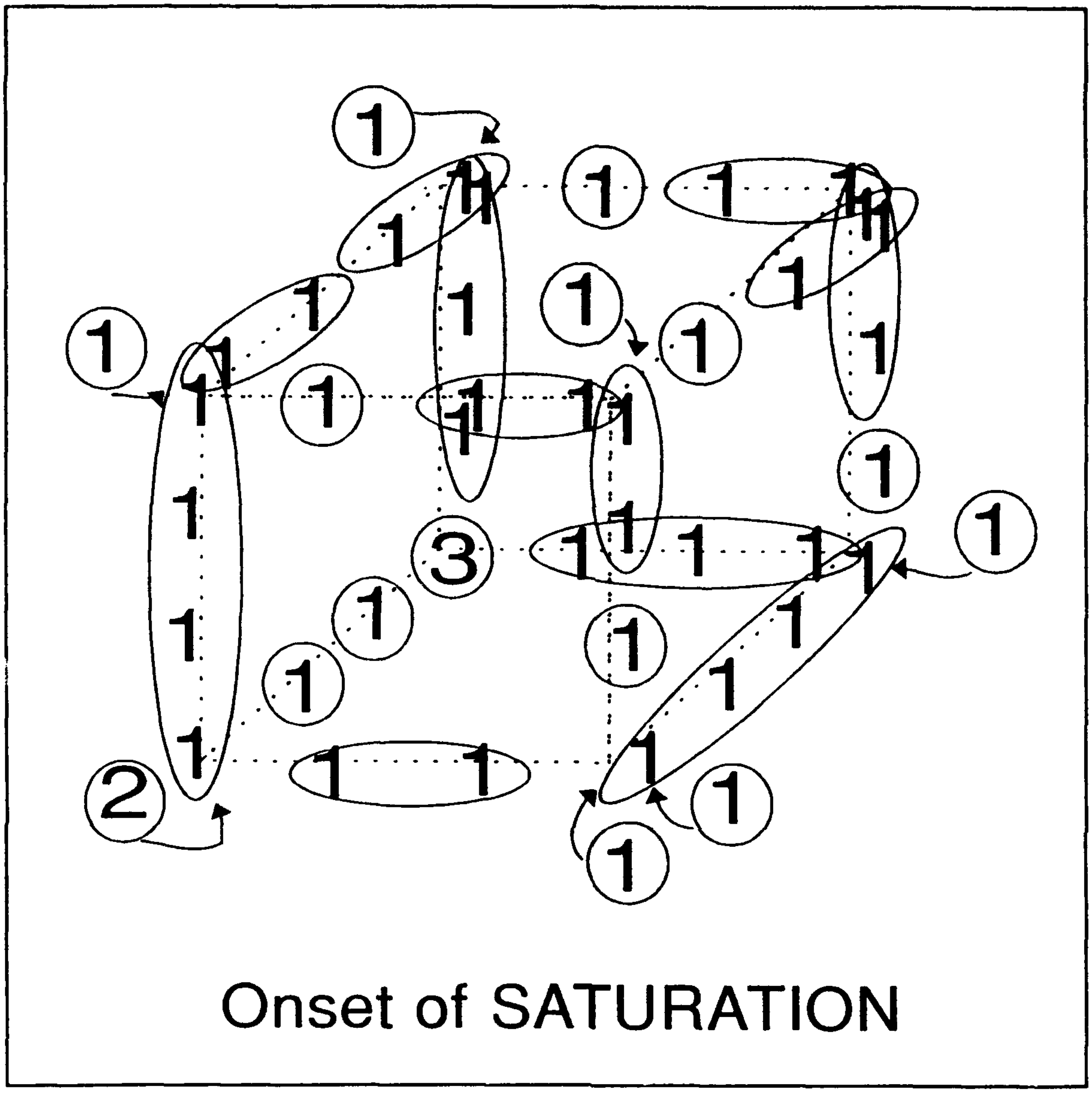

Figure 6. Plot of SF showing onset of saturation (cut_prob = 0.10, reshape_prob = 0.90, TABUQ_len = 0)

another plot showing saturation with four corners showing oversummed minterms. Parameter settings used: Maximum iterations = 1000, cut probability = 0.001, reshape probability = 0.999 (i.e., very few reshapes occurring), and zero TABU queue length. This plot clearly shows the oversumming which occurs and demonstrates the reformation of product terms after saturated minterm formation. It is important to add that a similar process must be repeated to transition back from a saturated solution to an unsaturated solution.

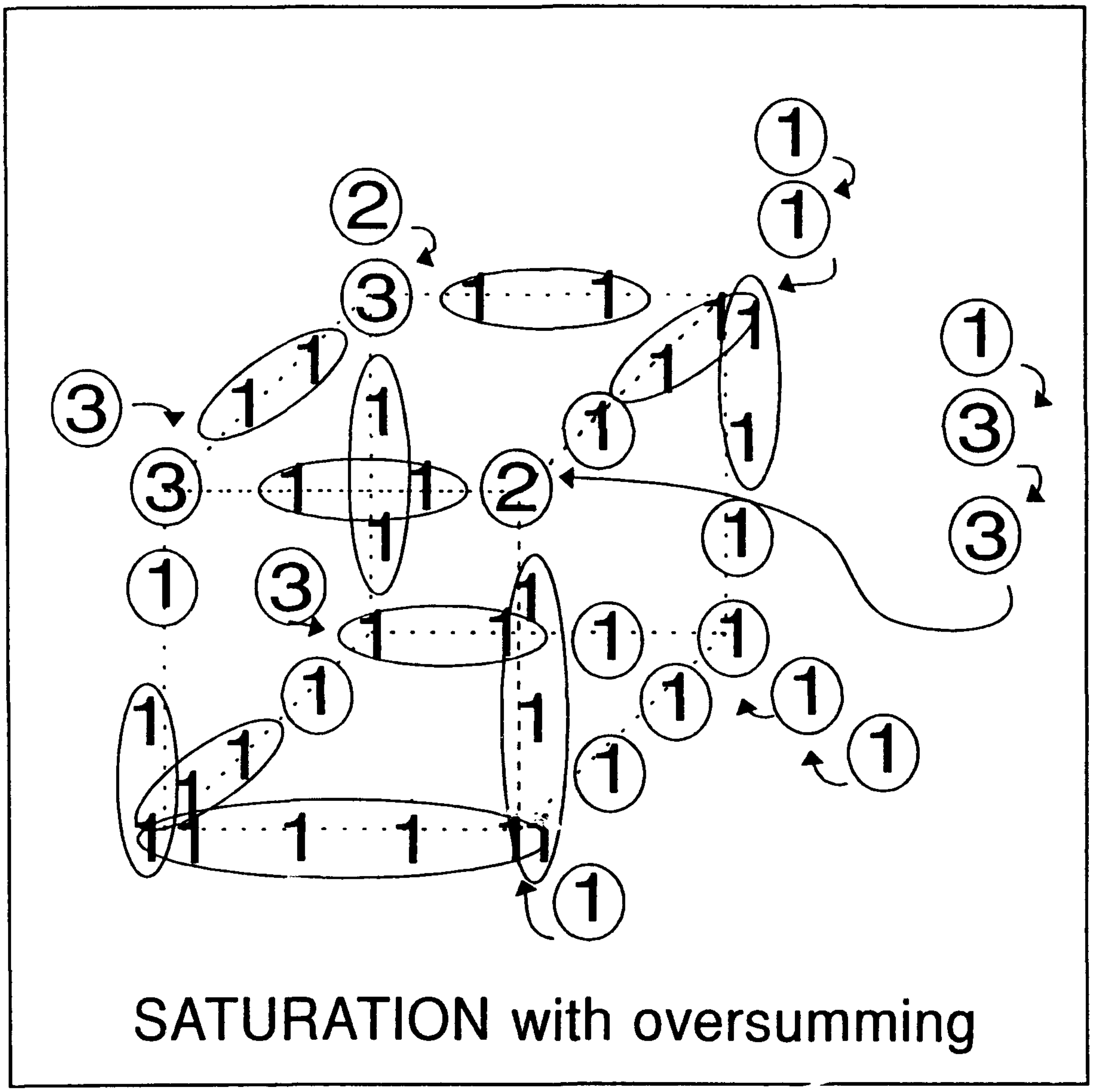

Figure 7. Plot of SF showing saturation with oversumming (cut_prob=0.001, reshape_prob=0.999, TABUQ_len = 0)

V. CONCLUSIONS

The solution space search heuristic provides a means to produce optimal or near-optimal MVL-PLA's. Analysis of test run results shows that the heuristic performs better, in certain circumstances, than both direct-cover and simulated annealing. The addition of a memory feature, while preventing repeated moves from the same state, introduces overhead proportional to the length of TABU queue selected. This may be the cause, at least with the present data structure, of some inefficiency. Employing a data structure that can be searched more efficiently will alleviate this problem. Possible schemes include using an array of minterms with pointers to adjacent minterms and product terms or a sorted linked list and hash table. Additionally, manual optimization of the C program code may yield further gains in efficiency.

Due to time considerations, no substantive testing was conducted using the variable-cost reshape mode of the heuristic. This mode may prove effective because of the unbalanced nature of the heuristic when using the variable-cost move. In particular, when near or in a local minimum, the variable-cost has the capability to provide more rapid movement than the zero-cost reshape move. On this basis, further research using the variable-cost move is recommended.

The relative merit of a zero-cost reshape move has not been investigated. MVLSA demonstrated improved performance using *reshape* over *cut & combine*, but functions that demonstrate the weakness of *reshape* can be found. Time considerations precluded a comparison of heuristic performance with and without a reshape move (i.e., setting reshape probability equal to 1.0). However, this comparison is recommended to ascertain the merits of the no-cost reshape move.

The fixed probability feature of the solution space search heuristic has provided some valuable insight into the solution space of MVL functions. Preliminary analysis of the special function demonstrated that restrictions to solution movement between unsaturated and saturated compositions. The existence of this restriction was previously unknown. Additionally, this restriction in movement becomes greater with increasing radix and number of variables in the expression. Because the ramifications of this discovery and others yet to be discovered, continued research using this mode of the heuristic is strongly recommended.

APPENDIX A - SOLUTION SPACE SEARCH CODE

1. Enclosed in this appendix is the C code for the Solution Space Search algorithm which runs as a module of HAMLET [Ref.10].

```
static char
rcsid[] = "$Id: sss.c,v 1.0 1993/07/06 10:17:40 wendt Exp
                                      wendt $";
/*************************************************************
      sss.c - This module implements the Solution Space Search
                  heuristic
*************************************************************/

/*************************************************************
 * Copyright (c) 1993 by Naval Postgraduate School
 *
 * Permission to use, copy, modify, and distribute this
 * software and its documentation for any purpose and
 * without fee is hereby granted, provided that the above
 * copyright notice appears in all copies and that both that
 * copyright notice and this permission notice appear in
 * supporting documentation, and that the name of Naval
 * Postgraduate School not be used in advertising or
 * publicity pertaining to distribution of the software
 * without specific, written prior permission.
 *
 * Naval Postgraduate School makes no representations about
 * the suitability of this software for any purpose. It is
 * provided "as is" without expressed or implied warranty.
 *
 * The sale of any product based wholely or in part upon the
 * technology provided by HAMLET is strictly forbidden
 * without specific, prior written permission from Naval
 * Postgraduate School. HAMLET technology includes, but is
 * not limited to, the source code, executable binary files
 * and expression specification language.
*************************************************************/

/* $Log: sss.c,v $
```

```
 * Revision 1.0  1993/07/06  10:17:40  wendt
 * "modifications to original code of yurchak, earle/
 *    dueck and others"
 */

#include "defs.h"
#define MAX_TABUQ        20000
/*   NOTE:  MAX length of TABUQ = MAX_TABUQ / 2    */

static int    better_found;

static Expression
      E_save = { NULL,0,0,0,MAX_INT },
      E_previous = { NULL,0,0,0,MAX_INT };

static struct sss_stats {
      int    sss_nterm;
      long   secs, tsecs;
      } *SSS_stats;

int    tot_cuts,
       tot_combines,
       tot_reshapes;

void  Soln_Space_Srch()
/*************************************************************
      :function:
            - Perform the Solution Space Search heuristic on the
            input expression
      :algorithm:
            Start with a working copy E_work of the original
            function E_orig;
                  While (total iterations less than max
                        iterations)
                   {
                        search the solution space
                  }
      :globals:
            E_orig
            opt_print_orig_expr
            opt_print_map
            opt_be_quiet
            sel_heur
```

```
        yyout
    :side_effects:
        STAT
        HEUR
        E_work
        E_final[]
    :called_by:
        main()
    :calls:
        dealloc_expr()
        dup_expr()
        print_terms()
        print_map()
        print_source()
*************************************************************/
{
    int     i, num_impl, first_prof, queue_exists,
                max_nterm = 0,
                tot_iterations = 0,
                tot_moves_made = 0,
                moves_this_incre = 0,
                move_attempts = 0,
                queue_hits_this_incre = 0;

    double      cut_prob = opt_SSS_cut_prob,
                incre_fac = (1.0-cut_prob)* ((1.0-opt_SSS_step_rate)
                                                    /opt_SSS_step_rate),
                ratio = 0.0;

        /* Incre_Fac is determined by taking the distance from
           Cut_Prob to 1.0 and multiplying by the ratio of
           1-step_rate/step_rate
        */

    Implicant  TQ[MAX_TABUQ];

    if (E_final[SOLN_SPACE_SRCH].I != NULL)
        dealloc_expr(&E_final[SOLN_SPACE_SRCH]);

#   ifdef KEEP_STATS
    STAT = &DM_stat;
#   endif
```

```
        HEUR = SOLN_SPACE_SRCH;
        dup_expr(&E_work,&E_orig);
        E_final[HEUR].nterm = 0;
        E_final[HEUR].radix = E_orig.radix;
        E_final[HEUR].nvar = E_orig.nvar;
        E_final[HEUR].I = NULL;

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

        better_found = opt_S_to_coverage;
        num_impl = E_orig.nterm;

        dup_expr(&(E_final[SOLN_SPACE_SRCH]),&E_orig);

        resource_used(START);

        if (SSS_stats == NULL) {

                SSS_stats = (struct sss_stats *)malloc(
                        (opt_SSS_max_iterations/10) * sizeof(struct
                        sss_stats));
                if (SSS_stats == NULL)
                        fatal("Soln_Space_Srch(): Out of memory
                                (SSS_stats[])");
        }

        dup_expr(&E_save,&E_orig);
        dup_expr(&E_previous,&E_work);

        first_prof = 1;

        queue_exists = build_tabu_q(&TQ[0]);

        SSS_stats[tot_moves_made].sss_nterm = E_work.nterm;

        while (tot_iterations < opt_SSS_max_iterations) {
```

```
int    I1_ndx, I2_ndx;

if (!choose_adjacent_pair(&E_work,&I1_ndx,&I2_ndx))
      goto done;

tot_iterations++;

if (combo = can_combine(&(E_work.I[I1_ndx]),
                                     &(E_work.I[I2_ndx])))
{
      if (combo == CAN_COMBINE) {
            sss_combine(&E_previous,&E_work.I1_ndx.I2_ndx);
      }
      else if (combo == BOUNDS_IDENT) {
            dup_expr(&E_previous,&E_work);
            csum = E_work.I[I1_ndx].coeff +
                                          E_work.I[I2_ndx].coeff;
            E_work.I[I1_ndx].coeff = min(radix-1, csum);
            E_work.nterm--;
            if (I2_ndx < E_work.nterm)
                  copy_impl(&(E_work.I[I2_ndx]),
                              &(E_work.I[E_work.nterm]));
      }
      tot_combines++;
}
else if (can_absorb(&(E_work.I[I1_ndx]),
                        &(E_work.I[I2_ndx])))
{
      dup_expr(&E_previous,&E_work);
      E_work.nterm--;
      if (I2_ndx < E_work.nterm)
            copy_impl(&(E_work.I[I2_ndx]),
                              &(E_work.I[E_work.nterm]));
      tot_combines++;
}
else if (can_absorb(&(E_work.I[I2_ndx]),
                                     &(E_work.I[I1_ndx])))
{
      dup_expr(&E_previous,&E_work);
      E_work.nterm--;
      if (I1_ndx < E_work.nterm)
            copy_impl(&(E_work.I[I1_ndx]),
                                    &(E_work.I[E_work.nterm]));
```

```
        tot_combines++;
}
else if (
        (((float)random()/RAND_MAX)>opt_SSS_reshape_prob)
                && (opt_SSS_method == SSS_ZERO_RESHAPE)
) {
        if (reshape_cost(&(E_work.I[I1_ndx]),
                &(E_work.I[I2_ndx])) == 0)
         {
                sss_reshape(&E_previous,&E_work,
                        I1_ndx,I2_ndx);
                tot_reshapes++;
        }
        else
                continue;
}
else if (
        (((float)random()/RAND_MAX)>opt_SSS_reshape_prob)
                && (opt_SSS_method == SSS_VARIABLE_RESHAPE)
){

        if (reshape_cost(&(E_work.I[I1_ndx])
                ,&(E_work.I[I2_ndx])) <= 0) {

                sss_reshape(&E_previous,
                        &E_work,I1_ndx,I2_ndx);
                tot_reshapes++;
        }
        else {
                continue;
        }
}
else if (((float) random()/RAND_MAX)>cut_prob) {

        if (!sss_random_cut(&E_previous,&E_work,
                        (rrandom(1) == 1)?I1_ndx:I2_ndx))
                continue;
        tot_cuts++;
}
else {
        continue;
}
```

```
if (queue_exists) {

	if (in_tabu_q(&TQ[0],&E_previous,
		I1_ndx,I2_ndx)) {

		dup_expr(&E_work,&E_previous);
		queue_hits_this_incre++;
		continue;
	}
	else update_tabu_q(&TQ[0],&E_previous,
		I1_ndx,I2_ndx);
}

if (E_work.nterm < E_save.nterm)
	dup_expr(&E_save,&E_work);

if (E_work.nterm > max_nterm)
	max_nterm = E_work.nterm;

if (E_work.nterm < num_impl) {
	num_impl = E_work.nterm;
	better_found = 1;
	dup_expr(&(E_final[SOLN_SPACE_SRCH]),&E_work);
}

if (tot_moves_made == opt_SSS_tabuq_len)
	moves_this_incre = 0;  /* re-zero counts after
					TABUQ fills */

tot_moves_made++;
moves_this_incre++;

if (opt_SSS_trace_profile) {
	if (first_prof) {
		printf("Max Iterations:   %10d\n",
			opt_SSS_max_iterations);
		printf("TABU Queue length:   %3d\n",
			opt_SSS_tabuq_len);
		printf("Initial Cut Prob:  %4f\n",
			opt_SSS_cut_prob);
		printf("Reshape Prob:      %4f\n",
			opt_SSS_reshape_prob);
		printf("Step Rate:         %3f\n",
```

```
                opt_SSS_step_rate);
            printf("TQ Hit Rate:        %3f\n\n",
                opt_SSS_tq_hit_rate);
            printf("Move Number  Terms  Total Iter's  Combines  Cuts
                 Reshapes Queue Hits  Incre Moves\n");
            first_prof = 0;
        }
        printf("%9d: %4d  %10d        %3d    %3d     %3d
             %3d        %4d\n",
            tot_moves_made,
            E_work.nterm,
            tot_iterations,
            tot_combines,
            tot_cuts,
            tot_reshapes,
            queue_hits_this_incre,
            moves_this_incre);
}


/*  if at equilibrium . . . increase cut_prob!  */
if ((float) queue_hits_this_incre/moves_this_incre
                        >=opt_SSS_tq_hit_rate) {

        incre_fac  *= opt_SSS_step_rate;
        cut_prob += incre_fac ;
        queue_hits_this_incre = 0;
        moves_this_incre = 0;
}

/* if in a local minimum . . . decrease cut_prob! */
if (queue_exists) {
        if (E_work.nterm == E_previous.nterm) {
            same_count++;
            if (same_count == 20) {
                cut_prob -= (1-opt_SSS_cut_prob)*0.667;
                incre_fac = (1.0-cut_prob)*
                  ((1.0-opt_SSS_step_rate) /opt_SSS_step_rate);
                same_count = 0;
                queue_hits_this_incre = 0;
                moves_this_incre = 0;
            }
```

```
			}
			else same_count = 0;
		}

		if (
				(tot_moves_made % 5 == 0) &&
				(of_name[0])
		) {
			fprintf(yyout,"%d\n",E_work.nterm);
		}

	}
done:
	resource_used(STOP);

	fprintf(yyout,"\n %d %d %d %d %d %d %d\n",
		num_impl,
		max_nterm,
		tot_moves_made,
		tot_combines,
		tot_cuts,
		tot_reshapes,
		queue_hits_this_incre);

	if (!verify_expr(&(E_final[SOLN_SPACE_SRCH])))
		fatal("Internal error; Solution Space Search
				verification failure");

	if (opt_SSS_show_stats) {
		printf("Move     Terms\n");
		for (i=0; i < tot_moves_made; i++) {
			printf("%5d:   %5d\n",
				i,
				SSS_stats[i].sss_nterm);
		}
	}

	if (opt_SSS_trace_profile)
		printf("\nMin terms: %4d   Max terms:
				%4d\n",num_impl,max_nterm);

	ratio = ((double)num_impl/(double)E_orig.nterm);
```

```
# ifdef ALEVEL_1
	if (opt_mvla && (is_redir || !opt_be_quiet)) {
		if (!better_found)
			printf("%-4d SSS: %4d/%-4d %4.2f %6ld:%3.3ld\n",
				expr_seq,num_impl, num_impl,
				 0.0,secs_used(),tsecs_used());
		else
			printf("%-4d SSS: %4d/%-4d %4.2f %6d:%3.3ld\n",
					expr_seq,num_impl,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: SSS     Perf: ");
		if (better_found)
			printf("%d\n\n",num_impl);
		else
			printf("no better\n\n");
			fflush(stdout);
	}
# endif

# ifdef ALEVEL_2
	if (opt_print_final_expr) {
		if (queue_exists)
			print_expr(&(E_final[SOLN_SPACE_SRCH]));
		else
			print_expr(&E_work);
	}
# endif

	dealloc_expr(&E_work);
}


int	build_tabu_q(T)
Implicant	*T;
/*************************************************************
	:function:
		- Allocate space for TABUQ of length MAX_TABUQ/2
*************************************************************/
{
	int	i;
```

```
	if (opt_SSS_tabuq_len != 0) {

		for (i = 0; i < (opt_SSS_tabuq_len * 2); i++) {

			*(T+i) = * alloc_implicant(NULL,1,1);
		}
		return(1);
	}
	else return(0);
}


void  sss_combine(P,E,I1_ndx,I2_ndx)
register Expression  *P,*E;
register int         I1_ndx,I2_ndx;
/****************************************************************
	:function:
		- Combines I2 INTO I1 and updates E appropriately.
		A copy of unmodified E is made to P for TABUQ entry
		if required.

		DANGER: Note the side effects on E and P
****************************************************************/
{
	register Bound  *B1,*B2;
	register int        i;

	dup_expr(P,E);

	B1 = E->I[I1_ndx].B;
	B2 = E->I[I2_ndx].B;
	for (i=0; i < nvar; i++) {
		B1[i].lower = min(B1[i].lower,B2[i].lower);
		B1[i].upper = max(B1[i].upper,B2[i].upper);
	}
	E->nterm--;
	if (I2_ndx < E->nterm)
		copy_impl(&(E->I[I2_ndx]),&(E->I[E->nterm]));
}


void  sss_reshape(P,E,I1_ndx,I2_ndx)
```

```
register Expression  *P,*E;
int   I1_ndx,I2_ndx;
/****************************************************************
      :function:
            - Reshape 2 implicants. The resulting implicants
            are added to E_work and a copy of unmodified E is
            made to P for TABUQ entry as required.
****************************************************************/
{
      static Implicant
            cons_imp,inter_imp;
      Implicant   *I1,*I2;
      register int        cost,dist,added;
      int            differ;

      dup_expr(P,E);

      if (cons_imp.B == NULL)
            cons_imp.B = alloc_bound();
      if (inter_imp.B == NULL)
            inter_imp.B = alloc_bound();

      I1 = &(E->I[I1_ndx]);
      I2 = &(E->I[I2_ndx]);

      dist = distance(I1,I2,&differ);
      if (dist == 1)
            consensus(I1,I2,&cons_imp,differ);
      else if (dist == 0)
            consensus_inter(I1,I2,&cons_imp);
      else
            fatal("reshape(): Implicants are not adjacent");

      consensus_inter(I1,&cons_imp,&inter_imp);
      inter_imp.coeff = min(I1->coeff,cons_imp.coeff);
      cost = sharp_cost(I1,&inter_imp);
      added = 0;

      if (cost == 0){
            added = 1;
            copy_impl(I1,&cons_imp);
      }
      else {
```

```
		random_sharp(E,I1_ndx,&inter_imp,cost);
	}
		/* CAUTION: Below this line, pointers I1,I2 may be defunct */

	I2 = &(E->I[I2_ndx]);
	consensus_inter(I2,&cons_imp,&inter_imp);
	inter_imp.coeff = min(I2->coeff,cons_imp.coeff);
	cost = sharp_cost(I2,&inter_imp);

	if (cost == 0){
		added = 1;
		copy_impl(I2,&cons_imp);
	}
	else {
		random_sharp(E,I2_ndx,&inter_imp,cost);
	}
		/* CAUTION: Below this line, pointers I1,I2 may be defunct */
	if (!added){
		E->I = alloc_implicant(E->I,cons_imp.coeff. ++(E->nterm));
		copy_impl(&(E->I[E->nterm-1]),&cons_imp);
	}
}

int	sss_random_cut(P,E,I_ndx)
Expression	*P,*E;
int		I_ndx;
/*************************************************************
	:function:
		- Perform random cut of Implicant.  A copy of
		unmodified E is made to P for TABUQ entry as
		required.
*************************************************************/
{
	static struct coeff_struct {
		short a,b;
	} *coeff_tab = NULL;
	static int	ncoeff,old_radix = 0;
	register Implicant	*I;
	register int		i,j;
	register int		bound_cuts,coeff_cuts,r_cut,max_coeff;
	bound_cuts = 0;

	dup_expr(P,E);
```

```
I = &(E->I[I_ndx]);

for (i=0; i < nvar; i++)
     bound_cuts += (I->B[i].upper - I->B[i].lower);

if (I->coeff == (radix -1)) {
     if ((coeff_tab == NULL) || (radix != old_radix)) { old_radix = radix;
          max_coeff = (((radix+1)/2)*((radix+2)/2))-1;
          if (coeff_tab != NULL)
               free(coeff_tab);
          coeff_tab = (struct coeff_struct *)
               malloc(sizeof(struct coeff_struct) *
                         max_coeff);
          if (coeff_tab == NULL)
               fatal("random_cut(): Out of memory\n");
          ncoeff = 0;
          for (i=1; i < radix; i++) {
               for (j = max((radix-1)-i,i); j<radix; j++ ) {
                    if (ncoeff >= max_coeff)
                         fatal("random_cut(): coeff table
                                   overflow");
                    coeff_tab[ncoeff].a = i;
                    coeff_tab[ncoeff++].b = j;
               }
          }
     }
     coeff_cuts = ncoeff;
}
else {
     coeff_cuts = I->coeff - 1;
}

/* If no cuts are possible ... */
if (!(coeff_cuts || bound_cuts)) {
     return(0);
}
r_cut = rrandom(bound_cuts + coeff_cuts) + 1;

if (r_cut <= bound_cuts) {
     /* Cut bounds */
     for (i=0; (I->B[i].upper - I->B[i].lower)<r_cut;
                                   i++)
          r_cut -= (I->B[i].upper - I->B[i].lower);
```

```
            cut(E,I_ndx,i,I->B[i].lower + (r_cut-1));
        }
        else if (I->coeff == (radix - 1)) {
            /* Cut coefficients */
            i = (r_cut - bound_cuts) - 1;
            cut_coeff(E,I_ndx,coeff_tab[i].a,coeff_tab[i].b);
        }
        else {
            r_cut -= bound_cuts;
            cut_coeff(E,I_ndx,I->coeff-r_cut,r_cut);
        }
        return(1);
}


int     in_tabu_q(T,E,I1_ndx,I2_ndx)
register Implicant *T;
register Expression *E;
int             I1_ndx, I2_ndx;
/**************************************************************
        :function:
            - search TABUQ for implicant pair
**************************************************************/
{
        register Implicant *I1,*I2,*TE1,*TE2,*Temp;
        int     i, j, a, b, bounds_good = 1;
        static int      tqscnt = 0;

        I1 = &(E->I[I1_ndx]);
        I2 = &(E->I[I2_ndx]);

        if (tqscnt < opt_SSS_tabuq_len) {

            tqscnt++;
            return(0);
        }

        for (i=0; i < opt_SSS_tabuq_len; i++) {

            TE1 = &T[2*i];
            TE2 = &T[2*i+1];

            /* order implicants by coeff */
```

```
            if (I1->coeff > I2->coeff) {
                Temp = I1;
                I1 = I2;
                I2 = Temp;
            }

            if (
              (TE1->coeff == I1->coeff) &&
              (TE2->coeff == I2->coeff)
              ) {

                for (j=0; j < nvar; j++) {

                    if (
                      (TE1->B[j].lower == I1->B[j].lower) &&
                      (TE1->B[j].upper == I1->B[j].upper)
                      ){

                        if (
                          (TE2->B[j].lower == I2->B[j].lower) &&
                          (TE2->B[j].upper == I2->B[j].upper)
                          ){
                          continue;
                        }

                        bounds_good = 0;
                        break;
                    }
                    bounds_good = 0;
                    break;
                }
            if (bounds_good) return(1);
            }
        }
        return(0);
}

void update_tabu_q(T,E,I1_ndx,I2_ndx)
Implicant    *T;
Expression        *E;
int    I1_ndx, I2_ndx;
/*************************************************************
        :function:
```

```
        - add implicant pair to TABUQ

        NOTE:  TABUQ is a FIFO queue
*************************************************************/
{
    register Implicant*TE1,*TE2,*I1,*I2,*temp;
    static int   qudcnt = 0;
    int          i;

    i = qudcnt % opt_SSS_tabuq_len;

    qudcnt += 1;

    TE1 = &T[2*i];
    TE2 = &T[2*i+1];

    I1 = &(E->I[I1_ndx]);
    I2 = &(E->I[I2_ndx]);

    /* order implicants by coeff  */
    if (I1->coeff > I2->coeff) {
        temp = I1;
        I1 = I2;
        I2 = temp;
    }
    copy_impl(TE1,I1);
    copy_impl(TE2,I2);
}
```

2. Other HAMLET files modified or use with SSS: *config.c*, *main.c* and *defs.h*. Major additions are listed below:

a. *config.c*:

(1) SSS help panel:

```
static char *SSS_help[] = {
" -ZSlx    - Set the TABU queue length to x (default = 1000)",
"                                  [MAX LENGTH = 10000]",
" -ZScx    - Set the Cut Probability to x (default = 0.99)",
" -ZSrx    - Set the Reshape Probability to x (default = 0.50)",
" -ZSsx    - Set the Step Rate to x (default = 0.90)",
" -ZSqx    - Set the TABU queue hit rate to x (default = 0.0001)",
```

```
" -ZSix     - Set Max Iterations to x (default = 12000000)",
" -ZSoFile - Output data for SSS to \"File\"",
" -ZSv      - Select Variable Cost Reshape move (default is",
"                Zero Cost Reshape)",
" -Zc       - Show the heuristic's performance even if the",
"             user's input could not be bettered (default is",
"             give up)",
" -Zs       - Show statistics",
" -Zt       - Trace the SSS profile",
NULL
};
```

(2) SSS global variables/initialization:

```
/* Globals for Soln Space Srch */
int
    opt_SSS_tabuq_len = SSS_INITIAL_TABUQ_LEN,
    opt_SSS_max_iterations = SSS_MAX_ITERATIONS,
    opt_SSS_method = SSS_ZERO_RESHAPE,
    opt_SSS_trace_profile = 0,
    opt_SSS_show_stats = 0;

double
    opt_SSS_cut_prob = SSS_CUT_PROB,
    opt_SSS_reshape_prob = SSS_RESHAPE_PROB,
    opt_SSS_step_rate = SSS_STEP_RATE,
    opt_SSS_tq_hit_rate = SSS_TQ_HIT_RATE;
```

(3) Code for parsing SSS command line options:

```
char *SSS_options(arg,p)
char *arg,*p;
{
    register    i;

    if (!p[0])
        return(p);

    if (*p == '-') {
        printf("\n%s\n%s",version,usage);
        printf("\nSolution Space Search options:\n");
        for (i=0; SSS_help[i]; i++)
            printf("%s\n",SSS_help[i]);
```

```
        exit(0);
    }

    if (*p++ == 'Z') {
        while (*p) {
            switch (*p++) {
            case 'c':
                opt_S_to_coverage++;
                break;
            case 's':
                opt_SSS_show_stats++;
                break;
            case 't':
                opt_SSS_trace_profile++;
                break;
            case 'S':
                if (*p == 'c') {
                    p++;
                    if (!(isdigit(*p) || (*p == '.'))) {
                        err_option(arg,p,  "positive float
                                    expected after -ZSc");
                    }
                    sscanf(p," %lf",&opt_SSS_cut_prob);
                    if (
                      (opt_SSS_cut_prob < 0.0) ||
                      (opt_SSS_cut_prob > 0.99999)
                    ){
                    err_option(arg,p,"Cut Prob. must be:
                                0.0 < c < 1.0");
                    }
                    *p = '\0';
                }
                else if (*p == 'r') {
                    p++;
                    if (!(isdigit(*p) || (*p == '.'))) {
                        err_option(arg,p,"positive float
                                   expected after -ZSr");
                    }
                    sscanf(p," %lf",&opt_SSS_reshape_prob);
                    if (
                          (opt_SSS_reshape_prob < 0.0) ||
                          (opt_SSS_reshape_prob > 0.99999)
                    ){
```

```
            err_option(arg,p,"Reshape Prob.
                must be: 0.0 < r < 1.0");
        }
        *p = '\0';
    }
    else if (*p == 's') {
        p++;
        if (!(isdigit(*p) || (*p == '.'))) {
            err_option(arg,p,"positive float
                    expected after -ZSs");
        }
        sscanf(p,"%lf",&opt_SSS_step_rate);
        if (
          (opt_SSS_step_rate < 0.0) ||
          (opt_SSS_step_rate > 0.99999)
        ){
            err_option(arg,p,"Step Rate must be:
                        0.0 < s < 1.0");
        }
        *p = '\0';
    }
    else if (*p == 'q') {
        p++;
        if (!(isdigit(*p) || (*p == '.'))) {
            err_option(arg,p,"positive float
                    expected after -ZSq");
        }
        sscanf(p,"%lf",&opt_SSS_tq_hit_rate);
        if (
            (opt_SSS_tq_hit_rate < 0.0) ||
            (opt_SSS_tq_hit_rate > 0.99999)
        ){
            err_option(arg,p,"TQ Hit Rate must
                        be: 0.0 < s < 1.0");
        }
        *p = '\0';
    }
    else if (*p == 'l') {
        p++;
        if (!isdigit(*p)) {
            err_option(arg,p,"positive integer
                    expected after -ZSl");
        }
```

```
                sscanf(p,"%d",&opt_SSS_tabuq_len);
                *p = '\0';
            }
            else if (*p == 'i') {
                p++;
                if (!isdigit(*p)) {
                    err_option(arg,p,"positive integer
                                expected after -ZSi");
                }
                sscanf(p,"%d",&opt_SSS_max_iterations);
                if (opt_SSS_max_iterations < 1) {
                        err_option(arg,p,
                            "max iterations must be > 0");
                }
                *p = '\0';
            }
            else if (*p == 'o') {
                p++;
                strcpy(sss_of_name,p);
                if ((yyout=fopen(sss_of_name,"w")) ==
                    NULL) {
                    fprintf(stderr,"SSS: Can't open
                                %s\n",sss_of_name);
                exit(1);
                }
                *p = '\0';
            }
            else if (*p == 'v') {
                opt_SSS_method =
                    SSS_VARIABLE_RESHAPE;
                *p = '\0';
            }
            else
                err_option(arg, p,"illegal option after
                                        -ZS");
            break;
        default:
            err_option(arg, p-1, "illegal option after -Z");
        }
    }
}
else
    err_option(arg, p, "unknown option");
```

```
    return(p);
}
```

b. *main.c*:

(1) Case option for SSS:

```
case SOLN_SPACE_SRCH:
        Soln_Space_Srch();
        FINAL = SOLN_SPACE_SRCH;
        break;
```

c. *defs.h*:

(1) SSS definitions:

```
#define SOLN_SPACE_SRCH              13
#define SSS_INITIAL_TABUQ_LEN        1000
#define SSS_MAX_ITERATIONS           12000000
#define SSS_ZERO_RESHAPE             0
#define SSS_VARIABLE_RESHAPE         1
#define SSS_CUT_PROB                 0.99
#define SSS_RESHAPE_PROB             0.50
#define SSS_STEP_RATE                0.90
#define SSS_TQ_HIT_RATE              0.0001
```

(2) SSS global variable definition:

```
/* Globals for Soln Space Srch */

extern int
        opt_SSS_tabuq_len,
        opt_SSS_max_iterations,
        opt_SSS_method,
        opt_SSS_trace_profile,
        opt_SSS_show_stats;

extern double
        opt_SSS_cut_prob,
        opt_SSS_reshape_prob,
        opt_SSS_step_rate,
        opt_SSS_tq_hit_rate;
```

LIST OF REFERENCES

1. Kerkhoff, H. G., "Theory and design of multiple-valued logic CCD's," *Computer Science and Multiple-Valued Logic* (ed. D. C. Rine), pp. 502-537, North Holland, New York, 1984.

2. Butler, J. T. and Kerkhoff, H. G., "Multiple-Valued CCD Circuits," *IEEE Computer*, pp. 58-69, March 1988.

3. Yurchak, J. M. and Butler, J. T., "HAMLET - An Expression Compiler/Optimizer for the Implementation of Heuristics to Minimize Multiple-Valued Programmable Logic Arrays," *Proceedings of the 20th International Symposium on Multiple-Valued Logic,* pp. 144-152, May 1990.

4. Naval Postgraduate School Technical Report NPS-6290-015. *HAMLET user reference manual*, J. M. Yurchak and J. T. Butler, July 1990.

5. Dueck, G. W., Earle, R. C., Tirumalai, P. P., and Butler, J. T., "Multiple-Valued Programmable Logic Array Minimization by Simulated Annealing," *Proceedings of the 22nd International Symposium on Multiple-Valued Logic*, pp. 66-74, May 1992.

6. Dueck, G. W. and Miller, D. M., "A direct cover MVL minimization using the truncated sum," *Proceedings of the 17th International Symposium on Multiple-Valued Logic*, pp. 221-227, May 1987.

7. Pomper, G. and Armstrong, J. A., "Representation of Multivalued functions using the direct cover method," *IEEE transactions on Computing*, pp. 674-679, September 1981.

8. Yang, C. and Wang, Y.-M., "A neighborhood decoupling algorithm for truncated sum minimization," *Proceedings of the 20th International Symposium on Multiple-Valued Logic*, pp. 153-160, May 1990.

9. Besslich, P. W., "Heuristic minimization of MVL functions: A direct cover approach," *IEEE Transactions on Computing*, pp. 134-144, February 1986.

10. Kirkpatrick, S., Gelatt, Jr., C. D., and Vecchi, M. P., "Optimization by Simulated Annealing," *SCIENCE*, Vol. 220, No. 4598, pp. 671-680, 13 May 1983.

11. Collins, N. E., Eglese, R. W., and Golden, B. L., "Simulated Annealing - An Annotated Bibliography," *American Journal of Mathematical and Management Science*, Vol. 8, Nos. 3 and 4, pp. 209-307, 1988.

12. Song, L. and Vannelli, A., "A VLSI Placement Method Using TABU Search," *Canadian Conference on VLSI*, August 1991.

13. Finney, R. L., and Thomas, G. B., Jr., *Calculus*, 1st ed., p. 615, Addison-Wesley Publishing Co., 1990.

14. Kerkhoff, H. G., and Butler, J. T., "Design of a high-radix programmable logic array using profiled peristaltic charge-coupled devices," *Proceedings of the 16th International Symposium on Multiple-Valued Logic*, pp. 100-103, May 1986.

BIBLIOGRAPHY

1. Tirumalai, P. P. and Butler, J. T., "Prime and Non-Prime Implicants in the Minimization of Multiple-Valued Logic Functions," *Proceedings of the 19th International Symposium on Multiple-Valued Logic*, May 1989.

2. VanLaarhoven, P. J. M. and Aarts, E. H. L., *Simulated Annealing: Theory and Applications*, D. Reidel Publishing Company, 1987.

3. Otten, R. H. J. M. and van Giancken, L. P. P. P., *The Annealing Algorithm*, Kluwer Academic Publishers, 1989.

4. Lam, J. and Delosme, J.-M., "Logic Minimization Using Simulated Annealing," *Proceedings of the IEEE International Conference on Computer Aided Design*, pp. 348-351, November 1986.

5. Yao, X. and Liu, C. L., "PLA Logic Minimization by Simulated Annealing," *INTEGRATION, the VLSI Journal 9*, pp. 243-257.

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 93943-5101	2
3.	Chairman, Code EC Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	1
4.	Professor Jon T. Butler, Code EC/Bu Department of Electrical and Computer Engineering Naval Postgraduate School Monterey, CA 93943-5121	1
5.	Visiting Assistant Professor David Erickson, Code CS/Er Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5118	1
6.	Professor Chyan Yang National Chiao Tung University Inst. of Management Sci. and Inst. of Information Sci. Hsinchu, TAIWAN Republic of Taiwan	1

7. Dr. George Abraham, Code 1005 1
Office of Research and Technology
Naval Research Laboratories
4555 Overlook Ave., N.W.
Washington, DC 20375

8. Dr. Robert Williams 1
Naval Air Development Center, Code 5005
Warminister, PA 18974-5000

9. Dr. James Gault 1
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709

10. Dr. Andre van Tilborg 1
Office of Naval Research, Code 1133
800 N. Quincy St.
Arlington, VA 22217-5000

11. Dr. Clifford Lau 1
Office of Naval Research
1030 E. Green St.
Pasadena, CA 91106-2485

12. LCDR John M. Yurchak, USN 1
JWC-OR
Hurlburt Field, FL 32544-5000

13. LCDR Charles G. Wendt, USN 1
3 Frank Hunt Court
Poquoson, VA 23662-1943