Using a search heuristic in an NP-complete problem in Ashenhurst-Curtis Decomposition.

Craig Files
Masters Student
Department of Electrical Engineering

University of Idaho
Moscow, Idaho 83844-1023
cfiles@gemini.mrc.uidaho.edu

Final Report for:
Graduate Summer Research Program
Wright Laboratory

Sponsored by:
Air Force Office of Scientific Research
Bolling Air Force Base, DC

and

Wright Laboratory

August 1994

Using a search heuristic in an NP-complete problem in Ashenhurst-Curtis Decomposition.

Craig M. Files
Masters Student
Department of Electrical Engineering
University of Idaho

Abstract

This paper reports on the application of several searching algorithms to Ashenhurst-Curtis decomposition. A final search algorithm is then presented that is base on using different values instead of just using column multiplicity to guide the search. The influence behind this algorithm is the analogous behavior of functional decomposition to decision trees. By using this algorithm 2^N possible partitions can be searched by evaluating $\frac{N*(N+1)}{2}$ partitions.

Introduction

This paper reports on the application of a search algorithm to an NP-complete combinatorial problem of pattern theory. Pattern theory is a concept for finding patterns in discrete and continuous mathematical functions. This approach has been developed at Wright Laboratories, in an attempt to find a more formal notation of a pattern. To gain an understanding of the object's "patterness" one has to be able to measure the complexity of a pattern. Measuring this complexity must be robust enough to handle many classes of applications. The applications considered by Wright Laboratories include pattern recognition, algorithm creation, data compression, machine learning, and logic minimization. Their approach is to use the Ashenhurst-Curtis decomposition for a binary function, called functional decomposition[1].

It is well known that functional decomposition is an NP-complete problem. Because of this complexity, the Ashenhurst-Curtis method is rarely used in industry, while algebraic factorization and binary decision diagram methods are used instead. There has been, however, some recent increase of interest in functional decomposition for applications in FPGA mapping and PLA decomposition[9].

The algorithm proposed here uses heuristic search to find close to minimum decomposition. The reference of minimum decomposition, in this paper, means the smallest representation of the function as a composition of two other functions. The advantage of this algorithm is in its ability to find solutions of this exponential problem in polynomial time. The algorithm "Ashenhurst-Curtis Search Heuristic (ASH)" is a reduction method that looks at groups of variables. By

using a figure of merit for each individual variable in the group, the variable with the best merit is then used to construct the next set of groups. The rest of this paper will discuss Ashenhurst-Curtis decomposition, previous work on this topic, the theory of ASH, and results.

2.0 Functional Decomposition

The decomposition of a function is an expression in terms of a composition of other functions. For example, if $f(x_1,x_2,..,x_4)=F(\Phi(\Im(x_1,x_2),x_3),x_4)$ then the term to the right is a set of compositions that are the equivalence of f. This process is called functional decomposition. It must be noted that functional decomposition has been recently formulated for multi-valued[9] and real-valued functions[3], but in this paper binary functions will be used.

In general, an N input function, $f: B^N \rightarrow B$, where $B=\{0,1\}$, has the set of input variables $\{x_1,x_2,..,x_N\} \in X$. In this paper we will use Ashenhurst-Curtis' definition of trivial decomposition to partition the input variables. A partition is a collection of subsets whose union is the whole set and whose intersections are empty. The partition is then used to separate the input variables into different functions to form a decomposition. For example, if $A \cup B=\{x_1,x_2,..,x_N\}$ and $A \cap B=\emptyset$, then the decomposition can be written as $f(A,B)=F(\Phi(A),B)$, where Φ and F are the composite functions of f.

Some simple forms of decomposition:

1. $f(A,B)=F(\Phi(A),B)$
2. $f(A,B)=F(\Phi(A),\Im(B))$

3. $f(A,B,C)=F(\Phi(A),\Im(B), C)$

4. $f(A,B,C)=F(\Phi(\Im(A), B), C)$

In each of these cases $A \cup B \cup C=X$ and $A \cap B=\emptyset$, $B \cap C=\emptyset$, and $A \cap C=\emptyset$. The decomposition $F(\Phi(A),B)$ is the most basic form of decomposition and is used in the implementation described in this paper.

A measure of a function's complexity is called the "function cardinality", and is defined as 2^N*M, for an N input, M output function. For means of representation this is called the decomposed function cardinality(DFC). DFC is defined as a cost of the minimum decomposition. Since we are only evaluating the decomposition of f(A,B) into $F(\Phi(A),B)$, we will denote the expression as the sum of child cardinality(SOCC). Thus, once a partition has been selected, and each of the functions in the composition have been found, the SOCC may be calculated. The SOCC is calculated as the cardinality of Φ, plus the cardinality of F. The example 2.0 illustrates calculating the SOCC of a given function.

Example 2.0

Starting with a 6 input function, cardinality = 2^6=64. After decomposing the function, the netlist of decomposed blocks from Fig 2.0 is created.

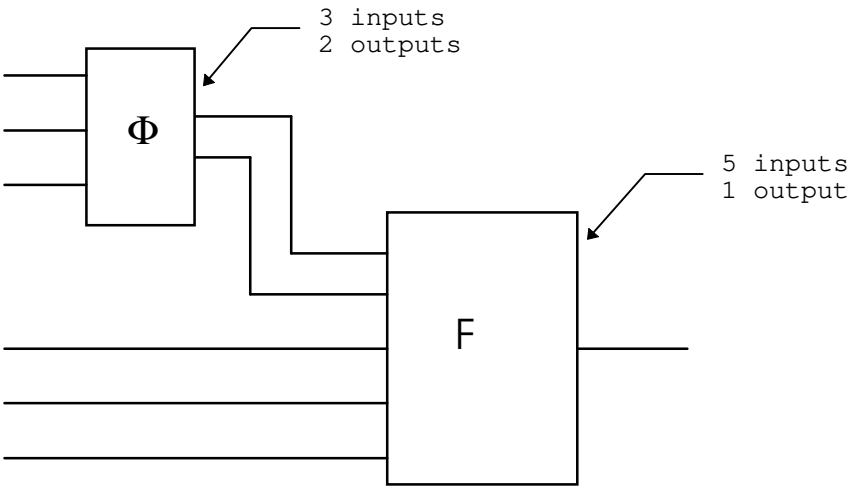


Fig 2.0 – Example of decomposed blocks

SOCC = cardinality(Φ) + cardinality(F)

SOCC = 2^3 * 2 + 2^5 * 1= 8 * 2 + 32 = 48

The concept of the DFC measure is used at Wright Labs as a robust measure of pattern. Thus, DFC can be used in common pattern recognition applications, such as the following machine learning approaches of geometric, syntactic and artificial intelligence. DFC has also been shown to be related to factors in data compression, time complexity and size complexity of a circuit[8]. SOCC is used because of its simple and obvious implementation of DFC.

A principal reason that Ashenhurst-Curtis decomposition hasn't been used much for practical engineering applications is because of the

search complexity that it involves. To find the minimum SOCC for an N variable input function to be placed in the form f(A,B)=F(Φ(A),B), there are 2^N partitions that could be evaluated. Because of the number of evaluations, the problem as stated is exponential.

3.0 Previous Work

Several approaches have been examined to reduce the complexity of finding the minimum decomposition. One such approach was the use of the Genetic Algorithm(GA)[6][7]. GAs have been used to find approximate solutions to NP-complete optimization problems[2], so it looked like an appropriate approach to this problem. But, according to the best of our knowledge there has been very few, if any, successful applications of genetic algorithms in the area of logic synthesis.

The GA is an evolutionary learning technique, developed by Holland(1975), that is based on genetic variation and natural selection. The GA is a simulation of the basic processes that are found in nature. Included in these processes are natural selection and reproduction. Highly fit individuals are allowed to reproduce with other individuals in the population. While other, not fit, individuals will be removed from the population to make way for the newly formed individuals. By using this type of selection, hopefully the most promising areas of the search space are explored. The GA should then converge towards an optimal solution of the problem. Figure 3.0 is the basic pseudo-code for a GA.

```
Initialize population

repeat
      evaluate population
      reproduce
until done;
```

Fig. 3.0

The initial population is probably the most important part of the GA. If the population is diverse then the probability of finding solutions will be much greater than if the population is localized in one area of the search space. This gives the GA the variance that it needs to complete the task. Another element in the implementation of the GA is evaluating the population. This is where the individuals of the population are given their fitness, or SOCC in the case of the work that has been conducted with functional decomposition.

Another approach to finding the best partitions in Ashenhurst-Curtis decomposition is the method of increasing row-column ratio(IRC)[5]. This search technique takes advantage of the fact that the minimum SOCC is usually found when there are very few row variables. Row variables are defined as the variables included in the set A of the expression $F(\Phi(A),B)$. Thus, row variables are the input variables to the composite function Φ. Column variables are then defined as the input variables not in A, but are the elements of the set B. The algorithm IRC evaluates the partitions of a given function by starting with all the possibilities of having one row variable. This is continued until all the possibilities of having N row variables have been evaluated.

Both of these approaches, on average, found the minimum SOCC faster than using a random search technique. The problem was that the algorithms didn't guarantee when the minimum SOCC would be found. Since the work was done experimentally, there was no real stopping condition for the algorithms. In some cases, the entire search space, all 2^N, would have to be searched before finding the minimum SOCC.

4.0 ASH

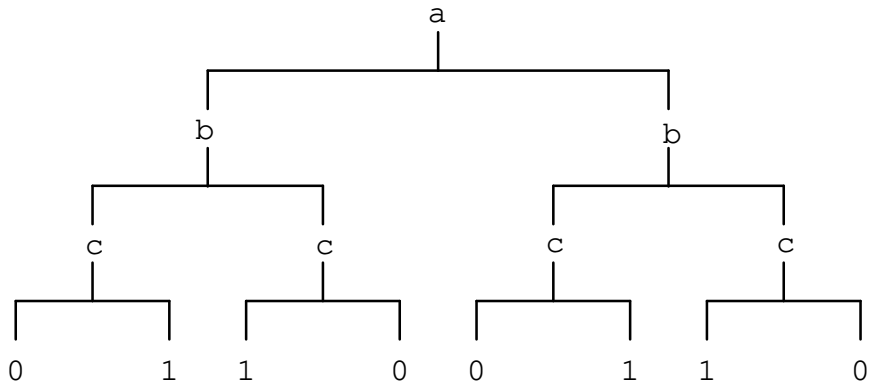
The concept of ASH is based on the idea of representing functional decomposition as a decision tree(DT), in the machine learning paradigm. The most general way of developing a DT is to start building it from the top, or the root. In the case of ASH, we are concerned with building the tree from the bottom, or the leaf nodes. Selecting the bottom node in the DT can be done by going through all N possible variable combinations. The variable or node that has the most folding, or matching elements, is then chosen as the best bottom node. The following is an example of doing this.

Example 4.0 – Decision Trees

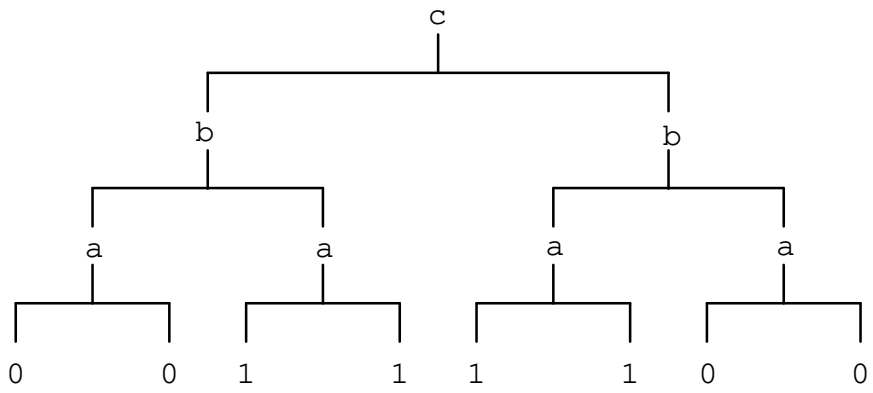
Given the following 3 input function.

f(a, b, c)= 01100110.

Then the following decision tree exists.



In this case, using c as the bottom node in the tree has four mismatches. The term mismatches is used to denote the elements under c that are not equal. The result that is wanted is to fold the elements under c, but this can only occur when the elements are equal. By switching the nodes in the tree such that a is the bottom node, the following tree results.



In this configuration there are no mismatches, thus indicating a is the best candidate for being the bottom node of the tree. Note that since there are no mismatches then a is a vacuous variable, meaning that the variable has no relevant effect on the output of the function.

This process is then continued to select the next level, and proceeding until choosing the top, or root of the tree.

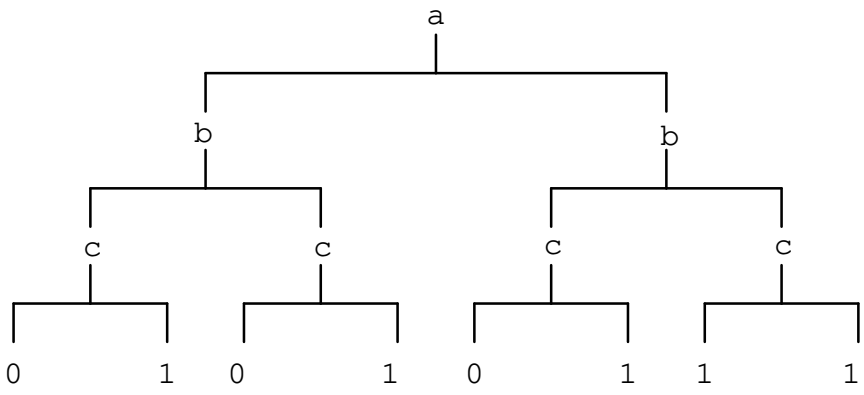
The process of choosing the bottom level of nodes in the tree is the same as selecting all the possible partitions that have only one column variable. The column variable that returns the best properties is then chosen as the best possible column variable. The next set of

partitions that are used will be the best column variable individually paired with all the remaining variables. Which is N-1 possible partitions. This process is continued until there is only 1 possible partition to evaluate.

The best partition is selected using the following cost function. Cost is defined as finding the partition with the smallest column multiplicity, then finding the smallest number of row mismatches. Example 4.1 shows the way ASH uses this Cost function to find the minimum SOCC.

Example 4.1 - Decision tree to functional decomposition

Given the following 3 input function
f(a, b, c) = 01010111
The following decision tree exists



Which can then be represented as the following decomp. table:

ab \ c		
0	1	- Mismatch
0	1	- Mismatch
0	1	- Mismatch
1	1	- Match

column multiplicity ν = 2
number of row mismatches = 3

b as the bottom level of nodes represented as a decomp. table:

ac \ b		
0	0	- Match
1	1	- Match
0	1	- Mismatch
1	1	- Match

column multiplicity ν = 2
number of row mismatches = 1

a as the bottom level of nodes represented as a decomp. table:

bc \ a		
0	1	- Mismatch
1	0	- Mismatch
0	1	- Mismatch
1	1	- Match

column multiplicity ν = 2
number of row mismatches = 3

From the above three partitions, the partition with b as the single column variable is chosen to be the best. So now b is paired with the other 2 variables as column variables.

b and a as column variables.

c \ ba				
0	0	1	1	- Mismatch
0	1	1	1	- Mismatch

column multiplicity ν = 3
number of row mismatches = 2

b and c as column variables.

a \ bc			
0	0	0	1
1	1	1	1

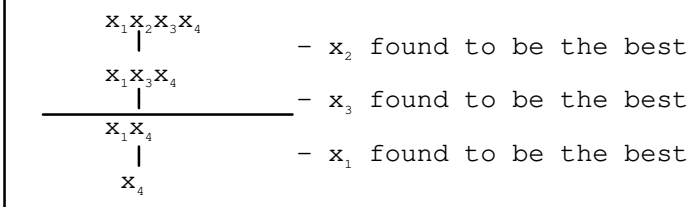
– Mismatch
– Match

column multiplicity ν = 2
number of row mismatches = 1

From the last two partitions, the partition with b and c as the column variables is chosen to be the best. Since the only partition that still exists is the situation when all three variables are column variables, means that ASH is done.

Summary of ASH; at each level of choosing the best partition, the variables in the best partition are then paired with the remaining individual variables. Figure 4.0 is graphical representation of this.

Figure 4.0 – 4 variable function



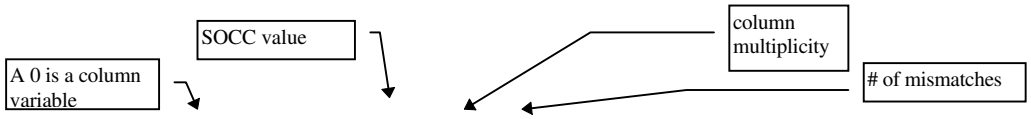
From Figure 4.0, there are four input variables to the function f, by using each variable as an independent column variable the best input variable for the composite function is chosen. That leaves three more variables to look at, and the process continues until there are no more variables to look at. At each level a new variable is selected to be column variable, x_2 is chosen first, then x_3 and so on. This process results in the evaluation of 4+3+2+1=10 partitions, since there are 4 partitions to be evaluated with 4 input variables, 3 partitions to be evaluated with 3 variables, and so on. For an N input function, Eulers

definition gives us $\frac{N*(N+1)}{2}$ evaluated partitions. Which states that ASH is a polynomial algorithm.

At some point, in this search, ASH will hold the best SOCC found during the search. From Figure 4.0, that is denoted by the horizontal line. This is the point when all the column variables chosen up until this point will return the best SOCC. After this point any additional column variables will make the SOCC grow in size.

4.1 ASH Example

The following is an example of the ASH algorithm decomposing an eight variable function.




```
 0 partition:  01111111 SOCC 256   2  64  best part: 01111111 SOCC 256  2  64
 1 partition:  10111111 SOCC 256   2  32  best part: 10111111 SOCC 256  2  32
 2 partition:  11011111 SOCC 256   2  16  best part: 11011111 SOCC 256  2  16
 3 partition:  11101111 SOCC 256   2   8  best part: 11101111 SOCC 256  2   8
 4 partition:  11110111 SOCC 256   2  64  best part: 11101111 SOCC 256  2   8
 5 partition:  11111011 SOCC 256   2  32  best part: 11101111 SOCC 256  2   8
 6 partition:  11111101 SOCC 256   2  16  best part: 11101111 SOCC 256  2   8
 7 partition:  11111110 SOCC 256   2   8  best part: 11101111 SOCC 256  2   8
best ----------- partition 11101111 SOCC  256   2  8
 8 partition:  01101111 SOCC 256   4  36  best part: 11101111 SOCC 256  2   8
 9 partition:  10101111 SOCC 256   4  20  best part: 11101111 SOCC 256  2   8
10 partition: 11001111 SOCC 256   4  12  best part: 11101111 SOCC 256  2   8
11 partition: 11100111 SOCC 256   4  36  best part: 11101111 SOCC 256  2   8
12 partition: 11101011 SOCC 256   4  20  best part: 11101111 SOCC 256  2   8
13 partition: 11101101 SOCC 256   4  12  best part: 11101111 SOCC 256  2   8
14 partition: 11101110 SOCC 132   2   8  best part: 11101110 SOCC 132  2   8
best ----------- partition 11101110 SOCC  132   2  8
15 partition: 01101110 SOCC 144   4  20  best part: 11101110 SOCC 132  2   8
16 partition: 10101110 SOCC 144   4  12  best part: 11101110 SOCC 132  2   8
17 partition: 11001110 SOCC 144   3   8  best part: 11101110 SOCC 132  2   8
18 partition: 11100110 SOCC 144   4  20  best part: 11101110 SOCC 132  2   8
19 partition: 11101010 SOCC 144   4  12  best part: 11101110 SOCC 132  2   8
20 partition: 11101100 SOCC 144   3   8  best part: 11101110 SOCC 132  2   8
best ----------- partition 11001110 SOCC  144   3  8
21 partition: 01001110 SOCC 176   6  12  best part: 11101110 SOCC 132  2   8
22 partition: 10001110 SOCC 176   5   8  best part: 11101110 SOCC 132  2   8
23 partition: 11000110 SOCC 176   6  12  best part: 11101110 SOCC 132  2   8
24 partition: 11001010 SOCC 176   5   8  best part: 11101110 SOCC 132  2   8
25 partition: 11001100 SOCC  48   2   4  best part: 11001100 SOCC  48  2   4
best ----------- partition 11001100 SOCC   48   2  4
26 partition: 01001100 SOCC  96   4   6  best part: 11001100 SOCC  48  2   4
27 partition: 10001100 SOCC  96   3   4  best part: 11001100 SOCC  48  2   4
28 partition: 11000100 SOCC  96   4   6  best part: 11001100 SOCC  48  2   4
29 partition: 11001000 SOCC  96   3   4  best part: 11001100 SOCC  48  2   4
best ----------- partition 10001100 SOCC   96   3  4
30 partition: 00001100 SOCC 224   5   2  best part: 11001100 SOCC  48  2   4
31 partition: 10000100 SOCC 224   5   3  best part: 11001100 SOCC  48  2   4
32 partition: 10001000 SOCC  72   2   2  best part: 11001100 SOCC  48  2   4
best ----------- partition 10001000 SOCC  72   2  2
```

```
33 partition: 10000000 SOCC 256   3   1  best part: 11001100 SOCC  48  2   4
34 partition: 00001000 SOCC 256   3   1  best part: 11001100 SOCC  48  2   4
best ----------- partition 00000100 SOCC  256   3  1
BEST : partition 11001100 SOCC   48   4   0
```

The next example is used to show how vacuous variables are found and then implemented as column variables. Note, the function being analyzed is completely specified so vacuous variables are specified as column variables.

```
0 partition:  01111111 SOCC 256   2  80  best part: 01111111 SOCC 256  2  80
1 partition:  10111111 SOCC 256   2  48  best part: 10111111 SOCC 256  2  48
2 partition:  11011111 SOCC 128   1   0  best part: 11011111 SOCC 128  1   0
3 partition:  11101111 SOCC 256   2  16  best part: 11011111 SOCC 128  1   0
4 partition:  11110111 SOCC 128   1   0  best part: 11011111 SOCC 128  1   0
5 partition:  11111011 SOCC 256   2  16  best part: 11011111 SOCC 128  1   0
6 partition:  11111101 SOCC 128   1   0  best part: 11011111 SOCC 128  1   0
7 partition:  11111110 SOCC 128   1   0  best part: 11011111 SOCC 128  1   0
vacuous variables
8 partition:  11010100 SOCC  16   1   0  best part: 11010100 SOCC  16  1   0
best ----------- partition 11010100 SOCC   16   1  0
9 partition:  01010100 SOCC  48   2   5  best part: 11010100 SOCC  16  1   0
10 partition: 10010100 SOCC  48   2   3  best part: 11010100 SOCC  16  1   0
11 partition: 11000100 SOCC  48   2   1  best part: 11010100 SOCC  16  1   0
12 partition: 11010000 SOCC  48   2   1  best part: 11010100 SOCC  16  1   0
best ----------- partition 11000100 SOCC   48   2  1
13 partition: 01000100 SOCC 144   3   3  best part: 11010100 SOCC  16  1   0
14 partition: 10000100 SOCC 144   3   2  best part: 11010100 SOCC  16  1   0
15 partition: 11000000 SOCC  72   2   1  best part: 11010100 SOCC  16  1   0
best ----------- partition 11000000 SOCC   72   2  1
16 partition: 01000000 SOCC 256   3   2  best part: 11010100 SOCC  16  1   0
17 partition: 10000000 SOCC 132   2   1  best part: 11010100 SOCC  16  1   0
best ----------- partition 10000000 SOCC  132   2  1
BEST : partition 11010100 SOCC   16   1  0
```

5.0 Results

The results of ASH are analyzed and compared to the other methods that have been implemented for finding the minimum partition. The research group at Wright Laboratories has created 53, 8-variable, and 14, 12-variable benchmark functions that have been used to evaluate the effectiveness of ASH. The results where obtained by using two software packages, FLASH, Function Learning and Synthesis Hotbed, and FLASHETTE. FLASH was implemented by the Systems Concepts Group of the Mission Avionics Division, Wright Laboratory[6]. FLASHETTE is a smaller version of FLASH that is the implementation of ASH.

In determining the effectiveness of the algorithms the following notation is used. The cost for each function is 1.0 if the minimum partition has not been found, 0.0 if the minimum has been found.

$$\text{Normalized Cost} = \frac{\text{Cost of each function}}{\text{number of functions}}$$

This method shows the speed in which the algorithm finds the optimal solution. See figures 5.0 and 5.1.

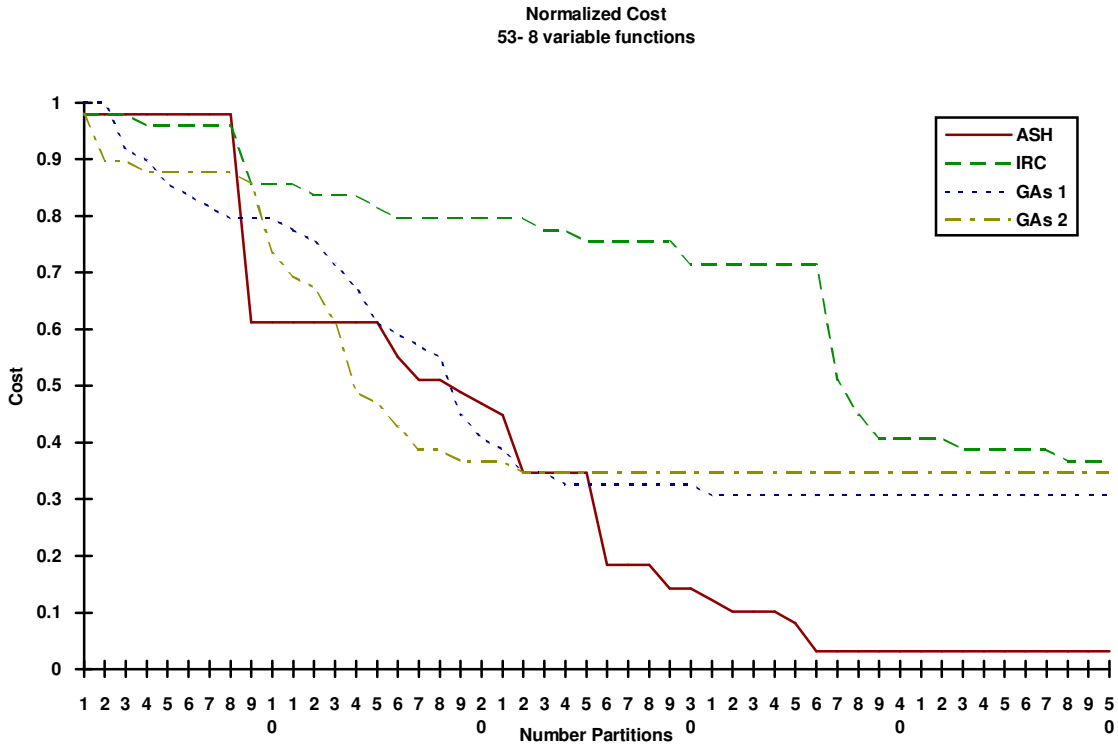


Fig 5.0

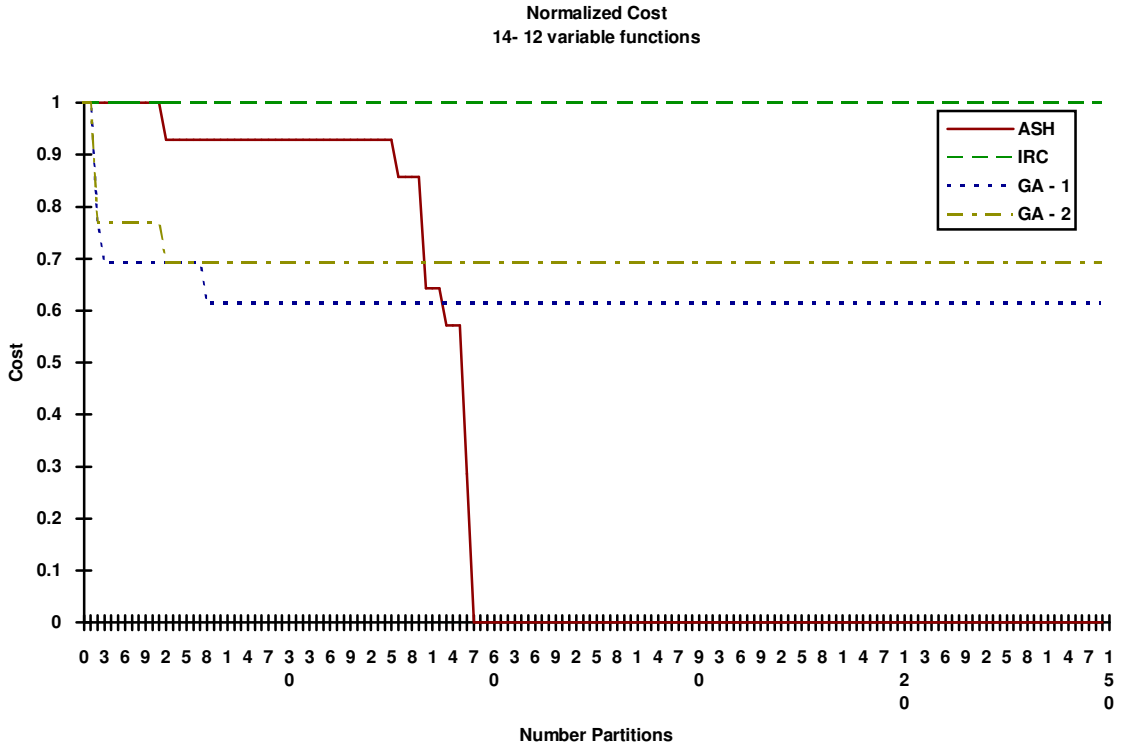


Fig 5.1

The two figures show the effectiveness and speed of ASH. In Fig 5.0, ASH doesn't find the known minimum two times, where in Fig 5.1 it finds the minimum in all 14 cases. The two genetic algorithms that are shown have a stopping condition. This implies that the algorithm will not do any better as the number of partitions increases. The reason for the stopping condition is that without it, the GAs would search all 2^N partitions, and the time that it would take to find new partitions would grow exponentially since it didn't want to reevaluate partitions. IRC in the two graphs will continue to evaluate all 2^N partitions, and so as the number of partitions increases its Normalized Error will approach zero. Thus, IRC will find the minimum partition in the figures shown above.

6.0 Conclusions

The paper proves that an algorithm can be successfully applied to an NP-complete problem of logic synthesis that has been traditionally treated as very difficult to solve. The results demonstrated that our ASH algorithm is able to reduce the search complexity to a polynomial complexity on pattern theory benchmarks.

Even though the algorithm may not find the minimum solution every time, we feel that this is a good start for actually understanding this area of functional decomposition. With more research into this area and other NP complete problems in decomposition, Ashenhurst-Curtis decomposition may be the best way of dealing with problems in machine learning and logic synthesis.

BIBLIOGRAPHY

[1] T. D. Ross et al. Logic Minimization as a Robust Pattern Finder. International Workshop on Logic Synthesis. Lake Tahoe, California. May 1993.

[2] S. Khuri. An Evolutionary Approach to Combinatorial Optimization Problems. Proceedings of CSC 1994. Phoenix Arizona. March 1994.

[3] T. D. Ross et al. On the decomposition of real-valued functions. 3rd International Workshop of Post-Binary VLSI Systems, International Symposium on Multiple Valued Logic. May 24, 1994.

[4] T. D. Ross et al. The Function Learning and Synthesis Hotbed. Wright Laboratory, WL/AART-2. WPAFB, OH 45433-6543. May 4, 1994.

[5] M. L. Axtell. Partition Selection Algorithms: Row/Column Ratio Experiment. Technical report, Veda Incorporated, c/o Wright Laboratory, WL/AART-2. WPAFB, OH 45433-7408. April 1994.

[6] M. J. Noviskey et al. Application of Genetic Algorithms to Functional Decomposition in Pattern Theory. Wright Laboratory, WL/AART-2. WPAFB, OH 4533-5543. Jan 1994.

[7] J. F. Frenzel. Application of Genetic Algorithms to Pattern Theory. Final report for summer faculty research program, AFOSR. Wright Laboratory, July 1993.

[8] T. D. Ross et al. Pattern Theory: An Engineering Paradigm for Algorithm Design. Final report for period Oct. 1988-Oct. 1990. Avionics Directorate, Wright Laboratory. WPAFB, OH 45433-6543. p 46.

[9] T. Luba, R. Lasocki. Decomposition of Multiple-Valued Boolean Functions. Applied Mathematics and Computer Science. Vol. 4 No.1 1994. pp 125-138.