EXACT GRAPH COLORING FOR FUNCTIONAL DECOMPOSITION: DO WE NEED IT?

Rahul Malvi, Marek Perkowski, Lech Jozwiak †
Portland State Univ., Dept. Electr. and Comp. Engng.,
Portland, OR, 97207-0751, USA, *mperkows@ee.pdx.edu*
† Faculty of Electronics Engineering, Eindhoven Univ. of Technology,
P.O.Box 513, 5600 MB Eindhoven, The Netherlands, *lech@eb.ele.tue.nl*

Abstract— Finding column multiplicity index is one of important component processes in functional decomposition of discrete functions for circuit design and especially Data Mining applications. How important it is to solve this problem exactly from the point of view of the minimum complexity of decomposition, and related to it error in Machine Learning type of applications? In order to investigate this problem we wrote two graph coloring programs: exact program EXOC and approximate program DOM (DOM can give provably exact results on some types of graphs). These programs were next incorporated into the multi-valued decomposer of functions and relations *MVGUD*. Extensive testing of *MVGUD* with these programs has been performed on various kinds of data. Based on these tests we demonstrated that exact graph coloring is not necessary for high-quality functional decomposers, especially in Data Mining applications, giving thus another argument that efficient and effective Machine Learning approach based on decomposition is possible.

1. INTRODUCTION

Functional Decomposition is used in many applications including FPGA mapping, custom VLSI design, regular arrays, Machine Learning, Data Mining and Knowledge Discovery in Data Bases (KDD). Unfortunately, many of the existing programs based on decomposition are slow, or, if they use heuristics for the speed of run, they find solutions that are much inferior to those found by more optimized decomposers. The question thus arises: *"how to create a decomposer that will be both effective and efficient?"*. To answer this question we investigated separately several issues related to this topic: - partitioning of variables to bound, free and shared sets [9], - creation of combined strategies that select point-oriented decomposition methods such as disjoint/nondisjoint, Ashenhurs/Curtis, serial/parallel, and other types of decompositions for any sub-function [9], - representations of data [2,3,10]; and in this paper we will concentrate only on the column minimization problem for a given bound set of input variables.

This problem affects considerably the overall success of any functional decomposer, because a high percentage of the decomposer's run time is spent on it [9,10,11]. We need an algorithm that is both fast and produces decomposed networks that have as small cost as possible. In case of circuits, the network cost relates to circuit's realization cost, area, or number of blocks; in Machine Learning and Data Minining applications the smaller cost directly corresponds to the reduced decision error (Occam Razor - [2,3]), so in both cases it is very important to obtain simple solution networks.

Here we want to find out what is the role of Column Minimization in the overall success of a decomposer; especially, in terms of the calculation time, the memory usage, and the quality of results. We want to investigate how the answers to these questions depend on the type of data, for instance on the percent of don't cares, or on the density of graphs in question. Presently the decomposer introduced by Pedram et al [6] achieves the best results in terms of data size; this program uses a set covering approach and does not assume many don't cares in functions, if any. This program is thus not useful for Machine Learning applications because it is not intended for types of functions with very many don't cares and other special characteristics of ML, KDD or Pattern Recognition benchmark sets. Several decomposers developed in the collaboration of Portland State University, Eindhoven University of Technology, and Technical University of Warsaw [2,3,10,11] achieve perhaps the best overall efficiency and efficacy on Machine Learning benchmarks, cube specified incomplete binary MCNC benchmarks and other binary functions.

There are basically four methods to find the Column Multiplicity in Functional Decomposition: *Set Covering,*

Graph Coloring, Clique Partitioning and *Clique Covering.* A relation between the columns of the Karnaugh map of a function, for a given bound and free sets can be represented as a Compatibility Graph or as an Incompatibility Graph. If represented as a Compatibility Graph, nodes which are connected together are compatible nodes and can be colored with the same color. This is called Clique Covering. If the graph is represented as an Incompatibility Graph then nodes which do not have a common edge can be colored with the same color. This is called Graph Coloring. It was shown to be efficient, especially for functions with high percent of don't cares [4,9,10,11]

Although few authors studied graph coloring in Functional Decomposition, nobody, to our knowledge, has compared these methods, or evaluated the importance of finding minimal solutions to the problem of Column Multiplicity in the Ashenhurst/Curtis Decomposition. In this paper we will compare two new graph coloring algorithms: exact (EXOC) and heuristic (DOM). DOM uses dominations to color the graph and is based on the approach that was first presented in [7,9]. The idea for the Exact Graph Coloring Algorithm(*EXOC*) is a modification of the one from [7] by more efficient implementation of backtracking.

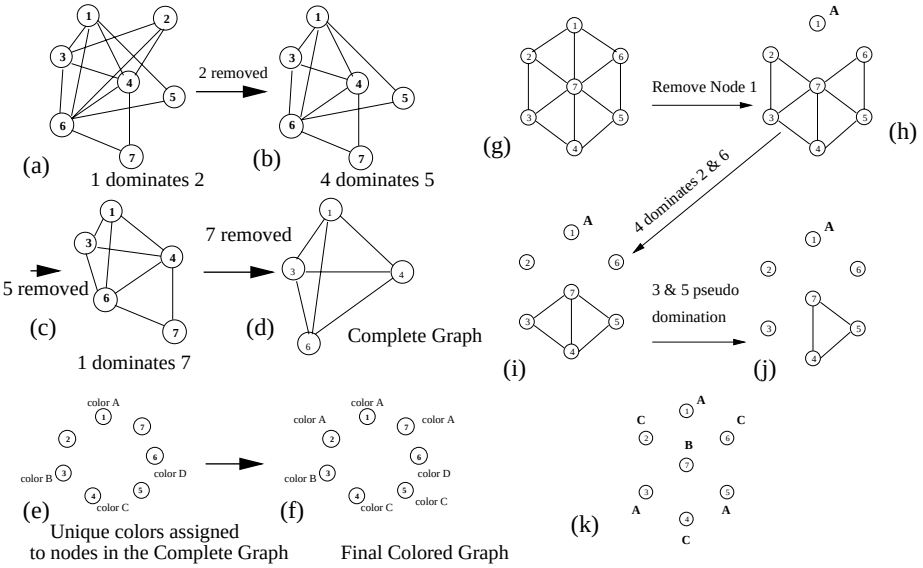

Fig. 1. *(a) - (f) Example showing how DOM colors a non cyclic graph, (g) - (k) Example showing how DOM colors a cyclic graph.*

2. DOM - NEW HEURISTIC GRAPH COLORING PROGRAM.

A node "A" in an incompatibility graph *dominates* some other node "B" in the graph if the following is satisfied: **1)** Node "A" and node "B" have no common edge. **2)** Node "A" has edges with all the nodes that node "B" has edges with. **3)** Node "A" has at least one more edge than node "B". In Fig. 1a, node "1" dominates node "2" since it satisfies the conditions for domination. When two nodes have a domination then both the nodes can be colored with the same color. If conditions 1) and 2) for dominations are satisfied and node "A" has the same number of edges as node "B", then it is called a *pseudo domination.* In Fig. 1i nodes 3 and 5 have a pseudo domination.

Theorem 1. If any node "A" in a graph dominates any other node "B" in the graph, node "B" can be removed from the graph, and in a pseudo domination any one of the nodes "A" or "B" can be removed.

A Complete graph is one in which all pairs of vertices are connected. In a complete graph, $total\_edges = \frac{nodes*(nodes-1)}{2}$, where $total\_edges$ is the sum of all the edges in the graph. In a Complete Graph no dominations or pseudo dominations can be found. In a Complete Graph all the nodes must have a unique color. A Complete Graph is a special case of a Cyclic Graph. A *Cyclic graph* is a graph that is not complete and has no dominated or pseudo-dominated node(s). All graphs from Fig. 1a-d are non-cyclic. The graph from Fig. 1g is cyclic.

Theorem 2. If a graph is not cyclic and can be reduced to a complete graph by successive removing of all its dominated and pseudodominated nodes, then Algorithm DOM finds the coloring with the minimum number of colors (the exact coloring).

Theorem 3. The graph created for SOP minimization of a non-cyclic single-output Boolean function using the method from [1,5,8,9] is not cyclic.

Thus, our approach, for either the SOP Minimization Problem, or the Column Minimization Problem in Functional Decomposition, is the following: For and arbitrary graph, we assume that the graph is not cyclic and we use our approximate algorithm. If it finds a solution without generating a cyclic graph, we know that this solution is exact. If a cyclic graph is generated, we have no proof of optimality, but still a good coloring is found if only few cyclic graphs were consecutively created in the process. Thus, if the characteristics of the graphs of some class is that a small number of cyclic graphs are created by DOM, this class is easily colorable by DOM. We are interested in classes of problems that are easily colorable by DOM. It can be shown that graphs created for two-level Sum-of-Products minimization are easily colorable, because DOM is equivalent to the algorithm based on finding essential primes and removing them, next finding secondary essential primes and removing them, and so on, until a cyclic remainder function is created [8]. On the other hand it was shown experimentally that most of SOP benchmarks are non-cyclic. In this paper we will show that a similar result is true for the functional decomposition graphs.

An Example showing how *DOM* colors a non cyclic graph. Here we illustrate use of the concept of dominations to color an incompatibility graph.
[1.] Fig. 1(a) shows an Incompatibility Graph. As can be seen Node 2 is dominated by Node 1, so in Fig. 1(b) Node 2 is removed and it is remembered that it was dominated by Node 1.
[2.] Next, in Fig. 1(b) Node 5 is removed as it is dominated by Node 4, and it is remembered that Node 4 dominates Node 5.
[3.] Then in Fig. 1(c), it can be seen that Node 7 is dominated by both nodes 1 and 3, the choice made is the first node which is Node 1, and Node 7 is removed. It is now remembered that Node 1 now dominates Node 2 and Node 7.
[4.] After removing Node 7 the resulting graph shown in Fig. 1(d) is a complete graph, so go to Step 5. In a complete graph each node is connected to all the other nodes, each node in the complete graph must have a unique color.
[5.] In Fig. 1(e), each node in the Complete Graph is given a unique color.
[6.] Finally in the last step in Fig. 1(f) the dominated nodes are colored with the same color as the dominating node. The color assignments are: Color A {1, 2, 7}, Color B {3}, Color C {4, 5}, Color D {6 }. Thus in this way the graph is colored. Fig. 1(e) shows the completely colored graph. Four colors were used which is the minimum required for this graph. **(end)**. As we see, in this example an exact solution was found without backtracking. This corresponds to the type of graphs that are created for non-cyclic Boolean functions in SOP minimization [8].

An Example showing how *DOM* colors a cyclic Graph. **[1.]** An incompatibility graph is shown in Fig. 1(g), as can be seen this graph has cycles. **[2.]** As the first step the graph is checked for dominations, but no dominations are found in this graph, so the first node is removed from the graph, which is node 1, and it is assigned a minimum possible color which in this case is color A. **[3.]** This results in a new graph, shown in Fig. 1(h). In this graph node 4 dominates node 2 and node 6. So node 2 and node 6 are removed from the graph, and it is remembered that node 4 dominates node 2 and node 6. **[4.]** On removing node 2 and node 6, in the resulting graph shown in Fig. 1(i) nodes 3 and 5 have a pseudo domination so the first one of these nodes which is node 3 is removed, and then node 4, 5, and 7 form a complete graph. The complete graph is shown in Fig. 1(j). **[5.]** Now nodes are colored with the minimum possible color, and each dominated node is given the same color as the node which dominated it. The coloring is shown in Fig. 1(k). Three colors were used to color the graph, which is the minimum required for this graph. The color assignments are: Color A {1, 3, 5}, Color B {7}, Color C {2, 4, 6}. **(end)**. As we see, in this example no proof of exact solution can be given but only few consecutive graphs (here, only the initial graph) were cyclic, so the solution is of a good quality. Such "cyclic graphs" are created for cyclic Boolean functions in SOP minimization [8], and this explains where is their name coming from.

3. EXOC - A NEW APPROACH TO EXACT GRAPH COLORING

Since Graph Coloring is an NP Complete problem, in general, nearly all possible solutions have to be evaluated to find the exact minimum solution. The algorithm used here for the exact Graph Coloring is a greedy algorithm with backtracking and cut-off. *EXOC* uses a tree search in order to color the graph, and colors successively nodes with an actually available color of a smallest number, remembering for each node all the remaining possibilities of coloring (it is assumed that initially the set of colors has as many elements as the set of nodes). The chromatic number of a graph is defined as the number of colors in the exact solution. *EXOC* stands for *Exact One Child*; it uses a "depth-first with one child" strategy. In this strategy at every stage only one branch of the tree is

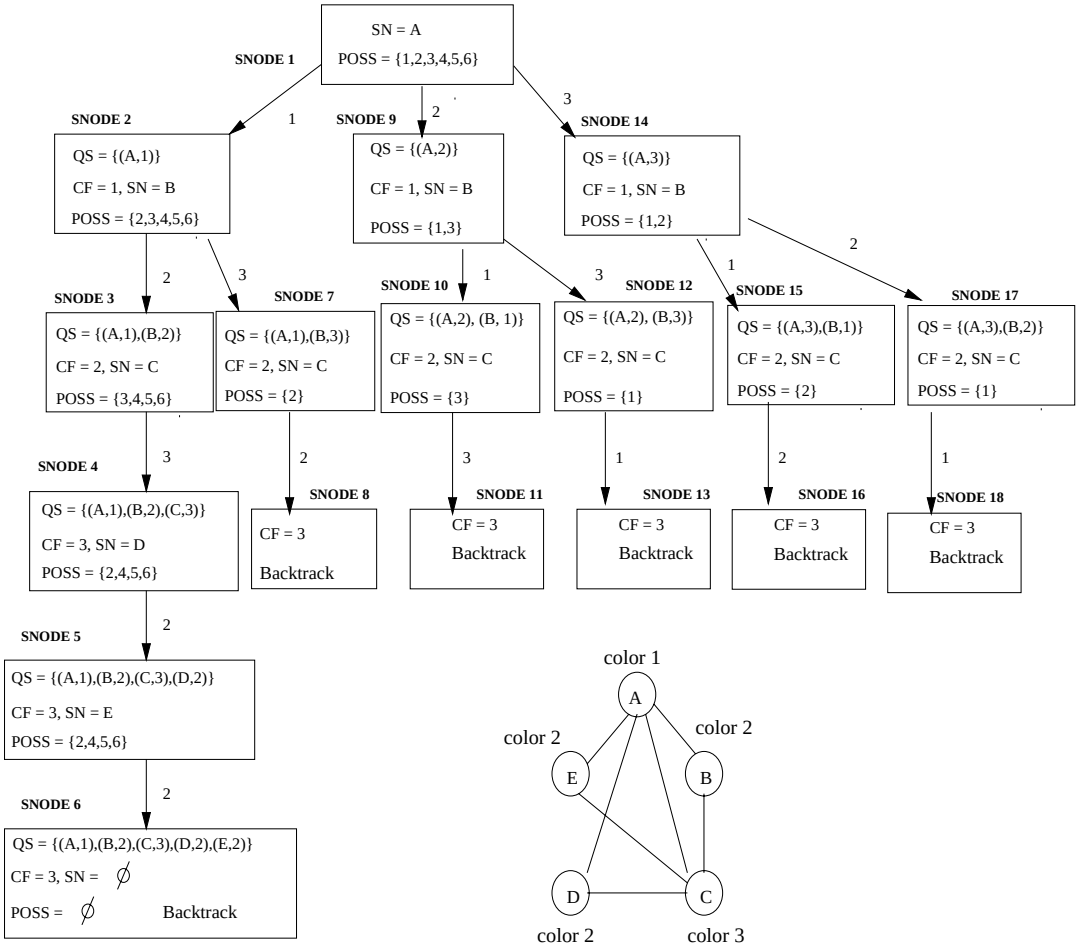

Fig. 2. *Tree Search for the Exact Graph Coloring Algorithm*

Comparison of *EXOC*, *DOM* and *CLIP* on MCNC Benchmarks											
Benchmark	in	out	cubes	option	DFC	nu of blocks	Av Edge%	T(s) (sec)	NP	TC	AC
5xp1	7	10	143	*EXOC*	344	17	62.75	2006	28	123	4.4
				CLIP	344	17		29.5	28	123	4.4
				DOM	344	17		29.9	28	123	4.4
9sym1	9	1	158	*EXOC*	96	3	-	108	11	54	4.9
				CLIP	96	3		55.2	10	52	5.2
				DOM	64	3		47.3	11	54	4.9
b12	15	9	172	*EXOC*	284	25	14.54	87	130	389	3
				CLIP	284	25		57.1	132	387	2.9
				DOM	284	25		46.4	130	389	3
bw	5	28	97	*EXOC*	560	56	55	51	115	361	3.14
				CLIP	560	56		50.9	115	361	3.14
				DOM	560	56		48.7	115	361	3.14
ex5p	8	63	214	*EXOC*	-	-	-	-	-	-	-
				CLIP	2472	186		565	8	35	4.4
				DOM	2472	186		516	8	35	4.4
misex1	8	7	40	*EXOC*	400	19	60.47	318	589	1831	3.1
				CLIP	388	19		24.7	587	1816	3.1
				DOM	400	19		22.5	589	1831	3.1
rd53	5	3	63	*EXOC*	80	6	63.5	5.4	8	26	3.25
				CLIP	80	6		4.7	8	26	3.25
				DOM	80	6		4.9	8	26	3.25
rd73	7	3	274	*EXOC*	160	6	71.6	46.6	10	38	3.8
				CLIP	160	6		41.6	10	38	3.8
				DOM	160	6		47.2	10	38	3.8
rd84	8	4	515	*EXOC*	220	12	-	189	23	75	3.3
				CLIP	220	12		127	24	80	3.3
				DOM	208	12		131	24	75	3.3
sao2	10	4	133	*EXOC*	416	12	67	423.8	52	337	6.5
				CLIP	416	12		124	52	337	6.5
				DOM	416	12		121	52	337	6.5
squar5	5	8	56	*EXOC*	200	16	58.6	124.8	31	94	3
				CLIP	200	16		10.2	31	94	3
				DOM	200	16		10.5	31	94	3
xor5	5	1	32	*EXOC*	20	2	53	1.1	2	4	2
				CLIP	20	2		1.2	2	4	2
				DOM	20	2		1.0	2	4	2

Table 1

A Comparison of results obtained by running MVGUD *with 4 variables in the Bound Set on MCNC Benchmarks*

Comparison of *EXOC*, *DOM* and *CLIP* on Machine Learning Benchmarks										
Function		**Algorithm**								
		CLIP			*EXOC*			*DOM*		
name	cubes	DFC	nu of blocks	T(s)	DFC	nu of blocks	T(s)	DFC	nu of blocks	T(s)
add0	77	28	5	12.5	28	5	24.7	28	5	12.8
add2	77	28	6	12.9	28	5	13.1	28	5	13.1
add4	77	10	3	7.3	10	3	7.1	10	3	7.6
and_or_chain8	77	22	6	12	22	6	11.9	22	6	12.5
ch15f0	77	56	6	20.4	68	6	417	68	6	24.6
ch176f0	77	14	4	11.7	22	6	13	22	6	14
ch177f0	77	6	2	6.3	6	2	6.3	6	2	6.7
ch22f0	77	28	5	12	28	5	12.3	28	5	13.4
ch30f0	77	40	5	16.4	40	5	15.8	40	5	17.1
ch47f0	77	28	5	14.3	28	5	15	28	5	16.4
ch52f4	77	64	5	18.5	76	5	19.8	76	5	20.4
ch70f3	77	28	6	14.6	28	6	14.6	28	6	15.8
ch74f1	77	52	6	15	44	5	19.5	52	6	15.5
ch83f2	77	64	5	19.5	64	4	25.7	64	4	22.6
ch8f0	77	22	5	13.8	22	5	13.8	22	5	15.6
contains_4_one	77	28	5	19.5	44	4	23.1	44	4	24.1
greater_than	77	28	7	13.3	28	7	13.1	28	7	14.6
interval1	77	44	4	20.5	56	5	20.3	56	5	21.9
interval2	77	64	4	17.2	64	4	21.7	64	4	18.2
kdd1	77	22	6	8.9	16	5	8.4	16	5	9.7
kdd2	77	22	6	12.9	22	6	12.3	22	6	14
kdd3	77	14	4	7.8	14	4	7.2	14	4	8.9
kdd4	77	2	1	6.4	2	1	5.9	2	1	6.9
kdd5	77	28	6	12.3	28	6	13.5	28	6	14.4
kdd6	77	14	4	8.3	14	4	8.3	14	4	9.2
kdd7	77	28	7	10.5	28	5	14.2	28	5	15.4
kdd8	77	14	4	7.1	14	4	6.5	14	4	7.6
kdd9	77	28	6	11.7	28	7	11.7	28	7	12.9
majority_gate	77	28	6	14.5	40	5	16.6	40	5	17.2
modulus2	77	28	6	17	28	5	17.3	28	5	18.8
monkish1	77	14	4	8.2	14	4	8.3	14	4	9
monkish2	77	28	6	13.9	28	6	13.6	28	6	15
monkish3	77	22	5	9.7	22	5	9.5	22	5	10.7
mux8	77	46	5	12.8	46	5	12.2	46	5	13.8
nnr1	77	64	5	18.5	64	5	19.7	64	5	19.1
nnr2	77	14	3	6.7	14	3	6.7	14	3	7.8
nnr3	77	88	5	22.6	68	4	33.2	68	4	26
or_and_chain8	77	22	6	9	22	6	9.2	22	6	10.3
pal	77	22	5	13.9	22	5	13.4	22	5	14.6
pal_dbl_output	77	92	5	23.4	92	6	29.1	92	5	25
parity	77	28	7	11	28	7	11.3	28	7	12.8
primes8	77	68	5	23.3	68	4	26.4	68	4	24.6
remainder2	77	68	4	23.9	68	4	29.8	68	4	25.1
rnd1	77	152	5	23.5	152	5	110	152	5	23.2
rnd2	77	88	4	23	88	4	187	88	4	24.9
rnd3	77	88	5	20.7	88	5	33.8	88	4	26.1

Table 2
A Comparison of results obtained by running MVGUD *with 2 variables in the Bound Set on 8 variable Machine Learning Benchmarks*

generated, and after finding a solution in a new branch, the solution that is better than a previous one, *EXOC* has a new, improved evaluation of the chromatic number. Thus, whenever a new solution is found, any color greater than the best solution is removed from the possible color choices of the other nodes. This best solution is used as the cut-off criterium. In any branch if the number of colors used is greater than or equal to the best solution, then *EXOC* does not proceed along that path and cuts it off.

The difference between the "depth-first with one child" and a "depth-first search" is that in a "depth-first search" all the branches of the tree are created, all these branches are stored in memory, and each branch is followed until a leaf node of the tree is reached. But in the "depth-first with one child" whenever a new branch of the tree is created, only if its solution is better than the previous solution, is the new branch kept in memory and the old branch deleted from it. Also in the "depth-first with one child" strategy, because there is a cut-off, *EXOC* does not go down to the leaf nodes of each branch. *EXOC* was written in ANSI C on a SUN SPARC workstation.

Theorem 4. The number of colors generated for an incompatibility graph by *EXOC* will always be equal to the chromatic number of the graph.

An Example showing how EXOC colors a graph. Fig. 2 is an example which shows how *EXOC* colors a graph. In this example the incompatibility graph being colored is a non cyclic graph, but the algorithm for *EXOC* does not consider if the graph is cyclic or not, it uses the same method to color a graph independent of whether the graph is cyclic or not. Since *EXOC* deals with stack nodes and with nodes of the graph, in each step given below *SNODE* refers to a node of the STACK and *GNODE* refers to a node of the incompatibility graph. The search process is executed as follows. **[1]** Select *GNODE* "A"and find the possible colors for it and push the *SNODE* labeled SNODE 1 in Fig. 2 on to the stack. Selected color for *GNODE* "A" is 1. **[2]** Now select a new *GNODE* which is node "B". *GNODE* "B" can be given the possible colors $2, 3, 4, 5, 6$. Store this information on *SNODE* labeled SNODE 2 in Fig. 2. Selected color is 2. **[3]** Now *GNODE* "C" becomes the selected node and it can obtain one of colors {3,4,5,6}. Selected color is 3. **[4]** In this way, the first branch of the

tree from Fig. 2 is created. After finding the first solution: color 1 = {A}, color 2 = {B,D,E}, color 3 = {C}. we know that three colors are enough, and these colors are 1, 2 and 3. **[5]** All non-used colors are then removed from sets $POSS$ of all previous nodes. Now backtrack, and at each stage check the $POSS$ colors of the node. If there is a $POSS$ color then go along the new path, else ignore and continue backtracking. **[6]** On reaching $SNODE$ 2 there is a $POSS$ color = 3. Select this color for $GNODE$ B, and go to SNODE 7. **[7]** IN $SNODE$ 7, $GNODE$ "C" is the selected node, and it has only one possible color, which is color 2. Select color 2, for $GNODE$ "C" thus reaching $SNODE$ 8. **[8]** In $SNODE$ 8, $CF = 3$, which is the same as the best solution, which indicates that any solution along this path will at most result in a solution equal to the best solution, but not better. So cutoff and backtrack. **[9]** Ultimately on reaching $SNODE$ 1, select color 2 for $GNODE$ "A" and go along that path. On reaching $SNODE$ 11 the CF is 3, so do not proceed any more along this path, and cut off here and backtrack. **[10]** Fig. 2 shows all the paths of the tree that are traversed. The $SNODEs$ are labeled in the order in which they are visited. **[11]** If at any SNODE a solution is obtained that is better than the one obtained at SNODE 6, store the new solution and discard the old solution. Continue till all the possible paths of the tree have been traversed. The solution saved is the minimum coloring of the graph.

Comparison of different Strategies of finding the Column Multiplicity on Machine Learning Benchmarks.

To see if the same results were obtained on Machine Learning Benchmarks, *MVGUD* was also tested on Machine Learning Benchmarks. These Benchmarks were obtained from the Wright Labs Database. These are completely specified functions with 8 and 12 variables in the input and one output. Since we were interested in don't-cares in the function the program *FLASH* from Wright Labs was used to convert the above functions into functions with 70% of don't cares. The following Tables illustrate the results obtained. *MVGUD* was run in the same way as before, with 2, 4 and 5 variables in the Bound Set.

Table 2 shows the results of running *MVGUD* on Machine Learning Benchmarks(MLB). These MLB have 8 variables in the Bound Set. In these tables they have 70% of don't cares (cubes = 30% * 256). Table 2 is a result of running *MVGUD* with 2 variables in the Bound Set.

On examining these Tables it was seen that here too *EXOC* fails to improve the quality of Decomposition. The results with all three algorithms prove to be nearly the same, with slight differences in some cases. Testing was not done on Bound Sets greater than 5 because for Bound Sets greater than 5, *EXOC* is too slow to be practical. Testing was also done to compare the total count of colors generated during the process of decomposition, but here too it was found that the number of times the algorithms for calculating the column multiplicity is called varies for the same function. Hence these Tables are not included here.

4. COMPARISON OF EXOC AND DOM ON COLUMN MULTIPLICITY

In this section, we will evaluate the importance of an Exact Graph Coloring in Curtis Decompositions. Our aim is to investigate if an Exact Graph Coloring is required in Functional Decomposition and if it leads to better results on the graphs that are created from practical function benchmarks. Or, is DOM sufficient? We used the Decomposer Multi-Valued General Universal Decomposer*MVGUD* written at Portland State University for the testing purpose. We instantiated three algorithms into *MVGUD*, a Greedy Clique Partitioning (*CLIP*), the Dominance Graph Coloring (*DOM*) and the Exact Graph Coloring(*EXOC*). The decomposer was run with different numbers of variables in the Bound Set on two kinds of benchmarks: *MCNC* benchmarks for circuits (presented below), and Machine Learning Benchmarks (from the Wright Labs Database) for data from Machine Learning, Pattern Recognition and Knowledge Discovery in Data Bases.

A comparison of *DOM* and *EXOC* was first done on randomly generated graphs, for varying number of nodes and varying percentage of edges (not shown because lack of space). Conclusions were reached about how well *DOM* and *EXOC* will perform on the different kinds of graphs. Tests were done to characterize the kind of graphs that are generated in decomposition with regard to the number of nodes in the graph and the percentage of edges in the graph in order to see if the same conclusions hold for the graphs generated during Functional Decomposition $F = H(G(bound\_set),\ free\_set)$.

Since *MVGUD* is a multi-valued decomposer, it has no encoding stage. Essentially *MVGUD* looks for the Curtis Binary Decomposition criteria in evaluating if a decomposition is acceptable, but then creates the output signal of the "G" function with as many values as the μ_{min} (minimal column multiplicity index) found. This approach results in one multi-valued output from each "G" block. It can be called *a multi-valued Ashenhurst Decomposition* [4,10]. Whether the method used by *MVGUD* to calculate *DFC* (a measure of multi-valued function complexity from [4,9,10] that is close to the equivalent total number of two-input gates in the circuit), is a good evaluation of the cost of the decomposed multi-valued blocks is not discussed here, but since the *DFC* is used for a comparison between different methods of calculating the Column Multiplicity in Decomposition, within the same decomposer, the method of calculation of the *DFC* does not matter for the purpose of evaluating algorithms

for calculating column multiplicity. What matters is that the same method is used for all the algorithms that are compared. Here a comparison is done between *DOM*, *EXOC* and *CLIP*. The goal of this testing is to see if an Exact Graph Coloring is necessary to calculate the Column Multiplicity in Functional Decomposition, and if the *DFC* can be improved in case that *MVGUD* is run with *EXOC*, in comparison to when it is run with *DOM* or with *CLIP*. *MVGUD* was tested with two, four, and five variables in the Bound set.

Notations Used in the Tables. The following is an explanation of the Notations used in the Tables in this section: **Benchmark** : Name of the Benchmark function; **in** : Number of inputs of the Benchmark; **out** : Number of outputs of the Benchmark; **cubes** : Number of cubes in the Benchmark; **DFC** : Decomposed function cardinality of the decomposed function; **Algorithm** : Name of Algorithm used in *MVGUD*; **Nu of Blocks** : Number of multi-valued blocks in the decomposed function; **NP** : *Number of passes*, or number of times the function to calculate the column multiplicity was called; **TC** : *Total Colors*, iterative sum of colors generated for each pass; **AC** : *Average Colors* = TC/NP; **T(s)** : User time in seconds.

A Comparison of the different Strategies of finding the Column Multiplicity on MCNC Benchmarks.

MVGUD was run with 4 variables and with 5 variables in the Bound Set (only a small sample of results is shown here). In Table 1 *Av Edge%* was calculated to see how dense or sparse the graphs generated during the decomposition are. This was calculated in the following way: For any graph with number of nodes = n, the *total_possible_edges* for this graph(100% edges) will be equal to $n * (n-1)/2$. Hence if the number of edges in the graph is equal to e, then the $edge\_percent = (e * 100)/total\_possible\_edges$. This will give the *edge_percent* in a graph with n nodes and e edges. Since the decomposer calls the function to calculate the Column Multiplicity a number of times, the *Av Edge%* was calculated by adding the *edge_percent* for a graph each time the function to find Column Multiplicity was called, and then dividing this total by the number of times the function to calculate the Column Multiplicity was called.

Looking at the results it is seen that *EXOC*, *DOM* and *CLIP* generate the same results in all the cases in terms of DFC and number of CLB's.

The reason for the slow times of *MVGUD* with *EXOC* can be explained as follows: when *MVGUD* is run with 5 variables in the Bound set, in most cases the average number of nodes in the graph is 32 and the edge percentage is always high with the highest being 77% and the lowest being 46.1%. This means that the graphs generated during decomposition were nearly always (since this is an average) dense graphs. It was found experimentally on random graphs that for dense graphs *EXOC* takes a long time to find the Exact solution, hence we have such slow times for *EXOC*. Whenever *DOM* does not generate an exact solution, it is usually 1 or 2 colors away from the Exact solution and rarely more than that, and this being on randomly generated graphs. Now considering that there were 5 variables in the Bound Set, then the Incompatibility graph will have 32 nodes, and for a Curtis decomposition to exist, if a coloring of the graph with 16 colors or less is found then one exists.

For 5 variables in Bound Set in the column for Average colors *AC* it can be seen that the largest average color is 7.65 for the benchmark *sao2*. But this means that these graphs generated during decomposition, had low chromatic numbers, which were much less than 16. So even if *DOM* or *CLIP* generate a solution that is 2 or 3 colors away, the solution will be accepted as a Curtis Decomposition because it will still be less than 16. The same in Table 1 where a comparison is made with 4 variables in the Bound Set. Hence we conclude that for 4 or 5 or greater number of variables in the Bound Set an Exact Graph Coloring does not produce better Curtis Decompositions, and having a good heuristic algorithm to find the Column Multiplicity or even a greedy algorithm to find the Column Multiplicity is good enough.

A Summary of the Results Obtained from Testing on Machine Learning Benchmarks.

As can be seen from Table 2 *EXOC* was unable to provide a better *DFC* for the Machine Learning Benchmarks. In order to see the total numbers of colors generated by *DOM*, *EXOC* and *CLIP* on the same graphs, which were generated during the process of Functional Decomposition, the following experiment was performed: *MVGUD* was made to run with all three algorithms *EXOC*, *CLIP*, and *DOM* calculating the Column Multiplicity, and only the results of one of them was accepted and the results from the other two was discarded. The count of the colors was kept for all three Algorithms, thus demonstrating how *EXOC*, *CLIP*, and *DOM* compare with respect to the total number of colors generated on the same graphs, only now these graphs have been generated from practical function Benchmarks. Table 3 shows the result of this comparison.

Table 3 shows the results of running *MVGUD* with all three algorithms, *DOM*, *EXOC*, and *CLIP* on the same graphs which were generated during decomposition. Table 4 is a summary of the results of Table 3. Table 4 shows how *DOM* and *CLIP* compare with respect to the number of times that the total number of colors generated by *DOM* and *CLIP* are the same as the total number of colors generated by *EXOC*, and the number of times the total colors generated by *DOM* and *CLIP* were not exact and by how much.

In Table 4, the row *Exact* stands for the case when the total numbers of colors generated by *DOM* and *CLIP* was the same as the total colors generated by *EXOC*. *Error 1* stands for the case in which the total numbers of

Comparison of the Total Colors generated on Graphs from Machine Learning 8 variable Benchmarks									
Benchmark	Total Colors								
	2 variables in Bound			4 variables in Bound			5 variables in Bound		
	DOM	*EXOC*	*CLIP*	*DOM*	*EXOC*	*CLIP*	*DOM*	*EXOC*	*CLIP*
add0	79	79	79	36	36	**38**	15	15	15
add2	90	90	90	**46**	45	**48**	21	21	**24**
add4	20	20	20	13	13	13	12	12	**14**
and_or_chain8	69	69	69	21	21	**22**	17	17	**18**
ch15f0	156	156	**158**	36	36	**38**	**27**	23	**24**
ch176f0	35	35	**36**	22	22	**23**	15	15	15
ch177f0	15	15	15	17	17	17	10	10	**11**
ch22f0	61	61	61	18	18	18	15	15	15
ch30f0	126	126	126	33	33	33	22	22	**23**
ch47f0	111	111	111	33	33	33	27	26	**27**
ch52f4	141	141	**142**	37	37	**40**	24	24	**25**
ch70f3	73	73	73	28	28	28	20	20	20
ch74f1	105	105	105	26	26	**29**	25	25	25
ch83f2	123	123	123	44	44	**45**	28	28	**29**
ch8f0	90	90	90	21	21	21	17	17	**18**
contains_4_ones	126	126	126	32	32	**34**	24	23	**25**
greater_than	100	100	100	35	35	35	20	20	**21**
interval1	134	134	**135**	31	31	**32**	**18**	17	**19**
interval2	123	123	**124**	35	35	**36**	30	30	**32**
kdd1	46	46	46	19	19	19	12	12	12
kdd2	58	58	58	18	18	18	12	12	12
kdd3	38	38	38	23	23	23	13	13	13
kdd4	16	16	16	7	7	7	3	3	3
kdd5	89	89	89	31	31	**32**	20	20	**24**
kdd6	29	29	29	9	9	9	12	12	12
kdd7	59	59	59	24	24	**25**	**21**	20	**21**
kdd8	30	30	30	11	11	11	13	13	**14**
kdd9	77	77	77	27	27	27	18	18	**20**
majority_gate	94	94	94	32	32	32	21	21	**23**
modulus2	118	118	**119**	36	36	**39**	20	20	**22**
monkish1	31	31	31	14	14	**15**	**12**	11	**13**
monkish2	84	84	**85**	34	34	**37**	23	23	**25**
monkish3	73	73	73	28	28	**29**	17	17	17
mux8	102	102	102	38	38	38	19	19	**22**
nnr1	131	131	**134**	27	27	**30**	26	26	**30**
nnr2	36	36	36	11	11	11	15	15	15
nnr3	187	187	**189**	42	42	**45**	31	31	**34**
or_and_chain8	56	56	56	24	24	24	15	15	**16**
pal	74	74	74	23	23	**24**	14	14	14
pal_dbl_output	219	219	**220**	43	43	**46**	**33**	31	**37**
parity	46	46	46	18	18	**19**	12	12	**14**
primes8	124	124	124	36	36	**37**	27	27	**28**
remainder2	157	157	**159**	34	34	**35**	24	24	**29**
rnd1	205	205	**207**	49	49	**50**	34	34	**39**
rnd2	215	215	**216**	54	54	**56**	37	37	**42**
rnd3	156	156	**160**	52	52	**54**	33	32	**34**

Table 3

A Comparison of the Total Colors obtained by running MVGUD *on Machine Learning Benchmarks*

Summary of Results of Comparison of the Total Colors												
Total Number of Program Runs = 46												
	DOM						*CLIP*					
	Bound = 2		Bound = 4		Bound = 5		Bound = 2		Bound = 4		Bound = 5	
	Nu	%	Nu	%	Nu	%	Nu	%	Nu	%	Nu	%
Exact	46	100	45	97.8	41	89.1	32	66.1	20	43.5	14	30.5
Error 1	-	-	1	2.1	3	6.5	8	17.4	13	28.3	11	23.9
Error 2	-	-	-	-	1	2.1	4	8.6	5	10.8	12	26.1
Error 3	-	-	-	-	-	-	1	2.1	8	17.4	3	6.5
Error 4	-	-	-	-	1	2.1	1	2.1	-	-	3	6.5
Error 5	-	-	-	-	-	-	-	-	-	-	2	4.3
Error 6	-	-	-	-	-	-	-	-	-	-	1	2.1

Table 4

A Comparison of Total Colors generated by DOM, *and* CLIP *compared with total colors generated by* EXOC *on the same graphs for two, four and five variables in the Bound Set for Machine Learning Benchmarks*

Sum of Table 4				
Total Number of Program Runs = 138				
	DOM		*CLIP*	
	Total Number	Total %	Total Number	Total %
Exact	132	95.6	66	47.8
Error 1	4	2.8	33	23.9
Error 2	1	0.7	21	15.2
Error 3	-	-	12	8.7
Error 4	1	0.7	4	2.8
Error 5	-	-	2	1.4
Error 6	-	-	1	0.7

Table 5

A Summary showing the Addition of the Total Colors obtained in Table 4

colors generated by *DOM* and *CLIP* were one color away from the total numbers of colors generated by *EXOC*, and so on, till *Error 6*. Corresponding to these rows, the column *Nu* gives the number of times, and column *%* is equal to $Nu/TotalNumberofProgramRuns * 100$. As can be seen from the Table 4, *DOM* performs extremely well, and *CLIP* does not perform so well. *DOM* thus proves to be a very good heuristic algorithm. Table 5 is a total of the rows of Table 4 for *DOM* and *CLIP*.

5. CONCLUSIONS.

Functional Decomposition has always been a very complex problem, and all the research done until now just tried to solve the problem more efficiently, but without trying to reason **why** Functional Decomposition is such a difficult problem. Here we have investigated only the part of Functional Decomposition which involves finding the Column Multiplicity, and we did not show all our experimental results that support our conclusions. But, **the results obtained here provide a deep insight into the Column Multiplicity part of Functional Decomposition.**

By the results of the testing we can definitely say that we have proved that an Exact Graph Coloring is not required to find the Column Multiplicity where Curtis Decompositions are considered. Exact Graph Colorings only take up more time and fails to produce any significant change in the results. This is true with respect to MCNC Circuit Benchmarks, Wright Labs Machine Learning Benchmarks, as well as other Machine Learning and Circuit benchmarks not shown here.

Also the results shown raise the question that in cases where *CLIP* did not generate the same total numbers of colors as *EXOC*, why did the *DFC* not improve when we used *EXOC*? The only possible answer to this question is that the decompositions generated by *CLIP* were still acceptable decompositions, even if they use non minimum numbers of colors which in turn means that these graphs generated during the decomposition process must be having low chromatic numbers. This provides a very valuable insight into the kinds of graphs that are generated during the decomposition process.

Our experiments clearly demonstrated that the graphs generated during the decomposition process are much simpler than graphs generated randomly. (For the lack of space the analysis of random graphs coloring is not discussed here). This was another important conclusion, because it provided us with a deeper insight into the entire decomposition process, and not only to the part of finding the Column Multiplicity. **With our detailed experimentation we proved that the DOM program itself is sufficient for column minimization to create an efficient and effective decomposer for large functions.**

References

[1] M.J. Ciesielski, S. Yang, and M.A. Perkowski, "Multiple-Valued Minimization Based on Graph Coloring," . *Proc. ICCD'89*, pp. 262 - 265, Oct. 1989.

[2] C. Files, R. Drechsler, and M. Perkowski, "Functional Decomposition of MVL Functions using Multi-Valued Decision Diagrams," *Proc. ISMVL'97*, May 1997, pp. 27 - 32.

[3] C. Files, and M. Perkowski, "An Error Reducing Approach to Machine Learning using Multi-Valued Functional Decomposition," *Proc. ISMVL'98.*

[4] S. Grygiel, M. Perkowski, M. Marek-Sadowska, T. Luba, L. Jozwiak, "Cube Diagram Bundles: A New Representation of Strongly Unspecified Multiple-Valued Functions and Relations," *Proc. ISMVL'97.*

[5] L. Nguyen, M. Perkowski, and N. Goldstein, "PALMINI - Fast Boolean Minimizer for Personal Computers," *Proc. DAC'87*, pp. 615 - 621.

[6] Y.T. Lai, M. Pedram, S. Sastry, "BDD-based Decomposition of Logic Functions With Application to FPGA Synthesis," *Proc of 30th DAC*, pp. 642-647, 1993.

[7] M.A. Perkowski, "A Method of Solving Combinatorial Problems in Automatic Design of Digital Circuits," *Ph.D. Thesis,* Warsaw Technical University, 1980.

[8] M. A. Perkowski, P. Wu, K. A. Pirkl, "KUAI-EXACT: A New Approach for Multi-Valued Logic Minimization in VLSI Synthesis," *Proc. ISCAS'89*, pp. 401 - 404.

[9] M.A. Perkowski, "A New Representation of Strongly Unspecified Switching Functions and it Application to Multi-Level AND/OR/EXOR Synthesis," *Proc. of RM'95,* pp. 143-151.

[10] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, M. Nowicka, R. Malvi, Z. Wang, J. Zhang, "Decomposition of Multiple-Valued Relations" *Proc. ISMVL'97*, pp. 13 - 18.

[11] P. Sapiecha, M. Perkowski, and T. Luba, "Decomposition of Information Systems Based on Graph Coloring Heuristics," *Proc. CESA'96 IMACS*, Lille, France.