Development of Search Strategies for MULTIS

Marek Perkowski, Michael Burns, Tadeusz Luba, Stanislaw Grygiel,
Cynthia Stanley, Robert Price, Zhi Wang, Jing Lu, Paul Burkey,
Dhinesh Manoharan, and Sanof Mohammad

Portland State University, Dept. of Electrical Engineering
Portland, Oregon 97207
Tel: 503-725-5411, Fax: 503-725-4882
e-mail: mperkows@ee.pdx.edu

December 29, 1995

Abstract

Although every Functional Decomposer program from the literature uses certain strategy for finding bound and free sets of variables (variable partitioning), and/or selecting various partial decomposition processes or auxiliary decomposition subroutines, there is nothing published on comparing the different decomposition strategies.

By general search strategies we understand programmed methods, like for instance deciding whether to execute a decomposition (and which type of), or to continue looking for a better bound set.

Using our decomposer MULTIS we found that the general search strategies of the decomposer influence its cost/speed tradeoff more than any other of its single components, such as the encoding algorithm or the column minimization algorithm.

In this report we present various search strategies for MULTIS, and we also propose how the code developer should create even more improved strategies of such type. We propose also ways to create good heuristic search strategies for partial combinatorial problems solved by the decomposer, such as the set covering or the graph coloring problems.

Contents

List of Figures

1 INTRODUCTION

It is our experience gained from experiments with MULTIS until now, that one of the most important components to create a successful multi-level functional decomposer is the *general strategy of decomposition*. By the general search strategy we understand the following:

1. how to select bound sets, free sets and sets of repeated variables at every decomposition step.
2. what types of decompositions to select at every step (serial, parallel, approximate serial, gate type decompositions, etc.)?
3. to which sub-functions (i.e., intermediate, multi-input, multi-output logic blocks) the next decomposition step should be applied?
4. what data to create for Column Minimization, Encoding, and Parallel Decomposition problems?

In principle, better search methods usually on one hand use some kind of heuristics, and on the other hand, utilize some **systematic** search strategies, that **guarantee**, at least local, optima. One convenient and popular way of describing such strategies is to use the concepts of **tree searching.**

In MULTIS, we have several problems that require tree searching. First, we have the entire strategy of selecting special kinds of decompositions and selecting bound, free and shared sets of inputs variables for them. We call it the "top-level" strategy. Next, we need such strategies for: Variable Partitioning Problem, Column Minimization Problem, Encoding, and other.

The problem of creating complex heuristic strategies to deal with combinatorial problems is very similar to that of general problem-solving methods in Artificial Intelligence. There are five main principles of problem solving in AI:

- state-space approaches,
- problem decomposition,
- automatic theorem proving,
- rule-based systems,
- learning methods (neural nets, abductive nets, immunological, fuzzy logic, genetic algorithm, genetic programming, Artificial Life, etc.).

It seems that all of the above principles can be utilized to construct the top-level search strategy for MULTIS, as well as to construct strategies for some partial combinatorial problems of MULTIS, such as the Column Minimization or Variable Partitioning algorithms.

Since we will limit the discussion in this report to the description of the state-space principle, the approach used is based on the assumption that any combinatorial problem can be solved by searching some space of states. This seems to be the best approach and, by doing so, the types of approaches to the problems of our interest that can be considered within this framework are not greatly restricted.

There are other reasons for choosing the state-space heuristic programming approach:

- The variable partitioning problem can be reduced to integer programming, dynamic programming, or graph-theoretic problems. It can also be reduced to decision diagram ordering problem. The Column Minimization problem can also be reduced to integer programming, dynamic programming, or graph-theoretic problems. In particular, to set covering, maximum clique, or graph coloring. Similarly, all other combinatorial problems that have been programmed in MULTIS, or not yet programmed but described in one of our reports. The computer programs, however, that would result from pure, classical formulations in these approaches would not take sufficiently into account the specific features and heuristics of the problems. Instead reducing to known models, we will create then our own general model, and next "personalize" this model to our problems. Thus, instead of popular graph coloring, we will formulate the compatible graph coloring problem, and moreover, we will use heuristics based on our data to solve this problem efficiently. The problems are rather difficult to describe using these standard formulations. The transition from the problem formulation, in these cases, to the working version of the problem-solving program is usually not direct and cannot be automated well. It is then difficult to experiment with strategy changes, heuristics, etc., which is our main goal here. We aim at model's flexibility, to be able to easily tune it experimentally. In a sense, we are looking at "fast prototyping" possibility.

- Some of these combinatorial problems (or similar problems) have been successfully solved using the state-space heuristic programming methods. The state-space methods include some methods that result from other AI approaches mentioned above. Some backtracking methods of integer programming, and graph-traversing programs used in graph partitioning and clustering methods, are for instance somewhat similar to the variable partitioning problem. They can be thus considered as special cases of the problem-solving strategies in the space of states.

Roughly speaking, several partial problems in the functional decomposition problem can be reduced to the following general model:

- The rules governing the generation of some set S, called *state-space,* are given. For instance in variable partitioning, the elements of set S are called *Partition Input-Output Vectors.* The *Partition Input-Output Vector* is the vector of sets: *Set of Bound Input Variables*, *Set of Free Input Variables*, *Set of Shared Input Variables* and *Set of Output Variables.*
- *Constraint conditions* exist which, if not met, would cause some set $S' \in S$ to be deleted from the set of potential solutions.
- The *solution* is an element of S that meets all the *problem conditions.*
- The *cost function* F is defined for all solutions.
- The solution (one, more than one, or all of them) should be found such that the value of the cost function is *optimal (quasi-optimal)* out of all the solutions.

A problem condition pc is a function with arguments in S and values For instance, if set S is the set of natural numbers:

$$pc_1(x) \ = \ true \ - \ if\ x\ is\ a\ prime\ number; \ \ false \ - \ otherwise$$

In general, a *problem* is then defined as an ordered triple:

$$P \ = \ (S, PC, F), \tag{1}$$

where:

1. PC is a set of predicates on the elements of S, called *problem conditions,*
2. F is the cost function that evaluates numerically the solutions. Solution is an element of S that satisfies all the conditions in PC.

The *tree search method* includes:

1. the problem P,
2. the constraint conditions,

3. *additional solution conditions* that are checked together with the problem conditions,

4. the *generator of the tree,*

5. the *tree-searching strategy.*

Additional solution conditions are defined to increase the search efficiency. Let us for instance assume that there exists an auxiliary condition that is always satisfied when the solution conditions are satisfied, but the auxiliary condition can be tested less expensively than the original solution conditions. In such case the search efficiency is increased by excluding the candidates for solutions that do not satisfy this auxiliary solution.

The additional solution conditions together with the problem conditions are called *solution conditions.* The method is *complete* if it searches the entire state-space and thus assures the optimality of the solutions. Otherwise, the method will be referred to as *incomplete.* Obviously, for practical examples most of our searches will be in incomplete spaces.

Let us illustrate these ideas with the minimal covering problem which we use for several applications in MULTIS, predominantly for the Column Minimization Problem. For instance:

1. the state-space S is a set that includes all of the subsets of the set of rows of the covering table (rows correspond to prime implicants contained in the function [32]).

2. The solution is an element of S that covers all the columns of the function.

3. A cost function assigns the cost to each solution. The cost of a solution is the number of selected rows. It may be also the total sum of costs of rows selected.

4. A solution (set of rows) should be found that is a solution and minimizes the cost function.

5. Additional quality functions are also defined that evaluate states and rows in the search process.

6. This process consists of successively selecting "good" rows (based on the value of the quality function), deleting other rows that cover fewer of the matrix columns (these are the *dominated rows*), and calculating the value of the cost function.

7. The cost value of each solution cover found can then be used to limit the search by *backtracking.*

8. This process can be viewed as a search for sets of rows in the state-space, and can be described as a generation of a tree *(solution tree)* using rows as *operators,* sets of rows as *nodes of the tree,* and solutions as *terminal nodes.*

A combinatorial problem of a set covering type can be either reduced to a covering table, or solved using its original data structures.

It is well known, that the PLA minimization problem, finding vacuous variables, column minimization, microcode optimization, data path allocation, Three Level AND/NOT Network with True Inputs (TANT) minimization problem, factorization, test minimization and many other logic synthesis problems can be reduced to the set covering. Hence, the latter problem can be treated as a *generic logic synthesis subroutine.* Several efficient algorithms for this problem have been created [6, 7, 18, 22, 66, 80]. We can use the search ideas from this report to solve efficiently all these problems. Equivalently, we believe that some of the ideas originated in these papers can be also used to extend the search framework presented by us.

Moreover, various methods of reduction of a problem to the Set Covering Problem exist. These methods would result in various sizes of the covering problem. By a smart approach, the problem may still be NP-hard, but of a smaller dimension. For a particular problem then, one reduction will make the problem practically manageable, while the other reduction will create a non-manageable problem. This is true, for instance, when the PLA minimization problem is reduced to the set covering of the signature cubes [Brayton] as columns of the covering table, instead of the minterms as the columns. Similar properties exist for the Graph Coloring, Maximum Clique, and other combinatorial problems of our interest in functional decomposer. Although the problems are still NP-hard as a class, good heuristics allows to solve a high percent of real life problems efficiently, because of the Occam's razor principle.

Many other partial problems in functional decomposition can also be reduced to a class of NP-hard combinatorial problems that can be characterized as *constrained logic optimization problems*. They are described using multi-valued Boolean functions, graphs, or arrays of symbols. On these data some constraints are formulated and some transformations are executed in order to optimize cost functions. These problems include Boolean satisfiability, tautology, complementation, set covering [22], clique partitioning [73], maximum clique [73], generalized clique partition, graph coloring, maximum independent set, set partitioning, matching, variable partitioning, linear and quadratic assignment, encoding, and others.

With respect to high importance of these problems, several different approaches have been proposed to solve them. These approaches include:

- mathematical analysis of the problems is done in order to find as efficient as possible algorithms (exact or approximate), or algorithms for particular sub-classes of these problems [8], in spite of the fact that the problems are NP-hard so that no efficient (polynomial) algorithms exist for them. For instance, the proper graph coloring problem is NP-hard, but for a non-cyclic graph there exists a polynomial algorithm. How practical is this polynomial algorithm depends only on how often non-cyclic graphs are met in any given area of application where the graph coloring is used.
- special hardware accelerators are designed to speed-up the most often executed or the slowest operations on the standard types of data used in the algorithms.
- general purpose parallel computers, like message-passing hypercubes, SIMD arrays, data flow computers and shared memory computers are used [10].
- the ideas of Artificial Intelligence, computer learning, genetic algorithms, and neural networks are used, also mimicking humans that solve these problems [61].

In future, we plan to use our FPGA-based Universal Hardware Accelerator from DEC, PERLE 1, to implement a special hardware architecture that will speed up the entire functional decomposition by speeding up the process that is repeated most often, and that contributes the most to the quality of results (perhaps checking the column compatibility, or the set covering). We have to develop some good respective models of our methods first.

QUESTIONS FOR SELF-EVALUATION

1. What are the main combinatorial optimization problems in Functional Decomposition?
2. What tree search problems do you know? Give examples. Why the solutions for them were successful?
3. What are the tree search problems that are used in functional decomposition.
4. How to create a tree search program for the following problems?
   (a) maximum clique generation.
   (b) all cliques generation.
   (c) graph coloring.

(d) set covering.

(e) Petrick Function.

(f) bound set selection.

(g) conversion from Sum-of-Products to Product-of-Sums forms and vice versa.

(h) Satisfiability of POS formula.

(i) finding all prime implicants.

(j) finding all sets of vacuous variables.

(k) parallel decomposition.

Give examples of trees and solutions in them. Give the pseudocodes of tree-searching programs based on the formulated by you methods and trees.

2 MULTISTRATEGICAL COMBINATORIAL PROBLEM-SOLVING

2.1 BASIC IDEAS

The goal of this section is to explain how the general objectives outlined above can be realized in C++ programs to be included in MULTIS. Our interest is in uniform explanation and creation of state-space tree search methods. Our goal is *Fast Prototyping.* It is desired that all these programs are written in such a way that the developer of the program will be able to experiment easily with problem description variants and create various search strategies for different tree search methods to optimize the efficiency of the search.

The tree-searching strategy is created by selecting respective classes and values of *strategy parameters.* The creation of multiple variants of a tree-searching program, that would have otherwise require weeks of code writing and debugging, would then be possible in a shorter period of time. Some efficiency of execution will be lost but the gain of being able to test very many variants of the algorithm will be much more substantial. The *behavior* of the variants of the tree search methods will be then compared and evaluated by the developer to create the final efficient algorithms.

Of course, these search programs will all use our *basic logic design subroutines* that include those that realize operations on *cubes* or *cube arrays,* like *sharp, intersection, or consensus* [14, 74]. It also uses BDD subroutines, partition subroutines, and all other partial subroutines that already exist in GUD.

2.2 DESCRIPTION OF THE SOLUTION TREE

The search strategy realizes some design task by seeking to find a set of solutions that fulfill all problem conditions. It checks a large number of partial results and temporary solutions in the tree search process, until finally it determines the optimality (optionally, the quasi-optimality) of the solutions. The *state-space* S for a particular problem solved by the program is a set which includes all the solutions to the problem. The elements of S are referred to as *states.* New states are created from previous states by application of operators. During the realization of the search process in the state-space, a memory structure termed *solution tree, solution space,* is used.

The solution tree is defined as a graph:

$$D = [NO, RS]$$

- Solution tree contains *nodes* from set NO, and *arrows* from set of arrows RS. Nodes correspond to the stages of the solution process. (see Figure 37).
- Each arrow is a pair of nodes (n_{i1}, n_{i2}). Arrows are also called *oriented edges.* They correspond to transitions from stages to stages of the solution process.
- An *open node* is the node without created *children*, or immediate successors. Child of child is called *grandchild.* If s is a child of p then p is a *parent* of s. A *successor* is defined recursively as a child or a successor of a child. A *predecessor* is defined recursively as a parent or a predecessor of a parent.
- A *semi-open node* is the node with an expanded part of children, but not all of its children.
- A *closed node* is a node for which all its children have been already created in the tree.
- The set of all nodes corresponding to solutions will be denoted by S_F.

In the solution space we can distinguish the following sub-spaces:

- **actual solution space** - the space which has a representation in the computer memory.
- **potential solution space** - the space that can be created froom the actual space using operators and taking into account constraints.

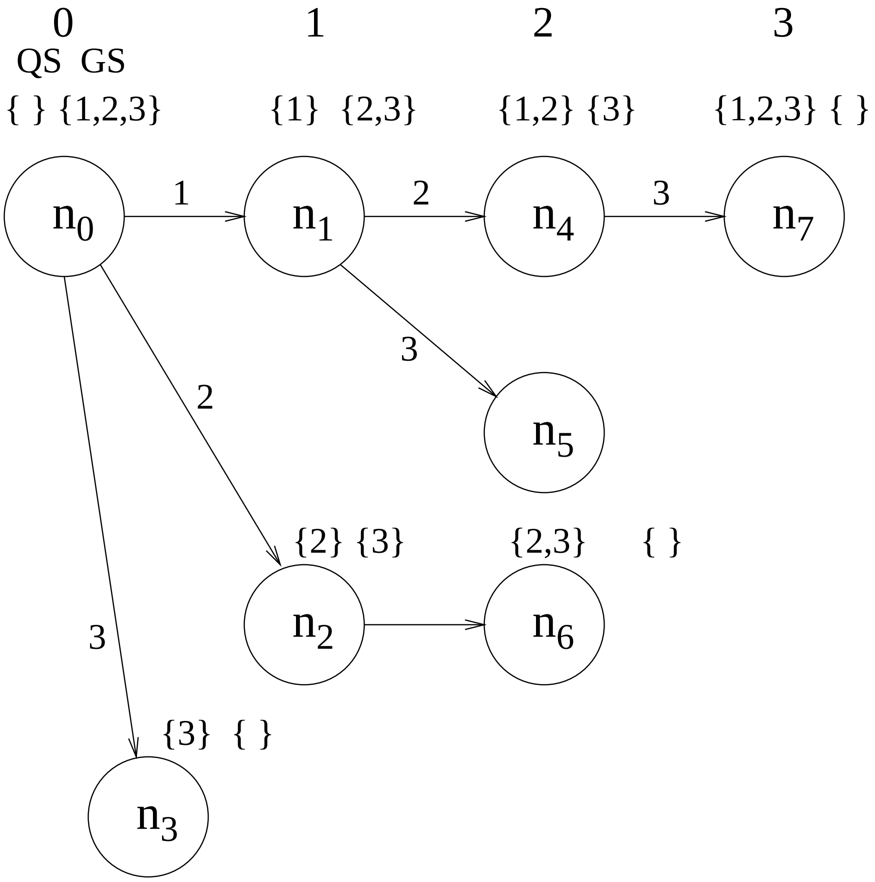


Figure 1: Example of T_1 type tree generator of a full tree

- **closed space** - the space which has already been an actual space for some time, but has been next removed from the memory (with except of the solutions).

The search process grows the actual space at the expense of the potential space, and grows the closed space at the expense of the actual space. The actual space is permanently modified by adding new tree segments and removing other segments. Sometimes the closed space is saved in hard disk, and re-used only if necessary.

The way to expand the space, called the *search strategy*, is determined by: (1) the way the open and semi-open nodes are selected, (2) the way the operators applied to them are selected, (3) the way the termination of search procedure is determined, (3) the conditions for which the new search process is started, and (4) the way the parts of the space are removed from the memory. By "opening a node" we will mean creating successors of this node.

The arrows are labelled by the *descriptors of the operators*. Each node contains a description of a state-space state and some other search related information, in particular descriptors of the operators to be applied. Descriptors are some simple data items. For instance, the descriptors can be: numbers, names, atoms, pairs of elements, sets of elements. Descriptors can be created and manipulated by the search program. They can be stored in nodes or removed from the nodes. For instance, let us discuss the case where partial solutions are sets of integers. In such case, descriptors can be pairs of symbols (*aritmetic_operator*, *integer*) and the application of an operator consists in taking a *number* from the partial solution and creating a new number, like this:

$$new\_number\ :=\ number\ aritmetic\_operator\ integer$$

and replacing the *number* in the partial solution of the successor node by the *new_number*.

The descriptors can be created by programs called *descriptor generators*. They generate descriptors for each node one by one or all of them at once. *Operators* traverse the tree from a node to a node, which is equivalent to searching among the states of S. Each of the solution tree's nodes is a *vector of data structures*. For explanation purposes, this vector's *coordinates* will be denoted as follows:

- N - the node number,
- SD - the node depth,
- CF - the node cost,
- AS - description of the hereditary structure,
- QS - partial solution,
- GS - set of descriptors of available operators,

Additional coordinates can then be defined, of course, as they are required.

Other notations used:

- NN - the node number of the immediately succeeding node (a child),
- OP - the descriptor of the operator applied from N to NN,
- NAS - actual length of list AS,
- NQS - actual length of list QS,
- NGS - actual length of list GS.

The operator is denoted by OP_i, and it's corresponding descriptor by r_i. An application of operator OP_i with the descriptor r_i to node N of the tree is denoted by $O(r_i, N)$. A *macro-operator* is a sequence of operators that can be applied successively without retaining the temporarily created nodes.

A prerequisite to formulate the combinatorial problem in the search model is to ascertain the necessary coordinates for the specified problem in the *initial node - the root of the tree.* The way in which the coordinates of the subsequent nodes are created from the preceding nodes must be ascertained as well. This leads to the description of the *generator of the solution space (tree generator). Solution conditions* and/or cost functions should be formulated for most of the problems. There are, however, generation problems (such as generating all the cliques of a specific kind), where only the generator of the space is used to generate all the objects of a certain kind.

- QS is the partial solution: that portion of the solution that is incrementally grown along the branch of the tree until the final solution is arrived at. A set of all possible values of QS is a state-space of the problem. According to our thesis, some relation $RE \in S \times S$ of partial order exists usually in S. Therefore, the state s $\in$ S symbolically describes the set of all $s' \in S$ such that $s\ RE\ s'$. The solution tree usually starts with $QS(N_0)$ which is either the *minimal* or the *maximal element of* S. All kinds of relations in S should be detected, since they are very useful to create efficient search strategies.
- The set $GS(N)$ of descriptors denotes the set of all operators that can be applied to node N.

- $AS(N)$ denotes the *hereditary structure*. By a hereditary structure we understand any data structure that describes some properties of the node N that it has inherited along the path of successor nodes from the root of the tree.

- The *solution* is a state of space that meets all the solution conditions.

- The *cost function* CF is a function that assigns the cost to each solution.

- The *quality function* QF can be defined as a function of integer or real values pertinent to each node, i.e., to evaluate its quality. It is convenient to define the cost and quality functions such that

$$QF(N) \le CF(N) \text{ and if } QS(N) \text{ is the solution, then } QF(N) = CF(N) \tag{2}$$

- $TREE(N)$ denotes a subtree with node N as a root. Often function $QF(N)$ is defined as a sum of function $F(N)$ and function $\hat{h}(N)$:

$$QF(N) = CF(N) + \hat{h}(N) \tag{3}$$

- $\hat{h}(N)$ evaluates the distance $h(N)$ of node N from the best solution in $TREE(N)$. $F(N)$ in such a case defines a partial cost of $QS(N)$. Then $\hat{h}(N)$ is called a *heuristic function.* We want to define $\hat{h}$ in such a way that it as close to h as possible (see [46] for general description).

A theoretical concept of function f is also useful to investigate strategies as well as cost and quality functions. This function is defined recursively on nodes of the extended tree, starting from the terminal nodes, as follows:

$$f(NN) = CF(NN) \qquad \text{when the terminal node NN is a solution from } \mathrm{S_F}, \tag{4}$$

$$f(NN) = \infty \qquad \text{when the terminal node NN is not a solution,} \tag{5}$$

$$f(N) = min(f(N_i)), \qquad \text{for all } \mathrm{N_i} \text{ which are the children of node N.} \tag{6}$$

This function can be calculated for each node only if all its children have known values, which means practically that the whole tree has been expanded. $f(N)$ is the cost of the least expensive solution for the path which leads through node N. We assume that such function CF can be created that for every node N (and not only for the nodes from set S_F of solutions), it holds that additionally hold:

$$CF(N) \le f(N) \tag{7}$$

and

$$CF(NN) \ge CF(N) \qquad \text{for NN} \in \text{SUCCESSORS(N)} \tag{8}$$

The general idea of the **Branch and Bound Strategy** consists in having a CF that satisfies equations 4, 7, 8. Then, knowing a cost CF_{min} of any intermediate solution that is temporarily treated as the minimal solution, one can cutt-off all subtrees $TREE(N)$ for which $CF(N) > CF_{min}$ (or, $CF(N) \ge CF_{min}$ when we look for only one minimal solution).

In many problems it is advantageous to use a separate function QF, distinct from CF, and then CF guides the process of cutting-off subtrees, while QF guides the selection of nodes for expansion of the tree. In particular, often the following functions are defined:

$g(N)$ the smallest from all the values of cost function calculated on all paths from N_0 to N.

$h(N)$ the smallest from all the values of increment of cost function calculated from N to some $N_k \in S_F$. This is the so-called **heuristic function**.

$f(N) = g(N) + h(N)$.

Since function h cannot be calculated in practice in node N during tree's expansion, and g is often difficult to find - a function CF can be introduced that approximates g and function $\hat{h}$ that approximates h, such that

$$QF(N) \;=\; CF(N) \;+\; \hat{h}(N) \tag{9}$$

$$h(N) \;\geq\; \hat{h}(N) \;\geq\; 0 \tag{10}$$

$$h(M) \;-\; h(N) \;\leq\; h(M,N) \tag{11}$$

where $h(M,N)$ is the smallest of all increment values of cost function from M to N, when $M,\ N \notin S_F$. It holds also:

$$QF(N) \;=\; CF(N) \qquad \text{for N } \in \mathrm{S_F} \tag{12}$$

$$h(N) \;=\; \hat{h}(N) \;=\; 0 \qquad \text{for N } \in \mathrm{S_F} \tag{13}$$

Function defined like that are useful in some search strategy, called *Nilsson A^* Search Strategy.* Sometimes while using branch-and-bound strategies it is entirely not possible to define cost function $g(N)$ for $N \notin S_F$. However, in some cases one can define function QF such that for each N

$$QF(N) \;\leq\; g(N) \tag{14}$$

For nodes $N \in S_F$ one calculates then $g(N) = CF(N)$, and next one uses standard cut-off principles, defining for remaining nodes N_i: $CF(N_i) = QF(N_i)$, and using function CF in a standard way for cutting-off. A second, standard role of QF is to control the selection of non-closed nodes. (By non-closed nodes we mean those that are either open or semi-open). One should then try to create QF that plays these both roles.

A *quasi-optimal* or *approximate solution* is one with no redundancy: i.e., if the solution is a set, all of its elements are needed. When the solution is a path in a certain graph, for example, it has no loops. An *optimal solution* is a solution $QS(N) = s \in S$ such that there does not exist $s' \in S$ where $F(s) > F(s')$. The problem can have more than one optimal solution. The set of all solutions will be denoted by SS. Additional *quality functions for operators* can also be used.

In many combinatorial problems, the set of all mathematical objects of some type are needed: sets, functions, relations, vectors, etc. For example, the following data are created:

- The set of all subsets of a set of prime implicants in the minimization of a Boolean function.
- The set of all subsets of bound variables in the variable partitioning problem.
- The set of all two-block partitions in the encoding problem.
- The set of maximal compatibles in the column minimization.

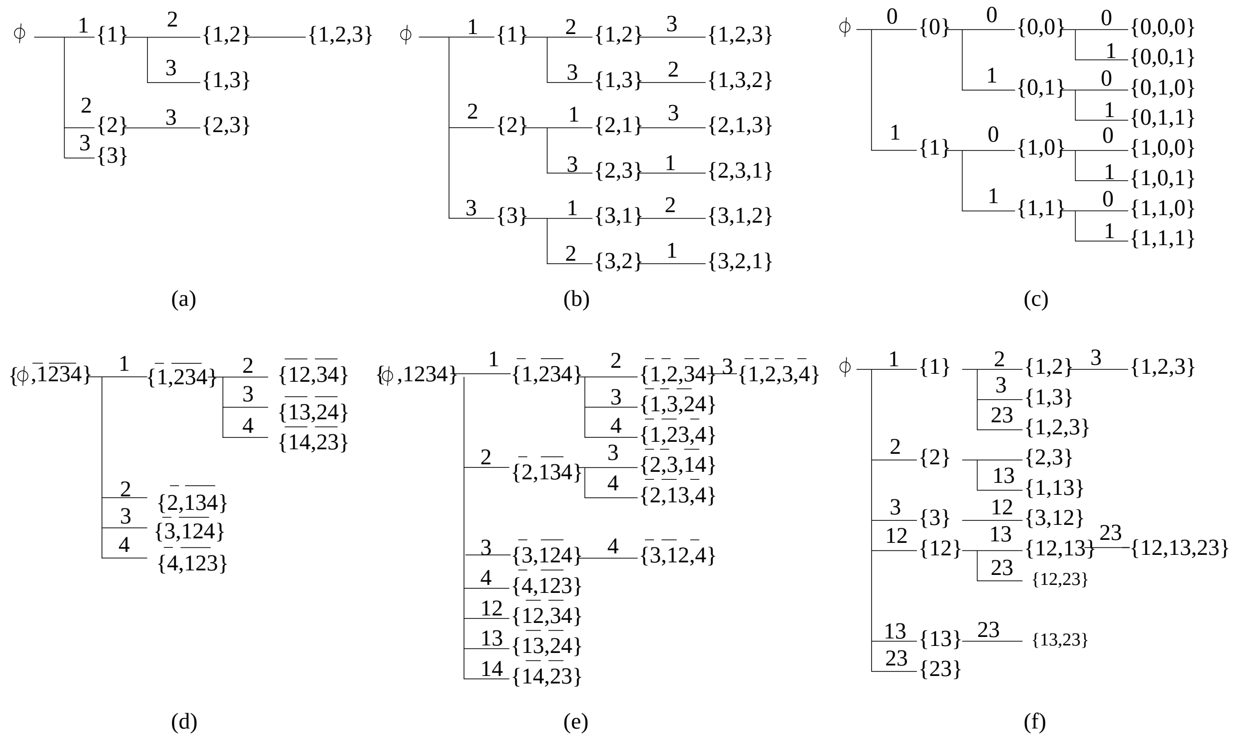


Figure 2: Examples of tree generators

These sets can be generated by respective search routines for them in a standard way using generators of trees. This is useful in *direct problem descriptions.*

It is desirable to develop descriptor generators for several standard sets, several types of tree generators, many ordering possibilities for each generation procedure, and several tree extension strategies for each ordering. The *type of tree* is defined by the generator that generates descriptors as well as the generator that generates the tree. A *full tree* is a tree generated by generators only, ignoring *constraint conditions,* quality functions, dominations, etc. Full trees of the following types exist:

- a tree of all subsets of a given set - type T_1.
- a tree of all permutations of a given set - type T_2.
- a tree of all one-to-one functions from a set A to set B - type T_3,

and many others.

The type T_1 tree generator of a full tree of the set of all subsets of the set {1,2,3}, shown in Fig. 37 can be described as below.

1. *Initial node* (root) is described as:

$$QS(N_0) \;=\; \emptyset; \tag{15}$$

$$GS(N_0) \;=\; \{1,2,3\}; \tag{16}$$

where $\emptyset$ is an empty set, and N_0 is the root of the tree.

2. The *children* of any node N are described *recursively* as follows:

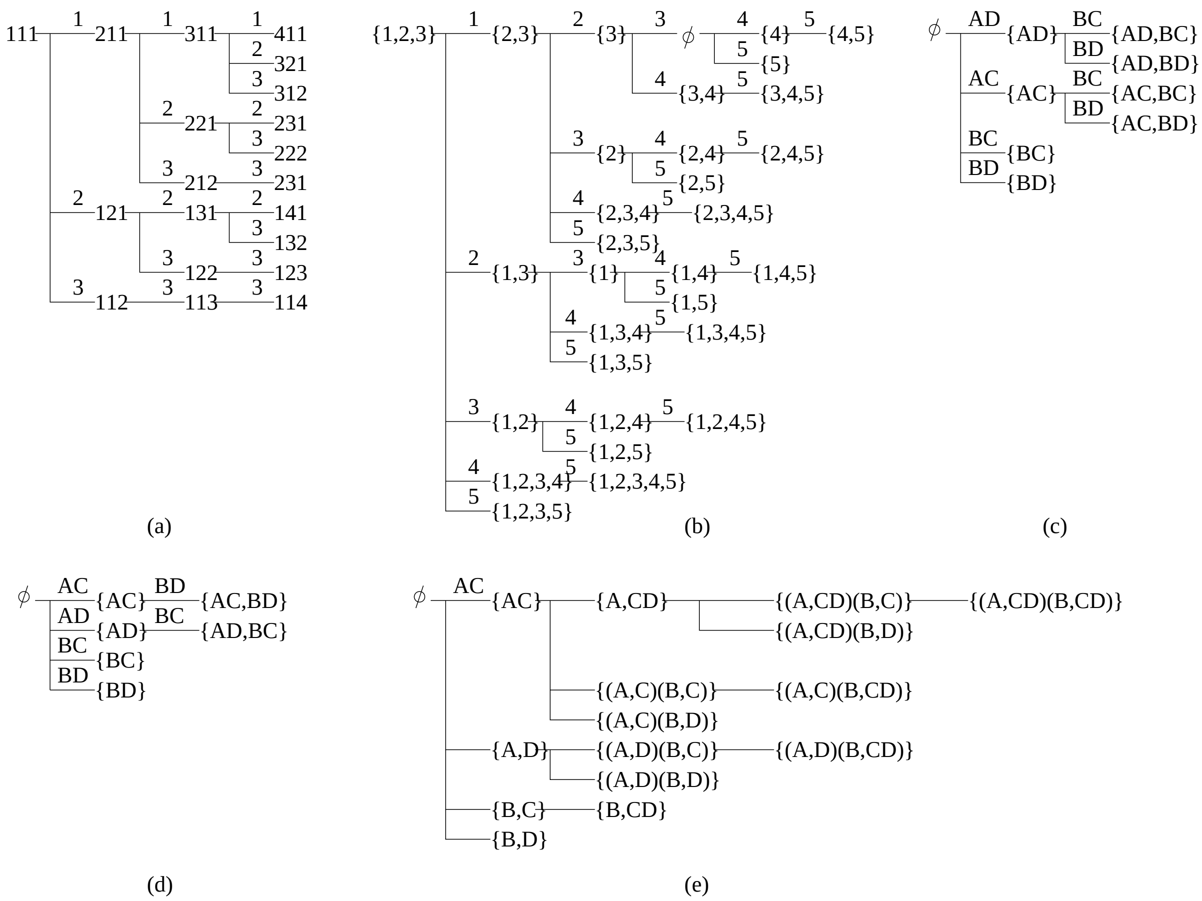


Figure 3: Further Examples of tree generators

$$(\forall\ r\ \in\ GS(N))\ [QS(NN) = QS(N)\ \cup\ \{r\};\quad GS(NN)\ =\ \{r_1\ \in\ GS(N)\ \mid\ r_1 > r\}] \qquad (17)$$

where NN is some child of node N, and r is the descriptor of the operator. Set GS is either stored in the node or its elements are generated one by one in accordance with the ordering relation $>$ while calculating the children nodes.

Figure 2 and Figure 3 present examples of full sets for many important combinatorial problems.

2.3 CREATING SEARCH STRATEGIES

A number of *search strategies* can be specified for the tree search procedure along with the quality functions. Beginning with the initial node, the information needed to produce the solution tree can be divided into *global information,* that relates to the whole tree, and *local information.* Local information in node N refers to subtree $TREE(N)$. The developer-specified search strategies are, therefore, also divided into a *global search* and a *local search.* The selection of the strategy by the user of the Universal Search Strategy from section 2.6 is based on a set of *strategy describing parameters.* By selecting certain values he can for instance affect the size of subsequent sets of bound variables or the types of codes in the encoding problem. We assume also that in the future we will create smart strategies that will allow to dynamically change the strategy parameters by the main program during the search process. Such strategies, that for instance search breadth-first and after finding a node with certain properties switch to depth-first search, have been used with successes in Artifficial Intelligence.

Let us distinguish the *complete search strategies* that guarantee finding all of the optimal solutions from the *incomplete strategies* that do not. Both the complete and the incomplete search strategies can be created for a complete tree search method. A tree searching strategy that is created for a complete tree search method and includes certain restricting conditions or cutting-off methods that can cause the loss of all optimal solutions is referred to as an *incomplete search strategy* for a complete search method. By removing such conditions a complete search strategy is restored, but it is less efficient.

This approach offers the following advantages:

- The *quasi-optimal solution* is quickly found and then, by backtracking, successive, better solutions are found until the *optimal solution* is produced. This procedure allows to investigate experimentally the trade-offs between the quality of the solution and speed of arriving at it.

- The search in the state-space can be limited by including as many *heuristics* as required. In general, a heuristic is any rule that directs the search.

- The application of various quality functions, cost functions, and constraints is possible.

- The problem can be described within several degrees of accuracy. The *direct description* is easy for the designer to formulate, even though it produces less efficient programs. It is created in the early prototype development stages, on the basis of the problem formulation only, and the heuristics are not yet taken into account. The only requirement is that the designer knows how to formulate the problem as a state-space problem using standard mathematical objects and relations. Only the standard node coordinates are used. The *detailed description of the tree search method,* on the other hand, provides the best program that is adequate for the specific problem but it requires a better understanding of the problem itself, knowledge about the program structure, and experimentation.

- By using macro-operators along with other properties, the main strategies require less memory than the comparable, well-known search strategies [33, 46, 49].

The search strategy is either selected from the *general strategies,* of which the following is a selection, or it is created by the developer's writing of the *sections' codes*, and next the user assigning values to the *strategy describing parameters.*

- *Breadth-First.* With this strategy, each newly created node is appended to the end of the so called *open-list* which contains all the nodes remaining to be extended: *open nodes.* Each time the first node in the list is selected to be extended, it is removed from the list. After all the available operators for this node have been applied, the next node in the open-list to be extended is focused on.

- *Depth-First.* The most recently generated node is extended first by this strategy. When the specified depth limit SD_{max} has been reached or some other cut-off condition has been satisfied, the program backtracks to extend the deepest node from the open-list. This newly created node is then placed at the beginning of the open-list. The consequence is that the first node is also always the deepest.

- *Branch-and-Bound.* The temporary cost B is assigned which retains the lowest cost of the solution node already found. Whenever a new node NN is generated, its cost $CF(NN)$ is compared to the value of B. All the nodes whose cost exceed B will be cut off from the tree.

- *Ordering.* This strategy, as well as the next one, can be combined with the Branch-and-Bound strategy. A quality function $Q(r, N)$ is defined for this strategy to evaluate the cost of all the available descriptors of the node being extended. These descriptors are applied in the operators in an order according to their evaluated cost.

- *Random.* With this strategy, the operator or the open node can be selected randomly for expansion, according to the probability distribution specified.
- *Simulated annealing*, that transforms nodes from open list using the respective algorithm.
- *Genetic algorithm*, uses open list as a genetic pool of parent chromosomes.

All the strategy creating tools should be defined as C++ classes. The strategy describing subroutines and parameters are outlined below.

There are two types of conditions for each node of the tree: *by-pass condition* and *cut-off condition.* The cut-off condition is a predicate function defined on node N as an argument. If the cut-off condition is met in node N, the subtree $TREE(N)$ is prevented from being generated and backtracking results. The by-pass conditions do not cause backtracking and the tree will continue to extend from node N. The following cut-off conditions exist:

- *bound condition* - it is known (possibly from information created in node N) that node N_1 exists (not yet constructed) such that $CF(N_1) < CF(N)$ and $QS(N_1)$ is a solution.
- *depth limit condition* - $SD(N)$ is equal to the declared depth limit SD_{max}.
- *dead position* - no operators can be applied to N, i.e. $GS(N) = \emptyset$.
- *restricting conditions* - $QS(N)$ does not fulfill certain restrictions, i.e. no solution can be found in $TREE(N)$.
- *solution conditions of the cut-off type* - if $QS(N)$ is a solution, then for each $M \in TREE(N)$, $CF(M) > CF(N)$ (or $CF(M) \geq CF(N)$) and M may be not taken into account.
- *Branch Switch Conditions* and *Tree Switch Condition.* Satisfaction of Switch condition causes modification of the actual search strategy to another strategy, resulting from the search context and previous conditional declaration of the user. This leads to the so-called *Switch Strategies* that dynamically change the search strategy during the process of search. For instance, the depth first strategy can be changed to breadth first if certain condition is met.
- *other types of conditions* formulated for some other type restrictions special to problems (selected by the user by setting flags in the main algorithm).

A value that interrupts the search when a solution node N is reached such that $CF(N) = CF_{min\ min}$ is denoted by $CF_{min\ min}$. This is a known minimum cost of the solution. This value can be arrived at through a calculated guess or by mathematical deduction. It may also be a known optimal cost. In most cases it is a guessed value that may be incorrect, so it will serve only here as one more control parameter.

When all the solution conditions are met in a certain node N, $QS(N)$ is a solution to the given problem. This is then added to the set of solutions and is eventually printed. The value of $CF(N)$ is retained. If one of the solutions is of the cut-off type, the program backtracks. Otherwise, the branch is extended.

2.4 RELATIONS ON OPERATORS AND STATES

It is often very useful that one can determine some relations on operators (descriptors) and/or relations on states of the solutions space or on nodes of the tree. This allows to cut-off nodes, and also to remove dispensable descriptors from the nodes. Specifically, in many problems it is good to check solution conditions immediately after creating a node, and next immediately reduce the set od descriptors that can be applied to this node.

The following relations between the operator descriptors (so called *relations on descriptors)* can be created by the program developer to limit the search process:

- *relation of domination,*
- *relation of global equivalence,*
- *relation of local equivalence.*

We will distinguish *local* and *global* domination relations. Operator O_2 is locally dominated in node N by operator O_2 (or descriptor r_2 is locally dominated by r_1) when:

$$(O_1,\ O_2)\ \in\ DOML(N) \tag{18}$$

while relation $DOML$ satisfies the following conditions:

$$DOML\ is \text{ transitive} \tag{19}$$

and

$$(O_1,\ O_2)\ \in\ DOML\ \Rightarrow\ f(O_1(N))\ \leq\ f(O_2(N)) \tag{20}$$

We will apply the notation:

$$(O_1,\ O_2)\ \in\ DOML\ \overset{define}{\Longleftrightarrow}\ O_1 \underset{L}{\succeq} O_2 \tag{21}$$

We will define operator O_2 as *locally subordinated* in node N with respect to operator O_1 (where $d_1,\ d_2\ \in\ GS(N)$), if

$$O_1 \underset{L}{\succeq} O_2\ \wedge\ \ O_2 \underset{L}{\not\succeq} O_1 \tag{22}$$

This will be denoted by

$$O_1 \underset{L}{\succ} O_2 \tag{23}$$

If $O_1 \underset{L}{\succ} O_2$ in node N, then tree $TREE(O_2(N))$ can be cut-off without sacrificing optimal solutions, since

$$f(O_1(N))\ <\ f(O_2(N)) \tag{24}$$

It is easy to check that relation $\underset{L}{\approx}$, defined as

$$O_1 \underset{L}{\approx} O_2\quad \overset{define}{\Longleftrightarrow}\quad O_1 \underset{L}{\succeq} O_2\ \wedge\ O_2 \underset{L}{\succeq} O_1 \tag{25}$$

is an *equivalence relation*, which we will call *Local Relation of Equivalency of Descriptors* in node N. Relation $\underset{L}{\approx}$ partitions set $GS(N)$ into classes of abstraction $[d_i]$. It is then obvious, that if we want to obtain only one optimal solution being a successor of N, then we have to select from each class of abstraction $[d_i]$ one element, and remove all remaining from $GS(N)$.

The relation of *global domination* gives better advantages that the local domination, if it can be defined. Operator O_2 is *globally dominated* in tree $TREE(N)$ by operator O_1 when

$$(O_1,\ O_2)\ \in\ DOMG(N)\ \subset\ O(TREE(N))\ \times\ O(TREE(N)) \tag{26}$$

By $O(TREE(N))$ we denote the set of operators to be applied in tree $TREE(N)$. Relation $DOMG$ satisfies the following conditions:

$$DOMG \ is \ transitive \tag{27}$$

and

$$(O_1,\ O_2) \in DOMG(N) \ \Rightarrow \ (\forall M_1 \in NO(TREE(N))) \\ [d_1 \in GS(M_1) \wedge d_2 \in GS(M_1) \wedge f(O_1(M_1)) \in f(O_2(M_1)) \vee d_2 \notin GS(M_1)] \tag{28}$$

Similar to local relations, one can define relation $\underset{G}{\succ}$ of *global subordination in tree* $TREE(N)$, and relation $\underset{G}{\approx}$ of *global equivalence* in tree $TREE(N)$.

Relation $\underset{G}{\approx}$ partitions every set $GS(M_1)$ for each $M_1 \in NO(TREE(N))$ into classes of abstraction. If we have no intention to find all optimal solutions, then from each class of abstraction we take just one element, and the remaining operators are removed from $GS(M_1)$.

It can be proven that if for each branch $N, N_1, N_2, ...N_k$ of tree $TREE(N)$ it holds $GS(N) \supseteq GS(N_2)$ $\supseteq GS(N_k)$ then the descriptors from set $[d_i] \setminus \{d_i\}$, where $[d_i]$ is the global equivalence class of operator O_i in node N, can be once forewer removed from GS for all nodes in $TREE(N)$.

When we want to use the relation of global equivalency, and the property from this theorem does not hold, then it is necessary to calculate the descriptors, which should **not** be applied in this node (sometimes it can be easily done from an analogous set for the node being the parent of this node).

Node M is *dominated* by node N if $f(N) \ \leq \ f(M)$

$$(N,\ M) \in DOMS \iff f(N) \leq f(M) \tag{29}$$

Analogously as before, we can introduce relations $\underset{S}{\succeq}$, $\underset{S}{\succ}$, and $\underset{S}{\approx}$.

If ST_1 and ST_2 are two strategies, which differ only in their domination relations D_1 and D_2 (these can be relations of domination of any of the presented types) and if $D_1 \supset D_2$ then ${k_i}^1 \ \leq \ {k_i}^2$ for each of the introduced coefficients k_i.

Let us also observe, that by incorporating checking the relation of domination (or equivalence) to an arbitrary strategy that generates all optimal solutions, there exists the possibility of sacrificing only some optimal solutions (or all but one optimal solutions). This decreases the number of generated nodes, which for many strategies brings gains both in time and in memory. On the other hand, if evaluating relations is very complex, the time of getting the solution can increase. The stronger the domination, the more complex its evaluation, or larger its domain - therefore the longer is the checking time. In turn, the more gain from the decreased number of generated nodes. Often it is convenient to investigate relation of domination only in nodes of the same depth, or on operators of some group. Theoretical analysis is often difficult and experimenting is necessary.

Let us finally observe that domination relations are not based on function f, which values are not known a'priori while creating the tree. The domination relations are also not based on costs, but on some additional problem-dependent information about the nodes of the tree, that is available to the program. This is based on some specific problem-related information. In most cases, implication in equation 20 cannot be replaced by an equivalence, since this would lead to optimal strategies with no search at all, and each branch would lead to optimal solutions.

2.5 COMPONENT SEARCH PROCEDURES

Main Universal Search subroutine is in charge of the global search which includes the selection of strategies, the arrangement of the open-list and the other lists as well as the decision making facilities related to the cut-off branch, and the configuration of the memory structure for the tree. The lines of code that realize the strategies of breadth-first, depth-first, or branch-and-bound are built into main search routine. Subroutines $RANDOM1$ and $RANDOM2$ are selectively linked for the random selection of the operator or the open node, respectively. The role of the subroutines linked to the Universal Search subroutine is as follows:

- $GENER$ is responsible for the local search that extends each node. $GENER$ cuts off the nodes which will not lead to the solution node when the description for the new node is created.
- GEN carries out the task of creating nodes.

Other subroutines, offered to create local search strategies, are the following:

- $MUSTAND$ and $MUSTOR$ are subroutines that serve to find two types of the *indispensable operators.* (The indispensable operators are the operators that must be applied.) All operators found are indispensable in $MUSTAND$ subroutine, and only one of operators is indispensable in case of $MUSTOR$ subroutine. The set of indispensable operators is next substituted as the new value of coordinate $GS(N)$.
- subroutine $MUSTNT$ deletes *subordinate operators.* Subordinate operators are those that would lead to worse solutions, or to no solutions at all. The set $MUSTNT(N)$ is subtracted from set $GS(N)$.

Domination and equivalence conditions for the tree nodes can also be declared as follows:

- $EQUIV$ cancels those nodes that are included in other nodes.
- $FILTER$ checks whether the newly created node meets the conditions.
- $SOLNOD$ checks the solution condition.
- $REAPNT$ is used to avoid the repeated applications of operators when the sequence of operator applications does not influence the solution.

These local strategies as well as the global strategies listed above can be selected by reading the parameter values as input data. $ORDER$ sorts the descriptors, QF calculates the quality function for the descriptors, and CF calculates the cost of the nodes.

2.6 UNIVERSAL SEARCH STRATEGY

In this section we will present the universal search strategy.

Meaning of Variables and Parameters

CF_{min} - cost of the solution that is actually considered to be the minimal one. After a full search, this is the cost of the exact minimum solution.

SOL - set of solutions actually considered to be minimal. If parameter $METHOD = 1$ then this set has always one element. After terminating full search, this set includes solutions of the exact minimal cost.

$OPERT$ - list of descriptors, which should be applied to the actual state of the tree.

$OPEN$ - list of open and semi-open nodes.

N - actual state of the space.

NN - next state of the space (being actually constructed from N).

$OUTPUT$ - a parameter that specifies the type of the actually created node;

when $OUTPUT = 0$ - the created node NN is a branching node;

when $OUTPUT = 1$ - the created node NN is an end of a branch;

when $OUTPUT = 2$ - a quasioptimal solution was found, whereby by a quasioptimal we understand any solution that has the value of the cost function not greater than the user-declared parameter $CF_{min\ min}$.

$CF_{min\ min}$ - parameter. assumed by the user, determined heuristically or methodically, the value that satisfies him.

QF_{min} - the actually minimal value of the quality function.

OPT - parameter; when $OPT = 1$ then any solution is sought, otherwise the minimal solution.

$PP9$ - parameter; when $PP9 = 1$ then called is the subroutine *"Actions on the Selected Node"*.

EL - actual descriptor from which the process of macro-generation starts (this is the first element of list $OPERT$).

$DESCRIPTOR$ - actual descriptor during the macrogeneration process.

$MUST$ - list of descriptors of operators, which must be applied as part of the macrooperator.

$PG5$ - parameter; if $PG5 = 1$ then it should be investigated, directly after creation of node NN, if node NN could be cut-off.

$PG6$ - parameter; if $PG6 = 0$ then it should be investigated if node NN can be cut-off with respect to the monotonically increasing cost function CF, and in respect to satisfaction of $CF_{min} = CF(NN)$.

$PG6D$ - parameter; if $PG6D = 1$ then value CF_{min} should be calculated with respect to a subroutine of a user, otherwise CF_{min} is calculated in a standard way as $CF(NN)$.

$PG6E$ - parameter; if $PG6E = 1$ then the learning subroutine is called.

$PG6F$ - parameter; if $PG6F = 1$ then after finding a solution the actions declared by the user are executed.

$PG7$ - parameter; if $PG7 = 1$ then descriptors defined by other parameters are removed from $GS(NN)$.

The Main Search Strategy

1. Set the values of parameters that determine the search strategy.
2. $CF_{min} := \infty$, $SOL := \emptyset$, $OPERT := \emptyset$, $OPEN := \emptyset$,
3. Call the macrogeneration subroutine $GENER$ for the user-declared initial state N_0.
4. If the value of variable $OUTPUT$ (this value is set by subroutine $GENER$) is 1 or 2

   then, according to the declared parameters, return to the calling program for the problem, or select a strategy corresponding to the declared data.
5. State N_0 has been (possibly) transformed by subroutine $GENER$.

   Store the new state N_0' in the tree. $OPEN := \{N_0'\}$.

6. If $OPEN = \emptyset$ then either return to the calling program, or change search strategy according to parameters and strategy change parameters for trees (*Tree Switch*). (see section 2.8.
7. If the threshold values for the tree have been exceeded (size, time, etc) then return to the calling program, or change strategy, as in step 6.

   If the Stopping Moment Learning Program decides termination of search, then this search process is terminated and return to the calling program, that will decide what to do next (see section 12.
8. $N :=$ selected node from list $OPEN$. This step is executed on the basis of Strategy Selecting Parameters, including minimal values QF or CF.

   If parameters specify A^* Strategy of Nillsson and $QF(N) \geq QF_{min}$

   then return to the calling program (since all minimal solutions have been already found).
9. If parameter $PP9 = 1$

   then

   call subroutine "Actions on the selected node" (this subroutine can for instance declare such actions as: cut-off a node upon satisfying some condition, sorting $GS(N)$, assigning $GS(N) := \emptyset$, deletion of redundant or dominated operators).

   $OPERT := GS(N)$.
10. Remove from list $OPEN$ all closed nodes.
11. If $OPERT = \emptyset$ then go to 6.
12. $EL := OPERT[0]$, remove EL from list $OPERT$.

    /* $OPERT[0]$ selects the first element of list $OPERT$ */
13. Call subroutine $GENER$.
14. If a Branch Switch Strategy has been declared and a respective switch condition is satisfied

    then execute the Branch Switch change of the search strategy.
15. If $OUTPUT = 0$ then store the node NN, created in $GENER$, in the tree (if a tree exist in addition to list $OPEN$), and add this node in some place of list $OPEN$. This place depends on the selected strategy.

    If $OUTPUT = 2$ then (if parameter $OPT = 2$ then return to the calling program, else go to 11.
16. Go to 11.

Subroutine GENER

1. If $GENER$ is executed in step 13 of the main search strategy then $MUST := EL$ (value of EL has been previously set in the main search routine).
2. If $MUST = \emptyset$ then set $OUTPUT := 0$, return.
3. $DESCRIPTOR := MUST[0]$.
4. Call subroutine $OPERATOR$ of the user: $O(N,\ DESCRIPTOR)$. It generates new state NN for the descriptor selected in step 3.

   $GS(N) := GS(N) \setminus DESCRIPTOR$.

5. If parameter $PG5 = 1$

   and

   (node NN satisfies on of the Branch Cut-Off Conditions **or** NN is dominated by another node (it means it is equal to it or is dominated by one of relations: Node Domination, Node Equivalence, Node Subordination))

   then cut-off node NN. $OUTPUT := 1$, return.

   If $CF(NN) > CF_{min}$ (while looking for all minimal solutions)

   or

   If $CF(NN) \geq CF_{min}$ (while looking for a single minimal solution)

   then

   cut-off node NN. $OUTPUT := 1$, return.

6. If parameter $PG6 = 0$ then:

   If $CF_{min} = CF(NN)$ and the parameter specifies that CF is monotonically increasing and node NN does not satisfy all the user-declared Solution Conditions

   then cut-off node NN, $OUTPUT := 1$, return.

   If node NN satisfies all the user-declared Solution Conditions then

   A If $CF(NN) \leq CF_{min}$ and the A^* Strategy of Nillsson is realized then: store $QF_{min} := QF(NN)$.

   B If $CF(NN) = CF_{min}$ then:

      a) (if all optimal solutions are sought then append $QS(NN)$ to the list of solutions SOL else do nothing.)

   C If $CF(NN) < CF_{min}$ then set $SOL := \{\text{QS(NN)}\}$.

   D If parameter $PG6D = 1$ then calculate CF_{min} using the User Subroutine Calculating CF_{min}, else $CF_{min} := CF(NN)$.

   E If parameter $PG6E = 1$ then call subroutine of parametric learning the quality function for operators.

   F If parameter $PG6F = 1$ then call subroutine "Actions after Finding a Solution". This is a subroutine in which the user specifies actions to be executed after the solution is found, such as printout, display, storage, etc.)

   G If $CF(NN) = CF_{min\ min}$ then $OUTPUT := 2$, return.

   H If $CF(NN) \neq CF_{min\ min}$ then $OUTPUT := 1$, return.

7. If $PG7 = 1$ then remove from $GS(NN)$ the indispensable descriptors (depending on the values of parameters, these are: Inconsistent Descriptors or Descriptors that result from: Local Subordination Relation, Local Domination Relation, Local Equivalence Relation, Local Equivalence Relation, Global Subordination Relation, Global Domination Relation, Global Equivalence Relation).

   Use subroutine $MUSTNT$.

   If a Condition of Node Expansion Termination is satisfied then set $GS(NN) := \emptyset$.

   If the set of Indispensable Operators of $MUSTOR$ type is declared

   and

   respective Condition of operators of $MUSTOR$ type is satisfied

   then set $GS(NN) := MUSTOR(GS(NN))$.

8. $N := NN$.
9. If $MUST \neq \emptyset$ then go to 2.

   else set $MUST$:= set of Indispensable Descriptors of $MUSTAND$ type in $GS(N)$ and go to 2.

The first call of subroutine $GENER$ is intended to check if in the initial state given by the user exist the indispensable operators of type $MUSTAND$. These operators are applied to the successively created states until a solution is found or a node in which no indispensable descriptors exist. When subroutine $GENER$ is returned from, the state N_0 may have been transformed. A condition to find a minimal solution is to terminate with an empty list $OPEN$. In steps 8 and 9, with respect to the strategy determining parameters, selected is a node for expansion, as well as the operators that will be applied to it. This node can be open or semi-open. *Open* means all possible operators have been applied to it. *Semi-open*, means some operators (descriptors) were applied but other still are ready to be applied in a future. Selected descriptors are successively applied to the node, until list $OPERT$ is cleared.

If the user-determined value of parameter OPT is 1 then the subroutine will return to the calling program after finding the first quasi-optimal solution.

Subroutine $GENER$ is used to find and apply macro-operators. On the list of indispensable operators $MUST$ placed is (except of the call in step 3) operator EL selected in the main search program. Such approach has been chosen in order to check if some indispensable operators exist in the initial state. In all subsequent nodes it is known that there are no indispensable operators, since if in a node created by $GENER$ there exists an indispensable operator, it is immediately applied. Therefore, the result of subroutine $GENER$ is always one child that has no indispensable operators.

In a general case, pure branch-and-bound strategy (discussed below) will terminate in steps 4 and 6 of the main search strategy. The A^* strategy of Nilsson will terminate in step 8.

Of course in lists $OPEN$, $OPERT$ and other, not objects but pointers to them are stored.

2.7 PURE SEARCH STRATEGIES

These strategies are the following.

1. Strategy ST_{QF} where:

   $$QF(SEL1_{QF}(OPEN)) = min_{N_i \in OPEN} QF(N_i), \quad SEL2(x) = x \tag{30}$$

   $SEL1$ is the node selection strategy and $SEL2$ is the descriptor selection strategy.

   In this strategy, all children of node N are generated at once. This corresponds to the "Ordered Search" strategy, as described in [46, 26]

   If additionally QF satisfies conditions (4.6) - (4.10) then it corresponds to A^* strategy of Nilsson.

2. Strategy ST_{CF} (strategy of equal costs), in which:

   $$CF(SEL1_{CF}(OPEN)) = min_{N_i \in OPEN} CF(N_i), \quad SEL2(x) = x \tag{31}$$

   This is a special case of the strategy from point 1..

3. Depth-first Strategy

   $SEL1_d(OPEN)$ = the node that was recently opened, $\quad SEL2(x) = x$

4. Breadth-first Strategy

   $SEL1_b(OPEN)$ = the first of the opened nodes, $\quad SEL2(x) = x$

5. Strategy $ST_{d,s,k}$ (depth, with sorting and selection of k best operators):

   $SEL1_{d,s,k}(NON\text{-}CLOSED)$ = the node that was recently opened,

   $SEL2_{d,s,k}(GS(SEL1(NON\text{-}CLOSED)))$ = set that is created by selecting the first k elements in the set $GS(SEL1(NON\text{-}CLOSED))$ sorted in nondecreasing order according to function ${q^N}_i$. A particular case of this strategy is $ST_{d,s,1}$, called the Strategy of Best Operators (Best Search Strategy).

6. Strategy $ST_{d,s,s,k}$ (depth, with selection of node, sorting, and selection of k best operators).

   $SEL1_{d,s,s,k}(NON\text{-}CLOSED)$ = a node of minimum value of function QF among all nodes that are created as the extension of the recently expanded node (not necessarily of the recently opened node).

   $SEL2_{d,s,s,k}$ = analogously to $SEL2_{d,s,k}$.

   Analogously, one can define "k-children" strategies $ST_{QF,k}$, $ST_{CF,k}$, $ST_{d,k}$.

7. Strategy ST_{RS} of Random Search.

   $SEL1_{RS}(NON\text{-}CLOSED)$ = randomly selected node from NON-CLOSED.

   $SEL2_{RS}(GS(SEL1_{RS}(NON\text{-}CLOSED)))$ = randomly selected subset of descriptors.

8. Strategy $ST_{RS,d}$ of Random Search Depth.

   $SEL1_{RS,d}(NON\text{-}CLOSED)$ = recently opened node from $NON\text{-}CLOSED$.

   $SEL2_{RS}(GS(SEL1_{RS,d}(OPEN)))$ = randomly selected descriptor.

Analogically, one can specify many other strategies by combining functions $SEL1$ and SEL given above.

Let us introduce now some measures of quality of strategies.

k_1 = CARD(B_a), where B_a is the set of all closed and semi-open nodes that were created until all minimal solutions have been found.

k_2 = CARD(B_s), where B_s is the set of all closed and semi-open nodes that were created until one minimal solution has been found.

k_3 = CARD(V_a), where $V_a \supseteq B_a$ is the set of all closed, semi-open, and open nodes that were created until all minimal solutions have been found.

k_4 = CARD(V_s), where $V_s \supseteq B_a$ is the set of all closed, semi-open, and open nodes that were created until one minimal solution has been found.

k_5 = CARD(T_a), where $T_a \supseteq B_a$ is the set of nodes that were created until proving the minimality of solutions, it means the total number of nodes that have been ever created by a strategy that searches all minimal solutions.

k_6 = CARD(T_s), analogously to k_5, but for a strategy that searches a single solution.

k_7 = max SD(N_i) - the length of the maximal path (branch) in the tree.

The advantage of the ordered search strategy is the relatively small total number of generated nodes (coefficients k_5 and k_6). The following theorem is true, analogous to one from [46].

Theorem 2.1 *If QF satisfies equations 9, 10, 11, 12, 13 and the ordered search strategy has been chosen (i.e. realized is the strategy A^* of Nilsson) and when some solution of cost QF' has been found, such that all nodes of costs smaller than QF' have been closed, then this solution is the exact minimal one.*

It is important to find conditions, by which this algorithm finds the optimal solution, generating relatively few nodes. The theorem below points to the fundamental role of function $\hat{h}$, the definition of which can substantially influence the quality of operation of the algorithm.

Let ST_1 and ST_2 be two A^* Nilsson strategies, and $\hat{h}_1$ and $\hat{h}_2$ their heuristic functions. We will define that strategy ST_2 is not worse specified than strategy ST_2 when for all nodes N it holds:

$$h(N) \;\geq\; \hat{h}_1(N) \;\geq\; \hat{h}_2(N) \;\geq\; 0 \tag{32}$$

which means, both functions evaluate h from the bottom, but function $\hat{h}_1$ does it more precisely than $\hat{h}_2$.

Theorem 2.2 *If ST_1 and ST_2 are A^* Nilsson strategies, and ST_1 is better specified than ST_2, than for each solution space the set of nodes closed by ST_1 before the minimal solution is found is equal to the set of closed nodes of ST_2, or is included in it.*

This theorem says in other words that if we limit ourselves to A^* Nilsson strategies only, then there exists one of them, better than all remaining ones, since it closes not more nodes than any other strategy. This is the strategy that most precisely evaluates the function h, preserving of course equations 10, 11, 13.

For many classes of problems the ordered search strategy is very inefficient, since it generates its first solution not earlier than very many nodes have been already created. Then it proves its optimality relatively quickly. In cases when one wants to find a good solution quickly, and only as the secondary criterium the **exact** solution, it is better to use one of the variants of the branch-and-bound strategies that search in depth.

Many properties can be proven for the branch-and-bound strategy presented above. We assume that:

$$for\ each\ N, NN, QF(N) \neq QF(NN)\ and\ QF(NN) > QF(N)\ for\ NN \in SUCCESSORS(N) \tag{33}$$

1) The branch-and-bound strategy is convergent, independent on function QF. Therefore, also the specific strategies included in it, such as "depth-first", "ordered search", etc, are also convergent if the user has not declared some additional cut-off conditions that may cause the loss of the optimal solution. Some of these strategies do not require calculating function QF that would satisfy certain conditions, which is the advantage of the given above universal search strategy, when compared with the A^* Nilsson Strategy.

2) If we are able to define the quality function QF^* such that:

$$(\forall N_i, N_j)[QF^*(N_i) < QF^*(N_j) \;\Rightarrow\; f(N_1) \;\leq\; f(N_j)] \tag{34}$$

then the strategy ST_{QF} is optimal in the sense of the number of opened nodes. Extended are only nodes that are on paths leading to solutions (opened are also other nodes).

3) If additionally we succeed to find such quality function for operators q^{*N} that is **consistent** with f, it means such that:

$$(\forall N, O_1, O_2)[q^{*N}(O_1) > q^{*N}(O_2) \;\Rightarrow\; f(O_1(N)) \;\leq\; f(O_2(N))] \tag{35}$$

the the strategy is optimal in the sense of the number of **generated** nodes. Expanded are only the nodes that lay on the paths leading to minimal solutions, and in addition none other nodes are opened (this concerns the 1-child strategies).

4) It is possible to introduce the relation of partial ordering $\ll$ on the set of all possible strategies ST_{QF}. It is shown that the strategies that are *adjacent* in the sense of this order have also similar behavior:

$$if\ ST_{QF_1} \ \ll\ ST_{QF_2}\ then\ {k_i}^1 \ \leq\ {k_i}^2,\ for\ i\ =\ 1, 2, ..., 6. \tag{36}$$

5) The strategy which is best with respect to relation $\ll$ (the minimal element of the lattice) is the strategy ST_{QF^*}. Next the "adjacent" strategies are defined, and it is proven, that if QF_0, QF_1, ..., QF_q is a sequence of such adjacent functions, then corresponding strategies ST_{QF_0}, ST_{QF_1}, ..., ST_{QF_q} are adjacent in the lattice. Therefore, in the class of the ordered search strategies function ST is in a sense a continuous function of function QF: small changes of QF cause small changes of ST_{QF}. If $QF\ \approx\ QF^*$ then behavior of ST_{QF} is close to optimal. If the user is then able to make choices among **all** functions QF then by the way of successive experimental modifications he can approach ST_{QF^*}.

1. Since strategies "depth-first" are very rarely located in the lattice (they have high distances), small changes of QF can cause a "jump" from ST_{QF^*} to a far-away from it lattice element. Analogously, small modification of QF in the direction of QF^* not necessarily need to lead to the improvement of the algorithm's behavior.

6) It can be shown, that in the sense of some of the introduced measures the proposed algorithm is better than the branch-and-bound algorithm investigated in literature [26].

In general case, we should always attempt at finding function QF close to QF^*. The better will be this function, the sooner will the program find a good solution of small value CF_{min} (the decrease of coefficients k_1 - k_4). Therefore, the cut-off of subsequent branches will be done with the smaller value, which will in turn decrease k_5 and k_6. In case of depth-first strategy, the changes in behavior can occur in jumps. In addition, with respect to 3), one has to select function q and with respect to equations 19- 29 respectively define relations on descriptors and states.

By constructing strategies one has also to keep in mind the following.

1. In general case, for those branch-and-bound strategies that search in depth, it is necessary that each branch of the tree either terminates with a solution, or that for each branch determined is certain constructive condition of cutting-off in certain moment (for instance, by reaching certain depth of the tree, or by having no more operators to apply). Lack of these conditions may lead to a danger of infinite or long search in depth, for instance in case of strategy $ST_{d,1}$. This condition is not necessary for A^* Nilsson strategy, which is a special case of the strategy.

2. In the sense of parameters k_1 - k_4, the best performance have strategies that combine properties of strategies $ST_{d,s,s,k}$, A^* Nilsson Strategy, and $ST_{d,s,k}$.

3. In sense of parameter k_7 the 1-child strategies are the best, and $ST_{d,s,1}$ in particular.

4. In case, when we look for a solution on a minimal depth in the tree, the breadth-search strategy theoretically creates the exact solution as the first solution. However, the tree can grow in some cases so quickly, that the strategy cannot be used. Still in some cases it is good to use disk memory. Also, the strategy is useful when the problem is small, or when one can define powerful relations on descriptors or relations on states of the search space.

5. Depth-first strategy allows to find solutions quicker, when the depth is limited or when we can find good upper bounds. This is a good strategy when there are many solutions. It requires not much memory. This strategy is not recommended when the cost function does not grow monotonically along the branches.

6. Strategies ST_{QF} and $ST_{d,s,k}$ allow often the most efficient time of calculations, and the second strategy requires smaller memory.

7. By constructing strategies that use quality functions one has to take into account that the evaluation of more complex function allows to decrease the search, but at the same takes more time. The trade-offs must be then experimentally compared.

8. It is possible to combine all presented strategies, and also to add new problem-specific properties to the strategies. One can for instance create from the depth-first strategy and breadth-first strategy a new strategy that will be modifying itself while searching the tree and finding intermediate solutions. Another useful trick is to cut-off with some heuristic values, for instance some medium value of CF_{min} and $CF_{min\ min}$.

9. An advantage of random strategies is a dramatic limitation of required space and time. These strategies are good to find many good starting points for other strategies that next find local optima.

2.8 SWITCH STRATEGIES

Switch strategies are:

- switch strategies for branches,
- switch strategies for trees.

Switch strategy is defined by using the conditional expression:

$$[\ sc_1 \ \longmapsto\ (MM_1,\ TREE_1), \ldots\ldots., sc_n\ \longmapsto\ (MM_n,\ TREE_n)\ ] \tag{37}$$

where

1. $sc_1, \ldots.., sc_n$ are *switch conditions*
2. $MM_i\ =\ (M_i, ST^i)$ are methods to solve problems by Universal Strategy,
3. M_i are tree methods,
4. ST^i are pure strategies,
5. $TREE_i$ are initial trees of methods MM_i (trees after strategy switchings).

The meaning of formula 37 is the following: If condition sc_1 is satisfied use method MM_1 with initial tree $TREE_1$. Else if condition sc_2 is satisfied use method MM_2 with initial tree $TREE_2$. And so on, until sc_1. This is like COND instruction of Lisp.

In practice, M_i, ST^i and $TREE_i$ are defined by certain *changes* to the actual data. These can be some symbolic transformations, or numeric transformation. Also, these can be selections of new data structures. Therefore one has to declare the initial data: M_0, ST^0 and $TREE_0$.

- In Switch Strategy for a Branch, the conditions $sc_i, i = 1, \ldots n$ are verified in *the moment of creating a new node* (or a new node being a solution, being a solution better than the previous one, etc - the type of the node is specified by the parameters).
- In the Tree Switch Strategy the conditions are checked after some type of full tree search has been completed.

In both types of strategies, the conditions of switching strategies can be defined on:

- nodes NN,
- branches leading from N_0 to NN,
- expanded trees.

There can exist various *Mixed Strategies* STM, defined as follows:

$$STM = (SST_T, SST_B) \tag{38}$$

where:

SST_T - is a Tree Switch Strategy,

SST_B - is a Switch Strategy for a Branch.

For both the switching strategies for tree, and switching strategies for branches there exist eight possible methods of selecting changes, specified by one of subsets of set of data $< M_i,\ ST^i,\ TREE_i >$ In a special case, by selecting an empty set, changes of M_i, ST_i or $TREE_i$ are not specified. This corresponds to pure strategy ST_0 (which was declared as the first one). As we see, pure strategies are a special case of switch strategies.

Analogously, complex methods, defined as $CM = (M_1, ..., M_r, STM)$ are generalizations of methods MM_i.

Now we will present changes of $TREE_i$, M_i, and ST_i.

1. We considered the following changes of $TREE_i$.
   - change of coordinates of nodes (locally, or in a branch, or in the whole tree),
   - adding or removing some coordinates (locally, or in a branch, or in the whole tree),
   - cut-off of the tree.
2. Changes M_i by switch strategy can be executed by specifying new components of the solution space.

   An example of switching strategy that changes both $TREE$ and M is a strategy for graph coloring - see section 5.
3. Strategy is changed by determining the change of strategy parameters. Change of strategy for switching strategy can consist for instance in a permutation of list $OPEN$, a selection of some its subset, or in some modification to list $OPERT$. Since the whole necessary information about the solution tree is stored in list $OPEN$ - after the Branch Switch the new strategy can start working immediately. The Main Universal Search Subroutine is constructed in such a way, that even by applying the switch search strategy the obtaining of the exact solution is still possible.

Examples of Switch Strategies

- **The Far-Jumps Strategy.** An example of switch strategy for branches is a strategy of "far jumps", which first finds solutions of high distance in the solution space. At first, the Breadth-First Strategy with macrooperators and dominance relations is used to develop a partial tree. Together with each node N of the tree we stored also its level in the tree, $SD(N)$. Selected is a node from $OPEN$ that has the smallest level. Next the "depth-first" strategy is used until the first solution is found. The program evaluates, using some additional method, whether this is a minimum solution, or a good enough solutions. When it is evaluated that this is not the minimum solution, the "strategy switch" is executed. The strategy switch consists in selection from list $OPEN$ the node of the actually lowest level, and next adding this node at the beginning of this list. Starting from this node, the tree is expanded again using the depth-first strategy, until the next solution is found, etc. With each solution, the order of nodes in $OPEN$ can be modified.

- **The Distance Strategy.** An advantage of the switch strategy called "distance strategy" is, that in contrast to the "depth-first" or other pure search strategies, the successively generated solutions are placed far away one from another. This gives the possibility of "sampling" in many parts of the space, which can lead to finding good cut-off values quicker (thanks to jumping out of local minima of the quality function).

- **The Strategy of Best Descriptors**. The Strategy of Best Descriptors consists in storing for some pure strategy (for instance the depth-first, or ordered-search) all or some, of the descriptors that proved to be the most useful by finding the previous solution. For instance, the dominating descriptors, or the descriptors with the highest values of cost or quality functions are stored. After switch, these descriptors are placed at the beginning of list $OPERT$, and are thus used as the first ones in the next tree expansion. The switch strategies of this type can be in some cases applied for fast finding of good cut-off values in branch-and-bound strategies.

- An example of switch strategies for trees can be a **"Strategy of Sequence of Trees".** It expands some full tree, or a tree limited by some global parameters (time, number of nodes). Next, using some additional principles, it selects few nodes of the expanded tree (for instance, the nodes with the minimum value of the cost function). Next, the strategy expands new trees that start from these nodes. It usually uses a different set of components of the space, and/or pure strategy in these new trees.

- In particular one of strategies is to select another set of descriptors.

- In another strategy of this type CF_{min} is calculated as some function of CF_{min} and other parameters, including the probabilistic evaluations of $CF_{min\ ,min}$, during the moment of switching. This strategy is not complete, but it can substantially limit the search by backtracking at the smaller depths.

QUESTIONS FOR SELF-EVALUATION

1. What are the main concepts in tree search?

2. What is the difference between state space and search tree?

3. For problems from the previous section, specify each tree-search process in terms of states, operators, coordinates and strategies explained in this section.

4. Do any of the problems from the previous section require modifications to the main Universal Search Strategy? What problems?

5. Create Search Methods for the Maximum Clique Problem:

   - searching starting from the smallest cliques,
   - searching starting from the largest cliques,
   - searching starting from cliques of certain size,
   - searching starting from certain clique and next cliques that differ from small amount and numbers of nodes from it.

6. Repeat point 4 above for the Problem of Finding the Bound Set of Variables.

7. Show full trees of the following types:
   - a tree of all subsets of a given set - type T_1.
   - a tree of all permutations of a given set - type T_2.
   - a tree of all one-to-one functions from a set A to set B - type T_3,
8. How to modify the strategy programs given above that they will include genetic algorithm?
9. How to modify the strategy programs given above that they will include simulated annealing?
10. Explain in more detail the possible pure and mixed strategies that have been only mentioned in the last two sections.
11. Using the framework of creating strategies shown, can you create a superior Universal Strategy?
12. Discuss the role of functins CF, QF, f and q. Compare and illustrate with an examples of problems from this and previous sections.
13. For type T_1 tree generator of a full tree of the set of all subsets of the set, discussed above, what are the possible applications in the combinatorial problems that occur in functional decomposition? List all of them.
14. Create a tree like T_1, but starting from an arbitrary subset of a set. Use a small adaptation of rules for T_1.
15. How to create a search program that will find all k-element subsets of a set with K elements?
16. How to create a search program that will find all k-element variations of a set with K elements?
17. How to create a search program that will find all k-element combinations of a set with K elements?
18. Give examples of descriptors.
19. What are the types of descriptors? Give examples of applications.
20. Give examples of macro-operators.
21. For each tree from Figure 2 do the following:
   - explain what is the set created in this tree.
   - explain what would be all possible applications of this set or this tree.
   - write the specification of the full tree generator.
   - draw an example how the tree can be expanded to larger dimension using your generator. explain what would be a possible application of this set or this tree.
22. For each tree from Figure 3 do the following:
   - explain what is the set created in this tree.
   - explain what would be all possible applications of this set or this tree.
   - write the specification of the full tree generator.
   - draw an example how the tree can be expanded to larger dimension using your generator. explain what would be a possible application of this set or this tree.

3 METHODOLOGY TO CREATE TREE SEARCH PROGRAMS

3.1 PROBLEM-SOLVING MODEL

The following is a model that describes how the developer, starting from the most general problem formulation, should proceed to create the completely finished search program.

3.1.1 Problem Formulation

The first task is to *formulate a given problem* to be solved according to the definition of the problem given in section 2.

- *The types of given objects* (sets, functions, graphs, etc.) as well as the types of objects sought by the program, should be specified. If these objects are not the standard objects of the MULTIS system, they should be defined or the problem needs to be re-formulated in such a way that it will be possible to use the system's standard objects.
- The developer should describe the problem conditions using these structures. He or she should also define the cost function, if it exists. It should be considered, how the problem conditions and the constraint conditions can be decomposed or re-formulated.
- It should be determined, whether the conditions need to be checked separately or jointly.
- The developer has also to determine the order in which the conditions should be checked. Let us assume, for example, that a set of objects being generated by generator G that satisfy the problem conditions C_1 and C_2 need to be found.
  1. One of the methods would be to generate the set S_1 out of all the objects generated by G that meet the condition C_1 first, and then to test the S_1 objects for the satisfaction of condition C_2.
  2. The procedure in another method would be identical except that the conditions would be reversed. Subsequently, the objects could be generated one at a time using generator G and both conditions checked for each generated object. In this case, we also have to decide on the order of checking the conditions.
  3. Finally, another generator that would generate only those objects generated by G that pass the C_1 test, etc. can also be constructed.

Many methods can be derived from the direct problem formulation based on the methodology provided here. The developer then determines which one is best, and in some cases creates a parametrized version to leave the final decisions to the user of the system for its testing on numerous benchmarks.

3.1.2 Creating the Method to Solve a Problem

The developer who already understands the problem structure well, can subsequently move on to the stage of formulating the *method of solving the problem.* In doing so, it is recommended that the *similarities of the problems* and the *possibilities of reducing problems to other problems* be systematically and intentionally applied, such as:

- is it a covering problem?
- is it an ordering problem?
- is it an constraing-satisfaction problem?
- is it a path search in some graph?

- is it a partitioning problem?
- is it an encoding problem?

The similarity of problems and the reduction of combinatorial problems offer an experienced developer many opportunities, which include using the already existing subroutines of the system for new problems. A number of useful reductions is presented for instance in [30]. Other reductions for digital design problems are discussed in [20, 44, 56, 77] and [57].

When the developer considers the combinatorial problems that are useful for decomposition, he should specify whether the new problem is *isomorphic, similar,* or *reducible* to one of the familiar problems. If such a relation between problems is found, it should be used, for example, for such applications as:

- restricting the solution space,
- constructing the generator for this space,
- proving theorems related to cutting off or to equivalences in the tree,
- detecting and utilizing symmetries,
- creating the quality and cost functions,
- specifying relations on descriptors or nodes in the tree,
- specifying other method conditions.

If only one similarity exists in the problems, such as: the cost function, the generator of the space, the conditions, the operators, then, using this similarity, the attempt should be made to re-formulate the problem in such a way that the number of similarities is increased. If there are no similarities and the problem cannot be reduced to some well-known problem, an attempt at decomposing the problem into well-known sub-problems can be made. In this way, part of the problems, or perhaps all of them, should be isomorphic, similar or reducible to the well-known problems.

It is also often useful to find a different problem that is simpler than the original problem, and for which one of the techniques mentioned above can be applied.

Finally, such problems can be transformed by modifying new problem conditions, or by adding new problem conditions, in order to conform with one of the previously mentioned cases. This can lead to another problem with respect to the specified earlier definition of the problem. This is often the case because the actual problems are not precisely formulated.

The successful application of the above techniques can simplify the selection of the generators and the conditions of the solution space, and make it easier to work with the prototype development later on.

Until now, we used most of these techniques in one or another form in MULTIS development, but we did not analyzed them systematically yet.

3.1.3 Precise definition of the algorithm

The next step is to formulate the *tree-search algorithm* precisely. The existing point subroutines of MULTIS, such as Graph Coloring or Boolean Operations on BDDs, determine certain *space of methods.* The task of the application developer is to determine how to link them in the most efficient way. A certain subspace in the space of methods should, therefore, be systematically investigated.

All of the steps taken are intended to define the method that is computationally efficient. Three basic directives result from the above general assumption:

1. The solution space needs to be limited by constructing a good tree generator.

Six Possible Cases to Improve Tree Search

	Specification of Tree Generation and Conditions	Selection of Strategy
Limiting the state-space	M1	M4
Specifications of the best way of extending nodes	M2	M5
Decrease of the processing time and the memory used for each node	M3	M6

Figure 4: Six Possible Cases to Improve the Tree Search Methods

2. The best possible way of extending the nodes in this space (order of generation) should be found.

3. The processing time and the memory used for each node should be decreased.

These directives can be complied with by specifying the corresponding generators and conditions for the solution space, or by the selection of strategies. Therefore, six possible cases of improving the tree search have to be dealt with, as shown in Table from Figure 4.

3.1.4 Discussion of methods to increase the search efficiency

Six methods to increase the search efficiency are outlined in detail below. Bear in mind that they are all mutually related.

Case M1.

In each problem, the developer should focus first on maximally decreasing the solution space, by specifying the corresponding generators and conditions. To achieve this, the developer should consider the way in which the states need to be represented, so as to cut-off and check the conditions in these states in the most efficient manner.

- What should the initial node be?
- What should be the coordinates, the operators, and the descriptors?
- What should the additive cost function be? The additive cost function should be formulated in such a way that it can be used for cutting off. Still, it may be more or less precize.

While maintaining a completeness of the strategy, the developer should focus on proving the theorems related to limiting the space.
These can be of the following types.

1. **The theorems to determine the solution space.** Sometimes it can be proven that some space S' exists, $S' \subset S$, such that $SS \subset S'$ (SS is a set of solutions). The theorems concerned with determining the solution space can be of two types. They either lead to a space-constructing mechanism that is built into a special tree generator, or they lead to new constraint conditions that are tested for nodes or descriptors.

2. **The theorems about the limit parameters that specify the solution tree.** In some problems, such parameters as SD_{max} or $CF_{min\ min}$ can be determined. Let us assume, for example, that a single, minimal coloring of a planar graph needs to be found, and it was shown that no solution of cost 3 exists, because a clique with 4 nodes exists in the graph. It is a known

mathematical fact that a planar graph can be colored using 4 colors. Hence we set the control parameter $CF_{min\ min} = 4$ and when a solution with 4 colors is found, the algorithm terminates.

3. **Theorems about relations on descriptors.** Often, relations of subordinance, dominance, equivalence (local and global), inconsistency, symmetry, or implication, can be found and proven. The utilization of analogies of problems is useful (see for example in [54] how analogies can be used to solve efficiently the highly cyclic covering problems). It needs to be ascertained whether the relations on the descriptors are independent, and also consider the possibility of checking only some of the subsets of those relations.

4. **Theorems about the relations between the tree nodes.** The relations of identity, isomorphisms, symmetry, and the similarity between the nodes in the solution tree should be considered, [54].

5. **Theorems about the construction of generators.** The possibility of moving the constraint conditions to the tree generator itself should be taken into consideration. Instead of first generating nodes and later removing them, fewer nodes would be generated in this approach. If the subset of a certain set is sought that does not fulfill a certain restricting condition W, for example, then, perhaps, instead of generating a tree of type T_1 and checking the condition W in the nodes of the tree, it is more reasonable to construct a more efficient generator that would only generate the subsets of the set that do not meet condition W. However, this may be a more time consuming way. Therefore, trade-offs must be thoroughly considered by the developer.

Case M2.

In this case, one contemplates which of the elements in the state space has to correspond to the initial node of the solution space. In the problems concerned with finding a path, for instance, the search can be performed from the final situation to the initial situation. The components discussed in the types 3,4 and 5 above also influence the search method as well as the ordering of the auxiliary sets (see example in [55]). In the problem of selecting a subset of a set (for instance a set of bound variables, or maximum clique) one can start from the largest, the middle, or the smallest elements of the lattice of subsets as the starting point.

Case M3.

The processing time can also be decreased for the following reasons:

a. **Selection of the corresponding order of checking the problem conditions and constraint conditions** while generating new nodes. This is also related to specifying the order in which the coordinates of the state vector are calculated. The problem conditions or constraint conditions can either be calculated jointly or the conditions can be decomposed to several successively calculated conditions. These are calculated intermittently using the values of some coordinates to allow for backtracking sooner. The various placement possibilities are considered along with the decomposition of conditions and the influence that they have on the cases M1 and M3.

b. **Formulation of weaker relations on descriptors** (which is opposed to the directive regarding the limiting of the solution space). Let us consider the following example as an illustration. It could be possible to generate a small tree by applying strong relations on descriptors but this could then take longer time because the relations would then be checked slower. On the other hand, replacing this strategy with one that applies weaker relations that are rapidly checked produces a larger tree but one in which each node is extended faster. The total processing time of some problems can, therefore, be decreased if no memory limitation exists.

c. **Simplification of the quality functions** (which is opposed to the directives resulting from M4 and M5). The argumentation for this case is similar to that in "b" above.

d. **Specification of the type of nodes that are suitable to be checked in the relations on nodes** (it is sometimes sufficient, for example, to investigate only the nodes of the same depth or in the same branch of the tree as the current node N).

e. **Selection of the** *additively calculated* **cost function and quality function** in such a way that they are both mutually related. See discussion in [46, 54, 55].

f. **Selection of the corresponding data structures of the lowest levels and the auxiliary functions** influences both the processing time and the memory size. In some problems, for example, it is reasonable to represent sets as the lists or vectors, and in other problems to represent them as binary words in which each set element corresponds to one bit of the computer word.

Case M4.
The solution space can be limited by the selection of the strategy. The only possibility for this in the case of complete strategies is the application of the cutting-off principle using the cost function. Then the problem is reduced to M5 in order to find the quasi-optimal solution as quickly as possible. The cutting-off, however, can result from the selection of some local strategy parameter values for incomplete strategies.

Case M5.
In this case, it needs to be considered whether some of the basic strategies match the specifics of the problem. The values of each strategy parameter, different combinations of parameter values, and the quality functions for states and operators should be considered. The possibility and the rationality of focusing on matching the quality function to the cost function should be analyzed so that they meet the special properties mentioned previously. When the characteristic of the problem is known, it should be determined which of the strategies discussed fits best the specifics of the problem. Applying the subroutines for ordering and selection should be considered by the developer, and next by the user, while selecting the corresponding strategy parameters.

Case M6.
If there is not enough memory for larger data, one should consider the following possibilities:

a. Changing the strategy (out of the complete strategies, the best is the Depth-First Strategy With One Child). If this is not sufficient, the Random search strategy should be applied or the disk memory for part of the tree should be used.

b. Increasing the number and the power of the cutting off rules that are controlled by the strategy parameters (this can lead to losing the completeness of the method). Such means can also reduce the calculation time.

With respect to the cases shown above, the developer can expand the concept by experimenting with the program from the simplest to the more complex problem descriptions, and from the standard strategies to those especially created strategies for the problem. These non-standard strategies could be for instance made by replacing the standard parameters and sections with the non-standard parameter values and code sections. Analysis of some of the above cases is important not only with respect to the efficiency, but is also very useful for the clarity of the results and the required interface to other programs. If the complete tree is being extended, for instance, the strategy does not affect the efficiency. The selection of the strategy is not irrelevant, however, when a certain order of generating the objects is mandatory or expected.

The methodology outlined above has allowed us to create several classical logic design algorithms in the past [38, 44, 50, 55, 57, 61, 62]. These include state minimization or various PLA minimization algorithms created by formal transformations of space generators and conditions of the direct problem description. Several new algorithm variants for these problems have also been derived using the same methodology. Therefore we believe that it will also be useful for functional decomposition, but this still must be verified by programs that we are in the process of developing.

3.2 EXPERIMENTING WITH DIRECTIVES BY THE DEVELOPER, AND BY THE USER

After having introduced in previous subsections the methodology of creating search programs, the manner in which the tree searching programs can be improved by experimenting with changes in various description sections will be discussed. This has already been illustrated to some extent in section 3.1. These principles will be further illustrated using several examples of problem descriptions.

The advantage of the state-space methods is the natural manner in which they are described and their similarity to the human problem-solving techniques. This allows the introduction of various *heuristic rules* to the state-space generation process that are direct and easy to modify. The prototype programs created as recommended in previous sections are flexible and adaptable and can be modified by replacing the section codes in problem descriptions, problem and solution conditions, the search strategy parameters, and other sections or by adding new classes derived from previous classes. The modification process is *orthogonal* and *incremental* in nature, which means that the changes in various segments of the code can be done separately, and one at a time. They exhaust the space of search methods.

3.2.1 Heuristics

A "heuristic" is any method, technique, or directive, that, in general, leads to a solution. We can only assume, with a certain degree of plausibility, that a given *heuristic directive* produces the desired results. In our case, the heuristics are expressed as subroutines, program segments, parameter segments, or computational mechanisms that expedite the generation of the optimal solution (or quasioptimal solution, if so desired) and even enables the developer to find a solution that would otherwise have been impossible.

If a heuristic directive is proven to lead to optimal solutions all of the time, it can be classified as a *methodic directive* and would no longer be referred to as a heuristic. It is usually necessary, however, to have both types of directives because, without directives, the enormously large state-space would often make finding solutions impossible. The methodic directives are based on the strict formal analysis and can relate to certain parameters that limit the solution space or that order the descriptors in the solution space nodes, or that order such nodes themselves in the $OPEN$ list. The solution method is complete when both the tree generator and the strategy are complete. The search efficiency can be increased through independent attempts to ease the condition governing the method's or the strategy's completness. This independence is advantageous when experimenting with variants of the program and with various sizes of data for it.

Heuristic directives of the type being considered here can be divided into three categories:

1. *Heuristic directives* that correspond to the manner in which the problem is formulated. These directives determine the state-space. The introduction of additional problem conditions (or reasonably motivated constraints) can essentially reduce the size of the state-space.

2. *Heuristic directives related to the manner in which the problem is represented in the solution space.*

3. *Heuristic directives that organize the search process.*

When it is not possible or reasonable to specify a complete solution space, we look for the heuristic directives that attempt to achieve the same goals as the methodic directives that were described in section 3.1. Each of the cases M1 - M6 has its counterparts in heuristic directives.

The categorization of any particular heuristic is not unique. It is often useful, when seeking heuristic directives, to rely on some analogies of the problems, and to use the analogies of the methods applied in their solutions. The existence and type of the heuristic directives influences the speed and the quality of the solutions generated. The developer should look for the best trade-offs. The analysis of various variants of programs for the same application also reveals the trade-offs between the enumeration and the knowledge-based reasoning that has already been presented. The initial, direct problem description uses only the problem conditions. The initial program does not, however, need to be, and usually is not, the most efficient one. It suffices if it works. It is obtained quickly so more experience is gained by working with it. A very large tree is generated and displayed. The developer can analyze the printouts of the large trees and suggest changes based on redundancies, symmetries and other information found in these trees. The obviously redundant or non-optimal sub-trees are noted. The analysis, as to why they were generated, leads usually the developer to the introduction of new problem conditions. Often also to formulation new heuristic directives, and sometimes also methodic directives. Incremental changes lead to new variants, the behavior and speed of which are analyzed in order to find the optimal trade-offs between the speed and the quality of results. Limiting the search can sometimes go too far - solutions are lost or the search time increases because the manner in which the partial solutions are evaluated becomes too complex. In such cases the developer should withdraw incrementally from some of his previous decisions. In some other cases, the developer overconstrains first the method description, and then he will have to experiment with sets of less constrained descriptions.

3.2.2 The Process of Fast Prototyping

Constructing the program is a complex process in which three phases are iterated:

- problem definition,
- construction of the algorithm, and
- coding of the algorithm.

Coding itself is not the most time consuming task of programming. Much more time is spent on the problem formulation, testing of variants, debugging, trying various program segments and modifications in the data structure, testing various controls of programs, analyzing the usefulness of heuristic directives, preparing and modifying documentation, etc. Usually the algorithm is initially not known. It is the goal of this report to facilitate the task of finding the correct algorithm and implementing the program. Experimenting with the program towards this end should be simplified by planning and keeping each individual phase independent from the other as much as possible. In order to solve a problem, when aided by the computer, the developer must have the knowledge required in the three phases outlined above. Since these phases of the design process, to varying degrees, are contained in the program experimentation process, the developer needs to bear in mind the aspects of the problem and method description already mentioned. The entire prototype development should consist of the sequence of the vertical transformations described in section 3.1 and the horizontal transformations from this section, interleaved with experimental variant evaluations.

Heuristic directives are learned with the experience of solving a great number of problems that are repetitive in nature. In fact, one of our methodology postulates is that *such learning is facilitated and expedited by the problem solver's use of the appropriate programmed computer and not only through the use of pencil and paper.*

Observing human techniques of solving problems we see that people apply several intuitive hypotheses that are often informal analogies to the cutting-off methods of our more formal search model. These hypotheses are often formed based on induction from observing examples of how the program behaves. Sometimes, with new methods, the representation of the problem is also modified or even totally changed. These hypotheses are often not true, not always true, or not sufficiently founded. They play a very fundamental role, however, in the trial-and-error process of constructing the final version of the prototype program. The methodology proposed here has the aim of *improving the fundamental structure of human heuristics by providing better proof based on the model of solution space.*

We believe that such an experimental search environment helps to design algorithms that are experimental in nature and must, therefore, be done in an experimental manner. We observed several times the dogmatical tendencies with which the students, engineers, CAD tool developers and even researchers view algorithms out of textbooks and research papers. Creating an algorithm systematically and playing with many variants of the algorithm crushes these dogmatical tendencies in them, and makes them more critical and also more creative. The proposed methodology encourages to look for new ways of solving problems.

A reader may ask, is it worthy to analyze all these problems that carefully, and with such a detail? Is it worthy? How much improvement will it bring practically to the Functional Decomposer? In our opinion, this analysis is of very big importance since it results from our preliminary tests of our decomposer that small changes in decomposition strategies can lead to dramatic changes in solution cost, decomposition times, and their mutual trade-offs (see section 13 and **??**).

QUESTIONS FOR SELF-EVALUATION

1. Select one combinatorial problem that is not trivial and that you know really well, possibly you wrote a program for it, or at least created an algorithm. Follow the methodology outlined above to create as many as possible new ideas and methods to solve this problem.
2. Follow the methodology outlined above to create as many as possible new ideas and methods to solve the Set Covering Problem.
3. Follow the methodology outlined above to create as many as possible new ideas and methods to solve the Graph Coloring Problem.
4. Follow the methodology outlined above to create as many as possible new ideas and methods to solve the Column Minimization Problem.
5. Follow the methodology outlined above to create as many as possible new ideas and methods to solve the Parallel Decomposition Problem.
6. Follow the methodology outlined above to create as many as possible new ideas and methods to solve the Input Encoding Problem.
7. Follow the methodology outlined above to create as many as possible new ideas and methods to solve the Output Encoding Problem.
8. Follow the methodology outlined above to create as many as possible new ideas and methods to solve the Variable Partitioning Problem.

4 EXAMPLE OF APPLICATION: THE COVERING PROBLEM

The following examples of some partial problems of MULTIS will illustrate the basic ideas involved in the state-space search. It will also show the methods used to formulate problems for multi-purpose search routines like those used, or to be soon used, in MULTIS.

4.1 THE FORMULATION OF THE SET COVERING PROBLEM

This problem is used in Column Minimization. It is also widely encountered in logic design (among others, in PLA minimization, test minimization, multilevel design - see many recent examples in [77]). Let us consider, as an example, the covering table shown in Fig. 5.

	1	2	3	4	5	6	
1	1	1	0	0	0	1	1
2	0	1	0	1	1	1	1
3	0	0	1	1	0	0	1
4	0	0	1	0	1	0	1
5	1	1	0	1	0	0	1

=

	1	2	3	4	5	6
1	X	X				X
2		X		X	X	X
3			X	X		
4			X		X	
5	X	X		X		

Figure 5: A Covering Table With Equal Costs of Rows

Each row has its own cost indicated by the value in the parentheses beside it. In this example, they are all equal. An **X** at the intersection of row r_i and column c_j means that row r_i covers column c_j. This can be described as:

$$(r_i, c_j) \in COV \subset R \times C, \tag{39}$$

or briefly, by $COV(r_i,\ c_j)$.

A set of rows which together cover all the columns and have a minimal total cost should be found.

The direct problem formulation is as follows:

1. **Given:**

   a. the set $R = \{\ r_1\ , r_2\ ,..., r_k\ \}$ (each r_i is a row in the table);

   b. the set $C = \{\ c_1\ , c_2\ ,..., c_n\ \}$ (each c_j is a column in the table);

   c. the costs of rows $f1(r_j)\ ,\ j\ =\ 1, ..., k$;

   d. the relation of covering columns by rows is $COV \subset R \times C$.

2. **Find** :

   Set $SOL \subset R$;

3. **Which fulfills the condition:**

$$(\forall\ c_j\ \in\ C)(\exists\ r_i\ \in\ SOL)[COV(r_i, c_j)] \tag{40}$$

4. **And minimizes the cost function:**

$$f2\ =\ \sum_{r_i\ \in\ SOL} f1(r_i) \tag{41}$$

It results from the above formulation that the state-space $S = 2^R$. This means that $SOL \subset R$. Hence, it results from the problem formulation that all the subsets of a set are being sought. Then, according to the methodology, the standard generator, called T_1, that generates all the subsets of a set is selected. Operation of such generator can be illustrated by a tree.

The previously mentioned relation RE on the set $S \times S$ can be found for this problem and used to reduce searching for a respective search method. It can be defined as follows:

$$s_1 \; RE \; s_2 \iff s_2 \supset s_1 \tag{42}$$

Therefore, when a solution is found a cut-off occurs in the respective branch.

There exists for each element $c_j \in C$ an element $r_i \in SOL$ such that their relation COV is met. In other words, r_i covers c_j which means that the predicate $COV(r_i,\ c_j)$ is satisfied. The cost function F assigns the cost to each solution. In this case, this means that $F = f2$ is the total sum of $f1$ costs of rows from the set SOL. Using the problem definition from section 2, the covering problem can be then formulated as the problem

$$P \;=\; (2^R, \{p_1\}, f2), \tag{43}$$

where

$$p_1(SOL) \;=\; (\forall\ c_j \in C)\ (\ \exists\ r_i \in SOL)\ [COV(r_i, c_j] \tag{44}$$

Tree Search Method # 1

The initial tree search method is based on the direct problem formulation, is then as follows:

1. The initial node N_0: $(QS,\ GS,\ F) := (\emptyset,\ R,\ 0)$.

2. The descriptors are rows r_i. The application of the operator is then specified by the subroutine:

   $O(N,\ r_i) =$

   $[\quad GS(NN) \ := \ GS(N) \ \setminus \ \{r_i\},$

   $QS(NN) \ := \ QS(N) \ \cup \ \{r_i\},$

   $CF(NN) \ := \ CF(N) \ + \ f_1(r_i).$

   $]$

3. Solution Problem and Condition (cut-off type):

$$p_1(NN) \;=\; (\forall\ c_j \in C)\ (\exists\ r_i \in QS(NN))\ [COV(r_i,\ c_j)] \tag{45}$$

Comments.

1. NN denotes a successor of node N.

2. $QS(N)$ is the set of rows selected as the subset of the solution in node N.

3. $F(N)$ is the cost function for node N. This is the total sum of costs of the selected rows from $QS(N)$.

As we can see in this problem, the formulation of the additive cost function is possible.

An example of the cover table is shown in Fig. 5. In this example, for simplification, equal cost of rows are assumed. The method can, however, be easily extended to arbitrary costs of rows. The solution tree obtained from such a formulation is shown in Fig. 6.

The nodes of the search tree are in the ovals. The arrows correspond to the applications of operators, and each descriptor of operator stands near the corresponding arrow. The solution nodes are shown in

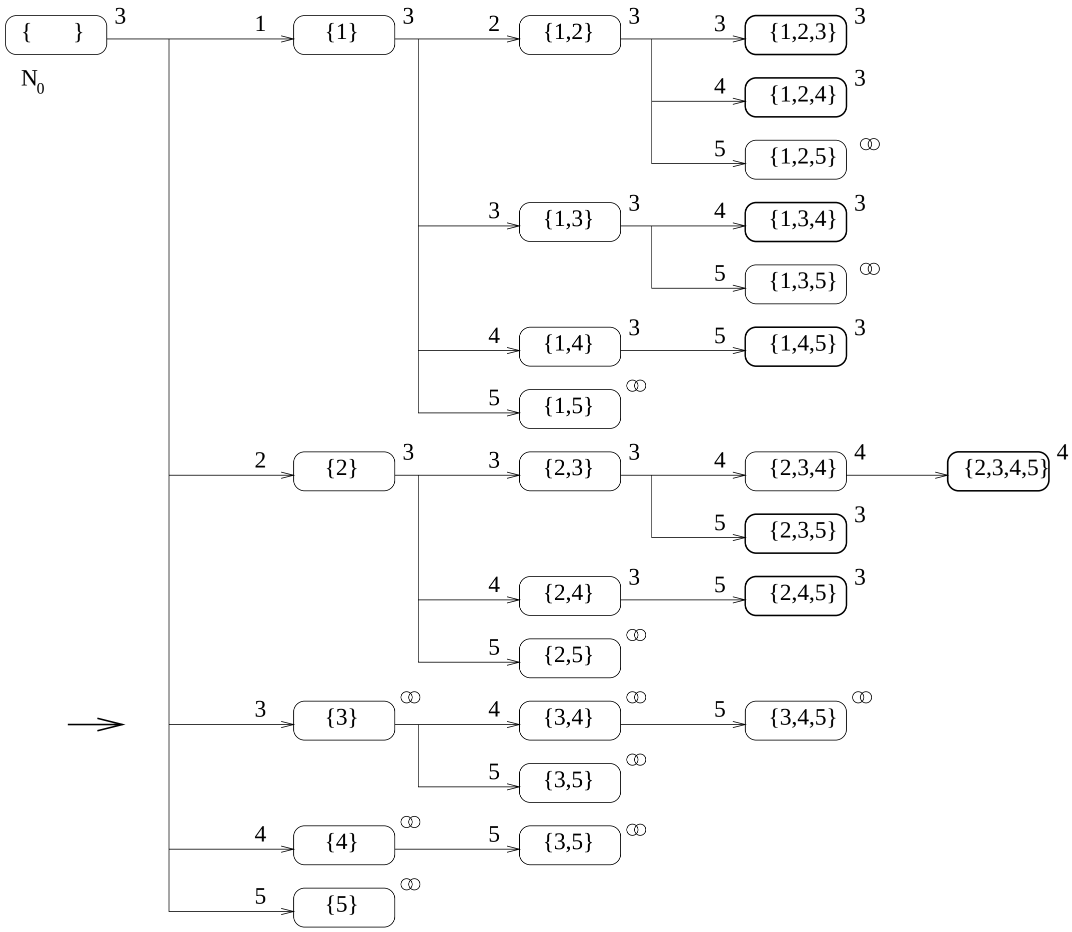


Figure 6: First Search Method for the Table from Figure 5

bold ovals. The costs of nodes are outside the ovals, to their right. The sets inside the ovals correspond to partial solutions in the nodes. Since the entire tree has been developed here, the sets GS for each node can be reconstructed as the sets of all descriptors from the outpointing arrows.

The cutting-off is realized based on the fact that the cost function increases monotonically along the branches of the tree; this is the cut-off condition. The nodes that are the solutions are therefore not extended. If the cut-off conditions were not defined, for example, the nodes {1,2,3} and {1,2,4} would be extended. Otherwise, the tree is produced under the assumption that the cutting-off is not done for the solutions with cost function values worse than for those nodes previously calculated. The values of the function f are shown next to the nodes.

Observe that the nodes in this tree are extended (for example, the node {3}) in such a way, that solutions may not be produced in their successor nodes. Based on the fact that each column must be covered by at least one row in the node, the generation of such nodes can be avoided. This is done by storing the columns c_j that are not yet covered in set AS. The branching for all rows r_i that cover the respective column for each individual column is also generated. These rows are such that $COV(r_i,\ c_j)$.

We can now formulate a new tree search method:

Tree Search Method #2

1. Initial node N_0:

$$(QS,\ GS,\ AS,\ CF)\ :=\ (\emptyset, \{r_k \in R\ \mid\ COV(r_k,\ c_1)\},\ C,\ 0) \tag{46}$$

   The first element of C is denoted by c_1 above.

2. Operator:
   $O(N,\ r_i) = [$
   $QS(NN)\ :=\ QS(N)\ \cup\ \{r_i\},$
   $AS(NN)\ :=\ AS(N)\ \setminus\ \{c_j \in C \mid COV(r_i, c_j)\},$
   $c_j\ :=$ the first element of $AS(NN)$,
   $GS(NN) := \{r_k \in R\ \mid\ COV(r_k,\ c_j)\ \},$
   $CF(NN) := CF(N)\ +\ f_1(r_i)$
   $]$

3. Solution condition (cut-off type):

$$p_1(NN)\ =\ (AS(NN)\ =\ \emptyset) \tag{47}$$

The corresponding tree is shown in Fig. 7.

Two disadvantages to this method become apparent from Fig. 7. The first one is in creating the redundant descriptor 4 in $GS(N_5)$ that can not be better than descriptor 2. This disadvantage can easily be overcome by using the new section code that defines the domination relation on descriptors. The second disadvantage is due to the repeated generation of the solution {1,3,4}, the second time as {1,4,3}. If the optimal solution is desired then there is no way to avoid the inefficiency brought about by the Tree Search Method #2.

Tree Search Method #3

Another method to avoid generating nodes for which $f = \infty$ is the application of the first method (the generation of the T_1 type of tree) and an additional filtering subroutine to check nodes to verify if the set of rows from $GS(N)$ covers all the columns from $AS(N)$. In addition, the following code of type *"Actions on the Selected Node"* is created:

If

$$AS(N)\ \not\subset\ \{c_j\ \in C\ \mid\ (\ \exists\ \ r_i \in GS(N))\ [COV(r_i,\ c_j)]\} \tag{48}$$

then

$$GS(N)\ :=\ \emptyset;$$

This means, that the cut-off is done by clearing set $GS(N)$ when the set of all the columns covered by the available descriptors from $GS(N)$ does not include the set $AS(N)$ of columns to be covered. For example, at the moment of generation shown by the arrow in Fig. 6, the set of $GS(N_0) = \{3,4,5\}$ and it does not cover $AS(N_0) = C$. Therefore, it is assigned $GS(N_0) := \emptyset$, and the generation of the subtree terminates. This constitutes the Tree Search Method #3.

Tree Search Method #4

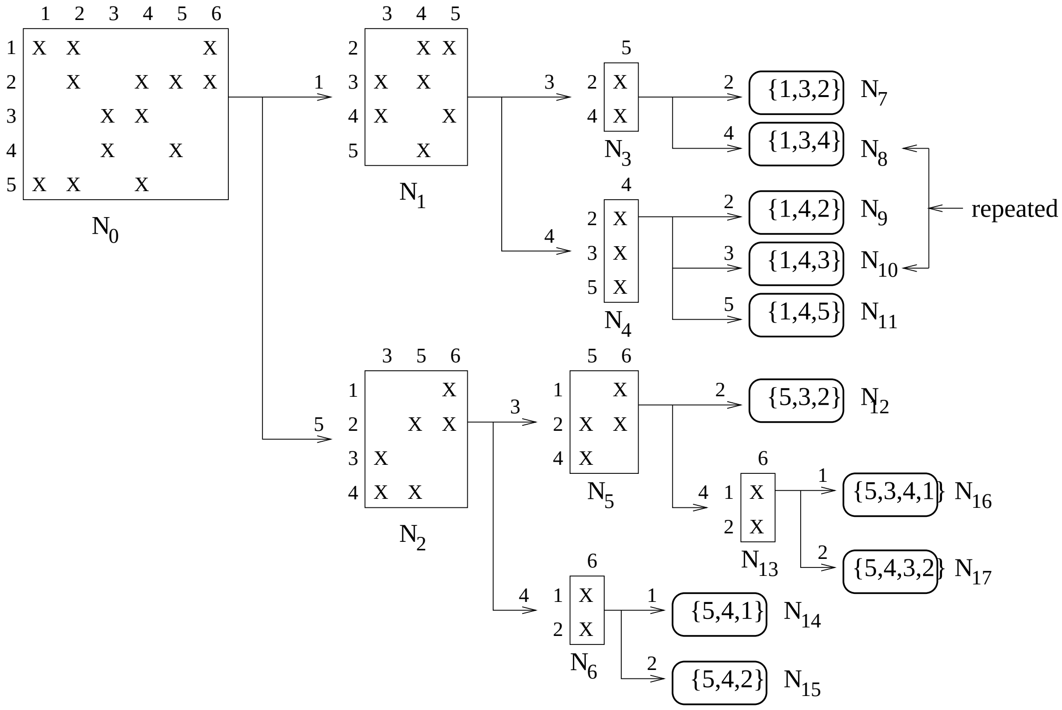


Figure 7: Second Search Method for the Table from Fig. 5

The generated tree can be decreased even further when the second method is used and it is declared in the operator that:

$$GS(NN) \ := \ \{r_k \in GS(N) \ \setminus \ \{r_i\} \ \mid \ COV(r_k, c_j)\} \tag{49}$$

Above, $\setminus$ denotes operation of set difference.

This approach, Tree Search Method #4, can lead however to the loss of the optimal solution. It is then a typical *heuristic directive* and not a *methodic directive* like those discussed previously.

In both trees, the cutting off condition based on the cost function has not yet been considered. If the solution {1,2,3} in the tree shown in Fig. 6 were first found, node {2,3,4} could be cut off, and the non-optimal solution {2,3,4,5} would not be generated. Until now, however, only methods of constructing the generator of complete, non-redundant trees have been presented. These are the trees calculated for the worst case of certain rules and heuristics that will be discussed in section 4.2.

4.2 SEARCH STRATEGIES

Various search strategies can now be illustrated to give the reader an intuitive feeling for the concepts and statements introduced in the previous sections.

The node enumeration order from Fig. 7 corresponds to the Breadth-First strategy, and to the strategy of Equal Costs (with respect to the equal cost of rows applied in this example). Eight nodes were generated in node N_7 to find the optimal solution {1,3,2}. The optimality of the solution {5,4,2} was proven after creating node N_{15}, which means, after generating 16 nodes. Nodes N_7 to N_{15} were temporary. Cost-related backtracking occurs in node N_{13} and, therefore, nodes N_{16} and N_{17} are not generated.

The strategy Depth-First generates the nodes in the order N_0, N_1, N_2, N_5, N_6, N_{14}, N_{15}, N_{12}, N_{13}, N_3, N_4, N_9, N_{10}, N_{11}, N_7, N_8. After finding N_{14}, i.e., generating six nodes, the optimal solution {5,4,1} is found. As previously, after generating 16 nodes, the optimality of the solution {1,3,4} is determined. We can say, *it is proven,* since the method is exhaustive, and we generated all nodes.

The strategy Depth-First-With-One-Successor, generates the nodes in the order: N_0, N_1, N_3, N_7, N_8, N_4, N_9, N_{10}, N_{11}, N_2, N_5, N_{12}, N_{13}, N_6, N_{14}, N_{15}. The optimal solution {1,3,2} is found after creating four nodes. After generating 16 nodes, the optimality of the solution {5,4,2} has been proven. Because the selection of the descriptor depends on the row order among the rows covering the first column, the selection is arbitrary. Hence, the order of generation in the worse case could be N_0, N_2, N_5, N_{13}, N_{16} (the temporary solution {5,4,3,1} of cost 4 has been found), N_{17}, N_{12}, N_6, N_{14}, N_{15}, N_1, N_3, N_7, N_8, N_4, N_9, N_{10}, N_{11}. A tree of 18 nodes would be generated to prove the optimality of {1,4,5}. This illustrates, how important are good heuristics to limit the size of the solution tree.

Tree Search Method #5

Subsequent advantages will result from the introduction of the heuristic functions that direct the order in which the tree is extended, with regard to the method #2 presented above. The introduction of such functions will not only lead to finding of the optimal solution more quickly, but also to expediting the proof of its optimality. This is due to a more complete application of the cutting-off property, which results in a search that is less extensive when the optimal solution is found sooner.

The quality function for the operators with regard to the selection of the best descriptors in the branching nodes as well as the quality function for nodes with regard to the selection of the nodes to be extended is defined below.

Quality function for nodes:

$$QF(NN) = CF(NN) \ + \ \hat{h}(NN), \tag{50}$$

where

$$\hat{h}(NN) \ = \ CARD(AS(NN)) \ \cdot \ CARD(GS(NN)) \ \cdot \ K \tag{51}$$

and

$$K \ = \ \frac{\sum_{r_i \ \in \ GS(NN)} f_1(r_i) \ \cdot \ CARD \ \{c_j \ \in \ AS(NN) \ \mid \ COV(r_i, c_j)\}}{\left(\sum_{r_i \ \in \ GS(NN)} CARD \ \{c_j \ \in \ AS(NN) \ \mid \ COV(r_i, c_j)\}\right)^2} \tag{52}$$

Such a defined function $\hat{h}$ is relatively easy to calculate. As proven in the experiments, it yields an accurate evaluation of the real distance of h of NN from the best solution. It is calculated as an additional coordinate of the node's vector. The function's form results from the attempt to take the following factors into account:

1. Such nodes N_i are extended that in the $AS(N_i)$ the fewest columns need to be covered. There is a higher probability that the solution is in the subtree $D(N_i)$ at the shallow depths for such nodes. Hence, the component $CARD(AS(NN))$.

2. The nodes in which the fewest decisions need to be made are extended. This is a general directive of tree searching that is especially useful when there exist, as in our problem, strong relations on descriptors. Hence, the component $CARD(GS(NN))$.

3. The coefficient K was selected in such a way that, with respect to the properties of the strategies discussed previously, the function $\hat{h}$ is as near to h as possible.

The quality function for operators is defined by the formula:

$$q^{NN}(r_i) \ = \ c_1 f_1(r_i) \ + \ c_2 f_2(r_i) \ + \ c_3 f_3(r_i), \tag{53}$$

where c_1, c_2, c_3 are arbitrarily selected weight coefficients of *partial heuristic functions* f_1, f_2, f_3, defined as follows:

$$f_1 \ has\ previously\ been\ defined\ as\ the\ cost\ function\ of\ rows \tag{54}$$

$$f_2(r_i) \ = \ CARD\{c_j \ \in \ AS(NN) \ | \ COV(r_i, \ c_j)\} \tag{55}$$

$$f_3(r_i) = \frac{1}{f_2(r_i)} \sum_{j=1}^{n} CARD \ [r_e \mid c_j \in AS(NN) \ \wedge \ COV(r_i, c_j) \ \wedge \ r_e \in GS(NN) \ \wedge \ COV(r_e, c_j)] \tag{56}$$

where n is a number of columns. Function f_3 (r_i) defines the "resultant usefulness factor of the row" r_i in node NN. Let us assume that there exist k rows covering some column in the set $GS(NN)$. The usefulness factor of each of these rows with respect to this column is k. When $k = 1$, the descriptor is *indispensable* (or with respect to Boolean minimization, the corresponding prime implicant is *essential*). The *resultant usefulness factor of the row* is the arithmetical average of the *column usefulness factors* with respect to all the columns covered by it. Then, one should add an instruction in the operator subroutine to sort the descriptors in $GS(NN)$ according to the non-increasing values of the quality function for descriptors q^{NN}.

The next way of decreasing the solution tree could be by declaring new section code that checks the relations on descriptors.

If the descriptors r_i and r_j are in the node N in the *domination relation* (such relation is denoted by $r_i \ > \ r_j$), r_j can be removed from $GS(N)$ with the guarantee that at least one optimal solution will be generated.

If the descriptors r_i and r_j are in node N in the *global equivalence relation,* one or the other of them can be selected and the other one should then be removed from $GS(N)$, as well as from $GS(M)$ where M is any node in the sub-tree $TREE(N)$. The *equivalence class* $[r]$ of some element r from $GS(N)$ is replaced in this coordinate by r itself. Descriptors declared as locally equivalent are treated similarly. The only difference is that the descriptors are then removed from $GS(N)$ only. Let us observe that these relations are not based on costs, but on some additional problem-dependent information about the nodes of the tree that is available to the program. The familiar to us covering problem may be a good example of this property.

The descriptors r_1 and r_2 are *globally equivalent* in node NN when they have the same cost and cover the same columns:

$$r_1 \ \equiv \ r_2 \iff f_1(r_1) = f_1(r_2) \ \wedge \ (\forall \in AS(NN)) \ [COV(r_1, c) \ = \ COV(r_2, c)]. \tag{57}$$

Descriptors (rows) r_1 and r_2 are *locally equivalent* in node NN when, after removing one of them from the array, the number of columns covered by j rows is the same for each $j \ = \ 1, ..., CARD(GS(NN)) - 1$ as after removing the second one.

$$r_1 \ \doteq \ r_2 \iff (\forall j = 1, ..., CARD(GS(NN)) - 1) \ [LK(j, r_1) \ = \ LK(j, r_2)] \tag{58}$$

where $LK(j, r)$ is the number of columns covered by j rows in the array that originates from $M(NN)$ after removing row r.

$$LK(j, r) \ = \ CARD \ \{c_k \in AS(NN) \ | \ CARD(X_k) = j\} \tag{59}$$

where X_k is the set of rows covering the column c_k;

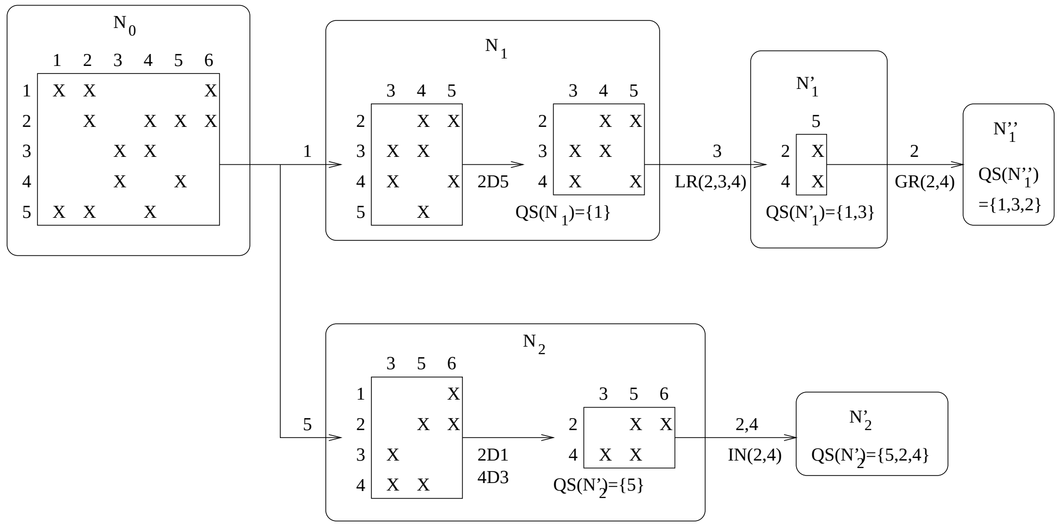


Figure 8: Final Search Method for the Table from Fig. 5

$$X_k \;=\; \{x \;\in\; GS(NN) \;\setminus \{r\} \;\mid\; COV(x, c_k)\} \tag{60}$$

Descriptor r_1 is *dominated* by descriptor r_2 when it has larger cost and covers at most the same columns as r_2, or if it has the same cost and covers the subset of columns covered by r_2,

$$\begin{aligned} r_1 \leq r_2 \Longleftrightarrow f_1(r_1) > f_1(r_2) \wedge (\forall\, c_k \in AS(NN))[COV(r_1,\ c_k) \;=\; COV(r_2,\ c_k)] \\ or\, f_1(r_1) = f_1(r_2) \wedge \{c_k \in AS(NN) \mid COV(r_1,\ c_k)\} \subset \{c_k \in AS(NN) \mid COV(r_2,\ c_k)\} \end{aligned} \tag{61}$$

The developer can program all of the relations given above or only some of them. If all the relations have been programmed and parametrized, the user can still select any of their subsets for execution using parameters. The solution process is shown in Fig. 8.

Column 1 and rows 1 and 5 are selected at the beginning ($GS(N_0) \;=\; \{1,5\}$). After the selection of row 1 to $QS(N_1)$, row 5 becomes dominated by 2 (or 3) and is removed. The domination of descriptor 5 by descriptor 2 is denoted in the Figure 8 by 2D5. Now descriptors 2,3,4 are locally equivalent, denoted as $LR(2,3,4)$. One of them, say 3, is selected. Descriptors 2 and 4 then become globally equivalent in node N_1' , denoted as $GR(2,4)$. One of them, say 2, is selected. This leads to the solution $QS(N_1'') \;=$ {1,3,2}. Now the backtracking to the initial node, N_0, occurs and descriptor 5 is selected. Next, descriptors 1 and 3 are removed since they are dominated and then descriptors 2 and 4 are selected as indispensable descriptors in node N_2' that is denoted as $IN(2,4)$ in Fig. 8. This produces the solution $QS(N_2') = $ {5,2,4}. After backtracking to the initial node $GS(N_0) = \emptyset$, node N_0 is removed from the open-list. The open-list $= \emptyset$ completing the search of the tree. The last solution of the minimal cost 3 is then proven to be the optimal solution.

Note the following facts:

- not all of the minimal solutions were obtained but more than one was produced,
- only node N_0 is permanently stored in the tree,
- if the user declared parameter $F_{min\ min} \;=\; 3$, the program would terminate after finding the solution {1,3,2}. In some problems, guessing or evaluating the cost of the function is not difficult.

	1	2	3	4	5	6	7
A (4)			X	X			
B (3)			X		X		
C (3)							X
D (3)	X	X	X		X		
E (2)				X	X	X	
F (4)	X	X				X	

Figure 9: A Covering Table with Costs of Rows not Equal

If all of the above relations, except the most expensively tested local descriptor equivalence, were declared, the complete tree consisting of 8 rows and 3 solutions would be obtained.

Yet another approach would be to select Branch-and-Bound and Ordering as global strategies and $MUST0$, $EQUIV$, $REAPNT$ to define the local strategy. $EQUIV$ only checks for the global equivalence of descriptors. In this example, the covering table shown in Fig. 9 will be solved. The cost of each row is entered next to its respective descriptor. In this example, the costs of the rows are not equal. The tree structured state-space for this problem is shown in Fig. 10. The details concerning the node descriptions for this tree are also illustrated in Table from Figure 11. (By *pred(N)* we denote the predecessor node of node N).

The search starts from node 0 where no column is covered so that the set AS consists of all the columns and all the rows available as descriptors, as well as QS which is an empty set since no descriptor has been applied. After being processed by $EQUIV$, it is found that descriptor B is dominated by the other descriptor, D. Descriptor B is, therefore, deleted from the descriptor list. $MUST0$ finds that descriptor C is indispensable (with respect to column 7), and it is then immediately applied by GEN to create the new node 1. The descriptor list is then ordered by $ORDER$ using the quality function mentioned above. Assuming the coefficients $c_1 = 0.5$, $c_2 = 0$, and $c_3 = 1$, the costs of descriptors are

$$Q(A) \;=\; 2 + 4/2 = 4.0, \qquad Q(D) \;=\; 1.5 + 4/4 = 2.5 \tag{62}$$

$$Q(E) \;=\; 1 + 3/3 = 2.0, \qquad Q(F) \;=\; 2 + 4/3 = 3.3 \tag{63}$$

The descriptor list is arranged according to the descriptor costs as $\{E,\ D,\ F,\ A\}$. The descriptors are applied according to this sequence.

There is no difference between the application of the descriptors in the sequence of E, D or D, E for the solution in this problem. Therefore, if descriptors E and D have already been applied, it is not necessary to apply them again in another sequence. This is why descriptor E is cancelled for node 3; E and D for node D_0 ; as well as E, D and F for node D_1. This cancellation is done by $REAPNT$. The above procedure prevents node D_0 from finding the descriptor to cover column 5. Therefore, this node is not in the path to the solution and should be cut off by $GENER$. This phenomena also happen for nodes D_1, D_3, and D_5. The cost of node D_2, which is 13, exceeds the temporary cost B which is the cost of solution node 4 that has already been found. It was, therefore, cut off by the $Branch-and-Bound$ strategy. So was node D_4. The whole search procedure in this example deals with 11 nodes but only stores the descriptions of 5 nodes in the memory structure. A total of two optimal solutions has been found in the search.

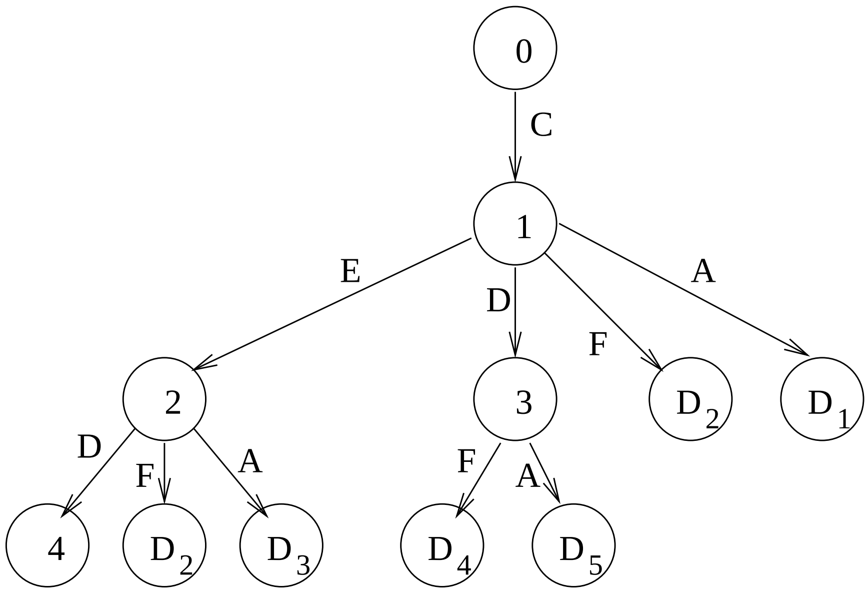


Figure 10: A Search Method for the Table from Figure 9

The methodology presented above illustrates the characteristic trade-off relationship between the knowledge-based reasoning and the exclusively intrinsic search already mentioned in section 3. The direct description of the problem allows us to find a solution based strictly on the generation of all possible cases not worse than the solution generated previously. The successive addition of the information in the form of new heuristic directives and methodic directives that are based on the analysis of the problem and the solution process (e.g. quality functions, domination relations, equivalences, $F_{min\ min}$, etc.) allows for the search to be decreased.

Until now, we have not focused on how the relation COV is *represented.* This could be an array, a list of pairs (r_i, c_j), a list of lists of columns covered by rows, a list of lists of rows covering the columns, etc. The selection of the representation is independent from the selection of the method and from the strategy, but various combinations of these can have different effects. At some stage in creating the program, the user decides on the selection of, for example, the binary array and writes the corresponding functions. The user can then work on the representation of the array next; by the words, by the bits. The arrays $M(N)$ also do not necessarily need to be stored in nodes as separate data structures, they can be re-created from $AS(N)$ and $GS(N)$. The local strategy parameters should also be matched to the representation. This is related to such factors as the total memory available for the program as well as the average times needed to select the node, to generate the individual node, to extend the node, to select the descriptor, and to check the solution condition.

In similar problems in the past, we have found that the application of each of the equivalence conditions, dominance conditions, or indispensable conditions in the covering problem reduces the search space by about 2 to 3 times. The joint application of all the conditions brought about a reduction of approximately 50 to 200 times the generated space.

QUESTIONS FOR SELF-EVALUATION

1. Create methods and strategies for Petrick Function Minimization Problem, that will make use of the similarity of this problem to the Set Covering Problem explained above.

2. Repeat point 1 for the Problem of Finding all Sets of Vacuous Variables for a binary single-output strongly unspecified function.

	0	1	2	3	4
N	0	1	2	3	4
SD	0	0	1	1	2
pred(NN)	-	0	1	1	2
OP	-	C	E	D	D
F	0	3	5	6	8
NAS	7	6	3	2	0
NQS	0	1	2	2	3
NGS	6	4	3	2	0
AS	1~7	1~6	1,2,3	4,6	-
QS	-	C	C,E	C,D	C,E,D
GS	A~F	E,D,F,A	D,F,A	F,A	-

Figure 11: Node Descriptions for the Tree from Fig. 9

3. Repeat point 1 for the Problem of Finding all Sets of Vacuous Variables for a multiple-valued single-output strongly unspecified function.

4. Create methods and strategies for the POS Satisfiability Minimization Problem, that will make use of the similarity of this problem to the Set Covering Problem explained above. Remember, that contrary to the Petrick Function Problem, the POS Satisfiability can have no solution. If solutions exists, you want to find one that has the minimal number of positive (non-negated) variables.

5. Use the Set Covering Problem from this section as an example to create various methods and strategies for the Generalized Clique Partitioning Problem. Assume that each node of the graph has a cost, and we look for a clique that maximizes the cost. What are the heuristics here? How can you use the heuristics?

5 EXAMPLE OF APPLICATION: THE GRAPH COLORING PROBLEM

The tree search algorithm for the Graph Coloring Problem given below colors successively nodes with actually available color of a smallest number, remembering for each node all 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 the graph is equal to the number of colors in the exact solution. We apply the strategy: "depth-first with one child". After finding a solution that is better than a previous one (in a new branch) the algorithm has a new, improved evaluation of the chromatic number, so it can remove from sets $GS(N)$ all the operators that correspond to colors which were not included in the solution. This is a very simple example of a Branch Switch Strategy. The process of tree creation is continued using the cut-off principle based on cost function.

Algorithm for Proper Graph Coloring

This algorithm uses the Universal Strategy.
$NODES$ is the set of nodes.
$EDGES \subset NODES \times NODES$ is the set of edges.
$G = < NODES, EDGES >$ is the graph.

1. **Creating the Initial State**:

1. $NODE_1 := NODES[0]$, /* take first node from $NODES$ */
   create initial QS := $\{NODE_1 \ . \ 1)\}$.
   $NODE_2 := NODES[1]$, /* take second node from $NODES$ */
   if ($(NODE_1, NODE_2) \in EDGES$) or ($(NODE_2, NODE_1) \in EDGES$)
   then append ($NODE_2$, 2) to QS; $COLORS\_USED := \{1,2\}$
   else append ($NODE_2$, 1) to QS; $COLORS\_USED := \{1\}$.
   $CF :=$ CARD($COLORS\_USED$).
2. $SET\_OF\_COLORS := \{1,2,...,\text{CARD(NODES)}\}$.
3. $(QS, GS, REMAINING\_NODES, COLORS\_USED, CF) :=$
   ({ QS, $REMAINING\_NODES$, $COLORS\_USED$, CF)

2. **Operator** O(N, $COLOR$) = [

   a) $SN :=$ first element from $REMAINING\_NODES(N)$, /* SN = selected node */
   b) $REMAINING\_NODES(NN) := REMAININB\_NODES(N) \setminus SN$,
   c) $QS(NN) := QS(N) \cup \{ (SN, COLOR) \}$
   d) if $COLOR \in COLORS\_USED(N)$ then $CF(NN) := CF(N)$
   /* color exists, no change in cost */
   else
   $(CF(NN) := CF(N) + 1, COLORS\_USED(NN) := COLORS\_USED(N) \cup \{COLOR\})$,
   e) if $CF(NN) \geq CF_{min} \wedge REMAINING_NODES(NN) \neq \emptyset$
   then (cut-off, backtrack),
   f) $GS(NN) := SET\_OF\_COLORS \setminus \{FK(SN) \mid ((SN_i, SN) \in EDGES$ or $(SN, SN_i) \in EDGES) \}$

   ]

3 **Selected Strategy:** Depth-First With One Child.

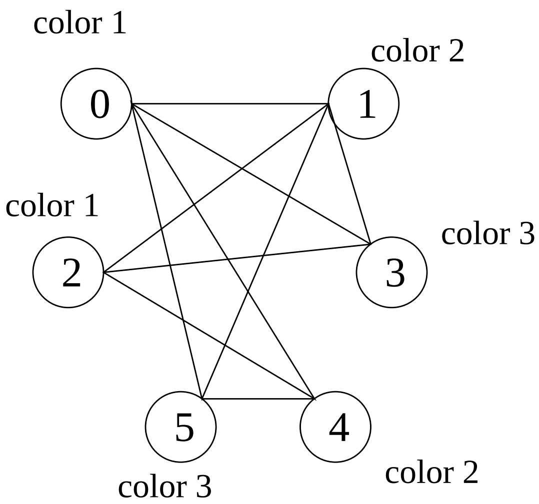


Figure 12: Graph for Coloring to Example 5.1

4 **Switch Condition for Branch:**

If $REMAINING\_NODES(NN) = \emptyset$ then

for each N_i in branch from N_0 to NN do

$GS(N_i) := GS(N_i) \cap COLORS\_USED(NN)$

COMMENTS

1. Coordinate $QS(N)$ includes a partial coloring of the graph, it means the set of pairs $(j,\ FK(j))$ where $j \in NODES$. In the initial state two inconsistent nodes are colored with different colors 1 and 2, and two nodes n_1 and n_2 such that $(n_i,\ n_2) \notin EDGES$ with the same color 1.

2. $GS(N)$ is the set of colors, which could be used to color the given node SN. $REMAINING\_NODES(N)$ are the set of non-colored nodes, and $COLORS\_USED(N)$ are the set of used colors.

3. In the operator, to increase the efficiency, the step e) of cutting-off before calculating $GS(NN)$ has been applied. As the candidates to color the selected node SN one can select the colors which are different from colors used for nodes that are linked with SN by edges (step f) above).

4. In the moment of finding solution $QS(NN)$ in node NN it is known that the minimial solution needs at most as many colors as in $COLORS\_USED(NN)$. Therefore one can use only a subset of $COLORS\_USED(NN)$. In the Switch Condition for Branch one can thus remove from all coordinates $GS(N_i)$ (for all nodes N_i in the branch from N_0 to the final node NN) the colors that do not belong to $COLORS\_USED(NN)$, since we are not interested in all possible colorings, but only in the colorings with the accuracy to an isomorphism (i.e the minimum chromatic number colorings).

5. Let us note that we have to deal here with two kinds of nodes: nodes of the graph, and nodes of the search tree.

Example 5.1

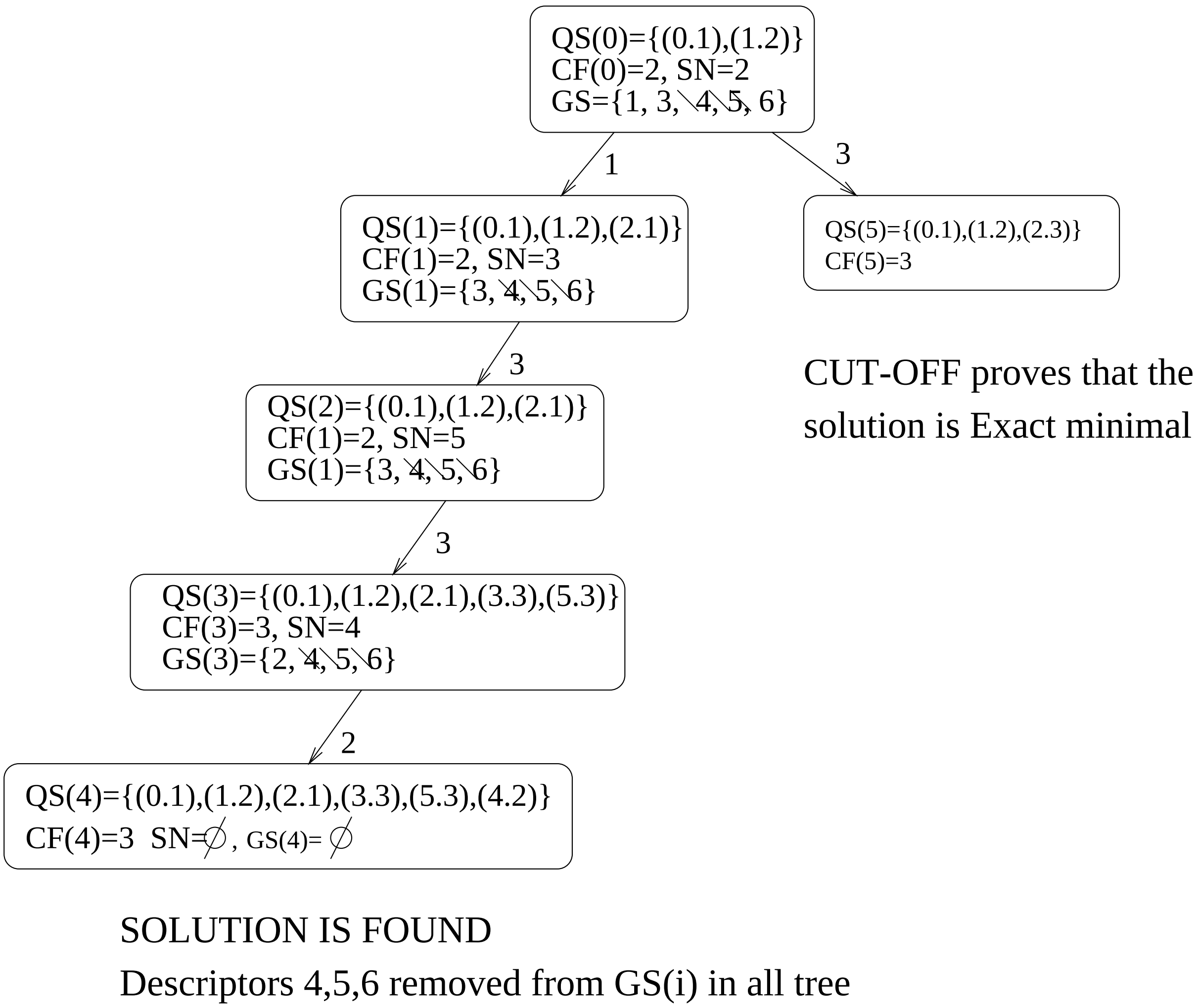


Figure 13: Tree Search for the Exact Graph Coloring Algorithm to Example 5.1

The tree for the graph from Figure 12 is shown in Figure 13. (For clarity, only some coordinates of the state vector are shown).

The search process is executed as follows.

1. The initial pair of graph nodes is 0 and 1, which obtain colors 1 and 2, respectively.

2. Next graph node 2 can obtain one of colors {1,3,4,5,6}. Selected is 1.

3. Now graph node 3 can obtain one of colors {3,4,5,6}. 3 is selected.

4. This way, the first branch of the tree from Figure 13 is created.

   After finding the first solution:

   color 1 = {0, 2},

   color 2 = {1, 4},

   color 3 = {3, 5},

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 $GS(3)$, $GS(2)$, $GS(1)$ and $GS(0)$. Now backtrack to node 3 of the search tree occurs.

6. $GS(3) = \emptyset$, so next backtrack is to node 2 of the search tree. Further, $GS(2) = \emptyset$, so next backtrack is to node 1 of the tree is done.

7. Now, $GS(1) = \emptyset$, so next backtrack is to node 0 of the search tree is done.

8. Now, $GS(0) = \{3\}$ so as a result of execution of operator O(0,3) node 5 of the tree is created. In this node, $CF(5) = 3$ and $NODES(5) = \{3, 5, 4\} \neq \emptyset$, so cut-off is executed and backtrack to node 0 of the tree.

9. Now $GS(0) = \emptyset$ and the program terminates its operation with the coloring found in node 4 of the tree. However, now we have also a proof that this is the exact minimal coloring.

QUESTIONS FOR SELF-EVALUATION

1. How to modify the above algorithm to make smarter choices of successive nodes for coloring? Can you use the node density or other local graph node-related parameters? Can you use the cycle measures or other local graph cycle-related parameters?

2. How to modify the above algorithm to make an additional, secondary, requirement that the numbers of nodes colored with the same color will be approximately equal?

3. How to modify the above algorithm in such a way that nodes not connected to other nodes remain not colored at all? What would be the use of this property in Column Minimization Problem.

4. How to modify the above algorithm to make it a Multi-Coloring Graph Coloring? By Multi-Coloring we understand an algorithm in which a node can be colored with as many colors as possible, but there is never a conflict between any two nodes, it means, sets of colors assigned to any two adjacent nodes are disjoint. What would be the use of this property in Column Minimization Problem? Write a pseudo-code of a program similar to one above.

5. How to modify this algorithm that it will start from coloring large cliques and using them to calculate the lower bound on the chromatic number?

6. How to use the algorithm to generate a set of relatively large, but not necessarily maximum, cliques, and not all of them? This problem would have applications to the Encoding Problem in Functional Decomposition.

6 CLASSICAL APPROACHES TO COLUMN MINIMIZATION

In previous sections we discussed how to create efficient search procedures, and we gave examples how Set Covering and Graph Coloring problems can be solved in our framework. How can we use these methods in decomposition?

As we know, the Column Minimization Problem is very important to the success of a Functional Decomposer, and most of the time of the calculations of a Decomposer are spent on the Column Minimization. This problem can be solved by either Graph Coloring or Set Covering. Moreover, our methodology teaches us that it is not only important to create an efficient solver, but it is also important to reduce a problem to standard search problem in such a way that this standard problem will be of the smallest possible dimension. When this cannot be done, we look for a non-standard problem of a smaller dimension.

An attempt to create various algorithms using our methodology will be done in this and forthcoming sections for the Column Minimization Problem. Next, the Variable Partitioning Problem will be presented in the same framework. This should help the reader to be able to formulate other problems, like this by himself. For instance, the Encoding Problem can be formulated in many ways using the general methodology outlined by us above.

Obviously, there are many approaches to solve the functional decomposition problem. Some approaches are better than others with regards to particular problem requirements and worse with regards to others. For example, one method may be very fast but may have a very large memory requirement. So this method would be not practical for functions of many variables and terms. Another method may be able to solve functions of many variables but will be very slow. Still other methods may be better suited for highly unspecified functions.

Here, we want to understand, what is the role of the Column Minimization Problem 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.

As the primary requirements for a decomposition program vary from one application area to another, it is important to understand well the characteristics of every application when determining the best method. Since our work is primarily concerned with Machine Learning applications where the functions have an extremely high percentage of don't care values, we take such functions into account here. Presently the method introduced by Pedram et al with the best overall results in decomposing Boolean functions ([48]) uses a BDD based approach using a Set Covering formulation of the problem. This approach was designed with circuit-based applications in mind.

However, for Machine Learning applications it is believed that reformulating the problem from a Set Covering formulation to a Graph Coloring formulation of the problem may significantly improve the efficiency of the decomposition. It should be noted that the Graph Coloring approach, first introduced for decomposition by Perkowski [58], is not a totally new idea in logic synthesis, but is often times overlooked as an alternative approach to Set Covering and is actually a more suitable choice in many problems.

What makes the approach of Pedram et al better than other approaches is perhaps a better combination of the following: choice of data structures, selection of free and bound sets, finding compatible columns, encoding techniques, and efficient programming. Perhaps, the choice of the data structure for Boolean functions was the main single contributor to the success of their approach. Because the problem of decomposing a functions has many parts, it seems very likely that only certain components of the Pedram's et al approach are more efficient than their counterparts in other programs. In fact, we believe that there are other methods which would be more efficient overall if a better combination of these techniques and data structures were used.

6.1 PARTITION CALCULUS FOR FUNCTIONAL DECOMPOSITION

6.1.1 Main Theorem

$H(A, G(B, C))$ represents the decomposed function such that $F = H(A, G(B, C))$. The sets A and B are the *free set* and *bound set*, respectively where $A \cup B = X$, and $A \cap B = \emptyset$. The set C is some subset of A when a *nondisjoint decomposition* is necessary, and is an empty set otherwise. It is called a **shared set of variables**. $P(A)$ is the input partition on the free set of input variables and $P(B)$ is the input partition on the bound set of input variables. P_F is the output partition on the set of output variables.

Definition 6.1 *Let B be a subset of the set of inputs X, an input partition generated by set B is denoted as:*

$$P(B) = \prod_{x \in B} P(x) \tag{64}$$

where $\prod$ denotes the product of partitions. $P(x)$ is a partition for variable x.

Definition 6.2 *Let B be a subset of outputs set Y, the output-consistency relation associated with B, i.e. $P_F(B)$, is called a $rough - partition$ (r-partition).*

Based on the definitions of partition, Luba developed the decomposition theorems.

Example 6.1 .

For a truth table from Table 1 (this is a version of example from our Report [63], but for simplicity, all cubes are disjoint here):

	X_0	X_1	X_2	Y_0	Y_1
0	1	0	0	0	1
1	1	0	1	0	1
4	-	1	1	0	-
5	-	1	0	0	0
6	0	0	-	1	0

Table 1: Truth table

The partitions of input variables are the following:

$$P(X_0) = (0, 1, 4, 5;\ 4, 5, 6); \tag{65}$$

$$P(X_1X_2) = (0, 6;\ 1, 6;\ 5;\ 4); \tag{66}$$

This is called the *discernibility relation* of input variables.
The partition of output signal is:

$$P_F \ = \ P(Y_0Y_1) \ = \ (4, 5;\ 0, 1, 4;\ 6) \tag{67}$$

This is called the consistency relation of output signals. As we can see from P_F, its sets are non-disjoint, this relation is called a rough-partition (r-partition).

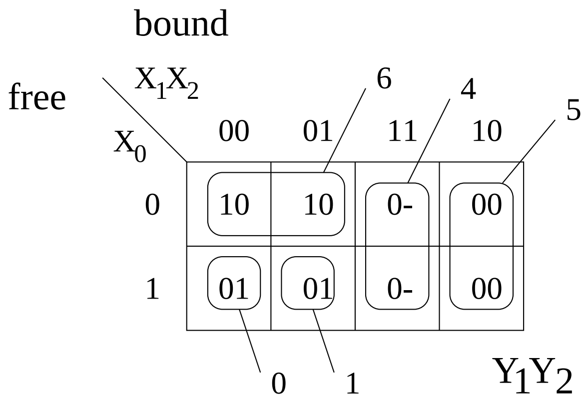


Figure 14: Karnaugh Map for Function from Example function F_1 with numbers of cubes from Table 1

The corresponding Karnaugh map is shown in Figure 14. The numbers of rows (cubes) are shown for each cube of the Kmap.

Sometimes this notation can lead to troubles. For example. Suppose we had a cube -01 with output 01 and cube 10- with output 10, because these two cubes intersect, there would be a conflict. We should watch for this in the situation of having DC sets both in input and output. This must be taken care of by the Reader Program that will read some files taken from other sources. We suppose that overlapping cubes with no conflict can also make some approaches more difficult, so for time being we assume that the cubes are all disjoint.

Theorem 6.1 *Functions G and H represent a serial decomposition of function F, i.e. $F = H(A, G(B, C))$ if there exists a partition $\Pi_G \geq P(B \cup C)$ such that $P(A) \cdot \Pi_G \leq P_F$, where all the partitions are over the set of cubes and the number of two-valued output variables of component Π_G is equal to $g = [\log_2 L(\Pi_G)]$, here $L(\Pi)$ denotes the number of blocks of partition Π, and $[x]$ denotes the smallest integer equal to or larger than x.*

Here $[\log_2 L(\Pi_G)]$ gives us the number of output signals from function G.

For example, in Table 1, we have:

$$P(B) \;=\; P(X_1X_2) = (0, 6;\; 1, 6;\; 5;\; 4); \tag{68}$$

$$P(A) \;=\; P(X_0) = (0, 1, 4, 5;\; 4, 5, 6); \tag{69}$$

if

$$\Pi_G = (0, 1, 6;\; 4, 5); \tag{70}$$

then

$$P(B) < \Pi_G \tag{71}$$

$$P(A) \cdot \Pi_G = (0, 1;\; 6;\; 4, 5) < P_F = (4, 5;\; 0, 1, 4;\; 6) \tag{72}$$

Therefore this is a feasible decomposition with bound set $B = \{X_1, X_2\}$ and free set $A = \{X_0\}$. The number of output signals from G is $[\log_2 2] = 1$.

The most important question, however, remains:

- How to find the bound and shared set B, C?

One method for determining the bound/shared set is to use the r-admissibility test.

6.1.2 The r-Admissibility Test

Definition 6.3 r-Admissibility Test. *Let P_i be a partition induced by some input variables x_i. The set of partitions $\{p_1, \ldots, p_k\}$ is called $r-admissible$ with respect to partition P_F if there exists a set $\{p_{k+1}, \ldots, p_r\}$ of two-block partitions, such that:*

$$p_1 \ldots p_k \cdot p_{k+1} \ldots p_r \leq P_F \tag{73}$$

and there exists no set of $r - k - 1$ two-block partitions which meet this requirement.

Given a set of input variables, this test helps us to know if function F is decomposable. If so, how many input variables the component H has.

Definition 6.4 Quotient partition. *Let γ be a partition and σ be an r-partition, in a quotient partition which is denoted as $\gamma|\sigma$, each block of γ is divided into a minimum number of elements being blocks of γ .*

For example:

$$\sigma = (1;\ 2,6;\ 3,6;\ 5,7;\ 4,5);$$

$$\gamma = (1,2,3,6;\ 4,5,7);$$

$$\gamma|\sigma = ((1)(2,6)(3,6);\ (4,5)(5,7));$$

Theorem 6.2 *For partitions γ and σ, such that $\sigma \leq \gamma$, let $\gamma|\sigma$ denote the quotient partition and $\eta(\gamma|\sigma)$ the number of elements in the largest block of $\gamma|\sigma$. Let $[log_2\eta(\gamma|\sigma)]$ denotes the smallest integer equal to or larger than $\log_2 \eta(\gamma|\sigma)$. (This is the lower bound of the multiplicity index). Let Π be the product of partitions $P_1, \ldots, P_k$, and $\Pi_F = \Pi \cdot P_F$. Then $\{P_1, \ldots, P_k\}$ is r-admissible in relation to P_F, with $r = k + [log_2(\Pi|\Pi_F)]$.*

Example 6.2

For function from Example 6.1 (Table 1), we select X_0 as free set, we can test if this is feasible by testing the r-admissibility.

$$P(X_0) \cdot P_F = (4,5;\ 0,1,4;\ 6); \tag{74}$$

$$P(X_0) \mid P(X_0) \cdot P_F = ((0,1,4)(5);\ (4,5)(6)); \tag{75}$$

$r = 1 + [\log_2 2] = 2$, i.e. $P(X_0)$ is 2-admissible, which means that the number of inputs to H is two, and we know X_0 is a free set, thus the number of outputs from G equals one, which is the same as what we obtained before.

The *r-admissibility* has the following interpretation. If a set of partitions $(P_1, \ldots, P_k)$ is r-admissible, then there exists a serial decomposition of $F = H(A, G(B, C))$ in which component H has r inputs: k

primary inputs corresponding to input variables which induce $P_1, \ldots, P_k$ and $r-k$ inputs being outputs of subfunction G.

A simple way to determine the admissibility is to solve the equation in Theorem 6.2 (i.e. $r = k + [log_2 \eta(\gamma \mid \sigma)]$). Where k is the number of variables in the input partition $P(A)$ and $\eta(\gamma \mid \sigma)$ as defined in Theorem 6.2 is equal to the number of sub-blocks in the largest block of $P(A) \mid P(A) \cdot P_F$.

6.2 PARTITION CALCULUS FORMALISM FOR DECOMPOSITION

6.2.1 Maximal Compatible Classes

Using the r-admissibility test, we can find a suitable set of input variables for component H. To find the corresponding set of inputs for component G, we have to find $P(B \cup C)$, such that there exists $\Pi_G \geq P(B \cup C)$ that satisfies Theorem 6.1. To solve this problem, consider a subset of primary inputs, $B \cup C$, and the q-block partition $P(B \cup C) = (B_1, B_2, \ldots, B_q)$ generated by this subset. Then we use a relation of compatibility of partition blocks to verify whether $P(B \cup C)$ is suitable for the decomposition or not.

Definition 6.5 Compatibility relation: *Two blocks B_i and $B_j \in P(B \cup C)$ are* **compatible** *iff partition $P_{ij}(B \cup C)$ obtained from partition $P(B \cup C)$ by merging blocks B_i and B_j into a single block satisfies $P(A) \cdot P_{ij}(B \cup C) \leq P_F$.*

For example, in Table 1, we have :

$$P(B) = (0,6;\ 1,6;\ 5;\ 4) = (B_1, B_2, B_3, B_4); \tag{76}$$

If we merge B_1 and B_2 together, we get:

$$P_{12}(B) = (0,1,6;\ 5;\ 4); \tag{77}$$

$$P(A) \cdot P_{12}(B) = (0,1;\ 6;\ 5;\ 4) < P_F, \tag{78}$$

therefore B_1 and B_2 are compatible, denoted as $B_1 \sim B_2$.

The same way we can check the compatibility relation of other blocks in $P(B)$.
The result is: $B_1 \sim B_2$, $B_3 \sim B_4$.

A compatible class is called Maximal Compatible Class (MCC) if and only if it can not be properly covered by any other compatible class.

Thus we have $MCC1 = \{B_1, B_2\}$, $MCC2 = \{B_3, B_4\}$. From this we can find the minimal cover, i.e.$\{\{B_1, B_2\}, \{B_3, B_4\}\}$, and $\Pi_G = (0,1,6;\ 4,5)$. We can easily see that Π_G corresponds to a 1-output function G. Therefore we have $F = (X_0, G(X_1, X_2))$.

6.2.2 Compatibility

Definition 6.6 Cubes: *Two cubes are said to be* **"compatible"** *if they belong to the same block of the partition on the free set being considered* **and** *the same two cubes belong to the same block in the output partition.*

Two cubes are said to be **"incompatible"** *if they belong to the same block of the partition on the free set being considered* **and** *the same two cubes* **do not** *belong to the same block in the output partition.*

Definition 6.7 *Two blocks B_i and B_j of a partition $\Pi(B)$ are said to be* **compatible blocks** *if they satisfy the requirement $P(A) \cdot (B_i \cup B_J) \leq P_F$.*

The submap corresponding to some combination of input variables will be also called a cofactor. If this is a combination of free variables, it will be also called a row - referring to a Kmap. If this is a combination of bound variables, it will be also called a column.

Definition 6.8 *Two columns of a Karnaugh Map are said to be* **compatible columns** *if every pair of outputs in corresponding cells (i.e. outputs corresponding to the same combination of input variables in the free set) are compatible. In binary single-output function, two outputs are compatible as long as they are not complements of each other.*

Definition 6.9 *A* **Compatible Class (CC)** *is a set of partition blocks that any two of them are compatible.*

Definition 6.10 *A* **Maximum CC (MCC)** *is a CC that cannot be covered by any other CC.*

Definition 6.11 *Columns that are not compatible will be called* **incompatible**.

Definition 6.12 **Decomposition Function Cardinality (DFC)** *is sometimes used as a measure to evaluate the quality or effectiveness of a decomposition (where a lower DFC is desired). More specifically, it is an integer value which is equal to*

$$DFC = \sum_{i=1}^{n} 2^{I_i} \times O_i$$

where I_i and Oi are the number of inputs and outputs to block i and n is the total number of blocks.

6.2.3 Encoding of Compatible Classes

As the truth table of H is partially constructed of G block outputs and the G block outputs are nothing but the codes allotted during the encoding process, it is very important to find the right code to reduce the number of logic blocks needed to implement a truth table. Next, functions G and H are calculated, (option) minimized, and realized.

In Table 2 the encoding problem is very simple; symbols A and B have to be encoded with 0 and 1. Next it is very easy to find functions G and H from Table 2 with its binary code from the last column.

$$g = X_1, Y_0 = X_1 X_0, Y_1 = X_0$$

The circuit of this function is shown in Figure 15.

	X_0	X_1	X_2	Y_0	Y_1	$g_{symbolic}$	g_{binary}
0	1	0	0	0	1	A	0
1	1	0	1	0	1	A	0
4	-	1	1	0	-	B	1
5	-	1	0	0	0	B	1
6	0	0	-	1	0	A	0

Table 2: Symbolic and binary function g to Examples 6.1 and 6.2

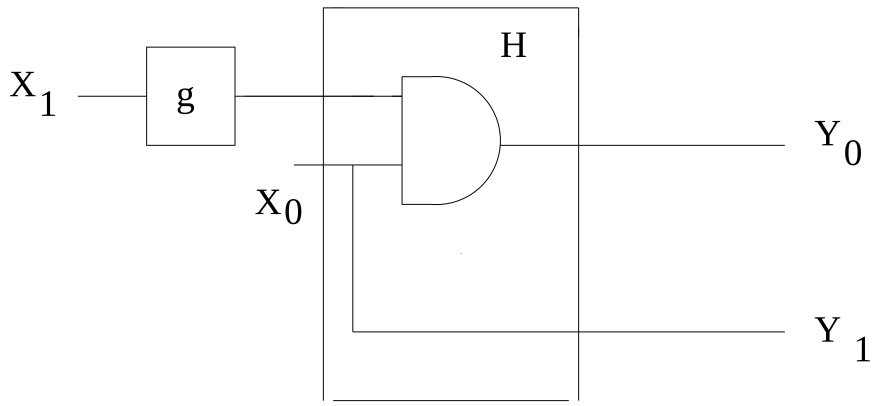


Figure 15: The Decomposed Circuit for Example 2

6.3 SET COVERING APPROACH TO COLUMN MINIMIZATION

Example 6.3

Given is the function described by the Karnaugh map in Figure 16, with the bound and free sets {c,d,e} and {a,b}, respectively. The steps for the set covering approach are as follows:

Step 1: Find all compatible pairs of columns. Compatible pairs are those columns which have the same output values in corresponding positions of every row. Any one of the following pairs of cells classifies as having the same values: (1,1),(1,X),(0,0),(0,X),(X,X). For simplicity let B_{ij} denote the compatible pair of columns B_i and B_j, denoted $(B_i,\ B_j)$.

Let C_B be the set of compatible pairs:

$$\begin{aligned} C_B \;=\; & (B_{12}, B_{13}, B_{14}, B_{15}, B_{17}, B_{18}, \\ & B_{23}, B_{25}, B_{27}, B_{28}, B_{34}, B_{35}, B_{36}, B_{37}, B_{38}, B_{45}, B_{56}, B_{57}, B_{58}, B_{78}) \end{aligned} \tag{79}$$

This results in the Compatibility Graph from Figure 17 with the nodes corresponding to the columns in the Karnaugh map and the edges between them indicating that the columns are compatible.

Step 2: Incrementally build CC's (MCC's-maximum compatible classes) from the compatible pairs of columns found in step 1 until each column is present in at least one CC. For simplicity, only the block number is used to denote each block (i.e. B_i is shown as number i).

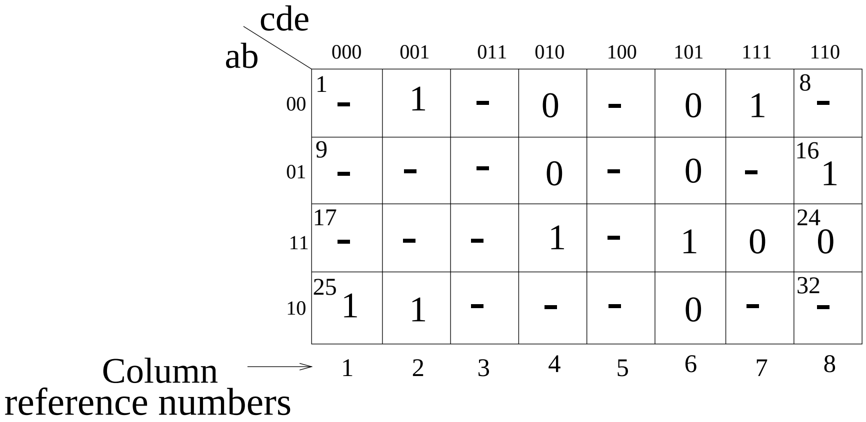


Figure 16: Karnaugh Map for function F_1 from Example 6.3 with numbers of columns shown below the map.

$CC_1 = 1$

Next, combine column 2 with the CC's of the previous incremental step.

$CC_2 = (1{,}2)$

Next, combine column 3 with the CC's of the previous incremental step.

$CC_3 = (1{,}2{,}3)$

Next, combine column 4 with the CC's of the previous incremental step.

It is found that column 4 is not compatible with column 2 and therefore a new CC must be introduced which includes column 4 and not column 2.

$CC_4 = \{\ (1{,}2{,}3),\ (1{,}3{,}4)\ \}$

Similarly, complete the CC's until all columns have been included in at least one CC.

$CC_5 = \{\ (1{,}2{,}3{,}5),\ (1{,}3{,}4{,}5)\ \}$
$CC_6 = \{\ (1{,}2{,}3{,}5),\ (1{,}3{,}4{,}5),\ (3{,}4{,}5{,}6)\ \}$
$CC_7 = \{\ (1{,}2{,}3{,}5{,}7),\ (1{,}3{,}4{,}5),\ (3{,}4{,}5{,}6)\ \}$
$CC_8 = \{\ (1{,}2{,}3{,}5{,}7{,}8),\ (1{,}3{,}4{,}5),\ (3{,}4{,}5{,}6)\ \}$

Step 3: Find the minimum number of MCC's which will completely cover all of the columns. The CC's in the last row of the previous step (CC_8) form the MCC's for this function on the given bound set.

$$MCC1 \ = \ (B_1, B_2, B_3, B_5, B_7, B_8)$$

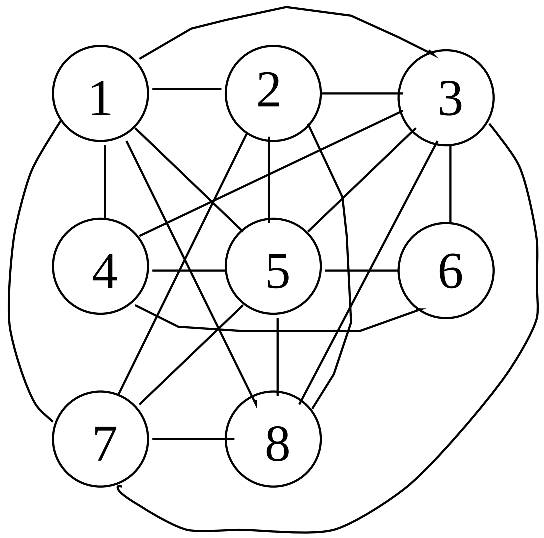


Figure 17: The Compatibility Graph for function F_1 from Example 6.3

$$MCC2 = (B_1, B_3, B_4, B_5)$$

$$MCC3 = (B_3, B_4, B_5, B_6)$$

Therefore the minimal cover is:

$$((B_1, B_2, B_3, B_5, B_7, B_8); \quad (B_3, B_4, B_5, B_6)) \tag{80}$$

Step 4: Finally, remove redundant columns from any of the MCCs of the minimal cover found in the previous step (i.e. $((B_1, B_2, B_7, B_8)$; (B_3, B_4, B_5, B_6). This results in the partition Π_G: where $\Pi_G = ((B_1, B_2, B_7, B_8); \quad (B_3, B_4, B_5, B_6))$

6.4 GRAPH COLORING APPROACH TO COLUMN MINIMIZATION

The following procedure uses the graph coloring approach from TRADE [75].

Execution of the Graph Coloring Algorithm

Step 1: Find all incompatible pairs of columns. Incompatible pairs of columns are found in the same way as in step 1 for the Set Covering approach. For simplicity let B_{ij} denote the incompatible pair $(B_i,\ B_j)$ and I_B be the set of incompatible pairs.

Then $I_B = (B_{16}, B_{24}, B_{26}, B_{47}, B_{48}, B_{67}, B_{68})$

The Incompatibility Graph is created as in Figure 18b with nodes corresponding to columns and edges representing incompatibility between the columns. It can be verified that this Incompatibility Graph is a complement of the Compatibility Graph from Figure 18a.

Step 2: Once the Incompatibility Graph has been constructed, start with the node which has the most edges and color that node with color A. Node 6 has the greatest number of adjacent edges

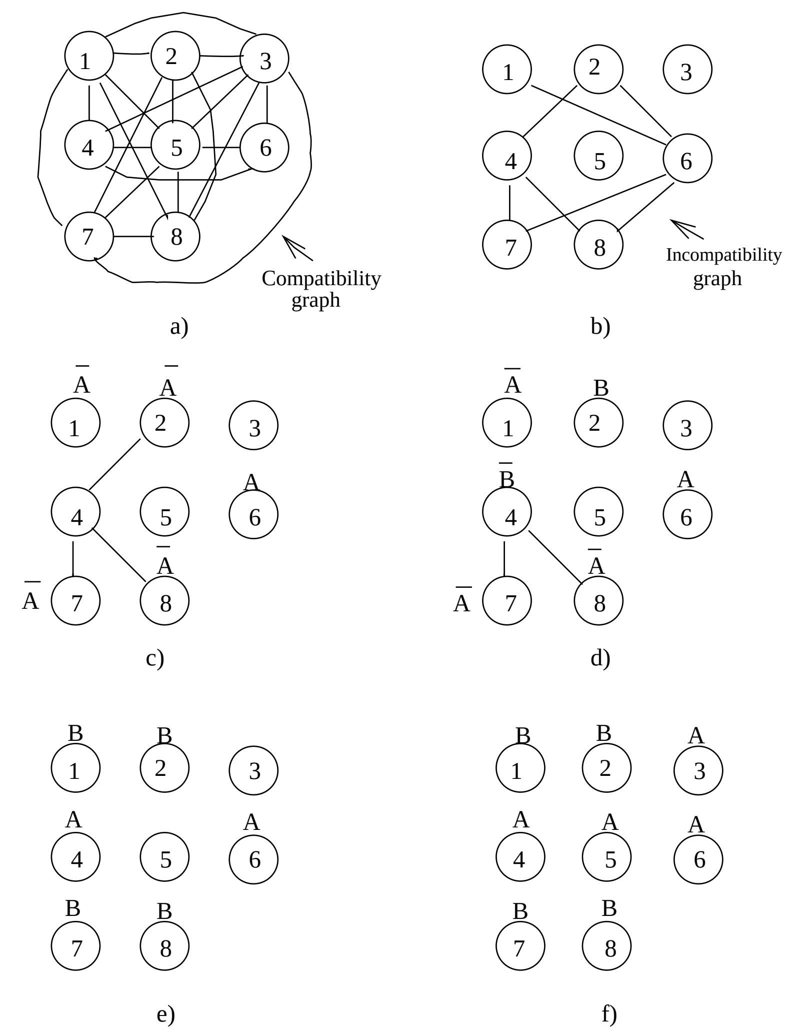


Figure 18: Incompatibility Graph for Example Function 1

and therefore it is colored with color A. Color all other nodes connected to that node in the Incompatibility Graph with the complement of A (denoted by $\overline{A}$). Then remove all edges to node colored with A. After this step has been completed, the graph appears as in Figure 18c.

Step 3: Next, color the next node which has the most "color-in-bars". The "color-in-bars" are the colors that a node may not be colored with. If more than one node has the same number of "color-in-bars" then choose the node which has the most edges. If there is still a tie, evaluate the influence of each color assignment, and then color the node which results in the minimum influence. Minimum influence is defined as the color assignment to a node which results in the greatest decrease in "color in bars". Repeat this process until all nodes connected by edges have been colored. For this example, the transitions from c to d and from d to e in Figure 18 correspond to this step.

Step 4: Remaining nodes may be colored with any color since they are compatible with all other nodes. In this example, nodes 3 and node 5 can be assigned to any color – randomly the color A was assigned to both of these nodes. This step corresponds to the transition from e to f in Figure 18.

Step 5: Form the desired partition Π_G by placing all columns with the same color in the same partitions. This results in the following partition for Π_G:

$$\Pi_G = ((B_1, B_2, B_7, B_8); \; (B_3, B_4, B_5, B_6)) \tag{81}$$

Comments

1. In another approach, the Graph Coloring algorithm (presented in section 5) can be applied to the Incompatibility Graph shown in Fig. 18b. The Reader may perform this coloring as an exercise. In future, we plan to have several graph coloring algorithms: exact and approximate.

2. The Graph Coloring algorithm from TRADE, illustrated above, can be easily converted to a tree search algorithm with backtracking, and still preserve the main advantage of its two heuristics: "colors in bars" and "order of coloring".

6.5 POSSIBLE APPROACHES TO THE COLUMN MINIMIZATION PROBLEM

The following should be considered for Machine Learning applications:

1. What is the best strategy for solving the Column Minimization Problem? Should it be solved together with Encoding, or separately? These issues are discussed in [64].

2. Assuming that the Column Minimization Problem is solved separately, is it better to use the Column Compatibility Graph or the Column Incompatibility Graph?

3. Assuming that the Column Compatibility Graph is used. What is the best method to find the groups of compatible columns?

   (a) By reducing to Set Covering with Maximum Cliques?

   (b) By reducing to Set Covering with large cliques found using heuristics?

   (c) By reducing to iterative finding and removing the maximum clique?

   (d) By reducing to iterative finding and removing large cliques?

   (e) By reducing to Graph Clique Partitioning?

   In addition, for each of the above formulations, several strategies in the sense of tree search strategies can be found, since each of the problems above, plus the problem of finding large or maximum cliques are the tree search problems.

4. Assuming that the Column Incompatibility Graph is used. What is the best method to find the groups of compatible columns (nodes)?

   (a) By Graph Coloring?

   (b) By finding Maximum Independent Set and iterating?

   (c) By finding large (but not necessarily maximum) Independent Set and iterating?

We have programs for:

1. finding one maximum clique

2. finding all maximum cliques

3. finding large clique
4. graph coloring
5. set covering

Some of these programs are already used in MULTIS. We plan to add other programs, and experiment with various search strategies as well as combined variants.

Some other interesting oportunities arise by considering new ways of creating the Compatibility and Incompatibility Graphs that will be introduced in section 7.

QUESTIONS FOR SELF-EVALUATION

1. Similarly as it was done in sections 4 and 5, write the pseudo-code of subroutines that collaborate with Universal Search Subroutine for the method of solving Column Minimization Problem based on iterative finding of large cliques.
2. Write the pseudo-code of subroutines that collaborate with Universal Search Subroutine for the graph coloring method from TRADE, but with added backtracking.
3. Show how the problems from this section can be solved using the Graph Coloring algorithm from section 5.
4. Solve in detail the problems outlined in Comments to section 6.4.

7 NEW APPROACH TO COLUMN MINIMIZATION

While in the previous section we compared two classical approaches to Column Minimization, here we will consider how they can be both improved by a new way of creating data for them. In subsection 7.1 another example function is solved, only this time a new formulation of the Graph Coloring problem is compared against the Set Covering approach, in section 7.4 another example function is presented to provide further illustration of the new approach that is presented in section 7.1, in section 7.5 comparisons are made between the new approach presented in step 1 of the Graph Coloring formulation and the existing approach used in step 1 for the Set Covering formulations of the compatibility problem.

7.1 COMPARISON OF THE NEW APPROACH VERSUS FORMER APPROACHES: FUNCTION F_2

In this subsection a Multiple-Valued-Input Multi-Output function will be used to compare the new approach to solving the "Column Compatibility Problem Combined with Graph Coloring" with the standard Set Covering approach to column minimization.

Cube	X_1	X_2	X_3	X_4	Y_1	Y_2
1	0	0	2	0	1	1
2	3	0	-	1	0	0
3	3	1	0	-	-	0
4	2	1	3	0	1	1
5	-	1	1	1	1	-
6	1	0	3	0	0	-
7	2	-	3	1	1	1
8	3	1	1	0	1	0

Table 3: Table for Example function F_2

Table 7.1 describes the next example function, F_2. The first column is the enumeration of cubes. The input variables are denoted X_1 thru X_4 and the output variables are Y_1 and Y_2.

The multi-valued map corresponding to Table 7.1 is shown in Fig. 19.

7.1.1 Problem Description

The following are partitions for the variables describing the function represented in the table. The blocks of each partition are formed by grouping together all cubes which have the same input values for the variable under consideration (or in the case of the output partition P_F, cubes are grouped according to common output classes).

Note: Semicolons are used to separate individual blocks of each partition.

$$
\begin{aligned}
P_1 &= (1,5;\ 2,3,5,8;\ 4,5,7;\ 5,6) \\
P_2 &= (1,2,6,7;\ 3,4,5,7,8) \\
P_3 &= (1,2;\ 2,3;\ 2,4,6,7;\ 2,5,8) \\
P_4 &= (1,3,4,6,8;\ 2,3,5,7) \\
P_F &= (1,4,5,7;\ 2,3,6;\ 3,5,8)
\end{aligned}
$$

Problem requirement: Find the decomposition $H(A, G(B, C))$ given the bound and free sets determined by the minimum admissibility.

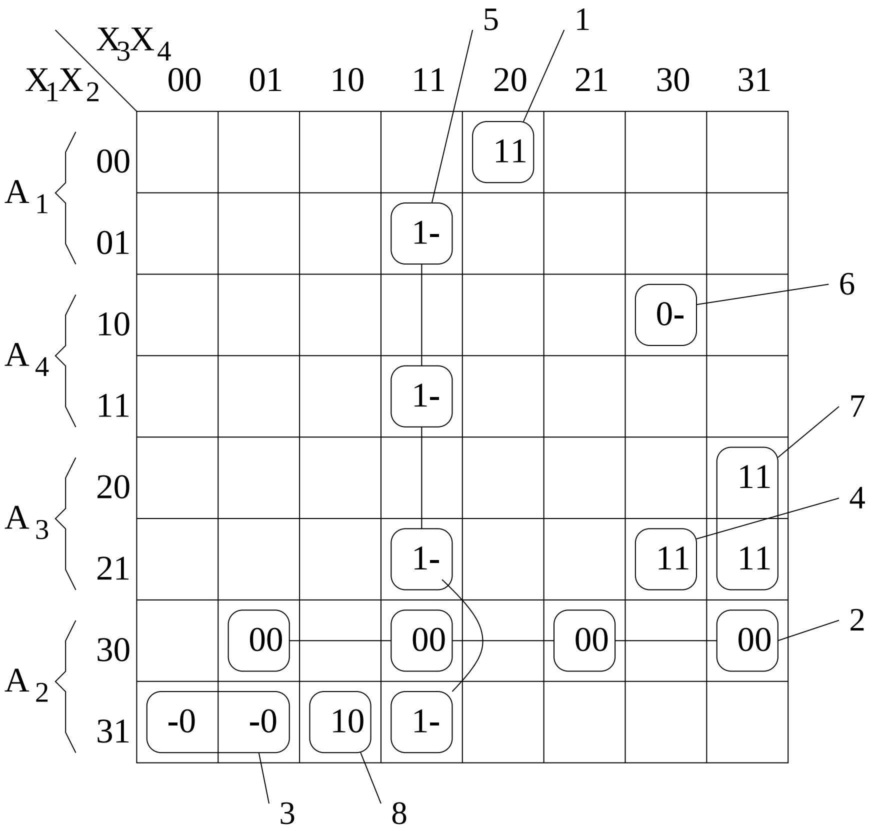


Figure 19: Multi-valued Map for Function F_2

7.1.2 r-admissibility used to determine bound and free sets

The choice of the bound and free sets depends on the specific problem requirements – sometimes using an estimate of the DFC as a measure. For simplicity, the only criterium used in this problem for determining bound and free sets was the minimum admissibility.

$P_1 \cdot P_F \;=\; (1,5;\; 2,3;\; 3,5,8;\; 4,5,7;\; 6)$

$P_1 \mid P_1 \cdot P_F \;=\; ((1,5);\; (2,3)(3,5,8);\; (4,5,7);\; (5)(6))$ (i.e. 2-admissible)

$P_2 \cdot P_F \;=\; (1,7;\; 2,6;\; 3,5,8;\; 4,7)$

$P_2 \mid P_2 \cdot P_F \;=\; ((1,7)(2,6);\; (3,5,8)(4,7))$ (i.e. 2-admissible)

$P_3 \cdot P_F \;=\; (1;\; 2,3;\; 4,7;\; 2,6;\; 5,8)$

$P_3 \mid P_3 \cdot P_F \;=\; ((1)(2);\; (2,3);\; (2,6)(4,7);\; (2)(5,8))$ (i.e. 2-admissible)

$P_4 \cdot P_F \;=\; (1,4;\; 2,3;\; 3,6;\; 3,5;\; 5,7;\; 3,8)$

$P_4 | P_4 \cdot P_F \;=\; ((1,4)(3,6)(8);\; (2,3)(5,7))$ (i.e. 3-admissible)

Similarly, one can find the r-admissibility for all combinations of bound and free sets, from which the one with the lowest admissibility is chosen for the decomposition. As it turns out P_1, P_2, and P_3 all yield the lowest admissibility of 2. Randomly P_1 was chosen for the partition for the free set $P(A)$. Hence the partition on the bound set $P(B)$ is $P_2 \cdot P_3 \cdot P_4$. The following partitions are the assumed bound, free, and output partitions used for the Set Covering approach as well as for the graph coloring approach.

$$P(A) = P_1 = (1,5;\ 2,3,5,8;\ 4,5,7;\ 5,6) = (A_1, ..., A_4)$$

$$P(B) = P_2 \cdot P_3 \cdot P_4 = (1;\ 4;\ 3;\ 2;\ 5;\ 8;\ 7;\ 6) = (B_1, ..., B_8)$$

$$P_F = (1,4,5,7;\ 2,3,6;\ 3,5,8) = (P_{F_1}, ..., P_{F_3})$$

7.1.3 Decomposition Using the Set Covering Approach

General description:
The basis of the following Set Covering approach by Luba et al [40] is to find all pairwise compatible blocks from the bound set and then incrementally build MCC's from the pairwise CC's until all blocks in the bound set have been assigned at least one MCC. Next, the minimum number of MCCs is selected, which will completely cover the blocks of the bound set. Finally, the redundant blocks are eliminated from the cover set to form the desired partition Π_G.

Specific problem description:

$$P(A) = P_1 = (1,5;\ 2,3,5,8;\ 4,5,7;\ 5,6)$$

and

$$P(B) = P_2 \cdot P_3 \cdot P_4 = (1;\ 4;\ 3;\ 2;\ 5;\ 8;\ 7;\ 6) = (B_1, ..., B_n)$$

where blocks of $P(B)$ are denoted B_i for $1 \leq i \leq n$ and

$$P_F = (1,4,5,7;\ 2,3,6;\ 3,5,8)$$

Execution of the Algorithm

Step 1: Find all compatible pairs of blocks from $P(B)$ such that

$$P(A) \cdot (B_i \cup B_j) \leq P_F$$

For simplicity let B_{ij} denote $B_i \cup B_j$. The set of all such pairs of blocks that satisfy $P(A) \cdot B_{ij} \leq P_F$ is denoted Π_G^δ.

$$\begin{aligned}\Pi_G^\delta = (&B_{12}, B_{13}, B_{14}, B_{15}, B_{16}, B_{17}, B_{18}, B_{23}, B_{24}, B_{25}, B_{26}, B_{27},\\ &B_{28}, B_{34}, B_{35}, B_{36}, B_{37}, B_{38}, B_{47}, B_{48}, B_{56}, B_{57}, B_{67}, B_{68}, B_{78})\end{aligned} \tag{82}$$

Figure 20 presents the respective Compatibility Graph with edges B_{ij} from Π_G^δ.

Step 2: Find the largest CC's (MCC's-maximum compatible classes) from the compatible pairs found in step 2.

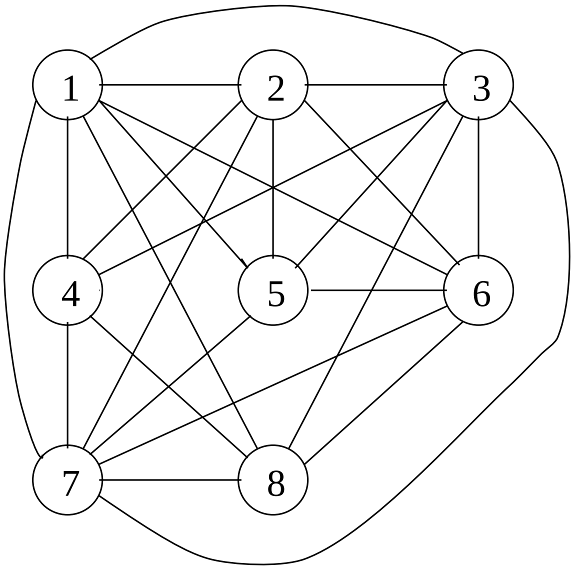


Figure 20: Compatibility Graph for Function F_2

$CC_1 = 1$

$CC_2 = (1,2)$

$CC_3 = (1,2,3)$

$CC_4 = (1,2,3,4)$

$CC_5 = (1,2,3,4),(1,2,3,5)$

$CC_6 = (1,2,3,4),(1,2,3,5,6)$

$CC_7 = (1,2,3,4,7),(1,2,3,5,6,7)$

$CC_8 = (1,2,3,4,7,8),(1,2,3,5,6,7)$

Step 3: Find the MCC's which will cover P(B) completely. The CC's in the last row of the previous step form the MCC's for this function on the given bound set.

$$MCC1 = (B_1, B_2, B_3, B_4, B_7, B_8)$$

$$MCC2 = (B_1, B_2, B_3, B_5, B_6, B_7)$$

Therefore the minimal cover is:

$$((B_1, B_2, B_3, B_4, B_7, B_8); \;\; (B_1, B_2, B_3, B_5, B_6, B_7))$$

Step 4: Finally, remove any repeated blocks from any of the classes of the minimal cover found in the previous step $((B_1, B_2, B_3, B_4, B_7, B_8); \;\; (B_5, B_6))$. This results in the partition Π_G which satisfies the requirement $P(A) \cdot \Pi_G \leq P(F)$,

where $\Pi_G = (1, 2, 3, 4, 6, 7; \; 5, 8)$ is the resulting partition of cubes.

Therefore the resulting function is $F = H(x_1, G(x_2, x_3, x_4))$ where G is a single-output function.

Note: The DFC of the resulting decomposition is 32 as compared with the DFC of the non-decomposed function which was 128.

7.2 A NEW APPROACH TO SOLVING THE COLUMN COMPATIBILITY PROBLEM

In this subsection we will present a new approach to solve the Column Compatibility Problem that combines the Partition-Based Method with the Graph Coloring method.

The new approach explained below finds exactly the same incompatible blocks of the bound set that could be found using the approach from section 6. However, they are found now indirectly and at a significant savings in the number of calculations executed. As an additional bonus, the new approach can produce even more savings in computational cost when the non-disjoint decompositions are to be performed.

In the Set Covering approach, the first set to be calculated is the set of "compatible" pairs of columns whereas in the Graph Coloring approach the first set to be calculated is the set of "incompatible" relationship pairs between columns. The incompatible columns that must be calculated for the Graph Coloring approach can be found in the same way as the compatible pairs of columns are found in the Set Covering approach. However, there exists another way to calculate the incompatible columns which significantly reduces the number of calculations required. This other way is the new approach that is presented in the next section.

General description:

A brief explanation of the decomposition using this approach is as follows:

1) Find the cubes "within" each block of the partition on the free set $P(A)$ which belong to the same output class. Group them accordingly. Repeat seperately for each block of $P(A)$.

2) Using these incompatible classes of cubes, generate the corresponding incompatibility classes of blocks. From these incompatible classes of blocks the incompatibility *"Block Graph"* can be formed.

3) Color the nodes of the incompatibility "Block Graph" with the minimum number of colors such that no two incompatible nodes connected by an edge are given the same color.

4) Each color corresponds to a separate partition block of Π_G. Assign all blocks which have the same color to the same partition block.

Specific Problem Description For Function F_2

Repeated below are the partitions for the free set, bound set, and and output partition.

$$P(A) = P_1 = (1,5;\ 2,3,5,8;\ 4,5,7;\ 5,6) = (A_1, ..., A_4)$$

$$P(B) = P_2 \cdot P_3 \cdot P_4 = (1;\ 4;\ 3;\ 2;\ 5;\ 8;\ 7;\ 6) = (B_1, ..., B_8)$$

$$P_F = (1,4,5,7;\ 2,3,6;\ 3,5,8) = (P_{F_1}, ..., P_{F_3})$$

Thus:

$$A_1 = (1,5); \quad A_2 = (2,3,5,8); \quad A_3 = (4,5,7); \quad A_4 = (5,6)$$

$$B_1 = (1);\ B_2 = (4);\ B_3 = (3);\ B_4 = (2);\ B_5 = (5);\ B_6 = (8);\ B_7 = (7);\ B_8 = (6)$$

$$P_{F_1} = (1,4,5,7); \quad P_{F_2} = (2,3,6); \quad P_{F_3} = (3,5,8)$$

7.2.1 The Block Algorithm for Column Minimization by Graph Coloring

This algorithm is a new approach which greatly reduces the number of calculations required to determine the column compatibility. Note that blocks of the free set and bound set are sometimes referred to as the rows and columns, respectively. This is in reference to rows and columns of a Karnaugh map, in order to provide an aid to the description of the approach. The basic idea of this approach is to find pairs of incompatible classes of cubes and then replace these incompatible classes of cubes with incompatible classes of columns (blocks of $P(B)$) which contain those cubes.

Algorithm 7.1

a) Classify cubes within each block of $P(A)$ according to output classes they belong to (i.e. $\forall i\, \forall j$ let $A_i \cdot P_{F_j} = I_{ij}$, where I_{ij} is a class of cubes from block A_i that are elements of the same output class P_{F_j}). Each of these classes, extracted from within each row A_i, are incompatible with each other, and are therefore referred to as incompatible classes.

$$I_{ij} = A_i \cdot P_{F_j}\ for\ all\ \ I_{ij}$$

b) Part b) is executed **only** for multi-output decompositions. If multi-output functions are split into multiple single-output functions then this part would be by-passed. However, if the option to split up a multi-output function is not chosen, then the following procedure should be used.

   Find all cubes which are incompatible with the "repeated cubes" (i.e. find all I_{r_k} where I_{r_k} denotes the class of cubes which are incompatible with the repeated cube r_k). Repeated cubes are cubes which appear in more than one output class. This is accomplished as follows:

   Let r_k denote repeated cubes and let R_{ir} denote the set of cubes which belong to at least one class I_{ij} which cube r_k also belongs to. (R_{ir} represents the set of all "care" cubes which are compatible with cube r_k in row i).

   i) For each row i, find r_k & R_{ir}. All repeated cubes r_k can be found by taking the intersection of each pair of output blocks for all combinations. This results in the repeated cubes $r_1 = 3$ and $r_2 = 5$ as can be seen from P_F. For R_{ir}: $\forall j\ if\ r_k \in I_{ij}$ then $R_{ir} = R_{ir} \cup I_{ij}$ (R_{ir} is initially empty).

ii) For each row i with $r = r_k$, if $R_{ir} \neq \emptyset$, do the following: check $\forall j\ if \ \ r_k \notin I_{ij}$ then $I_{r_k} = I_{r_k} \cup ((I_{ij} \cap R_{ir}) \oplus I_{ij})$ where I_{r_k} denotes the class of cubes which are incompatible with r_k.

c) For decompositions where multi-output functions are not split up into single-output functions, remove the repeated cubes from all incompatible classes I_{ij}. Otherwise, don't make any changes to I_{ij}.

d) For each pair of incompatible classes of cubes $(I_{ij}, I_{ik}) \in I_c$, generate the corresponding pair of incompatible classes of columns $(I^B_{ij}, I^B_{ik}) \in I_B$ (where I_B is the set of pairs of incompatible classes of columns generated from I_c). This is done by identifying all blocks in P(B) which have at least one cube in common with a particular class of incompatible cubes and then placing them together in a class I^B_{ij}. This is done for all classes in I_c. If we let I_{ij} denote each of the individual classes within I_c. Then I_B is found accordingly:

$$\forall i \forall j \forall k\ if\ I_{ij} \cap B_k \neq \emptyset\ then\ add\ B_k\ to\ class\ I^B_{ij}$$

7.2.2 Step by Step Execution of Algorithm 7.1 on Example Function F_2

a) For A_1 = (1,5) we have:

$$I_{11} = (1,5), \quad I_{12} = \emptyset, \quad I_{13} = (5).$$

For A_2 = (2,3,5,8) we have:

$$I_{21} = (5), \quad I_{22} = (2,3), \quad I_{23} = (3,5,8).$$

For A_3 = (4,5,7) we have:

$$I_{31} = (4,5,7), \quad I_{32} = \emptyset, \quad I_{33} = (5).$$

For A_4 = (5,6) we have:

$$I_{41} = (5), \quad I_{42} = (6), \quad I_{43} = (5).$$

b) Execution of the algorithm for this step results in the following:

For cube 3 we have:

$R_{13} = \emptyset$,
$R_{23} = (2,3,5,8)$,
$R_{33} = \emptyset$,
$R_{43} = \emptyset$.

For cube 5 we have:
$R_{15} = (1,5)$,
$R_{25} = (3,5,8)$,
$R_{35} = (4,5,7)$,
$R_{45} = (5)$.

ii) For cube $r_1 = 3$, we find that $R_{i3} = \emptyset$ for i = 1,3 and 4. This means that cube 3 is not an element of rows A_1, A_3 or A_4 and therefore cannot be incompatible with any cubes in these rows. Since $R_{23} = (2,3,5,8)$ we know that cube $r_1 = 3$ is an element of row A_2 and that it is only compatible with cubes 2,3,5, and 8. There are no other cubes besides the compatible cubes 2,5 and 8 in row A_2, therefore there are no incompatible cubes with cube 3 in row A_2 as well as in rows A_1, A_3 and A_4. What does this mean? It means that since there are no incompatible cubes with cube 3 in within any row that it occupies, we are guaranteed that cube 3 will not be a factor in preventing any columns from being combined.

However, for cube $r_2 = 5$, we find that $R_{25} = (3,5,8)$ and since cube 5 is not an element of I_{22}, then any cubes in I_{22} which are not contained in R_{25} are incompatible with cube 5. It turns out that cube 2 is an element of I_{22} but not an element of R_{25} and is therefore incompatible with cube 5. Similarly in row A_4, we check membership in the classes I_{4j} and we find that cube 5 is incompatible with cube 6. Therefore using the formula above for I_{r_k} we were able to find that cubes 2 and 6 are incompatible with cube 5. These incompatible cubes form the pairs ((5),(2)) and ((5),(6)) or ((5),(2,6)), which is of the form $((r_k),(I_{r_k}))$.

c) In this example, it was decided to use a multi-output function in order to illustrate all parts of this approach. Therefore, the following is the set of incompatible classes I_{ij} after the repeated cubes (cubes 3 and 5) were removed.

For A_1 = (1,5):

$I_{11} = (1) \quad I_{12} = \emptyset \quad I_{13} = \emptyset$

For A_2 = (2,3,5,8):

$I_{21} = \emptyset \quad I_{22} = (2) \quad I_{23} = (8)$

For A_3 = (4,5,7):

$I_{31} = (4,7) \quad I_{32} = \emptyset \quad I_{33} = \emptyset$

For A_4 = (5,6):

$I_{41} = \emptyset \quad I_{42} = (6) \quad I_{43} = \emptyset$

Now pair all combinations of incompatible classes corresponding to each row (ignore pairs containing an emptyset as one of its two classes). Each pair is of the form (I_{ij}, I_{ik}).

Then include all pairs from all rows together to form the list of incompatible pairs of classes I_c (also include the pairs from part b) if there are any, and treat the same as other pairs (I_{ij}, I_{ik})). Pairing all the classes (I_{ij}, I_{ik}) we find only one pair ((2),(8)) which does not include $\emptyset$. The following is the resulting list of incompatible pairs of classes:

I_c = (((2),(8)), ((5),(2)), ((5),(6))).

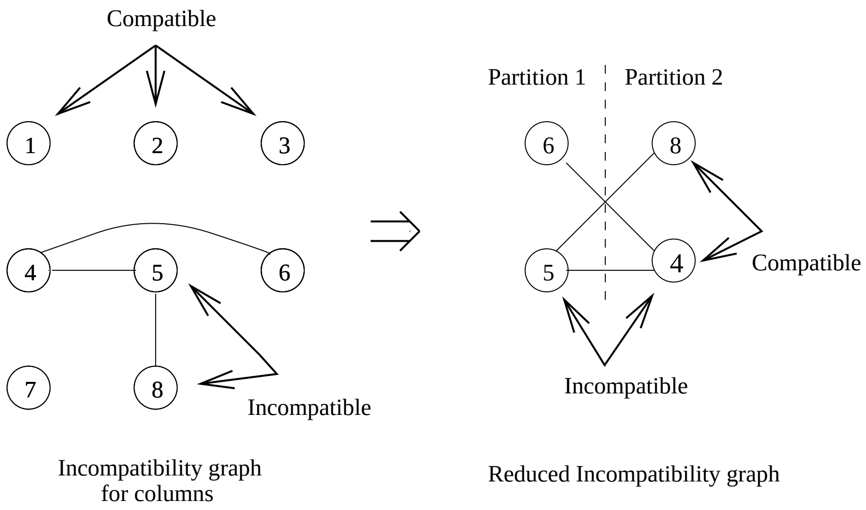


Figure 21: Incompatibility Graphs for function F_2

d) This step results in the following pairs of incompatible columns:

$$I_B = ((4),(6)),\ ((5),(4)),\ ((5),(8))$$

These incompatible pairs of columns represent the Incompatibility Graph shown in Figure 21. For this simple example, there is only one column in each class. However, for larger examples, classes will often times have several columns within each class. It is important to note that every column in one class is incompatible with all columns in the opposite class within a pair of classes.

7.3 CONTINUATION OF THE COLORING APPROACH FOR FUNCTION F_2

The next two steps of the Graph Coloring Approach include:

1. Coloring the Reduced Incompatibility Graph (this can be done with an adaptation of any of the discussed previously graph coloring algorithms).
2. Elimination of redundant blocks (optional).

   Actually, skipping this step allows much greater flexibility for the encoding. However, algorithms for encoding must be modified to ensure an equivalence relation between the original function and the decomposed function.

Coloring the Reduced Incompatibility Graph. Color the reduced incompatibility graph with the minimum number of colors such that no two nodes connected by an edge are given the same color. The number of colors required corresponds to the number of partition blocks in Π_G. Assign all columns with the same color to the same partition. Finally, assign the remaining blocks (the blocks which are not connected by any edges in the incompatibility graph) to any of the partitions already established. Using an algorithm for Graph Coloring such as the ones from sections 5 and 6, we find that nodes corresponding to columns 4 and 8 may be assigned the same color and nodes 5 and 6 may be assigned another color different than the color which nodes 4 and 8 are colored with.

Elimination of Redundant Blocks (optional).

Eliminate any redundant blocks from the partitions to form the desired partition Π_G.

$$\Pi_G = ((B_4, B_8, ...);\ (B_5, B_6, ...))$$

or

$$\Pi_G = ((B_4, B_8, B_1, B_2, B_3, B_7);\ (B_5, B_6, B_1, B_2, B_3, B_7))$$

In the case of this example, several choices for Π_G are available. Which choice is the best, it is very difficult to determine in general, as it requires "looking ahead" to the next decomposition. Shown below are the possible choices for Π_G after redundant blocks have been removed and the remaining blocks are replaced with the cubes that were elements of these blocks.

Possible choices for Π_G are:

$$\Pi_G = ((2, 6, ...);\ (5, 8, ...))$$

or

$$\Pi_G = ((2, 6);\ (5, 8, 1, 3, 4, 7))$$

. . .

$$\Pi_G = ((2, 6, 1, 3);\ (5, 8, 4, 7))$$

. . .

$$\Pi_G = ((2, 6, 1, 3, 4, 7);\ (5, 8))$$

7.4 ANOTHER EXAMPLE ILLUSTRATING THE NEW APPROACH

The following are the partitions of the free set, bound set and output set corresponding to function F_3 shown in Figure 22. Each number corresponds to a minterm in the figure.

$$\begin{aligned} P(A) = {} & ((2,3,5,6,7,8)\,(11,13,14,16)\,(17,19,22,24)\,(28,29,32) \\ & (33,34,35,38,40)\,(42,44,48)\,(53,54,55)\,(58,59,60,62)) = (A_1 \ldots A_8) \end{aligned} \quad (83)$$

$$\begin{aligned} P(B) = {} & ((8,17,33)\,(7,34,42,58)\,(6,11,19,35,59)\,(5,28,44,60)\,(13,29,53) \\ & (3,14,22,38,54,62)\,(2,55)\,(16,24,32,40,48)) = (B_1 \ldots B_8) \end{aligned} \quad (84)$$

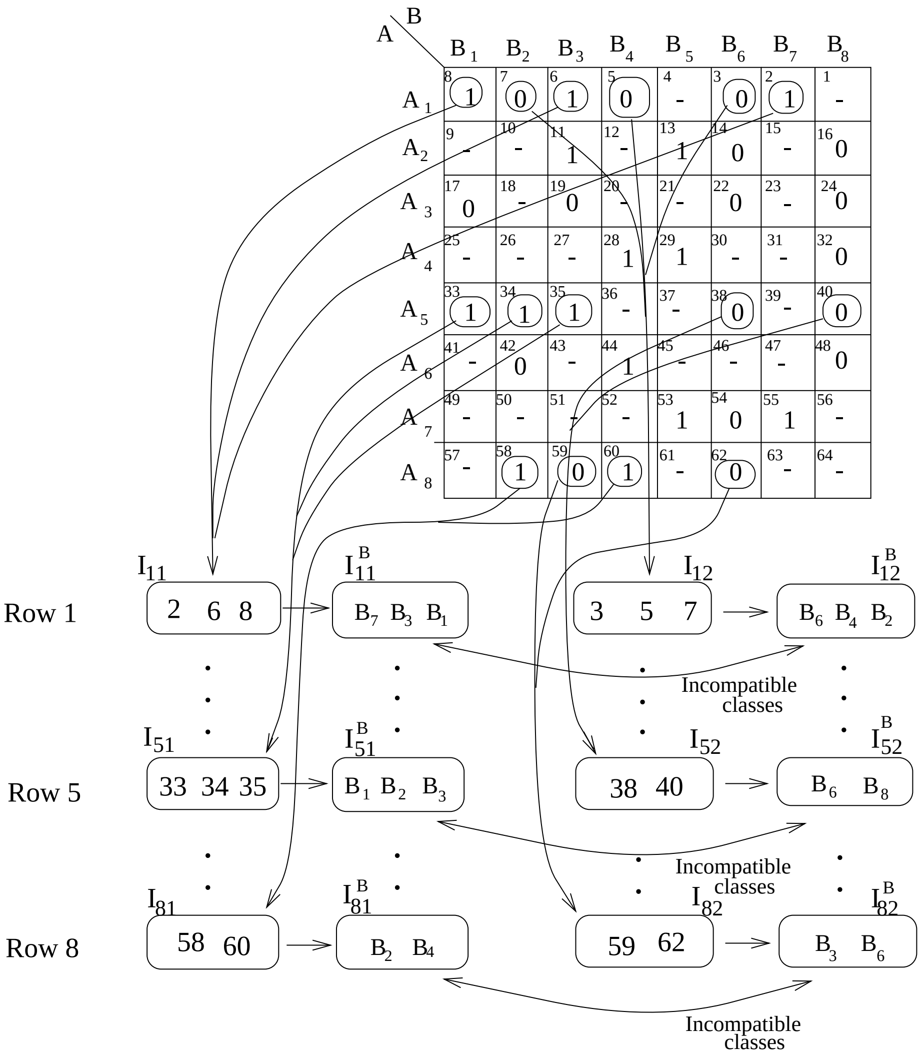


Figure 22: Karnaugh Map for Function F_3

$$
\begin{aligned}
P_F &= ((2,6,8,11,13,28,29,33,34,35,44,53,55,58,60) \\
&\quad (3,5,7,14,16,17,19,22,24,32,38,40,42,48,54,59,62)) \;=\; (P_{F_1} P_{F_2}) \qquad (85)
\end{aligned}
$$

Step By Step Execution of Algorithm 7.1 for function F_3.

Algorithm 7.1 is executed as follows.

Find the pairs of incompatible classes of columns for the function described by the above partitions:

a) Classify cubes within each block of P(A) according to output classes they belong to(ie. $\forall i \forall j \;\; I_{ij} = A_i \cdot P_{F_j}$ where I_{ij} is a class of cubes from block A_i that are elements of the same output class P_{F_j}).

$I_{11} = (2,6,8)$, $I_{12} = (3,5,7)$,

$I_{21} = (11,13)$, $I_{22} = (14,16)$,

$I_{31} = (17,19,22,24)$, $I_{32} = \emptyset$,

$I_{41} = (28,29)$, $I_{42} = (32)$,

$I_{51} = (33,34,35)$, $I_{52} = (38,40)$,

$I_{61} = (44)$, $I_{62} = (42,48)$,

$I_{71} = (53,55)$, $I_{72} = (54)$,

$I_{81} = (58,60)$, $I_{82} = (59,62)$.

b) Skip part b) since this is a single output function.

c) Now pair all combinations of incompatible classes corresponding to each row (ignore pairs containing an emptyset as one of its two classes) to form the list of incompatible pairs of classes I_c. Each pair is of the form$(I_{ij},\ I_{ik})$. The following is the resulting list of incompatible pairs of classes:

$I_c =$

(((2,6,8), (3,5,7)),

((11,13), (14,16)),

((28,29), (32)),

((33,34,35), (38,40)),

((44), (42,48)),

((53,55), (54)),

((58,60), (59,62))).

d) For each pair of incompatible classes of cubes $(I_{ij}, I_{ik}) \in I_c$, generate the corresponding pair of incompatible classes of columns $(I_{ij}^B, I_{ik}^B) \in I_B$ (where I_B is the set of pairs of incompatible classes of columns generated from I_c). This is done by identifying all blocks in P(B) which have at least one cube in common with a particular class of incompatible cubes and then placing them together in a class I_{ij}^B.

This is done for all classes in I_c. I_B is found accordingly:

$$\forall i\ \forall j\ \forall k\ if\ \ I_{ij} \cap B_k \neq \emptyset,\ then\ add\ B_k\ to\ class\ I_{ij}^B$$

$I_{11}^B = (7,3,1)$, $\quad I_{12}^B = (6,4,2)$,

$I_{21}^B = (3,5)$, $\quad I_{22}^B = (6,8)$,

$I_{41}^B = (4,5)$, $\quad I_{42}^B = (8)$,

$I_{51}^B = (1,2,3)$, $\quad I_{52}^B = (6,8)$,

$I_{61}^B = (4)$, $\quad I_{62}^B = (2,8)$,

$I_{71}^B = (5,7)$, $\quad I_{72}^B = (6)$,

$I_{81}^B = (2,4)$, $\quad I_{82}^B = (3,6)$,

Note that pair $((I_{31}^B),\ (I_{32}^B))$ is not included because it would be a nonsense to have columns that are incompatible with an empty set.

Therefore I_B =

(((7,3,1), (6,4,2)),

((3,5), (6,8)),

((4,5), (8)),

((1,2,3), (6,8)),

((4), (2,8)),

((5,7), (6)),

((2,4), (3,6))).

Note that each column in a pair of classes is incompatible with all columns in the opposite class. Each of these pairs represents the sets of columns which have a conflicting output in a particular row and therefore cannot be combined. Also, each of these pairs of classes could be broken down into pairs of incompatible columns in an extra step if desired (i.e. ((3,5), (6,8)) could be broken down into the pairs ((3),(6)), ((3),(8)), ((5),(6)), and ((5),(8)). However, this is not necessary. In fact, the larger classes can actually help speed-up the coloring process in the next step.

The coloring step is executed as follows.
The following graph coloring description is based on the approach by Wei Wan [75].
Color the columns with the minimum number of colors needed.
Recall that the minimum number of colors corresponds to the minimum column multiplicity and hence the minimum number of partitions required for the sub-function Π_G.

1. Start by coloring the column which has the greatest number of edges color A.
2. Then for each class in which that column appears, color all columns included in the opposite class with $\overline{A}$.
3. Once a column has been assigned a new color, it is removed from that class as soon as the columns in the opposite class have been colored with it's "color-in-bar".
4. Check all the remaining classes for occurences of the column that was assigned color A.
5. When an occurrence of this column is encountered, color all blocks in the opposited class $\overline{A}$ and remove the occurrence of the column colored with color A.
6. Similarly, repeat for the remainder of the columns (until no more colors are required): color the column which cannot be colored with any of the previous colors with a new color and color all columns which appear in classes opposite this column with it's "color-in-bar".
7. If more than one column cannot be colored with any of the previous colors, then assign the next new color to the column which has the greatest number of edges.
8. If there are more than one column which has the same number of edges, then color the column which results in the minimum influence (the *minimum influence* is defined as the color assignment to a column which results in the greatest decrease in "color-in-bars").

See Figure 23 for a breakdown of the steps in the graph coloring process for this example. Note that columns which have not been assigned a color before the minimum number of colors have been assigned, can be assigned to any of the colors for which they do not have the corresponding "color-in-bar". For simplicity, assign the columns which can be assigned to more than one color, to any of the color options available. Since each color corresponds to a separate partition block, the color assignment for this example results in the following partition:

$$P_G = ((6,8)(2,5)(1,3,5,7)(4,5))$$

The Elimination Step (this step is optional). Redundant columns are eliminated from the partitions formed in the previous step to form Π_G. For simplicity, the redundant columns are removed from the largest partition blocks resulting in the desired partition Π_G.

$$\Pi_G = ((6,8)(2,5)(1,3,7)(4))$$

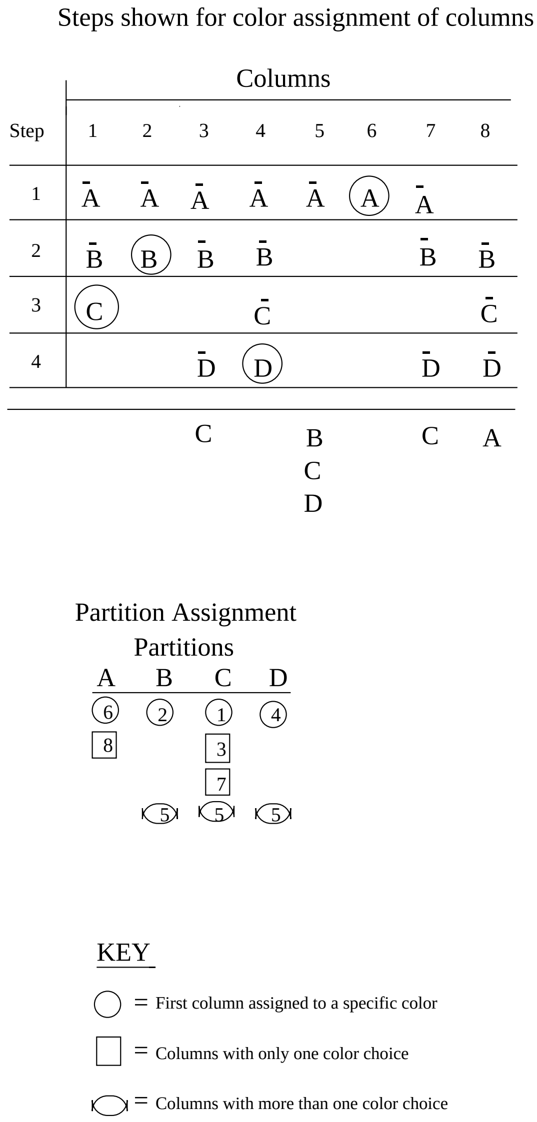


Figure 23: Breakdown of steps in Graph Coloring

7.5 ANALYSIS OF THE NEW GRAPH COLORING APPROACH VERSUS THE SET COVERING APPROACH USED BY LUBA

In this section we will perform an analysis of the new approach introduced in the Graph Coloring example (section 7.1) versus the corresponding approach used with the Set Covering formulation of the compatibility problem.

The following are expressions used in each approach for finding column compatibility and the worst case formulas used to determine the number of calculations (intersection and union operations) required by each.

Note: The following analysis is done only for single output funtions. The reasons for this are:

1) because any multiple-output function may be replaced by multiple single-output functions and

2) because the formula for single-output functions for the new approach is much simpler to work with.

For step 1 using the Set Covering approach of Luba et al, [40], the expression for finding pair-wise column compatibility is:

$$P(A) \cdot (B_i \cup B_j) \leq P_F \tag{86}$$

or more specifically,

$$A_k \cdot (B_i \cup B_j) \leq P_{F_m} \tag{87}$$

and the number of required calculations is:

$$AEP \;=\; R \times 0 \times \binom{C}{2} \tag{88}$$

where

$$\binom{n}{r} \;=\; \frac{n!}{r!\,(n-r)!} \tag{89}$$

For step 1 using the new approach, the expression for finding pairs of incompatible classes of columns is:

$$P(A) \cdot P_F \cdot P(B) \tag{90}$$

or more specifically,

$$A_k \cdot P_{F_m} \cdot B_i \tag{91}$$

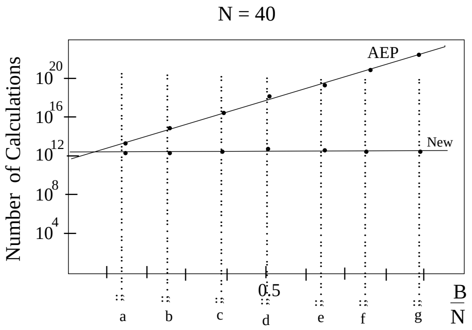
and the number of calculations required is:

$$New \;=\; R \times O \times C \tag{92}$$

Variables defined:

AEP = stands for "approach existing previously"

	N	A	B	AEP	New
a	40	35	5	3.4×10^{13}	2.2×10^{12}
b	40	30	10	1.1×10^{15}	2.2×10^{12}
c	40	25	15	3.6×10^{16}	2.2×10^{12}
d	40	20	20	1.1×10^{18}	2.2×10^{12}
e	40	15	25	3.7×10^{19}	2.2×10^{12}
f	40	10	30	1.2×10^{21}	2.2×10^{12}
g	40	5	35	3.8×10^{22}	2.2×10^{12}



	N	A	B	AEP	New
h	30	25	5	3.3×10^{10}	2.1×10^{9}
i	30	20	10	1.1×10^{12}	2.1×10^{9}
j	30	15	15	3.5×10^{13}	2.1×10^{9}
k	30	10	20	1.1×10^{15}	2.1×10^{9}
l	30	5	25	3.6×10^{16}	2.1×10^{9}

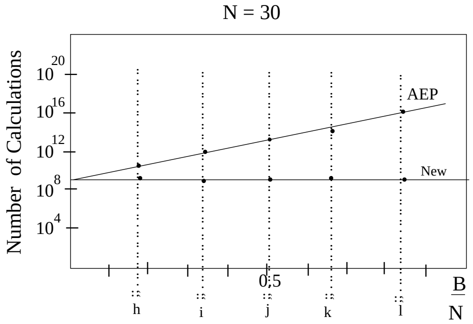


Figure 24: Plots of the two approaches represented by the formulas AEP and New for a constant total number of variables(N)

$$
\begin{aligned}
A_k &= \text{individual blocks of the free set } P(A),\\
B_k &= \text{individual blocks of the bound set } P(B),\\
P_{F_m} &= \text{individual blocks of the output set } P_F,\\
\mid A \mid &= \text{number of variables in the free set},\\
\mid B \mid &= \text{number of variables in the bound set},\\
C = 2^{|B|} &= \text{number of columns in the bound set},\\
R = 2^{|A|} &= \text{number of rows in the free set},\\
Y &= \text{number of output variables},\\
O = 2^{|Y|} &= \text{number of blocks in the output partition.}
\end{aligned}
$$

Figure 24 presents plots of the two approaches represented by the formulas AEP and New for a constant total number of variables (N) with varying numbers of variables in the bound and free sets. Note that when the number of variables in the bound set is much larger than the number of variables in the free set, there exists a several orders of magnitude difference in the number of calculations (intersections and unions) required.

Similarly in Figure 25, we see several orders of magnitude difference in the number of calculations when the number of variables in the bound set are much greater than the number of variables in the free set.

	N	A	B	AEP	New
m	15	10	5	1.0×10^{6}	6.5×10^{4}
n	20	10	10	1.1×10^{9}	2.1×10^{6}
o	25	10	15	1.1×10^{12}	6.7×10^{7}
p	30	10	20	1.1×10^{15}	2.1×10^{9}
q	35	10	25	1.1×10^{18}	6.9×10^{10}
r	40	10	30	1.2×10^{21}	2.2×10^{12}
s	45	10	35	1.2×10^{24}	7.0×10^{13}

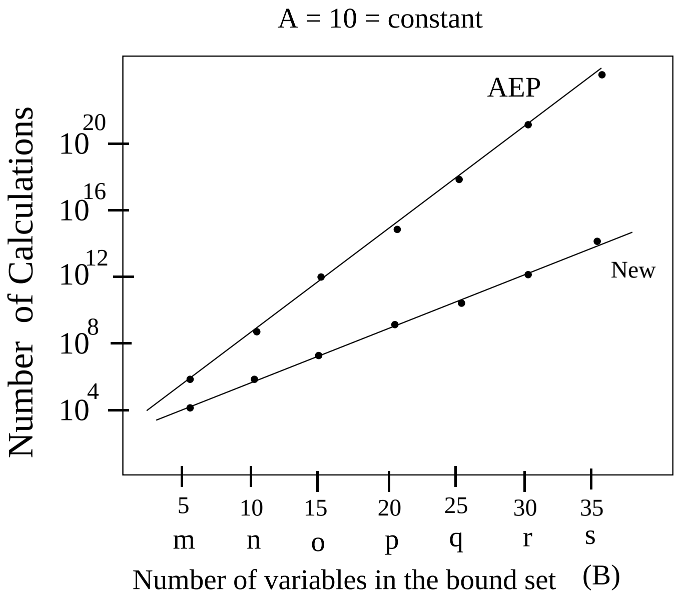


Figure 25: Plots of the two approaches represented by the formulas AEP and New when the number of variables in the bound set are much greater than the number of variables in the free set

Concluding, the worst case analysis of the two approaches compared in sections 6 and 7 illustrates dramatic differences in the number of calculations required by each of the approaches. Though, it can be expected that actual results for computation time required for each approach would not differ as dramatically as for the worst case number of calculations illustrated. The comparisons made do however suggest a potential for significant savings using the new approach introduced in this section. The variants of this approach for Graph Coloring and for Set Covering should be created and evaluated on very large and sparse functions. A further analysis will be required in order to determine a more accurate assessment of the potential for the new approach. This would include implementing the algorithm in a program and comparing the results on numerous benchmark examples (which is currently being done by us).

The proposed approach can be used with Set Covering as well as with Graph Coloring. However, using the new approach with Set Covering requires an additional cost in converting from incompatibility classes to compatible classes. Though not every single comparison is calculated for each graphical approach, the shear number of comparisons that can be calculated, indicate that there are far more calculations required in more steps using the Set Covering approach than in the Graph Coloring approach. However, the Graph Coloring approach and the Set Covering approach need to be analyzed more carefully in the context of numerous Machine Learning examples to determine the impact on the overall cost of each approach.

QUESTIONS FOR SELF-EVALUATION

1. Complete the decomposition of function F_2 from this section to the stage of final decomposition and circuit.
2. Do the same for function F_3.
3. Perform the decomposition of function F_2 using classical decomposition approach. Compare.
4. Perform the decomposition of function F_2 using classical decomposition approach. Compare.

8 VARIABLE PARTITIONING SEARCH STRATEGY THAT IMPROVES THE TRADE STRATEGY

In previous sections we learned how tree search strategies and related methodologies can be used for the Column Minimization Problem in Functional Decomposition.

Another very important and difficult problem in functional decomposition is that of finding the sets of bound, free and shared variables. This is called the *Variable Partitioning Problem.* Various variable partitioning algorithms were implemented in TRADE, DEMAIN and GUD as component decomposers of MULTIS. In this and next section we will present several approaches to the Variable Partitioning Problem.

Finding a good bound set of input variables is one of the most important problems to be solved in Ashenhurst/Curtis decomposition (both disjoint and non-disjoint), and also in the generalized decompositions, such as those discussed by Steinbach, Luba, and Perkowski. The variable partitioning problem can not be solved exhaustively, since it is exponential or super-exponential (depending on the formulation of the decomposition problem). Therefore, different authors are looking for good variable partitioning heuristics. Especially remarkable are the methods proposed by Tadeusz Luba, Bernd Steinbach, Craig Files [15], and Haomin Wu [78]. In our final report we will compare the above, and also other methods, and we will create a new method that will take into account particular ideas of most of these methods.

In this section, the emphasis will be on comparing current methods of variable partitioning, and we will suggest a few possible new methods and combinations of old methods to produce a smarter and quicker method of Variable Partitioning.

8.1 COMPARISON OF CURRENT METHODS FOR VARIABLE PARTITIONING

Method	Don't Cares	m outp	m-valu	Number of Bond Sets Tested	Node Selection Cost	Search Strategy	Possible Improvements
Luba r-admis	Yes	+	+	All	good	no	not sure
Craig Files	No	-	-	$\frac{N(N-1)}{2}$	u(B) and # of mismatches	Best bound hill-climbing (no backtr.)	Some
S.Grygiel inform.th.	No	-	-	various	good	best bound	Some
MULTIS	Yes	+	+	User Specified Parameter	u(B) for free set and bond set	1)Best Bound 2)Depth First 3)Breadth First 4)Ordered Search 5)Above variants	Best
Wei Wan	Limited	-	-	4 or less	Cube Calculus	No	Some

Figure 26: Comparison of Variable Partitioning Strategies

There are currently several ways to find the best partitions as shown in Figure 26. In the first row of Figure 26 the r-admissability test [63] of Luba has shown the ability to work well with don't cares, for multi-output, multi-valued simple functions. The r-admissability method has, however, the following properties:

- this method is totally useless for binary, single-output functions, since it always evaluates the multiplicity index of such functions to be not more than 2. This is a too rough evaluation of the lower bound.
- the method can be modified by not only counting the number of elements in the largerst block, but also, the avarage size of blocks, number of blocks of size 1, number of blocks of size 2, and other global measures that may be useful to create a more accurate evaluation of the multiplicity index.
- it has no "smart" searching.

The overall evaluation at this point is that the method should and can be improved, but that it handles don't cares better than the rest, especially for multi-otput, multi-valued functions.

The second row of Figure 26 shows the method of Craig Files from [15]. It does not work with don't cares, but uses smart searching to find solutions, and therefore gives very good results for small completely specified functions. It's failure to deal with don't cares keeps it from becoming the method used in MULTIS, where the existence of don't cares is the single most important factor of finding the minimized solution. The method has the following properties:

- its search idea - successive evaluation of single variables is good, but it can be further generalized and improved using the tree searching methods outlined in previous sections. Instead of Hill Climbing approach, any other strategy can be used.
- the most important is to find certain quality function for evaluating single variables, and subsets of variables. The approach used by Craig Files cannot be directly extended to incomplete functions, but perhaps some other methods of this type can be created. For instance, a total of lower bounds for Incompatibility Graphs created for bound and free sets. This can be done using the maximum cliques or large cliques of these graphs. The method of Haomin Wu has some similarities to this method, but handles don't cares better.

Next, the third row of Figure 26 shows the information theory method [41]. Such methods have been used for years in Machine Learning applications to build Decision Trees. They are also used to construct Decision Diagrams. This method uses no smart search, but can be adapted to don't cares. It can be also adapted for smart search, but some good heuristics must be found.

The fourth row of Figure 26 shows the Pair Weighting method of Wei Wan from TRADE [75]. This *The Variable Partitioning Pair Weighting* is a high quality heuristic method used in TRADE to find the best partition of input variables into the bound set and free set, where the bound set is the collection of variables forming the columns of a decompostion chart and the free set is the collection of variables forming the rows of a decomposition chart. The Pair Weighting method was developed by Wei Wan [76] to avoid the very time consuming testing of all possible sets of bound variables, which becomes impractical for large numbers of input variables in the input function, given the current limitations and speed of computer technology. The method has the following properties:

- can handle only a small percent of don't cares,
- it uses no smart searching.
- it is not perfectly clear why this method work and how well it works indeed.

The method from DEMAIN is difficult to compare and it is not in table. This is basically, in the worst case, a full search method. It sorts variables heuristically and next creates all subsets with k

elements, going next from k one variable up and one variable down, if possible, for instance, 5, 6, 4, 7, 3, 8, 2, 9, 10, 11, ... This is done for all subsets of k elements until multi-output Curtis decomposition is found. This strategy is basically blind, since the sorting of variables is done once and is not very useful to improve results. Concluding, all these methods need improvements, and all can be combined into one high-quality method.

Unfortunately, we were not able to find any papers dealing with comparison of variable partitioning approaches in the literature. Theoretical analysis seems also difficult. Therefore we plan to do the following:

1. Implement the methods as they are given in the literature and compare them in GUD. The Variable Partitioning Subroutine in GUD will have several algorithms implemented and will be able to use with any decomposition steps and encoding algorithms. We will be then able to compare various variable partitioning approaches for a broad spectrum of decomposition algorithms.

2. Implement a general tree-searching framework based on universal strategies, such as those from previous section. Apply these strategies to evaluate single variables, pairs of variables, triples of variables, etc.

3. Use different heuristics to evaluate variables and groups of variables. In particular, use the methods from the literature and their variants.

8.2 NEW METHODS FOR VARIABLE PARTITIONING

One approach to develop Variable Partitioning methods is to find good variable orderings, and next the best cuts in the sorted sequence of variables. These methods were applied in TRADE, and also in the BDD-based decomposer of Pedram et al. There is a guess that such an approach gives good results, so an experimental comparison to other strategies should be performed. Especially, comparison to the exact and full search strategies should be done.

There is also much room for improvement of the variable ordering methods used in TRADE, and the BDD-based method by Pedram et al. The ideal method would allow large amounts of don't cares, use a smart heuristic multistrategy tree searching, and should run fast. One of the possible approaches would be to concentrate on adding smart searching to the Pair Weighting Variable Partitioning method, and estimate its overall effect on the decomposer.

The method presented here improves on the TRADE algorithm. It consists in adding searching and backtracking to the Pair Weighting algorithm. This will add some time to calculations, but the time will be very minimal compared to "dumb search" from DEMAIN, and the increased performance of TRADE should far outweight any loss of extra time needed for backtracking.

There are two fundamental methods of tree searching to which heuristics can be added: breadth first searching, and depth first searching. The breadth-first searching, used in DEMAIN, always finds the shortest path to a goal node (assuming the shorter bound sets are better). But it takes up large amounts of time and memory in doing so. The strategy of DEMAIN has basically no information, and searches nearly totally blindly. It also assumes that the smaller bound sets are better than ones with more variables, which heuristic may be good for CLBs in Xilinx, but is not generally good for decompositions that attempt at minimizing the DFC. It is for that reason that the Depth-First searching is better suited for improving TRADE. The Depth-First searching is fast and goes deeply into the desired search space. The main problem is to find some good heuristics for ordering new nodes, another problem is telling the search subroutine when to stop going deeper and start backtracking, this is especially important in very deep trees.
Listed below is the Pseudo-code for Variable Partitioning from TRADE.

```
variable_partitioning() {
      repeat for ON-Y, ON-N, OFF-Y, and OFF-N Tables
     {
```

```
                create the Relationship Factor Table;
                sort the Relationship Factor Table in a decreasing order;
                bond_set := first pair in the queue;
                while (| bond_set | < maximum_bond_set_number)
                          bond_set = bond_set ∪ a pair in the next position;
        }
}
```

As we see, the procedure uses the *Relationship Factor Table* as a heuristic to find correlations of pairs and next larger groups of variables. Another observation is that it creates a sequence of increasing bound sets, but actually returns only the last one of them.

In the presented depth-first searching algorithm, the forming of the "best" partitions would be altered to include depth-first searching. An example algorithm for this type of depth-first searching is shown below.

```
depth_first_search( )
{
        OPEN := {variable_i, variable_j};
           /*initialize list with two random variables*/
        CLOSED := ∅;
        while OPEN ≠ ∅          /*nodes remain*/
              do
                remove leftmost node from OPEN, call it X;
                if for partition X there exists Curtis decomposition
                        then return ("success":  partition X) /*goal found*/
                else
                {
                  generate children of X. Each child is generated by adding one variable;
                  calculate (or evaluate heuristically) multiplicity index for each child.
                  put X on CLOSED;
                  eliminate children of X with small cost     /*loop check*/
                  /* these are children with high multiplicity index or r-admissibility
*/

                  put remaining children on the left end of OPEN; /* OPEN is a stack*/
                }
             return(failure)      /*no tree nodes left*/
}
```

This strategy has very little heuristics, it blindly searches the increasing sets of variables; its only heuristics is to eliminate children of X with small cost. One can use r-admissability or maximum clique to evaluate heuristically the multiplicity index instead of actually calculating it. Thus, the algorithm could be run several times. In the first run all bound sets could be found that would have the smallest value of rough evaluation of the multiplicity index - for instance using the r-admissability. In the second run the algorithm would go through these bound sets and would find all that would have the smallest value of a more accurate evaluation of the multiplicity index - for instance using the maximum clique of the Column Incompatibility Graph as a lower bound. Finally, in the last run, the real value of the multiplicity index would be calculated using the Graph Coloring or Set Covering algorithms.

In another variant, all these runs can be combined in a simple loop.

An algorithm of this type would take over in the code of TRADE right after the *ON-Y*, *ON-N*, *OFF-Y* and *OFF-N* triangle tables have been filled according to the methods discussed in [76] to form the final "best" partitions. The state space being searched would be the third item of the 3-tuple list from the previously generated *Relationship Factor Table*.

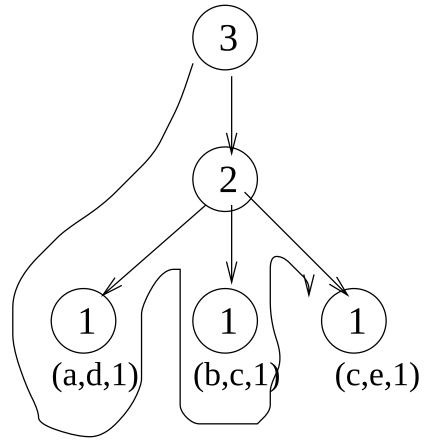
Another similar idea related to depth-first searching the state space of the triangle tables is just simple backtracking of all similar weighted values of the third item of the 3-tuple list to find the best bound set. The original method from TRADE has no way of checking similar weighted values of the triangle tables, so any addition of backtracking couldn't hurt, it could only help. A more descriptive form of depth-first backtracking algorithm is shown below.

```
backtracking( )
{
STATELIST := {Start};
NEWSTATELIST := {Start};
CURRENTSTATE := Start; /* initializations */
while NEWSTATELIST ≠ ∅ /*while there are states to be searched*/
   do
   {
   if search_end conditions are satisfied
    then return(best_bound_sets(STATELIST);
        /* when search terminated, evaluate all sets of variables found */
  if CURRENTSTATE has no children
    (excluding nodes already on STATELIST, NEWSTATELIST)
    then {
        while STATELIST not empty
          and CURRENTSTATE = first element of STATELIST
        {
        remove first element from STATELIST; /* backtrack */
        remove first element from NEWSTATELIST;
        CURRENTSTATE := first element of NEWSTATELIST;
        }
        add CURRENTSTATE to STATELIST
        }
 else {
     place children of CURRENTSTATE;
     STATELIST, or NEWSTATELIST) onto NEWSTATELIST;
     sort and filter NEWSTATELIST heuristically, according to method from section 2;
     CURRENTSTATE := first element in NEWSTATELIST;
     add CURRENTSTATE to STATELIST
      }
 }
  return FAIL; *state space has been exhausted*
}
```

With proper modification and incorporation, either of the two algorithms listed above could be used to better search similarly weighted values. For example, the triangle table in Figure 27a has three possibilities for it's last weighted item for 1 of (a,d,1), (b,c,1), or (c,e,1). Instead of just stopping at the first set (a,d,1), backtraking methods described above can be used to further check the state space and use them for the final comparison of best bond sets. This will take up exponentially larger amounts of memory, so code will also have to be written to allocate large amounts of space for running of the code, or the number of times backtracking is used must be limited. The inclusion of backtracking is shown in Fig. 27b for the traingle table next to it. First a goal of say (a,d,1) is found from say the ON-Y Table and compared with the three other ON and OFF tables, then using backtracking as shown in Fig. 27b another goal of (b,c,1) is found than compared again as before.

	b	c	d	e
a	0	2	1	0
	b	1	3	0
		c	0	1
			d	0

(a)



(b)

Figure 27: TRADE Approach: Triangular Table for pairs of variables, and Search with Backtracking

This whole process can be repeated a number of times for extremely large input functions, or it could be built into the code to limit the number of backtracking searches for large input functions.

The additional comparisons created from backtracking should result in "better" partitions for variable ordering, without having to test all possible decompostion charts. This in turn will improve TRADE (or any algorithm that uses TRADE's approach to variable partitioning), which will in turn improve MULTIS, that together with new methods, mborrows also all variable partitioning methods from TRADE.

The depth first searching method is just one of many possible ways of making variable ordering better in MULTIS. All other approaches to strategies presented before can be also used. The method developed by Craig Files will then become just one special case of the best bound search, but we will be able to search a larger space by using backtracking on top of his approach. What is most important is to combine the r-admissability approach of Luba, which gives good results for incompletely specified functions. This will be done in the next version of the MULTIS system.

QUESTIONS FOR SELF-EVALUATION

1. In sections 2-4 (and especially in section 3) we described methods to generate subsets of a set. How to create a variable partitioning strategy that will be analogous to these methods? For instance, generating the subsets of the set can be done: (1) starting from the smallest set of variables, (2) starting from the largest sets of variables, (3) starting from sets with certain number of elements, (4) starting from sets that are the closest to some subset of variables evaluated heuristically as good.

2. Compare the heuristics for the bound set selection from TRADE [75] and from ASH [15]. Create one more powerful heuristic that will include those from TRADE and ASH as its special cases.

9 UNCERTAINTY ANALYSIS APPROACH TO MULTI-VALUED FUNCTION VARIABLE ORDERING

Below we present a new Variable Partitioning method that has been adapted to multiple-valued logic decomposition from the area of general system analysis. The method is explained here only in its application to Binary Decision Diagrams ordering, which is the same as disjoint partitioning, because having a well-ordered diagram, the bound set is the set of variables above the min-cut of the diagram, and the (disjoint) free set is the set of variables below the min-cut. Having thus a minimal tree, the tree is factorized to a diagram by recursively combining the isomorphic nodes, and next the min-cut is found. This method can be applied only for partitioning with disjoint sets of free and bound variables, i.e. empty set of shared variables C.

As it will be shown in the forthcoming paper, an extension of this method can also be used for non-disjoint decompositions. Moreover, actual creation of the tree or diagram structure as a data structure is not necessary, and the selection of several good candidate sets of (possibly overlapping) free and bound variables can be done during the creation of successive cofactors of the input function. It will be also shown that the method can be applied to:

- single-output and multiple-output functions,
- binary, multiple-valued and fuzzy functions,
- completely and incompletely specified functions.
- general Ashenhurst/Curtis decomposition and special "gate-type" decompositions.

9.1 ELEMENTS OF SYSTEM ANALYSIS

Definition 9.1 *A system S is a set of elements (state variables) V, and relations R:*

$$S = \{V, R\} \tag{93}$$

where:

$$V = \{V_1, V_2, ..., V_n\}$$

The state of a system is a point in n-dimensional discrete space where n is equal to the number of state variables. If the set R is empty then each variable V_i may take any value from permissible for it set of values.

For instance, if a system can be described by two state variables A and B, taking n_a and n_b values:

$$A = \{a_1, a_2, \ldots, a_{n_a}\}$$

$$B = \{b_1, b_2, \ldots, b_{n_b}\}$$

and relation set is empty ($R = \emptyset$) then the system has $n_a \cdot n_b$ states. Relations between state variables introduce constraints to the system and may be defined as subsets of original (full) state space of the system. For a two-variable system:

$$R_{AB} \subset A \otimes B \tag{94}$$

Relation R_{AB} consists of a set of tuples (a_i, b_j) where each of possible values of a_i, b_j must show up at least once in the set. For example if $n_a = n_b = 2$ and each then one of the possible relations is:

$$R_{AB} = \{(a_1, b_1), (a_2, b_2)\}$$

Set $\{(a_1, b_1), (a_1, b_2)\}$ cannot be considered as a relation because a_2 didn't show up in the set.

The observations on the system may be represented by a contingency table, n-dimensional table which every cell contains the information about the number of cases a given value of the state of the system has been observed. Our two-variable system contingency table is a two-dimensional one and has $n_a \cdot n_b$ cells.

An important notion in system analysis is uncertainty or entropy of a system.

Definition 9.2 *An* uncertainty *or* entropy *of a system, for a two-variable case, is defined as follows (Shannon [70]):*

$$u(AB) = -\sum_i \sum_j p(a_i\ b_j) \log_2 p(a_i\ b_j) \tag{95}$$

where:
$p(a_i\ b_j)$ *is the probability of variable A taking the value of a_i and variable B taking the value of b_j*

and

$$\sum_i \sum_j p(a_i\ b_j) = 1$$

Conditional uncertainty of variables A and B is equal to:

$$u(A \mid B) = -\sum_j p(b_j) \sum_i p(a_i \mid b_j) \log_2 p(a_i \mid b_j) = u(AB) - u(B) \tag{96}$$

where:
$p(a_i|b_j)$ is the probability of variable A taking the value of a_i given variable B taking the value of b_j.
Those equations can be easily extended to any number of system state variables.

The uncertainty as defined above is a measure of the system variety, it is maximum if all the probabilities are equal (there is no privileged state of a system, there is maximum freedom of choice). The uncertainty is equal to zero if the system remains in one state only (one privileged state, no freedom of choice).

9.2 SYSTEM ANALYSIS AND MULTI-VALUED FUNCTION DECOMPOSITION

Let us consider an n-valued function f of k inputs and m outputs:

$$f : D \to U$$

$$D = \{0, 1 \ldots n-1\}^k = \{D_1, \ldots, D_k\}$$
$$U = \{0, 1 \ldots n-1\}^m = \{U_1, \ldots, U_m\}$$

Union of the function domain and the function range may be interpreted as a set of system state variables $V = D \cup U$, and the function mapping as a set of relations R_{DU}. Each element of the set R is a correspondence between an element of the function domain and element of the function range. Each state variable may take one out of n values ($n = 2$ for binary function). Probabilities in equations (95, 96) are defined using classical definition of probability:

$$p(a) = \frac{n_a}{n} \tag{97}$$

where:
n is the number of possible outcomes,

n_a is the number of outcomes that are favorable to the event a.

From the system's point of view, function decomposition consists in forming a partition P of the function domain D, where a partition is a collection of mutually exclusive subsets whose union equals D.

The partition is then used to separate the input variables into different functions to form a decomposed circuit.

Let us consider a decision tree (DT) representation of a multi-valued function. Decision tree complexity strongly depends on the function variable ordering. The problem of finding variable ordering which would minimize DT complexity is a key issue of many partitioning algorithms.

This problem may be addressed by means of calculating uncertainty of function taking certain output value given the value of an input variable(s) $u(Q \mid T)$, $Q = U_1 \dots U_m$, $T = D_i \dots D_j$. The value of uncertainty represents a degree of unimportance of a given input variable for the function output determination. The smaller variable uncertainty is, the bigger is the contribution of the variable in the determination of the function output. Such variables are to be placed close to the DT root.

To determine the optimal variable ordering however, $u(Q \mid T)$ must be calculated for every possible T. Calculation complexity of such procedure is $O(2^k)$, which means exponential increase of calculations. Calculation complexity may be reduced to $O(k^2)$ by applying the following heuristic procedure.

Algorithm 9.1 *1. Calculate $u(Q \mid D_i)$, $i = 1, 2, \dots k$.*

2. Given $u(Q \mid D_a)$ is the minimum value from the previous step, calculate $u(Q \mid D_a D_i)$, $i = 1, 2, \dots k$, $i \neq a$.

3. Given $u(Q \mid D_a D_b)$ is the minimum value from the previous step, calculate $u(Q \mid D_a D_b D_i)$, $i = 1, 2, \dots k$, $i \neq a$, $i \neq b$.

4. Continue until there is only one possible D_i to evaluate.

This procedure corresponds to building *DT* from the root (top). A similar procedure may be used to start building the tree from the leaf nodes (bottom). In this case however, the maximum, instead of minimum, uncertainty value is to be considered from step to step. Another approach is to consider both maximum and minimum uncertainty values, which corresponds to building the tree from both top and bottom.

Example 9.1

Given the following 3 input binary function [15]:

$$f(A, B, C) = 01010111$$

which is described by the truth table from Table 4 and Karnaugh Map from Figure 28. Find the decision tree of minimal complexity.

A	B	C	D
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 4: Table for Example 9.1

AB \ C	0	1
00	0	1
01	0	1
11	1	1
10	0	1

Figure 28: K-map for Example 9.1

The decision tree for ABC variable ordering is shown in Fig. 29.

The results of applying equation (96) to the function f is shown in Table 5.

Conditional uncertainties calculated for every single input variable A, B and C show how much uncertainty each of them introduces to the output variable determination. Since uncertainty due to the variable C is the smallest one, C is the best candidate for being placed in the root of the tree. Variables A and B when paired with C give the same uncertainty value which is smaller then uncertainty calculated for pair AB. Therefore the best variable order is CAB or CBA. The trees for each of these cases are shown in Fig. 30, 31.

As a comparison, the minimal tree obtained for the same function using ASH algorithm described in [15] is shown in Fig. 32.

Example 9.2

For the function from the previous example we will repeat the tree construction method, but the evaluation of variables will be done in a different way. For every variable we calculate the evaluation as follows.

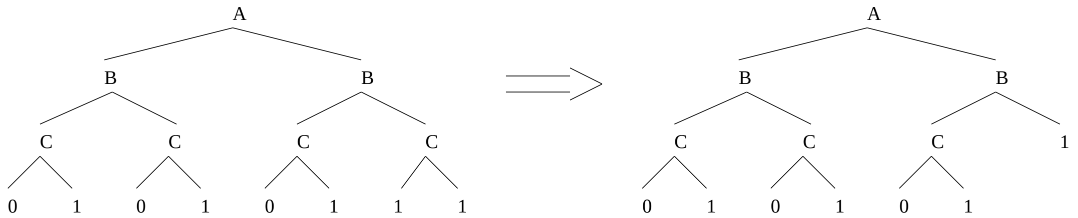


Figure 29: The decision tree for ABC variable ordering

Structure	Uncertainty
D	0.9544
$D \mid A$	0.9056
$D \mid B$	0.9056
$D \mid C$	0.4056
$D \mid AB$	0.7500
$D \mid AC$	0.2500
$D \mid BC$	0.2500
$D \mid ABCD$	0.0001

Table 5:

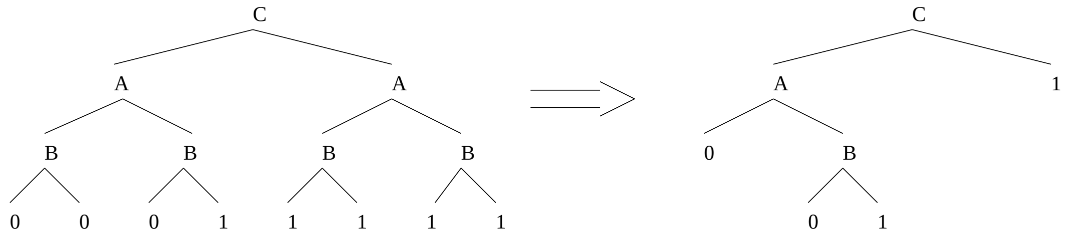


Figure 30: Tree for order CAB

$$EVAL(V_i) \ = \mid \ (ev(V_i)) - ev(\overline{V_i}) \ \mid = \mid \ (num\_1\_in(V_i) - num\_0\_in(V_i)) - (num\_1\_in(\overline{V_i}) - num\_0\_in(\overline{V_i})) \ \mid \quad (98)$$

The formula calculates the difference $ev(V_i)$ of the numbers of true and false minterms inside the positive literal V_i and the difference $ev(\overline{(V_i)})$ of the numbers of true and false minterms inside the negative literal $\overline{V_i}$. Next it calculates the absolute value of these two differences. Let us observe that $EVAL(V_i)$ is the measure of absolute separation of care minterms by variable V_i.

From Karnaugh Map we get:

$$EVAL(C) \ = \mid \ (4-0)-(1-3) \ \mid = \ 6$$

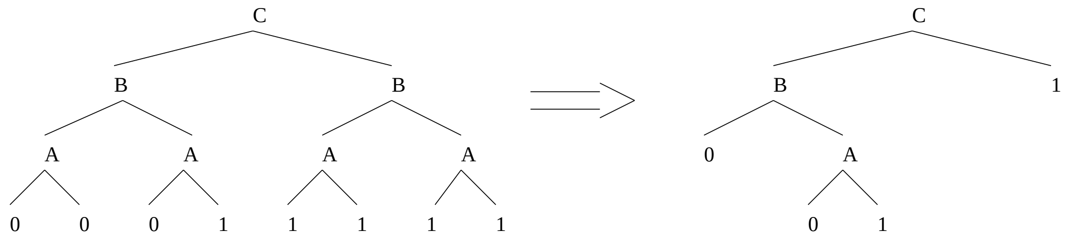


Figure 31: Tree for order CBA

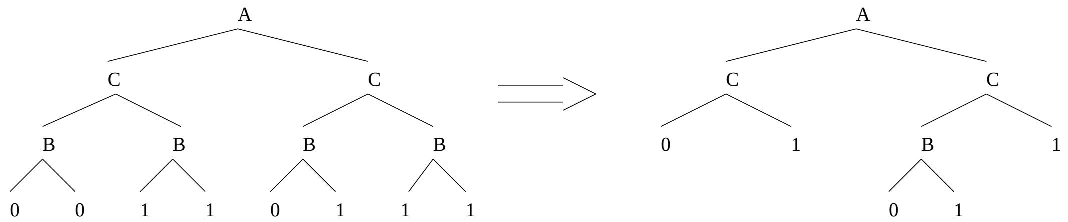


Figure 32: Tree for order found by ASH algorithm

$$EVAL(A) \;=\; |\;\; (3-0)-(2-2) \;\; | = \; 3$$

$$EVAL(B) \;=\; |\;\; (3-0)-(2-2) \;\; | = \; 3$$

Therefore, variable C is the selected as the root. Now the Karnaugh Map is separated to two Karnaugh Maps, one for C and one for $\overline{C}$, which means, we calculate cofactors for C and $\overline{C}$.

The number of maps on lower level is 2^r where r is the number of variables in the set of already selected variables. In our case there is one selected variable C. For each of the new maps that correspond to the set of selected variables (to all cofactors of this set), we calculate the total costs:

$$Total\_Cost \;=\; \sum_{cofactor_i \;\in\; Cofactors(set)} EVAL(cofactor_i) \tag{99}$$

for all remaining variables. In this case, the remaining variables are: A and B.

In this particular case the total costs are equal and lead to the same solutions as those found in the previous examples.

QUESTIONS FOR SELF-EVALUATION

1. What are the main concepts in the presented variable partitioning method?
2. How can the presented variable partitioning method be generalized to:
   - multiple-valued variables,
   - incompletely specified functions,
   - multi-output functions,
3. Find a more accurate cost function for the last method presented above.

10 APPLICATION OF R-ADMISSIBILITY IN VARIABLE PARTITIONING

We discussed already the use of r-admissibility in Column Minimization. Here we will discuss its application in variable partitioning. This time, we will do this for multiple-valued functions.

10.1 THE R-ADMISSIBILITY TEST

Direct application of Theorem 6.1 to find functions G and H would make the problem computationally intractable. To overcome this difficulty, we present conditions that allow us to check if, for a given set of input variables $A \subseteq X$, function F is decomposable so that component H has a given number of input variables, and variables in A directly feed H. These conditions are based on the concept of r-admissibility of a set of partitions.

Let P_i be a partition on M induced by some input variable X_i. The set of partitions $\{P_1, ..., P_k\}$ is called *r-admissible* with respect to partition P_F if there exists a set $\{P_{k+1}, ..., P_r\}$ of two-block partitions, such that:

$$P_1 \cdot \ ... \ P_k \cdot P_{k+1} \cdot ... \cdot P_r \leq P_F, \tag{100}$$

and there exists no set of r - k - 1 two-block partitions which meets this requirement.

The r-admissibility has the following interpretation. If a set of partitions $\{P_1, ..., P_k\}$ is r-admissible, then there exists a serial decomposition of F in which component H has r inputs: k primary inputs corresponding to input variables which induce $\{P_1, ..., P_k\}$ and r-k inputs being outputs of G. Thus, to find a decomposition of F in which component H has r inputs, we must find a set of input variables which induces an r-admissible set of input partitions.

To formulate a simple condition that can be used to check whether or not a given set of partitions is r-admissible, we introduce the concept of a quotient partition.

Let τ be a partition and σ an r-partition, such that $\tau \geq \sigma$. In a quotient partition of τ over σ, denoted $\tau \mid \sigma$, each block of τ is divided into a minimum number of elements being (not necessarily disjoint) blocks of σ.

For example, if

$$\sigma \ = \ (1;\ 2,6;\ 3,6;\ 5,7;\ 4,5)$$

$$\tau \ = \ (1,2,3,6;\ 4,5,7)$$

then we have quotient partition $\tau \mid \sigma$:

$$\tau \mid \sigma \ = \ ((1)(2,6)(3,6);\ (5,7)(4,5))$$

The following theorem can be applied to check whether or not a set of input partitions is r-admissible.

Theorem 10.1 *For partition σ and τ, such that $\sigma \leq \tau$, let $\sigma|\tau$ denote the quotient partition and $\eta(\tau|\sigma$ the number of elements in the largest block of $\tau|\sigma$. Let $\zeta(\tau|\sigma)$ denotes the smallest integer equal to or larger than $log_2\eta(\tau|\sigma)$, i.e. $\zeta(\tau|\sigma) = \lceil \log_2 \eta(\tau|\sigma) \rceil$. Let Π be the product of partitions $p_1, \ldots, p_k$ and $\Pi_F = \Pi \cdot P_k$. Then, $p_1, \ldots, p_k$ is $r-admissible$ in relation to P_F, with $r = k + \zeta(\Pi|\Pi_F)$.*

Example 10.1

	X_1	X_2	X_3	X_4	y_1	y_2	y_3
1	0	0	0	0	0	0	0
2	0	0	1	1	0	1	0
3	0	0	1	0	1	-	0
4	0	1	2	0	0	1	1
5	0	1	2	1	0	0	1
6	0	1	3	0	-	1	0
7	0	1	0	0	0	0	1
8	1	1	0	0	0	-	-
9	1	1	1	0	0	0	0
10	1	1	2	0	1	0	0
11	1	1	3	1	0	1	1
12	1	1	3	0	-	1	-
13	1	0	0	1	0	0	1
14	1	0	1	1	-	-	0
15	1	0	1	0	1	0	0

Figure 33: Table to Example 10.1

The following set of partitions on $M = \{1,...,15\}$ represents the function F of three two-valued variables, X_1, X_2, X_4, and one four-valued variable, X_3, as shown in Table from Fig. 33.

$$P_1 = (1,2,3,4,5,6,7;\ 8,9,10,11,12,13,14,15)$$

$$P_2 = (1,2,3,13,14,15;4,5,6,7,8,9,10,11,12)$$

$$P_3 = (1,7,8,13;\ 2,3,9,14,15;\ 4,5,10;\ 6,11,12)$$

$$P_4 = (1,3,4,6,7,8,9,10,12,15;\ 2,5,11,13,14)$$

$$P_F = (1,8,9,14;\ 2,6,8,12,14;\ 3,6,12,14;\ 3,10,14,15;\ 4,8,11,12;\ 5,7,8,13)$$

By examining the admissibility of $\{P_1\}$ we obtain:

$$P_1 \cdot P_F = (1;\ 8,9,14;\ 8,12,14;\ 5,7;\ 8,13;\ 2,6;\ 4;\ 8,11,12;\ 3,6;\ 10,14,15) \tag{101}$$

$$P_1 \mid P_1 \cdot P_F = ((1)(2,6)(3,6)(4)(5,7);\ (8,13)(8,9,14)(10,14,15)(8,11,12)). \tag{102}$$

Hence, $r = 1 + \mid log_2\ 5 \mid = 4$, i.e. $\{P_1\}$ is 4-admissible. Also, as:

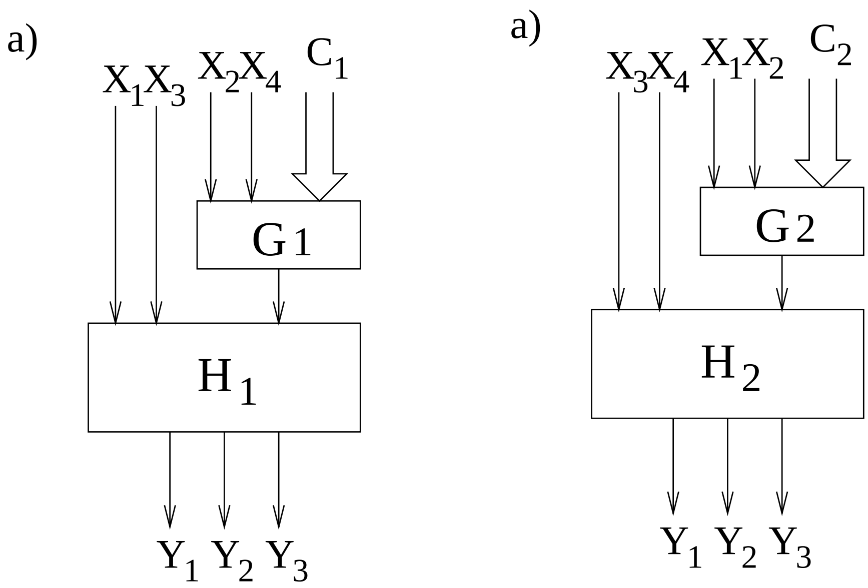


Figure 34: Two decompositions of function F with single-output functions G

$$P_1 \cdot P_3 \mid P_1 \cdot P_3 \cdot P_F \;=\; ((1)(7);\ (8,13);\ (2)(3);\ (9,14)(14,15);\ (4)(5);\ (10);\ (6);\ (11)(12)), \quad (103)$$

$\{P_1, P_3\}$ is 3-admissible. Similarly, we can show that $\{P_3, P_4\}$ is 3-admissible. Therefore, $F = H(X_1, X_3, G1(X_2, X_4, C_1))$ with $C_1 \subset \{X_1, X_3\}$ or $F = H(X_3, X_4, G_2(X_1, X_2, C_2))$ with $C_2 \subset \{X_3, X_4\}$, where both G_1 and G_2 are single-output functions - as shown in Fig. 34.

However, if we calculate the admissibility of $\{P_2, P_3\}$:

$$P_2 \cdot P_3 \;\mid\; P_2 \cdot P_3 \cdot P_F \;=\; ((1)(13);\ (7,8);\ (2,14)(3,14,15);\ (9);\ (4)(5)(10);\ (6,12);\ (11)(12)), \quad (104)$$

we will conclude that decomposition of F with the set $A = \{X_2, X_3\}$, where G is single-output function, does not exist. This is because $\mathrm{r}(P_2, P_3) = 4$, and the only possibility for decomposition of F is with the two-output function G.

Therefore we can conclude that taking into considerations the following subsets of input variables: $\{X_1, X_3\}$, $\{X_3, X_4\}$, $\{X_2, X_3\}$, the disjoint decomposition $F = H(A, (G(B))$, if only exists, includes the following subsets of variables: $\{X_1, X_3\}$ and $\{X_3, X_4\}$. In other words, the analysis of r-admissibility makes it possible to select some interesting subsets of variables for which the best disjoint decomposition can exist, however to be sure of this fact, the sufficient condition for existence of the decomposition (6.1) should be verified next. This is, however not done here, and we only use r-admissibility only for variable paritioning.

Sometimes it is more reasonable to unify the term k in r-admissility formula (i.e. $r = k + \mathrm{e}(\Pi \mid P_F)$, with respect to the total number of binary signals, b, which represent the set A:

$$b \;=\; \sum_{i:x_i \in A} \lceil log_2 \mid V_{x_i} \mid \rceil \qquad (105)$$

Then a *binary admissibility*, r_b, can be used to estimate the size of the set A. For the variables from example 10.1 we have:

$$
\begin{aligned}
r_b(X_1) &= 4 \\
r_b(X_1, X_3) &= 4 \\
r_b(X_2, X_3) &= 5
\end{aligned}
$$

and it is easier to comprehend, that the decomposition with the set $A = \{X_2, X_3\}$, i.e. $F = H(X_2, X_3, G(X_1, X_4))$ is not reasonable, since the number of equivalent binary input signals to block H matches the "binary equivalent", b_e, of the set $X = \{X_1, X_2, X_3, X_4\}$:

$$
b_e = \sum_{i=1}^{4} \mid log_2 \mid V_{X_i} \mid \mid = log_2 2 + log_2 2 + log_2 4 + log_2 2 = 5 \tag{106}
$$

11 USE OF SYMMETRY IN VARIABLE PARTITIONING

In this section we will present one more approach to variable partitioning that makes advantage of symmetry of Boolean functions, a simple but effective approach for disjoint decomposition of symmetric functions.

Although the recently proposed Boolean decomposition methods are intended for general multi-level network synthesis and are used for both symmetric and nonsymmetric functions, there is no special treatment for symmetric functions and the results for those functions are still quite inferior to the best designs.

For example, by analysing the quality of circuits designed by various design automation systems it was was found that the designs for totally symmetric functions had on average more than twice as many literals as the best designs, while the designs of nonsymmetric functions had on average only 20% more literals.

Taking advantage of symmetry, we will create a method particularly suitable for symmetric function synthesis, while some parts of the decomposition procedure will be simplified.

If the function is strongly unspecified then it is very likely that we are able to find many symmetric variables in it.

11.1 BASIC DEFINITIONS AND THEOREMS FOR SYMMETRY

Definition 11.1 *A Boolean function $f(X_1, \ldots, X_n)$ is symmetric in X_i and X_j if the interchange of X_i and X_j leaves the function identically the same.*

Lemma 11.1 *A Boolean function $f(X_1, \ldots, X_n)$ is symmetric X_i and X_j if and only if $f_{X_i\bar{X}_j} = f_{\bar{X}_iX_j}$.*

However, Lemma 11.1 is mostly suitable for the symmetry detection of completely specified functions. For the functions with don't cares, there is a theorem states as following:

Theorem 11.1 *$f = (ON, OFF, DC)$ is symmetric with respect to $A = \{X_i, X_j\}$ if and only if:*

$$(ON_{X_i\bar{X}_j} \sqcap OFF_{\bar{X}_iX_j} = \emptyset) \;\; and \;\; (ON_{\bar{X}_iX_j} \sqcap OFF_{X_i\bar{X}_j} \; = \; \emptyset \tag{107}$$

Detection of larger symmetry sets of completely specified functions is relatively easy, for it is an equivalence relation on the variables of symmetry.

Theorem 11.2 *$f = (ON, OFF, DC)$ is symmetric with respect to $\{X_i, X_j, X_k\}$ if and only if f is pairwise symmetric with respect to $\{X_i, X_j\}, \{X_j, X_k\}, \{X_i, X_k\}$.*

A maximum clique of variables that are pairwise symmetrical is the set of symmetric variables of a function.

11.2 APPROACH TO USE SYMMETRY

One of advantages of our approach is to use symmetry of functions to increase efficiency of partition-based and cofactor-based decomposition methods. It is well known that if f is totally symmetric then the output of f depends only on the weights of the input vectors. This can be carried over to multi-output functions: if F is symmetric with respect to λ, then every single-output function f_i in F depends on the λ-part weights of input vectors. This weight dependency suggests the disjoint decomposition.

$$F(X) \;=\; H(X - \lambda, G(\lambda)) = H(A, Z) \tag{108}$$

where $G = \{g_1, g_2, \ldots\}$ is the multiple-output function totally symmetric on λ.

One difficult problem in functional decomposition is to choose bound set. But for a symmetric function, our conjecture is that there exists at least one optimal disjoint decomposition using the symmetric variables as the bound set.

Following is the procedure.
Algorithm to find symmetric variables.

1. Using cofactors $F_{\{a,\bar{b}\}}$, $F_{\{\bar{a},b\}}$ find all pairs (a,b) of symmetric variables of function F.
2. Create a graph called *Symmetry Graph* with nodes corresponding to variables and edges corresponding to pairs of symmetric variables: if variables a and b are symmetric, the edge (a,b) exists in the graph.
3. Find maximum cliques in the graph.
4. For each maximum cliques create a bound set B_i.
5. For each bound set B_i, let $A_i = X - B_i$.
6. To the partitions of each set A_i, do the r-admissible test, calculate the value of r_i.
7. Determine the number of outputs of the recoder function $G(B_i)$, k, which equal to $r - |A_i|$.
8. Create Function $G(B_i)$.
9. Determine the image function H for this decomposition.
10. Select best B_i.

There are two methods to create function $G(B)$.

- The first method performs standard decomposition with set B of bound variables, calculates the multiplicity index and executes encoding.
- The second method knows the multiplicity index and which columns to combine from previous symmetry calculations.

  It can either find function G and next do encoding, or knowing the symmetry type directly find recoder functions by special synthesis method for symmetric functions, in case of smaller functions lookup table approach can be used.

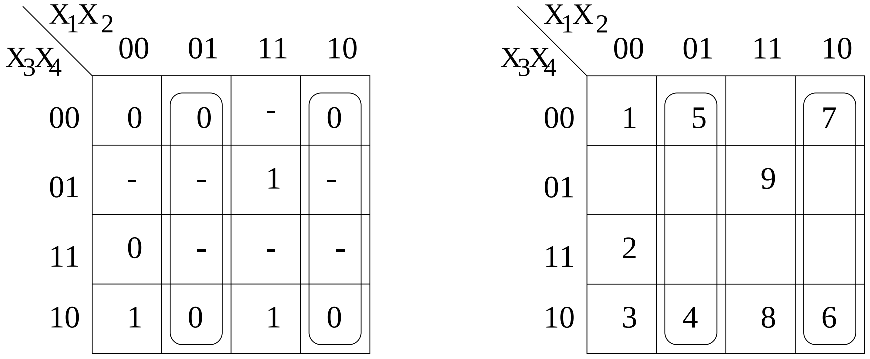
  In the second case, according to the value of k, certain recording functions are chosen.
  For instance, if $k = 2$, choose $G(B)$ as

  $$g_1(X_i, X_j) = X_i + X_j$$
  $$g_2(X_i, X_j) = X_i \cdot X_j$$

We can also benefit from the symmetry properties in a function if we need to perform encoding. As shown in the example.

	X_1	X_2	X_3	X_4	y
1	0	0	0	0	0
2	0	0	1	1	0
3	0	0	1	0	1
4	0	1	1	0	0
5	0	1	0	0	0
6	1	0	1	0	0
7	1	0	0	0	0
8	1	1	1	0	1
9	1	1	0	1	1

Table 6: Table for Example 11.1.



a) b)

Figure 35: Kmaps to Example 11.1: (a) K-map with cofactors $\bar{X}_1X_2$ and $X_1\bar{X}_2$ shown, (b) K-map numbers of rows from Table 7.11

Example 11.1 *An example of a binary function is given in Table 6 and also in Figure 35.*

As we can see, in this example, the variables X_1 and X_2 are symmetric variables, so we select these two variables as the bound set.

Do the partition on X_1 and X_2:

$$P(X_1) = \{\overline{1,2,3,4,5};\ \overline{6,7,8,9}\}$$

$$P(X_2) = \{\overline{1,2,3,6,7};\ \overline{4,5,8,9}\}$$

The partition of bound set $\{X_1, X_2\}$ is:

$$P(B) = P(X_1, X_2) = \{\overline{1,2,3};\ \overline{6,7};\ \overline{4,5};\ \overline{8,9}\}$$

The partition of free set $A = \{X_3, X_4\}$ is:

$$P(A) = \{\overline{1,5,7};\ \overline{2};\ \overline{3,4,6,8};\ \overline{9}\}$$

The output partition is:

$$P(F) = \{\overline{1,2,4,5,6,7};\ \overline{3,8,9}\}$$

Since, because of symmetry of variables from the bound set there must exist a disjoint decomposition, we can go directly to the r-admissibility test.

$$P(A) \cdot P(F) \;=\; \{\overline{1,5,7};\; \overline{3,8};\; \overline{9};\; \overline{2};\; \overline{4,6}\}$$

$$P(A)|P(A) \cdot P(F) \;=\; \{(1,5,7);\; (2);\; (9);\; (3,8)(4,6)\}$$

$$r = 2 + log_2 2 = 3$$

$$k = 1$$

Encoding:

We can benefit from symmetry in a function when performing the compatibility test. As in this example, if we group the two equivalent cofactor blocks of $P(B)$ together, we can get the corresponding partition Π_G, from which the image function H can be derived together with $P(A)$ and $P(F)$.

Group the blocks in $P(B)$ into two blocks. $(1,2,3,8,9)$, $(6,7,4,5)$

$$P(A) \cdot P(B) = (1;\; 2;\; 3,8,9;\; 7;\; 4;\; 6)$$

Assign $(3,8,9)$ as 1, we can get the image function H which is consistent with the original function y.

It is clear if we try another grouping: $(1,2,3,6,7)$; $(4,5,8,9)$

$$P(A) \cdot P(B) = (1,7;\; 2;\; 3,6;\; 5;\; 8,9)$$

which can not lead to a solution.

Depending on the type of symmetry, if there are two variables in the bound set the multiplicity index is either 2 or 3. Since multiplicity index of three requires two wires, this decomposition would reduce the number of columns from 4 to 3 and will not qualify for Curtis decomposition. In case of both symmetry and antysymmetry:

$$(ON_{X_iX_j} \sqcap OFF_{\bar{X}_i\bar{X}_j} = \emptyset) \;\; and \;\; (ON_{\bar{X}_i\bar{X}_j} \sqcap OFF_{X_iX_j}) \;=\; \emptyset \tag{109}$$

the multiplicity index equals 1 and there exist Curtis decomposition. The case of two variables is a special case. As we can see, in case of three variables in the bound set the multiplicity index is at most 4. Then 2 wires are needed on output of G, versus 3 on input, so the Curtis decomposition exists. In case of 4 variables in the bound set, in the worst case there the multiplicity index is 5, so 3 wires are needed and Curtis decomposition exists. In general, with m variables in bound set the multiplicity index in the worst case is $m+1$, so for $m > 2$ the Curtis decomposition exists.

Even if a set of nodes of a graph of variables with relation of symmetry as edges has no clique, a dense subset is a good choice for bound set in variable partitioning. By a dense subset we mean one which is nearly a clique and has only few edges missing to be a clique.

Example 11.2 *This second example will be very similar to the first one, we added only one row, number 10. This single care causes the columns $\bar{X}_1$ $\bar{X}_2$ and X_1 X_2 are no longer compatible. This is to make a point that the r-admissibility for single output binary functions has the component coming from G always at most 2, and is thus useless.*

An example of a binary function is given in Table 7.

As we can see, in this example, the variables X_1 and X_2 are symmetric variables, so we select these two variables as bound set.

Do the partition on X_1 and X_2:

	X_1	X_2	X_3	X_4	y
1	0	0	0	0	0
2	0	0	1	1	0
3	0	0	1	0	1
4	0	1	1	0	0
5	0	1	0	0	0
6	1	0	1	0	0
7	1	0	0	0	0
8	1	1	1	0	1
9	1	1	0	1	1
10	0	0	0	1	0

Table 7: Table of a binary function to Example 11.2.

$$P(X_1) = \{\overline{1,2,3,4,5,10};\ \overline{6,7,8,9}\}$$

$$P(X_2) = \{\overline{1,2,3,6,7,10};\ \overline{4,5,8,9}\}$$

The partition of bound set $\{X_1, X_2\}$ *is:*

$$P(B) = P(X_1, X_2) = \{\overline{1,2,3,10};\ \overline{6,7};\ \overline{4,5};\ \overline{8,9}\}$$

The partition of free set $A = \{X_3, X_4\}$ *is:*

$$P(A) = \{\overline{1,5,7};\ \overline{2};\ \overline{9,10};\ \overline{3,4,6,8}\}$$

The output partition is:

$$P(F) = \{\overline{1,2,4,5,6,7,10};\ \overline{3,8,9}\}$$

Since there must exists a disjoint decomposition, we can go directly to r-admissibility test without check the condition stated in Theorem 11.1.

$$P(A) \cdot P(F) = \{\overline{1,5,7};\ \overline{3,8};\ \overline{9};\ \overline{10};\ \overline{2};\ \overline{4,6}\}$$

$$P(A)|P(A) \cdot P(F) = \{(1,5,7);\ (2);\ (3,8)(4,6);\ (9);\ (10)\}$$

$$r = 2 + log_2 2 = 3$$

$$k = 1$$

As we see, in this case r-admissibility is of no help. On the other hand, it is easy to find a maximum clique of the Incompatibility Graph with 3 nodes, which is a good evaluation of the multiplicity index.

12 AUTOMATIC LEARNING OF SEARCH STRATEGIES

The ideas of Artificial Intelligence, being ther learning and neural networks, as well as methods mimicking humans while solving problems [52, 61] can be used to program a functional decomposer.

This report proposes such an approach. Since the problems of the constrained logic optimization class, such as NP-hard ones, will be always difficult to solve, one tries to find good heuristics that take into account the peculiarities of real life data in order to maximize the efficiency of execution. This can be done even at the cost of sacrificing the human's understanding why this or other technique works well. When a problem of developing a new algorithm is encountered, the logic theorist/program developer has, based on his experience and a large collection of similar problems, to find an appropriate tree-searching algorithm and the corresponding heuristics. A software system plus the outlined methodology should help the designer in this task.

The program should help the user to choose values for several parameters that all together define the searching strategy These parameters are: depth-first, ordered search, branch-and-bound, and other search heuristic routines.

The developer learns strategies and heuristics in the process, but he cannot test by hand too many examples. He cannot also perform very many experiments using the computer program, because making changes in source code and recompiling takes time. Therefore, we want to automate at least partially this computer-based experimentation process. This way, the program designer will be able to quickly evaluate the usefulness of his various ideas; and particular, how much each of them contributes to the success of the tree search.

Therefore, some of the parameters will also be used to select a combination of the *learning methods*. By the *learning methods* we will mean the methods that change some strategy parameters in order to improve the future behavior of the program, as the result of previous experience of it runs on categories of data.

Two learning methods will be proposed in this report. The first method learns criteria of selecting good operators by using a weighted evaluation function and learning its weight coefficients. The same approach is used for selecting nodes as well, where the coefficients of the evaluation function for solution tree nodes are learned. Another variant of this method observes the fact that a search strategy in a program can be described by a set of subroutine flags that are used to connect or disconnect the respective subroutines from the main tree-searching program. The subroutine with a flag taking values $0 \leq p \leq 1$ is connected with a probability of p. The probability coefficients of the flags can be learned in a similar way that the coefficients of the evaluation function are learned. Analogous approaches have been successfully applied in game playing programs and pattern recognizers [4, 5, 9, 21, 39, 42, 45, 67, 68, 69, 71, 81] but to the best of our knowledge, they have never been used to solve the class of combinatorial design problems considered by us here.

The second approach is an original one. It is used in Branch-And-Bound strategies and determines the Stopping Moment - such a moment of search in which backtracking can be terminated, because a better solution is very unlikely to be obtained in future. In several problems a strategy can be constructed that quickly leads to a minimal solution, however, it takes a very long time for the computer to prove that the solution is really the minimal one. Therefore, it is important to know the moment when the further backtracking can be terminated. The method uses a normalized shape diagram to determine the stopping method. The diagram plots the cost function values to the number of subsequently generated solutions. It is assumed, that these shapes are the same for combinatorial data of the same type.

The predictions of the stopping moment are calculated for some estimated probability of not loosing the optimal solution.

Section 12.1 presents the evaluation function learning method and illustrates with the *set covering problem* from section 4. Section 12.2 presents the backtracking stopping learning method, and section 12.3 illustrates this approach with a *linear assignment problem.* We are interested in this problem because it finds application to the Encoding Problem in decomposition, However, it has also several applications in: VLSI layout, logic synthesis, operations research, production scheduling, marriage

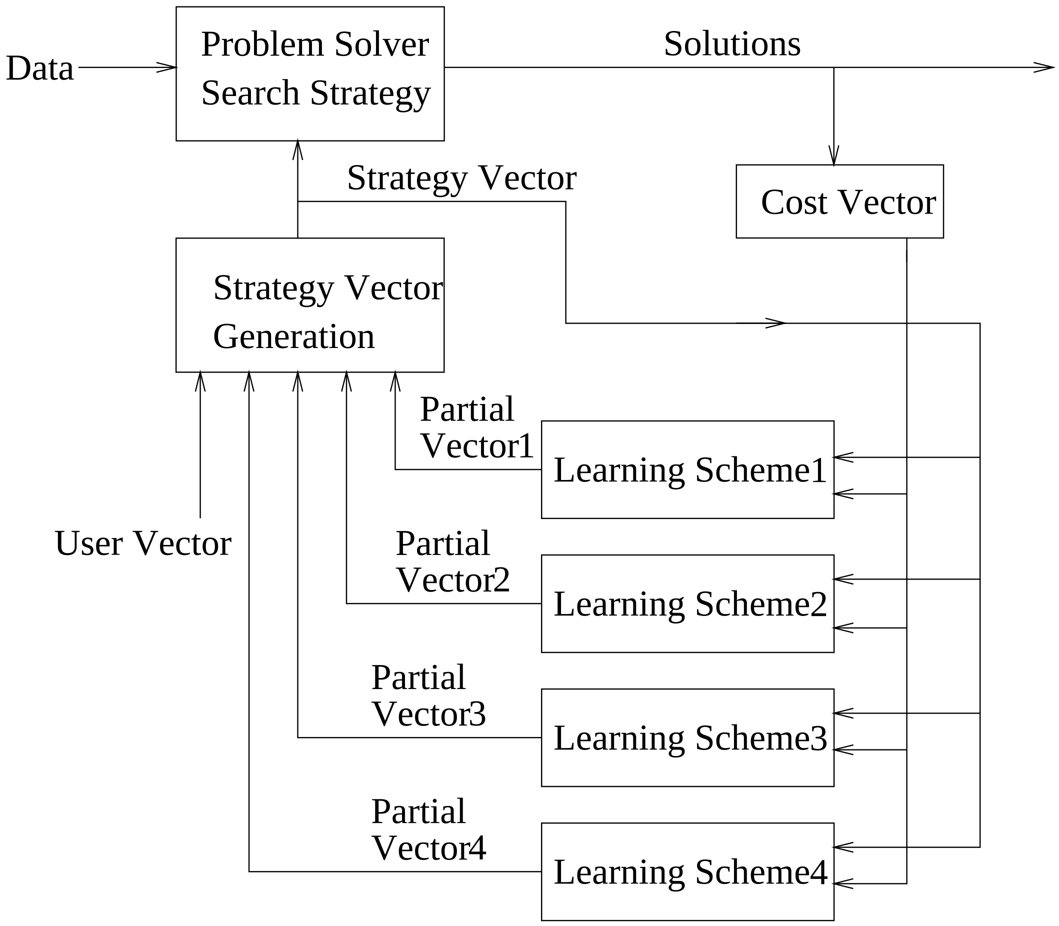


Figure 36: Collaboration of the Learning Methods with the Problem Solver

counseling, economics, communication networks, personnel administration, clustering, psychometrics, and statistical inference [1, 10, 17, 18, 23, 24, 27, 33, 34, 43, 65, 72, 79]. It is closely related to the traveling salesman problem [37]. Interestingly, the same "linear assignment" problem finds also applications to create general-purpose efficient heuristic learning schemes.

The conditions, relations, sorting functions, selecting functions, and strategy parameters, all together describe some "personalized" solution tree searching method and respective strategy. The possibility of dynamic modification of flags and coefficients used in them is *the basic principle of learning in program.* Fig. 36 presents schematic diagram of our approach. The Problem Solver in the top left corner is the tree searching one described in the previous sections. Its strategy is specified by a Strategy Vector - an ordered vector of numerical coefficients called **Strategy Parameters** (in general, real numbers, often zeros and ones or numbers from [0, 1]). Each value of the Vector describes one search strategy that can be realized by the Problem-Solver. Assuming r coordinates of the vector, and each of them with w values, one can realize w^r strategies. The sub-vectors of the System Vector (subsets of Strategy Parameters) are created by four Learning Schemes:

1. Learning Scheme # 1 - learning the Operators Evaluation Function.

2. Learning Scheme # 2 - learning the Tree Nodes Evaluation Function.

3. Learning Scheme # 3 - learning the moment when to stop the searching.

4. Learning Scheme # 4 - learning the probabilities of calling various subroutines and using parameters.

Strategy Defining Subroutines.

The user can set the initial values of all Strategy Parameters and some of them cannot be modified by learning. The Strategy Vector Generator checks the consistency of parameters. Certain subroutine flags are *conditionally disjoint.* For instance, there can be three subroutines, $SS1$, $SS2$, and $SS3$, all used to perform the sorting of operators. They use three different methods with different comparison criteria to be used in a node of the tree. Since only one of them can be used at given time, selecting $SS1$ (flag for $SS1$ enabled) will disable flags for $SS2$ and $SS3$. Let us observe, that the sum of probabilities for parameters $SS1$, $SS2$ and $SS3$ does not have to be equal one, since in any case, selection of one of the subroutine flags will disable the other ones from the group. In general, the *Exclusion Conditions* can be of the form: $Si \wedge Sj$ or $Sk \wedge Sl \wedge Sf \rightarrow \neg Su \wedge \neg Sv$ which means that if flags Si and Sj or flags Sk, Sl and Sf have been selected then the flags Su and Sv must be disabled. The four learning schemes have their local memories that store sets of pairs [*partial vector sample, its cost*], or other similar data. The costs are provided by the calculations of some solution parameters, such as, the values of the cost function, the total solution times, the solution cost improvement times, the sizes of the used memory. All these methods are "orthogonal" (independent) and can be applied separately or together. For strategies #1, 2, and 4, the improper learning, or the lack of the convergence cannot result in non-minimal solutions, but the lack of convergence will cause the program to take more time to obtain the optimal solution.

12.1 THE FIRST METHOD: LEARNING THE EVALUATION FUNCTION

Several studies lay the stress on the importance of selecting an appropriate Evaluation Functions (Quality Functions) while searching a tree [71, 46]. This section considers only Branch-And-Bound strategies, or mixed strategies, that combine the Branch-And-Bound and Ordered-Search Strategies. This principle is used in Learning Schemes # 1, 2, and 4.

Let the measure of the intelligence of the system be the number of nodes which have to be generated to find the optimal solution. It is easy to observe that this number depends on how quickly the program will encounter the appropriate branch (it means one that terminates with the optimal solution) and this will in turn depend on the good selection of evaluation functions for operators and nodes. For instance, in program the quality function for operators can be defined in a form

$$\sum_{i=0}^{M} c_i \ f_i \ = \ C^T \ \cdot \ F \tag{110}$$

where

$$F^T \ = \ (f_1, \ f_2, \ ..., f_n) \tag{111}$$

is a vector of *partial quality functions* and

$$C^T \ = \ (c_1, \ c_2, \ ...., c_n) \tag{112}$$

is a *vector of weight coefficients.* For each new problem, or even particular set of data for this problem, there is a question of how to select the vector C^T. The program can automatically learn C^T while solving the given problem or a set of examples. The learning problem can be formulated as follows. On the basis of the already searched part of the solution tree, and the information gathered from the previously solved examples of the same set, select such C^T that if a subsequent searches of the same tree were executed, the program would directly extend the branch leading to the best of the solutions selected until now, opening only those nodes, which are on the branch from the initial state of the tree.

Let us denote by N any node on this path, except the solution, and by NN one of its successors on this path. Let $O(N)$ be the operator transforming N to NN on the best branch from all the previously searched branches, and let $O_i(N)$ stands for all other operators in N.

To solve the formulated above problem we need to select such vector C^T that the following set of inequalities be satisfied:

$$(\forall \ \ N) \ [C^T \ \cdot \ F(O(N)) \ > \ C^T \ \cdot \ F(O_i(N))] \tag{113}$$

or, after transformation

$$(\exists \ \ N) \ [C^T(F(O(N)) \ - \ F(O_i(N))) \ > \ 0] \ . \tag{114}$$

Such set is usually inconsistent, so the problem of learning is reformulated to the problem of selecting C^T which satisfies the greatest possible number of inequalities from the above set.

For illustration, let us consider a two-dimensional case (Fig. 37a).

Let us assume that in some node there are 7 operators, for which the quality vectors can be represented as in Fig. 37a. Let us also assume that after finding the next solution the vector 4 is best. Using 114 we create then the difference vectors $(F(O) \ - \ F(O_i))$. We can observe that no vector C exists in Fig. 3.1b that would satisfy the set of equations 114.

Let us, therefore, reformulate the learning problem in the following manner: Select such a vector C for which as many as possible of the inequalities from the set of equations 114 are satisfied. As it can be noticed, this problem is equivalent to the problem of calculating such a *hyperplane* running across the coordinates system center, that as many as possible of the differences $(F(O(N)) \ - \ F(O_i(N))$ are located on one side of this hyperplane, and as few as possible on the other side. In the case of

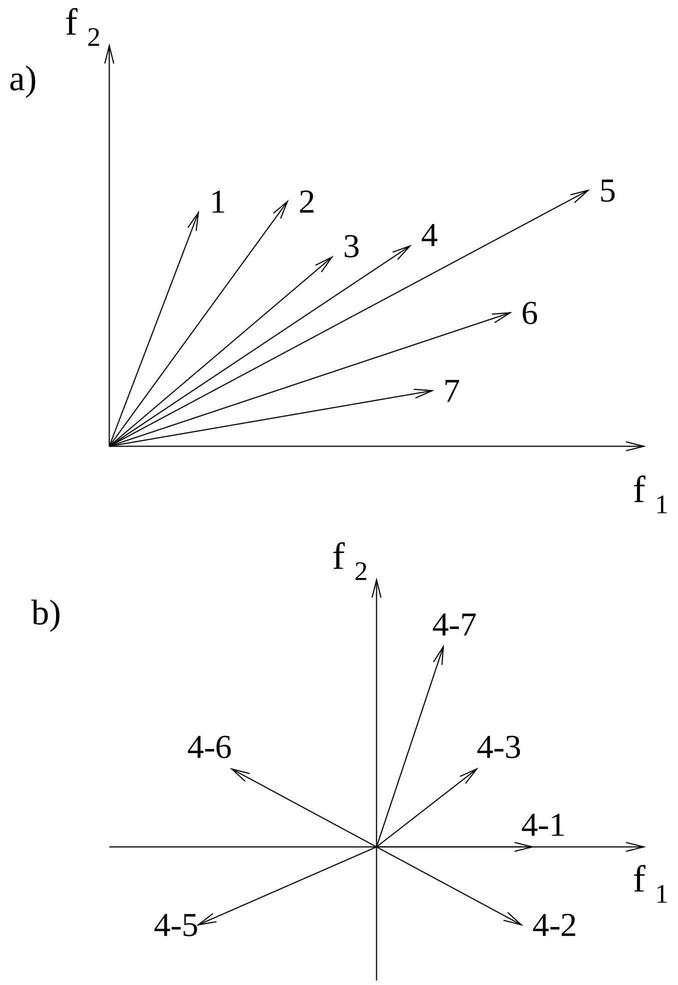


Figure 37: Two-Dimensional Case to Illustrate the Learning Method

n-dimensional space this problem is time consuming to solve, so we have implemented an approximate solution according to [71]. A hyperplane will be considered optimal, when

$$\sum_{i=1}^{n} C^T \cdot D_i = max, \tag{115}$$

where $\mid C \mid = 1$ and

$$D_i = \frac{F(O) - F(O_i)}{\mid F(O)) - F(O_i) \mid} \tag{116}$$

is a *normalized vector of differences*.
The normalized vector

$$C = \frac{\sum_{i=1}^{n} D_i}{\mid \sum_{i=1}^{n} D_i \mid} \tag{117}$$

is taken as the solution. Whenever new solution is found or when a backtrack occurs, the new inequalities are added to the learning procedures databases and the Learning Schemes #1, 2, and/or 3 are called in order to update the values of respective vectors C, using the above formulas.
An algorithm using the above principles is very fast and gives satisfactory results. Application of this type of learning increases the time (in the learning phase) by about 20% - 30%, but reduces the tree size by about 15%- 20%.

12.1.1 Experimental Results of the Set Covering Problem.

This problem was discussed previously. Now we will only discuss the learning aspect of it. It has been found by us in past on another applications, that the application of each of the equivalence conditions, dominance conditions, or indispensable conditions in the covering problem reduces the search space by about 2 to 3 times. The joint application of all the conditions brings about a reduction of approximately 50 to 200 times of the generated space. The influence of learning method on solution efficiency has been investigated. The computation time was increased in the learning phase by 20% to 30%. The vector of coefficients obtained in this learning was: $C = [$ -0.85, 0.25, -0.35]. This vector brings next 15% - 20% decrease in the generated solution space size, as compared with the initial vector: $C = [$ -1, 0, 0].

12.2 THE SECOND METHOD: LEARNING THE STOPPING MOMENT

The stopping process can be explained using an *"improvement curve"* in a diagram where the x-axis is a solution time expressed as a number of expanded nodes and the y-axis is a value of the cost function of the best of the solutions found until then.

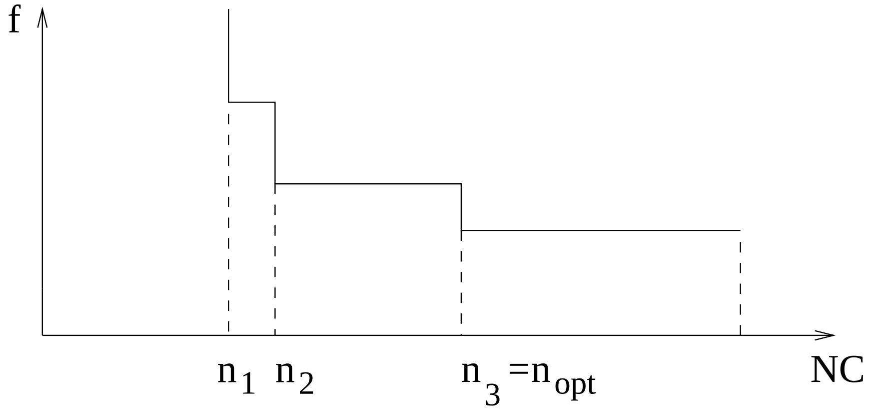


Figure 38: Improvement Curve for the Second Learning Method. Dependency of the cost function on the number of expanded nodes.

An example of an improvement curve is shown in Figure 38.
The solution process can be in this case terminated after expanding n_3 nodes, while the further expansion will not bring any cost function improvements. A question can be thus raised *"can one construct a system that would learn to predict the effects (or, specifically, lack of effects) of the further solution space expansion ?"*. A positive answer to this question would additionally improve the solution space method efficiency.

A new approach to this problem will be proposed below. It takes the following assumptions:

1. There exist combinatorial problems for which for all examples from some set of data the improvement curve shapes are similar.
2. There exists a *statistical dependency* (proportionality) between the number of nodes which have to be expanded in order to find successive, better solutions, i.e., $n_1,\ n_2,\ n_3,\ \ldots$. (See Fig. 38.).

In order to establish this proportionality, a sufficiently large number of examples must be tested. Also, it is necessary to introduce certain "normalization" that would make possible comparison of various improvement curves.

Let us introduce the concept of the *dimension of the problem with the respect to the i-th improvement,* or *i-th dimension* for short, as the number of nodes that must be expanded in order to obtain the i-th improvement of the cost function value. It will be denoted by n_i. By n_{ij} we will denote the number of nodes to obtain the i-th improvement in the j-th problem's sample. In Fig. 38 the dimension with respect to the first improvement is n_1, with respect to the second is n_2, and so on. For the given search strategy, the dimension sequence determines a unique *improvement curve* for this problem. Various problems can have sequences of dimensions of various lengths.

Solution of each example by a computer creates some number of pairs

$$(n_i,\ n_{opt}) \tag{118}$$

In the discussed example the pairs $(n_1,\ n_3)$, and $(n_2,\ n_3)$ are created. Each pair is reduced to a standard dimension N, by multiplying both its coordinates by a factor of N/n_i. The following pairs are obtained

$$(N,\ \frac{n_{opt}\ \cdot\ N}{n_i}), \tag{119}$$

where N is a fixed coefficient.
The second coordinate of each pair determines the standard number of nodes N_{oi} that results from the i-th dimension and which also has to be expanded in order to find the optimal solution. By disposing several improvement curves for many examples, one can calculate average standard numbers of nodes $\overline{N_{oi}}$ as weighted averages, while the weight of each result N_{oi} grows with the value of the i-th dimension n_i. It is based on the assumption that with an increase of example dimension the results obtained from the example are more reliable, i.e. they are characterized by a smaller expected value of standard deviation. Hence:

$$\overline{N_{oi}} = \frac{\sum_{j=1}^{k_i} N_{oij}\ \cdot\ n_{ij}}{\sum_{j=1}^{k_i} n_{ij}} = N\ \cdot\ \frac{\sum_{j=1}^{k_i} N_{j\,opt}}{\sum_{j=1}^{k_i} n_{ij}} \tag{120}$$

where k_i is a number of already solved examples that have the i-th dimension. The standard deviations of the obtained results are as follows:

$$\sigma_i = \sqrt[2]{\frac{\sum_{j=1}^{k_i} n_{ij} \cdot (N_{oij}\ -\ \overline{N_{oi}})^2}{\sum_{j=1}^{k_i} n_{ij}}} \tag{121}$$

In order to assure a sufficiently high probability that the space expansion would be not terminated before obtaining the optimal solution, a width of the half-interval of confidence with some coefficient is added to the calculated values of $\overline{N_{oi}}$:

$$N_{i\ max} = \overline{N_{oi}}\ +\ m\ \cdot\ \sigma_i \tag{122}$$

Coefficient $m = 3$ assures the 99,7% probability of assuming the normal probability density of the results. Therefore, $N_{i\ max}$ determines the maximal standard number of nodes - calculated with respect to the i-th dimension - which should be expanded in order to find the optimum solution.

The calculated values are applied as follows. Having found the first solution by expanding n_1 nodes, the maximum number of nodes n_{1_max} (already not the standard one) is calculated,

$$n_{1max} = \frac{N_{1max}\ \cdot\ n_1}{N} \tag{123}$$

If a better solutions is not found after expansion of n_{1max} nodes, the system stops. However, if after expansion of $n_2\ <\ n_{1max}$ nodes a better solution is found, then the value of n_{2max} is calculated according to the formula:

$$n_{2max} = \frac{(k_1\ \cdot\ n_{1max}\ +\ k_2\ \cdot\ \frac{N_{2max}\ \cdot\ n_2}{N})}{(k_1\ +\ k_2)} \tag{124}$$

where k_1 and k_2 determine number of examples which have been used to calculate the values of N_{1max} and N_{2max}. The system behavior after generating next solutions is based on the same principles.

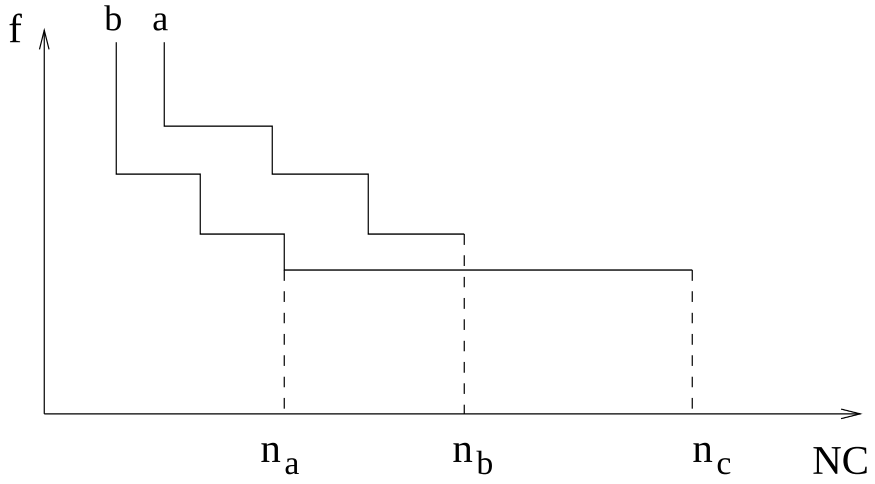


Figure 39: The Effect of Concurrent Application of Both Learning Methods, (a) without learning the quality function coefficients, (b) with learning the quality function coefficients.

The method described above bypasses several difficulties related to the normalization of several individual properties of the problems, such as different numbers of improvement curve steps, or various values of quality function decrements. It can be observed that this method collaborates well with the one to learn quality functions coefficients, and gives better results when the other method produces better results.

Fig. 39 presents schematically the effect that can be obtained by concurrent application of the both learning methods. Without learning, n_c nodes should be expanded. By applying the learning method from this section one would need n_b nodes while applying additionally the method from section 12.3 one would need only n_a nodes.

12.3 EXAMPLE OF APPLICATION OF THE SECOND METHOD: THE LINEAR ASSIGNMENT PROBLEM

This problem, similarly to the previous one, has *several* CAD applications, most importantly here, for encoding.

Problem Formulation.

Given is n machines and n workers and the productivity of the worker i on machine j is denoted by w_{ij}. The assignment of machines to workers is sought that maximizes the total productivity of all workers. The assignment matrix $W_{n \times n} = [w_{ij}]$ is given from which n elements must be selected in such a way that any two of them are taken from a different row and column and the sum of the elements is maximum. Since program looks for a minimum, the problem is reformulated as below.

Each object has n properties. The value of property x_i determines the column number from which the element from row i has been selected. In order to simplify the program the elements of the matrix are selected in the following order: first an element from row one is chosen, next an element from row two, and so on. In this case the operator is a number specifying only the second coordinate of the selected element w_{ij}, and the first coordinate is specified by the depth of the node in the tree. Secondly, the list $AS(N)$ of nodes specifying the state of the object in node N becomes in this case unnecessary, while it would contain the set of non-selected rows, and this set is already specified by the depth of the node in the tree. For instance, in such description, the solution

$< 3, 1, 2 >$

for a 3×3 problem means that elements w_{13}, w_{21}, w_{32} have been selected from matrix W. The description of the **initial state** is as follows:

$QS(0) = \emptyset,$

$GS(0) =$ set of numbers of all columns.

Operation of **operator** $O(N,\ r_i)$ can be described as follows:

[
$QS(NN) = QS(N)\ \cup\ \{\ r_i\ \},$
$GS(NN) = GS(N)\ \setminus\ \{\ r_i\ \}\ .$
]

The **solution condition** is $GS(NN)\ =\ \emptyset.$

The specification of the **quality function** values for operators is in this case obvious - they are the negated values of the respective elements w_{ij} of the matrix. The value of cost function for states is equal to the sum of negated costs of operators on the path from the initial state to the given state. **A heuristic quality function for nodes,** that specifies the minimal cost of the path from the given node to the solution is also useful. It is defined as follows:

$$\hat{h}\ =\ \sum_i w_{i,\ min}$$

where summing is extended over the non-selected rows, and $w_{i,\ min}$ denotes the least element of the i-th row among the elements from the non-selected columns.

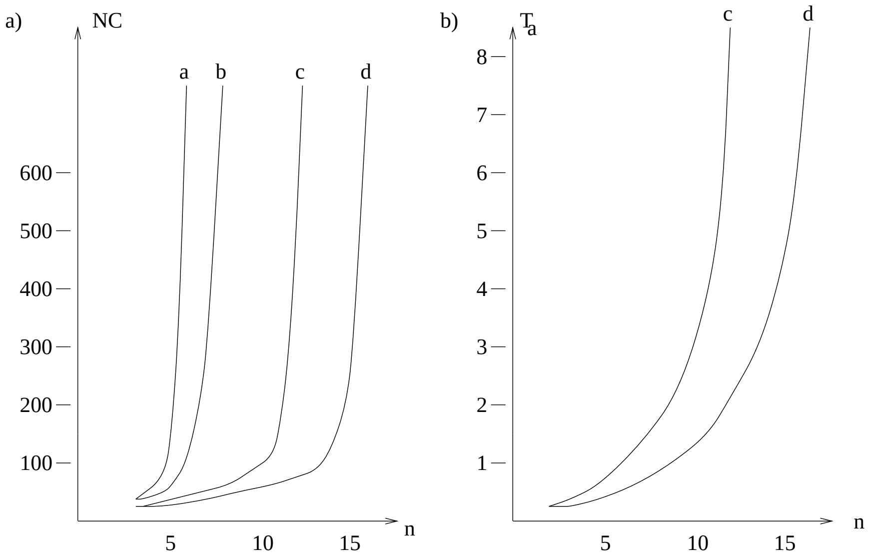


Figure 40: The dependence of (a) number of nodes and (b) the processing time on the problem size for strategies: a - Breadth First, b - Depth First, c - Branch-and-Bound, d - Ordered Search

Experimental Results.

The influence of the search strategy on the solution efficiency has been investigated. Fig. 40. presents the statistical dependency of the solution time and the number of nodes expanded with respect to the problem's dimension, for various strategies. The Ordered-Search strategy has been the most efficient one for this problem. It is due to the usage of a very accurate heuristic function which permitted for cutting branches at small depth of the tree.

This problem is one for which the Stopping Learning Method gives good results, because it is characterized by a improvement curve with sufficiently large number of steps. (For instance about 4-10 steps for Branch-And-Bound strategy and $n\ <\ 14$). Additionally, it was verified that about 30% of the solution tree was expanded "redundantly" after finding the first optimal solution. Using the above method permitted to decrease this part of tree by about 40%. For instance, when the entire space generated by the system included, on average, 600 nodes and the optimal solution was found after 420 nodes, by using this method it was sufficient to expand on 530 nodes.

13 TESTING OF TRADE

The improved TRADE program was tested on the "circuit design benchmarks" from MCNC and ISCAS. The shell scripts required for this testing will be presented. The testing was performed assuming various parameters as the number of inputs to the block. The tests were done on both the old and the new version of TRADE. This may explain some differences in tables.

13.1 BASIC STAGES OF TRADE

The term TRADE stands for TRAnsformation and DEcomposition. Cube Calculus is used in TRADE for all operations. TRADE reads the input in Espresso (.type fr) format and outputs in Bliff format with the input variables of each node less than or equal to five. In new version there is no any constraint of this type.

The basic steps in TRADE are as follows:

- Read in the input file written in Espresso ".type fr" format.
- Select an output. Perform partition analysis(the number of bound set variables is fixed to five) to obtain the best partitions and the Additional Partitions. Additional Partitions are the partitions whose bound sets consist of the variables that are the input varaibles of some CLBs in the CLB Pool, and these variables of the current decomposition. The CLB Pool is a list of all CLBs previously generated by the program.
- Execute decompositions using the best and Additional Partitions. Encode the bound set and try to use as many CLBs from the CLB Pool as possible. From the program it is observed that the Graph Coloring Technique is applied at this stage to get a quasi-optimum don't care assignment. Select a partition which results in the smallest Cover Ratio. The Cover Ratio of the number of newly created CLBs over the difference of the input variables before and after decomposition. At this stage, if the functions happens to be non-decomposable, then perform the local transformation to make it decomposable.
- Repeat the last two steps for the blocks left until all decomposed blocks are with five or less inputs.
- Repeat the last three steps for all Modifying Functions which were created by local transformations.
- Repeat the last four steps for all outputs.
- Merge all possible nodes into the FG mode CLBs.

```
TRADE()
{
    read_input_file(); /*read in input file*/
    for ( i = 0; i < number_of_primary_outputs; i++)
    { /*loop for all output functions*/
      do
      {
            variable partitioning(); /*find best partition*/
            create_incompatibility_graph();/*create inompatibility graph */
            graph_coloring();/* quasi-optimum don't care assignment */
            if ( decomposable ≠ true )
                  local_transfomation(); /*make decomposable */
            bound_set_encoding();/* encode bound set */
```

```
            CLB reusing(); /*use CLBs in CLB Pool*/
        } while (function(s) from local transformation == true)
    }
    CLB MERGing(); /*merge nodes into FG mode CLBs */
    output_result(); /*output results*/
```

The syntax of the command line for running the TRADE is:

trade [-r] [-m] <u>bound-input</u> <u>bound -output</u> <u>file-name</u>

bound-input is the allowed maximal number of input variables to each subblock. The range of its value is from 2 to 5.

bound-output is the allowed maximal number of output variables from each subblock. It must be no more than the value of bound-input.

-r By default, trade processes each output of a multi-output function according to the complexity of each output. It starts to process from the simple output to the complex output. -r option reverses this order, making trade to start to process from the complex output to the simple output.

-m By default, trade merges the possible nodes into FG mode CLBs of Xilinx architecture. -m option prohibits the merging process, making it possible to apply trade to non Xilinx architecture applications.

13.2 TESTING TRADE WITH BENCHMARKS

In the testing with benchmarks the following parameters generated by TRADE were analysed.

The Number of CLBs generated by the TRADE program for various benchmarks is compared for all possible acceptable combinations of options. There are 10 combinations of acceptable bound-input and bound-output with bound-input ranging from 2 to 5 and bound-output taking a value that is no more than the value of bound-input.

Also, the 'r' and 'm' option can be combined to form 4 possible combinations.

xx No options are given.

rx 'r' option alone active.

xm 'm' option alone active.

rm both 'r' and 'm' option active.

The execution time is computed in seconds for all the possible combinations.

The DFC measures is compared for the multi-output examples of the MCNC benchmarks. The DFC measure is sum of DFC measures of blocks. $DFC\ of\ block = 2^k * M$, where k is the number of inputs to block and M is the number of outputs of the block.

The TRADE program is tested for various MCNC benchmarks for multiple output functions. The program is tested for all acceptable combinations of options for the each example. A comparison is made for the number of CLBs, number of levels, the execution time and the DFC for all various MCNC multi-output examples.

The TRADE program transforms the input logic into a network of minimum number of sub-blocks.

13.2.1 Testing on special functions

The first test uses a short test file of 10 inputs and 3 outputs. The program run time, total CLB's and levels, and cardinality are listed. All tests were done assuming no CLB merging.

As shown above the lowest number of CLB's and/or levels is not necessarily the best solution according to DFC scores. In this case the 4 input one output blocks (16) with a 4 level structure is the cheapest according to DFC by far, with a score of 174 compared to the next lowest score of 264. Thus depending on what the user's design constraints, the "best" solution could be based on program run time, number of CLB's, number of levels, or lowest cardinality.

The program will work with the number of bound outputs specified at greater than or equal to the number of bound inputs. However, in the case where the two are equal, the program cannot always find a solution (see 3 in's and 3 outs, and 2 in's and 2 outs above).

In comparison with the above test file, whose number of inputs and outputs are small, a larger file standard, duke2.w was also tested with the following results:

Again, the "best" solution depends on the desired design parameters. Duke2.w has 22 inputs and 29 outputs.

We wanted to find out which affected TRADE more, higher number of inputs, or outputs. Thus the next set of tests use two test files; temp.w has 6 inputs and 30 outputs and temp2.w has 30 inputs and 6 outputs.

Though the sample is not large enough for statistical analysis, it can be easily seen that the number of CLB's in a case with fewer inputs the number of CLB's is much greater than in a case with more inputs. In this example the number of CLB's in the case with fewer inputs is four time that of the case with more inputs. This will also drive up cardinality on the cases with fewer inputs, indicating that finding a good minimal solution is more difficult in this case.

13.2.2 Number of ONSET in INPUT

time trade -m 5 4 filename

The output was composed of 40% onset, 40% offset, and 20% DC's. The input was made up of the onset percentages shown in the table with the balance split evenly between Offset anf DC's. There were 20 input variables, 5 output variables and 20 cubes per file. Times are in seconds and are an average of two runs.

File	Onset	CLB's	Levels	CPU	Kernal	Elapsed	%CPU
t1.w	10%	257	15	61.1u	1.7s	1:16	82%
t2.w	30%	37	5	7.3u	.2s	:11	68%
t3.w	50%	22	3	4.4u	.2s	:08	58%
t4.w	80%	23	4	1.8u	.3s	:04	57%
t5.w	100%	5	1	.1u	.1s	:01	16%

13.2.3 Number of OUTPUT VARIABLES

time trade -m 5 4 filename

The input was composed of 30% onset, 35% offset and 35% DC's. The output was composed of 40% onset, 40% offset, and 20% DC's. There were 20 input variables and 30 cubes per file. The number of output variables varied. Times are in seconds and are an average of two runs.

File	Output	CLB's	Levels	CPU	Kernal	Elapsed	%CPU
t7.w	5	63	6	13.4u	.3s	:16	83%
t14.w	1	11	5	2.2u	0s	:02	81%
t15.w	10	144	9	32.9u	.4s	:39	84%
t17.w	20	453	12	167.3u	1.2s	3:06	90%

13.2.4 Number of INPUT VARIABLES

time trade -m 5 4 filename

The input was composed of 35% onset, 35% offset and 30% DC's. The output was composed of 40% onset, 40% offset, and 20% DC's. There were 5 output variables and 30 cubes per file. The number of input variables varied. Times are in seconds and are an average of two runs.

File	Inputs	CLB's	Levels	CPU	Kernal	Elapsed	%CPU
t20.w	5	5	1	0.0u	.2s	:01	18%
t21.w	10	91	7	13.5u	3s	:17	78%
t7.w	20	63	6	13.4u	.3s	:16	85%
t22.w	30	59	5	6.7u	.1s	:09	80%

13.2.5 Number of DON'T CARES in INPUT

time trade -m 5 4 filename

The output was composed of 40% onset, 40% offset, and 20% DC's. The input was made up of the DC percentages shown in the table with the balance split evenly between Offset anf Onset. There were 20 input variables, 5 output variables and 30 cubes per file. Times are in seconds and are an average of two runs.

File	DC's	CLB's	Levels	CPU	Kernal	Elapsed	%CPU
t30.w	30%	32	4	9.0u	.1s	:11	82%
t31.w	10%	18	3	3.6u	1s	:05	68%
t35.w	40%	171	11	43.7u	.8s	:50	87%

13.2.6 Number of CUBES in INPUT FILE

time trade -m 5 4 filename

The input was composed of 30% Onset, 35% Offset, and 35% DC's. The output was composed of 40% onset, 40% offset, and 20% DC's. There were 20 input variables and 5 output variables. Times are in seconds and are an average of two runs.

File	Cube's	CLB's	Levels	CPU	Kernal	Elapsed	%CPU
t6.w	10	16	5	2.7u	.3s	:12	29%
t2.w	20	37	5	7.3u	4s	:15	50%
t7.w	30	63	6	13.2u	.5s	:23	61%
t8.w	50	125	9	35.3u	.6s	:51	71%
t10.w	60	201	12	75.5u	.9s	1:25	89%
t11.w	65	385	13	327.2u	1.9s	5:40	97%
t12.w	70	289	12	165.1u	1.2s	2:53	96%

13.2.7 Number of CLBs

The Number of CLBs generated by the TRADE program for various benchmarks is compared for all possible acceptable combinations of options. There are 10 combinations of acceptable bound-input and bound-output with bound-input ranging from 2 to 5 and bound-output taking a value that is no more than the value of bound-input.

Also, the 'r' and 'm' option can be combined to form 4 possible combinations.

xx No options are given.

rx 'r' option alone active.

xm 'm' option alone active.

rm both 'r' and 'm' option active.

Figures 1 to 4 give the values for the number of CLBs for the different combinations.

13.2.8 Number of levels

The Number of levels is computed for all possible combinations for bound-input, bound-output, r and m. Figure 5. gives the values for the number of levels. If the values are represented by slashes, then the values are different for the four different combinations of xx/xm/rx/rm given in sequence. The values represented in bold represent the same value for all the 4 different combinations.

13.2.9 Execution time

The execution time is computed in seconds for all the possible combinations. Figures 6 to 9 give the execution time on running the TRADE program for various MCNC benchmarks on a Sparc 10.

13.2.10 Comparison of DFC measure

Figures 10 - 13 gives the DFC measures for the various combination of options.

#	Examples	Var		Bound input - Bound output									
		i/p	o/p	2-1	3-1	3-2	4-1	4-2	4-3	5-1	5-2	5-3	5-4
1	5xp1	7	10	63	30	25	14	12	13	11	11	11	11
2	9sym	9	1	83	27	13	11	8	8	11*	7	6*	6*
3	Z5xp1	7	10	84	31	26	14	12	13	11	11	11	11
4	adr2	4	3	5	3	3	2	2	2	2	2	2	2
5	b12	15	9	55	25	20	17	17	17	20	18	18	18
6	bw	5	28	140	55	48	31	31	31	27	27	27	27
7	cc	21	20	35	22	20	23	23	23	24	23	23	23
8	clip	9	5	258	68	69	45	34	36	60	49	37	38
9	duke2	22	29	853	243	355	157	179	234	170	224	248	253
10	f51m	8	8	48	25	21	13	12	8	12	11	9	9
11	inc	7	9	118	40	37	26	21	22	18	19	19	19
12	misex2	25	18	69	41	41	31	30	30	33	31	31	31
13	rd53	5	3	15	5	4	5	5	5	3*	3*	3*	3*
14	rd73	7	3	41	15	11	10	7	8	5	5	5	5
15	rd84	8	4	67	24	16	14	9	10	14	10	11	11
16	sao2	10	4	271	75	70	37	30	25	37	33	30	30
17	squar5	5	8	31	14	12	8	8	8	7	7	7	7
18	t481	16	1	11	25	12	6	6	6	13	9*	9*	9*
19	vg2	25	8	521	147	256	83	89	87	126	123	78	83
20	xor5	5	1	2	2*	2*	2*	2*	1*	1*	1*	1*	1*

Figure 41: Number of CLBs for no option.

#	Examples	Var		Bound input - Bound output									
		i/p	o/p	2-1	3-1	3-2	4-1	4-2	4-3	5-1	5-2	5-3	5-4
1	5xp1	7	10	124	58	56	25	24	22	14	14	14	14
2	9sym	9	1	165	46	27	15	12	12	11*	8	6*	6*
3	Z5xp1	7	10	163	60	56	25	24	22	14	14	14	14
4	adr2	4	3	9	6	3	3	3	3	3	3	3	3
5	b12	15	9	103	51	44	35	29	29	29	20	20	20
6	bw	5	28	271	109	104	64	64	64	28	28	28	28
7	cc	21	20	64	37	34	37	36	36	29	29	29	29
8	clip	9	5	500	135	150	83	72	65	64	59	39	38
9	duke2	22	29	1629	487	822	295	383	496	217	261	277	276
10	f51m	8	8	93	49	48	25	23	14	13	14	12	12
11	inc	7	9	228	83	87	49	43	43	22	22	22	22
12	misex2	25	18	137	81	97	60	57	57	45	39	39	39
13	rd53	5	3	30	10	8	8	8	3	3*	3*	3*	3*
14	rd73	7	3	80	25	17	17	12	12	6	6	6	6
15	rd84	8	4	131	48	31	26	16	18	15	11	12	12
16	sao2	10	4	530	158	158	60	52	44	44	38	31	31
17	squar5	5	8	57	27	28	16	16	16	8	8	8	8
18	t481	16	1	21	53	35	7	8	8	15	9*	9*	9*
19	vg2	25	8	989	297	576	148	178	166	150	138	82	86
20	xor5	5	1	4	2*	2*	2*	2*	2*	1*	1*	1*	1*

Figure 42: Number of CLBs for 'm' option alone

#	Examples	Var		Bound input - Bound output									
		i/p	o/p	2-1	3-1	3-2	4-1	4-2	4-3	5-1	5-2	5-3	5-4
1	5xp1	7	10	64	31	26	16	16	14	13	13	13	13
2	9sym	9	1	83	27	13	11	8	8	11*	7	6*	6*
3	Z5xp1	7	10	82	30	26	16	16	14	14	14	14	14
4	adr2	4	3	5	3	3	2	2	2	2	2	2	2
5	b12	15	9	47	25	25	20	21	20	20	18	18	18
6	bw	5	28	134	53	42	29	32	32	27	27	27	27
7	cc	21	20	31	22	20	23	24	23	23	23	23	23
8	clip	9	5	193	83	75	45	51	42	45	38	28	29
9	duke2	22	29	711	241	329	150	170	262	159	243	250	289
10	f51m	8	8	66	25	23	18	15	13	14	13	13	13
11	inc	7	9	95	42	39	24	20	23	18	18	19	19
12	misex2	25	18	62	37	39	31	30	30	33	32	32	32
13	rd53	5	3	16	5	4	4	4	4	3*	3*	3*	3*
14	rd73	7	3	46	15	4	10	7	5	5	5	5	5
15	rd84	8	4	70	22	8	15	8	8	13	10	8	8
16	sao2	10	4	254	70	56	42	26	32	44	32	27	27
17	squar5	5	8	27	16	13	8	8	8	7	7	7	7
18	t481	16	1	11	25	12	6	6	6	13	9*	9*	9*
19	vg2	25	8	555	157	151	98	118	196	95	132	134	146
20	xor5	5	1	2	2*	2*	2*	2*	2*	1*	1*	1*	1*

Figure 43: Number of CLBs for ‘r’ option alone.

#	Examples	Var		Bound input - Bound output									
		i/p	o/p	2-1	3-1	3-2	4-1	4-2	4-3	5-1	5-2	5-3	5-4
1	5xp1	7	10	125	61	57	28	30	25	16	16	16	16
2	9sym	9	1	165	46	27	15	12	12	11*	8	6*	6*
3	Z5xp1	7	10	159	59	57	28	30	24	17	17	17	17
4	adr2	4	3	9	5	5	3	3	3	3	3	3	3
5	b12	15	9	89	54	53	40	35	34	34	20	20	20
6	bw	5	28	259	105	96	62	65	65	28	28	28	28
7	cc	21	20	56	37	34	37	37	37	29	29	29	29
8	clip	9	5	380	154	162	86	88	72	49	45	29	29
9	duke2	22	29	1356	485	755	275	375	539	197	274	278	310
10	f51m	8	8	124	50	49	30	27	24	17	16	15	15
11	inc	7	9	183	88	89	46	47	44	22	22	22	22
12	misex2	25	18	124	74	84	59	60	60	45	40	40	40
13	rd53	5	3	31	10	8	7	7	7	3*	3*	3*	3*
14	rd73	7	3	91	27	8	16	12	8	6	6	6	6
15	rd84	8	4	140	44	16	27	15	15	14	11	9	9
16	sao2	10	4	501	141	131	68	47	53	52	37	27	27
17	squar5	5	8	52	31	29	16	16	16	8	8	8	8
18	t481	16	1	21	53	35	7	8	8	15	9*	9*	9*
19	vg2	25	8	1054	324	338	167	240	373	111	147	139	148
20	xor5	5	1	4	2*	2*	2*	2*	2*	1*	1*	1*	1*

Figure 44: Number of CLBs for both 'r' and 'm' option

#	Examples	Var		Bound input - Bound output									
		i/p	o/p	2 - 1	3 - 1	3 - 2	4 - 1	4 - 2	4 - 3	5 - 1	5 - 2	5 - 3	5 - 4
1	5xp1	7	10	8/8/9/9	5	6	3	4	3	2	2	2	2
2	9sym	9	1	13	7	7	3	4	5	3	4	3	3
3	Z5xp1	7	10	9	5	6/6/7/7	3	3/4/4/4	4/3/3/3	2	2	2	2
4	adr2	4	3	4	2	2	1	1	1	1	1	1	1
5	b12	15	9	8/8/7/7	5	5	4	3/3/4/4	3/3/4/3	3	3	3	3
6	bw	5	28	7	3	4	2	2	2	1	1	1	1
7	cc	21	20	5	4	4	3	3	3	2	2	2	2
8	clip	9	5	11/11/12/12	7	9	5	5/5/6/6	6/6/7/7	4	4	4	4
9	duke2	22	29	16/16/15/15	9	16	7	9	15/15/14/14	5/5/6/6	7/7/6/6	9/9/8/8	10
10	f51m	8	8	8/8/9/9	6/6/5/5	6/6/8/8	4	4	3	3	3	3	3
11	inc	7	9	9/9/8/8	5	7	4	4	5	2	2	2	2
12	misex2	25	18	10/10/9/9	6/6/7/7	8/8/7/7	5	5	5	4	4	4	4
13	rd53	5	3	6	3	3	2	2	2	1	1	1	1
14	rd73	7	3	8	4	4	3	4	3	2	2	2	2
15	rd84	8	4	10	6	6	3	3/3/4/4	4	3	3	3	3
16	sao2	10	4	13/13/12/12	7	9/9/10/10	5	5	5/5/8/8	4/4/5/5	4	4/4/3/3	4/4/3/3
17	squar5	5	8	6	3	4	2	2	2	1/1/2/1	1	1	1
18	t481	16	1	7	9	8	4	4	4	4	4	4	4
19	vg2	25	8	16	9/9/10/11	16/16/13/13	7	9/9/10/10	10/10/14/14	7	8/8/7/7	8/8/9/9	8/8/10/10
20	xor5	5	1	4	2	2	2	2	2	1	1	1	1

Figure 45: Number of Levels for xx/xm/rx/rm options in sequence. The value in bold represent the same levels for all combinations.

#	Examples	Var		Bound input - Bound output									
		i/p	o/p	2-1	3-1	3-2	4-1	4-2	4-3	5-1	5-2	5-3	5-4
1	5xp1	7	10	23.8	12	8.2	7.6	7.2	6.6	5.6	3.6	3.6	4.4
2	9sym	9	1	39.8	14.9	6.5	1.3	6	3.5	12.3	4.4	2.6	2.6
3	Z5xp1	7	10	26.4	11.7	8.8	7.6	9.2	7.7	6.6	3.7	3.9	4.1
4	adr2	4	3	2	1.2	1.4	1.4	1.5	1.9	1.6	0.4	0.4	0.6
5	b12	15	9	17.8	10.9	7.5	8.1	7.1	6.1	9.6	5.9	5.5	6
6	bw	5	28	51.2	26.1	14.7	12.8	12.6	13.2	11.3	4.6	4.0	6.2
7	cc	21	20	14.9	10.3	7.9	9.7	11.2	8.9	11.1	5.8	6.2	5.8
8	clip	9	5	84.6	36	24	44.1	17.3	10.5	63.4	20.4	10.3	8.2
9	duke2	22	29	381	143	202	164	136	574	308	145	122	124
10	f51m	8	8	16.6	9.3	10.5	5.4	6.1	5.5	4.8	2.8	2.2	2.3
11	inc	7	9	35.7	13.3	18.3	11.6	9.4	7.1	6.1	4.3	4.3	4.5
12	misex2	25	18	27.4	15.2	14.1	15.4	11.2	14.6	27.1	11.7	11.5	11.7
13	rd53	5	3	4.9	2.1	1.9	1.9	1.6	1.9	1.5	1.5	0.4	0.5
14	rd73	7	3	13.9	6.5	2.6	6.1	3.8	3.3	3.1	2.4	2.2	2.8
15	rd84	8	4	33.7	26.6	7.5	22.9	7.8	6.3	22.3	11.2	4.9	5
16	sao2	10	4	89.1	34.2	25.7	27.9	15	90.5	34.8	13.4	8.5	8.2
17	squar5	5	8	11.9	7.2	5.1	3.7	3.5	3.9	3.3	1.2	1.1	1.1
18	t481	16	1	130	482	169	78.6	78.6	78.8	778	87.1	86.9	87.4
19	vg2	25	8	843	169	562	131	85.9	77.2	14867	99.5	76.6	72.6
20	xor5	5	1	70	0.7	0.7	0.6	0.7	0.9	1.1	0.3	0.2	0.2

Figure 46: Execution time for no option in seconds

#	Examples	Var		Bound input - Bound output									
		i/p	o/p	2-1	3-1	3-2	4-1	4-2	4-3	5-1	5-2	5-3	5-4
1	5xp1	7	10	8.9	6.4	4.4	5.1	4.5	3.1	4.9	3.3	3.3	3.2
2	9sym	9	1	23.9	11.6	4.9	11.9	5.1	2.8	11.3	4.4	2.4	2.4
3	Z5xp1	7	10	10.4	5.7	3.4	5.3	4.9	4.9	5	3.3	3.3	3.2
4	adr2	4	3	0.6	0.5	0.5	1	1	0.7	1	0.4	0.4	0.4
5	b12	15	9	5.7	5.9	3	6.6	4.6	4.5	7.6	5.3	5.2	5.2
6	bw	5	28	16.6	13.4	6.1	9.8	8.3	10.2	7.3	3.6	3.5	3.6
7	cc	21	20	5.7	4.8	3.4	7.4	6	7.7	8.6	6.1	5.5	5.5
8	clip	9	5	45.8	23.7	15.2	38	14.5	9.7	56.5	19.5	9.8	7.8
9	duke2	22	29	210	108	140	140	111	150	289	139	255	119
10	f51m	8	8	6.3	5.6	3.1	4.5	3.6	2.9	3.9	2.4	2.1	2.1
11	inc	7	9	12..3	10.1	5	10.7	5.1	4.9	6.4	4.1	4.1	4.2
12	misex2	25	18	10.4	9.8	6.9	14.6	10.7	10.9	14.9	11.2	11.2	11.2
13	rd53	5	3	1.6	0.9	0.6	1.2	1	1.3	0.7	0.4	0.4	0.4
14	rd73	7	3	6.7	3.7	1.6	4	2.5	2.3	2.6	2.2	2.1	2.2
15	rd84	8	4	21.9	23.5	6.1	21.6	6.2	5.9	21.3	11	5	5
16	sao2	10	4	62.6	22.3	13.3	19.7	9.7	9.5	32.2	13	8.2	8.4
17	squar5	5	8	8.5	2.9	1.9	2.3	1.8	2.7	1.9	1.1	0.9	1.2
18	t481	16	1	128	1976	166	78.6	78.1	78.4	357	407	87	87
19	vg2	25	8	292	138	334	124	75.6	67.6	1350	98.6	75.4	71.8
20	xor5	5	1	1.1	0.2	0.5	0.4	0.6	0.4	0.2	0.2	0.2	0.2

Figure 47: Execution time for ‘m’ option alone in seconds.

#	Examples	Var		Bound input - Bound output									
		i/p	o/p	2-1	3-1	3-2	4-1	4-2	4-3	5-1	5-2	5-3	5-4
1	5xp1	7	10	10.3	11.7	6.1	6.8	4.5	4.2	5.3	5.6	6.5	6.1
2	9sym	9	1	29.5	14.2	7.1	12.2	4.9	3.1	12.1	5..3	3.9	2.9
3	Z5xp1	7	10	18.7	12.3	5	6.5	4.1	4	6.1	7.4	5.8	6.1
4	adr2	4	3	1.8	0.7	0.6	1	0.5	1	0.8	0.9	3.6	1.1
5	b12	15	9	10.8	7.2	4.3	8.9	3.8	5.4	8	6.8	7.5	6.7
6	bw	5	28	34.2	19.8	11.1	12.8	9.1	11.6	7	6.2	11.6	9.1
7	cc	21	20	10.2	8	7	9.6	5.3	7.5	10.4	11.5	11.9	10.2
8	clip	9	5	51.9	33	21.2	48.2	20.1	14.1	49.7	24.9	12.2	8.7
9	duke2	22	29	498	149	152	163	106	161	263	164	151	190
10	f51m	8	8	11.4	9.9	6.7	6.2	5.5	3.5	6	4.8	5.1	4.2
11	inc	7	9	18	15.7	8.3	9.3	7.3	7.3	5.4	5.6	5.4	5.1
12	misex2	25	18	12.2	17.6	12.1	14.8	10.2	11.7	20.2	13.4	15.9	15.1
13	rd53	5	3	1.9	2.5	1.3	1.3	1.4	1.2	1.1	14	1.7	1.1
14	rd73	7	3	9.8	6	2.3	5.1	2.6	2.1	3.3	2.8	3.2	3
15	rd84	8	4	27.7	24.4	6.3	22.2	15.6	7.6	21.7	12.4	5.9	6.3
16	sao2	10	4	45.8	27.2	19.4	26.9	12.7	8	45.1	16.1	9	1.9
17	squar5	5	8	3.7	8.4	3.5	6.2	3.6	2.4	11.2	1.9	2	2
18	t481	16	1	128	481	168	78.9	78.9	78.6	368	93.2	87.5	87.5
19	vg2	25	8	475	236	181	122	142	168	285	209	280	149
20	xor5	5	1	0.6	0.3	0.4	0.4	0.7	0.5	0.4	0.4	0.7	0.4

Figure 48: Execution time for 'r' option alone in seconds.

#	Examples	Var		Bound input - Bound output									
		i/p	o/p	2-1	3-1	3-2	4-1	4-2	4-3	5-1	5-2	5-3	5-4
1	5xp1	7	10	21.5	8.9	5.7	7.5	13.6	3.3	5.5	5.4	5	4.6
2	9sym	9	1	38	36.1	12	13	4.9	7.4	12.3	4.9	2.5	2.7
3	Z5xp1	7	10	21.7	13.7	5.6	7.4	5.2	4.5	5.5	5.7	4.9	8
4	adr2	4	3	1.2	0.9	0.7	1.7	0.9	1.1	1.1	1.2	0.9	0.7
5	b12	15	9	13.7	12.3	11.6	11.6	4.2	5.3	7.6	7.3	6.5	6.3
6	bw	5	28	37.7	27.7	13.2	10.2	11.9	15.6	7.1	21.7	6.7	7
7	cc	21	20	14.2	7.7	6.1	5.9	11.3	10.8	8.3	8.6	8.1	7.9
8	clip	9	5	53.8	44.1	29.5	50.6	19.9	15.8	41.2	31.1	9.6	8.2
9	duke2	22	29	259	178	200	207	155	172	281	207	144	171
10	f51m	8	8	17.6	15.2	10.1	9.7	4.5	4.9	4.2	4.2	3.6	3.1
11	inc	7	9	24.7	11.9	10.9	13.9	6.3	6.0	5	5	4.6	5.7
12	misex2	25	18	19.8	24.3	18.8	12.8	10.8	13.8	15.3	23.8	15.1	13.8
13	rd53	5	3	4.6	1.6	2.2	0.9	14.6	1.3	0.8	0.9	0.8	0.6
14	rd73	7	3	11.6	5.5	5.5	5	2.6	2.7	2.6	3.1	2.5	2.6
15	rd84	8	4	34.5	25.4	15.7	24.6	6.5	6.1	24.8	21.1	5.8	5.5
16	sao2	10	4	66.3	27.1	31.3	34.4	11.5	8.9	38.7	18.9	9.6	8.6
17	squar5	5	8	6.1	19.7	5.8	2.3	19.7	3.5	2.5	2.4	1.8	1.7
18	t481	16	1	129	489	347	81.9	92.8	80.5	357	94.5	87.3	87.2
19	vg2	25	8	550	300	232	152	155	185	310	167	289	141
20	xor5	5	1	0.9	0.7	0.5	0.4	0.5	0.4	0.6	0.3	0.3	0.2

Figure 49: Execution time for both ‘r’ and ‘m’ option in seconds.

#	Examples	Var		Bound input - Bound output									
		i/p	o/p	2-1	3-1	3-2	4-1	4-2	4-3	5-1	5-2	5-3	5-4
1	5xp1	7	10	506	428	378	342	290	310	286	286	286	286
2	9sym	9	1	662	366	206	242	178	194	354	206	178	178
3	Z5xp1	7	10	642	444	386	342	290	310	286	286	286	286
4	adr2	4	3	42	46	46	42	42	42	42	42	42	42
5	b12	15	9	398	372	324	402	392	392	562	502	502	502
6	bw	5	28	1082	820	780	816	816	816	920	920	920	920
7	cc	21	20	284	236	224	448	452	452	476	476	476	476
8	clip	9	5	1879	970	1060	1102	920	914	1812	1348	1102	1166
9	duke2	22	29	6071	3339	5407	3700	4694	6333	4802	6340	7216	7496
10	f51m	8	8	376	352	328	334	282	174	294	266	214	214
11	inc	7	9	876	578	602	584	562	594	542	558	558	558
12	misex2	25	18	576	578	686	690	740	740	940	920	920	920
13	rd53	5	3	126	78	62	106	106	106	102	102	102	102
14	rd73	7	3	324	198	142	270	166	190	142	142	142	142
15	rd84	8	4	528	352	244	336	224	264	400	284	336	336
16	sao2	10	4	2030	1066	1132	904	730	688	996	928	880	880
17	squar5	5	8	230	214	204	192	196	196	204	204	204	204
18	t481	16	1	80	368	210	102	106	106	310	214	214	214
19	vg2	25	8	3598	2110	3681	1898	2336	2276	3628	3524	2238	2458
20	xor5	5	1	18	18	18	22	22	22	34	34	34	34

Figure 50: DFC measures for no option

#	Examples	Var		Bound input - Bound output									
		i/p	o/p	2-1	3-1	3-2	4-1	4-2	4-3	5-1	5-2	5-3	5-4
1	5xp1	7	10	506	428	378	342	290	310	286	286	286	286
2	9sym	9	1	662	366	206	242	178	194	354	206	178	178
3	Z5xp1	7	10	642	444	386	342	290	310	286	286	286	286
4	adr2	4	3	42	46	46	42	42	42	42	42	42	42
5	b12	15	9	398	372	324	402	392	392	562	502	502	502
6	bw	5	28	1082	820	780	816	816	920	920	920	920	920
7	cc	21	20	284	236	224	448	452	452	476	476	476	476
8	clip	9	5	1879	970	1060	1102	920	914	1812	1348	1102	1166
9	duke2	22	29	6071	3339	5407	3700	4694	6333	4802	6340	7216	7496
10	f51m	8	8	376	352	328	334	282	174	294	266	214	214
11	inc	7	9	876	578	602	584	562	594	542	558	558	558
12	misex2	25	18	576	578	686	690	740	740	940	920	920	920
13	rd53	5	3	126	78	62	106	106	106	102	102	102	102
14	rd73	7	3	324	198	142	270	166	190	142	142	142	142
15	rd84	8	4	528	352	244	336	224	400	284	336	336	336
16	sao2	10	4	2030	1066	1132	904	730	688	996	928	880	880
17	squar5	5	8	230	214	204	192	196	204	204	204	204	204
18	t481	16	1	80	368	210	102	106	106	310	214	214	214
19	vg2	25	8	3598	2110	3681	1898	2336	2276	3628	3524	2238	2458
20	xor5	5	1	18	18	18	22	22	22	34	34	34	34

Figure 51: DFC measures for 'm' option alone

#	Examples	Var		Bound input - Bound output									
		i/p	o/p	2-1	3-1	3-2	4-1	4-2	4-3	5-1	5-2	5-3	5-4
1	5xp1	7	10	502	456	390	390	382	320	320	350	350	350
2	9sym	9	1	662	366	206	242	178	194	354	206	178	178
3	Z5xp1	7	10	636	428	394	390	382	328	320	320	320	320
4	adr2	4	3	42	38	38	42	42	42	42	42	42	42
5	b12	15	9	358	376	350	484	506	486	554	530	530	530
6	bw	5	28	1042	796	730	764	840	840	920	920	920	920
7	cc	21	20	252	236	224	448	468	468	476	476	476	476
8	clip	9	5	1488	1152	1138	1170	1196	972	1260	1052	760	800
9	duke2	22	29	5045	3365	4867	3396	4520	7208	4476	6956	7138	8496
10	f51m	8	8	474	352	318	380	324	338	374	334	326	326
11	inc	7	9	702	616	636	572	578	618	542	542	558	558
12	misex2	25	18	528	540	616	674	784	784	912	908	908	908
13	rd53	5	3	130	78	62	90	90	90	102	102	102	102
14	rd73	7	3	370	214	70	254	166	122	142	142	142	142
15	rd84	8	4	568	320	124	352	208	224	368	284	236	236
16	sao2	10	4	1927	978	918	1028	634	812	1166	900	800	800
17	squar5	5	8	214	214	216	196	196	196	204	204	204	204
18	t481	16	1	80	368	210	102	106	310	214	214	214	214
19	vg2	25	8	3869	2276	2244	2258	3126	5246	2710	3764	3784	4252
20	xor5	5	1	18	18	18	22	22	22	34	34	34	34

Figure 52: DFC measures for ‘r’ option alone

#	Examples	Var		Bound input - Bound output									
		i/p	o/p	2-1	3-1	3-2	4-1	4-2	4-3	5-1	5-2	5-3	5-4
1	5xp1	7	10	502	456	390	390	382	320	350	350	350	350
2	9sym	9	1	662	366	206	242	178	194	354	206	178	178
3	Z5xp1	7	10	636	428	394	390	382	328	320	320	320	320
4	adr2	4	3	42	38	38	42	42	42	42	42	42	44
5	b12	15	9	358	376	350	484	506	486	554	530	530	530
6	bw	5	28	1042	796	730	764	840	840	920	920	920	920
7	cc	21	20	252	236	224	448	468	468	476	476	476	476
8	clip	9	5	1488	1152	1138	1170	1196	972	1260	1052	760	800
9	duke2	22	29	5045	3365	4867	3396	4520	7208	4476	6956	7138	8498
10	f51m	8	8	474	352	318	380	324	338	374	334	326	702
11	inc	7	9	702	616	636	572	578	618	542	542	558	558
12	misex2	25	18	528	540	616	674	784	784	912	908	908	908
13	rd53	5	3	130	78	62	90	90	90	102	102	102	102
14	rd73	7	3	370	214	70	254	166	122	142	142	142	142
15	rd84	8	4	568	320	124	352	208	224	368	284	236	236
16	sao2	10	4	1927	978	918	1028	634	812	1166	900	800	800
17	squar5	5	8	214	242	216	196	196	196	204	204	204	204
18	t481	16	1	80	368	210	102	106	106	310	214	214	214
19	vg2	25	8	3869	2276	2244	2258	3126	5246	2710	3764	3784	4252
20	xor5	5	1	18	18	18	22	22	22	34	34	34	34

Figure 53: DFC measures for both 'r' and 'm' option

small.w	bound_in	bound_out	time	CLB's	level	DFC
	5	6	:01	9	3	
	5	5	:01	9	3	292
	5	4	:01	9	3	292
	5	3	:01	9	3	292
	5	2	:01	9	3	292
	5	1	:01	13	4	268
	4	4	:01	21	6	308
	4	3	:03	21	6	308
	4	2	:01	19	5	264
	4	1	:02	16	4	174

Table 8: Small function with different inputs and outputs of bound set

duke2.w	bound_in	bound_out	time	CLB's	level
	5	6	1:55	270	9
	5	5	1:56	270	9
	5	3	2:01	272	9
	4	3	2:07	481	13
	3	1	2:00	508	7

Table 9: Function duke.2

temp.w	bound_in	bound_out	time	CLB's	level
	5	6	:08	99	2
	5	3	:08	99	2
	4	3	:09	133	4
temp2.w	boundi_in	bound_out	time	CLB's	level
	5	5	:10	24	4
	5	3	:09	24	4
	4	3	:06	22	4

Table 10: temp and temp2

13.3 TESTING VARIOUS VARIABLE PARITIONING STRATEGIES IN TRADE

TRADE: Variable Partitioning Test. Stanislaw Grygiel
CLB: 5 inputs, 3 outputs July 19, 1995
number in the 1-st line of description = number of CLBs
number in the 2-nd line of description = number of levels
number in parenthesis (if any) = number of EXOR decompositions

	full orig	full mod	full+ lrand48	get_part	get_part +lrand48	add_part	get_part add_part	get_part pick_clb	add_part pick_clb
bw.w	27	27	27	27	27	27	27	27	27
i= 5, o=28	1	1	1	1	1	1	1	1	1
bench.w	16	16	16	16	18	18	16	16	17
i= 6, o= 8	2	2	2	2	2	2	2	2	2
fout.w	26	26	26	26	26	26	26	27	26
i= 6, o=10	2	2	2	2	2	2	2	2	2
5xp1.w	11	11	11	18	19	16	16	11	11
i= 7, o=10	2	2	2	2	2	2	2	2	2
rd73.w	5	5	5	7	7	7	7	5	5
i= 7, o= 3	2	2	2	2	2	2	2	2	2
z4ml.w	4	4	4	8	7	8	8	4	4
i= 7, o= 4	2	2	2	2	2	2	2	2	2
f51m.w	9	9	9	15	12	13	13	9	9
i= 8, o= 8	3	3	3	3	3	3	3	3	3
misex1.w	14	15	16	15	16	15	15	15	15
i= 8, o= 7	2	2	2	2	2	2	2	2	2
rd84.w	11	11	11	12	12	12	12	11	11
i= 8, o= 4	3	3	3	3	3	3	3	3	3
root.w	21	21	22	27	32	23	22	25	23
i= 8, o= 5	4(1)	4(1)	4(1)	4(2)	4(2)	4(1)	3(1)	4(2)	4(1)
test1.w	68	70	65	74	70	73	73	68	69
i= 8, o=10	4	4(5)	4(4)	4(7)	4(4)	4(7)	4(6)	4(6)	4(6)
9sym.w	6	6	6	7	7	6	6	7	6
i= 9, o= 1	3	3	3	3	3	3	3	3	3
9symml.w	6	6	6	7	6	6	6	7	6
i= 9, o= 1	3	3	3	3	3	3	3	3	3
clip.w	37	38	29	48	32	40	40	42	40
i= 9, o= 5	4(2)	4(2)	4(1)	4(3)	3	4(2)	4(2)	4(3)	4(2)
alu2.w	22	23	21	25	44	38	24	23	21
i=10, o= 8	3	3	3	3	5(2)	5(1)	3	3	3
sao2.w	34	33	32	29	36	39	29	34	34
i=10, o= 4	5(1)	4(1)	4	4(1)	5(1)	5(3)	4(1)	4(1)	4(1)
b9.w	39	29	46	46	59	50	35	43	33
i=16, o= 5	6(1)	4	6(3)	5(2)	7(5)	6(5)	4	7(2)	3
duke2.w	246	235	259	280	360	292	264	253	266
i=22, o=29	9(11)	9(8)	9(11)	8(14)	8(24)	8(13)	9(9)	8(10)	9(11)
misex2.w	31	31	32	38	42	45	38	31	40
i=25, o=18	4	3	3	3	4	4	3	3	4

Table 11: TRADE: Variable partitioning test.

14 COMPARISON OF TRADE and DEMAIN option in MULTIS

Table 12: Comparison of DEMAIN and TRADE

Program				DEMAIN				TRADE			
Benchmark				CLB5/3		CLB5/1		CLB5/3		CLB5/1	
name	in	out	cubes	cells	time(s)	cells	time(s)	cells	time(s)	cells	time(s)
9sym	9	1	158	3	1.2	13	69.6	6	2.3	11	10.3
rd84	8	4	515	4	3.0	13	2.7	11	5.3	14	21.0
rd73	7	3	274	2	1.2	9	1.1	5	2.0	5	2.1
sao2	10	4	133	14	395.0	31	579.8	30	7.8	39	28.6
clip	9	5	271	19	1656.5	42	1972.8	37	9.5	57	40.5
5xp1	7	10	143	7	3.5	18	4.1	11	2.8	11	2.7
b12	15	9	72	20	1387.7	17	7691.4	17	4.0	18	3.9
f51m	8	8	154	8	59.3	18	11.2	9	1.7	12	2.1
bw	5	28	97	[1]	-	28	1.0	27	0.4	27	0.5
duke2	22	29	406	[2]	-	[2]	-	244	122.6	153	179.4
cc	21	20	96	[2]	-	[2]	-	24	3.7	24	3.6
misex1	8	7	40	5	1.0	19	1.4	15	2.1	14	2.2
cu	14	8	1150	10	2.3	28	80.7	13	3.1	14	2.7
misex2	25	18	101	[2]	-	[2]	-	31	9.9	31	10.2
i1	25	16	69	[3]	-	[3]	-	18	4.4	18	4.5
c8	28	18	166	[3]	-	[3]	-	35	9.5	35	10.0
alu2	10	6	315	[3]	-	[3]	-	69	32.7	79	180.7
9symm1	9	1	165	3	1.1	13	70.6	6	2.3	11	11.0

1 : Command terminated abnormally.

2 : Table inconsistent.

3 : Could not be completed.

All benchmarks used for DEMAIN and TRADE are Espresso format. .type fr sets the logic interpretation of the character matrix of output array. (1: ON set, 0: OFF set, -/ : no meaning)

Table 13: Comparison of Demain and Trade - new variants

Program				DEMAIN				TRADE			
Benchmark				CLB5/3		CLB5/1		CLB5/3		CLB5/1	
name	in	out	cubes	cells	time(s)	cells	time(s)	cells	time(s)	cells	time(s)
9sym	9	1	158	3	1.0	13	70.0	6	2.3	11	10.3
rd84	8	4	515	6	5.6	19	62.5	11	5.3	14	21.0
rd73	7	3	274	2	1.3	10	13.2	5	2.0	5	2.1
sao2	10	4	133	14	108.0	64	240.7	30	7.8	39	28.6
clip	9	5	271	9	153.1	54	349.8	37	9.5	57	40.5
5xp1	7	10	143	6	3.4	28	13.3	11	2.8	11	2.7
b12	15	9	72	12	10.5	33	30.0	17	4.0	18	3.9
f51m	8	8	154	7	4.0	18	21.6	9	1.7	12	2.1
bw	5	28	97	[1]	-	28	1.0	27	0.4	27	0.5
duke2	22	29	406	[2]	-	[2]	-	244	122.6	153	179.4
cc	21	20	96	[2]	-	[2]	-	24	3.7	24	3.6
misex1	8	7	40	6	0.9	23	7.9	15	2.1	14	2.2
cu	14	8	1150	9	2.3	27	128.7	13	3.1	14	2.7
misex2	25	18	101	[2]	-	[2]	-	31	9.9	31	10.2
i1	25	16	69	[2]	-	[2]	-	18	4.4	18	4.5
c8	28	18	166	[2]	-	[2]	-	35	9.5	35	10.0
alu2	10	6	315	[3]	-	[3]	-	69	32.7	79	180.7
9symm1	9	1	165	3	1.1	13	70.6	6	2.3	11	11.0

If the number of output is greater than the number of CLB's output, do parallel decomposition. [1] Command terminated abnormally [2] Table inconsistent. [3] Still running without result.

15 CONCLUSIONS

In this report, first we presented a general methodology to create search strategies for combinatorial problems encountered in a general-purpose Functional Decomposer.

We demonstrated that the method to create such strategies is general enough to solve several combinatorial problems that we have to deal with in functional decomposition. These problems include the following:

- set covering for Column Minimization.
- graph coloring for Column Minimization.
- set covering for combined Column Minimization and Encoding.
- set covering for combined Column Minimization and Encoding.
- argument selection for parallel decomposition.
- Variable Paritioning.
- encoding.

We hope, that our readers, including the first-year graduate students at PSU, will be able to use the text, its examples, and also the problems given in sections "Questions for Self-Evaluation" to learn our approach well enough to be able to apply it to other combinatorial problems that we already found must be solved in the decomposer, or we will find in the future.

We used a similar approach extensively in the past: the multipurpose programs [49, 51, 52], designed over the years, have been used to prototype several CAD programs including implicant generation, covering and covering/closure problems, layering, placement and partitioning problems, state assignment of FSM's, microprogram transformation, multi-output TANT network minimization [49], state minimization [55], concurrent state minimization and state assignment [38], optimization of asynchronous FSMs [53], PAL minimization [44], and column minimization in FSM [57]. It allowed to prototype algorithms quickly. For some problems we found a good algorithm without much effort, and next the corresponding optimized code in C was written. The presented approach allows for the creation of *both optimal and approximate algorithms.* Access to the optimal algorithm allows one to *evaluate the approximate algorithms* (since the optimal cost is known for each small size data) and to improve them. The way in which the developer transforms a problem formulation to a ready and running program is shorter and more straightforward. The approach systematizes and uniformizes the problem-solving and optimization methods as well as their applications to different problems that are of interest to us. In this way, these methods are founded on standard fundamentals that provide a *general development methodology.*

It is our goal now to implement all the versions that have not yet been implemented to gain practical understanding on the role of strategies and heuristics in combinatorial problems of functional decomposition.

This will allow us to answer the most important questions for construction of decomposers, to which answers cannot be found in the literature:

- What is the role of variable partitioning as compared to column minimization, to encoding in a decomposer?
- How much effect is gained by various entire decomposition strategies?
- How much effect is gained by various variable partitioning strategies?
- How much effect is gained by various Column Minimization strategies?
- How much effect is gained by various Encoding strategies?

References

[1] S.B. Akers, "On the use of linear assignment algorithm in module placement," *25 Years of Electronic Design Automation,* 1988, pp. 218-223.

[2] A. Angyal, "The Structure of Wholes," *Philosophy of Science*, 1939, pp. 25-37.

[3] W.R. Ashby, "An Introduction to Cybernetics," *Methuen & Co., London*, 1964.

[4] A. Bagchi, and A. Mahanti, "Search Algorithms under Different Kinds of Heuristics - A Comparative Study," *JACM,* Vol. 30., 1983, pp. 1-21.

[5] H. Berliner, "On the Construction of Evaluation Functions for Large Domains," *Proceedings IJCAI-79,* Tokyo, Japan, 1979, pp. 53-55.

[6] P. Bertolazzi, and A. Sassano, "A class of polynomially solvable set-covering problems," *SIAM J. Discrete Math.,* Vol. 1., No. 3., August 1988, pp. 306-316.

[7] P. Bertolazzi, and A. Sassano, "An O(mn) algorithm for regular set-covering problems," *Theor. Comput. Sci.,* Vol. 54., 2,3, Oct. 1987, pp. 237-247.

[8] R. K. Brayton, G.D. Hachtel, C.T. McMullen, and A. L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis,* Kluwer Academic Publishers, 1984.

[9] B. G. Buchanan, C.R. Johnson, T. M. Mitchell, and R. G. Smith, "Models of Learning Systems," in: Belzer, J. (Ed.), *Encyclopedia of Computer Science and Technology, 11, Marcel Dekker,* New York, 1978, pp. 24-51.

[10] B. Carpenter, and IV. N. Davis, "Implementation and performance analysis of parallel assignment algorithms on a hypercube computer," *Proc. "Hypercube Concurrent Computers and Applications,* Vol. 2., Pasadena, CA, Jan. 19-20, 1980, pp. 1231-1235.

[11] A. Chan, "Using decision trees to derive the complement of a binary function with multiple-valued inputs," *IEEE Trans. Comp.,* Vol. C-36, No. 2., Febr. 1987, pp. 212-214.

[12] R.C. Conant, "Detecting Subsystems of a Complex System," *IEEE Transactions on Systems, Man, and Cybernetics*, 1972, pp. 350-353.

[13] R.C. Conant, "Set-Theoretic Structure Modeling," *International Journal of General Systems,* 1981, No. 38, Vol. 7, pp. 93-107.

[14] D.L. Dietmeyer, *Logic Design of Digital Systems,* Allyn and Bacon, Boston 1971.

[15] C. Files, "Using a Search Heuristic in an NP-Complete Problem in Ashenhurst-Curtis Decomposition," *Graduate Summer Research Program,* Wright Laboratory, August, 1994.

[16] J. Frackowiak, "The synthesis of minimal hazardless TANT networks," *IEEE Trans. on Comp.,* Vol. C-21, No. 10, pp. 1099-1108, Oct. 1972.

[17] J. Frenk, M. Van Houweninge, and R. Kan, "Order statistics and the linear assignment problem", *Computing,* N.Y., Vol. 39, April 1, 1987, pp. 165-174.

[18] R. S. Garfinkel, and G.L. Nemhauser, *Integer Programming,* Wiley, N.Y., 1972.

[19] W.R. Garner, and W.J. McGill, "The Relation Between Information and Variance Analyses," *Psychometrica,* 1956, Vol. 21, No. 3., pp. 219-228.

[20] J.F. Gimpel, "The minimization of TANT networks," *IEEE Trans. on Comp.* Vol. EC-16, pp. 18-38, February 1967.

[21] A. K. Griffith, "A Comparison and Evaluation of Three Machine Learning Procedures as Applied to the Game of Checkers," *Artificial Intelligence,* Vol. 5., 1974, pp. 137-148.

[22] D. S. Hochbaum, "Approximation algorithms for the weighted set covering and node covering problems," *SIAM J. Comput.,* Vol. 11, 1982, pp. 535-556.

[23] L. J. Hubert, "Assignment Methods in Combinatorial Data Analysis," *Marcel Dekker Inc.,* New York/Basel, 1987.

[24] L. J. Hubert, "Statistical applications of linear assignment," *Psychometrica,* Vol. 49, 1984, pp. 449-473.

[25] S. L. Hurst, *The Logical Processing of Digital Signals.* Crane-Russak, New York and Edward Arnold, London, 1978.

[26] Ibaraki, to be found.

[27] R. Jonker, and Volgenant, "A shortest augmenting path algorithm for dense and sparse linear assignment problems," *Computing,* N.Y., Vol. 38., No. 4., March 1987, pp. 325-340.

[28] G.J. Klir, "Architecture of Systems Problem Solving," *Plenum Press,* New York, 1985.

[29] G.A. Miller, "What is Information Measurement," *American Psychologist*, 1963, No. 8, pp. 3-11.

[30] R.M. Karp, *Reducibility Among Combinatorial Problems, Complexity of Computer Computation,* Plenum, ed. Miller, New York, 1972, pp. 85-103.

[31] D. Knoke, and P.J. Burke, "Log-Linear Models," *Sage Publications,* Beverly Hills, California, 1980.

[32] Z. Kohavi, *Switching And Finite Automata Theory,* McGraw Hill, New York, 1970.

[33] T. C. Koopmans, and M. Beckmann, "Assignment problems and the location of economic activities," *Econometrica,* 25., pp. 53-76, 1957.

[34] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Res. Logist. Quart.,* 2., pp. 83-87, 1955.

[35] K. Krippendorf, "On the Identification of Structures in Multivariate Data by the Spectral Analysis of Relations," *Proc. of the 23rd Annual Meeting of the Society for General Systems Research*, 1979, pp. 82-91, Louisville, Kentucky.

[36] K. Krippendorf, "Information Theory: Structural Models for Qualitative Data," *Sage Publications,* Beverly Hills, California, 1986.

[37] E. L. Lawler, J. K. Lenstra, A.H.G. Rinooy Kan, and D.B., Shmoys, "The Traveling Salesman Problem. A Guided Tour of Combinatorial Optimization," *Wiley,* 1985.

[38] E. B. Lee, and M. Perkowski, "Concurrent minimization and state assignment of Finite State Machines," *Proc. IEEE Intern. Conf. on Systems, Man, and Cybernetics,* Halifax, Nova Scotia, Canada, October 9-12, 1984.

[39] K. F. Lee, and S. Mahajan, "A Pattern Classification Approach to Evaluation Function Learning," *Artificial Intelligence,* Vol. 36., pp. 1-25, 1988.

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

[41] G. Luger, and W. Stubblefield, "Artificial intelligence: structures and strategies for complex problem solving," *The Benjamin/Cumnnings Publ. Comp.*, Ed. 2. 1993.

[42] T. Mitchell, J. Carbonell, and R. Michalski, (eds), *Machine Learning: A Guide to Current Research. (Knowledge Representation, Learning, and Expert Systems).* Kluwer Academic Publishers, Nowell, MA, 1986.

[43] R. Mirchandaney, and J. Stankovic, "Using Stochastic Learning Automata for Job Scheduling in Distributed Processing Systems," *J. Parallel Distrib. Comput.,* Vol. 3., No. 4., Dec. 1987, pp. 527-552.

[44] L. Nguyen, M. Perkowski, and N. Goldstein, "PALMINI - fast Boolan minimizer for personal computers," *Proc. of 24th Design Automation Conference,* June 28 - July 1, Miami, Florida, June 1987, pp. 615-621.

[45] N. J. Nilsson, *Learning Machines,* McGraw-Hill, New York, 1965.

[46] N. J. Nilsson, *Problem-Solving Methods in Artificial Intelligence,* McGraw Hill, New York 1971.

[47] A. Papoulis, "Probability, random variables, and stochastic processes," *McGraw-Hill, Inc.*, 1991.

[48] Y.T. Lai, K.R. Pan, M. Pedram, S. Vrudhula, "FGMap: A Technology Mapping Algorithm for Look-up Table Type FPGA Synthesis," *Proc. of 30-th DAC,* pp. 642-647, 1993.

[49] M. Perkowski, "Synthesis of multioutput three level NAND networks," *Proc. of the Seminar on Computer Aided Design,* Budapest, Hungary, 3-5 November 1976, pp. 238-265.

[50] M. Perkowski, "Multipurpose and multistrategical program to solve combinatorial problems," *Proc. III Symp. Methods of Heuresis,* Polish Cybernetical Society, Warsaw, 1976, Vol. 3, pp. 23-90.

[51] M. Perkowski, "An application of general problem solving methods in Computer Aided Design. The Multicomp system and its problem oriented source language," *Proc. IV Intern. Symp. Methods of Heuresis,* Polish Cybernetical Society, Warsaw, 1977, Vol. 3, pp. 55-102.

[52] M. Perkowski, "The state space approach to the design of multipurpose problem solver for logic design," *Proc. IFIP WG 5.2. Conference "Artificial Intelligence and Pattern Recognition in Computer Aided Design,* Grenoble 17-19 March, North Holland Publ. Comp., Amsterdam, 1978, pp. 123- 140.

[53] M. Perkowski, and A. Zasowska, "Minimal area MOS asynchronous automata," *Proc. of International Symposium on Applied Aspects of Automata Theory,* Bulg. Acad. Sci., Warna, Bulgaria, 14-19 May 1979, pp. 284-298.

[54] M. Perkowski, "General methods for solving combinational problems," Chapter 4 in (A. Goralski ed.), *Problem, Method, Solution,* Vol. 4, Scientific-Technical Publishers, Warsaw 1982, pp. 110-149, (in Polish).

[55] M. Perkowski, and N. Nguyen, "Minimization of Finite State Machines in SuperPeg," *Proc. of the Midwest Symposium on Circuits and Systems,* Louisville, Kentucky, 22-24 August 1985, pp. 139-147.

[56] M. Perkowski, "Minimization of Two-Level Networks from Negative Gates," *Proc. of Midwest 86 Symposium on Circuits and Systems,* Lincoln, Nebraska, 1-2 August 1986, pp. 756-761.

[57] M. Perkowski, H. Uong, and H. Uong, "Automatic Design of Finite State Machines with Electronically Programmable Devices," record of *Northcon 87,* Portland, 1987, paper 16/4, pp. 1-15.

[58] M. Perkowski, H. Uong, "Generalized Decomposition of Incompletely Specified Multioutput, Multi-Valued Boolean Functions," *Unpublished manuscript, Department of Electrical Engineering,* PSU 1987.

[59] M. Perkowski, J. Brown, "A Unified Approach to Designs with Multiplexers and to the Decomposition of Boolean Functions," *Proc.ASEE Annual Conference,* pp.1610-1619, 1988.

[60] M.A. Perkowski, J. Liu, J.E. Brown, "Rapid Software Prototyping: CAD Design of Digital CAD Algorithms," In G. W. Zobrist (ed), *Progress in Computer-Aided VLSI Design,* Vol. 1, pp. 353-401, 1989.

[61] M. Perkowski, and J. Liu, "A System for Fast Prototyping of Logic Design Programs," *Proc. 1987 Midwest Symposium on Circuits and Systems,* Syracuse, New York.

[62] M. Perkowski, and J. Liu, "Minimization of TANT networks," *Proc. ISCAS'90.*

[63] M. Perkowski, et al., Report to WL, 1995.

[64] M. Perkowski et al, Encoding Report for WL, 1995.

[65] D. Pertsekas, "The auction algorithm: a distributed relaxation method for the assignment problem," *Oper. Res.,* Vol. 14., 1-4, June 1988, pp. 105-123.

[66] L. Pyber, "Clique covering of graphs," *Combinatorica,* Vol. 6., No. 4., 1986, pp. 393-398.

[67] L. Rendell, "A New Basis for State-Space Learning Systems and a Successful Implementation," *Artificial Intelligence,* Vol. 20., 1983, pp. 369-392.

[68] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," *IBM J.,* Vol. 3., 1959, pp. 210-229.

[69] A. L. Samuel, "Some Studies in Machine Learning Using the Game of Checkers," II, *IBM J.,* Vol. 11., 1967, pp. 601-617.

[70] C.E. Shannon, and W. Weaver, "The Mathematical Theory of Communication," *University of Illinois Press,* 1975 (first published in 1949).

[71] J. R. Slagle, "Artificial Intelligence: the Heuristic Programming Approach," *McGraw Hill,* New York 1970.

[72] R. L. Thorndike, "The problem of classification of personnel," *Psymetrika,* 15, 1950, pp. 215-235.

[73] Ch. J. Tseng, and D. P. Siewiorek, "Automated synthesis of data path in digital systems," *IEEE Trans on CAD,* Vol. CAD-5, No. 3, July 1986, pp. 379-395.

[74] M.E. Ulug, and B.A. Bowen, "A unified theory of the algebraic topological methods for the synthesis of switching systems," *IEEE Trans. Comp.,* Vol. C-23, pp. 255 - 267, March 1974.

[75] W. Wan, "A New Approach to the Decomposition of Incompletely Specified Functions Based on Graph Coloring and Local Transformation and Its Application to FPGA Mapping," *M.S. Thesis,* Portland State University, 1992.

[76] W. Wan, M.A. Perkowski, "A New Approach to the Decomposition of Incompletely Specified Functions based on Graph-Coloring and Local Transformations and Its Application to FPGA Mapping", *Proc. of the IEEE EURO-DAC '92, European Design Automation Conference,* Sept. 7-10, Hamburg, 1992, pp. 230 - 235.

[77] *Proc. of International Workshop on Logic Synthesis,* Vol. 1 + 2, Research Triangle Park, North Carolina, May 12-15, 1987.

[78] H. Wu, and M. A. Perkowski, "Synthesis for Reed-Muller Directed-Acyclic-Graph networks with applications to Binary Decision Diagrams and Fine Grain FPGA Mapping", *Proc. of IWLS '93,* Tahoe City, CA, May 1993.

[79] G. Vanderstraeten, and M. Bergeron, "Automatic assignment of aircraft to gates at a terminal," *Comput. Ind. Eng.,* Vol. 14, No. 1, Jan. 1988, pp. 15-25.

[80] F. Vasko, and F. Wolf, "Solving large set covering problems on a personal computer," *Comput. Oper. Res.,* Vol. 15., No. 2., Febr. 1988, pp. 115-121.

[81] C. Yu, and B. Wah, "Learning dominance relations in combinatorial search problems," *IEEE Trans. Software Engng.,* Vol. 14., No. 8., August 1988, pp. 1155-1175.

[82] W. Zhao, "Two-dimensional minimization of Finite State Machines," *Master Thesis, Dept. Electr. Engn.,* PSU, 1989.

[83] J.C. Hosseini, R.R. Harmon, and M. Zwick, "An Information Theoretic Framework for Exploratory Multivariate Market Segmentation Research," *Decision Sciences,* 1991, Vol. 22, No. 3, pp. 663-677.