Research Report DC-2001-0x

BOOM - a Boolean Minimizer

Petr Fišer, Jan Hlavička

June 2001

Department of Computer Science and Engineering

Faculty of Electrical Engineering

Czech Technical University in Prague
Karlovo nám. 13,
CZ-121 35 Prague 2
Czech Republic

Abstract

This report presents an algorithm for two-level Boolean minimization (BOOM) based on a new implicant generation paradigm. In contrast to all previous minimization methods, where the implicants are generated bottom-up, the proposed approach uses a top-down approach. Thus instead of increasing the dimensionality of implicants by omitting literals from their terms, the dimension of a term is gradually decreased by adding new literals. One of the drawbacks of the classical approach to prime implicant generation, dating back to the original Quine-McCluskey method, is the use of terms (be it minterms or terms of higher dimension) found in the definition of the function to be minimized, as a basis for the solution. Thus the choice of terms used originally for covering the function may influence the final solution. In the proposed method, the original coverage influences the final solution only indirectly, through the number of literals used. Starting from an n-dimensional hypercube (where n is the number of input variables), new terms are generated, whereas only the on-set and off-set are consulted. Thus the original choice of the implicant terms is of a small importance.

Most minimization methods use two basic phases introduced by Quine-McCluskey, known as prime implicant generation and the covering problem solution. Some more modern methods, including the well-known ESPRESSO, combine these two phases, reducing the number of implicants to be processed. A sort of combination of prime implicant generation with the solution of the covering problem is also used in the BOOM approach proposed here, because the search for new literals to be included into a term aims at maximum coverage of the output function (coverage-directed search). The implicants generated during the CD-search are then expanded to become primes. Different heuristics are used during the CD-search and when solving the covering problem.

The function to be minimized is defined by its on-set and off-set, listed in a truth table. Thus the don't care set, which normally represents the dominant part of the truth table, need not be specified explicitly. The proposed minimization method is efficient above all for functions with several hundreds of input variables and with a large portion of don't care states.

The minimization method has been tested on several different kinds of problems. The MCNC standard benchmarks were solved several times in order to evaluate the minimality of the solution and the runtime. Both "easy" and "hard" MCNC benchmarks were solved and compared with the solutions obtained by ESPRESSO. In many cases the time needed to find the minimum solution on an ordinary PC was non-measurable.

The procedure is so fast that even for large problems with hundreds of input variables it often finds a solution in a fraction of a second. Hence if the first solution does not meet the requirements, it can be improved in an iterative manner. Larger problems (with more than 100 input variables and more than 100 terms with defined output values) were generated randomly and solved by BOOM and by ESPRESSO. BOOM was in this case up to 166 times faster. For problems with more than 300 input variables no comparison with any other minimization tool was possible, because no other system, including ESPRESSO, can solve such problems. The dimension of the problems solved by BOOM can easily be increased over 1000 input variables, because the runtime grows linearly with the number of inputs. On the other hand, as the runtime grows roughly with the square of the size of the care set, for problems of very high dimension the success largely depends on the number of care terms. The quality of the proposed method was also tested on other problems like graph coloring and symmetric function minimization.

Keywords
Boolean minimization, PLA minimization, prime implicant, implicant expansion, implicant reduction, mutations, covering problem, ESPRESSO

Acknowledgment

This research was in part supported by grant 102/99/1017 of the Czech Grant Agency (GACR).

Table of Contents

1. Introduction

The problem of two-level minimization of Boolean functions is old, but surely not dead. It is encountered in many design environments, e.g., multi-level logical design, PLA design, artificial intelligence, software engineering, etc. The minimization methods started with the papers by Quine and McCluskey [McC56], [Qui52], which formed a basis for many follow-up methods. They mostly copied the structure of the original method, implementing the two basic phases known as prime implicant (PI) generation and covering problem (CP) solution. Some more modern methods, including the well-known ESPRESSO [Esp1], [Hac96], try to combine these two phases. This is motivated above all by the fact that the problems encountered in modern application areas like design of control systems, design of built-in self-test equipment, etc., often require minimization of functions with hundreds of input variables, where the number of PIs is prohibitively large. Also the number of don't care states is mostly so large that modern minimization methods must be able to take advantage of all don't care states without enumerating them.

One of the most successful Boolean minimization methods is ESPRESSO and its later improvements. The original ESPRESSO generates near-minimal solutions, as can be seen from the comparison with the results obtained by using alternative methods – see Section 9. ESPRESSO-EXACT [Rud87] was developed in order to improve the quality of the results. The improvement consisted above all in combining the PI generation with set covering. Finally, ESPRESSO-SIGNATURE [McG93] was developed, accelerating the minimization by reducing the number of prime implicants to be processed by introducing the concept of a "signature", which is an intersection of all primes covering one minterm. This in turn was an alternative name given to the concept of "minimal implicants" introduced in [Ngu87].

A combination of PI generation with solution of the CP, leading to a reduction of the total number of PIs generated, is also used in the BOOM (BOOlean Minimizer) approach proposed here. The most important difference between the approaches of ESPRESSO and BOOM is the way they work with the on-set received as function definition. ESPRESSO uses it as an initial solution, which has to be modified (improved) by expansions, reductions, etc. BOOM, on the other hand, uses the input sets (on-set and off-set) only as a reference, which determines whether a tentative solution is correct or not. This allows us to remain to a great extent independent of the properties of the original function coverage. The second main difference is the top-down approach in generating implicants. Instead of expanding the source cubes in order to obtain better coverage, BOOM reduces the universal hypercube until it no longer intersects the off-set while the coverage of the source function is satisfied. The basic principles of the proposed method and the BOOM algorithms were published in some previous reports [1-5]. BOOM was programmed in Borland C++ Builder and tested under MS Windows NT.

This report has the following structure. After a formal problem statement in Section 2, the structure of the BOOM system is described in Section 3 and its iterative mode in Section 4. The initial generation of implicants is described in Section 5 and their expansion into prime implicants in Section 6. The extension of the method to multi-output functions is described in Section 7 and covering problem solution in Section 8. Experimental results are evaluated and commented in Section 9. Section 10 evaluates the time complexity of the algorithm and in Section 11 the BOOM program is described together with its data formats and controls.

2. Problem Statement

2.1. Boolean Minimization

Let us have a set of *m* Boolean functions of *n* input variables $F_1(x_1, x_2, \dots x_n)$, $F_2(x_1, x_2, \dots x_n)$, $\dots$ $F_m(x_1, x_2, \dots x_n)$, whose output values are defined by truth tables. These truth tables describe the **on-set** $F_i(x_1, x_2, \dots x_n)$ and **off-set** $R_i(x_1, x_2, \dots x_n)$ for each of the functions F_i. The terms not represented in the input field of the truth table are implicitly assigned don't care values for all output functions. The **don't care set** $D_i(x_1, x_2, \dots x_n)$ of the function F_i is thus represented by all the terms not used in the input part of the truth table and by the terms to which the don't care values are assigned in the *i*-th output column. Listing the two care sets instead of an on-set and a don't care set, which is usual, e.g., in MCNC benchmarks, is more practical for problems with a large number of input variables, because in these cases the size of the don't care set exceeds the two care sets. We will assume that *n* is of the order of hundreds and that only a few of the 2^n minterms have an output value assigned, i.e., the majority of the minterms are don't care states. Moreover, using off-set in the function definition simplifies checking whether a term is an implicant of the given function. Without the explicit off-set definition, more complicated methods, such as tautology checking used in ESPRESSO [Bra84], must be used, which slows down the minimization process.

Our task is to formulate a synthesis algorithm which will for each output function F_i produce a sum-of-products expression $G_i = g_{1i}+g_{2i}+\dots+g_{ti}$, where $F_i \subseteq G_i$ and $G_i \cap R_i = \emptyset$. The expression $T = \Sigma t_i$ ($i = 1 \dots m$) should be kept minimal.

This formulation of the minimization process uses the number of product terms (implicants) as a universal quality criterion. This is mostly justified, but it should be kept in mind that the measure of minimality should correspond to the needs of the intended application. Thus, e.g., for PLAs, the number of product terms is what counts, whereas the total number of literals has no importance. In some other cases, like in custom design, the total number of literals and the output cost, i.e., the number of inputs into all output OR gates, may be important. Hence we will formulate the method in such a way, that all criteria can be used on demand and allow the user to choose among them.

2.2. Motivation

An example of a design problem with many input variables and many don't care states can be found in the design of built-in self-test (BIST) devices for VLSI circuits. A very common method of BIST design is based on the use of a linear feedback shift register (LFSR) generating a code whose code words are used as test input patterns for the circuit under test. However, before being used as test patterns, these words usually have to be transformed into the patterns needed for fault detection [Cha95]. The LFSR may have more than one hundred stages and the sequence used for testing may have several thousands of states. Thus, e.g., for a circuit with 100 LFSR stages and 1000 test patterns the design of the decoder is a problem with 100 input variables and 2^{100}-1000 don't care states.

Another typical problem with a large number of input variables and only a few care terms is the design of a logic function given by its behavioral description. It is mostly very difficult - or even impossible - to enumerate explicitly all terms to which the output value 1 should be assigned. More likely we will formulate some rules specifying, which outputs have to have a certain value (0,1 or not influenced) for given values of input variables. These rules can be described by a truth table where the on-sets and off-sets of output functions are specified and the rest are don't cares.

3. BOOM Structure

Like most other Boolean minimization algorithms, BOOM consists of two major phases: **generation of implicants** (PIs for single-output functions, group implicants for multi-output functions) and the subsequent **solution of the covering problem**. The generation of implicants for single-output functions consists of two steps: first the **Coverage-Directed Search (CD-Search)** generates a sufficient set of implicants needed for covering the source function and these are then passed to the **Implicant Expansion (IE)** phase, which converts them into PIs.

Multi-output functions are minimized in a similar manner. Each of the output functions is first treated separately; the CD-search and IE phases are performed in order to produce primes covering all output functions. However, to obtain the minimal solution, we may need implicants of more than one output function that are not primes of any (group implicants). Here, **Implicant Reduction** takes place. Then the **Group Covering Problem** is solved and **Output Reduction** is performed. Fig. 3.1 shows the block schematic of the BOOM system.

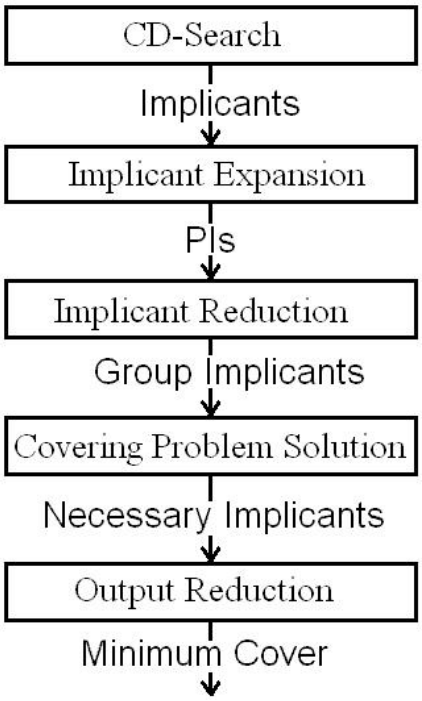

Fig. 3.1 Structure of BOOM

The BOOM system improves the quality of the solution by repeating the implicant generation phase several times and recording all different implicants that were found. At the end of each iteration we have a set of implicants that is sufficient for covering the output function. In each following iteration, another sufficient set is generated and new implicants are added to the previous ones (if the solutions are not equal). After that the covering problem is solved using all obtained primes.

4. Iterative Minimization

Most current heuristic Boolean minimization tools use deterministic algorithms. The minimization process leads then always to the same solution, never mind how many times it is repeated. On the contrary, in the BOOM system the result of minimization depends to a certain extent on random events, because when there are several equal possibilities to choose from, the decision is made randomly. Thus there is a chance that repeated application of the same procedure to the same problem would yield different solutions.

4.1. The Effect of Iterative Approach

The iterative minimization concept takes advantage of the fact that each iteration produces a new set of prime implicants satisfactory for covering all minterms of all output functions. The set of implicants gradually grows until a maximum reachable set is obtained. The typical growth of the size of a PI set as a function of the number of iterations is shown in Fig. 4.1 (thin line). This curve plots the values obtained during the solution of a problem with 20 input variables and 200 minterms. Theoretically, the more primes we have, the better the solution that can be found after solving the covering problem, but the maximum set of primes is often extremely large. In reality, the quality of the final solution, measured by the number of literals in the resulting SOP form, improves rapidly during the first few iterations and then remains unchanged, even though the number of PIs grows further. This fact can be observed in Fig. 4.1 (thick line).

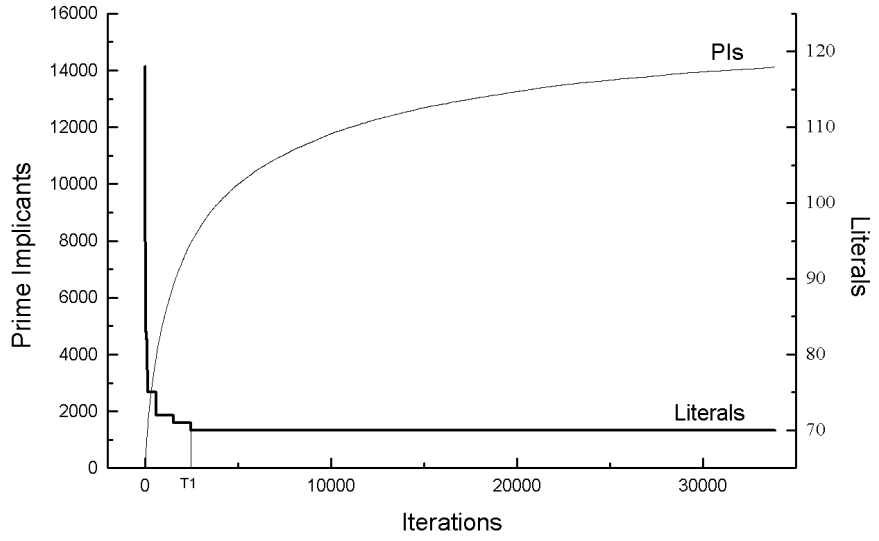

Fig. 4.1 *Growth of PI number and decrease of SOP length during iterative minimization*

From the curves in Fig. 4.1 it is obvious that selecting a suitable moment T1 for terminating the iterative process is of key importance for the efficiency of the minimization. The approximate position of the stopping point can be found by observing the relative change of the solution quality during several consecutive iterations. If the solution does not change during a certain number of iterations (e.g., twice as many iterations as were needed for the last improvement), the minimization is stopped. The amount of elapsed time may be used as an emergency exit for the case of unexpected problem size and complexity.

The iterative minimization of a group of functions F_i (i = 1,2,…m) can be described by the following pseudo-code. The inputs are the on-sets F_i and off-sets R_i of the m functions, the output is a minimized disjunctive form G = (G_1, G_2,...G_m).

Algorithm 1

```
BOOM(F[1..m], R[1..m]) {
     G = ∅
     do
          I = ∅
          for (i = 1; i <= m; i++)
               I′ = CD_Search(F[i], R[i])
               Expand(I′, R[i])
               Reduce(I′, R[1..m])
               I = I ∪ I′
          G′ = Group_cover(I, F[1..m])
          Reduce_output(G′, F[1..m])
          if (Better(G′, G)) then G = G′
     until (stop)
     return G
}
```

4.2. Accelerating Iterative Minimization

When the CD-search phase is repeated, identical implicants are quite often generated in different iterations. These are then passed to the Implicant Expansion phase, which might be unnecessarily repeated. To prevent this, all implicants that were ever produced by the CD-search are stored in the I-buffer (Implicant buffer). Each new implicant is looked up in this buffer, and if it is already present, its further processing is stopped. A flow diagram of the whole minimization algorithm for a multi-output function is shown in Fig. 4.2.

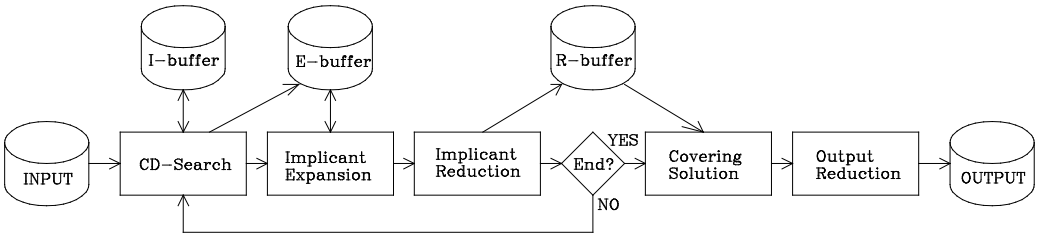

Fig. 4.2 *Iterative minimization schematic plan*

Each newly generated implicant is first looked up in the I-buffer and, if it is not present, it is stored both in the I-buffer and E-buffer (Expansion buffer). The E-buffer serves as a storage of implicants that are candidates for expansion into PIs. After expansion, they are removed from the E-buffer. Then they are reduced to group implicants and, after duplicity and dominance checks, the newly created group implicants are stored in the R-buffer (Reduced implicants buffer). Finally, the covering problem is solved using all the implicants from the

R-buffer. For multioutput functions there are separate I- and E-buffers for each output. The R-buffer is common.

The main implementation requirement for the I-buffer is its high look-up speed, hence it is structured as a ternary tree whose depth is equal to *n*. At the *k*-th level of the tree the direction is chosen according to the polarity (0,1,-) of the *k*-th variable in the searched term. The presence of an implicant is represented by the existence of its corresponding leaf. The tree is dynamically constructed during the addition of implicants into the buffer. An example of such a tree is shown in Fig. 4.3.

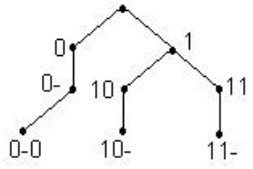

Fig. 4.3 *I-buffer tree structure*

The example shows the structure of a three-variable I-buffer containing implicants 0-0, 10- and 11-. If, e.g., implicant 0-1 is looked for, the search will fail in the node 0- where no path leading to 0-1 is present.

The main properties and advantages of a tree buffer can be visualized by Tab. 4.1 as a comparison between a tree buffer and a linear buffer. Here *i* denotes the number of stored implicants and *n* is the number of input variables.

Tab. 4.1 *Tree buffer properties*

	Tree buffer	Linear buffer
buffer size	$< n.i$, when $i << 3^n$	$n.i$
	$< 3^{n+1}$, when $i \sim 3^n$	$n.i \sim n.3^n$
maximal lookup & insert time	n (on success)	$n.i$ (on failure)
minimal lookup time	1 (on failure)	i (on success)

5. Coverage-Directed Search

5.1. Basis of the Method

The idea of combining implicant generation with the covering problem solution gave rise to the coverage-directed search (CD-search) method used in the BOOM system. This consists in a directed search for the most suitable literals that should be added to some previously constructed term. Thus instead of increasing the dimension of an implicant starting from a 1-minterm, we reduce an *n* dimensional hypercube by adding literals to its term, until it becomes an implicant of F_i. This happens at the moment when this hypercube does not intersect with any 0-term.

The search for suitable literals that should be added to a term is directed towards finding an implicant that covers as many 1-terms as possible. To do this, we start implicant generation by selecting the most frequent input literal from the given on-set, because any term corresponding to a subcube of cube **H** must contain all literals describing **H**. The selected literal describes an *n-1* dimensional hypercube, which may be an implicant, if it does not intersect with any 0-term. If there are some 0-minterms covered, we add one literal and verify whether the new term already corresponds to an implicant by comparing it with 0-terms that can intersect this term. We continue adding literals until an implicant is generated, then we record it and start searching for other implicants.

During the CD-search, the key factor is the efficient selection of literals to be included into the term under construction. After each literal selection we temporarily remove from the on-set the terms that cannot be covered by any term containing the selected literal. These are the terms containing that literal with the opposite polarity. In the remaining on-set we find the most frequent literal and include it into the previously found product term. Again we compare this term with 0-terms and check if it is an implicant. After obtaining an implicant, we remove from the original on-set those terms that are covered by this implicant. Thus we obtain a reduced on-set containing only uncovered terms. Now we repeat the procedure from the beginning and apply it to the uncovered terms, selecting the next most frequently used literal, until another implicant is generated. In this way we generate new implicants, until the whole on-set is covered. The output of this algorithm is a set of product terms covering all 1-terms and not intersecting any 0-term.

The basic CD-search algorithm for a single-output function can be described by the following function in pseudo-code. The inputs are the on-set (F) and the off-set (R), and the output is the sum of products (H) that covers the given on-set. Let us remark that H in this case consists of implicants, which need not be prime.

Algorithm 2

```
CD_Search(F, R) {
     H = ∅                  // H is the cube that is being created
     do
          F′ = F            // F′ is the reduced on-set
          t = true          // t is the term in progress
          do
               v = most_frequent_literal(F′)
               t = t AND v
               F′ = F′ – cubes_not_including(t)
          while (t ∩ R ≠ ∅)
          H = H ∪ t
          F = F - F′
     until (F == ∅)
     return H
}
```

5.2. Immediate Implicant Checking

When selecting the most frequent literal, it may happen that two or more literals have the same frequency of occurrence. In these cases another decision criterion can be applied. We construct terms as candidates for implicants by multiplying all newly selected literals by the

previously selected one(s). Among these terms we select only **the implicants** (if any) and reject the rest. When there are still more possibilities to choose from, we select one at random.

Sometimes this feature prevents a term from being unnecessarily prolonged, because it would have to be shortened during the IE. The effects of using this additional criterion are following:

- it reduces the runtime of CD-search and the whole minimization
- it reduces the number of PIs that are generated

This can be illustrated by the following table. A single-output function with 20 input variables and 500 defined terms was minimized for 1000 iterations. In the first experiment the immediate implicant checking was not used, in the second it was used.

Tab. 5.1 *Immediate implicant checking effects*

	not used	used
total CD-search time [s]	318,9	265,1
total minimization time [s]	6688,3	4782,8
number of PIs found	27194	21741

5.3. CD-Search Example

Let us have a partially defined Boolean logic function of ten input variables and ten defined minterms given by a truth table in Tab. 5.2. Input variables are named x_0 ..x_9. The 1-minterms are highlighted.

Tab. 5.2

```
var: 0123456789
0.   0000000010 1
1.   1000111011 1
2.   0000011001 1
3.   1111011000 0
4.   1011001100 0
5.   1111000100 1
6.   0100010100 0
7.   0011011011 0
8.   0010111100 1
9.   1110111000 1
```

As the first step we count the occurrence of literals in the 1-minterms. The “0”-line and “1”-line in Tab. 5.3 give counts of x_i’ and x_i literals respectively. In this table we select the most frequent literal.

Tab. 5.3

```
var: 0123456789
0:   3435322444
1:   3231344222
```

The most frequent literal is x_3‘ with five occurrences. This literal alone describes a function that is a superset of an implicant, because it covers the 6th minterm (0-minterm) in the original function. Hence another literal must be added. When searching for the next literal, we can reduce the scope of our search by suppressing 1-minterms containing the selected literal with the opposite polarity (in Tab. 5.4 shaded dark). An implicant containing a literal x_3‘ cannot cover the 5th minterm, because it contains the x_3 literal. Thus, we temporarily suppress this minterm. In the remaining 1-minterms we find the most frequent literal.

Tab. 5.4

```
var: 0123456789
0.   0000000010 1
1.   1000111011 1
2.   0000011001 1
3.   1111011000 0
4.   1011001100 0
5.   1111000100 1
6.   0100010100 0
7.   0011011011 0
8.   0010111100 1
9.   1110111000 1

var: 0123456789
0:   343-211433
1:   212-344122
```

As there are several literals with maximal frequency of occurrence 4 (x_1’, x_5, x_6, x_7’), the second selection criterion must be applied. We use these literals tentatively as implicant builders and create four product terms using the previously selected literal x_3‘: x_3‘x_1’, x_3‘x_5, x_3‘x_6, x_3‘x_7’. Then we check which of them are already implicants. The term x_3‘x_5 is not an implicant (it covers the 6th minterm), so it is discarded. Now one of the remaining 3 terms representing implicants must be chosen. We should choose a term, which covers the maximum of yet uncovered 1-minterms (in Tab. 5.4 shaded lightly). As each of these implicants covers four 1-minterms, we can select randomly – e.g. x_3‘x_6. This implicant is stored and the search continues.

The search for literals of the next implicants is described in Tab. 5.5. We omit minterms covered by the selected implicant x_3‘x_6 (dark shading) and select the most frequent literal in the remaining minterms.

Tab. 5.5

```
var: 0123456789
0.   0000000010 1
1.   1000111011 1
2.   0000011001 1
3.   1111011000 0
4.   1011001100 0
5.   1111000100 1
6.   0100010100 0
7.   0011011011 0
8.   0010111100 1
9.   1110111000 1

var: 0123456789
0:   1111222112
1:   1111000110
```

As seen in the lower part of Tab. 5.5, we have four equal possibilities, so we choose one randomly – e.g. x_5'. When we add the x_6' literal we have an implicant covering the remaining two 1-minterms.

The resulting form of the function is $x_3'x_6 + x_5'x_6'$.

5.4. Weights

The fact that the input file may contain both 1-minterms and 1-terms of higher dimension may complicate the search for the most frequent literal. In fact, every term with k don't care input values (representing a k-dimensional hypercube) might be replaced by 2^k minterms, thus increasing the occurrence of the used literals 2^k times. Strictly speaking, each of these literals should be assigned a **weight** corresponding to this factor. This is, however, not feasible, because for functions with several hundreds of input variables the number of vertices of any hypercube may reach astronomic values. Different approaches to the solution of this problem have been evaluated and tested. The best results were obtained when no weights were assigned to the literals in connection with the dimensions of the input terms.

5.5. Mutations

The heuristics used to implement individual steps of this procedure are based on a study of the statistical properties of the given Boolean functions. In some cases this selection criterion may prevent reaching the minimum solution. In other words, there may exist implicants that are unreachable by a strict CD-search, although they are necessary for obtaining the minimum solution. In such cases the mutations, implemented as a random choice used in place of a deterministic decision, may be of help. These mutations may be used in several places in the procedure. This subsection will investigate their usefulness, i.e., the quality of the solution obtained and the time needed to find the solution.

First let us introduce a formal definition of a mutation. We assume a set $\boldsymbol{X}$ of elements x_i with weights $w(x_i)$ assigned. The selection function S($\boldsymbol{Y}$), $\boldsymbol{Y} \subset \boldsymbol{X}$, is a mapping into $\boldsymbol{X}$ such that the function S returns the element $x_j \in \boldsymbol{Y}$ with the highest value of w. As a **mutation** of the function S($\boldsymbol{Y}$) we will denote a random selection of an element from $\boldsymbol{Y}$, i.e., a selection that is

not influenced by the weights. The selection function affected by the mutations will be denoted as $S_m(\boldsymbol{Y}, k)$, where $k \in \langle 0,1 \rangle$ is the mutation rate. S_m then returns a random element from $\boldsymbol{Y}$ with the probability k and the element $S(\boldsymbol{Y})$ with the probability 1-k.

The more mutations are implanted, the faster is the growth of the number of primes during iterative minimization. This is caused by the variety of implicants that are being produced. Figs. 5.1 and 5.2 show the growth of the number of prime implicants as a function of the number of iterations and the time, respectively. The mutation rate k was changed as a parameter from 20 to 100 %. The problem solved was the minimization of a single-output function of 20 input variables with 300 defined minterms.

Although the number of PIs grows faster for higher mutation rates, the CD-search is slowed down. This is because implicants that cover fewer 1-terms are produced and thus more of them must be generated to cover all the on-set. The time needed for one pass of the CD-search as a function of the mutation rate is shown in Fig. 5.3.

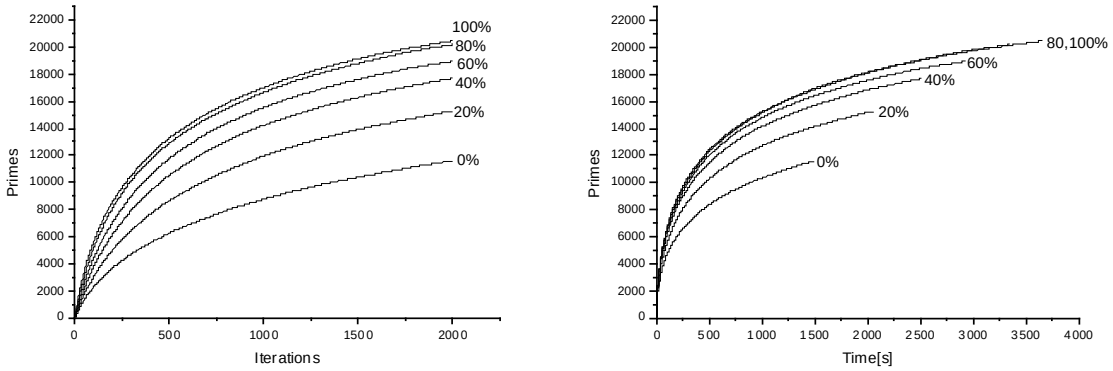

Fig. 5.1, 5.2 *PI set growth for various mutation rates*

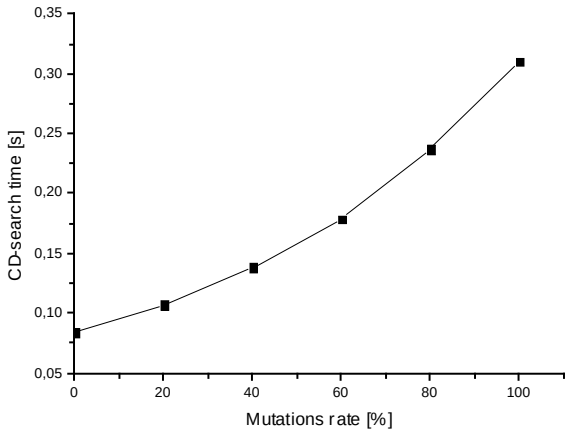

Fig. 5.3 *CD-search run-time growth with increasing mutation rate*

The effects documented above can be summarized in the following way: the mutations slow down the whole minimization process and make it less effective, hence selecting literals with maximum frequency of occurrence is the best method of literal selection. The necessary set of implicants for covering the on-set is then reached in the shortest time, and any deviation from this rule will slow down the algorithm.

However, experiments show that 2-5% of mutations can improve the result by producing some originally unreachable implicants. For example, it can easily be proved that for the problem given by the truth table in Tab. 5.6 a simple CD-search without the use of mutations cannot reach the minimum two-term solution shown below.

Tab. 5.6 *Mutations example*

```
abcdef
001000 0
000101 0
111011 0
010111 0
101011 0
101110 1
001110 1
000110 0
100000 1
010000 1

exact solution: cd + c'd'
solution without the use of mutations: af' + bf' + cef'
```

5.6. CD-Search History

The key motivation for the iterative minimization is the generation of new implicants in the CD-search phase. However, it often generates equal solutions. It is difficult to prevent this, but we may easily prevent the CD-search from following the same way it went ever before.

Each pass through the CD-search can be symbolized as a path from the root of a tree to its leaf. The root represents the beginning (no literals were selected yet), every level represents one literal selection and the leaf is the final solution. Every branching represents more choices of literal selection, while some of the branches are preferable.

When running the CD-search, we may construct this tree and mark every node with a symbol indicating whether there is any possibility of branching. If there is a branching possible, we keep this node "open" for the next pass, otherwise we close it and next time the algorithm should not enter this node. This will prevent reaching the same leaf twice. When all possibilities of choice are exhausted, the root node will become closed and we cannot produce any new implicants any more. This tree will be denoted as a **CD-search history**.

Until now we have assumed that the branching occurs only when two equal literal selections are possible and one is made randomly. The criteria for literal selection are the following:

1. the literal has maximal frequency of occurrence and makes an implicant
2. the literal has maximal frequency of occurrence

3. the literal has non-zero frequency of occurrence

All of these are possible, while only the first one is used in the simple CD-search, the others are used e.g. in mutations, even if randomly and spontaneously. Intuitively, the preference of these criteria is decreasing (1 is the best one), but also the other possibilities may be of any use (see mutations). Using the history we can gradually try out all these possibilities by a simple modification of the previous algorithm. Instead of marking whether a node is simply "open" or "closed" we mark whether it is closed for a particular level of literal selection (we assign a value to the attribute *closed*). For example, if all literal selections according to the first criterion are exhausted for all branches of a node, this node is marked as closed for the first criterion (*closed* = 1). Initially the root and all newly created nodes are marked *closed* = 0. Now when repeating the CD-search we simply do not enter nodes that have a **bigger** *closed* attribute than the node we are currently in. The effect of this is that first we exhaust all possibilities of a "strict" literal selection according to the first criterion. When there are no more possibilities and the root is marked closed for 1, we try the selection according to the second criterion and finally the third one. Note that when using the third criterion, we do not select a literal at random, but again the literal with the maximal frequency of occurrence among the "free" ones is selected. At the end, when the root is closed for all criteria, no other implicants can be ever produced.

The progress of the CD-search when using history is shown in Fig. 5.4. A very simple problem of 5 input variables and 10 minterms defined was solved, in more complex problems the process is much slower. The three curves show the numbers of nodes marked 0, 1 and 2. When the solid curve representing the 0-marked nodes sinks to zero (near 2000 iterations), all best possible selections are exhausted (according the first selection criterion) and the root node is marked 1. When both solid and dashed lines sink to zero (near 11000 iterations), all selections according criterion 2 are exhausted, the root node is marked 2 and next even the third criterion can be used. After that the scope of possible selections is very large and the number of 0, 1 and 2-marked nodes grows very fast.

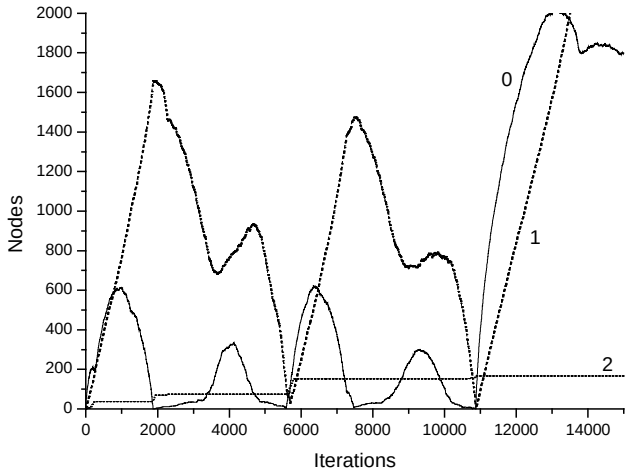

Fig. 5.4 *CD-Search history illustration*

6. Implicant Expansion

As mentioned above, the implicants constructed during the CD search need not be prime. To reduce the number of implicants needed to cover all 1-terms of the given function, we

have to increase their size by IE, which means removing literals (variables) from their terms. When no literal can be removed from the term any more, we get a prime implicant. The expansion is a very sensitive operation in the sense that much effort may be wasted if a bad strategy is chosen, and the result may be far from optimum at that.

There are basically two problems to be solved in connection with implicant expansion. One of them is the mechanism that effectively checks whether a tentative literal removal is acceptable. The other is the selection of the literals and the order in which they are to be removed from the implicant term. First let us discuss the checking mechanism.

6.1. Checking a Literal Removal

Removing a variable from a term doubles the number of minterms covered by the term. The newly covered minterms may be 1-minterms or DC-minterms, but none of them should be a 0-minterm. In BOOM, individual literals are tried for removal and checked whether the expanded term does not intersect the off-set. This means that the DC terms need not be enumerated explicitly, because every newly created implicant is compared with the 0-terms. If an intersection is found, the removal is cancelled.

6.2. Expansion Strategy

The second problem is the selection strategy for the literals to be removed. The expansion of one implicant may yield several different prime implicants. To find them all, we have to try systematically to remove each literal, whereas the order of the literals selected plays an important role. Trying all possible sequences of literals to be removed will be denoted as an exhaustive search or **Exhaustive Implicant Expansion.** Using recursion or queue, all possible literal removals can be systematically tried until all primes are obtained. Unfortunately, the complexity of this algorithm is exponential. Hence this method is usable only in problems with up to 20 input variables. Nevertheless, the modification of this method called Distributed Exhaustive Implicant Expansion is generally usable and quite effective - see below.

There exist several other IE methods differing in complexity and quality of results obtained. Some of them that are used in BOOM are described below.

The simplest one, namely a **Sequential Search**, systematically tries to remove from each term all literals one by one, starting from a randomly chosen position. Every removal is checked against the off-set as above, but if the removal is successful, we make it permanent. If, on the contrary, some 0-minterm is covered, we put the literal back and proceed to the next one. After removing all possible literals we obtain one prime implicant covering the original term. This algorithm is greedy, i.e., we stay with one PI even if there are more than one PIs that can be derived from the original implicant. The complexity of this algorithm is linear.

A sequential search obviously cannot reduce the number of product terms, but it reduces the number of literals. The experimental results show that this reduction may reach approximately 25%.

With a **Multiple Sequential Search** we try all possible starting positions and each implicant thus may expand into several PIs. The upper bound of the number of PIs that can be produced from one implicant is n-d, where n is the number of input variables and d is the dimension of the original implicant. The complexity of this algorithm is $O(n.p)$, where p is the number of defined on-terms.

Some search algorithms, especially Exhaustive Implicant Expansion and the Multiple Sequential Search are rather time consuming for large problems. Therefore a distributed version for each of them was proposed and tested. **Distributed Expansion** is based on the idea of distributing the expansion steps among several consecutive iterations. The advantage of this approach is the possibility to stop the expansion at the moment when an acceptable result is reached and save a considerable amount of time, because the quality of the solution is checked after each iteration.

In the **Distributed Multiple Sequential Search** only one pass of a sequential search is made for every implicant. After that, these implicants are stored in the E-buffer. In the following iteration they are processed again together with the newly created implicants, while another starting position for the sequential search is used. When all meaningful starting positions are exhausted, the corresponding implicant is removed from the buffer. In other words, if the Multiple Sequential Search produces *j* primes from some implicant in one pass, the Distributed Multiple Sequential Search will find all of them in *j* iterations.

Distributed Exhaustive Implicant Expansion uses the same mechanism as the Distributed Multiple Sequential Search. In this case, also the partially expanded implicants are stored in the E-buffer. This ensures the exhaustiveness of the expansion.

6.3. Evaluation of Expansion Strategies

The properties of the proposed IE methods and their influence on the minimization process (time and quality of the final solution) will be discussed in this section. The distributed mode of the implicant expansion methods will not be studied separately, as it is always a simple modification of the original algorithm. Hence the results for the given example are similar.

The choice of IE method may influence two properties of the minimization process: the time of minimization and the quality of the result obtained. Fig. 6.1 shows the time of the minimization of a single-output function of 30 input variables and 500 defined minterms as a function of the number of iterations. The growth for the sequential search is linear, which means that an equal time is needed for each iteration. The time for the multiple sequential search and the exhaustive expansion grows faster at the beginning and then it turns to linear with a slower growth. At this point the CD-search no longer produces new implicants and thus the IE and the following phases are not executed. This causes the simple sequential search, which is seemingly the fastest, to become the slowest after a certain number of iterations.

Fig. 6.2 illustrates the growth of the PI set as a function of time. We can see that the Sequential Search achieves the lowest values, although it is the fastest implicant expansion method. However, when this method is used we cannot take advantage of the I-buffer. The implicants are repetitively expanded, even if they have already been expanded in all possible ways. The two other methods achieve higher values, because they put an implicant into the E-buffer only once and then they are blocked by the I-buffer. Hence when the same implicants are generated repetitively by the CD-search, they are not processed any more, which speeds up the whole minimization. We can see that the most complex method, namely exhaustive expansion, produces PIs at the fastest rate.

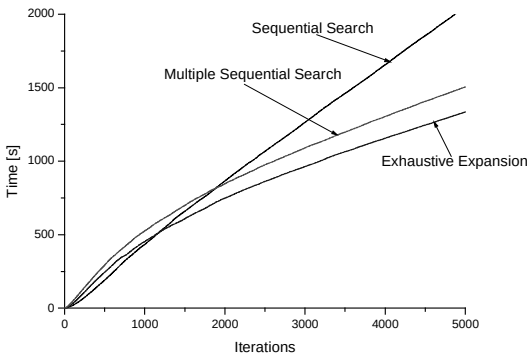

Fig. 6.1 *Growth of time for different IE methods*

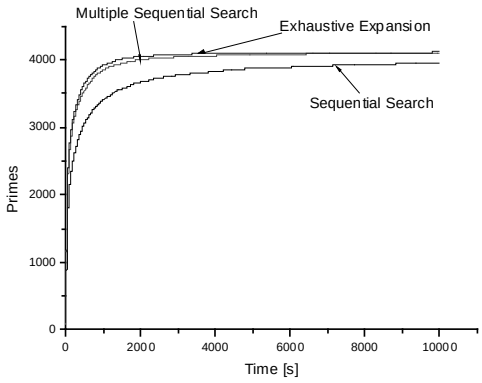

Fig. 6.2 *Growth of PIs for different IE methods*

We saw in subsection 4.1 that even a small number of PIs may give the minimum solution. Moreover, the quality of the final solution strongly depends on the CP solution algorithm. With a large number of PIs exact solving is impossible and some heuristic must be used. Here the large number of implicants may misguide the CP solution algorithm and thereby prevent the minimum solution being achieved. Practice shows that the more complex IE methods are more advantageous for less sparse functions, where the number of implicants in the final solution is big, while the simplest sequential search is better for very sparse functions.

7. Minimizing Multi-Output Functions

To minimize multi-output functions, only few modifications of the algorithm must be done.

First each of the output functions F_i is treated separately, the CD-search and IE phases are performed. After that, we have a set of primes sufficient for covering all m functions. However, for obtaining the minimal solution we may need implicants of more than one input function that are not primes of any F_i. Here the next part of minimization – the **Implicant Reduction** finds place.

As a solution of the covering problem we get a set of implicants needed to cover all F_1 ... F_m. For every output we may find all implicants that do not intersect the off-set of the output function. However, to generate the required output values, some of these implicants may not be necessary. These implicants would create redundant inputs into the output OR gate. Sometimes it is harmless (e.g. in PLAs), moreover it could prevent hazards. Nevertheless, for hardware-independent minimization the redundant outputs should be removed. This is done at the end of the minimization by solving m covering problems for all m functions independently.

7.1. Implicant Reduction (IR)

All obtained primes are tried for reduction by adding literals in order to become implicants of more output functions. The method of implicant reduction is similar to the CD-search. Literals that prevent intersecting given term with most off-set terms are added until there is no chance that the implicant will be used for more functions. Preferably there are selected literals that prevent intersecting with most of the terms of the off-set of all F_1 ... F_m (i.e. yielding reduced terms that cover the least zeros). When no further reduction leads to any possible

improvement, the reduction is stopped and the implicant is recorded. A term that no longer intersects with the off-set of any F_i becomes its implicant. All implicants that were ever found are stored and output functions are assigned to them. Then simple dominance checks are performed in order to eliminate implicants that are dominated by another implicant in all functions F_i. Fig. 7.1 shows the typical growth of the number of group implicants (non-primes) as a function of the number of iterations. Here the function of 13 input variables, 13 output variables and 200 defined terms was used for demonstration. We can see that the number of implicants first rapidly grows, but then it falls to approx. 15 % of the maximum value. This is due the fact that new prime implicants are constantly produced and the most of group implicants are absorbed by them in the preliminary dominance checks.

When group implicants are generated, the **Group Covering Problem** is solved using the same heuristic as described in Section 8.

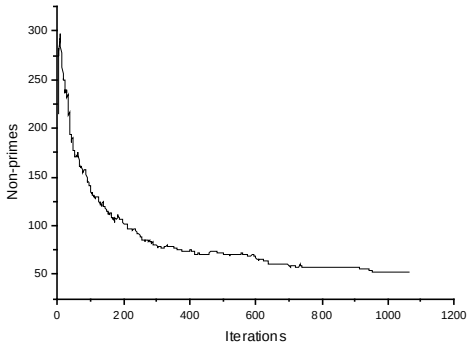

Fig. 7.1 *Growth of the number of non-primes*

7.2. Implicant Reduction Mutations

Not only the CD-search, but also implicant reduction can be enhanced by mutations. Here the mutation is a random selection of a literal that prevents intersecting a given term with at least one zero term. The number of group implicants (non-primes) grows very rapidly with increasing mutation rate – see Fig. 7.2. However, the experimental results show that the final result of minimization (the quality of the solution) depends on the rate of IR mutations only slightly. The group implicants that arise from the mutations are mostly not necessary for reaching the minimum solution. However, there may exist functions that require the use of mutations in order to find the minimum solution.

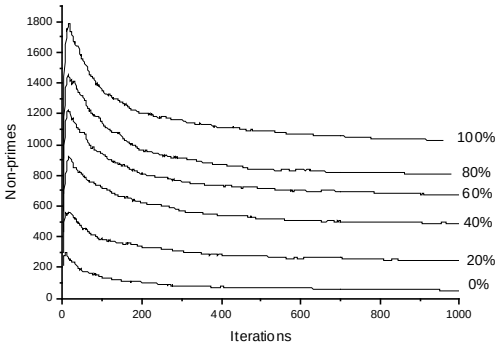

Fig. 7.2 *Growth of the number of non-primes for different IR mutation rates*

8. Covering Problem Solution

The covering problem can be solved either exactly, or by using some heuristics. The exact CP solving may be sometimes rather time-consuming, especially when it is performed after more iterations, when a large number of implicants needs to be processed. In this case, a heuristic approach is the only possible solution. We will describe three heuristic methods: LCMC cover, Contribution-based selection and Contribution-based removal.

8.1. LCMC Cover

LCMC (Least covered, most covering) algorithm is a common heuristic for the solution of covering problem. First, implicants covering minterms covered by the lowest number of other implicants are preferred. If there are more such implicants, implicants covering the highest number of yet uncovered 1-minterms are selected. From these primes we select the "shortest" ones, i.e., terms constructed of the lowest number of literals. When still more primes could be selected, we select one randomly.

8.2. Contribution-Based Techniques

More sophisticated heuristic methods for CP solution are based on computing the contributions (scoring functions) of terms as a criterion for their inclusion into the solution. Similar methods were also discussed in [Ser75] and [Rud89]. In the following text we will describe these methods as they are used in BOOM.

First, we will formulate some basic definitions:

Definition 8.1

Let $X=\{x_1, x_2, \ldots, x_r\}$ be a set of rows and $Y=\{y_1, y_2, \ldots, y_s\}$ a set of columns in a Boolean covering matrix A of dimensions r x s. An element $y_j \in Y$ covers an element $x_i \in X$ if A[i, j] = 1, otherwise A[i, j] = 0. The **unate set covering problem** consists in finding a minimum subset of Y that covers X.

Definition 8.2

The **strength of coverage SC** of the row x_i is defined as

$$\mathrm{SC}(x_i) = \frac{1}{\sum_{j=1}^{s} A[i, j]}$$

In other words: if the row x_i is covered by k columns, then the SC of the row x_i is equal to $1/k$. Hence the more ones are in one row, the weaker is the role of each of them. An essential row has SC = 1.

Definition 8.3

The **column contribution** (CC) of the column y_j is defined as

$$CC(y_j) = \sum_{i=1}^{r} A[i, j] \cdot SC(x_i)$$

CC expresses the importance of a column, because it summarizes the strengths of all ones that it contains.

Example 8.1

The following example shows the calculation of SC and CC (zeros in the matrix are omitted).

Tab. 8.1 *Contributions example*

	y_1	y_2	y_3	y_4	y_5	y_6	*SC*
x_1	1	1	1		1		1/4=0.25
x_2	1	1		1			1/3=0.33
x_3	1				1		1/2=0.50
x_4			1	1			1/2=0.50
x_5				1			1/1=1.00
x_6					1	1	1/2=0.50
x_7	1					1	1/2=0.50
CC	1.58	0.58	0.75	**1.83**	1.25	1.00	

8.3. Contribution-Based Selection

This solution method uses the CC values as a criterion for the selection of columns to be included into the final solution. First the contributions are evaluated and the column with the highest value is selected. After the selection this column is removed from the matrix together with all rows it covers. Then all values are recomputed and again the column with the highest CC value is selected. This is repeated until all rows are covered. When there are more columns with equal maximal CC values, one is selected at random.

Note that this method is used not only for a heuristic cyclic core solution (like e.g. in [Rud89]), but for solution of the whole CP. In most cases the stand-alone algorithm is capable to resolve the row and column dominances and, if properly implemented, this is done even faster than when using standard algorithms (see the Recomputing of contributions).

Example 8.2

In Example 8.1 the column y_4 has the highest CC value and is thus selected. The matrix is reduced to:

Tab. 8.2

	y_1	y_2	y_3	y_4	y_5	y_6	*SC*
x_1	1	1	1		1		1/4=0.25
x_3	1				1		1/2=0.50
x_6					1	1	1/2=0.50
x_7	1					1	1/2=0.50
CC	1.25	0.25	0.25		1.25	1.00	

Columns y_1 and y_5 have now the same CC value, one (y_1) is selected at random:

Tab. 8.3

	y_1	y_2	y_3	y_4	y_5	y_6	*SC*
x_6					1	1	1/2
CC	0	0	0	0	0.50	0.50	

Again, both remaining columns have an equal CC value, one has to be selected at random *(y_5)*. The final solution is $Y=\{ y_1, y_4, y_5\}$.

8.4. Recomputing of contributions

After every column selection the column contributions must be recomputed in order to select the next column with the highest contribution. However, we will show that not all row and column contribution values need to be recomputed because some are not affected by the reduction of the covering matrix. Values that are to be modified can be recomputed on-line, when a column is being selected.

Let us consider the first move in Example 8.1:

Tab. 8.4

	y_1	y_2	y_3	y_4	y_5	y_6	*SC*
x_1	1	1	1		1		1/4=0.25
x_2	1	1		1			1/3=0.33
x_3	1				1		1/2=0.50
x_4			1	1			1/2=0.50
x_5				1			1/1=1.00
x_6					1	1	1/2=0.50
x_7	1					1	1/2=0.50
CC	1.58	0.58	0.75	1.83	1.25	1.00	

	y_1	y_2	y_3	y_4	y_5	y_6	*SC*
x_1	1	1	1		1		1/4=0.25
x_3	1				1		1/2=0.50
x_6					1	1	1/2=0.50
x_7	1					1	1/2=0.50
CC	1.25	0.25	0.25		1.25	1.00	

Row strengths are not affected by the selection, because all rows whose values should be modified are removed from the matrix (shaded dark). This removal causes that only CC values of columns covering these rows must be modified (light shading). Let *X'* be a set of rows covered by the column *y*. After *y* is selected all column contributions have to be modified in the following way:

$$\forall j : CC(y_j) = CC(y_j) - \sum_{i}^{x_i \in X'} A[i, j] \cdot SC(x_i)$$

For columns, that do not cover removed rows, all $A[i, j]$ values are equal to zero, thus the CC remains unchanged.

8.5. Contribution-Based Removal

A modification of the above method is the use of the CC value for elimination, instead of selection of columns. Then the terms (columns) with the lowest CC value would be eliminated from the table, until the irredundant cover of the on-set is reached.

9. Experimental Results

Extensive experimental work was done to evaluate the efficiency of the proposed algorithm, especially for problems of large dimensions. Both runtime in seconds and result quality were evaluated. The processor used was a Celeron 433 MHz with 160 MB RAM. The quality of the results was measured by three parameters: total number of literals, output cost, and number of product terms (implicants). However, one of them had to be chosen as a minimization criterion. Several groups of experiments, listed in the following subsections, were performed.

9.1. Standard MCNC Benchmarks

A set of 123 standard MCMC benchmarks [Esp2] was solved by ESPRESSO, ESPRESSO-EXACT and BOOM. The so-called "hard" benchmarks were treated separately and the results are presented in Subsection 9.2. Of the 123 standard problems, 51.22 % were solved by BOOM in a shorter time than ESPRESSO (in Tab. 9.1 the shaded values in column "time"). In all cases only one iteration of BOOM was used. In 45.52 % of problems BOOM gave the same result as ESPRESSO (in column "lit/out/terms" the shaded entries). In one case (dc2) we found even a better solution than ESPRESSO or ESPRESSO-EXACT. In 30.89 % these results were reached faster than by ESPRESSO. It is also worth mentioning, that in 28 cases the BOOM runtime was non-measurable and the timer inserted a default value of .01 sec. The benchmarks were also solved by ESPRESSO-EXACT in order to obtain the minimum solution for comparison. Note that in this case the minimality criterion is only the number of terms and thus some "exact" solutions are worse than those reached by ESPRESSO or BOOM. Some benchmarks were not solved by ESPRESSO-EXACT because of its extremely long runtimes (blank entries in Tab. 9.1).

ESPRESSO solutions that are equal to the exact ones are shaded in the ESPRESSO column. In BOOM column there are shaded run times that are shorter than ESPRESSO times and solutions that are equal or better than solutions obtained by ESPRESSO.

Tab. 9.1 *Runtimes and minimal solutions for the standard MCNC benchmarks*

		ESPRESSO		ESPRESSO-EXACT		BOOM – 1it.	
bench	i/o/p	time	lit/out/terms	time	lit/out/terms	time	lit/out/terms
5xp1	7/10/141	0,10	260/87/65	0,23	263/97/63	0,06	260/87/65
9sym	9/1/158	0,12	516/86/86	0,12	516/86/86	0,05	516/86/86
al2	16/47/139	0,15	324/103/66	41,3	324/103/66	0,95	324/103/66
alcom	15/38/90	0,13	174/49/40	11,0	174/49/40	0,30	177/45/42
alu1	12/8/39	0,10	41/19/19	0,52	41/19/19	0,01	41/19/19
alu2	10/8/241	0,20	268/79/68	0,45	268/79/68	0,06	268/79/68
alu3	10/8/273	0,19	279/70/65	0,54	278/74/64	0,07	279/68/66
alu4	14/8/1184	1,63	4443/644/575	39,9	4495/648/575	24,6	4443/644/575
amd	14/24/272	0,16	442/212/66	0,47	443/214/66	1,12	480/194/74
apex1	45/45/1440	0,69	1739/1103/206	118	1742/1108/206	146,00	1850/1084/219
apex2	39/3/1576	5,90	14453/1075/1035	301	14453/1075/1035	29,4	14489/1065/1041
apex3	54/50/1036	1,49	2270/1020/280	6,12	2284/1023/280	115,00	2433/867/314
apex4	9/19/1907	2,98	3684/1736/435	3,09	3646/1804/427	36,5	4270/1424/532
apex5	117/88/2710	15,2	6089/1192/1088			2940	6089/1192/1088
apla	10/12/119	0,14	163/58/25	0,45	163/58/25	0,04	187/49/31
b10	15/11/333	0,19	818/182/100	1,29	821/181/100	0,60	837/182/103
b11	8/31/74	0,11	122/59/27	0,13	122/59/27	0,11	124/52/28
b12	15/9/72	0,13	149/59/42	2,67	158/75/41	0,03	148/58/43
b2	16/7/509	0,25	972/969/106	1,20	967/1003/104	3,68	1191/825/136
b3	32/20/621	0,55	2120/392/211	11,7	2117/390/210	7,37	2146/361/216
b4	33/23/680	0,22	437/109/54	22,7	437/109/54	6,51	471/103/58
b7	8/31/74	0,11	122/59/27	0,13	122/59/27	0,10	124/52/28
b9	16/5/292	0,18	754/119/119	2,54	754/119/119	0,22	754/119/119
br1	12/8/107	0,12	206/48/19	0,13	206/48/19	0,02	206/48/19
br2	12/8/83	0,11	134/38/13	0,14	134/38/13	0,01	134/38/13
bw	5/28/93	0,15	102/246/22	0,23	102/251/22	0,19	114/203/27
clip	9/5/271	0,16	630/162/120	0,53	614/155/117	0,15	548/161/127
clpl	11/5/40	0,12	55/20/20	0,15	55/20/20	0,01	55/20/20
con1	7/2/18	0,10	23/9/9	0,13	23/9/9	0,01	23/9/9
cordic	23/2/2105	5,19	13825/914/914	9,39	13843/914/914	22,6	13825/914/914
cps	24/109/855	0,90	1890/946/163	7,69	1884/983/158	194,00	1950/860/171
dc1	4/7/25	0,12	27/27/9	0,13	27/27/9	0,01	27/27/9
dc2	8/7/101	0,13	207/52/39	0,15	208/52/39	0,03	206/51/39
dekoder	4/7/20	0,11	18/29/9	0,12	19/29/9	0,01	18/26/10
dist	8/5/314	0,20	709/160/123	0,28	710/169/120	0,16	710/160/123
dk17	10/11/59	0,13	103/32/18	0,30	103/32/18	0,01	114/30/21
dk27	9/9/24	0,10	31/15/10	0,23	31/15/10	0,01	31/15/10
dk48	15/17/64	0,24	115/28/22	1,73	115/28/22	0,03	115/28/22
duke2	22/29/404	0,18	751/245/86	0,90	759/256/86	2,72	796/241/90
e64	65/65/327	0,27	2145/65/65	0,25	2145/65/65	26,6	2145/65/65
ex5	8/63/208	0,90	373/884/72			5,97	422/493/96
ex7	16/5/292	0,19	754/119/119	2,52	754/119/119	0,23	754/119/119
exep	30/63/643	0,52	1175/110/110	2,11	1170/108/108	15,00	1175/110/110
exp	8/18/317	0,18	405/153/59	0,28	400/175/57	0,36	432/141/66
exps	8/38/851	0,51	961/947/133	0,96	961/936/132	17,4	1063/748/153
f51m	8/8/178	0,14	323/77/77	0,28	326/76/76	0,06	323/77/77
gary	15/11/372	0,22	896/221/107	0,53	899/218/107	0,75	927/218/113
check	4/1/5	0,09	2/1/1	0,10	2/1/1	0,01	2/1/1
chkn	29/7/370	0,26	1598/141/140	0,62	1602/142/140	0,97	1598/141/140
in0	15/11/376	0,21	896/221/107	0,53	899/218/107	0,78	928/218/113
in1	16/17/509	0,29	972/969/106	1,29	967/1003/104	4,61	1137/858/129
in2	19/10/399	0,25	1169/251/136	0,53	1172/257/134	1,14	1172/247/137
in3	35/29/341	0,18	508/263/74	2,04	508/263/74	5,31	544/257/82
in4	32/20/642	0,53	2183/413/213	10,1	2162/411/212	9,78	2233/384/219

		ESPRESSO		ESPRESSO-EXACT		BOOM – 1it.	
bench	i/o/p	time	lit/out/terms	time	lit/out/terms	time	lit/out/terms
in5	24/14/348	0,15	533/208/62	0,84	533/208/62	0,71	580/204/67
in6	33/23/317	0,15	437/110/54	14,6	437/110/54	1,46	458/104/57
in7	26/10/142	0,14	337/90/54	3,43	337/90/54	0,20	337/90/54
inc	7/9/94	0,09	136/61/30	0,14	133/62/29	0,03	137/58/31
intb	15/7/1190	1,58	5260/631/631	28,4	5282/629/629	4,29	5262/631/631
luc	8/27/126	0,14	130/238/26	0,27	130/238/26	0,24	139/199/29
m1	6/12/53	0,10	99/118/19	0,15	99/118/19	0,03	99/69/22
m2	8/16/163	0,19	319/320/47	0,23	322/337/47	0,33	338/259/52
m3	8/16/201	0,20	426/385/64	0,28	419/412/62	0,41	487/244/81
m4	8/16/329	0,30	640/518/105	0,86	634/580/101	1,26	663/461/113
mark1	20/31/72	0,59	97/57/19	3,59	97/57/19	0,09	92/52/22
max1024	10/6/744	0,65	1874/383/273			0,97	1932/373/286
max128	7/24/256	0,31	451/632/80	0,49	450/714/78	1,18	493/411/97
max46	9/1/155	0,14	395/46/46	0,15	395/46/46	0,02	395/46/46
max512	9/6/382	0,27	869/213/145	0,57	816/216/133	0,25	880/212/148
misex1	8/7/41	0,12	51/45/12	0,12	51/45/12	0,01	51/45/12
misex2	25/18/101	0,13	183/30/28	0,15	183/30/28	0,19	183/30/28
misex3	14/14/1391	4,97	6380/1288/676			19,8	6956/1289/745
misex3c	14/14/1566	2,39	1306/253/197			3,26	1327/252/207
mlp4	8/8/373	0,21	736/156/128	1,76	709/160/121	0,27	764/154/136
mp2d	14/14/114	0,16	122/76/31	1,44	123/78/30	0,09	122/76/34
newapla	12/10/60	0,10	74/28/17	0,12	74/28/17	0,02	74/28/17
newapla1	12/7/25	0,10	64/12/10	0,10	64/12/10	0,01	64/12/10
newapla2	6/7/22	0,09	42/7/7	0,11	42/7/7	0,01	42/7/7
newbyte	5/8/16	0,09	40/8/8	0,11	40/8/8	0,01	40/8/8
newcond	11/2/72	0,11	208/31/31	0,12	208/31/31	0,01	208/31/31
newcpla1	9/16/90	0,12	201/63/38	0,18	204/69/38	0,07	203/62/40
newcpla2	7/10/62	0,11	87/42/19	0,11	87/42/19	0,01	91/42/20
newcwp	4/5/24	0,10	31/19/11	0,10	31/19/11	0,01	31/19/11
newill	8/1/18	0,13	42/8/8	0,12	42/8/8	0,01	42/8/8
newtag	8/1/12	0,16	18/8/8	0,13	18/8/8	0,01	18/8/8
newtpla	15/5/63	0,15	176/23/23	0,14	176/23/23	0,01	176/23/23
newtpla1	10/2/15	0,16	33/4/4	0,13	33/4/4	0,01	33/4/4
newtpla2	10/4/26	0,19	54/15/9	0,13	54/15/9	0,01	54/15/9
newxcpla1	9/23/93	0,17	197/86/39	0,19	201/108/39	0,14	200/75/41
opa	17/69/382	0,36	559/540/79	0,76	554/559/77	13,20	581/487/85
p82	5/14/74	0,17	93/56/21	0,13	93/56/21	0,03	95/54/22
prom2	9/21/1470	3,53	2572/2954/287			51,00	3249/1705/407
rd53	5/3/67	0,09	140/35/31	0,12	140/35/31	0,01	140/35/31
rd73	7/3/274	0,14	756/147/127	0,17	756/147/127	0,10	756/147/127
rd84	8/4/511	0,35	1774/296/255	0,37	1774/296/255	0,51	1774/296/255
risc	8/31/71	0,12	129/53/29	0,11	127/60/28	0,09	129/53/29
root	8/5/157	0,14	297/88/57	0,15	300/76/57	0,05	308/77/61
ryy6	16/1/119	0,15	624/112/112	0,11	624/112/112	0,10	624/112/112
sao2	10/4/137	0,11	421/75/58	0,19	420/75/58	0,04	421/75/58
seq	41/35/2014	1,21	4369/1876/336	74,8	4343/1877/334	266,00	4687/1791/368
sex	9/14/61	0,12	69/36/21	0,11	69/36/21	0,02	69/36/21
spla	16/46/837	1,61	2558/643/251	31,6	1564/450/181	17,5	2793/543/280
sqn	7/3/95	0,10	184/46/38	0,12	184/45/38	0,01	184/46/38
sqr6	6/12/119	0,11	195/65/49	0,29	199/75/47	0,06	199/64/50
sqrt8	8/4/66	0,11	144/44/38	0,13	151/44/38	0,01	144/44/38
squar5	5/8/65	0,12	87/32/25	0,13	88/32/25	0,01	88/30/26
t1	21/23/211	0,26	449/163/102	178	478/176/101	0,80	449/163/102
t2	17/16/183	0,14	286/74/53	0,39	285/73/53	0,27	290/74/54
t3	12/8/97	0,11	217/33/33	0,12	218/33/33	0,03	217/33/33
t4	12/8/55	0,12	61/28/16	0,34	61/28/16	0,01	65/23/19

		ESPRESSO		ESPRESSO-EXACT		BOOM – 1it.	
bench	i/o/p	time	lit/out/terms	time	lit/out/terms	time	lit/out/terms
t481	16/1/841	0,59	4752/481/481	0,51	4752/481/481	3,48	4752/481/481
table3	14/14/1686	0,51	2001/642/175	0,62	2001/642/175	10,5	2069/622/182
table5	17/15/1600	0,39	1895/606/158	0,55	1896/607/158	12,5	1944/603/163
tms	8/16/119	0,20	217/248/30	0,22	208/204/30	0,17	213/170/32
vg2	25/8/304	0,15	804/110/110	0,98	804/110/110	0,54	804/110/110
vtx1	27/6/305	0,17	964/110/110	0,88	964/110/110	0,45	964/110/110
wim	4/7/22	0,11	18/25/9	0,11	18/29/9	0,01	19/24/11
x1dn	27/6/305	0,17	964/110/110	0,88	964/110/110	0,45	964/110/110
x6dn	39/5/310	0,17	641/177/82	0,85	632/186/81	0,29	663/177/85
x9dn	27/7/315	0,20	1138/120/120	1,04	1138/120/120	0,53	1138/120/120
xor5	5/1/32	0,08	80/16/16	0,12	80/16/16	0,01	80/16/16
z5xp1	7/10/159	0,12	262/97/63	0,16	263/97/63	0,08	261/89/65
z9sym	9/1/72	0,09	226/34/34	0,14	226/34/34	0,01	226/34/34

9.2. Hard MCNC Benchmarks

The set of 19 "hard" MCNC benchmarks was solved by BOOM and ESPRESSO. The common property of these problems is a large number of PIs, hence, the minimum solution is really hard to find. Moreover, the number of output variables and/or terms is also high and thus they are very difficult for BOOM to handle, hence the BOOM runtimes are in all but one case longer. Tab. 9.2 shows that for 10 problems BOOM found the same solution as ESPRESSO, once in a shorter time (shaded cells), 4 problems gave slightly worse solutions and 5 problems could not be solved because of high memory demands. This is due to the high number of terms, because for BOOM the runtime (and memory demand) grows with the square of the number of terms - see Section 10.

Tab. 9.2 *Hard MCNC Benchmarks*

		ESPRESSO		BOOM – 1 iteration	
bench	i/o/p	time	lit/out/terms	Time	lit/out/terms
ex1010	10/10/1304	1,47	1982/758/283	5,91	2422/734/370
ex4	128/28/654	1,45	1649/279/279	32,3	1649/279/279
ibm	48/17/499	0,24	882/173/173	2,81	882/173/173
jbp	36/57/402	0,41	807/220/122	11,9	838/202/131
mainpla	27/54/2507	1,91	2338/6383/172		
misg	56/23/120	0,23	172/75/69	0,58	172/75/69
mish	94/43/158	0,25	147/91/82	3,71	147/91/82
misj	35/14/55	0,11	54/48/35	0,08	54/48/35
pdc	16/40/822	2,26	828/432/136	7,33	901/328/167
shift	19/16/200	0,12	388/105/100	0,32	388/105/100
signet	39/8/3627	1,09	490/146/119	4,67	490/146/119
soar	83/94/779	2,58	2454/549/353		
test2	11/35/8181	82,3	9936/5686/1097		
test3	10/35/3970	13,1	4139/2484/541		
ti	47/72/839	0,99	1882/741/214	117	1975/696/226
ts10	22/16/262	0,16	896/128/128	0,81	896/128/128
x2dn	82/56/345	0,33	528/119/105	15,40	528/119/105
x7dn	66/15/1456	1,60	4128/539/539	17,70	4128/539/539
xparc	41/73/3226	3,28	4813/2653/254		

9.3. Test Problems with *n*>50

The MCNC benchmarks have relatively few input terms and few input variables (only for 3 standard benchmarks does *n* exceed 50) and then also have a small number of don't care terms. To compare the performance and result quality achieved by the minimization programs on larger problems, a set of problems with up to 300 input variables and up to 300 minterms was solved. The truth tables were generated by a random number generator, for which only the number of input variables, number of care terms and number of don't cares in the input portion of the truth table were specified. The number of outputs was set equal to 5. The on-set and off-set of each function were kept approximately of the same size. For each problem size ten different samples were generated and solved and average values of the ten solutions were computed.

First the minimality of the result was compared. BOOM was always run iteratively, using **the same total runtime** as ESPRESSO needed for one pass. The quality criterion selected for BOOM was the sum of the number of literals and the output cost to match the criterion used by ESPRESSO. The first row of each cell in Tab. 9.3 contains the BOOM results, the second row shows the ESPRESSO results. We can see that in most cases BOOM found a better solution than ESPRESSO. The missing ESPRESSO results in the lower right-hand corner indicate the problems for which ESPRESSO could not be used because of the long runtimes. Hence only one iteration of BOOM was performed, and its duration in seconds is given as a last value.

Tab. 9.3 *Solution of problems with* n*>50 - comparing the result quality*

p/n	60	100	140	180	220	260	300
20	22/12/9(67) 23/15/9/1.01	18/11/8(96) 23/13/8/1.95	18/10/8(127) 22/14/8/3.47	16/10/8(161) 19/13/8/5.59	17/10/8(201) 19/12/7/9.52	16/10/8(219) 18/11/7/12.03	16/9/8(262) 20/12/8/16.44
60	76/29/22(54) 86/40/21/6.54	68/24/20(77) 75/34/19/14.26	65/22/19(127) 73/34/19/28.99	61/21/19(151) 68/30/17/42.46	58/21/17(183) 62/28/16/57.53	56/20/17(218) 64/29/17/78.68	55/19/17(271) 65/27/17/111.62
100	143/42/35(45) 150/61/33/13.85	127/38/32(74) 133/55/29/41.26	118/36/30(100) 127/52/28/69.02	110/32/28(157) 121/46/27/124.22	108/31/28(162) 116/46/26/152.44	105/31/27(215) 116/45/26/248.67	102/30/27(260) 112/44/25/328.37
140	206/56/47(46) 215/80/43/28.70	190/50/44(70) 191/72/39/71.23	177/46/41(94) 177/66/36/129.66	165/44/39(127) 171/63/36/206.76	159/44/37(160) 164/60/34/273.15	154/39/36(210) 164/60/33/452.93	149/40/36(231) 156/55/32/516.63
180	288/70/61(45) 284/101/54/48.70	251/61/54(79) 253/92/48/141.97	230/56/51(111) 233/84/44/261.95	220/55/49(139) 228/80/44/397.36	209/50/46(181) 220/77/42/630.53	255/49/48/1.36 -	250/48/48/1.60 -
220	363/85/72(48) 352/120/63/80.68	310/74/64(88) 310/109/57/256.40	291/68/60(118) 290/103/53/392.86	273/65/57(146) 285/98/52/632.04	329/61/60/1.79 -	320/60/59/2.09 -	308/58/57/2.43 -
260	436/98/84(46) 427/144/74/108.50	374/84/74(87) 382/124/67/336.32	353/81/70(116) 348/119/61/580.84	420/75/73/2.15 -	398/71/70/2.55 -	391/70/69/2.98 -	372/66/65/3.39 -
300	521/109/96(40) 489/160/83/120.69	450/97/87(81) 447/149/75/427.72	422/88/81(107) 416/139/71/719.54	493/84/8/32.88 -	469/80/79/3.48 -	449/77/77/3.91 -	441/77/75/4.75 -

Entry format: BOOM: #of literals/output cost/#of implicants(# of iterations).

ESPRESSO: #of literals/output cost/#of implicants/time in seconds

A second group of experiments for n>50 was performed to compare the runtimes. Again random problems were solved, but this time BOOM was running until a solution of **the same or better quality** was reached. The quality criterion selected was the sum of the number of literals and the output cost. The results given in Tab. 9.4 show that for all samples the same or better solution was found by BOOM in much shorter time than by ESPRESSO (up to 300 times faster).

Tab. 9.4 *Solution of problems with n>50 - comparing the runtime*

p/n	50	100	150	200	250	300
50	111/0.06 (1) 132/5.71	92/0.08 (1) 92/7.15	83/0.12 (1) 84/20.00	77/0.59 (4) 88/42.77	77/0.39 (2) 77/51.29	75/8.69 (35) 76/110.74
100	219/2.36 (9) 220/7.38	190/2.57 (7) 190/27.95	174/4.19 (9) 176/104.38	163/31.05 (35) 165/114.65	155/14.74 (19) 158/184.31	154/1.40 (2) 154/317.39
150	330/2.34 (4) 334/21.42	287/9.44 (10) 287/79.47	289/1.11 (1) 289/129.20	249/31.23 (20) 253/367.19	231/57.38 (29) 233/396.01	247/44.66 (19) 248/569.44
200	338/20.26 (11) 447/55.24	401/37.79 (15) 404/209.27	349/91.96 (25) 350/297.20	344/63.23 (20) 347/557.54	331/2.27 (1) 334/794.97	321/2.89 (1) 328/857.19
250	576/32.38 (9) 576/80.27	460/242.27 (36) 463/323.27	443/142.71 (23) 450/404.09	409/481.63 (50) 445/934.13	423/196.56 (27) 425/1607.45	385/507.23 (52) 389/2354.24
300	594/83.35 (13) 597/105.20	580/203.06 (22) 588/333.90	505/446.42 (38) 508/798.84	506/416.01 (34) 512/847.05	500/470.90 (38) 500/1822.01	465/205.76 (32) 466/3012.90

Entry format: BOOM: #of literals+output cost / time in seconds (# of iterations)

ESPRESSO: #of literals+output cost / time in seconds

9.4. Solution of Very Large Problems

A third group of experiments aims at establishing the limits of applicability of BOOM. For this purpose, a set of large test problems was generated and solved by BOOM. For each problem size (# of variables, # of terms) 10 different problems were generated and solved. Each problem was a group of 10 output functions. For problems with more than 300 input variables ESPRESSO cannot be used at all. Hence when investigating the limits of applicability of BOOM, it was not possible to verify the results by any other method. The results of this test are listed in Tab. 9.5, where the average time in seconds needed to complete one iteration for various problem sizes is shown. We can see that a problem with 1000 input variables, 10 outputs and 2000 care minterms was solved by BOOM in less than 5 minutes.

Tab. 9.5 *Time for one iteration on very large problems*

p/n	200	400	600	800	1000
200	0.21	0.38	0.55	0.90	1.06
400	0.98	1.90	3.30	4.84	5.96
600	2.48	4.73	6.94	11.52	18.10
800	4.89	9.76	14.56	24.06	38.58
1000	8.34	15.51	27.88	48.85	74.29
1200	17.64	29.66	42.15	58.37	64.18
1400	23.72	41.49	58.58	74.09	106.65
1600	36.05	73.43	104.90	118.98	161.42
1800	49.53	95.78	146.28	178.29	210.99
2000	60.62	118.39	206.44	204.16	288.87

9.5. Graph Coloring Problem

The graph coloring problem can be formulated using Boolean functions, as was shown in [Ost74]: a Boolean function has as many input variables as there are regions in the graph, each variable representing one region. The on-set specifies the areas that can share the same color, while the off-set defines the neighboring areas that cannot be colored by the same color.

The minimal cover of this function corresponds to the minimal number of colors needed for coloring the graph.

An example with 14 regions presented in [Ost74] was solved by BOOM, ESPRESSO and ESPRESSO-EXACT. The on-set consists of 14 terms, and the off-set consists of 33 terms. The results listed in Tab. 9.6 show that ESPRESSO-EXACT reached the minimum of 4 terms, while ESPRESSO could not find the minimum solution. BOOM found the minimum solution in the shortest time.

Tab. 9.6 *Solutions of the 4-color problem*

Method	terms	time [s]
ESPRESSO-EXACT	4	0.22
ESPRESSO	5	0.17
BOOM	4	non-measurable

9.6. Minimization of a Symmetric Function

Symmetric functions are notoriously difficult to minimize. The S^9_{3456} function was used in [Hon74] to test the minimization procedure. This function has 420 minterms and 1680 prime implicants. The minimum two-level solution consists of 84 implicants. This result was also obtained by application of BOOM in about 9 seconds, whereas ESPRESSO found a non-minimal solution with 86 implicants in 0.5 second. ESPRESSO-EXACT found the minimum solution in 5 seconds.

10. Time Complexity Evaluation

As for most heuristic and iterative algorithms, it is difficult to evaluate the time complexity of the proposed algorithm exactly. We have observed the average time needed to complete one pass of the algorithm for various sizes of input truth table. The truth tables were generated randomly, following the same rules as in the previous case. Fig. 10.1 shows the growth of an average runtime as a function of the number of care minterms (20-300) where the number of input variables is changed as a parameter (20-300). The curves in Fig. 10.1 can be approximated with the square of the number of care minterms. Fig. 10.2 shows the runtime growth depending on the number of input variables (20-300) for various numbers of defined minterms (20-300). Although there are some fluctuations due to the low number of samples, the time complexity is almost linear. Fig. 10.3 shows a three-dimensional representation of the above curves.

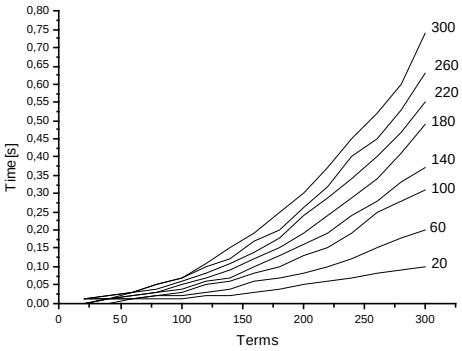

Fig. 10.1 *Time complexity (1)*

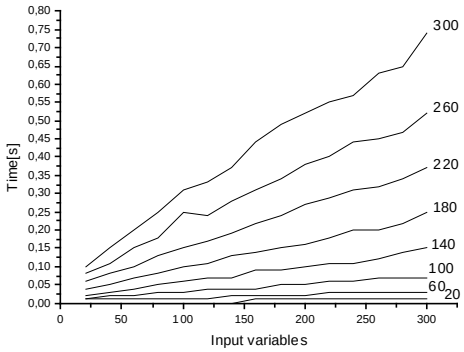

Fig. 10.2 *Time complexity (2)*

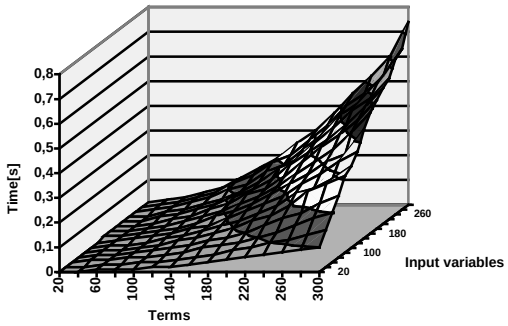

Fig. 10.3 *Time complexity (3)*

11. The BOOM Program

The BOOM minimizer has been placed on a web page [6], from where it can be downloaded by anybody who wants to use it.

11.1. Program Description

The BOOM program is a tool for minimizing two-valued Boolean functions. The output is a near-minimal or minimal two-level disjunctive form. The input and output formats are compatible with Berkeley standard PLA format that is described in Section 11.2.

BOOM runs as a Win32 console application with the following command line syntax:

```
BOOM [options] [source] [destination]
```

Tab. 11.1 *BOOM options*

-CM*n*	Define CD-search mutations ratio *n* (0-100)
-RM*n*	Define implicant reduction mutations ratio *n* (0-100)
-E*x*	Select implicant expansion type: 0: Sequential search 1: Distributed multiple IE 2: Distributed exhaustive IE 3: Multiple IE (default) 4: Exhaustive IE
-CP*x*	Select the CP solution algorithm: 0: LCMC 1: Contribution-based selection (default) 2: Contribution-based removal 3: Exact
-S*xn*	Define stopping criterion *x* of value *n*: t: stop after *n* seconds (floating point number is expected) i: stop after *n* iterations (default is Si1) n: stopping interval equal to *n* the minimization is stopped when there is no improvement of the solution for *n*-times more iterations than it was needed for the last improvement q: stop when the quality of solution meets *n* more criteria can be specified at the same time
-Q*x*	Define quality criterion *x*: t: number of terms l: number of literals o: output cost b: number of literals+output cost (default)
-endcov	Solve CP only at the end of minimization
-c	Checks the input function for consistence, i.e., checks if the off-set doesn't intersect the on-set.

11.2. PLA Format

Input to the BOOM system, as well as its output, has the format of a two-level SOP expression. This is described as a character matrix (truth table) with keywords embedded in the input to specify the size of the matrix and the logical format of the input function.

The following keywords are recognized by BOOM. The list shows the probable order of the keywords in a PLA description. The symbol *d* denotes a decimal number and *s* denotes a text string. The minimum required set of keywords is .i, .o and .e. Both keywords .i and .o must precede the truth table.

***Tab. 11.2** Keywords in PLA format*

.i *d*	Specifies the number of input variables (necessary)
.o *d*	Specifies the number of output functions (necessary)
.ilb *s1 s2 . . . sn*	Gives the names of the binary valued variables. This must come after .i. There must be as many tokens following the keyword as there are input variables
.ob *s1 s2 . . . sn*	Gives the names of the output functions. This must come after .o. There must be as many tokens following the keyword as there are output variables
.type *s*	Sets the logical interpretation of the character matrix. This keyword (if present) must come before any product terms. *s* is either fr or fd (which is default)
.p *d*	Specifies the number of product terms
.e (.end)	Marks the end of the PLA description

11.3. Logical Description of a PLA

When we speak of the ON-set of a Boolean function, we mean those minterms, which imply the function value is a 1. Likewise, the OFF-set are those terms which imply the function is a 0, and the DC-set (don't care set) are those terms for which the function is unspecified. A function is completely described by providing its ON-set, OFF-set and DC-set. Note that all minterms lie in the union of the ON-set, OFF-set and DC-set, and that the ON-set, OFF-set and DC-set share no minterms.

A Boolean function can be described in one of the following ways:

- By providing the ON-set. In this case the OFF-set can be computed as the complement of the ON-set and the DC-set is empty.
- By providing the ON-set and DC-set. The OFF-set can be computed as the complement of the union of the ON-set and the DC-set. This is indicated with the keyword **.type fd** in the PLA file. This Boolean function specification is used by BOOM as the output of the minimization algorithm.
- By providing the ON-set and OFF-set. In this case the DC-set can be computed as the complement of the union of the ON-set and the OFF-set. It is an error for any minterm to belong to both the ON-set and OFF-set. This error may not be detected during the minimization, but it can be checked with the "consistency check" option. This type is indicated with the keyword **.type fr** in the input file. This is the only possible Boolean function specification for the input to BOOM.

12.4. Symbols in the PLA Matrix and Their Interpretation

Each position in the input plane corresponds to an input variable where a 0 implies that the corresponding input literal appears complemented in the product term, a 1 implies that the input literal appears uncomplemented in the product term, and - implies the input literal does not appear in the product term.

With **.type fd** (default option), for each output, a 1 means this product term belongs to the ON-set, a 0 means this product term has no meaning for the value of this function.

With **.type fr**, for each output, a 1 means this product term belongs to the ON-set, a 0 means this product term belongs to the OFF-set, and a - means this product term has no meaning for the value of this function.

Regardless of the type of PLA, a ~ implies the product term has no meaning for the value of this function.

Example

A two-bit adder, which takes in two 2-bit operands and produces a 3-bit result, can be completely described with minterms as:

Tab. 11.3

```
.i 4
.o 3
.p 16
0000 000
0001 001
0010 010
0011 011
0100 001
0101 010
0110 011
0111 100
1000 010
1001 011
1010 100
1011 101
1100 011
1101 100
1110 101
1111 110
.e
```

Note that BOOM **does not** accept all features of the current Berkeley PLA format. When any features of this format not described here are used, they are ignored or an error is returned.

12 Conclusions

An original Boolean minimization method has been presented. Its most important features are its applicability to functions with several hundreds of input variables and very short minimization times for sparse functions. The function to be minimized is defined by its on-set and off-set, whereas the don't care set need not be specified explicitly. The entries in the truth table may be minterms or terms of higher dimensions. The implicants of the function are constructed by reduction of *n*-dimensional cubes; hence the terms contained in the original truth table are not used as a basis for the final solution.

The properties of the BOOM minimization tool were demonstrated on examples. Its application is advantageous above all for problems with large dimensions and a large number of don't care states where it beats other methods, like ESPRESSO, both in minimality of the result and in runtime. The PI generation method is very fast, hence it can easily be used in an

iterative manner. However, for sparse functions it mostly finds the minimum solution in a single iteration. Thus, e.g., for more than one fifth of the MCNC standard benchmark problems the runtime needed to find the minimum solution on an ordinary PC was less than 0.01 sec., and in more than a half of the cases the solution was found faster than by ESPRESSO. Among the hard benchmarks, BOOM found the minimum for only one half of the problems. Random problems with more than 100 input variables were in all cases solved faster and mostly with better results than by ESPRESSO. The dimension of the problems solved by BOOM can easily be increased over 1000, because the runtime grows linearly with the number of input variables. For problems of very high dimension, success largely depends on the size of the care set. This is due to the fact that the runtime grows roughly with the square of the size of the care set.

BOOM Publications

So far, the BOOM algorithm or some of its specific features have been published in the following papers

[1] Hlavička, J. - Fišer, P.: Algorithm for Minimization of Partial Boolean Functions, Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems (DDECS'00) Workshop, Smolenice (Slovakia), 5-7.4.2000, pp.130-133

[2] Fišer, P. - Hlavička, J.: Efficient Minimization Method for Incompletely Defined Boolean Functions, Proc. 4th Int. Workshop on Boolean Problems, Freiberg (Germany), Sept. 21-22, 2000, pp. 91-98

[3] Fišer, P. - Hlavička, J.: Implicant Expansion Method used in the BOOM Minimizer. Proc. IEEE Design and Diagnostics of Electronic Circuits and Systems Workshop (DDECS'01), Gyor (Hungary), 18-20.4.2001, pp.291-298

[4] Hlavička, J. - Fišer, P.: A Heuristic method of two-level logic synthesis. Proc. The 5th World Multiconference on Systemics, Cybernetics and Informatics SCI'2001, Orlando, Florida (USA) 22-25.7.2001 (in print)

[5] Fišer, P. - Hlavička, J.: On the Use of Mutations in Boolean Minimization. Proc. Euromicro Symposium on Digital Systems Design, Warsaw (Poland) 4-6.9.2001 (in print)

[6] http://cs.felk.cvut.cz/~fiserp/boom/

References

[Are78] Arevalo, Z. - Bredeson, J. G.: "A method to simplify a Boolean function into a near minimal sum-of-products for programmable logic arrays", IEEE Trans. on Computers, Vol. C-27, No.11, Nov. 1978, pp. 1028-1039

[Bar88] Bartlett, K., et al.: Multi-level logic minimization using implicit don't cares. IEEE Trans. on CAD, 7(6) 723-740, Jun. 1988

[Bow70] Bowman, R.M. - McVey, E.S.: A method for the fast approximation solution of large prime implicant charts. IEEE Trans. on Comput., C-19, p.169, 1970

[Bra84] Brayton, R.K. et al.: Logic minimization algorithms for VLSI synthesis. Boston, MA, Kluwer Academic Publishers, 1984

[Cha95] Chatterjee, M. -Pradhan, D.J.: A novel pattern generator for near-perfect fault coverage. Proc. of VLSI Test Symposium 1995, pp. 417-425

[Cou92] Coudert, O. - Madre, J.C.: Implicit and incremental computation of primes and essential primes of Boolean functions, In Proc. of the Design Automation Conf. (Anaheim, CA, June 1992), pp. 36-39
[Cou94] Coudert, O.: Two-level logic minimization: an overview. Integration, the VLSI journal, 17-2, pp. 97-140, Oct. 1994
[Hac96] Hachtel, G.D. - Somenzi, F.: Logic synthesis and verification algorithms. Boston, MA, Kluwer Academic Publishers, 1996, 564 pp.
[Hon74] Hong, S.J. - Cain, R.G. - Ostapko, D.L.: MINI: A heuristic approach for logic minimization. IBM Journal of Res. & Dev., Sept. 1974, pp.443-458
[McC56] McCluskey, E.J.: Minimization of Boolean functions. The Bell System Technical Journal, 35, No. 5, Nov. 1956, pp. 1417-1444
[McG93] McGeer, P. et al.: ESPRESSO-SIGNATURE: A new exact minimizer for logic functions. In Proc. of the Design Automation Conf.'93
[Ngu87] Nguyen, L. – Perkowski, M. – Goldstein, N.: Palmini – fast Boolean minimizer for personal computers. In Proc. of the Design Automation Conf.'87, pp.615-621
[Ost74] Ostapko, D.L. - Hong, S.J.: Generating test examples for heuristic Boolean minimization. IBM Journal of Res. & Dev., Sept. 1974, pp. 459-464
[Qui52] Quine, W.V.: The problem of simplifying truth functions, Amer. Math. Monthly, 59, No.8, 1952, pp. 521-531
[Rud87] Rudell, R.L. – Sangiovanni-Vincentelli, A.L.: Multiple-valued minimization for PLA optimization. IEEE Trans. on CAD, 6(5): 725-750, Sept.1987
[Rud89] Rudell, R.L.: Logic Synthesis for VLSI Design, PhD. Thesis, UCB/ERL M89/49, 1989
[Ser75] Servít, M.: A Heuristic method for solving weighted set covering problems. Digital Processes, vol. 1. No. 2, 1975, pp.177-182
[Sla70] Slagle, J.R. - Chang, C.L. - Lee, R.C.T.: A new algorithm for generating prime implicants. IEEE Trans. on Comput., C-19, p.304, 1970
[Esp1] http://eda.seodu.co.kr/~chang/ download/espresso/
[Esp2] ftp://ic.eecs.berkeley.org