EFFICIENT MINIMIZATION METHOD FOR INCOMPLETELY DEFINED BOOLEAN FUNCTIONS

Petr Fišer, Jan Hlavička
Czech Technical University, Karlovo nám. 13, 121 35 Prague 2
e-mail: hlavicka@fel.cvut.cz, fiserp@fel.cvut.cz

Abstract

The paper presents a minimization algorithm for Boolean functions whose values are defined only for a small part of their range. In contrast to other known minimization algorithms, it uses the "start big" strategy gradually reducing the dimensionality of a term until an implicant is generated. This approach leads to a very fast solution even for problems with several hundreds of input variables and several hundreds of minterms with defined output values. Its programmed version gave in these cases better results (in terms of runtime and minimality of the solution) than the state-of-the-art ESPRESSO.

1 Introduction

Synthesis of combinational functions with large numbers of input variables is a problem that appears in different contexts. It may be encountered e.g. in the design of control systems, where a large number of sensors deliver their input data to be processed by an automaton, or in diagnostics of logic circuits, where a sequence of code words is to be transformed into a sequence of test patterns. A common feature of those problems is the disproportion between the total number of minterms existing for the given number of input variables and the number of really used minterms, for which the output value is defined (care minterms). An efficient synthesis method, whose time and memory complexity is acceptable even for large problems with several hundreds of input variables and several hundreds of care minterms, is needed for the solution of such problems.

When generating a prime implicant (PI) for a given function *F*, the usual approach is to start from a 1-minterm and combine it with other 1-minterms (neighbors) in order to create an implicant with highest possible dimension. This was the principle of the original Quine's method [6] and of all minimization methods proposed later, whose representative survey can be found e.g. in [4]. The Quine-McCluskey minimization method, yielding all prime implicants and possibly a minimal form (if we succeed in solving the covering table), is applicable to problems with the maximum of 10 input variables. To solve also problems with higher number of variables, different heuristics have been proposed, accelerating the prime implicant generation and/or selection. To the most frequently used ones belongs the principle of consensus, starting the implicant generation from minterms having the lowest number of neighbors. However, the increase of tractable problem size gained thanks to the use of this method is rather modest, reaching e.g. 15 to 20 input variables – see e.g. [1].

One of the best known and probably the most successful of all programs for minimization of switching functions, called ESPRESSO [2], [4], uses a heuristic procedure for local search. This means that starting from some initial solution, the procedure tries to improve the quality of the solution through successive modifications. These modifications are directed by some quality criterion evaluating each newly reached situation. A similar approach was used also in other methods like e.g. [3]. A kind of local search is used also in our case, although the heuristic is different.

Organization of the paper is the following. The principle of the minimization meethod is formulated in section 3. Then an illustrative example is solved using the proposed method in section 4.

Statistical data based on results of experimental evaluation of the method are listed and commented in section 5. Some concluding remarks are offered in section 6.

2 Problem Statement

Let us have a Boolean function of n input variables $F(x_1, x_2, \ldots x_n)$, whose output values are defined by a truth table. Let the number of 1-minterms and 0-minterms be equal to u and z respectively, the rest are don't care states. The function is highly undefined, i.e. only few of the 2^n minterms have an output value assigned ($u+z<<2^n$). Our task is to formulate a synthesis algorithm, which will produce a two-level disjunctive form of F, whose complexity is close to the minimal disjunctive form. The measure of minimality generally corresponds 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, the total number of literals may be important, hence we will respect both possibilities here.

3 Prime Implicant Generation

3.1 "Start Big" Approach

Instead of increasing the dimension of an implicant starting from a 1-minterm, we will reduce the size of a hypercube, which contains an implicant as its subset, by adding literals to its term, until the hypercube becomes an implicant. This happens at the moment, when no 0-minterm is covered by this hypercube any more. Then we generate a PI by removing literals from the term, until we reach the maximum dimension without covering any 0-minterm. These two principles will be used as a basis for two phases of implicant generation. The first one, denoted as **Coverage-directed search** (or CD-search), combines the implicant generation with solution of the covering problem. The second one is denoted as **Sequential search**, because the literals that are candidates for rejection from a term are selected one by one.

3.2 Coverage-Directed Search

First we describe the implicant generation method based on hypercube reduction. Let us have a single-output Boolean function F defined by its on-set and off-set (set of 1-minterms and set of 0-minterms respectively). As the first step, we select the most frequent input literal from the given on-set and use it as a term from which an implicant will be derived. This term describes an implicant, if it does not cover any 0-minterm. If it is not an implicant, we add one literal and verify whether the new term already corresponds to an implicant by comparing it with all 0-minterms. If we obtain an implicant, we record the term and start searching for other implicants (see below). If not, some other literal must be added. We divide the given on-set into two subsets. One subset contains those minterms, which cannot be covered by any term containing the selected literal (minterms containing the literal with the opposite polarity). This subset will not be considered any more. In the other subset (containing the minterms, which can be covered by the selected literal) we again find the most frequent literal and multiply it with the previous one, so we have a two-literal product term. Again we compare this term with all 0-minterms and check if it is an implicant. We repeat this procedure until no 0-minterm is covered any more, i.e. until we get an implicant. We record this implicant and remove from the original on-set those minterms, which are covered by this implicant. Thus we obtain a reduced on-set containing only uncovered minterms. Now we repeat the procedure from the beginning and apply it to uncovered minterms, selecting the next most frequently used literal, until the next 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-minterms and no 0-minterm.

When selecting the most frequent literal, it may happen that two or more literals have the same frequency of occurrence. In these cases another heuristics can be applied. We construct terms as candidates for implicants by multiplying all selected literals with the previously selected one(s). From them we select those terms which are implicants. This will prevent a useless term prolongation. When there are still more possibilities to choose from, we select one at random.

A certain drawback of this algorithm is that it is greedy. Thus, once a literal is selected, it is kept till the end of the term generation. Therefore the obtained implicants need not be prime. In other words, we have to check whether some literals can be removed without losing any implicant. Here the second part of the PI generation algorithm, namely the Sequential search, finds its application.

3.3 Sequential Search

To check the results of the previous search, we may try and systematically remove from each term all literals one by one, starting from a randomly chosen position. If the hypercube obtained after removal of one literal does not cover any 0-minterm, we make the removal permanent. If, on the contrary, some 0-minterm is covered, we put the literal back and proceed to the next literal. After removing all removable literals we obtain one prime implicant covering the original term. This algorithm is also greedy, i.e. we stay with one PI even if there are more PIs that can be derived from the original term.

The sequential search obviously cannot reduce the number of product terms. On the other hand, the experimental results show that it reduces the number of literals by approximately 25%. After reducing the number of literals in the terms (and therefore extending the range they cover) some implicants may absorb others. Although this situation doesn't occur very often, solving the covering problem is desirable in this case.

3.4 Solution of the Covering Problem

A usual heuristic algorithm was used for solution of the covering problem. First, we prefer implicants covering minterms covered by the lowest number of other implicants. If there are more such implicants, we select implicants covering the highest number of yet uncovered 1-minterms. 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.

3.5 Iterative Minimization

The minimization process consists of the three above-mentioned phases (CD-search, Sequential search and Covering problem solution). The results of all these phases depend to a certain extent on random events, because whenever there are more possibilities to choose from and no selection criterion is given, a random number generator is used. Thus there is a chance, that the repeated application of the same procedure to the same problem will yield different solutions. We can improve the quality of the solution by repeating the CD-search phase followed by the Sequential search several times and recording all different PIs that were found. After the first pass we have a set of primes that is sufficient for covering the function. After every new iteration, another sufficient set could be added to the previous one (if the solutions are not equal). Finally, the covering problem is solved using all obtained primes. The set of primes gradually grows until a maximal reachable set is obtained. A typical growth of the size of the prime implicant set as a function of the number of iterations is shown in Fig. 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 solution could be found. In most of problems the maximal set of primes is extremely large. In the reality, the quality of the final solution improves rapidly during the first few passes and then remains unchanged, even though the number of prime implicants grows further. This fact can be observed in Fig. 1 (thick line). After reaching the minimal (or near-minimal) solution, its quality remains unchanged, even though the number of prime implicants grows further.

From the curves in Fig. 1 it is obvious, that selecting a suitable criterion for the termination of the iterative process has a key importance for the efficiency of the minimization. The most appropriate moment to stop the computation is marked with T1 on the horizontal axis in Fig. 1. However, determining its position is quite a difficult task. One possibility is to estimate the number of primes needed to make a good solution. This number strongly depends on a nature of the input function. Another possibility is to find the stopping point judging by the size of the temporary solution. In this case the covering problem must be solved repetitively after every iteration (or after several iterations). This means, that after every iteration there is known the best solution that could be obtained so far. This comfort is paid for by the "useless" loss of time when solving the covering problem. 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 size of 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 stopping point can also be defined explicitly, like e.g. when the current solution meets some criterion, (the maximum allowed number of product terms to fit into the PLA chip). The criteria described above are more flexible than terminating the minimization by the number of iterations or amount of elapsed time, although these last criteria may be used as an emergency exit for the case of unexpected problem size and complexity.

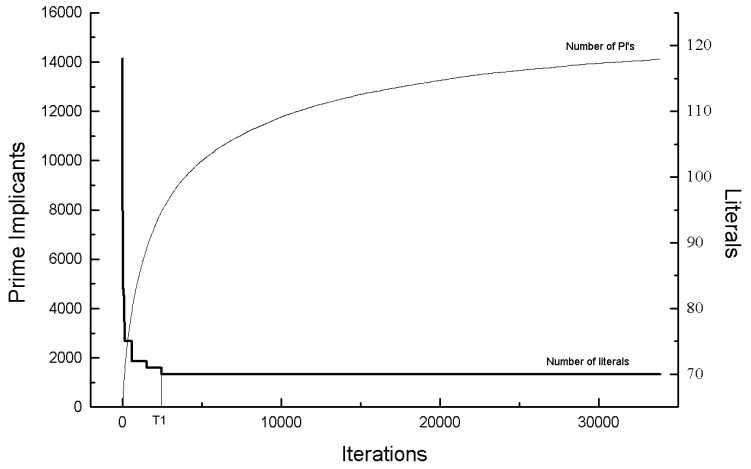

Fig. 1. *Growth of PI number and decrease of SOP length during iterative minimization*

4 Illustrative Examples

4.1 CD-Search Example

Let us have an incompletely defined Boolean function F(x_0 ..x_9) of ten input variables with ten care minterms given by a truth table shown below (the 1-minterms are shaded).

```
var: 0123456789.F
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 of the minimization we count the numbers of literals in the input table. The "0"-line and "1"-line give counts of x_n' and x_n literals respectively.

```
var: 0123456789
0:   3435322434
1:   3231344232
```

In this table we can locate x_3' as the most frequent literal with five occurrences (underlined). This literal is our candidate for an implicant, but is not yet an implicant, because it covers a 0-minterm (the 6th minterm) in the original function. Hence another literal must be added. When searching for the next literal, we can reduce the range of our search by suppressing all 1-minterms containing the selected literal with the opposite polarity. In our case it is only the 5th minterm, which contains the x_3 literal and

thus cannot be covered by an implicant containing the x_3' literal (in the next table shaded dark). In the remaining 1-minterms we look for the most frequent literal:

```
var: 0123456789 F
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
```

This time we find several literals with maximal frequency of occurrence (x_1', x_5, x_6, x_7'), hence the second selection criterion must be applied. Multiplying the x_3' literal by all literals found so far, we create four product terms: $x_3'x_1'$, $x_3'x_5$, $x_3'x_6$, $x_3'x_7'$. First we check whether some of them are already implicants. The term $x_3'x_5$ is not an implicant (it covers the 6th minterm), so it is discarded, whereas the remaining three terms represent implicants. Now we must choose one of them. As they are all equal, we may select at random – e.g. $x_3'x_6$. This implicant is stored and the search continues. For further search we discard the minterms covered by this implicant (minterms 1, 2, 8 and 9 – dark shading in the next table) and the next most frequent literal can be selected:

```
var: 0123456789 F
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
```

Now we find four literals with equal frequency (2) and choose one at random – e.g. x_9'. This is not an implicant, thus we must add more literals. The selected literal does not suppress any of the remaining 1-minterms, therefore we need not reduce the range of our search. We construct three product terms: $x_4'x_9'$, $x_5'x_9'$, $x_6'x_9'$. None of them is an implicant, we select e.g. $x_4'x_9'$. Now we have two possibilities to choose the next literal: $x_4'x_5'x_9$ *or* $x_4'x_6'x_9'$. Again, none of these terms is an implicant. Hence, we must add one more literal to create an implicant. The result of the CD-search is thus $x_3'x_6 + x_4'x_5'x_6'x_9'$.

4.2 Sequential Search Example

We will continue with our running example. The result of the CD-search, i.e. the function $x_3'x_6 + x_4'x_5'x_6'x_9'$ is now to be simplified by the Sequential Search algorithm. We must process both product terms one by one, the order is not significant. We start with the term $x_3'x_6$ and try to remove one literal. If we remove x_3' and compare the remaining term (x_6) with all the 0-minterms, we'll find that it collides with 3rd, 4th and 7th minterm and thus x_3' cannot be removed. We continue with the x_6 literal. It cannot be removed either, because the remaining term covers the 6th minterm. This term cannot be reduced any more and thus it is a prime implicant. Now we try to reduce the second term ($x_4'x_5'x_6'x_9'$). We remove x_4' and find that the remaining term $x_5'x_6'x_9'$ is an implicant. So we keep the removal permanent. Now by removing x_5' we get $x_6'x_9'$. This is not an implicant (it covers the 6th minterm). Similarly, when we remove x_6', the term $x_5'x_9'$ covers the 4th minterm, hence x_6'cannot be removed either. Finally, after removing x_9', the term $x_5'x_6'$ will be an implicant. As we exhausted all

possibilities of removal, this is a prime implicant. The minimal SOP form of the function is $x_3'x_6 + x_5'x_6'$.

5 Experimental Results

The proposed algorithm was programmed in Borland C++ Builder and tested under MS Windows NT. The processor used was a Celeron 433 MHz and 160 MB RAM, the runtime was measured in seconds. The quality of the results was measured by two parameters: number of product terms (implicants) and total number of literals. Although every implementation basis requires different evaluation criteria, these two figures are good representatives of the overall complexity of the solution obtained.

An extensive experimental work was done to evaluate the efficiency of the proposed algorithm, especially for the problems of large dimensions. These experiments can be divided into three groups.

First a small example with 10 input variables and 20 care minterms, where also the true minimization could be used, was solved by different methods. The results were compared with the Quine-McCluskey minimization method [5], [6] (programmed by ourselves) and with ESPRESSO. All methods gave the same results (minimal disjunctive form). The runtimes in seconds were the following: CD-search 0.03, Espresso 0.03 Quine-McCluskey 438.32.

In the second group of experiments the efficiency was compared with ESPRESSO 2.3 [8]. The problems solved were at the beginning the standard Berkeley benchmarks [7]. As the size of these benchmark problems is relatively small, the runtimes needed by the CD-search were longer than those of ESPRESSO. Hence some larger problems had to be used to prove the capabilities of the method. The truth tables of single-output functions were generated by a random number generator, for which only the number of input variables and number of care minterms in the truth table were specified. The on-set and off-set were kept approximately of the same size.

A third group of experiments aimed at establishing the limits of applicability of the newly designed method. Therefore a set of large problems was generated, where only the proposed algorithm was tested (see paragraph 5.2).

		Number of input variables							
Number of care minterms		20	60	100	140	180	220	260	300
	20	0.01/3/8 (1) 0.09/3/8	0.01/2/4 (2) 0.20/2/4	0.00/2/4 (1) 0.41/2/6	0.01/2/5 (1) 0.72/2/6	0.01/2/4 (1) 1.36/2/4	0.02/2/5 (1) 1.30/2/7	0.01/2/4 (1) 3.24/2/4	0.02/2/4 (2) 3.60/2/4
	60	0.20/7/25 (19) 0.19/6/25	6.49/5/15 (254) 1.09/5/15	0.15/5/17 (4) 2.51/5/20	0.03/4/12 (1) 4.68/4/14	0.04/4/17 (1) 6.84/4/17	0.06/4/15 (1) 11.96/4/16	0.11/4/12 (2) 18.83/4/12	0.15/4/13 (2) 21.87/4/13
	100	0.12/11/42 (4) 0.35/10/43	0.84/7/28 (14) 1.86/6/28	0.18/7/29 (2) 6.62/7/31	4.00/7/27 (29) 10.22/7/28	1.50/6/27 (10) 23.22/6/27	0.12/6/24 (1) 26.64/6/24	1.33/6/24 (6) 39.12/6/24	16.74/5/19 (78) 40.58/5/19
	140	1.30/13/64 (19) 0.51/14/66	4.83/10/45 (37) 3.93/10/45	4.79/9/39 (26) 12.68/9/40	2.65/9/38 (11) 20.99/9/40	17.74/8/36 (63) 35.73/8/36	4.82/7/33 (14) 35.76/7/35	0.29/7/31 (1) 76.03/8/36	0.76/6/27 (2) 74.47/7/32
	180	6.28/15/74 (49) 0.86/15/76	5.72/13/60 (24) 7.92/13/61	0.60/11/55 (2) 19.27/11/55	26.06/10/48 (63) 33.27/10/48	47.68/10/45 (90) 91.56/10/45	2.20/9/45 (4) 73.71/9/45	66.20/9/42 (92) 99.91/9/43	2.38/10/47 (3) 149.46/10/47
	220	2.51/21/104 (14) 1.34/21/105	10.55/16/76 (28) 9.87/15/77	36.41/12/58 (71) 39.35/12/61	28.40/13/61 (40) 54.43/13/63	65.91/12/59 (75) 87.40/12/59	42.47/12/58 (43) 141.97/11/59	0.99/11/60 (1) 167.00/12/61	44.89/11/54 (37) 179.06/10/54
	260	1.78/25/136 (6) 1.32/26/146	18.21/18/89 (34) 15.06/17/89	62.85/14/72 (81) 41.78/14/73	26.20/14/74 (26) 70.06/14/79	31.17/14/74 (26) 135.27/14/76	11.02/12/69 (8) 149.86/13/71	50.19/13/66 (33) 207.24/12/69	43.04/13/68 (25) 314.29/12/68
	300	1.88/28/153 (5) 1.41/29/163	11.28/20/114 (16) 15.77/20/118	21.38/16/93 (23) 42.18/16/94	56.51/17/86 (42) 99.59/16/86	253.29/14/71 (145) 201.60/14/76	257.16/14/75 (136) 294.64/13/75	81.32/16/81 (33) 334.32/15/85	11.57/13/77 (5) 347.02/14/83

Tab. 1. *Comparison of CD-search and ESPRESSO*

5.1 Comparison of CD-Search and ESPRESSO

To compare the performance and result quality achieved by the two minimization programs, a set of problems with up to 300 input variables and up to 300 minterms were used. Larger problems could not be solved, because ESPRESSO would not reach a solution within acceptable computing time.

Every problem was at first solved by ESPRESSO and then by iterative CD-search. The minimization was stopped when CD-search reached equal or better solution than ESPRESSO did.

The results of the comparison are shown in Tab.1. The first row of every cell contains the CD-search results, the last row shows ESPRESSO results. The entry format is: “time in seconds/ #of implicants/ #of literals”. The number of iterations is indicated in parentheses. When the CD-search reached the same or better solution than ESPRESSO in shorter time, the appropriate cell is shaded gray.

5.2 Solution of Very Large Problems

For problems with more than 300 input variables ESPRESSO cannot be used at all. Hence when investigating the limits of applicability of the CD-search, there was no possibility to verify the results by any other method. The results of this test are listed in Tab. 2. The time in seconds needed to complete one iteration for various sizes of the problem is shown here. We can see that a problem with 1000 variables and 1000 minterms was solved by the CD-search in less than 3 minutes.

#minterms/ #vars	200	400	600	800	1000
200	0.73	1.81	1.46	2.77	2.34
400	3.35	5.43	8.24	11.03	13.27
600	21.87	14.22	17.35	28.49	37.96
800	43.65	29.58	32.33	67.78	90.09
1000	71.87	55.83	56.90	122.03	155.55

Tab. 2. *Time for one iteration on big problems*

5.3 Time Complexity Evaluation

Like for most heuristic and iterative algorithms, it is impossible to evaluate the time complexity of CD-search algorithm exactly. We have observed the average time needed to complete one pass of this algorithm for various sizes of the input truth table. For every problem size ten different samples were generated and solved and the average of runtimes was taken. Fig. 2 shows the growth of an average runtime as a function of the number of care minterms (20-260) where the number of input variables is changed as a parameter (20-300). The curves in Fig. 2 can be approximated with the square of the number of care minterms. Fig. 3 on the other hand shows the runtime growth depending on the number of input variables (20-300) for various numbers of care minterms (20-260). Here the time complexity is almost linear.

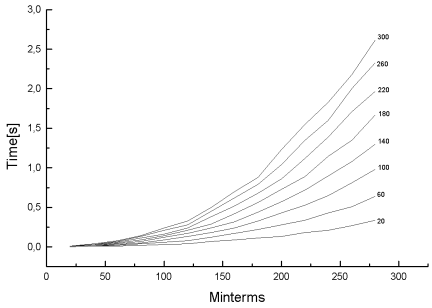

Fig. 2. *Time complexity (1)*

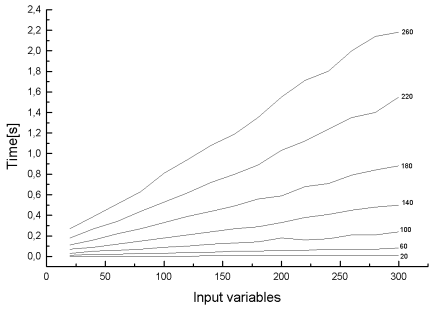

Fig. 3. *Time complexity (2)*

6 Conclusions

A new method for single-output Boolean function minimization method has been presented. It is applicable above all to problems with large dimensions and large number of don't care states. The PI generation method is quite straightforward and therefore very fast. Hence it can easily be used in an iterative manner. The strength of the method consists in the possibility to choose between a very fast solution, obtained in one iteration and minimal (or near-minimal) solution which is obtained during several iterations. The same results as with ESPRESSO can be achieved, but the runtimes are much shorter. For large problems with several hundreds of variables the program beats ESPRESSO both in minimality of the result and in the runtime.

The future research will be oriented towards the possibility to process the care terms (not only minterms) in the input file and towards the group minimization in order to minimize also the multi-output functions.

Acknowledgment

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

References

[1] 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

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

[3] 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

[4] HACHTEL, G.D. - SOMENZI, F.: Logic synthesis and verification algorithms. Boston, MA, Kluwer Academic Publishers, 1996, 564 pp.

[5] McCLUSKEY, E.J.: Minimization of Boolean functions. The Bell System Technical Journal, 35, No. 5, Nov. 1956, pp. 1417-1444

[6] QUINE, W.V.: The problem of simplifying truth functions. Amer. Math. Monthly, 59, No. 8, 1952, pp. 521-531

[7] ftp://ftp.mcnc.org/pub/benchmark/Benchmark_dirs/LGSynth93/testcases/pla/

[8] http://eda.seodu.co.kr/~chang/download/espresso/