Report Nr 08

Semantic and Syntactic Simplification of Truth Functions

H.H.F.M. Verstralen

Cito

3.4

69^8

Semantic and Syntactic Simplification of Truth Functions

by

Huub H.F.M. Verstralen,

Report Nr 08

Project 'Optimal Item Selection'

CITO

Arnhem, The Netherlands, 1990

Submitted for publication

General Introduction

The purpose of Project 'Optimal Item Selection' is to solve a number of issues in automated test design, making extensive use of optimization techniques. In each report, one or several theoretical issues are raised and an attempt is made to solve them. Furthermore, each report (where appropriate) is accompanied by one or more computer programs, which are implementations of the methods that have been investigated. Some of these programs will be available in the future.

T.J.J.M. Theunissen,
project director.

Abstract

Some restrictions in Linear Programming (LP) problems are easier formulated as a Boolean expression. Here we describe a way to translate a Boolean expression, or the associated function, via a Disjunctive Normal Form (DNF) representation, to an equivalent set of linear restrictions. The transformation of a truth function f into the wanted DNF proceeds in two steps. First the set PI(f) of prime implicants of f is calculated, and second a 'cheapest' subset of PI, equivalent with f is chosen. The best known algorithms for the first step, to find PI(f), are the semantic algorithm of Quine (1952) and McCluskey (1956) or, if one has a Boolean expression for f, the syntactic algorithm of Quine (1955). The first algorithm is appreciably improved, and the scope of the second is enlarged to include valid formula's. As a consequence it can also be used as a validity test for Boolean expressions. The second step is discussed as well, where the suggestions of Quine (1955) are supplemented with two properties of redundant elements of PI(f), that make their identification more easy. However, the Set Covering approach to the second step appears to be better suited.

Content Indicators:
CR classification :
G. Mathematics of Computing
G.4. Mathematical Software, Algorithm analysis, Algorithm efficiency
General terms: Algorithms, Theory
Key words : Boolean Expressions, Boolean functions, Normal Forms, Validity test, Linear Programming

Contents

1 Introduction 9

2 Algorithms for simplification of truth functions 11

3 Two algorithms that find all the prime implicants of a truth function 15

3.1 The Quine and McCluskey algorithm 15

3.2 The syntactic method of Quine 23

4 Algorithmic search for the cheapest equivalent combination of prime implicants 27

5 Conclusion 31

References 33

Titles of other research reports 35

1 Introduction

Our interest in truth functions derives from a wish to incorporate restrictions that are formulated as Boolean expressions in linear optimization algorithms for (educational) test construction. The construction of tests can be greatly aided by the application of optimization techniques like Linear Programming (LP), Theunissen (1985), Cito (1990). An LP program searches an optimum for a goal function in a space constrained by linear inequalities. However, several test construction constraints, among which are interitem dependencies, are more naturally formulated as Boolean expressions. As a simple example of an interitem dependency one can think of: 'if the test contains item 1, it must also contain items 2 and 3'. Therefore we face the problem to adapt Boolean expressions to the LP approach. We proceed by finding a 'cheapest' Conjunctive Normal Form for an expression, which is easily transformed into a system of linear inequalities. However, for an application of such an algorithm to be succesful in an on line test construction program, speed is an important desideratum. The speed of the application is influenced by the speed of the transformation itself, and by what it delivers to the LP program. Especially the number of linear inequalities must be kept as small as possible.

Because it is the best known algorithm for Boolean function simplification, we first involved ourselves with the Quine and McCluskey algorithm. Although we devised a substantial improvement on the original algorithm, especially for more complex expressions, processing times were considered not entirely satisfactory for the intended application. Therefore, and because we start with a Boolean expression, the syntactic algorithm of Quine (1955) was considered as well. His original proof of the algorithm explicitly exluded valid expressions. However, it turned out that some small changes in the proof are sufficient to also include valid expressions. Moreover, the processing times of our implementation of the algorithm are entirely satisfactory for the class of expressions that we can expect.

The just mentioned algorithms yield the complete set of Prime Implicants of a Boolean function. This set may contain redundant elements, which would lead to superfluous linear inequalities. Therefore, the second step in the process is to find a cheapest subset of the set of prime implicants that represents the original expression. In principle this problem can be solved by following Quine's (1955) suggestions. We supplemented this approach with two theorems on the redundancy of prime implicants that enable to simplify and speed up the algorithm, but are considered interesting by themselves.

However, we did not succeed in constructing a satisfactory algorithm from this basis. A greedy algorithm from the set covering literature proved to be much better suited.

Although we may consider this part of the problem as satisfactorily solved, concerns remain about the number of linear inequalities that it produces.

2 Algorithms for simplification of truth functions

A Boolean variable takes values from the set B = {0,1} and a truth function is a function on B^n (n=0,1,2,...) with values in B. We denote a Boolean variable, also called a letter, by an index number or by a lower case letter. The usual basic set of truth functions is denoted as follows:
Unary : - (NOT, Negation)

Binary: + (OR, Disjunction)
* (AND, Conjunction)
> (IF .. THEN, Implication)

The unary function is notated as prefix, the binary functions are notated as infix.

All truth functions can be expressed with the help of this notation. A Boolean expression defines precisely one truth function, while a single truth function can, in general, be expressed in many ways as a Boolean expression. Two Boolean expressions are called equivalent if they define the same truth function. The problem of how to transform a Boolean expression to an equivalent system of linear inequalities is readily solved by realizing what kind of expressions hardly needs a transformation. An example of such an expression is:

a+b * c+d

which is equivalent to the following system of linear inequalities:

a+b > 0
c+d > 0

where a..d and + and > are interpreted in their usual arithmetical sense.

In general every Boolean expression of the form:

$$\prod_i \sum_j a_{ij} \tag{1}$$

is equivalent to the following system of linear inequalities:

$$\sum_j a_{ij} > 0 \quad (i=1,\ldots,n) \tag{2}$$

where n equals the number of Boolean sums in (1).

A Boolean expression of the form (1) is said to be in Conjunctive Normal Form (CNF). The negation of (1) together with the property that $-(a+b) = -a*-b$ (The Morgans Law) produces a so called Disjunctive Normal Form (DNF); it can be written like (1) but the sequence of Π and Σ is reversed.

The expressions (1) and (2) would not be of very great help were it not that it can be easily proved that every Boolean expression can be transformed in an equivalent CNF or DNF. In the sequel we will be concerned exclusively with DNF representations of Boolean expressions.

Because especially the number of inequalities, but also the number of variables impede the algorithms for optimal test construction, it is important to find a DNF representation of a Boolean expression that is as simple as possible.

From Wegener (1987) and from Hammer (1974) it appears that Quine (1952, 1955) and McCluskey (1956) are still the leading articles on the subject of simplification of Boolean functions. We found that Wegener (1987), although he refers to Quine (1955), does not in any way hint at its contents. And Hammer (1974) mentions the consensus method in Quine (1955), but does not refer to its source. It seems therefore, that these methods on the one hand are considered as part of the anonymous mathematical heritage, but on the other hand partly tend to fade into oblivion.

To explain the algorithms, first some terminology has to be introduced. We thereby partly follow Quine (1952) and Wegener (1987). A *litteral* is a letter or a negation of a letter. A *monom* or a *clause* is a litteral or a conjunction of litterals. A *fundamental formula* is a monom containing no letter twice. Because the only monoms that we are concerned with are fundamental formulas, in the sequel the term 'monom' means 'fundamental formula'. An alternation, or Boolean sum, of monoms is called a *normal formula*. We denote monoms by lower case Greek letters and arbitrary Boolean expressions by uppercase Greek letters. Let α and β be two monoms. We say that α *subsumes* β if all the litterals in β are among those of α. α is called an *implicant* of Φ if α implies Φ, α is called a *prime implicant* of Φ if it is an implicant of Φ and subsumes no shorter formula which implies Φ. When two litterals share the same letter we say that they *agree* when their signs are equal. Two monoms agree when the letters they share have the same sign. Two monoms α and β have a *consensus* γ, also a monom, when they agree except for one letter, call the disagreeing letter v. The consensus γ of α and β consists of exactly all the litterals of α and β (duplicates only once) except v and -v. We call α and β the *parents* of their consensus γ. Let x and x' ($\in \{0,1\}^n$) be assignments for the variables a_i ($i=1, .. ,n$). The assignment x can be related to precisely the one monom α in the variables a_i such that $\alpha(x) = 1$ and $\alpha(x') = 0$ for all $x' \neq x$. Given this relation between monoms and assignments we feel free to switch terms from an assignment x and its thus related monom. (This avoids the more cumbersome introduction of minterms.) We say that a truth function Φ *accepts* an assignment x if $\Phi(x) = 1$. The symbol \ is used for omission. If, e.g., a is a litteral of a monom α, $\alpha\backslash a$ is the monom we get from α by omitting a.

This ends the introduction of basic concepts and notation. We now turn to the process of finding a DNF-expression for a Boolean function f. All methods to this end proceed in two major steps. In the first step all the prime implicants of a truth function are constructed. The second step looks for the cheapest disjunction of prime implicants that accepts precisely the same set of assignments as f. These procedures build on theorem 1 from Quine (1952) that every simplest DNF representation of a Boolean expression is a disjunction of prime implicants. We first treat two methods that are designed to find all the prime implicants of an arbitrary Boolean expression or its associated function.

3 Two algorithms that find all the prime implicants of a truth function

Although both algorithms build on the same two principles, consensus and subsumption, they nevertheless function quite differently. The first and best known is based on the articles of Quine (1952) and McCluskey (1956) and is further referred to as Q&M. The second algorithm is based on Quine (1955), although he was not the first discoverer, as he himself recognised. This algorithm is further referred to as Syn.

3.1 The Quine and McCluskey algorithm

Given a truth function Φ Q&M first finds the set of assignments $\Phi^{-1}(1) = \{ x \mid \Phi(x) = 1 \}$. Relate, as indicated above, the assignments with their unique accepting monoms, then $\Phi^{-1}(1)$ is a DNF representation of Φ and is called the *developed normal form* of Φ, because every monom in this DNF representation of Φ, contains all variables of Φ. The algorithm proceeds by testing for every pair of implicants α and β whether they have a consensus γ. If so, then γ is an implicant of Φ, which contains all the variables of its parents α and β except the disagreeing letter, and agrees with both its parents. It follows that it is subsumed by both and therefore can replace them in the normal form of Φ without changing the truth function Φ. The consensus, which is one variable shorter, is placed at the end of the chain of implicants that represent Φ and its parents are marked. This process is continued with the shorter implicants, until no longer a consensus can be found. It is important to note that in this process only implicants with the same variables have to be tested for a consensus. Because, if pairs that do not share their variables yield a consensus, this consensus must be an implicant of Φ with at least as much variables as its parents, and therefore it is already produced or even already processed. Therefore every pair is first tested on equality of variable set. It is not difficult to verify that the unmarked implicants in the chain are precisely the prime implicants of Φ. This is the process that Quine (1952) proposed for finding the prime implicants of a truth function.

McCluskey (1956) improved the processing time of the algorithm by noting that many comparisons between implicants could be skipped because they surely do not yield a consensus. Two implicants will surely not yield a consensus if their number of negated letters differs not precisely by one. They possibly yield a consensus when this difference equals one. Therefore McCluskey arranged the implicants in such a way that first the implicants with no letters affirmed (at most one implicant) are grouped together, then those with one letter affirmed, and so on. Comparisons can then be confined to implicants of consecutive groups, all with the same number of variables and differing by one in number of negated letters. According to Mileto and Putzolu (1965) the number of comparisons in relation to Quines (1952) algorithm is reduced by a factor that approaches 3.

Subsuming: Quine makes a comparison within all pairs of implicants with the same number of variables, McCluskey restricts comparison to pairs of which the members differ by one in number of negations.

However, this approach still produces, in general, a majority of unneccessary tests for equality of variable set. Two implicants may only produce a consensus in this process when they share their letters. This becomes an increasingly rare event as the proportion of left out variables of the processed monoms approaches 0.5, e.g., with a twelve variable expression when four variables are left out the odds may be about 1 to 3000 (if the expression is almost valid). If it can be arranged that comparisons are (primarily) restricted to implicants that share their variables, the algorithm gains in efficiency. The next paragraphs describe how this can be achieved and what increase in efficiency is to be expected.

Code an implicant α of Φ as an ordered pair of binary vectors $[x,y]$, where $x = (x_1, \ldots ,x_n)$, and y similarly, where n equals the number of different letters in Φ. $x_i = 1$ if letter i occurs in α else $x_i = 0$. $y_i = 1$ if $x_i = 1$ and variable i is negated in α. Call the set of implicants of Φ with n-k variables of which p ($p = 0, \ldots ,n-k$) are negated $\Phi I(k,p)$. In Quine (1952) all elements of $\cup_p \Phi I(k,p)$ are mutually compared to form $\cup_p \Phi I(k+1,p)$, while the comparisons in McCluskey are restricted to pairs (α,β) with $\alpha \in \Phi I(k,p)$ and $\beta \in \Phi I(k,p+1)$. As already mentioned, this restriction results in an improvement factor approaching 3.

By (quick)sorting the implicants $\alpha = [\alpha x,\alpha y]$ in $\Phi I(k,p)$ according to αx, we achieve that monoms that share their variables are grouped together. This allows for a great reduction in tests for equality of variable set to the extent that almost only comparisons are made between monoms that share their variables. Some small changes (indicated with * in the pseudocode below) in the Quine & McCluskey algorithm accomplish this.

Another difficulty that hinders the process is the fact that the same consensus can be produced multiple times from different pairs of parents. For example, abc can be produced by abcd and abc$\underline{d}$ (underline indicates negation) or by abce and abc$\underline{e}$ etc. until the last variable. And it not only <u>can</u> be, but it will be. Because if abc is an implicant, all its possible parent-pairs necessarily show up in the two higher regions of the monom chain, because they are implicants of Φ. Therefore, if there is no check for double implicants, they very soon consume all the available memory (for more details, see Verstralen, 1988). The above mentioned literature suggests that it must be tested whether a consensus already has been found. But such a test is very time consuming.

However, this test can be avoided, by identifying a unique monom among its doubles, and to store only this one. A clue for identifying a unique monom is the one-one relation between a set of identical monoms and its parent pairs. In the set of parent pairs there is precisely one member that has no irrelevant variable lower indexed than its non-agreeing variable, the new irrelevant variable in their consensus. This can be tested efficiently, as can be seen in the part with the Window variable in the following pseudocode.

```
{
   Given the assignments that Φ accepts in McCluskey's order.
   ( # indicates 'number of' )
   n : #variables in Φ;
   k : #left out variables --> n-k: #variables in implicant;
   p : #variables negated
}
    for k := 0 to n-1 do
     for p := 1 to n-k do begin
       ox := 0;                      { * initialize ox = previous αx }
       Firstβ := First β in ΦI(p,k); { * Firstβ is a variable }
         for α ∈ ΦI(p-1,k) do begin
         First := αx <> ox;          { * Boolean to indicate variable
                                         change}
         if First then ox := αx      { * update ox }
         else β := Firstβ            { * return to Firstβ if α
                                         contains same vars }
         while βx <= αx do begin     { * check only β with same vars }
           if αx = βx then begin     { * just after First maybe
                                         there are some βx < αx }
             if First then begin
               First  := false;
               Firstβ := β           { * update Firstβ }
             end;
             DisVars := αy xor βy;   { find the disagreeing vars }
             if DisVars and βy = DisVars  { just one disagr. var ? }
             then
               mark(α); mark(β);     { α and β have a consensus }
               Window := DisVars - 1;{ Window on all vars with lower
                                       index than disagreeing var }
               if Window and xβ = Window then  { test to avoid duplicates }
                  Adjoin consensus(α,β) to ΦI(p-1,k+1)
           end; { if αx = βx }
           next β ∈ ΦI(p,k)
```

```
        end;   { while $\beta x <= \alpha x$ }
        if $\alpha$ not marked then Adjoin $\alpha$ to PI($\Phi$)
        next $\alpha \in \Phi I(p-1,k)$;
      end; { for $\alpha$ }
      quicksort($\Phi I(p-1,k+1)$);
    end;   { for p }
    Adjoin all not marked $\beta \in \Phi I(p=n-k,k)$ to PI($\Phi$)
  end;    { for k }
```

The time for sorting turns out to be a minor ratio of total processing time.

To get an impression of the gain that can be expected from the indicated changes, we calculated the expected number of comparisons between implicants under the assumption that only implicants with the same variables are compared. From Monte Carlo experiments we got estimates of the percentages of the number of comparisons between implicants with the same variables compared to the total number of comparisons, for the improved and the original algorithm as well. The expected number of comparisons in the original Quine & McCluskey algorithm are given in Miletu and Putzolu (1965). Also estimates of processing times are given.

To derive the expected number of comparisons between implicants of the same variables, the framework given in Miletu and Putzolu is helpful. To get a linear notation let $a@b = a!/(a-b)!b!$: the number of combinations of size b from a set of size a. Consider a function $U(i,u)$ on B^n with u out of 2^n true assignments or ones with $i \in \{1, .. ,2^n@u=z\}$. Let $v(s)$ be a subset of $(n-k)$ variables of $U(i,u)$ given by a subset s of the index set $\{1, .. ,n\}$, $s \in \{1, ... ,n@k\}$. Denote by $\Gamma(i,s,k,p)$ the number of k-cubes of $U(i,u)$ with p false or 0 assignments for the n-k variables in $v(s)$. Then the number of pairs of k-cubes $c(j,s,k)$ and $c(l,s,k)$ in $U(i,u)$ that share the same variable set $v(s)$ and such that the number of 0 assignments in $c(j)$ exceeds the number in $c(l)$ by one equals:

$$!\xi(i,k) = \sum_{p=1}^{n-k} \sum_{s=1}^{\binom{n}{k}} \Gamma(i,s,k,p)\, \Gamma(i,s,k,p-1) \qquad (3)$$

Then the expected number of same variable comparisons between k cubes for functions with u ones is:

$$!\underset{i}{E}(\xi(i,k)) = \zeta(u,k) = \frac{1}{z}\sum_{i=1}^{z}\sum_{s=1}^{\binom{n}{k}}\sum_{p=1}^{n-k}\Gamma(i,s,k,p)\,\Gamma(i,s,k,p-1) \tag{4}$$

By symmetry of i and s this can be simplified to:

$$!\zeta(u,k) = \frac{\binom{n}{k}}{z}\sum_{p=1}^{n-k}\sum_{i=1}^{z}\Gamma(i,1,k,p)\,\Gamma(i,1,k,p-1) \tag{5}$$

There are (n-k)@p k-cubes c(1,j,k,p) with variable set v(1) and p negated variables. Now let

$$\begin{aligned} a(i,j,1,k,p) &= 1 \text{ if } U(i,u) \supset c(1,j,k,p) \\ &= 0 \text{ else} \end{aligned}$$

then $$!\Gamma(i,1,k,p) = \sum_{j=1}^{\binom{n-k}{p}} a(i,j,1,k,p) \tag{6}$$

Substitution of (6) in (5) yields after reordering summation terms:

$$!\zeta(u,k) = \frac{\binom{n}{k}}{z}\sum_{p=1}^{n-k}\sum_{i=1}^{z}\sum_{j=1}^{\binom{n-k}{p}}\sum_{h=1}^{\binom{n-k}{p-1}} a(i,j,1,k,p)\,a(i,h,1,k,p-1) \tag{7}$$

By symmetry of indices j and i this reduces to:

$$!\zeta(u,k) = \frac{\binom{n}{k}}{z}\sum_{p=1}^{n-k}\binom{n-k}{p}\sum_{h=1}^{\binom{n-k}{p-1}} N(1,h,1) \tag{8}$$

where

$$!N(1,h,1) = \sum_{i=1}^{z}\sum_{h=1}^{\binom{n-k}{p-1}} a(i,1,1,k,p)\,a(i,h,1,k,p-1) \tag{9}$$

which can be interpreted as the number of functions U(i,u) that contain the k-cubes c(1,1,k,p) and c(1,h,k,p-1) e.g., in the last n-k variables:

$$\begin{array}{lccccccccc} c(1,1,k,p) = & x_1 & .. & x_k & 0 & .. & 0 & 1 & .. & 1 \\ & 1 & & k & k+1 & & k+p & k+p+1 & & n \end{array}$$

N(1,h,1) is the number of sets formed with u out of z elements wich contain the w(k+1) = $2^{(k+1)}$ elements of c(1,1,k,p) $\cup$ c(1,h,k,p-1). The intersection of both k-cubes is empty because their sign differs at least in one variable in v(1). It follows that N(1,h,1) = (z-w(k+1))@(u-w(k+1)). So that finally we have:

$$\zeta(u,k) = \binom{n}{k} \frac{N(1,h,1)}{z} \sum_{p=1}^{n-k} \binom{n-k}{p} \binom{n-k}{p-1} \tag{10}$$

Which implies that the expected number of comparisons between cubes with equal variable sets for an arbitrary Boolean function of n variables with u true assignments is:

$$\sum_{k=0}^{n} \zeta(u,k) \tag{11}$$

Table 1 gives some data on expected numbers of comparisons for the improved Quine and McCluskey algorithm (NQ&M), combined with data from Mileto and Putzolu (1965) on the number of comparisons in the original Quine and McCluskey algorithm (OQ&M). The processing times refer to an 8MHz IBM-AT compatible computer (NSI = 7.1), for an implementation in Pascal (Borland, 1987). Under Mean #Comparisons NQ&M are given the number of comparisons between pairs of k-cubes with the same variables. Under %Tot one finds the percentage of this number in comparison with the total number of comparisons of NQ&M, as estimated by Monte Carlo experiments, 25 per case if #vars<11 else 10 per case. For 11 and 12 variables we extrapolated the Mileto and Putzolu (1965) numbers on OQ&M.

Table 1 Results on performance of the improved Quine & McCluskey algorithm

U(i,u)		Mean #Comparisons				Mean Processing time (sec)		
#vars	%true	OQ&M	NQ&M	%Tot	OQ&M/NQ&M	OQ&M	NQ&M	OQ&M/NQ&M
6	25	74.0	50.8	91	1.5	0.04	0.05	0.80
	50	696.4	268.2	94	2.6	0.10	0.09	1.11
	75	4056.2	902.1	96	4.5	0.28	0.22	1.27
7	25	330.5	202.0	94	1.6	0.08	0.07	1.14
	50	3487.4	1088.7	97	3.2	0.29	0.22	1.32
	75	23621.3	3834.1	98	6.2	1.41	0.63	2.24
8	25	1456.0	794.2	97	1.8	0.17	0.16	1.06
	50	17200.2	4392.6	98	3.9	1.04	0.57	1.82
	75	135158.1	16207.2	99	8.3	7.05	1.79	3.94
9	25	6379.7	3108.4	99	2.1	0.46	0.37	1.24
	50	83938.1	17678.0	99	4.8	4.24	1.68	2.52
	75	761392.3	68290.1	99	11.2	32.42	6.07	5.34
10	25	27888.1	12153.3	99	2.3	1.60	0.99	1.62
	50	406393.7	71098.6	100	5.7	18.48	5.29	3.49
	75	4228973.6	287182.7	100	14.7	153.49	21.27	7.21
11	25	121667.9	47545.1	100	2.6	6.31	3.29	1.92
	50	1945486.7	286016.9	100	6.8	82.58	19.68	4.20
	75	23089747.0	1206171.8	100	19.1	826.35	79.18	10.44
12	25	531339.5	186238.9	100	2.9	25.67	11.55	2.22
	50	9318678.3	1151306.9	100	8.1	388.28	69.36	5.60
	75	125956770.3	5061350.6	100	24.9	4030.46	302.70	13.32

As can be seen from table 1 the improvement of NQ&M is considerable when processing times start to count, from nine or ten variables onward. Not only is the improvement, theoretically as well as in practice, strongly dependent on the percentage of ones, but also on the number of variables. Moreover the practical improvement factors, although less than theoretically expected, reflect the theoretical factors better with the greater number of variables. For 9 variables and higher and 75% true assignments the practical improvement factors are about 50% or better of the theoretical improvement factors. It can also be inferred from table 1 that NQ&M compares 'almost' only implicants with equal variables. For 10 variables and more the percentage non-equal variable comparisons is less than 0.5%.

If we try to extrapolate the improvements for larger numbers of variables, from table 1 the following relations can be extracted, which can be assumed to approximately hold:

$$c(25,n) = 2.30\ 1.12_{(n-10)}$$

$$c(50,n) = 5.72\ 1.19^{(n-10)}$$

$$c(75,n) = 14.73\ 1.30^{(n-10)}$$

where c(x,n) is the theoretical improvement factor with x% ones and n variables. For example, for the following values of n we get the theoretical factors as specified in table 2.

Table 2 Extrapolated approximate theoretical improvement factors for larger numbers of variables and execution times (seconds) for NQ&M (1 E5 seconds = 24 hours, 3 E7 seconds = 1 year)

	NQ&M Improvement factors				NQ&M Execution times	
#vars	20	30	50	80	20	30
%true						
25	7	20	70	6000	8 E5	8 E11
50	30	180	6000	11 E5	5 E6	5 E12
75	200	3000	5 E5	15 E8	2 E7	2 E13

Inspection of table 2 readily reveals that numbers of variables appreciably greater than 20 cannot be processed in the foreseeable future, by NQ&M, just by the factor time alone. The problem of storing intermediate implicant sets is not even mentioned here.

The above results were obtained for random Boolean functions. However, for our problem we also have a Boolean *expression*. Although I rather randomly entered some expressions for the program, humanly entered expressions are by nature of very restricted length. This seems to have the side effect that the associated functions have much less prime implicants than random functions. For example, the two twelve variable expressions that I entered produced resp. 37 and 73 prime implicants, whereas the expected number of prime implicants for 12 variable random functions of 25, 50 and 75% true assignments is resp. about 1000, 3000 and 6000. On the negative side this implies longer processing times for NQ&M and much longer for OQ&M. For the two 12 variable expressions we got resp. 3 minutes against 1.5 hours (34% true, improvement factor 30) and 14 minutes against 16.3 hours (66% true, improvement factor 70). These execution times are prohibitive for our application, at least with the current hardware. However, the fact that we possess an expression, opens the possibility for a syntactic approach, which is treated in the following section.

3.2 The syntactic method of Quine

In many cases where Truth functions are not given by a function table but by an expression, it proves profitable to directly process the expression itself, without exploiting its semantics in producing the developed normal form as above. The developed normal form is indeed identical to the function table with the function values of 0 left out. Quine (1955) developed the following syntactic algorithm: " (i) *Drop these obvious superfluities* : If one of the clauses of alternation subsumes another, drop the subsuming clause. Also supplant $a \vee \underline{a}\alpha$ by $a \vee \alpha$ (and $\underline{a} \vee a\alpha$ by $\underline{a} \vee \alpha$) where a is a single letter.

(ii) *Adjoin, as an additional clause of alternation, the consensus of two clauses.* ... Operation (ii) is to be regarded as not applying in case the consensus subsumes a clause already present."

Quine (1955, pg 628) proves that repeated application of these two operations to any *non-valid* formula Φ produces precisely all prime implicants of Φ. However, some minor changes in the proof ensure that valid formulas can be included as well, which means that a validity test prior to the application of the algorithm can be omitted. Moreover, the algorithm provides an alternative method for testing validity as an alternative for testing truth for all assignments. We will give the adapted proof below, with the adaptations indicated.

A second remark concerns the superfluity of the second sentence in the formulation of operation (i), which means that the consensus is adjoined and the subsuming longer parent omitted. The implementation of the second sentence as a preliminary step might speed up the algorithm. It is irrelevant for the proof of the algorithm, however.

Before proceeding we need two additional concepts: the *empty monom* denoted by $\mathfrak{m}$ and the *empty expression* denoted by $\mathfrak{E}$. Some authors that mention the empty monom say that it is a valid expression by convention. But this choice is not that arbitrary. Consider the expression Φ with one monom $\alpha = ab\underline{c}$, Φ accepts only the assignment 110 for the variables a, b and c. Now omit c from Φ to get $\Phi = ab$, which accepts 110 and 111 as well. In other words, the assignment for not present variables doesn't matter, it is accepted, whatever value the variable is given. Therefore, to be consistent, a monom with no variables at all must accept all assignments.

The opposite is $\mathfrak{E}$, the empty expression. An expression in DNF accepts an assignment iff one of its monoms does. Therefore $\mathfrak{E}$ cannot accept any assignment, because it does not posses any monom, including the empty monom. $\mathfrak{E}$ and $\mathfrak{m}$ are each others opposites: $\mathfrak{E}$ is inconsistent, and $\mathfrak{m}$ is valid. We have the following lemma:

Lemma 1: A formula Φ is valid iff ħ is a prime implicant.
If ħ is a prime implicant of Φ it is the only one.
Proof: Suppose Φ is valid. ħ clearly implies Φ, because both are valid. Therefore ħ is an implicant of Φ. Because its shortest subclause is ħ itself, ħ is a prime implicant of Φ. Furthermore ħ is subsumed by any monom, and therefore no other implicant can be a prime implicant of Φ.
Next, suppose ħ is a prime implicant of Φ. Because ħ accepts any assignment, Φ accepts any assignment, therefore Φ is valid. ■
The proof of Quine's (1955) theorem can now be reproduced here except for some changes to include valid formulas. We insert comments between {}. The proof proceeds by proving that a formula Φ is still susceptible for an application of operation (i) or (ii) as long as there is a prime implicant α of Φ which is not a clause of Φ.
{ omit the sentence: 'Φ is implied by ..', it is not true for the empty monom, and in the original proof the sentence is redundant. } Since α is a *prime* implicant, it has no letters foreign to Φ; for any such could be dropped without impairing the implication. Moreover, since α is a prime implicant of Φ, not contained in Φ, and each clause of Φ also implies Φ, no clause of Φ is subsumed by α. So there is at least one fundamental formula (α itself, for one) fulfilling these three conditions: (a) it subsumes α, (b) it subsumes no clause of Φ (c) it contains no letters foreign to Φ {slight change}. Let β be a longest fundamental formula fulfilling (a), (b) and (c). Still β will lack some letter d of Φ. (for if β contained all letters of Φ, then, by (b), β would conflict with each clause of Φ, which is impossible because β implies Φ). Now, since β is a longest formula fulfilling (a), (b), and (c), the longer formulas $d\beta$ and $\underline{d}\beta$ must fail to fulfill (b); for they do fulfill (a) and (c). So $d\beta$ and $\underline{d}\beta$ each subsumes a clause of Φ. These subsumed clauses must contain d and $\underline{d}$ respectively, since they were not subsumed by β alone. { replace the next sentences to the end of the paragraph by: } They can be denoted by $d\gamma$ and $\underline{d}\gamma'$ and β subsumes γ and γ'. Clearly Operation (ii) 'Adjoin consensus' is applicable to Φ, because the consensus $\gamma\gamma'$ (minus any duplicate litterals) contains no letter both affirmed and negated, since it is subsumed by a fundamental formula β; and it subsumes no clause of Φ since β subsumed none. If γ = ħ xor γ' = ħ then subsequently operation (i) will be applicable. If γ = ħ and γ' = ħ then conclude that Φ is valid because ħ is an implicant (Lemma 1).

This proves that Operations (i) or (ii) are applicable as long as there is a prime implicant α of Φ, which is not a clause of Φ. { Quine concludes that this implies that (i) and (ii) produce all prime implicants of Φ. In my opinion he proved that it does so only if the algorithm stops. But that it stops does not follow from the proof. } If it stops it clearly produces precisely all prime implicants, because any other clause, that Φ might

contain is an implicant of Φ and therefore subsumes one of the prime implicants and can therefore be omitted by Operation (i). ■
It follows that if a valid disjunction of monoms is processed by Q&M or Syn the empty monom $\mathbb{1}$ is produced.

To see that the alternation of Operations (i) and (ii) stops it is sufficient to show that a monom that is removed by Operation (i) does not reappear in the process. This means that a circularity is impossible. And circularity is the only mechanism in this finite space that can keep the algorithm going for ever. The only way a monom α disappears is by Operation (i) because it subsumes a shorter one α'. This shorter one can only disappear again because it subsumes a shorter α''. As a result α cannot be adjoined again by Operation (ii) because this is precluded by its subsumed followers α', α''...

Our implementation of Syn does not alternate between (i) and (ii), because this produces many unnecessary subsumption tests in (i). First, the list of monoms is sorted according to length, the shortest first. Thereafter (i) is applied, including the combination of (ii) and (i) for monoms of one litteral. Sorting facilitates the subsumption test, because it is known from the start, that if subsumption is true, which of the two subsumes the other. After Operation (i) all pairs of monoms are checked whether they have a consensus. If so then for every monom in the list of monoms, beginning with the first, it is checked whether the consensus subsumes it until the first monom in the list that is longer than the consensus. There the consensus will be inserted in the sorted list, if there was no subsumed monom in the list. From the first monom in the list that is longer than the consensus to the end of the list, Operation (i) is tried. This prevents that over and over again the same pairs of monoms are tested whether one subsumes the other. After the first application of Operation (i), only a newly adjoined consensus can be subsumed. Of course, our implementation follows Quine's (1952) advice to first find all independent separations of a set of prime implicants, and to do all the processing for each separation independently. Two monoms from different *separations* do not share a letter.

A program that applies Syn to an arbitrary Boolean expression Φ, must first transform Φ to an equivalent DNF. Our implementation proceeds by first parsing the expression Φ, which produces a tree structure with operators at its nodes and letters at its leaves. Some relatively straightforward recursive operations on this tree produce a DNF tree of Φ that is easily transformed to a disjunction of monoms that is equivalent to Φ. More details are given in Verstralen (1988). On a series of test expressions, the time for this transformation seemed negligible.

The algorithm Syn is very fast and powerful. The expressions we tested with our implementation of Syn, left no doubt whatsoever about this. All expressions except one, were processed in less then one second. The twelve variable expression that took OQ&M 16 hours and NQ&M 16 minutes was processed by Syn in 0.28 seconds. The exception was the twelve variable expression that took the OQ&M 1.5 hours and NQ&M 3.5 minutes. It took Syn 7.5 seconds. An improvement factor of about 30 relative to NQ&M and of 900 relative OQ&M.

4 Algorithmic search for the cheapest equivalent combination of prime implicants

After having found the complete set PI(Φ) of prime implicants of a function Φ, the next step in the simplification process is to find an equivalent cheapest combination of them. Cheapest is defined here in relation to the following cost function C. Denote the number of variables in α by v(α). Denote the number of prime implicants in a DNF representation D(Φ) van Φ with p(D). C is any function that fulfills the following conditions:

IF $p(D_1) > p(D_2)$ THEN $C(D_1) > C(D_2)$
IF ($p(D_1) = p(D_2)$ AND $\Sigma_1 v(\alpha) > \Sigma_2 v(\alpha)$) THEN $C(D_1) > C(D_2)$
where Σ_i means summation over the monoms in D_i.

As Paul (1974) and Wegener (1987) point out, the search for the cheapest subset CPI(Φ) of PI(Φ) that is equivalent with PI(Φ) can be viewed as a Set Covering problem. Call T the set of assignments that is accepted by Φ and so by PI(Φ). Each member α of PI(Φ) covers a subset T(α) of T. Let S = $\{ T(\alpha) \mid \alpha \in PI(\Phi) \}$. The problem is to find a cheapest subset of S that covers T. Because the sets T and T(α) tend to be very large already for relatively modest expressions, we first explored another route, to solve this combinatorial NP-complete problem (Paul, 1974).

The second part of Quine (1955) offers a start. We will describe his algorithm here and add two theorems of which especially the last appreciably contributes to the simplicity of the algorithm. Moreover, I think that both theorems are also interesting from a theoretical point of view.

Let Φ' be a DNF representation of a truth function consisting exactly of all its prime implicants. Let α' be a redundant prime implicant of Φ'. To see whether α' is redundant one tests whether α' implies its complement $\Phi'\backslash\alpha'$. Quine (1955): 'This may be quickly decided by testing $\Phi'\backslash\alpha'$ for truth when the letters affirmed by α' are marked true and those negated in α' are marked false.'

Let $\Phi'(\alpha')$ denote $\Phi'\backslash\alpha'$ in which α' is 'substituted' as indicated. Normally by "testing $\Phi'(\alpha')$ for truth" is meant: enumerate all assignments of the remaining variables and check whether $\Phi'(\alpha')$ accepts them all. But this procedure can be simplified considerably. First it helps to remark that all monoms of $\Phi'(\alpha')$ of which the original in Φ' did not agree with α' are false for every assignment of their remaining variables. Therefore they do not contribute to the validity of $\Phi'(\alpha')$ and can therefore be omitted from $\Phi'(\alpha')$ from the start. This not only reduces the number of monoms that have to be tested, but, in general, also the number of variables over which all assignments have to be considered. Because the number of assignments equals

2^n, where n is the number of variables, each neglected variable reduces testing time by a factor 2.

Therefore, to test the redundancy of α', all implicants of Φ' that do not agree with α' are discarded. The monoms that agree with α' and share one or more litterals with it, can also be simplified. The part of a monom that agrees with α' is by the 'substitution' collectively set to TRUE. Consequently the monom accepts an assignment iff its part accepts that it doesn't share with α'. The substituted variables that the implicants share with α' can be omitted from $\Phi'(\alpha')$, without changing the truth function connected with $\Phi'(\alpha')$. These simplifications produce the expression Φ, which is of course again a DNF and we have: α' is redundant iff Φ is valid. Call A' the set of implicants of Φ' that agree with α' and share at least one letter with α', call the same set A when the shared litterals with α' are left out. If α is a monom in A then denote with α' its original in A'. Call B the set of monoms of Φ' that do not share a letter with α'. Clearly B = B'. Φ is the disjunction of A and B. Call I the set of variables of A. We will prove the following theorems:

1. If I does not contain all variables of α' then α' is not redundant.
   An algorithm could start with testing whether A' contains all litterals of α'. Although the test itself is clearly a shortcut, we did not succeed in preventing the associated administrative overhead to consume the time gained. But, when parallel processing is possible, the property is useful.
2. Φ is valid if A is valid.
   The helpful consequence of this theorem in devising an algorithm is that we can limit the investigation of the validity of Φ to A, and to the variables in I.

Because in the transition from $\Phi'$to Φ only the monoms in A have changed, it is natural investigate whether the monoms in B can be dispensed with in the construction of the empty monom in Φ as a validity test.

Proof of theorem 1:

Assume that α' is redundant, the variables of α' do not all occur in A', and let z be a litteral in α' not in A'. Then $\Phi'\backslash\alpha'$ accepts all assignments that α' accepts, because α' is redundant. But $\Phi'\backslash\alpha'$ also accepts all assignments that are accepted by α' and -z, because there is no monom in $\Phi'\backslash\alpha'$ to notice the change in the assignment for the letter in z. This means that α' is not prime because $\alpha'\backslash z$ is also an implicant of Φ', contrary to the assumption. Therefore, for α' to be a redundant prime implicant all litterals of α' must occur in A'. In particular A cannot be empty. ■

Proof of theorem 2:

If A is valid then surely Φ is.

Suppose that Φ is valid, we are ready if we show that any assignment that is accepted by B is accepted by A as well. Suppose B is not empty and contains a prime implicant β of Φ'. Then Φ must have an implicant α that contains no letters foreign to β and either (1) is subsumed by β because it contains less letters or (2) forms a conjugate parent pair with β with a consensus γ. Otherwise Φ would not be valid. Moreover, α cannot be an implicant of B, for then β would not be a prime implicant of Φ'. Therefore, α is an implicant of A. It follows that (2) reduces to (1), because the consensus $\gamma'$of α' and β must be an implicant of A′ and therefore γ an implicant of A. And γ necessarily is subsumed by β, because β is a parent of γ and γ does not contain letters foreign to β, like its other parent α. It follows that if B accepts an assignment then so does A. ■
Therefore, B can be neglected in proving the validity of Φ. Again, the greatest gain is achieved by the reduction of the number of variables.

Each member of the set RPI(Φ') of prime implicants that is redundant can only be dismissed individually. As soon as one of them is dismissed, it is not certain that the rest still is, because there is one monom less in their complement. But we are certain that those prime implicants that cannot even be dismissed individually must be a member of the cheapest representation. The set of prime implicants that cannot be dismissed individually is called the *core*. Because we are certain that the core is in the cheapest representation, we can also be certain that those redundant prime implicants that imply the core are redundant in every representation and therefore can surely be dismissed. Of the rest of RPI(Φ) the most 'expensive' redundant subset must be found. Here we are left again with an NP-complete combinatorial problem. In our implementation we do in fact a so called intelligent exhaustive search, that can be interrupted at any moment to give the best solution found.

We proceed as follows. Sort RPI(Φ) áccording to length, the longest first. If RPI(Φ) is not empty a best one element solution is to omit the first element of RPI(Φ). Therefore, a better solution must contain two elements of RPI(Φ). Starting with the first two elements of RPI all two element-combinations are enumerated until a pair is found that implies its complement. A subset SR of RPI(Φ) is redundant iff each member of SR implies PI(Φ)\SR. Here again theorem 3 pays its service. Before continuing with a search among all triples if RPI(Φ) it is good to realize that all already found nonredundant pairs surely cannot be part of a redundant triple. Because one of them did not imply the complement of the pair, this implicant will certainly not imply a complement with less monoms. Therefore the enumeration with triples is continued in such a way that the already failed pairs are not included.

Nevertheless, complete search for a reasonable complex expression is not a realistic option. It is, therefore, necessary to revert to heuristic search. In stead of development of heuristic procedures in the above Quinean approach it appeared to be easier to implement a heuristic procedure from the Set Covering literature. We chose the Chvàtal algorithm, Syslo e.a. 1983, pg 215. This greedy algorithm searches for the prime implicant α that covers its set of true assignments with the least cost per assignment and makes it a member of the cover. Next it deletes all assignments that are covered by α and considers the rest problem, etc..

The next table gives an overview of the results of out implementation of the Chvàtal algorithm along with results of Q&M and Syn. |PI| denotes the number of prime implicants and ||PI|| the size of the smallest cover according to Chvàtal's algorithm.

Table 3 Processing times (h:m:s) for alternative routines for construction of the set of prime implicants and of Chvàtal's SC algorithm

Exp	%1	Name	#vars	Syn	NQ&M	OQ&M	fac	Chvàtal	|PI|	||PI||
1	84	Hammer	6	0.22	0.55	1.04	2	0.11	10	6
2	76	NegAet	8	0.11	4.01	21.42	5	0.55	10	10
3	34	ExtwT10	10	2.97	12.20	1:38.64	8	5.99	37	14
4	34	ExtwT	12	7.69	3:05.32	1:37:49.12	32	5.31	73	28
5	66	-ExtwT	12	0.17	14:02.61	16:20:45.00	70	3.24	9	6

The expressions (The negation of the expression is processed):

1 (1*2*-5) + (1*2*-6) +
(3*-4*6) + (-1*3*-4) + (3*-4*5) + (-1*3*6) + (3*5*6) + (-1*-4*-6) +
(-4*5*6) + (1*-2*6) + (-2*3*6) + (-2*4*6) + (1*-2*3) +
(1*3*6) + (-1*-6) + (3*-4*-6)

2 -((1+2) * (3+4)) + (1+5*-7) -> (6*-8*3)

3 ((-((1+2) * (3+4))) + (1+5) ->
(6*8) + ((9*10) -> (-2+8))) ->
(1*((-2+3+4)*(4+5+6)) -> (4*8*7*6))

4 = 3 except underlined = (-11+12)

5 = -4

From table 3 it can be concluded that the Chvàtal procedure is very fast, and, because these kind of SC problems in general have relatively few columns (elements of PI) in relation to the number of rows (true assignments), probably will result in an optimal cover.

5 Conclusion

It seems that the presented results promise the application of Boolean expressions as constraints in LP programs. However, expression 4 results in 28 prime implicants. This means that the current approach would add 28 inequality constraints to a test construction problem with this constraint. This shows that in an average test construction problem, where probably several interitem constraints must be considered, the size of the additional number of linear inequalities will appreciably hamper the algorithms for test construction. Therefore, there are two directions for further research: (1) devise algorithms that can handle a greater number of constraints in acceptable time and (2) devise an algorithm for transformation of Boolean expressions into less linear inequalities. The second direction offers some direct prospects because the type of linear inequalities that are produced by direct translation of a CNF are rather constrained.

References

Cito. OTD. A computer program for Optimal Test Design.
Arnhem: Cito, Department for Research and Psychometrics, (1990).

Hammer, P.L. Boolean procedures for bivalent programming.
In: Hammer,P.L. and Zoutendijk,G. *Mathematical Programming in Theory and Practice*. Amsterdam: North Holland, (1974).

McCluskey, E.J., Jr. Minimization of Boolean functions.
The Bell System Technical Journal, 35 (1956), 1417-1444.

Mileto, F. and Putzolu, G. Statistical complexity of algorithms for Boolean function minimization. *Journal of the Association for Computing Machinery, 12* (1965), 364-375.

Paul, W.J. Boolesche Minimalpolynome und Überdeckungsprobleme.
Acta Informatica, 4 (1974), 321-336.

Quine, W.V. The problem of simplifying truth functions.
American Mathematical Monthly, 59 (1952), 521-531.

Quine, W.V. A way to simplify truth functions.
American Mathematical Monthly, 62 (1955), 627-631.

Syslo,M.M., Deo,N. and Kowalik,J.S. *Discrete Optimization Algorithms*.
Englewood Cliffs, NJ: Prentice-Hall, (1983).

Theunissen, T.J.J.M. Binary programming and test design.
Psychometrika, 50 (1985), 411-420.

Verstralen, H.H.F.M. *Processing of Boolean functions in the context of test construction*. Cito, Arnhem, (1988).

Wegener, I. *The Complexity of Boolean Functions*.
New York: Wiley, (1987).

Titles of other research reports