

Two-level logic minimization: an overview

Olivier Coudert

DEC, Paris Research Laboratory,
85 Avenue Victor Hugo,
92500 Rueil Malmaison, France

INTEGRATION, the VLSI journal, **17-2**, pp. 97–140, October 1994.

Contents

1	Introduction	5
2	Minimization over a product space	6
2.1	Product space and Boolean functions	6
2.2	Implicants, primality, essentiality	8
2.3	Example	9
2.4	Set covering problem	10
3	The Quine–McCluskey procedure	10
3.1	Prime implicant computation	11
3.2	Reduction processes	12
3.2.1	Partitioning	12
3.2.2	Essentiality	12
3.2.3	Dominance relations	12
3.2.4	Gimpel’s reduction	13
3.3	Branch and bound	14
3.3.1	Branching	15
3.3.2	Lower bound	15
3.4	Conclusion	15
4	Formalizing covering matrix reduction	16
4.1	Strong reduction with dominance relations	16
4.2	Transposing functions	17
4.3	Conclusion	19
5	The signature cube based procedure	19
6	Set covering problem over lattices	21
6.1	Lattices	22
6.2	Solving set covering problems over a lattice	22
6.2.1	Examples and intuition	22

6.2.2	The transposing function $\tau_{W, \sqsubseteq}$	24
6.2.3	Essential elements	25
6.2.4	Recovering the original solutions	25
7	A new two-level logic minimization procedure	26
7.1	Implicit prime implicant computation	27
7.2	Computing Q	28
7.3	Computing $\max_{\subseteq} \tau_P(Q)$	28
7.4	Computing $\max_{\subseteq} \tau_Q(P)$	29
7.5	Improvements for solving the covering matrix	30
7.5.1	Pruning with the left-hand side lower bound	31
7.5.2	The limit lower bound theorem	31
7.6	Conclusion	32
8	Experimental results	32
9	Conclusion	35
A	Binary Decision Diagrams	35
B	Combinational sets	37
	Bibliographie	39

List of Figures

1	A total Boolean function on $\{0, 1\}^3$ and its associated on-set.	7
2	Karnaugh map.	9
3	Karnaugh maps of an irredundant and two minimal prime covers.	10
4	A covering matrix.	11
5	Reduction by essentiality.	12
6	Reduction by domination on X	13
7	Reduction by domination on Y	13
8	Gimpel's reduction on x	14
9	Strong dominance reduction with equivalence classes.	17
10	Universal transposing functions based reduction.	19
11	Signature cubes and minterm dominance.	20
12	Naive computation of $\max_{\subseteq} \sigma(f)$	21
13	Computing $\tau_{X, \sqsubseteq}(Y)$ on a lattice.	23
14	Computing $\tau_{Y, \supseteq}(X)$ on a lattice.	23
15	Computing the essential elements.	24
16	Algorithm <i>Prime</i>	28
17	Algorithm <i>DiveMinIntoP</i>	28
18	Algorithm <i>NotSubSet</i>	29
19	Algorithm <i>MaxTauP</i>	29
20	Algorithm <i>SupSet</i>	29
21	Algorithm <i>MaxTauQ</i>	30

22	Espresso-Exact, Espresso-Signature, and Scherzo.	34
23	Shannon tree and BDD of the function $x_1(x_3 \oplus x_4) + x_2(x_3 \Leftrightarrow x_4)$	36
24	Reduction rules rewriting a Shannon tree into a BDD.	36
25	Reduction rules rewriting a set of sparse combinations into a CS.	38
26	CS of the set of products $\{b\bar{d}, \bar{a}b\bar{c}d, acd\}$	39

List of Tables

1	Computation time for 33 examples	44
2	The hard Espresso-Exact problems	45
3	Hard espresso examples with Espresso-Signature and Scherzo	46
4	Results of Scherzo on hard examples from the MIS and ISCAS benchmarks	47

Notation

X, Y, Z, \dots	Sets.
x, y, z, \dots	Elements of a set.
$\cup, \cap, -, \times$	Union, intersection, set difference, cartesian product.
f	Function.
f^{-1}	Inverse function of f .
2^S	Power set of a set S .
$f(S)$	Image of a function f on a set S .
i, j, k, n, m	Integers, indices.
\cdot	Ordered cartesian product.
\mathcal{L}	Product space, i.e., set $\prod_k L_k = L_1 \cdot L_2 \cdots$.
l_k	Literal, i.e., subset of a set L_k .
$\prod_k l_k$	Product built on a product space \mathcal{L} .
$\mathcal{P}(\mathcal{L})$	Set of all products built on a a product space \mathcal{L} .
P, Q	Sets of products.
p, q, r	Products.
P_{l_k}	Set of products $\{p \mid l_k \cdot p \in P\}$.
\mathcal{B}	The particular product space $\{0, 1\}^n$.
minterm	An element of \mathcal{B} .
a, b, c, d, x, x_k	Boolean variables and positive literals.
\bar{x}	Negation of the Boolean variable x , negative literal.
$\bar{a}bcd, \bar{x}_1x_2$	Products built on \mathcal{B} .
$P_{l_k}, P_{\bar{x}_k}, P_{x_k}$	Canonical decomposition of a set of products built on \mathcal{B} .
f^1	On-set of a partial boolean function f .
f^{1*}	Union of the on-set and the don't care set of f .
f_{l_k}	Largest set such that $l_k \cdot f_{l_k} \subseteq f$.
$f_{\bar{x}}, f_x$	Shannon expansion of the Boolean function f w.r.t a Boolean variable x .
$Prime(f)$	Set of prime implicants of f .
X, Y	Set of rows and set of columns.
x, x', y, y'	Rows and columns.
$Cost(y)$	Cost of a column.
R	Subset of $X \times Y$.

$\langle X, Y, R \rangle$	Set covering problem.
E	Set of essential columns.
$Sol(C)$	Set of minimal solution of a set covering problem C .
\prec	Quasi order.
\preceq, \sqsubseteq	Partial order.
\equiv	Equivalence relation.
π	Map an element onto its equivalence class.
ρ	Map an equivalence class onto one of its element.
\rightarrow	Rewriting rule, mapping.
Z/\equiv	Quotient set of Z by an equivalence relation \equiv .
\prec_Y	Dominance relation on rows.
\prec_X	Dominance relation on columns.
\max	Maximal elements of a set w.r.t. a quasi or partial order.
τ	Transposing function.
$\sigma(x)$	Signature cube of a minterm x .
(Z, \sqsubseteq)	Lattice.
\sup	Least upper bound of a set.
\inf	Greatest lower bound of a set.
\otimes	Inner product, i.e., $P \otimes Q = \{p \cdot q \mid p \in P, q \in Q\}$.

Two-level logic minimization: an overview

INTEGRATION, the VLSI journal, **17-2**, pp. 97–140, October 1994.

Olivier Coudert

Abstract

Fourty years ago Quine noted that finding a procedure that computes a minimal sum of products for a given propositional formula is very complex, even though propositional formulas are fairly simple. Since this early work, this problem, known as two-level logic minimization, has attracted much attention. It arises in several fields of computer science, e.g., in logic synthesis, reliability analysis, and automated reasoning. This paper exposes the classical approach for two-level logic minimization, and presents the recent developments that overcome the limitations of the procedures proposed in the past. We show that these new techniques yield a minimizer that is 10 up to 50 times faster than the previously best known ones, and that is able to handle more complex functions.

Keyword

Set covering problem, Two-level logic minimization, Binary Decision Diagram, Combinational Set, Transposing function

1 Introduction

In 1952, Quine observed that finding a procedure that reduces propositional formulas to their “smallest” equivalent forms is complex, even though Propositional Logic is quite “simple” [45]. Quine was interested in propositional formulas represented in disjunctive normal form. A disjunctive normal form, also called sum-of-products, is a disjunction of products. A product is a conjunction of literals (each literal is a propositional variable or its negation). For instance, a and \bar{b} are literals, $a\bar{b}c$ is a product, and $a\bar{b}c + acd + \bar{b}d$ is a sum-of-products. The problem is to find, given a Boolean function f , the minimal sum-of-products that represents f . This problem is known as two-level logic minimization.

Consider the following sum-of-products:

$$abd + acd + a\bar{b}c\bar{d} + \bar{a}b\bar{c} + \bar{a}c\bar{d} + \bar{b}c\bar{d}.$$

A simpler equivalent sum-of-products can be found by merging some products, or by removing some literals from some products. For instance, abd logically implies $acd + \bar{b}c\bar{d}$ (one says that $acd + \bar{b}c\bar{d}$ contains abd), thus abd can be removed from the expression. Also $\bar{a}c\bar{d}$ contains the minterm $a\bar{b}c\bar{d}$, so the product $a\bar{b}c\bar{d}$ in the original expression can be replaced with $\bar{b}c\bar{d}$. We then obtain:

$$acd + \bar{a}b\bar{c} + \bar{a}c\bar{d} + \bar{b}c\bar{d} + \bar{b}c\bar{d}.$$

At this step we can neither remove any product from this sum-of-products, nor a literal from any product, without modifying the function the original expression represents. This sum-of-products is locally minimal, or *irredundant*. However there are smaller sums-of-products. Indeed,

$$acd + \bar{a}\bar{b}\bar{d} + \bar{b}\bar{c}d + \bar{b}\bar{c}\bar{d} \quad \text{and} \quad abd + \bar{a}\bar{b}c + \bar{a}b\bar{c} + \bar{a}\bar{c}\bar{d}$$

are the two absolute minimal sums-of-products representations of the original expression. Note that in these two minimal sums-of-products, some products do not occur in the original expression, e.g., $\bar{a}\bar{b}c$ and $\bar{b}\bar{c}\bar{d}$, even as subexpressions, e.g., $\bar{a}\bar{b}\bar{d}$.

Computing an irredundant or minimal sum-of-products has several applications in computer science. In logic synthesis a minimal sum-of-products provides the user with an efficient implementation of a single or multi-output Boolean functions with NOT, AND and OR gates [26, 6, 55]. In reliability analysis, a sum-of-products is a way for either exhaustively, or concisely, representing the causes of failure of a system [25, 18]. In automated reasoning, a sum-of-products can be either a set of minimal demonstrations, or the normalization of a formula into a disjunctive normal form for further computations [22, 30, 31, 49, 50, 39].

Since Quine established the problem of computing minimal sums-of-products in the 1950's [45], efforts have been made to discover efficient minimization procedures [48, 34, 3, 26, 6, 56, 57, 37, 27]. These procedure encounter two bottlenecks. The first one is the number of prime implicants they have to generate, the second one is that solving a set covering problem is itself NP-hard. Most of the procedures proposed in the past are limited by huge numbers of products one has to manipulate for some functions, typically their number of *prime implicants*. This paper shows how recent developments yield new techniques for computing minimal sum-of-products, whose complexities no longer depend on the numbers of products to manipulate.

This paper aims at presenting the standart techniques used to compute a minimal cost sum-of-products representation of a given Boolean function, as well as the last recent developments in this field. Section 2 defines the two-level minimization problem in the more general terms of a paving problem, and explains some useful notions such as implicants, primality, essentiality, and set covering problems. Section 3 presents the well known *Quine–McCluskey algorithm* that is widely used for two-level logic minimization. Section 4 formalizes the concept of *dominance relation* used by the Quine–McCluskey algorithm and introduces the notion of *transposing function*. Section 5 presents the *signature cube* based minimization method that is a direct consequence of the results derived in Section 4. Section 6 addresses the resolution of a generic set covering problem, namely set covering problem over a lattice, and presents an original minimization algorithm. Section 7 is a direct application of the results introduced in Section 6 to two-level logic minimization. Section 8 presents and discusses some experimental results.

2 Minimization over a product space

This section defines the two-level logic minimization problem in terms of a paving problem, and introduces some notions that will be used in the sequel.

2.1 Product space and Boolean functions

The paving problem can be expressed as follows. Given a set \mathcal{L} , given a collection $\mathcal{P}(\mathcal{L})$ of some of its subsets (i.e., $\mathcal{P}(\mathcal{L}) \subseteq 2^{\mathcal{L}}$), how can we represent a subset f of \mathcal{L} as a (minimal cost) union of some elements of $\mathcal{P}(\mathcal{L})$? The two-level logic minimization of a function f is a paving problem where $\mathcal{P}(\mathcal{L})$ is the set of all products built on a product space \mathcal{L} , as it is explained below.

$$\begin{aligned}
f(x_1, x_2, x_3) &= x_1(x_2 + x_3) + \overline{x_1}x_2\overline{x_3}, \\
f^1 &= \{(0, 1, 0), (1, 0, 1), (1, 1, 0), (1, 1, 1)\}
\end{aligned}$$

Figure 1: A total Boolean function on $\{0, 1\}^3$ and its associated on-set.

We use the symbol “ \cdot ” as a cartesian product that orders the sets on which it operates with respect to their indices, e.g., both $L_1 \cdot L_2$ and $L_2 \cdot L_1$ are equal to $L_1 \times L_2$. A *product space* \mathcal{L} is a cartesian product $\prod_k L_k = L_1 \cdot L_2 \cdots$. A *literal* is a subset l_k of L_k . A *product* is a cartesian product of literals $\prod_k l_k$. The set of all products built on a product space \mathcal{L} will be noted $\mathcal{P}(\mathcal{L})$. The structure $(\mathcal{P}(\mathcal{L}), \subseteq)$ is a complete lattice. Note that any subset of \mathcal{L} can be represented as a union of products.

We will particularly study the case where the product space is $\mathcal{B} = \{0, 1\}^n$. Its set of products is denoted $\mathcal{P}(\mathcal{B})$. The literal l_k , that can be $\{\}$, $\{0\}$, $\{1\}$, or $\{0, 1\}$, will be denoted 0_k , $\overline{x_k}$, x_k , and 1_k respectively. For instance the product $\{1\} \times \{0, 1\} \times \{0\}$ built on $\{0, 1\}^3$ is $x_1\overline{x_3}$ (we will omit the \cdot 's and 1_k 's in a product built on \mathcal{B}). A product built on \mathcal{B} is then identified with its *characteristic function* (see below). To illustrate our notations, we have on $\{0, 1\}^4$: $x_1\overline{x_2}x_4 \subseteq x_1x_4$; $\max_{\subseteq} \{x_1, x_1\overline{x_2}, \overline{x_1}x_2\overline{x_3}, x_2\overline{x_3}, x_4\} = \{x_1, x_2\overline{x_3}, x_4\}$; $\inf_{\subseteq} \{x_1x_2, x_2\overline{x_3}\} = x_1x_2\overline{x_3}$; $\sup_{\subseteq} \{x_1x_2, x_2\overline{x_3}\} = x_2$.

A (multi-valued input variable) Boolean function is a (partial, or incompletely specified) function from \mathcal{L} into $\{0, 1\}$. A Boolean function f can be seen as a total function from \mathcal{L} into $\{0, 1, *\}$, where $f^{-1}(*)$, also called the don't-care set of f , is the set on which f is not properly defined. The set $f^{-1}(1)$, i.e., the subset of \mathcal{L} on which f evaluates to 1, is called the on-set of f . To shorten the notation, we will denote f^1 the on-set, and f^{1*} the set $f^{-1}(1) \cup f^{-1}(*)$. In the sequel, we will omit the term “partial” when speaking of Boolean functions, and we will specify whether the function is total when necessary. A vectorial Boolean function, or multi-output Boolean function, is a vector $[f_1 \cdots f_m]$ of m Boolean functions f_k , and is considered as a total function from \mathcal{L} into $\{0, 1, *\}^m$.

The characteristic function of a set is a function that evaluates to 1 on the elements of this set, and to 0 elsewhere. Since the characteristic function of the on-set of a total Boolean function f is f itself, we will make no distinction between a total Boolean function and the on-set it represents. Figure 1 shows a total Boolean function defined on $\{0, 1\}^3$.

A set of products P is a *cover* of a Boolean function f iff $f^1 \subseteq (\bigcup_{p \in P} p) \subseteq f^{1*}$. A set of products P is a *cover* of a vectorial Boolean function $[f_1 \cdots f_m]$ iff there exists a subset P_k of P that is a cover of f_k for $1 \leq k \leq m$. We can now define the minimization problem we study in this paper.

Definition 1 *Let $f = [f_1 \cdots f_m]$ be a vectorial Boolean function from \mathcal{L} into $\{0, 1, *\}^m$. The minimization of f consists in finding a set P of products built on \mathcal{L} that covers f and that minimizes a given cost function.*

Actually this problem can be reduced to the case $m = 1$ [60]. Let ff be the (multi-valued input) Boolean function defined from the product space $\mathcal{L} \times \{1, \dots, m\}$ into $\{0, 1, *\}$ by $ff(x, k) = f_k(x)$, where x belongs to \mathcal{L} , and k to $\{1, \dots, m\}$. Then a minimal cover of f can be obtained from a minimal cover P of ff by removing from the products of P the literals associated with $\{1, \dots, m\}$. Moreover the cover of each function f_k is obtained by taking products of P whose literal associated with $\{1, \dots, m\}$ contains k . For this reason, we only consider single-output Boolean function in the sequel.

In the case of two-level logic minimization, namely the case where the product space is $\mathcal{B} = \{0, 1\}^n$, one can even reduce the minimization of the vectorial Boolean function f defined from $\{0, 1\}^n$ into

$\{0, 1, *\}^m$ to the minimization of a Boolean function ff defined from $\{0, 1\}^{n+m}$ into $\{0, 1, *\}$ by:

$$ff^{1*}(x, y) = \left(\bigwedge_{k=1}^m (y_k \Rightarrow f_k^{1*}(x)) \right),$$

$$ff^1(x, y) = \left(\bigvee_{k=1}^m (y_k \wedge \left(\bigwedge_{\substack{1 \leq j \leq m \\ j \neq k}} \neg y_j \right) \wedge f_k^1(x)) \right)$$

where x takes its value in $\{0, 1\}^n$, and y in $\{0, 1\}^m$ [2, 20, 16].

For arbitrary cost functions one may need to consider all covers of f , which may be computationally infeasible. Therefore we add the following natural constraints on the cost function, which hold for real-life minimization problems.

Definition 2 *The cost function Cost that applies on (sets of) products is such that¹:*

- *Cost is positive and additive, i.e.,*
 $Cost(P) \geq 0$ and $Cost(P) = \sum_{p \in P} Cost(p)$.
- *A product costs no more than any product it contains, i.e.,*
 $p \subseteq p' \Rightarrow Cost(p') \leq Cost(p)$.

For most of the applications, the cost of a product is a linear function of its literals. In the examples shown in the sequel, we will consider the cost of a product as 1.

2.2 Implicants, primality, essentiality

Let f be a Boolean function defined on the product space \mathcal{L} . An element p of $\mathcal{P}(\mathcal{L})$ is an *implicant* of f if and only if (iff) $p \subseteq f^{1*}$. The set of implicants of a Boolean function f is trivially a cover of f since each element of f^1 , which belongs to $\mathcal{P}(\mathcal{L})$, is an implicant of f . For the purpose of two-level logic minimization, Quine introduced the key notion of *prime implicant* that he defined as follows [45]:

“[A] prime implicant of a formula f [is] a fundamental formula [i.e., a product] that logically implies f but ceases to when deprived of any one literal”

In other words, the set of prime implicants of f , which we will note $Prime(f)$, is the set of maximal implicants with respect to \subseteq . For instance $x_1x_2x_3$ is an implicant of the function given in Figure 1, but it is not a prime implicant because it is contained by the implicant x_1x_2 .

It is clear that a Boolean function f is covered by the set of all its prime implicants. Moreover, with the constraints we put on the cost functions, all minimal covers of f are composed of prime implicants. Quine also introduced the notion of *essential* prime implicant [35, 47]. A prime implicant of f is essential iff it is the only prime implicant of f that contains an element of f^1 . Thus essential prime implicants necessarily occur in any minimal cover of f .

One of the difficulties of two-level logic minimization is that the number of prime implicants of a Boolean function can be very large. Indeed the number of prime implicants can be exponential with respect to the number of variables. First Quine showed that the number of prime implicants of a function can exceed its number of minterms, i.e., assignments of its variables that value it to 1 [46]. It

¹For any function f defined on a set S , we denote by $f(S)$ the *image* of f on a subset S of X .

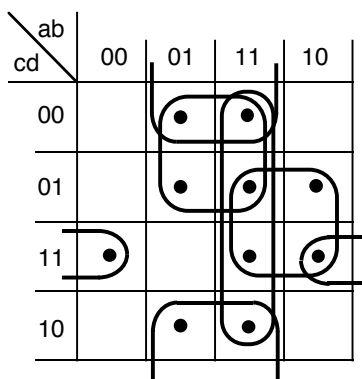


Figure 2: Karnaugh map.

is shown in [13] that the number of prime implicants of functions of n variables is at most $O(3^n/\sqrt{n})$ and can be at least $\Omega(3^n/n)$. In [37] is even shown a function that has a minimal sum-of-products made of n products, but that has $2^n - 1$ prime implicants.

A cover P of a Boolean function f is *irredundant* iff there does not exist any proper subset of P that covers f . A prime and irredundant cover P of f is *locally* minimal in the sense that if one eliminates a literal or a product from P one does no longer have a cover of f . Any minimal prime cover is necessarily prime and irredundant, but the converse is of course false. Irredundant prime covers are nonetheless interesting because it can be much less costly to compute such a locally minimal cover than an absolute minimal one. Most of the techniques developed for exact two-level logic minimization can be modified so that they produces irredundant prime cover with a lower computational cost. Very different irredundant prime cover computation procedures can be found in [6, 7, 17, 43, 57].

2.3 Example

We illustrate the notions that have been introduced with Karnaugh maps [29]. A Karnaugh map is a matrix that provides a planar representation of elements of the product space \mathcal{B} so that products appear as rectangles of the matrix. Each element of the matrix corresponds to an element of \mathcal{B} . Variables are partitioned in two ordered sets, one for the rows of the matrix, and the other for its columns. The rows and columns are labeled by the values of the corresponding variables so that two consecutive rows or columns differ in only one variable value. With this representation, a product of \mathcal{B} is a rectangle whose width and height are powers of 2.

Figure 2 shows the Karnaugh map of a function f that evaluates to 1 for the squares with a black dot. An implicant of f is a rectangle whose width and height are powers of 2, and that only covers black dots. An implicant is prime if it is not contained by another implicant. Here there are 5 prime implicants, i.e., from top to bottom and from left to right: $b\bar{d}$, $b\bar{c}$, ab , ad , and $\bar{b}cd$. All of these primes are essential except ab that covers dots already covered by other primes. Thus the minimal sum-of-products representation of this function is $b\bar{d} + b\bar{c} + ad + \bar{b}cd$. Clearly such graphical methods are of limited utility in solving more complex two-level logic minimization problems.

Figure 3 shows the Karnaugh map of the function presented in the introduction, with some different covers. The first one is the irredundant prime cover with 5 prime implicants, the two other ones are the two minimal covers made of 4 prime implicants.

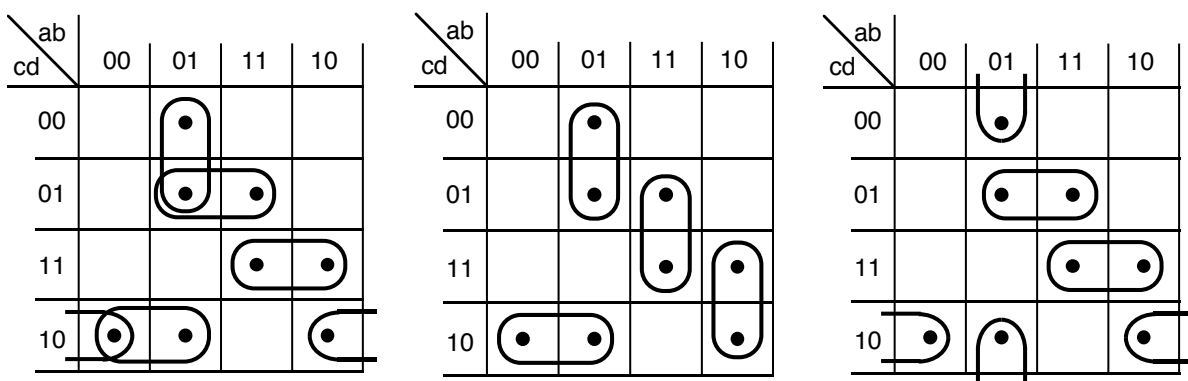


Figure 3: Karnaugh maps of an irredundant and two minimal prime covers.

2.4 Set covering problem

Sum-of-products minimization is simply a set covering problem. This section introduces notation relevant to set covering problems.

We say that a subset R of $X \times Y$ is a relation defined on set $X \times Y$. We will write $x R y$ when $(x, y) \in R$.

Definition 3 Let X and Y be two sets, R be a relation defined on $X \times Y$, and $Cost$ a cost function that applies on subsets of Y . The set covering problem $\langle X, Y, R \rangle$ consists of finding a minimal cost subset S of Y such that for any x of X , there exists an element y of S with $x R y$.

In the context of set covering, we say that y covers x when $x R y$. We will note $Sol(C)$ the set of all minimal solutions of a set covering problem C . We say that a set covering problem C_2 is *weakly equivalent* to a set covering problem C_1 if any solution of C_2 can be used to build some minimal solutions of C_1 . We say that C_1 and C_2 are *strongly equivalent* if there is a way of computing $Sol(C_1)$ with $Sol(C_2)$ and vice versa.

It is convenient to illustrate a set covering problem using a matrix, called its *covering matrix*. The matrix associated with the set covering problem $\langle X, Y, R \rangle$ has rows labeled with elements of X and columns labeled with elements of Y , such that the element $[x, y]$ of the matrix is equal to 1 iff $x R y$. The set covering problem $\langle X, Y, R \rangle$ consists in finding a minimal cost subset of columns that cover all the rows.

The two-level logic minimization of a Boolean function f consists in solving the set covering problem $\langle f^1, Prime(f^{1*}), \in \rangle$. Figure 4 shows the covering matrix associated with the two-level minimization of the function $f(a, b, c, d)$ shown on Figure 2, where $Cost$ is the number of products. It is easy to determine essential prime implicants with a covering matrix: these are the ones that label a column containing the unique “1” occurring in some row. For instance, $b\overline{d}$ is essential because it is the only prime that covers the minterm “0110”. Also one can check that ab is the only prime implicant that is not essential, since all elements it covers are also covered by some other primes.

3 The Quine–McCluskey procedure

The Quine–McCluskey procedure performs the following tasks to compute a minimal cost sum-of-products representation of a Boolean function f :

\in	$b\bar{d}$	ab	$b\bar{c}$	ad	$\bar{b}cd$
0100	1		1		
1100	1	1	1		
0101			1		
1101		1	1	1	
1001				1	
0011					1
1111		1		1	
1011				1	1
0110	1				
1110	1	1			

Figure 4: A covering matrix.

- (a) Compute all the prime implicants of f .
- (b) Build the covering matrix consisting of rows labeled by the minterms of f and of columns labeled by the prime implicants of f .
- (c) Find a subset of columns that cover all the rows and that is minimal with respect to Cost.

We know that task (a) has an exponential complexity. Task (b) can also be exponential, since all minterms of f may have to be considered. Task (c) is NP-hard. Next section deal with task (a), and the following one describes how the covering matrix can be reduced to its *cyclic core* by removing rows and columns using the concepts of essentiality, minterm dominance, and prime dominance [6, 26, 34, 48, 57]. The last section shows how the Quine–McCluskey procedure, as typically ESPRESSO-EXACT [57] does, performs a branch and bound algorithm to solve the resulting covering matrix.

3.1 Prime implicant computation

One needs first to compute all the prime implicants of the Boolean function. Since the definition of prime implicants by Quine in 1952, a lot of work has been done to develop efficient prime implicants computation procedures. These techniques include the method based on consensus [2, 28, 51], that comes from a restriction of the resolution principle [21, 52]; the Karnaugh map method presented Section 2.3 that is well known of circuit designers [29]; the Quine–McCluskey method [4, 34, 48]; the semantic resolution method [62, 63]; Tison’s method [66] that has been proved to be correct later in [33]. The techniques that have been proposed to compute essential prime implicants are enhancements of prime implicant computation procedures. They are quite complex, and have been optimized for some classes of Boolean functions [54].

Most of these works represent incremental improvement over Quine’s algorithm. It consists of picking a minterm of f , considering it as a product, and of trying to remove as many as possible literals from it while preserving the implication of f by the products. These procedures iteratively compute one prime implicant at a time, which makes their cost directly related to the number of prime implicants to be found. From the practical point of view, these techniques are limited to Boolean functions that have no more than 20000 prime implicants².

²A recent technique that is not limited by the number of prime implicants is presented Section 7.1.

R	y_1	y_2	y_3	y_4	y_5
x_1	1	1	1	1	
x_2	1	1	1	1	
x_3	1			1	
x_4	1			1	1
x_5	1			1	
x_6					1

→

R	y_1	y_2	y_3	y_4
x_1	1	1	1	1
x_2	1	1	1	1
x_3	1			1
x_5	1			1

Figure 5: Reduction by essentiality.

3.2 Reduction processes

Once the covering matrix has been built, it can be split into several independent minimizing problems, and it can be reduced by iteratively removing its rows and columns. We recall here the main techniques that can be used to reduce the size of the optimization problem [24, 44, 57].

3.2.1 Partitioning

If the rows and the columns of a covering matrix can be permuted to yield a covering matrix that is partitioned in diagonal blocks B_k as follows, where 1's occur only in these blocks, then any minimal solution of the original problem is the union of minimal solutions of the blocks B_k . The partitioning of a set covering matrix is easily obtained by noting that two rows that cover a common column are in the same block.

B_1			
	B_2		
		\ddots	
			B_n

3.2.2 Essentiality

Let $\langle X, Y, R \rangle$ be a set covering problem. An element y of Y is *essential* iff it is the only one that covers an element x of X . When Y is a set of prime implicants, this notion is the same as essential prime implicants. Since essential elements belong necessarily to any minimal solution, we can remove from the covering matrix all columns labeled by essential elements as well as the rows they cover. This reduction yields a strongly equivalent set covering problem, since all minimal solutions of the original problem can be obtained by adding all essential elements to all minimal solutions of the reduced problem [47].

For instance consider the left-hand side covering matrix of Figure 5. The only element that covers x_6 is y_5 . Thus y_5 is necessary in all minimal solutions of this set covering problem. Thus the column labeled by y_5 can be removed from the matrix, as well as all rows covered by y_5 , i.e., x_4 and x_6 . This generates the strongly equivalent set covering problem whose matrix is shown on the right-hand side of Figure 5.

3.2.3 Dominance relations

Dominance relations have been introduced to reduce the size of a set covering problem by removing rows and columns from its matrix and yielding a new covering matrix that is (weakly) equivalent to the original one. We give here an intuitive idea of dominance relations, and the way they are used in the

R	y_1	y_2	y_3	y_4	y_5
x_1	1	1	1	1	
x_2	1	1	1	1	
x_3	1			1	
x_4	1			1	1
x_5	1			1	
x_6					1

→

R	y_1	y_2	y_3	y_4	y_5
x_3	1			1	
x_6					1

Figure 6: Reduction by domination on X .

R	y_1	y_2	y_3	y_4	y_5
x_1	1	1	1	1	
x_2	1	1	1	1	
x_3	1			1	
x_4	1			1	1
x_5	1			1	
x_6					1

→

R	y_1	y_5
x_1	1	
x_2	1	
x_3	1	
x_4	1	1
x_5	1	
x_6		1

Figure 7: Reduction by domination on Y .

Quine–McCluskey procedure to simplify the set covering problem [34]. Section 4 will go back to these concepts in order to formalize and enlighten some basic properties of dominance relations.

One says that an element x' of X *dominates* x iff all elements of Y that cover x' also cover x . This means that once x' is covered, one does not care about covering x . Thus the row labeled with x can be removed from the matrix without modifying the set of minimal solutions. The reduced covering matrix is strongly equivalent to the original one.

In the example shown in Figure 6 on the left, x_3 dominates x_1 , x_2 , x_4 and x_5 , and x_6 dominates x_4 . This covering matrix can be then reduced to the strongly equivalent one on the right. After this reduction, it is evident that y_2 and y_3 are useless to solve the set covering problem.

One says that an element y' of Y *dominates* y iff all elements of X covered by y are also covered by y' , and if $Cost(y') \leq Cost(y)$. This means that y' covers at least all the elements of X covered by y without being more costly. In the case where $Cost(y') = Cost(y)$, though y' seems to be a “better” choice than y when solving the set covering problem, it is not true that removing y from the covering matrix does not modify the set of minimal solutions, even if y' covers at least one element of X not covered by y , i.e., if y' strictly dominates y . Some minimal solutions using y can be lost, but there still exists a minimal solution that uses y' . For instance, if y and y' have an equal cost and cover exactly the same elements of X , then y and y' can be chosen equivalently to build a minimal solution, provided that y and y' are not strictly dominated by another element of Y .

In the example shown in Figure 7 on the left, y_1 dominates y_2 , y_3 , and y_4 . This covering matrix can be then reduced to the weakly equivalent one on the right. Note that the latter yields only the minimal solution $\{y_1, y_5\}$ and loses the minimal solution $\{y_4, y_5\}$ of the original problem.

3.2.4 Gimpel’s reduction

Gimpel proposed a reduction that applies when some rows x is only covered by two columns y_1 and y_2 that moreover have the same cost [24]. Let $\{x, x_1^1, \dots, x_1^n\}$ and $\{x, x_2^1, \dots, x_2^m\}$ be the set of elements covered by y_1 and y_2 respectively. Gimpel’s reduction consists in removing column y_1 , removing the

R	y_1	y_2	y_3	y_4	y_5
x	1	1			
x_1^1	1				1
x_1^2	1		1		
x_2^1		1	1		1
x_2^2		1		1	
x_6				1	1

→

R	y_2	y_3	y_4	y_5
$x^{1,1}$	1	1		1
$x^{1,2}$	1		1	1
$x^{2,1}$	1	1		1
$x^{2,2}$	1	1	1	
x_6			1	1

Figure 8: Gimpel’s reduction on x .

$n + m + 1$ rows covered by y_1 and y_2 , and adding nm new rows $x^{i,j}$ ($1 \leq i \leq n$ and $1 \leq j \leq m$) such that the row $x^{i,j}$ is covered by the set of columns

$$\{y \in Y \mid x_1^i R y \vee x_2^j R y\} - \{y_1\}.$$

Let S be a minimal solution of this new problem. Then a minimal solution of the original set covering problem is derived as follows [53]. If S is such that $S \cap \{y \in Y \mid x_1^i R y\} \neq \emptyset$ for $1 \leq i \leq n$, then add y_2 to S . Otherwise add y_1 to S .

Consider the example shown in Figure reffig:gimpel. The row x is only covered by two columns that have the same cost, namely y_1 and y_2 . All rows covered by y_1 and y_2 are removed, namely rows x up to x_2^2 , and columns y_1 is removed from the matrix. Here we have $n = m = 2$, so we add $nm = 4$ new rows as described above. For instance rows $x^{2,1}$ is covered by columns that cover either x_1^2 or x_2^1 beside y_1 , i.e., y_2, y_3, y_5 . Gimpel’s reduction produces the covering matrix shown on the right-hand side of Figure 8. One of its minimal solution is $S = \{y_3, y_5\}$, which intersects both the sets $\{y \in Y \mid x_1^i R y\}$ for $1 \leq i \leq 2$, i.e., $\{y_1, y_5\}$ and $\{y_1, y_3\}$, thus adding y_2 to S produces $\{y_2, y_3, y_5\}$ which is a minimal solution of the original problem. Another minimal solution of the right-hand side problem is $S = \{y_2, y_5\}$. The set S does not intersects the set $\{y_1, y_3\}$, and adding y_1 to S produces $\{y_1, y_2, y_5\}$ which is still a minimal solution of the original problem.

Gimpel’s reduction yields a new set covering problem with one less column but with $nm - n - m - 1$ more rows. Adding new rows can produce new dominations between rows and columns, which can reduce the covering matrix. However, from the practical point of view, it is better to allow Gimpel’s reduction when it guarantees to produce a smaller covering matrix, i.e., when $nm - n - m - 1 \leq 0$.

3.3 Branch and bound

When the reduction processes described above (essentiality, and dominance relations on X and Y) are iteratively applied, one eventually produced a covering matrix that the reduction processes do not modify anymore. This fixpoint is called the *cyclic core* of $\langle X, Y, R \rangle$. If this cyclic core is empty, the set of all essential elements that have been found during the reduction process constitutes a minimal solution of the original problem.

If the cyclic core is not empty, the minimization can be terminated using a branch-and-bound algorithm. One chooses an element of y , and generates two subproblems, one resulting from choosing y in the minimal solution, the other resulting from discarding y from Y . These two subproblems are consequently simplified, i.e., their cyclic cores are computed, and then recursively solved. The minimal solution of the original problem is the minimum of both of the minimal solutions of the two subproblems.

3.3.1 Branching

Heuristics to properly choose an element of Y as belonging to the minimal solution are discussed in [6, 34, 57]. A natural heuristics consists in choosing an element y that cover the largest number of elements of X . However experimental results show that it is not the best heuristics. A much better heuristics proposed in [57] consists in, given a cyclic core $\langle X, Y, R \rangle$, choosing an element y that maximizes $Choose(y)$:

$$Choose(y) = \sum_{xRy} \frac{1}{|\{y \in Y \mid x R y\}| - 1}$$

This cost function increases with the number of elements that y cover, but also with the “quality” of the elements it covers. The less an element x is covered, the harder it is to cover it, the larger will be its contribution to the cost function. Thus this cost function favors elements y that cover difficult x ’s.

3.3.2 Lower bound

If we know a lower bound of the minimal solution of a subproblem yielded at some point of the branching algorithm, one can prune the recursion as soon as the lower bound is greater or equal to the best solution found so far. It is critical to provide an accurate lower bound to terminate useless searches as early as possible.

Let $\langle X, Y, R \rangle$ be a set covering problem, and let X' be a subset of X such that for any two different elements x_1 and x_2 of X' , all elements y that cover x_1 do not cover x_2 and conversely. The set X' , which we will call an *independent* subset of X , provides us with the lower bound

$$\sum_{x \in X'} \min_{y \text{ } xRy} Cost(y),$$

since this is the minimum cost necessary to cover the elements of X' . Though finding an independent subset that maximizes this lower bound is an NP-complete problem, heuristics in practice yields a quite good lower bound, except for examples that have a large number of y ’s compared to the number of x ’s (for instance the big random examples of the MCNC benchmark [68]).

3.4 Conclusion

The two-level logic minimization of a Boolean function f consists in solving the set covering problem $\langle f^1, Prime(f^{1*}), \in \rangle$. The Quine–McCluskey procedure, e.g., ESPRESSO-EXACT, computes the cyclic core of this covering matrix using row/column removal based on dominance and essentiality. If the cyclic core is empty, the minimization is done, else a row is selected and the procedure is iterated. A branch-and-bound algorithm can then provide a minimal solution [6, 34, 57].

In the Quine–McCluskey procedure, the elements of f to be covered (i.e., the set X) and the prime implicants of f (i.e., the set Y) are *explicitly* represented. This implies that the cost of this procedure is directly related to the number of prime implicants, and it cannot be applied on functions that have large numbers of prime implicants.

To overcome this limitation, Swamy, McGeer, and Brayton have introduced in [65] an implicit cyclic core computation procedure, in which the covering matrix is *implicitly* represented with a couple of BDDs (Binary Decision Diagrams are a canonical representation of Boolean functions, see appendix A). The set of rows, i.e., the minterms to be covered, is represented with a BDD, and the set of columns is represented with the BDD of the characteristic function of the prime implicants that can be used

to cover the minterms. Initially, the covering matrix is represented by the BDD of the function f to minimize, and by the BDD of all the prime implicants of f . The idea is to compute, at each step of the cyclic core computation, two BDDs representing the minterm dominance relation and the prime dominance relation respectively, and then to extract from these two BDDs a new implicit covering matrix. The problem with this procedure is that the BDDs of the dominance relations are computed by evaluating complex quantified Boolean formulas, which makes the procedure difficult to apply on large examples [65]. It has been shown later that the use of recursive operators for the building of the minterm and prime dominance relations make the implicit cyclic computation much more efficient. However, it happens that the computation fails for some examples because the BDDs of the dominance relations are too big to allow extraction of the reduced covering matrix.

4 Formalizing covering matrix reduction

This section formalizes the reduction processes presented above, and shows that some new concepts can be figured out to reduce and solve a set covering problem [19] $\langle X, Y, R \rangle$ with respect to *any* additive cost function $Cost$.

4.1 Strong reduction with dominance relations

We first clarify the underlying structure of the dominance relations, and show how to reduce a covering matrix to a *strongly* equivalent problem.

A *quasi order* is a reflexive and transitive relation, and will be noted \prec . A *partial order* is a reflexive, transitive, and antisymmetric relation, and will be noted \preceq or \sqsubseteq . An *equivalence relation* is a reflexive, transitive, and symmetric relation, and will be noted \equiv . Let \equiv be an equivalence relation on Z . The *quotient set*, noted Z/\equiv , is the partition of Z made of the equivalence classes of \equiv . Given a set Z quasi ordered by \prec , we denote by $\max_{\prec} Z$ its set of maximal elements, i.e., the set $\{z \in Z \mid \forall z' \in Z, z \prec z' \Rightarrow z = z'\}$.

Let $\langle X, Y, R \rangle$ be a set covering problem, and $Cost$ any additive cost function. The dominance relations that we will note \prec_Y and \prec_X in the following, are defined on X and Y respectively as follows:

$$\begin{aligned} x \prec_Y x' &\Leftrightarrow \{y \in Y \mid x' R y\} \subseteq \{y \in Y \mid x R y\}, \\ y \prec_X y' &\Leftrightarrow \{x \in X \mid x R y\} \subseteq \{x \in X \mid x R y'\} \wedge Cost(y') \leq Cost(y). \end{aligned}$$

Since \subseteq and \leq are reflexive and transitive, the dominance relations \prec_Y and \prec_X are quasi orders on X and Y respectively. We now introduce the following notation.

Definition 4 *Let \prec be a quasi order on a set Z . We associate with \prec the equivalence relation \equiv defined by $(z \equiv z') \Leftrightarrow ((z \prec z') \wedge (z' \prec z))$. The projection of the quasi order \prec on Z/\equiv is a partial order, which we denote with \preceq . We denote by π the function that maps each element of Z onto its equivalence class, and by ρ a function that map each equivalence class onto one of its element.*

The process of row and column removing described in Section 3.2 generates a weakly equivalent set covering problem, because an arbitrary choice is made between two columns that dominate each other. A stronger reduction process consists in considering two elements that dominate each other as belonging to the same class. Indeed, this consists in partitioning X (respectively Y) into the equivalence classes X/\equiv_Y (respectively Y/\equiv_X) induced by the dominance relation \prec_Y (respectively \prec_X). Then we only

R	y_1	y_2	y_3	y_4	y_5
x_1	1	1	1	1	
x_2	1	1	1	1	
x_3	1			1	
x_4	1			1	1
x_5	1			1	
x_6					1

R	$\{y_1, y_4\}$	$\{y_2, y_3\}$	$\{y_5\}$
$\{x_1, x_2\}$	1	1	
$\{x_3, x_5\}$	1		
$\{x_4\}$	1		1
$\{x_6\}$			1

R	$\{y_1, y_4\}$	$\{y_5\}$
$\{x_3, x_5\}$	1	
$\{x_6\}$		1

Figure 9: Strong dominance reduction with equivalence classes.

keep the equivalence classes that are maximal with respect to \preceq_Y and \preceq_X , which produces a reduced and *strongly* equivalent set covering problem $\langle \max_{\preceq_Y}(X/\equiv_Y), \max_{\preceq_X}(Y/\equiv_X), R \rangle$. No minimal solution is lost, which is the case when one chooses between elements of Y that dominate each other.

Figure 9 shows the reduction obtained by rewriting $\langle X, Y, R \rangle$ into $\langle X/\equiv_Y, Y/\equiv_X, R \rangle$, and then into $\langle \max_{\preceq_Y}(X/\equiv_Y), \max_{\preceq_X}(Y/\equiv_X), R \rangle$. Each class of X/\equiv_Y and of Y/\equiv_X are made of elements that dominate each other. Note how the only minimal solution of the rewritten problem, namely $\{\{y_1, y_4\}, \{y_5\}\}$, yields *all* the minimal solutions of the original problem: $\{y_1, y_4\} \cdot \{y_5\} = \{\{y_1, y_5\}, \{y_4, y_5\}\}$. This reduction process is formalized by Theorem 1.

Theorem 1 *Let $\langle X, Y, R \rangle$ be a set covering problem. The following transformation produces a strongly equivalent set covering problem (the relation R is modified in the usual way on the quotient sets, and the cost of an equivalence class is the cost of any of its element).*

$$\langle X, Y, R \rangle \rightarrow \langle \max_{\preceq_Y}(X/\equiv_Y), \max_{\preceq_X}(Y/\equiv_X), R \rangle$$

All the minimal solutions of the original problem are built as follows, where C is the problem generated by the transformation above:

$$\text{Sol}(\langle X, Y, R \rangle) = \bigcup_{S \in \text{Sol}(C)} \prod_{s \in S} s$$

Proof. As explained in section 3.2.3, $\langle X, Y, R \rangle$, $\langle X/\equiv_Y, Y, R \rangle$, and $\langle \max_{\preceq_Y}(X/\equiv_Y), Y, R \rangle$ are all strongly equivalent, since the set Y is not modified. Problems $\langle X, Y/\equiv_X, R \rangle$ and $\langle X, \max_{\preceq_X}(Y/\equiv_X), R \rangle$ are clearly strongly equivalent, since the dominance relation on Y/\equiv_X is a partial order. We just have to check that $\langle X, Y, R \rangle$ and $\langle X, Y/\equiv_X, R \rangle$ are strongly equivalent.

Let $\{y_1, \dots, y_n\}$ be a minimal solution of $\langle X, Y, R \rangle$. By definition of \equiv_X , $\{\pi(y_1), \dots, \pi(y_n)\}$ is a minimal solution of $\langle X, Y/\equiv_X, R \rangle$. Conversely let $\{s_1, \dots, s_n\}$ be a minimal solution of $\langle X, Y/\equiv_X, R \rangle$. Then the n -tuple of $s_1 \times \dots \times s_n$ are minimal solutions of $\langle X, Y, R \rangle$. So $\langle X, Y, R \rangle$ and $\langle X, Y/\equiv_X, R \rangle$ are strongly equivalent. \square

4.2 Transposing functions

We have shown that reducing $\langle X, Y, R \rangle$ to $\langle \max_{\preceq_Y}(X/\equiv_Y), \max_{\preceq_X}(Y/\equiv_X), R \rangle$ preserves all minimal solutions and reduces the size of the covering problem. However it can be difficult to manipulate equivalence classes since they are subsets of X and Y . One can view the Quine–McCluskey-like minimization procedures as representing these equivalence classes through projections ρ_Y and ρ_X that map each class onto one of its elements. These projections guarantee that the reduced covering problem is equivalent

to the original one, but it is only weakly equivalent if one cannot evaluate ρ_X^{-1} , i.e., π_X . This section shows that, instead of using such projections, one can use an isomorphism that maps the classes that must be manipulated onto objects whose manipulation is less costly [19].

Definition 5 Let (Z, \prec) be a quasi ordered set. A transposing function τ is a morphism³ that maps (Z, \prec) onto a partially ordered set $(\tau(Z), \sqsubseteq)$, such that $\tau(z) \sqsubseteq \tau(z')$ iff $z \prec z'$.

Any quasi ordered set (Z, \prec) has at least one transposing function. For instance, the function $\tau(z) = \{z' \in Z \mid z' \prec z\}$ is a transposing function that maps (Z, \prec) into the partially ordered set $(2^Z, \sqsubseteq)$. We call this particular morphism the *universal* transposing function of (Z, \prec) .

If τ is a transposing function from the quasi ordered set (Z, \prec) into $(\tau(Z), \sqsubseteq)$, then the latter is isomorphic to $(Z/\equiv, \preceq)$ through $\tau \circ \rho$. We thus obtain the following commutative diagram:

$$\begin{array}{ccc}
 (Z/\equiv, \preceq) & \longrightarrow & \max_{\preceq}(Z/\equiv) \\
 \nearrow \tau & \uparrow \downarrow \tau \circ \rho & \uparrow \downarrow \tau \circ \rho \\
 (Z, \prec) & & (\tau(Z), \sqsubseteq) \longrightarrow \max_{\sqsubseteq} \tau(Z)
 \end{array}$$

Note that the range of τ can be any set, as soon as it is partially ordered. Thus the transformation that τ performs is useful because it can be much more efficient to represent and manipulate elements of $\tau(Z)$ than elements of 2^Z .

Using transposing functions to reduce set covering problems is then straightforward. Figure 10 illustrates how transposing functions reduce a covering matrix $\langle X, Y, R \rangle$. Let τ_Y and τ_X be the universal transposing functions of the dominance relations \prec_Y and \prec_X respectively. Then $\max_{\preceq_Y}(X/\equiv_Y)$ (respectively $\max_{\preceq_X}(Y/\equiv_X)$) is isomorphic to $\max_{\sqsubseteq} \tau_Y(X)$ (respectively $\max_{\sqsubseteq} \tau_X(Y)$). Thus the reduced and strongly equivalent set covering problem $\langle \max_{\preceq_Y}(X/\equiv_Y), \max_{\preceq_X}(Y/\equiv_X), R \rangle$ introduced in previous section is isomorphic to $\langle \max_{\sqsubseteq_Y} \tau_Y(X), \max_{\sqsubseteq_X} \tau_X(Y), R' \rangle$, where R' is the relation induced by R through τ_Y and τ_X . From the covering matrix given on the left, we compute the sets $\tau_Y(x_k)$ and $\tau_X(y_k)$, and produce the reduced and strongly equivalent covering matrix on the right by keeping sets that are maximal w.r.t. \sqsubseteq . The only minimal solution of this covering matrix is $\{\tau_X(y_1), \tau_X(y_5)\}$. So the minimal solutions of the original problem is given by $\tau_X^{-1}(\tau_X(y_1)) \cdot \tau_X^{-1}(\tau_X(y_5)) = \{y_1, y_4\} \cdot \{y_5\} = \{\{y_1, y_5\}, \{y_4, y_5\}\}$. Theorem 2 formalizes this new transformation.

Theorem 2 Let $\langle X, Y, R \rangle$ be a set covering problem, τ_Y be a transposing function from (X, \prec_Y) into $(\tau_Y(X), \sqsubseteq_Y)$, and τ_X be a transposing function from (Y, \prec_X) into $(\tau_X(Y), \sqsubseteq_X)$. The following transformation, where R' is defined by $\tau_Y(x) R' \tau_X(y) \Leftrightarrow x R y$, produces a strongly equivalent set covering problem.

$$\langle X, Y, R \rangle \rightarrow \langle \max_{\sqsubseteq_Y} \tau_Y(X), \max_{\sqsubseteq_X} \tau_X(Y), R' \rangle.$$

All the minimal solutions of the original problem are built as follows, where C is the problem generated by the transformation above:

$$\text{Sol}(\langle X, Y, R \rangle) = \bigcup_{S \in \text{Sol}(C)} \prod_{s \in S} \tau_X^{-1}(s).$$

³Most of the time it is not injective.

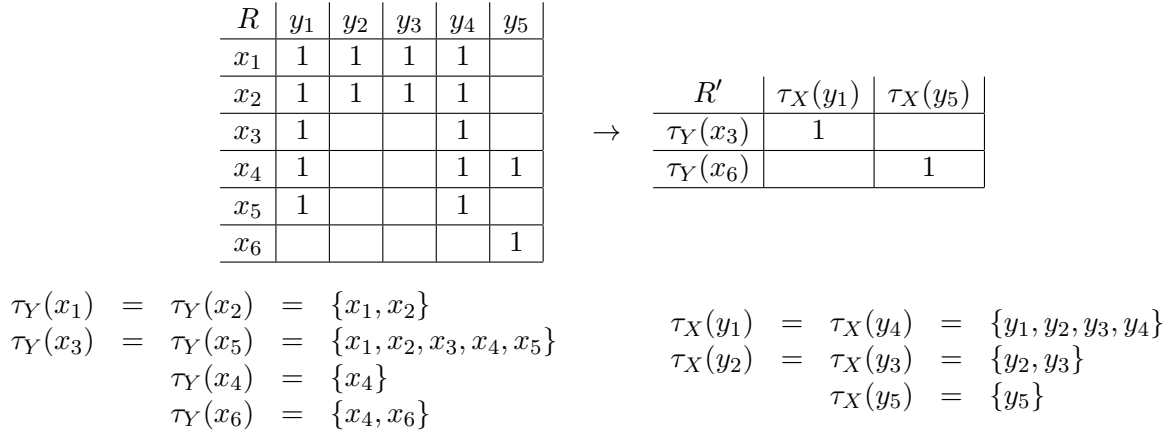


Figure 10: Universal transposing functions based reduction.

Proof. We need to show that R' is well defined, i.e., $x R y$ is equivalent to $x' R y'$ for all x' and y' such that $\tau_Y(x') = \tau_Y(x)$ and $\tau_X(y') = \tau_X(y)$. This is the case because $\tau_Y(x') = \tau_Y(x)$ and $\tau_X(y') = \tau_X(y)$ implies that x and x' are covered by exactly the same elements of Y , and that y and y' cover exactly the same elements of X .

The two set covering problems are strongly equivalent because transposing functions establish the isomorphisms $\tau \circ \rho$ presented in this Section. \square

4.3 Conclusion

This section has shown how the dominance concept can be used to strongly reduce covering matrices when dominance relations are considered as quasi-orders with their associated equivalence relations. Furthermore, morphisms can be used to transpose a set covering problem onto a strongly equivalent one that operates on an arbitrary domain, which allows the use of more suitable workspace and data structure to solve the minimization problem. The next section illustrates these ideas with a reduction procedure whose behavior is very different from that of Quine–McCluskey. Section 7 will present a minimization algorithm that more fully uses the concepts that have been presented.

5 The signature cube based procedure

The first step of Quine–McCluskey-like minimization procedures, as ESPRESSO-EXACT, is a bottleneck because it requires the computations of all the prime implicants of the function to minimize. Brayton, McGeer, and Sanghavi proposed a procedure that does not necessarily need to compute all prime implicants to generate a covering matrix. For this purpose, they introduce the following function σ [7]:

$$\sigma(x) = \bigcap_{\{p \in P \mid x \in p\}} p,$$

where x is a minterm, and P is the set of prime implicants of the function f to minimize. They call the product $\sigma(x)$ the *signature cube* of the minterm x . The schema of their minimization procedure, called ESPRESSO-SIGNATURE, can be viewed as follows [7, 36]:

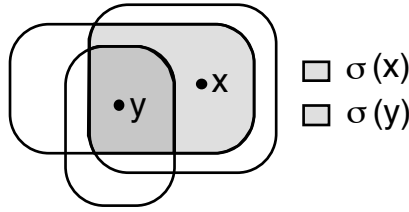


Figure 11: Signature cubes and minterm dominance.

- (a) Build a covering matrix whose rows are labeled with the elements of $\max_{\subseteq} \sigma(f)$, and whose columns are labeled with the prime implicants of f that contain some rows.
- (b) Solve this covering matrix.

The key point of this minimization procedure is that $\max_{\subseteq} \sigma(f)$ can be computed without necessarily computing all prime implicants of f .

We can prove very easily the correctness of this approach by showing that σ is a transposing function from (f, \prec_P) onto $(\sigma(f), \subseteq)$. By definition, x' belongs to $\sigma(x)$ if and only if x' is covered by all elements of P that cover x , i.e., if x' is dominated by x . In other words we have $\sigma(x) = \{x' \in f \mid x' \prec_P x\}$, so σ is the universal transposing function of the minterm dominance relation \prec_P . Thus σ transposes problem $\langle f, P, \in \rangle$ into $\langle \max_{\subseteq} \sigma(f), P, R \rangle$ where R is by Theorem 2: $\sigma(x) R p \Leftrightarrow x \in p$. Since $x \in p$ is clearly equivalent to $\sigma(x) \subseteq p$, we have $R = \subseteq$. The signature cube based minimization method can be then resumed by the strong reduction:

$$\langle f, P, \in \rangle \rightarrow \langle \max_{\subseteq} \sigma(f), P, \subseteq \rangle$$

which also proves that the only prime implicants that are needed are those that contain some maximal signature cubes.

Figure 11 shows an example of two signature cubes. All elements belonging to the signature cube $\sigma(x)$ are exactly those dominated by x . So the set $\max_{\subseteq} \sigma(f)$ of maximal signature cubes is isomorphic to $\max_{\prec_P} (f / \equiv_P)$.

The first task consists in computing $\max_{\subseteq} \sigma(f)$, as well as the prime implicants of f that contain at least one maximal signature cubes. The naive algorithm of Figure 12 shows how it can be done iteratively without necessarily computing all the prime implicants of f . Indeed it is explained in [7, 36] how the maximal signature cubes can be efficiently computed from any sum-of-products representation of f without generating a single prime implicant (unless it happens that some signature cube is also a prime implicant).

This naive algorithm works as follows. Initially, the set Σ of signature cubes found to be maximal so far is empty, and the set F of elements that are candidates to generate new signature cubes is the set of minterms of f . An element x is picked in F , and $\sigma(x)$ is computed. All elements in $\sigma(x)$ are dominated by x , so they cannot have larger signature cubes than $\sigma(x)$, and can be removed from F . All previously computed signature cubes that are contained in $\sigma(x)$ can be removed from Σ , because they cannot be maximal. Note that at this step, it is impossible to have an element of Σ that contains $\sigma(x)$. If not, another signature cube $\sigma(y)$ containing $\sigma(x)$ must have been generated, but in this case $\sigma(y)$, that contains x , has been removed from F , and so x cannot have been picked. Thus the potential maximal element $\sigma(x)$ is added to Σ . This process is iterated until F becomes empty. The prime implicants that contain at least one maximal signature cube are necessarily some of those that have been used to

```

function ComputeMaxSigma( $f$ );
 $\Sigma = \emptyset$ ;
 $F = f^1$ ;
while  $F \neq \emptyset$  do {
     $x$     = an element of  $F$ ;
     $P(x)$  = all prime of  $f^{1*}$  that covers  $x$ ;
     $\sigma(x)$  =  $\bigcap_{p \in P(x)} p$ ;
     $\Sigma$    =  $\{q \in \Sigma \mid q \not\subseteq \sigma(x)\} \cup \{\sigma(x)\}$ ;
     $F$      =  $F - \sigma(x)$ ;
}
return  $\Sigma$ ;

```

Figure 12: Naive computation of $\max_{\subseteq} \sigma(f)$.

compute the maximal signature cubes, and so they can be kept in memory during the computation of $\max_{\subseteq} \sigma(f)$. The reader is referred to [36] for a much more effective procedure.

Note that the covering matrix ESPRESSO-SIGNATURE generates cannot be larger than the one produced by the Quine–McCluskey algorithm (ESPRESSO-EXACT), since the latter has its columns labeled by all the prime implicants of the Boolean function to minimize. The covering matrix is then solved by the techniques described in Section 3.2, as ESPRESSO-EXACT does.

Though ESPRESSO-SIGNATURE yields a covering matrix that may be much larger than the cyclic core of f , it is nevertheless interesting because it does not require the computation of all the prime implicants of f , and it has been successfully used to generate covering matrices for functions with huge numbers of prime implicants in such a way that the exact minimization can be achieved using this reduced matrix [7, 36]. However this reduction is still of limited power because for some functions the reduction it allows is not sufficient, or the number of maximal signature cubes is too large to allow their exhaustive computation (See Section 8). For instance the function $x_1 \oplus \dots \oplus x_n$ has 2^{n-1} maximal signature cubes. Actually it is important to note that the number of maximal signature cubes can be larger than the number of prime implicants, since signature cubes are related to both prime implicants *and* minterms of the function to minimize.

6 Set covering problem over lattices

This section deals with set covering problems over a lattice (we recall what is a lattice below). Set covering problems $\langle X, Y, \subseteq \rangle$, where both X and Y are subsets of a lattice (Z, \subseteq) , are the most general in the sense that any set covering problem $\langle U, V, R \rangle$ can be rewritten into a strongly equivalent set covering problem over a lattice. Consider any set covering problem $\langle U, V, R \rangle$. Now let $X = \{\{u\} \mid u \in U\}$ and $Y = \{\{u \in U \mid u R v\} \mid v \in V\}$. Both X and Y are subsets of the lattice $(2^U, \subseteq)$. Clearly $x \subseteq y \Leftrightarrow u R v$, which proves that $\langle X, Y, \subseteq \rangle$ is isomorphic to the original problem $\langle U, V, R \rangle$.

A particular interesting application of this idea consists in rewriting the two-level logic minimization problem into a set covering problem over the lattice $(\mathcal{P}(\mathcal{B}), \subseteq)$. For this purpose, just note that $\langle f, P, \subseteq \rangle$ is isomorphic to $\langle Q, P, \subseteq \rangle$, where $Q = \{q \in \mathcal{P}(\mathcal{B}) \mid x \in f, \{x\} = q\}$. Section 7 will be a consequence of this rewriting.

We assume here that the cost of a cover is the number of its elements (one can generalize to any additive cost function but this requires technical details that are irrelevant for this presentation). This

section introduces an essential result for solving set covering problems over a lattice.

6.1 Lattices

A complete lattice (Z, \sqsubseteq) is a partially ordered set such that any non empty subset of Z has a greatest lower bound and a least upper bound with respect to \sqsubseteq . If X is a subset of Z , we denote $\sup_{\sqsubseteq} X$ the least upper bound of X w.r.t \sqsubseteq , i.e., the unique minimal (w.r.t \sqsubseteq) element z of Z such that $x \sqsubseteq z$ for all elements x of X . We denote $\inf_{\sqsubseteq} X$ the greatest lower bound of X , i.e., the unique maximal element z of Z such that $z \sqsubseteq x$ for all elements x of X .

Since the structure (Z, \supseteq) is also a lattice, the principle of duality implies that, if a statement is true on (Z, \sqsubseteq) , then the dual statement obtained by interchanging \sqsubseteq and \supseteq can also be deduced. Note that $\inf_{\sqsubseteq} = \sup_{\supseteq}$.

6.2 Solving set covering problems over a lattice

The aim of this section is to prove the following fundamental result.

Theorem 3 *Let (Z, \sqsubseteq) be a complete lattice. Let X and Y be subsets of Z . The fixpoint obtained by applying the following transformations is strongly equivalent to the cyclic core of the set covering problem $\langle X, Y, \sqsubseteq \rangle$.*

$$\begin{aligned} \langle X, Y, \sqsubseteq \rangle &\xrightarrow{1} \langle X, \max_{\sqsubseteq} \tau_{X, \sqsubseteq}(Y), \sqsubseteq \rangle, \\ \langle X, Y, \sqsubseteq \rangle &\xrightarrow{2} \langle \max_{\sqsubseteq} \tau_{Y, \supseteq}(X), Y, \sqsubseteq \rangle, \\ \langle X, Y, \sqsubseteq \rangle &\xrightarrow{3} \langle X - E, Y - E, \sqsubseteq \rangle \quad \text{with } E = X \cap Y \end{aligned}$$

where $\tau_{W, \sqsubseteq}$ is the function defined from Z into Z by:

$$\tau_{W, \sqsubseteq}(z) = \sup_{\sqsubseteq} \{w \in W \mid w \sqsubseteq z\}.$$

All the minimal solutions of the original problem are built as follows. Let C be the fixpoint generated by the transformations above, and F be the union of the sets E yielded by the third rewriting rule during the fixpoint computation. Then

$$\text{Sol}(\langle X, Y, R \rangle) = \bigcup_{S \in \text{Sol}(C)} \prod_{z \in S \cup F} \{y \in Y \mid z \sqsubseteq y\}.$$

We first present examples to give the intuitive ideas beyond this rewriting system. We then prove Theorem 3 in three steps. We first justify rule (1) and (2) by showing that $\tau_{W, \sqsubseteq}$ is a transposing function that can be used to capture dominance on both X and Y . We show that rule (3) consists of computing the set E of essential elements of Y and applying the corresponding reduction. Finally we prove that recovering the minimal solutions of the original problem can be done as described above.

6.2.1 Examples and intuition

Consider the set covering problem over a lattice given on the left-hand side of Figure 13. Each dot is an element of the lattice (Z, \sqsubseteq) . Black dots are the elements of X , and gray boxes are the elements of

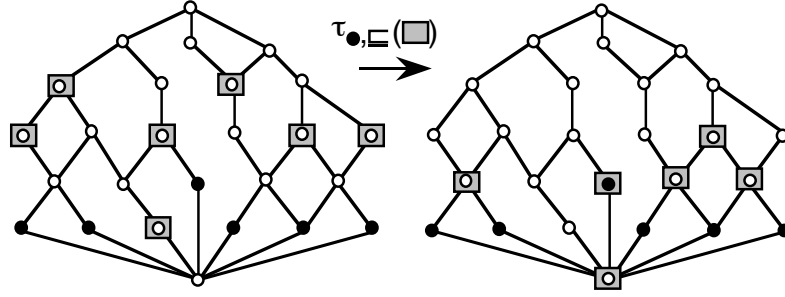


Figure 13: Computing $\tau_{X,\subseteq}(Y)$ on a lattice.

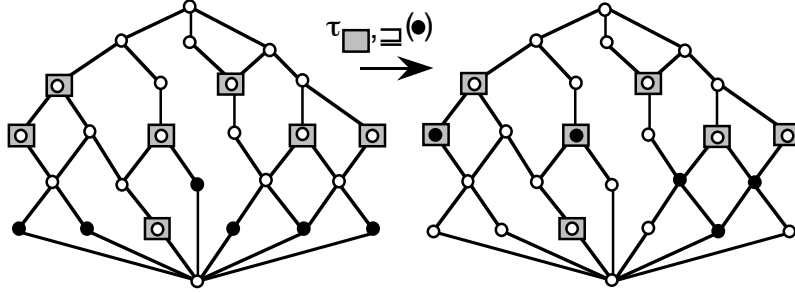


Figure 14: Computing $\tau_{Y,\supseteq}(X)$ on a lattice.

Y . The maximal element of Z is the top dot of the diagram, and its minimal element is the bottom dot. The partial order \subseteq is denoted by edges between an upper dot and a lower dot.

The rewriting of Y into $\tau_{X,\subseteq}(Y)$ is given by the right-hand side diagram of Figure 13. Each box y is mapped on the least upper bound of the set of black dots it covers. The two right-hand side boxes are mapped on the same element of Z because they cover exactly the same elements of X . Note that some boxes of Y remain invariant, while some others are mapped onto new elements of Z , some of them even being elements of X . The cardinality of $\tau_{X,\subseteq}(Y)$ is strictly less than the cardinality of Y because the two left-hand side boxes have been mapped on the same box by $\tau_{X,\subseteq}$, since they cover exactly the same black dots. Keeping the maximal boxes with respect to \subseteq produces a set made of three boxes, which is the only minimal covering of the reduced problem. From this set we can recover the two minimal solutions of the original problem as it is described by Theorem 3.

Figure 14 gives the rewriting of X into $\tau_{Y,\supseteq}(X)$ on the same covering problem. Each black dot x is mapped on the greatest lower bound of the set of boxes that cover it. The two right-hand side black dots are mapped on the same element of Z because they are covered by the same set of boxes. Note that $\tau_{Y,\supseteq}(x) = \inf_{\subseteq}\{y \in Y \mid x \subseteq y\}$. Here again, some elements of X remains invariant, while some others are mapped onto new elements of Z , some of them even being elements of Y . The cardinality of $\tau_{Y,\supseteq}(X)$ is strictly less than X 's one because the two left-hand side black dots have been mapped on the same dot by $\tau_{Y,\supseteq}$, since they are covered by the same boxes. There are only four black dots that are maximal with respect to \subseteq . We can then obtain directly the two minimal solution of the original problem.

Figure 15 gives on the left-hand side a set covering problem $\langle X, Y, \subseteq \rangle$ over a lattice (Z, \subseteq) where Y is maximal. Then it shows the set covering problem obtained after having applied $\tau_{Y,\supseteq}$, and finally the diagram obtained by extracting from $\tau_{Y,\supseteq}(X)$ the maximal elements. The elements of sets $Y \cap \tau_{Y,\supseteq}(X)$ and $Y \cap \max_{\subseteq} \tau_{Y,\supseteq}(X)$ are denoted by a box with a black dot inside. Both of these sets are equal to

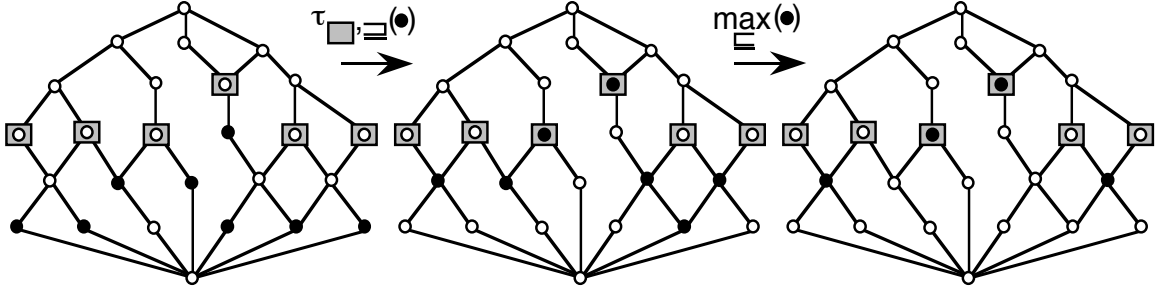


Figure 15: Computing the essential elements.

a set E . Clearly E is the set of essential elements of the original problem. Removing E from the set of boxes consists in removing essential elements from Y , and removing E from the set of black dots consists in removing the elements of X that are covered by essential elements. This problem has finally four minimal solutions.

6.2.2 The transposing function $\tau_{W, \sqsubseteq}$

We prove the first result that will justify rule (1) and (2).

Lemma 1 *Let (Z, \sqsubseteq) be a complete lattice. Let W be a subset of Z . Let $\prec_{W, \sqsubseteq}$ be the quasi-order defined on Z by*

$$z \prec_{W, \sqsubseteq} z' \Leftrightarrow \{w \in W \mid w \sqsubseteq z\} \subseteq \{w \in W \mid w \sqsubseteq z'\},$$

and $\tau_{W, \sqsubseteq}$ be the function defined from Z into Z by

$$\tau_{W, \sqsubseteq}(z) = \sup_{\sqsubseteq} \{w \in W \mid w \sqsubseteq z\}.$$

The function $\tau_{W, \sqsubseteq}$ is a transposing function from $(Z, \prec_{W, \sqsubseteq})$ into (Z, \sqsubseteq) .

Proof. The proof is based on the two key properties of $\tau_{W, \sqsubseteq}$ that come directly from its definition: (1) for all z of Z we have $\tau_{W, \sqsubseteq}(z) \sqsubseteq z$; (2) for all w of W and all z of Z , $w \sqsubseteq z$ is equivalent to $w \sqsubseteq \tau_{W, \sqsubseteq}(z)$.

Since for any subsets S and S' of Z , $S \subseteq S'$ implies that $\sup S \sqsubseteq \sup S'$, we have $z \prec_{W, \sqsubseteq} z'$ implies $\tau_{W, \sqsubseteq}(z) \sqsubseteq \tau_{W, \sqsubseteq}(z')$. Conversely, assume that $\tau_{W, \sqsubseteq}(z) \sqsubseteq \tau_{W, \sqsubseteq}(z')$. Then by (1) and (2) we have $w \sqsubseteq \tau_{W, \sqsubseteq}(z) \sqsubseteq \tau_{W, \sqsubseteq}(z') \sqsubseteq z'$ for all element w of W such that $w \sqsubseteq z$. This means that we have $w \sqsubseteq z'$ for all element w of W such that $w \sqsubseteq z$, i.e., $z \prec_{W, \sqsubseteq} z'$. So we have shown that $z \prec_{W, \sqsubseteq} z'$ is equivalent to $\tau_{W, \sqsubseteq}(z) \sqsubseteq \tau_{W, \sqsubseteq}(z')$. Thus $\tau_{W, \sqsubseteq}$ is a transposing function from $(Z, \prec_{W, \sqsubseteq})$ into (Z, \sqsubseteq) . \square

Corollary 1 *Let (Z, \sqsubseteq) be a complete lattice. Let $\langle X, Y, \sqsubseteq \rangle$ be a set covering problem, where X and Y are subsets of Z . It is strongly equivalent to $\langle \max_{\sqsubseteq} \tau_{Y, \sqsupseteq}(X), \max_{\sqsubseteq} \tau_{X, \sqsubseteq}(Y), \sqsubseteq \rangle$.*

Proof. Lemma 1 proved that $\tau_{X, \sqsubseteq}$ is a transposing function from $(Z, \prec_{X, \sqsubseteq})$ into (Z, \sqsubseteq) . By duality and Lemma 1, the function $\tau_{Y, \sqsupseteq}$ is a transposing function from $(Z, \prec_{Y, \sqsupseteq})$ into (Z, \sqsupseteq) , i.e., a transposing function from $(Z, \succ_{Y, \sqsupseteq})$ into (Z, \sqsubseteq) . The dominance relation on Y is $\prec_{X, \sqsubseteq}$, and the dominance relation on X is $\succ_{Y, \sqsupseteq}$. So by Theorem 2, the problem $\langle X, Y, \sqsubseteq \rangle$ is strongly equivalent to $\langle \max_{\sqsubseteq} \tau_{Y, \sqsupseteq}(X), \max_{\sqsubseteq} \tau_{X, \sqsubseteq}(Y), R \rangle$, where R is defined by $x \sqsubseteq y \Leftrightarrow \tau_{Y, \sqsupseteq}(x) R y$. By (2) and duality, $y \sqsupseteq x$ is equivalent to $y \sqsupseteq \tau_{Y, \sqsupseteq}(x)$ for all y of Y , so we have $R = \sqsubseteq$. \square

We have justified the first and second reduction, namely the set covering problem $\langle X, Y, \sqsubseteq \rangle$ is strongly equivalent to $\langle X, \max_{\sqsubseteq} \tau_{X, \sqsubseteq}(Y), \sqsubseteq \rangle$ and $\langle \max_{\sqsubseteq} \tau_{Y, \sqsupseteq}(X), Y, \sqsubseteq \rangle$. The advantage of the transposing functions $\tau_{X, \sqsubseteq}$ and $\tau_{Y, \sqsupseteq}$ is that they map Z into Z , which preserves the nature of the set covering problem: they produce strongly equivalent problems that have still the form $\langle X, Y, \sqsubseteq \rangle$, where both X and Y are subsets of the lattice (Z, \sqsubseteq) .

6.2.3 Essential elements

We now prove the second result. This nice property is a consequence of the definition of $\tau_{Y, \sqsupseteq}$, which is a transposing function that acts as a filter on the lattice and maps (Z, \sqsubseteq) into itself.

Lemma 2 *Let (Z, \sqsubseteq) be a complete lattice. Let $\langle X, Y, \sqsubseteq \rangle$ be a set covering problem, where X is a subset of Z , and Y is a maximal subset of Z w.r.t \sqsubseteq . Then the set of essential elements of Y is $Y \cap \tau_{Y, \sqsupseteq}(X)$, which is also equal to $Y \cap \max_{\sqsubseteq} \tau_{Y, \sqsupseteq}(X)$.*

Proof. Let y be an essential element of Y . Then it is the only element of Y such that $x \sqsubseteq y$ for an element x of X . Then we have $\tau_{Y, \sqsupseteq}(x) = \sup_{\sqsupseteq} \{y \in Y \mid y \sqsupseteq x\} = \sup_{\sqsupseteq} \{y\} = y$, and so y belongs to $\tau_{Y, \sqsupseteq}(X)$.

Conversely, let $y \in Y \cap \tau_{Y, \sqsupseteq}(X)$. Then there exists an element x of X such that $\tau_{Y, \sqsupseteq}(x) = y$, and we have $x \sqsubseteq y$. Let y' be an element of Y such that $x \sqsubseteq y'$. We then have $y = \tau_{Y, \sqsupseteq}(x) \sqsubseteq y'$, i.e., $y \sqsubseteq y'$. Since Y is maximal w.r.t \sqsubseteq , this implies that $y = y'$. So y is the only element of Y such that $x \sqsubseteq y$, i.e., y is essential.

We have proved that the set of essential elements of Y is $Y \cap \tau_{Y, \sqsupseteq}(X)$. This set is also equal to $Y \cap \max_{\sqsubseteq} \tau_{Y, \sqsupseteq}(X)$. Clearly the latter is included in the former. Conversely, assume that there exists y such that $y \in Y \cap \tau_{Y, \sqsupseteq}(X)$ and $y \notin Y \cap \max_{\sqsubseteq} \tau_{Y, \sqsupseteq}(X)$. This means that y is not a maximal element of $\tau_{Y, \sqsupseteq}(X)$ with respect to \sqsubseteq , so there exists an element x of X such that $y \sqsubseteq \tau_{Y, \sqsupseteq}(x)$ and $y \neq \tau_{Y, \sqsupseteq}(x)$. Then let y' be an element of Y such that $x \sqsubseteq y'$. We have $y \sqsubseteq \tau_{Y, \sqsupseteq}(x) \sqsubseteq y'$, i.e., $y \sqsubseteq y'$ with $y \neq y'$, which is impossible because Y is maximal with respect to \sqsubseteq . \square

Provided that Y is *maximal* with respect to \sqsubseteq , the set of essential elements is thus $Y \cap \max_{\sqsubseteq} \tau_{Y, \sqsupseteq}(X)$. The first rewriting rule produces a set Y that has the form $\max_{\sqsubseteq} \tau_{X, \sqsubseteq}(Y)$, which is obviously maximal. The second rewriting rule produces a set X that has the form $\max_{\sqsubseteq} \tau_{Y, \sqsupseteq}(X)$. This establishes the correctness of the third rewriting rule.

6.2.4 Recovering the original solutions

Recovering the original solutions from the minimal solutions of the fixpoint can be done very easily thanks to the uniformization of the workspace to the lattice (Z, \sqsubseteq) . To establish the last part of Theorem 3, by Corollary 1 and Theorem 2, we just have to prove the following lemma.

Lemma 3 *The restriction of the function $\tau_{X, \sqsubseteq}^{-1}$ to Y satisfies the following identity:*

$$\tau_{X, \sqsubseteq}^{-1}(y) = \{y \in Y \mid z \sqsubseteq y\}.$$

Proof. Let y be an element of $\tau_{X, \sqsubseteq}^{-1}(z)$. This means that $\tau_{X, \sqsubseteq}(y) = z$, and since we have $\tau_{X, \sqsubseteq}(y) \sqsubseteq y$, we have $z \sqsubseteq y$. Conversely, let y be an element of Y such that $z \sqsubseteq y$. Then $\tau_{X, \sqsubseteq}(z) \sqsubseteq \tau_{X, \sqsubseteq}(y)$. But since $\tau_{X, \sqsubseteq}(z)$ belongs to $\max_{\sqsubseteq} \tau_{X, \sqsubseteq}(Y)$, it is maximal w.r.t \sqsubseteq within $\tau_{X, \sqsubseteq}(Y)$, and so we must have $\tau_{X, \sqsubseteq}(z) = \tau_{X, \sqsubseteq}(y)$. Moreover since z belongs to $\tau_{X, \sqsubseteq}(Y)$, and since $\tau_{X, \sqsubseteq}$ is idempotent, we have $\tau_{X, \sqsubseteq}(z) = z$. Thus $z = \tau_{X, \sqsubseteq}(y)$, and so $y \in \tau_{X, \sqsubseteq}^{-1}(z)$. \square

7 A new two-level logic minimization procedure

This section presents SCHERZO, a two-level logic minimization procedure that fully uses the ideas presented in the previous section. SCHERZO reduces the minimization of a vectorial Boolean function f to the one of a single-output Boolean function using the one-to-one mapping between the implicants of f and those of the function ff defined in Section 2.1. The outline of the procedure that SCHERZO uses to compute a prime cover with a minimal number of products is as follows [19].

- (a) Compute the set P of all prime implicants of f .
- (b) Transpose the set covering problem $\langle f, P, \in \rangle$ into the strongly equivalent problem $\langle Q, P, \subseteq \rangle$ over the lattice $(\mathcal{P}(\mathcal{B}), \subseteq)$, where $Q = \{q \in \mathcal{P}(\mathcal{B}) \mid x \in f, \{x\} = q\}$.
- (c) Compute the strongly equivalent cyclic core of the set covering problem $\langle Q, P, \subseteq \rangle$ by iterating the following rewriting rules:

$$\begin{aligned} \langle Q, P, \subseteq \rangle &\xrightarrow{1} \langle \max_{\subseteq} \tau_P(Q), P, \subseteq \rangle, \\ \langle Q, P, \subseteq \rangle &\xrightarrow{2} \langle Q - E, P - E, \subseteq \rangle \quad \text{with } E = Q \cap P, \\ \langle Q, P, \subseteq \rangle &\xrightarrow{3} \langle Q, \max_{\subseteq} \tau_Q(P), \subseteq \rangle \end{aligned}$$

where τ_P and τ_Q are defined from $\mathcal{P}(\mathcal{B})$ into $\mathcal{P}(\mathcal{B})$ by

$$\begin{aligned} \tau_Q(w) &= \sup_{\subseteq} \{q \in Q \mid q \subseteq w\}, \\ \tau_P(w) &= \inf_{\subseteq} \{p \in P \mid w \subseteq p\}. \end{aligned}$$

- (d) Solve the covering matrix if it is non empty.

The correctness of this minimization procedure is a direct consequence of Theorem 3, since $(\mathcal{P}(\mathcal{B}), \subseteq)$ is a lattice, and since the rewriting process starts with a set P that is the set of prime implicants of f , which is obviously maximal. This minimization procedure is very different from all the other ones for several reasons. First, the set covering problems it works on use the subset relation “ \subseteq ” instead of the belongness relation “ \in ”. Second, it uses two transposing functions τ_P and τ_Q to capture dominance relations. Moreover these transposing functions are endomorphisms, i.e., morphisms whose domains and codomains are the same set (here $\mathcal{P}(\mathcal{B})$). Third, the simplicity of the second rewriting rule (that is equivalent to finding the essential elements of P and consequently simplifying the covering problem) compares favorably with the complexity of the essential elements computation procedures known so far. Fourth, Combinational Sets (CSs, see Appendix B) will be used to represent the sets of products Q and P , which makes the cost of this algorithm independent of the number of products we need to manipulate, in particular the number of prime implicants. In other words the strongly equivalent cyclic core is computed *implicitly*.

If the cost of a cover is not its number of product, but some more complex function satisfying definition 2 (for instance the number of literals), then SCHERZO does not use the third reduction rule so that the covering matrix it generates remains strongly equivalent to the initial optimization problem.

The different sets of products we need to compute can be easily expressed with formulas with quantifiers. Thus one can use the standart BDDs and CSs operators (namely: intersection, union, set difference, substitution, and quantified variable elimination) to perform the different tasks. However,

this direct approach is not interesting from the computational point of view, because substitutions and quantified variable eliminations do not have a polynomial cost.

All the algorithms presented in the sequel are based on quantifier free recursive equations. These algorithms use a divide and conquer strategy based on canonical decompositions of Boolean functions and of sets of products. These decompositions are outlined in Section 7.1. BDDs and CSs described in Appendices A and B are graph data structures based on these canonical decompositions. This yields a straightforward implementation. For the sake of clarity, the algorithms will be described on explicit sets of products, since they can be directly translated into *implicit* manipulations of sets of products, by replacing sets of products with their CSs, and set operations with their correspondent graph combinators. All these algorithms use caches to avoid redundant computations, but we omitted the cache look up for the sake of legibility.

We now explain how to perform the required computations by taking full advantage of the BDD and CS data structure, namely: computing all the prime implicants; computing $\{q \in \mathcal{P}(\mathcal{B}) \mid x \in f, \{x\} = q\}$; computing $\max_{\subseteq} \tau_P(Q)$; and computing $\max_{\subseteq} \tau_Q(P)$.

7.1 Implicit prime implicant computation

Let P be a set of products built on the product space $\mathcal{L} = \prod_k L_k$, and l_k be a literal (i.e., a subset of L_k). We denote P_{l_k} the set of products $\{p \mid l_k \cdot p \in P\}$. We can canonically decompose P as $P = \bigcup_{l_k \subseteq L_k} \{l_k\} \otimes P_{l_k}$, where $P \otimes Q = \{p \cdot q \mid p \in P, q \in Q\}$. In the case where the product space in \mathcal{B} , we have

$$P = P_{1_k} \cup \{\overline{x_k}\} \otimes P_{\overline{x_k}} \cup \{x_k\} \otimes P_{x_k}.$$

For instance, if $P = \{x_1 \overline{x_4}, x_2, x_2 x_4, \overline{x_2} x_3\}$ we have : $P_{1_2} = \{x_1 \overline{x_4}\}$; $P_{\overline{x_2}} = \{x_3\}$; $P_{x_2} = \{1, x_4\}$. A set of products built on \mathcal{B} is completely and uniquely defined by its three components P_{1_k} , $P_{\overline{x_k}}$, and P_{x_k} , which founds the divide and conquer computation strategy.

Theorem 4 . *Let f be a subset of $\mathcal{L} = \prod_k L_k$, and l_k a literal. We denote f_{l_k} the largest subset of $\prod_{j \neq k} L_j$ such that $l_k \cdot f_{l_k} \subseteq f$. We have for any literal l_k*

$$Prime(f)_{l_k} = Prime(f_{l_k}) - \bigcup_{\substack{l'_k \supseteq l_k \\ l'_k \neq l_k}} Prime(f_{l'_k}).$$

Proof. Let us note $Impl(f)$ the set of implicants of f . By definition of f_{l_k} , it is clear that for any subset p of $\prod_{j \neq k} L_j$, the proposition $l_k \cdot p \subseteq f$ is equivalent to $p \subseteq f_{l_k}$. This proves that $Impl(f)_{l_k} = Impl(f_{l_k})$.

A product p belongs to $Prime(f)_{l_k}$ iff the two following properties are satisfied: (1) it is an implicant of f_{l_k} , and (2) $l_k \cdot p$ is not contained by some other prime implicant of f . Thus p must belong to $Prime(f_{l_k})$, and satisfy (2). To falsify (2), one must find a prime implicant of f containing $l_k \cdot p$, which implies that this prime implicant must be written as $l'_k \cdot p'$, with $l'_k \supseteq l_k$ and $p' \supseteq p$, and $p' \in Impl(f_{l'_k})$. First, note that we cannot have $l'_k = l_k$, for if this would mean that p is not maximal in $Impl(f_{l_k})$. Second, if $l'_k \supseteq l_k$ and $l'_k \neq l_k$, the only way for the prime implicant $l'_k \cdot p'$ of f to contain $l_k \cdot p$ where p is maximal in $Impl(f_{l_k})$ is to have $p' = p$. This concludes the proof. \square

It is clear that $f_{l_k} = \bigcap_{x_k \in l_k} f_{\{x_k\}}$. Applying the result above to the particular case of a total Boolean function f defined on \mathcal{B} , we obtain

$$\begin{aligned} Prime(f)_{1_k} &= Prime(f_{\overline{x_k}} \cap f_{x_k}), \\ Prime(f)_{\overline{x_k}} &= Prime(f_{\overline{x_k}}) - Prime(f)_{1_k}, \\ Prime(f)_{x_k} &= Prime(f_{x_k}) - Prime(f)_{1_k}. \end{aligned}$$

```

function Prime(f, k);
if f = 0 return  $\emptyset$ ;
if f = 1 return {1};
let P = Prime( $f_{\bar{x}_k} \cap f_{x_k}$ , k + 1)
     P0 = Prime( $f_{\bar{x}_k}$ , k + 1) - P
     P1 = Prime( $f_{x_k}$ , k + 1) - P in
     return  $P \cup \{\bar{x}_k\} \otimes P0 \cup \{x_k\} \otimes P1$ ;

```

Figure 16: Algorithm *Prime*.

```

function DiveMinIntoP(f, k);
if f = 0 return  $\emptyset$ ;
if k = n + 1 return {1};
return  $\{\bar{x}_k\} \otimes \text{DiveMinIntoP}(f_{\bar{x}_k}, k + 1) \cup$ 
      $\{x_k\} \otimes \text{DiveMinIntoP}(f_{x_k}, k + 1)$ ;

```

Figure 17: DiveMinIntoP.

The CS of $\text{Prime}(f)$ is recursively computed from the BDD of f using this recursive bottom-up scheme [15]. Evaluating $\text{Prime}(f, 1)$, which is defined by the algorithm given in Figure 16, returns the set of prime implicants of the total Boolean function $f(x_1, \dots, x_n)$. Thanks to BDDs and CSs, the cost of this algorithm is independent of the number of prime implicants, which makes it able to handle Boolean functions that have a huge number of prime implicants, as it will be shown in Section 8.

7.2 Computing Q

The evaluation of $\text{DiveMinIntoP}(f, 1)$ returns the CS of $Q = \{q \in \mathcal{P}(\mathcal{B}) \mid x \in f, \{x\} = q\}$ from the BDD of $f(x_1, \dots, x_n)$, see Figure 17.

7.3 Computing $\max_{\subseteq} \tau_P(Q)$

This section shows how the CS of $\max_{\subseteq} \tau_P(Q)$ can be recursively computed without computing $\tau_P(Q)$. We first introduce the function *NotSubSet* defined from $\mathcal{P}(\mathcal{B})^2$ into $\mathcal{P}(\mathcal{B})$ by:

$$\text{NotSubSet}(P, Q) = \{p \in P \mid \forall q \in Q, p \not\subseteq q\}.$$

The term $\text{NotSubSet}(P, Q)$ is the set of products of P that are not contained by any product of Q . This function can be evaluated on CSs using the algorithm given in Figure 18.

Theorem 5 *The CS of $\max_{\subseteq} \tau_P(Q)$ is built by evaluating $\text{MaxTauP}(Q, P, 1)$, where the function MaxTauP is described by the algorithm in Figure 19.*

Proof. The terminal cases are obvious. Since K is the set of products q of Q such that neither the literal \bar{x}_k nor x_k occurs in $\tau_P(q)$, R is the set $(\max_{\subseteq} \tau_P(Q))_{1_k}$. The set $R0$ is $\max_{\subseteq} (\tau_P(Q)_{\bar{x}_k} \cup W)$, where W is dummy complementary term that is a subset of R . Since only the maximal products with respect to $\tau_P(Q)$ have to be kept, we have to remove from $R0$ the products that are contained by a product of R to obtain $(\max_{\subseteq} \tau_P(Q))_{\bar{x}_k}$. The latter is then $\text{NotSubSet}(R0, R)$. The treatment for $R1$ is similar. \square

```

function NotSubSet( $P, Q, k$ );
if  $Q = \emptyset$  return  $P$ ;
if  $P = \emptyset$  or  $1 \in Q$  return  $\emptyset$ ;
if  $P = \{1\}$  return  $\{1\}$ ;
return  $NotSubSet(P_{1_k}, Q_{1_k}, k + 1) \cup$ 
 $\{\bar{x}_k\} \otimes NotSubSet(P_{\bar{x}_k}, Q_{1_k} \cup Q_{\bar{x}_k}, k + 1) \cup$ 
 $\{x_k\} \otimes NotSubSet(P_{x_k}, Q_{1_k} \cup Q_{x_k}, k + 1)$ ;

```

Figure 18: Algorithm *NotSubSet*.

```

function MaxTauP( $P, Q, k$ );
if  $P = \emptyset$  or  $Q = \emptyset$  return  $\emptyset$ ;
if  $P = \{1\}$  return  $\{1\}$ ;
if  $1 \notin P$  and  $Q = \{1\}$  return  $\emptyset$ ;
let  $K = Q_{1_k} \cup NotSubSet(Q_{\bar{x}_k}, P_{\bar{x}_k}) \cup NotSubSet(Q_{x_k}, P_{x_k})$ ;
 $R = MaxTauP(P_{1_k}, K, k + 1)$ ;
 $R0 = MaxTauP(P_{1_k} \cup P_{\bar{x}_k}, Q_{\bar{x}_k}, k + 1)$ ;
 $R1 = MaxTauP(P_{1_k} \cup P_{x_k}, Q_{x_k}, k + 1)$  in
return  $R \cup$ 
 $\{\bar{x}_k\} \otimes NotSubSet(R0, R) \cup$ 
 $\{x_k\} \otimes NotSubSet(R1, R)$ ;

```

Figure 19: Algorithm *MaxTauP*.

7.4 Computing $\max_{\subseteq} \tau_Q(P)$

This section shows how a recursive evaluation scheme directly produces the CS of $\max_{\subseteq} \tau_Q(P)$. We first introduce the function *SupSet* defined from $\mathcal{P}(\mathcal{B})^2$ into $\mathcal{P}(\mathcal{B})$ by

$$SupSet(P, Q) = \{p \in P \mid \exists q \in Q, q \subseteq p\}.$$

The term *SupSet*(P, Q) is the set of products of P that contain at least one product of Q . This function can be evaluated on CSs using the algorithm in Figure 20.

Theorem 6 *The CS of $\max_{\subseteq} \tau_Q(P)$ is built by evaluating $MaxTauQ(Q, P, 1)$, where the function $MaxTauQ$ is described by the algorithm in Figure 21.*

```

function SupSet( $P, Q, k$ );
if  $P = \emptyset$  or  $Q = \emptyset$  return  $\emptyset$ ;
if  $P = \{1\}$  return  $\{1\}$ ;
if  $1 \notin P$  and  $Q = \{1\}$  return  $\emptyset$ ;
return  $SupSet(P_{1_k}, Q_{1_k} \cup Q_{\bar{x}_k} \cup Q_{x_k}, k + 1) \cup$ 
 $\{\bar{x}_k\} \otimes SupSet(P_{\bar{x}_k}, Q_{\bar{x}_k}, k + 1) \cup$ 
 $\{x_k\} \otimes SupSet(P_{x_k}, Q_{x_k}, k + 1)$ ;

```

Figure 20: Algorithm *SupSet*.

```

function MaxTauQ( $Q, P, k$ );
if  $Q = \emptyset$  or  $P = \emptyset$  return  $\emptyset$ ;
if  $1 \in Q$  and  $1 \in P$  return  $\{1\}$ ;
let  $K = \text{SupSet}(P_{1_k}, Q_{1_k}) \cup$ 
       $(\text{SupSet}(P_{1_k}, Q_{\bar{x}_k}) \cap \text{SupSet}(P_{1_k}, Q_{x_k}))$ ;
 $R = \text{MaxTauQ}(Q_{1_k} \cup Q_{\bar{x}_k} \cup Q_{x_k}, K, k + 1)$ ;
 $R0 = \text{MaxTauQ}(Q_{\bar{x}_k}, P_{1_k} \cup P_{\bar{x}_k}, k + 1)$ ;
 $R1 = \text{MaxTauQ}(Q_{x_k}, P_{1_k} \cup P_{x_k}, k + 1)$  in
return  $R \cup$ 
       $\{\bar{x}_k\} \otimes \text{NotSubSet}(R0, R) \cup$ 
       $\{x_k\} \otimes \text{NotSubSet}(R1, R)$ ;

```

Figure 21: Algorithm *MaxTauQ*.

Proof. The terminal cases are obvious. By definition K is the set of products of P_{1_k} that contain a product of Q_{1_k} , or that contain a product of $Q_{\bar{x}_k}$ and a product of Q_{x_k} . This means that K is the set of products p of P such that $\tau_Q(p)$ does not contain the literal x_k nor \bar{x}_k . The set R is then the set of products of $\max_{\subseteq} \tau_Q(P)$ that do not contain the literal x_k or \bar{x}_k .

It remains to compute the set of maximal sets of $\tau_Q(P)$ that contain either the literal x_k or the literal \bar{x}_k . The set $R0$ is the set of products of $\tau_Q(P)$ that contain the literal \bar{x}_k , and that are maximal with respect to $\tau_Q(P)_{\bar{x}_k}$. In addition with them, $R0$ includes other products containing the literal \bar{x}_k , which belongs to $\tau_Q(P)$, but which are not maximal with respect to products of the set $\max_{\subseteq} \tau_Q(P)$. Since only the maximal products with respect to $\tau_Q(P)$ have to be kept, i.e., the set $\text{NotSubSet}(R0, R)$. A similar proof shows that $\text{NotSubSet}(R1, R)$ is the subset of products of $\max_{\subseteq} \tau_Q(P)$ that contain the literal x_k . \square

7.5 Improvements for solving the covering matrix

Once SCHERZO has produced the two CSs that implicitly represent the covering matrix (indeed the cyclic core if the cost function is the number of products), it solves it using a branch and bound algorithm that switches from the CS representation to an explicit representation when the set covering problem is small enough (roughly less than 20 000 products). However the branch and bound algorithm used by SCHERZO differs from the one used by ESPRESSO-EXACT and ESPRESSO-SIGNATURE by two important improvements that are presented in the next two sections.

Let $C = \langle X, Y, \epsilon \rangle$ be a set covering problem. If the cyclic core of C is not empty, we choose an element y as described in Section 3.3.1, and generate two new set covering problems, the first one defined as $C_l = \langle X - y, Y - \{y\}, \epsilon \rangle$ that assumes that y belongs to the minimal solution, and the second one defined by $C_r = \langle X, Y - \{y\}, \epsilon \rangle$ that assumes that y does not belong to the minimal solution. For a set covering problem C generated at some point of this binary search tree, we note $C.min$ the cost of its minimal solution, $C.lower$ a lower bound of $C.min$ computed thanks to an independent set (see Section 3.3.2), and $C.path$ the cost of the path that yields C , i.e., the sum of the costs of all y 's that have been assumed to belong to the minimal solution before reaching C . We note $C.upper$ the global upper bound, i.e., the cost of the best global solution found so far. We ensure that $C.path + C.lower < C.upper$ is always true, if not, the corresponding branch of root C is pruned, as it is done in ESPRESSO-EXACT.

7.5.1 Pruning with the left-hand side lower bound

Theorem 7 *Let C be a set covering problem. If $C.path + C_l.lower \geq C.upper$, then both C_l and C_r can be pruned, and a strictly better lower bound for C is $C_l.lower$.*

Proof. By definition we have always $C_r.min \geq C_l.min$ since C_r has exactly the same y 's than C_l but more elements to cover. The cost of the best global solution that can be found in C_l is $C.path + Cost(y) + C_l.min$, and this quantity exceeds the global upper bound under the given condition, so C_l can be obviously pruned.

The cost of the best global solution that can be found in C_r is:

$$\begin{aligned} C_r.path + C_r.min &= C.path + C_r.min \\ &\geq C.path + C_l.min \\ &\geq C.path + C_l.lower \end{aligned}$$

So under the given condition, this cost exceeds the upper bound and consequently C_r can be pruned. Moreover since $C.min = \min(Cost(y) + C_l.min, C_r.min)$, we have in this case $C.min \geq \min(Cost(y) + C_l.lower, C_l.lower) = C_l.lower$, since $Cost$ is positive. But $C_l.lower$ is necessarily greater than $C.lower$, for if $C.lower \geq C_l.lower$ we have $C.path + C.lower \geq C.upper$, which is impossible. \square

What is interesting is that if the lower bound of C_l satisfies the given condition, we do not even need to examine C_r . In practice, pruning the right-hand side branch thanks to the lower bound of the left-hand side does not occur very often, but sufficiently enough to reduce the number of recursions of about 10% on some examples.

7.5.2 The limit lower bound theorem

This result is the most important in practice. It reduces the number of recursions by a factor of 10 or more on examples that have a very large search space.

Theorem 8 (Limit Lower Bound) *Let $C = \langle X, Y, \epsilon \rangle$ be a set covering problem and an independent set X' of C . Let $C.lower = \sum_{x \in X'} \min_{y \ni x} Cost(y)$ the lower bound of C obtained thanks to X' . Let*

$$Y' = \{y \in Y \mid y \cap X' = \emptyset, C.path + C.lower + Cost(y) \geq C.upper\}$$

be the set of y 's that do not cover any element of X' , and whose cost added to $C.path + C.lower$ exceeds the upper bound. Then C can be reduced to $\langle X, Y - Y', \epsilon \rangle$.

Proof. Let $y \in Y'$, and assume that y belongs to the minimal solution of C . This produces the subproblem C_l , and X' is still an independent set of C_l since y does not cover any element of X' . Thus we have $C_l.lower \geq C.lower$. So the best global solution that can be found in C_l has the cost:

$$\begin{aligned} C_l.path + C_l.min &= C.path + Cost(y) + C_l.min \\ &\geq C.path + Cost(y) + C_l.lower \\ &\geq C.path + Cost(y) + C.lower \end{aligned}$$

which exceeds the upper bound. So y can be removed from Y . \square

What is interesting is that when the limit lower bound is reached for some y 's, reducing $\langle X, Y, R \rangle$ to $\langle X, Y - Y', R \rangle$ makes in practice the recursion terminate immediately, i.e., the lower bound of the latter nearly always exceeds the upper bound, or a better solution is found. To illustrate the gain the limit

lower bound produces, assume that $Cost(y) = 1$ for all y . Then instead of terminating the recursion when the global lower bound (i.e., $C.path + C.lower$) reaches $C.upper$, it is nearly always pruned when the global lower bound reaches $C.upper - 1$. This gain of 1 in the depth of the search can produce an exponential reduction of the space search and reduces dramatically the exploration time. This effect is even more significant when dealing with more discriminant cost functions, for instance cost functions that depend on the literals.

The use of the limit lower bound enables us to solve or improve the best known solutions of some unsolved cyclic cores⁴, and to reduce dramatically the time needed to explore the search space: the gain in CPU time is more than 10 on some examples.

7.6 Conclusion

We have presented a two-level logic minimization algorithm that is based on rewriting rules within the lattice $(\mathcal{P}(\mathcal{B}), \subseteq)$ [19]. These transposing function based transformations produce a set covering problem strongly equivalent to the cyclic core of the Boolean function to minimize. They also allow the use of the efficient CS data structure all along the computation. The effectiveness of CSs to implicitly represent sets of products makes the complexity of this algorithm independent from the number of minterms and prime implicants of f , and the uniformization of the workspace to the lattice $(\mathcal{P}(\mathcal{B}), \subseteq)$ makes this algorithm able to handle problems that are out of reach of any other technique known so far (see Section 8).

In the case when the (strongly equivalent) cyclic core of the Boolean function f to be minimized is empty, a minimal prime cover of f can be immediately built by choosing, for each of the essential elements that have been discovered during the fixpoint computation, a prime implicant of f with a minimal cost that contains it. It is interesting to note that if the cost function has the form $Cost(l_1 \cdots l_n) = c_1(l_1) * c_2(l_2) * \cdots * c_n(l_n)$, where $*$ is any associative operator and l_k 's are literals, then extracting from a set of products P the products that minimize the cost function can be done with a complexity linear with respect to the size of the CS representing P [18].

In the case when the cyclic core is not empty, a branch-and-bound algorithm is used to solve it. It mixes both implicate CS based techniques and explicit techniques. Its efficiency comes mainly from the use of the *limit lower bound* that dramatically reduces the search space.

8 Experimental results

We compare 3 two-level logic minimization methods. The first one is ESPRESSO-EXACT [57], based on the Quine–McCluskey procedure presented in Section 3. The second one is ESPRESSO-SIGNATURE, the signature cube based method [7] presented in Section 5. The third one [19] is SCHERZO, described in Section 7.

There are two reasons for which ESPRESSO-EXACT can fail in minimizing a function. Either the number of prime implicants is too large to be computed explicitly, or the set covering problem ESPRESSO-EXACT generates is too complex to be solved. Note that if ESPRESSO-EXACT can minimize a function, then ESPRESSO-SIGNATURE can too, since the latter generates smaller covering. However ESPRESSO-SIGNATURE is still limited by the number of maximal signature cubes. SCHERZO generates two graphs, namely CSs, that implicitly represent a strongly equivalent covering matrix (this matrix is strongly

⁴For the MCNC benchmark [68], a solution of size 65 (instead of 67) has been found for *ex5*, 259 (instead of 261) for *max1024*, 122 (instead of 125) for *bench1*, 100 (instead of 104) for *test4*, and 477 (instead of 491) for *test3*.

equivalent to the cyclic core when the cost function is the number of products). The advantage is that SCHERZO is not limited by the size of the covering matrix.

Any function that can be minimized with ESPRESSO-SIGNATURE can also be minimized by SCHERZO, providing that one can build the BDD of the function and the CS of its prime implicants. As far as we looked from the theoretical point of view, nothing prevents the existence of a function that could be treated by ESPRESSO-SIGNATURE but not by SCHERZO: a function with a polynomial sum-of-products representation but that has an exponential

BDD representation would be such an example. However it is not clear whether such a function can still have a polynomial number of maximal signature cubes. So far we have not found such a counter example.

Among the 134 examples of the MCNC benchmark [68] that describe partial vectorial Boolean functions, ESPRESSO-EXACT cannot compute explicitly the prime implicants of 17 of them, and 3 are unsolved though their covering matrices have been built. The 114 examples ESPRESSO-EXACT can solve are the “easy” examples. For solving the 114 “easy” examples on a DEC 5000, ESPRESSO-EXACT takes 9264 seconds [59], ESPRESSO-SIGNATURE 7777 seconds [59], and SCHERZO requires 471 seconds. The CPU time measured for SCHERZO includes: reading the file, building the BDDs of the function ff described in Section 2.1 (its variable ordering is determined with the help of the heuristics introduced in [67] and the “sifting” technique [58]), minimizing ff , and recovering the original solution. The most time consuming operation consists of building the BDDs of ff and computing implicitly its cyclic core. This process is especially costly when dynamic variable ordering is necessary. On these “easy” examples SCHERZO is 16 times faster than ESPRESSO-SIGNATURE and 19 times faster than ESPRESSO-EXACT. Table 1 shows the CPU time in seconds on a DEC 5000 to minimize 33 of these examples that represent about 90% of the cumulated time for solving the 114 easy examples. On these “easy” examples, most of the efficiency of SCHERZO comes from the use of the limit lower bound.

The remaining 20 “hard” examples, in addition with 3 other examples on which ESPRESSO-EXACT fails, are known as the 23 hard ESPRESSO-EXACT examples. Table 2 gives the characteristic of the hard ESPRESSO-EXACT examples. We clearly see that the algorithm presented Section 7.1 is not limited by the huge number of prime implicants of some functions, thanks to the implicit representation of sets of products it uses. For instance the CS of the 1.12 $e+15$ prime implicants of *mish* is computed in 0.3 seconds on a DEC 5000, and it has only 2435 vertices. A “?” in column **B** indicates that it is only an upper bound on the minimal sum-of-products, obtained thanks to an irredundant prime cover that has not been proven as minimal. A “*” indicates an improvement due to SCHERZO, either by solving a previously open problem, or by providing with a better solution than the one known so far.

Table 3 compares the results of ESPRESSO-SIGNATURE reported in [36] and those of SCHERZO on the hard espresso examples. Column **row** \times **col** give the number of rows and columns of the covering matrix that is produced by both methods. For ESPRESSO-SIGNATURE, the number of rows is the number of maximal signature cubes, and the number of columns is the number of prime implicants that contain a maximal signature cube. For SCHERZO, it is the size of the cyclic core. Column **Tmatrix** gives the CPU time that ESPRESSO-SIGNATURE needs to generate the covering matrix. Column **Tcc** gives the CPU time that SCHERZO needs to implicitly compute the cyclic core. A “na” indicates a non available datum. A “-” indicates that the computation failed at the step given by the column when it occurs.

The size of the matrix SCHERZO produces is obviously smaller than the one generated by ESPRESSO-SIGNATURE since the latter does not compute directly the cyclic core. SCHERZO is able to generate the cyclic core of all examples, while ESPRESSO-SIGNATURE is sometimes unable to generate a covering matrix because of a too large number of maximal signature cubes. For instance, *mainpla*, *soar*, and *ti*

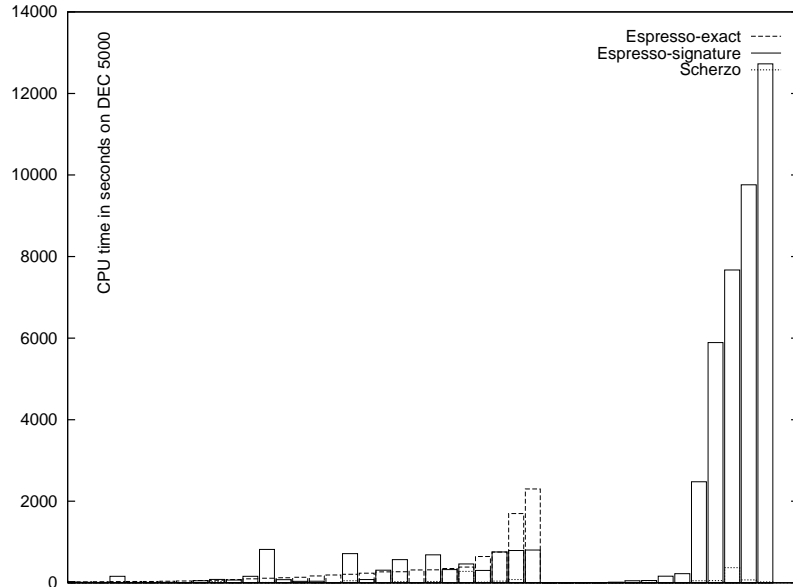


Figure 22: Espresso-Exact, Espresso-Signature, and Scherzo.

have 10 364, 6 707 717, and 17 716 maximal signature cubes respectively ⁵.

On the 14 hard examples that both methods can solve, SCHERZO is nearly 50 times faster than ESPRESSO-SIGNATURE. Moreover, SCHERZO is able to minimize 6 other hard examples that ESPRESSO-SIGNATURE cannot handle because of the too large number of maximal signature cubes, or because the covering matrix cannot be solved without the limit lower bound theorem. The 3 remaining examples are still unsolved by both methods, but SCHERZO improved the best solution known so far of one of them.

Table 4 gives results of SCHERZO on some other hard examples. Some of these examples cannot be handled by ESPRESSO-EXACT because of too many prime implicants. Also ESPRESSO-SIGNATURE cannot handle some of these examples because of the too large number of maximal signature cubes. For instance, *rip08*, *s382*, *seq*, *s1196*, *s344*, and *addsub* have respectively 11131, 29 455, 44 374, 197 423, 357 178, and 33 270 071 maximal signature cubes.

Figure 22 shows the \log_{10} of the CPU time required by the three minimizer ESPRESSO-EXACT, ESPRESSO-SIGNATURE, and SCHERZO, on the last 45 examples of increasing complexity that ESPRESSO-SIGNATURE can solve. We clearly see the double exponential of both ESPRESSO-EXACT and ESPRESSO-SIGNATURE, due to the fact that when treating a function of “size” n , the covering matrix can be in $\Omega(2^n)$, and solving it is NP-complete. SCHERZO has a completely different behavior. Its exponential growing is expected to occur on much more complex examples. It is significant to notice that the more complex the problem is (because of the number of prime implicants of the function, the size of its covering matrix, or the inherent difficulty of solving the covering matrix), the greater the efficiency of SCHERZO is compared to the one of ESPRESSO-EXACT or ESPRESSO-SIGNATURE. However it appears very difficult to capture the complexity of its behavior to explain this efficiency. We claim that this efficiency is primarily due to a shifting of the complexity thanks to the BDD and CS data structures.

⁵Note that given the set covering problem $\langle f, P, \epsilon \rangle$, we have $\sigma(f) = \tau_P(Q)$, where Q is the set of products resulting from diving f into $\mathcal{P}(\mathcal{B})$. This allows us to count the number of maximal signature cubes by counting the number of elements of $\max_{\subseteq} \tau_P(Q)$, which can be done with a cost linear w.r.t the size of the CS representing this set.

9 Conclusion

Two-level logic minimization is a well known problem of computer science that applies in logic synthesis, reliability analysis, and automated reasoning. This paper gave an overview of the methods that have been proposed to solve this problem. We presented the widely used Quine–McCluskey algorithm (ESPRESSO-EXACT), that is limited by the number of prime implicants of the Boolean function to be minimized. We introduced new concepts to establish transformations of the minimization problem into some other ones, in which good properties can be exploited to yield an original minimization algorithm. A good example is the signature cube based minimization method (ESPRESSO-SIGNATURE) that does not necessarily require the computation of all prime implicants. Going further we studied how to solve set covering problems over lattices by exploiting the properties of lattices, and by using transformations that preserve the structure of the problem. The application of this essential result to two-level logic minimization yields the new minimization procedure SCHERZO. First, this algorithm is no longer limited by the number of prime implicants thanks to the use of efficient graph representations of Boolean functions and sets of products. Second, it fully exploits isomorphic transformations of sets covering problems within the same workspace, namely the complete lattice made of products and the subset relation. Third, it improves the branch and bound technique thanks to the limit lower bound theorem that dramatically reduces the number of recursions. These techniques enables SCHERZO to be significantly faster than the best methods known so far, and allows the two-level minimization of functions that are out of reach of any other algorithm.

Acknowledgment

The author would like to thank Anna Karlin, Hervé Touati, and the reviewers for their constructive comments on the early draft of this paper. SCHERZO has been developed with the TiGeR library written by Jean Christophe Madre, Hervé Touati, and the author.

A Binary Decision Diagrams

Several normal forms of Propositional Logic have been used in the past as the basis of automated propositional provers and constraint solvers. Some of these normal forms have the remarkable property of being *canonical*. These canonical forms are very appealing because they support very simple proof procedures. Two formulas written in canonical form are equivalent if and only if they are syntactically equal.

The computational cost of a proof procedure based on canonical rewriting depends on the time needed to rewrite the formula to be proved valid into its canonical form. This time is in the worst case, for all canonical forms, exponential with respect to the size of the formula to be rewritten. The problem is that this worst case was very easily obtained for all the canonical forms that were known until very recently.

Binary decision diagrams (BDDs) are a graph based canonical representation of Boolean functions that is widely used in CAD for verification and synthesis [11]. This representation is strongly related to Boole’s expansion theorem [5], also called Shannon decomposition since his work on switching functions [61]. The Shannon decomposition⁶ of a total Boolean function f with respect to a variable x is the couple of functions $(f_{\bar{x}}, f_x)$ [5] such that: $f = \bar{x}f_{\bar{x}} + xf_x$. When iterated for all the variables of

⁶This is a particular case of the decomposition given Section 7.1

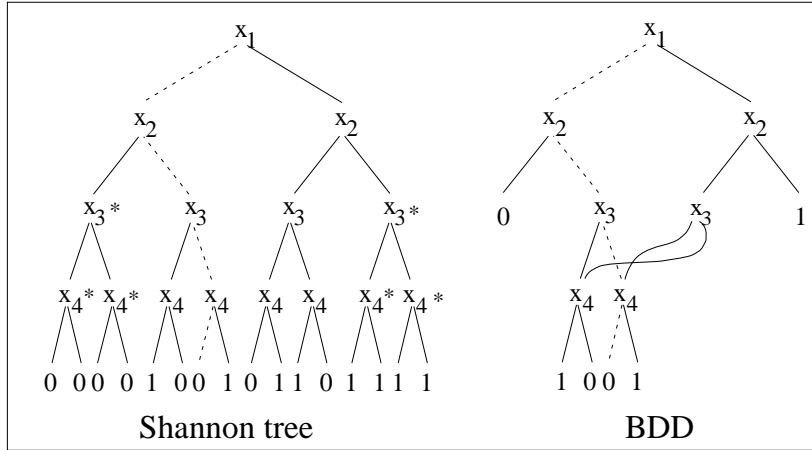


Figure 23: Shannon tree and BDD of the function $x_1(x_3 \oplus x_4) + x_2(x_3 \Leftrightarrow x_4)$.

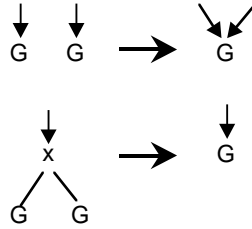


Figure 24: Reduction rules rewriting a Shannon tree into a BDD.

f , the decomposition process defined above associates the function f with its *Shannon tree* which is unique modulo the variable ordering used during the decomposition [1]. The left-hand side of Figure 23 shows the Shannon tree of the function $x_1(x_3 \oplus x_4) + x_2(x_3 \Leftrightarrow x_4)$. The value of the function for any assignment of its variables can be found by following the path that this assignment defines in the tree, for instance the path defined by the assignment $x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0$ is marked with a dotted line.

Since the size of a Shannon tree is exponential with respect to the number of variables, several attempts have been done to compress this tree [1, 9]. The binary decision diagram (BDD) of the function f is a compacted representation of its Shannon tree obtained for a given variable ordering. It can be obtained by applying on Shannon tree the two compaction rules shown in Figure 24, which yields a canonical form [9]. The first compaction rule consists in eliminating all the vertices of the tree that have isomorphic sons because these vertices do not contain any valuable information about the function. For instance the vertices marked with a “*” in Figure 23 are useless. The second compaction rule consists of identifying all remaining isomorphic subtrees, so that the tree becomes an acyclic directed graph called the BDD of f . The right-hand side of Figure 23 shows the BDD obtained of the Shannon tree given on the left-hand side.

Both compaction rules participate in making the size of BDDs, i.e., their number of vertices, much smaller than the number of vertices in their associated Shannon trees. However the size of the BDD of a function f , noted f , depends heavily on the variable ordering that has been chosen to build this graph [9]. For instance the BDD of the function $(x_1 \oplus x_{2n}) + (x_2 \oplus x_{2n-1}) + \dots + (x_n \oplus x_{n+1})$ has a $O(n)$ size with the variable ordering $x_1 < x_{2n} < x_2 < x_{2n-1} < \dots < x_{n-1} < x_n$, but has a $O(2^n)$ size with the variable ordering $x_1 < x_2 < \dots < x_{2n}$. Moreover, there exist functions that do not have tractable

BDDs for any variable ordering [10].

An algorithm has been developed to compute variable orderings that minimize the number of vertices in the BDD of a function f [23]. However this algorithm has a complexity in $O(n^2 3^n)$, where n is the number of variables of f , so it cannot be used for functions with more than 10 variables. For this reason, many efforts have been spent to develop ordering heuristics that exploit the structure of the expression representing a function to produce a good variable ordering, that is an ordering for which the size of the generated BDD is tractable [12, 40].

The binary Boolean operators can be evaluated with quadratic complexity on BDDs built with the same variable ordering [9]. Negation can be evaluated in linear time on BDDs, and in constant time on BDDs with binary attribute on edges to denote the negation [38, 41]. These polynomial complexities are a remarkable property that makes BDDs very different from previously used representations of Boolean functions. For instance, the complementation of Achille's heel function is exponential when representing it with disjunctive normal forms [6]. BDDs are a very interesting representation of Boolean functions and sets because there is *no relation at all* between the number of elements in a set and the size of the BDD that denotes this set through its characteristic function, so that huge sets can potentially be represented by small BDDs [14]. The correspondence existing between the set operators and the Boolean operators allows us to perform operations on sets with a complexity that is not related to the number of elements of these sets but to the sizes of the BDDs that denote them.

B Combinational sets

The purpose of this appendix is to explain the CS data structure used to represent sets of products in a compact and canonical way.

A set of products is nothing but a set of combinations of the literals $\{x_1, \bar{x}_1, \dots, x_n, \bar{x}_n\}$. Moreover, when dealing with prime implicants, most of the literals (indeed at least half of them) do not occur in the combinations. So we are interested in representing *sparse* sets of combinations.

Let $S = \{s_1, \dots, s_n\}$ be a set of objects. A combination is a subset of S . A combinational set C is a set of combinations. For instance $C = \{\{\}, \{s_1, s_2\}, \{s_1, s_3, s_7\}, \{s_4\}\}$ is a combinational set. The set C can be decomposed w.r.t to an element s_k in a unique way as

$$C = C_{\bar{s}_k} \cup \{s_k\} \times C_{s_k}.$$

The set $C_{\bar{s}_k}$ is the set of combinations belonging to C that do not contain s_k , and C_{s_k} is made of combinations belonging to C that contain s_k from which s_k has been removed. For instance with $k = 1$, we have $C_{\bar{s}_1} = \{\{\}, \{s_4\}\}$ and $C_{s_1} = \{\{s_2\}, \{s_3, s_7\}\}$.

When this decomposition is recursively applied with respect to all elements of S in a given order, the combinational set C can be represented by a binary tree whose leaves are $\{\}$ and $\{\{\}\}$. This tree is canonical with respect to the element ordering used during the decomposition, and it has obviously an exponential size with respect to n .

Note that if $\{\}$ is represented by 0, if $\{\{\}\}$ is represented by 1, and if the elements s_k are seen as Boolean variables that values to 0 (respectively to 1) when the element s_k does not appear (respectively appears) in a given combination, then the resulting binary tree yielded by this decomposition is nothing but the Shannon tree of the characteristic function of C . Thus a combinational set, in particular a set of products, can be represented by a BDD [14, 15]. But assume that we are interested in representing a combinational set C made of *sparse* combinations. In this case, on any path from the root of the Shannon tree of the characteristic function of C to the leaf 1, most of the Boolean variables s_k will be assigned to 0. This suggests another compression scheme to reduce the size of the tree [16]. S. Minato

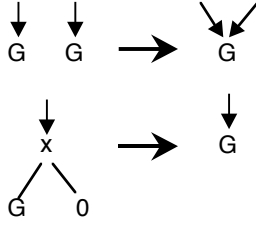


Figure 25: Reduction rules rewriting a set of sparse combinations into a CS.

proposed the two following compaction rules [42] shown on the right-hand side of Figure 25: (1) all isomorphic subgraphs are shared in memory; (2) a vertex that has a right branch that points to zero is removed and replaced with the vertex pointed to its left branch. The resulting graph, which we will call the CS (for Combinational Set) of C , is then a compacted representation of C that is canonical w.r.t. to the ordering of the elements $\{s_1, \dots, s_n\}$.

Set operations can be performed on CSs with a cost related to the size of the CSs, but not to the number of combinations of the combinational sets denoted by the CSs. Union, intersection, and set difference can be performed on CSs with a quadratic worst case complexity, as disjunction and conjunction for BDDs. Complementation is also quadratic on CS, while the negation on BDDs can be done in constant time. Attributed edges can also be used to indicate whether or not $\{\}$ belongs to a combinational set. This allows to add or remove the empty set from a combinational set in constant time, which is very useful when combinational sets represent sets of products. It is also very useful for implementing low cost set operations [42], as already pointed out in [16]. The main advantage of CSs compared to BDDs is their efficiency to represent sets of sparse combinations thanks to compaction rule (2). CSs are then particularly effective when addressing prime cover computation problems.

As mentioned in Section 7.1, any set of products P built on the variables $\{x_1, \dots, x_n\}$ can be partitioned in the following canonical way:

$$P = P_{1_k} \cup (\{\bar{x}_k\} \otimes P_{\bar{x}_k}) \cup (\{x_k\} \otimes P_{x_k}),$$

where P_{1_k} , $P_{\bar{x}_k}$, and P_{x_k} are sets of products in which the variable x_k does not occur. For instance, for $P = \{x_1\bar{x}_4, x_2, x_2x_4, \bar{x}_2x_3\}$, we have $P_{1_2} = \{x_1\bar{x}_4\}$, $P_{\bar{x}_2} = \{x_3\}$, and $P_{x_2} = \{1, x_4\}$. The CS representation can be used to implicitly represent any set of products by associating an element x_k^+ (respectively x_k^-) with each literal x_k (respectively \bar{x}_k). Let $CS(P)$ be the CS of P . Then we have

$$\begin{aligned} P_{1_k} &= CS(P)_{x_k^+ x_k^-}, \\ P_{\bar{x}_k} &= CS(P)_{x_k^+ x_k^-}, \\ P_{x_k} &= CS(P)_{x_k^+ x_k^-}. \end{aligned}$$

Since the CS of the sets P_{1_k} , $P_{\bar{x}_k}$, and P_{x_k} resulting from the canonical decomposition of the set of products P with respect to the variable x_k , can be obtained from the CS of P in constant or linear time, it is very easy to implement sets of products manipulation algorithms that use such a divide and conquer strategy with CSs.

Figure 26 shows the CS of the set of products $P = \{\bar{b}\bar{d}, \bar{a}c\bar{d}, \bar{a}b\bar{c}d, acd\}$. Every path from the root of this graph to the leaf “1” can be interpreted, knowing the compaction rule (2), to produce an assignment of the variables x^+ and x^- that defines a product belonging to P . For instance, the dotted path in this CS corresponds with the product $\bar{b}\bar{d}$.

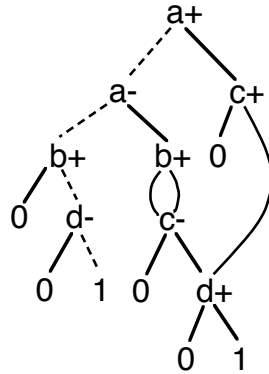


Figure 26: CS of the set of products $\{\bar{b}, \bar{a}b\bar{c}d, acd\}$.

References

- [1] S.B. Akers, Binary Decision Diagrams, *IEEE Transactions on Computers*, Vol. 27, pp. 509–516, 1978.
- [2] T.C. Bartee, Computer Design of Multiple-output Logical Networks, in: *IRE Transactions on Electronic and Computers*, pp. 21–30, March 1961.
- [3] T.C. Bartee, I.L. Lebow, I.S. Reed, *Theory and Design of Digital Machines* (McGraw-Hill, New York, 1962).
- [4] N.N. Biswas, *Introduction to Logic and Switching Theory* (Gordon & Breach Science, New York, 1975).
- [5] G. Boole, *An Investigation of the Laws of Thought* (Walton, London, 1854) (reprinted by Dover Books, New York, 1954).
- [6] R.K. Brayton, G.D. Hachtel, C.T. McMullen, A.L. Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis* (Kluwer Academic Publishers, Dordrecht, 1984).
- [7] R.K. Brayton, P.C. McGeer, J. Sanghavi, A.L. Sangiovanni-Vincentelli, A New Exact Minimizer for Two-level Logic Synthesis, in: *Logic Synthesis and Optimization*, T. Sasao (Ed.) (Kluwer Academic Publishers, Dordrecht, 1993) pp. 1–31.
- [8] F.M. Brown, *Boolean Reasoning* (Kluwer Academic Publishers, Dordrecht, 1990).
- [9] R.E. Bryant, Graph-based Algorithms for Boolean Functions Manipulation, *IEEE Transactions on Computers*, Vol. C-35, No. 8, pp. 677–692, August 1986.
- [10] R.E. Bryant, On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication, Carnegie Mellon University Research Report, September 1988.
- [11] R.E. Bryant, Symbolic Boolean Manipulations with Ordered Binary Decision Diagrams, *ACM Computing Surveys*, Vol. 24, No. 3, pp. 293–318, September 1992.

- [12] K.M. Butler, D.E. Ross, R. Kapur, M.R. Mercer, Heuristics to Compute Variable Orderings for Efficient Manipulations of Ordered Binary Decision Diagrams, *Proc. 28th Design Automation Conference*, pp. 417–420, San Francisco, California, June 1991.
- [13] A.K. Chandra, G. Markowsky, On the Number of Prime Implicants, *Discrete Mathematics*, Vol. 24, pp. 7–11, 1978.
- [14] O. Coudert, J.C. Madre, A New Method to Compute Prime and Essential Prime Implicants of Boolean Functions, *Proc. Brown/MIT Conference on Advanced Research in VLSI and Parallel Systems*, Cambridge, MA, USA, March 1992.
- [15] O. Coudert, J.C. Madre, Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions, *Proc. 29th Design Automation Conference*, Anaheim, CA, USA, pp. 36–39, June 1992.
- [16] O. Coudert, J.C. Madre, A New Graph Based Prime Computation Technique, in *Logic Synthesis and Optimization*, T. Sasao (Ed.) (Kluwer Academic Publishers, Dordrecht, 1993) pp. 33–57.
- [17] O. Coudert, J.C. Madre, Towards a Symbolic Logic Minimization Algorithm, *Proc. VLSI Design*, Bombay, India, January 1993.
- [18] O. Coudert, J.C. Madre, Fault Tree Analysis: 10^{20} Prime Implicants and Beyond, *Proc. Annual Reliability and Maintainability Symposium*, Atlanta, GA, USA, pp. 240–245, January 1993.
- [19] O. Coudert, J.C. Madre, H. Fraisse, A New Viewpoint on Two-Level Logic Minimization, *Proc. 30th Design Automation Conference*, Dallas, TX, USA, pp. 625–630, June 1993.
- [20] R.B. Cutler, S. Muroga, Useless Prime Implicants of Incompletely Specified Multiple-Output Switching Functions, *International Journal of Computer and Information Sciences*, Vol. 9, No. 4, 1980.
- [21] M. Davis, H. Putnam, A Computing Procedure for Quantification Theory, *Journal of the ACM*, Vol. 7, pp. 201–215, 1960.
- [22] J. Doyle, A Truth Maintenance System, *Artificial Intelligence*, Vol. 12, pp. 231–271, 1979.
- [23] S.J. Friedman, K.J. Supowit, Finding the Optimal Variable Ordering for Binary Decision Diagrams, *IEEE Transactions on Computer*, Vol. C-39, No. 5, pp. 710–713, May 1990.
- [24] J.F. Gimpel, A Reduction Technique for Prime Implicant Tables, *IEEE Transactions on Electronics Computers*, Vol. EC-14, pp. 535–541, 1965.
- [25] D.F. Hasl, Advanced Concepts in Fault Tree Analysis, *Proc. System Safety Symposium*, Seattle, USA, June 1965.
- [26] S.J. Hong, R.G. Cain, D.L. Ostapko, MINI: A Heuristic Approach for Logic Minimization, *IBM Journal R&D*, pp. 443–458, 1974.
- [27] S.J. Hong, S. Muroga, Absolute Minimization of Completely Specified Switching Functions, *IEEE Transactions on Computers*, Vol. 40, pp. 53–65, 1991.
- [28] H.R. Hwa, A Method for Generating Prime Implicants of a Boolean Expression, *IEEE Transactions on Computers*, pp. 637–641, June 1974.

- [29] M. Karnaugh, The Map Method for Synthesis of Combinational Logic Circuits, *AIEE Transactions on Communications & Electronics*, Vol. 9, pp. 593–599, 1953.
- [30] J. De Kleer, An Assumption-based TMS, *Artificial Intelligence*, Vol. 28, pp. 127–162, 1986.
- [31] J. De Kleer, B.C. Williams, Diagnosing Multiple Faults, *Artificial Intelligence*, Vol. 32, pp. 97–130, 1987.
- [32] B. Lin, O. Coudert, J.C. Madre, Symbolic Prime Generation for Multiple-Valued Functions, *Proc. 29th Design Automation Conference*, Anaheim, CA, USA, pp. 40–44, June 1992.
- [33] M.C. Loui, G. Bilardi, The Correctness of Tison’s Method for Generating Prime Implicants, Report R-952, UILU-ENG 82-2218, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1982.
- [34] E.L. Jr. McCluskey, Minimization of Boolean Functions, *Bell System Technical Journal*, Vol. 35, pp. 1417–1444, April 1959.
- [35] E.L. Jr. McCluskey, H. Schorr, Essential Multiple Output Prime Implicants, *Proc. Symposium on Mathematical Theory of Automata*, Vol. 12, Polytechnic Institute of Brooklyn, New York, NY, pp. 437–457, April 1962.
- [36] P.C. McGeer, J. Sanghavi, R.K. Brayton, A.L. Sangiovanni-Vincentelli, ESPRESSO-SIGNATURE: A New Exact Minimizer for Logic Functions, *IEEE Transactions on VLSI*, Vol. 1, No. 4, pp. 432–440, December 1993.
- [37] C. McMullen, J. Shearer, Prime Implicants, Minimum Covers, and the Complexity of Logic Simplification, *IEEE Transactions on Computers*, Vol. C-35, pp. 761–762, August 1986.
- [38] J.C. Madre, J.P. Billon, Proving Circuit Correctness using Formal Comparison Between Expected and Extracted Behaviour, *Proc. 25th Design Automation Conference*, Anaheim, CA, USA, July 1988.
- [39] J.C. Madre, O. Coudert, A Logically Complete Reasoning Maintenance System Based on Logical Constraint Solver, *Proc. Int. Joint Conference on Artificial Intelligence*, Sydney, Australia, August 1991.
- [40] S. Malik, A.R. Wang, R.K. Brayton, A. Sangiovanni-Vincentelli, Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment, *Proc. Int. Conference on Computer Aided Design*, Santa Clara, USA, pp. 6–9, November 1988.
- [41] S. Minato, N. Ishiura, S. Yajima, Fast Tautology Checking Using Shared Binary Decision Diagrams—Experimental Results, in: *Formal VLSI Correctness Verification*, L.J.M. Claesen (Ed.) (North-Holland, Amsterdam, 1989) pp. 107–111.
- [42] S. Minato, Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems, *Proc. 30th Design Automation Conference*, Dallas, TX, pp. 272–277, June 1993.
- [43] E. Morreale, Recursive Operators for Prime Implicant and Irredundant Normal Form Determination, *IEEE Transactions on Computers*, 1970.
- [44] I.B. Pyne, E.L. McCluskey, An Essay on Prime Implicant Tables, *SIAM*, Vol. 9, pp. 604–632, 1961.

- [45] W.V.O. Quine, The problem of Simplifying Truth Functions, *American Mathematics Monthly*, Vol. 59, pp. 521–531, 1952.
- [46] W.V.O. Quine, Two Theorems about Truth Functions, *Bol. Society of Mathematics Mexicana*, Vol. 10, pp. 64–70, 1953.
- [47] W.V.O. Quine, A Way to Simplify Truth Functions, *American Mathematics Monthly*, Vol. 62, pp. 627–631, 1955.
- [48] W.V.O. Quine, On Cores and Prime Implicants of Truth Functions, *American Mathematics Monthly*, Vol. 66, pp. 755–760, 1959.
- [49] R. Reiter, A Theory of Diagnosis From First Principles, *Artificial Intelligence*, Vol. 32, pp. 57–95, 1987.
- [50] R. Reiter, J. de Kleer, Foundations for Assumption-based Truth Maintenance Systems, *Proc. AAAI National Conference '87*, Seattle, pp. 183–188, July 1987.
- [51] V.T. Rhyne, P.S. Noe, M.H. McKinney, U.W. Pooch, A New Technique for the Fast Minimization of Switching Functions, *IEEE Transactions on Computers*, Vol. C-26, No. 8, pp. 757–764, 1977.
- [52] J.A. Robinson, A Machine-oriented Logic Based on the Resolution Principle, *Journal of ACM*, Vol. 12, pp. 23–41, 1965.
- [53] S. Robinson, R. House, Gimpel's Reduction Technique Extended to the Covering Problem With Costs, *IEEE Transactions on Electronic Computer*, Vol. EC-16, pp. 509–514, August 1967.
- [54] J.P. Roth, Algebraic Topological Methods for the Synthesis of Switching Systems, *Transactions of American Mathematical Society*, Vol. 88, No. 2, pp. 301–326, 1958.
- [55] R.L. Rudell, *Multiple-Valued Logic Minimization for PLA Synthesis*, Research Report, UCB M86/65, 1986.
- [56] R.L. Rudell, A.L. Sangiovanni-Vincentelli, Multiple Valued Minimization for PLA Optimization, *IEEE Transactions on CAD*, Vol. 6, No. 5, pp. 727–750, September 1987.
- [57] R.L. Rudell, *Logic Synthesis for VLSI Design*, PhD Thesis, UCB/ERL M89/49, 1989.
- [58] R.L. Rudell, Dynamic Variable Ordering for Binary Decision Diagrams, *Proc. Int. Conference on Computer Aided Design*, Santa Clara, USA, pp. 42–47, November 1993.
- [59] J. Sanghavi, Personal Communication, June 1994.
- [60] T. Sasao, An Application of Multiple-valued Logic to a Design of Programmable Logic Arrays, *Proc. Int. Symposium on Multiple-Valued Logic*, 1978.
- [61] C.E. Shannon, The Synthesis of Two-terminal Switching Function, *Bell System Technical Journal*, Vol. 28, No. 1, pp. 59–98, 1949.
- [62] J.R. Slage, C.L. Chang, R.C.T. Lee, Completeness Theorems for Semantics Resolution in Consequence Finding, *Proc. Int. Join Conference on Artificial Intelligence*, pp. 281–285, 1969.

- [63] J.R. Slagel, C.L. Chang, R.C.T. Lee, A New Algorithm for Generating Prime Implicants, *IEEE Transactions on Computers*, Vol. C-19, No. 4, pp. 304–310, 1970.
 - [64] M. Stone, The Theory of Representations for Boolean Algebra, *Transactions of American Mathematical Society*, Vol. 40, pp. 37–111, 1936.
 - [65] G.M. Swamy, P. McGeer, R.K. Brayton, A Fully Quine—McCluskey Procedure using BDD's, Report #UCB/ERL M92/127, November 1992. Also in: *Proc. Int. Workshop on Logic Synthesis*, Lake Tahoe, CA, May 1993.
 - [66] P. Tison, Generalized Consensus Theory and Application to the Minimization of Boolean Functions, *IEEE Transactions on Electronic Computers*, Vol. EC-16, No. 4, pp. 446–456, 1967.
 - [67] H.J. Touati, H. Savoj, B. Lin, R.K. Brayton, A. Sangiovanni-Vincentelli, Implicit State Enumeration of Finite State Machines using BDDs, *Proc. of Int. Conference on Computer Aided Design*, Santa Clara, USA, November 1990.
 - [68] S. Yang, *Logic Synthesis and Optimization Benchmarks User Guide*, Microelectronics Center of North Carolina, January 1991.
-

About the author

Olivier Coudert received the engineering degree and the master degree in computer sciences and applied mathematics from Ecole Centrale de Paris (Paris, France) in 1987, and the Ph.D. degree in computer sciences from Ecole Nationale Supérieure des Télécommunications (Paris, France) in 1991. From 1988 to 1993 he was at Bull Research Center, where he worked on formal verification of sequential systems, and on reliability analysis (2 patents). He developed with Jean Christophe Madre the fault tree analysis tool MetaPrime marketed in 1993. From 1993 to 1994 he was a member of research staff at DEC Paris Research Labs, where he was mainly active in logic synthesis, verification, and real-time software compilation. He has authored or coauthored more than 30 technical papers. His main current interests are in symbolic and algebraic computations, logic synthesis, and hardware verification.

Dr. Olivier Coudert served as a member of the technical program committees of ICCAD, DAC, EDAC, and CAV. He received a best paper award at ICCD'90 for a paper on formal verification, and a best paper award at DAC'92 for a paper on implicit prime implicant computation.

Name	i/o	#P	B	Esp-Exc	Esp-Sig	Scherzo
<i>add6</i>	12/7	8568	355	234	78.44	0.5
<i>al2</i>	16/47	9179	66	349	324.89	4.3
<i>alcom</i>	15/38	4657	40	97	160.09	2.6
<i>b2</i>	16/17	928	104	14	47.76	3.9
<i>b9</i>	16/5	3002	119	27	7.93	1.0
<i>bc0</i>	26/11	6596	177	643	302.99	18.2
<i>bca</i>	26/46	305	180	267	308.62	8.3
<i>bcb</i>	26/39	255	155	79	69.07	5.3
<i>bcd</i>	26/38	172	117	56	49.16	2.0
<i>ex7</i>	16/5	3002	119	31	7.88	1.0
<i>exep</i>	30/63	558	108	30	160.28	12.3
<i>in1</i>	16/17	928	104	15	47.54	4.0
<i>in3</i>	35/29	1114	74	25	11.65	3.3
<i>in6</i>	33/23	6174	54	191	1.66	2.8
<i>in7</i>	26/10	2112	54	44	2.53	1.1
<i>prom1</i>	9/40	9326	472	1698	790.12	86.7
<i>Z9sym</i>	9/1	1680	84	63	83.15	4.5
<i>addm4</i>	9/8	1122	189	9	27.42	5.6
<i>b3</i>	32/20	3056	210	168	38.63	4.5
<i>bcc</i>	26/45	237	137	123	76.37	6.1
<i>cps</i>	24/109	2487	157	111	816.77	38.6
<i>exps</i>	8/38	852	132	11	40.12	9.1
<i>in4</i>	32/20	3076	211	134	35.14	5.3
<i>intb</i>	15/7	6522	629	270	567.94	28.2
<i>lin.rom</i>	7/36	1087	128	753	755.85	55.2
<i>m181</i>	15/9	1636	41	39	10.92	2.1
<i>mlp4</i>	8/8	606	121	21	29.91	5.0
<i>mp2d</i>	14/14	469	30	16	27.00	3.4
<i>pope.rom</i>	6/48	593	59	15	13.97	11.5
<i>spla</i>	16/46	4972	248	209	714.10	25.1
<i>sym10</i>	10/1	3150	210	386	463.29	18.1
<i>t1</i>	21/33	15135	100	2302	801.83	6.0
<i>tial</i>	14/8	7145	575	319	685.70	36.3
Total	1496	110701	5559	8749	7558.7	421.9
Mean	45	3354	168	265.1	229.0	11.5
Ratio				20.7	17.9	1.0

i/o: #inputs/#outputs
#P: #prime implicants
B: size of the minimal sum-of-products

Table 1: Computation time for 33 examples

Name	i/o	#M	#P	#E	B
<i>accpla</i>	50/69	1.60 <i>e</i> +16	1758057	97	175
<i>ex4</i>	128/28	1.59 <i>e</i> +39	1.83 <i>e</i> +14	138	279
<i>ibm</i>	48/17	1.55 <i>e</i> +15	1.04 <i>e</i> +09	172	173
<i>jbp</i>	36/57	8.00 <i>e</i> +11	2496809	0	122
<i>misg</i>	56/23	1.05 <i>e</i> +18	6.49 <i>e</i> +09	3	69
<i>mish</i>	94/43	4.14 <i>e</i> +29	1.12 <i>e</i> +15	3	82
<i>misj</i>	35/14	2.61 <i>e</i> +11	139103	13	35
<i>pd</i>	16/40	120958	23231	2	96
<i>shift</i>	19/16	4194304	165133	100	100
<i>signet</i>	39/8	1.83 <i>e</i> +12	78735	104	119
<i>ts10</i>	22/16	4194304	524280	128	128
<i>x2dn</i>	82/56	8.84 <i>e</i> +25	1.14 <i>e</i> +16	2	104
<i>x7dn</i>	66/15	3.50 <i>e</i> +20	566698631	378	538
<i>xparc</i>	41/73	1.08 <i>e</i> +13	15039	140	254
<i>mainpla</i>	27/54	3.55 <i>e</i> +09	87692	29	172*
<i>soar</i>	83/94	1.74 <i>e</i> +26	3.30 <i>e</i> +14	2	352*
<i>ti</i>	47/72	4.13 <i>e</i> +14	836287	46	213*
<i>prom2</i>	9/21	3027	2635	9	287*
<i>max1024</i>	10/6	3232	1278	14	259*
<i>ex5</i>	8/63	7620	2532	28	65*
<i>test3</i>	10/35	3543	41344	0	477*?
<i>ex1010</i>	10/10	1471	25888	0	246?
<i>test2</i>	11/35	7122	109099	0	995?

i/o: #inputs/#outputs
#M: #minterms
#P: #prime implicants
#E: #essential prime implicants
B: size of the minimal sum-of-products.
A “?” indicates only an upper bound, and
a “*” indicates an improvement due to SCHERZO.

Table 2: The hard Espresso-Exact problems

Name	ESPRESSO-SIGNATURE				SCHERZO			
	row	col	Tmatrix	Time	row	col	Tcc	Time
<i>accpla</i>	385	768	na	7670	8	8	430	431
<i>ex4</i>	509	838	na	163.2	20	20	6.6	6.7
<i>ibm</i>	173	174	na	1.6	0	0	8.4	8.4
<i>jbp</i>	5192	37644	na	5891	324	209	52.3	72.2
<i>misg</i>	134	200	na	13.9	0	0	7.3	7.3
<i>mish</i>	160	239	na	49.2	0	0	9.6	9.6
<i>misj</i>	79	101	na	2.1	0	0	2.7	2.7
<i>pd</i>	6550	18923	na	9759	1498	1141	91.1	106.8
<i>shift</i>	100	100	na	0.2	0	0	4.1	4.1
<i>signet</i>	132	153	na	55.3	0	0	56.1	56.1
<i>ts10</i>	128	128	na	0.3	0	0	4.5	4.5
<i>x2dn</i>	846	2006	na	223.8	0	0	8.8	8.8
<i>x7dn</i>	2602	5966	na	2478	1968	664	51.3	67.2
<i>xparc</i>	1843	2974	na	12726	224	149	23.8	26.3
Total				39033.6				811.7
Ratio				48.0				1.0
<i>mainpla</i>	–	–	j10 h	–	0	0	1480	1480
<i>soar</i>	–	–	j10 h	–	22540	8927	421.1	1638
<i>ti</i>	–	–	j10 h	–	874	430	28.2	34.1
<i>prom2</i>	1763	2604	282.5	–	1524	1813	60.2	5.5 h
<i>max10241054</i>	1278	1278	47.4	–	916	904	5.5	20.3 h
<i>ex5</i>	795	2451	123.2	–	686	974	64.0	34.4 h
<i>test3</i>	3520	40664	5286	–	3518	26461	1794	–
<i>ex1010</i>	1468	25200	684	–	1466	11568	125.0	–
<i>test2</i>	7105	106933	11400	–	7104	69096	5458	–

row × col: Size of the yielded matrix
Tmatrix: Time needed by ESPRESSO-SIGNATURE to yield the matrix
Tcc: Time needed by SCHERZO to yield the cyclic core
Time: Total minimization time (limited to 10h)

Table 3: Hard espresso examples with Espresso-Signature and Scherzo

Name	i/o	#M	#P	#E	B	Time
<i>cbp16</i>	33/17	7.30 <i>e</i> +10	6.87 <i>e</i> +10	327662	655287*	0.4
<i>cbp32</i>	65/33	6.08 <i>e</i> +20	2.84 <i>e</i> +21	2.14 <i>e</i> +10	42949672823*	1.0
<i>cont</i>	30/28	8.12 <i>e</i> +09	17060286	79	229	1.9
<i>s298</i>	17/20	868352	106307	36	58	1.4
<i>s382</i>	24/27	143293440	11142527	6	150*	27.7
<i>s420</i>	35/18	3.86 <i>e</i> +10	125924	38	58	2.1
<i>s526</i>	24/27	139917312	21523813	40	130	9.2
<i>seq</i>	70/58	2.21 <i>e</i> +22	9.85 <i>e</i> +09	36	472*	62.4
<i>addsub</i>	31/15	1.61 <i>e</i> +10	3.60 <i>e</i> +09	32794	163649*	171
<i>pitch</i>	16/48	750848	27560	35	95	20.5
<i>rip08</i>	16/9	294784	182893	631	1499*	1.9
<i>add8</i>	17/9	589824	338732	1270	2519	0.1
<i>achil8n</i>	24/1	5764801	6561	6561	6561	0.1
<i>alupla</i>	25/5	59653904	25677	1791	2144	37.5

i/o: #inputs/#outputs

#M: #minterms

#P: #prime implicants

#E: #essential prime implicants

B: size of the minimal sum-of-products.

A “*” indicates an improvement due to SCHERZO.

Table 4: Results of Scherzo on hard examples from the MIS and ISCAS benchmarks