COMPLEXITY OF TWO-LEVEL LOGIC MINIMIZATION

by

Christopher Umans, Tiziano Villa and
Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M04/45

4 October 2004

COMPLEXITY OF TWO-LEVEL LOGIC MINIMIZATION

by

Christopher Umans, Tiziano Villa and
Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M04/45

4 October 2004

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Complexity of Two-Level Logic Minimization

Christopher Umans* Tiziano Villa† Alberto L. Sangiovanni-Vincentelli‡

October 4, 2004

Abstract

The complexity of two-level logic minimization is a topic of interest to both CAD specialists and computer science theoreticians.

In the logic synthesis community, two-level logic minimization forms the foundation for more complex optimization procedures that have significant real-world impact. At the same time, the computational complexity of two-level logic minimization has posed challenges since the beginning of the field in the 60's; indeed some central questions have been resolved only within the last few years, and others remain open. This recent activity has classified some logic optimization problems of high practical relevance, such as finding the minimal sum-of-products form, and maximal term expansion and reduction.

This paper surveys progress in the field, with self-contained expositions of fundamental early results, an account of the recent advances, and some new classifications. We include an introduction to the relevant concepts and terminology from computational complexity, as well a discussion of the major remaining open problems in the complexity of logic minimization.

1 Introduction

Computer-aided design specialists [10, 24] and computer science theoreticians [18] alike investigated the computational complexity of logic minimization problems. The motivation lies both in their practical importance for design automation [3, 2, 20], and in their paradigmatic nature in the landscape of computational complexity classes.

Even though logic synthesis has grown increasingly sophisticated, building complex optimization scenarios that include multi-level and multi-valued logic realizations [5] up to regular fabrics of newer kinds [13], two-level logic minimization (good old time PLAs) retains a central role as a key procedure in more complex logic optimization packages.

Therefore an in-depth investigation of two level logic minimization has a special place in studying the complexity of logic synthesis problems. The history of classifying the complexity of two-level logic minimization accompanies the field of computational complexity from its beginnings in the 60s [7] to now. This history extends to a result of only a few years ago settling a conjecture open for more than 20 years, namely, that the following problem is Σ_2^p-complete [23, 22]: "given a sum-of-producte (SOP) form of a logic function, is there an equivalent SOP form with at most a given

*Department of Computer Science, California Institute of Technology, Pasadena, CA 91125. Email: umans@cs.caltech.edu. This research was supported by NSF grant CCF-0346991.

†DIEGM, Universita' di Udine, Via delle Scienze, 208, 33100 Udine, Italy and PARADES, Via di S. Pantaleo, 66, 00186 Roma, Italy. Email: villa@parades.rm.cnr.it.

‡Department of EECS, University of California at Berkeley, Berkeley, CA 94720 and PARADES, Via di S. Pantaleo, 66, 00186 Roma, Italy. Email:alberto@parades.rm.cnr.it.

number of terms?". This result spawned the complexity classification of the main optimization problems routinely solved by software packages like ESPRESSO, such as maximal term expansion and reduction. With this recent activity, important theoretical achievements are linked with state-of-the-art exact or heuristic algorithmic practice.

The relevant material is scattered in various sources and some key older items are hard to find and/or inaccurate in their original version. This paper presents the progress of the field with a self-contained exposition that sets the record straight, states precisely the results, and introduces the most valuable proof techniques. The CAD practitioner should gain an updated understanding of the complexity of relevant fundamental problems, while the theorist will find a gentle survey of this literature.

The remainder of this paper is organized as follows. Sec. 2 introduces basic definitions relating to logic functions. Sec. 3 is a brief introduction to the relevant complexity classes and to their canonical complete problems. Sec. 4 discusses the relation between two-level minimization and covering problems, by introducing Gimpel's reduction that shows the NP-completeness of two versions of this problem. Sec. 5 reports a revised version of Masek's proof that SOP minimization is NP-complete when the input is given by the full truth table of a completely specified function. Sec. 6 discusses the complexity of two-level minimization and various sub-problems used in the major logic minimization suites (i.e., ESPRESSO), when the input is given as a sum-of-products. Many of these problems turn out to be Σ_2^p-complete. Sec. 7 reviews past achievements and major open problems.

2 Minimization of Boolean Functions

Let f be a Boolean function. An *implicant* of f is a conjunction C of literals that implies f. For a conjunction C we denote by $lit(C)$ the set of literals appearing in C. In this notation, if C_1 and C_2 are conjunctions with $lit(C_1) \subseteq lit(C_2)$, and C_1 is an implicant of f, then C_2 also is an implicant of f.

A conjunction C *covers* those assignments with no variable set in contradiction to its setting in C. We write $C_1 \subseteq C_2$ to mean that the minterms covered by C_1 are a subset of the minterms covered by C_2. Note that $C_1 \subseteq C_2$ is equivalent to $lit(C_1) \supseteq lit(C_2)$.

An implicant C is a *prime implicant* (or just a "prime") if the only conjunction $C' \supseteq C$ that is an implicant of f is C itself. An *essential prime implicant* is a prime implicant that covers some assignment that is covered by no other prime implicant.

An implicant is also called a *product* or *term*, and a disjunction of products of literals is called a *sum-of-products (SOP)*. We write $\phi = \vee_{i \in I} t_i$ to mean the SOP formula ϕ with terms t_i for $i \in I$. Formula ϕ is *equivalent* to formula ϕ', written $\phi \equiv \phi'$, if and only if the two SOPs cover exactly the same points (i.e., the functions computed by ϕ and ϕ' are the same).

3 An Introduction to Computational Complexity

In this section we introduce the main concepts from computational complexity needed in the paper. We refer to [6, 14, 1] as standard references.

For the purposes of this paper, a computational problem is formalized as a *decision problem*. Given as input a string (the *instance*), the solution to a decision problem is either "yes" or "no." The set of strings whose answer is "yes" completely specifies the problem, and is sometimes called the *language* associated with the decision problem.

If Σ is a finite alphabet, and $L \subseteq \Sigma^*$ is a language, we define the *complement language*, denoted $\overline{L}$, as the set of strings that are not in L (in other words, $\overline{L} = \Sigma^* - L$). In most reasonable schemes for encoding instances of real-world problems as strings, some strings do not correspond to valid encodings of any instance. For this reason, we often abuse notation slightly and use $\overline{L}$ to refer to only those strings in $\Sigma^* - L$ that are valid encodings of some instance. This is standard and does not affect any of the results we cite or prove.

Example 3.1 *CNF-SAT is the problem of deciding if a given Boolean expression in conjunctive normal form (CNF) has a satisfying assignment. Given a reasonable rule to encode CNF expressions, the language CNF-SAT will contain all strings in Σ^* that encode CNF expressions that are satisfiable. The complement problem $\overline{\text{CNF-SAT}}$ is the problem of deciding if given CNF is unsatisfiable. The language $\overline{\text{CNF-SAT}}$ contains all strings that encode CNF expressions that are unsatisfiable. As noted above, strings in Σ^* that do not encode any CNF expression are neither in CNF-SAT nor in $\overline{\text{CNF-SAT}}$.*

Computational complexity studies the computational resources required to solve problems. Formally, an *algorithm* for a decision problem is a Turing Machine that accepts exactly those strings in the associated language, and rejects those not in the language[1]. We are primarily interested in the worst-case running time of such an algorithm, as measured in steps taken by the Turing Machine. This measure is polynomially related to the running time of the all other commonly used computational mechanisms (for instance a C program running on a Von Neumann computer).

A *complexity class* is a set of languages. As we are interested in running time, an important class is P, the set of languages possessing algorithms that run in time that is a polynomial in the length of the input. Another important class of languages is NP, the set of languages possessing algorithms that run in *nondeterministic* polynomial time. For this paper, we will use the following alternate definition:

Definition 3.1 *A language L is in the class NP if and only if there is another language R in the class P and an integer k for which:*

$$L = \{x : \exists y, |y| \leq |x|^k, (x, y) \in R\}.$$

For example, CNF-SAT is a language in NP. This is true because we can define the language R in P to consist of those pairs (ϕ, A) for which ϕ is a CNF formula, and A is a satisfying assignment for ϕ. We see that CNF-SAT $= \{\phi : \exists A, (\phi, A) \in R\}$, which shows that CNF-SAT is in NP. It can be seen from the above definition that NP is exactly those languages whose "yes" instances possess succinct *witnesses* that can be verified in polynomial time.

To understand the importance of NP in studying the running time required for various problems, we must describe the central notions of *reductions* and *completeness* in computational complexity. A reduction from language A to language B is a transformation T that maps "yes" instances of A to "yes" instances of B and "no" instances of A to "no" instances of B. Formally T is a function from A to B that satisfies $x \in A \Leftrightarrow T(x) \in B$. Intuitively, a reduction T from A to B that runs in polynomial-time implies that B is at least as hard as A.

Let C be a complexity class. A language L is C-hard if every language in C reduces to L in polynomial time. Such a language may be regarded as "at least as hard" as any language in C.

[1]Some problems are undecidable and possess no such algorithm, but they are not of interest here.

A language L that is $\mathcal{C}$-hard and also in the class $\mathcal{C}$ is called $\mathcal{C}$-complete. Such a language may be regarded as a "hardest" language in $\mathcal{C}$. Complete languages are very useful because they allow one to reason about an abstract complexity class by studying a concrete, natural computational problem. Perhaps the most well-known example of a complete problem is CNF-SAT:

Theorem 3.1 (Cook (see [6])) *CNF-SAT is NP-complete.*

The importance of the class NP in studying the time complexity of problems stems from (1) the fact that many natural problems are known to be NP-complete, and (2) the widely-believed assumption that $P \neq NP$. If any NP-hard problem is in P then $P = NP$, and so showing that a problem is NP-hard amounts to a proof (subject to the above assumption) that the problem does not have any polynomial time algorithm.

We now describe some complexity classes beyond P and NP that we will need to correctly classify some of the problems we encounter in this paper.

In general, for a complexity class $\mathcal{C}$, we can define the complement class, denoted $co\mathcal{C}$, to be the set of all complements of languages in $\mathcal{C}$; i.e., $co\mathcal{C} = \{L : \overline{L} \in \mathcal{C}\}$. The complement of P is just P again, but the complement of NP is the class $coNP$, which is believed to be different than NP. Using the fact that P is closed under complement, and Definition 3.1 we have

Definition 3.2 *A language L is in the class $coNP$ if and only if there is another language R in the class P and an integer k for which:*

$$L = \{x : \forall y, |y| \leq |x|^k, (x, y) \in R\}.$$

The canonical coNP-complete language is SOP-VALIDITY (also called DNF-TAUTOLOGY). This language consists of all strings encoding SOP expressions that are satisfiable by all truth assignments, or "valid." From Theorem 3.1 we know that $\overline{CNF - SAT}$ is coNP-complete, and a simple reduction (mapping CNF formula ϕ to SOP expression $\neg\phi$) then shows that SOP-VALIDITY is coNP-complete.

Both NP-complete and co-NP complete problems are "hard" in the sense that they cannot have polynomial-time algorithms under the assumption $P \neq NP$. Still, it is useful to maintain the distinction, because in practice heuristics attacking NP problems can stop once they find a witness; and this positive feature cannot be exploited for a coNP-complete problem (assuming $NP \neq coNP$.)

A more esoteric class that we will need is the class DP, which is defined in terms of NP and coNP:

Definition 3.3 *The class DP is the set of languages L that can be expressed as*

$$L = \{x : x \in A \text{ and } x \in B\}$$

where A is a language in NP and B is a language in $coNP$.

The canonical complete language for DP is the language SAT-UNSAT, which consists of all pairs (ϕ_1, ϕ_2) where ϕ_1 is a satisfiable CNF expression, and ϕ_2 is an unsatisfiable CNF expression. Note that DP contains all of NP (because given a language L in NP, it can be seen to lie in DP by taking $A = L$ and $B = \Sigma^*$) and DP also contains all of coNP (because given a language L in coNP, we can take $A = \Sigma^*$ and $B = L$). Thus a problem that is DP-hard is both NP-hard *and* coNP-hard (although the converse is not necessarily true).

3.1 Oracles and the Polynomial Hierarchy

We say that a Turing Machine is equipped with an *oracle* L, when it has available a subroutine that charges one unit of computation to answer whether a given string is in L. For example, a Turing Machine equipped with a CNF-SAT oracle can in a single step determine whether a CNF expression generated in the course of its computation is satisfiable, by "querying" its oracle. We define complexity classes involving oracles using a standard shorthand: if $\mathcal{C}$ and $\mathcal{B}$ are complexity classes, then $\mathcal{C}^{\mathcal{B}}$ is the class of languages decided by a machine of the type that defines $\mathcal{C}$, augmented with a oracle language in $\mathcal{B}$. This is not a precise definition, but the meaning should be clear for the classes we apply this to.

An example of an oracle complexity class is P^{NP}. This class includes all languages decidable by a Turing Machine running in polynomial time that is equipped with a CNF-SAT oracle, since CNF-SAT is in NP.[2] An example of a language in P^{NP} is the language consisting of all m-tuples of CNF expressions for which an odd number of them are satisfiable. Given such an instance $(\phi_1, \phi_2, \ldots, \phi_m)$, we can make m queries to the CNF-SAT oracle to determine exactly which of the m expressions are satisfiable, and accept if the number of satisfiable expressions is odd.

A refinement of P^{NP} is the class $P_{||}^{NP}$, in which the queries to the oracle are required to be *nonadaptive*. That is, no query depends on the outcome of previous queries, and so the computation can always be organized as follows: first compute in polynomial time a set of polynomially-many oracle queries, then perform the queries "in parallel," and finally decide whether to accept or reject the input with a polynomial-time computation on the input and the outcomes of the parallel queries. The example above in fact lies in $P_{||}^{NP}$. Clearly $P_{||}^{NP}$ is contained in P^{NP}, and we observe that DP, NP and coNP are all contained in $P_{||}^{NP}$. Complexity theorists believe that all of these classes are distinct.

Just as it is meaningful to augment a polynomial-time deterministic Turing Machine with an oracle, we can also augment a polynomial-time nondeterministic Turing Machine with an oracle. This gives rise to an infinite hierarchy of complexity classes collectively called the *Polynomial Hierarchy* (PH). We describe the levels of the PH below using the shorthand for oracle classes, and follow that with an alternate (formal) definition that generalizes Definition 3.1.

$$\begin{array}{rl} \Sigma_0^P = P & \Pi_0^P = P \\ \Sigma_1^P = NP & \Pi_1^P = coNP \\ \Sigma_2^P = NP^{NP} & \Pi_2^P = coNP^{NP} \\ \vdots & \vdots \\ \Sigma_i^P = NP^{\Sigma_{i-1}^P} & \Pi_i^P = coNP^{\Sigma_{i-1}^P} \\ \vdots & \vdots \end{array}$$

Definition 3.4 *A language L is in the class Σ_i^P if and only if there is another language R in the class P and an integer k for which*

$$L = \{x : (\exists y_1)(\forall y_2)(\exists y_3)\cdots(Qy_i), |y_i| \le |x|^k \text{ for all } i, [(x, y_1, y_2, \ldots, y_i) \in R]\},$$

where the sequence of quantifiers alternates, ending with $Q = \exists$ if i is odd, or $Q = \forall$ if i is even.

[2] In fact $P^{CNF-SAT} = P^{NP}$, because for any Turing Machine running in polynomial time that is equipped with *any* language $L \in NP$ as an oracle, we can build another Turing Machine running in polynomial time equipped with a CNF-SAT oracle that decides the same language. The new Turing Machine simply applies the polynomial-time reduction from L to $CNF - SAT$ prior to making each of its oracle calls.

Definition 3.5 *A language L is in the class Π_i^P if and only if there is another language R in the class P and an integer k for which*

$$L = \{x : (\forall y_1)(\exists y_2)(\forall y_3)\cdots(Qy_i), |y_i| \leq |x|^k \textit{ for all } i, [(x, y_1, y_2, \ldots, y_i) \in R]\},$$

where the sequence of quantifiers alternates, ending with $Q = \forall$ if i is odd, or $Q = \exists$ if i is even.

A few facts can be immediately gleaned from these definitions. First $\Pi_i^P = co\Sigma_i^P$ for all i. Second, for each i, Σ_i^P contains Π_{i-1}^P and Σ_{i-1}^P. In particular, Σ_2^P contains both NP and coNP, and it is not much more difficult to show that it contains in addition all of the other classes we have considered, namely DP, $P_{||}^{NP}$ and P^{NP}. The most important class in the PH for this paper will indeed be Σ_2^p; an example of a natural problem in this class follows:

Example 3.2 *The language* EQUIVALENT FORMULAS *consists of those pairs (ϕ, k) where ϕ is a Boolean expression for which there exists an equivalent Boolean expression ϕ' of length at most k.*

This language is in Σ_2^P because we can define a language R that lies in P as follows: R accepts those tuples $((\phi, k), \phi', A)$ for which ϕ and ϕ' are Boolean formulas, ϕ' has length at most k, and ϕ agrees with ϕ' on assignment A. We then see that

$$(\phi, k) \in \text{EQUIVALENT FORMULAS} \Leftrightarrow (\exists\phi')(\forall A)[((\phi, k), \phi', A) \in R].$$

It is not known whether *EQUIVALENT FORMULAS* is Σ_2^p-complete or not, although many people believe that it is. Some natural problems have been shown to be complete for various levels of the PH above the first level (see the surveys [16, 17]), although not nearly the same number as are known to be NP-complete.

The canonical complete problems for the PH are "quantified satisfiability" problems. The instances of these problems are Boolean expressions ϕ built on a set of Boolean variables $\cup_i X_i$ where $X_i = \{x_{i,j} : 1 \leq j \leq m_i\}$ for positive integers m_i. The language $k - QBF$ consists of those expressions for which

$$(\exists X_1)(\forall X_2)\cdots(QX_k)[\phi(X_1, X_2, \ldots, X_k)],$$

where the sequence of quantifiers alternates, ending with $Q = \exists$ if k is odd, or $Q = \forall$ if k is even. Here "$(\exists X_i)$" is to be read as "there exists an assignment of values to the variables $x_{i,1}, \cdots, x_{i,m_i}$", and "$(\forall X_i)$" is to be read as "for all assignments of values to the variables $x_{i,1}, \cdots, x_{i,m_i}$".

Theorem 3.2 (see [14]) *For all $k \geq 1$, the problem k-QBF is Σ_k^p complete.*

4 Two-Level Logic Minimization

We begin our examination of the complexity of two-level logic minimization by considering several variants of the problem of finding a minimum SOP representation of a specified Boolean function, described below:

INCOMPLETE TRUTH TABLE MIN SOP
INSTANCE: Onset and offset of incompletely specified function f (i.e., disjoint sets $A, B \subseteq \{0,1\}^n$), and a positive integer k.
QUESTION:Is there a SOP representation of f (i.e., a SOP formula ϕ for which $x \in A \Rightarrow \phi(x) = 1$ and $x \in B \Rightarrow \phi(x) = 0$) with at most k terms?

FULL TRUTH TABLE MIN SOP
INSTANCE: Onset and offset of completely specified function f (i.e., disjoint sets $A, B \subseteq \{0,1\}^n$ with $A \cup B = \{0,1\}^n$), and a positive integer k.
QUESTION:Is there a SOP representation of f (i.e., a SOP formula ϕ for which $x \in A \Rightarrow \phi(x) = 1$ and $x \in B \Rightarrow \phi(x) = 0$) with at most k terms?

ONSET TRUTH TABLE MIN SOP
INSTANCE: Onset of completely specified function f (i.e., a set $A \subseteq \{0,1\}^n$), and a positive integer k.
QUESTION:Is there a SOP representation of f (i.e., a SOP formula ϕ for which $\phi(x) = 1 \Leftrightarrow x \in A$) with at most k terms?

All three of these problems are in NP because given a "candidate" SOP ϕ with at most k terms, one can determine whether it is a representation of f, in polynomial time in the size of the instance. For the first and second problems, this simply requires evaluating f at all of the points in A and B. For the third problem, we can check that ϕ is a representation of f as follows: we explicitly construct the set $\{x : \phi(x) = 1\}$ by adding the minterms covered by each term in ϕ one at a time and only once. If at any point our set size exceeds $|A|$, we know that ϕ is not a representation of f; otherwise if at the end our set equals A, we know that ϕ is indeed a representation of f.

In fact all three of these problems are NP-complete. We will give proofs of this fact for the first and the third problem in this section. The NP-completeness proof for the problem *FULL TRUTH TABLE MIN SOP* is contained in Section 5. We first discuss the "covering problems" which play an important role in the reductions.

4.1 Covering problems and INCOMPLETE TRUTH TABLE MIN SOP

A classical exact procedure for finding the minimum SOP representation of a Boolean function f is due to Quine-McCluskey [12]. This procedure first computes all of the prime implicants of f, and then finds a minimum cardinality subset of these prime implicants that cover all of the minterms of f.

The first part of this procedure requires polynomial time in the size of the onset of f [19](and so is efficient for any of the three variants of *TRUTH TABLE MIN SOP* that we consider above). The second part of the Quine-McCluskey procedure is a special case of the well-known NP-complete problem below:

MINIMUM COVER
INSTANCE: Collection C of subsets of a finite set S, and a positive integer $k \leq |C|$.
QUESTION:Does C contain a cover for S of size at most k; i.e., is there a subset of $C' \subseteq C$ with $|C'| \leq k$ such that every element of S belongs to at least one member of C'?

MINIMUM COVER has been shown to be NP-complete in Karp's seminal paper [9]. It is also well-known that the problem remains NP-complete even when we restrict the subsets in C to all have size exactly three [6].

Now it *might* be the case that when we restrict to the subproblem of *MINIMUM COVER* consisting of only those instances arising from the Quine-McCluskey procedure, the problem becomes tractable. A couple of papers [7, 15] investigated this question and gave transformations that show that even this subproblem remains NP-complete. We summarize here their main results.

It will be convenient to represent covering problems as matrices, or *covering tables*. The elements of S index the rows, and the columns are indexed by the subsets of C. Entry (i, j) is 1 if subset j contains element i, in which case column j is said to *cover* row i. Formally,

Definition 4.1 *Let X and Y be finite sets. A function $A : X \times Y \to \{0,1\}$ is a* covering table *iff*

- $\forall x \in X \;\; \exists y \in Y : A(x,y) = 1$
- $\forall y \in Y \;\; \exists x \in X : A(x,y) = 1$

A subset $Z \subseteq Y$ is called a cover *of X if $\forall x \in X \; \exists z \in Z : A(x,z) = 1$.*

As noted, a Boolean function f, possibly incompletely specified, gives rise to a covering table. Adapting terminology from [15], we denote by $PI(f)$ the prime implicants of f and by $A_{f^{-1}(1),PI(f)}$ the covering table whose rows are the the minterms of f and whose columns are the prime implicants of f, called also the *Minterm-Prime (MP) table of f.*

Despite the fact that a MP table seems to possess additional structure, it turns out that *every* covering table can arise as the MP table of an incompletely specified function:

Theorem 4.1 *Let $A : X \times Y \to \{0,1\}$ be a covering problem, such that A has no equal rows, and let $k = |Y|$. Then there is an incompletely specified Boolean function $f : D \to \{0,1\}$ with $D \subseteq \{0,1\}^k$ for which $A_{f^{-1}(1),PI(f)}$ equals A (possibly after renaming rows and columns).*

Proof. We define the unate function f as follows. The offset contains a single point: the all zeros point. The onset contains one minterm for each row of A: the minterm whose coordinates are exactly that row of A.

It is easy to verify that the primes of f are exactly those points with a single 1, with all zeros raised to "-".[3] Thus $|PI(f)| = k$. Moreover, the j-th column of A equals the column of $A_{f^{-1},PI(f)}$ labeled with the prime p that has a 1 in the j-th position. This is because for each i, the i-th row of A corresponds to a minterm of f that is covered by p if and only if entry $A(i,j) = 1$. □

Example 4.1 *Consider the following covering table A with $X = \{1,2,3\}$ and $Y = \{1,2,3\}$:*

	1	2	3
1	*0*	*1*	*1*
2	*1*	*0*	*1*
3	*1*	*1*	*0*

The above transformation gives the incompletely specified function f that has $f^{-1}(0) = \{000\}$ and $f^{-1}(1) = \{011, 101, 110\}$. The primes of f are: $1--, -1-, --1$ and $A_{f^{-1}(1),PI(f)}$ is

	$1--$	$-1-$	$--1$
011	*0*	*1*	*1*
101	*1*	*0*	*1*
110	*1*	*1*	*0*

The following problem formalizes the second step in the Quine-McCluskey algorithm:

MINTERM-PRIME MINIMUM COVER
INSTANCE: Minterm-Prime table $A : X \times Y \to \{0,1\}$, and a positive integer $k \leq |Y|$.
QUESTION:Does A contain a cover of size at most k; i.e., is there a cover $Z \subseteq Y$ with $|Z| \leq k$?

Th. 4.1 can be interpreted as a reduction from an instance of *MINIMUM COVER* to an instance of *MINTERM-PRIME MINIMUM COVER*, which gives:

[3]In this section we use an alternative notation for implicants: a string of 0's, 1's, and -'s. A 0 (resp. 1) in location i means that literal $\overline{x}_i$ (resp. x_i) appears in the implicant.

Theorem 4.2 *MINTERM-PRIME MINIMUM COVER is NP-complete.*

Also using Th. 4.1 we have a reduction from *MINIMUM COVER* to *INCOMPLETE TRUTH TABLE MIN SOP*, giving:

Theorem 4.3 *INCOMPLETE TRUTH TABLE MIN SOP is NP-complete.*

This can be interpreted to mean that there are in fact no "better ways" to solve the problem of *INCOMPLETE TRUTH TABLE MIN SOP* than Quine-McCluskey – any alternative approach must also be NP-hard.

4.2 ONSET TRUTH TABLE MIN SOP

The transformation in Th. 4.1 cannot be used directly to prove that *ONSET TRUTH TABLE MIN SOP* or *FULL TRUTH TABLE MIN SOP* are NP-complete. The reason is that it produces a function f whose number of variables k is equal to the number of columns of the original covering table A. A completely specified function consistent with f might have exponentially many minterms, and would certainly have an exponentially large truth table. Thus a reduction to *ONSET TRUTH TABLE MIN SOP* or *FULL TRUTH TABLE MIN SOP* based directly on Th. 4.1 would not be a polynomial-time reduction.

However, a different transformation than the one in Th. 4.1 will allow us to prove NP-hardness of *ONSET TRUTH TABLE MIN SOP.* The same authors [7, 15] describe how to produce a completely specified function that in a certain sense still represents the original covering table:

Theorem 4.4 *Let $A : X\times Y \to \{0,1\}$ be a covering problem, such that A has no row that dominates another* [4]*, and let $n = |X|$. Then there is a completely specified function $g : \{0,1\}^{n+2} \to \{0,1\}$ whose MP table equals A after removal of the essential prime implicants and of the minterms they cover (possibly after renaming rows and columns).*

Proof. We construct g in several steps. We first construct a function g' whose variables are $z_1, z_2, \ldots, z_n$.

- For each $i \in \{1, \ldots, |X|\}$, define minterm $q_i = z_1 \ldots z_{i-1}\overline{z}_i z_{i+1} \ldots z_n$.
- For each $j \in \{1, \ldots, |Y|\}$, define $Q_j = \prod_{i:A(i,j)=0} z_i$.
- Let $p_1, p_2, \ldots, p_m$ be an enumeration of all minterms covered by the Q_j for some j, that are not among the q_i. We may assume that the first ℓ of these minterms have an even number of negated variables, and the remaining $m - \ell$ have an odd number of negated variables.
- We define g whose variables are $z_1, z_2, \ldots, z_{n+2}$ as follows:

$$g = \sum_{i=1}^{n} q_i z_{n+1} z_{n+2} + \sum_{i=1}^{m} p_i z_{n+1} z_{n+2} + \sum_{i=1}^{\ell} p_i z_{n+1} \overline{z}_{n+2} + \sum_{i=\ell+1}^{m} p_i \overline{z}_{n+1} z_{n+2}$$

We can verify that the prime implicants of g are exactly:

- $Q_j z_{n+1} z_{n+2}$ for each j, and

[4]A row dominates another if the 1's in the first row are a superset of the 1's in the second. We can remove dominant rows in polynomial-time. The resulting covering problem is equivalent to the initial one.

- $p_i z_{n+1}$ for $i = 1, 2, \dots, \ell$, and
- $p_i z_{n+2}$ for $i = \ell + 1, \ell + 2, \dots, m$.

The last two of these sets of primes are actually essential prime implicants. After removing these essential prime implicants and the minterms that they cover, we are left with minterms $q_i z_{n+1} z_{n+2}$, and prime implicants $Q_j z_{n+1} z_{n+2}$. Notice that $A(i,j) = 1$ iff $q_i z_{n+1} z_{n+2}$ is covered by $Q_j z_{n+1} z_{n+2}$, and hence the MP table of g is equivalent to A after removal of the essential prime implicants and the minterms they cover. □

Example 4.2 *Starting with the covering table A from Ex. 4.1, we have $q_1 = \bar{z}_1 z_2 z_3$, $q_2 = z_1 \bar{z}_2 z_3$, $q_3 = z_1 z_2 \bar{z}_3$ and $Q_1 = z_1$, $Q_2 = z_2$, $Q_3 = z_3$.*

There are 4 minterms covered by the Q_j that are not among the q_i. They are $p_1 = z_1 z_2 z_3$, $p_2 = z_1 \bar{z}_2 \bar{z}_3$, $p_3 = \bar{z}_1 z_2 \bar{z}_3$ and $p_4 = \bar{z}_1 \bar{z}_2 z_3$. In this example all of the p_i have an even number of negated variables, so we obtain:

$$g = \sum_{i=1}^{3} q_i z_4 z_5 + \sum_{i=1}^{4} p_i z_4 z_5 + \sum_{i=1}^{4} p_i z_4 \bar{z}_5.$$

The primes of g are: $z_1 z_4 z_5$, $z_2 z_4 z_5$, $z_3 z_4 z_5$, $z_1 z_2 z_3 z_4$, $z_1 \bar{z}_2 \bar{z}_3 z_4$, $\bar{z}_1 z_2 \bar{z}_3 z_4$, and $\bar{z}_1 \bar{z}_2 z_3 z_4$, of which all but the first three are essential. After removing the essential prime implicants and the minterms they cover, we are left with minterms $\bar{z}_1 z_2 z_3 z_4 z_5$, $z_1 \bar{z}_2 z_3 z_4 z_5$, $z_1 z_2 \bar{z}_3 z_4 z_5$ and primes $z_1 z_4 z_5$, $z_2 z_4 z_5$, $z_3 z_4 z_5$. This portion of g's MP table is shown below:

	1 − −11	−1 − 11	− − 111
01111	*0*	*1*	*1*
10111	*1*	*0*	*1*
11011	*1*	*1*	*0*

which can be seen to equal the covering table A.

Czort [4] described how to use this construction to prove that *ONSET TRUTH TABLE MIN SOP* is *NP*-complete.

Theorem 4.5 *ONSET TRUTH TABLE MIN SOP is NP-complete.*

Proof. We have already argued that *ONSET TRUTH TABLE MIN SOP* is in *NP*. We reduce from the variant of *MINIMUM COVER* in which every set has size exactly 3 (which remains *NP*-complete). An instance of this problem gives a covering table A in which each column has exactly three ones. Our reduction outputs the function g coming from the transformation in Th. 4.4.

We critically use the fact that A has only three ones per column to ensure that the onset of g has size polynomial in the size of A. Using the terminology of the proof of Th. 4.4, the onset of g has $n + 2m$ points. Each Q_j has exactly $n - 3$ variables, and so it can cover at most 2^3 points. Thus m is at most $|Y|2^3$. So the onset of g has at most $|X| + 16|Y|$ points, which is polynomial in the size of A.

There are exactly m essential prime implicants of g, and the remaining MP table is equivalent to A. Thus g has a SOP representation with at most k terms iff A has a cover of size at most $k - m$. This completes the reduction. □

5 FULL TRUTH TABLE MIN-SOP is NP-complete

Masek [11] showed that the problem of finding a minimum SOP representation is NP-complete *even when the input is given as a fully specified truth table.* For this problem, an input of size n specifies a function of only $\log n$ variables. An efficient algorithm for this problem (one that runs in polynomial-time in the size of the input) is therefore allowed *exponential* time in the number of *variables.* It is further evidence of the hardness of this problem that even under these favorable circumstances, an efficient solution is unlikely.

The heart of Masek's reduction[5] is a method for "embedding" any Boolean circuit C of size n into a $O(\log n)$-dimension Boolean hypercube. Roughly speaking, the gates of the circuit are embedded by setting a cluster of adjacent vertices to one; specified vertices in this cluster are the input and output vertices. These clusters are the *gadgets* used in the reduction. Wires between an input vertex and an output vertex belonging to different gadgets are embedded by setting the hypercube vertices along a path between them to one. The gadgets and wires are separated sufficiently so that no product in a SOP cover of the resulting function can cover vertices belonging to different gadgets, or distinct wires.

By an interpretation described in more detail below, every SOP cover assigns truth values (TRUE or FALSE) to the input and output vertices according to how they are covered. Each gadget is designed so that it is covered most efficiently (i.e. by the fewest products) when it "computes" the intended function of its inputs. Similarly, the wires are designed so that they are covered most efficiently when they "transmit" a consistent truth value from their source to their destination. The final gadget in the embedding has a single input, and is most efficiently covered when that input has value TRUE. By connecting the output of the embedded circuit to this gadget, we ensure that the function on $O(\log n)$ variables defined by this embedding has a small SOP cover if and only if the circuit C is satisfiable.

We will describe the desired embedding with the aid of an *adjacency diagram*, whose vertices correspond to nodes of the hypercube that are set to one, and whose edges connect exactly those pairs of nodes that differ in exactly one bit. As a final step we will argue that the adjacency diagram we produce is realizable in a hypercube of $O(\log n)$ dimensions.

As mentioned, the basic components of the embedding are gadgets with distinguished I/O nodes, and wires between them. In our embedding, every I/O node will be adjacent to a node belonging to a wire. Given a SOP cover, we say that an I/O node is TRUE if it is covered by a product that also covers the adjacent wire node, and FALSE otherwise.

In counting the number of terms in a SOP cover for our embedding, we would like to be able to count the terms that cover each component separately, and then sum them to get the total number. This is not quite possible, because some product may simultaneously cover nodes in a gadget and an adjacent wire. To avoid double counting this product, we adopt the following convention: for a product covering both an I/O node and the adjacent wire node, we charge 1/2 to the gadget and 1/2 to the wire. This simple accounting trick will simplify the exposition that follows significantly.

The AND/OR gadget. The AND/OR gadget has two inputs, i_1 and i_2, and two outputs, which "compute" the AND and OR of the two input truth values, respectively. It has the adjacency structure shown in Fig. 1.

Lemma 1 *Given a SOP cover C, let x and y be the truth values assigned to the I/O nodes labelled i_1 and i_2 in the figure, respectively. Then C covers the AND/OR gadget with 11 terms if it assigns*

[5]Our exposition in this section largely follows Czort [4], with some simplifications.

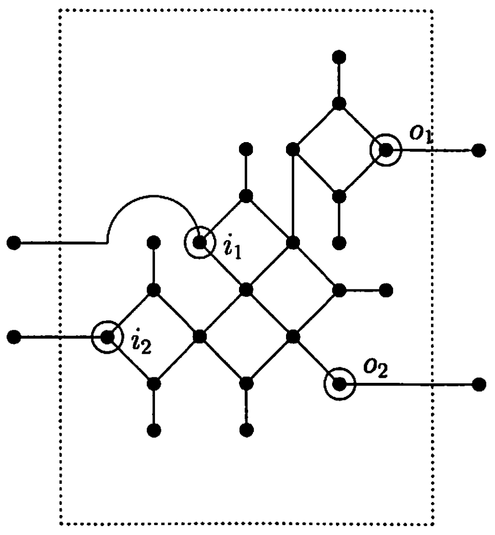

Figure 1: AND/OR gadget.

truth values $x \wedge y$ and $x \vee y$ to the I/O nodes labelled o_1 and o_2, respectively. Otherwise C covers the AND/OR gadget with more than 11 terms.

Fig. 2 shows one possible cover of an AND/OR gadget. In this example, x is FALSE and y is TRUE. Note that the two terms that extend outside the dotted line each contribute 1/2 to the overall sum of 11, according to our accounting convention.

The GENERATOR gadget. The GENERATOR gadget has a single input i_1, and three outputs, which "generate" three copies of the input truth value. It has the adjacency structure shown in Fig. 3.

Lemma 2 *Given a SOP cover C, let x be the truth values assigned to the I/O node labelled i_1 in the figure. Then C covers the GENERATOR gadget with 7 terms if it assigns truth values x to the I/O nodes labelled o_1, o_2, and o_3. Otherwise C covers the GENERATOR gadget with more than 7 terms.*

Fig. 4 shows two possible covers of a GENERATOR gadget, each using 7 terms. In the first cover, x is TRUE; in the second cover x is FALSE.

The WIRE gadget. The WIRE gadget has no I/O nodes of its own; it is used to connect two I/O nodes belonging to GENERATOR or AND/OR gadgets. A WIRE gadget is simply a path between the two I/O nodes. The *length* of the wire is the number of nodes in the path, not counting the I/O nodes at either end. WIRE gadgets of length 2 and 3 are shown in Fig. 5.

Lemma 3 *A SOP cover C covers a WIRE gadget of length k with at most $k/2$ terms if either*

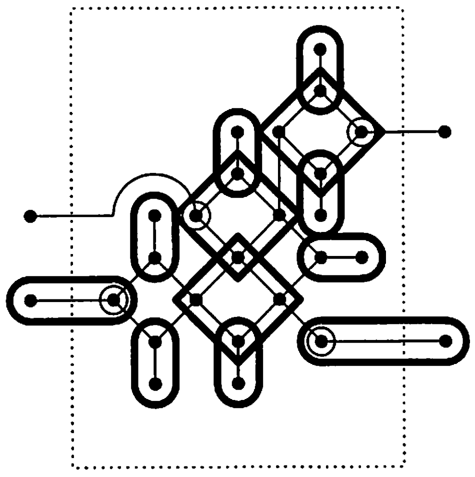

Figure 2: Cover of AND/OR gadget.

- *the wire has* even *length and C assigns the* same *truth values to the I/O nodes connected by the wire, or*
- *the wire has* odd *length and C assigns* opposite *truth values to the I/O nodes connected by the wire.*

Otherwise C covers the WIRE gadget with more than k/2 terms.

Fig. 6 shows covers of the two WIRE gadgets. The first cover uses 1 term, and assigns TRUE to the I/O nodes connected by the wire. The second cover uses 3/2 terms and assigns TRUE to the I/O node on the left and FALSE to the I/O node on the right.

Building the adjacency diagram. We now describe how to attach these gadgets together to "simulate" a given Boolean formula $f(x_1, x_2, \ldots x_n)$. We assume that f has fan-in 2, fan-out 1 AND and OR gates, and WLOG that all the negations occur at the leaves. In fact we will simulate both f and its complement; the added symmetry is needed, for example, because we do not have a separate AND gadget at our disposal, but only a combined AND/OR gadget.

The final adjacency diagram will be coverable by at most N terms if and only if f is satisfiable, where N is the sum over all gadgets in the diagram of the optimal cover numbers given in the previous three lemmas.

We first create a set of "variables" sufficient to supply the variable values at the leaves of the formula. For each i, let n_i be the number of occurrences of the variable x_i as a leaf of the f. Because we are simulating f and its complement, we need to supply $2n_i$ occurrences of x_i; however we also know that there will be an equal number of positive and negative occurrences of x_i, which will make the construction easier. If $n_i = 1$, a single odd-length wire will supply a positive and negative occurrence of x_i, one at either end of the wire.

If $n_i = 2$, we use two GENERATOR gadgets. We connect the input I/O nodes of these two GENERATOR gadgets with an odd length wire, and we connect the third outputs of these two

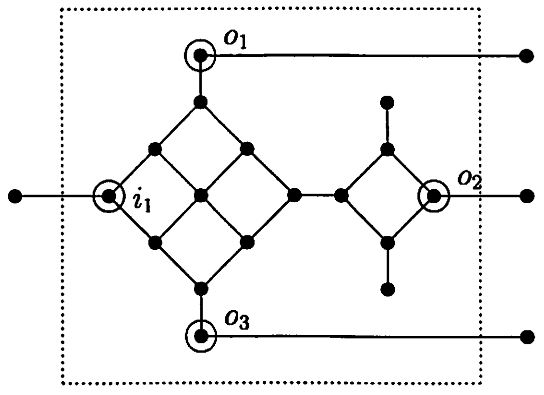

Figure 3: GENERATOR gadget.

generator gadgets with an odd length wire. The remaining 4 outputs (two belonging to each of the GENERATOR gadgets) provide two copies each of positive and negative occurrences of x_i. For $n_i > 2$, we repeat this process: pick an output supplying a positive occurrence of x_i and connect a fresh GENERATOR gadget to it with an even length wire, and similarly, pick an output supplying a negative occurrence of x_i and connect a fresh GENERATOR gadget to it with an even length wire. Finally connect the third outputs of each of these two fresh GENERATOR gadgets with an odd length wire. After connecting $2(n_i - 1)$ GENERATOR gadgets in this fashion, we have, altogether, n_i I/O nodes that supply positive occurrences of x_i and n_i I/O nodes that supply negative occurrences of x_i. Fig. 7 shows a cover of the adjacency diagram of an example of generators for $n_i = 3$; the generator gadgets supply 3 copies of variable x_i and 3 copies of its complement. In the pictured cover x_i is TRUE for all copies and $\overline{x}_i$ is FALSE for all copies.

We can now construct the "gates" that simulate the gates of f. For each AND or OR gate g in the lowest level of the formula we have an AND/OR gadget. Let x_i^a and x_j^b (where $a, b \in \{0,1\}$) be the two literals that are inputs to gate g in the formula f. We attach the two input I/O nodes of the AND/OR gadget to I/O nodes supplying the literals x_i^a and x_j^b with two even length wires. We also add an additional AND/OR gadget that will compute the the complement of g. We attach the two input I/O nodes of this AND/OR gadget to I/O nodes supplying the literals x_i^{1-a} and x_j^{1-b} with two even length wires. Finally, if g was an AND gate, we attach the second output (the "OR" output) of the first AND/OR gadget to the first output (the "AND" output) of the second AND/OR gadget with an odd length wire. If g was an OR gate, we attach the first output (the "AND" output) of the first AND/OR gadget to the second output (the "OR" output) of the second AND/OR gadget with an odd length wire. The remaining two outputs supply the value $g(x_i^a, x_j^b)$ and its complement $\neg g(x_i^a, x_j^b)$.

Now, we move on to the next level of gates, using the outputs from the previous level as the inputs for for the current level, and so on. The final gate's output is attached to an odd length wire, and the complementary output is attached to an even length wire.

Realizing the adjacency diagram. In the previous section we have built an adjacency diagram with $m = O(|f|)$ gadgets. These gadgets have certain points that play the role of input nodes, and

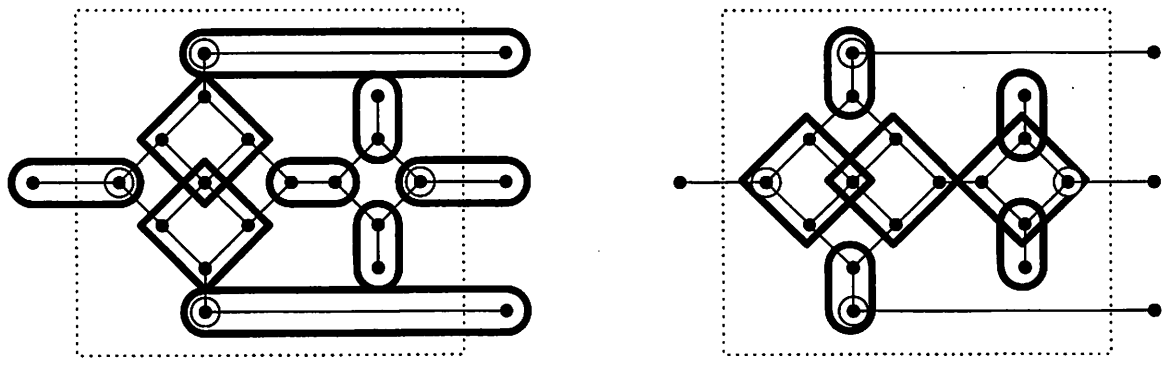

Figure 4: Covers of GENERATOR gadget.

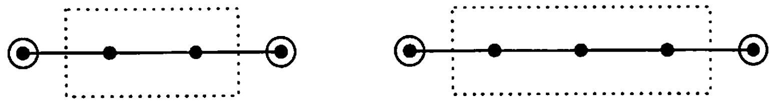

Figure 5: WIRE gadget.

certain points that play the role of output nodes. In the above construction we have used odd-length wires to connect input nodes to input nodes, and also to connect output nodes to output nodes. We have used even length wires to connect output nodes to input nodes.

In this section we describe how to map the points of the adjacency diagram from the previous section into points of a hypercube of dimension $O(\log m)$. This mapping will realize the adjacency diagram: a pair of nodes in the adjacency diagram will be mapped to two nodes in the hypercube that differ in exactly one bit *if and only if* there is an edge in the adjacency diagram between those two nodes. In order to construct this realization, we need to allow ourselves the freedom to choose wirelengths that may differ from the ones in the adjacency diagram; however the odd/even parity of their lengths (which is what matters) will be preserved.

As an example of a realization of the adjacency diagrams of the AND/OR and GENERATOR gadgets in a hypercube of dimension 6, see Fig. 8. This embedding will play an important role in

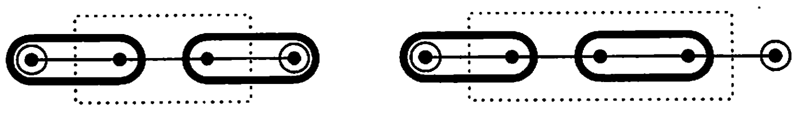

Figure 6: Covers of WIRE gadget.

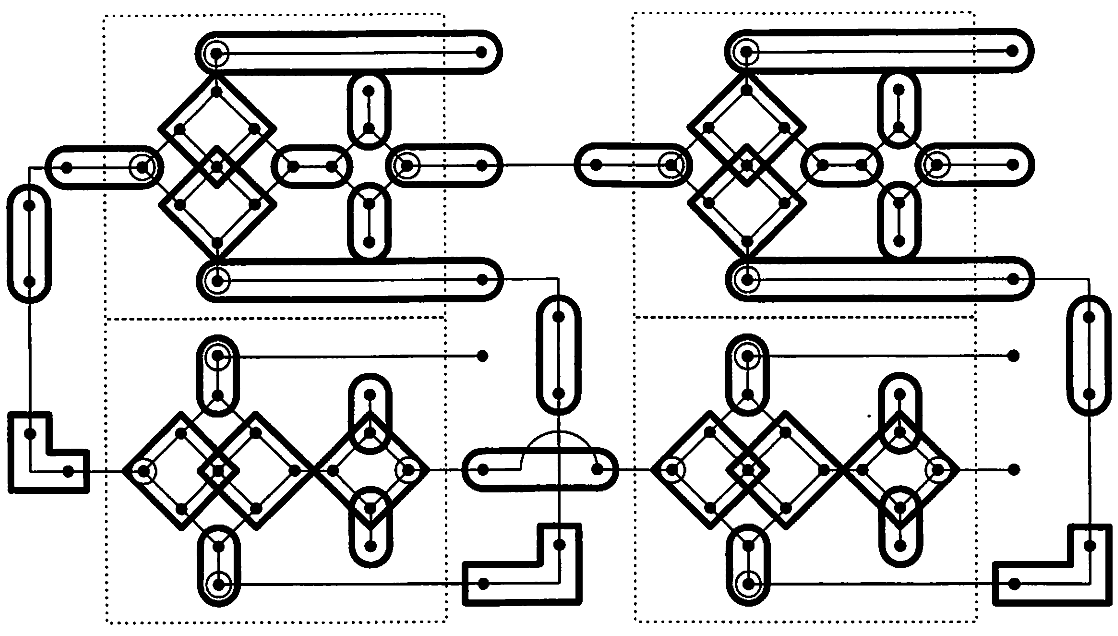

Figure 7: Cover of generators supplying 3 copies of variable x_i (and 3 copies of its complement).

the "global" realization momentarily.

We first describe how to map all of the nodes belonging to AND/OR and GENERATOR gadgets into the hypercube. For each of the m gadgets in the diagram, we assign a unique even parity ID string of $\ell = \lceil \log_2 m \rceil + 1$ bits. Our hypercube will have dimension $2\ell + 12$, and for convenience we routinely refer to nodes of hypercube with 4-tuples whose fields have bitlengths $\ell, \ell, 6$ and 6 respectively. For gadget i that has been assigned ID $id(i)$, we map the node labelled with the 6-bit string x in Fig. 8 to the hypercube node $(id(i), id(i), x, 000000)$. The fact that this is a valid embedding so far follows from the validity of the embeddings in Fig. 8.

We now turn to the wires in the adjacency diagram. Note that, from the previous paragraph and Fig. 8, all of the input I/O nodes are embedded at hypercube nodes with even parity, and all of the output I/O nodes are embedded at hypercube nodes with odd parity (our construction requires to connect input points to input points and output points to output points with odd length wires, and output points to input points with even length wires, this implies that the parity - number of ones in the encoding vectors - of the outputs must be the same and opposite to the parity of inputs). For each wire, we will describe a sequence of adjacent points whose endpoints are the I/O nodes that the wire connects. By the above observation regarding the parity of the various I/O node labels, the length of these embedded wires will retain the required even/odd parity that they have in the adjacency diagram (even if their lengths may change).

A wire between gadgets i and j with ID's $id(i)$ and $id(j)$ respectively is embedded as follows. Let x and y be the 6-bit labels of the I/O nodes we are connecting, belonging to gadget i and j, respectively. The wire will contain the following hypercube nodes:

1. $(id(i), id(i), x, 000000)$

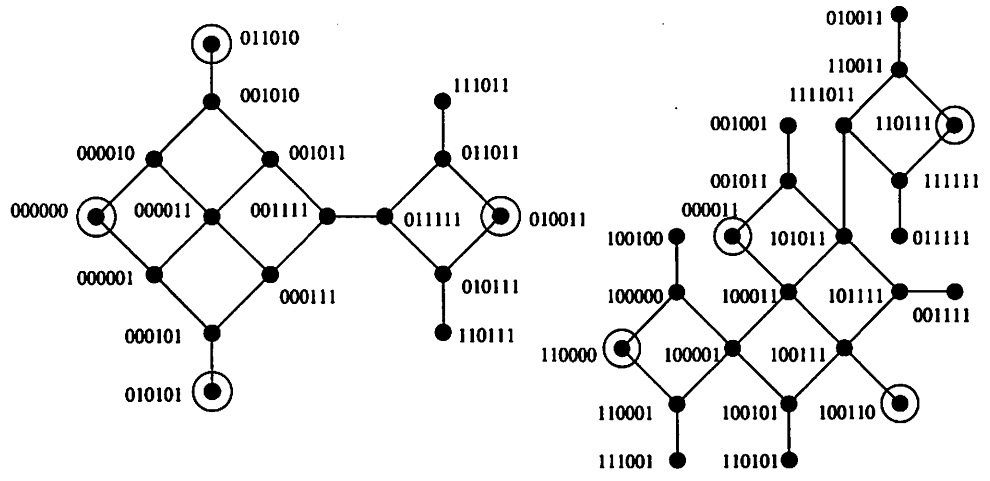

Figure 8: Embedding of GENERATOR (left) and AND/OR (right) gadgets.

2. $(id(i), id(i), x, 110000)$
3. $(id(i), id(j), x, 110000)$
4. $(id(i), id(j), x, 111100)$
5. $(id(i), id(j), y, 111100)$
6. $(id(i), id(j), y, 111111)$
7. $(id(j), id(j), y, 111111)$
8. $(id(j), id(j), y, 000000)$

In addition, we include the hypercube nodes along an arbitrary shortest path between each of these points, changing one bit a time. For example, between the first two points in the above list, we can include hypercube node $(id(i), id(i), x, 100000)$, and then between the second and third points, we include as many additional hypercube nodes as the Hamming distance from $id(i)$ to $id(j)$ minus one, and so on.

There are two special wires in our adjacency diagram that do not terminate at I/O nodes; these are the odd length wire connected to the output of the final gate of the circuit, and the even length wire connected to its complement output. We may take this even length wire to be a length 0 wire (and therefore we need to add no new nodes to our embedding). For the odd-length wire, which is connected to some node with 6-bit label x in some gadget i with label $id(i)$, we add the single hypercube node $(id(i), id(i), x, 100000)$, thus forming a wire of length 1.

The validity of the overall embedding may be verified by a case analysis. The main effort involves verifying that we have not introduced any "unintended" adjacencies in our embedding (to that purpose the second and fourth field of the encoding vectors play a role, especially to enforce that there is no spurious adjacency between points across wires). We obtain the following lemma:

Lemma 4 *The embedding just described realizes the adjacency diagram of the Boolean formula f in a $O(\log|f|)$-dimension Boolean hypercube.*

Our main theorem is:

Theorem 5 *In the above embedding of f, let N be the sum of*

- *the total length of wires divided by 2, and*
- *the number of AND/OR gadgets times 11, and*
- *the number of GENERATOR gadgets times 7.*

Then the function whose onset is exactly those hypercube nodes in the above embedding has a SOP cover of size N if and only if the function f is satisfiable.

Proof. If f is satisfiable, we can have each gadget and each wire behave "as intended," and this defines a cover with exactly N products, using Lemmas 1, 2 and 3.

In the other direction, suppose we have a cover C with exactly N products. Then by Lemmas 1, 2 and 3, *every* gadget must behave "as intended" - for if even a single gadget did not, the sum number of products of the cover would exceed N. We conclude that the output of the f is one for some setting of its input variables. This setting is determined from the truth values C defines at the input I/O nodes. The fact that C optimally covers the special odd length wire extending from the final output node implies that f evaluates to TRUE for this setting of the variables. □

Observe that if we write down the complete truth table of the function on $2\ell + 12$ variables whose onset is exactly those hypercube nodes in the above embedding, it will have length $2^{O(\log|f|)}$ which is polynomial in $|f|$. The mapping from formula f to this truth table is straightforward to compute, and thus the overall transformation can be computed in polynomial time in $|f|$. This observation together with the Th. 5 gives us:

Theorem 5.1 *FULL TRUTH TABLE MIN SOP is NP-complete.*

6 SOP Minimization Problems are Σ_2^P-complete

In this section we discuss logic minimization problems whose input is a sum-of-products form, whereas in the previous formulations the input was a truth table. The prototypical such problem is *MIN SOP-2-SOP* asking, given a SOP formula ϕ, what is the smallest SOP ϕ' equivalent to ϕ [6]. Can this problem be in *NP*? It cannot, if one believes (as complexity theorists do) that *co-NP* is different from *NP* - because *MIN SOP-2-SOP* can easily be seen to be *co-NP-hard*[7]. In fact, *MIN SOP-2-SOP* exhibits the $\exists\,\forall$ quantifier alternation of Σ_2^P - it asks if there exists a short SOP $\phi'(x)$ such that for all x, $\phi'(x) = \phi(x)$ - so we might expect it to be Σ_2^P-complete. Indeed, *MIN SOP-2-SOP* was conjectured to be complete for Σ_2^P in a seminal 1976 paper by Stockmeyer [18].

[6] *MIN SOP-2-SOP*, called also *MIN DNF* in the computational complexity literature, is a specialized version of the problem *MIN FORMULA* where the input is a generic Boolean formula. The latter problem motivated the definition of the polynomial hierarchy.

[7] A satisfiable DNF ϕ has an equivalent DNF of size 0 iff ϕ is a tautology, and one can check whether a DNF is satisfiable in polynomial time.

For over 20 years it was an open problem to prove completeness for *MIN SOP-2-SOP*, until Umans proved it in his dissertation [23, 22]. Moreover, he proved the Σ_2^P-completeness for a number of other logic minimization problems that model the most common operations performed in modern logic minimization packages (see *espresso* [2]), like maximal expansion and reduction of implicants.

Just as NP-complete problems probably require exponential time, $\Sigma_2^P = NP^{NP}$ complete problems probably require exponential time with access to an NP oracle, so Σ_2^P-completeness allows us to classify what can or cannot be done in polynomial time with access to an NP oracle. Notice that almost every heuristic, approximate, or exact method for solving logic minimization problems employs a tautology subroutine. If we assume the presence of such a subroutine, Σ_2^P-completeness plays exactly the role of NP-completeness for ordinary optimization problems: it distinguishes the intractable from the efficiently solvable.

We will review now the complexity of *MIN SOP-2-SOP* and of various logic minimization steps used in the major logic minimization suites. when the input is given as a sum-of-products.

6.1 Complexity of Term Expansion and Reduction

The *expand* step in ESPRESSO attempts to expand as much as possible the set of minterms covered by a given term, by removing literals from that term. One way to formalize the computational problem underlying this routine is as follows:

SHORTEST IMPLICANT CORE
INSTANCE: A SOP formula $\phi = \vee_{i \in I} t_i$, $j \in I$, and an integer k.
QUESTION:Is there an implicant t'_j of ϕ, with $lit(t'_j) \subseteq lit(t_j)$, of length at most k?

Theorem 6.1 *SHORTEST IMPLICANT CORE is* Σ_2^P*-complete.*

The proof is by reduction from *2-QBF* and can be found in [23, 22].

A variant of this problem asks not for a *minimum* length implicant t'_j for which $lit(t'_j) \subseteq lit(t_j)$ but rather a *minimal* length (in other words, prime) implicant t'_j for which $lit(t'_j) \subseteq lit(t_j)$. Usually this latter variant is used in practice; in complexity terms it is also "easier" since it lies in the class P^{NP}, as can be seen from the following algorithm. For each literal in term t_j, we use an NP query to determine if t_j with that literal deleted remains an implicant of ϕ. If so, delete that literal and repeat from the resulting expanded term the process of deleting literals. Otherwise (no expanded term with a deleted literal is an implicant of ϕ), the term is a prime. Overall the running time is at most quadratic in the number of literals in t_j.

In a similar manner, we can formalizes the step of *reduce* in ESPRESSO. This procedure attempts to reduce as much as possible a given term by adding literals, while still covering the given function.

LONGEST IMPLICANT EXTENSION (TERM REDUCTION)
INSTANCE: A SOP formula $\phi = \vee_{i \in I} t_i$, a $j \in I$, and an integer k.
QUESTION:Is there an implicant t'_j of ϕ, with $lit(t'_j) \supseteq lit(t_j)$, such that $\phi \equiv \vee_{i \in I \setminus \{j\}} t_i \vee t'_j$ and t'_j has length at least k?

It is noted in [23, 22] that this problem is in the class $P_{||}^{NP}$. This is true because t'_j can contain literal ℓ if and only if the conjunction $t_j \wedge \bar{\ell}$ is an implicant of $\vee_{i \in I \setminus \{j\}} t_i$. For each literal ℓ, an NP query will tell us whether the above condition holds. Thus with polynomially many parallel NP queries we can determine exactly which literals can be added to t_j to form t'_j.

6.2 Complexity of Minimum SOP

This is the key problem introduced at the beginning of the section and asking, for a given a SOP formula ϕ, what is the smallest SOP ϕ' equivalent to ϕ. The size of the SOP can be measured by number either of literals or of terms, yielding respectively *MIN LIT SOP-2-SOP* or *MIN TERM SOP-2-SOP*.

MIN LIT SOP-2-SOP
INSTANCE: A SOP formula ϕ and an integer k.
QUESTION:Is there a SOP ϕ' with at most k occurrences of literals and for which $\phi' \equiv \phi$?

Theorem 6.2 *MIN LIT SOP-2-SOP is Σ_2^P-complete.*

The proof is by reduction from *SHORTEST IMPLICANT CORE* and can be found in [23, 22]. Given an instance ($\phi = \vee_{i \in I} t_i, j, k$) of *SHORTEST IMPLICANT CORE*, the general idea of the reduction is to construct a SOP ϕ' from ϕ in which every term except one corresponding to t_j must occur in any equivalent SOP. The size of the minimum SOP equivalent to ϕ' is then determined by the size of the smallest implicant core contained in that term.

As noted in [23, 22], the proof also shows that a seemingly easier SOP minimization problem, which requires only minimality term-by-term, is also Σ_2^P-complete: given $\phi = t_1 \vee t_2 \vee \ldots \vee t_n$ and a bound k, find an equivalent SOP ϕ', with at most k occurrences of literals, of the form $\phi' = t'_1 \vee t'_2 \vee \ldots \vee t'_n$, such that for all i, $lit(t'_i) \subseteq lit(t_i)$.

The second main variant of *MIN SOP-2-SOP* asks for an equivalent SOP that is minimum with respect to terms (rather than occurrences of literals).

MIN TERM SOP-2-SOP
INSTANCE: A SOP formula ϕ and an integer k
QUESTION:Is there a SOP ϕ' with at most k terms and for which $\phi' \equiv \phi$?

Theorem 6.3 *MIN TERM SOP-2-SOP is Σ_2^P-complete.*

The proof can be found in [22](Th. 5.5, page 44), by reduction from *SHORTEST IMPLICANT CORE*.

6.3 Complexity of Irredundant Cover

The following problem formalizes the step of *irredundant* in ESPRESSO, namely, remove as many terms as possible from a given SOP:

IRREDUNDANT COVER
INSTANCE: A SOP formula $\phi = \vee_{i \in I} t_i$ and an integer k.
QUESTION:Is there a subset $I' \subseteq I$ such that $\phi \equiv \vee_{i \in I'} t_i$ and $|I'| \leq k$?

Theorem 6.4 *IRREDUNDANT COVER is Σ_2^P-complete.*

The proof can be found in [22](Th. 5.7, page 50), by reduction from *SHORTEST IMPLICANT CORE*.

6.4 Complexity of Detecting Implicants

The most most basic logic containment operation is to check if a conjunction t is an implicant of SOP ϕ:

IMPLICANT
INSTANCE: A SOP formula ϕ and a term t.
QUESTION:Is t an implicant of ϕ?

Theorem 6.5 *IMPLICANT is coNP-complete.*

Proof. This problem is in coNP because the complement problem of determining whether t is *not* an implicant of ϕ can be solved by guessing a point covered by t but not ϕ.

We show it is coNP-complete by reducing from SOP-VALIDITY. Given SOP ϕ which is an instance of SOP-VALIDITY, we produce the instance $(\phi, 1)$ of IMPLICANT. It is clear that 1 is an implicant of ϕ if and only if ϕ is a tautology. □

The next problem formalizes the step of detecting essential implicants. Here we want to determine if a term t_j of a SOP ϕ covers some point that no other term in ϕ covers.

ESSENTIAL IMPLICANT
INSTANCE: A SOP formula $\phi = \vee_{i \in I} t_i$ and $j \in I$.
QUESTION:Is $\phi \not\equiv \phi'$, where $\phi' = \vee_{i \in I \setminus \{j\}} t_i$?

Theorem 6.6 *ESSENTIAL IMPLICANT is NP-complete.*

Proof. The problem is in NP, because one can guess a point and verify that it is covered by t_j but is not covered by $\vee_{i \in I \setminus \{j\}} t_i$ (note that checking whether a given point is covered by a term is a polynomial-time operation).

Completeness is shown by reduction from CNF-SAT. Let $\gamma(x_1, ..., x_n)$ be an instance of *CNF-SAT*. Introduce a new variable z and construct the SOP formula $\phi = z \vee (z \wedge \neg\gamma)$. Set j so that t_j is the term z in this SOP.

We claim that γ is satisfiable iff z is essential (i.e., $\phi \not\equiv \phi' \equiv (z \wedge \neg\gamma)$). If γ is satisfiable, then there exists some $(a_1, ..., a_n)$ for which $\gamma(a_1, .., a_n) = 1$; this implies that $(z \wedge \neg\gamma)$ does not cover the extension of this point that sets $z = 1$, while z clearly does cover it. This means that z is essential. In the other direction, if γ is not satisfiable, then $(z \wedge \neg\gamma)$ is equivalent to z, and so z is not essential. □

Another basic operation is that of detecting prime implicants:

PRIME IMPLICANT
INSTANCE: A SOP formula ϕ and a term t.
QUESTION:Is t a prime implicant of ϕ?

Theorem 6.7 *PRIME IMPLICANT is DP-complete.*

Proof. It is in DP because t is a prime implicant iff t is an implicant (a problem in *co-NP* as noted above), *and* every shortening of t by deleting one literal is not an implicant (a problem in NP).

We show it is DP-hard by reduction from *SAT-UNSAT*. Let (ϕ, ϕ') be an instance of *SAT-UNSAT* (both ϕ and ϕ' are CNFs). We produce the following SOP (z is a fresh variable): $\phi'' \equiv (\neg z \wedge \neg\phi) \vee (z \wedge \neg\phi')$. Our instance of *PRIME IMPLICANT* is (ϕ'', z).

We claim that z is a prime implicant of ϕ'' iff ϕ is satisfiable and ϕ' is unsatisfiable. If ϕ' is unsatisfiable, then it is clear that z is an implicant of ϕ''; at the same time if ϕ is satisfiable, then z must be prime, because 1 is not an implicant of ϕ''. In the other direction, if z is an implicant of ϕ'' then ϕ' must be unsatisfiable; and if z is also *prime* then 1 cannot be an implicant of ϕ'' which implies that ϕ must be satisfiable. □

6.5 Complexity of Shortest Implicant

A relaxation of *SHORTEST IMPLICANT CORE* is to look for a shortest implicant of SOP ϕ, not necessarily obtained by expanding a term of ϕ.

SHORTEST IMPLICANT
INSTANCE: A SOP formula $\phi = \vee_{i \in I} t_i$ and an integer k.
QUESTION:Is there an implicant t'_j of ϕ of length at most k?

One would expect that this problem is Σ_2^P-complete too, since it is in Σ_2^P, like the others. However, it turns out it is likely easier than Σ_2^P-complete, by exploiting the following fact, proved by Umans in [23, 22]: any shortest implicant of ϕ may be obtained from some term t_i of ϕ with at most $\log n$ additions of literals and at most $\log n$ deletions of literals. As a consequence to decide *SHORTEST IMPLICANT* it requires only $O(\log^2)$-limited non-determinism and a single coNP query. Given an instance $(\phi = t_1 \wedge t_2 \wedge \ldots \wedge t_n, k)$, one guesses an index i, a set I of $\log n$ literals to add, a set D of $\log n$ literals to delete and construct a candidate conjunction $C = (t_i \cup I) \setminus D$. We answer "yes" if C is well-formed, has length at most k, and a query to a coNP oracle indicates that it is an implicant of ϕ.

Moreover, Umans showed in [23, 22] that this problem is in fact complete for a new complexity class including the problems solvable by $O(\log^2 n)$ limited non-determinism and a single interaction with a coNP oracle, denoted by GC($\log^2 n$, coNP) and lying between coNP and Σ_2^P. This result indicates that while the problem is unlikely to be Σ_2^p-complete, it is also unlikely to be any "easier" than the above algorithm indicates; in particular it is not likely to be in coNP.

Theorem 6.8 *SHORTEST IMPLICANT is GC($\log^2 n$, coNP)-complete.*

The formula and circuit versions of *SHORTEST IMPLICANT* instead are Σ_2^P-complete (see again [23, 22]).

6.6 Complexity of Complementation

In ESPRESSO, given a SOP cover of the offset and dcset, a SOP cover of the offset is computed, because the maximal expansion of a cube is obtained by checking that it does not intersect any cube of the offset. Therefore given a SOP ϕ, it is important to compute a small SOP representation of the complement of ϕ. This problem is captured as follows

MIN SOP COMPLEMENT
INSTANCE: A SOP formula ϕ and a integer k in unary (i.e., k is given in the input by a string of k ones).
QUESTION:Is there a SOP formula ϕ' with at most k occurrences of literals for which $\phi' \equiv \neg\phi$?

Note that if k were represented in binary, then we would not readily be able to show that the problem is even in Σ_2^P, since there are SOP formulas ϕ whose complement requires *exponential* size in ϕ. However, we can avoid this difficulty if k is represented in unary. In that case, the input has size at least k and then the problem lies in Σ_2^P, since any potential ϕ' has size only polynomial in the size of the input.

In fact Schaefer and Umans [16] show that this problem is Σ_2^P-complete:

Theorem 6.9 *MIN SOP COMPLEMENT is Σ_2^P-complete.*

Proof. In [16] they show that a problem called SHORT CNF is Σ_2^P-complete. In this problem the input is a SOP formula ϕ and an integer k represented in unary; the question is whether there is a CNF formula ϕ' with at most k occurrences of literals for which $\phi' \equiv \phi$. It is clear that this occurs if and only if there is a SOP formula ϕ'' with at most k occurrences of literals for which $\phi'' \equiv \neg\phi$, since the negation of a SOP formula ϕ'' can be represented by a CNF formula of the same size, and vice versa. $\square$

6.7 Complexity of State Assignment

A *finite state machine* (or "finite state transducer") is specified by a transition function $\delta : Q \times \{0,1\}^s \rightarrow Q \times \{0,1\}^t$, where Q is a finite set of states, and $\{0,1\}^s$ is the set of inputs, and $\{0,1\}^t$ is the set of outputs. Equivalently, we may specify the transition function with a function

$$f : Q \times \{0,1\}^s \times Q \times \{0,1\}^t \rightarrow \{0,1\},$$

with the property that $f(q,i,q',o) = 1$ iff $\delta(q,i) = (q',o)$. It is desirable to express f as a small SOP. To do this we need a *state encoding function* $e : Q \rightarrow \{0,1\}^m$. Then we may ask for a minimum SOP ϕ representing the function f_e defined by [8]

$$f_e(e(q), i, e(q'), o) = 1 \quad \Leftrightarrow \quad f(q,i,q',o) = 1.$$

The decision problem associated with this optimization is:

STATE ASSIGNMENT
INSTANCE: A SOP ϕ representation of a function f_e specifying a finite state machine with states Q, and an integer k.
QUESTION:Is there an integer m' and an encoding function $e' : Q \rightarrow \{0,1\}^{m'}$ for which $f_{e'}$ has a SOP representation ϕ' with at most k terms[9]?

It had been shown by K. Keutzer and D. Richards [10] that STATE ASSIGNMENT is coNP-hard (lower bound) and contained in Σ_2^p (upper bound). A complete account of the early results on the complexity of this problem is available in [24]. In the degenerate case in which there is only a single state STATE ASSIGNMENT becomes MIN TERM SOP-2-SOP. We thus obtain the new result:

Theorem 6.10 *STATE ASSIGNMENT is Σ_2^P-complete.*

The practitioner may protest that lumping together two-level minimization and state assignment in the same complexity classes is highly unsatisfactory since the latter problem is much harder than the former in practice. This is but one example where the coarse classification afforded by the classes of the Polynomial Hierarchy would benefit from some refinement that captures the observed difference in the difficulty of problems within the same class. The classification of NP-complete problems according to their approximability is an example of such a refinement; however here it seems that approximability would not separate, e.g. MIN SOP-2-SOP from STATE ASSIGNMENT, as MIN SOP-2-SOP is already extremely hard to approximate [22, 21].

[8]This is the relational representation of a finite state machine; alternatively one may use a functional representation by means of a multiple-output function whose outputs correspond to all the encoded next state and external output variables.

[9]We can also consider the variant in which ϕ' must have at most k literals.

7 Conclusions

Two-level logic minimization is the quintessential problem in digital design automation. In this paper we have presented a consistent account of the computational complexity of the decision problems capturing two-level logic minimization. We hope that this provides a coherent picture of results scattered in many sources and fills a few gaps in the older literature.

The major points to take away are:

- When the function is *explicitly* described by its onset and offset (or even just its onset), the problem of finding a minimum SOP representation is NP-complete. This problem is essentially a covering problem, and NP-completeness follows by either the Gimpel reduction in Sec. 4, or the more involved Masek reduction in Sec. 5.
- When the function is *implicitly* described by a SOP form, the complexity of finding a minimum SOP representation is Σ_2^p-complete. This means that (subject to accepted assumptions from computational complexity) no polynomial-time algorithm can solve this problem *even when charging only a single time step for every call to a TAUTOLOGY subroutine.*
- When the function is *implicitly* described by a SOP form, many of the sub-problems used in the major logic minimization suites (i.e., ESPRESSO) turn out to lie between NP and Σ_2^p, and several are Σ_2^p-complete.

In closing we mention two very natural problems whose complexity remains open:

MIN CIRCUIT-2-CIRCUIT
INSTANCE: A Boolean circuit C and an integer k.
QUESTION:Is there a Boolean circuit C' of size at most k that computes the same function as C?

This problem is in Σ_2^p and conjectured to be Σ_2^p-complete. In fact even when C and C' are required to be circuits of depth 3 (i.e. sum-of-products-of-sums, or product-of-sums-of-products), no proof of Σ_2^p-completeness is known.

The "explicit" version of this problem is:

FULL TRUTH TABLE MIN CIRCUIT
INSTANCE: Onset and offset of a completely specified function f and an integer k.
QUESTION:Is there a Boolean circuit C of size at most k that computes f?

This problem is in NP and it is not known whether it is NP-complete or not. This problem has some interesting connections to other questions in computational complexity, that have been outlined in [8].

References

[1] D. Bovet and P. Crescenzi. *Introduction to the theory of complexity.* Prentice Hall, 1994.

[2] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984.

[3] R. Brayton, A. Sangiovanni-Vincentelli, and G. Hachtel. Multi-level logic synthesis. *Proceedings of the IEEE*, vol. 78(no. 2):264–300, February 1990.

[4] S.L.A. Czort. The complexity of minimizing disjunctive normal form formulas. Master's thesis, University of Aarhus, Denmark, 1999.

[5] M. Gao, J.-H. Jiang, Y. Jiang, Y. Li, A. Mishchenko, S. Sinha, T. Villa, and R. Brayton. Optimization of multi-valued multi-level networks. In *The Proceedings of the International Symposium on Multiple-Valued Logic*, pages 168–177, May 2002.

[6] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* W. H. Freeman and Company, 1979.

[7] J. Gimpel. A method of producing a boolean function having an arbitrarily prescribed prime implicant table. *IRE Transactions on Electronic Computers*, EC-14:485–488, June 1965.

[8] V. Kabanets and J-Y. Cai. Circuit minimization problem. In *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing (STOC)*, pages 73–79, 2000.

[9] R. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[10] K. Keutzer and D. Richards. Computational complexity of logic optimization. *Unpublished manuscript*, March 1994.

[11] W.J. Masek. Some NP-complete set covering problems. Unpublished manuscript, May 1978.

[12] E. McCluskey. Minimization of Boolean functions. *Bell Laboratories Technical Journal*, 35:1417–1444, November 1956.

[13] F. Mo and R. Brayton. Regular fabrics in deep sub-micron integrated-circuit design. In *International Workshop on Logic and Synthesis*, pages 7–12, May 2002.

[14] C. Papadimitriou. *Computational complexity.* Addison Wesley, 1994.

[15] Wolfgang Paul. Boolean minimum polynomials and covering problems. *Acta Informatica*, 4:321–336, 1975. (in German). Original Title: Boolesche Minimalpolynome and Ueberdeckungsprobleme.

[16] M. Schaefer and C. Umans. Completeness in the polynomial-time hierarchy: A compendium. *SIGACT News*, September 2002.

[17] M. Schaefer and C. Umans. Completeness in the polynomial-time hierarchy: Part II. *SIGACT News*, December 2002.

[18] L.J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1976.

[19] T. Strzemecki. Polynomial-time algorithms for the generation of prime implicants. *Journal of complexity*, 8:37–63, 1992.

[20] M. Theobald and S. Nowick. Fast Heuristic and Exact Algorithms for Two-Level Hazard-Free Logic Minimization. *IEEE Transactions on Computer-Aided Design*, 17(11):1130–1147, November 1998.

[21] C. Umans. Hardness of approximating Σ_2^p minimization problems. In *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 465–474, 1999.

[22] C. Umans. *Approximability and completeness in the polynomial hierarchy.* PhD thesis, U.C. Berkeley, 2000.

[23] C. Umans. The minimum equivalent DNF problem and shortest implicants. *Journal of Computer and System Sciences*, 63(4):597–611, 2001.

[24] T. Villa, T. Kam, R. Brayton, and A. Sangiovanni-Vincentelli. *Synthesis of FSMs: logic optimization.* Kluwer Academic Publishers, 1997.