The Combinational-Complexity Game For Symmetric Functions

Andrea Costamagna
EPFL
Lausanne, Switzerland

Alan Mishchenko
UCBerkley
Berkley, California

Giovanni De Micheli
EPFL
Lausanne, Switzerland

***Abstract*—The *combinational complexity* of a Boolean function is the minimum number of binary logic operators needed to represent it. Finding such representations is crucial to achieving faster and less resource-intensive computing. Since this problem is intractable, state-of-the-art techniques rely on solving it using Boolean satisfiability (SAT). However, finding the optimum solution for functions with combinational complexity higher than 15 is usually not feasible. We treat synthesis as a time series of steps, each adding a few Boolean operators to the representation. From interpreting this procedure as a game, we devise a solver acting as a player that chooses the next move based on estimating the expected reward. A greedy version of the method yields close-to-optimum results for symmetric functions. For instance, the solver finds an And-Inverter Graph (AIG) of 72 gates for the majority-of-15 in less than 10 seconds. After thorough hyperparameter tuning, state-of-the-art synthesis and optimization require minutes to obtain a 71-gate AIG.**

***Index Terms*—Combinational complexity, symmetric functions, set covering**

I. Introduction

The increasing cost of silicon and the need for faster computation call for systems with higher performance. Boolean functions are the cornerstone of digital computation. Hence compactly representing them is crucial for reducing resource usage and computation time. However, despite decades of extensive research, we still do not know the optimal representations for many 6-input Boolean functions.

The problem of interest is to find a Boolean function representation in terms of the minimum number of binary logic operators. This number is called the *combinational complexity* of the function, and the problem of finding it is called *exact synthesis*. State-of-the-art techniques rely on encoding exact synthesis as an instance of the *satisfiability* problem (SAT) [1], [2], for which efficient solvers are available [3]. However, when the combinational complexity is higher than 15, SAT-based approaches become infeasible and finding close-to-optimum representations remains an open problem.

Inspired by the advances in computer programs capable of beating humans in complex games [4], we investigate ways of representing the problem as a game played by the solver, the *combinational complexity game* (CCG). Rather than using a monolithic problem encoding and a SAT solver, this approach treats synthesis as a time series of steps. The goal of the solver is to transform a truth-table into a structural representation. At any step of CCG, the solver analyses the partial solution and decides what action to take. We target synthesis of single output Boolean functions that can be manipulated using truth tables, and for which SAT-based methods do not work, i.e., single output functions with more than 6 inputs.

Motivated by recent research efforts aiming to reduce the resource usage of machine learning [5], we focus on *threshold* functions. Indeed, Binarized Neural Networks (BNNs) are hardware-efficient machine learning models in which a neuron is a Boolean threshold function. [6]. BNNs have hundreds of thousands of neurons. Hence, efficient and compact representations for the individual nodes determine the system's performance. Since threshold functions are symmetric in all their variables, this first motivation leads to a more general question: *Is it possible to define a solver capable of obtaining close-to-optimum results in* CCG *when functions present symmetries?*

Our work takes inspiration from a classical synthesis technique based on detecting symmetries through spectral methods. In the seminal work, Edwards et al. proposed to combine symmetry detection and operator insertion in a Boolean representation through a synthesis procedure called *remapping* [7]. They implemented the method as an interactive program and mentioned the need to develop automatic criteria to identify the most suitable synthesis option at each stage. To the best of the authors' knowledge, formulating symmetry-based synthesis for practical automation remains an open problem.

Our formulation of the problem relies entirely on truth-table manipulation. This results in a more lightweight symmetry-detection strategy compared to spectral methods [8]. We analytically derive remapping equations induced by a synthesis step. These equations allow us to estimate the advantage of a synthesis step before performing it. In line with the idea in the seminal paper and follow-up work [7], [9], we propose a greedy selection strategy based on maximizing *don't-cares*.

Symmetry-based synthesis fails when the function to synthesize does not have two-variable symmetries. Furthermore, the set of actions identified by symmetry analysis is not necessarily exhaustive, resulting in sub-optimal results. To extend the applicability of the method to generic functions and improve the quality of results, we devise a novel technique relying on set covering [10]. Since the proposed solvers consider a set of Boolean variables named cut, we call them CUSCO, which stays for *cut-by-cut set covering-based synthesis*. We formalize both strategies as instances of the CCG paradigm, whose generality goes beyond the details of this paper.

The rest of this paper is organized as follows. Section II covers the background. Section III formulates CCG and describes the devised solvers. Section IV shows experimental results. Section V concludes the paper.

II. Background

A. Boolean Basics

Let $f : \mathbb{B}^n \mapsto \mathbb{B}$ be a single-output Boolean function. We describe the function in terms of the partition that its value induces on the Boolean cube $\mathbb{B}^n$ [11]. Its *support* $\mathcal{S} = (x_i)_{i=1}^n$ is the ordered set of variables on which f depends.

A *literal* is a Boolean variable or its negation, so its positive value identifies a subset of $\mathbb{B}^n$. A *cube* is a literal or an intersection of literals. We represent the operations $\{\cap, \cup, \subset, \subseteq, \triangle\}$ between cubes in logic notation $\{\cdot, +, <, \leq, \oplus\}$. Also, we indicate cube complementation as C'.

We equivalently refer to a cube as intersections of literals, as the binary labelling of their negation, or as the decimal value of the binary labelling. For example, the two-literal cubes in terms of variables (x_i, x_j) are equivalently $\mathbb{B}^2$, $\{x_j x_i, x_j x_i', x_j x_i', x_j' x_i'\}, \{11, 10, 01, 00\}$, or $\{3, 2, 1, 0\}$.

Two cubes C and D are *independent* if they have an empty intersection ($C \cdot D = 0$). A cube C is *contained* in a cube D if $C \leq D$. For instance, $x_i x_j'$ is contained in x_i because $x_i x_j' \leq x_i$ is satisfied $\forall (x_i, x_j) \in \mathbb{B}^2$.

The *minterms* are the 2^n n-literal cubes contained in $\mathbb{B}^n$. The *cofactor* of f with respect to a cube C is the function f_C of the variables not appearing in C and whose minterms are the subsets of minterms of f contained in C. When C is a minterm, f_C is the value of the function. The minterms can be partitioned based on the value of f_C. If $f_C = 0$, C is in the *offset* of f, otherwise C is in the *onset* of f.

An *incompletely specified* function f is a function whose output is not defined (or is a *don't-care*) for some input minterms. This condition can happen when f is a sub-function of an *environment*, e.g., a larger block of logic. *Controllability don't-cares* (CDCs) are patterns never appearing at the function inputs. We represent the CDCs with a function named CDC-*mask* $\mu : \mathbb{B}^n \to \mathbb{B}$. Given a minterm M, $\mu_M \in \mathbb{B}$ indicates if M can appear at the input. There are $2^{|\text{CDC}|}$ pairs (τ, μ) representing f. Indeed τ_M is not important if $\mu_M = 0$.

A *Boolean chain* for n variables $(x_1, ..., x_n)$ is a sequence $(x_{n+1}, ..., x_{n+r})$ where each step combines two previous steps $x_i = x_{j(i)} \circ_i x_{k(i)} \quad n + 1 \leq i \leq n + r$ where $1 \leq j(i) < i$ and $\circ_i$ is a binary Boolean operator [12]. The *cut* at position k in the Boolean chain is the set of variables $C = \{x_i\}_{i \leq k}$, named *divisors*, such that there is a variable $x_{j>k}$ having x_i in its support.

B. Boolean Function Representations and Optimization

The structure of the Boolean chain allows us to represent it as a directed acyclic graph (DAG). We consider two types of graph, differing in the set of Boolean operators that can be represented by a node and its edges. In an *and-inverter graph* (AIG), $\circ_i$ is either in $\{\cdot, +, <, \leq\}$ or in the set of the complements of these operations. An *xor-and-inverter graph* (XAIG) extends the node functionalities of an AIG with the exclusive-or $\oplus$ and its complement. A Boolean chain of length r *synthesizes* $f : \mathbb{B}^n \to \mathbb{B}$ if $x_{n+r} = f(x_1, \dots, x_n)$.

A *functionally reduced AIG* (FRAIG) is an AIG in which every node has a unique functionality. We rely on efficient implementations of FRAIGs, leveraging functional simulation and SAT-solving [13]. When the number of inputs is smaller than 17, the FRAIG implementation in ABC [14] synthesizes a FRAIG from a truth table within a few milliseconds.

Unless optimal, the size of a FRAIG is further optimizable. We use an efficient combination of traditional logic synthesis optimization strategies named `deepsyn` (command `&deepsyn` in ABC). `Deepsyn` is the most aggressive optimization strategy available in open-source EDA. Hence, `deepsyn` can yield the most compact AIG representation achievable with state-of-the-art synthesis and optimization.

C. Constrained Optimization of Boolean Representations

Finding the minimum length chain synthesizing f is a fundamental problem [11]. We name *optimum* representation the Boolean chain of shortest length, and the problem of finding it is *exact synthesis*. The length of the optimum chain is the *combinational complexity* of the function it synthesizes $\mathcal{C}(f)$. When finding the optimum is not possible, we target close-to-optimum solutions.

Theoretical results show that the combinational complexity of an n-input function generally exceeds $2^n/n$ [12]. Since the number of chains we can define with $n_\circ$ binary operations is of the order of $[n_\circ \cdot (n + \mathcal{C}(f))^2]^{\mathcal{C}(f)}$, the dimensionality of the *search space* makes the optimization problem intractable.

Since an exhaustive search for a solution is impossible, any solver must contain efficient mechanisms for ruling out sub-optimal solutions. In practice, this corresponds to a dynamic reduction of the explored portion of the search space. Exact synthesis solvers are engines that guarantee the optimality of the chain in case of success. State-of-the-art approaches rely on encoding the problem as an instance of *Boolean satisfiability* (SAT). However, despite the efficiency of modern SAT solvers, the resulting problems are too hard even for the general case of 6-input functions.

D. Two-Variable Symmetries

Classical symmetries correspond to functional equivalences in subspaces defined by two-literal cubes. Cofactor comparisons give information on these functional properties.

The fundamental two-variables symmetries are *non-equivalence symmetry* (NES), *equivalence symmetry* (ES), and *multiform symmetry* (MS). Fig. 1(a-c), show these symmetries illustrated on the Karnough map (KM). Instead, the explicit functional form is:

1) NES$\{x_i, x_j\}$: $f_{01} = f_{10}$.
2) ES$\{x_i, x_j\}$: $f_{00} = f_{11}$.
3) MS$\{x_i, x_j\}$: $f_{01} = f_{10} \wedge f_{00} = f_{11}$.

The red numbers offer an example in which f is a two-input function. In the general case, the equivalence identified by the lines is among cofactors. For instance, Fig. 1(b) means that $f_{00} = f_{11}$, where $f_{00}, f_{11} : \mathbb{B}^{n-2} \to \mathbb{B}$. In the presence of *don't-cares*, they can be allocated to complete a cofactor and satisfy a symmetry. Fig. 1(b) offers an example: assigning the *don't-care* $*$ to 0 results in ES.

Single-variable symmetries (SVS) correspond to cofactor equalities in the space identified by a single variable:

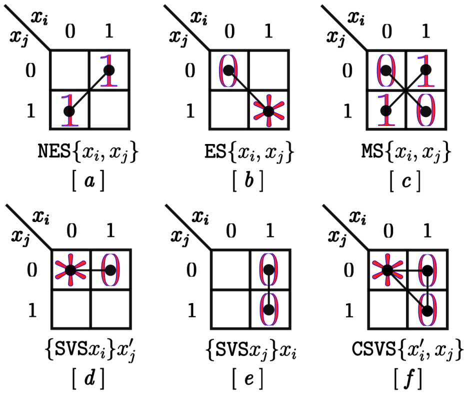
1) $\{\text{SVS}x_i\}x_j$: $f_{10} = f_{11}$.
2) $\{\text{SVS}x_i\}x_j'$: $f_{00} = f_{01}$.
3) $\{\text{SVS}x_j\}x_i$: $f_{11} = f_{01}$.

4) $\{\texttt{SVS}x_j\}x_i' : f_{10} = f_{00}$.

Fig. 1 (d-e) demonstrates two of these symmetries.

Fig. 1 (f) shows that more SVSs can be present at the same time. They are called *compatible single-variable symmetries* (CSVS). The compatibility check in the presence of *don't-cares* can result in an incorrect result, induced by conflicting assignments of *don't-cares*. To avoid this, it is sufficient to combine the SVS-check with an ES-check. The compatible symmetries and their equivalence checks are:

1) $\texttt{CSVS}\{x_j', x_i'\} : f_{00} = f_{01} = f_{10}$.
2) $\texttt{CSVS}\{x_j', x_i\} : f_{00} = f_{01} = f_{11}$.
3) $\texttt{CSVS}\{x_j, x_i'\} : f_{00} = f_{10} = f_{11}$.
4) $\texttt{CSVS}\{x_j, x_i\} : f_{01} = f_{10} = f_{11}$.



$\texttt{NES}\{x_i, x_j\}$ [*a*]
$\texttt{ES}\{x_i, x_j\}$ [*b*]
$\texttt{MS}\{x_i, x_j\}$ [*c*]
$\{\texttt{SVS}x_i\}x_j'$ [*d*]
$\{\texttt{SVS}x_j\}x_i$ [*e*]
$\texttt{CSVS}\{x_i', x_j\}$ [*f*]

Fig. 1. Schematic representations of the symmetry classes.

E. Synthesis Method Using Symmetries

Edwards et al. [7] devised a synthesis strategy based on *remapping*, i.e., the exploitation of two-variable symmetries to progressively map the problem to a simpler one. The simplicity of the problem corresponds to the fact that each synthesis stage increases the number of CDCs. Hence, at each synthesis stage, the number of exploitable symmetries is either the same or higher. Sec. II-D shows that two-variable symmetries correspond to the equivalence of a function in the subspace identified by two two-literal cubes. Given two variables x_i and x_j, let $\Pi = (C_k)_{k=0}^{3}$ be any permutation of the two-literal cubes in $\mathbb{B}^2$. If we define the *simple symmetries* (SS) as the group of symmetries including NES, ES, and SVS, we observe

1) SS $\Leftrightarrow \exists\, p \neq q : f_{C_p} = f_{C_q}$.
2) CSVS $\Leftrightarrow \exists\, p \neq q \neq r : f_{C_p} = f_{C_q} = f_{C_r}$.
3) MS $\Leftrightarrow \exists\, p \neq q \neq r \neq s : f_{C_p} = f_{C_q}$ and $f_{C_r} = f_{C_s}$.

All the subscripts in the previous equalities are distinct. Without loss of generality, we group the cubes into subsets

1) SS : $\mathcal{S} = (C_p)$ $\mathcal{D} = (C_q)$ $\mathcal{N} = (C_r, C_s)$.
2) CSVS : $\mathcal{S} = (C_p, C_q)$ $\mathcal{D} = (C_r, C_r)$ $\mathcal{N} = (C_s)$.
3) MS : $\mathcal{S} = (C_p, C_q)$ $\mathcal{D} = (C_r, C_s)$ $\mathcal{N} = \varnothing$.

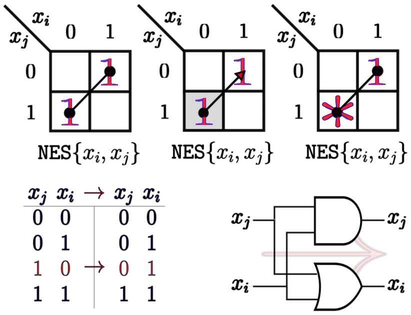
Remapping is based on the definition of the following map

$$\varphi : \mathbb{B}^2 \to \mathbb{B}^2 \quad \begin{cases} \varphi(C_k) = D_k & C_k \in \mathcal{S}, D_k \in \mathcal{D} \\ \varphi(C_k) = C_k & C_k \in \mathcal{D} \cup \mathcal{N} \end{cases} \tag{1}$$

The map identifies the truth tables of two Boolean operators. Explicitly, given two symmetric variables x_i, x_j, the map reads

$$(x_j, x_i) \to (\varphi_j(x_j, x_i), \varphi_i(x_j, x_i)) \begin{cases} 00 \to \varphi(00) \\ 01 \to \varphi(01) \\ 10 \to \varphi(10) \\ 11 \to \varphi(11) \end{cases} \tag{2}$$

Substituting x_j and x_i with the newly defined variables, results in an equivalent synthesis problem named *remapped problem*. The *remapped problem* is simpler in the sense that the minterms contained in the cubes in $\mathcal{S}$ will not appear in the remapped problem, enlarging the CDC set at the next synthesis stage. As soon as a one-literal cube C is not observable, it can be removed from the inputs list, so that the progressive increase of the CDC set eventually results in obtaining a unique intermediate variable, representing the function. Fig. 2 shows the detail of a remapping step in presence of NES. More details for the other symmetries can be found in the seminal paper [7].


x_i
x_j
0
1
1
1
$\texttt{NES}\{x_i, x_j\}$
$\texttt{NES}\{x_i, x_j\}$
$\texttt{NES}\{x_i, x_j\}$
x_j
x_i
x_j
x_i

x_j	x_i	→	x_j	x_i
0	0		0	0
0	1		0	1
1	0	→	0	1
1	1		1	1

Fig. 2. Remapping induced by $\texttt{NES}\{x_i, x_j\}$.

III. The Combinational Complexity Game

In this section, we formulate synthesis as the *combinational complexity game* (CCG). CCG is a synthesis paradigm in which the solver chooses among a set of moves based on the expected reward. We propose ad-hoc definitions of the reward function, modeling the intuition that adding a correct operation in the Boolean chain should map the problem to a simpler one.

A. Synthesis Based on Controlability Don't-Cares

We now formulate symmetry-based synthesis in terms of truth-table manipulations. This formulation allows us to efficiently automatize the synthesis procedure using CDC-maximization as the synthesis guiding principle.

1) Symmetry Detection: Differently from the seminal paper, we check the presence of symmetries directly using truth tables. Sec. II presented the two-variables symmetries in terms of the cofactors of the function. Let f be an incompletely specified function with CDC-mask μ and synthesized as τ. If A and B are two-literal cubes defining a symmetry via the equivalence $f_A = f_B$, then

$$f_A = f_B \Leftrightarrow \mu_A \mu_B \tau_A = \mu_A \mu_B \tau_B \tag{3}$$

Hence checking the presence of a symmetry for truth tables corresponds to the bitwise verification of its definition.

2) Remapping Equations: For the sake of readability, we rename the cubes of the partition of $\mathbb{B}^2$ introduced in Sec. II-E: $\Pi = \{A, B, C, D\}$. We consider the general remappings:

1) `SS` $: A \mapsto C$.
2) `CSVS` $: A \mapsto C$ and $B \mapsto C$.
3) `MS` $: A \mapsto C$ and $B \mapsto D$.

The *remapping equations* are the transformations of f from step t to step $t+1$, i.e., the assignment $\tau^{t+1}, \mu^{t+1} \leftarrow \tau^t, \mu^t$.

The remapping equations for the mask are:

$$\mu \xleftarrow{\texttt{SS}} A'\mu + C\mu_A$$
$$\mu \xleftarrow{\texttt{CSVS}} A'B'\mu + C(\mu_A + \mu_B)$$
$$\mu \xleftarrow{\texttt{MS}} A'B'\mu + (C\mu_A + D\mu_B)$$

The first term in the disjunctions removes the minterms contained in the source cubes from the *care set*. The second term removes a minterm from the `CDC` set when a minterm in the *care set* is mapped to it. Indeed, cofactoring the mask to a source cube A identifies all cubes K in the remaining variables that, when put in conjunction with A are in the *care set* ($KA \in \texttt{CDC}'_f$). Hence, after the conjunction with the target cube C, the new minterms generated by the remapping process are in the *care set* ($KC \in \texttt{CDC}'_f$).

The remapping equations for τ are

$$\tau \xleftarrow{\texttt{SS}} B'\tau + B(\mu_B\tau + \mu_A\tau_A)$$
$$\tau \xleftarrow{\texttt{CSVS}} (C' + \mu_C)\tau + C \cdot (\mu_A\tau_A + \mu_B\tau_B)$$
$$\tau \xleftarrow{\texttt{MS}} (C'D' + C\mu_C + D\mu_D)\tau + (C\mu'_C\mu_A\tau_A + D\mu'_D\mu_B\tau_B)$$

Also in this case the truth table computation has two terms. The first contribution preserves the value of the function for all the minterms that were observable at time t. Meanwhile, the second contribution reallocates the value of τ when there is a `CDC` in the source cubes of the remapping. Indeed, `CDC`s can become visible at later stages, and ignoring the second contribution would break the equivalence of the remapped problem with the original one.

3) `CDC`-Based Automation: Edwards et al. [7] provide two guiding principles for using their interactive tool:

1) Do not perform remapping if the destination set $\mathcal{D}$ contains only *don't-cares*.
2) Do not remap *care* minterms into the *don't-care* area. Avoid reallocating definite values to present *don't-cares*.

The remapping equations allow us to include the first principle in an automatic method. Indeed, if $\{a_{t,i}\}_{i=1}^m$ is the set of available synthesis actions at time t, we compute the remapped mask $\mu[a_{t,i}] \xleftarrow{a_{t,i}} \mu$. Next, by defining the *reward* function as the number of `CDC`s, we can select the action as

$$a^* = \arg\max_a |\texttt{CDC}[a]| = \arg\max_a |\mu[a]|_0 \tag{4}$$

The move maximizing the number of `CDC`s maps the problem to a simpler one, either reducing the number of variables or increasing the number of exploitable symmetries.

when it comes to the second principle, the remapping equations for τ perform reallocation when it is advantageous. In the experimental section we show that `CDC`-maximization is a powerful guiding principle for symmetric functions.

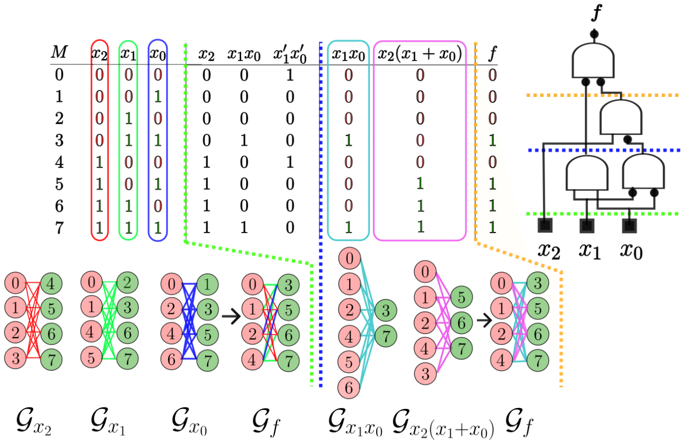
B. Cut-by-cut Synthesis Using Set Covering

Symmetry-based synthesis fails for functions without symmetries. Furthermore, the symmetry-based remappings are a subset of the legal actions. Hence, the portion of the search space containing the optimum might be unreachable. We propose a new approach generalizing symmetry-based synthesis to arbitrary functions and increasing its exploratory potential.

1) Set Covering-Based Verification of the Validity of a Cut: We introduce a graph representation for Boolean functions. The *information graph* (IG) of the function $f : \mathbb{B}^n \to \mathbb{B}$ is the undirected bipartite graph $\mathcal{G}_f = (\mathcal{V}, \mathcal{E})$, whose parts are the onset and the offset of f. The edges identify minterms with different function value, i.e., the adjacency matrix is

$$A^f_{MK} = f_M \oplus f_K \quad \forall M, K \in \mathcal{V} \tag{5}$$

Existing works use similar functional representations [15]. Fig. 3 shows the node functions in a Boolean chain for the majority-of-3, together with the IG representation of the target function $\mathcal{G}_f$ and of the node functionalities at two cuts. Every IG explicitly stores the information on the separation induced by its defining function on $\mathbb{B}^n$. The existence of a function f of x_2, x_1 and x_0 implies that the combination of these variables contains enough information to describe f. In terms of the IGs, the subsets of the Boolean space identified by $\mathcal{G}_{x_2}$, $\mathcal{G}_{x_1}$ and $\mathcal{G}_{x_0}$ cover $\mathcal{G}_f$. This must be true at every cut of the circuit.



M	x_2	x_1	x_0	x_2	x_1x_0	$x'_1x'_0$	x_1x_0	$x_2(x_1+x_0)$	f
0	0	0	0	0	0	1	0	0	0
1	0	0	1	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0
3	0	1	1	0	1	0	1	0	1
4	1	0	0	1	0	1	0	0	0
5	1	0	1	1	0	0	0	1	1
6	1	1	0	1	0	0	0	1	1
7	1	1	1	1	1	0	1	1	1

Fig. 3. Example to discuss the set covering formulation.

Proposition III.1 (Covering). *Let $f : \mathbb{B}^n \to \mathbb{B}$ and $h : \mathbb{B}^n \to \mathbb{B}^m$. Let $\mathcal{E}$ be the edges of $\mathcal{G}_f$, and $\mathcal{E}_h = \bigcup_{i=1}^m \mathcal{E}_i$ be the edges in the information graphs of the components of h ($\mathcal{G}_{h_i}$). Then, if $\mathcal{E}_h$ covers $\mathcal{E}$ there is a function $g : \mathbb{B}^m \to \mathbb{B}$ such that*

$$f(x) = g(h(x))$$

Proof. We define $g_M \doteq g(h_M) = g(h_{1M}, h_{2M}, \ldots, h_{mM})$. Then, for every minterm pairs $M, K \in (\texttt{CDC}_f)^c$ we require

$$f_M \neq f_K \Rightarrow g_M \neq g_K \tag{6}$$
$$f_M = f_K \Rightarrow g_M = g_K \tag{7}$$

The equality is possible for any map h, as g can be arbitrarily complex. Instead, the inequality is possible only if $h_M \neq h_K$, i.e., $\exists i \in \{1,\dots,m\}$ s.t. $h_{iK} \neq h_{iM}$. At the level of the adjacency matrices, these statements read

$$\forall M,K \in \mathbb{B}^n \quad A^f_{MK} \leq \sum_{i=1}^{m} A^{h_i}_{MK} \Rightarrow A^f \leq \sum_{i=1}^{m} A^{h_i} \tag{8}$$

If for every edge $\{K,M\} \in \mathcal{E}$, $\{K,M\}$ is covered by a set in $\mathcal{E}_h$, the functions in h contain enough information to perform the partition of $\mathbb{B}^n$ identified by the target function. □

2) Cut-By-Cut Set Covering-Based Synthesis: The observation that the variables in a cut contain enough information to solve the synthesis problem allows us to generalize symmetry-based synthesis. The idea is to address the covering problem instead of performing the symmetry analysis. Given a cut C, we define the candidate *divisors* by combining the variables in C using the Boolean operators $\{\cdot,+,<,\leq,\oplus\}$ [16] and the divisors in C. This set of operators is exhaustive since we can introduce complementations at the next synthesis stage. If the cut has n variables, this results in $n+5\binom{n}{2}$ candidate divisors. Next, we define the IG for each candidate divisor and the target function, and we find several solutions to the covering problem for $\mathcal{G}_f$ using the sets defined by the IGs of the divisors. Given a set of solutions to the covering problem, we randomly select a solution that was not previously considered.

Symmetry-based synthesis also synthesizes Boolean chains one cut at a time. For instance, the synthesis process in Fig. 3 can be equivalently obtained with symmetry-based synthesis, where the first cut corresponds to a NES remapping (see Fig. 2). However, the cuts allowed by the symmetry-based remappings is a subset of the ones obtainable with set covering. The enlarged set of actions allows the solver to explore regions of the search space that are unreachable by symmetry-based analysis, enabling the synthesis of generic functions as well as better results for symmetric functions. Furthermore, the set covering solutions target the minimization of the cut size. This criteria acts as a proxy for the *don't care* maximization principle in symmetry-based synthesis.

We name our solver CUSCO, which stays for *cut-by-cut set covering-based synthesis*. In the next section, we characterise the features of our approach for the symmetric case SYM-CUSCO and the general case COV-CUSCO. The two methods differ in the data structures they manipulate. SYM-CUSCO leverages the remapping techniques to work uniquely with truth tables having the same dimensionality as the original function, i.e., 2^n for an n-inputs Boolean function. Instead, the dimensionality of the IGs is $\mathcal{O}(2^{2n})$, so it comes at the cost of runtime and memory. The experimental section shows that COV-CUSCO obtains the optimum XAIGs for almost all the symmetric functions of 4 and 5 inputs.

C. CUSCO-solvers for CCG

Algorithm 1 describes the structure of a generic CUSCO-solver. The method takes a function $f : \mathbb{B}^n \rightarrow \mathbb{B}$ as the input and returns a network representation of type NTK. In this work we consider and-inverter graphs (AIGs) [13], [17] and xor-and-inverter graph (XAIGs) [18]. CUSCO is also specified by a method-type (TYPE). We give two examples of TYPE to showcase the synthesis paradigm:

1) SYM-CUSCO uses as the core synthesis engine our automation of symmetry-based synthesis (Sec. III-A).
2) COV-CUSCO extends SYM-CUSCO using set covering.

Algorithm 1 is an iterative procedure, in which we store the best result after applying synthesis a number of times. First, we initialize the state $\texttt{state}_t$, which corresponds to an encoding of the partial solution. Next, we synthesize the network a few-gates at a time. At each step, the analyzer<TYPE>$(\cdot)$ function identifies a set of candidate moves from the current state of the solution. After selecting an action using an iteration-dependent policy<TYPE>$(\cdot,\texttt{it})$, move<TYPE>$(\cdot)$ updates the partial solution. When $\texttt{state}_t$ corresponds to a circuit satisfying the specifications defined by f, we synthesize the desired network representation. The method returns the smallest size representation identified in the synthesis process.

Algorithm 1 NTK ntk ← CUSCO<TYPE,NTK>(f)

```
|ntk_best| ← ∞
while ( it<number of iterations ) do
    $t \leftarrow 0$
    $\texttt{state}_t \leftarrow$ initialize<TYPE>($f$)
    while ( $f$ not satisfied ) do
        actions ← analyzer<TYPE>($\texttt{state}_t$)
        $\texttt{action}_t \leftarrow$ policy<TYPE>(actions,it)
        $\texttt{state}_{t+1} \leftarrow$ move<TYPE>($\texttt{state}_t$, $\texttt{action}_t$)
        ++$t$
    ntk ← synthesize<NTK>($\texttt{state}_t$)
    if |ntk| < |$\texttt{ntk}_{\texttt{best}}$| then
        $\texttt{ntk}_{\texttt{best}} \leftarrow$ ntk
    ++$it$;
return $\texttt{ntk}_{\texttt{best}}$
```

D. Characterization of the Solvers

This section characterizes Algorithm 1 for SYM-CUSCO and COV-CUSCO using concepts from Sec. III-A and III-B.

The *Boolean chain* for the CUSCO solvers is a collection of *cuts* defined at each synthesis stage. The $\texttt{state}_t$ object in Algorithm 1 is a representation related to the partial *Boolean chain*, not yet synthesizing the function. In light of the discussion on the set covering generalization, we characterize the state $\texttt{state}_t$ with the following attributes:

1) The cut of the last synthesis stage.
2) The target function f.
3) The functions of the cut variables.
4) The CDCs of the cut (only for SYM-CUSCO).

We win the game when a node functionality is equal to f.

The analyzer<TYPE> function for SYM-CUSCO follows directly from the truth table-based formulation of symmetry-based synthesis. For COV-CUSCO, the analyzer returns the n cut functionalities and the $5\binom{n}{2}$ functionalities obtainable by combining them with the Boolean operators $\{\cdot,+,<,\leq,\oplus\}$.

The policy<TYPE>$(\cdot)$ is the core of the synthesis procedure. In SYM-CUSCO, the remapping equations for the mask allow us to choose the action maximizing the number of CDCs at the next cut. In COV-CUSCO, a solution to the set covering problem targeting low sets count acts as a proxy for *don't cares* maximization. The policies are:

1) SYM−CUSCO Randomly selects among the actions maximizing *don't-cares* after remapping
2) COV−CUSCO Randomly selects among the actions covering the information graph.

In SYM−CUSCO, we further characterize the heuristic criterion. First, when multiple logic blocks yield the same *don't care* gain, we prioritize lower numbers of Boolean operators. The difference arises when synthesizing AIGs over XAIGs since the number of Boolean operators needed to represent an XOR in an AIG is 3. Furthermore, when running a single-shot version of SYM−CUSCO (SYM−CUSCO$_{\times 1}$), we prioritize symmetries acting on the same groups of variables. Intuitively, we do so to eliminate variables from the representation as soon as possible. When considering multiple iterations, say N, we relax this filter to explore more synthesis solutions (SYM−CUSCO$_{\times N}$). Similarly, COV−CUSCO$_{\times N}$ returns the best result found in N random games.

IV. Experiments

This section investigates the performances of the CUSCO solvers on symmetric functions. Section IV-A compares SYM−CUSCO$_{\times 1}$ with a state-of-the-art technique. Section IV-C analyses the validity of the assumptions defining SYM−CUSCO. Section IV-D compares SYM−CUSCO and COV−CUSCO on functions with known combinational complexity.

A. Comparison of Single-Shot Synthesis With State-of-the-Art

This experiment has two goals:

1) Verify if extreme optimization can improve the result of single-shot symmetry-based synthesis.
2) Compare the result against the ones achievable with state-of-the-art synthesis and optimization.

Due to space limitations, in this experiment we focus on the *majority* functions, evaluating to 1 when the majority of the inputs is 1. Following experiments will discuss other symmetric functions. Table I shows the experimental results. We vary the input size from 3 to 20. Column |AIG| shows the results of running SYM−CUSCO$_{\times 1}$, i.e., the version introducing most a-priori information in the reward function.

As a first test of the quality of results, we perform aggressive optimization on top of the result obtained with our solver. We use the ABC command deepsyn. Deepsyn contains a random initialization procedure on which the quality of results depends. We consider 20 random initializations and halt the optimization after 500 iterations without any improvement (&deepsyn -I 20 -J 500). The entries of type $X \mapsto Y$ show that an AIG with X gates can be optimized to an AIG of size Y. Column T[s] reports the synthesis time. With up to 17 inputs, we can synthesize functions in less than a minute. Overall, high-effort optimization fails in finding optimizations in most of the representations obtained by SYM−CUSCO$_{\times 1}$, showing the stability of the minimum found by our solver.

The absence of optimization does not guarantee the global minimum. To obtain a strong baseline, we perform aggressive optimization after FRAIG synthesis. Column |FRAIG| reports the number of AND gates in the FRAIG generated from the truth table. Next, we repeat deepsyn-based optimization with high-effort hyperparameter tuning to obtain the most compact representation possible. The real-world time to obtain most of the results in Table I is of the order of minutes. However, we report an underestimation of the time invested by removing the time needed for the hyperparameter optimization and the time invested by deepsyn in performing unsuccessful optimizations. Despite the aggressive optimization and the underestimation of the time, our method finds the same solution or a better one in most cases. Furthermore, SYM−CUSCO can find the result within seconds for all the majority functions that we can represent as truth tables in ABC ($n = 3$ to 16).

TABLE I
Comparison of SYM−CUSCO$_{\times 1}$ and FRAIG-based synthesis.

	HYPERPARAMETER TUNING			SYM−CUSCO$_{\times 1}$	
n	|FRAIG|	|FRAIG*|	T$_\infty$[s]	|AIG|	T[s]
3	6	**4**	0.02	**4**	**0.00**
4	13	**7**	0.02	**7**	**0.00**
5	23	**10**	0.02	**10**	**0.01**
6	36	**15**	0.05	**15**	**0.01**
7	68	**20**	1.94	**20**	**0.02**
8	99	**25**	2.59	**25**	**0.04**
9	142	**30**	14.52	**30**	**0.07**
10	207	40	23.61	**37**	**0.11**
11	301	49	13.47	**44**	**0.27**
12	452	**49**	29.18	**49**	**0.51**
13	637	64	33.09	**58**	**1.39**
14	985	**62**	31.54	$67 \mapsto 66$	**3.24**
15	1357	**71**	86.46	72	**7.71**
16	2155	105	120.76	**79**	**21.24**
17	–	–	–	**92**	**57.78**
18	–	–	–	**101**	**118.0**
19	–	–	–	**116**	**295.09**
20	–	–	–	**121**	**600.00**

B. Other Symmetric Functions

We use the notation $S_{k_1,k_2,\ldots,k_m}$ to indicate the symmetric function which is satisfied when the number of ones at the input is $k_1 \vee k_2 \vee \ldots, k_m$. Threshold functions are perhaps the most practically interesting symmetric functions. However, we also investigates the synthesis of other symmetric functions. Symmetric functions with known optimum are:

1) *Gamble functions* $S_{0,n}$: $\mathcal{C}(S_{0,n}) = 2n - 1$.
2) *Parity functions* S_{2k} $k = 0 \ldots \lfloor \frac{n}{2} \rfloor$: $\mathcal{C}(S_{2k}) = n - 1$.
3) 2*-threshold functions* $S_{\geq 2} = S_{2,\ldots,n}$: $\mathcal{C}(S_{2k}) = 3n - 5$.
4) 1*-hot functions* S_1: $\mathcal{C}(S_1) = 2n - 3$.

SYM−CUSCO$_{\times 1}$ finds these exact results.

These experiments show that the a-priori assumptions used in SYM−CUSCO allow us to obtain compact representations of symmetric functions from a truth table representation.

C. Evaluating the A-Priori Reward Function

SYM−CUSCO$_{\times 1}$ relies on many assumptions:

1) It only considers the actions maximizing the number of remapped *don't cares*.
2) It prioritizes those actions introducing the lowest numbers of Boolean operators in the representation.
3) It prioritizes exploiting symmetries on the same subgroup of variables at different iterations.

The last requirement comes from the fact that actions with the same reward are equally likely to map the problem to a simpler one. Then, selecting actions involving variables manipulated in previous synthesis steps is likely to induce the removal of variables from the netlist, simplifying the problem.

Table II considers two classes of functions:

- The *threshold* functions $S_{\geq k}$ evaluate to 1 when there are at least k ones at the input.
- The *k-hot-encoding* functions S_k evaluate to 1 when there are k ones at the input.

The functions not represented in the table relate to one entry by duality. The vertical line identifies the *majority function*.

TABLE II
XAIGS FOR THRESHOLD FUNCTIONS AND k-HOT ENCODING FUNCTIONS.

THRESHOLD FUNCTIONS

2	${}_0 1^0$	${}_0 1^0$						
3	${}_0 2^0$	${}_0 4^0$						
4	${}_0 3^0$	${}_0 7^0$	${}_0 7^0$					
5	${}_0 4^0$	${}_0 10^0$	${}_0 10^0$					
6	${}_0 5^0$	${}_0 13^0$	${}_0 15^0$	${}_0 15^0$				
7	${}_0 6^0$	${}_0 16^0$	${}_0 18^0$	${}_0 20^0$				
8	${}_0 7^0$	${}_0 19^0$	${}_0 21^0$	${}_0 25^0$	${}_0 25^0$			
9	${}_0 8^0$	${}_0 22^0$	${}_0 24^0$	${}_0 30^0$	${}_0 30^2$			
10	${}_0 9^0$	${}_0 25^0$	${}_0 29^0$	${}_0 35^0$	${}_0 37^2$	${}_0 37^2$		
11	${}_0 10^0$	${}_0 28^0$	${}_0 32^0$	${}_0 40^0$	${}_0 42^4$	${}_0 44^4$		
12	${}_0 11^0$	${}_0 31^0$	${}_0 35^0$	${}_0 45^0$	${}_2 49^4$	${}_0 49^6$	${}_0 49^6$	
13	${}_0 12^0$	${}_0 34^0$	${}_0 38^0$	${}_0 50^0$	${}_2 52^6$	${}_0 56^6$	${}_2 58^4$	
14	${}_0 13^0$	${}_0 37^0$	${}_0 41^0$	${}_0 55^0$	${}_2 61^4$	${}_0 63^6$	${}_2 63^8$	${}_2 65^8$
15	${}_0 14^0$	${}_0 40^0$	${}_0 44^2$	${}_0 60^0$	66^4	${}_0 70^6$	${}_0 72^6$	${}_0 72^{10}$
n	$S_{\geq 1}$	$S_{\geq 2}$	$S_{\geq 3}$	$S_{\geq 4}$	$S_{\geq 5}$	$S_{\geq 6}$	$S_{\geq 7}$	$S_{\geq 8}$

k-HOT ENCODING FUNCTIONS

2	${}_0 1^0$	${}_0 1^0$						
3	${}_0 4^0$	${}_0 4^0$						
4	${}_0 7^0$	${}_0 7^0$	${}_0 7^0$					
5	${}_0 10^0$	${}_0 12^0$	${}_0 12^0$					
6	${}_0 13^0$	${}_0 15^0$	${}_0 17^0$	${}_0 15^0$				
7	${}_2 16^2$	${}_2 18^2$	${}_0 22^0$	${}_0 22^0$				
8	${}_0 19^0$	${}_0 21^0$	${}_0 27^0$	${}_0 27^4$	${}_0 27^0$			
9	${}_0 22^0$	${}_0 26^0$	${}_0 32^0$	${}_0 32^4$	${}_0 32^4$			
10	${}_0 25^0$	${}_0 29^0$	${}_0 37^0$	${}_2 39^4$	${}_0 39^6$	${}_2 39^4$		
11	${}_0 28^0$	${}_0 32^0$	${}_0 42^0$	${}_0 44^4$	${}_2 46^6$	${}_0 46^6$		
12	${}_0 31^0$	${}_0 35^0$	${}_0 47^0$	${}_2 51^6$	${}_2 53^6$	${}_0 53^8$	${}_0 53^6$	
13	${}_0 34^0$	${}_2 38^2$	${}_0 52^0$	${}_0 56^6$	${}_2 60^8$	${}_0 62^8$	${}_0 62^8$	
14	${}_0 37^0$	${}_0 41^0$	${}_0 57^0$	${}_0 65^4$	${}_0 67^8$	${}_0 69^8$	${}_0 71^8$	${}_4 67^{10}$
15	${}_0 40^0$	${}_2 44^2$	${}_0 62^0$	${}_0 68^8$	${}_4 72^8$	${}_2 78^8$	${}_4 76^{10}$	${}_0 78^{10}$
n	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8

We consider 33 iterations of the solver (SYM−CUSCO$_{\times 33}$) and indicate the quality of the results as ${}_{\delta_0}X^{\delta_M}$, where:

1) X is the minimum number of Boolean operators observed during synthesis.
2) δ_0 is the difference between the number of Boolean operators obtained by the SYM−CUSCO$_{\times 1}$ and X.
3) δ_M is the difference between the maximum number of Boolean operators observed during synthesis and X.

We highlight in red (${}_{\delta_0}X^{\delta_M}$) the cases in which SYM−CUSCO$_{\times 1}$ yields an empirically provable sub-optimal results. This experiment shows that, in general, SYM−CUSCO$_{\times 1}$ yields the best observed result in most cases, empirically validating the a-priori assumptions. Nevertheless, in some cases, breaking ties at random can identify more compact representations. Multiple runs of fast CUSCO-synthesis with stochastic selections of moves with high expected rewards result in a random process between high-quality solutions in the design space. This remark is interesting in light of the analysis done in Sec. IV-A because each one of these solutions is hardly reachable with traditional optimization strategies.

D. Comparison With Exact Synthesis

In this experiment, we investigate the quality of results for four- and five-input symmetric functions with known combinational complexity [12]. Table III compares SYM−CUSCO$_{\times 1}$ and COV−CUSCO$_{\times 100}$ (Sec. III-D). We consider a solution "close-to-optimum" if it is within 2 gates from the exact result, highlighting the results with the following colour code:

- $|XAIG|$: if $|XAIG| - \mathcal{C}(f) = 0$.
- $|XAIG|$: if $|XAIG| - \mathcal{C}(f) = 1$.
- $|XAIG|$: if $|XAIG| - \mathcal{C}(f) = 2$.
- $|XAIG|$: if $|XAIG| - \mathcal{C}(f) \geq 3$.

The notation $X \mapsto Y$ indicates that an XAIG optimization [19] reduces the gates count X of our synthesis strategy to Y. We also run the experiment with SYM−CUSCO$_{\times 100}$ but there was no major variation in the results, confirming the effectiveness of SYM−CUSCO$_{\times 1}$.

TABLE III
COMPARISON OF EXACT XAIGS SYNTHESIS WITH THE CUSCO SOLVERS.

function			SYM−CUSCO$_{\times 1}$		CUSCO$_{\times 100}$	
	f	$\mathcal{C}(f)$ [12]	|XAIG|	$T[s]$	|XAIG|	$T[s]$
$n = 4$	S_4	3	3	0.00	3	0.17
	S_3	7	7	0.00	7	1.28
	$S_{3,4}$	7	7	0.00	7	0.77
	S_2	6	7	0.00	6	0.33
	$S_{2,4}$	6	7	0.00	6	0.32
	$S_{2,3}$	6	$9 \mapsto 8$	0.00	7	0.83
	$S_{2,3,4}$	7	7	0.00	7	0.81
	S_1	7	7	0.00	7	1.54
	$S_{1,4}$	7	$9 \mapsto 8$	0.00	7	0.95
	$S_{1,3}$	3	3	0.00	3	0.24
	$S_{1,3,4}$	6	7	0.00	6	0.35
	$S_{1,2}$	6	$9 \mapsto 8$	0.00	7	0.92
	$S_{1,2,4}$	7	$9 \mapsto 8$	0.00	7	0.90
	$S_{1,2,3}$	5	7	0.00	5	0.48
	$S_{1,2,3,4}$	3	3	0.00	3	0.21
$n = 5$	S_4	10	10	0.00	10	6.47
	$S_{4,5}$	10	10	0.00	10	6.43
	S_3	9	12	0.01	9	20.14
	$S_{3,5}$	10	10	0.00	10	16.77
	$S_{3,4}$	10	13	0.01	$14 \mapsto 13$	8.17
	$S_{3,4,5}$	9	10	0.00	11	101.67
	$S_{2,5}$	10	14	0.01	11	18.71
	$S_{2,4}$	8	10	0.00	8	3.33
	$S_{2,4,5}$	9	12	0.00	$11 \mapsto 10$	47.33
	$S_{2,3,5}$	10	15	0.01	11	26.08
	$S_{2,3}$	8	15	0.01	9	4.76
	$S_{2,3,4}$	10	$13 \mapsto 12$	0.01	12	3.45
	$S_{1,5}$	9	$13 \mapsto 12$	0.01	10	7.77
	$S_{1,4}$	9	15	0.01	9	2.70
	$S_{1,3,4}$	11	13	0.01	11	11.18
	$S_{1,2,5}$	9	16	0.01	9	157.11

SYM−CUSCO$_{\times 1}$ finds compact XAIGs in a few milliseconds. However, the results are not optimal for most 5-input functions. On the other hand, COV−CUSCO$_{\times 100}$ finds close-to-optimum representations in most cases. However, the better results come at the cost of runtime and memory. Indeed, IGs require manipulating 2^{2n}-dimensional bit-strings, preventing the direct application of the method for more than 8 variables.

E. The Importance of Compact Representations

This experiment discusses the importance of close-to-optimum representations of Boolean functions. We consider the majority-of-9 and N input simulation patterns $\{p_i\}_{i=1}^{N}$, with $p_i \in \mathbb{B}^9$. We vary N with logarithmic increments in the range $N = 2^k \in [2^{10}, 2^{17}]$. Fig. 4 shows two ways of performing the input-output simulation. The *bit-level* simulation considers one pattern at a time. The *word-level* simulation considers multiple patterns at a time. We use words of length $2^l = 64$ bits, so we need 2^{k-l} words to store 2^k simulation patterns. Word-level simulation leverages the structural representation by word-wise application of the Boolean operators. The computational cost of a bit-level simulation is $\mathcal{O}(2^k)$. Instead, word-level simulation for an XAIG representation has a cost $\mathcal{O}(|XAIG|2^{k-l})$ because we need to perform a binary operation for each node in the XAIG. Fig. 4 shows the runtime of the simulation as a function of N. As expected, both graphs present an exponential dependence on n. The word-level simulation presents a shift of approximately $\Delta = \log|XAIG| - 6 \simeq 4.9 - 6 = -1.1$, where the size of the representation reduces the speed-up induced by parallelism. Hence, minimum-size representations have runtime advantages on top of resource advantages. A BNN simulator repeats this computation thousands of times. Hence, compact representations result in higher simulation performance, possibly enabling training and inference of BNNs without GPUs.

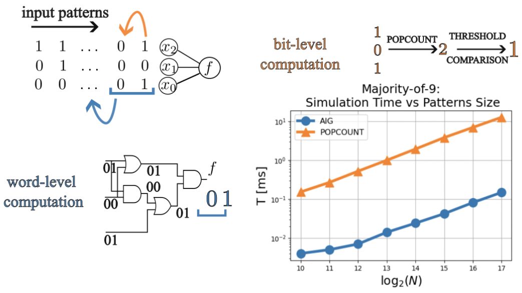


Fig. 4. Word-level simulation of a threshold function.

V. Conclusions

We propose a new EDA paradigm treating logic synthesis as a game (`CCG`). `CUSCO` is a class of synthesis heuristics acting as players in `CCG`. `CUSCO` solvers perform design space exploration guided by *don't care*-maximization:

1) `SYM-CUSCO` is our implementation of a classical symmetry-based synthesis technique.
2) `COV-CUSCO` is our generalization of `SYM-CUSCO`.

Experiments with `SYM-CUSCO` show that the exploration of the subspace identified by the topological constraints of the method yields close-to-optimum AIG sizes for symmetric functions. Experiments show that our generic solver performs better than the symmetry-based one on problems with known combinational complexity. Furthermore, `COV-CUSCO` allows extending *don't care*-maximization heuristics to functions without symmetries. Future works will investigate more advanced policies and reward functions, and ways of overcoming the runtime and memory limitations of `COV-CUSCO`.

Acknowledgment

This research was supported in part by Synopsys inc., in part by SRC Contract 3173.001 "Standardizing Boolean transforms to improve quality and runtime of CAD tools". The authors thank Satrajit Chatterjee and Alessandro Tempia Calvino for the inspiring discussions.

References

[1] M. Soeken, W. Haaswijk, E. Testa, A. Mishchenko, L. G. Amarù, R. K. Brayton, and G. De Micheli, "Practical exact synthesis," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 309–314.

[2] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli, "Sat-based exact synthesis: Encodings, topology families, and parallelism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 4, pp. 871–884, 2019.

[3] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*. Springer, 2008, pp. 337–340.

[4] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[5] T. Simons and D.-J. Lee, "A review of binarized neural networks," *Electronics*, vol. 8, no. 6, p. 661, 2019.

[6] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," *Advances in neural information processing systems*, vol. 29, 2016.

[7] C. R. Edwards and S. L. Hurst, "A digital synthesis procedure under function symmetries and mapping methods," *IEEE Transactions on Computers*, vol. 27, no. 11, pp. 985–997, 1978.

[8] S. Hurst, "Detection of symmetries in combinatorial functions by spectral means," *IEE Journal on Electronic Circuits and Systems*, vol. 1, no. 5, pp. 173–180, 1977.

[9] C. Scholl, "Multi-output functional decomposition with exploitation of don't cares," in *Proceedings Design, Automation and Test in Europe*. IEEE, 1998, pp. 743–748.

[10] R. M. Karp, "Reducibility among combinatorial problems, complexity of computer computations (re miller and jw thatcher, editors)," 1972.

[11] G. D. Micheli, *Synthesis and optimization of digital circuits*. McGraw-Hill Higher Education, 1994.

[12] D. E. Knuth, *The art of computer programming, volume 4A: combinatorial algorithms, part 1*. Pearson Education India, 2011.

[13] A. Mishchenko, S. Chatterjee, R. Jiang, and R. K. Brayton, "Fraigs: A unifying representation for logic synthesis and verification," ERL Technical Report, Tech. Rep., 2005.

[14] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22*. Springer, 2010, pp. 24–40.

[15] S. Yamashita, H. Sawada, and A. Nagoya, "A new method to express functional permissibilities for lut based fpgas and its applications," in *Proceedings of International Conference on Computer Aided Design*. IEEE, 1996, pp. 254–261.

[16] N. Modi and J. Cortadella, "Boolean decomposition using two-literal divisors," in *17th International Conference on VLSI Design. Proceedings.* IEEE, 2004, pp. 765–768.

[17] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," in *Proceedings of the 34th annual Design Automation Conference*, 1997, pp. 263–268.

[18] G. Meuli, M. Soeken, and G. De Micheli, "Xor-and-inverter graphs for quantum compilation," *npj Quantum Information*, vol. 8, no. 1, p. 7, 2022.

[19] S.-Y. Lee, H. Riener, A. Mishchenko, R. K. Brayton, and G. De Micheli, "A simulation-guided paradigm for logic synthesis and verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 8, pp. 2573–2586, 2021.