Three-Input Gates for Logic Synthesis

Dewmini Sudara Marakkalage*, Eleonora Testa*, Heinz Riener*,
Alan Mishchenko†, Mathias Soeken*, Giovanni De Micheli*

*Integrated Systems Laboratory, EPFL, Lausanne, Switzerland
†Department of EECS, UC Berkeley, Berkeley, California, USA

***Abstract*—Most logic synthesis algorithms work on graph representations of logic functions with nodes associated with arbitrary logic expressions or simple logic functions and attempt to iteratively optimize such graphs. While early logic synthesis efforts focused primarily on graphs with 2-input nodes such as AND and OR gates, the recently proposed paradigm of Majority-Inverter Graphs instead uses the 3-input Majority gate as the node function. As this technique proved to be a success, it is natural to ask: Are there other 3-input gates better suited for logic synthesis?**

Motivated by this question, we investigate the relative advantages of 3-input gates as constituents of logic networks. We consider representative gates from each of the ten non-degenerate 3-input NPN classes and study how powerful they are at representing Boolean functions. Using SAT-based exact synthesis, we evaluate each 3-input gate using the minimum number of such gates (together with inverters) needed to synthesize all 4-input Boolean functions and a subset of commonly used 5-input and 6-input Boolean functions. We show that the logic gate $\mathrm{Dot}(x,y,z) := x \oplus (z \vee xy)$ outperforms the rest in terms of expressive power. Motivated by this result, we introduce a set of rewriting rules to manipulate *Dot*-Inverter Graphs.

***Index Terms*—Logic synthesis, Exact synthesis, 3-input gates.**

I. Introduction

Given a Boolean function, what is the minimum-size circuit that computes it? This is one of the driving questions in logic synthesis, which is the process of optimizing logic representations under various criteria. Decades of research on this problem have considered various circuit models and produced many synthesis algorithms [1], [2], [3], [4], [5]. In general, the problem of finding the smallest circuit for a given Boolean function is a computationally difficult task, and exact minimization can be done reasonably fast only for Boolean functions with a small number of inputs. Consequently, most synthesis algorithms do not find optimum representations but focus on heuristic solutions. They usually work on graph representations of logic functions where each node is associated with an arbitrary logic expression (e.g., YLE [6], MIS [7]) or a simple logic function, and they try to incrementally modify such graphs in order to minimize the size or depth.

Early logic synthesis efforts on this front considered graphs with nodes computing 2-input ANDs and ORs together with inverters (or NANDs and NORs), which included the well-known And-Inverter Graphs (AIGs). An AIG is a Directed Acyclic Graph (DAG) where each internal node has in-degree two and represents a 2-input AND gate, and each directed edge is either complemented or regular indicating the presence/absence of an inverter along that edge. As an example use-case, the logic synthesis tool ABC [3] uses AIGs as the primary logic representation and implements associated re-writing techniques [8], [9] for optimizing them. We call graph representations such as AIGs *homogeneous* as each internal node in the graph computes the same logic function. Optimizing homogeneous multilevel logic representations is typically more scalable and leads to better results.

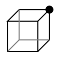
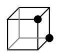
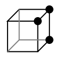
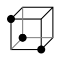
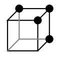
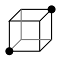
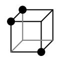
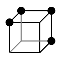
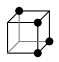
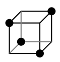
Recently, Amarú et al. [10], [11] proposed Majority-Inverter Graphs (MIGs) as a new paradigm for logic synthesis. An MIG is also a homogeneous DAG representation similar to an AIG: The only difference is that internal nodes of an MIG have in-degree *three* and represent *3-input majority* gates. The authors further introduced a set of algebraic rules for manipulating MIGs, proposed new synthesis algorithms, and showed significant results for depth optimization over ASIC and FPGA designs. The success of MIGs for logic synthesis begs the following question: Are there other Gate-Inverter Graphs (i.e., homogenous DAGs where each internal node is associated with some fixed 3-input logic gate[1]) that are better suited for logic synthesis?

The answer depends on several criteria, such as the expressive power of the 3-input gate, the rules to manipulate such gate networks, and practical considerations such as the suitability of the gate for physical design. This work primarily focuses on the expressive power of the 3-input gates. To elaborate, we study how succinctly different Gate-Inverter Graphs can represent a given Boolean function, assuming that the use of inverters does not count towards the size of such graphs. We note that the homogeneity of Gate-Inverter Graphs makes their manipulation using synthesis algorithms easier compared to the non-homogeneous ones.

There are $2^8 = 256$ 3-input Boolean functions to be considered as potential 3-input gates. However, due to the zero cost of inverters and the ability to rewire a gate's inputs in any order, many of these 3-input gates can be considered equivalent (the transformations do not incur any additional cost). For instance, the function $g(x,y,z) := x \wedge (y \vee z)$ is equivalent to the function $h(x,y,z) := \overline{(z \wedge (\bar{x} \vee y))}$ with respect to input negations, input permutations, and output negations because $g(x,y,z) = \overline{h(\bar{y},z,x)}$. This notion of equivalence is called NPN (Negation-Permutation-Negation)

[1]Unlike the 3-input majority gate, a general 3-input gate can be asymmetric, thus in such a DAG representation, the ordering of the fan-in must be specified as well.

Table I: 10 NPN classes for 3-input functions and their representative functions. The column *Class* is the class representative, which is the lexicographically smallest truth table (as two hexadecimal digits) for each NPN class, and the column *Function* represents the truth table of the candidate 3-input function selected from each class. The column *MC* reports the multiplicative complexity of each function [14], [15].

Class	Expression	Function	Cube	Name	MC
#01	$x\,y\,z$	#80		And3	2
#06	$x\,(y \oplus z)$	#28		XorAnd	1
#07	$x\,(y \vee z)$	#a8		OrAnd	2
#16	$x\,\bar{y}\,\bar{z} \oplus \bar{x}\,y\,\bar{z} \oplus \bar{x}\,\bar{y}\,z$	#16		Onehot	2
#17	$\langle x\,y\,z\rangle$	#e8		Majority	1
#18	$x\,y\,z \oplus \bar{x}\,\bar{y}\,\bar{z}$	#81		Gamble	1
#19	$x \oplus (z \vee x\,y)$	#52		Dot	2
#1b	$x\,?\,y : z$	#d8		Mux	1
#1e	$x \oplus y\,z$	#6a		AndXor	1
#69	$x \oplus y \oplus z$	#96		Xor3	0

equivalence [12], [13]. Formally, two Boolean functions are NPN equivalent if one can be obtained by the other using a combination of input negations, input permutations, and output negation. Consequently, all n-input Boolean functions can be partitioned into NPN equivalence classes. The 256 3-input functions fall into 14 different NPN classes, out of which, four classes only depend on at most two variables ($f(x,y,z) = 0$, $f(x,y,z) = x$, $f(x,y,z) = x \wedge y$, and $f(x,y,z) = x \oplus y$). Thus, we only consider the remaining ten 3-input NPN classes that depend on all three variables and select one function from each class as candidate gates for building Gate-Inverter Graphs. Table I lists the ten 3-input NPN classes and the candidate functions, together with their names.

As the main result of this work, we present a comparison of the expressive powers of the 3-input gates mentioned in Table I. To this end, we measure the minimum size of a Gate-Inverter Graph of each type needed to compute each 4-input Boolean function using *SAT-based exact synthesis* [16], [17]. Note that the size of Gate-Inverter Graph is the number of gates in it excluding inverters. We omit the 3-input Xor (Xor3) gate from our analysis because it is not a universal 3-input gate as we show in Section III-A.

Our results show that *Dot* gate has the highest expressive power closely followed by the *Onehot* gate: Dot-Inverter Graph and Onehot-Inverter Graph require at most four gates to compute any given 4-input Boolean function. On the opposite end, 3-input *And* (*And3*) gates have the least expressive power, as there exist some 4-input Boolean functions that need up to nine gates. We further confirm these results by running SAT-based exact synthesis for some commonly used 5-input and 6-input Boolean functions.

For a Gate-Inverter Graph to be useful for optimizing logic networks, in addition to the expressive power of the gates, we also need the ability to easily manipulate such graphs. For example, there exists a sound and complete axiomatization and a comprehensive set of rewriting rules for MIGs [10], [18]. Since the Dot gate has high expressive power, we also present several non-trivial rewriting rules for Dot-Inverter Graphs, which can be employed to manipulate and optimize them.

II. SAT-based Exact Synthesis

In this section, we introduce *SAT-based exact synthesis* as presented in [16], [17], [19], and [20], and show how it is applied in the context of 3-input gate networks.

Exact synthesis is the problem of finding a logic network that exactly meets its specification or determines whether it is impossible to do so. In our case, given a 3-input gate $\mathcal{T}$, a non-negative integer r, and a Boolean function f, the goal is to find whether there exists a $\mathcal{T}$-Inverter Graph of size r that computes f. Starting with $r = 0$ and incrementing it until the synthesis algorithm finds a valid circuit, we determine the minimum number of gates to compute f. In the following, we first formalize the notion of 3-input gate networks, and then we show how to encode the exact synthesis problem as a Boolean satisfiability (SAT) problem [17]. The exact synthesis algorithm uses a SAT solver to find a satisfying assignment to the problem or to determine its unsatisfiability. If a satisfying assignment is found, the algorithm decodes it into a valid logic network. We refer the interested reader to [21], [20] for a more detailed review on exact synthesis, while the first example of SAT-based exact synthesis can be found in [22], and successive analyses and improvements have been considered in [16], [17].

A. 3-Input Gate Networks

Let P be a collection of 3-input Boolean operators $\phi : \mathbb{B}^3 \to \mathbb{B}$ where $\mathbb{B} = \{0,1\}$ is the Boolean alphabet. We call P the set of *primitives.* (For the purpose of encoding as a SAT problem, we will describe how to construct P in Section II-B.) Let $f(x_1,\ldots,x_n)$ be a Boolean function on n inputs, and for notational convenience, define $x_0 = 0$. We call a sequence $(x_{n+1}, x_{n+2}, \ldots, x_{n+r})$ a 3-input gate network of size r if $x_j = \phi_j(x_{j_1}, x_{j_2}, x_{j_3})$ for all $n+1 \le j \le n+r$ where $\phi_j \in P$ and $0 \le j_1, j_2, j_3 < j$. Note that such a network corresponds to a Directed Acyclic Graph (DAG) where each leaf node corresponds to an input variable or constant 0, and each non-leaf node has in-degree 3 and corresponds to some Boolean operator in P. The sequence $(x_{n+1},\ldots,x_{n+r})$ defines a topological ordering of non-leaf vertices.

We say a given gate network $(x_{n+1}, x_{n+2}, \ldots, x_{n+r})$ computes f if $f(x_1, \ldots, x_n) = x_{n+r}$. In general, if $F = \{f_1, f_2, \ldots f_k\}$ is a set of k Boolean functions on the common support $x_1, \ldots, x_n$, we say the gate network computes F if each function $f_j \in F$ is computed by some x_ℓ in the network.

B. Encoding as a SAT Problem

We encode the problem of finding a 3-input gate network of size r that computes a given set F of output functions on n input variables as a SAT problem using the Single Selection Variable (SSV) encoding [19], [20], [21]. The SSV encoding uses a single variable per Boolean operator to encode the inputs of the operator. Namely, it uses binary variables $s_{\ell,i,j,k}$ which are set to 1 if x_i, x_j, and x_k are the inputs for the ℓ-th operator in the network.

In the SSV encoding, to reduce the number of variables, the variables $s_{\ell,i,j,k}$ are only defined for $i < j < k < \ell$. However, since we should instead consider repeated inputs in a gate's fan-in as well as different orderings of those inputs for non-associative operators, we add all input-permuted (with repetitions) versions of an operator as primitives for SAT-based exact synthesis. To elaborate, if $\phi(x, y, z) \in P$, we make sure that $\phi(x, x, x), \phi(x, x, y), \phi(x, x, z), \phi(x, y, x), \phi(x, y, y)$, etc. also belong to P.

To allow inverters at no additional cost, for every Boolean operator $\phi(x, y, z)$, we add input negated versions to P, i.e., we add $\phi(x, y, \bar{z}), \phi(x, \bar{y}, z), \phi(x, \bar{y}, \bar{z})$ and so on to P. Furthermore, to avoid explicitly considering constants as inputs, for each operator $\phi(x, y, z) \in P$ we also add its versions where subsets of inputs are replaced by constants, i.e., we add $\phi(x, y, 0), \phi(x, y, 1), \phi(x, 0, z), \phi(x, 0, 0), \phi(x, 0, 1), \phi(x, 1, z)$, etc. to P.

Note that the different primitives in P correspond to the different fan-in configurations a 3-input gate in a Gate-Inverter Graph can have. The primitives obtained by different permutations with repetitions take care of the different fan-in orders a gate can have and the fact that it can have multi-edges (repeated inputs). Similarly, having input negated versions of the primitives take care of the fact that a gate's fan-in edges being either regular or complemented.

The SSV encoding also uses the symmetry-breaking assumption that all logic primitives are *normal* (i.e., the output is zero when all inputs are zero), thus, we negate any primitive that is not normal. Note that, due to the zero-cost inverters, this normality assumption does not affect the accuracy of the SAT-based exact synthesis.

III. Evaluation Method and Results

In this section, we study how the 3-input gates of Table I can be used as a basis of representing logic functions with larger support. Using exact methods, we investigate the minimum number of each such 3-input gate needed to compute each 2-input (Section III-A), 3-input (Section III-B), and 4-input (Section III-C) logic functions. As the main result of this work, we summarize our findings for 4-input logic functions in Table IV, which serves as a relative measure of the expressive power of the considered 3-input gates. Finally, in Section III-D, we present synthesis results of some common 5-input and 6-input functions. Note that applying the same exact methods for all 5-input NPN functions would be vastly time-consuming in a conventional computing setting, and prohibitively so for functions with even larger supports.

For exact synthesis, we use the SAT-based exact synthesis library *percy*,[2] which is a part of EPFL logic synthesis libraries [23], and we choose the SSV encoding described in Section II as our encoding method. Given a 3-input logic gate $\mathscr{T} : \mathbb{B}^3 \to \mathbb{B}$, to synthesize a Gate-Inverter Graph using base gate $\mathscr{T}$ with *percy*, we first compute the correct set of primitives. Note that $\mathscr{T}$ is one of the ten gates from Table I and recall that the primitives should represent all versions of $\mathscr{T}$ obtained by

- permuting the inputs (with repetition),
- negating a subset of the inputs, and
- replacing a subset of the inputs with constants.

However, also recall that the SSV encoding uses the assumption that all primitives are normal. Hence, we replace any primitive that is not normal with its complement. Formally, let

$$Q_{\mathscr{T}} = \left\{\mathscr{T}(a, b, c) : (a, b, c) \in \{x, \bar{x}, y, \bar{y}, z, \bar{z}, 0, 1\}^3\right\}.$$

Then, we define the set of primitives $P_{\mathscr{T}}$ as follows: For each $q \in Q_{\mathscr{T}}$, if $q(0,0,0) = 0$ then we add q to $P_{\mathscr{T}}$, otherwise we add $\bar{q}$ to $P_{\mathscr{T}}$. Using the set of primitives $P_{\mathscr{T}}$, we invoke *percy*'s exact synthesis algorithm using the standard synthesis engine with ABC's BSAT2 as the default SAT solver, which is a modified MiniSAT solver [3], [24].

Using the aforementioned procedure, for each 3-input gate type, we study the minimum number of such gates needed to compute all n-input NPN classes for $n = 2, 3, 4$, and a selected set of NPN classes for $n = 5$ and 6 . These results are presented next.

A. Synthesizing 2-Input NPN Classes

The only two non-trivial NPN classes of 2-inputs are the AND and the XOR function.[3] Table II shows how to compute the 2-input AND and the 2-input XOR functions using each 3-input gate type. For example, using the 3-input Dot gate, we can construct the 2-input AND gate as $Dot(x, \bar{y}, 0)$. Concerning the 2-input XOR function, note that it takes two 3-input OrAnd gates to implement the 2-input XOR gate, while the Majority and And3 gates need three instances each. Indeed, these three are the only monotone[4] ones out of the ten functions. Moreover, the Majority and And3 gates are also symmetric. It turns out that their best representation for a 2-input XOR is to implement the usual $x\bar{y} + \bar{x}y$ formula.

[2]Available at: *https://github.com/lsils/percy*

[3]The remaining two classes are the constant function $f(x, y) = 0$ and the projection function $f(x, y) = y$.

[4]The representative function is monotone. For other NPN classes, no member function is monotone.

Table II: Representations of 2-input And and Xor gates

	2-input AND		**2-input XOR**	
Gate Type	**Number of Gates**	$\mathbf{x \wedge y}$	**Number of Gates**	$\mathbf{x \oplus y}$
Dot	1	$(x, \bar{y}, 0)$	1	$(x, 0, y)$
Onehot	1	$(\bar{x}, \bar{y}, 1)$	1	$(x, y, 0)$
Mux	1	$(x, y, 0)$	1	$(x, \bar{y}, y)$
AndXor	1	$(0, x, y)$	1	$(x, y, 1)$
XorAnd	1	$(x, y, 0)$	1	$(1, x, y)$
Gamble	1	$(x, y, 1)$	1	$(x, x, \bar{y})$
OrAnd	1	$(x, y, 0)$	2	$(\overline{(\bar{x}, \bar{y}, 0)}, \bar{x}, \bar{y})$
Majority	1	$(x, y, 0)$	3	$((x, \bar{y}, 0), 1, (\bar{x}, y, 0))$
And3	1	$(x, y, 1)$	3	$\overline{(\overline{(x, \bar{y}, 1)}1\overline{(\bar{x}, y, 1)})}$
Xor3	-	-	1	$(x, y, 0)$

As evident from Table II, all the gates except Xor3 can be used to construct the 2-input AND gate. Note that AIGs can represent any Boolean function. We call such representations *universal*, and the results of Table II thus implies that all 3-input Gate-Inverter Graphs where "Gate" is any 3-input gate from Table I except Xor3 are universal representations. The Xor3-Inverter Graph is not a universal representation as the truth-table of each node in such a graph must have an even number of '1's in the output column which prevents them from representing Boolean function that has an odd number of '1's in the output column. Hence, in the following, we further restrict our focus to the remaining nine 3-input gates.

B. Synthesizing 3-Input NPN Classes

In this section, we synthesize the ten 3-input NPN classes using the selected gates. Note that these classes are the 3-input functions we considered in Table I.

Table III summarizes the synthesis results. Each column represents one of the nine candidate gates for universal 3-input Gate-Inverter Graphs and shows the number of gates to compute each of the other 3-input functions. For example, the third column is for 3-input Mux gates. It means that, to implement the Dot function, 2 Muxes are needed, while, to implement the Onehot function, we need 3 Muxes, and so on.

It is worth noting that Dot and Onehot gates need the smallest number of gates to compute the remaining three input classes. As noted previously, the monotone functions OrAnd, Majority, and And3 need a high number of gates to synthesize other functions. However, the property of symmetry does not appear to affect the expressiveness as demonstrated by the Onehot gate.

Also note that no 3-input function pair needs five gates, whereas implementing Xor3 needs six And3 gates. This implies that for And3 gates, implementing the Xor3 function is significantly more complex than implementing the remaining functions in Table III.

Table III: Minimum number of gates to synthesize 3-input NPN classes.

Gate Type / Function	**Dot**	**Onehot**	**Mux**	**AndXor**	**XorAnd**	**Gamble**	**OrAnd**	**Majority**	**And3**
Dot	1	2	2	2	2	2	2	4	3
Onehot	2	1	3	3	3	3	4	4	4
Mux	2	2	1	2	2	3	2	3	3
AndXor	2	2	2	1	2	2	3	4	4
XorAnd	2	2	2	2	1	2	2	3	3
Gamble	3	3	3	3	2	1	3	4	3
OrAnd	2	2	2	2	2	2	1	2	2
Majority	2	2	2	3	3	2	2	1	4
And3	2	2	2	2	2	2	2	2	1
Xor3	2	2	2	2	2	2	4	3	6
Total	20	20	21	22	21	21	25	30	33

C. Synthesizing 4-Input NPN Classes

The results of the previous section hint that Dot and Onehot gates seem to outperform the rest in their expressive power. We further confirm this claim by synthesizing all 222 4-input NPN classes using each type of 3-input gates.

Table IV shows, for each 3-input gate $\mathscr{T}$ and for each gate count r, the number of NPN classes (number of functions) that need r gates of type $\mathscr{T}$. For example, consider the first column: There are two 4-input NPN classes (ten functions) that do not need any Dot gate (i.e., the trivial functions), three classes (252 functions) that need only one Dot gate (2-input And and Xor, and the Dot function itself), 32 NPN classes (9128 functions) that need two Dot gates, etc.

As shown in Table IV, the nine 3-input gates roughly correspond to three categories. The Dot and Onehot gates clearly outperform the rest and use the smallest number of gates to represent 4-input functions. On the other side of the spectrum, OrAnd, Majority, and And3 gates use the most number of gates. Notice that, for the And3 gate, there is one NPN class whose synthesis needs nine gates. Unsurprisingly, it turns out to be the NPN class of the 4-input Xor function (recall that the synthesis of the 3-input Xor (Xor3) function needed six 3-input And gates). The remaining four gates, Mux, AndXor, XorAnd, and Gamble fall in the middle of the two former categories.

D. Synthesizing Common 5-Input and 6-Input Functions

In this section, we search for an indication of the expressive power of the considered nine 3-input gates on some commonly used 5-input and 6-input functions. At this purpose, we computed the 50k most popular 5-input and 6-input functions from the LUT mapping of the EPFL benchmarks,[5] and classified them according to NPN-classification. We obtained 387 5-input NPN classes and 1905 6-input NPN classes on which we apply our exact synthesis method. To keep the runtime under control, we

[5]Available at: *https://github.com/lsils/benchmarks*

Table IV: Classification of the 222 4-input classes (65536 4-input functions). This shows, for each 3-input gate $\mathcal{T}$ and for each gate count r, the number of 4-input NPN classes (number of 4-input functions) whose synthesis needs r gates of type $\mathcal{T}$. The last row shows the total number of gates needed to separately synthesize all 4-input NPN classes (all 4-input functions).

Gate Count	Dot	Onehot	Mux	AndXor	XorAnd	Gamble	OrAnd	Majority	And3
0	2 (10)	2 (10)	2 (10)	2 (10)	2 (10)	2 (10)	2 (10)	2 (10)	2 (10)
1	3 (252)	3 (124)	3 (156)	3 (156)	3 (156)	3 (92)	2 (240)	2 (80)	2 (112)
2	32 (9128)	18 (2856)	17 (3224)	15 (2776)	18 (2336)	13 (1272)	14 (3020)	5 (640)	4 (544)
3	158 (51770)	151 (50490)	92 (31554)	86 (27202)	83 (26786)	63 (14242)	46 (14528)	18 (3300)	13 (2508)
4	27 (4376)	48 (12056)	100 (29936)	110 (34864)	109 (35032)	115 (41856)	89 (30854)	42 (10352)	46 (14944)
5			8 (656)	6 (528)	7 (1216)	26 (8064)	55 (15064)	117 (40064)	68 (24024)
6							14 (1810)	35 (11058)	55 (17376)
7								1 (32)	26 (5680)
8									5 (336)
9									1 (2)
Total	649 (191322)	684 (205530)	753 (224290)	761 (229410)	759 (231394)	808 (253106)	883 (259500)	1036 (319560)	1134 (335342)

Table V: Common 5-input classes. This shows, for each 3-input gate $\mathcal{T}$ and for each gate count r, how many of the 316 common NPN classes need r gates in exact synthesis. The last row shows the total number of gates needed to separately synthesize each such NPN class.

Gate Count	Dot	Onehot	Mux	AndXor	XorAnd	Gamble	OrAnd	Majority	And3
2			2	1	3		4		2
3	97	7	49	58	68	18	72	11	12
4	208	249	204	189	180	154	136	60	73
5	11	60	56	68	65	143	90	131	94
6			5			1	14	114	83
7									52
Total	1178	1317	1277	1272	1255	1391	1302	1612	1664

Table VI: Common 6-input classes. This table, similarly to Table V, shows the results for 1905 6-input classes. The row CLE reports the number of conflict limit exceeded cases – thus the number of non-synthesized classes.

Gate Count	Dot	Onehot	Mux	AndXor	XorAnd	Gamble	OrAnd	Majority	And3
3	6		15	31	51	1	65	7	12
4	551	33	262	400	424	75	377	55	96
5	931	923	965	771	815	796	490	296	323
6			32			31	162	421	254
CLE	417	949	631	703	615	1002	811	1126	1220

set a maximum conflict limit for the SAT solver of 10M (5-input) and 1M (6-input), respectively.

For the 5-input classes, 316 (out of 387) classes have been synthesized – within the conflict limit – by all nine Gate-Inverter Graphs. In Table V, we summarize the results for these 316 NPN classes which confirms the trend observed in Table IV: Dot uses fewer number of gates, while And3 and Majority instead have the highest number of gates. It is also worth mentioning that And3 and Majority have the highest number of non-synthesized classes (conflict limit exceeded) equal to 59 and 30 classes (out of a total of 387), respectively.

Regarding the 6-input classes, only 213 classes (out of 1905) were synthesized by all 9 Gate-Inverter Graphs within the conflict limit of 1M. In this case, we thus present the complete set of results on all 1905 classes, showing also the number of classes that could not be synthesized within the conflict limit. The results are shown in Table VI, and they confirm our previous observations as well. The Dot gate synthesizes the highest number of classes (1488) within the conflict limit, and it uses at most 5 gates on each synthesized class. On the other side, And3 and Majority could synthesize only 779 and 685 classes within the conflict limit, respectively.

IV. Rewrite Rules for Dot-Inverter Graphs

As the results of Section III suggest, the Dot gate seems to minimize the number of gates required to represent general Boolean functions. Consequently, the exact synthesis of Dot-Inverter Graphs is faster and hence can be applied to relatively large logic synthesis problems, suggesting that such graphs could be useful as representations for logic optimization. However, as mentioned earlier, the exact synthesis of logic functions with five or more inputs is highly time-consuming and hence it is impractical today. Motivated by this computational hurdle, in this section, we show that Dot-Inverter Graphs admit a set of rewrite rules that can be useful in heuristic-based optimization algorithms.

To find useful rewrite rules, we considered all possible Dot-Inverter Graphs of up to five primary inputs and up to three Dot gates, and considered the output function of the top gate of each such graph. Then we grouped together the Dot-Inverter Graphs that compute the same function and manually observed the graph structures to identify non-trivial rewrite rules. As the second major contribution of this work, we present the identified rules below with their proofs.

For convenience, we use the notation (x, y, z) to denote the Dot gate with inputs x, y, and z. In the proofs of the rewrite rules, we use the following simplifications of the Dot gate when one input is a constant.

- $(0, y, z) = 0 \oplus (z \vee 0\, y) = z$,
- $(1, y, z) = 1 \oplus (z \vee 1\, y) = \overline{y \vee z} = \bar{y}\,\bar{z}$,
- $(x, 0, z) = x \oplus (z \vee x0) = x \oplus z$,
- $(x, 1, z) = x \oplus (z \vee x1) = x \oplus (z \vee x) = \bar{x}z$,
- $(x, y, 0) = x \oplus (0 \vee x\,y) = x \oplus x\,y = x\bar{y}$,
- $(x, y, 1) = x \oplus (1 \vee x\,y) = x \oplus 1 = \bar{x}$.

Rule 1 - $(x\,y\,z) = (x\,(z\,x\,y)\,z)$:

Proof. If $z = 0$, then $RHS = (x,(0,x,y),0) = (x,y,0) = LHS$. On the other hand, if $z = 1$, then both LHS and RHS are $\bar{z}$ according to the last simplification rule. □

Rule 2 - $(z\,x\,(x\,y\,z)) = (y\,x(x\,z\,y))$:

Proof. When $x = 0$, we have $LHS = (z,0,(0,y,z)) = (z,0,z) = (z \oplus z) = 0$. Since RHS is equal to swapping y and z in LHS, $RHS = (y \oplus y) = 0$ as well. On the other hand, when $x = 1$, we have $LHS = (z,1,(1,y,z)) = (z,1,\bar{y}\,\bar{z}) = \bar{z}\,\bar{y}\,\bar{z} = \bar{y}\,\bar{z}$, and swapping y and z as before, RHS is also $\bar{y}\,\bar{z}$. □

Rule 3 - $(x\,z\,(y\,x\,z)) = (y\,z\,(x\,y\,z))$:

Proof. When $z = 0$, then $LHS = (x,0,(y,x,0)) = (x,0,(y\,\bar{x})) = x \oplus y\bar{x} = (x \vee y)$. As before, since RHS is obtained by swapping x and y in LHS, $RHS = x \vee y$ as well. For the case when $z = 1$, we have $LHS = (x,1,(y,x,1)) = (x,1,\bar{y}) = \bar{x}\,\bar{y}$, and again by swapping x and y, RHS is also the same. □

Rule 4 - $(x\,y\,(y\,z\,u)) = (u\,y\,(y\,z\,x))$:

Proof. When $y = 0$, $LHS = (x,0,(0,z,u)) = (x,0,u) = x \oplus u$. Swapping x and u, we get $x \oplus u = RHS$ as well. When $y = 1$, $LHS = (x,1,(1,z,u)) = (x,1,\bar{z}\,\bar{u}) = \bar{x}\,\bar{z}\,\bar{u}$, and again swapping x and u as before, we get $LSH = RHS$. □

Rule 5 - $(((x\,y\,z)\,x\,u)\,y\,u) = (((x\,y\,z)\,y\,u)\,x\,u)$:

Proof. For $u = 0$, we have $LHS = (((x,y,z),x,0),y,0) = ((x,y,z)\bar{x},y,0) = (((x,y,z)\bar{x})\,\bar{y}) = (((x,y,z)\,\bar{y})\,\bar{x}) = (((x,y,z),\bar{y}),x,0) = (((x,y,z),y,0),x,0) = RHS$. For $u = 1$, we have $LHS = (((x,y,z),x,1),y,1) = ((\overline{(x,y,z)}),y,1) = (x,y,z)$ and a similar simplification yields $RHS = (x,y,z)$ as well. □

Finding a concise set of rewriting rules for 3-input functions is challenging and requires more fundamental research, but we are already able to prove several useful rewrite rules for the manipulation of Dot-Inverter Graphs.

Nevertheless, we think that there remain many other such rules to be discovered, and extending it to a more useful algebra will be an interesting research challenge.

V. Discussion

State-of-the-art logic optimization tools use data structures such as AIGs or MIGs which are based on simple logic primitives such as And gates, 3-input Majority gates, and Inverters to represent logic networks. In the optimization stage, these data structures are manipulated using various algorithms to achieve different objectives such as depth or size reduction. Two desired qualities of such data structures are (i) the expressive power of the used logic primitives and (ii) the ability to easily manipulate them.

In this paper, we studied whether there exist better logic primitives for this task. In particular, we considered different 3-input logic gates and analyzed their expressive power using SAT-based exact synthesis. We show that the 3-input logic gate Dot is the most powerful in terms of expressibility and it uses a significantly fewer number of gates to represent 4-input functions as compared to And3 or Majority gates. We provided further evidence to support this observation by synthesizing a set of common 5-input and 6-input NPN classes. Finally, we presented a few rewriting rules that are useful for manipulating Dot-Inverter Graphs.

Our results show that monotone gates (And3, Majority, and OrAnd) have less expressive power as compared to their non-monotone counterparts even in the presence of zero-cost inverters. This is intuitive as we certainly need a combination of monotone gates to represent non-monotone functions. However, somewhat counter-intuitively, the symmetric property of the gates seems not to affect their expressibility as suggested by the results for Onehot gates when synthesizing 4-input functions. In general, if a function is highly asymmetric in its variables, this means that permuting the inputs makes it compute different logic functions. In particular, a single Dot gate can compute six different logic functions by just permuting its inputs. Thus, intuitively, a few of them are sufficient to compute a large number of logic functions. On the other hand, a symmetric function such as Onehot stays the same when inputs are permuted, hence one would expect its expressive power to be relatively low.

In terms of manipulative power, the existence of non-trivial rewrite rules seems promising for Dot-Inverter Graphs. However, we need further research to expand the set of rewriting rules for Dot-Inverter Graphs and to determine whether such rules exist for other types of Gate-Inverter Graphs as well. In particular, it seems reasonable to explore such rules for logic networks of gates with moderate expressive power such as AndXor, XorAnd, and Gamble as there seems to be a trade-off between a gate's expressive power and the ability to manipulate networks of such gates.

Acknowledgments

We acknowledge Niklas Een for having suggested the interesting problem addressed in this work. This research was supported by the Swiss National Science Foundation (200021-169084 MAJesty), by H2020-ERC-2014-ADG 669354 CyberCare, by the EPFL Open Science Fund, and by SRC contract 2867.001.

References

[1] E. Testa, M. Soeken, L. Amarú, and G. De Micheli, "Logic synthesis for established and emerging computing," *Proceedings of the IEEE*, pp. 1–20, 2018.

[2] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proceedings of the IEEE*, vol. 78, no. 2, pp. 264–300, 1990.

[3] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *Computer Aided Verification*, 2010, pp. 24–40.

[4] H. Riener, E. Testa, W. Haaswijk, A. Mishchenko, L. Amarú, G. De Micheli, and M. Soeken, "Scalable generic logic synthesis: One approach to rule them all," in *Design Automation Conference*, 2019, pp. 1–6.

[5] S. Muroga, "Logic design and switching theory," 1979.

[6] R. K. Brayton, "The yorktown silicon compiler," *Silicon Compilation*, pp. 204–311, 1988.

[7] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli, and A. R. Wang, “MIS: A multiple-level logic optimization system,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 6, no. 6, pp. 1062–1081, 1987.

[8] P. Bjesse and A. Boralv, “DAG-aware circuit compression for formal verification,” in *Int’l Conf. on Computer-Aided Design*, 2004, pp. 42–49.

[9] A. Mishchenko, S. Chatterjee, and R. K. Brayton, “DAG-aware AIG rewriting a fresh look at combinational logic synthesis,” in *Design Automation Conference*, 2006, pp. 532–535.

[10] L. Amarú, P. Gaillardon, and G. De Micheli, “Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization,” in *Design Automation Conference*, 2014, pp. 1–6.

[11] ——, “Majority-inverter graph: A new paradigm for logic optimization,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 35, no. 5, pp. 806–819, 2016.

[12] E. Goto and H. Takahasi, “Some theorems useful in threshold logic for enumerating Boolean functions,” in *IFIP Congress*, 1962, pp. 747–752.

[13] S. L. Hurst, D. M. Miller, and J. C. Muzio, “Spectral techniques in digital logic,” 1985.

[14] M. Turan Sönmez and R. Peralta, “The multiplicative complexity of Boolean functions on four and five variables,” in *Lightweight Cryptography for Security and Privacy*, Cham, 2015, pp. 21–33.

[15] J. Boyar, R. Peralta, and D. Pochuev, “On the multiplicative complexity of Boolean functions over the basis $(\wedge, \oplus, 1)$,” *Theoretical Computer Science*, vol. 235, no. 1, pp. 43–57, 2000.

[16] A. Kojevnikov, A. S. Kulikov, and G. Yaroslavtsev, “Finding efficient circuits using SAT-solvers,” in *Int’l Conf. on Theory and Applications of Satisfiability Testing*, 2009, pp. 32–44.

[17] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley, 2015.

[18] E. Testa, M. Soeken, L. Amarú, W. Haaswijk, and G. De Micheli, “Mapping monotone Boolean functions into majority,” *IEEE Trans. on Computers*, pp. 1–1, 2018.

[19] W. J. Haaswijk, “SAT-based exact synthesis for multi-level logic networks,” Ph.D. dissertation, EPFL, Lausanne, 2019.

[20] M. Soeken, W. Haaswijk, E. Testa, A. Mishchenko, L. Amarú, R. K. Brayton, and G. De Micheli, “Practical exact synthesis,” in *Design, Automation and Test in Europe*, 2018, pp. 309–314.

[21] W. Haaswijk, M. Soeken, A. Mishchenko, and G. De Micheli, “SAT-based exact synthesis: Encodings, topology families, and parallelism,” *IEEE Trans. on CAD of Integrated Circuits and Systems*, pp. 1–1, 2019.

[22] N. Éen, “Practical SAT - a tutorial on applied satisfiability solving,” 2007, slides of invited talk at FMCAD.

[23] M. Soeken, H. Riener, W. Haaswijk, E. Testa, B. Schmitt, G. Meuli, F. Mozafari, and G. De Micheli, “The EPFL logic synthesis libraries,” Nov. 2019, arXiv:1805.05121v2.

[24] N. Eén and N. Sörensson, “An extensible SAT-solver,” in *Int’l Conf. on Theory and Applications of Satisfiability Testing*, 2003, pp. 502–518.