The Complexity of Self-Dual Monotone 7-Input Functions

Eleonora Testa, Winston Haaswijk, Mathias Soeken, Giovanni De Micheli

Integrated Systems Laboratory, EPFL, Switzerland

***Abstract*—The study of the complexity of Boolean functions has recently found applications in logic synthesis and optimization algorithms, as for instance in logic rewriting. Previous works have focused on the minimum length of Boolean chains for functions up to 5 inputs, being represented in terms of 2- and 3-input operators. In this work, we study the complexity of self-dual monotone 7-input Boolean functions in terms of 3-input majority operators. We use enumeration-based and SAT-based exact methods to find (i) the minimum number of operators in the shortest formula of a Boolean function, and (ii) the minimum length of its Boolean chain. Different generalizations and restrictions of majority Boolean chains are considered to represent functions. For instance, we consider leafy Boolean chains in which each step has at least one fanin that is an input variable, or majority Boolean chains that use complemented edges.**

I. Introduction

The study of Boolean functions and in particular their complexity [1] plays a central role in logic synthesis and optimization. In this work, we focus on self-dual monotone functions and their representation as multi-level logic networks that only use the majority operator. In order to define the complexity of a logic Boolean function, we thus introduce the concept of majority Boolean chain.[1] Given a Boolean function $f(x_1, \ldots, x_n)$ of n input variables, a *majority Boolean chain* [2] is a way of representing functions defined as a sequence $(x_{n+1}, ..., x_{n+r})$, with the property that each step i in the chain combines 3 previous steps or inputs using a 3-input majority operator, such that for $n + 1 \leq i \leq n + r$:

$$\begin{aligned} x_i &= \langle x_{1(i)} x_{2(i)} x_{3(i)} \rangle \\ x_{1(i)} &< x_{2(i)} < x_{3(i)} < x_i \end{aligned} \tag{1}$$

A *leafy* majority Boolean chain is a majority Boolean chain for which $x_{1(i)} \leq n$ for all steps i. In other words, each step in such Boolean chain is constrained to have at least one previous step which is an input variable. The inherent complexity of a Boolean function is studied here according to two different measures, being (i) the *combinational complexity*, and (ii) the *length*. The combinational complexity of a Boolean function f, denoted as $C(f)$, is defined [2] as the minimum length r of the majority or leafy majority Boolean chain such that $x_{n+r} = f(x_1, \ldots, x_n)$. Note that the definition of Boolean chain allows for multiple fanouts: multiple distinct steps in the chain may refer to the same input or step x_i. On the other hand, the length $L(f)$ is defined as the number of 3-input majority operators (leafy or not leafy) in the shortest formula for f. It can be easily verified that $L(f) = C(f)$ for $n \leq 3$, and that $L(f) \geq C(f)$ [2].

Generally, the study of the complexity of Boolean functions deals with finding some upper bounds [3], [4] or lower bounds [5] over a set of primitives. In our case, we are instead concerned with finding exact numbers both for the length and the combinational complexity over majority operators. Similar problems have already found application in the logic synthesis and optimization field [6]–[8], as, for instance, logic rewriting algorithms optimize logic networks by replacing small subnetworks with optimized Boolean chains [6], [7]. In [2], the complexity for all 4- and 5-input Boolean functions in terms of 2-input Boolean operators have been studied, while 3-input Boolean operators have been used in [9]. Having exact numbers for the combinational complexity of some small functions can help to find tighter upper bounds for larger functions by using arguments from Boolean decomposition. Recently, the complexity for the majority-of-n-input (majority-n, [10], [11]) functions over majority Boolean chains has been considered in [12]. The work in [12] uses a BDD-based method to find the combinational complexity of monotone functions, with particular stress on the use of leafy majority Boolean chains. It has been demonstrated that for the majority-n functions up to 7 inputs, the combinational complexity is invariant when considering majority or leafy majority Boolean chains. Further, the work in [12] proves that all majority-n functions can recursively be constructed from self-dual monotone functions. For this reason, the study of minimum Boolean chains for self-dual monotone functions plays a key role in finding new upper bound for the complexity of majority-n functions.

In this paper, partly motivated by the recent results in [12], we enumerate all self-dual monotone 7-input functions and classify them according to their length and their combinational complexity over both majority and leafy majority Boolean chains. There are 1,422,564 such functions, distributed over 716 classes according to input permutation (P-equivalence). The classification over majority Boolean chains has first been presented in [2], both according to $C(f)$ and $L(f)$. Here, we use our own implementation of the algorithm from [2] to enumerate all 1,422,564 functions and to reproduce their classification according to their length. On the other hand, we propose our own strategy to compute the combinational

[1] In [2] Knuth called them *median* Boolean chains.

complexity for all 716 classes, which is a SAT-based exact synthesis method. By comparing our results with the ones in [2], we confirm that, when using majority Boolean chains, the largest $L(f)$ is 11, while the largest $C(f)$ is 8. Moreover, we also present results on leafy majority Boolean chains, focusing on the differences with respect to the majority case. We demonstrate an increase in the maximum length, while we show that the worst combinational complexity remains unaffected. However, the combinational complexity does not remain unchanged for all functions, i.e., 40 functions have increased combinational complexity when built using leafy majority chains. As a last result, we show that inverters have an impact on the minimum Boolean chain of self-dual monotone functions. It is theoretically known that inverters can decrease complexity even in monotone functions [13], here we demonstrate and give concrete examples for the complexity of 7-input functions.

II. Preliminaries

This work focuses on the complexity of self-dual monotone functions. While the concept of complexity has already been defined, we give here some notions about self-dual monotone functions. Further, we also introduce input permutation classes and SAT-based exact synthesis, as they will be used in the following discussion.

A. Self-Dual Monotone Functions

A Boolean function $f(x_1, x_2, \ldots, x_n)$ is *monotone* if and only if $f(x) \leq f(y)$ whenever $x \subseteq y$. This means that for the bitstrings $x = x_1 \ldots x_m$ and $y = y_1 \ldots y_m$, it follows $x_i \leq y_i$ for all i. A monotone Boolean function can be expressed using only AND ($\wedge$) and OR ($\vee$) operators, without using complementation [2]. Being $\langle xyz \rangle$ the majority-of-three-input (majority-3) operator and considering that $\langle x0y \rangle = x \wedge y$ and $\langle x1y \rangle = x \vee y$, it follows that any monotone Boolean functions can be written using only majority-3 operators and constants, without using inverters.

A Boolean function $f(x_1, x_2, \ldots, x_n)$ is *self-dual* if it satisfies

$$\bar{f}(x_1, x_2, \ldots, x_n) = f(\bar{x}_1, \bar{x}_2, \ldots, \bar{x}_n)$$

where the $\bar{}$ is used to represent the complementation. In other words, it states that negations can be freely propagated from the inputs to the output. A monotone Boolean function expressed over $\wedge$ and $\vee$ operators is self-dual if the symbols $\wedge$ and $\vee$ can be interchanged without affecting the value of the function. The majority-3 is an example of self-dual function.

Another useful definition is the one of *normal* Boolean function, also called *0-preserving* function [14]. A Boolean function $f(x_1, x_2, \ldots, x_n)$ is normal if

$$f(0, 0, \ldots, 0) = 0$$

More generally, a Boolean chain is normal if and only if all its operators are normal.

The majority operator is monotone, self-dual and normal. When considering functions over 7 inputs, there are 1,422,564 self-dual monotone functions. A majority chain always computes a monotone and self-dual function. Also, for each monotone and self-dual function, there exists a majority Boolean chain that computes it.

B. Input Permutation Equivalence

Two Boolean functions are *P-equivalent* if they are equivalent up to permutation of their inputs. As an example, functions $f = a \wedge \bar{b}$ and $g = \bar{a} \wedge b$ are equal if we swap the input a with input b and thus are said to be P-equivalent. P-equivalence is used to group functions into P-equivalent classes which consist of all functions equivalent up to permutation of the inputs. A class for a function f is denoted here as $[f]$. When two function f and g belong to the same class, i.e., $g \in [f]$, they are P-equivalent. Each class can be represented using the *canonical representative* $\hat{f}$ of the class, which is the function $f \in [f]$ that has the truth table corresponding to the smallest integer value. More details about efficient exact and heuristic algorithms for P classification can be found in [2].

In this work, we are interested in studying the complexity of Boolean functions. A key property is that all P-equivalent functions, i.e., functions which are in the same class, have the same combinational complexity $C(f)$ and the same length $L(f)$, which means that when $g \in [f]$, $C(f) = C(g)$ and $L(f) = L(g)$. All self-dual monotone 7-input Boolean functions can be grouped into 716 classes according to the permutation of their inputs. The key idea is that, thanks to P-equivalence, we can study the complexity of self-dual monotone 7-input functions by looking at $C(f)$ and $L(f)$ for only 716 functions, instead of for all 1,422,564 functions. This is preferable, since the number of P classes is significantly smaller than the number of functions. Thanks to this property, P-equivalence and its generalization to NPN [15] are largely used in logic synthesis, for instance in logic rewriting and exact synthesis [7], [16].

C. SAT-based Exact Synthesis

Exact synthesis is the problem of finding optimum Boolean chains that represent given Boolean functions and respect given constraints [17], e.g., in the type of operators. In this paper, we concentrate on size optimum results, but extensions that aim at depth optimum have also been considered [18], [19]. The first example of SAT-based exact synthesis can be found in [20], and successive analyses and improvements have been considered in [21], [22]. The key idea behind all these methods is to verify if it is possible to realize a function f with a Boolean chain of size r, using a sequence of SAT formulas, i.e., encoding the problem in *Conjunctive Normal Form* (CNF). The size r is at first initialize at 0, or at some given value, and at each loop it is increased if a solution is not found, i.e., the result is UNSAT. If the results is SAT, an optimum size Boolean chain can be extracted from the obtained solution. In this paper, we focus on SAT-based exact synthesis to study the combinational complexity of Boolean functions. It has already been proven in [9] that SAT-based exact synthesis methods can efficiently be employed to address

this task. Indeed, the combinational complexity of Boolean functions can be extracted directly from an optimum size Boolean chain, as it corresponds to the number of steps in the optimum solution.

We refer the interested reader to [17], [23] for a more detailed review on exact synthesis.

III. Self-Dual Monotone 7-input Functions Classification

In this section, we describe the algorithms used to enumerate and classify self-dual monotone 7-input functions with respect to their length and their combinational complexity over majority operators. First, we illustrate the implementation of an algorithm to classify functions according to their $L(f)$. The same algorithm allowed us to obtain the truth table for all the 716 self-dual monotone 7-input functions. Then, we propose a SAT-based exact synthesis method to classify the obtained 7-input functions according to their combinational complexity. Both the majority and the leafy majority Boolean chains are considered.

A. Length L(f): Algorithm L

This section describes an exact algorithm to evaluate the length of self-dual monotone functions in terms of majority operators. We start by describing the algorithm for majority operators, which is inspired by the one in [2]. In particular, it is a 3-input majority-based version of "Algorithm L" presented in Section 7.1.2 of [2]. Finally, we address the changes necessary to make the algorithm work for the leafy majority case.

The idea is to compute the length of all 1,422,564 functions by enumerating all functions with length $0, 1, 2, \ldots r$. Each function f with $L(f) = r$ is built as $\langle ghi \rangle$, where g, h, and i are three functions already enumerated and whose sum of lengths is equal to $r - 1$. This is practically obtained by enumerating and storing all self-dual monotone functions from previous lengths. As we are using only majority operators, each function built using previously obtained self-dual monotone functions is self-dual and monotone and can be added to the list itself, if not already there. The algorithm is also called [2] *Find normal length* as it works only on normal Boolean chains.

The pseudocode is depicted in Alg. 1. The input is a vector containing the truth tables for the 7-input variables. The first part of the procedure initializes the count to the total number of functions (line 1 in Alg. 1) and the length for the input variables to 0 (lines [3–5] in Alg. 1). The table $function\_to\_length$ maps each function, represented as truth table, to its length. The algorithm's outer loop takes into account the total number of functions, and it ends once the counter hits 0. The inner while loop considers different values for j, k, and l, which are the lengths of functions g, h, and i, respectively. Function f is computed as the majority of functions g, h, and i, using all combinations over the three functions whose lengths sum is equal to $current\_length - 1$ (lines [21–27] of Alg. 1). The length of f is equal to $current\_length$, further, as all functions from previous steps

Algorithm 1 Algorithm L to compute $L(f)$

Input: Truth tables of input variables x_k
Output: $function\_to\_length$

```
 1: count ← 1,422,564
 2: current_length = 0
 3: for each variable ∈ x_k do
 4:     function_to_length(variable) ← current_length
 5: end for
 6: while count > 0 do
 7:     current_length = current_length + 1
 8:     j, k = 0
 9:     l = current_length − 1
10:     while l > 0 do
11:         for each combination of g, h, i ∈ function_to_length with
    length j, k, l respectively do
12:             f ← ⟨ghi⟩
13:             if f ∉ function_to_length then
14:                 function_to_length(f) ← current_length
15:                 count ← count − 1
16:                 if count = 0 then
17:                     return function_to_length
18:                 end if
19:             end if
20:         end for
21:         if j + k = current_length − 1 then
22:             j = j + 1
23:             k = j
24:         else
25:             k = k + 1
26:         end if
27:         l = current_length − 1 − j − k
28:     end while
29: end while
30: return function_to_length
```

are self-dual and monotone, it follows also function f is self-dual and monotone. Thus, if the function is not already present in the $function\_to\_length$ map, it is saved in the map together with its length. The algorithm ends when all functions have been computed, and it returns all functions and their corresponding lengths.

In practice, few changes to Alg. 1 allowed us to save not only all 1,422,564 self-dual monotone 7-input functions, but also the 716 representatives of the P-classes. The for loop in line 11 of Alg. 1 consists practically of 3 loops over g, h, and i. The algorithm consists thus of 5 nested loops, whose complexity grows with $current\_length$. Experimentally, to save runtime, we used the map $function\_to\_length$ to search for the existence of function f (line 13 of Alg. 1), while the loops over g, h, and i have been performed using vectors of truth tables. Further runtime has been saved by using the commutativity property of majority, thus by disregarding all combinations of g, h, and i already considered in different orders.

Alg. 1 works over majorities operators. We also designed a second algorithm to work on leafy majority formulas. To constraint Alg. 1 with leafy operators, we constrained variable j to be always equal to 0, thus to consider only input variables. It means function g loops over all input variables x_k, while h and i can consider functions with larger lengths. The rest of the algorithm and the general idea remain the same as Alg. 1.

Algorithm 2 SAT-based exact method to compute $C(f)$

```
1: function synthesize_maj(f, r, S)
2:     S ← Restart SATSolver
3:     CreateVariables(S, f, r)
4:     AddMainClause(S, f, r)
5:     AddFanInClauses(S, r)
6:     AddOtherClauses(S, f, r)
7:     if Solve(S) then return Majority Boolean chain
8:     else
9:         return synthesize_maj(f, r + 1, S)
10:    end if
11: end function
```

B. Combinational Complexity C(f): Exact Synthesis

In this section, we present a SAT-based exact method to evaluate the combinational complexity of self-dual monotone 7-input functions in terms of majority operators. The combinational complexity is invariant under input permutation, thus we apply the SAT-based exact method only to the 716 P-classes, whose number is significantly smaller than the total number of functions. The truth tables for the 716 functions have been obtained by using Algorithm L described in Section III-A. As in the previous section, we first present the general method that works over majority Boolean chains, we then describe the differences to the implementation in order to consider leafy majority Boolean chains.

The implemented exact synthesis algorithm starts by trying to find a Boolean chain for function f using $r = 0$. If a solution exists with r steps, the algorithm returns a majority Boolean chain that implements function f, otherwise it searches a solution with larger size $(r + 1)$. The algorithm increases the number of steps until a solution is found.

This idea is described in the recursive procedure depicted in Alg. 2, which is applied to each function f separately. The inputs of the algorithm are (i) the function specification f (for instance, given as truth table), (ii) the number of steps r, and (iii) the SAT solver S. For the majority Boolean chain, we used the same encoding presented in [8], [24]. This is an extension that works over 3-input Boolean operators of the encoding first proposed by Knuth [22] for 2-input Boolean operators. In our case, the operations of each step are limited to the majority operator, without allowing inversions. The clauses are the same as discussed in [22]. The main clause is the one which encodes the truth table of the circuit, while the fanin clause assures that each step has exactly 3 distinct inputs. $\mathrm{AddOtherClauses}$ consists of both necessary and additional clauses proposed in [19], which can be added to reduce the solving time of the SAT solver. More details about both clauses formalization and additional clauses can be found in [19], [22], [23]. Alg. 2 is first applied to each function using $r = 0$, r is then increased until a solution is found. It is worth noting that this method allows to not only count the number of functions for each combinational complexity, but also to obtain their implementation in terms of majority operators.

In order to constrain Alg. 2 to work only with leafy majority Boolean chains, we changed the $\mathrm{AddFanInClauses}$ in order to constrain the first input of each step to be one of the input variables. The fanin of each step i is encoded in the *select variable* s_{ijkl}, which is true if steps x_j, x_k and x_l are the children of step x_i. In the leafy case, the fanin clause is changed to ensure $x_j \leq n$. The rest of the algorithm remains the same as Alg. 2.

IV. Experimental Results

In this section, we describe the experimental results both for the length and the combinational complexity of self-dual monotone 7-input functions. First, we present our results for the majority case. Then, we give a comparison between majority and leafy majority Boolean chains.

We have implemented the proposed algorithms using the open source EPFL Logic Synthesis Libraries [25]. Algorithm L has been implemented using the truth table library *kitty*.[2] The SAT-based exact method has been implemented using the exact logic synthesis library *percy*.[3] We have used the "maj_encoder" and the `Glucose` SAT solver [26], [27]. All the experiments have been carried out on Intel Xeon E5-2680 CPU with 2.5 GHz and with 256 GB of main memory.

Regarding the majority Boolean chains, the results of classification have already been presented in [2]. Our results have been obtained using Algorithm L (Alg. 1) and are shown in the first part of Table I. The first three columns of Table I show both the number of classes and the number of functions for each length. The largest length is equal to 11 (in agreement with the results from [2]). Consider as an example function $f = \langle x_1 x_2 x_3 x_4 x_5 x_6 x_7 \rangle$, which is the majority-of-seven-inputs (majority-7). Its $L(f)$ has been demonstrated both here and in [2] equal to 8, given by:

$$\langle x_1 \langle x_2 \langle x_3 x_4 x_5 \rangle \langle x_3 x_6 x_7 \rangle \rangle \langle x_4 \langle x_2 x_6 x_7 \rangle \langle x_3 x_5 \langle x_5 x_6 x_7 \rangle \rangle \rangle \rangle \tag{2}$$

While for the length we have implemented the same algorithm presented in [2] to obtain our results, for the combinational complexity we have used an alternative approach, i.e., SAT-based exact synthesis. The results are obtained applying Alg. 2 on all 716 functions, with a total runtime of 4038 seconds and 2997 seconds for the majority and leafy majority Boolean chains, respectively. The runtime for the method presented in [2] on majority Boolean chains is 6894 seconds. Note also that while Knuth's algorithm only counts the number of functions for each class, in our case extra memory and runtime are necessary in order to also get the majority networks implementations.[4] Our results for the combinational complexity are shown in columns 4 and 5 of Table I. As we used a SAT-based exact synthesis method, we have computed the combinational complexity only for the 716 classes. The maximum combinational complexity is 8 (in agreement with the results in Table II). The shortest chain for function $f = \langle x_1 x_2 x_3 x_4 x_5 x_6 x_7 \rangle$ needs 7 steps and it is given by:

[2]Available at: https://github.com/msoeken/kitty

[3]Available at: https://github.com/whaaswijk/percy

[4]Available at: https://github.com/eletesta/7input_classification

TABLE I: $L(f)$ and $C(f)$ for all 716 self-dual monotone 7-input classes

Majority Computation					Leafy Majority Computation				
$L(f)$	Classes	Functions	$C(f)$	Classes	$L_{\text{leafy}}(f)$	Classes	Functions	$C_{\text{leafy}}(f)$	Classes
0	1	7	0	1	0	1	7	0	1
1	1	35	1	1	1	1	35	1	1
2	2	350	2	2	2	2	350	2	2
3	8	3745	3	9	3	8	3745	3	9
4	38	35203	4	48	4	35	33628	4	45
5	139	270830	5	201	5	123	233660	5	191
6	313	699377	**6**	**354**	6	272	600887	6	347
7	176	367542	**7**	**98**	7	210	449673	7	114
8	34	43135	8	2	8	50	84519	8	6
9	3	2310	9	0	9	12	14770	9	0
10	0	0	10	0	10	1	1260	10	0
11	1	30	11	0	11	0	0	11	0
12	0	0	12	0	12	1	30	12	0

$$x_8 = \langle x_1 x_3 x_4 \rangle, \quad x_9 = \langle x_6 x_7 x_8 \rangle, \quad x_{10} = \langle x_5 x_8 x_9 \rangle$$
$$x_{11} = \langle x_5 x_6 x_7 \rangle, \quad x_{12} = \langle x_1 x_3 x_{11} \rangle, \quad x_{13} = \langle x_4 x_{11} x_{12} \rangle$$
$$x_{14} = \langle x_2 x_{10} x_{13} \rangle \tag{3}$$

This last result matches the one demonstrated both in [2] and [12].

The second half of Table I shows the leafy majority results. The worst $L(f)$ is increased to 12, while the largest combinational complexity is still 8. As a general trend, functions need more steps when the majority operators are constrained to have at least one primary input. For example, note that the number of chains with 8 steps is increased from 2 to 6. By comparing the majority results with the leafy ones, we can conclude that:

1) The class with the largest length is the same both in majority and leafy majority computation, and it is the function with truth table (in hexadecimal form):

`fefefeaafeccf080fef0cc80aa808080`

Its combinational complexity in both majority and leafy is equal to 8 steps, obtained using the Boolean chain given by:

$$x_8 = \langle x_5 x_6 x_7 \rangle, \quad x_9 = \langle x_2 x_3 x_5 \rangle, \quad x_{10} = \langle x_1 x_8 x_9 \rangle$$
$$x_{11} = \langle x_3 x_7 x_{10} \rangle, \quad x_{12} = \langle x_2 x_6 x_{10} \rangle, \quad x_{13} = \langle x_4 x_{11} x_{12} \rangle$$
$$x_{14} = \langle x_1 x_4 x_{13} \rangle, \quad x_{15} = \langle x_5 x_{13} x_{14} \rangle \tag{4}$$

2) Regarding the length, 98 functions out of 716 have different length when comparing majority with leafy majority formulas. The maximum difference in length is equal to 3. This is true for functions:

`feeaeaeaeeeaaaaa0faaaaa88a8a8a880`
`feeeeeeaeeeaecc0fcc8a888a8888880`
`feeaece0faeaa8a0faeaa8a0f8c8a880`

3) For the combinational complexity, 40 functions out of 716 have a different combinational complexity when comparing majority with leafy majority Boolean chains. In this case, the difference is equal to 1 for all the functions.

4) Up to 3 steps, the results are the same both for majority and leafy majority case. This was expected, if we consider that we are not allowing steps in the Boolean chains to have two equal inputs.

TABLE II: $L(f)$ and $C(f)$ for all self-dual monotone 7-input functions taken from Section 7.1.2 in [2]

$L(f)$	Classes	Functions	$C(f)$	Classes	Functions
0	1	7	0	1	7
1	1	35	1	1	35
2	2	350	2	2	350
3	8	3745	3	9	3885
4	38	35203	4	48	42483
5	139	270830	5	201	406945
6	313	699377	**6**	**353**	798686
7	176	367542	**7**	**99**	169891
8	34	43135	8	2	282
9	3	2310	9	0	0
10	0	0	10	0	0
11	1	30	11	0	0

5) For the majority-7, the length in the majority case is equal to 8 and its complexity is 7. The combinational complexity in the leafy case is unchanged (w.r.t. the majority case) and equal to 7, as it has been demonstrated in [12] and seen in (3). Further, we prove that also its length is unchanged when considering leafy majority operators.

To further stress the difference between majority and leafy majority Boolean chains, consider function `feeaeaeaeeeaaaaa0faaaaa88a8a8a880` as an example. When considering its length, the difference between majority and leafy majority is equal to 3. i.e., $L(f) = 4$ and $L_{\text{leafy}}(f) = 7$. The smallest majority Boolean chain needs 4 steps:

$$x_8 = \langle x_1 x_3 x_6 \rangle, \quad x_9 = \langle x_1 x_4 x_5 \rangle,$$
$$x_{10} = \langle x_1 x_2 x_7 \rangle, \quad x_{11} = \langle x_8 x_9 x_{10} \rangle \tag{5}$$

In the leafy case, the same Boolean chain cannot be used, as the last step x_{11} is the majority of three majority operators in the previous steps. The function needs 5 steps in the leafy majority Boolean chain, given by:

$$x_8 = \langle x_1 x_3 x_6 \rangle, \quad x_9 = \langle x_1 x_2 x_7 \rangle,$$
$$x_{10} = \langle x_1 x_8 x_9 \rangle, \quad x_{11} = \langle x_5 x_8 x_9 \rangle, \quad x_{12} = \langle x_4 x_{10} x_{11} \rangle \tag{6}$$

TABLE III: $C(f)$ for 715 self-dual monotone 7-input functions over majority Boolean chains with inverters

$C(f)$	Classes
0	1
1	1
2	2
3	9
4	48
5	**210**
6	**374**
7	**70**
8	**0**

As a last result, it is worth mentioning that our SAT-based exact synthesis method was able to demonstrate better combinational complexity for one of the 716 classes with respect to the original results in [2]. The discrepancy was due to a bug in the original version of the algorithm in [2], which has been found and solved as a result of this work. The original results obtained in [2] are listed in Table II. In Table II, the number of functions with complexity 6 and 7 respectively are not in agreement with our results in Table I (see highlighted numbers). In particular, for one function, the SAT-based synthesis method was generating a smaller combinational complexity.

The discrepancy has been discussed with the author, and corrected in the most recent version of [2].

V. Conclusions and Future Works

We study the complexity of self-dual monotone 7-input functions in terms of 3-input majority operators. Finding minimum chains is not only of interest from a theoretical point of view, but it also has practical application. For example, they can be used in logic optimization and technology mapping. Our method uses both state-of-the-art algorithms and exact synthesis, based on P classification, to compute the complexity of Boolean functions according to their (i) length, and (ii) combinational complexity. We study the complexity in terms of both majority and leafy majority Boolean chains, in which majority operators are constrained to have at least one primary input. In future work, we plan to further study self-dual monotone 7-input functions in terms of different majority Boolean chains. For instance, majority Boolean chains which allow the use of constant signals, i.e., and/or operators, could be considered. Also limited fan-out configurations in which the fan-out of each node is constrained to be smaller than a given value can be taken into account. A key role for future developments is the use of inverters. It is already known from complexity theory that the inverters could help to reduce asymptotic bounds of size and depth of logic networks for monotone functions [13]. In the future, we plan studying the effect of inversions both on the length and the combinational complexity of self-dual monotone 7-input functions. Also for this case, leafy majority Boolean chains could be considered. Towards this direction, we show in Table III some preliminary results for the combinational complexity of the 716 self-dual monotone Boolean functions when using majority Boolean chains and inverters. The results are obtained using a SAT-based exact synthesis method for *Majority Inverter Graphs* [28], as presented in [8]. This method is similar to the one presented in Alg. 2. At the time of submission, 715 functions out of 716 were synthesized by our exact synthesis method. Even if not complete, Table III shows that the combinational complexity of 39 functions is decreased thanks to the use of inverters. For instance, consider function `feeeeee8faa8aaa0faaaeaa0e8888880`. The combinational complexity over majority Boolean chain (with no inversion) is equal to 7, given by the Boolean chain:

$$\begin{gathered} x_8 = \langle x_1 x_3 x_4 \rangle, \quad x_9 = \langle x_2 x_6 x_8 \rangle, \\ x_{10} = \langle x_7 x_8 x_9 \rangle, \quad x_{11} = \langle x_3 x_4 x_{10} \rangle, \\ x_{12} = \langle x_5 x_6 x_{11} \rangle, \;\; x_{13} = \langle x_2 x_5 x_{12} \rangle, \quad x_{14} = \langle x_1 x_{10} x_{13} \rangle \end{gathered} \tag{7}$$

The same function can be synthesized using only 6 steps, if we allow inverters:

$$\begin{aligned} &x_8 = \langle x_1 \bar{x}_5 x_7 \rangle, \quad x_9 = \langle x_1 x_5 \bar{x}_7 \rangle, \quad x_{10} = \langle x_3 x_4 x_8 \rangle, \\ &x_{11} = \langle x_2 x_6 x_9 \rangle \quad x_{12} = \langle x_1 x_5 \bar{x}_9 \rangle, \quad x_{13} = \langle x_{10} x_{11} x_{12} \rangle \end{aligned} \tag{8}$$

Acknowledgments

We would like to offer special thanks to Professor Donald E. Knuth for sharing his implementation of Algorithm L and for his valuable help and assistance. We further acknowledge fruitful discussions with Alan Mishchenko.

This research was supported by the Swiss National Science Foundation (200021-169084 MAJesty), by H2020- ERC-2014-ADG 669354 CyberCare, and by the EPFL Open Science Fund.

References

[1] I. Wegener, *The complexity of Boolean functions*. John Wiley, 1987.

[2] D. E. Knuth, *The Art of Computer Programming, Volume 4A*. Addison-Wesley, 2011.

[3] W. Hesse, E. Allender, and D. A. M. Barrington, "Uniform constant-depth threshold circuits for division and iterated multiplication," *Journal of Computer and System Sciences*, vol. 65, no. 4, pp. 695–716, 2002.

[4] M. Goldmann, J. Håstad, and A. Razborov, "Majority gates vs. general weighted threshold gates," *Computational Complexity*, vol. 2, no. 4, pp. 277–300, 1992.

[5] W. J. Paul, "A 2.5n-lower bound on the combinational complexity of Boolean functions," *SIAM Journal on Computing*, vol. 6, no. 3, pp. 427–443, 1977.

[6] M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Optimizing majority-inverter graphs with functional hashing," in *Design, Automation and Test in Europe*, 2016, pp. 1030–1035.

[7] W. Haaswijk, M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "A novel basis for logic optimization," in *Asia and South Pacific Design Automation Conference*, 2017, pp. 151–156.

[8] M. Soeken, L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Exact synthesis of majority-inverter graphs and its applications," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 36, no. 11, pp. 1842–1855, 2017.

[9] W. Haaswijk, E. Testa, M. Soeken, and G. De Micheli, "Classifying functions with exact synthesis," in *Int'l Symp. on Multiple-Valued Logic*, 2017, pp. 272–277.

[10] S. Amarel, G. E. Cooke, and R. O. Winder, "Majority gate networks," *IEEE Trans. Electronic Computers*, vol. 13, no. 1, pp. 4–13, 1964.

[11] A. Neutzling, F. S. Marranghello, J. M. Matos, A. Reis, and R. P. Ribas, "Maj-n logic synthesis for emerging technology," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2019.

[12] E. Testa, M. Soeken, L. Amaru, W. Haaswijk, and G. De Micheli, "Mapping monotone boolean functions into majority," *IEEE Trans. on Computers*, 2018.

[13] T. Hofmeister, "The power of negative thinking in constructing threshold circuits for addition," in *Proceedings of the Seventh Annual Structure in Complexity Theory Conference*, 1992, pp. 20–26.

[14] E. L. Post, *The Two-Valued Iterative Systems of Mathematical Logic*. Princeton University Press, 2016, vol. 5.

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

[16] 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.

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

[18] L. Amarú, M. Soeken, P. Vuillod, J. Luo, A. Mishchenko, P.-E. Gaillardon, J. Olson, R. Brayton, and G. De Micheli, "Enabling exact delay synthesis," in *Int'l Conf. on Computer-Aided Design*, 2017, pp. 352–359.

[19] M. Soeken, G. De Micheli, and A. Mishchenko, "Busy man's synthesis: Combinational delay optimization with SAT," in *Design, Automation and Test in Europe*, 2017, pp. 830–835.

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

[21] 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.

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

[23] 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*, 2019.

[24] E. Testa, M. Soeken, O. Zografos, F. Catthoor, and G. De Micheli, "Exact synthesis for logic synthesis applications with complex constraints," *Int'l Workshop on Logic and Synthesis*, 2017.

[25] M. Soeken, H. Riener, W. Haaswijk, and G. De Micheli, "The EPFL logic synthesis libraries," may 2018, arXiv:1805.05121.

[26] G. Audemard and L. Simon, "Predicting learnt clauses quality in modern sat solvers," in *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, ser. IJCAI'09, 2009, pp. 399–404.

[27] G. Audemard and L. Simon, "Glucose and Syrup in the SAT Race 2015," in *Reports on the SAT 2015 Competition*, 2015.

[28] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Design Automation Conference*, 2014, pp. 194:1–194:6.