Enumeration of Minimum Fanout-Free Circuit Structures

Siang-Yun Lee Jie-Hong R. Jiang
Department of Electrical Engineering
National Taiwan University

Alan Mishchenko Robert Brayton
Department of EECS
University of California, Berkeley

ABSTRACT

The paper focuses on logic synthesis of minimum-size circuits for small Boolean functions using structural enumeration. Known results for completely-specified functions up to five inputs are reproduced and independently verified. Additionally, for the first time, the results are presented on the number of *structurally different* minimum circuits for each NPN class of Boolean functions up to five inputs. The actual structures are made publicly available with potential applications in FPGA architecture exploration, peephole optimization by circuit rewriting, and approximate logic synthesis.

1 INTRODUCTION

Logic synthesis transforms functionality into structure. The input is a functional representation given in some form, and the output is a structural network composed of some primitives. For example, in the case of completely-specified Boolean functions, the functional representation can be a truth table, a sum of products, or a binary decision diagram, while the output can be a network containing gates from a technology library or nodes with arbitrary functions with bounded support size.

Quality of results produced by logic synthesis is evaluated based on how well the resulting network meets practical requirements, which includes minimizing parameters such as circuit size (the total number of nodes), depth (the number of nodes on a longest path from an input to an output), power dissipation (which depends on what gates are used), the number and topology of internal connections, testability, verifiability, etc.

Small network size is typically the primary metric of quality or at least an important concern. Indeed, who would choose to use more primitives if the function can be completely and correctly implemented with fewer of them? Given the obvious preference for minimum-sized networks, it is ironic that claims of exact minimality are very rare in logic synthesis.

The exact results are known for specific function types, including some of those frequently appearing in practical designs, in particular, read-polarity-once [16] and disjoint-support decomposable functions [4][6]. However, for general Boolean functions, the exactness remains elusive. It is known, for example, what is the minimum circuit composed of two-input gates for all functions up to 5 inputs and for some larger functions [1]. However, even for functions with as few as 6 inputs, with truth tables containing only 64 bits, it is not known what is the largest size of a minimum two-input-gate circuit for the special classes mentioned above. Moreover, if a 6-input function cannot be implemented with fewer than 10 gates, its minimum circuit size is also typically unknown.

Besides the problem of finding one minimum circuit for a given function, it is often useful to have access to all structurally different minimum or near-minimum circuits. The following applications may benefit from the circuit library: peephole optimization, FPGA architecture evaluation, and approximate logic synthesis. A more detailed discussion of these applications is found in Section 5.

The contribution of this paper is three-fold:

(1) It reproduces and independently verifies minimum circuit sizes for Boolean functions up to five inputs. These results were first published by Donald Knuth in the recent volume of his book, *The Art of Computer Programming* [1].
(2) It presents, for the first time, the number of structurally different minimum circuits for all functions up to five inputs.
(3) It generates the library of all minimum circuit structures of all Boolean functions up to five inputs and makes it publicly available in the AIG format.

The rest of the paper is organized as follows: After preliminaries are given in Section 2, the bottom-up enumeration and library construction algorithms are described in Section 3. Experimental results are shown in Section 4, and practical applications are discussed in Section 5. Section 6 concludes the paper.

2 PRELIMINARIES

A *Boolean variable* is a variable taking value on the *Boolean space* $\mathbb{B} = \{0, 1\}$. An n-input *Boolean function* f is a relation from n *input* Boolean variables to an *output* Boolean variable, or $f : \mathbb{B}^n \to \mathbb{B}$. Boolean functions can be represented with its *truth table* by placing the output values in the order of the corresponding input values. In this paper, a truth table of an n-input function has 2^n bits, where the least significant bit corresponds to the input value $00\ldots00$, the second least significant bit corresponds to the input value $00\ldots01$, and the most significant bit corresponds to the input value $11\ldots11$.

Two Boolean functions are said to be in the same *NPN class* if we can get one of them from the other by negating and permuting any input(s) and/or adding a negation at the output. We can always choose a *representative* function to represent its NPN class. In this paper, we choose the one with the smallest value of truth table when the truth table is interpreted as a binary number.

In this paper, the *circuit* of a function is composed of any two-input nodes. Without limiting generality, it can be assumed that the circuit is composed of two-input AND-/XOR-gates, and optional inverters with no cost on the connecting wires.

There are several types of minimum circuits for Boolean function introduced in *The Art of Computer Programming* [1]:

- The *combinational complexity* $C(f)$ of a function f, is the number of two-input nodes in the smallest circuit representing the function. Note that the internal nodes of the circuit can have multiple fanouts.
- The *length* $L(f)$ of a function f is the number of binary operators in its shortest formula. It is the same as the number of nodes of the minimum-node leaf-DAG, or the minimum possible number of nodes in a circuit without gate sharing.
- The *depth* $D(f)$ of a function f is the minimum number of levels among all circuits representing f.

3 MINIMUM CIRCUIT ENUMERATION

In this work, we focus on enumerating the minimum circuits in terms of $L(f)$.

3.1 Enumerating $L(f)$ and $D(f)$

First, we show how the $L(f)$ of each NPN class is found. For circuits without logic sharing, the problem of finding minimum circuits has *optimal substructure*. That is, if a circuit is minimum, the two subcircuits of the fanins of the topmost gate must also be minimum, because otherwise, we can replace the subcircuit to get an even smaller circuit, which contradicts our assumption that the original circuit is minimum. This approach is similar to that in [3]. However, in this paper we focus on the complete enumeration of circuit structures rather than on building a general synthesis framework.

With this property, we can enumerate all the minimum circuits of NPN functions having a given number of input variables using a bottom-up strategy. In this algorithm, we combine two minimum circuits for n_1 and n_2 gates with an additional gate to get a larger circuit composed of $n_1 + n_2 + 1$ gates. We compute the function of the resulting circuit and check if we already computed a smaller circuit for any of the functions in its NPN class. If not, this circuit is minimum. We record it with the cost $L(f) = n_1 + n_2 + 1$. Note that the two NPN classes with 0 gates are constant zero whose truth table has all 0s, and the buffer class whose truth table has value 1 whenever a certain input variable has value 1.

The pseudo code of the algorithm is shown in Figure 1 where *Table* is a look-up table with 2^{2^I} bits, each bit representing an I-input function. We use this table to keep track of the functions we have enumerated and the functions belonging to the NPN class of these functions. The procedure *InitializeTable* simply creates an array of 2^{2^I} bits and initializes all the bits to 0. Once a function is enumerated with $L(f) = n$, all the functions in the same NPN class are known to have $L(f) = n$ and are also marked in the table. Hence the procedure *Mark* performs permuting and negating of the original function to get all functions belonging to the same NPN class and mark all of them in *Table*. In contrast, procedure *Record* records in the library that all the functions in an NPN class, represented by a single function, have the same $L(f)$.

Variable n_{max} in line `07` can be set by the user to limit the runtime; otherwise, it is `INT_MAX` to perform complete enumeration. In the later case, the algorithm stops when n does not yield new functions, compared to those found with $n - 1$.

The enumeration of $D(f)$ can be done by changing the code in line `08` of Figure 1, by setting j to be $n - 1$ and making i go from 0 to $n - 1$. Now n represents the number of levels of the circuit, and minimum-level circuits must be composed of the topmost gate with one fanin subcircuit of $n - 1$ levels and the other subcircuit with $\leq n - 1$ levels.

```
***FunctionEnumeration***
**input**: The number of input variables $I$
**output**: $L(f)$ of each NPN class with up to $I$ inputs
**begin**
01 Library := empty vector
02 Table := InitializeTable($I$)
03 Library.Record(CONST_ZERO, 0)
04 Table.Mark(CONST_ZERO)
05 Library.Record(INPUT_VARIABLE, 0)
06 Table.Mark(INPUT_VARIABLE)
07 for $n = 1 : n_{max}$
08     for $i = 0 : \lceil n/2 \rceil - 1, j = n - 1 : n - \lceil n/2 \rceil$ // n=i+j+1
09         foreach NPN class $c_i$ in Library with $i$ nodes
10             foreach function $f_i$ in class $c_i$
11                 foreach NPN class $c_j$ in Library with $j$ nodes
12                     $f_j$ := representative function in class $c_j$
13                     if not Table.Marked($f_i \wedge f_j$)
14                         Library.Record($f_i \wedge f_j$, $n$)
15                         Table.Mark($f_i \wedge f_j$)
16                     if not Table.Marked($\neg f_i \wedge f_j$)
17                         Library.Record($\neg f_i \wedge f_j$, $n$)
18                         Table.Mark($\neg f_i \wedge f_j$)
19                     if not Table.Marked($f_i \wedge \neg f_j$)
20                         Library.Record($f_i \wedge \neg f_j$, $n$)
21                         Table.Mark($f_i \wedge \neg f_j$)
22                     if not Table.Marked($\neg f_i \wedge \neg f_j$)
23                         Library.Record($\neg f_i \wedge \neg f_j$, $n$)
24                         Table.Mark($\neg f_i \wedge \neg f_j$)
25                     if not Table.Marked($f_i \oplus f_j$)
26                         Library.Record($f_i \oplus f_j$, $n$)
27                         Table.Mark($f_i \oplus f_j$)
28     if there is no new function with $L(f) = n$ recorded
29         break
30 return Library
```

Figure 1: Bottom-up enumeration algorithm to record $L(f)$

3.2 Enumerating all *different* minimum circuits

Two circuits implementing the same NPN class and with the same number of gates are considered *different* if their topmost gates are different, or if at least one fanin of the topmost node of one circuit has NPN class that is not present among the NPN classes of the fanins of the topmost node of the other circuit. To construct a library of all *different* minimum circuits, we modify the above algorithm to record minimum circuits in terms of its topmost gate and the NPN classes of its two fanin functions, as shown in Figure 2.

The parameters of procedure *Record* are the NPN class of the function of the circuit, the number of gates in its minimum circuit, the type of the topmost gate, and its two fanin functions, respectively.

3.3 Constructing the library

In the above section, we have constructed an intermediate representation of the library of all different minimum circuit implementations in terms of the topmost node and the NPN classes of two fanins of this node. For practical applications, it is helpful to create a library containing specific circuit structures for each function. To derive the library in the AIG format, we first convert the intermediate representation into Boolean formulas using recursive procedure shown in Figure 3. The formula can be converted to the AIG easily using any AIG package.

```
MinimumCircuitEnumeration
input: The number of input variables $I$
output: All different minimum circuits of functions with up to $I$ inputs
begin
01 Library := InitializeLibrary()
02 Library.Record(CONST_ZERO, 0, NULL, NULL, NULL)
03 Library.Record(INPUT_VARIABLE, 0, NULL, NULL, NULL)
04 for $n = 1 : n_{max}$
05   for $i = 0 : \lceil n/2 \rceil - 1, j = n - 1 : n - \lceil n/2 \rceil$ // n=i+j+1
06     foreach NPN class $c_i$ in Library with $i$ nodes
07       foreach function $f_i$ in class $c_i$
08         foreach NPN class $c_j$ in Library with $j$ nodes
09           $f_j$ := representative function in class $c_j$
10           if $f_i \wedge f_j$ does not appear in Library with $< n$ nodes
11             Library.Record(NPN($f_i \wedge f_j$, $n$), Type1, $f_i, f_j$)
12           if $\neg f_i \wedge f_j$ does not appear in Library with $< n$ nodes
13             Library.Record(NPN($\neg f_i \wedge f_j$), $n$, Type2, $f_i, f_j$)
14           if $f_i \wedge \neg f_j$ does not appear in Library with $< n$ nodes
15             Library.Record(NPN($f_i \wedge \neg f_j$), $n$, Type3, $f_i, f_j$)
16           if $\neg f_i \wedge \neg f_j$ does not appear in Library with $< n$ nodes
17             Library.Record(NPN($\neg f_i \wedge \neg f_j$), $n$, Type4, $f_i, f_j$)
18           if $f_i \oplus f_j$ does not appear in Library with $< n$ nodes
19             Library.Record(NPN($f_i \oplus f_j$), $n$, Type5, $f_i, f_j$)
20   if there is no new function with $L(f) = n$ recorded
21     break
22 return Library
```

Figure 2: Bottom-up enumeration to generate all *different* minimum circuits

Procedure *MakePI* computes and returns one of the input variables based on a combination of permutations. Procedure *CombineFormula* combines two sub-formulas into one with the given type of the topmost gate, as well as a combination of all negations throughout the process. The details of combining permutations and negations are described in Section 3.4.

3.4 Permutation and negation

Since the library stores the minimum circuits for each NPN classes, a query to the circuit implementation of any given function f involves NPN transformation by input permutation and negation. To denote such a transformation of a function from its NPN representative, we use an array P of n integers and an array N of n bits, where n is the number of input variables. Suppose originally the order of input variables for the NPN representative function is $v_1, v_2, \ldots, v_n$, all in positive phase. After the transformation defined by arrays P and N, the variable order becomes $b'_1v'_1, b'_2v'_2, \ldots, b'_nv'_n$, where $b'_i = 0$ denotes the positive phase of variable v'_i, and $b'_i = 1$ denotes the negative phase of this variable. The application rules of arrays P and N are:

$$v'_{P[i]} = v_i, \, b'_{P[i]} = N'[i]$$

Throughout the process of constructing the formula of the circuit with recursive calls described in Section 3.3, we may encounter many P and N transformations. For example, in order to get the circuit of an arbitrary function f, we learn from the library that the circuit of its NPN representative, f^*, is composed of gate g and two fanin functions f_{i1} and f_{i2}. However, from the library we can only get the circuits of the NPN representatives, f^*_{i1} and f^*_{i2} of f_{i1} and f_{i2}, respectively. Hence, given the circuit of f^*_{i1}, we first transform it into f_{i1} by P_1 and N_1, compose the resulting circuit with g and the circuit of f_{i2} to make f^*, and then transform again by P_2 and N_2 to obtain the circuit of f.

```
ConstructFormula
input: Intermediate Library returned by MinimumCircuitEnumeration
output: Circuit structures represented by the Boolean formulas in the file
begin
01 foreach function $f$ in Library
02   foreach implementation $M$ of $f$ in Library
03     MakeFormula($M$, TRUE)

procedure MakeFormula
input: Circuit implementation $M$ ($g, f_1, f_2$),
Flag $flagDump$ indicating whether the formula will be completed
output: (Partial) formula of $M$
begin
01 if M.$f_1$ is PI // INPUT_VARIABLE
02   $str1$ := MakePI($f_1$)
03   $str$ := MakeFormulaFI2($str1$, $M$, $flagDump$)
04 else
05   foreach $M^*$ in Library implementing $f_1$
06     $str1$ := MakeFormula($M^*$, FALSE)
07     $str$ := MakeFormulaFI2($str1$, $M^*$, $flagDump$)
08 return $str$

procedure MakeFormulaFI2
input: Partial formula of the first half $str1$,
Circuit implementation $M$ ($g, f_1, f_2$),
Flag $flagDump$ indicating whether the formula will be completed
output: (Partial) formula of $M$
begin
01 if M.$f_2$ is PI // INPUT_VARIABLE
02   $str2$ := MakePI($f_2$)
03   $str$ := CombineFormula($str1$, $str2$, $g$)
04   if $flagDump$
05     Dump $str$ to output file
06 else
07   foreach $M^*$ in Library implementing $f_2$
08     $str2$ := MakeFormula($M^*$, FALSE)
09     $str$ := CombineFormula($str1$, $str2$, $g$)
10     if $flagDump$
11       Dump $str$ to output file
12 return $str$
```

Figure 3: Converting the circuit implementation into Boolean formula

To compose two transformations P_1, P_2 and N_1, N_2, we compute P' (resp. N') to be equivalent by first applying P_1 (resp. N_1) and then applying P_2 (resp. N_2) as follows:

$$P'[i] = P_1[P_2[i]], \, N'[i] = N_1[P_2[i]] \oplus N_2[i]$$

Below is an example of applying and composing P, N transformations.

Example 1. *Suppose originally the variables are (x_0, x_1, x_2), all in positive phase, and the function is $f_1 = x_0 \land x_1$ (x_2 not used), represented as an AND gate. After the first transformation $P_1 = \{2, 0, 1\}$, $N_1 = \{0, 0, 1\}$, the variable order becomes $(x_1, \neg x_2, x_0)$, and the function becomes $f_2 = x_1 \land \neg x_2$. Note that, before applying the second transformation, the variable order is restored as (x_0, x_1, x_2) and the function as an AND gate of the last two inputs with a negation at the last input. Now, after the second transformation $P_2 = \{1, 0, 2\}$, $N_2 = \{1, 0, 0\}$ is applied, the variables become $(x_1, \neg x_0, x_2)$ and the final function is $f_3 = \neg x_0 \land \neg x_2$. By composing the two transformations, we get $P' = \{0, 2, 1\}$ and $N' = \{1, 0, 1\}$. By applying P' and N' directly to the original function f_1, we get variables $(\neg x_0, \neg x_2, x_1)$ and hence the resulting function $f_4 = \neg x_0 \land \neg x_2$ equals f_3.*

4 EXPERIMENTAL RESULTS

The enumeration algorithms are implemented in `ABC` [2] as command *funenum*. First, we reproduce the results for the number of NPN classes having each number of $L(f)$ and $D(f)$. Next, we generate all the different minimum circuits and show in Section 4.2 the distribution of NPN classes in terms of the number of different minimum structures they have.

4.1 Distribution of minimum circuit sizes

First, the bottom-up enumeration algorithm is applied to compute $L(f)$ and $D(f)$ of all functions up to five inputs. The distribution of NPN classes over the number of nodes in $L(f)$ and $D(f)$ is shown in Table 1. This table appeared to be coherent with the data in *The Art of Computer Programming* [1].

Table 1: Distribution of minimum circuit sizes

	upto 4-input		upto 5-input	
evaluation	L(f)	D(f)	L(f)	D(f)
	# NPN classes			
0	2	2	2	2
1	2	2	2	2
2	5	17	5	17
3	20	179	20	1789
4	34	22	93	614316
5	75		366	
6	68		1730	
7	16		8782	
8			40297	
9			141422	
10			273277	
11			145707	
12			4423	
Total	222	222	616126	616126

4.2 Distribution of the counts of different minimum circuits

Next, we generate and count all *different* minimum circuits of each NPN class. Note that the circuits here are recorded as the topmost gate and the NPN classes of its two fanin functions, hence alternative implementations of the fanin sub-circuit do not count as *different* minimum circuits if the sub-circuits represent the same NPN class. The distributions of minimum circuit counts for 4-input and 5-input functions are shown in Table 2 and Table 3, respectively.

In Table 3, it is interesting to note that among all 11-node NPN classes, there are 5450 classes with a unique minimum circuit and

Table 2: Minimum circuit sizes for 4-input functions

L(f)	# classes	# classes having n different min-circuits with n in range						
		[1]	[2]	[3;4]	[5;8]	[9;16]	[17;32]	Ave.
0	2	2						1.00
1	2	2						1.00
2	5	5						1.00
3	20	13	7					1.35
4	32	16	12	5	1			1.79
5	75	24	17	18	15	1		2.85
6	68	8	4	24	15	16	1	6.06
7	16			2	2	8	4	13.81
Total	222	70	40	49	33	25	5	

one class with exactly 523 different minimum circuits. The next closest to it is one class with exactly 314 different minimum circuits. Similar outliers can be found in other parts of Table 2.

As part of future work, we plan to evaluate how many practical functions have more than one minimum circuit structure. One way of doing this, is to exclude the NPN classes not belonging to the set of classes observed in [8]. The remaining functions could be profiled in terms of the numbers of minimum circuits. If a substantial number of these functions has more than one circuit, these circuits could be useful in practical applications.

Another interesting experiment is to optimize the resulting minimum fanout-free circuits using ABC. If logic sharing can be detected in some cases, the number of two-input nodes can be further reduced, moving from $L(f)$ closer to $C(f)$.

4.3 Library of all different minimum circuits

Finally, as described in Section 3.3, a library of all different minimum circuits represented as formulas and in the AIG form, for all NPN classes up to five inputs is constructed by recursive calls for the two fanin functions of the topmost gate. The library can be downloaded via the following link [17].

Figure 4 shows an example of the several minimum structures of a symmetric function with truth table `16696996` (represented in hexadecimal) that can be found in the library. This function has $L(f) = 8$ and has a total of 15 structures recorded in the library. Some of them are just variations on associativity, and we only show 4 selected ones here. If we consider the consecutive XOR gates as a larger XOR gate, we can see that the four structures are different in terms of the topmost gate: Figure 4(a) is a 3-input XOR gate, Figure 4(b) is a 4-input XOR gate, Figure 4(c) is a 5-input XOR gate, and Figure 4(d) is an AND gate. On the other hand, Figure 4(d) achieves the smallest depth among all the minimum-$L(f)$ structures.

5 PRACTICAL APPLICATIONS

In this section, we discuss several applications that can benefit from the availability of multiple circuit structures for Boolean functions.

One such application is field-programmable gate-array (FPGA) architecture evaluation. FPGAs contain programmable cells, which can implement all or some Boolean functions of a given size. A question may be asked: given a constraint on the size of the cell, what should be its programmable structure so that it could implement as many practically important functions as possible? In mainstream FPGAs, the problem is often solved by utilizing k-input lookup-tables, which can implement any k-input function. However, in alternative FPGA designs [5, 7, 9], one can build custom

Table 3: Minimum circuit sizes for 5-input functions

#Nodes	L(f)	# classes having n different min-circuits with n in range										
		[1]	[2]	[3;4]	[5;8]	[9;16]	[17;32]	[33;64]	[65;128]	[129;256]	[257;512]	[513;1024]	Ave
0	2	2											1.00
1	2	2											1.00
2	5	5											1.00
3	20	13	7										1.35
4	93	58	25	9	1								1.52
5	366	209	76	58	22	1							1.86
6	1730	865	344	315	172	31	3						2.29
7	8782	3876	1727	1681	1037	404	52	4	1				2.85
8	40297	15147	7057	8134	5856	3098	871	127	5	2			3.78
9	141422	43610	21297	27757	23861	16333	6894	1525	136	9			5.27
10	273277	53148	31217	48088	52334	46734	29891	10434	1359	68	4		8.82
11	145707	5450	5327	11946	21956	33912	36615	23435	6621	436	8	1	21.50
12	4423		1	2	7	64	365	1468	1949	555	12		80.27
Total	616126	122385	67078	97990	105246	100577	74691	36993	10071	1070	24	1	

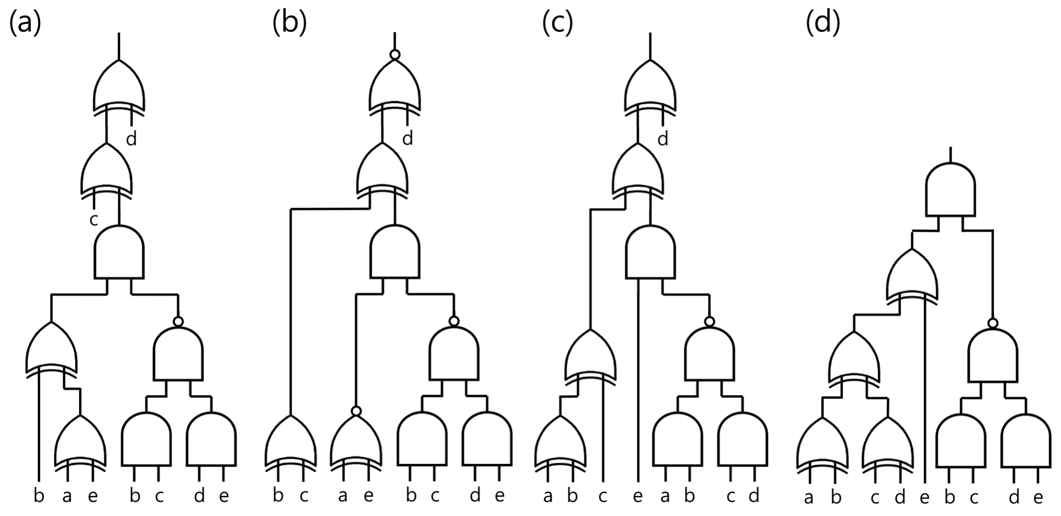

Figure 4: Several circuit structures of a symmetric function (truth table: 16696996).

programmable cells capable of implementing functions using a network of simple gates. The structure of such a cell can be determined by analyzing available structures for a subset of practically important functions. These structures can be found by enumerating minimum circuits.

Another potential application is in logic optimization. When applied to an existing circuit, the optimization identifies a small subcircuit, derives its Boolean function, and replaces it with a pre-computed minimum circuit, while improving a cost function. This type of optimization is called peephole optimization [10] or circuit rewriting [11, 12]. It greatly benefits from having access to many circuit structures computed for each Boolean function.

Finally, a popular research trend in recent years has been approximate logic synthesis [13]. The goal of approximate synthesis is the same as that of regular synthesis, except that the resulting circuit is allowed to be incorrect for some (typically small) percentage of input variable assignments. A typical question asked by approximate synthesis is: given a circuit, what gate(s) can be removed or changed while minimizing the number of errors introduced at the output? This hard problem can be approached in a number of ways. Having a library of all circuit structures for simple functions might be useful because it reduces search for a better structure to evaluating structures available for functions whose output is different from the target function for a few input combinations.

6 CONCLUSIONS AND FUTURE WORK

This paper discusses the enumeration of minimum circuit structures and their potential applications. A bottom-up enumeration method is proposed and implemented to produce statistics for $L(f)$ and $D(f)$ minimum-circuits for 4- and 5-input NPN classes. This computation independently verifies the information from *The Art of Computer Programming* [1]. Novel contributions are: enumerating all *different* minimum circuit structures and providing a library of these structures for use in practical applications. The future work may include extending the results to a practical subset of 6-input functions.

7 ACKNOWLEDGEMENTS

We thank the anonymous reviewers for many helpful comments and suggestions. This work was supported in part by SRC contracts

2710.001 "SAT-based methods for scalable synthesis and verification" and 2867.001 "Deep integration of computation engines for scalability in synthesis and verification".

REFERENCES

[1] D. E. Knuth. *The Art of Computer Programming, Volume 4, Section 7.1.2, Boolean evaluation.* Pearson Education, Inc., Boston, MA, USA, 2015.

[2] Berkeley Logic Synthesis and Verification Group. ABC: A System for Sequential Synthesis and Verification. Available: https://eecs.berkeley.edu/~alanmi/abc/

[3] M. G. Martins, V. Callegaro, L. Machado, R. P. Ribas, and A. I. Reis. Functional composition paradigm and applications. In *Proc. IWLS*, 2012.

[4] V. Bertacco and M. Damiani. Disjunctive decomposition of logic functions. In *Proc. ICCAD*, pp. 78-82, 1997.

[5] A. Ling, D. Singh, and S. Brown. FPGA PLB evaluation using Quantified Boolean Satisfiability. In *Proc. FPGA*, 2005.

[6] A. Mishchenko and R. Brayton. Faster Logic Manipulation for Large Designs. In *Proc. IWLS*, 2013.

[7] A. Mishchenko, R. Brayton, W. Feng, and J. Greene. Technology mapping into general programmable cells. In *Proc. FPGA*, 2015.

[8] A. Kennings, A. Mishchenko, K. Vorwerk, V. Pevzner, and A. Kundu. Generating efficient libraries for use in FPGA resynthesis algorithms. In *Proc. IWLS*, pp. 147-154, 2010.

[9] W. Feng, J. Greene, and A. Mishchenko. Improving FPGA performance with a S44 LUT structure. In *Proc. FPGA*, 2018.

[10] L. Amaru, M. Soeken, P. Vuillod, J. Luo, A. Mishchenko, P.-E. Gaillardon, J. Olson, R. K. Brayton, and G. De Micheli. Enabling exact delay synthesis. In *Proc. ICCAD*, 2017.

[11] P. Bjesse and A. Boralv. DAG-aware circuit compression for formal verification. In *Proc. ICCAD*, pp. 42-49, 2004.

[12] A. Mishchenko, S. Chatterjee, and R. Brayton. DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In *Proc. DAC*, pp. 532-536, 2006.

[13] Y. Wu, Ch. Shen, Y. Jia, and W. Qian. Approximate logic synthesis for FPGA by wire removal and local function change. In *Proc. ASP-DAC*, pp. 163-169, 2017.

[14] E. Lehman, Y. Watanabe, J. Grodstein, and H. Harkness. Logic decomposition during technology mapping. In *IEEE Trans. on CAD*, 16(8): 813-833, 1997.

[15] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam. Reducing structural bias in technology mapping. In *Proc. ICCAD*, pp. 519-526, 2005.

[16] V. Callegaro, M. G. A. Martins, R. P. Ribas, and A. Reis. Read-polarity-once Boolean functions. In *Proc. SBCCI*, 2013.

[17] A library of minimum circuit structures of small Boolean functions: https://eecs.berkeley.edu/~alanmi/research/funenum