Electronic Colloquium on Computational Complexity, Report No. 44 (2021)

SAT-based Circuit Local Improvement

Alexander S. Kulikov ORCID
Steklov Mathematical Institute at St. Petersburg, Russian Academy of Sciences
St. Petersburg State University

Nikita Slezkin ORCID
St. Petersburg State University

Abstract

Finding exact circuit size is a notorious optimization problem in practice. Whereas modern computers and algorithmic techniques allow to find a circuit of size seven in blink of an eye, it may take more than a week to search for a circuit of size thirteen. One of the reasons of this behavior is that the search space is enormous: the number of circuits of size s is $s^{\Theta(s)}$, the number of Boolean functions on n variables is 2^{2^n}.

In this paper, we explore the following natural heuristic idea for decreasing the size of a given circuit: go through all its subcircuits of moderate size and check whether any of them can be improved by reducing to SAT. This may be viewed as a local search approach: we search for a smaller circuit in a ball around a given circuit. We report the results of experiments with various symmetric functions.

2012 ACM Subject Classification Theory of computation → Circuit complexity

Keywords and phrases circuits, algorithms, complexity theory, SAT, SAT-solvers, heuristics

Supplementary Material Source code:
repository: `GitHub`

Funding *Alexander S. Kulikov*: Supported by Russian Science Foundation (18-71-10042).

1 Boolean Circuits

A Boolean *straight line program* of size r for input variables $(x_1, \ldots, x_n)$ is a sequence of r instructions where each instruction $g \leftarrow h \circ k$ applies a binary Boolean operation $\circ$ to two operands h, k each of which is either an input bit or the result of a previous instruction. If m instructions are designated as outputs, the straight line program computes a function $\{0,1\}^n \to \{0,1\}^m$ in a natural way. We denote the set of all such functions by $B_{n,m}$ and we let $B_n = B_{n,1}$. For a Boolean function $f\colon \{0,1\}^n \to \{0,1\}^m$, by $\text{size}(f)$ we denote the minimum size of a straight line program computing f. A Boolean *circuit* shows a flow graph of a program.

Figure 1 gives an example for the $\text{SUM}_n\colon \{0,1\}^n \to \{0,1\}^l$ function that computes the binary representation of the sum of n bits:

$$\text{SUM}_n(x_1, \ldots, x_n) = (w_0, w_1, \ldots, w_{l-1})\colon \sum_{i=1}^{n} x_i = \sum_{i=0}^{l-1} 2^i w_i, \quad \text{where } l = \lceil \log_2(n+1) \rceil .$$

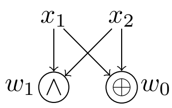
This function transforms n bits of weight 0 into l bits of weights $(0, 1, \ldots, l-1)$. The straight line programs are given in `Python` programming language. This makes particularly easy to verify their correctness. For example, the program for SUM_3 can be verified with just three lines of code:

```
from itertools import product

for x1, x2, x3 in product(range(2), repeat=3):
```

ISSN 1433-8092

```python
def sum2(x1, x2):
    w0 = x1 ^ x2
    w1 = x1 * x2
    return w0, w1
```




```python
def sum3(x1, x2, x3):
    a = x1 ^ x2
    b = x2 ^ x3
    c = a | b
    w0 = a ^ x3
    w1 = c ^ w0
    return w0, w1
```

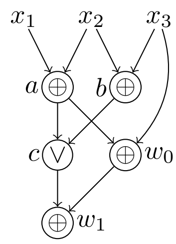


Figure 1 Optimal size straight line programs and circuits for SUM_2 and SUM_3. These two circuits are known as *half adder* and *full adder*.

```python
    w0, w1 = sum3(x1, x2, x3)
    assert x1 + x2 + x3 == w0 + 2 * w1
```

Determining $\text{size}(f)$ requires proving lower bounds: to show that $\text{size}(f) > \alpha$, one needs to prove that *every* circuit of size at most α does not compute f. Known lower bounds are far from being satisfactory: the strongest known lower bound for a function family in NP is $(3+1/86)n - o(n)$ [7]. Here, by a *function family* we mean an infinite sequence of functions $\{f_n\}_{n=1}^{\infty}$ where $f_n \in B_n$.

Even proving lower bounds for specific functions (rather than function families) is difficult. Brute force approaches become impractical quickly: $|B_n| = 2^{2^n}$, hence already for $n = 6$, one cannot just enumerate all functions from B_n; also, the number of circuits of size s is $s^{\Theta(s)}$, hence checking all circuits of size s takes reasonable time for small values of s only. Knuth [9] found the exact circuit size of all functions from B_4 and B_5.

Finding the exact value of $\text{size}(f)$ for $f \in B_6$ is already a difficult computational task for modern computers and techniques. One approach is to translate a statement "there exists a circuit of size s computing f" to a Boolean formula and to pass it to a SAT-solver. Then, if the formula is satisfiable, one decodes a circuit from its satisfying assignment; otherwise, one gets a (computer generated) proof of a lower bound $\text{size}(f) > s$. This circuit synthesis approach was proposed by Kojevnikov et al. [11] and, since then, has been used in various circuit synthesis programs (`abc` [1], `mockturtle` [18], `sat-chains` [8]).

The state-of-the-art SAT-solvers are surprisingly efficient and allow to handle various practically important problems (with millions of variables) and even help to resolve open problems [3]. Still, already for small values of n and s the problem of finding a circuit of size s for a function from B_n is difficult for SAT-solvers. We demonstrate the limits of this approach on *counting* functions:

$$\text{MOD}_n^{m,r} = [x_1 + \cdots + x_n \equiv r \bmod m]$$

(here, $[\cdot]$ is the Iverson bracket: $[S]$ is equal to 1 if S is true and is equal to 0 otherwise). Using SAT-solvers, Knuth [10, solution to exercise 480] found $\text{size}(\text{MOD}_n^{3,r})$ for all $3 \le n \le 5$ and all $0 \le r \le 2$. Based on the found numbers, he made the following conjecture:

$$\text{size}(\text{MOD}_n^{3,r}) = 3n - 5 - [(n+r) \equiv r \bmod 3] \text{ for all } n \ge 3 \text{ and } r. \quad (1)$$

He was also able to find the circuit size for the $n = 6, r = 0$ case and wrote: "The case $n = 6$ and $r \neq 0$, which lies tantalizingly close to the limits of today's solvers, is still unknown."

To summarize, our current abilities for checking whether there exists a Boolean circuit of size s are roughly the following:

- for $s \leq 6$, this can be done in a few seconds;
- for $7 \leq s \leq 12$, this can (sometimes) be done in a few days;
- for $s \geq 13$, this is out of reach.

In this paper, we explore the limits of the following natural idea: given a circuit, try to improve its size by improving (using SAT-solvers, for example) the size of its subcircuit of size seven. This is a kind of a local search approach: we have no possibility to go through the whole space of all circuits, but we can at least search in a neighborhood of a given circuit. This allows us to work with circuits consisting of many gates.

As the results of experiments, we show several circuits for which the approach described above leads to improved upper bounds. In particular, we support Knuth's conjecture (1) by proving the matching upper bound. Also, we present improvements for size(SUM_n) for various small n. Finally, we provide examples of circuits that are optimal locally, but not globally: our program is not able to find a (known) smaller circuit since it is "too different" from the original circuit.

2 Program Overview

The program is implemented in `Python`. We give a high-level overview of its main features below. All the code shown below can be found in the file `tutorial.py` at [4]. One may run it after installing a few `Python` modules. Alternatively, one may run the Jupyter notebook `tutorial.ipynb` in the cloud without installing anything. To do this, press the badge "Colab" at [4].

2.1 Manipulating Circuits

This is done through the `Circuit` class. One can load and save circuits as well as print and draw them. A nicely looking layout of a circuit is produced by the `pygraphviz` module [16]. The program also contains some built-in circuits that can be used as building blocks. The following sample code constructs a circuit for SUM_5 out of two full adders and one half adder. This construction is shown in Figure 2(a). Then, the circuit is verified via the `check_sum_circuit` method. Finally, the circuit is drawn. As a result, one gets a picture similar to the one in Figure 2(b).

```
circuit = Circuit(input_labels=['x1', 'x2', 'x3', 'x4', 'x5'])
x1, x2, x3, x4, x5 = circuit.input_labels
a0, a1 = add_sum3(circuit, [x1, x2, x3])
b0, b1 = add_sum3(circuit, [a0, x4, x5])
w1, w2 = add_sum2(circuit, [a1, b1])
circuit.outputs = [b0, w1, w2]
check_sum_circuit(circuit)
circuit.draw('sum5')
```

2.2 Finding Efficient Circuits

The class `CircuitFinder` allows to check whether there exists a circuit of the required size for a given Boolean function. For example, one may discover the full adder as follows. (The

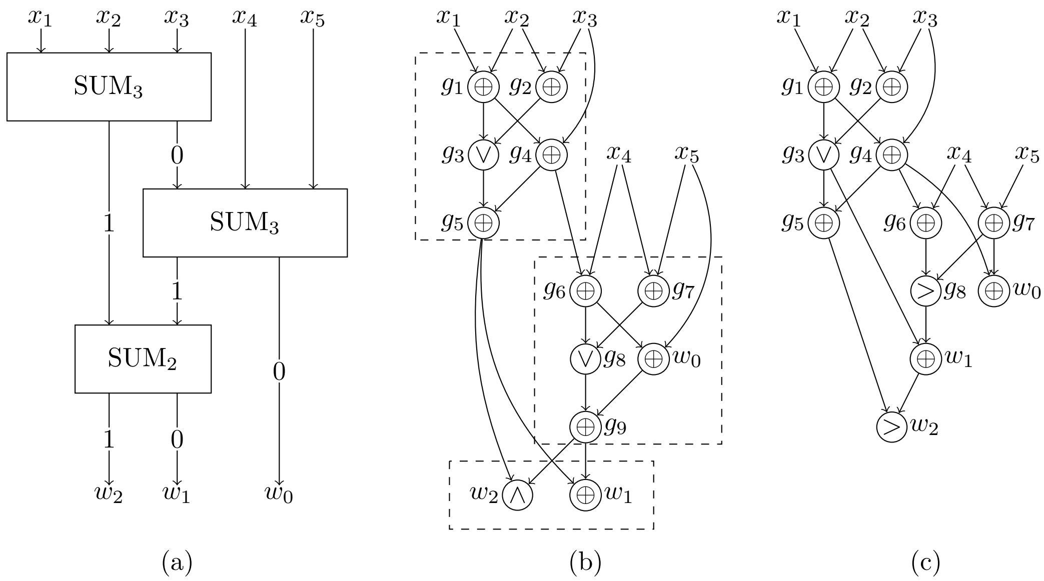


Figure 2 (a) A schematic circuit for SUM$_5$ composed out of two full adders and one half adder. (b) The corresponding circuit of size 12. (c) An improved circuit of size 11.

function `sum_n` returns the list of $\lceil \log_2(n+1) \rceil$ bits of the binary representation of the sum of n bits.)

```
def sum_n(x):
    return [(sum(x) >> i) & 1 for i in range(ceil(log2(len(x) + 1)))]


circuit_finder = CircuitFinder(dimension=3, number_of_gates=5,
                               function=sum_n)
circuit = circuit_finder.solve_cnf_formula()
circuit.draw('sum3')
```

This is done by encoding the task as a CNF formula and invoking the PicoSAT solver [2] (via the `pycosat` module [15]). The reduction to SAT is described in [11].

As mentioned in the introduction, the limits of applicability of this approach (for finding a circuit of size s) are roughly the following: for $s \leq 6$, it usually works in less than a minute; for $7 \leq s \leq 12$, it may already take up to several hours or days; for $s \geq 13$, it becomes almost impractical. The running time may vary a lot for inputs of the same length. In particular, it usually takes much longer to prove that the required circuit does not exist (by proving that the corresponding formula is unsatisfiable).

The program allows to predefine some of the gates and wires of a circuit. We demonstrate this functionality later in the text.

2.3 Improving Circuits

The method `improve_circuit` goes through all subcircuits of a given size of a given circuit and checks whether any of them can be replaced by a smaller subcircuit (computing the same function) via `find_circuit`. For example, applying this method to the circuit from Figure 2(b) gives the circuit from Figure 2(c) in nine seconds.

```
circuit = Circuit(input_labels=[f'x{i}' for i in range(1, 6)], gates={})
circuit.outputs = add_sum5_suboptimal(circuit, circuit.input_labels)
improved_circuit = improve_circuit(circuit, subcircuit_size=5,
                                   connected=True)
print(improved_circuit)
improved_circuit.draw('sum5')
```

This circuit can be also found via `find_circuit` directly, but it takes about seven hours.

3 Evaluation

In this section, we report the results of experiments with various symmetric functions. A function $f(x_1, \ldots, x_n)$ is called *symmetric* if its value depends on $\sum_{i=1}^{n} x_i$ only. They are among the most basic Boolean functions:

- to specify an arbitrary Boolean function from B_n, one needs to write down its truth table of length 2^n; symmetric functions allow for more compact representation: it is enough to specify $n+1$ bits (for each of $n+1$ values of $\sum_{i=1}^{n} x_i$);
- circuit complexity of almost all functions of n variables is exponential ($\Theta(2^n/n)$), whereas any symmetric function can be computed by a linear size circuit ($O(n)$).

Despite simplicity of symmetric functions, we still do not know how optimal circuits look like for most of them. Below, we present new circuits for some of these functions.

3.1 Sum Function

The SUM function is a fundamental symmetric function: for any symmetric $f \in B_n$, $\mathrm{size}(f) \leq \mathrm{size}(\mathrm{SUM}_n) + o(n)$. The reason for this is that any function from B_n can be computed by a circuit of size $O(2^n/n)$ by the results of Muller [14] and Lupanov [13]. This allows to compute any symmetric $f(x_1, \ldots, x_n) \in B_n$ as follows: first, compute $\mathrm{SUM}_n(x_1, \ldots, x_n)$ using $\mathrm{size}(\mathrm{SUM}_n)$ gates; then, compute the resulting bit using at most $O(2^{\log n}/\log n) = o(n)$ gates. For the same reason, any lower bound $\mathrm{size}(f) \leq \alpha$ for a symmetric function $f \in B_n$ implies a lower bound $\mathrm{size}(\mathrm{SUM}_n) \leq \alpha - o(n)$. Currently, we know the following bounds for SUM_n:

$$2.5n - O(1) \leq \mathrm{size}(\mathrm{SUM}_n) \leq 4.5n + o(n)\,.$$

The lower bound is due to Stockmeyer [19], the upper bound is due to Demenkov et al. [5].

A circuit for SUM_n can be constructed from circuits for SUM_k for some small k. For example, using full and half adders as building blocks, one can compute SUM_n (for any n) by a circuit of size $5n$ as follows. Start from n bits $(x_1, \ldots, x_n)$ of weight 0. While there are three bits of the same weight k, replace them by two bits of weights k and $k+1$ using a full adder. This way, one gets at most two bits of each weight $0, 1, \ldots, l-1$ ($l = \lceil \log_2(n+1) \rceil$) in at most $5(n-l)$ gates (as each full adder reduces the number of bits). To leave exactly one bit of each weight, it suffices to use at most l half or full adders ($o(n)$ gates). Let us denote the size of the resulting circuit by $s(n)$. The second row of Table 1 shows the values of $s(n)$ for some $n \leq 15$ (see (28) in [9]).

n	2	3	4	5	6	7	8	9	10	15
$s(n)$	2	5	9	12	17	20	26	29	34	55
size(SUM$_n$)	2	5	9	11	≤ 16	≤ 19	≤ 25	≤ 27	≤ 32	≤ 53

Table 1 The first line shows the value of n. The second line gives the size $s(n)$ of a circuit for SUM$_n$ composed out of half and full adders. The third row shows known bounds for size(SUM$_n$).

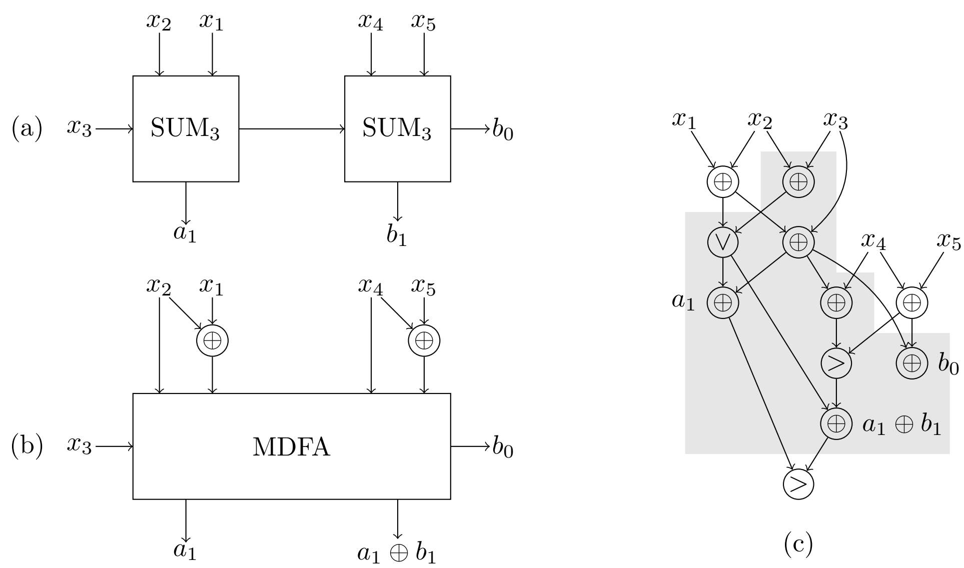


Figure 3 (a) Two consecutive SUM$_3$ blocks. (b) The MDFA block. (c) The highlighted part of the optimal circuit for SUM$_5$ computes MDFA.

In a similar fashion, one can get an upper bound (see Theorem 1 in [12])

$$\text{size}(\text{SUM}_n) \le \frac{\text{size}(\text{SUM}_k)}{k - \lceil \log_2(k+1) \rceil} \cdot n + o(n)\,. \tag{2}$$

This motivates the search for efficient circuits for SUM$_k$ for small values of k. The bottom row of Table 1 gives upper bounds that we were able to find using the program (the upper bounds for $n \le 7$ were found by Knuth [9]). The table shows that the first value where $s(n)$ is not optimal is $n = 5$. The best upper bound for SUM$_n$ given by (2) is $4.75n + o(n)$ for $n = 7$. The upper bound for $n = 15$ is $53n/11 + o(n)$ which is worse than the previous upper bound. But if it turned out that size(SUM$_{15}$) ≤ 52, it would give a better upper bound.

The found circuits for SUM$_n$ for $n \le 15$ does not allow to improve the strongest known upper bound size(SUM$_n$) $\le 4.5n + o(n)$ due to Demenkov et al. [5]. Below, we present several interesting observations on the found circuits.

3.1.1 Best Known Upper Bound for the SUM Function

The optimal circuit of size 11 for SUM$_5$ shown in Figure 2(c) can be used to get an upper bound $4.5n + o(n)$ for size(SUM$_n$) (though not through (2) directly). To do this, consider two consecutive SUM$_3$ circuits shown in Figure 3(a). Its specification is: $x_1 + \cdots + x_5 = b_0 + 2(a_1 + b_1)$, its size is equal to 10. One can construct a similar block, called MDFA (for

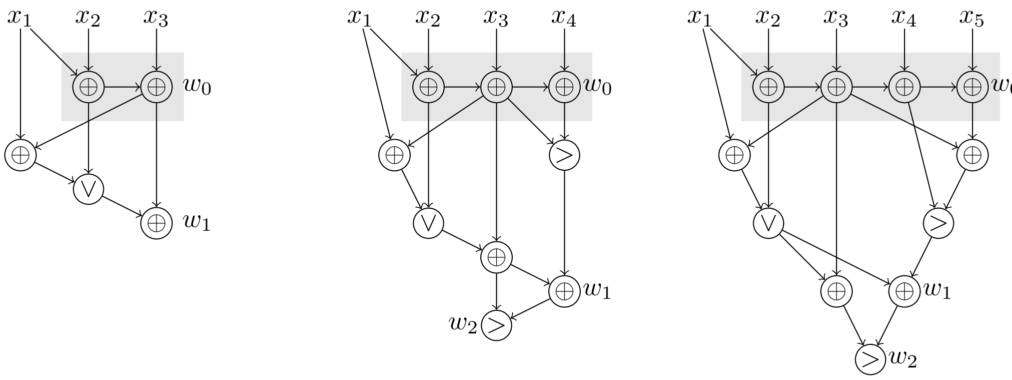


Figure 4 Optimal circuits computing SUM_n for $n = 3, 4, 5$ with a specific structure: every input, except for x_1, has out-degree one.

modified double full adder), of size 8, whose specification is

$$\mathrm{MDFA}(x_1 \oplus x_2, x_2, x_3, x_4, x_4 \oplus x_5) = (b_0, a_1, a_1 \oplus b_1)\,,$$

see Figure 3(b). The fact that MDFA uses the encoding $(p, p \oplus q)$ for pairs of bits (p, q), allows to use it recursively to compute SUM_n: first, compute $x_1 \oplus x_2, x_3 \oplus x_4, \dots, x_{n-1} \oplus x_n$ ($n/2$ gates); then, apply $n/2$ MDFA blocks ($4n$ gates). The MDFA block was constructed by Demenkov et al. [5] in a semiautomatic manner. And it turns out that MDFA is just a subcircuit of the optimal circuit for SUM_5! See Figure 3(c).

3.1.2 Best Known Circuits for SUM with New Structure

For many upper bounds from the bottom row of Table 1, we found circuits with the following interesting structure: the first thing the circuit computes is $x_1 \oplus x_2 \oplus \cdots \oplus x_n$; moreover the variables $x_2, \dots, x_n$ are used for this only. This is best illustrated by an example — see Figure 4.

These circuits can be found using the following code. It demonstrates two new useful features: fixing gates and forbidding wires between some pairs of gates.

```
def sum_n(x):
    return [(sum(x) >> i) & 1 for i in range(ceil(log2(len(x) + 1)))]


for n, size in ((3, 5), (4, 9), (5, 11)):
    circuit_finder = CircuitFinder(dimension=n, number_of_gates=size,
                                   function=sum_n)
    circuit_finder.fix_gate(n, 0, 1, '0110')
    for k in range(n - 2):
        circuit_finder.fix_gate(n + k + 1, k + 2, n + k, '0110')
    for i in range(1, n):
        for j in range(n, n + size):
            if i + n - 1 != j:
                circuit_finder.forbid_wire(i, j)
```

```
circuit = circuit_finder.solve_cnf_formula(verbose=0)
circuit.draw(f'sum{n}')
```

3.1.3 Optimal Circuits for Counting Modulo 4

The optimal circuit for SUM_5 can be used to construct an optimal circuit of size $2.5n + O(1)$ for MOD_4^n due to Stockmeyer [19]. To do this, note that there is a subcircuit (of the circuit at Figure 2(c)) of size 9 that computes the two least significant bits (w_0, w_1) of $x_1 + \dots + x_5$ (one removes the gates g_5, w_2). To compute $x_1 + \dots + x_n \bmod 4$, one first applies $\frac{n}{4}$ such blocks and then computes the parity of the resulting bits of weight 1. The total size is $9 \cdot \frac{n}{4} + \frac{n}{4} = 2.5n$. Thus, the circuit that Stockmeyer constructed in 1977 by hand, nowadays can be found automatically in a few seconds.

3.2 Modulo-3 Function

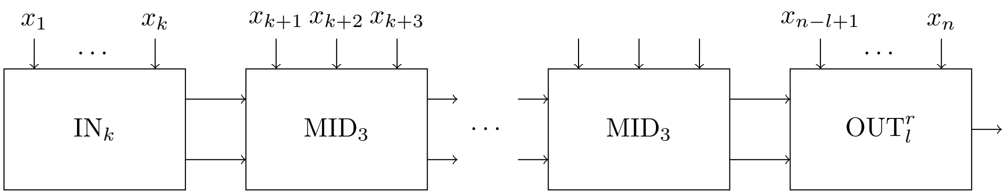
In [11], Kojevnikov et al. presented circuits of size $3n + O(1)$ for $\mathrm{MOD}_n^{3,r}$ (for any r). Later, Knuth [10, solution to exercise 480] analyzed their construction and proved an upper bound $3n-4$. Also, by finding the exact values for $\mathrm{size}(\mathrm{MOD}_n^{3,r})$ for all $3 \le n \le 5$ and all $0 \le r \le 2$, he made the conjecture (1). Using our program, we proved the conjectured upper bound for all n.

▶ **Theorem 1.** *For all $n \ge 3$ and all $r \in \{0, 1, 2\}$,*

$$\mathrm{size}(\mathrm{MOD}_n^{3,r}) \le 3n - 5 - [(n + r) \equiv r \bmod 3]\,.$$

To prove Knuth's conjecture, one also needs to prove a lower bound on $\mathrm{size}(\mathrm{MOD}_n^{3,r})$. The currently strongest known lower bound for $\mathrm{size}(\mathrm{MOD}_n^{3,r})$ is $2.5n - O(1)$ due to Stockmeyer [19] (and no stronger lower bound is known for any other symmetric function).

Proof. As in [11], we construct the required circuit out of constant size blocks. Schematically, the circuit looks as follows.



Here, the n input bits are passed from above. What is passed from block to block (from left to right) is the pair of bits (r_0, r_1) encoding the current remainder r modulo 3 as follows: if $r = 0$, then $(r_0, r_1) = (0, 0)$; if $r = 1$, then $(r_0, r_1) = (0, 1)$; if $r = 2$, then $r_0 = 1$. The first block IN_k takes the first k input bits and computes the remainder of their sum modulo 3. It is followed by a number of MID_3 blocks each of which takes the current remainder and three new input bits and computes the new remainder. Finally, the block OUT_l^r takes the remainder and the last l input bits and outputs $\mathrm{MOD}_n^{3,r}$. The integers k, l take values in $\{2, 3, 4\}$ and $\{1, 2, 3\}$, respectively. Their exact values depend on r and $n \mod 3$ as described below.

The theorem follows from the following upper bounds on the circuit size of the just introduced functions: $\mathrm{size}(\mathrm{IN}_2) \le 2$, $\mathrm{size}(\mathrm{IN}_3) \le 5$, $\mathrm{size}(\mathrm{IN}_4) \le 7$, $\mathrm{size}(\mathrm{MID}_3) \le 9$, $\mathrm{size}(\mathrm{OUT}_2^0) \le 5$, $\mathrm{size}(\mathrm{OUT}_1^1) \le 2$, $\mathrm{size}(\mathrm{OUT}_3^2) \le 8$. The corresponding circuits are presented

	$n=3t$	$n=3t+1$	$n=3t+2$
$r=0$	$(4,t-2,2)$, $(7,5)$, $3n-6$	$(2,t-1,2)$, $(2,5)$, $3n-5$	$(3,t-1,2)$, $(5,5)$, $3n-5$
$r=1$	$(2,t-1,1)$, $(2,2)$, $3n-5$	$(3,t-1,1)$, $(5,2)$, $3n-5$	$(4,t-1,1)$, $(7,2)$, $3n-6$
$r=2$	$(3,t-2,3)$, $(5,8)$, $3n-5$	$(4,t-2,3)$, $(7,8)$, $3n-6$	$(2,t-1,3)$, $(2,8)$, $3n-5$

Table 2 Choosing parameters k,m,l depending on $n \mod 3$ and r. The circuit is composed out of blocks as follows: $\mathrm{IN}_k + m \times \mathrm{MID}_3 + \mathrm{OUT}^l_r$. For each pair $(n \mod 3, r)$ we show three things: the triple (m,k,l); the sizes of two blocks: $\mathrm{size}(\mathrm{IN}_k)$ and $\mathrm{size}(\mathrm{OUT}^l_r)$; the size of the resulting circuit computed as $s = \mathrm{size}(\mathrm{IN}_k) + 9m + \mathrm{size}(\mathrm{OUT}^l_r)$. For example, the top left cell is read as follows: when $r=0$ and $n=3t$, we set $k=4, m=t-2, l=2$; the resulting circuit is then $\mathrm{IN}_4 + (t-2) \times \mathrm{MID}_3 + \mathrm{OUT}^2_0$; since $\mathrm{size}(\mathrm{IN}_4)=7$ and $\mathrm{size}(\mathrm{OUT}^2_0)=5$, the size of the circuit is $7+9(t-2)+5 = 9t-6 = 3n-6$.

in the Appendix by a straightforward `Python` code that verifies their correctness. (The presented code proves the mentioned upper bounds by providing explicit circuits. We have also verified that no smaller circuits exist meaning that the inequalities above are in fact equalities.)

Table 2 shows how to combine the blocks to get a circuit computing $\mathrm{MOD}_n^{3,r}$ of the required size. (Technically, it requires n to be at least 4. For $n=3$, the corresponding circuits are easy to construct.)

◀

3.3 Threshold-2 Function

Here, we present an example of a reasonably small circuit that our program fails to improve though a better circuit is known. The reason is that these two circuits are quite different. The function we are going to consider is the threshold-2 function:

$$\mathrm{THR}^2_n(x_1,\dots,x_n) = [x_1+\dots+x_n \geq 2]\,.$$

Figures 5 and 6 show circuits of size 31 and 29 for THR^2_{12}. They are quite different and our program is not able to find out that the circuit of size 31 is suboptimal. One can construct the two circuits in the program as follows.

```
c = Circuit(input_labels=[f'x{i}' for i in range(1, 13)], gates={})
c.outputs = add_naive_thr2_circuit(c, c.input_labels)
c.draw('thr2naive')

c = Circuit(input_labels=[f'x{i}' for i in range(1, 13)], gates={})
c.outputs = add_efficient_thr2_circuit(c, c.input_labels, 3, 4)
c.draw('thr2efficient')
```

4 Further Directions

In the paper, we focus mainly on proving asymptotic upper bounds for function families (that is, that work for every input size). A natural further step is to apply the program to specific circuits that are used in practice.

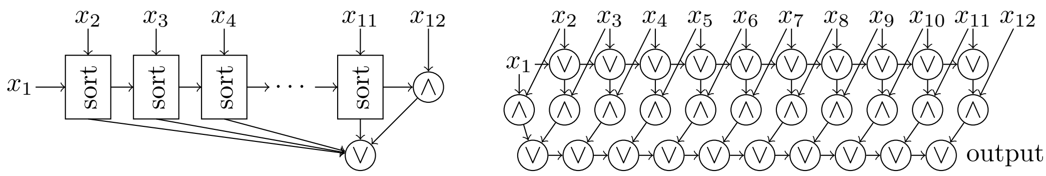


Figure 5 A circuit of size 31 for THR^2_{12}: (a) block structure and (b) gate structure. The $\mathrm{SORT}(u, v)$ block sorts two input bits as follows: $\mathrm{SORT}(u, v) = (\min\{u, v\}, \max\{u, v\}) = (u \wedge v, u \vee v)$. The circuit performs one and a half iterations of the bubble sort algorithm: one first finds the maximum bit among n input bits; then, it remains to compute the disjunction of the remaining $n - 1$ bits to check whether there is at least one 1 among them. In general, this leads to a circuit of size $3n - 5$.

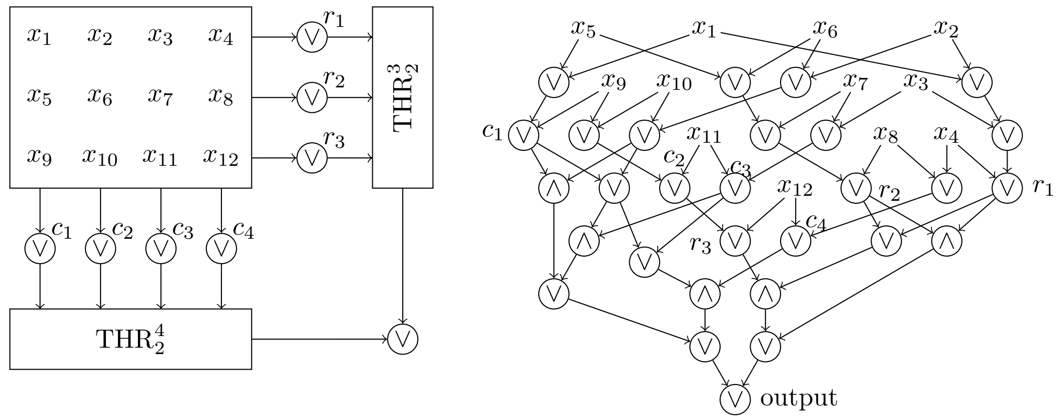


Figure 6 A circuit of size 29 for THR^2_{12}: (a) block structure and (b) gate structure. It implements a clever trick by Dunne [6]. Organize 12 input bits into a 3×4 table. Compute disjunctions r_1, r_2, r_3 of the rows and disjunctions c_1, c_2, c_3, c_4 of the columns. Then, there are at least two 1's among $x_1, \ldots, x_{12}$ if and only if there are at least two 1's among either r_1, r_2, r_3 or c_1, c_2, c_3, c_4. This allows to proceed recursively. In general, it leads to a circuit of size $2n + o(n)$. (Sergeev [17] showed recently that the *monotone* circuit size of THR^2_n is $2n + \Theta(\sqrt{n})$.)

It would also be interesting to extend the program so that it is able to discover the circuit from Figure 6.

References

1 `https://github.com/berkeley-abc/abc`.

2 Armin Biere. PicoSAT essentials. *J. Satisf. Boolean Model. Comput.*, 4(2-4):75–97, 2008. URL: `https://doi.org/10.3233/sat190039`, `doi:10.3233/sat190039`.

3 Joshua Brakensiek, Marijn Heule, John Mackey, and David Narváez. The resolution of Keller's conjecture. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, volume 12166 of *Lecture Notes in Computer Science*, pages 48–65. Springer, 2020. URL: `https://doi.org/10.1007/978-3-030-51074-9_4`, `doi:10.1007/978-3-030-51074-9\_4`.

4 `https://github.com/alexanderskulikov/circuit_improvement`.

5 Evgeny Demenkov, Arist Kojevnikov, Alexander S. Kulikov, and Grigory Yaroslavtsev. New upper bounds on the boolean circuit complexity of symmetric functions. *Inf. Process. Lett.*, 110(7):264–267, 2010. URL: `https://doi.org/10.1016/j.ipl.2010.01.007`, `doi:10.1016/j.ipl.2010.01.007`.

6 Paul E. Dunne. *Techniques for the analysis of monotone Boolean networks*. PhD thesis, University of Warwick, 1984.

7 Magnus Gausdal Find, Alexander Golovnev, Edward A. Hirsch, and Alexander S. Kulikov. A better-than-3n lower bound for the circuit complexity of an explicit function. In Irit Dinur, editor, *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS 2016, 9-11 October 2016, Hyatt Regency, New Brunswick, New Jersey, USA*, pages 89–98. IEEE Computer Society, 2016. URL: `https://doi.org/10.1109/FOCS.2016.19`, `doi:10.1109/FOCS.2016.19`.

8 `http://www-cs-faculty.stanford.edu/~knuth/programs.html`.

9 Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 0: Introduction to Combinatorial Algorithms and Boolean Functions (Art of Computer Programming)*. Addison-Wesley Professional, 1 edition, 2008.

10 Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*. Addison-Wesley Professional, 1st edition, 2015.

11 Arist Kojevnikov, Alexander S. Kulikov, and Grigory Yaroslavtsev. Finding efficient circuits using SAT-solvers. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 32–44. Springer, 2009. URL: `https://doi.org/10.1007/978-3-642-02777-2_5`, `doi:10.1007/978-3-642-02777-2_5`.

12 Alexander S. Kulikov. Improving circuit size upper bounds using sat-solvers. In Jan Madsen and Ayse K. Coskun, editors, *2018 Design, Automation & Test in Europe Conference & Exhibition, DATE 2018, Dresden, Germany, March 19-23, 2018*, pages 305–308. IEEE, 2018. URL: `https://doi.org/10.23919/DATE.2018.8342026`, `doi:10.23919/DATE.2018.8342026`.

13 Oleg Lupanov. A method of circuit synthesis. *Izvestiya VUZov, Radiofizika*, 1:120–140, 1959.

14 David E. Muller. Complexity in electronic switching circuits. *IRE Transactions on Electronic Computers*, EC-5:15–19, 1956.

15 `https://pypi.org/project/pycosat/`.

16 `https://pygraphviz.github.io/`.

17 Igor Sergeev. On monotone circuit complexity of threshold boolean functions. *Diskretnaya Matematika*, 32:81–109, 2020. `doi:10.4213/dm1547`.

18 Mathias Soeken, Heinz Riener, Winston Haaswijk, Eleonora Testa, Bruno Schmitt, Giulia Meuli, Fereshte Mozafari, and Giovanni De Micheli. The EPFL logic synthesis libraries, November 2019. arXiv:1805.05121v2.

19 Larry J. Stockmeyer. On the combinational complexity of certain symmetric boolean functions. *Mathematical Systems Theory*, 10:323–336, 1977. `doi:10.1007/BF01683282`.

A Blocks for the Modulo 3 Function

The following code justifies the existence of circuits needed in the proof of Theorem 1.

```
from itertools import product

enc = {(0, 0): 0, (0, 1): 1, (1, 0): 2, (1, 1): 2}

# in_2
for x1, x2 in product(range(2), repeat=2):
    g1 = x1 ^ x2
```

```
    g2 = x1 & x2
    assert (x1 + x2) % 3 == enc[g2, g1]

# in_3
for x1, x2, x3 in product(range(2), repeat=3):
    g1 = x1 == x2
    g2 = 1 - (x1 | x2)
    g3 = g2 == x3
    g4 = g1 == g3
    g5 = g2 < g4
    assert (x1 + x2 + x3) % 3 == enc[g5, g3]

# in_4
for x1, x2, x3, x4 in product(range(2), repeat=4):
    g1 = x1 == x2
    g2 = g1 ^ x3
    g3 = g2 ^ x2
    g4 = g1 & g3
    g5 = g4 == x4
    g6 = g2 == g5
    g7 = g4 < g6
    assert (x1 + x2 + x3 + x4) % 3 == enc[g7, g5]

# mid_3
for x1, x2, x3, z0, z1 in product(range(2), repeat=5):
    g1 = x1 == z1
    g2 = g1 | z0
    g3 = g2 ^ x2
    g4 = g3 ^ z0
    g5 = g4 ^ x1
    g6 = g2 & g5
    g7 = g6 == x3
    g8 = g3 == g7
    g9 = g6 < g8
    assert (enc[z0, z1] + x1 + x2 + x3) % 3 == enc[g9, g7]

# out_1^1
for x1, z0, z1 in product(range(2), repeat=3):
    g1 = x1 ^ z1
    g2 = z0 < g1
    assert ((enc[z0, z1] + x1) % 3 == 1) == g2

# out_2^0
for x1, x2, z0, z1 in product(range(2), repeat=4):
    g1 = z0 == x2
    g2 = x1 ^ z1
    g3 = g1 == x1
    g4 = z0 < g2
```

```
    g5 = 1 - (g3 | g4)
    assert ((enc[z0, z1] + x1 + x2) % 3 == 0) == g5

# out_3^2
for x1, x2, x3, z0, z1 in product(range(2), repeat=5):
    g1 = x1 == z1
    g2 = g1 | z0
    g3 = g2 ^ x2
    g4 = g3 ^ z0
    g5 = g4 ^ x1
    g6 = g2 & g5
    g7 = g3 == x3
    g8 = 1 - (g6 | g7)
    assert ((enc[z0, z1] + x1 + x2 + x3) % 3 == 2) == g8

# mod3_3^0
for x1, x2, x3 in product(range(2), repeat=3):
    g1 = x2 == x3
    g2 = x1 ^ x2
    g3 = g1 > g2
    assert ((x1 + x2 + x3) % 3 == 0) == g3

# mod3_3^2
for x1, x2, x3 in product(range(2), repeat=3):
    g1 = x2 ^ x3
    g2 = x3 | g1
    g3 = x1 < g2
    g4 = g1 ^ g3
    assert ((x1 + x2 + x3) % 3 == 2) == g4

# mod3_4^2
for x1, x2, x3, x4 in product(range(2), repeat=4):
    g1 = x1 ^ x2
    g2 = x3 ^ x4
    g3 = x1 ^ x3
    g4 = g2 | g3
    g5 = g1 < g4
    g6 = g2 ^ g5
    assert ((x1 + x2 + x3 + x4) % 3 == 2) == g6
```

ECCC ISSN 1433-8092

https://eccc.weizmann.ac.il