EXTENDING CC^0 CIRCUIT UPPER BOUNDS BEYOND SYMMETRIC FUNCTIONS

LUV UDESHI AND BRYNMOR CHAPMAN

MIT PRIMES-USA

ABSTRACT. The class $\mathsf{CC}^0[m]$ of constant-depth circuits built from unbounded-fan-in MOD_m gates reveals a remarkable landscape: when m is prime, computational ability is limited in small size, yet composite m unlocks unexpectedly rich capabilities. In this work, we illuminate and extend the true power of composite-modulus counting by generalizing Chapman and Williams' result for computing symmetric functions in CC^0, helping to support the fact that $\mathsf{CC}^0[m]$ is a versatile class capable of capturing complex structures with surprising efficiency. We additionally provide a Satisfiability Modulo Theories (SMT)-based framework for explicitly constructing and enumerating small $\mathsf{CC}^0[m]$ circuits, offering a computational tool for future research.

1. INTRODUCTION

Constant-depth counting circuits, which are built exclusively from unbounded-fan-in MOD_m gates, form the class $\mathsf{CC}^0[m]$, and their study has revealed a striking difference in computational power between prime and composite moduli. Classical results of Razborov and Smolensky [10, 11] showed that a MOD_p gate cannot be simulated by small $\mathsf{AC}^0[q]$ circuits when $p \neq q$ are prime, firmly establishing the hardness of prime-modulus counting in alternating circuits. In contrast, Barrington, Beigel and Rudich demonstrated that over composite modulus m, low-degree polynomials can capture significant counting power, and subsequent work by Barrington–Immerman–Straubing [3] and Chen–Papakonstantinou [6] gave depth-two $\mathrm{MOD}_a \circ \mathrm{MOD}_b$ simulations of AND.

More recently, Chapman and Williams [5] proved that every *symmetric* Boolean function, those which depend only on the number of 1s in the input, on n bits admits a depth-three

$$\mathrm{MOD}_{p_1} \circ \mathrm{MOD}_{p_2 \cdots p_r} \circ \mathrm{MOD}_{p_1}$$

circuit of size $\exp(O(n^{1/r} \log n))$, where $p_1, \ldots, p_r$ are distinct primes. Their results highlight the unexpectedly strong power of composite-modulus gates at constant depth, but leave open a natural question to further gauge the power of MOD_m circuits. What other, non-symmetric Boolean functions have similarly efficient CC^0 implementations? In this paper, we shed some light on this, as well as show minor improvements to their symmetric function construction.

The specific structure of the paper is as follows:

- In Section 2, we state some preliminary results that will guide what follows, heavily discussing symmetric function ideas from [5].
- In Section 3, we show a constant factor improvement in the symmetric function constructions of [5], as well as generalize their $\mathbb{Z}/m\mathbb{Z}$ polynomial detection result.

Date: February 3, 2026.

- In Section 4, we use the symmetric result of [5] and apply it two different ways. The first partitions the inputs into blocks of symmetry, yielding Theorem 4.2, while the second tweaks their results by applying the generalization from Section 3, yielding Theorem 4.3. We then combine both of these ideas, leading to Theorem 4.6, and explore some examples for how these results can be used.
- In Section 5, we compose symmetric functions with other natural classes of Boolean functions, to further analyze the scope of the power of MOD gates.
- In Section 6, we discuss directions for future research, including potential improvements and broader generalizations of our results, as well as provide a solver tool to facilitate further investigation of CC^0 circuits.
- Appendix A provides the full satisfiability-modulo-theories (SMT) encoding used to synthesize small CC^0 circuits, as described in Section 6.

2. Preliminaries

We will assume basic computational complexity [1], and start with a few relevant definitions and results that will be useful. We study *counting circuits*, which are composed of gates that count the inputs modulo a fixed modulus. More formal definitions follow.

Definition 2.1. For a positive integer m, a MOD_m *gate* is a gate that outputs 1 iff the sum of its input wires is an integer multiple of m. When m is irrelevant or implicitly understood, we may say simply MOD *gate*.

Definition 2.2. A MOD_m *circuit* is a circuit consisting entirely of MOD_m gates. When m is irrelevant or implicitly understood, we may say simply *counting circuit*.

We now note that although negation is not explicitly allowed by the definition, the inputs to MOD gates can be negated with little overhead.

Lemma 2.3 (Folklore)**.** *For any vector of n wires $\mathbf{x}$, any bit vector $\mathbf{v} \in \{0,1\}^n$ and any modulus m, $\mathrm{MOD}_m(\mathbf{x} \oplus \mathbf{v})$ can be computed with a MOD_m gate of fan-in at most $n + m + ||v||_1(m-2)$, where $\oplus$ denotes the bitwise XOR operation.*

Proof. For each i where $v_i = 0$, we use the input wire x_i as is. For each i where $v_i = 1$, we instead use m input wires: $m-1$ copies of x_i and a constant 1. Finally, if there are at least m constant 1 wires, we reduce the number of such wires modulo m. □

Similarly, the inputs (and output) of a $\sum \mathrm{MOD}_m$ gate can be similarly negated. For the remainder of this work, use of the above lemma will not always be explicitly noted.

Our next tool, proved in the 1800s, gives us a way to compute binomial coefficients modulo a prime number.

Theorem 2.4 (Lucas' Theorem [9])**.** *For non-negative integers n, k and a prime p,*

$$\binom{n}{k} \equiv \prod_{i=0}^{r} \binom{n_i}{k_i} \pmod{p}$$

where $n = (\overline{n_r n_{r-1} \dots n_0})_p$ and $k = (\overline{k_r k_{r-1} \dots k_0})_p$ are the base p representations of n, k.

This can be directly used for the polynomial representations of Boolean functions.

Lemma 2.5 (Barrington, Beigel, Rudich [2]). *Let p be a prime, let $n \in \mathbb{N}$, and let $e_i(\mathbf{x})$ denote the i-th elementary symmetric polynomial on n variables. For a binary vector $\mathbf{x}$, let*

$$\sum y_i \cdot p^i = |\mathbf{x}|_1$$

be the base p expansion of $|\mathbf{x}|_1$. Here, $|\mathbf{x}|_1$ is the number of 1s in the binary vector $\mathbf{x}$. Then for every i, $e_{p^i}(\mathbf{x}) \equiv y_i \mod p$.

Proof. Notice that $e_{p^i}(\mathbf{x}) = \binom{|\mathbf{x}|_1}{p^i}$. Since p^i is of the form $100\ldots0_p$, Lucas' Theorem can directly be applied to get the result. □

We look at past results relating to polynomial representations of various functions over $\mathbb{Z}_m$.

Theorem 2.6 (Hansen [8]). *If m is the product of r primes and k is smaller than each of the prime factors of m, then MOD_k can be represented by a degree $O(n^{1/r})$ polynomial over $\mathbb{Z}_m$.*

We now present a result that we will directly improve later.

Theorem 2.7 (Chapman, Williams [5]). *Let $m = p_1 \ldots p_k$ be a square-free integer. For every $n \in \mathbb{N}$ and every $T \in \{0, 1, \ldots, n\}$, there is a polynomial $P_T(x_1, \ldots, x_n)$ of degree*

$$\max_{j=1}^{k}\{p_j\}\sqrt[k]{n}$$

such that for all $a \in \{0,1\}^n$, $P_T(a) = 0 \bmod m$ if and only if $\sum_i a_i = T$.

We can follow ideas presented in BIS90 [3] to get depth-two MOD_m circuits for AND, which can be useful for constructing CC^0 circuits.

Proposition 2.8 ([3, 6]). *Let $a, b \geq 2$ be fixed integers with $\gcd(a, b) = 1$. Every AND of k MOD_b gates can be represented by an $\mathrm{MOD}_a \circ \mathrm{MOD}_b$ circuit of $O(b^k)$ gates. Furthermore, on all k-bit inputs, the sum of the inputs to the output gate of the circuit is always $0 \pmod a$ or $1 \pmod a$.*

Chapman and Williams noted that increasing the depth from two to just three gives a drastically stronger bound for general symmetric functions.

Theorem 2.9 (Chapman, Williams [5, Theorem 1.1]). *For every $\varepsilon > 0$, there is a modulus $m \leq (1/\varepsilon)^{2/\varepsilon}$ such that every symmetric function on n bits can be computed by depth-3 MOD_m circuits of $\exp(O(n^\varepsilon))$ size. In fact, the circuits have the form $\mathrm{MOD}_{p_1} \circ \mathrm{MOD}_{p_2 \cdots p_r} \circ \mathrm{MOD}_{p_1}$, where $p_1, \ldots, p_r$ are distinct primes.*

They then looked at varying the depth to $d \geq 3$, and getting an asymptotic size-depth tradeoff for large depths.

Theorem 2.10 (Chapman, Williams [5, Theorem 1.2]). *Let $d \geq 3$ be an integer, and let m be a product of $r \geq 2$ distinct primes. Then every symmetric function on n bits can be computed by depth-d MOD_m circuits of size $\exp(\tilde{O}(n^{1/(r+d-3)}))$.*

Theorem 2.11 (Chapman, Williams [5, Theorem 1.3])**.** *There is a constant $c \geq 1$ such that, for all sufficiently large depths d, and all composite m with r prime factors, every symmetric function can be computed by a MOD_m circuit of depth d and size $\exp\left(O\left(n^{c/((d-c)(r-1))}\right)\right)$.*

3. Polynomial Detectors of Boolean Functions

We begin by showing a constant improvement in the symmetric function construction described in [5].

Recall Theorem 2.7. We show the following stronger statement:

Theorem 3.1. *Let $m = p_1 \dots p_k$ be a square-free integer. For every $n \in \mathbb{N}$ and every $T \in \{0, 1, \dots, n\}$, there is a polynomial $P_T(x_1, \dots, x_n)$ of degree*

$$\max_{j=1}^{k} \left\{ \frac{p_j}{p_j - 1} \right\} \sqrt[k]{n}$$

such that for all $a \in \{0,1\}^n$, $P_T(a) = 0 \bmod m$ if and only if $\sum_i a_i = T$.

Proof. We follow much of the same structure as the proof of the original theorem. By Lemma 2.5, $e_{p^i}(a_1, \dots, a_n) \mod p$ is equal to the ith digit in the base p representation of $\sum_i a_i$.

Now, for each $p_r \mid m$, choose an integer t_r such that $p_r \cdot n^{1/k} \geq p_r^{t_r} > n^{1/k}$. Suppose when we write T in base p_r, the t_r lowest order digits are $b_r(t_r - 1), \dots, b_r(0)$. Then, for each p_r, we construct the polynomial

$$p_{p_r}(y_1, \dots, y_n) = 1 - \prod_{j=0}^{t_r - 1} (1 - (b_r(j) - e_{p_r{}^j}(y))^{p_r - 1}) \bmod p_r,$$

noting that the degree of p_{p_r} is $p_r^{t_r} - 1$. Now, here is where we make the constant factor improvement. We can reduce the degree of the p_{p_r} by directly multi-linearizing each of them. Explicitly, since for any Boolean variable x and positive integer k, we have $x^k = x$, we can reduce the power of any variable in the expansion of p_{p_r} to one, which will divide its degree by $p_r - 1$.

Note that for each p_r, we have that p_{p_r} will be zero modulo p_r precisely when the base p_r representation of $\sum a_i$ matches $b_r(t_r - 1), \dots, b_r(0)$ in the last t_r digits. Then, by the Chinese Remainder Theorem, $\sum a_i = T$ if and only if

$$\begin{aligned} \bigwedge_{r=1}^{k} \left(\sum_i a_i \equiv T \pmod{p_r^{t_r}} \right) &\iff \bigwedge_{r=1}^{k} \left(p_{p_r}(y) \equiv 0 \pmod{p_r} \right) \\ &\iff \sum_{r=1}^{k} \left(\tfrac{m}{p_r}\, p_{p_r}(y) \right) \equiv 0 \pmod{m}. \end{aligned}$$

This polynomial has degree $\max_{j=1}^{k} \left\{ \frac{p_j}{p_j - 1} \right\} \sqrt[k]{n}$ and satisfies the necessary condition. □

Applying this in the constructions for symmetric function CC^0 circuits, we can reduce the overall size of the circuit by the same constant factor.

We now generalize the statement of Theorem 2.7 beyond just symmetry.

Theorem 3.2. *Let $m = p_1 \dots p_k$ be a square-free integer. Let $I(x) \in \mathbb{Z}[x_1, \dots, x_n]$ of degree d, whose maximum value over all Boolean inputs is M. For every $n \in \mathbb{N}$ and every T lying in the range of I, there is a polynomial $P_T(x_1, \dots, x_n)$ of degree $O(d \cdot M^{1/k})$ such that for all $a \in \{0,1\}^n$, $P_T(a) = 0 \bmod m$ if and only if $I(a_1, \dots, a_n) = T$.*

Proof. We use a similar proof as in Theorem 2.7. Instead of the polynomials constructed there, we can construct

$$p_{p_r}(y_1, \dots, y_n) = 1 - \prod_{j=0}^{t_r - 1} \left(1 - \left(b_r(j) - \binom{I(y)}{p_r^j}\right)^{p_r - 1}\right) \bmod p_r$$

which has degree $O(d \cdot M^{1/k})$. Once again combining via Chinese Remainder Theorem gives us that the polynomial $P(y) = \sum_{1 \le r \le k} \left(\frac{m}{p_r} \cdot p_{p_r}(y)\right)$ is zero modulo m precisely when $I(y) = T$. □

4. Block Partitioning and Polynomial Invariants

In this section, we adapt the construction of Theorem 2.9 to extend to non-symmetric functions.

We start with an extension of the concept of a symmetric function. The symmetric group S_n acts naturally on the n-dimensional Boolean hypercube $\{0,1\}^n$ by

$$(x_1, x_2, \dots, x_n)^\sigma = (x_{1^\sigma}, x_{2^\sigma}, \dots, x_{n^\sigma}) .$$

This also induces an action on the set of n-variate Boolean functions given by

$$f^\sigma(x) = f\left(x^\sigma\right) .$$

Definition 4.1. The *symmetry group* $G(f)$ of a Boolean function $f : \{0,1\}^n \to \{0,1\}$ is its stabilizer $\mathrm{Stab}(f) := \{\sigma : f^\sigma = f\}$ under the above group action by S_n.

Note that f is symmetric iff its symmetry group is S_n. More generally, if $G(f)$ contains as a subgroup a large product of symmetric groups, then f can still be computed with a small depth-three MOD circuit.

Theorem 4.2. *Let $f\colon \{0,1\}^n \to \{0,1\}$ and suppose its symmetry group $G(f)$ contains a subgroup $H \cong S_{n_1} \times \cdots \times S_{n_r}$, where $\sum_{i=1}^r n_i = n$ and $L = \max n_i$. Then, for integer $k \ge 2$ and modulus $m = p_1 \cdots p_k$, there is a depth-3 MOD_m circuit for f of size*

$$\exp\bigl(O(r \log n + L^{1/k} \log L)\bigr).$$

Proof. We can decompose the input variables as a product of symmetric groups of the blocks. This means that the value of f depends on the bits $\{x_j : j \in B_i\}$ only through $w_i = \sum_{j \in B_i} x_j$. Then, f has a companion function $g : \{0, 1, \ldots, n_1\} \times \cdots \times \{0, 1, \ldots, n_r\} \to \{0, 1\}$ such that $f(x) = g(w_1, \ldots, w_r)$. Now, define $\mathscr{T} = g^{-1}(1)$ to be the set of r-tuples of integers $(T_1, \ldots, T_r)$ where g outputs 1. Using this, we can express f in disjunctive normal form as

$$f(x) = \bigvee_{T \in \mathscr{T}} \left[(w_1 = T_1) \wedge \cdots \wedge (w_r = T_r) \right].$$

To implement each conjunctive clause, we can use the exact construction as in Theorem 2.9 to get $\exp(O(n_i^{1/k} \log n_i))$ size circuits for block B_i. Now, using Proposition 2.8 allows us to eliminate the AND gates, while the OR gate can be replaced by a MOD gate, since it is simply an arithmetic sum always in $\{0, 1\}$ (as exactly one set of values can be true). Our size is the combination of $\sum_{i=1}^{r} (n_i + 1) \exp\left(O\left(n_i^{1/k} \log n_i\right)\right)$ gates for each block and $|\mathscr{T}| \cdot O(p_1^r) = \exp(O(r \log n))$ gates for the AND gadgets. This gives an overall size at most

$$\exp\left(O\left(r \log n + L^{1/k} \log L\right)\right).$$

□

An illustration of how this construction works is provided in Figure 1.

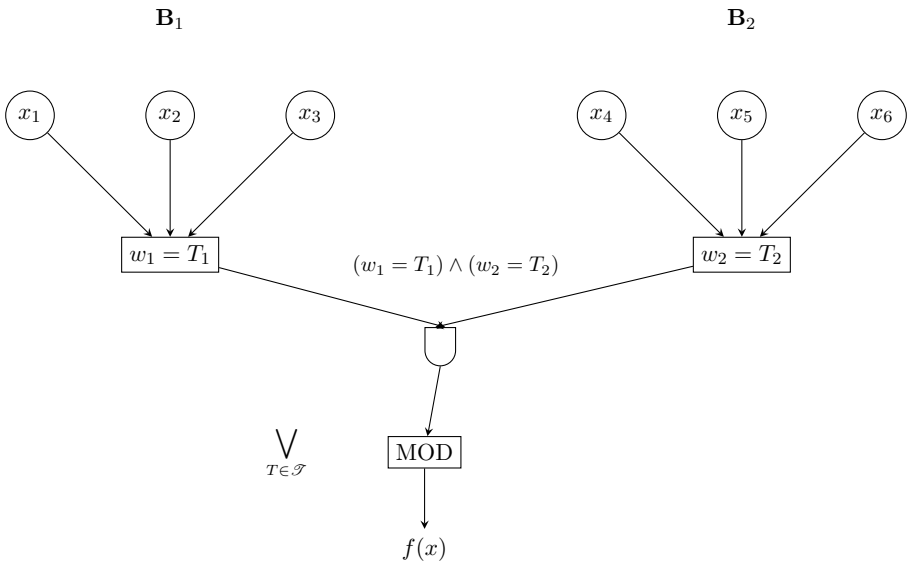

FIGURE 1. Sketch of the block-partitioning circuit from Theorem 4.2, showing how each block B_i checks $w_i = T_i$, the conjunction $(w_1 = T_1) \wedge (w_2 = T_2)$, and the final $\bigvee_{T \in \mathscr{T}}$ implemented by a MOD gate.

Our next idea is using different polynomial invariants. In particular, symmetric functions are those that depend only on the polynomial $\sum_{i=1}^{n} x_i$. If we have a function that is instead dependent only on the value of a different polynomial, we can compute it in a similar manner.

Theorem 4.3. *Let f be a Boolean function on n bits $x_1, \ldots, x_n$ and m be a product of $k \geq 2$ distinct primes. Let $I(x) \in \mathbb{Z}[x_1, \ldots, x_n]$ of degree d, whose maximum value over all Boolean inputs is M. Then, every f that depends solely on the value of $I(x)$ can be computed by depth-3 MOD_m circuits of size $\exp(O(d \cdot M^{1/k} \log M))$.*

Proof. Let $g\colon \{0, 1, \ldots, M\} \to \{0, 1\}$ be the companion of f, so that $f(x) = g\big(I(x)\big)$, and set $\mathscr{T} = \{T : g(T) = 1\}$. By Theorem 3.2, for each T in the range of I, there is a polynomial $P_T(x_1, \ldots, x_n)$ of degree $D = O\big(d\, M^{1/k}\big)$ such that

$$P_T(x) \equiv 0 \pmod{m} \iff I(x) = T.$$

Write

$$P_T(x) = \sum_{j=1}^{r_T} c_{T,j}\, M_{T,j}(x),$$

where each monomial $M_{T,j}(x)$ is an AND of at most D literals and $r_T \leq \exp(O(D))$.

The circuit for f has three layers of MOD_m gates as follows. The *output gate* is a MOD_{p_1} gate which

(a) sums over all $T \in \mathscr{T}$, and
(b) for each such T sums the terms $c_{T,1}M_{T,1}(x), \ldots, c_{T,r_T}M_{T,r_T}(x)$.

Since exactly one $T \in \mathscr{T}$ can satisfy $I(x) = T$, and in that case all the corresponding monomials sum to $P_T(x) \equiv 0 \pmod{m}$, the MOD_{p_1} sum is either 0 or 1 and in fact equals $f(x)$.

Each block of terms $\{c_{T,j}M_{T,j}(x)\}_{j=1}^{r_T}$ is computed in the *middle layer* by a single $\mathrm{MOD}_{m'}$ gate (with $m' = m/p_1$) summing those monomials. Finally, each bottom-layer monomial $M_{T,j}(x)$ is an AND of at most D input bits; by Proposition 2.8 (since $\gcd(p_1, m') = 1$) each such AND can be replaced by a small $\mathrm{MOD}_{p_1} \circ \mathrm{MOD}_{m'}$ subcircuit, exactly as in the proof of Theorem 2.9. Then, the resulting depth-3 MOD_m circuit has size

$$\exp\big(O(d\, M^{1/k} \log M)\big),$$

as claimed. □

Example 4.4. Consider $f(x) = \bigvee_{i=1}^{n}(x_i \wedge x_{i+1})$. Taking $I(x) = \sum_{i=1}^{n} x_i x_{i+1}$ gives us a degree 2 polynomial invariant on all the variables, and so we get depth-three MOD_m circuits of size $\exp(O(n^{1/k} \log n))$ computing f.

Example 4.5. We can compute the inversion-count threshold

$$f_{\mathrm{inv}\leq R}(x) \;=\; \big[\#\{\, i < j : x_i = 1 > x_j = 0\} \leq R\big] \;=\; \big[I(x) \leq R\big]$$

in $\exp\big(O(n^{2/k} \log n)\big)$ size by taking

$$I(x) \;=\; \sum_{1 \leq i < j \leq n} x_i\,(1 - x_j).$$

In fact, we can combine these results by partitioning the inputs into different invariant blocks, by using the circuits in Theorem 4.3 for each block from Theorem 4.2, rather than just the pure symmetric one.

Theorem 4.6. *Let $f\colon \{0,1\}^n \to \{0,1\}$ and suppose its symmetry group $G(f)$ contains a subgroup $H = H_1 \times \cdots \times H_r$, where each $H_i \leq S_{n_i}$ preserves an integer-valued polynomial $I_i(x_{i,1}, \ldots, x_{i,n_i}) \in \mathbb{Z}[x_{i,1}, \ldots, x_{i,n_i}]$ of degree d_i whose maximum value over all Boolean inputs is M_i. Let $L = \max n_i$. Then, for $k \geq 2$ and $m = p_1 \cdots p_k$, f can be computed by a depth-three MOD_m circuit of size*

$$\exp\Big(O\big(r \log n \;+\; \max_{1\leq i\leq r}(d_i\, M_i^{1/k}\, \log M_i)\big)\Big).$$

Essentially, the idea behind using this theorem is to strategically partition the input variables into various blocks, so that the value of the overall function only depends on each block. In particular, it should depend only on the value of a fixed polynomial on each block.

5. Composing Symmetric Functions

We now explore composing symmetry with other classes of functions, and seeing how to compute these using MOD_m circuits.

Lemma 5.1 ($\mathrm{SYM} \circ \mathrm{AND}$)**.** *Let $m = p_1 \cdots p_k$ be the product of the first k primes, and let $g : \{0,1\}^n \to \{0,1\}$ be symmetric. For each $1 \leq i \leq n$, let $h_i(x)$ be an* AND *of fan-in s. Then $f(x) := g\big(h_1(x), \ldots, h_n(x)\big)$ has a depth-5 MOD_m circuit of size*

$$\exp\big(O(n^{1/k} \log n + s\, k \log k)\big).$$

Proof. First, for each input, feed it and $p_1 - 1$ constant 1 wires into a MOD_{p_1} gate. We observe then that AND of s literals can be implemented by a two-layer modular gadget that replaces an s-input AND by a $\mathrm{MOD}_{m/p_1} \circ \mathrm{MOD}_{p_1}$ subcircuit of size $(m/p_1)^s$ via Proposition 2.8. Using the estimate $p_k = O(k \log k)$ one checks

$$\frac{m}{p_1} = \prod_{i=2}^{k} p_i \leq p_k^{k-1} = (O(k \log k))^{k-1},$$

so $(m/p_1)^s = \exp\big(O(s\, k \log k)\big)$. On the other hand, by Theorem 2.9, any symmetric g on n bits has a depth-3 MOD_m circuit of size $\exp(O(n^{1/k} \log n))$. Substituting each of the n AND-gadgets into the bottom layer and collapsing gives a depth-5 circuit whose size is

$$\exp\big(O(n^{1/k} \log n + s\, k \log k)\big).$$

□

Lemma 5.2 ($\mathrm{SYM} \circ \mathrm{OR}$)**.** *Let $m = p_1 \cdots p_k$ be the product of the first k primes and let $g : \{0,1\}^n \to \{0,1\}$ be symmetric. Let $h_i(x)$ be an* OR *of fan-in t for each i. Then $f(x) = g\big(h_1(x), \ldots, h_n(x)\big)$ has a depth-5 MOD_m circuit of size*

$$\exp\big(O(n^{1/k} \log n + t^{1/k} \log t)\big).$$

Proof. We first observe that each OR of t literals can be implemented by Theorem 2.9. Substituting these n OR-gadgets into the depth-3 MOD_m circuit for the symmetric function g of size $\exp\big(O(n^{1/k}\log n)\big)$ yields a depth-6 circuit of size

$$\exp\big(O(n^{1/k}\log n + t^{1/k}\log t)\big).$$

The third and fourth layers of MOD_{p_1} gates can then collapse, as the fourth is a sum mod p_1. This yields a depth-5 circuit with the same size. □

Theorem 5.3 ($\mathrm{SYM}\circ\mathrm{OR}\circ\mathrm{AND}$)**.** *Let $m = p_1\cdots p_k$ and let $g : \{0,1\}^n \to \{0,1\}$ be symmetric. Let $h_i(x)$ be a DNF with $O(t)$ terms each of width $O(s)$ for each i. Then $f(x) = g\big(h_1(x),\ldots,h_n(x)\big)$ admits a depth-7 MOD_m circuit of size*

$$\exp\big(O(n^{1/k}\log n + t^{1/k}\log t + s\,k\log k)\big).$$

Proof. Simply composing the circuit for Lemma 5.2 with the AND construction from Lemma 5.1, we get depth-7 circuits computing $\mathrm{SYM}\circ\mathrm{OR}\circ\mathrm{AND}$. □

Corollary 5.4. *Let g be an n-variate symmetric function. For each $1 \le i \le n$, let h_i be a DNF with polynomially many terms each of subpolynomial width. Then*

$$f(\mathbf{x}) := g(h_1(\mathbf{x}),\ldots,h_n(\mathbf{x}))$$

can be computed with counting circuits of depth seven and subexponential size.

Theorem 5.5 ($\mathrm{SYM}\circ\mathsf{NC}$)**.** *Let $m = p_1\cdots p_k$ be a product of k distinct primes, and let g be an n-variate symmetric function. For each $1 \le i \le n$, let h_i be a function that depends only on n^ϵ inputs. Then $f(\mathbf{x}) := g(h_1(\mathbf{x}),\ldots,h_n(\mathbf{x}))$ can be computed with depth-three MOD_m circuits of size $\exp(O(n^{2/k+\epsilon}\log m))$.*

Proof. For simplicity of notation, let $b = n^\epsilon$. As in Theorem 2.9, we represent f as a sum mod p (of fan-in $2^{O(n^{1/k}\log n)}$) of ANDs (of fan-in $O(n^{1/k})$) of MOD_{m/p_1} (of fan-in $2^{O(n^{1/k})}$) of ANDs (of fan-in $O(n^{1/k})$) of functions of fan-in b. The second layer ANDs can be immediately replaced via Proposition 2.8 to give a sum mod p_1 of fan-in $(m/p_1)^{O(n^{1/k})}$. We now observe that each AND in the fourth layer computes a function that depends only on $O(bn^{1/k})$ inputs. Using Proposition 2.8 again, these can be replaced with a sum mod m/p_1 of fan-in $O\left(p_1^{bn^{1/k}}\right)$. Collapsing the sum mod p_1 and sum mod m/p_1 into the layers above them, we get a $\mathrm{MOD}_{p_1}\circ\mathrm{MOD}_{m/p_1}\circ\mathrm{MOD}_{p_1}$ circuit of size

$$\exp(O(n^{2/k}\log n\log m + bn^{2/k}\log m)),$$

which is $\exp(O(n^{2/k+\epsilon}\log m))$. □

Corollary 5.6. *Let g be an n-variate symmetric function. For each $1 \le i \le n$, let h_i be a function that depends only on a subpolynomial number of inputs. Then*

$$f(\mathbf{x}) := g(h_1(\mathbf{x}),\ldots,h_n(\mathbf{x}))$$

can be computed with a counting circuit family of depth three and subexponential size.

Corollary 5.7. *Let g be an n-variate symmetric function. For each $1 \leq i \leq n$, let h_i be a function computable with an* NC *circuit family of depth $o(\log n)$. Then*

$$f(\mathbf{x}) := g(h_1(\mathbf{x}), \ldots, h_n(\mathbf{x}))$$

can be computed with a counting circuit family of depth three and subexponential size.

6. Future Work and SMT-Based Circuit Synthesis

Our work here further helps to show the power of CC^0 compared to initial belief, even at low depth levels. A natural direction for future work would be to consider computing other Boolean functions, which may not have the nice forms described in this paper, using MOD_m circuits. In addition, it would be helpful to find ways to more naturally characterize some of the results described, especially those from Section 4. Extending these ideas, another step that could be taken would be to apply similar ideas to compute these classes of functions in ACC^0, similar to what was done in [5]. Improving the explicit symmetric function construction itself would also be relevant, as that would immediately improve all these other constructions relying on that one.

6.1. SMT for Explicit Computation of Small $\mathsf{CC}^0[m]$ Circuits.

To experimentally explore the structure of small $\mathsf{CC}^0[m]$ circuits, we implemented an SMT-based synthesizer using the Z3 solver. For fixed input size n, modulus m, depth bound d, and gate budget s, the solver searches for a depth-d circuit of s MOD_m gates computing a given Boolean function exactly on all 2^n inputs.

Each input bit is treated as a depth-0 gate. Every additional gate g_i computes a modular linear form

$$g_i(x) = \Big[c_i + \sum_{k<i} c_{i,k}\, g_k(x) \equiv 0 \pmod{m}\Big],$$

with coefficients $c_{i,k} \in \{0, \ldots, m-1\}$. Correctness is enforced simultaneously for all inputs by equating the output gate with the target truth table.

The encoding uses several symmetry-breaking constraints that are essential for improving speed and scalability. First, each gate is assigned an explicit depth variable, enforcing acyclicity and bounding the circuit depth. Second, gates at the same depth are required to be lexicographically ordered by their coefficient vectors, eliminating permutation symmetry within layers. Third, multiplicative symmetry in modular equations is removed by requiring the first nonzero coefficient of each gate to lie in the set of proper divisors of m, yielding a canonical representative. Additional normalization constraints on constant terms further reduce equivalent encodings.

The solver is run incrementally over increasing gate budgets, guaranteeing minimality of the synthesized circuit when a solution is found. All arithmetic is performed using fixed-width bit-vectors, allowing efficient use of Z3's quantifier-free bit-vector engine.

While this approach is limited to small n, it provides concrete examples of minimal $\mathsf{CC}^0[m]$ circuits and may guide future analytic constructions. The full implementation is provided in Appendix A. Appendix A.1 contains code for extracting the solver's first satisfying model, while Appendix A.2 provides code for enumerating all circuits of a given size up to symmetry.

Acknowledgments

We would like to express gratitude to MIT PRIMES-USA for making this research experience possible.

References

[1] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*, Cambridge University Press, 2009.
[2] David A. Mix Barrington, Richard Beigel, and Steven Rudich. Representing Boolean functions as polynomials modulo composite numbers. *Comput. Complexity*, 4:367–382, 1994.
[3] David Barrington, Neil Immerman, and Howard Straubing. On uniformity within NC1. *Journal of Computer and System Sciences*, 41, 1990.
[4] David A. Mix Barrington, Howard Straubing, and Denis Therien. Non-uniform automata over groups. *Inf. Comput.*, 89(2):109–132, 1990.
[5] Brynmor Chapman and Ryan Williams. Smaller ACC0 circuits for symmetric functions. *arXiv preprint arXiv:2107.04706*, 2021.
[6] Shiteng Chen and Periklis A. Papakonstantinou. Depth reduction for composites. *SIAM J. Comput.*, 48(2):668–686, 2019.
[7] Vince Grolmusz and Gábor Tardos. Lower bounds for (MODp-MODm) circuits. *SIAM J. Comput.*, 29(4):1209–1222, 2000.
[8] Kristoffer Arnsfelt Hansen. On modular counting with polynomials. In *21st Annual IEEE Conference on Computational Complexity (CCC 2006)*, pages 202–212. IEEE Computer Society, 2006.
[9] Edouard Lucas. Sur les congruences des nombres eulériens et des coefficients différentiels des fonctions trigonométriques suivant un module premier. *Bulletin de la Société Mathématique de France*, 6:49–54, 1878.
[10] Alexander A. Razborov. Lower bounds on the size of bounded-depth networks over the complete basis with logical addition. *Mathematical Notes of the Academy of Sciences of the USSR*, 41(4):333–338, 1987.
[11] Roman Smolensky. Algebraic methods in the theory of lower bounds for Boolean circuit complexity. In STOC, pages 77–82, 1987.
[12] Andrew Chi-Chih Yao. On ACC and threshold circuits. In *FOCS*, pages 619–627, 1990.

Appendix A. Full Code for Computation of Small $\mathsf{CC}^0[m]$ Circuits

A.1. **First Model.** This code uses an SMT encoding to synthesize a minimum-size depth-bounded $\mathsf{CC}^0[m]$ circuit computing a specified Boolean function.

```
import time
from z3 import *

def adder_bitwidth(m):
    return max(2, (2 * m - 1).bit_length())

def mod_add(a, b, m_bv, w):
    s = a + b
    return If(UGE(s, m_bv), s - m_bv, s)

def mod_wsum(const_bv, coeffs, sel_bools, m_bv, w):
    zero = BitVecVal(0, w)
    acc = const_bv
    for c, sel in zip(coeffs, sel_bools):
        term = If(sel, c, zero)
        acc = mod_add(acc, term, m_bv, w)
    return acc

def prop_div(m):
    return [d for d in range(1, m) if m % d == 0]

def build_circ(f, m, max_s, max_depth):
    n = (len(f) - 1).bit_length()
    N = 1 << n
    assignments = [list(map(int, format(i, '0' + str(n) + 'b'))) for i in range(N)]
    w = adder_bitwidth(m)
    m_bv = BitVecVal(m, w)

    allowed_first_vals = prop_div(m)

    for s in range(4, max_s + 1):

        total = n + s
        max_d_cap = max_depth if max_depth is not None else total
        wd = max(2, (max_d_cap + 1).bit_length())
        zero_d = BitVecVal(0, wd)
        one_d  = BitVecVal(1, wd)
        max_dv = BitVecVal(max_d_cap, wd)

        solver = SolverFor("QF_BV")

        depth = [BitVec("d_" + str(i), wd) for i in range(total)]
        for i in range(n):
            solver.add(depth[i] == zero_d)
        if max_depth is not None:
            for i in range(total):
                solver.add(ULE(depth[i], max_dv))
```

```
const_conn = {}
connections = {}

def bv_in_range(x):
    return ULT(x, m_bv)

for i in range(n, total):
    c1 = BitVec("c1_" + str(i), w)
    const_conn[i] = c1
    solver.add(bv_in_range(c1))

    for k in range(i):
        c = BitVec("c_" + str(i) + "_" + str(k), w)
        connections[(i, k)] = c
        solver.add(bv_in_range(c))

        used = (c != BitVecVal(0, w))
        solver.add(Implies(used, UGE(depth[i], depth[k] + one_d)))

    solver.add(UGE(depth[i], one_d))

zero_w = BitVecVal(0, w)

def row_vec(i):
    base = [const_conn[i]] + [connections[(i, k)] for k in range(i)]
    need = total - len(base)
    if need > 0:
        base += [zero_w] * need
    return base

def lex_le(xs, ys):
    eq_so_far = BoolVal(True)
    lt_acc = BoolVal(False)
    for a, b in zip(xs, ys):
        lt_here = And(eq_so_far, ULT(a, b))
        lt_acc = Or(lt_acc, lt_here)
        eq_so_far = And(eq_so_far, a == b)
    return Or(lt_acc, eq_so_far)

rows = {i: row_vec(i) for i in range(n, total)}
for i in range(n, total):
    for j in range(i + 1, total):
        same_layer = (depth[i] == depth[j])
        solver.add(Implies(same_layer, lex_le(rows[i], rows[j])))

allowed_vals_bv = [BitVecVal(v, w) for v in allowed_first_vals]
for i in range(n, total):
    head = [const_conn[i]] + [connections[(i, k)] for k in range(i)]
    L = len(head)
    for t in range(L):
        if t == 0:
```

```
                    pre_zero = BoolVal(True)
                else:
                    pre_zero = And(*[head[p] == zero_w for p in range(t)])
                is_first_nonzero = And(pre_zero, head[t] != zero_w)
                solver.add(Implies(is_first_nonzero, Or(*[head[t] == v for v in
    allowed_vals_bv])))

        g = [[Bool("g_" + str(i) + "_" + str(j)) for j in range(N)] for i in range(total)]

        for j, bits in enumerate(assignments):
            for i in range(n):
                solver.add(g[i][j] == (bits[i] == 1))

        for i in range(n, total):
            coeffs_row = [connections[(i, k)] for k in range(i)]
            for j in range(N):
                sels = [g[k][j] for k in range(i)]
                total_mod = mod_wsum(const_conn[i], coeffs_row, sels, m_bv, w)
                solver.add(g[i][j] == (total_mod == BitVecVal(0, w)))

        for j in range(N):
            solver.add(g[total - 1][j] == (f[j] == 1))

        if solver.check() == sat:
            return solver.model(), s

    return None, None


# execution for specific function:
# function's truth table goes here
f = [int(bin(i).count("1") % 4 == 0) for i in range(1 << 7)] #MOD4 function on 7 bits

m = 5
max_s = 5
max_d = 2

start_time = time.time()
model, s = build_circ(f, m, max_s, max_d)
end_time = time.time()
elapsed = end_time - start_time

if not model:
    print("no circuit for size<=" + str(max_s) + " and depth<=" + str(max_d))
    exit(1)

print("circuit with " + str(s) + " gates (depth<=" + str(max_d) + ")")
print("elapsed time: {:.3f} seconds".format(elapsed))

n = (len(f) - 1).bit_length()
total_gates = n + s
```

```
connections = {}
const_conn = {}
for i in range(n, total_gates):
    const_conn[i] = BitVec("c1_" + str(i), adder_bitwidth(m))
    for k in range(i):
        connections[(i, k)] = BitVec("c_" + str(i) + "_" + str(k), adder_bitwidth(m))

depth = [BitVec("d_" + str(i), max(2, (max_d + 1).bit_length())) for i in range(total_gates)]

print("\ninputs:")
for i in range(n):
    print("  gate " + str(i) + ": input x" + str(i))

print("\nMOD gates:")
for i in range(n, total_gates):
    parts = []
    mult1 = model.eval(const_conn[i]).as_long()
    if mult1 > 0:
        if mult1 == 1:
            parts.append("1")
        else:
            parts.append(str(mult1) + "*1")
    for k in range(i):
        mult = model.eval(connections[(i, k)]).as_long()
        if mult > 0:
            if mult == 1:
                parts.append("g" + str(k))
            else:
                parts.append(str(mult) + "*g" + str(k))
    if parts:
        driver_str = ", ".join(parts)
    else:
        driver_str = "-"
    d_val = model.eval(depth[i]).as_long()
    print("  Gate " + str(i) + " (depth " + str(d_val) + "): MOD_" + str(m) + "(" +
    driver_str + ")")

final_depth = model.eval(BitVec("d_" + str(total_gates - 1), max(2, (max_d + 1).bit_length())
    )).as_long()
print("\noutput at gate " + str(total_gates - 1) + " with depth " + str(final_depth))
```

A.2. **Enumeration.** This code extends the SMT encoding to enumerate all depth-bounded $\mathsf{CC}^0[m]$ circuits of a fixed size computing a given Boolean function, up to symmetry.

```
import time
from z3 import *

def adder_bitwidth(m):
    return max(2, (2 * m - 1).bit_length())

def mod_add(a, b, m_bv, w):
    s = a + b
```

```
    return If(UGE(s, m_bv), s - m_bv, s)

def mod_wsum(const_bv, coeffs, sel_bools, m_bv, w):
    zero = BitVecVal(0, w)
    acc = const_bv
    for c, sel in zip(coeffs, sel_bools):
        term = If(sel, c, zero)
        acc = mod_add(acc, term, m_bv, w)
    return acc

def prop_div(m):
    return [d for d in range(1, m) if m % d == 0]

def lex_le(xs, ys):
    eq_so_far = BoolVal(True)
    lt_acc = BoolVal(False)
    for a, b in zip(xs, ys):
        lt_here = And(eq_so_far, ULT(a, b))
        lt_acc = Or(lt_acc, lt_here)
        eq_so_far = And(eq_so_far, a == b)
    return Or(lt_acc, eq_so_far)

def row_vec(i, total, const_conn, connections, zero_w):
    base = [const_conn[i]] + [connections[(i, k)] for k in range(i)]
    need = total - len(base)
    if need > 0:
        base += [zero_w] * need
    return base

def canonical_forms_by_depth(model, n, total, m, max_depth):
    w  = adder_bitwidth(m)
    wd = max(2, (((max_depth if max_depth is not None else total)) + 1).bit_length())

    const_conn = {}
    connections = {}
    for i in range(n, total):
        const_conn[i] = BitVec(f"c1_{i}", w)
        for k in range(i):
            connections[(i, k)] = BitVec(f"c_{i}_{k}", w)
    depth = [BitVec(f"d_{i}", wd) for i in range(total)]

    depth_vals = [model.eval(d).as_long() for d in depth]

    max_d_in_model = max(depth_vals) if total > 0 else 0
    gates_at_depth = {d: [] for d in range(max_d_in_model + 1)}
    for i in range(n, total):
        gates_at_depth[depth_vals[i]].append(i)

    gate_cf = {}

    def coef(i, k):
        return model.eval(connections[(i, k)]).as_long()
```

```
    def const(i):
        return model.eval(const_conn[i]).as_long()

    for d in range(1, max_d_in_model + 1):

        layer_items = []
        for i in gates_at_depth.get(d, []):
            c0 = const(i)

            input_coeffs = []
            for k in range(n):
                val = coef(i, k)
                if val != 0:
                    input_coeffs.append(val)
            input_coeffs.sort()

            ref_items = []
            for k in range(n, i):
                val = coef(i, k)
                if val == 0:
                    continue
                src = k
                ref_items.append((val, gate_cf[src]))

            ref_items.sort()

            cf = (d, c0, tuple(input_coeffs), tuple(ref_items))
            layer_items.append((i, cf))

        for i, cf in layer_items:
            gate_cf[i] = cf

    return depth_vals, gate_cf

def circuit_fingerprint_perm_invariant(model, n, total, m, max_depth):
    depth_vals, gate_cf = canonical_forms_by_depth(model, n, total, m, max_depth)
    all_cfs = [gate_cf[i] for i in range(n, total)]
    all_cfs.sort()
    return tuple(all_cfs)

def enumerate_minimal_circuits(f, m, max_s, max_depth=2, max_models=None):
    n = (len(f) - 1).bit_length()
    N = 1 << n
    assignments = [list(map(int, format(i, '0' + str(n) + 'b'))) for i in range(N)]

    w = adder_bitwidth(m)
    m_bv = BitVecVal(m, w)
    allowed_first_vals = prop_div(m)

    for s in range(1, max_s + 1):
        total = n + s
```

```
    max_d_cap = max_depth if max_depth is not None else total
    wd = max(2, (max_d_cap + 1).bit_length())
    zero_d = BitVecVal(0, wd)
    one_d  = BitVecVal(1, wd)
    max_dv = BitVecVal(max_d_cap, wd)

    solver = SolverFor("QF_BV")

    depth = [BitVec(f"d_{i}", wd) for i in range(total)]
    for i in range(n):
        solver.add(depth[i] == zero_d)
    if max_depth is not None:
        for i in range(total):
            solver.add(ULE(depth[i], max_dv))

    const_conn = {}
    connections = {}

    def bv_in_range(x):
        return ULT(x, m_bv)

    for i in range(n, total):
        c1 = BitVec(f"c1_{i}", w)
        const_conn[i] = c1
        solver.add(bv_in_range(c1))

        for k in range(i):
            c = BitVec(f"c_{i}_{k}", w)
            connections[(i, k)] = c
            solver.add(bv_in_range(c))
            used = (c != BitVecVal(0, w))
            solver.add(Implies(used, UGE(depth[i], depth[k] + one_d)))

        solver.add(UGE(depth[i], one_d))

    zero_w = BitVecVal(0, w)

    rows = {i: row_vec(i, total, const_conn, connections, zero_w) for i in range(n, total
)}
    for i in range(n, total):
        for j in range(i + 1, total):
            same_layer = (depth[i] == depth[j])
            solver.add(Implies(same_layer, lex_le(rows[i], rows[j])))

    allowed_vals_bv = [BitVecVal(v, w) for v in allowed_first_vals]
    for i in range(n, total):
        head = [const_conn[i]] + [connections[(i, k)] for k in range(i)]
        L = len(head)
        for t in range(L):
            pre_zero = BoolVal(True) if t == 0 else And(*[head[p] == zero_w for p in
range(t)])
            is_first_nonzero = And(pre_zero, head[t] != zero_w)
```

```
            solver.add(Implies(is_first_nonzero, Or(*[head[t] == v for v in
allowed_vals_bv])))

    g = [[Bool(f"g_{i}_{j}") for j in range(N)] for i in range(total)]

    for j, bits in enumerate(assignments):
        for i in range(n):
            solver.add(g[i][j] == (bits[i] == 1))

    for i in range(n, total):
        coeffs_row = [connections[(i, k)] for k in range(i)]
        for j in range(N):
            sels = [g[k][j] for k in range(i)]
            total_mod = mod_wsum(const_conn[i], coeffs_row, sels, m_bv, w)
            solver.add(g[i][j] == (total_mod == BitVecVal(0, w)))

    for j in range(N):
        solver.add(g[total - 1][j] == (f[j] == 1))

    unique_models = []
    seen_fps = set()

    block_vars = []
    for i in range(n, total):
        block_vars.append(const_conn[i])
        for k in range(i):
            block_vars.append(connections[(i, k)])
    block_vars += [depth[i] for i in range(total)]

    while solver.check() == sat:
        model = solver.model()
        fp = circuit_fingerprint_perm_invariant(model, n, total, m, max_depth)
        if fp not in seen_fps:
            seen_fps.add(fp)
            unique_models.append(model)

        diseqs = []
        for v in block_vars:
            try:
                val = model.eval(v, model_completion=True)
            except Z3Exception:
                continue
            diseqs.append(v != val)
        if not diseqs:
            break
        solver.add(Or(*diseqs))

        if max_models is not None and len(unique_models) >= max_models:
            break

    if unique_models:
        return unique_models, s
```

```
    return [], None

def print_circuit_from_model(model, f, m, s, max_depth):
    n = (len(f) - 1).bit_length()
    total_gates = n + s
    w = adder_bitwidth(m)
    wd = max(2, (((max_depth if max_depth is not None else total_gates)) + 1).bit_length())

    const_conn = {}
    connections = {}
    for i in range(n, total_gates):
        const_conn[i] = BitVec(f"c1_{i}", w)
        for k in range(i):
            connections[(i, k)] = BitVec(f"c_{i}_{k}", w)
    depth = [BitVec(f"d_{i}", wd) for i in range(total_gates)]

    print("\ninputs:")
    for i in range(n):
        print(f"  gate {i}: input x{i}")

    print("\nMOD gates:")
    for i in range(n, total_gates):
        parts = []
        mult1 = model.eval(const_conn[i]).as_long()
        if mult1 > 0:
            parts.append("1" if mult1 == 1 else f"{mult1}*1")
        for k in range(i):
            mult = model.eval(connections[(i, k)]).as_long()
            if mult > 0:
                parts.append(f"g{k}" if mult == 1 else f"{mult}*g{k}")
        driver_str = ", ".join(parts) if parts else "-"
        d_val = model.eval(depth[i]).as_long()
        print(f"  Gate {i} (depth {d_val}): MOD_{m}({driver_str})")

    final_depth = model.eval(depth[total_gates - 1]).as_long()
    print(f"\noutput at gate {total_gates - 1} with depth {final_depth}")

#execution for specific function
# function's truth table goes here
f = [int(bin(i).count("1") % 4 == 0) for i in range(1 << 8)]  # MOD4 function on 8 bits

m = 5
max_s = 5
max_d = 2

start_time = time.time()
models, s = enumerate_minimal_circuits(f, m, max_s, max_depth=max_d, max_models=None)
end_time = time.time()
elapsed = end_time - start_time

if not models:
```

```
    print("no circuit for size<=" + str(max_s) + " and depth<=" + str(max_d))
    exit(1)

print("Found " + str(len(models)) + " unique circuit(s) of minimal size " + str(s) +
      " (depth<=" + str(max_d) + ")")
print("elapsed time: {:.3f} seconds".format(elapsed))

for idx, model in enumerate(models, 1):
    print("\n=== Circuit #" + str(idx) + " ===")
    print_circuit_from_model(model, f, m, s, max_d)
```

Thomas Jefferson High School for Science and Technology
Email address: luv.udeshi@gmail.com

Massachusetts Institute of Technology
Email address: brynmor@mit.edu