Topics in Circuit Complexity **(CS354, Fall'11)**

Week 1: An Overview of Circuit Complexity

Lecture Notes for 9/27 and 9/29

Ryan Williams

1 Welcome

The area of circuit complexity has a long history, starting in the 1940's. It is full of open problems and frontiers that seem insurmountable, yet the literature on circuit complexity is fairly large. There is much that we do know, although it is scattered across several textbooks and academic papers. I think now is a good time to look again at circuit complexity with fresh eyes, and try to see what can be done.

2 Preliminaries

An n-*bit Boolean function* has domain $\{0,1\}^n$ and co-domain $\{0,1\}$. At a high level, the basic question asked in circuit complexity is: *given a collection of "simple functions" and a target Boolean function f, how efficiently can f be computed (on all inputs) using the simple functions?* Of course, efficiency can be measured in many ways. The most natural measure is that of the "size" of computation: how many copies of these simple functions are necessary to compute f?

Let $\mathcal{B}$ be a set of Boolean functions, which we call a *basis set*. The *fan-in* of a function $g \in \mathcal{B}$ is the number of inputs that g takes. (Typical choices are fan-in 2, or *unbounded* fan-in, meaning that g can take any number of inputs.)

We define a circuit C with n inputs and size s over a basis $\mathcal{B}$, as follows. C consists of a directed acyclic graph (DAG) of $s+n+2$ nodes, with n sources and one sink (the sth node in some fixed topological order on the nodes). The nodes are often called *gates*. For $i = 1, \ldots, n$, the ith gate in topological order is a source associated with the ith input bit. (These nodes are often called *input gates*.) The gates numbered $j = n+1, \ldots, s$ in topological order are each labeled with a function f_j from $\mathcal{B}$. Each function f_j has fan-in equal to the indegree of j. We'll sometimes call such a C a $\mathcal{B}$-*circuit*, for short.

Evaluating a circuit C on an n-bit input $x = (x_1, \ldots, x_n) \in \{0,1\}^n$ is done as follows. The *value on* x of the ith gate is defined to be x_i. For $j = n+1, \ldots, s$, the *value on* x of the jth

gate is $v_j = f_j(v_{i_1}, \ldots, v_{i_k})$, where $i_1, \ldots, i_k < j$ are those nodes with edges to node j (listed in topological order), v_{i_ℓ} is the value of node i_ℓ, and f_j is the function assigned to node j. The *output of* C is the value of the sth gate on x. We often write $C(x)$ to refer to the output of C on x. A circuit computes a function $f : \{0,1\}^n \to \{0,1\}$ if for all n-bit strings x, $f(x) = C(x)$.

An essentially analogous circuit definition also places the constants 0 and 1 in the circuit: we may add two more sources v_0 and v_1 to the DAG, which are defined to always take the values 0 and 1 (respectively). These values can then be fed into the other gates in an arbitrary way. When the basis $\mathcal{B}$ can simulate OR, AND, and NOT, these constants can be easily simulated by having a gate computing the function $ZERO = (x \wedge \neg x)$ and a gate computing $ONE = (x \vee \neg x)$ for an arbitrary input x. So for all the basis sets we'll consider, the difference in circuit size between the two models is only an additive factor.

2.1 Complexity Measures

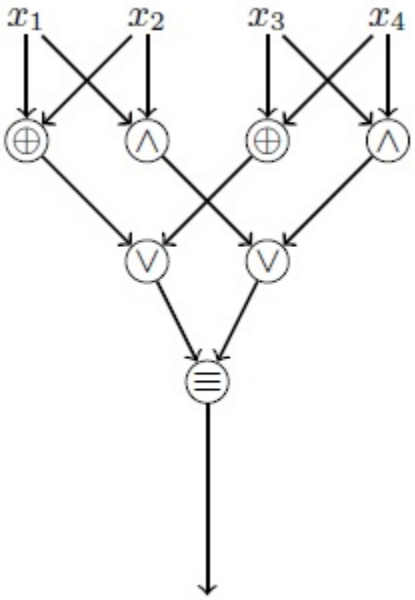
We sometimes use $size(C)$ to refer to the size s of C.

Size roughly measures the amount of time that would be needed to evaluate the circuit, if we used a single processor for circuit evaluation. However, if we could devote arbitrarily many processors for evaluating a circuit, then size is not the bottleneck for running time. The *depth* of a circuit is the longest path from any source to the sink. This measure would be more appropriate in measuring the time efficiency of a circuit which is being evaluated over many processors.

Another interesting measure of complexity is the *wire-complexity* of a circuit, which is the total number of edges in the DAG. Note that as long as the indegree of each node is bounded by a constant (i.e., the fan-in of each gate is constant), the wire-complexity and the size are related by constant factors. (For indegree at most d, the wire-complexity is at most d times the size.) So in most cases we'll study, the wire complexity is approximately the circuit size. But sometimes we will study circuits where the gates have unbounded fan-in, and there it is easy to have complex circuits with few gates and many wires.

2.2 An Example

Consider the MOD3 function on 4 bits: $MOD3(x_1, x_2, x_3, x_4) = 1 \iff \sum_i x_i = 0 \pmod 3$. The function is 1 when the sum of bits is either 3 or 0. Here is an *optimal* size circuit for computing MOD3 on 4 bits where the circuit is over all 2-bit functions, where $\oplus$ is the XOR of two bits and $\equiv$ is the negation of XOR (the EQUALS function):



The circuit was found by Kojevnikov, Kulikov, Yaroslavtsev 2009 using a SAT solver, and the solver verified that there is no smaller circuit computing this function. Pretty awesome.

2.3 Circuit Complexity

Given a Boolean function f and a basis set $\mathcal{B}$, we define $C_{\mathcal{B}}(f)$ to be the minimum size over all $\mathcal{B}$-circuits that compute f. The quantity $C_{\mathcal{B}}(f)$ is called the *circuit complexity* of f over $\mathcal{B}$.

What possible $\mathcal{B}$'s are interesting? For circuits with bounded fan-in, there are two which are generally studied. (For circuits with unbounded fan-in, there are many more, as we'll see.) One basis used often is B_2, the set of all $2^{2^2} = 16$ Boolean functions on two bits. (In general, B_k refers to the set of all functions $f : \{0,1\}^k \to \{0,1\}$.) This makes the circuit complexity problem very natural: *how efficiently can we build a given $f \in B_n$ from functions in B_2?*

The next most popular basis is U_2, which informally speaking, uses only AND, OR, and NOT gates, where NOT gates are "for free" (NOT gates are not counted towards size).

More formally, for a variable x, define $x^1 = x$ and $x^0 = \neg x$. Define the set

$$U_2 = \{f(x,y) = (x^{b_1} \wedge y^{b_2})^{b_3} \mid b_1, b_2, b_3 \in \{0,1\}\}.$$

U_2 consists of 8 different functions. When we have constants 0 and 1 built-in to the circuit, we can get more functions out of U_2 without any trouble. By plugging constants into functions from U_2, we can obtain $f(x,y) = x$, $f(x,y) = \neg x$, $f(x,y) = y$, $f(x,y) = \neg y$, $f(x,y) = 0$,

and $f(x, y) = 1$. That's six more functions, so we're up to 14. The only two functions we're missing from B_2 is

$$XOR(x, y) = (x \wedge \neg y) \vee (\neg x \wedge y)$$

and its complement, $EQUALS(x, y) = \neg XOR(x, y)$.

Clearly for every f, $C_{B2}(f) \leq C_{U2}(f)$, because B_2 only contains more functions than U_2. From the above discussion, we also have:

Proposition 1. *For all f, $C_{U_2}(f) = C_{B_2 - \{XOR, EQUALS\}}(f)$.*

Proof. Because every function from U_2 is in $B_2 - \{XOR, EQUALS\}$, we have $C_{U_2}(f) \geq C_{B_2-\{XOR,EQUALS\}}(f)$. Every function from $B_2 - \{XOR, EQUALS\}$ can be simulated with a function from U_2 (using built-in constants), so $C_{U_2}(f) \leq C_{B_2-\{XOR,EQUALS\}}(f)$: given a minimum size circuit over $B_2 - \{XOR, EQUALS\}$ we can get a circuit of the same size over U_2. □

Proposition 2. *For all f, $C_{U_2}(f) \leq 3C_{B_2}(f)$.*

Proof. To get a U_2-circuit from a B_2-circuit, we only have to replace the XOR and NOT(XOR) gates with U_2 functions. As

$$EQUALS(x, y) = (x \wedge y) \vee (\neg x \wedge \neg y),$$

hence we can simulate EQUALS with 3 gates from U_2. Taking the negation of this we also get XOR. □

The above shows that caring about whether you use U_2 or B_2 only makes sense if you care about constant leading factors, since the two circuit complexities within (small) constant factors of each other. Such linear factors could make a difference though, if the circuit complexity were only a linear function in n, i.e. $C_{B2} = cn$. (For many functions, this *is* unfortunately the best we know!)

A more general result can be shown. A basis $\mathcal{B}$ is defined to be *complete* if for every Boolean function, there exists a Boolean circuit over $\mathcal{B}$ that computes f. (For example $\mathcal{B} = \{AND\}$ is not a basis, while $\mathcal{B} = \{NOR\}$ is a basis.) We can see that B_2 and U_2 are complete, because we can write any function f in disjunctive normal form (DNF), as a huge OR of $O(2^n)$ ANDs, where each AND outputs 1 on exactly one input of f. So writing f as a "tree" of up to 2^n ORs, where each leaf has n ANDs of variables and their negations, we can get a circuit for f over U_2 with size $O(n2^n)$.

Theorem 2.1. *Let $\mathcal{B}$ be any* complete *basis of Boolean functions with constant fan-in. Then for all Boolean functions f, $C_{\mathcal{B}}(f) = \Theta(C_{U_2}(f)) = \Theta(C_{B_2}(f))$.*

Here the constant factors depend on $\mathcal{B}$. The proof is simple: if a basis $\mathcal{B}$ is complete, then we can write all the functions from U_2 as small circuits over the basis $\mathcal{B}$. Each of these circuits has constant size since the number of inputs to U_2 is two. Conversely, every function from $\mathcal{B}$ can be written as an DNF, which can then be converted to a constant-size U_2-circuit (due to the constant fan-in of the functions in $\mathcal{B}$).

So we see a certain strong "robustness" with circuit complexity: because the complexity only changes by small constant factors in front, choose your favorite complete basis, and work with that. Maybe you like all your circuits to be composed of only NOR gates. That's fine up to constant factors.

3 Why Study Circuit Complexity?

Here are four good reasons.

3.1 Reason $\#1$: Complexity of Finite Functions

Circuit complexity is a sensible difficulty measure for functions that take only finitely many inputs (you know, the functions we see in reality). Let INTEGER-FACTORING$_{1024}$ be the problem of finding a prime factorization of a given 1024-bit integer n. (We can make this a function that outputs a single bit, by having the input be $\langle n, i \rangle$ and changing the objective to outputting the ith bit of the prime factorization under some reasonable encoding.) After the encodings of numbers and factorizations have been fixed, it is sensible to ask: *what is* $C_{U_2}(\text{INTEGER-FACTORING}_{1024})$*?* Is it at most 10^{10}? That would be amazing – integer factorization *would* be practical, given the appropriate circuit. There would be some circuit out there which, if we can just discover it and build it, we'd be able to factor without much trouble (using today's technology).

Maybe instead the circuit complexity is at least 10^{80}? That would be a huge step towards provable security! (For provable security, we'd also need further strengthenings, e.g., that there is no circuit of small size that can factor a sufficiently large fraction of all 1024-bit integers.)

So the chief advantage of circuit complexity is that you can talk directly about finite-input problems and say very meaningful things. We cannot achieve such understanding of finite problems by studying things like the time complexity of Turing machines, because every finite problem (i.e., having only finitely many inputs) can be solved in "linear time" on a Turing machine, by just embedding a huge lookup table for each instance of the problem in the transition table of the Turing machine.

Circuit complexity is more fine-grained than this. We are studying not only the "computation time" of the algorithm but also the "program size" of algorithms that run on n-bit instances.

However, we don't wish to go too far with this, and suggest that the usual "uniform" time/space complexity notions are inferior to circuit complexity in the long run. On the contrary, it is extremely useful to have finite programs that will work for huge inputs now and even larger inputs in the future. So we'll still talk a lot about problems in time/space complexity and relate them to circuit complexity.

Here is a concrete circuit complexity lower bound.

Theorem 3.1 (Meyer-Stockmeyer)**.** *There is a natural logic problem* Π *such that every Boolean circuit over* B_2 *that decides every sentence over* Π *on all* 3660*-bit instances requires size at least* 10^{125}.

(Incidentally, Π is the decision problem for sentences in the theory WS1S, or "Weak Monadic Second-Order Theory of 1 Successor". Very briefly, each sentence can contain first-order variables $x_1, ..., x_n$ that range over the natural numbers, and "monadic" second-order variables $S_1, ..., S_k$ that are allowed to range over all subsets of variables. The vocabulary constants 0 and 1, the $+$ function, the relation $\leq$ on the first-order variables, and the equality relation on the second-order variables. The problem is, given a sentence ϕ over some quantified first-order and second-order variables with the above vocabulary, is ϕ true or not?)

3.2 Reason #2: Circuit Complexity Lower Bounds Imply Complexity Class Separations

If there are "efficient enough" functions that still require large circuits, then we can separate complexity classes of languages. Although circuits are defined to compute finite functions, we can also use them to compute infinite languages (like in the usual model of complexity theory), by giving a new circuit for each input length. So when we talk about circuits computing infinite languages (like SAT, TSP, etc.) we'll give an infinite family of circuits $\{C_n\}$ where C_n works on n-bit inputs.

For a language $L \subseteq \{0,1\}^\star$, let $L_n = \{0,1\}^n \cap L$. Let $s : \mathbb{N} \to \mathbb{N}$ and $d : \mathbb{N} \to \mathbb{N}$ be functions.

Definition 1. *A language* $L \subseteq \{0,1\}^\star$ *has* $s(n)$*-size* $d(n)$*-depth circuits if for all* n*, there is a circuit* C_n *such that* $size(C_n) \leq s(n)$*,* $depth(C_n) \leq d(n)$*, and for all* x*,* $C_n(x) = 1 \iff x \in L_n$.

A language L has polynomial size circuits if L has $p(n)$-size circuits for some polynomial $p(n)$. Having infinitely many circuits computing a language effectively gives us an "infinite" computational model.

Can we show that presumably hard problems like TSP and SAT do not have polynomial-size circuits? This is *only harder* than the P versus NP question:

Theorem 3.2. *If there is a language $L \in \mathsf{NP}$ such that L does not have n^k size circuits for every k, then $\mathsf{P} \neq \mathsf{NP}$.*

An analogous statement holds for PSPACE. The theorem follows directly from the fact that P has polynomial-size circuits.

3.3 Reason #3: Circuit Complexity *Upper Bounds* Imply Complexity Class Separations

If "hard enough" functions have small circuits, then we can again we can separate complexity classes.

Theorem 3.3. *If every language in EXP has polynomial-size circuits, then $\mathsf{P} \neq \mathsf{NP}$.*

Observe that, if we could prove *if every language in* PSPACE *has polysize circuits, then* $\mathsf{P} \neq \mathsf{PSPACE}$ then we would have proved $\mathsf{P} \neq \mathsf{PSPACE}$ unconditionally.

3.4 Reason #4: Circuit Complexity Lower Bounds Imply Universal Derandomization

Given a hard function in the form of its truth table, we can "de-randomize" every randomized algorithm: for every randomized algorithm A we can run an algorithm that uses the hard function to efficiently simulate A without randomness. Here is one amazing theorem from a few years back:

Theorem 3.4 (Umans'05)**.** *For every k, there is a polytime computable function $G : \{0,1\}^{2^m} \times \{0,1\}^m \to \{0,1\}^{m^k}$ such that, if f is an m-bit Boolean function which doesn't have circuits of size m^{3k}, then for every circuit C of size m^k (and at most m^k inputs), $|Pr_y[C(y) = 1] - Pr_x[C(G(f,x)) = 1]| < 1/m^k$.*

That is, assuming f is *hard*, G is a pseudorandom generator that "fools" m^k-size circuits: the output of $G(f,x)$ over all m-bit x "looks random" to all m^k-size circuits. So if we want to deterministically simulate a randomized algorithm that uses m^k random bits, we can convert that algorithm into a circuit, then run it on the output of $G(f,x)$ over all x of m bits. This takes only $O(2^m \text{poly}(m^k))$ time, rather than the $2^{O(m^k)}$ time of exhaustive search over all random bits. So we'd get a subexponential time simulation of randomized algorithms:

Theorem 3.5. *If there is a function in $\mathsf{TIME}[2^{O(n)}]$ that doesn't have polynomial size circuits, then BPP can be simulated in 2^{n^ε} time on infinitely many input lengths, for all $\varepsilon > 0$.*

Notice the parameters are not like those of pseudorandom generators used in cryptography. The running time of the generator G is high – higher than the circuit size. Exponential-size circuit lower bounds for exponential time would imply an even stronger consequence:

Theorem 3.6 (Impagliazzo-Wigderson '97)**.** *: If there is a language $L \in \mathsf{TIME}[2^{O(n)}]$ and some $c > 0$ such that for all n, L_n requires 2^{cn} size circuits, then $P = BPP$.*

These are amazing results, because they are turning negatives into positives! Even if there are very hard exponential time functions out there, life is not so bad: we can use them for positive algorithmic effect.

In general, there are incredible connections between lower bounds and algorithms that can be formally realized through the study of circuit complexity, and we are still only beginning to understand these connections.

4 Goals, Methods, Mindsets

Right now I have two basic goals for this course.

- The basics of circuit complexity: understand how small and large the complexity is, for interesting functions and interesting types of circuits. We'll look at how these results have been proved, the ideas behind them, and how we might do better.
- Connect circuit complexity to the "usual" uniform complexity theory that you've seen before. We'll use the vehicle of circuit complexity to tour and introduce other areas of complexity, such as exponential-time algorithms.

4.1 Methods

Broadly speaking, there have been three classes of techniques for proving circuit complexity lower bounds:

1. **Restriction methods.** Here is the basic idea.
   - We pick a collection of n-bit functions, e.g. the PARITY function $\{PARITY_n\}$, where computing the n-bit version has a close relationship with computing the k-bit version for $k < n$. In the case of PARITY, there are relations like

$$\text{PARITY}_n(x_1, \ldots, x_n) = \text{PARITY}_{n-k+1}(\text{PARITY}_k(x_1, \ldots, x_k), x_{k+1}, \ldots, x_n).$$

- Argue that, if you set some of the input bits to a small n-bit circuit (i.e., replace some of the input variables with fixed constants), and simplify the resulting circuit, the circuit becomes "too small" to compute the function on k bits. This is generally an inductive method.

Examples of "restriction methods" include gate elimination and the method of random restrictions. They may be the most versatile methods we currently know.

2. **Polynomial methods.** The basic idea is to represent the circuit (approximately or exactly) as a "nice" polynomial, then prove your target function simply can't be represented with this "nice" polynomial.

   The most prominent examples of the polynomial method is that the circuit class AC^0 (constant-depth circuits of polynomial size over the basis of unbounded fan-in AND and OR, with NOTs for free) can be "approximated" by poly$(\log n)$-degree polynomials. This can be used to prove significant lower bounds on AC^0 circuits.

3. **Brute force methods.** Here we try to construct functions that *by design* do not have small-size circuits. We try to do this as efficiently as possible. If could do this in PSPACE or NP, we'd separate P from PSPACE (or NP).

4.2 Mindsets

What are these classes of methods used for? Broadly speaking, we can imagine three basic approaches to proving results in circuit complexity that people have taken over the years:

1. **Bottom-up:** We look at restricted circuits (typically the circuits are restricted by depth, or by gate basis), prove nice lower bounds for these weak circuit models, and try to relax the restrictions gradually. In short, we make the model so weak that finding functions they can't compute isn't so difficult.

2. **Top-down:** We start with functions that we already know to be hard for circuits in some sense, and try to algorithmically produce these functions more efficiently. We make the function so strong that ruling out all circuits to compute it isn't so difficult.

3. **Middle:** We start with fairly easily computable functions, and try to prove unrestricted circuit lower bounds anyway.

Polynomial methods seem inherently based on a bottom-up approach: your circuit has to be somewhat restricted, to have a chance of looking like a polynomial. Brute-force methods are definitely top-down. Restriction methods have been used in both "middle-first" and "bottom-up" approaches.

Recently, we've been studying a combination of "top-down" and "bottom-up": we focus on restricted circuits, and functions that are "hard by definition", but exploit the restrictions on the circuits to get more efficiently computable functions that are still hard. (What I am saying is hopelessly vague, but I hope things will become clearer later on.)

5 Some Popular Circuit Complexity Classes

There are many complexity classes related to circuit complexity. Here we briefly list them, in decreasing order of power:

- P/poly = languages computed by polynomial-size circuits
- NC^k = languages computed in $O(\log^k n)$ depth, polynomial size, bounded fan-in
  A particularly well-studied case is NC^1, which is equivalent in power to polynomial size Boolean formulas.
- TC^{k-1} = languages computed in $O(\log^{k-1} n)$ depth, polynomial size, unbounded fan-in over MAJORITY gates (with NOTs for free). A MAJORITY gate outputs the most popular input; if there is a tie then it outputs 1.
  Here the main focus is the class TC^0.
- $\mathsf{AC}^{k-1}[m]$ = languages computed in constant-depth, polynomial size, unbounded fan-in over the basis AND, OR, MODm, where a MODm gate outputs 1 iff the sum of its input bits is divisible by m. Note that NOT can be simulated with MODm, but it is open whether AND (or OR) can be simulated with only MODms in constant depth.
  Here the main focus is the class $\mathsf{AC}^0[m]$.
- $\mathsf{ACC} = \bigcup_m AC^0[m]$.
- AC^{k-1} = $\mathsf{AC}^{k-1}[m]$ without the MODm gates.

We have the following containments, for all constants m:

$$\mathsf{NC}^0 \subseteq \mathsf{AC}^0 \subseteq \mathsf{AC}^0[m] \subseteq \mathsf{ACC} \subseteq \mathsf{TC}^0 \subseteq \mathsf{NC}^1 \subseteq \mathsf{AC}^1 \subseteq \mathsf{NC}^2 \subseteq \mathsf{AC}^2 \subseteq \cdots \subseteq \mathsf{P}/\text{poly}.$$

5.1 Known Limitations

How many of these containments are proper? We don't know many of them. It is easy to prove that $\mathsf{NC}^0 \neq \mathsf{AC}^0$:

Proposition 3. *If $f : \{0,1\}^n \to \{0,1\}$ is computed by an NC^0 circuit, then f only depends on a constant number of input bits.*

Proof. If the depth is d, and fan-in is b, then the maximum number of variables that f can depend on is b^d, which is constant. □

Corollary 5.1. *The AND function on n variables is not in NC^0.*

Proof. Suppose it were; we will find a circuit in the NC^0 family and an input on which AND is computed incorrectly. Choose $n > b^d$ where b is the fan-in and d is the depth. Let C_n be the NC^0 circuit for AND_n. C_n only depends on b^d variables; setting all of these variables to 1 will make C_n output a value v. If $v = 1$, then set one of the unset variables to 0, so the AND is 0. If $v = 0$, then set all the unset variables to 1, so the AND is 1. □

5.2 AC0 and ACC

So much for NC^0. What about AC^0? In the 80's, there were many upper and lower bounds proved for AC^0. We will cover most of the below results in this course. For now, we will just state them.

Recall the PARITY function: $\text{PARITY}_n(x_1, ..., x_n) = \sum_i x_i \pmod 2$.

Theorem 5.1 (Hastad '85)**.** *PARITY$_n$ has depth-d AC^0 circuit complexity $2^{\Theta(n^{1/(d-1)})}$.*

PARITY is the negation of the MOD2 function, so we have:

Corollary 5.2. $\mathsf{AC}^0 \neq \mathsf{AC}^0[2]$.

Theorem 5.2 (Razborov '87)**.** *The MAJORITY of n bits requires $\mathsf{AC}^0[2]$ circuits of size exponential in n.*

Corollary 5.3. $\mathsf{AC}^0[2] \neq \mathsf{TC}^0$.

Theorem 5.3 (Smolensky '86)**.** *The MOD3 function requires $\mathsf{AC}^0[2]$ circuits of exponential size. In fact, for all distinct primes $p \neq q$, and all constants k, the MODp function is not in $\mathsf{AC}^0[q^k]$.*

Corollary 5.4. $\mathsf{AC0}[3] \neq \mathsf{AC}^0[2]$.

Question: What about MOD4? Where does it lie in this hierarchy?

Answer: Good question!

Exercise: *Is MOD4* $\in \mathsf{AC}^0[2]$*? What about* MOD_{2^k} *for all constants* $k \geq 2$*?*

The above lower bounds hinge on properties of polynomials over finite fields; this is why we see prime powers pop up. What about constant-depth circuits over $\{OR, AND, NOT, MOD3, MOD2\}$? How powerful are multiple moduli?

Claim 1. *Any circuit over the basis* $\{$*OR, NOT, AND, MOD3, MOD*$2\}$ *can be efficiently expressed as a circuit over* $\{$*OR, AND, NOT, MOD6*$\}$.

Proof. Follows from the equations:
$\text{MOD6}(x_1, \ldots, x_n) = \text{MOD3}(x_1, \ldots, x_n) \wedge \text{MOD2}(x_1, \ldots, x_n)$
$\text{MOD2}(x_1, x_2, \ldots, x_n) = \text{MOD6}(x_1, x_1, x_1, x_2, x_2, x_2, \ldots, x_n, x_n, x_n)$
$\text{MOD3}(x_1, \ldots, x_n) = \text{MOD6}(x_1, x_1, x_2, x_2, \ldots, x_n, x_n)$. □

So by looking into constant-depth circuits over MOD3, MOD2, AND, OR, NOT, we're really studying the class $\mathsf{AC}^0[6]$.

HERE, WE'RE REALLY STUCK! MAYBE YOU CAN HELP!

It has been conjectured that MAJORITY $\notin AC^0[6]$. (This would imply that $\mathsf{AC}^0[6] = \mathsf{TC}^0$.) However it is consistent with current knowledge that $\mathsf{AC}^0[6] = \mathsf{P/poly}$, which would be amazing if true. But could we really achieve a massive parallelism of arbitrary serial computations by using just MOD6's? Honestly this looks ridiculous. Furthermore, it's possible that every exponential-time computable function could be efficiently simulated by a $\mathsf{AC}^0[6]$ circuit family, even though this family is only polynomial size! (The fact that we are allowed a separate circuit for each input length is absolutely critical: if there were an efficient algorithm that could generate all circuits in the family, separating exponential time from $\mathsf{AC}^0[6]$ is easy.) Only last year did we rule out that NEXP (nondeterministic exponential time) is contained in $\mathsf{AC}^0[6]$.

Even though we are rather stuck at MOD classes like $\mathsf{AC}^0[6]$ for proving *super-polynomial size* lower bounds, interesting results can still be proved for more general circuit classes.

5.3 TC0 and higher

As we relax the restrictions we place on circuits, our knowledge becomes weaker. Concerning the class TC^0, we do know that depth-3 TC^0 is strictly more powerful than depth-2 TC^0. Let TC^0_d denote depth-d TC^0.

Theorem 5.4 (Hajnal, Maass, Pudlák, Szegedy, and Turán '93)**.** $\mathsf{TC}^0_2 \subsetneq \mathsf{TC}^0_3$.

Also, TC^0 with depth-3 already looks quite powerful. We can simulate all of ACC with three layers of MAJORITY gates (and NOTs for free), provided our size bound is relaxed from polynomial to quasi-polynomial:

Theorem 5.5. *Every problem in* ACC *can be simulated with* TC^0 *circuits of depth 3 with* $n^{\text{poly}(\log n)}$ *size.*

It is known that NC^1 corresponds to the languages recognized by an infinite family of Boolean formulas $\{F_n\}$ of polynomial size. For Boolean formulas, we know some interesting size lower bounds, like quadratic size and cubic size lower bounds. The best known to date is:

Theorem 5.6 (Håstad)**.** *There is a function in* P *that requires* $n^3/\log^2 n$ *size formulas over* U_2.

Open Problem 1. *Is there a language in* NP *that requires formulas of size at least* n^4 *over* U_2*?*

Open Problem 2. *Is there a language in* NP *that requires circuits of size at least* $6n$ *over* U_2*?*

The problems are still open if we replace NP with the much larger class $\mathsf{TIME}[2^n]^{\mathsf{NP}}$.

5.4 Uniform Circuit Classes

The above circuit classes are called *non-uniform*, in that we get a separate algorithm for each input length, and the number of lines of code in that algorithm can grow with the input. This is in contrast to the one-algorithm-for-all *uniform* model that is more common. But there are also *uniform* versions of all these circuit complexity classes, where there exists an fixed, efficient algorithm that will generate the nth circuit on demand for us, when we give the algorithm an n-bit input (say, 1^n).

Let U be a small uniform complexity class (like $\mathsf{LOGSPACE}$) and let $\mathcal{C}$ be a circuit class. Then the class U*-uniform* $\mathcal{C}$ is defined to be the set of languages recognized by a circuit family $\{C_n\}$ from $\mathcal{C}$, and there is an algorithm A implementable in U such that $A(1^n)$ prints C_n as output.

There is a very rich theory of *how much* uniformity is needed to generate circuits: how efficiently can we generate a circuit with an algorithm. We can even have $\mathsf{LOGTIME}$-uniformity, where there is an algorithm A that, given n and an index $i = 1, ..., s$, runs in $O(\log n)$ time and prints the gate information for the ith gate of C_n. That is, $A(n, i)$ prints the gate type of i (whether it is AND, OR, NOT, etc.), and $A(n, i)$ prints gate numbers $j_1, j_2 < i$ such that there are wires $(j_1, i), (j_2, i)$ in the circuit. Such an algorithm can very quickly print "local information" about the overall circuit. I am not planning to talk too much about these uniform circuit classes, but you should definitely know that they are out there.