IAS/PCMI Summer Session 2000
Clay Mathematics Undergraduate Program
Advanced Course on Computational Complexity

Lecture 6: Arithmetic and Threshold Circuits

David Mix Barrington and Alexis Maciel
July 24, 2000

1. Overview

The realm of arithmetic on binary integers provides a wide array of problems whose complexity we can compare. In this second week of advanced lectures we will look at the arithmetic problems that can be solved within a particular model of computation called *threshold circuits*. As with boolean circuits, threshold circuit families have uniformity conditions that limit how complicated they can be. By the end of the week we will see a new algorithm (due to Chiu, Davida, and Litow) for dividing one integer by another using log-space uniform threshold circuits of constant depth and polynomial size (L-uniform TC^0). This algorithm solved the longstanding problem of dividing integers using only $O(\log n)$ space.

Today we will introduce the model of threshold circuits, the complexity class TC^0, and some of the arithmetic problems we will solve within it. Specifically:

- We define majority gates, threshold circuits, and the class TC^0 of languages decidable by threshold circuits of constant depth and polynomial size.
- We define the model of first-order formulas with MAJORITY gates (and the BIT predicate), and the class FOM of languages definable by such formulas. We prove that FOM is equal to DLOGTIME-uniform TC^0, adapting our proof that $\mathrm{FO} + \mathrm{BIT}$ equals DLOGTIME-uniform AC^0.
- We note that TC^0 is contained within L, and then prove that TC^0 is contained within the ordinary circuit class NC^1, using *signed-digit notation* for integers. (An alternate proof of $\mathrm{TC}^0 \subseteq \mathrm{NC}^1$ using *three-for-two addition* will appear in the exercises.)

- We survey the arithmetic problems on integers that will be our subject for the rest of the week. These include some we can already solve in $\mathrm{FO}+\mathrm{BIT}$, some we will solve next lecture in FOM, and some that we can only solve in the larger class TC^1 that allows threshold circuits of logarithmic depth.

2. Threshold Circuits and TC^0

Our unbounded fan-in AND and OR gates can be thought of as counting the number of their inputs that are one (or simply adding up their inputs as integers) and testing the result to see whether it is at least one (in the case of an OR gate) or equal to the number of inputs s (for an AND gate). It is natural to generalize this process to adding up the inputs and comparing the result to some other *threshold*, between zero and s. A *threshold gate* does just this, outputting 1 iff the sum of its inputs is at least some threshold k.

AND and OR gates are special cases of threshold gates, but in general threshold gates are strictly more powerful. Gates with a threshold of $O(1)$ or $s - O(1)$ can be simulated in FO, and it turns out that *polylogarithmic* thresholds ($(\log n)^{O(1)}$ or $s - (\log n)^{O(1)}$) can be simulated in $\mathrm{FO}+\mathrm{BIT}$ by a more complicated argument. But MAJORITY gates, for example, where the threshold is $s/2$, are provably more powerful because the majority function is known to be outside of $\mathrm{FO}+\mathrm{ARB}$. (This will follow from Basic Lecture 10 and a result in the exercises.)

On the other hand, as shown in the exercises, the MAJORITY function is powerful enough to simulate all the other types of threshold functions. We thus have two alternative definitions for the complexity class TC^0: languages decidable by threshold circuits of constant depth and polynomial size, or languages decided by circuits of AND, OR, and MAJORITY gates with constant depth and polynomial size. In each case we allow NOT gates just above the input gates. Our definitions of uniformity restrictions are similar to those for FO, with one exception. If we use gates with arbitrary thresholds, the number k is part of the gate type, and we must be able to answer questions such as "is gate number i a k-threshold gate?" within the given complexity bound.

Why study threshold circuits? Their computational power is interesting simply because they are a natural generalization of AND and OR, and MAJORITY is a natural problem that we'd like to relate to other problems. But another reason is that threshold gates can be implemented in hardware, particularly by analog computers. In fact, the earliest known analog computer, the animal brain, seems to use something like threshold computation. A neuron fires when the sum of voltages coming into it surpasses some threshold, and when it fires it delivers particular voltages to certain other neurons. A considerable body of research has looked into the power of *weighted* threshold gates, where instead of comparing the sum of the boolean inputs to the threshold, a gate compares a particular linear combination or weighted sum to the threshold. (It is known, though, that constant-depth poly-size circuits of weighted threshold gates still compute only TC^0.)

3. First-Order Formulas With Majority

In the logical model, we can make the majority function definable only by somehow changing the rules to get us outside of $\mathrm{FO} + \mathrm{ARB}$. We have seen a relationship between the gates of an unbounded fan-in circuit and the quantifiers in a formula, suggesting that we design a quantifier that somehow simulates a MAJORITY or threshold gate. If $\Phi(x)$ is a formula with one free variable, we define $Mx : \Phi(x)$ to be true iff $\Phi(x)$ is true for a majority of the possible values of x. We define FOM to be the languages definable by formulas with $\exists$, $\forall$, and M quantifiers, using atomic predicates for input, $=$, $<$, and BIT. (Why include BIT? It turns out that FOM without BIT is a weaker class that cannot perform the most basic machine operations like operating a counter, and is thus less relevant to machine computation. The proof of the lower bound on its power is quite interesting, but outside of our scope here.)

In FOM we can define some other useful predicates that will be convenient later (note that the BIT predicate is important here). Throughout Φ is an arbitrary FOM formula with the appropriate free variables:

- "$\Phi(x)$ holds for exactly half of the possible x." This sentence, written $Hx : \Phi(x)$, can be expressed as $(Mx : \Phi(x)) \wedge (Mx : \neg\Phi(x))$.
- "There are exactly y values of x less than $n/2$ such that $\Phi(x)$." This can be expressed as

  $$Hx : (\Phi(x) \wedge (x < n/2)) \vee ((x \geq n/2) \wedge (x < n - y))$$

  .
- "There are exactly y values of x such that $x \geq n/2$ and $\Phi(x)$." This can be expressed in the same way as the one above.
- "There are exactly y values of x such that $\Phi(x)$." This is easy using addition of index numbers (in $\mathrm{FO} + \mathrm{BIT}$): we guess the number in the first half and in the second half, and add them.

To prove that FOM and DLOGTIME-uniform TC^0 are the same, we follow our proof that $FO + BIT$ and DLOGTIME-uniform AC^0 are the same. That proof was really about the relationship between DLOGTIME-uniform circuits, with any gate type, and formulas with BIT and the matching type of quantifiers. Where do the MAJORITY gates and majority quantifiers enter into the proof?

As we simulate the formula by a circuit, a majority quantifier gives rise to a level of gates in the tree, and these will be MAJORITY gates. Each gate is reached by a particular path from the output gate, and the choices made along this path correspond to a choice of values for some of the variables. At a given gate, the values already chosen are those that are bound by quantifiers before the quantifier for that level. Each child of that gate represents a choice of a value for a new variable, the one bound by the majority quantifier for that level.

Each MAJORITY gate will output 1 iff a majority of its children output 1. This will happen iff a majority of values of the variable bound by the majority quantifier give rise to marked words that satisfy the rest of the formula, and *this* is precisely the definition for the given marked word to satisfy the subformula beginning with the majority quantifier. The tree part of the circuit, and in fact the whole circuit, are DLOGTIME uniform just as before — the presence of majority quantifiers has not changed the relationships among the gate numbers.

In simulating the circuit by a formula, we need add only one new type of predicate to our inductive construction of definable predicates: "Gate $\mathbf{i}$ is a MAJORITY gate with value 1 at depth d". (Remember that $\mathbf{i}$ is a vector of $O(1)$ variables.) This is true iff for a majority of numbers $\mathbf{j}$ such that $\mathbf{j}$ is the number of a child of gate $\mathbf{i}$, $\mathbf{j}$ is a gate with value 1. This is *almost* a formula with a majority quantifier in front of a previously defined predicate, but not quite. We need the following technical lemma:

Lemma 1 *Let* $\Phi(\mathbf{x})$ *and* $\Psi(\mathbf{x})$ *be* FOM *formulas with a common vector of free variables* $\mathbf{x}$*. Then the predicate "for a majority of the* $\mathbf{x}$ *such that* $\Phi(\mathbf{x})$, $\Psi(\mathbf{x})$ *is true" is definable in FOM.*

Proof First consider the special case where Φ is always true — we need to show that we can take the majority of the n^k values of $\Psi(\mathbf{x})$ using just the majority quantifiers that bind one variable at a time. (Note that $Mx : My : \Psi(x, y)$ does *not* tell whether $\Psi(x, y)$ holds for a majority of the pairs $\langle x, y\rangle$.) For the moment we will *cheat* on this, and assume that we have a k-ary majority quantifier that checks the majority of k-tuples. (We'll derive majority-of-k-tuples from majority in the exercises.) Once we have that, we can express "there are exactly m values of $\mathbf{x}$ such that $\Psi(\mathbf{x})$ by counting the first and second halves and adding the answers, as above. (Here m is a number of $O(\log n)$ bits stored as a vector of variables.)

To tell whether Ψ is true for a majority of the $\mathbf{x}$ such that $\Phi(\mathbf{x})$ is true, we need only count how many $\mathbf{x}$ satisfy $\Phi(\mathbf{x}) \wedge \Psi(\mathbf{x})$, count how many satisfy $\Phi(\mathbf{x}) \wedge \neg\Psi(\mathbf{x})$, and compare the answers. $\square$

4. Upper Bounds on TC^0

Let's first demonstrate that we can evaluate a constant-depth, poly-size threshold circuit with a log-space machine. We use a simple recursive attack: for each gate we recursively evaluate each of its children in turn, add up the number of children outputting true and the number outputting false, and return true iff the number returning true is appropriate (for example, if it is half or more in the case of a majority gate). We can cycle through all the children of a gate by cycling through *all* nodes and checking which ones are children with the child predicate. (We thus need the child predicate to be log-space uniform, at least. We also need the gate-type predicate to be log-space uniform so we can evaluate leaves.) The storage needed for this process is that needed for the name of each gate on the stack ($O(1)$ gate numbers of $O(\log n)$ bits each) and a counter for each gate on the stack ($O(1)$ counters of $O(\log n)$ bits each), a total of $O(\log n)$ bits.

A more difficult task is to simulate TC^0 in NC^1, which requires simulating a MAJORITY gate in NC^1. We'll do this in two ways, one here and one in the exercises. In both cases we will solve the more general problem of *iterated addition* of n numbers of $O(\log n)$ bits each. (Majority is the special case of iterated addition where all the inputs are one or zero.) It will not do to have a binary tree of addition circuits, because the addition circuit we know (and indeed, any circuit that solves the standard addition problem) involves more than $O(1)$ depth when we restrict the fan-in of gates to two. Our two tricks will each solve a *variant* of the addition problem in depth $O(1)$ and fan-in two, allowing us to solve a variant of iterated addition in NC^1. We will then have to show how to use the variant to solve the standard iterated addition problem.

The first trick is to represent integers in *signed-digit notation*. We will use a base of four (actually any constant base greater than two will do). Along with the ordinary base-four digits $\{0, 1, 2, 3\}$ we will have three additional digits $\{\overline{1}, \overline{2}, \overline{3}\}$ representing -1, -2, and -3 respectively. A string $d_k \ldots d_1 d_0$ represents the integer $\sum_{i=0}^{k} d_i 4^i$, just as for ordinary base-four. Note that there are several different strings to represent a given number — for example, "seven" could be 13, $2\overline{1}$, or $1\overline{3}3$, among other possibilities.

A binary number may be put *into* signed-digit notation by simply grouping its bits into pairs and negating all the digits if the number is negative. Getting a number from signed-digit to ordinary notation is in FO + BIT, as it involves one binary subtraction (of the number formed by the negative digits from the number formed by the positive digits). All we need to do is to show how to take two numbers in signed-digit notation and compute their sum in signed-digit notation, using $O(1)$ depth and fan-in two.

We begin by adding each pair of corresponding digits to get a number ranging from -6 to 6. We record this number as a two-signed-digit number, where the fours digit (a $\overline{1}$, 0, or 1) will be carried. But we need to take care that carries cannot propagate as in ordinary binary addition. To do this, we make sure that the uncarried units digit is never a $\overline{3}$ or a 3: we record -3 as $\overline{1}1$, -1 as $0\overline{1}$, 1 as 01, and 3 as $1\overline{1}$. Then the addition of the carries to the answer string causes no further carries and can be done in $O(1)$ depth and fan-in two.

In the exercises we take the alternate approach of *three-for-two addition* and also prove that $\mathrm{TC}^0 \subseteq \mathrm{NC}^1$.

5. The Spectrum of Arithmetic Problems

Let's now look at the variety of arithmetic problems we want to solve. First, some terminology. We'll be concerned mostly with integers in two ranges: numbers of $O(\log n)$ bits, which we'll call *short*, and those of $n^{O(1)}$ bits, which we'll call *long*. (The language is deliberately reminiscent of the various integer data types in programming languages.) Short numbers can be represented by variables, or vectors of $O(1)$ variables, in the logical formalism. Long numbers cannot be represented directly, but they can be represented implicitly as predicates of $O(1)$ variables. (The input, thought of as a number of n bits, is already given this way by the input predicate.)

We've seen that in FO + BIT we can add two long numbers and perform a variety of operations on short numbers. We showed that we can add up $O(\log n)$ short numbers, which is sufficient to multiply short numbers or in fact to add $O(\log n)$ long numbers. We also have the two integer division operators (Pascal `div` and `mod`, C or Java `/` and `%`) on short numbers, because we can guess and verify both the quotient and remainder. For that matter we can use the sequence capability of FO + BIT to perform exponentiation on short numbers, as long as the answer is short.

Extending the logical formalism from FO + BIT to FOM increases our ability to do arithmetic, at least by letting us add up n one-bit numbers (the definition of the threshold operation). What more can we do? The two obvious targets of our ingenuity are *multiplication* of long numbers and *iterated addition* of polynomially many long numbers. The latter will suffice to carry out the former, by the standard algorithm for multiplication. We'll show how to do these operations in FOM in the next lecture.

Here we'll solve a special case of the iterated addition problem, that of adding a polynomial number of short numbers to get a short result. As we'll see in the exercises, this will suffice to implement the majority-of-pairs quantifier (and generalizing, the k-ary majority quantifier) with the simple n-ary majority quantifier. To be precise, we will add together n short numbers using simple majority quantifiers and note that a polynomial number of short numbers may be added by repeating this construction.

The argument is similar to that for the BITSUM predicate in Advanced Lecture 4. For each of the $O(\log n)$ input positions in the inputs, we have a column of n bits which can be added with a single threshold operation using simple majority quantifiers. The results of these operations, properly shifted, constitute $O(\log n)$ short numbers whose sum is our desired sum. Now as before we can let t be the square root of $\log n$, reformat these numbers as having $O(t)$ segments of t bits each, divide each number into the pieces consisting of its odd-numbered and even-numbered segments, add together the odd pieces and the even pieces, and finally add the two results. (The $O(\log n)$ odd-numbered pieces cannot cause a carry to propagate because each column of segments has a sum of at most $t + (\log \log n) < 2t$ bits.)

The next set of problems to concern us are:

- *Division* of one long number by another,
- *Powering* of a long number by a short exponent (giving a long result), and
- *Iterated multiplication* of a polynomial number of long numbers, also giving a long result.

We could carry out iterated multiplication, and its special case powering, with circuits by constructing a binary tree of multiplication operations. This would be a threshold circuit family of unbounded fan-in and depth $O(\log n)$, putting these problems in TC^1, a subclass of NC^2. As natural as this algorithm seems, we will see better ones in the remainder of this week that will put these three problems in L-uniform TC^0.

It is not trivial that these three problems should have the same complexity. We can see that powering is a special case of iterated multiplication, and it is possible to use division to calculate powering as well because $1/(1-A2^n)$, for example, is the sum for all i of $A^i 2^{ni}$, a number from which we can read off all powers of A of at most n bits. With the aid of some FOM problems, we can use almost this same relationship to solve the division problem given powering. To divide X by Y, for example, we calculate and add together powers of $(1-Y)$ to get a close approximation of $1/Y$, then multiply this by X. We may have two candidates for $\lfloor X/Y \rfloor$ because of the potential error of the approximation, but we can check each of them using multplication.

A final class of arithmetic operations included iterated multiplication and powering of integer *matrices*. Multiplying two matrices is in FOM, because each entry of the result is the iterated sum of binary products of entries of the input matrices, and both iterated sum and binary product are in FOM. To multiply together polynomially many matrices, however, seems to require a binary tree of these FOM operations, and thus a TC^1 circuit. It turns out that an appropriately chosen iterated matrix product is equivalent to finding the *determinant* of an integer matrix, so the class of problems that can be solved by reduction to iterated matrix product is sometimes called DET.

6. Exercises

1. Show that the parity function can be computed by a depth-2 threshold circuit.

2. A *symmetric* function is one whose value depends only on the number of inputs that are one (that is, no permutation of the input values can change the output). Show that any symmetric function is in non-uniform TC^0.

3. Prove that the class TC^0 remains invariant whether we define it in terms of arbitrary threshold gates, or only MAJORITY gates. This is true for non-uniform, P-uniform, L-uniform, and FO + BIT-uniform circuits. Use the equivalence to FOM to argue that it is true for the DLOGTIME-uniform versions as well. (Hint: the FO + BIT-uniform and DLOGTIME-uniform versions are equal.)

4. Show that if x, y, and z are three binary numbers (of up to n bits), we can in $O(1)$ depth and fan-in two compute two new numbers u and v such that $x + y + z = u + v$. Conclude that we can reduce n n-bit numbers to two numbers with the same sum using depth $O(\log n)$ and fan-in two. Explain how this leads to an alternate proof that $\mathrm{TC}^0 \subseteq \mathrm{NC}^1$.

5. Using the fact that iterated addition of n short numbers is in FOM (using only majority quantifiers binding one variable), show that for any k the k-variable majority quantifier can be simulated in this version of FOM. (The k-variable majority quantifier $M_k x_1, \ldots, x_k : \Phi(x_1, \ldots, x_k)$ determines whether Φ is true for a majority of the n^k possible k-tuples of variable values.)

6. Let M be a nondeterministic log-space machine. For any input string w, let $f(w)$ be the *number of computation paths* on which M accepts w. Show that the function f can be computed in the class DET. (The class of all such functions f from Σ^* to non-negative integers is called #L.)