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

Lecture 5: Boolean Formulas, NC^1, and M-Programs

David Mix Barrington and Alexis Maciel
July 21, 2000

1. Overview

We've now shown the circuit class AC^0 to be equivalent to first-order formulas at various levels of uniformity, making it a "non-uniform analogue" of the first-order or star-free languages. With M-programs over arbitrary finite monoids, we have a similar non-uniform analogue of the regular or recognizable languages. In this lecture we characterize the computational power of M-programs, bringing in the models of *log-depth circuits* and *boolean formulas*. Specifically:

- We define the model of boolean formulas, show their relationship to circuits, and (in an exercise) show how arbitrary poly-size boolean formulas are equivalent to balanced, log-depth boolean formulas.
- We argue that poly-length M-programs (over any finite monoid) can be simulated by log-depth boolean formulas, putting languages decided by such formulas in (non-uniform) NC^1.
- We develop some finite group theory, showing how to represent finite groups as permutations and defining commutator subgroups and solvable groups.
- We show that any log-depth boolean formula can be simulated by a poly-length M-program over the group S_5, proving that the languages decided by poly-length M-programs are *equal* to non-uniform NC^1. In the exercises we discuss uniform versions of this result.
- We look briefly at what algebra might tell us about the internal structure of NC^1.

2. Boolean Formulas

A *boolean formula* is combination of boolean input variables by the binary operations AND and OR and the unary operation NOT. (Without loss of generality, we assume that all NOT operations have been pushed to the bottom, so they may only be applied to input gates.) If we represent the formula as a circuit, it has fan-in two and *fan-out* one, meaning that as a graph it is a tree rather than a general directed acyclic graph. But we can also represent the formula as a string, in the familiar *infix*, *postfix*, or *prefix* notation. Just like circuits, boolean formulas represent a function from Σ^n to $\{0,1\}$, and a *family* of boolean formulas represents a function from Σ^* to $\{0,1\}$ and thus a formal language. We can measure the size and depth functions of a formula family just as for a circuit family. There are several different predicates we could examine to define the uniformity of a formula family, and the choice is sometimes significant depending on the restrictiveness of the uniformity condition.

In the basic lectures we have defined the complexity class NC^1, of languages decided by circuit families of fan-in two, $O(\log n)$ depth, and polynomial size. The circuits in such a family need not represent formulas, but given any NC^1 circuit family we can construct an equivalent formula family of polynomial size and the same depth. We do this by eliminating any situations where the fan-out in the circuit is greater than one, making a separate copy of the gate (and everything under it) for each wire that wants to access that gate. (Inductively, for every possible sequence of left and right branchings through the circuit, we have a gate in the formula that is a copy of the circuit gate reached by the path.) This process clearly preserves depth but may increase the size. However, it is clear that a formula of fan-in two and depth d can have at most 2^d leaves and at most $2^{d+1} - 1$ total gates. So logarithmic depth and fan-in two *imply* polynomial size, and the "polynomial size" condition in the definition of NC^1 is redundant.

Is this process uniform, that is, does it produce a uniform formula from a uniform circuit? We can choose a layout of the formula as a string, so that the index of a bit records the sequence of left and right choices leading to a particular operator. But to determine which gate goes in which place by reference only to the circuit, we need to be able to take a sequence of left and right choices and produce a gate number. This predicate is sometimes called the *extended connection language* of the circuit. We can see that if the extended connection language is in DLOGTIME, for example, we can arrange the formula so that it is also DLOGTIME uniform. But if we are given only the child and gate-type predicates for the circuit, more computation will apparently be required to answer uniformity questions about the formula.

One might wonder about the class of languages decided by poly-size boolean

formulas of *any* depth, not just the relatively balanced formulas of $O(\log n)$ depth. At least in the non-uniform case, as we will see in the exercises, it is possible to take an arbitrary boolean formula and *balance* it — construct an equivalent formula of polynomially greater size but depth logarithmic in its size. It is not clear even from this construction that an NC^1 circuit family can be designed that will solve the *boolean formula value problem*, which is to input a formula and an input string and to evaluate the formula. (The equivalent *boolean sentence value problem* is to evaluate a formula that has constant gates in place of its inputs.) In fact this problem is known to be in uniform NC^1 by a theorem of Sam Buss that works through the intermediate model of *log-time alternating machines*. We will define alternating machines in Basic Lecture 11, but we won't be able to prove Buss' theorem here.

3. Simulating M-Programs With Formulas

Let M be an arbitrary finite monoid, and let L be defined by a family of M-programs of polynomial size. That is, for each n there is a program of length $p(n)$ that computes a mapping ϕ from Σ^n to M, such that $L = \phi^{-1}(X)$ for some set $X \subseteq M$. We will show in this section that L has poly-size log-depth boolean formulas and is thus in NC^1.

In Advanced Lecture 4 we showed that every regular language is in NC^1, and the present proof is only a slight generalization of this earlier argument. For a given input size n, the instructions of the program define a sequence of monoid elements that multiply to $\phi(w)$ for any given input word w. We can make a sequence of $O(1)$ size circuits each of which accesses a letter of w and produces the yield of one instruction. This sequence of circuits is essentially as uniform as the program is, since each circuit implements a lookup function given by the instruction.

The largest computational task in evaluating the program is to multiply $p(n)$ monoid elements to get a single element $\phi(w)$. Once we have this, we need only $O(1)$ size to determine whether $\phi(w)$ is in X and thus whether w is in L. The iterated product operation can be carried out by a binary tree of binary product operations, and each binary product operation takes $O(1)$ size (as does any function with $O(1)$ inputs and $O(1)$ outputs). The resulting circuit can be converted to a formula as in the previous section, and the formula family is quite uniform (unless the program family itself was not uniform).

4. Permutation Groups and Solvability

A finite *group* is a monoid where each element x has an *inverse* y such that $xy = yx = e$. A finite group can be represented as a set of *permutations* (one-to-one, onto functions) of a finite set under the operation of composition — one way to do this is to represent each group element x as the permutation that takes an element y to the product yx. We define S_n to be the group of *all* permutations of an n-element set, and note that we have shown that every finite group is isomorphic to a subgroup of S_n for some n.

We can represent an element x of S_n by a sequence of n distinct numbers in the range from 1 through n, in two separate ways. The obvious way is the *pointwise notation*, giving the sequence $x(1), x(2), \ldots, x(n)$. Another way is *cycle notation*, using the fact that any permutation divides the point set into *orbits* or *cycles*. For example, the element of S_5 with pointwise representation $5, 3, 2, 1, 4$ has two orbits: 1 goes to 5 which goes to 4 which goes to 1, and 2 and 3 are interchanged. We can write the permutation as (154)(23) to record this information. Of course there are many ways to write the same permutation, such as (32)(541) or (415)(32), but we can pick a canonical one if we want. The standard choice is to start each cycle with its smallest-numbered element, then order the cycles in descending order of first element. (Our example thus becomes (23)(154).) This allows us to omit the parentheses and still convert from pointwise to cycle notation and vice versa (see the exercises).

We need to do a bit of group theory in this section, developing the concepts of *commutator subgroups* and *solvability*. Let G be any finite group. The *commutator* of two elements x and y of G is the element $xyx^{-1}y^{-1}$, where x^{-1} is the inverse of x guaranteed by the group axioms. The *commutator subgroup* of G is the subgroup *generated by* the set of all commutator elements for all x and y. (Note that the commutator subgroup can contain elements that are not commutators themselves, as there is no guarantee that the set of commutators is closed under multiplication.) In an *abelian group*, where every two elements x and y satisfy $xy = yx$, every commutator is equal to the identity and so the commutator subgroup is just $\{e\}$. An example of a non-abelian finite group is S_3, the permutations of three elements. The reader may verify that the 36 commutators are each equal to one of e, (123), or (132). (For example, (12)(123)(21)(321) is equal to (123). The cycle notation is easy to compute with given a bit of practice. You get the inverse of a permutation by reversing each cycle. A product where the cycles are not disjoint may be computed by tracing the effect of successive cycles on each element.) Since these three elements form a subgroup, the commutator subgroup of S_3 is the group containing these three elements, called A_3 and isomorphic to the integers modulo 3.

Beginning with any finite group G, we can define a series of groups $G_0 = G$, G_1, G_2,..., by letting G_{i+1} be the commutator subgroup of G_i. With S_3, as we have seen, $G_0 = S_3$, $G_1 = A_3$, and $G_2 = \{e\}$. Since the groups are finite, one of two things must happen to this sequence: it will reach $\{e\}$ or it will reach *some other group* whose commutator subgroup is itself. Groups in which the former happens are called *solvable*, the others are called *non-solvable*. (The reason for the terminology comes from Galois theory, the setting in which groups were originally defined. An equation in which the possible permutations of the roots forms a solvable group can be solved by radicals, other equations cannot. The fact that S_5 is not a solvable group is the basis of Abel's proof that arbitrary fifth-degree equations cannot be solved by radicals.)

There is an intuitive sense in which solvable groups are "nearly abelian", while non-solvable groups are non-abelian in a very strong sense. In the next section we will show a computational consequence of non-solvability.

5. Simulating Formulas With M-Programs

We are now ready to show that M-programs are surprisingly powerful. They can simulate *all* of NC^1, which (as we will see in next week's lectures) includes the majority language, integer multiplication, and much more. We will give the original argument that constructs a program over the group S_5, leaving it to the exercises to show that a similar construction works for arbitrary non-solvable groups.

Theorem 1 *(Barrington's Theorem:) Polynomial-length programs over S_5 recognize exactly those languages in NC^1.*

Proof We will argue that given any boolean formula F of depth d, there exists a program B of length 4^d over S_5 that simulates F in a particular way. On an input word w, B will define a permutation $B(w)$ in S_5, and we will ensure that $B(w) = (12345)$ if $F(w) = 1$ and $B(w) = e$ otherwise. We say in this case that the language of F "is *five-cycle recognized* by B". (A *five-cycle* is a permutation that has exactly one orbit of size five.) Before we get to the key step in the proof, we need a series of lemmas.

Lemma 2 *If B is a program of length at least one, and σ and τ are fixed permutations, there exists a program C, of the same length as B, such that for any w we have $C(w) = \sigma B(w)\tau$.*

Proof Alter the first instruction of B by multiplying each possible yield on the left by σ, and alter the last instruction of B by multiplying each possible yield on the right by τ. The product of the yields of the instructions of C will then be σ times the product of the yields of B times τ. □

Lemma 3 *The language of any input gate is five-cycle recognized by a program of length 1.*

Proof The instruction can query the variable of the gate, output (12345) if the variable has the desired value, and output e if it doesn't. □

Lemma 4 *A language L is five-cycle recognized by a program of length t iff for any five-cycle σ, there is a program of length t that outputs σ if $w \in L$ and outputs e if $w \notin L$.*

Proof If σ is a five-cycle, then there exists a permutation θ such that $\sigma = \theta(12345)\theta^{-1}$. Using an earlier lemma, then, we can convert a program that outputs (12345) iff $w \in L$ to one that outputs σ iff $w \in L$ and vice versa. Since $\theta e \theta^{-1} = e$ for any θ, these programs all output e if $w \notin L$. □

Lemma 5 *If L is five-cycle recognized by a program of length t, then its complement is also recognized by a program of length t.*

Proof Using the earlier lemma, multiply the first program on the right by (54321) so that it outputs e if $w \in L$ and (54321) otherwise. By the last lemma, since (54321) is a five-cycle, the complement of L is five-cycle recognized by a program of length t. □

We are now ready for the main proof. We show by induction that a formula of depth d has a language five-cycle recognized by a program of length at most 4^d. The base case of $d = 0$, where the formula is an input gate, is handled above. We have also handled the case of a NOT gate, and provided a way to convert OR gates to AND gates, with the complementation lemma above. The argument we need to finish is the following, the implementation of an AND gate:

Lemma 6 *If L_1 and L_2 are languages five-cycle recognized by programs A_1 and A_2 respectively, each of length t, then $L_1 \cap L_2$ is five-cycle recognized by a program of length $4t$.*

Proof Using the previous lemmas, we design programs as follows:

- $B_1(w) = (12345)$ iff $w \in L_1$, $B_1(w) = e$ otherwise,
- $B_2(w) = (13542)$ iff $w \in L_2$, $B_2(w) = e$ otherwise,
- $C_1(w) = (54321)$ iff $w \in L_1$, $C_1(w) = e$ otherwise,
- $C_2(w) = (24531)$ iff $w \in L_2$, $C_2(w) = e$ otherwise.

Let the program B be the concatenation $B_1B_2C_1C_2$. There are four cases to consider:

- $w \notin L_1$, $w \notin L_2$, $B(w) = eeee = e$.
- $w \notin L_1$, $w \in L_2$, $B(w) = e(13542)e(24531) = e$.
- $w \in L_1$, $w \notin L_2$, $B(w) = (12345)e(54321)e = e$.
- $w \in L_1$, $w \in L_2$, $B(w) = (12345)(13542)(54321)(24531) = (13254)$.

The language $L_1 \cap L_2$ is thus five-cycle recognized by a program of length $4t$. This closes the induction and proves Barrington's Theorem. □

□

The permutation fact used at the end of the proof may seem somewhat *ad hoc*, but let's look at it more closely. (The exercises ask you to in effect show that a similar trick can be used in any non-solvable group.) By concatenating the four programs, we recognize $L_1 \cap L_2$ with output equal to the *commutator* of the two outputs we had available. This commutator has to be in the commutator subgroup of G. If this subgroup were not all of G, we would be restricted at the next level in what the output of our programs could be. The construction works only because we can take two sufficiently *generic* elements of the group, take their commutator, and get a sufficiently generic element in return. The determining factor in whether we can repeatedly take commutators as long as we want, before being forced to the identity, is exactly the non-solvability of the group.

6. The Internal Structure of NC^1

Earlier we mentioned the *structure theory of finite monoids* due to Krohn and Rhodes, which explains how all monoids may be built up from building blocks that are either aperiodics or *simple groups*. A simple group is one that has no non-trivial normal subgroups (in particular, its commutator subgroup is either itself or $\{e\}$). The most familiar examples of simple groups are the integers modulo a prime (*abelian simple groups*) and the groups A_n, the commutator subgroup of S_n, for all $n \geq 5$. (The full set of finite simple groups was characterized in the 1970's, in one of the great triumphs of modern mathematics.)

If a monoid contains any non-abelian simple group, then poly-size programs over it recognize exactly the languages in NC^1. Is the converse of this statement true? The composition operation in the Krohn-Rhodes theorem corresponds fairly closely to stacking of levels of gates in a circuit. If a monoid contains only solvable group and aperiodic pieces, it is called a *solvable monoid*. Solvable monoids turn out to be closely related to another circuit complexity class, which was proposed (in Barrington's Ph.D. thesis, actually) as an intermediate step between AC^0 and larger classes:

Definition: A MOD-m gate for a number $m > 1$ takes any number of boolean inputs and outputs 1 iff their sum (as integers) is *not* divisible by m. The complexity class ACC^0 consists of the languages recognized by circuit families where the circuits have constant depth, unbounded fan-in, polynomial size, and have AND, OR, and MOD-m gates where m is fixed for the circuit family.

Theorem 7 *(Barrington-Thérien, not proved here) A language has poly-size programs over a solvable monoid iff it is in* ACC^0.

Conjecture 8 *(The "ACC Conjecture")* $\mathrm{ACC}^0 \neq NC^1$.

In Basic Lecture 10 we will prove an important special case of the ACC Conjecture due to the late Roman Smolensky — that if all the MOD gates in an ACC^0 circuit have the same *prime* modulus p (actually we'll only prove it for $p = 3$), then the circuit cannot simulate a MOD-q gate where q is any number except for a power of p. This result followed fairly quickly upon the Furst-Saxe-Sipser theorem that AC^0 circuits cannot simulate a MOD-2 gate, and led to considerable optimism about further progress toward resolving the ACC Conjecture and proving lower bounds for larger and larger class. But now, fifteen years after Smolensky's proof, the ACC Conjecture remains unresolved without much apparent progress. It is still consistent with our current knowledge that ACC^0 might contain all problems in P, or even all

problems in NP! No one really believes that MOD-6 gates, for example, contain any strange computational power, but proving that they *don't* has proved very difficult.

7. Exercises

1. Show that any formula of size s is equivalent to another formula of size $s^{O(1)}$ and depth $O(\log s)$. (Hint: A useful lemma is that any binary tree has an edge splitting it into two pieces each containing at least a third of the nodes. Using this, divide an arbitrary tree into two pieces R and S such that R contains the output gate and S fits into a leaf of R. Let R_0 and R_1 be the trees with a 0 and a 1 respectively substituted for S's leaf. Note that the original formula is equivalent to $(R_0 \wedge \neg S) \vee (R_1 \wedge S)$. Recursively apply this transformation to R and S and use a recurrence to measure the size and depth of the resulting formula.)

2. Show that the pointwise form of a permutation can be determined from the given canonical cycle form, with no parentheses, in FO + BIT. Show that this canonical cycle form can be determined from the pointwise form in L. (Explain why any ordering of the n numbers is the canonical cycle form of exactly one permutation.)

3. Show that S_4 is a solvable group by finding the series of subgroups given in the definition of solvability.

4. Show that any formal language can be decided by a family of programs over the group S_3, in general with exponential program length. Show that this is *not* true for the group S_2.

5. Argue carefully that a language is in DLOGTIME-uniform NC^1 iff it has DLOGTIME-uniform poly-size programs over S_5. For NC^1, define uniformity by the extended connection language: given any sequence of $O(\log n)$ L's and R's and a gate number, in DLOGTIME we can determine whether that sequence of branchings from the output leads to that gate. You may make other convenience assumptions about the circuits if you want.

6. Show that if G is any non-solvable group, any boolean formula of depth d may be simulated by a program over G of length c^d for some number c depending on G.

7. Let G be a solvable group. Show that any poly-length program family over G can be evaluated by a circuit family where the circuits have $O(1)$ depth,

polynomial size, and *only MOD gates* of unbounded fan-in. (This puts these language into a subset of ACC^0 called "CC^0". It is not known whether the AND function on n inputs is in CC^0 — if it is then $\mathrm{CC}^0 = \mathrm{ACC}^0$.)