NC^1 and Barrington's Theorem

COMPSCI 501 Guest Lecture
David Mix Barrington
17 April 2019

NC^1 and Barrington's Theorem

- NC^1 in Context
- Branching Programs
- Regular Languages and Monoid Multiplication
- Simulating Circuits With S_5 Programs
- Extensions: Algebra and Complexity

NC^1 in Context

- **NC^1** is the set of decision problems solvable by boolean circuits (AND, OR, and NOT gates) of fan-in two, polynomial size, and depth O(log n).
- It sits inside L (deterministic log space) and thus inside NL, P, and NP. A log space DTM can evaluate an NC^1 circuit by depth-first search, using a stack to remember its location, as long as the circuit is sufficiently **uniform**.

Circuit Uniformity

- For most circuit classes, it suffices to define a "uniform" circuit to be one that can be constructed by a logspace Turing machine.
- But when we are interested in complexity classes inside L, this won't always do.
- "First-order uniformity" means that structural questions about the circuit can be defined by first-order formulas — this is the standard notion in CS 601.

Basic Results About NC^1

- NC^1 **circuits** and **formulas** (circuits that are trees) have the same computational power. Evaluating a formula is a complete problem for NC^1.
- NC^1 strictly includes AC^0, the class of problems solvable by circuits of unbounded fan-in, polynomial size, and depth O(1). The "strict" is because the parity language can be shown *not* to be in AC^0.

Basic Results About NC^1

- Lots of arithmetic operations on binary integers can be done in NC^1. But even the majority function, counting the number of 1's in the input string, is not obvious. Adding two binary numbers is AC^0, so adding n of them together in the obvious way would be AC^1.
- But there are tricks — using a different notation for binary numbers, two can be added in NC^0, so n can be added in NC^1, and converting back to normal notation is AC^0.

Basic Results About NC^1

- Sipser and others showed in the early 1980's that parity (and therefore majority) is not in AC^0. This was thought to be the first step in a series of lower bounds against larger and larger circuit classes, culminating in a lower bound against poly-size general circuits that would prove $P \neq NP$.
- In the second step, Razborov and Smolensky showed that even with parity gates (or mod p gates for a single prime p), AC^0 still cannot do majority.

Classes Within NC^1

- If we augment AC^0 with mod-m gates for some fixed m (possibly composite), but keep constant depth and polynomial size, we get the class ACC^0. There seems to be no reason that the majority function should be in this class.
- If we use the mod-m gates alone, with constant depth and polynomial size, we get the class CC^0. There seems to be no reason that the AND function should be in this class.

Branching Programs

- Also in the mid-80's, there was an interest in another combinatorial model of computation, that of **branching programs**, and its relation to circuit classes.
- A branching program is just a flowchart, where at each node the program queries one of the input bits, and goes to either the 0-successor or the 1-successor of that node depending on the result. A **decision tree** is the special case where the fan-in is 1.

Branching Programs

- Poly-size branching programs are equivalent to deterministic log space TM's, as long as the programs are uniform.
- An LSTM can trace the correct path through the program, using log space to remember where it is.
- Since a configuration of an LSTM is given by the state, tape contents, and head positions, there are only polynomially many, and we can make a BP with a node for each one.

Branching Programs

- Could we get a lower bound against a restricted class of branching programs, like the lower bounds against AC^0 and AC^0 with mod p gates?
- Borodin *et al.* proved a lower bound against programs of **width** 2. Here we'll define width as follows. The nodes of a width-w BP are divided into levels of size w, and the two successors of a node on level i are on level i+1. Also, all nodes on a level query the same input variable.

Branching Programs

- So for my Ph.D. research I took on the problem of extending the lower bound, say to all constant widths.
- This class BWBP might be interesting because it strictly includes AC^0 (and even ACC^0, which allows modular gates of any fixed modulus), but still looks weak.
- It also includes all regular languages, but we have lower bound techniques against that class. Could we show MAJORITY $\notin$ BWBP?

Regular Languages and Monoids

- Think back to the argument that the parity language is in NC^1. You make a binary tree of XOR gates, each of which has constant size and depth.
- You can think of this as "multiplying" together n elements in the group $\mathbb{Z}_2$, using a binary tree of binary $\mathbb{Z}_2$ multiplications.
- Actually the decision problem for any regular language can be thought of similarly.

Regular Languages and Monoids

- If X is any finite set with n elements, the bijections on X form a group with n! elements, called S_n, under the operation of composition. The functions from X to X form a **monoid** with n^n elements, called T_n, under the same operation.
- If M is a DFA with n states, every input letter a defines a function φ(a) on the states, given by (φ(a))(q) = δ(q, a). Given any string w, we can define a function φ(w) on the states as the ordered composition of the φ(a)'s in w.

Regular Languages and Monoids

- So to determine whether w is in L(M), we can look up φ(a) for each letter in w, compose these n functions together to get φ(w), and determine whether $(\varphi(w))(q_0) \in F$. From a circuit point of view, this is all easy except for the iterated multiplication of n elements of T_k, where k is the number of states in M.
- But this iterated multiplication is clearly in NC^1 if k is a constant, since any function with O(1) inputs and outputs is in NC^0.

Programs Over Monoids

- This gives us an equivalent way to look at BWBP. Fix a finite monoid M. An **M-program** of **length** t is a sequence of t **instructions**, each of which is a triple (i, σ, τ) where σ and τ are in M. The **yield** of (i, σ, τ) is σ if $x_i = 0$ and τ if $x_i = 1$. The yield of the program is the composition of the yields of the instructions. The language of the program is the set of strings whose yield is in some fixed subset F of M.

Programs Over Monoids

- It turns out that classes of monoids, previously studied by algebraists, correspond to circuit classes. Poly-length programs over **aperiodic** monoids, for example, are equivalent to AC^0.
- Programs over **groups** (or "permutation branching programs) are an interesting special case. I was able to prove that programs over S_3 could do AND in exponential size, but not in polynomial size.

Simulating Circuits with S_5

- A reasonable conjecture would be that no program over a group could do AND in polynomial size, much less majority.
- But it turns out that once the group is complicated enough, programs over it are surprisingly powerful.
- **Theorem:** The language of any fan-in two circuit of depth d can be decided by an S_5 program of length 4^d. (Hence BWBP = NC^1.)

Permutation Preliminaries

- If $X = \{1,2,3,4,5\}$, we can write a permutation in **cycle form**. For example, (1 3 5 4 2) is the permutation that takes 1 to 3, 3 to 5, 5 to 4, 4 to 2, and 2 to 1. If a permutation has more than one cycle we concatenate the cycles, as in "(1 5 3)(2 4)".
- Two permutations α and β with the same cycle structure are **conjugate**, meaning that there exists γ such that $\beta = \gamma\alpha\gamma^{-1}$. In particular, any two five-cycles are conjugate.

Five-Cycle Recognition

- Let L be a subset of $\{0,1\}^n$. We say that an S_5 program f **five-cycle recognizes** L if f yields a five-cycle (a b c d e) when $w \in L$ and yields the identity id when $w \notin L$. We'll show it doesn't matter what a, b, c, d, and e are.
- We will prove that if L is decided by a circuit of depth d, it is five-cycle recognized by a program of length at most 4^d. Of course we will prove this by induction on d.

Adjusting Programs

- **Lemma:** Let f be a non-empty S_5-program of length t, and let α and β be any permutations. Then there exists a program g of length t such that for any string w, $g(w) = \alpha f(w)\beta$.
- **Proof:** Multiply the permutations in the first instruction of f on the left by α, and the permutations in the last instruction of f on the right by β.

Starting the Proof

- Base case: d = 0, so we need a program of length 1 to simulate an input gate or negated input gate. The single instruction is just (i, id, (1 2 3 4 5)) or (i, (1 2 3 4 5), id).
- NOT case: Given a program that yields id when $w \notin L$ and (a b c d e) when $w \in L$, use the Lemma to multiply the yield by (e d c b a). This gives (e d c b a) when $w \notin L$ and id when $w \in L$. Since one five-cycle is as good as another by the Lemma, we are done without increasing the length at all.

The Key Step

- Let's say that $L = L_1 \cap L_2$, so that our circuit of depth d has an AND gate at the top. (We can simulate OR gates with AND and NOT.)
- By the IH, we have programs f_1 and f_2 five-cycle recognizing L_1 and L_2. Since we can pick the five-cycles at will, we will have f_1 yield (1 2 3 4 5) if $w \in L_1$ and have f_2 yield (1 3 5 4 2) if $w \in L_2$.

The Key Step

- We will also make g_1 that five-cycle recognizes L_1 yielding (5 4 3 2 1), and g_2 five-cycle recognizing L_2 with yield (2 4 5 3 1).
- Our program f will just be the concatenation $f_1f_2g_1g_2$. Since each of the IH-derived programs has length at most 4^{d-1}, f has length at most 4^d.
- Now we just have to verify that f five-cycle recognizes $L = L_1 \cap L_2$.

The Key Step

- If w ∉ L_1 and w ∉ L_2, f(w) = (id)(id)(id)(id) = id.
- If w ∉ L_1 and w ∈ L_2, f(w) = (id)(1 3 5 4 2)(id)(2 4 5 3 1) = id.
- If w ∈ L_1 and w ∉ L_2, f(w) = (1 2 3 4 5)(id)(5 4 3 2 1)(id) = id.
- If w ∈ L_1 and w ∈ L_2, f(w) = (1 2 3 4 5)(1 3 5 4 2)(5 4 3 2 1)(2 4 5 3 1) = (1 3 2 5 4).
- So it works, and we are done.

Why Did This Work?

- S_5 happens to have two elements that are conjugate both to one another and to their **commutator**.
- This can only happen in a **non-solvable group**, the smallest of which is A_5 with 60 elements (the even permutations in S_5).
- No similar trick will work in S_4, for example, but there could conceivably be a way to simulate NC^1 with S_4 programs.

Lower Bounds

- There is basically one known lower bound technique for programs over groups.
- It works for S_3 and A_4, showing that while the AND function has exponential-length programs, it doesn't have polynomial-length ones.
- We've conjectured that AND requires exponential length over any solvable group, which is equivalent to "AND ∉ CC^0".

Algebra and Complexity

- So iterated multiplication over a finite group or groupoid is in NC^1.
- Over constant-dimension integer matrices, it seems to be "close to" NC^1 but not in it.
- Over a fixed non-associative structure (a **groupoid**), where iterated multiplication becomes nondeterministic, it becomes complete for the class LOGCFL or SAC^1, which contains NL and is contained in AC^1.

Algebra and Complexity

- We can also ask about the **generation problem**. The input is a structure, a subset, and a target element, and we are asked whether any product of elements from the subset equals the target.
- Over groupoids this problem is P-complete.
- Over finite groups it is in L, but "almost in" AC^0 so that it is not complete for any class that includes parity, such as NC^1 or L.