6.841 Advanced Complexity Theory Feb 17, 2009

Lecture 4

Lecturer: Madhu Sudan *Scribe: Adam Spanbauer*

1 Introduction

Today's lecture will focus on the proof of Barrington's Theorem:

Barrington's Theorem: All polynomial-size formulas have an O(1) width braching program.

Barrington proved, in particular, that all polynomial-size formulae have a width 5 branching program. His proof used a lot of advanced algebra, so the proof presented here will show that all polynomial-size formulae have a width 8 branching program. The proof was discovered by Ben-Or and Cleve.

2 Summary of concepts covered in previous lectures

2.1 Non-uniform complexity

For each input length, we are allowed to do something different – the mechanism for doing this is assigning an *advice* string to each integer, one for each input length, and giving it to the TM along with the input.

The languages that can be decided by a polynomial time TM with polynomial length advice is called P/POLY and is equivalent to polynomial sized circuits.

An interesting unresolved question in complexity theory is whether $NP \not\subseteq P/POLY$.

2.2 Circuits

Given by a Directed Acyclic Graph (DAG). Has source nodes for inputs, a collection of gates, connections, and an output. Circuit gates can have arbitrary fan-out, which means that the output from a gate can be used arbitrarily many times as an input elsewhere.

2.3 Branching programs

Branching programs are a weaker model of computation than circuits.
BPs roughly correspond to space complexity, that is, they can be thought of as trying to measure non-uniformly what the space-complexity is.

A BP is a DAG with a few extra restrictions. Every BP has a starting point, variable nodes, and output nodes. Being at a variable node corresponds to reading that variables value. There are two edges out of each variable node: a zero edge and a one edge. The BP outputs a zero or a one based on the type of output node it lands on. To run a BP, follow the edges corresponding to the read variable values until an output node is reached.

Every function has a branching program of $O(2^n)$ that computes it, since the branching program can just branch and have 2^n output nodes, one corresponding to each possible input to the function.

We can compute any function using a constant width branching program by writing down a 3-CNF formula for it, and then checking each term, one after the other, outputting zero if any of the terms are zero, and outputting one if it gets through all of the terms.

2.4 Formulae

Formulae are the weakest model of computation of the three.

They don't really correspond to anything, but they are useful in trying to prove lower bounds. The idea would be to prove lower bounds for formulae, then use those results to prove things about branching programs, and then to prove things about circuits.

Formulae are just circuits with a fan-out of one, that is, the output of each gate can only be used once.

2.5 Relations between the constructs

Formula size roughly $\geq$ BP size roughly $\geq$ circuit size, in the sense that if you have a formula, you can always come up with the same size or only slightly larger equivalent BP, and if you have a BP you can always come up with the same size or only slightly larger equivalent circuit.

While a branching program can compute any function in $O(2^n)$ nodes, or alternatively in constant width, the real question is whether or not one can have both small size and constant width. In particular, we are looking to show that every formula can be simulated by a constant width branching program.

3 Example for BP, checking if the sum of the inputs is 0 mod 5

An example of a linear length, constant width BP is a BP to compute the function that returns 1 if the sum of the inputs is 0 mod 5, and 0 otherwise. The BP simply keeps track of what the current sum of the inputs mod 5 is by having 5 nodes at every level. (Picture will be here as soon as I know how to do it.)

4 Majority function

MAJ gives 1 if more than $\frac{n}{2}$ of the inputs are 1, or 0 otherwise.

You can compute MAJ with a polynomial size formula, here are some useful facts:

Define the function:
$E_{[a,b,i]}(x_a, x_{a+1}, ..., x_b) = 1$ if exactly i of the $x_a..x_b$ are 1s. Then MAJ is $\bigvee_{i=\frac{n}{2}+1}^{n}(E_{[1,N,i]})$.
Additionally, $E_{[a,b,i]} = \bigvee_{j=0}^{i}(E_{[a,mid,j]} \wedge E_{[mid+1,b,i-j]})$.

5 Buildup to Barrington's theorem

Excercise 1: verify that there is a polynomial sized (2-AND, 2-XOR, NOT) formula for MAJ.
Excercise 2: if f has a polynomial sized (2-AND, 2-XOR, NOT) formula then it has a log-depth formula.

There was some effort to show that MAJ does not have an O(1) width polynomial length BP. It is trivial to show that any width 2 BP for MAJ is exponentially long, and there is a non-trivial proof that any width 3 BP for MAJ is exponentially long. Nothing is known about width 4 BPs, but Barrington showed that there is a width 5 BP for MAJ that is polynomial length.

Let us define a model of computation that will make the proof much easier.
A *Register machine* holds 3 bits of information (R_1, R_2, R_3), and at each step it can perform some operation on these bits. It has polynomially many instructions. Each instruction is of the form $R_i \leftarrow R_j + x_n R_k + (R_l$ if needed).

Our input will be (R_1, R_2, R_3) and $x_1..x_n$, and we get our output by looking at the final register values: $(R_1, R_2, R_3 + f(x_1..x_n)R_1$. This means that if we give $(1, 0, 0)$ as input, we'll get $(1, 0, f(x_1..x_n))$.

Each instruction can be simulated by a width 8 branching program by just keeping track of the 3 bits using the $2^3 = 8$ nodes per level, and hard-coding in the instructions to the BP.

Now we have everything neccesary to prove Barrington's Theorem.

6 Proof of Barrington's Theorem

Lemma: Let f be a function computed by the depth d formula of 2-AND, 2-XOR, and NOT gates. Then f has a register machine of length 4^d computing it.
Proof by induction on d: depth=0 case:
The only possibility is that $f(x_1, ...x_n) = x_i$. Then the register machine with the single instruction $R_3 \leftarrow R_3 + x_i R_1$ computes it.
$f =$ NOT f_1 case:
$R_3 \leftarrow R_3 + f_1 R_1$
$R_3 \leftarrow R_3 + R_1$
This gives the output $(R_1, R_2, R_3 + R_1 + R_1 f_1) = (R_1, R_2, R_3 + R_1(NOT f_1))$

$f = f_1$ XOR f_2 case:
$R_3 \leftarrow R_3 + f_1 R_1$
$R_3 \leftarrow R_3 + f_2 R_1$
This gives the output $(R_1, R_2, R_3 + R_1 f_1 + R_1 f_2) = (R_1, R_2, R_3 + R_1(f_1 + f_2))$

$f = f_1$ AND f_2 case:
$R_2 \leftarrow R_2 + f_1 R_1$
$R_3 \leftarrow R_3 + f_2 R_2$
$R_2 \leftarrow R_2 + f_1 R_1$
$R_3 \leftarrow R_3 + f_2 R_2$
This gives the output $(R_1, R_2, R_3 + R_1 f_1 f_2)$

Since the largest number of instructions used for any gate was 4, the most instructions our register machine could have is 4^d. ■

Barrington's Theorem: All polynomial-size formulas have an O(1) width braching program.
Proof: By excercise 2, the formula has an equivalent log-depth formula. Then by the previous lemma, there is a register machine with $4^{log\ n}$ instructions, that is, polynomial length instructions. ■