6.841 Advanced Complexity Theory February 9, 2005

Lecture 3

Lecturer: Madhu Sudan *Scribe: Ilya Baran*

1 Overview

In this lecture we looked at three non-uniform computation models, defined measures of their complexity, and proved some relationships between them. The models were circuits, formulas, and branching programs and the complexity measures were size, depth and width.

2 Non-Uniform Turing Machines

In the last lecture we defined circuits, the basic non-uniform model of computation as a DAG of input nodes, logic gates, and output nodes. To relate circuit families to Turing Machines, we provide a Turing Machine with a different "advice string" for every input length. So a *non-uniform Turing Machine* is a two-input TM, $M(x, y)$ together with a sequence of advice strings $\bar{a} = a_1, a_2, \ldots$. Its language is defined as

$$L(M, \bar{a}) = \{w \mid M(w, a_{|w|}) = 1\}.$$

If M's resource use corresponds to a complexity class $\mathcal{C}$ and the advice string length is bounded as $|a_n| = O(\ell(n))$ for some function ℓ, then $L(M, \bar{a})$ is said to be in the complexity class $\mathcal{C}/\ell$. In particular, languages recognizable by polynomial time TMs with polynomial length advice strings are in the class $\mathbf{P}/poly$.

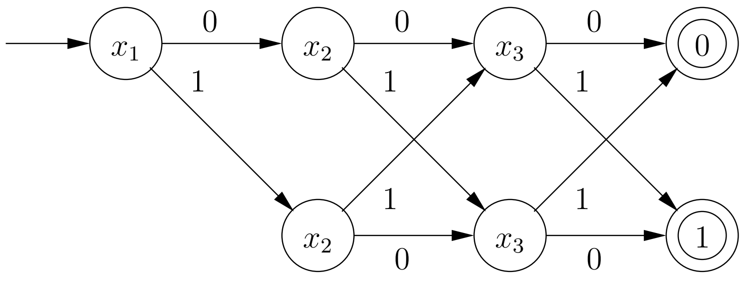
It is easy to show that a $\mathbf{P}/poly$ non-uniform TM is equivalent to a polynomial size circuit family: a circuit can be simulated by a non-uniform TM if a_n is the description of the circuit for inputs of size n. Conversely, a circuit family can simulate a non-uniform TM if a_n is hard coded into the circuit for simulating the TM on inputs of size n.

3 Weaker Non-Uniform Models

We also considered two weaker non-uniform computation models: formulas and branching programs. Formulas are circuits with outdegree 1 (except that input nodes may have arbitrary outdegree).

A branching program (BP) computing a function $f(x_1, \ldots, x_n)$ is a DAG with many branching nodes and two output nodes. Each branching node is labeled with one of the x_i's and has outdegree 2, with the outgoing edges labeled 0 and 1. One of the branching nodes is marked as the starting node. The output nodes have no outgoing edges and are labeled 0 and 1. The BP computes by starting at the starting node and at each node following the edge whose label corresponds to the value of the variable with which the node is labeled. The computation ends at one of the output nodes and that node's label determines the value of f.

For example, the following branching program computes $x_1 + x_2 + x_3 \pmod 2$:



4 Complexity Measures

Circuits and formulas have two natural measures: size and depth. The size of a circuit or a formula is the number of nodes and in some sense corresponds to time. The depth of a circuit or a formula is the length of the longest path in the DAG. This corresponds to parallel computation time.

For BPs, size and depth are defined analogously, but size represents only an "upper bound" on time (because for any particular input, most of the BP may be irrelevant), while depth represents only a "lower bound" on time (because for any boolean function we can make a BP of depth n).

We may also consider another measure of Branching Program complexity: width. A branching program is called *layered* if its nodes can be partitioned into sets (layers) $L_1, L_2, \ldots, L_m$ so that edges always go from layer L_i to L_{i+1}. Then the width of a Branching Program is defined to be the maximum size of a layer. The logarithm of the width is in some sense a lower bound on space.

5 Relationships Between Complexity Measures

We can show some easy relationships between the complexities of different models.

Given a boolean function f, let $\text{size}_C(f)$, $\text{size}_F(f)$ and $\text{size}_B(f)$ denote the minimum size of a circuit, formula, and branching program for f, respectively. Similarly define $\text{depth}_C(f)$, $\text{depth}_F(f)$, and $\text{depth}_B(f)$.

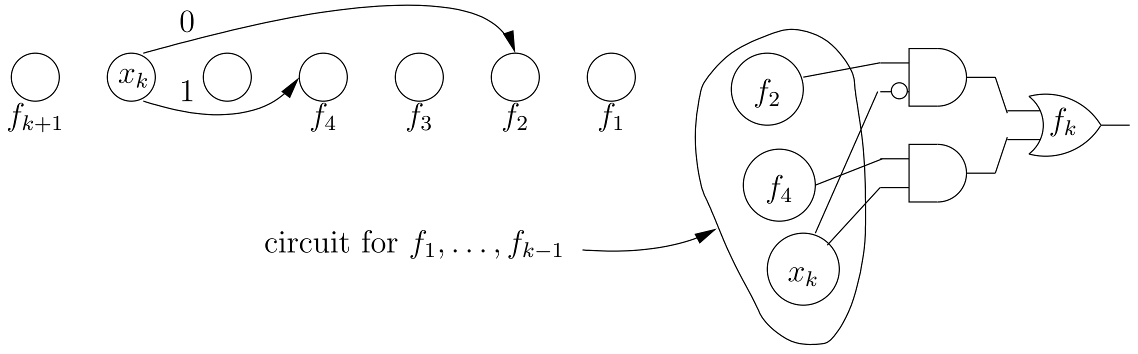
We can show that for all f

1. $\text{size}_C(f) \leq \text{size}_F(f)$
2. $\text{size}_C(f) = O(\text{size}_B(f))$
3. $\text{size}_B(f) \leq \text{size}_F(f) + 2$ (if formulas contain only NOTs, ANDs and ORs)
4. $\text{depth}_C(f) = \text{depth}_F(f)$
5. $\text{depth}_F(f) = \Theta(\log \text{size}_F(f))$ (if formulas have bounded fan-in)

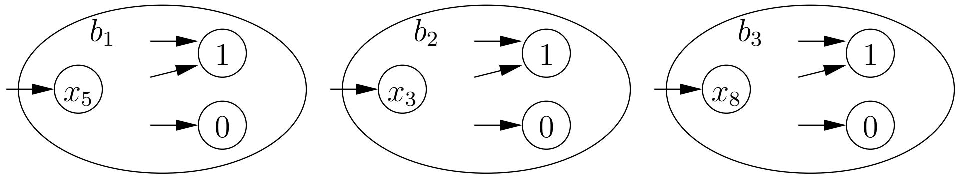
The reasons the above are true are:

1. is obvious because every formula is a circuit.

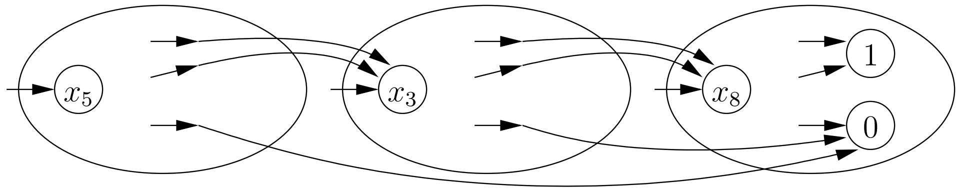
2. follows because we can simulate every BP with a circuit of roughly the same size. Lay out the nodes of the BP in topological order from left to right and construct the circuit inductively from right to left. Suppose that the functions at the nodes of a branching program (from right to left) are $f_1, f_2, \ldots$. Given a circuit that has nodes whose outputs are $f_1, \ldots, f_{k-1}$, we can construct a circuit with $O(1)$ more gates that also has a node that outputs f_k. Suppose the k^{th} node in the BP (from right) reads x_i and branches to x_{i-i_1} and x_{i-i_2} on 0 and 1 respectively. Then we add a node to the circuit whose value is $f_k = (x_i \wedge f_{i-i_2}) \vee (\bar{x}_i \wedge f_{i-i_1})$, which can be done with $O(1)$ gates.



3. is true because we can construct a BP that simulates a formula and whose size is the number of leaves in the formula plus the two output nodes. The construction proceeds by induction: assuming that we constructed BPs for several subformulas, we can combine them into a BP that computes their AND or OR. To combine BPs $b_1, b_2, \ldots, b_n$ into a BP b that computes their AND, merge b_i's 1 output node with b_{i+1}'s start node. Merge all of the 0 output nodes together. Make b's start node b_1's start node and $b's$ outputs are the b_n's 1 output and the merged 0 outputs. See the figure below. ORs are obtained similarly and NOTs are trivial.



the BP for the AND of the 3 BPs above is:



4. has two parts: $\text{depth}_C(f) \leq \text{depth}_F(f)$ holds because every formula is a circuit, and $\text{depth}_C(f) \geq \text{depth}_F(f)$ holds because the straightforward method for converting a circuit into a formula (and possibly causing exponential increase in size) does not increase depth.

5. also has two parts: $\text{depth}_F(f) = \Omega(\log \text{size}_F(f))$ is trivial because the fan-in is bounded. $\text{depth}_F(f) = O(\log \text{size}_F(f))$ was left as an exercise. The idea is that we can convert any

formula to one of logarithmic depth by "rebalancing" it. This can be done by picking an edge that splits the formula tree into roughly equal pieces, recursively rebalancing the pieces, and using a multiplexer as the new root.

6 Neciporuk's Lower Bound on Formula Size for an Explicit Function

A simple counting argument shows that some boolean functions require exponentially large formulas. It is easy to see that there are 2^{2^n} boolean functions of n variables. On the other hand, there are approximately $4^{2^{n/c}}$ binary trees of size $2^{n/c}$ and each tree may be converted into a formula in at most $g^{2^{n/c}}$ distinct ways (where g is the number of different gates allowed). Therefore, there are at most approximately $(4g)^{2^{n/c}} = 2^{2^{n/c}\log_2(4g)}$ formulas of size $2^{n/c}$. For any $c > 1$, this is smaller than 2^{2^n} for all sufficiently large n.

It is much harder to construct an explicit family of boolean functions that requires large formulas. We describe a family that requires formulas of almost quadratic size. This is argued by restricting the function to a subset of its inputs and then doing a counting argument similar to the one above.

The function f_n is defined on $2n \log n$ boolean variables, grouped into n groups. So for $i = 1$ to n, let $x_i = (x_{i_1}, x_{i_2}, \ldots, x_{i_{2\log n}})$. Define f_n to be:

$$f_n(x_1, x_2, \ldots, x_n) = \begin{cases} 1 & \text{if for some } i \neq j,\ x_i = x_j \\ 0 & \text{otherwise} \end{cases}$$

We will show that this function family requires formulas of size $\Omega(n^2)$ by proving that for every i, there must be $\Omega(n)$ leaves reading from x_i.

Assume that some x_i is read in k leaves. Consider the functions obtained by fixing all inputs except x_i. They are $f_{\bar{a}}(x_i) = f(a_1, a_2, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_n)$ where the a's are all distinct. Then $f_{\bar{a}}(x_i) = 1$ precisely when $x_i = a_j$ for some j, so these functions are in one-to-one correspondence with subsets of size $n-1$ from among the $2^{2\log n} = n^2$ possible a_i's. There are clearly $\binom{n^2}{n-1} \geq 2^n$ of them.

On the other hand, from our original formula, we can construct a formula for $f_{\bar{a}}$ by collapsing and propagating the a's values until the only leaves left are those that read x_i. Using the counting argument in the first paragraph, we conclude that there are at most $2^{O(k)}$ formulas on k leaves. But each of the $f_{\bar{a}}$'s needs a different formula, so $2^{O(k)} \geq 2^n$ and therefore $k = \Omega(n)$.