6.841 Advanced Complexity Theory Feb 14, 2007

Lecture 3

Lecturer: Madhu Sudan *Scribe: Eitan Reich*

1 Overview

In this lecture we look at several Non-uniform models of computation:

- Turing Machines with Advice
- Curcuits
- Formulas
- Branching Programs

We will also define their complexity measures and prove some relationships between the models.

When we say that a model of computation is non-uniform, we mean that the non-uniformity is with respect to the size of the input. Given a language L, we will define the members L_n of that language of a given length as:

$$L_n = L \cap \{0,1\}^n$$

A non-uniform solution to a computation problem is a collection of Algorithms $\{A_n\}_{n\geq 0}$, where A_n solves L_n.

2 Turing Machines with Advice

A Turing Machine M with advice $\bar{a} = \{a_1, a_2, ..., a_n, ...\}$, where $a_n \in \{0,1\}^*$ is a two input Turing Machine whose second input is a member $a_n \in \bar{a}$ where n is the size of the first input. We define the language accepted by such a Turing Machine as follows:

$$L(M, \bar{a} = \{x | M(x, a_n) \text{ accepts where n} = |\text{x}|\}$$

The resources used by M can be measured as a combination of two measures. One is a measured as a function of the length of the first input length. The novel part of the resources is $|a_n|$, the length of the advice on inputs of length n. When a machine M runs in time $t(n)$, where the advice $\hat{a}$ is fixed and $|a_n| = l(n)$, we say that:

$$L(M, \hat{a}) \in DTIME(t(n))/l(n)$$

One particular class of languages we will be looking at is those recognized by Turing Machines running in polynomial time with polynomial size advice:

$$P/poly = \bigcup_{polyp} DTIME(p(n))/p(n)$$

Two other complexity classes that are studied are $NP/poly$ and $L/poly$. It is obvious that P is contained in $P/poly$. We also note that $P/poly$ contains some interesting languages not in P, such as a modified version of the halting problem (because a bit representing halts/does not halt can be given as advice) which is not even solvable in the standard computation model. This class is interesting because people believe it might be useful for showing NP different from P. The idea is that showing NP different from $P/poly$ would imply NP different from P, and the first problem might be easier to tackle because certain combinatorial techniques can be applied, due to the equivalence between $P/poly$ and polynomial size circuits, which we will study in the next section.

3 Polynomial Sized Circuits

A Circuit is a model of computation in which the input bits are wired through a collection of logical gates to produce a set of output bits. These logical gates will typically be the AND, OR, and NOT gates, but in general we will define circuits in terms of a Basis of gates $\{g_1, ..., g_k\}$. The resulting circuit is a Directed Acyclic Graph (DAG) whose vertices represent instances of one of the basis gates and whose edges represent wires carrying a bit as output from one gate and input into another gate. There are n source vertices (having in-degree 0) $x_1, ..., x_n$ representing the input to the circuit. The remaining vertices are associated with a basis function. A number of the vertices, say m, will be designated as output.

4 Complexity Measures

- One complexity measure for a circuit is its size, which we will define as the number of edges in the DAG. The size of a circuit can be seen as the non-uniform time measure for the computation using the circuit. Given this definition for the size of a circuit, we can now define the circuit size of a function, $Ckt - Size(f)$ as the size of the smalest circuit computing the function f.
- Another complexity measure to consider is the depth of a circuit, which is defined as the length of the longest path in the DAG. The depth is equivalent to the measure of non-uniform parallel time.

In general we will be considering functions over the following basis:

$$NOT, \{\text{k} - \text{ary AND}\}_{\text{k}\geq 0}, \{\text{k} - \text{ary OR}\}_{\text{k}\geq 0}$$

We can see that there is an equivalence between Polynomial Circuits and Turing Machines with Advice. Given a collection of Polynomial Circuits, we can create a Turing Machine that accepts the circuit description as advice and simulates the computation of a circuit. Similarly, given a Turing Machine with Advice, we can construct circuits that simulate the execution of the Turing Machine on each member a_n of the advice. We will state without proof the following two relations between the complexity measures of Turing Machines with Advice and Polynomial Circuits:

- $DTIME(t(n))/l(n) \subseteq Ckt - Size(o(t(n) + l(n))^2)$
- $Ckt - Size(s(n)) \subseteq DTIME(s(n)^2)/s(n)logs(n)$

5 Formulas

To model formulas as a computation, we wish to capture the idea that a formula is a function of the input variables in which no intermediary values are stored. To model this in terms of circuits, we define formulas as circuits in which the out-degree of non-input nodes is 1.

We define the complexity measures of Formula-Size and Formula-Depth just as we did for Circuits.

6 Branching Programs

Like Circuits, Branching Programs are defined in terms of DAGs. Branching programs capture the idea of non-uniform space just as Circuits did for non-uniform time. In a Branching Program, the DAG has a designated start node with in-degree 0 and two specified output nodes representing the states accept and reject. All nodes except the output nodes have out-degree 2. The two edges emanating from each node are labeled by complementary literals, denoting the path to be taken from the node based on which literal evaluates to true.

The two complexity measures we define for Branching Programs are the Size and the Width. The Size of a Branching Program is defined as the number of nodes in the DAG. We could just as well define size in terms of edges because the number of edges is always twice the number of nodes.

The measure we will be most interested in is the Width because this captures the amount of space necessary to perform the computation. In order to define the width we first define a Layering of the Branching Program. A Branching Program has layers $L_1, ..., L_k$ if $L_1, ..., L_k$ form a partition of the nodes of the DAG and if every edge goes from L_i to $L_i \cup L_{i+1}$. In other words, every edge must connect a node with another node in the same layer or with a node in the next layer. The Width of a Branching Program can then be defined as:

$$Width(BP) = \min_{layering\ L_1,...,L_k} \{\max_i \{|L_i|\}\}$$

We note that we can compute the XOR function using a Branching Program with width 2. In fact, Regular Languages have constant width Branching Programs because the Finite State Machine for a Regular Language can serve as the Branching Program.

We can define the Non-uniform space complexity of a function as the log of the minimum width Branching Program computing this function (so long as the BP also has polynomial size). We can use this definition of space complexity to show that the existence of a function f that is computable in P but doesn't hvae a polynomial size Branching Program would imply that $P \neq L$.

7 Relationships between Complexity Measures

We first state some relationships between the size and depth of formulas:

- Formula-Size(L) $\leq$ exp(Formula-Depth(L)) (for any finite basis)
- Formula-Depth(L) $\leq$ log(Formula-Size(L)) (for the $\{\vee, \wedge, \neg\}$ basis)
- Formula-Size in Basis $B_1 = O$(F-Size in Basis $B_2)^{O(1)}$

Next we note the following relationship between the size complexity of the three non-uniform computation models discussed in lecture today (under the $\{\vee, \wedge, \neg\}$ basis):

$$Formula - Size(f) \geq \Omega(BP - Size(f)) \geq \Omega(Ckt - Size(f))$$

We will now show an existential lower bound for the Circuit Size of a function:

$$\exists f : \{0,1\}^n \rightarrow \{0,1\}\ s.t.\ Ckt - Size(f) \geq \Omega(\frac{2^n}{n})$$

We just give the beginning of the proof here, the rest of which is left as an exercise. We first claim that the number of circuits of size $\leq S$ is $2^{O(S \log(S))}$. Since the number of functions on n bits is 2^{2^n}, we know that if $2^{S \log(S)} \leq 2^{2^n}$ then there exists such a function f that is not computable using a circuit of size S. Choosing S appropriately we achieve the desired lower bound.

We can show an upper bound on formula size in that every function f on n bits has F-Size(f) $\leq O(2^n)$. To see why this is true, we note that we can write the function f in terms of two functions f_0 and f_1 on $n-1$ bits:

$$f(x_1, ..., x_n) = ((x_1 = 0) \wedge f_0(x_2, ..., x_n)) \vee ((x_1 = 1) \wedge f_1(x_2, ..., x_n))$$

By induction we can show that the size of the formula is bounded by $O(2^n)$.

Just as we have space hierarchy theorems in the standard computational model, we also have the Circuit-Hierarchy theorem which states that:

$$\forall s(n) \ll \frac{2^n}{n^2}, Size(o(s(n))) \subsetneq Size(s(n)\log(s(n)))$$

The following are the best lower bounds we have achieved so far in our non-uniform models:

- Best known Circuit Size lower bound is $4.5n$ (due to [Raz])
- Best known Formula Size lower bound, using $\{\vee, \wedge, \neg\}$ as a basis, is around n^3
- Best known Branching Program Size lower bound is $(\frac{n}{\log n})^2$