CSE531: Computational Complexity Theory January 11, 2012

Lecture 3 Hierarchies for Circuits, and NP

Lecturer: Anup Rao

1 Counting based Lowerbounds for Circuits

We define the class $\mathsf{SIZE}(s(n))$ to be the set of functions $f : \{0,1\}^n \to \{0,1\}$ that can be computed by a circuit family of size $s(n)$.

Last time we discussed the following theorem:

Theorem 1. *Every function $f : \{0,1\}^* \to \{0,1\}$ is in $\mathsf{SIZE}(O(2^n/n))$.*

This time we shall prove that the theorem is close to being tight:

Theorem 2. *For every large enough n, there is a function $f : \{0,1\}^n \to \{0,1\}$ that cannot be computed by a circuit of size $2^n/3n$.*

Proof To specify a circuit of size s, we need to describe the function that is computed at each gate (which takes 4 bits) and specify where each of its inputs come from (which takes $2 \log s$) bits. Thus, in total it takes $2s \log s + 4s$ bits to define the entire circuit.

Thus the total number of circuits of size s, for s large enough is at most $2^{3s \log s}$. Thus the total number of circuits of size $2^n/n$ is less than $2^{3 \cdot \frac{2^n}{3n} \cdot n} = 2^{2^n}$. On the other hand, the number of functions $f : \{0,1\}^n \to \{0,1\}$ is exactly 2^{2^n}, thus not all these functions can be computed by a circuit of size $2^n/(3n)$. ■

We can use this theorem to prove a hierarchy bound for space.

Theorem 3. *There is a constant c such that for every functions $s(n), s'(n)$ satisfying $2^n/n > s'(n) > cs(n) > n$, we have that $\mathsf{SIZE}(s(n)) \subsetneq \mathsf{SIZE}(s'(n))$.*

Proof Suppose every function on n bits can be computed using a circuit of size $k2^n/n$. Let ℓ be such that $k2^\ell/\ell = s'(n)$. Then every function on ℓ bits can be computed by a circuit of size $s'(n)$. On the other hand, there is some function on ℓ bits that cannot be computed using a circuit of size $2^\ell/3\ell$. Thus, as long as $s'(n) > 3ks(n)$, no circuit of size $s(n)$ can compute everything computed by a circuit of size $s'(n)$. ■

2 Some interesting complexity classes

In order to give a definition that captures the notion of *efficient* computation, there are two important considerations. First, efficient computation must allow at least linear (i.e. n) time just to read the input. Secondly, if an efficient computation makes a call to a subroutine that is also efficient, then we would want to consider the composed computation to also be efficient. The smallest computational class that allows both of these is:

Definition 4. $\mathbf{P} = \bigcup_{c \geq 1} \mathsf{DTIME}(n^c)$

Of course there is a whole spectrum of classes above P. For example:

Definition 5. $\mathbf{EXP} = \bigcup_{c \geq 1} \mathsf{DTIME}(2^{n^c})$.

3 NP

Definition 6. *(Class* **NP***)* $f : \{0,1\}^* \to \{0,1\}$ *is in* **NP** *if there exists a polynomial* p *and a polynomial time machine* M *such that for every* $x \in \{0,1\}^*$,

$$f(x) = 1 \Leftrightarrow \exists w \in \{0,1\}^{p(|x|)}, M(x,w) = 1$$

M is usually called the *verifier* and w is usually called the witness or certificate.
Examples:

- Independent set: (G, k) does the graph have an independent set of size k?
- Subset sum: Given a list of numbers $A_1, \ldots, A_\ell, T$, is there some subset of the numbers that sums to T?
- Composite numbers: Given a number N decide if it is composite or not.

4 Reductions

Definition 7. *A function* f *is polynomial time reducible to a function* g *if there is a polynomial time computable function* h *such that* $f(x) = g(h(x))$. *We write* $f \leq_P g$.

Note that the above definition is not the only one that makes sense. In general it makes sense to allow our reductions to make multiple calls to the problem being reduced to. However, we will be able to prove many of our results using the stronger notion above, so that is what we shall use.

Definition 8. *We say* f *is* **NP***-hard if* $g \leq_P f$ *for every* $g \in \mathbf{NP}$. *We say* f *is* **NP***-complete if* f *is* **NP***-hard and* $f \in \mathbf{NP}$.

Theorem 9.

- *If* $f \leq_P g$ *and* $g \leq_P h$, *then* $f \leq_P h$.
- *If* f *is* **NP***-hard and* $f \in \mathbf{P}$, *then* $P = NP$.
- *If* f *is* **NP***-complete, then* $\mathbf{P} = \mathbf{NP}$ *if and only if* $f \in P$.

5 NP-complete problems

Of course, the above definitions only make sense because we do know of examples of **NP**-complete problems.

Definition 10. $\mathsf{CKT-SAT} : \{0,1\}^* \to \{0,1\}$ *is the function that views its input as a circuit* C *and outputs* 1 *iff* $\exists x$ *such that* $C(x) = 1$.

Theorem 11. $\mathsf{CKT} - \mathsf{SAT}$ *is* **NP***-complete.*

Proof Suppose $g \in \mathbf{NP}$, and let M be a verifier for g. Then the reduction will build the circuit C_x that computes $M(x, w)$, where here w are the input variables and x is the input. Since $g(x) = 1$ if and only if there exists w such that $C_x(w) = 1$, we can determine the value of g by computing $\mathsf{CKT} - \mathsf{SAT}(C_x)$. ■