In the last lecture we introduced the model of polynomial-size boolean circuits. This is a computational model that has some realistic features. Also, there is hope that by studying what is difficult for circuits we can make progress on the P versus NP question. Although proving that NP has no polynomial-size circuit families is formally more difficult than proving P does not equal NP, circuits may be easier to reason about than Turing Machines.

1 Constant depth circuits

Questions in complexity theory are often quite hard. To get started we look for easier variants of the question that might be more tractable. To show that NP $\not\subseteq$ P/poly it is sufficient to show that SAT (or some other NP-complete problem) does not have polynomial-size circuit families. However reasoning about the power of general circuit families for SAT has turned out to be quite difficult. One way to make the problem easier is to put some restrictions on the type of circuit we are considering.

We have already seen a particular very simple kind of circuit: A CNF formula. We can think of a CNF formula as a circuit with two "layers" of non-input gates: the bottom layer are the OR gates (the clauses) and the top layer has one AND gate, representing the output of the formula. There is another kind of circuit with two layers, in which the bottom layer has AND gates and the top layer has an OR gate.

In general, given any circuit, we can layer the gates in a similar manner, so that layer 0 contains the input gates, and layer i contains exactly those gates whose values can be computed from the gates at levels $i-1$ or lower. With a little extra work we can arrange the gates so that all gates in the same layer are of the same type, and the layers alternate (so even layers have AND gates and odd layers have OR gates, or vice versa). This transformations does not affect the number of layers.

The *depth* of a circuit is the layer in which the output gate is found. Alternatively, it is the length of the longest directed path in the underlying graph.

In general, a circuit of size s can have depth as large as $\Omega(s)$. It is believed that in the containment P $\subseteq$ P/poly, there are problems in P that require circuits of depth $\Omega(n)$ on inputs of length n. Therefore polynomial-size circuit families of depth $o(n)$ are considered unlikely to capture all of P. However, coming up with examples of problems in P that cannot be computed by such circuits could be a useful warmup towards P $\neq$ NP.

In fact, we will study the class of circuits that have *constant* depth – that is, the depth is completely independent of the input length. In some sense, these are the simplest kinds of circuits that can do some nontrivial computations. CNFs and DNFs can already compute all functions, although some of them require circuits of exponential size. However, it seems plausible that as we increase the depth of the circuits, more and more interesting things can be computed.

Definition 1. The class AC^0 consists of those decision problems L that admit polynomial-size circuit families $\{C_0, C_1, \dots\}$ such that the depth of C_i is at most c for come constant c.

Although AC^0 circuit families appear extremely limited, they can already compute some nontrivial things. For example the partial function family

$$\mathrm{APXMAJ}(x_1, \dots, x_n) = \begin{cases} 1, & \text{if } x_1 + \cdots + x_n \geq 0.6n, \\ 0, & \text{if } x_1 + \cdots + x_n \leq 0.4n. \end{cases}$$

can be computed by a polynomial-size circuit family of depth 3.

However, we do not even know how to prove that some problem in P requires circuits of depth $\omega(\log \log n)$ on inputs of length n.

2 Circuit lower bounds for parity

What kinds of problems are difficult for AC^0 circuits to solve? To develop some intuition, let's start by looking at polynomial-size depth 2 circuits, in particular DNFs. A DNF is an OR of clauses, each of which is the AND of many literals.

One class of functions that are hard for DNFs to compute are those that are "sensitive" to the values of all their variables – so that changing the value of any variable is likely to affect the value of the function. We illustrate this point by the most extreme such function:

$$\mathrm{XOR}(x_1, \dots, x_n) = \begin{cases} 1, & \text{if } x_1 + \cdots + x_n \text{ is odd}, \\ 0, & \text{if } x_1 + \cdots + x_n \text{ is even}. \end{cases}$$

How can we compute such a function by a DNF? Assume that a DNF C on n inputs computes the XOR function. We claim that each clause of C must depend on all n variables: If any variable is omitted from the clause, then it is possible to set the values of the others so that the clause (and therefore C) is always true; but we can make XOR false by setting the omitted variable to be the XOR of the other ones. So each clause depends on all n variables. Now suppose C has t such clauses $A_1, \dots, A_t$ that depend on all its variables. If we choose a random input $x \in \{0,1\}^n$, we have that

$$\Pr_{x \sim \{0,1\}^n}[C(x) = 1] \leq \sum_{i=1}^{t} \Pr_{x \sim \{0,1\}^n}[A_t(x) = 1] \leq t \cdot 2^{-n},$$

while $\Pr_{x \sim \{0,1\}^n}[\mathrm{XOR}(x) = 1] = 1/2$. So if C computes XOR, it must be that $t \geq 2^{n-1}$. So any DNF for XOR must have size at least $n2^{n-1}$. One can make a similar argument for CNF.

Can circuits of larger depth help compute XOR more efficiently? In fact they can: For example XOR on n variables can be computed by a circuit of depth 3 and size $O(2^{\sqrt{n}})$. Although this is an improvement, it is far from a polynomial.

In fact, XOR cannot be computed by any circuit family of any constant depth and polynomial size:

Theorem 2. *The function* XOR *is not in* AC^0.

This is an example of a circuit lower bound: XOR is a function which is in NP (and in fact in P), but it is not in AC^0. Therefore, NP $\not\subseteq$ AC^0. And while we cannot hope to extend this result to show NP $\not\subseteq$ P/poly, perhaps we can expect some ideas from the proof to be valuable in this endeavor.

To prove Theorem 2 we have to show that for every circuit C that is not too large, C and the XOR function differ on at least one input. Extending our intuition for DNF, we can hope to show that the reason that C and XOR are different is that XOR depends on all its variables, while C might not.

Indeed, we will show that if C is not too large, there is at least one input variable x_i of C and one setting of all the other input variables so that for this setting, C does not depend on the value of x_i. Clearly this is not true for the XOR function, so C cannot compute XOR. But circuits can be very different, so how do we know for which x_i and for which setting of the other variables C will be independent of x_i? To do this we apply the probabilistic method: We will show that for a random choice of i and a random setting of values for the variables $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$, C does not depend on x_i with some nonzero probability.

To establish this fact we will proceed in stages. At each stage we will fix some of the inputs of C in such a way that this fixing will simplify the circuit C: We will show that after an appropriate fixing of some of the inputs, the circuit C can be replaced by an equivalent circuit C' such that C' has about the same size as C (it is only slightly bigger), but its depth goes down by 1. After repeating this process a constant number of times, we will end up with a "small" DNF. By the above argument, a "small" DNF cannot depend on all its variables, so the original C could not have depended on all its variables either.

The heart of the argument is in showing how to reduce the depth of the circuit C after restricting the values of some of the variables. To do so we consider those gates of C which are at the second level from the bottom. Let us assume that this is an OR layer, so each such gates computes a DNF of the input. We will argue that we can always restrict the values in such a way that after restriction, each of these second layer DNFs can be converted to an equivalent CNF of comparable size. After applying this transformation to all level 2 gates the circuit C, whose three bottom layers are of the type AND-OR-AND, is converted to an equivalent circuit C' whose three bottom layers are AND-AND-OR. We can now merge levels 2 and 3 and we end up reducing the depth of C by one (as long as C has depth at least 3 to begin with).

How can we find a choice of restrictions for the input that yields this collapse in the bottom layers of C? Again we use the probabilistic method: We will argue that a random choice of a restriction is likely to yield such a collapse.

We now state the main lemma that allows us to replace DNFs with CNFs after a restriction that randomly assigns some subset of the variables. The parameters may look a bit strange but their choice will hopefully become clear in the proof. For a collection of variables $x_1, \ldots, x_n$, a $\rho(n)$-*random restriction* is a partial assignment to a subset of the variables obtained by the following

random process. For each x_i independently at random,

$$x_i \text{ is assigned } \begin{cases} 0, \text{ with probability } (1-\rho(n))/2, \\ 1, \text{ with probability } (1-\rho(n))/2, \\ \text{remains unassigned, with probability } \rho(n). \end{cases}$$

The lemma says that a random restriction is likely to reduce D to a formula that depends on only a *constant* number of its variables.

Lemma 3. *Let c be a constant and assume n is sufficiently large (in terms of c). Let D be a DNF with n inputs and at most n^c clauses. With probability $1-n^{-c}$, after a $n^{-1/2}$-random restriction, there exists a CNF C of size $n^{c'}$ over the unrestricted variables y such that $D^R(y) = C(y)$ for all unrestricted y, where D^R denotes the restricted version of D. Here c' is some constant that depends only on c.*

With this lemma in hand, we can finish the proof of Theorem 2 as follows. Let's start with an AC^0 circuit family of size $n^c/2$ and depth $d \geq 3$. Assume that the gates at the second layer in this circuit are OR gates. The case of AND gates is similar. By the lemma, each of these DNF reduces to a CNF of size $n^{c'}$ with probability $1-n^{-c}$. By a union bound, all the layer 2 DNFs reduce to CNFs of size $n^{c'}$ with probability greater than $1/2$. Also, by a deviation inequality, the probability that after the restriction, D depends on fewer than $\sqrt{n}/2$ variables is at most $1/2$. It follows that for some choice of restriction, after replacing the layer 2 DNFs with CNFs and merging layer 2 and layer 3, we can replace the original circuit by a new equivalent circuit, whose size is still polynomial in the length of its input, but its depth is one less than the one of the original circuit.

After repeating this operation sufficiently many times we end up with a depth 2 circuit family whose size is polynomial in the number of its variables. We argued that the circuits in such a family cannot depend on all their variables. On the other hand, even after applying all these restrictions, the XOR function will still depend on all of its variables. Therefore the original circuit family cannot compute the XOR function.

Proof of Lemma 3. We will do the proof in two stages. An $n^{-1/2}$-random on the variables of D can be viewed as first applying an $n^{-1/4}$ random restriction, then applying another independent $n^{-1/4}$ random restriction on the unassigned variables.

We will show that with probability $1-n^{-c}/2$, after the first restriction, each clause of D depends on at most $9c$ of the restricted variables. Then we will show that conditioned on this, with probability $1-n^{-c}/2$, after the second restriction D can be replaced with C. The lemma then follows by a union bound.

The first restriction. We write $D = A_1 \vee \cdots \vee A_s$, where $s \leq n^c$ and each A_i is an AND of literals. Let A_i' denote the clause A_i after the first random restriction and after simplifying each clause. For each i, we will bound the probability that A_i' depends on $9c$ or more literals. We consider two cases. If A_i has more than $4c \log n$ literals, then the probability that none of its literals is assigned to 0 (FALSE) in the restriction is at most

$$[(1-n^{-1/4})/2]^{4c\log n} \leq n^{-2c}/2$$

so with probability at least $1 - n^{-2c}/2$, one of the literals of A_i is assigned 0, and A'_i depends on zero literals.

If A_i has at most $4c \log n$ literals, we will argue that the probability that at least $9c$ of them survive the restriction is at most $n^{-2c}/2$. To do this we look at all possible subsets of $9c$ literals and take a union bound. The probability that A'_i has $9c$ or more unassigned literals is at most

$$\binom{4c \log n}{9c} \cdot (n^{-1/4})^{9c} \leq \left(\frac{4c \log n}{n^{1/4}}\right)^{9c} \leq n^{-2c}/2.$$

Now by a union bound, we have that after the first random restriction,

$$\Pr[\text{some } A'_i \text{ has} \geq 9c \text{ literals}] \leq \sum_{i=1}^{s} \Pr[A'_i \text{ has} \geq 9c \text{ literals}] \leq s \cdot n^{-2c}/2 \leq n^{-c}/2.$$

The second restriction. Now let's consider the formula $D' = A'_1 \vee \cdots \vee A'_s$ obtained after the first restriction and let's assume that each clause A'_i depends on less than $9c$ literals. We will show that after the second restriction, it is quite likely that D' can be converted into the desired CNF C.

Let $k = 5c2^{9c} \log n$ and $k_i = (40c2^{9c})^i k$. We will build a tree whose nodes are labeled by DNFs as follows. The root is labeled by D'. Here is how the rest of the tree is constructed. Suppose we have constructed the tree up to level i. To construct level $i+1$, we do the following for every node at level i:

1. If F is a constant (always 0 or always 1), make F a leaf.
2. If there is a collection of $k_i + 1$ clauses of F that are disjoint (i.e. they do not share any variables), make F a leaf.
3. Otherwise, there exists a collection $x_{i_1}, \ldots, x_{i_{9ck_i}}$ of variables that cover all the clauses of F (namely, each clause of F contains at least one of these variables). For every possible partial assignment $x_{i_1} = a_{i_1}, \ldots, x_{i_{9ck_i}} = a_{i_{9ck_i}}$, make a child of F and label it by the DNF derived from F by this partial assignment. Notice that when we go from a node to its children, the number of literals in every clause goes down by 1.

We now claim that with probability at least $1 - n^{-c}/2$, after the second random restriction, all leaves in the tree become constants. Consider the effect of the second restriction on leaves F of the second type at level i. Let F' be the restricted version of F. Each clause of F has at most $9c$ literals, so the restriction sets it to 1 with probability at least 2^{-9c}. Since the clauses do not share variables, they are each set to 1 independently with this probability and so

$$\Pr[F' \not\equiv 1] \leq (1 - 2^{-9c})^{k_i} \leq e^{-k_i 2^{-9c}}$$

Notice that the depth of the tree is at most $9c$, since the size of the clauses shrinks by 1 at each

step. The number of nodes at level i is at most $2^{9c(k_0+k_1+\cdots+k_{i-1})} < 2^{k_i 2^{-9c}/4}$, so by a union bound

$$\begin{aligned} 4\Pr[\text{some leaf } F' \text{ is not constant}] &\leq \sum_{i=0}^{9c} \Pr[\text{some level } i \text{ leaf } F' \text{ is not constant}] \\ &\leq \sum_{i=0}^{9c} 2^{k_i 2^{-9c}/4} e^{-k_i 2^{-9c}} \leq \sum_{i=0}^{9c} 2^{-k_i 2^{-9c}/4} \leq 2 \cdot 2^{-k 2^{-9c}/4} \leq n^{-c}/2. \end{aligned}$$

Now let's assume all leaves of the tree are constant under the second random restriction. In this case, we will derive the CNF C from the tree. Some branches in the tree may be incompatible with the second restriction. After throwing out these branches we get a smaller tree. We look at each path that terminates at a leaf labeled by the constant 0. Each such path represents a partial assignment of some of the variables $x_{i_1} = a_{i_1}, \ldots, x_{i_t} = a_{i_t}$. Such an assignment always makes the restricted DNF D^R false, so we add the following clause to C to represent this fact:

$$\Big(\bigwedge_{a_{i_j}=0} x_{i_j}\Big) \wedge \Big(\bigwedge_{a_{i_j}=1} \overline{x_{i_j}}\Big).$$

The collection of all such clauses exactly represents the formula D^R. How many such clauses are there? Their number is bounded by the number of leaves in the tree, which is at most

$$2^{9c(k_0+\cdots+k_{9c-1})} \leq 2^{k_{9c} 2^{-9c}/4} = n^{2^{O(c^2)}}. \qquad \square$$

3 Increasing the depth

What is the significance of the circuit lower bound for AC^0 we just proved to the P versus NP question? On the positive side, we have an example of a nontrivial model of computation – AC^0 circuits – that provably cannot compute a function in NP. Using reductions we can derive the consequence that AC^0 circuits cannot compute other problems in NP like SAT.[1] On the negative side, the lower bound we proved is way too strong: We have shown that AC^0 not only fails to solve problems in NP but also problems in P! In particular, we expect any reasonable computational device to easily compute the XOR function, so the model of constant depth circuits is much too weak.

How can we beef up constant depth circuits to make them a more realistic model of computation? A natural way is to try and increase the depth. We'll do so in a very careful way, so as to keep the circuit as simple as possible. Let's start with the simple case of an AND gate of unbounded fan-in (in-degree) n. One way to "compute" this gate is to draw a circuit AND gates of depth $\log n$ where each gate in the circuit now has fan-in 2. We can do a similar transformation on the OR gates. If we start with an AC^0 circuit family and apply this transformation to all the AND/OR gates, we obtain a new polynomial-size circuit family where the depth of the nth circuit is now $O(\log n)$, but every gate has fan-in 2. This is the circuit class NC^1.

[1] We have to be careful when we do the reductions – when talking about AC^0 it is not sufficient that the reductions preserve polynomial running time but they must also preserve constant depth.

Definition 4. The class NC^1 consists of those decision problems L that admit circuit families $\{C_0, C_1, \dots\}$ such that every gate in every circuit in the family has fan-in 2 and the depth of C_i is at most $O(\log n)$.

The fact that NC^1 circuit families have polynomial size is automatic: If every gate has fan-in 2 and the depth is $O(\log n)$, then the circuit size is at most $2^{O(\log n)} = n^{O(1)}$.

We just saw that $\mathrm{AC}^0 \subseteq \mathrm{NC}^1$. But is NC^1 more powerful than AC^0? Yes, because NC^1 circuits can compute the XOR function: To compute XOR on n variables, recursively compute XOR on $n/2$ variables and XOR the results together. This gives a circuit of size $O(n)$ and depth $O(\log n)$. It turns out that NC^1 circuit families can also compute the majority functon, which is also not in AC^0:

$$\mathrm{MAJ}(x_1, \dots, x_n) = \begin{cases} 1, & \text{if } x_1 + \cdots + x_n \geq n/2, \\ 0, & \text{if } x_1 + \cdots + x_n < n/2. \end{cases}$$

Which problems are hard for NC^1? Is SAT hard for NC^1? We don't know. So we turn to the most fruitful trick of complexity theory: When you can't prove, reduce to something simpler. Let's look more deeply into NC^1 and see what insights we can get.

For the rest of this lecture we will assume that the fan-in of all circuits and formulas in question is 2.

4 Circuits and formulas

A *(boolean) formula* is a circuit where every gate (apart from the input gates) has out-degree 1. It is believed that polynomial-size families of formula are less powerful than the corresponding circuit families, as they are not allowed to reuse previously computed values.

However, in the case of NC^1 circuit families, it turns out that the out-degree does not make a difference. This follows from the following theorem, which gives a general method for converting a circuit into a slightly larger formula:

Theorem 5. *If f has a (fan-in 2) circuit of size s, and depth d, then it has a formula of size $s2^d$ and depth d.*

Proof. By induction on the depth d. Let C be the circuit for f of size s and depth d and look at the topmost gate G of C. Then $C(x) = G(f_1(x), f_2(x))$, where f_1 and f_2 are the functions computed by the gates that connect into G. By assumption, f_1 and f_2 each have circuits of size at most $s-1$ and depth at most $d-1$, so they can be computed by formulas of size $(s-1)2^{d-1}$ each. Putting these two formulas together we obtain a formula for C of size $2(s-1)2^{d-1} + 1 < s2^d$. □

A slightly more surprising fact is that is a family of boolean formulas has polynomial size, we can assume without loss of generality that it also has logarithmic depth:

Theorem 6. *If f has a formula of size s, then it has a formula of size $O(s^2)$ and depth $O(\log s)$.*

Proof. When we count the gates in this analysis we will disregard the $2n$ input gates. If f has size s, then there must be a wire in the circuit that splits the other gates into sets of size at most $s/2$ each. Suppose this wire goes out of gate G. Let g be the formula computed by G and f_0, f_1 be the formulas obtained when G is replaced by the constants 0 and 1, respectively (and the formula is simplified). Then we can write the expression

$$f(x) = (f_0(x) \wedge \overline{g(x)}) \vee (f_1(x) \wedge g(x))$$

All of the formulas f_0, f_1, and g have size $s/2$, so we can recursively apply the same argument to them to obtain a formula of depth $O(\log s)$ for f. The size of the new formula obeys the recursive relation $\text{size}(s) = 4\text{size}(s/2) + 3$, which solves to $\text{size}(s) = O(s^2)$. □