6.841/18.405J: Advanced Complexity Theory March 6, 2002

Lecture 9

Lecturer: Madhu Sudan *Scribe: George Savvides*

Topics for today:

1. Proof of Lemma 1 from last lecture
2. Qualitative overview of the proof that $PARITY \notin \mathcal{AC}_0$
3. "Promise" problems, complexity of Unique SAT

1 Proving Lemma 1

Recall Lemma 1 from last lecture:

Lemma 1 *If $\mathbb{C}$ is a circuit of size s and depth d and computes function f then there exists a polynomial $p : \mathbb{Z}_3^n \longrightarrow \mathbb{Z}_3$ of degree $D \leq (\log s)^{o(d)}$ and a set $S \subseteq \{0,1\}^n$ with cardinality $|S| \geq \frac{3}{4}2^n$ such that $\forall x \in S,\ \ p(x) = f(x)$*

1.1 Proof sketch

- WLOG we will assume that the circuit $\mathbb{C}$ consists of NOT and OR gates only (i.e we don't allow AND gates). This restriction can only increase in the size and the depth of the circuit by a constant factor since, using DeMorgan's law, we can replace AND gates with an equivalent configuration that uses only OR and NOT gates (the number of OR and NOT gates will be proportional to the fanin of the AND gate which is a constant).
- We then replace each gate by a polynomial as follows:

  NOT gates We will use the polynomial $p = 1 - x$. This polynomial does the right thing when $x \in \{0,1\}$. We won't bother with what happens when $x = -1$ (recall that we are working in $\mathbb{Z}_3$)

  OR gates Likewise, we would like to replace each OR gate with a *low degree* polynomial. In order to do this, we will replace each OR gate with a randomly chosen polynomial of degree $\log s$ (the polynomial will be chosen *independently* for each gate).

- We will then show that for a fixed input to the circuit, any of these polynomials (replacing the gates) will correctly compute the gate's output with probability at least $1 - \frac{1}{4s}$ [trivial for NOT gates, more interesting for OR gates].
- For any fixed input, the error probability for each (replaced) gate is at most $\frac{1}{4s}$. Thus, according to the union bound, the probability that there exists at least one polynomial in the entire circuit which computed the wrong result is at most $\frac{1}{4s} \cdot s = 1/4$ and so circuit produces the right result with probability at least $1 - 1/4 = 3/4$.
- We conclude that there must exist polynomials of degree $\log s$ replacing the gates of a circuit $\mathbb{C}$ so that the entire circuit correctly computes the output for $3/4$ of all possible inputs.
- Finally, we conclude that the degree of the output function must be $(\log s)^{O(d)}$.

Questions and answers

Q: Do we use the same polynomial to replace all the OR gates?

A: No. Each polynomial is chosen independently

Q: What's our gain?

A: In the original circuit, an OR gate may have up to s inputs. Instead of replacing them with degree-s polynomials (which compute the right result 100% of the time), we will randomly select polynomials of much lower degree that will still end up computing the right result most of the time.

1.2 How to replace gates with low-degree polynomials

1.2.1 The polynomial representing the entire circuit

In order to see what the function computed by the "poly-replaced" circuit looks like, we notice that throughout the circuit we are replacing OR gates (with a fanin of size k) with polynomials of degree $\log s$ (in $x_1, x_2, \ldots, x_k$). At the lowest level of the circuits, where all the the inputs to the gates (polynomials) are constants in $\{0, 1\}$, we have polynomials of degree $\log s$. At the second level, where the inputs to the polynomials are themselves polynomials of degree $\log s$, we will have polynomials of degree $\log^2 s$. In general, if we have a polynomial of degree d_1 and all its inputs are polynomials of degree d_2, the output will be a polynomial of degree $d_1 d_2$ (can be proved using induction). So continuing all the way to depth d, we see that the resulting polynomial (which computes the output of the circuit) has degree $\leq (\log s)^d$.

This polynomial will not be computing the correct value for all inputs. However, for a fixed input and an *independent*, random choice of the polynomials to replace the OR gates, we will show that the probability that a (replaced) gate computes the wrong result is at most $1/4s$. By the union bound, we get the right values throughout the circuit with probability $\geq 3/4$. Notice that this requires that the choice of polynomials be made *independently* of the choice of inputs (i.e. the polynomials were not chosen to suit the input, or the other way around).

Since for a particular input we have a $3/4$ probability of getting the right result when the replacing polynomials are chosen randomly, we conclude that there must exist a particular choice of polynomials (not necessarily the same for all gates) which produces the correct result for at least $3/4$ of all possible inputs to the circuit.

1.3 Finding suitable $\log s$-degree polynomials

In order to get some intuition about what those $\log s$-degree polynomials might look like, it helps to recall that $\mathrm{OR}(y_1, y_2, \ldots, y_k)$ is 1 if at least one of these variables is 1 and 0 otherwise. The polynomial $1 - \prod_{i=1}^{k}(1 - y_i)$ always produces the right result. Unfortunately, it has degree k. If we allow for a non-zero error probability, we can bring the degree down to $\log s$.

We will see how we can construct such polynomials in steps: We will first start with a degree-1 polynomial, then go to degree-two polynomials, and finally degree-s polynomials which compute the right result with probability $1 - \frac{1}{4s}$.

Degree 1 In order to construct a degree-1 polynomial which approximates $\mathrm{OR}(y_1, y_2, \ldots, y_n)$, we simply pick $a_1, a_2, \ldots, a_n \in_R \{1, 0, -1\}$ ($\in_R$ $\equiv$ "randomly selected from"). We then construct the polynomial

$$p_{\bar{a}}(\bar{y}) \quad = \quad \sum_{i=1}^{k} a_i y_i$$

which is of degree 1 and whose output is *always* 0 when all the y's are 0 (as it should be). But what happens when its input is non-zero? It turns out that we can use the Schwarz lemma to show that

$$\Pr_{\bar{a}}[p_{\bar{a}}(\bar{y}) = 0 \text{ if } \bar{y} \neq 0] \leq \frac{1}{3}$$

To see why the Schwarz lemma applies in this case, consider fixing $\bar{y} \neq 0$ and defining $Q(\bar{z}) = \sum_{i=1}^{k} z_i y_i$. Then the probability that $Q(\bar{z}) = 0$ if $Q \not\equiv 0$ is at most $\frac{1}{3}$. If we let $\bar{z}$ be the random choice of $\bar{a}$ we used above, this probability remains the same.

So, on a non-zero input, our polynomial is non-zero with probability $\geq 2/3$. But then it could be either 1 or -1!

Degree 2 We can fix this -1 issue by simply squaring everything. Then:

$$p^2_{\bar{a}}(\bar{y}) \quad = \quad [p_{\bar{a}}(\bar{y})]^2 = \begin{cases} 0 & \text{with prob. 1 if } \bar{y} = 0 \\ 1 & \text{with prob} \geq 2/3 \text{ if } \bar{y} \neq 0 \end{cases}$$

Therefore, $\forall \bar{y}$, $p^2_{\bar{a}}(\bar{y})$ produces the right result with probability $\geq 2/3$.

Higher degree polynomials In order to boost this result, we notice that our degree-2 polynomial always produces the right result when $\bar{y} = 0$, but when $\bar{y} = 1$, it will not necessarily produce a 1. This motivates the following approach: pick $\bar{a}_1, \bar{a}_2, \ldots, \bar{a}_\ell$ at random and come up with a polynomial that computes (with 100% accuracy)

$$\text{OR}\left(p^2_{\bar{a}_1}, p^2_{\bar{a}_2}, \ldots, p^2_{\bar{a}_\ell},\right)$$

A polynomial that computes the OR function of ℓ variables, each of which is a polynomial of degree 2, will have degree 2ℓ. As for the correctness of the output, the probability that all ℓ polynomials inside the OR function fail to produce the correct result is $\leq (\frac{1}{3})^\ell$, so the probability that its result is correct is $\geq 1 - (\frac{1}{3})^\ell$. If we pick $\ell = \log s$ we get the result we wanted: a polynomial of degree $\log s$ computing the OR function with accuracy $1 - \frac{1}{4s}$.

We then repeat the process for all the gates independently.

2 Qualitative overview of the lower bound proof for PARITY

Here's a summary of the proof of the main theorem we set out to prove last time (PARITY $\notin \mathcal{AC}_0$).

1. If PARITY has a small depth, small size circuit $\mathbb{C}$ computing it, then, since $\mathbb{C}$ can be approximated by a polynomial, PARITY will have a low-degree polynomial computing it on *most* inputs.
2. By changing notations[1], we have seen that $\prod_{i=1}^{n} x_i$ has a "small degree" (say D) polynomial computing it correctly on most inputs in $\{-1, +1\}^n$.
3. This allows us to say that every boolean function has a low degree[2] polynomial computing it on most inputs. Therefore, every boolean function is in the linear span of a small number[3] of monomial functions.
4. This leads to a contradiction: the boolean functions (and in particular the δ_x functions given by $\delta_x(y) = 1$ iff $x = y$) require large bases on large domains. You cannot have *every* boolean function in the span of a small set of monomials. (see Lecture 08 for details).

[1] Notice that we can change the representation from $\{0, 1\}$ to $\{+1, -1\}$ "for free" without changing the degree of the polynomials involved.

[2] $n/2 + D$ to be precise

[3] $\sum_{i=0}^{n/2+D} \binom{n}{i}$ to be precise

Alternative Proof: There's an alternative proof in Spielman's notes from 2000 (lecture 13?)
`http://www-math.mit.edu/ spielman/AdvComplexity/2000/`

3 Complexity of Unique SAT

3.1 Motivation

The complexity of Unique SAT was initially studied in the context of cryptography. In particular, Diffie and Hellman had proposed a list of candidate one-way functions and processes which contained, among others, the following examples:

1. Take a piece of code written in a high-level language and compile it. The process is demonstrably easy in one direction, but conjectured to be"hard" in the other
2. Take a formula ϕ and one of its satisfying assignments $\mathbf{a}$, and delete $\mathbf{a}$. That is: $(\phi, \mathbf{a}) \longrightarrow \phi$
   While the forward direction is easy, the reverse is also conjectured to be hard (clarification: it is not necessary to retrieve the same $\mathbf{a}$: any satisfying assignment will do).
3. Take two prime numbers p, q and multiply them to get $p \cdot q$. That is: $p,\ q \rightarrow p \cdot q$
   Again, one direction (multiplication) is easy, while the other (factoring) seems harder.

But how does the difficulty of factoring relate to the difficulty of finding a satisfying assignment $\mathbf{a}$ of the previous example? Several similar questions were being asked at the time:

1. How did "changing the problem" affect its hardness?
2. What if $(\phi, \mathbf{a})$ were chosen at random?
3. What if ϕ were an instance with a known satisfying assignment $\mathbf{a}$?

These were clearly major gaps in our understanding of hardness. However, what bothered the cryptographers of the time most was another important question: going from $(\phi, \mathbf{a})$ to ϕ clearly results in *loss of information*, while going from p and q to pq doesn't. In other words,

- $(\phi, \mathbf{a}) \longrightarrow \phi$ is an information-hiding map
- $p,\ q \rightarrow p \cdot q$ is an information-preserving map (and actually a one-to-one function)

Question: Could hardness be a property of information-hiding maps only?

The hardness of unique-SAT provided a good starting point towards answering this question. Obviously, if ϕ is known to have only one satisfying assignment, there is no information loss when going from $(\phi, \mathbf{a})$ to ϕ. However, does this make USAT any easier than SAT?

3.2 Formalizing the problem

3.2.1 Promise Problems

A lot of questions in Complexity are phrased in the following way: "Can you solve the problem on instances I care about?"

That's where promise problems comes into play. A promise problem Π consists of two sets, Π_{YES} and Π_{NO}, both of which are subsets of $\{0,1\}^*$ and whose intersection is the empty set. That is:

- $\Pi = (\Pi_{\text{YES}}, \Pi_{\text{NO}})$
- $\Pi_{\text{YES}}, \Pi_{\text{NO}} \subseteq \{0,1\}^*$
- $\Pi_{\text{YES}} \cap \Pi_{\text{NO}} = \emptyset$

Given a problem in Π, our goal is to come up with an algorithm A which satisfies the following two conditions:

1. **Completeness** $x \in \Pi_{\text{YES}} \implies A(x)$ accepts
2. **Soundness** $x \in \Pi_{\text{NO}} \implies A(x)$ rejects

If algorithm A satisfies these two conditions, then we say that it solves our problem. Note that other variants of this definition are also available. In particular, the definition can be naturally extended to the probabilistic world.

Strings we don't care about: Notice that if $\Pi_{\text{YES}} \cup \Pi_{\text{NO}} = \{0,1\}^*$ then solving the promise problem Π is the same as deciding the language Π_{YES}. Loosely, we say that in this case the promise problem is just a language. The novelty of of promise problems is that it allows us to to specify strings we don't care about, namely the strings in $\{0,1\}^* - \Pi_{\text{YES}} - \Pi_{\text{NO}}$. Our algorithms can feel free to do whatever they want when given an input from that set. (However, they do need to halt, a condition that can easily be enforced by using a counter.)

There is a broad collection of problems that fall into the category of promise problems, including approximation algorithms and randomized algorithms.

3.3 USAT as a promise problem

Consider the following definition of USAT as a promise problem:

- $\text{USAT} = (\text{USAT}_{\text{YES}}, \text{USAT}_{\text{NO}})$
- $\text{USAT}_{\text{YES}} = \{\phi \mid \phi \text{ has exactly one satisfying assignment}\}$
- $\text{USAT}_{\text{NO}} = \{\phi \mid \phi \text{ is not satisfiable}\}$

3.4 Hardness of USAT

Moving back to the cryptographic motivation, suppose we consider the map $(\phi, \mathbf{a}) \longrightarrow \phi$, but now restrict ϕ to be a member of USAT_{YES}. Since $\mathbf{a}$ is unique, this map is information preserving. However, now it is not clear if this map is hard to invert. If we could show that it is, then we would answer, negatively, the question we raised earlier about complexity: I.e., "is intractability a consequence of information-hiding?" To be more formal, we will explore the question of whether $\text{USAT} \in \mathcal{P}$.

It turns out that it isn't, as a direct consequence of the following theorem due to Valiant and Vaziram:

Theorem 2 *SAT reduces to USAT probabilistically.*

If USAT $\in \mathcal{P}$ then there exists an algorithm running in polynomial time which solves this particular problem with probability $\frac{1}{\text{poly}(n)}$. More precisely, if USAT $\in \mathcal{P}$ then $\mathcal{NP} = \mathcal{RP}$.

The probabilistic reduction takes a formula ϕ and creates another formula ψ with the following properties:

1. $\phi \in \text{SAT} \implies \psi \in \text{USAT}_{\text{YES}}$ with probability $\frac{1}{\text{poly}(n)}$
2. $\phi \notin \text{SAT} \implies \psi \in \text{USAT}_{\text{NO}}$ with probability 1

Notice that we have no idea how to boost the probability from $\frac{1}{\text{poly}(n)}$ to something better in the first case.

Related question: Suppose that the reduction is given and that USAT $\in \mathcal{P}$. Can we use it to compute a satisfying assignment to the SAT problem?

Answer: Yes. Use the self-reducibility of USAT.

3.5 Main ideas for reduction from SAT to USAT

Say we have the universe $\{0,1\}^n$ and a formula ϕ. Let $S = \{a \mid \phi(a) = 1\}$. Suppose further that we know $M = |S|$. We will do the following:

We pick a random (hash) function $h : \{0,1\}^n \longrightarrow \{0, \ldots, M-1\}$. Then, with probability at least some constant c, there exists an $x \in S$ such that $h(x) = 0$. Define $\psi(x)$ as follows:

$$\psi(x) = \phi(x) \wedge [h(x) = 0]$$

Notice that this is not really a SAT formula, but the completeness of SAT for NP indicates we can convert the above expression into a SAT formula, specifically by solving the following exercise.

Exercise: Given a circuit $C : \{0,1\}^n \to \{0,1\}$ show that it is possible to construct, in time poly($|C|$), a 3CNF formula $\phi(\mathbf{x}, \mathbf{y})$, where $\mathbf{x}$ is a vector of n variables and $\mathbf{y}$ is a vector of at most $|C|$ variables such that for every $\mathbf{x} \in \{0,1\}^n$

$$C(\mathbf{x}) = 1 \Rightarrow \exists! \mathbf{y} \text{ s.t. } \phi(\mathbf{x}, \mathbf{y}) \text{ and } C(\mathbf{x}) = 0 \Rightarrow \forall \mathbf{y}, \phi(\mathbf{x}, \mathbf{y}) = 0.$$

A simple probability calculation (omitted from today's lecture) shows that if ϕ is satisfiable, then with some constant probability we also have $h(x) = 0$ for a *unique* $x \in S$. If h could be computed efficiently, then this reduction would also take only polynomial time and we would be done.

Problems with this reduction:

1. If h is random: we can not be able to specify it succinctly or compute it efficiently.
2. We don't really know M.

Solution:

1. We will not require h to be completely random. Instead, we will require only *pairwise independence*(defined below).
2. We will guess M. It turns out that if we are within a factor of 2, the probability calculations will still work out (so we only need to make logarithmically many guesses).

3.5.1 Pairwise independence

Definition 3 *$\mathcal{H} \subseteq \{f : \{0,1\}^n \longrightarrow \{0,1\}^m\}$ is a pairwise independent family of hash functions if for all $a \neq b \in \{0,1\}^n$ and $c, d \in \{0,1\}^m$ the following holds:*

$$\Pr_{h \in \mathcal{H}}[h(a) = c \wedge h(b) = d] = \left(\frac{1}{2^m}\right)^2$$

In the definition above, notice two things:

1. We only required that $a \neq b$. We do want the condition to hold for all pairs (c, d) including when $c = d$.
2. 2^m is the size of the range of $\mathcal{H}$.

Definition 4 *$\mathcal{H}$ is* nice *if $h \in \mathcal{H}$ can be (1) efficiently sampled and (2) efficiently computed*

Example The following family of hash functions is both pairwise independent and nice:

- $\mathcal{H} = \{h_{A,b}\}$ where
- A is an $m \times n$, $\{0,1\}$ matrix
- b is an m-dimensional $\{0,1\}$ vector
- $h_{A,b}(x) = Ax + b$ (all operations done mod 2)

The proof is left as an exercise.

In the next lecture we will complete the reduction from SAT to USAT using the two ideas above (pairwise independent hash h, and guessing M approximately).