6.841 Advanced Complexity Theory May 11, 2009

Lecture 25

Lecturer: Madhu Sudan *Scribe: Rishi Gupta*

Write your feedback for the course at https://sixweb.mit.edu/student/evaluate/6.841-s2009. It will help future students decide whether they should take the class.

Motivation

A few examples of where randomization (and derandomization) results are used:

- Algorithmic, for instance RP. This might be less exciting than we think though if BPP = P.
- Distributed Computing. A classic problem is: given n computers that are pairwise connected, each with a single bit. If all the bits are 0 they should agree that they all have 0, if they all have 1 they should agree they all have 1, otherwise they can do whatever. The communication links are arbitrarily speeds; some computers might even be asleep for an hour. We can't solve this problem deterministically.
- Cryptography. Given that the inputs, outputs, and algorithm aren't secret, randomness is essential to have any secrets at all.
- Game Theory. Equilibrium exists if and only if randomness exists. For instance, optimal rock-paper-scissors playing is dependent on access to random bits.

Extracting Randomness

Nature provides unpredictability. Physics at a small enough level is assumed to be random. However, as a practical matter, it's hard to use this randomness to come up with unbiased, independent coins. Electrons are expensive to deal with; they don't do things with exactly 50-50 probability, and it's hard to do multiple independent trials on the same chip. Though intel does sell a costly randomness generating chip.

So maybe we can start with a large number (say n^2) weakly random (biased, dependent) bits, and use those to generate n truly random bits.

Von Neumann: If we start with a stream of random, independent bits, all biased with the same unknown probability p (they are 0 with probability p, 1 with probability $1-p$), we can convert them into a stream of truly random bits by looking at pairs of bits, discarding 00 and 11, and turning 01 into 0 and 10 into 1.

This idea of purifying random bits out of a source of randomness is called *extraction*.

Blum: Gave an extractor for Markovian sources, where the state jumps around, and each state is biased differently.

Vazzirani ('83): Ph.d thesis on extracting randomness.

Nissan-Zuckermann ('93): Defined randomness and randomness extraction as given below:

We say a distribution has *entropy* k if no sequence is produced with probability more than $1/2^k$. Goal: take sequence of n bits with entropy k, and extract some $m \leq k$ truly random bits from it.

Turns out to be impossible deterministically. BUT, you can do it if you start with a small random seed. Lots of work is done, and then Trevisan ('99) comes up with an efficient extractor, where the seed is of length $\log n$, and m is maybe $k/10$.

Pseudorandom Generators

It's not hard to imagine that BPP = P. How would one prove such a thing? If we could generate random-looking distributions over n bits using a small number of random bits $l(n)$ (called a *seed*), we could deterministically use the BPP algorithm over every string in the distribution, and explicitly calculate a probability. In particular,

A pseudorandom generator $G : \{0,1\}^l \to \{0,1\}^n$ *looks random* to algorithm A if $Pr_{x \leftarrow U_n}(A(x) = 1) \approx_\epsilon Pr_{s \leftarrow U_l}(A(G(s)) = 1)$, where U is the uniform distribution and $\epsilon > 0$.

What we're most interested in are PRGs (pseudorandom generators) G that fool *all* polysize circuits A. We're interested in polysize instead of polytime for technical reasons.

Yao: If there exists a PRG G that runs in polytime, stretches $l(n)$ bits to n bits, and fools all polysize algorithms A, then BPP $\in$ DTIME($2^{l(n^k)}$). In particular, if l is $O(\log n)$, then BPP = P. The proof is straightforward; you just loop over all the possible seeds.

Blum and Micali's PRG

We use $RSA : \{0,1\}^l \to \{0,1\}^l$ to construct a PRG. RSA is a one-way function, meaning that $RSA(x)$ is easy to compute but $RSA^{-1}(y)$ is hard.

Define $G_1 : \{0,1\}^l \to \{0,1\}^{l+1}$ as follows: $G_1(x)$ prepends the most significant bit of x to $RSA(x)$. Blum and Micali proved that this is in fact a pseudorandom generator, but it's a very hard and fragile proof. For instance, prepending the least significant bit doesn't work. We then define $G_k : \{0,1\}^{l+k}$ by simply iterating the procedure.

Claim: For every $k \in$ poly(l) and polynomial q, $G_k(x)$ looks random with $\epsilon = 1/q$ to all poly sized circuits.

Proof: We reduce from G_1, using general techniques called *reconstruction* and *hybridization.*

Define D_i for $0 \le i \le l+k$ to be the distribution over strings of length $l+k$ where the first i bits are from U_i and the last $l+k-i$ bits are the last $l+k-i$ bits of G_{l+k}. Say for contradiction we could distinguish $U_{l+k} = D_0$ and $G_{l+k} = D_{l+k}$ in polysize with probability ϵ. Then for any i we can distinguish D_i and D_{i+1} in polysize with expected probability ϵ/k.

D_i looks like i random bits + $G_{k-i}(U_l)$. D_{i+1} looks like i random bits + 1 random bit + $G_{k-i-1}(U_l)$. We can generate D_i (resp D_{i+1}) by applying G_{k-i-1} to the last l bits of $G_1(x)$ (resp a random bit + $RSA(x)$), and then prepending the first bit of $G_1(x)$ (resp the random bit) and U_l. Since we can't distinguish $G_1(x)$ from a random bit + $RSA(x)$, we have a contradiction. ◁

This work has gone in several directions since then.

- Håstad, Impagliazzo, Levin, Luby: One way functions are necessary and sufficient for the existence of PRGs. By one-way function we mean something that is computable in a fixed polynomial time, but fools all polytime algorithms.

- Since the RSA stuff is so touchy — it depends on the specifics of RSA, and the fact that we took the most significant bit — there was hope for a more general method of gaining an extra bit.

  Goldreich-Levin: If f is a general one-way permutation, there is no way to come up with a generic bit b such that $(b(x), f(x))$ looks random.

  But we can come up with such a bit for a slight variant. Given $\hat{f}(x,r) = (f(x), r)$, $b(x,r) = \langle x, r\rangle = \otimes x_i r_i$ is such a desired bit. Note that $\hat{f}$ is quite similar to f.

- If we could find a G that runs in some big poly time, but fools all small poly time algorithms, it would still show BPP = P.

  Say $f : \{0,1\}^m \to \{0,1\}$ is hard for circuits of size $2^{\epsilon m}$. We set the i^{th} bit of our output string to $f(s_i)$, where $\{s_i\}$ is a set of not-necessarily disjoint subsets of size m in a seed s.

  Nisan-Wigderson: One way to make this idea work (to still have each s_i contain enough independent randomness) is to insist $|s_i \cap s_j| < 10m^2/l$. (Note m^2/l is the expected intersection between any s_i and s_j.)

  Using hybridization as above, we reduce to the case when the seed is fixed except over s_n. All the other seeds intersect this in at most $10m^2/l$ bits, and so $f(s_i), i < n$ can be computed in a circuit of size $2^{10m^2/l}$, which is tiny if you choose m and l right [look at the notes for more details].

Essentially, the problem reduces to finding a function f computable in DTIME(2^{10000n}) but that isn't in SIZE($2^{.0001n}$), in other words, a function for which non-uniformity doesn't help much. Though everything can be computed in SIZE(2^n), there are lots of functions in SIZE($2^{.0001n}$); it's mainly a matter of finding one in the intersection of the two sets.