| | |
|---|---|
| **6.841 Advanced Complexity Theory** | April 27, 2005 |
| Lecture 22 | |
| *Lecturer: Madhu Sudan* | *Scribe: Paul Valiant* |

# 1 Outline

This lecture consists of two parts. First we discuss some of the feedback given on the previous four lectures. Then we develop the notion of randomness, from its roots in physics, information theory, and cryptography, to its applications to complexity theory. We aim towards some version of "BPP=P", namely that randomness "is" or "should be" free.

# 2 Lecture Comments

The complete comments are posted online at

`http://theory.csail.mit.edu/~madhu/ST05/comments18-21.txt`

We note here a few things of general interest that were brought up.

## 2.1 Self-correction

Recall that in the PCP framework, when we wanted to evaluate a function $f(a)$ given a table of function evaluations, we did not simply query the value indexed by $f$, but rather constructed a random function $f_1$, queried the table at the entries indexed by $f + f_1$ and $f_1$, and took their difference.

Similarly, when we wished to evaluate $(f \cdot g)(a)$ we did not query the table at $fg$ but rather constructed a random function $h$, and took the difference between the values indexed by $fg + h$ and $h$.

The reason for this is the following. We can show statements about PCPs along the lines of "if a string is close to an honestly generated PCP then the theorem must be true". Further, we ask the verifier to probabilistically estimate closeness of a given string to an "honest" PCP by conducting random tests and seeing if any fail; if they all succeed, then the verifier can be fairly sure the string lies close to an "honest" PCP and the theorem is true.

Thus it had better not be the case that by changing a tiny fraction of the entries of a PCP one could significantly change the results of the verifier's tests.

Consider, for example, what would happen above if we queried $fg$ directly. Note that since $f$ and $g$ take values in $\{0, 1\}$, their product will have expected value $\frac{1}{4}$ when $f$ and $g$ are drawn randomly. The fraction of $\{0, 1\}$-functions that have expected value close to $\frac{1}{4}$ is tiny compared with the total number of

$\{0,1\}$-functions. Thus it had better not be the case that values from this tiny set are queried with high probability. Specifically, we cannot query $fg$ directly.

This suggests the trick of making the distribution of queries uniform by adding an independent random function $h$. In this way we can ensure that the PCP is sound.

This is an instance of what is more generally referred to as a "self-correcting" probe, i.e. a probe that corrects its own mistakes. This is an area with many applications outside the scope of this class.

## 2.2 Approximation Ratios

Recall that we showed the PCP theorem to be equivalent to the nonexistence of an $\alpha$-approximation to the max-$k$-SAT-$\Sigma$ problem unless P = NP. The question was raised whether we know anything more specific about the approximation ratio of the max-$k$-SAT-$\Sigma$ problem.

We note that a clause of such a problem depends on at most $k$ bits, so will be satisfied by a random assignment with probability at least $2^{-k}$. This implies we have a $2^k$ approximation algorithm for max-$k$-SAT-$\{0,1\}$.

This bound looks awful. However, a recent result by Samorodnitsky and Trevisan shows that one cannot do better than $2^{k-\sqrt{k}}$, unless P = NP.

To put this in context, we note that the existence proof of 3-query PCPs implies that one cannot do better than 2-approximate max-3-SAT-$\{0,1\}$ unless P = NP, while we could do a $\frac{1}{8}$ approximation by just guessing. Thus our three queried bits have (in a sense) an efficiency of $\frac{1}{3}$, since these three bits give us 1 factor of 2 soundness. The surprising fact is that with each additional bit of the proof, we can almost double the completeness/soundness ratio, since $2^{k-\sqrt{k}}$ is so close to $2^k$.

## 2.3 Dinur's Result

There was some confusion about various aspects of Dinur's result. Everyone agreed the proof went too fast. More details will be covered in an optional session this Friday (April 29).

Also, there was some discussion of *which* of her techniques/results were new/interesting. Madhu cited as a key feature of her proof a new connection between expanders and randomness amplification.

## 2.4 Computationally Sound Proofs

In class so far we have seen a wide variety of notions of what a "proof" is. The standard notion allows us to prove anything in NP. Interactive proofs allow us to prove anything in the (much bigger?) class PSPACE, but at the cost of relaxing the notion of soundness to be only statistical; i.e., unless the prover gets very lucky, he will not be able to prove a false statement.

We may relax the soundness criteria even further to be *computational*. Namely, a computationally sound proof (CS-proof) is one where the prover cannot prove a false statement unless he either gets really lucky, or expends an inordinate amount of time. We note that the relaxation from "statistical" to "computational" will appear later in this lecture with regards to a definition of randomness, and is an important technique.

In the proof paradigms we have considered so far, we have generally disregarded the computational difficulties involved with the prover's side of the picture. Here they play a central role. In Arthur-Merlin protocols, by contrast, Merlin, the prover is portrayed as "magical". The general idea is that a computationally bounded verifier might trust a prover more if he knows the prover is similarly computationally bounded and cannot perform "magic" to fool him.

Using this notion, Micali demonstrated how to construct polynomial sized CS-proofs of any statement in EXPTIME. Unfortunately, his results, unlike those of the PCP theorem, rely on some extremely strong cryptographic assumptions, specifically something known as the *random oracle model/methodology.* The details of this are beyond the scope of this lecture.

# 3 Randomness

We move on to our main topic: randomness. We have discussed randomized algorithms, constructions proofs, etc. but have not discussed what randomness *means*, nor how to *find* it.

One place to start is with physics. We believe randomness exists because physicists say that there is randomness. Physicists cite a device called a "zener diode", which they tell us behaves randomly.

(We note however, that just because physicists say something exists does not mean we should interpret it as "free" in our model of computation. Physicists say that quantum computing exists, and that is a far from easy assumption for complexity theorists to assimilate.)

At this point, we must consider what physicists mean by "random". In the case of the zener diode, they mean *not completely predictable.* If we measure the voltage across a zener diode at regular intervals, we get a sequence that is not completely predictable. However, if we measure at short enough intervals, we notice that the voltage changes only so fast, and there is significant serial correlation. In other words, physicists do not guarantee *uniform* randomness, of the kind we need for BPP and other complexity theory tasks.

We might call randomness without such a guarantee "weak" randomness. There is a developing computer science field of *randomness extractors*, where the goal is to extract "pure" randomness from weak randomness.

We back up here and ask again what we mean by randomness. Consider the following sequence:

$$010110111000011011110010001.$$

Is it random? What about the following sequence:

$$000000000000000000000000000?$$

We might consider a hypothetical "Turing test" for randomness, namely we put a person in a room who tries to generate random numbers, and we are asked to distinguish between his sequences and those produced by a coin. Which of the above sequences would pass the test? Some of the class said that the all-zeros sequence would never be seen from a random coin; someone else said that no person would be stupid enough to try to pass off an all-zeros sequence as random.

In light of this disagreement, we consult the experts.

- Shannon's answer is *no*: no single string can be random; randomness is a property of distributions.
- Kolmogorov's answer is *no*: no finite string can be random; he defines a string as random if no finite length (deterministic) program could produce it.
- Adleman's answer is *maybe*: recall his proof that $\mathrm{BPP} \subseteq \mathrm{P}|_{\mathrm{poly}}$; the technique was to find a *single* string of length $n^2$ that could stand in for all the randomness in the uniform distribution on $\{0,1\}^n$, at least in the context of the fixed BPP algorithm $A$ under consideration.

The interesting part of Adleman's definition is that the notion of randomness is conditional on the use to which it will be put. Namely, whether a string is random or not depends on the algorithm $A$ we wish to "fool".

We clarify the model here before moving on.

## 3.1 Pseudo-Random Generators

A pseudo-random generator (PRG) is a (deterministic) algorithm $G$ that takes a "short" random seed $\sigma$ as input, and produces a longer string as output, where the output meets a criteria for randomness that we will define later.

In formal notation, we have

$$G : \{0,1\}^l \to \{0,1\}^n,$$

where $l < n$. We consider the distribution of $G(\sigma)$, when $\sigma$ is uniformly random, notated as

$$\{G(\sigma)\}_{\sigma \leftarrow U_l},$$

where $U_m$ is defined to be the uniform distribution on $\{0,1\}^m$.

For the sake of completeness, we note that a distribution $D$ is formally defined to be a function

$$D : \{0,1\}^n \to [0,1]$$

satisfying

$$\sum_x D(x) = 1.$$

We consider the distribution $\{G(\sigma)\}_{\sigma \leftarrow U_l}$, and ask how "close" it is to our target $U_n$, the uniform distribution on length $n$ strings.

The standard distance metric on distributions $D_1, D_2$ is the *statistical difference*, defined as follows:

$$|D_1 - D_2| \stackrel{\text{def}}{=} \sum_{x : D_1(x) > D_2(x)} D_1(x) - D_2(x).$$

We note that the statistical difference is symmetric:

$$|D_1 - D_2| = \sum_{x : D_1(x) < D_2(x)} D_2(x) - D_1(x),$$

and we may thus equivalently define it in the explicitly symmetric form

$$|D_1 - D_2| = \frac{1}{2} \sum_x |D_1 - D_2|.$$

We call two distributions $D_1, D_2$ *$\epsilon$-close* if $|D_1 - D_2| \leq \epsilon$. It turns out that the statistical difference is very much related to our task of distinguishing a distribution from random. We have the following lemma.

**Lemma 1** *Distributions $D_1$ and $D_2$ are $\epsilon$-close if and only if for all algorithms $A$,*

$$\Pr_{x \leftarrow D_1}[A(x) = 1] - \Pr_{x \leftarrow D_2}[A(x) = 1] \leq \epsilon.$$

This means that if we could pseudo-randomly generate a distribution very close to uniform, it would pass all random tests. However, we note immediately that this is not possible. Suppose the distribution $D_1 : \{0,1\}^n \rightarrow [0,1]$ is uniform. Thus is has full support on $\{0,1\}^n$. Clearly a pseudo-randomly generated distribution can have support on at most half this set, since $G$ is seeded with a string of length $l$ at most $n-1$, and $G$ is deterministic. Looking at those elements outside the support of $G(U_l)$, we bound the statistical difference as

$$|U_n - G(U_l)| \geq \sum_{x \notin G(U_l)} (U_n(x) - 0) \geq 2^{n-1}2^{-n} = \frac{1}{2}.$$

This means that a pseudo-random generator can never hope to fool every $A$, and we must cut back our expectations if we wish to succeed.

We we introduce the notion of *computational indistinguishability*. The idea is that while we can never fool all adversaries, we may realistically only concern ourselves with computationally bounded ones.

**Definition 2** *We say that distribution $D_1$ is* $(\epsilon, s)$-computationally indistinguishable *from $D_2$ if for all algorithms $A$ that are computed with a circuit of size $s$,*

$$\Pr_{x \leftarrow D_1}[A(x) = 1] - \Pr_{x \leftarrow D_2}[A(x) = 1] \leq \epsilon.$$

As an exercise: prove that that there exists a distribution $D_1$ with support polynomial in the size of $s$ such that $D_1$ is $(\frac{1}{10}, s)$ computationally indistinguishable from random. The idea is very similar to Adleman's proof that $\mathrm{BPP} \subseteq \mathrm{P_{poly}}$.

We may thus define a pseudo-random generator as a function that produces random-looking distributions:

**Definition 3** *(Blum and Micali) A function $G : \{0,1\}^l \rightarrow \{0,1\}^n$ is an $\{\epsilon, s\}$-PRG if $\{G(\sigma)\}_{\sigma \leftarrow U_l}$ is $(\epsilon, s)$ computationally indistinguishable from $U_n$.*

Yao noted that if we have a function $G$ that is computable in time $t(l)$ and is a $(\frac{1}{10}, n^{\omega(1)})$-PRG then

$$\mathrm{BPP} \subseteq \mathrm{TIME}(2^{\mathrm{l}} \cdot \mathrm{t(l)} \cdot \mathrm{poly(n)}),$$

since no program in BPP could distinguish $\{G(\sigma)\}_\sigma$ from random.

It remains to construct such generators, and examine their running time $t(l)$.

We introduce a few such constructions from the literature.

Blum and Micali introduced the first PRG $G : \{0,1\}^l \rightarrow \{0,1\}^{l+1}$, which has running time $t = poly(n)$, and is a $(\frac{1}{l^{\omega(1)}}, l^{\omega(1)})$-PRG.

While this generator only expands the input by 1 bit, they further showed how PRGs could be composed to expand the input to any length. Specifically, they showed how to transform an $(\epsilon, s)$-PRG $G : \{0,1\}^l \to \{0,1\}^{l+1}$ into an $(n\epsilon, s)$-PRG $\hat{G} : \{0,1\}^l \to \{0,1\}^n$, by iteratively applying $G$ to the last $l$ digits of its output.

Their construction, however, relies on several assumptions, including that $\mathrm{P} \neq \mathrm{NP}$; it would thus be awkward for us to use such a generator to prove that $\mathrm{BPP} = \mathrm{P}$.

Recall Yao's time bound from above of $2^l \cdot t(l) \cdot poly(n)$. Since there is already a factor of $2^l$, we do not need a PRG with $t(l) = poly(l)$, but could in fact allow exponential time. Thus we do not need the efficient PRGs that Blum and Micali provide.

To make Yao's bound polynomial in $n$, we set $l = O(\log n)$, and let $t = exp(l)$. We also note that it suffices to let $G : \{0,1\}^l \to \{0,1\}^n$ be a $(\frac{1}{10}, n)$-PRG by a simple padding argument: suppose we want to fool a BPP machine $A$ that takes time $p(n)$ for some polynomial $p$. Thus we could express $A$ on inputs of size $n$ as a circuit of size $q(n)$ for some polynomial $q$. Then if we constructed a generator $G : \{0,1\}^l \to \{0,1\}^{q(n)}$, it would be sufficient for it to be a $(\frac{1}{10}, q(n))$-PRG, as claimed above. This argument is from Nisan and Wigderson.

Impagliazzo and Wigderson extended this argument by relating languages that are undecidable over circuits of size $s$ to distributions that are pseudo-random over circuits of size $s$. Their result is the following:

**Claim 4** *If there exists a language $L$, and constants $\epsilon > 0, c$ such that language $L$ on inputs of length $l$ is decidable in time $2^{cl}$, but not decidable by circuits of size $2^{\epsilon l}$ then* $\mathrm{BPP} = \mathrm{P}$*.*

Thus circuit complexity lower bounds could prove that $\mathrm{BPP} = \mathrm{P}$. We note that it is crucial here that the circuits are allowed to be non-uniform, because otherwise $\mathrm{BPP} = \mathrm{P}$ would follow from the time hierarchy theorem.

We will prove this claim in the next lecture.