6.841: Advanced Complexity Theory Fall 2012

Lecture 13 (Madhu Sudan) — October 25, 2012

Prof. Dana Moshkovitz *Scribe: Josh Alman*

1 Overview

Today we will move on to the topic of pseudorandomness. We will begin by introducing pseudorandom generators and why they are important, with emphasis on their relation to the **P** versus **BPP** problem. We will also look at the history of the the theory of pseudorandomness, to gain an insight into the throught process that went into it. Finally, we will introduce the Nisan-Wigderson generator and begin to prove that it is a pseudorandom generator under certain assumptions. We will finish the proof next class.

2 Pseudorandomness

2.1 Defining Pseudorandomness

We have already seen randomness in a variety of contexts in the class. However, it is important to consider exactly what randomness means. For instance, is the following bit sequence random?

$$0101101110000110111110010001$$

What about the following sequence?

$$0000000000000000000000000000$$

It seems like the former sequence is more random than the latter. We cannot easily find any patterns in it, for instance. However, by most definitions of randomness, a single string can never be random.

In [1], Blum and Micali sought to give a useful definition of randomness that could be used in modern Complexity and Cryptography. The main idea they devised is that *randomness is in the eye of the beholder.* They thus define a pasudo-random generator to be a (deterministic) algorithm G which takes in a seed of length s and produces a string longer than s as its output, with the condition that any distinguisher algorithm with some constraints on it cannot distinguish the output from random. More formally:

Definition 1. *For a generator*

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

and a distinguisher

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

we say that D is ϵ-successful *at distinguishing G if*[1]

$$\Pr_{x \leftarrow \{0,1\}^s}[D(G(x)) = 1] - \Pr_{y \leftarrow \{0,1\}^n}[D(y) = 1] > \epsilon.$$

The idea is that D somehow distinguishes strings generated by G from strings actually generated at random. However, notice that for any G, there is a D which simply returns 1 if the input is a string that can be output by G, and returns 0 otherwise. For instance, it could do this by running G on all possible seed strings and seeing if D ever outputs the current input. This D would $(1 - 2^{r-s})$-distinguish G. However, this D might take a very long time to run. This inspires the following definition:

Definition 2. *([1]) A generator G is* ϵ-pseudorandom *for a class C of distinguishers if, for all distinguishers D in C, D is not ϵ-successful at distinguishing G. This means:*

$$\Pr_{x \leftarrow \{0,1\}^s}[D(G(x)) = 1] - \Pr_{y \leftarrow \{0,1\}^n}[D(y) = 1] \leq \epsilon.$$

We choose C to be a class with certain constraints that we are interested in examining. Picking $C = \mathbf{P}$ is often a good choice. However, for this lecture we will consider $C = \mathbf{P/poly}$, which will turn out to be more useful.

Notice that if, for a particular G, D, we have that

$$\Pr_{x \leftarrow \{0,1\}^s}[D(G(x)) = 1] - \Pr_{y \leftarrow \{0,1\}^n}[D(y) = 1] = v < 0,$$

then the distinguisher D' which runs D and then outputs 1 if D would output 0, and 0 if D would output 1, gives

$$\Pr_{x \leftarrow \{0,1\}^s}[D'(G(x)) = 1] - \Pr_{y \leftarrow \{0,1\}^n}[D'(y) = 1] = -v > 0.$$

Thus, it would be equivalent to put absolute values around the differences in probabilities in the two definitions above. But, by convention, we will not, and we will say that D tries to output 1 on valuges generated by G and 0 on values generated at random.

There are two other common definitions of randomness. Shannon defines randomness as a property of distributions. Meanwhile, Kolmogorov says that finite strings cannot be random, but defines the randomness of an infinite string to be proportional to how big a deterministic program would have to be to generate it. Note that the Blum-Micali definition of randomness lies in between the Shannon and the Kolmogorov definitions. It does apply to finite strings, but it says that single finite strings can not be random.

2.2 Pseudorandomness and P versus BPP

The general idea behind pseudorandom functions is to find a G that runs fast, and that fools distinguisher algorithms that run fast. These two 'fast's might not necessarily be the same.

In [3], Yao showed that the existence of some pseudorandom generators would imply that $\mathbf{P} = \mathbf{BPP}$. This would have profound implications, for instance, for our ability to prove statements that we

[1]Throughout these notes, all the probabilities will be over the uniform distribution on the given set.

know how to randomly check. As an example, consider the problem of deciding whether a given n-variable polynomial is identically zero. We have a randomized algorithm for this problem, which is to evaluate the polynomial at many randomly-chosen points and confirm that it is zero at all of those. However, if we had a particular example of a pseudorandom generator, we could use that generator to produce a set of points for that algorithm. This would yield a set of points such that any polynomial is identically zero if and only if it is zero at those points. We do not currently know of such a set.

We state more precisely and prove Yao's theorem:

Theorem 3. *([3]) Suppose we have a $\frac{1}{10}$-pseudorandom generator $G : \{0,1\}^{s(n)} \to \{0,1\}^n$ for **P/poly**, which runs in time $t(n)$. Then,*

$$\boldsymbol{BPP} \subseteq \boldsymbol{DTIME}(2^{s(n^{O(1)})} \cdot t(n^{O(1)}) \cdot n^{O(1)}).$$

Proof. For a given algorithm $A \in \mathbf{BPP}$, we want to simulate A with a deterministic algorithm. Recall that if $L = L(A)$ is the language of A, then

$$x \in L \Leftrightarrow \Pr_y[A(x,y) = 1] > \frac{2}{3},$$

$$x \notin L \Leftrightarrow \Pr_y[A(x,y) = 1] < \frac{1}{3}.$$

The y above are taken over all binary strings of a certain length ℓ that is polynomial in n.

Here is what our deterministic algorithm can do: Run $A(x, G(z))$ for every $z \in \{0,1\}^{s(\ell)}$. Taking the average of the results gives

$$\Pr_{z \in \{0,1\}^{e(\ell)}}[A(x, G(z)) = 1].$$

Since for each input x, we could create the distinguisher $D(y) = A(x,y)$, it follows by assumption that the above probability is within $\frac{1}{10}$ of $\Pr_y[A(x,y) = 1]$. Then, since $\frac{1}{3} + \frac{1}{10} < \frac{1}{2}$ and $\frac{2}{3} - \frac{1}{10} > \frac{1}{2}$, we can just accept if the probability we get is more than a half, and reject if it is less than a half.

Computing each $G(z)$ takes $t(\ell)$ time, running A takes $n^{O(1)}$ time, and we do this for $2^{s(\ell)}$ strings, for a total runtime of $O(2^{s(\ell)} t(s(\ell)) n^{O(1)})$, which is indeed $O(2^{s(n^{O(1)})} \cdot t(n^{O(1)}) \cdot n^{O(1)})$. We conclude by noting that the algorithm we just described is indeed deterministic. ☐

3 Blum-Micali generator

We next give a pseudorandom generator due to Blum and Micali. Note that in the above proof, we want $s(n) = O(\log n)$ so that $2^{s(n^{O(1)})} \cdot t(n^{O(1)}) \cdot n^{O(1)}$ is polynomial in n. However, Blum, Micali, and Yao did not think this would be possible to achieve at the time, under the constraint that we also have $t(n) = poly(n)$.

Blum and Micali sought to make a generator $G : \{0,1\}^{\sqrt{n}} \to \{0,1\}^n$ which runs in polynomial time and fools all polynomial time algorithms. Their idea was to use one way permutations, which can easy generate strings, but cannot be easily recognized.

Definition 4. *A* one way permutation *is a bijection $f : \{0,1\}^k \to \{0,1\}^k$ such that, loosely, f is easy to compute, but f is hard to invert.*

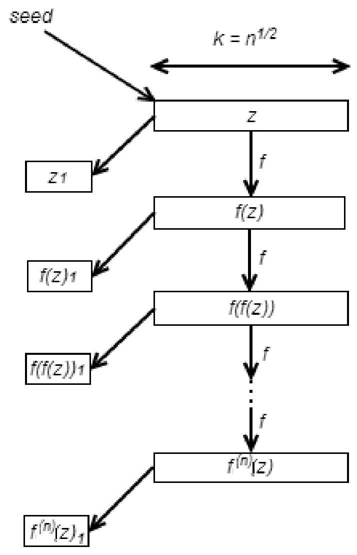

Figure 1: The Blum-Micali generator. The bits on the left column are the output.

The Blum-Micali generator works as follows: given a one way permutation f, we start using a random seed to generate a random bit string z. Then, we repeatedly apply f to the current bit string, and take the first bit to add to the output of our generator. See Figure 1. More formally:

Definition 5. *The* Blum-Micali generator $g : \{0,1\}^k \to \{0,1\}^n$ *is given by*

$$g(y)_i = f^{(i)}(z)_1,$$

where z is a string yielded form the seed y.

Theorem 6. *g is a pseudorandom generator for $\boldsymbol{P}$, using $f = RSA$ and some assumptions.*

Proof. Not proved in class. See [1]. □

Unfortunately, we cannot pick $k = \log n$ in a scheme like the Blum-Micali generator. Indeed, if $f : \{0,1\}^k \to \{0,1\}^k$ is any permutation that can be computed in time $t(k)$, then we know that f^{-1} can be computed in time $2^k t(k)$ by just trying all possible inputs to f and seeing which one yields the string we want to invert. If $k = \log n$, then this is polynomial in n, and so f is not hard to invert.

4 Nisan-Wigderson generator

To rectify the above problem, we seek a function f that is easy to compute, but not too easy to compute. For instance, perhaps we want an f that can be computed in time n^{10} but not in time n^9. By the time hierarchy theorem, such a function exists!

Nisan and Wigderson sought to use such a function to make a pseudoranfom generator. Unfortunately, the function they used is one that we do not know for sure exists by using the time hierarchy theorem or any other known result. In particular, they wanted a generator G for which:

1. G runs slower than the algorithms it fools, but,
2. G must stretch the randomness of the seed vastly. In particular, the want $s(n) = O(\log n)$ so that $t(n)$ is exponentially larger than $s(n)$.

They found such a function under the assumption that there exists an $f : \{0,1\}^k \to \{0,1\}$ such that:

1. f is computable in time 2^{100k}
2. The $k+1$ bit string $(z, f(z))$ looks random to **P/poly**.

To make the second assumption more precise, we make the definition:

Definition 7. *A function $f : \{0,1\}^k \to \{0,1\}$ is* ϵ-easy for size m *there exists a circuit C of size at most m such that*

$$Pr_{x \leftarrow \{0,1\}^k}[f(x) = C(x)] \geq \frac{1}{2} + \epsilon.$$

For instance, if f is $\frac{1}{2}$-easy then a circuit exactly computes it, while if a function is not ϵ-easy for any $\epsilon > 0$ then no circuit can compute it correctly on more than half the inputs.

Assumption 2 for the Nisan-Wigderson generator is then stated formally as follows:

2. f is not ϵ-easy for size $2^{k/100}$, for any $\epsilon = \exp(-k)$.

These assumptions are called the Nisan-Wigderson assumption. Note that any function is computable by a circuit of size 2^k, although $2^{k/100}$ is much smaller than this. Thus, it seems plausible that the assumption is true.

Theorem 8. *([2]) Under the Nisan-Wigderson assumption, there exists a generator $G : \{0,1\}^{O(\log n)} \to \{0,1\}^{O(n)}$ such that G can be computed in poly(n) time and G $\frac{1}{10}$-fools all n^2-size circuits.*

By Theorem 3, this implies that under the Nisan-Wigderson assumption, $\mathbf{P} = \mathbf{BPP}$. We begin the proof now, and will finish it next class.

Proof. We define our generator G. It will be defined by some $k, s \in O(\log n)$ with $k < s$, a function $f : \{0,1\}^k \to \{0,1\}$ that satisfies the Nisan-Wigderson assumption, and n subsets $T_1, T_2, \cdots, T_n \subseteq \{1, 2, \cdots, s\}$ with $|T_1| = k$ for each i. Then, for a s-bit string z, and for each i, we will write $z|_{T_i}$ to refer to the k-bit string taken from the indices in z that are in T_i. Our generator $G : \{0,1\}^s \to \{0,1\}^n$ will then be defined by

$$G(z)_i = f(z|_{T_i})$$

for all i.

Notice that the T_i will overlap with each other. A priori, this seems bad, since it means some bits of z will contribute to multiple bits of $G(z)$, which might make it seem less random. However, here we will restrict the overlaps of the T_i by picking the subsets such that they satisfy some property, which will make G as described above a pseudorandom generator. We will fill in that property as we go on in the proof next lecture.

We will now use a proof technique developed by Goldwasser and Micali, called a *Hybrid Argument.* We will proceed by contradiction; assume to the contrary that G is not a pseudorandom generator. Then, there is a destinguisher circuit D, which is smaller than the Nisan-Wigderson assumption allows, such that

$$\Pr_{z \leftarrow \{0,1\}^s}[D(G(z)) = 1] - \Pr_{y \in \{0,1\}^n}[D(y) = 1] > \epsilon,$$

where here $\epsilon = \frac{1}{10}$.

We will now define $n+1$ 'hybrid' distributions. For each $i \in \{0, 1, \cdots, n\}$, Let H_i be the distribution of strings, the first i digits of which are the first i digits of $G(z)$ for a uniformly chosen z as above, and the last $n-i$ digits of which are the last $n-i$ digits of y for y uniformly chosen as above. See Figure 2.

Then, we define α_i by

$$\alpha_i = \Pr_{w \leftarrow H_i}[D(w) = 1].$$

For instance, H_0 is the distribution of choosing all the bits from y, and so H_0 is the uniform distribution on n bits. Meanwhile, H_n is the distribution of taking all the bits from $G(z)$, and so H_k is the same distribution that G gives from a uniformly random seed. We have that

$$\alpha_n - \alpha_0 > \epsilon$$

and so by the Pigeonhole principle, there exists an i such that

$$\alpha_i - \alpha_{i-1} > \frac{\epsilon}{n}.$$

The idea, now, is to show that $\alpha_i - \alpha_{i-1}$ cannot be large. We consider a concrete case; say that $i = 1$. We have that:

$$\alpha_1 = \Pr_{z, y_1, \cdots, y_n}[D(f(z|_{T_1}), y_2, \cdots, y_n) = 1],$$
$$\alpha_0 = \Pr_{z, y_1, \cdots, y_n}[D(y_1, y_2, \cdots, y_n) = 1].$$

In particular, since

$$\Pr_{z, y_1, \cdots, y_n}[D(f(z|_{T_1}), y_2, \cdots, y_n) = 1] - \Pr_{z, y_1, \cdots, y_n}[D(y_1, y_2, \cdots, y_n) = 1] > \frac{\epsilon}{n},$$

there is a particular setting of $y_2, \cdots, y_n$ such that

$$\Pr_{z, y_1}[\overline{D}(f(z|_{T_1})) = 1] - \Pr_{z, y_1}[\overline{D}(y_1) = 1] > \frac{\epsilon}{n},$$

where $\overline{D}(\cdot) = D(\cdot, y_2, \cdots, y_n)$, since if that inequality happens randomly, then there must be a situation in which it happens.

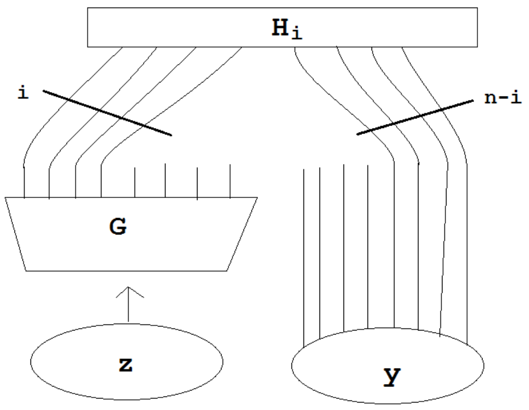

Figure 2: The ith distribution, H_i, in the hybrid argument.

But then we can use $\overline{D}$ to compute f, contradicting our assumption about it. Indeed, say we had any algorithm $\overline{\overline{D}}$ such that

$$Pr_{z,y_1}[\overline{\overline{D}}(z, f(z)) = 1] - Pr_{z,y_1}[\overline{\overline{D}}(z, y_1) = 1] > \frac{\epsilon}{n}.$$

Then, we can use $\overline{\overline{D}}$ to compute f. On a given input z, we compute $\overline{\overline{D}}(z, 0)$ and $\overline{\overline{D}}(z, 1)$. If the former is 1 and the latter is 0, then we pick $f(z) = 0$, while if the former is 0 and the latter is 1 then we pick $f(z) = 1$. Otherwise, we flip a coin. Relatively simple analysis will show that this method usually computes the correct values of f.

Next class we will continue the proof with the harder case where $i \approx n$. □

References

[1] M. Blum, S. Micali, *How To Generate Cryptographically Strong Sequences Of Pseudo Random Bits*, SIAM Journal on Computing, 13(4):850-864, 1984.

[2] N. Nisan, A. Wigderson, *Hardness vs randomness*, Journal of Computer and System Sciences 49(2):149-167, 1994.

[3] A. Yao, *Theory and Applications of Trapdoor Functions*, SFCS '82 Proceedings of the 23rd Annual Symposium on Foundations of Computer Science, 80-91, 1982.