CS 880: Advanced Complexity Theory	2/21/06

Lecture 11: Pseudorandom Generators

Instructor: Dieter van Melkebeek — Scribe: Scott Diehl, Jeff Kinne

Last time we discussed how expanders can be used to achieve randomness-efficient error reduction. Such a process can be thought of as improving the quality of a randomized algorithm by reducing the error probability while using no more (or only slightly more) randomness. Today, we turn to improving the quality of randomized algorithms in a different sense. This time, we use expanders to reduce the number of random bits used by space-bounded randomized algorithms while allowing for a slightly larger error probability.

Specifically, we develop a pseudorandom generator, which is a procedure that efficiently transforms a short string chosen uniformly at random (called a seed) into a larger string which "looks" close to uniform to a certain class of algorithms (in this case, randomized algorithms running in space S). Such generators have been previously developed by Nisan, who shows how two-universal hash functions can be used to construct a pseudorandom generator for space S algorithms which has seed length $O(S^2)$ and runs in space $O(S^2)$ while only increasing the error by an arbitrarily small constant. The pseudorandom generator which we present today matches these parameters, and can even be applied to a wider class of algorithms computed by certain "nice" communication networks.

1 Intuition

The intuition behind the construction of this pseudorandom generator stems from the expander mixing lemma, which we proved in lecture 3:

Lemma 1 (Expander Mixing Lemma)**.** *Let $G = (V, E)$ be of degree d and $|V| = N$. For all $S, T \subseteq V$, we have*

$$\left| \frac{E(S,T)}{dN} - \mu(S)\mu(T) \right| \leq \lambda_2 \sqrt{\mu(S)\mu(T)}.$$

Suppose we wish to select two vertices of G at random such that one vertex is in S and the other is in T. Notice that the probability that this occurs when the two vertices are chosen independently at random is $\mu(S)\mu(T)$. On the other hand, we could select only the first vertex v uniformly at random, and then generate the second vertex by choosing an edge emanating from v at random and using the vertex adjacent to v along this edge. As this process amounts to selecting an edge uniformly at random and using its two endpoints, the probability that these endpoints are in S and T respectively is precisely $\frac{E(S,T)}{dN}$. The expander mixing lemma tells us that the probability of these two processes succeeding is close when G is an expander. However, the latter process uses less randomness– selecting two vertices at random costs $2 \log N$ random bits, whereas selecting a random vertex and a random edge emanating from this vertex costs only $\log N + \log d$. As we typically have $\log d \ll \log N$, this offers a significant savings in randomness without deteriorating the quality of the trial too badly.

This affords us the insight that we can mimic the selection of two independent strings by taking a random edge in an expander and using its endpoints. The end result is almost the same, but we use less random bits. This idea forms the basis for our pseudorandom generator.

2 Basic Construction

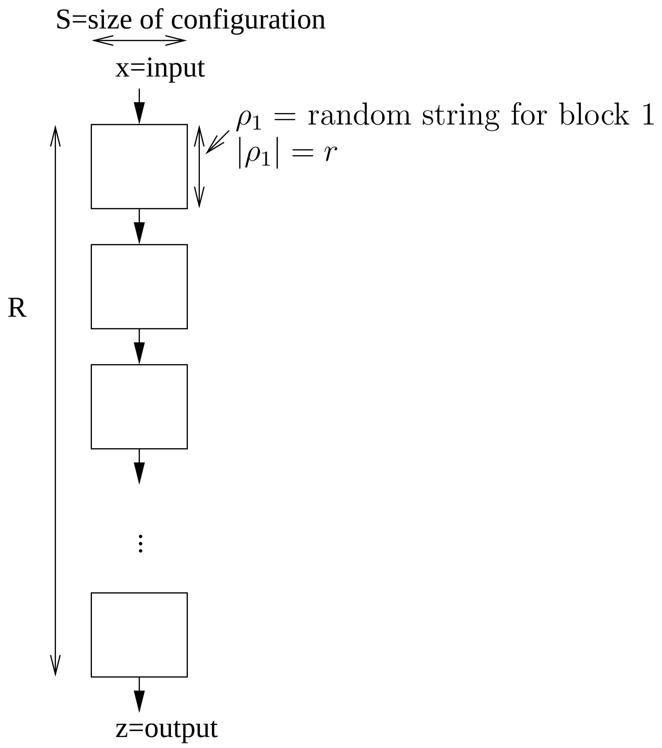

Figure 1: A randomized space bounded computation, broken up into parts that each use the same amount of randomness.

By applying this intuition in a recursive manner, we can construct pseudorandom generators for computations that can be broken up into a number of processes that use only a small amount of communication. Namely, space-bounded computations can be viewed in such a manner. Consider the computation of a space S algorithm which uses R random bits. The computation on a given input can be broken up into a number of consecutive processes, each taking a configuration of size $O(S)$ as input and using r random bits to generate a configuration as output to the next block (see figure 1). Naturally, the input to the first process is the initial configuration, and we accept if and only if the output of the final process is an accepting configuration. Therefore, when S is small, such a network of processors simulates the given algorithm with a small amount of communication.

Consider two consecutive processes in this network (see figure 2), where the first takes input x and random string ρ_1, producing output y given to the second process as input. The second process takes random string ρ_2 and produces output z. As $|\rho_1| = |\rho_2| = r$, the number of random bits required to run these two processes is $2r$. Following the intuition described in the previous section, our pseudorandom generator replaces (ρ_1, ρ_2) chosen independently at random with the vertices adjacent to a randomly chosen edge in an expander of size 2^r. More formally, the generator $g = (g_\ell, g_r)$ is a mapping

$$g : (\rho, \sigma) \to (\rho, \sigma^{\text{th}} \text{ neighbor of } \rho),$$

where $|\rho| = r$ and $|\sigma| = \log d$. A straightforward application of the expander mixing lemma shows that the distribution of the output when we run these two processes on the output of g (where the

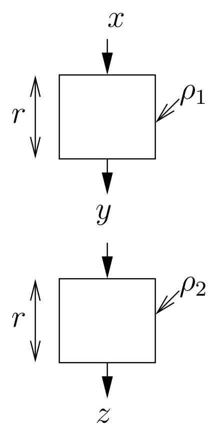

Figure 2: 2 consecutive parts of a randomized process, each using the same amount of randomness.

first process is given g_ℓ and the second g_r as random strings) is not too different than that when the two strings are chosen independently at random. Namely, if the output distribution of this two process computation (as a function of x, ρ_1 and ρ_2) is $Z(\cdot,\cdot,\cdot)$, then we have:

Theorem 1. *For any distribution X,*

$$\|Z(X, U_r, U_r) - Z(X, g(U_r, U_{\log d}))\|_1 \leq (\lambda_2 2^S)^{\Omega(1)}.$$

3 Extending the Construction

Before proving this theorem, we discuss how to extend the above construction to get a pseudorandom generator with seed length $O(S^2)$ computable in space $O(S^2)$. We also discuss how such a generator allows for a space $O(S^2)$ deterministic simulation of a space S randomized computation.

Given a space S computation using R random bits, we break this computation up into 2^ℓ processes, each using $r = R/2^\ell$ bits. We then apply the above construction to each successive pair, resulting in a computation consisting of $2^{\ell-1}$ processes each using $r + \log d$ bits of randomness. The output distribution of each process is only slightly different than that of each corresponding pair in the previous network, as per Theorem 1. This strategy can now be repeated using a slightly larger expander (holding d constant) to arrive at a computation consisting of $2^{\ell-2}$ processes each using $r + 2\log d$ random bits. Repeating this technique ℓ times, we arrive at a computation consisting of one process (see figure 3), which we use to simulate our original computation. It remains to calculate how good this simulation is, namely, its error, number of random bits used, and complexity.

Error

We compute the error of the pseudorandom generator by computing the 1-norm of the difference between output distributions at successive recursive levels of the construction. Within a level, we break it up further by looking at the distribution difference when we only replace the first two processes by one process using the pseudorandom generator (see figure 4). The output distribution of either computation is then determined by running the remaining randomized processes on the output z of the appropriate process. Since we know that applying a randomized function f to two

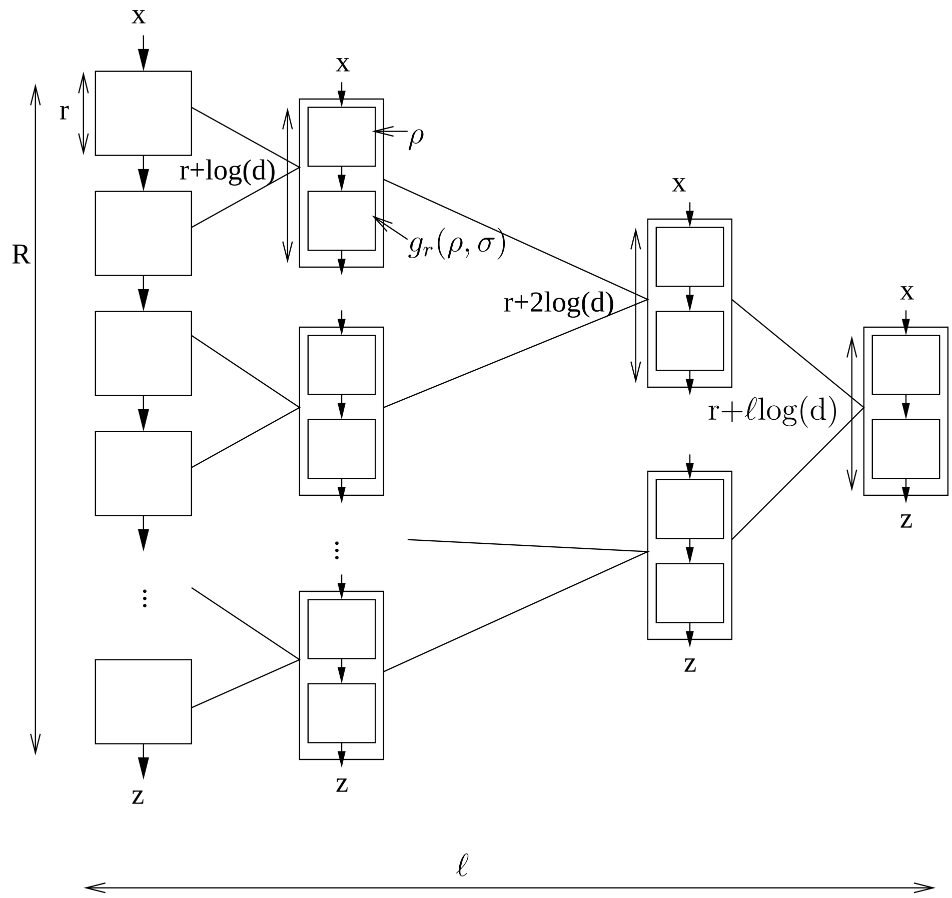

Figure 3: Construction of the pseudorandom generator. At each level in the tree, 2 adjacent "processes" are collapsed into a single process that takes pseudorandom random bits.

distributions X and Y cannot increase the norm of their difference,

$$\|f(X) - f(Y)\|_1 \leq \|X - Y\|_1,$$

the difference of the output distributions of these two computations can be upper bounded by Theorem 1 (where X is the output distribution of the first 2 processes of the original randomized machine, Y is the output distribution of the first 2 processes of the machine with random bits from the pseudorandom generator instead of completely uniform random bits, and f is the remaining $2^\ell - 2$ processes of the original randomized machine).

Similarly, we can apply the same argument for distributions X' and Y' where: X' is the output distribution of the 4^{th} process when the first 2 processes are given random bits from the pseudorandom generator, and the next 2 processes are given true random bits; Y' is the output distribution of the 4^{th} process when both the first 2 and the next 2 processes are given random bits from the pseudorandom generator. Again, the distribution difference is upper bounded by Theorem 1. Proceeding in this manner, we switch from one level to the next with half as many processes using the pseudorandom generator while accumulating error at most $2^{\ell-1} \cdot (\lambda_2 2^S)^{\Omega(1)}$, and summing up over

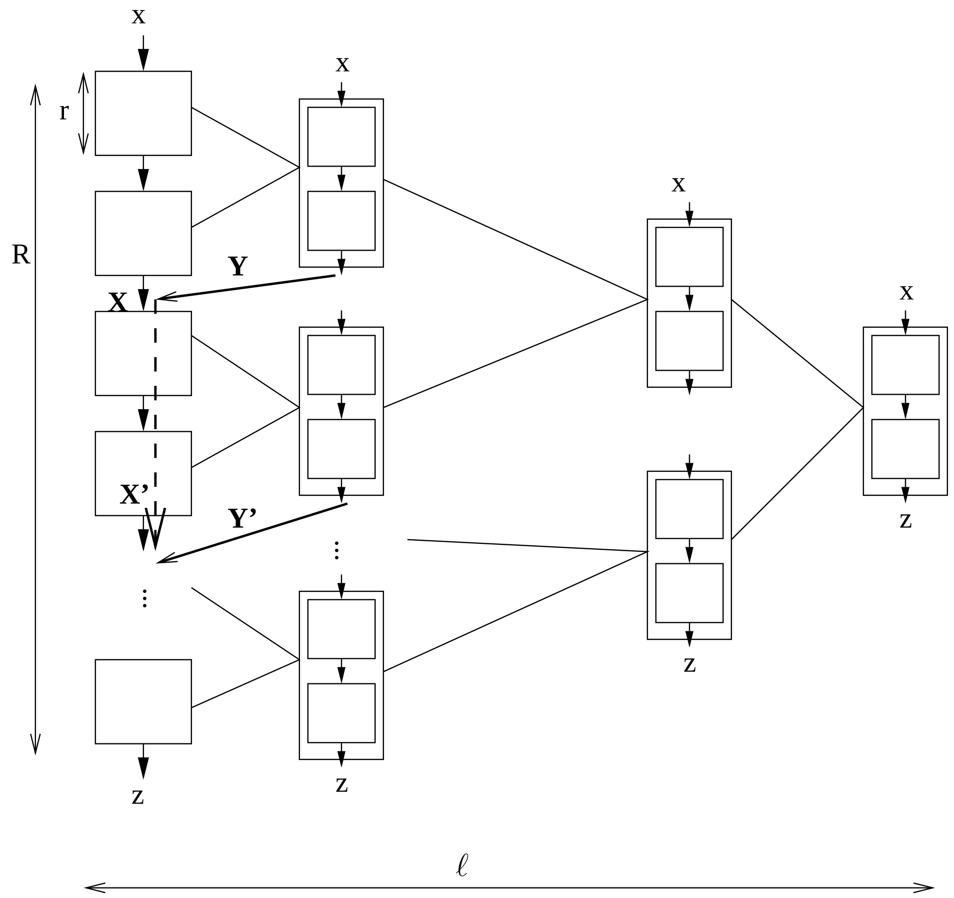

Figure 4: To calculate the difference between the original output distribution and that achieved using the pseudorandom generator, we analyze the difference in distributions that use increasingly more of the pseudorandomness.

all ℓ levels gives error at most

$$(\lambda_2 2^S)^{\Omega(1)}(2^{\ell-1} + 2^{\ell-2} + \cdots + 1) = (\lambda_2 2^S)^{\Omega(1)}(2^\ell - 1) \leq (\lambda_2 2^S)^{\Omega(1)}(2^\ell). \tag{1}$$

Random bits

Each level of the construction requires only an additional $\log d$ random bits so long as the degree d of the expanders used at each level is constant. Therefore, the total number of random bits used is $r + \ell \log d = \frac{R}{2^\ell} + \ell \log d$.

Notice that we have freedom to choose the depth ℓ of the tree to optimize the performance of the algorithm. We would like to minimize the number of random bits used while introducing error at most ϵ. First consider the $\log d$ term of the randomness. We know that for optimal expanders, we have $\lambda_2 = O(\frac{1}{\sqrt{d}})$, which means that $\log d = \Omega(\log \frac{1}{\lambda_2})$, and the amount of randomness can be written as

$$\frac{R}{2^\ell} + \ell \cdot \Omega(\log \frac{1}{\lambda_2}). \tag{2}$$

If we introduce error at most ϵ, condition (1) tells us that $2^\ell(\lambda_2 2^S)^{\Omega(1)} = \epsilon$. Some manipulations gives an expression for $\log\frac{1}{\lambda_2}$ that we can plug into (2):

$$2^\ell(\lambda_2 2^S)^{\Omega(1)} = \epsilon$$

$$\implies \ell + \Omega(\log(\lambda_2 + S)) = \log\epsilon$$

$$\implies \log\frac{1}{\lambda_2} = S + \Omega(\ell + \log\frac{1}{\epsilon})$$

$$\text{for } \ell \le S \implies \log\frac{1}{\lambda_2} = \Omega(S + \log\frac{1}{\epsilon}).$$

Plugging this into (2) for $\log d$ means we would like to minimize

$$\frac{R}{2^\ell} + \ell \cdot \Omega(S + \log\frac{1}{\epsilon})$$

by picking ℓ appropriately. Notice that we are adding 2 functions, one of which decreases with ℓ (namely $\frac{R}{2^\ell}$) and the other increases with ℓ (namely $\ell \cdot \Omega(S + \log\frac{1}{\epsilon})$). It is easy to see that choosing ℓ to be the point of intersection of the two functions will be optimal to within a factor of 2 (to see why, consider a graph of the two functions and their sum). In our case, this would mean setting ℓ so that $\ell \cdot 2^\ell = \frac{R}{\Omega(S+\log\frac{1}{\epsilon})}$, which means that ℓ will be slightly smaller than S but very close to S. If we choose $\ell = S$, we will use randomness $O(S^2)$ provided ϵ is not too small (in fact ϵ can be as small as $\frac{1}{2^{O(S)}}$).

Note that this analysis assumes we can actually achieve the expansion ratio $\lambda_2 = O(\frac{1}{\sqrt{d}})$ to run the construction. See the notes below for a potential problem with this and how it can be dealt with.

Complexity

The random string at any given level of the pseudorandom generator construction describes a vertex v and the label of an edge leaving this vertex. Computing the output of the pseudorandom generator at this level can be done by recursively computing the output at the previous level with the label of v as input, computing the appropriate neighbor w of v, and then recursively computing the output of the previous level on input w. The neighbor relation can be computed in linear space for many of the expander families we've seen, which in this case is $\Theta(S^2)$ at the topmost level. As it only requires $\log\ell = \log S$ space to write down the recursive level, and we can reuse the space required to compute the neighbor relation between recursive calls, the total space to compute the pseudorandom generator is $\Theta(S^2)$.

As a corollary, we obtain space $O(S^2)$ deterministic simulations of space S randomized computations by running the latter on the output of the pseudorandom generator on all seeds of length $O(S^2)$ and taking the majority vote.

Corollary 1.

$$\mathrm{BPSPACE}(S) \subseteq \mathrm{DSPACE}(S^2).$$

4 Notes

- In calculating the optimal parameters for minimizing randomness, we assumed that we could use expander graphs with $\lambda_2 = O(\frac{1}{\sqrt{d}})$. At first glance this should not pose a problem because we have seen expander constructions that achieve this spectral gap. However, in order to optimize the randomness, we set $\ell = S$. This means that $r = \frac{R}{2^\ell} = 1$. Recall that r is the amount of randomness used by the individual leaf processes of the tree, and is also the number of bits needed to specify a vertex in our expander graph. It clearly may not be possible to have a good enough expander on a graph with a small constant number of vertices.

  We get around this problem in one of two ways. One way is to pick ℓ slightly less than S (our analysis of the randomness allowed for this), so that the leaf processes use a greater amount of randomnes (hence making our expander graphs large enough to realize the spectral gap). Another way to get around this problem is to change the construction slightly near the leaf levels. Our analysis of randomness assumed an additional additive term of $\log d$ random bits at each level. For the first few levels, we could simply supply these random bits without using an expander. We can do this for all of the levels until more than $\log d$ random bits are needed for the individual processes (and then at that point use the pseudorandom generator as we've already stated it).

- In calculating the error of the pseudorandom generator, we assumed that the randomized computation always halts. As most familiar randomized algorithms have such a property (e.g., the random walk algorithm for undirected S-T connectivity), this is not really a limiting assumption.

  Also, it is important to note that the machine model we are using treats its random bits as coin flips, where each can only be accessed once (unless copied down at the cost of space). We may also consider the more powerful model which allows two-way access to the random bits. In this case, our argument breaks down and the pseudorandom generator doesn't work for this model. This is not surprising, as such a model can be seen to allow for shared random bits between the processes comprising the computation. As we achieve provably better communication complexity bounds for models with shared randomness than without (i.e., equality), this indicates that such a network can communicate with far fewer bits than before, and is therefore much harder to fool.

- Although this pseudorandom generator matches Nisan's in many parameters, the latter can be used to show that BPSPACE(S) $\subseteq$ DTISP($2^{O(S)}, S^2$) (where $DTISP(s,t)$ is the class of languages decidable in simultaneous deterministic time t and space s). It is not clear if the former allows for the same result. On the other hand, this pseudorandom generator does extend to other models of computation which have small communication complexity between processes, while Nisan's generator does not.

All that remains is to prove Theorem 1. Today we present a proof based on the expander mixing lemma, and tomorrow we will present an alternate proof via extractors.

5 Proof 1 of Theorem 1

First notice that we can prove Theorem 1 by only considering point distributions. Point distributions place all probability in a single point. Any distribution D is a convex combination of point distributions $X_1, X_2, ..., X_k$. If we consider the quantity

$$\Delta_D = \|Z(D, U_r, U_r) - Z(D, g(U_r, U_{\log d}))\|_1,$$

we know by the triangle inequality for 1-norm that $\Delta_D \leq \Sigma_k \Delta_{X_k}$. Hence, we will consider a point distribution on X with $\Pr[X = x] = 1$.

With this in mind, consider any fixed z. The probability that the output is z for this particular point distribution X can be written as

$$\Pr[Z = z | X = x] = \sum_y \underbrace{\Pr[\rho_1 \in S_{xy}] \Pr[\rho_2 \in T_{yz}]}_{(*)},$$

where S_{xy} is the set of strings that cause the first process to output y on input x, and T_{yz} is the set of all strings that cause the second process to output z on input y.

When ρ_1 and ρ_2 are chosen uniformly at random, we know that the value of $(*)$ is $\mu(S_{xy})\mu(T_{yz})$ (because ρ_1 and ρ_2 are independent). However, when chosen by an edge from the expander, we have that $(*)$ is the probability that such an edge goes from S_{xy} to T_{yz}, namely, $E(S_{xy}, T_{yz})/(dN)$. By the expander mixing lemma, we can bound the absolute value of the difference in $(*)$ for the two cases by

$$\lambda_2 \sqrt{\mu(S_{xy})\mu(T_{yz})}.$$

This gives that the absolute value of the difference in probability that each case outputs z is at most

$$\lambda_2 \sum_y \sqrt{\mu(S_{xy})\mu(T_{yz})}. \tag{3}$$

As this represents the difference in probability for a single point of the distribution Z, the 1-norm difference of Z is at most the sum of (3) over all z,

$$\|Z(X, U_r, U_r) - Z(X, g(U_r, U_{\log d}))\|_1 \leq \lambda_2 \sum_{y,z} \sqrt{\mu(S_{xy})\mu(T_{yz})}.$$

By Cauchy-Schwartz, we have that the latter expression is at most

$$\lambda_2 \sqrt{\underbrace{\sum_{y,z} \mu(S_{xy})}_{(\alpha)}} \sqrt{\underbrace{\sum_{y,z} \mu(T_{yz})}_{(\beta)}}.$$

To calculate (α) we break up the sum as $\sum_{y,z} \mu(S_{xy}) = \sum_z \sum_y \mu(S_{xy})$. Notice that the inner sum represents the probability that x produces *some* y, which must be 1. Thus, we have

$$(\alpha) = \sum_z 1 = 2^S$$

since there are 2^S possible strings z. Similarly, $(\beta) = 2^S$ since for a fixed y, the sum over all z is 1 (y definitely produces some output z), and there are 2^S strings y. Thus, we can conclude that

$$\|Z(X, U_r, U_r) - Z(X, g(U_r, U_{\log d}))\|_1 \leq \lambda_2 2^S,$$

which is of the desired form.