

Lecture 15: Pseudorandomness for Logarithmic Space

Instructors: Holger Dell and Dieter van Melkebeek

Scribe: Gautam Prakriya

DRAFT

In the last lecture, we discussed a recursive construction of a pseudorandom generator that is based on expanders: the INW generator. In this lecture we will show that the INW-generator with a suitable instantiation of an expander family produces pseudorandomness for machines that run in logarithmic space. We also introduce *Nisan's generator*, which is a pseudorandom generator that has a recursive structure that is slightly different from, but related to, the recursive structure of the INW-generator, and which employs hash families instead of expanders. We begin with a description of the class of algorithms that we want to produce pseudorandomness for.

1 Logspace Turing Machines

One way to model bounded space computation uses the Turing machine model and restricts the length of the work tapes. A logspace machine M is a Turing machine with a read-only input tape that contains the input x , a work tape of size $O(\log |x|)$ that corresponds to the memory, and a write-only output tape. We usually equate access to randomness with the ability to flip a fair coin, and we model this situation by allowing the machine to access an additional infinite read-once tape that is initialized with bits which are sampled independently and uniformly at random. Notice that the read-once restriction captures the fact that the outcome of a coin toss must be stored in the configuration of the machine if it is to be used later in the computation. As usual, we denote the random string used by the machine as ρ and its length by r .

For notational simplicity we will only talk about decision problems, though much of what we discuss here is easily extended to more general settings. We further restrict ourselves to randomized logspace machines with bounded, one-sided error, i.e., machines that may falsely reject a YES-instance, but that never falsely accept a NO-instance. This corresponds to RL-machines as formally defined below.

Definition 1. A logspace-machine M is an RL-machine for a language L if, for all x , we have

- if $x \in L$, then $\Pr_{\rho}[M(x, \rho) \text{ accepts}] \geq 1/2$,
- if $x \notin L$, then $\Pr_{\rho}[M(x, \rho) \text{ accepts}] \leq 0$, and
- M halts after at most $\text{poly}(|x|)$ steps.

The third condition states that the worst-case running time of the algorithm should be polynomial. Notice that any deterministic logspace machine that halts, halts in polynomially many steps. This is not the case for a randomized machine, which is why we need the termination condition in the definition. Consider for example the machine that loops until it finds a 1 in the random string ρ and then accepts. With probability one, this machine halts and accepts, but if the random string happens to be the infinite 0-string, it never halts, so the worst-case running time is unbounded.

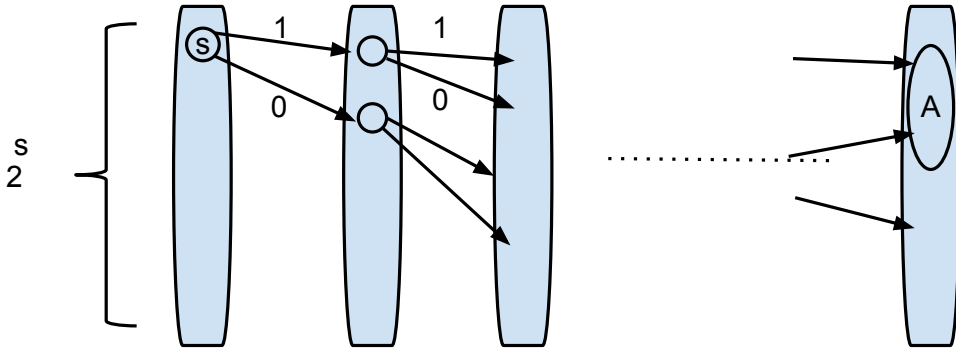


Figure 1: Pictorial representation of a branching program of width $w = 2^S$ and accepting set A . Every non-final vertex has two outgoing edges to the next layer, and in each layer, a particular bit ρ_i of the random string is read.

1.1 Branching Programs

Branching Programs are the non-uniform analogues of bounded-space Turing machines. For our purposes, we think of a branching program as a layered directed acyclic graph in which each layer has the same number of nodes (see Figure 1). Every node except the ones in last layer have two outgoing edges, labeled with 0 and 1, leading to nodes in the next layer. More formally, a branching program B of length r and width w is a function $B : [r - 1] \times [w] \times \{0, 1\} \rightarrow [r] \times [w]$ so that $B(i, *, *) = (i + 1, *)$ holds for all $i < r$, where $[r] = \{1, \dots, r\}$. Note that branching programs essentially constitute a special case of finite state machines.

A branching program B can be viewed as a model of computation by picking a start node $s = (1, *)$ in the first layer and a set of accepting nodes $A \subseteq \{r\} \times [w]$ in the last layer. Then a string ρ is *accepted* by B if there is a path from s to A such that the edge labels in the path correspond to the bits in the string ρ .

An RL-machine M can be translated to a branching program $B = B_{M,x}$ as soon as the input x is known: We let w be the total number of configurations that the logspace machine M can be in when x is fixed. Since M has $S = O(\log |x|)$ space and $O(1)$ internal states, the total number of distinct configurations of M is upper bounded by $O(2^S) = \text{poly}(|x|)$. In each layer, the set of nodes $(i, *)$ corresponds to the set of configurations that M can be in right before step i . Let s correspond to the initial configuration of M , that is, the configuration that M on input x is in right before it reads the first random bit $\rho_1 \in \{0, 1\}$. Let C_b be the configuration of M right before reading ρ_2 in case $\rho_1 = b$. Then we define $B(s, \rho_1) = (2, C_{\rho_1})$. In general, every time a new random-bit is read and depending on the value of the bit, the machine M moves to a new configuration, and this transition is modeled by B . Notice that we only model changes in the configuration of the machine when a random bit is read; the deterministic part of the computation and its dependency on the actual input x is hidden in the non-uniformity of the model. The length of the branching program is equal to the number of random bits r read by the machine and therefore upper bounded by the running time $\text{poly}(|x|)$ of M . Finally, the accepting states A in the final layer correspond to the accepting configurations of M .

Let us introduce some notation to talk about specific runs of B when the original input is fixed

and the randomness input is given. On (randomness) input $\rho \in \{0, 1\}^{r'}$, we overload our notation of B to a function $B : [r-1] \times [w] \times \{0, 1\}^{r'} \rightarrow [r] \times [w]$ defined recursively as $B(i, j; \rho_1 \rho_2 \dots \rho_\ell) = B(B(i, j; \rho_1); \rho_2 \dots \rho_\ell)$. Finally, we may also write $B : \{0, 1\}^r \rightarrow \{0, 1\}$ for the function computed by the branching program B , that is,

$$B(\rho) \doteq \begin{cases} 1 & \text{if } B(s; \rho) \in A, \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

Since M was an RL-machine for a language L , the program $B = B_{M,x}$ satisfies $\Pr_\rho(B(\rho) = 1) = 0$ if $x \notin L$, and it satisfies $\Pr_\rho(B(\rho) = 1) \geq 1/2$ if $x \in L$. In either case, we expressed the acceptance probability of M on input x in terms of the probability that B outputs one. Since B can be seen as a directed graph with out-degree two, the latter is equal to the probability that a random walk in B , started at s , will end in A after r steps.

2 Pseudorandomness for Branching Programs

In this section we prove that the INW-generator instantiated with a suitable expander family fools read-once branching programs. After that, we introduce another recursive construction of a PRG for space bounded computation known as Nisan's generator.

For convenience, let us discuss what it means to be pseudorandom for branching programs. We say that a generator $G : \{0, 1\}^\ell \rightarrow \{0, 1\}^r$ is ϵ -pseudorandom for branching programs of length r and width w if, for all such branching programs B , we have

$$\left| \Pr_{\rho \sim U_r} (B(\rho) = 1) - \Pr_{\rho \sim G(U_\ell)} (B(\rho) = 1) \right| \leq \epsilon. \quad (1)$$

2.1 INW-Generator

We recall the definition of the INW-generator, Definition 1 from Lecture 14, but we specialize the definition to the case in which the output of the generator is a binary string; this is what we ultimately need as the input of the branching program. There are several ways of doing this specialization, and we choose to simply restrict the initial symbol set $[D]$ to be binary.

Definition 2 (Impagliazzo–Nisan–Wigderson Generator (binary version)).

Let H_1, H_2, \dots be a sequence of directed outregular graphs such that H_i has out-degree d_i , vertex set $\{0, 1\} \times [d_1] \times \dots \times [d_{i-1}]$ and neighbor function $\Gamma_{H_i} : V(H_i) \times [d_i] \rightarrow V(H_i)$. Then the INW-generator is the family of functions $G_i : \{0, 1\} \times [d_1] \times \dots \times [d_i] \rightarrow \{0, 1\}^{2^i}$ such that

$$\begin{aligned} G_0(h_0) &\doteq h_0 \\ G_{i+1}(h_0 \dots h_{i+1}) &\doteq G_i(h_0 \dots h_i) \circ G_i(\Gamma_{H_{i+1}}(h_0 \dots h_i; h_{i+1})), \end{aligned}$$

where $h_0 \in [D]$ and $h_j \in [d_j]$ for all $j > 0$.

We let H_1, H_2, \dots be a family regular expanders of degree at most d each and with some sufficiently good spectral expansion at most λ to be chosen later. We have seen before that $d \sim 1/\lambda^2$ can be achieved. Unfortunately, it is technically tricky to use the fully explicit family that we constructed

based on the replacement product, because we have to worry about the exact degrees and number of vertices that we need in the construction of the INW-generator. We ignore these issues here and assure the reader that these difficulties can be overcome.

Note that the first $d - 1$ graphs above, that is, H_1, \dots, H_{d-1} , will be chosen to be complete graphs, that is, H_1 is the complete graph on two vertices and all self-loops, and H_{d-1} is the complete graph with d vertices and all self-loops. Only after that point will we employ a d -regular family of expanders with second-largest eigenvalue at most λ .

To transform the INW-generator G_k to a generator $G_k : \{0, 1\}^\ell \rightarrow \{0, 1\}^r$ that takes a binary input, we represent the elements of $\{0, 1\} \times [d_1] \times \dots \times [d_k]$ in binary, which can be done with strings of length $\ell \doteq 1 + \sum_{i=1}^k \log d_i \leq k \cdot \log d$. Since $r = 2^k$ and $d \sim 1/\lambda^2$, we achieve a seed length of $\ell \sim \log r \cdot \log(1/\lambda)$. To get the error down to $\epsilon > 0$ for general branching programs, we will require $\lambda = \text{poly}\left(\frac{\epsilon}{rw}\right)$ in the analysis below, for which reason the seed length of the INW-generator is

$$\ell \sim \log r \cdot \left(\log r + \log w + \log(1/\epsilon) \right).$$

More succinctly, in the most interesting case where the width w is polynomial in r and the error ϵ is a constant, say $1/2$, the seed length simplifies to $\ell \sim \log^2 r$.

We now prove that the INW-generator is pseudorandom for branching programs and analyze the error. Unfortunately, we don't know how to analyze the error using the derandomized squaring of expanders as we did for undirected connectivity in Lecture 14. The analysis below appears to be different, and relies directly on the quasirandomness property of expanders that we proved in the expander mixing lemma (Lemma ?? of Lecture 10).

Lemma 1. *For all $k \in \mathbb{N}$, the INW-generator G_k instantiated with expanders of second-largest eigenvalue at most λ is $(3^k \cdot w \cdot \lambda)$ -pseudorandom for branching programs of length $r = 2^k$ and width w .*

Proof. Let B be a branching program length $r = 2^k$ and width w . We need to prove (1). To this end, we will actually prove a stronger condition so that we can use induction. The stronger condition is that the statistical distance between the distributions of the state that we reach when reading a uniform input and the one when reading a pseudorandom input is small. Formally, the claim is that, for all $i \in \mathbb{N}$ and all branching programs B of length 2^i , the following holds for all states $s \in \{1\} \times [w]$:

$$d_{\text{stat}}\left(B(s; U_{2^i}), B(s; G_i(U))\right) \leq \epsilon_i \doteq 3^i w \lambda. \tag{2}$$

Since we want to avoid specifying the seed length of G_i here, we let the distribution U be uniform on all valid inputs of G_i . Recall that $B(s; \rho)$ denotes the *state* in $[r] \times [w]$ that we reach when starting B at s and reading ρ ; thus, $B(s; D)$ is a distribution on the states that we reach when the randomness comes from a distribution D .

To see that (2) implies (1), note that the statistical distance in (2) is by definition equal to the maximum of the left-hand side of (1) taken over all sets $A \subseteq [r] \times [w]$.

The proof of (2) proceeds by induction on i . For $i = 0$, we have $r = 1$ and both U_r and $G_i(U)$ are just a single uniform bit; so the distributions are the same and the statistical distance in (2) is equal to zero. For the induction step $i \rightarrow i + 1$, we essentially split the computation of the branching program in the middle and use the induction hypothesis on both parts. We use the

triangle inequality to show an upper bound on the error of the entire execution in terms of the errors of the two parts.

So first consider the input to be truly random. Then we can rewrite the output distribution under truly random input as $B(s; U_{2^{i+1}}) = B(B(s; U_{2^i}); U_{2^i}) = B(M_U; U_{2^i})$ where M_U is the random variable $B(s; U_{2^i})$ that holds the state we reach in the middle layer after reading the first half of the truly random input, and $B(M_U; U_{2^i})$ is the state that we reach after reading the second half of the truly random input provided that we started at a state sampled from M_U .

Because of the recursive structure of the INW-generator, we can do a similar decomposition for the pseudorandom distribution. Namely, we split the uniform input distribution U for G_{i+1} into two parts X and Y , so that X corresponds to the uniform distribution on the edge labels $h_0 \dots h_i$ and Y corresponds to the uniform distribution on the edge labels h_{i+1} . Then we can write

$$G_{i+1}(XY) = G_i(X) \circ G_i(\Gamma(X; Y)), \quad (3)$$

where $\Gamma = \Gamma_{H_{i+1}}$ is the function from the definition of the INW-generator. As above, we can then write

$$B(s; G_{i+1}(U)) = B\left(B(s; G_i(X)); G_i(\Gamma(X; Y))\right) = B(M_P; G_i(\Gamma(X; Y))),$$

where $M_P = B(s; G_i(X))$ is the random variable that holds the state that we reach in the middle layer after reading a random string sampled from the pseudorandom distribution $G_i(X)$.

Our goal is to show 2, namely that $B(s; U_{2^{i+1}})$ and $B(s; G_{i+1}(U))$ are close in statistical distance. We are now in position to use the triangle inequality to break up the computation in the middle:

$$\begin{aligned} d_{\text{stat}}\left(B(s; U_{2^{i+1}}), B(s; G_{i+1}(U))\right) &= d_{\text{stat}}\left(B(M_U; U_{2^i}), B(M_P; G_i(\Gamma(X; Y)))\right) \\ &\leq d_{\text{stat}}\left(B(M_U; U_{2^i}), B(M_P; U_{2^i})\right) \end{aligned} \quad (4)$$

$$+ d_{\text{stat}}\left(B(M_P; U_{2^i}), B(M_P; G_i(U))\right) \quad (5)$$

$$+ d_{\text{stat}}\left(B(M_P; G_i(U)), B(M_P; G_i(\Gamma(X; Y)))\right). \quad (6)$$

In the following claims, we will use the induction hypothesis and the expansion properties to bound this sum by $\epsilon_i + \epsilon_i + w \cdot \lambda$; by the definition in (2), this is equal to $2 \cdot 3^i w \lambda + w \lambda \leq 3^{i+1} w \lambda = \epsilon_{i+1}$, which then concludes the induction step. Let us now bound the three terms separately.

To bound the first term of the sum, we use the induction hypothesis on the first half of the branching program.

Claim. (4) $\leq \epsilon_i$.

Proof. We get $d_{\text{stat}}(M_U, M_P) \leq \epsilon_i$ since M_U and M_P are the output distributions of a branching program of length 2^i under uniform input U_{2^i} and under pseudorandom input sampled from $G_i(U)$, respectively. Furthermore, the process $F : S \mapsto F(S) \doteq B(S; U_{2^i})$ is a (randomized) procedure applied to the (outcome of) the random variable S . It is a general fact about the statistical distance that no process can increase the statistical distance between two random variables, that is, (4) $= d_{\text{stat}}(F(M_U), F(M_P)) \leq d_{\text{stat}}(M_U, M_P)$ holds, which establishes the lemma. \square

To bound the second term of the sum, we use the induction hypothesis on the second half of the branching program.

Claim. (5) $\leq \epsilon_i$.

Proof. By the triangle inequality, we get

$$(5) \leq \mathbb{E}_{m \sim M_P} \left[d_{\text{stat}} \left(B(m; U_{2^i}), B(m; G_i(U)) \right) \right] \leq \max_{m \in \text{middle layer}} d_{\text{stat}} \left(B(s; U_{2^i}), B(s; G_i(U)) \right).$$

The second half of B started at any vertex m from the middle layer of B is a branching program of length 2^i . Hence the induction hypothesis applies and we get an upper bound of ϵ_i . \square

The following claim is the crux of the INW-generator and the only part of the proof that uses the facts that the branching program has width at most w and that the function Γ in the definition of the INW-generator corresponds to the neighbor function in an expander graph with second-largest eigenvalue at most λ .

Claim. (6) $\leq w \cdot \lambda$.

Proof. We will again upper bound the statistical distance by the worst-case state m that we might reach in the middle layer as we did in the proof of the previous claim. However, there is a complication since the distribution X may change if we condition on reaching a certain state m in the middle layer. This is because the distributions M_P and $G_i(\Gamma(X; Y))$ are not independent since M_P and X may be correlated. However, since M_P is a distribution on at most w states, the amount of dependence between the two distributions is actually fairly small. Since Y and M_P are fully independent, this limited dependence allows us to use Y to reshuffle X enough for the second half of the branching program not to notice any significant correlation of its input to the input that the first half received.

Formally, we define the random variable $X_m \doteq (X | M_P = m)$ for each possible outcome of the middle state m . That is, for all x in the support of X , we define $\Pr(X_m = x) \doteq \Pr(X = x | M_P = m)$. Note that, because X was uniform, the new distribution X_m is uniform on its support $\text{supp}(X_m)$.

We use the 1-norm characterization of the statistical distance:

$$\begin{aligned} (6) &= \frac{1}{2} \sum_{f \in \text{final layer}} \left| \Pr(B(M_P; G_i(U)) = f) - \Pr(B(M_P; G_i(\Gamma(X; Y))) = f) \right| \\ &= \frac{1}{2} \sum_{f \in \text{final layer}} \left| \mathbb{E}_{m \sim M_P} \Pr(B(m; G_i(U)) = f) - \mathbb{E}_{m \sim M_P} \Pr(B(m; G_i(\Gamma(X_m; Y))) = f) \right| \\ &\leq w \cdot \max_{\substack{m \in \text{middle layer} \\ f \in \text{final layer}}} \left| \Pr(B(m; G_i(U)) = f) - \Pr(B(m; G_i(\Gamma(X_m; Y))) = f) \right|. \end{aligned}$$

The inequality above follows from the triangle inequality and the fact that the final layer has at most w states f . It remains to bound the absolute value for all m and f , which we will do using the quasirandomness property of the expander H_{i+1} , whose neighbor function is $\Gamma = \Gamma_{H_{i+1}}$. Recall that the vertex set of H_{i+1} is the support of X , and the labels used on the edges is the support of Y . Now we define the sets $S \doteq \text{supp}(X_m) = \{x \mid B(s; G_i(x)) = m\}$ and $T \doteq \{x' \mid B(m; G_i(x')) = f\}$. Note that $\Pr(B(m; G_i(U)) = f) = \frac{|S||T|}{|V(H_{i+1})|^2}$ and $\Pr(B(m; G_i(\Gamma(X_m; Y))) = f) = \frac{|E(S, T)|}{|E(H_{i+1})|}$. Since

the second-largest eigenvalue of H_{i+1} is at most λ , the expander mixing lemma (Lemma ?? of Lecture 10) implies

$$(6) \leq w \cdot \max_{m,f} \left| \frac{|S| \cdot |T|}{|V(H_{i+1})|^2} - \frac{|E(S,T)|}{|E(H_{i+1})|} \right| \leq w \cdot \lambda. \quad \square$$

We completed the analysis of the pseudorandomness of the INW-generator, and therefore the proof of Lemma 1. \square

We already stated the resulting seed length of the generator, but for reference, we restate it as a theorem.

Theorem 2 ([INW94]). *For all $r, w \in \mathbb{N}$ and $\epsilon > 0$, the INW-generator $G_{\log r} : \{0, 1\}^\ell \rightarrow \{0, 1\}^r$ with seed length*

$$\ell \sim \log r \cdot \left(\log r + \log w + \log(1/\epsilon) \right)$$

is ϵ -pseudorandom for branching programs of width w and length r . Furthermore, the generator can be computed in space $O(\ell)$.

Proof. Let H_1, H_2, \dots be an expander family with $\lambda < \epsilon/(r^2w)$ and let $k = \log r$. Then, by Lemma 1, the INW-generator G_k is ϵ -pseudorandom for branching programs of length r and width w since $3^k w \lambda \leq \epsilon$. Furthermore, the seed length is $\ell \leq k \cdot \log d$, where d is an upper bound on the degree of the expanders. Since $d \sim 1/\lambda^2$ can be achieved, we have $\ell \leq C \cdot (\log r(\log r + \log w + \log(1/\epsilon)))$ for some absolute constant $C > 0$ independent from r, w , and ϵ .

Furthermore, G_k can be computed in space $O(\ell)$ because of Fact 4 in Lecture 14 and the fact that the expander family is fully explicit. \square

Finally, we consider the consequences of this theorem for the derandomization of randomized logspace RL. Recall that L denotes the complexity class for deterministic logspace, and we write $L^2 = \text{DSpace}(\log^2 n)$ for the class of problems that can be decided by machines that use space $O(\log^2 n)$.

Corollary 3. $\text{RL} \subseteq L^2$.

Proof. Let $L \in \text{RL}$ and let M be an RL-machine that decides L , and let $r = \text{poly}(n)$ be an upper bound on the number of random bits used by M on an input of length n and $w = \text{poly}(n)$ is an upper bound on the size of M 's configuration graph. We construct an L^2 -machine M' for L . On input x of length n , we will use the INW-generator G from Lemma 1 with $\epsilon = 1/3$. Then M' works as follows:

1. Cycle through all seeds $\sigma \in \{0, 1\}^\ell$, and
 - (a) Simulate M on input x
 - (b) Keep track of the position $j \in [r]$ in M 's random tape
 - (c) Whenever M requests a random bit, provide $\rho_j \doteq (G(\sigma))_j$ and increment j .
2. M' accepts if and only if at least one run of M above accepts.

This algorithm uses space $\Theta(\log^2 n)$ because $\ell \sim \log^2 n$. For the correctness of the algorithm, let x be some input. If $x \notin L$, then $\Pr_\rho(M(x; \rho) = 1) = 0$ and hence no run of M can accept. On the other hand, if $x \in L$, then $\Pr_\rho(M(x; \rho) = 1) \geq 1/2$. For the analysis in this case, let $B = B_{M,x}$ be the corresponding branching program of width w and length r so that $M(x; \rho) = B(\rho)$ for all ρ . By Lemma 1, we have $|\Pr(B(U_r) = 1) - \Pr(B(G(U_\ell)) = 1)| \leq \epsilon = 1/3$, and therefore, $\Pr(B(G(U)) = 1) = \Pr_\sigma(M(x; G(\sigma)) = 1) \geq \frac{1}{2} - \frac{1}{3} > 0$. This implies that the algorithm M' above finds at least one seed σ that leads to acceptance. \square

Note that Corollary 3 can be proved *without* using pseudorandom generators. In fact, $\text{RL} \subseteq \text{L}^2$ as well as the more general result $\text{NL} \subseteq \text{L}^2$ about non-deterministic logspace were known long before and essentially work in the same manner as the repeated squaring of matrices that we discussed in the beginning of Lecture 14. However the proof above provides a “black-box derandomization” in that the algorithm can be seen as a black box and only the randomness is chosen from a pseudorandom distribution with small support. This is structurally more elegant, and many people believe that a possible proof of $\text{RL} = \text{L}$ would make use of pseudorandom generators. We will see in Lecture 16 that a pseudorandom generator for RL can indeed be used to prove $\text{RL} \subseteq \text{L}^{1.5}$, which constitutes a strict improvement on Corollary 3.

2.2 Nisan’s generator

We now have a look at Nisan’s generator, a different pseudorandom generator for branching programs. It came historically before the INW-generator and is related to it, but it is based on pairwise independent hash functions instead of expanders. Recall from Definition 2 in Lecture 5 that a family \mathcal{H} of pairwise uniform hash functions $h : \{0, 1\}^m \rightarrow \{0, 1\}^m$ is a family that satisfies $\Pr_{h \in \mathcal{H}}(h(x) = x' \wedge h(y) = y') = 2^{-2m}$ for all $x, y, x', y' \in \{0, 1\}^m$. We also noted in Lecture 5 that the family $\mathcal{H} = \{h_{a,b}\}$ of all linear functions $h_{a,b}(x) = a \cdot x + b$ over \mathbb{F}_{2^m} is such a family. Once hash functions h_1, \dots, h_i from the family are chosen, Nisan’s generator is defined as follows.

Definition 3. *Let m be a positive integer and let $h_1, \dots, h_i : \{0, 1\}^m \rightarrow \{0, 1\}^m$ be a sequence of (hash) functions. Then we define the Nisan’s generator $G_{h_1 \dots h_i} : \{0, 1\}^m \rightarrow (\{0, 1\}^m)^{2^i}$ as follows for all seeds $\sigma \in \{0, 1\}^m$:*

$$\begin{aligned} G_\emptyset(\sigma) &\doteq \sigma \\ G_{h_1 \dots h_i}(\sigma) &\doteq G_{h_1 \dots h_{i-1}}(\sigma) \circ G_{h_1 \dots h_{i-1}}(h_i(\sigma)) \end{aligned}$$

For example, if we only have a single hash function h , this yields $G_h(\sigma) = \sigma \circ h(\sigma)$. The interpretation in the case of branching programs is that we first divide the branching program into *blocks* of length m each. Then a random walk of length $2m$ in the branching program is specified by the assignment $\sigma_1 \circ \sigma_2$ to the first two blocks. In a uniform walk, these two assignments are independent and uniform; however, in a pseudorandom walk, the first assignment σ_1 is chosen uniformly at random, but the second step is taken to be $\sigma_2 = h(\sigma_1)$ in Nisan’s generator. The intuition is that, if h was drawn from a family of pairwise uniform hash functions, then the pseudorandom string looks random to any branching program.

Let us point out why we need to choose $m \geq \Omega(\log w)$ where w is the width of the branching program: If $\log w$ is larger than $6m$, then there is a simple branching program that can detect whether its input is an output of Nisan’s generator. For simplicity, we consider the case of three

hash functions h_1, h_2, h_3 ; here we have

$$G_{h_1, h_2, h_3}(\sigma) = \sigma \circ h_1(\sigma) \circ h_2(\sigma) \circ h_2(h_1(\sigma)) \circ h_3(\sigma) \circ h_3(h_1(\sigma)) \circ h_3(h_2(\sigma)) \circ h_3(h_2(h_1(\sigma))).$$

Since $\log w \geq 6m$, the branching program can hold six blocks entirely in its memory and make arbitrary decisions based on their value. Let a, b, c, d, e, f, g, h be the eight blocks above. Then the linear equations $e = h_3(a)$ and $f = h_3(b)$ can be used to determine the linear function h_3 since a and b are different with high probability. The branching program will then claim that the sequence is not random if $g = h_3(c)$ holds. Thus, in order for Nisan's generator to give something useful, the block length m must be $\Omega(\log w)$.

Nisan's generator is ϵ -pseudorandom for length r and width w branching programs in the following sense: When h_1, \dots, h_k for $k = \log r$ and $\sigma \in \{0, 1\}^m$ for $m = \Theta(\log r + \log w + \log(1/\epsilon))$ are drawn uniformly and independently from \mathcal{H} and U_m , respectively, then replacing the branching programs' uniform random input with the output of Nisan's generator incurs an error of at most ϵ in statistical distance. This result suffices to prove Corollary 3, but we will actually need a stronger version in Lecture 16 when we prove $\text{RL} \subseteq \text{L}^{1.5}$.

A careful analysis of Nisan's generator yields that we can actually divide the seed into two parts: In a first phase, the "outer" seed h_1, \dots, h_k is chosen uniformly at random. Then the function defined in Definition 3 is a procedure that stretches an "inner" seed σ of $m \sim \log r$ bits to $r \sim m \cdot 2^k$ pseudorandom bits; in order to compute it, only $O(m + k) = O(\log r)$ additional work space is required (where we do not count the work space required to store h_1, \dots, h_k). The stronger pseudorandomness statement is that, for all branching programs B , we have a very high probability that a uniformly sampled outer seed will instantiate the generator G_{h_1, \dots, h_k} such that it is ϵ -pseudorandom for B . We make this formal below.

Theorem 4 ([Nis92]). *Let B be any branching program of length $r = 2^k$ and width w and let $\epsilon > 0$. Then there is an $m \sim \log r + \log w + \log(1/\epsilon)$ such that*

$$\Pr_{h_1, \dots, h_k \sim \mathcal{H}} \left[G_{h_1, \dots, h_k} \text{ is } \epsilon\text{-pseudorandom for } B \right] \leq \frac{1}{\text{poly}(r)}.$$

Moreover, the following slightly stronger statement is true as well:

$$\Pr_{h_1, \dots, h_k \sim \mathcal{H}} \left[d_{\text{stat}} \left(B(s; G_{h_1, \dots, h_k}(U_m)), B(s; U_{2^k}) \right) \leq \epsilon \right] \leq \frac{1}{\text{poly}(r)}. \quad (7)$$

Recall that $B(s; D)$ refers to the distribution on the states that B reaches when its input is sampled from D . Furthermore, let us remark that G_{h_1, \dots, h_k} outputs $m2^k$ bits, but we actually only use 2^k of them; this is an insignificant technicality.

We leave it as an exercise for the reader to prove Theorem 4, it follows along the lines of the proof of Lemma 1; the only important difference is the bound on the corresponding third term (6). Instead of the expander mixing lemma, we use a mixing lemma for pairwise uniform hash functions. This is also the step that we get the concentration result from. We state this lemma and provide a proof sketch.

Lemma 5. *Let m be a positive integer, and let \mathcal{H} be a family of pairwise uniform hash functions $h : \{0, 1\}^m \rightarrow \{0, 1\}^m$. Furthermore, let $S, T \subseteq \{0, 1\}^m$. Then we have*

$$\Pr_{h \sim \mathcal{H}} \left[\left| \frac{|S||T|}{2^{2m}} - \frac{|\{s \in S \mid h(s) \in T\}|}{2^m} \right| < \epsilon \right] > 1 - 1/(\epsilon^2 \cdot 2^m).$$

Proof. Let X be the random variable $|\{s \in S | h(s) \in T\}|/2^m$. We now compute its expected value using the 0/1-indicator variable $X_s = I[h(s) \in T]$:

$$\mathbb{E}_{h \sim \mathcal{H}}[X] = \mathbb{E}_{h \sim \mathcal{H}} \left[\sum_{s \in S} X_s / 2^m \right] = \sum_{s \in S} \mathbb{E}_{h \sim \mathcal{H}}[X_s] / 2^m = \sum_{s \in S} |T| / 2^m = |S| \cdot |T| / 2^{2m}.$$

To prove the theorem, we need to prove concentration. Below, we first apply Chebyshev's inequality (Proposition 1 in Lecture 5), which requires us to bound the variance of X . Since $X = \sum_{s \in S} X_s / 2^m$ holds and the X_s are pairwise independent, we can apply Proposition 3 in Lecture 5. We get:

$$\begin{aligned} \Pr_{h \sim \mathcal{H}} \left[|\mathbb{E}[X] - X| \geq \epsilon \right] &\leq \frac{\text{Var}[X]}{\epsilon^2} = \frac{\sum_{s \in S} \text{Var}[X_s / 2^m]}{\epsilon^2} = \frac{\sum_{s \in S} \text{Var}[X_s] / 2^{2m}}{\epsilon^2} \\ &\leq \frac{\sum_{s \in S} \mathbb{E}[X_s] / 2^{2m}}{\epsilon^2} = \frac{\mathbb{E}[X] / 2^m}{\epsilon^2} \leq \frac{1}{\epsilon^2 2^m}. \end{aligned} \quad \square$$

References

- [INW94] Russell Impagliazzo, Noam Nisan, and Avi Wigderson. Pseudorandomness for network algorithms. In *Proceedings of the 26th Annual Symposium on Theory of Computing, STOC 1994*, pages 356–364, 1994.
- [Nis92] Noam Nisan. Pseudorandom generators for space-bounded computation. *Combinatorica*, 12(4):449–461, 1992.