CS59200-PRS: Pseudorandomness Sept 10, 2025

Lecture 5: PRG for Derandomization and Branching Programs

Lecturer: Wei Zhan *Scribe: Hongao Wang*

1 PRG for Derandomization

1.1 Derandomizing BPP

We have already seen several construction of pseudorandom generators (PRGs) in the previous lectures that fools very specific classes of functions, and several useful tools to analyze them. In this and the following many lectures, we will focus on the application of pseudorandom generators for derandomization.

First, we will discuss the goal of derandomization. Vaguely speaking, we want to show that every randomized algorithm can be converted to a deterministic one with only a small overhead in time complexity. More precisely, we want to show that $\mathsf{BPP} = \mathsf{P}$. Here BPP is the class of languages that can be decided by a randomized polynomial-time algorithm with error probability at most $1/3$ for all inputs. Formally, we have the following definition of BPP.

Definition 1. *A language $L \subseteq \{0,1\}^*$ is in BPP if there exists $m(n) = \mathrm{poly}(n)$, where n is the length of the input, and a deterministic polynomial-time algorithm A such that for every input x,*

- *If $x \in L$, then $\mathop{\mathbb{E}}_{r\sim\{0,1\}^{m(n)}}[A(x,r)] \geq 2/3$.*
- *If $x \notin L$, then $\mathop{\mathbb{E}}_{r\sim\{0,1\}^{m(n)}}[A(x,r)] \leq 1/3$.*

Therefore, to derandomize BPP, it suffices to find a family of pseudorandom generator $G : \{0,1\}^{\ell(n)} \to \{0,1\}^{m(n)}$ such that:

- G ε-fools every polynomial-time algorithm $A(x,r)$ with every input x, for some small constant ε, say $\varepsilon = 0.1$;
- $\ell(n) = O(\log n)$;
- G can be deterministically computed within polynomial time in n, or exponential time in its input length.

In this case, we can go through all possible $2^{\ell(n)}$ seeds s to simulate the behavior of $A(x, G(s))$ in polynomial time. However, this goal is impossible to achieve, as we showed in the first lecture that such PRG must have seed length $\omega(\log n)$ (otherwise there exists a polynomial time algorithm that computes

$$A(x, r) = \begin{cases} 1 & \text{if } r \in \text{range}(G) \\ 0 & \text{otherwise} \end{cases}$$

which is not fooled by G).

This is fine because we don't really need a universe PRG that fools every polynomial-time algorithm. It would be enough if for every polynomial-time algorithm A, we can find a pseudorandom generator G_A that specifically fools A, with seed length $\ell(n) = O(\log n)$ and that G_A can be computed in polynomial time. This is called *targeted PRG*. We have already seen some targeted PRGs, for instance, the one for the approximate MAX-CUT algorithm, which turn out to be pairwise independence.

But we also cannot just enumerate all polynomial-time algorithms and design a PRG targeted for each one individually. We still want some general construction that works for a group of polynomial algorithms. Therefore, our actual goal is that for every c, find a PRG G_c that fools every polynomial-time algorithm A with time complexity at most n^c, with $\ell(n) = O(\log n)$ and G_c can be computed in polynomial time. Notice that the counterexample above would not prevent this possibility as the running time of the algorithm A there depends on G.

1.2 Derandomizing BPL

We have some strong evidence that these PRGs G_c exists, by the Nisan-Wigderson hardness vs. randomness program, but that is left for another day. For now, we will discuss the analogous problem of derandomizing algorithms with *logspace*, on which we have even more progress. We will first define the class BPL.

Definition 2. *A language $L \subseteq \{0,1\}^*$ is in* BPL *if there exists $m(n) = \text{poly}(n)$, where n is the length of the input, and a deterministic logspace algorithm A with one-way access to r, such that for every input x,*

- *If $x \in L$, then $\mathop{\mathbb{E}}_{r \sim \{0,1\}^{m(n)}}[A(x, r)] \geq 2/3$.*
- *If $x \notin L$, then $\mathop{\mathbb{E}}_{r \sim \{0,1\}^{m(n)}}[A(x, r)] \leq 1/3$.*

Remark. We require A to have only one-way access to r, so that it is consistent with the actual behavior of a logspace probabilistic Turing machine: The random coins it uses can be modeled by a stream of random bits, and to access previously used random coins it must store them. This way we avoid the problem in our definition where the random bits r are part of the input and does not count towards the space usage of A.

Note that if we instead remove this restriction and allow A to have full access to r, the corresponding class is then called $\mathsf{BP} \cdot \mathsf{L}$ (Here $\mathsf{BP}\cdot$ works as an operator and can be followed by any complexity class, by just requiring A to be in that class). It is not known to be the same as BPL; in fact, it is not even known to be in P.

Now to derandomize BPL, it also suffices to find a PRG $G : \{0,1\}^{\ell(n)} \to \{0,1\}^{m(n)}$ such that:

- G 0.1-fools every logspace algorithm $A(x, r)$ that has one-way access to r with every input x;
- $\ell(n) = O(\log n)$;
- G can be deterministically computed within logspace in n, or linear space in its input length.

However, this is again impossible to achieve. The counterexample at the start does not exactly work, but we can modify it a bit so that it can be computed with one-way access to r. For instance, we let $A(x, r) = 1$ if the first 2ℓ bits of r are the same with the first 2ℓ bits of any string in range(G), and 0 otherwise. Then A can be computed in logspace by simply storing the first 2ℓ bits of r, but

$$\mathop{\mathbb{E}}_{r\sim\{0,1\}^m}[A(x,r)] \le 2^{-\ell}, \text{ while } \mathop{\mathbb{E}}_{s\sim\{0,1\}^\ell}[A(x,G(s))] = 1.$$

Therefore, we need to relax our goal a bit as previously. Similarly, the actual goal is that for every c, find a PRG G_c that fools every log-space algorithm A with one-way access to r with space complexity at most $c \log n$, with $\ell(n) = O(\log n)$ and G_c can be computed in logspace.

Then we are going to formally discuss the computation model with bounded space, and introduce *Read-Once Branching Program* (ROBP).

2 Log Space Turing Machine and Read-Once Branching Program (ROBP)

The most common way to define space-bounded computation is via Turing Machines with bounded work tape. Formmaly speaking, A SPACE(s) Turing Machine is a Turing Machine with three tapes: a read-only input tape contains the input x, a read-write work tape with only $O(s)$ cells, and a write-only output tape.

Then we will introduce Read-Once Branching Program (ROBP), which has two parameters: width w and length n. A ROBP is a directed acyclic graph with $n + 1$ layers of vertices, where each layer has at most w vertices. The first layer is the starting layer with a specific initial vertex. Each vertex in the first n layers has exactly two outgoing edges, one labeled by 0 and the other labeled by 1. Each vertex in the last layer has only one outgoing edge to either 0 or 1, which is the output. See Figure 1 for an example. This is a function

$B : \{0,1\}^n \to \{0,1\}$, where for every input $x \in \{0,1\}^n$, we start from the first layer and follow the edges according to the bits of x until we reach the last layer. The label of the outgoing edge of the vertex in the last layer is the output $B(x)$.

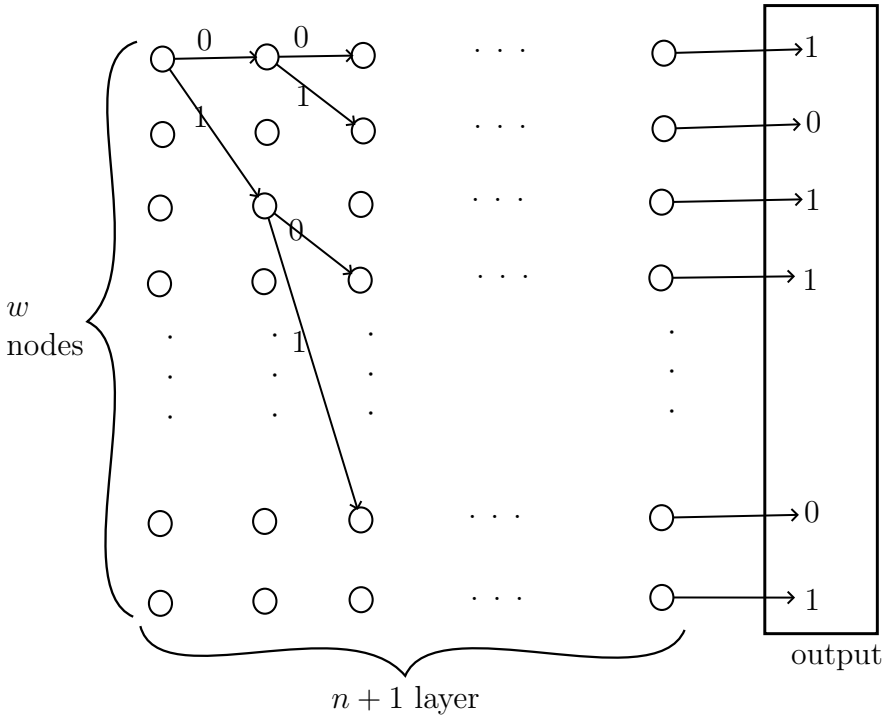

Figure 1: An example of ROBP.

Then we want to claim that any SPACE(s) Probabilistic Turing Machine can be simulated by a ROBP with width $w = 2^{O(s)}$ and length $n = 2^{O(s)}$ that is SPACE(s)-uniform (that a deterministic SPACE(s) Turing machine can output the description of the ROBP). As the number of possible configurations of the Turing Machine (excluding the input) is at most $2^{O(s)}$, we can represent each configuration as a vertex in each layer of the ROBP. Then each layer of the ROBP represents the configurations after reading one more bit of the random string. Therefore, there are at most $n = 2^{O(s)}$ layers and each layer has at most $w = 2^{O(s)}$ vertices. However, we should notice that we need to know the input in advance to construct the ROBP, as the transition function of the Turing Machine depends on the input.

Therefore, $A(x, r)$ can be represented as $B_x(r)$ with width-poly(n) and length-poly(n), where B_x is the ROBP constructed according to the input x. Hence, our goal becomes to fool the ROBP B_x. Formally, we have the following theorem.

Theorem 1. *Suppose that for every n, w there exists $\ell = O(\log(nw))$ and a PRG $G_{n,w} : \{0,1\}^\ell \to \{0,1\}^n$ that 0.1-fools every ROBP of width w and length n, and $G_{n,w}$ can be computed in space $O(\ell)$. Then* $\mathsf{BPL} = \mathsf{L}$.

2.1 Simulating ROBP by Matrix Powering

Here we share some common recipes for simulating a ROBP B with width w and length n. The goal is to (approximately) compute $\mathop{\mathbb{E}}_{r\sim\{0,1\}^n}[B(r)]$.

First, we can in general assume the transition function (edges in the ROBP) is time-invariant. This is because for every ROBP with transition functions $t_i : [w] \times \{0,1\} \to [w]$ at the i-step, we can construct a equivalent ROBP with width $(n+1)w$ and a new transition function $t : [n] \times [w] \times \{0,1\} \to [n] \times [w]$ as

$$t((i,q),b) = (i+1, t_i(q)).$$

Then the transition function t is time-invariant.

We can define the transition matrices $M_0, M_1 \in \{0,1\}^{w\times w}$ of the ROBP as

$$M_b[q',q] = \mathbb{1}_{t(q,b)=q'}.$$

Here $M_b[q',q]$ is the entry of the matrix M_b at the row q' and column q. Then we can write $B(x) = \langle v_{out}, M_{x_n} M_{x_{n-1}} \cdots M_{x_1} v_{in}\rangle$, where v_{in} is the indicator vector of the start vertex, and v_{out} is the indicator vector of the accepting states. Therefore, we have:

$$\begin{aligned}
\mathop{\mathbb{E}}_{r\sim\{0,1\}^n}[B(r)] &= \mathop{\mathbb{E}}_{r\sim\{0,1\}^n}[\langle v_{out}, M_{r_n} M_{r_{n-1}} \cdots M_{r_1} v_{in}\rangle] \\
&= v_{out}^{\mathrm{T}} \mathop{\mathbb{E}}_{r\sim\{0,1\}^n}[M_{r_n} M_{r_{n-1}} \cdots M_{r_1}] v_{in} \\
&= v_{out}^{\mathrm{T}} \left(\frac{M_0+M_1}{2}\right)^n v_{in}.
\end{aligned}$$

Thus, derandomizing ROBP is equivalent to computing specific entry of the matrix power M^n for $M = (M_0+M_1)/2$ by deterministic algorithms. A trivial algorithm is to compute the matrix power directly and this shows that $\mathsf{BPL} \subseteq \mathrm{P}$. However, there is a recursive algorithm that uses only $O(\log^2 n)$ space to compute this matrix power, which shows that $\mathsf{BPL} \subseteq \mathsf{L}^2 = \mathrm{SPACE}(\log^2 n)$. The idea is to use the following recursion: for every $i,j < w$, $M^n[i,j] = \sum_{k=1}^{n} M^{n/2}[i,k] \cdot M^{n/2}[k,j]$. Then we can recursively compute $M^{n/2}[i,k]$ and $M^{n/2}[k,j]$ for every k. The depth of the recursion is $O(\log n)$ and each level of the recursion uses $O(\log n)$ space to store $i,j,k < w$. Therefore, the total space complexity is $O(\log n \log w)$, which is $O(\log^2 n)$. Thus, we got the space bound as disired.