CS 880: Pseudorandomness and Derandomization 3/18/2013

Lecture 16: RL is in $L^{1.5}$

Instructors: Holger Dell and Dieter van Melkebeek Scribe: Lubos Krcal

DRAFT

In Lecture 15, we have seen two pseudorandom generators that use a seed of length $\log^2 n$ and produce pseudorandomness for RL-machines. This time, we will see that every randomized logspace machine with bounded error can be simulated by a deterministic machine that uses $\log^{1.5} n$. All of these results can be easily transferred to BPL-machines, where the error can be two-sided.

1 The idea

Recall that Nisan's generator consisted of two phases: In the first phase, a logarithmic number $k \sim \log n$ of hash functions is chosen; to sample a single hash function, a logarithmic number of random bits are required, and thus, the outer seed consists of $O(k \log n) = O(\log^2 n)$ random bits. In the second phase, an inner seed of logarithmic length $m \sim \log n$ chosen and the output is computed using at most a logarithmic amount of additional work space. It is this discrepancy between the logarithmic space requirement of the second phase and the quadratically logarithmic length of the outer seed length that we will be able to trade off with each other to get an overall seed length of $O(\log^{1.5} n)$.

A natural idea for decreasing the seed length of the outer seed is to simply choose fewer hash functions and reuse them a few times. In particular, we want to choose $k << \log n$ so that $k \log n << \log^2 n$. We will see below that we can in fact do so if we are willing to pay for it with an increased space requirement of the inner phase. In the construction and analysis below, the space requirement for the inner phase will eventually go up to $\frac{\log n}{k} \cdot \log n$. The two quantities $k \log n$ and $\log^2 n/k$ embody the trade off mentioned above, and they are simultaneously minimized if $k \sim \sqrt{\log n}$, which is how the bound of $\log^{1.5} n$ comes about.

We will first attempt to pick a single hash function, fail, and successively try to make this idea work.

2 Attempt 1: Pick only a single hash function

Naïvely, we might pick a single hash function $h_1 : \{0,1\}^m \to \{0,1\}^m$ from the distribution $\mathcal{H}$ and set $h_2 = \cdots = h_k = h_1$. This is not so unreasonable since the concentration result for Nisan's generator, Theorem 4 in Lecture 15, implies that $G_{h_1} : \{0,1\}^m \to \{0,1\}^{2m}$ is with high probability a pseudorandom walk generator for a given branching program B: That is, it only requires m bits of seed but produces randomness that B cannot distinguish from $2m$ uniform bits. At least on the surface, this is similar to the derandomized squaring based on expander graphs that we discussed in Lecture 14. The idea to simply apply the same derandomized squaring k times to get $r \leq 2^k m$ pseudorandom bits out then directly leads to our attempt to choose all hash functions to be equal.

More formally, we define the *derandomized square* of a branching program $B : [r-1] \times [w] \times \{0,1\}^m \to [r] \times [w]$ with respect to a function $h : \{0,1\}^m \to \{0,1\}^m$ to be the branching program $B_h : [r-2] \times [w] \times \{0,1\}^m \to [r] \times [w]$ where $B_h(s;\sigma) \doteq B(s; G_h(\sigma))$ holds for all states $s \in [r-2] \times [w]$. Since $r = 2^k$ and $G_h(\sigma)$ is $2m$ bits long, we can drop all states $s = (i, *)$ for odd numbers i; thus, the length of the derandomized square is $r/2$, half of the length of B, and its width is again at most w since the computation of $G_h(\sigma)$ is hidden in the non-uniformity of the branching program model (this is why we make the branching program read a block of length m in each step, which we can do without loss of generality since any pseudorandom generator for this model will also be pseudorandom for the usual model with block length 1).

An immediate consequence of Nisan's pseudorandomness guarantee is that the derandomized square of a branching program has a similar behavior as the original, which we formalize below.

Lemma 1. *For all k, w, and $\epsilon > 0$, there is an $m \sim \log r + \log w + \log(1/\epsilon)$ such that, for every branching program $B : [r-1] \times [w] \times \{0,1\}^m \to [r] \times [w]$ of length $r = 2^k$ and width w and for all states s of B, we have*

$$\Pr_{h_1 \sim \mathcal{H}} \left[d_{\text{stat}}\Big(B_h(s; U_m) \ , \ B(s; U_{2m}) \Big) < \epsilon \right] \geq 1 - \frac{1}{\text{poly}(r)} \, . \tag{1}$$

Note that the distributions in (1) are over states two layers after s.

If we iterate the derandomized square process with hash functions $h_1, \ldots, h_k$ sampled from $\mathcal{H}$ independently, Lemma 1 implies that $B_{h_1,h_2,\ldots,h_k}$ is $k\epsilon$-close to the original branching program, except with probability at most $k/\text{poly}(r) = 1/\text{poly}(r)$. Another reason why this is true is the fact that $B_{h_1,\ldots,h_k}(s;\sigma) = B(s; G_{h_1,\ldots,h_k}(\sigma))$ holds, and Nisan's generator has the stated property.

In light of Lemma 1, it is tempting to use the same hash function $h_1 = \cdots = h_k$. However, it is not hard to see that Nisan's generator $G_{h_1,\ldots,h_1}$ fails to be pseudorandom in this case since there exists a simple distinguisher D that can tell the generator's output apart from a uniform distribution: For example, if $k = 2$, we have $G_{h_1,h_1}(\sigma) = \sigma \circ h_1(\sigma) \circ h_1(\sigma) \circ h_1(h_1(\sigma))$. A branching program D of width two can then store the $(m+1)$st bit and compare it to the $(2m+1)$st bit. If they are equal, the branching program will guess that the distribution is not uniform. Indeed, in the uniform case the bits are equal with probability $\frac{1}{2}$ whereas they are always equal under the output distribution of G_{h_1,h_1}. In other words, we have $d_{\text{stat}}(D_{h_1,h_1}(s;U), D(s;U)) \geq 1/2$ for *all* functions h_1 when s is the start state, and (1) cannot hold. Therefore, we have to do something slightly more clever if we are to reuse the same hash function.

2.1 Attempt 2: Decoupling by Rounding

We have seen in the last section that we cannot simply re-use the same hash function in Nisan's generator. Formally, the reason that Lemma 1 fails to give a guarantee for the branching program $B' = B_{h_1}$ after the first iteration is that the second hash function h_2 is assumed to be independent if we were to reapply Lemma 1, but we don't do that since we chose it to be equal to h_1. However, if we can start from B_{h_1} and modify it a little bit to get to a related B' that is *independent* from h_1, then we can pick h_1 such that B_{h_1} is a good approximation for B and *simultaneously* B'_{h_1} is a good approximation for B', which would enable us to hold on to the idea of choosing the same hash function in each derandomized squaring step. This "modification" of B_{h_1} will no longer work in a black-box way, but recall that we have the relaxed goal to derandomize B in as little space as possible in a possibly non-black-box way. This means that we are given B as *input* to a Turing

machine and we are allowed to transform it arbitrarily; we first transform it into B_{h_1} and then decouple it from h_1 to obtain B'.

To see what good candidates for this "decoupled" B' could be, note that B_{h_1} *is* a good approximation for B, so B' should also be something close to B that is, however, independent from h_1. We cannot choose B itself since it is not clear how to get from B_{h_1} back to B, and in fact, some information about B may have gotten lost in B_{h_1}. To get a clue, let us look back at (1) and only consider the good case for h_1. Then the 1-norm interpretation of the statistical distance states that $\sum_{t\in\text{next even layer}} |\Pr(B_h(s;U)=t) - \Pr(B(s;U)=t)| < 2\epsilon$ holds for all states s in an even layer. This implies that $\Pr(B_h(s;U)=t)$ and $\Pr(B(s;U)=t)$ are equal up to an additive error of at most 2ϵ. In particular, it seems very likely that the binary expansions of these two numbers coincide in their first $M = \log(1/\epsilon)/100$ most significant bits, and that the only part of the probabilities that depends on h is in the lower-order bits. It is then natural to cut the lower-order bits off, which simply corresponds to rounding down to an integer multiple of 2^{-M}. Thus, we can define $B' \doteq \lfloor B_h \rfloor_M$ such that the probabilities are rounded down to the first M positions. The hope is that the probabilities and B' itself will be independent from h. Moreover, when B_h is given, a description of B' can be computed in space $O(m + \log w + \log r + \log(1/\epsilon))$. We leave this fact as an exercise.

Exercise 1. *There is a deterministic algorithm that, when given a branching program B of length r, block length b, and width w, and given a number $\epsilon > 0$ so that $M \doteq \log(1/\epsilon)/100 \le m$, computes a branching program $\lfloor B \rfloor_M$ of length r, block length m, and width w that satisfies*

$$\Pr(\lfloor B \rfloor_\epsilon(s;U_m)=t) = \lfloor \Pr(B(s;U_b)=t) \rfloor_M \,, \tag{2}$$

for all states s in the i-th layer and all states t in the $(i+1)$-st layer. Moreover, the algorithm uses space $O(m+b+\log(wr/\epsilon))$ and the program $\lfloor B \rfloor_M$ depends only on the right-hand side of (2) (and not on the structure of B).

The rounded program approximates the original in the following sense: For all states s,

$$d_{\text{stat}}\Big(\lfloor B \rfloor_M(s;U_m), B(s;U_b)\Big) \le w2^{-M} = O(\epsilon w)\,. \tag{3}$$

So our second attempt for obtaining a derandomization of B proceeds as follows: We are given B of length $r = 2^k$ and with block length m as input, and we pick a single hash function $h \sim \mathcal{H}$. Then we compute the branching program P_k defined iteratively as follows:

$$\begin{aligned} P_0 &\doteq B \\ P_{i+1} &\doteq \lfloor B_h \rfloor_M \,. \end{aligned}$$

The block length of all these programs is $m \sim \log r$, and the length of P_i is $r/2^i$. Therefore, P_k has length one and can be fully derandomized by brute force. By Lemma 1, we know that one step in B_h is ϵ-close to two steps in B. So we can compare the above sequence to the "truly random" sequence:

$$\begin{aligned} T_0 &\doteq B \\ T_{i+1} &\doteq \lfloor B^2 \rfloor_M \,. \end{aligned}$$

In this sequence, the block length of B^2 is $2m$, but the rounding cuts the block length back to m and we loose some precision in the lower-order bits. In the truly random sequence, we iteratively apply Exercise 1 and get that T_k and B^{2^k} are $O(kw\epsilon)$-close in statistical distance.

Recall that our goal is to apply Lemma 1 *once* to find an h that bounds the distance between P_i and T_i for *all* $i \in [k]$ simultaneously. This would indeed be possible by a simple application of the union bound if the rounding made the P_i's independent from h. We have not quite achieved that goal yet: Consider the case in which B transitions from s to a state t two layers after s with probability exactly $p = 7 \cdot 2^{-M}$. Then Lemma 1 guarantees that, for most choices of h, the program B_h transitions from s to t with probability 2ϵ-close to p; this probability could be $< p$ or $\geq p$. In the former case, P_1 would round this probability down to $6 \cdot 2^{-M}$, and in the latter case it would get rounded down to $7 \cdot 2^{-M}$. Note that M was chosen so that $\epsilon = 2^{-100M}$, so only these two cases can occur since the error is much smaller than the rounding intervals. The problem is that the program P_1 still depends on h since we cannot rule out that some h with lead to the larger number and some will lead to the smaller one.

2.2 Attempt 3: Decoupling by Rounding + Random Translation

To circumvent the problem that the branching program may depend on h after the rounding step, note that there are only few problematic values for the probability p from above: The problematic cases of p occur in the 2ϵ-neighborhood around integer multiples of 2^{-M}. In each interval $[i2^{-M}, (i+1)2^{-M}]$, the total measure of these problematic boundary cases is $2\epsilon/2^{-M} = 2^{-100M+1+M} \leq 2^{-98M}$ by our choice of $M = \log(1/\epsilon)/100$. By choosing $\epsilon = 1/\operatorname{poly}(r)$, we can make this measure polynomially small in r.

In light of the small measure of problematic cases, we can just add some random noise to the transition probabilities. The noise will be such that it is unlikely that we will be in the problematic region, and so that the error does not go up significantly. The details can be worked out in the following exercise.

Exercise 2. *There is a deterministic algorithm that, when given a branching program B of length r, block length m, and width w, and given numbers $\epsilon > 0$ and $\delta \in [0,1] \cap 2^{-100M} \cdot \mathbb{N}$ so that $M \doteq \log(1/\epsilon)/100 \leq m$, computes a branching program $\lfloor B \rfloor_M$ of length r, block length m, and width w that satisfies*

$$\Pr(\lfloor B - \delta \rfloor_\epsilon(s; U_m) = t) = \lfloor \Pr(B(s; U_b) = t) - \delta \rfloor_M \,, \tag{4}$$

for all states s in the i-th layer and all states t in the $(i+1)$-st layer. Moreover, the algorithm uses space $O(m + b + \log(wr/\epsilon))$ and the program $\lfloor B - \delta \rfloor_M$ depends only on the right-hand side of (4) (and not on the structure of B).

The rounded program approximates the original in the following sense: For all states s,

$$d_{\mathrm{stat}}\Big(\lfloor B - \delta \rfloor_M(s; U_m), B(s; U_b)\Big) \leq w(2^{-M} + \delta) = O((\epsilon + \delta)w) \,. \tag{5}$$

Finally, if h is a good hash function for B in the sense of (1), we want to have that $\lfloor B_h - \delta \rfloor$ is the same regardless of the choice of which good hash function we chose. Note that there are at most $r \cdot w = \operatorname{poly}(r)$ transition probabilities between subsequent layers of B_h, and therefore, the probability that there exists a transition probability that gets shifted into a bad region by picking δ uniformly at random is at most $1/\operatorname{poly}(r)$, where the degree of the polynomial can be made as large as needed to apply the union bound over all transition probabilities.

We are done with our third attempt, and this algorithm actually works. Let's analyze the space requirement: The space for a single iteration is $\sim logr$ plus the space required to store the hash function h, which is $\log m \sim \log(1/\epsilon) \sim \log r$. So over all k iterations, we require space $k \cdot \log r \sim \log^2 r$, which is no improvement over Nisan's generator.

2.3 Final algorithm

To reduce the space requirements further, we need to decrease the number of iterations and rounding steps required, and we will do so by increasing the number of hash functions used. In particular, our final algorithm proceeds as follows:

1. The input is a branching program B of length $r = 2^k$ and width w.
2. We set $\epsilon = 1/\operatorname{poly}(r)$, $m = O(\log(1/\epsilon))$, and $M = \log(1/\epsilon)/100$.
3. We sample $\sqrt{k}$ hash functions $h_1, \dots, h_{\sqrt{k}} \sim \mathcal{H}$.
4. We set $P_0 = B$.
5. For each i from 0 to $\sqrt{k} - 1$, we sample δ_i uniformly at random and compute
$$P_{i+1} \doteq \left\lfloor (P_i)_{h_1, h_2, \dots, h_{\sqrt{k}}} - \delta_i \right\rfloor_M,$$
via Exercise 2.
6. We cycle through all seeds of length m to compute the acceptance probability of $P_{\sqrt{k}}$.

Note that P_0 has block length 1 initially, and we silently pad it to block length m without loss of generality. The successive application of Lemma 1 with $\sqrt{k}$ independent hash function decreases the length of the program by a factor of $2^{\sqrt{k}}$, so all programs P_i have block length m and length $r/2^{i \cdot \sqrt{k}}$. Importantly, computing this composition and applying Exercise 2 only requires space $\sim \log r$ since the hash functions are fixed globally. In the end, $P_{\sqrt{k}}$ has length one, and cycling through all seeds of length m can be done in space $O(\log r)$. Thus, we need space $m \cdot \sqrt{k}$ to sample the hash functions once, and we need space $\log r \cdot \sqrt{k}$ to iteratively compute $P_{\sqrt{k}}$. By the choice of parameters, this works out to be $\sim \log^{1.5} r$.

Theorem 2 ([SZ99]). $\mathrm{RL} \in \mathrm{L}^{1.5}$.

References

[SZ99] Michael Saks and Shiyu Zhou. $\mathrm{BP_HSPACE(S)} \subseteq \mathrm{DSPACE(S^{3/2})}$. *Journal of Computer and System Sciences*, 58(2):376–403, 1999.