Spring 5-31-2023

Exploring Improvements to Space-Bounded Derandomization from Better Pseudorandom Generators

Boxian Wang
Dartmouth College, boxian.wang.23@dartmouth.edu

Exploring Improvements to Space-Bounded Derandomization from Better Pseudorandom Generators

Boxian Wang

Advisor: Amit Chakrabarti

A thesis presented for the degree of

Bachelor of Arts

Department of Computer Science

Dartmouth College

June 2023

Abstract

Saks and Zhou [SZ99] used Nisan's PRG [Nis90] in a recursive manner to obtain $\mathbf{BPL} \subseteq \mathbf{L}^{3/2}$. We describe how this framework could be generalized to use arbitrary PRGs following Armoni's sampler idea [Arm99]. We then give a theorem relating the seed length of a better PRG to the implied improvements in derandomizing **BPL**. Recently, Hoza [Hoz21a] used Armoni's PRG [Arm99] in the Saks-Zhou framework to obtain an even better derandomization. We describe the construction of Armoni's PRG and conjecture that by using basic components other than extractors, parameters in that construction could be improved. Under some assumptions, we calculate the extent to which such parametrical improvements could produce better derandomization of **BPL**, noting that proving results better than $\mathbf{BPL} \subseteq \mathbf{L}^{4/3}$ requires ideas beyond our current suite of techniques.

Contents

Chapter 1

Introduction

The use of randomness has become one of the most important approaches in algorithm design. However, like space and time, randomness itself could be viewed as a precious resource that algorithm designers should aim to conserve. From a theoretical perspective, it is natural to ask if computation containing randomness has more intrinsic capabilities than deterministic computation.

The quest to reduce the need for randomness algorithm while preserving space/time efficiency is called *derandomization*, and ultimate goal of derandomization is to find a way to simulate randomized with deterministic computation within the same space/time bounds. While derandomizing time-bounded computation is widely believed to be still far out of reach, there has been considerable optimism for derandomizing space-bounded computation.

Pseudorandom generators (PRGs) are important tools of derandomization. Conceptually, a PRG stretches a short random seed to a long pseudorandom seed, which can then be used in randomized computation as if it was truly random. Using a PRG thus drastically saves the amount of randomness needed. Nisan designed a PRG with seed length $O(\log^2 n)$ [Nis90], which is only a $\log n$ factor away from the optimal (logarithmic) seed length.

However, using Nisan's PRG directly only gives us $\mathbf{BPL} \subseteq \mathbf{L}^2$; Saks and Zhou [SZ99] later gave a non-trivial way of using Nisan's PRG recursively to obtain $\mathbf{BPL} \subseteq \mathbf{L}^{3/2}$. However, their original paper relied heavily on the particular structure of Nisan's PRG

and is thus not immediately applicable to other PRGs; Armoni [Arm99] later generalized the Saks-Zhou framework to use arbitrary PRGs by composing a sampler. More recently, Hoza [Hoz21a] combined Armoni's PRG [Arm99] with error-reduction technique [CDR$^+$21] enabled by *weighted pseudorandom generators* (WPRGs), a generalization of PRG, to obtain an even better derandomization: $\mathbf{BPL} \subseteq \mathbf{DSPACE}\left(\frac{\log^{3/2} N}{\sqrt{\log \log N}}\right)$. For a more on recent advancements, see Hoza's excellent survey paper [Hoz22].

In this paper, we expand on these previous results to discuss the relationship between better (W)PRGs and improvements to derandomization of **BPL** using the current suite of techniques. First, we discuss Saks and Zhou's technique and explain how it can be applied to an arbitrary PRG. We show what improvements over the current derandomization might be possible, given that we have a better PRG that improves upon Nisan's seed length. Second, we discuss the construction of Armoni's PRG. We then calculate, under some assumptions, the extent to which better parameters in Armoni's construction improves derandomizing **BPL**.

Chapter 2

Derandomization and Pseudorandom Generators

2.1 Space-Bounded Computation

We start by introducing the model of space-bounded computation.

Definition 2.1.1. *For $S(N) \geq \log N$,* **DSPACE**(S) *is the set of languages decidable by a deterministic Turing machine using a read-only input of length n and a work tape of length at most $O(S(n))$.*

We are particularly interested in the case where S is logarithmic, which prompts the following definition.

Definition 2.1.2. $\mathbf{L} = \mathbf{DSPACE}(\log N)$

On the other hand, we would like to introduce randomness into space-bounded computation. However, because we are interested in space complexity, special care needs to be taken to insure the Turing machine behave as expected.

Definition 2.1.3. *For $S(N) \geq \log N$,* **BPSPACE**(S) *is the set of languages decidable by the following type of randomized Turing machines. It receives a read-only input of length N, and have* ***one–way*** *access to a tape with uniformly random bits. It uses a work tape of length at most $O(S(N))$, and is guaranteed to always to halt. Here "decid-*

ing" means if the input is in the language, the Turing machine accepts with probability $\geq 2/3$. *If not, it rejects with probability* $\geq 2/3$.

Note the requirements that the randomness access be one-way and the machine always halt. Forgoing these two restrictions gives us classes of languages that are not trivially equivalent to the what we defined to be **BPSPACE**.

Similarly, we are interested in the case where $S = \log N$.

Definition 2.1.4. $\mathbf{BPL} = \mathbf{BPSPACE}(\log N)$.

It has been conjectured that **BPL** and **L** are equivalent in power, i.e. $\mathbf{BPL} = \mathbf{L}$. This problem is considered to be much more accessible than other long-standing conjectures in complexity theory, such as **P** vs. **NP**, where we could not even prove **NP**-complete problems require more than linear time. It is also thought to be easier than its sibling conjecture $\mathbf{BPP} \stackrel{?}{=} \mathbf{P}$, as the existence of a strong pseudorandom generator for derandomizing **BPP** implies some very strong circuit lower bound results [Hoz21b] considered to be far out of reach, whereas pseudorandom generators for **BPL** do not have such unfortunate implications.

A prominent feature of space-bounded Turing machines is that their behaviors can be understood through *configuration graphs*. A configuration graph is simply a directed graph where each vertex is a configuration of the Turing machines, and each configuration is connected to all possible next configurations by directed edges. Many types of space-bounded computation can be thought of as graph algorithm problems. For instance, **NL** is defined to be the class of problems solvable by a *nondeterministic* log-space Turing machine, and path-finding between two nodes of a directed graph (the starting and accepting configurations) is **NL**-complete. Complexity classes like **SL** also originates from finding paths in (undirected) configuration graphs.

When randomness is present, one configuration can lead to two different configurations, depending on the random bit read. Note the requirement that the machine always halt disallows loops in the configuration graph, so it is actually a DAG. However, simulating **BPL** requires solving a much more difficult graph problem than just finding a path – it requires us to calculate whether more than 2/3 of the paths from

the starting vertex lead to the accepting vertex.

Fix the input to the Turing machine so that we have a fixed configuration graph. If we further label each edge with the associated random bit, this is now more reminiscent of a finite state machine (FSM), where the states are configurations and the inputs are the random bits.

In practice, it is often beneficial to consider **read-once branching programs** (ROBPs), a generalization of FSM, when studying randomized space-bounded computation.

Definition 2.1.5. *A (w, n)-ROBP is a directed graph with $n + 1$ layers of vertices $V_0, \ldots, V_n$, where each layer V_i contains at most w vertices. Each vertex in layer V_i has two outgoing edges to vertices in layer V_{i+1}, labeled 0 and 1. A starting vertex $v_0 \in V_0$ and a subset of accepting vertices $V_{acc} \subseteq V_n$ are designated.*

An ROBP defines a decision function $f : \{0,1\}^n \to \{0,1\}$ in the following fashion. On input $x = x_1 \ldots x_n \in \{0,1\}^n$, the ROBP starts at v_0, follows the edge corresponding to each bit read, and accepts or rejects based on the final vertex it arrives at.

In the above definition, w is called the width of the ROBP and n the length. In the context of a **BPSPACE** machine, w would be equal to the number of configurations and n would be equal to the number of random bits needed. But our requirement that the machine always halt means that n is at most w, since there are only w nodes in the acyclic configuration graph, whence a traversal takes at most w steps. For a general randomized algorithm, it is useful to look at ROBPs whose width is close to its length ($w = \text{poly}(n)$). Thus we have the following proposition:

Proposition 2.1.6. *Let T be a randomized log-space machine, and let N be the length of its input. Then there exists $n, w = \text{poly}(N)$ such that, for every input x and random seed y, there exists a (w, n)-ROBP f_x where $f_x = T(x, y)$. Furthermore, f_x can be constructed in $O(\log N)$ space, if n is constructible in $O(\log N)$ space.*

However, we will later see that studying cases where $n \ll w$ is useful as well.

2.2 Pseudorandom Generators

Pseudorandom generators (PRGs) are powerful tools of derandomization. Intuitively, a PRG stretches a short uniformly random seed to a longer output, such that the output is almost indistinguishable from a true uniformly random seed by a specific class of functions. When this happens, we say that the PRG "fools" that class of functions.

Definition 2.2.1. *Let $\mathcal{F}$ be a class of functions $f : \{0,1\}^n \to \{0,1\}$. An ε-PRG is a function $G : \{0,1\}^r \to \{0,1\}^n$ such that for every $f \in \mathcal{F}$,*

$$\left| \mathop{\mathbb{E}}_{x \in \{0,1\}^r} [f(G(x))] - \mathop{\mathbb{E}}_{x \in \{0,1\}^n} [f(x)] \right| \leq \varepsilon.$$

Here r is the seed length of G.

We are chiefly interested in PRGs for ROBPs, although PRGs for other models of computation have also been studied with considerable interests. We are further only interested in *explicit* constructions of PRGs for them to be useful in space-bounded derandomization, which means G must be computable in small space. Therefore, although probabilistic method guarantees the existence of a PRG with optimal seed length $O(\log(n/\varepsilon))$ for (n,n)-ROBPs [Hoz21b], it cannot be used directly to derandomize **BPL** due to it being non-explicit.

A breakthrough in PRG research came when Nisan designed an explicit PRG for (w,n)-ROBPs with seed length $O(\log n \cdot \log(nw/\varepsilon))$ [Nis90].

Theorem 2.2.2 ([Nis90])**.** *For every $w, n \in \mathbb{N}$ and $\varepsilon > 0$, there exists an explicit ε-PRG for (w,n)-ROBPs with seed length $r = O(\log n \cdot \log(wn/\varepsilon))$ and uses space $O(r)$.*

Nisan's generator has been the cornerstone in space-bounded derandomization research and serves as an inspiration for PRG designs. It has also seen applications in streaming algorithms where it helps reduces randomness usage [Ind06]. Since it is only a $O(\log n)$ factor away from the optimal seed length, it was considered as a source of optimism for moving towards proving $\mathbf{BPL} = \mathbf{L}$. Despite intense effort in the last three

decades, however, it has stubbornly resisted any improvements and remains the best known explicit PRG for (n, n)-ROBPs to this day.

Difficulties in improving over Nisan's PRG has prompted new approaches to derandomization. Braverman, Cohen, and Garg [BCG19] recently introduced a generalization of PRGs called **weighted pseudorandom generators** (WPRGs) . Essentially, WPRGs give each outcome a real-valued weight.

Definition 2.2.3. *Let $\mathcal{F}$ be a class of functions $f : \{0,1\}^n \to \{0,1\}$. An ε-WPRG is a function $G : \{0,1\}^r \to \{0,1\}^n$ together with a weight $\rho : \{0,1\}^r \to \mathbb{R}$, such that for every $f \in \mathcal{F}$,*

$$\left| \mathop{\mathbb{E}}_{x \in \{0,1\}^r} [\rho(x) \cdot f(G(x))] - \mathop{\mathbb{E}}_{x \in \{0,1\}^n} [f(x)] \right| \leq \varepsilon.$$

If ρ is bounded by K, we say the WPRG is K-bounded.

Note that when $\rho \equiv 1$, WPRGs turn into the standard PRGs. Better seed length for (n, n)-ROBPs than Nisan's generator has been achieved using WPRGs: see for instance [Hoz21a]. The generalization from PRG to WPRG provides a valuable insight into the derandomization process. When using a PRG to derandomize space-bounded computation, we approximate the true accpetance probability by calculating it on a smaller set of random seeds sampled via the pseudorandom output of the PRG. However, we are not limited to using a uniform weighting/distribution on these sampled seeds as a plain PRG would imply – we are allowed to apply a customized weighting to our sampled results, as WPRGs suggest.

2.3 Derandomizing Space-Bounded Computation

We will now point out the connection between pseudorandom generators and derandomizing **BPL**. An obvious way of using a PRG G to derandomize computation f is to compute $\mathbb{E}_{x \in \{0,1\}^r}[f(G(x))]$ directly, by iterating through all $x \in \{0,1\}^r$. If G is explicitly computable in space $O(r)$, then this process takes $O(r)$ space as well. If we take G to be Nisan's generator, we have the following result:

Theorem 2.3.1. BPL $\subseteq \mathbf{L}^2$.

Proof. Let T be a probabilistic log-space machine and N be the length of its input. Then the number of configurations, together with the maximum steps of computation, which is also the randomness needed, are both $2^{O(\log N)}$. Therefore (c.f. proposition 2.1.6) there exists a (w, n)-ROBP f_x where $n, w = \text{poly}(N)$ for every input x such that $f_x(y) = T(x, y)$, where y denotes the random seed. Fix $\varepsilon = 0.01$, and Nisan's PRG for (w, n)-ROBPs have seed length $O(\log(nw) \log n) = O(\log^2 N)$. Therefore, iterating over all the seeds to the PRG and computing the average only takes $O(\log^2 N)$. Furthermore, constructing the ROBP f_x takes only $O(\log N)$ space, so the whole deterministic process takes $O(\log^2 N)$ space. □

However, this straightforward averaging process is not the only way to use PRGs for derandomization. Saks and Zhou later devised a recursive procedure that gives a better derandomization of **BPL** [SZ99] using Nisan's PRG, which yields the following result:

Theorem 2.3.2 ([SZ99])**.** $\mathbf{BPL} \subseteq \mathbf{L}^{3/2}$.

It was later shown that Saks and Zhou's method can be adapted to use any PRGs or WPRGs [CL20]. Hoza showed that, using a PRG by Armoni [Arm99] and error reduction techniques enabled by WPRG construction, an even better derandomization can be achieved [Hoz21a]:

Theorem 2.3.3 ([Hoz21a])**.** $\mathbf{BPL} \subseteq \mathbf{DSPACE}\left(\frac{\log^{3/2} N}{\sqrt{\log \log N}}\right)$.

The focus of this paper will be on the outcome of applying Saks and Zhou's approach to potentially "better" PRGs like Armoni's. Specifically, we aim to investigate the question: using current techniques, what improvements in derandomization of **BPL** do improvements in seed length of (W)PRGs imply?

Chapter 3

The Saks-Zhou Framework

3.1 Nisan's PRG

It is necessary to study the particular structure of Nisan's PRG before we describe Saks and Zhou's original idea of recursive derandomization, in which Nisan's PRG plays a crucial part. We start by giving the construction of Nisan's PRG.

Definition 3.1.1. *Let H be a family of hash functions $h : D \to M$. If for any $x \neq y$ in D,*

$$\mathbb{P}_{h \in H}[h(x) = h(y)] \leq 1/|M|,$$

H is called a universal hash family.

Theorem 3.1.2 ([Nis90])**.** *Let H be a family of universal hash functions $h : \{0,1\}^c \to \{0,1\}^c$. Note that we can find simple H such that h can be described with a string of length $O(c)$. For $k \geq 0$, define $G_r : \{0,1\}^c \times H^r \to (\{0,1\}^c)^{2^r}$ recursively as follows:*

$$G_0(x) = x,$$

$$G_r(x, h_1, \ldots, h_r) = G_{r-1}(x, h_1, \ldots, h_{r-1}) \circ G_{r-1}(h_r(x), h_1, \ldots, h_{r-1}),$$

where $\circ$ means concatenation. We call c the block size of the generator.

Then, setting $c = O(\log(wn/\varepsilon))$ and $r = O(\log n)$, G_r is a ε-PRG for (w, n)-ROBPs with seed length $O(\log n \cdot \log(wn/\varepsilon))$.

We omit the proof and refer the reader to Nisan's original paper [Nis90].

It is worth pointing out some peculiarities in Nisan's generator. First, note that the PRG is of the form $G(x, y)$ where the seed can effectively be split into two different parts: x is a short uniform random seed, and y is a sequence of (randomly picked) hash functions. These hash functions are then applied to x in a tree-like fashion, and after k levels stretch x by a factor of 2^k.

The second point is more technical: for any ROBP f, if we fix y (the hash functions) at random, then with high probability, the y we pick will satisfy the property

$$\left| \mathop{\mathbb{E}}_{x \in \{0,1\}^c} [f(G(x, y))] - \mathop{\mathbb{E}}_{u \in \{0,1\}^n} [f(u)] \right| \leq \varepsilon.$$

That is, we are allowed to pick y in advance and only average over the first seed x to approximate the average outcome of the ROBP f, and we expect our choice of y to be good the vast majority of the time. We will formalize this property in the next subsection and use it devise a "randomized" matrix powering algorithm.

3.2 Recursive Derandomization

Space-bounded derandomization can be solved by calculating the acceptance probability of an ROBP on a uniformly random input, which, in a sense, is just matrix exponentiation. As discussed in Section 2, a randomized Turing machine T that uses space S can be thought of as a FSM with $w = 2^{O(S)}$ states, and the random seeds to the Turing machine can be thought of as input to the FSM. This random transition between states is a Markov chain, and we associate the computation with a size w matrix M, where $M[i, j]$ denotes the probability that the FSM transitions from state i to state j, given uniformly random input. To find out the probability that T accepts after n steps, we only need to compute M^n and look at the corresponding entry. Actually, it is good enough to *approximate* M^n up to some constant error, since **BPSPACE** requires the acceptance probability to differ significantly depending on whether the input is actually in the language.

The next proposition summarizes what we just said about M:

Proposition 3.2.1. *Let $T(x, y)$ be a randomized Turing machine where x is the input, y the random seed. Suppose $|x| = N$ and T uses space $O(S)$. Then for each x, there exists a stochastic matrix M_x of size $2^{O(S)}$. and an entry $[i, j]$ (corresponding to the starting and accepting configuration), such that $M_x^n[i, j] = \mathbb{P}_{y\in\{0,1\}^n}[T(x, y)$ accepts$]$. Furthermore, each entry of M_x can be computed in constant space.*

Consequently, if we can find a space-efficient way of computing $M^n[i, j]$, we obtain a derandomization of **BPSPACE**(S).

Proposition 3.2.2. *Let $S(N) \geq \log N$. Let $w = 2^{O(S)}$. If there exists a deterministic algorithm A that takes M, i, j, w as input, where M is a $w \times w$ stochastic matrix and $i, j \in [w]$, and approximates $M^w[i, j]$ using space $O(S')$, then* **BPSPACE**$(S) \subseteq$ **DSPACE**(S').

Proof. Let $T(x, y)$ be a randomized Turing machine. We construct a deterministic algorithm D that derandomizes T as follows. D "knows" how to compute each entry of M_x in constant time, but it does not precompute M_x. Instead, it only computes (i, j) corresponding to the start and accepting configurations and w the exponent of the matrix, and send them to algorithm A. Whenever A needs an entry of M_x, D can compute that in constant time. Finally, depending on whether $M^w[i, j]$ is greater than $2/3$, D accepts or reject the input. It is easy to verify everything D does is in $O(\log N)$ space, and so the whole process takes $O(S')$ space. □

Saks and Zhou gave a space-efficient way of approximating matrix exponentiation using two main ideas.

The first is that we are actually allowed to use randomness to compute matrix powering even if the final algorithm has to be deterministic – we just need to iterate through all the random seeds and average the result, given the random seed works with high probability. By doing so, we only need to pay extra space for the random seed. We will next show how Nisan's PRG gives a way to compute matrix exponentiation using randomness.

The second idea is that if we apply the aforementioned randomized matrix exponentiation many times, we do not need a new seed every time – we can get away with using the same random seed, thus saving space. There is one caveat, however: we cannot square the matrix in a bottom-up fashion, as we do not have the space to store the result in full. Therefore, we must compute each entry of the matrix recursively. However, note that the recursion adds extra overhead as each level of recursion needs its own "stack space". Additionally, it turns out we cannot directly reuse the random seed – some cleverness is needed.

Randomized Matrix Exponentiation

Remark. *From now on, we use $\|M\|$ to denote the max norm of matrix M: $\|M\|_{\max}$.*

We now point out how we can allow randomness in matrix exponentiation.

Lemma 3.2.3 ([SZ99])**.** *For a randomized matrix algorithm, iterating over the random seed and calculating the average of all matrix output is called the naive derandomization of that algorithm. Let F be some function on stochastic matrices and A an algorithm that uses a random seed of length R and space S, and outputs a stochastic matrix. If A approximates F with accuracy 2^{-a} and error probability $\beta \leq 2^{-(a+1)}$, then the naive derandomization approximates F with accuracy 2^{-a+1} using space $O(S+R)$.*

We omit the proof here as it is straightforward. Essentially, we only need the error probability to be close to the approximation accuracy for the naive derandomization to work.

Now we show how to use Nisan's PRG to perform matrix exponentiation. Given a FSM Q and some distribution of its input D, let $Q(D)$ denote the the distribution matrix, i.e. each entry is denotes the transition probability between states of Q given input distribution conforming to D. Nisan's PRG has the following property:

Lemma 3.2.4 ([SZ99])**.** *Let Q be a FSM with w states and input alphabet size 2^m. Let $G : \{0,1\}^m \times H^r \to (\{0,1\}^m)^{2^r}$ be Nisan's PRG. If $y \in H^r$ is picked at random, for*

any $\varepsilon > 0$, with probability $1 - O(rw^5 2^{2r}/2^m\varepsilon^2)$, the following holds:

$$\|Q(G(U_m, y)) - Q^{2^r}(U_m)\| < \varepsilon,$$

where U_m denotes the uniform distribution on $\{0,1\}^m$.

In fact, this property was used as an intermediate step to prove the correctness of the PRG. For a detailed proof, see [SZ99].

Thus Nisan's Generator provides a way to approximate the exponent of transition matrix of a FSM. How should we use it to calculate the exponent of an arbitrary stochastic matrix? We must first realize the matrix M as an FSM Q, such that $Q(U_m) = M$, then calculate an entry of $Q(G(U_m, y))$ by directly iterating over all $x \in \{0,1\}^m$.

Theorem 3.2.5 ([CL20])**.** *For every n, w, ε, there exists an algorithm Pow_n that takes a $w \times w$ stochastic matrix M and a random seed y of length $O(\log n \log(nw/\varepsilon))$, such that with probability at least $1 - \varepsilon$,*

$$\|\mathrm{Pow}_n(M, y) - M^n\| < \varepsilon.$$

Furthermore, this algorithm uses space $O(\log(nw/\varepsilon))$.

Proof. Pow_n is carried out as follows. In Lemma 3.2.4, set $m = O(\log(nw/\varepsilon)), r = O(\log n)$. Given matrix M, construct a FSM Q with w states and alphabet $\Sigma = \{0,1\}^m$, such that $Q(U_m) = \lfloor M \rfloor_m$ (truncating to precision 2^{-m}). To calculate entry (i, j), iterate through $x \in \{0,1\}^m$. For each x, start from state i, calculate the next length-m block of $G(x, y)$, and transition to the next state according to Q. If after n steps state j is reached, increment count. After the iteration of x is complete, output the count$/2^m$. Verifying the space usage is indeed $O(\log(nw/\varepsilon))$ is straightforward. We only note that Nisan's PRG allows us to generate its output block by block in space $O(\log(nw/\varepsilon))$. □

Reusing The Random Seed

Suppose we want to calculate the n-th power of $w \times w$ matrix M. Let $2^r = n$ and $2^s = w$. Instead of calculating the 2^r-th power directly, we factor $r = r_1 r_2$. Theorem

3.2.5 gives us an algorithm Pow to calculate the 2^{r_1}-th power of M, and we can apply Pow r_2 times to obtain the 2^r-th power (as $(2^{r_1})^{r_2} = 2^r$).

Although it seems for r_2 applications of Pow we need r_2 different random seeds y_i, it is natural to ask whether we can get away with just using one seed y. Assuming this is possible, let us calculate the space complexity. By the naive derandomization, deterministic space usage is the sum of the space needed for the random seed and the space needed for computation. The random seed, y, has length $O(sr_1)$. There are r_2 levels of recursion, and each requires $O(s)$ space, for a total of $O(sr_2)$. Therefore, if $r_1 = r_2 = \sqrt{r}$, total space needed is $O(s(r_1 + r_2)) = O(s\sqrt{r})$. Therefore, in the context of derandomizing **BPL**, where N is the length of input and $s, r = O(\log N)$, this translates to a deterministic derandomization in $O(\log^{3/2} N)$ space.

The tricky part, however, is proving that using the same seed y does succeed with high probability for all randomized exponentiation. If we fix a sequence of M_i' that is independent from seed y, it is easy to say that a random y works for all of them with high probability. However, this ceases to work for our sequence from recursive application $M_{i+1} = \text{Pow}(M_i, y)$, as the whole sequence is dependent on y.

To overcome the dependence on y, we will try to "regulate" M_i via a "randomized snapping" operation. Essentially, we randomly perturb the entries of M and truncate them to a lower precision. Consider the sequence $M_{i+1}' = (M_i')^{2^{r_i}}$, which is independent from y. Since M_i is supposed to approximate M_i', performing randomized snapping on these two matrices will yield the same result with high probability. Thus, at least with high probability, we can fix the value of M_i without losing too much accuracy.

We now formalize these ideas, starting by defining randomized snapping:

Definition 3.2.6. *Given value $x \in \mathbb{R}$ and random seed $y \in \{0,1\}^d$, define*

$$\text{Snap}_d(x, y) = \max(\lfloor 2^d \cdot x - 2^{-d} y \rfloor \cdot 2^{-d}, 0).$$

For a matrix M, let $\text{Snap}_d(M, y) = M'$ such that $M'[i,j] = \text{Snap}_d(M[i,j], y)$.

Note that the randomized snapping operation first perturb x by a random value in

$[0, 2^{-d}]$ (in 2^{-2d} increments), then truncate to d bits. Thus if two matrices are close, then with high probability they become equal after the snap operation.

Lemma 3.2.7 ([CL20]). *If $\|M - M'\| < \varepsilon$, then*

$$\mathop{\mathbb{P}}_{y}[\mathrm{Snap}_d(M, y) \neq \mathrm{Snap}_d(M', y)] \leq w^2(2^d\varepsilon + 2^{-d}).$$

Proof. The snap operation is equal to choosing a grid of length 2^{-d} at random and snapping to the left grid point. For two real entries that are ε apart, they can only be snapped to different points when a grid point falls between them. This happens with probability at most $\varepsilon/2^{-d} + 2^{-d}$ (the last term due to the choice of grid being discrete). Union bound over the entire matrix gives $w^2(2^d\varepsilon + 2^{-d})$. □

Now, if we compose Pow with Snap, there is a high chance that we can "fix" the outcome at each step. That is, we take $M_{i+1} = \mathrm{Snap}_d(\mathrm{Pow}(M_i), z_{i+1})$. Note that we have to pay for the extra randomness z_i for each level of recursion, but we shall see the cost is not too much.

Theorem 3.2.8 ([CL20]). *Let M be a $w \times w$ stochastic matrix. Assume $n \leq w$. Let $r_1 r_2 = r = \log n, s = \log w$. Let Pow_{n_1} be the powering algorithm given by Theorem 3.2.5, where $n_1 = 2^{r_1}$, and let ε be the error of that algorithm. Let y be the random seed to* Pow *and $z_1, \ldots, z_{r_2} \in \{0,1\}^d$ be random seeds to Snap_d. Define $M_0 = M$, $M_{i+1} = \mathrm{Snap}_d(\mathrm{Pow}(M_i), z_{i+1})$. With probability at least $1 - O(w^2 r_2(2^d\varepsilon + 2^{-d}))$ over y, z_i,*

$$\|M_{r_2} - M^n\| \leq nw2^{-d+1}.$$

Furthermore, M_{r_2} can be deterministically computed in space $O(r_1 s + r_2(s + d))$.

Proof. (sketch) Define $M'_0 = M$, $M'_{i+1} = \mathrm{Snap}_d((M'_i)^{n_1}, z_{i+1})$. Consider the situation when the following two events happen at once:

1. z_i's are so chosen such that for each i the following holds: $\mathrm{Snap}_d(M'_i, z_i) = \mathrm{Snap}_d(M, z_i)$ for all M where $\|M - M'_i\| \leq \varepsilon$.

2. Given fixed z_i's, y is chosen such that for all i, $\mathrm{Pow}(M'_i, y)$ successfully approximates the exponent up to error ε.

Using the union bound, the above situation fails with probability at most $O(w^2 r_2(2^d\varepsilon + 2^{-d}))$.

When all the conditions are met, however, evidently $M'_{r_2} = M_{r_2}$. The snapping operation gives $\|M'_i - (M'_{i-1})^{n_1}\| \leq 2^{-d+1}$. Lemma 5.4 in [SZ99] the gives $\|M^n - M'_{r_2}\| \leq nw2^{-d+1}$.

For the space complexity, randomness usage are y and z_i's, which total $O(r_1 s + r_2 d)$. There are r_2 levels of recursion, each using $O(s)$ space to compute $\mathrm{Snap} \circ \mathrm{Pow}$, totaling $O(r_2 s)$ space. Thus the space complexity is $O(r_2(s+d)+r)$. □

Finally, we now prove $\mathbf{BPL} \subseteq \mathbf{L}^{3/2}$.

Theorem 3.2.9. $\mathbf{BPL} \subseteq \mathbf{L}^{3/2}$.

Proof. In theorem 3.2.8, we may assume $r = s = O(\log N)$ where N is the input length to the **BPL** machine. Set $r_1 = r_2 = \sqrt{r}$, $d = O(\log w)$, $\varepsilon = 2^{-2d}$. Then $nw2^{-d+1}$ and $O(w^2 r_2(2^d\varepsilon + 2^{-d}))$ both turn into some very small constant, which is good enough for derandomization purposes (see Lemma 3.2.3 also). The space usage is $O(\sqrt{r}s + \sqrt{r}(s+r)) = O(\log^{3/2} N)$. We have thus obtained a very good matrix exponent approximation algorithm, and by proposition 3.2.2, we are done. □

3.3 Generalization Using Arbitrary PRGs

Saks and Zhou's proof relies crucially on a particular property of Nisan's PRG that implies a randomized algorithm which calculates matrix exponent using only small space. However, not every PRG has that property, and it is not clear how such an algorithm can be devised using arbitrary PRGs. Fortunately, Armoni [Arm99] found a way to use an arbitrary PRG to perform matrix powering by composing the PRG with a sampler.

First, we introduce the definition of a sampler.

Definition 3.3.1. *A function $g : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ is a (ε, δ) sampler if for every function $f : \{0,1\}^m \to [0,1]$,*

$$\mathop{\mathbb{P}}_{x\in\{0,1\}^n}\left[\left|\mathop{\mathbb{E}}_{s\in\{0,1\}^d}[f(g(x,s))] - \mathop{\mathbb{E}}_{y\in\{0,1\}^m}[f(y)]\right| \le \varepsilon\right] \le 1-\delta.$$

To try to unpack the definition a bit: if we choose a seed x at random, then with high probability, the average value of f can be approximated by the average value of $f(g(x,s))$ where the expectation is taken over s.

It had been shown that good, small-space samplers exist:

Lemma 3.3.2 ([CL20], Appendix B)**.** *For every $\delta, \varepsilon > 0$ and m, there exists a (ε, δ) sampler $g : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ such that $d = O(\log\log(1/\delta) + \log(1/\varepsilon))$ and $n = m + O(\log(1/\delta) + \log(1/\varepsilon))$. f can be computed in space $O(m + \log(1/\delta) + \log(1/\varepsilon))$.*

For technicality, we allow a ROBP to have a larger alphabet size, just like FSM. A (w, n, d)-ROBP consumes d bits of input between layers, instead of just 1 bit.

Now we are ready to state Armoni's randomized matrix powering algorithm.

Theorem 3.3.3 ([CL20])**.** *Suppose there exists a PRG G for $(w, n, \log(3nw/\varepsilon))$-ROBP with error $\varepsilon/3$, seed length ℓ, and computable in space $O(l)$. Then there is a powering algorithm Pow_G that approximates the n-th power of any stochastic matrix M with accuracy ε and error probability $\ge 1-\varepsilon$. Pow_G uses $l + O(\log(w/\varepsilon))$ amount of randomness, and needs at most $O(\log(nw/\varepsilon))$ persistent stack space.*

Proof. (sketch) We first explain what we mean by persistent stack space. Since Pow will be used recursively, every time it reads an entry from its input matrix M, it actually enters into a recursion to calculate that entry. Therefore, Pow needs stack space to save the current state of computation, and persistent stack space means the space we cannot erase before each read from M.

The proof itself is analogous to that of theorem 3.2.5. Given M, we first truncate M to $d = \log(3nw/\varepsilon)$ bits. Call the resulting matrix M'. Then the accuracy loss is small:

$$\|M^n - (M')^n\| \le \varepsilon/3.$$

We then construct a ROBP B such that its transition matrix is equal to M'. We omit the details of the construction as it is straightforward. Let $B_{i,j}$ denotes the indicator function of whether the input leads from state i to state j. Then $\mathbb{E}_x[B_{i,j}(x)] = (M')^n[i,j]$.

By the definition of a PRG, we have

$$\left|\mathop{\mathbb{E}}_{r}[B_{i,j}(G(r))] - \mathop{\mathbb{E}}_{x}[B_{i,j}(x)]\right| \leq \varepsilon/3.$$

Now let Samp be a $(\varepsilon/3, \varepsilon/w^2)$ sampler. Define Pow to be

$$\mathrm{Pow}(M,y)[i,j] = \mathop{\mathbb{E}}_{z}[B_{i,j}(G(\mathrm{Samp}(y,z)))].$$

Then by definition of Samp, with probability at least $(1-\varepsilon/w^2)$ over choice of y,

$$\left|\mathrm{Pow}(M,y)[i,j] - \mathop{\mathbb{E}}_{r}[B_{i,j}(G(r))]\right| \leq \varepsilon/3.$$

Taking the union bound, then with probability at least $(1-\varepsilon)$, the above holds for each i,j.

Thus by triangular inequality,

$$\|\mathrm{Pow}(M,y) - M^n\| \leq \varepsilon.$$

Let us now calculate the space complexity of Pow. y is the only source of randomness, which has length $\ell + O(\log(w^2/\varepsilon) + \log(1/\varepsilon)) = l + O(\log(w/\varepsilon))$. For the stack space needed, note that the space needed to calculate $G \circ \mathrm{Samp}$ can be reclaimed and we only need to save the length d block (saving as much as $O(l)$ space). Together with some other bookkeeping needed for traversing the ROBP, the space usage comes to $O(\log(wn/\varepsilon))$. □

Therefore, an arbitrary PRG with seed length ℓ can be used to implement a randomized matrix powering algorithm using $\log(nw/\varepsilon)$ space and $\ell + O(\log(w/\varepsilon))$ bits of randomness. When this algorithm is used recursively as in Chapter 3.2, we could

obtain results similar to Theorem 3.2.9. If, then, we can find a PRG with seed length better than Nisan's PRG, we could potentially obtain a better derandomization of **BPL** just by using the above framework. In fact, this is exactly what Hoza did in [Hoz21a], although with a slight twist.

Lastly, we state a similar result which will be used later, stating that WPRGs can also be used to power matrices in small spaces:

Theorem 3.3.4 ([CL20])**.** *Suppose there exists a* $\mathrm{poly}(nw/\varepsilon)$ *bounded WPRG* G *for* $(w, n, \log(3nw/\varepsilon))$*-ROBP with error* $\varepsilon/3$*, seed length* ℓ*, and computable in space* $O(l)$*. Then there is a powering algorithm* Pow_G *that approximates the* n*-th power of any stochastic matrix* M *with accuracy* ε *and error probability* $\geq 1 - \varepsilon$*.* Pow_G *uses* $l + O(\log(w/\varepsilon))$ *amount of randomness, and needs at most* $O(\log(nw/\varepsilon))$ *persistent stack space.*

The proof is almost the same as 3.3.3, so we omit it here.

3.4 Derandomization Consequences of Better PRGs

We begin by generalizing theorem 3.2.8 by allowing arbitrary PRG to be used.

Lemma 3.4.1. *Let* G *be a PRG satisfying Theorem 3.3.3 with seed length* ℓ*, and let* Pow_G *be the resulting powering algorithm. Then, using* Pow_G *instead in Theorem 3.2.8 gives an equally good approximation with space usage* $O(\ell + r_2(s + d))$*. Furthermore, to achieve constant accuracy with constant error (as is appropriate for Lemma 3.2.3), assuming the parameters for the PRG satisfy* $\varepsilon = 1/\mathrm{poly}(w)$*, the space usage is* $O(\ell + r_2 s)$*.*

Proof. Replacing all mentions of Pow by Pow_G in the proof of theorem 3.9 suffices to prove the first assertion. For the second assertion, note that we only need to set $d = O(\log(w))$ and $\varepsilon = 2^{-O(d)}$. □

As we have said before, assuming the existence of a PRG with seed length ℓ that is "better" than Nisan's, applying the Saks-Zhou framework with that PRG yields an

improved derandomization of **BPL**. We formalize this observation in the next theorem, which will be used extensively in the rest of this section and Chapter 4.

Theorem 3.4.2. *Let N be any natural number. Let $w = 2^{O(\log N)} = 2^s$, so $s = O(\log N)$. Assume that $r(s) = O(s/g(s))$ for some function $g(s) = O(\sqrt{s})$, and let $n = 2^r$. Suppose there exists a PRG for $(w, n, d = \Theta(\log w))$-ROBPs with error $\varepsilon = O(1/\text{poly}(w))$.[1] Assume this PRG has seed length ℓ and can be computed in space $O(l)$. If $\ell = O(s^{3/2}/t(s))$, then a randomized log-space Turing machine with input length N can be derandomized in space $O(s^{3/2}/t(s) + s \cdot g(s))$.*

Proof. Given such as PRG G, by Theorem 3.3.3, there exists an algorithm Pow_G that takes a $w \times w$ matrix M and approximate M^n. Now, let $r' = O(g)$, so that $r \cdot r' = s$. By the previous lemma, Pow_G applied r' times recursively approximates M^w with high probability. The space usage is thus $O(\ell + r' \cdot s) = O(s^{3/2}/t(s) + s \cdot g(s))$. By Proposition 3.2.2, a log-space randomized Turing machine with input length N can be derandomized by approximating M^w where $w = 2^{O(\log N)}$ and M is a $w \times w$ stochastic matrix, and so the conclusion follows. □

It is important to mention that this theorem still holds true if G is a $\text{poly}(nw/\varepsilon)$ bounded WPRG and the proof is essentially the same.

Note that when G is Nisan's PRG, $g(s) = \sqrt{s}$, $t(s) = 1$, then the derandomization is exactly Saks-Zhou's original result. However, if a PRG exists such that when $n \ll w$ it is better than Nisan's PRG ($s^{3/2}$), then a better derandomization of **BPL** ensues.

If good PRGs are conjectured to exist, then Theorem 3.4.2 tells us the corresponding improvement in derandomizing **BPL**.

Corollary 3.4.3 ([CL20])**.** *If there exists a ε-PRG for $(w, n, O(\log w))$-ROBPs which has seed length $\ell = O(\log^2 n + \log^{4/3}(w/\varepsilon))$, then $\mathbf{BPL} \subseteq \mathbf{L}^{4/3}$.*

Proof. Note that this PRG satisfies Theorem 3.4.2 with $g = s^{1/3}$ and $t = s^{1/6}$. It is also easy to see that this is also the optimal setting of g for this particular ℓ. $O(s^{3/2}/t(s) + s \cdot g(s)) = O(s^{4/3}) = O(\log^{4/3} N)$. □

[1] d and ε can be fixed explicitly.

It was suggested in [BCG19] that a research program of constructing WPRGs with seed length that decouples n, w, and ε may eventually lead to proving much better derandomization results, such as $\mathbf{BPL} \subseteq \mathbf{L}^{4/3}$. In [BCG19], [CL20], and [Hoz21a], advancements have been made to construct WPRGs where the error term $\log(1/\varepsilon)$ is decoupled, but it is still unclear what a WPRG construction that decouples n and w might look like.

Armoni was able to construct a PRG that slightly improves upon Nisan's seed length when $n \ll w$. We defer its construction to the next section and merely demonstrate its derandomization consequences assuming its existence.

Corollary 3.4.4. *If there exists a* $1/\mathrm{poly}(w)$*-PRG for* (w, n, d)*-ROBP with seed length*

$$\ell = O\left(d + \frac{\log(wn)\log n}{\max\{1, \log\log w - \log\log n\}}\right),$$

then

$$\mathbf{BPL} \subseteq \mathbf{DSPACE}\left(\frac{\log^{3/2} N}{\sqrt{\log\log N}}\right).$$

Proof. Setting $d = O(\log w)$, this PRG satisfies Theorem 3.4.2 with $t = \sqrt{\log s}$ and $g = \frac{\sqrt{s}}{\sqrt{\log s}}$. Thus, $O(s^{3/2}/t(s) + s \cdot g(s)) = O\left(\frac{s^{3/2}}{\sqrt{\log s}}\right) = O\left(\frac{\log^{3/2} N}{\sqrt{\log\log N}}\right)$. □

There is just one last snag – Armoni's original PRG only has the desired seed length when the error is $1/\mathrm{poly}(n)$, not $1/\mathrm{poly}(w)$. The saving grace is that we are allowed to reduce the error by turning the original PRG into a WPRG without adding much seed length.

Chapter 4

Armoni's PRG

4.1 Armoni's Recursive Construction

In this section we describe the construction of Armoni's PRG [Arm99], using an improved extractor design as suggested in [KNW08].

First, we define extractors.

Definition 4.1.1. *A distribution D on $\{0,1\}^n$ is called a δ source if $D(x) \leq 2^{-\delta}$ for all $x \in \{0,1\}^n$.*

Definition 4.1.2. *A function $E : \{0,1\}^k \times \{0,1\}^t \to \{0,1\}^m$ is a (δ, ε) extractor if for every δ source D on $\{0,1\}^k$, $X \sim D$, and $Y \sim U_t$, $\|E(X,Y) - U_m\|_{TV} \leq \varepsilon$.*

Guruswami, Umans and Vadhan gave an explicit extractor construction in [GUV09] with near optimal parameters, and Kane, Nelson, Woodruff showed in [KNW08] this extractor can be computed in linear space.

Theorem 4.1.3 ([KNW08])**.** *For $k > 0, \delta \leq k, \varepsilon > 0$, there is a (δ, ε) extractor $E : \{0,1\}^k \times \{0,1\}^t \to \{0,1\}^m$ with $t = O(\log k + \log(1/\varepsilon))$, $m \geq \delta/2$. E can be computed in space $O(n + \log(1/\varepsilon))$.*

We only need $m = \Theta(k)$ for Armoni's construction to work, so we shall use the following corollary.

Corollary 4.1.4. *For $k, \varepsilon > 0$, there is a $(k/2, \varepsilon)$ extractor $E : \{0,1\}^k \times \{0,1\}^t \to \{0,1\}^m$ where $t = O(\log(k/\varepsilon))$ and $m = k/4$.*

We start by constructing a basic PRG just by concatenating outputs of E (with different second seeds).

Lemma 4.1.5. *Suppose n, w, r, γ are given. Let E be the extractor in Corollary 4.1.4. There exists constants c_1, c_2, c_3, such that when $\varepsilon = c_1(\gamma/n)$, $k = c_2(\log(w/\varepsilon))$, and $t = c_3(\log(k/\varepsilon))$,[1] the following holds:*

Define $\hat{G} : \{0,1\}^{k+nt} \to \{0,1\}^{nr}$ as:

$$\hat{G}(x, y_1, \ldots, y_n) = E(x, y_1) \circ \cdots \circ E(x, y_n), \quad x \in \{0,1\}^k, y_i \in \{0,1\}^t.$$

Then, $\hat{G}$ is a γ-PRG for (w, n, r)-ROBPs.

(Note we only keep r bits from each output block if $m = k/4 > r$.)

We omit the proof and refer the reader to [Arm99] for details. It is illustrative, however, to calculate the length of each parameter. Assuming $\log w \ll n \ll w$, we have

$$k = \Theta(\log(w/\varepsilon)) = \Theta(\log(wn/\gamma)),\ m = \Theta(k),$$

$$t = \Theta(\log(k/\varepsilon)) = \Theta\left(\log \frac{n \log(nw/\gamma)}{\gamma}\right) = \Theta(\log(n/\gamma)).$$

Next we recursively compose $\hat{G}$ to obtain a much better PRG. We point out one crucial property of $\hat{G}$ that allows easy recursive composition: $\hat{G}$ only needs to read y_i's once to generate the output. That is to say, if we let $\hat{G}_x(y) = \hat{G}(x, y)$ for a fixed x, then for any ROBP B, $B \circ \hat{G}_x$ is again an ROBP. Therefore, instead of using many y_i's, we use the pseudorandom output of $\hat{G}$ again, and proceed recursively.

Theorem 4.1.6 ([Arm99])**.** *Let n, w, r, α be given. Let $\gamma = \Theta(\alpha/\log n)$, $k = \Theta(\log(wn/\gamma))$, $t = \Theta(\log(n/\gamma))$, $m = \Theta(k)$. By Lemma 4.1.5, let $\hat{G}$ be the PRG thus defined with parameter k and t. Let $h = \Theta(\frac{\log n}{\log(k/t)})$, and define $n_1 = n$, $n_i = n_{i-1}/(m/t)$. Define $G_i : \{0,1\}^{k \cdot i} \times \{0,1\}^{n_i \cdot t} \to \{0,1\}^{n \cdot r}$:*

$$G_1 = \hat{G},$$

[1] For ease of reading, in future results we will express this as $t = \Theta(\log(k/\varepsilon))$ without mentioning the constant c_3 explicitly.

$$G_i(x_1, \ldots, x_i, y_1, \ldots, y_{n_i}) = G_{i-1}(x_1, \ldots, x_{i-1}, \hat{G}(x_i, y_1, \ldots, y_{n_1})),$$

where we split each output block from $\hat{G}$ of length m into m/t blocks of length t, and treat them as y_i's to G_{i-1}. Now define $G(x_1, \ldots, x_h, y) = G_h(x_1, \ldots, x_h, y)$.

Then, G is a α-PRG for (w, n, r)-ROBPs with seed length

$$\ell = O\left(r + \frac{\log n \cdot \log(wn/\alpha)}{\max\{1, \log\log w - \log\log(n/\alpha)\}}\right)$$

and can be computed in $O(\ell)$.

Proof. (sketch) We shall skim over the proof of correctness and focus on calculating the seed length.

Let P be a (w, n, r)-ROBP. We claim that for each G_i, when $x = x_1 \cdots x_i$ is fixed, $P \circ G_{i,x}$ is a (w, n_i, t)-ROBP, as each G_i still possess the read-once property regarding y_i's as discussed. Now this (w, n_i, t)-ROBP can be "compressed" into a (w, n_{i+1}, m)-ROBP by grouping transitions between m/t layers into a transition between two layers, thereby reducing the number of layers while enlarging the dictionary size. But since $\hat{G}$ is a PRG for (w, n_{i+1}, m)-ROBPs, $\|P \circ G_{i,x} - P \circ G_{i,x} \circ \hat{G}\| < \gamma$ (where we represent each function with its probabilistic matrix on uniformly random input). By induction on i, we are done.

Let's now calculate the seed length. We first explain where the r comes from: to produce length r blocks, length of x_1 has to be at least r. Thus $\ell = O(r + k \cdot h + t) = O(r + k \cdot h)$ as $t = O(k)$. Therefore,

$$O(k \cdot h) = O\left(\log(wn/\alpha) \cdot \frac{\log n}{\log(k/t)}\right)$$

On the other hand, we have

$$k/t = \frac{\Theta(\log(nw/\alpha))}{\Theta(\log(n/\alpha))} = \Theta\left(\frac{\log w}{\log(n/\alpha)}\right).$$

Thus

$$O(k \cdot h) = O\left(\log(wn/\alpha) \cdot \frac{\log n}{\log\log w - \log\log(n/\alpha)}\right).$$

One can also verify the space complexity, which we also omit here. □

By setting $n = \exp(\sqrt{\log w \cdot \log\log w})$, $\alpha = 1/\mathrm{poly}(n)$ and $r = O(\log w)$, the seed length is

$$\ell = O\left(\frac{\log^{3/2} w}{\sqrt{\log\log w}}\right),$$

which almost fits the requirements of Corollary 3.4.4. However, the error is $1/\mathrm{poly}(n)$ instead of $1/\mathrm{poly}(w)$.

To resolve this conundrum, it turns out a result by Cohen, Doron, Renard, Sberlo, and Ta-Shma [CDR+21] enable us to construct a WPRG with arbitrary error ε from a $1/\mathrm{poly}(n)$-PRG. If the original seed length is r, the new WPRG has seed length $r + O(\log(w/\varepsilon)\log\log_n(1/\varepsilon))$ and can be computed in space $O(r+\log\log_n(1/\varepsilon)\cdot(\log\log(w/\varepsilon))^2)$ if the original PRG can be computed in space $O(r)$. Furthermore, the WPRG is $\mathrm{poly}(w)$-bounded. Since all the added costs can be safely neglected, we have the following:

Theorem 4.1.7 ([Hoz21a])**.** *There exists a* $\mathrm{poly}(w)$*-bounded WPRG for* (w, n, d)*-ROBPs with error* $\varepsilon = 1/\mathrm{poly}(w)$ *and seed length*

$$\ell = O\left(d + \frac{\log(wn)\log n}{\max\{1, \log\log w - \log\log n\}}\right).$$

And now combining with Theorem 3.4.2, we obtain the result promised by Corollary 3.4.4:

Theorem 4.1.8.

$$\mathbf{BPL} \subseteq \mathbf{DSPACE}\left(\frac{\log^{3/2} N}{\sqrt{\log\log N}}\right).$$

4.2 Potential and Limits of Improvement in Parameters

We now generalize Armoni's PRG in the last section, noticing that for the recursive construction to work, $\hat{G}$ is only required to have the "read-once" property. Thus, we may speculate whether we could use functions other than extractors to construct G, and calculate the corresponding seed length along with its derandomization consequences.

We start by generalizing Armoni's recursive construction.

Theorem 4.2.1. *For any $k, t > 0$, let $E : \{0,1\}^k \times \{0,1\}^t \to \{0,1\}^m$ be a function, where $m = \Theta(k)$ and $t = O(k)$. For $w \geq n > 0,\ r > 0$, suppose there exists functions $k(w,n),\ t(w,n)$ such that when $k = k(w,n),\ t = t(w,n)$, $\hat{G}$ defined as*

$$\hat{G}(x, y_1, \ldots y_n) \stackrel{\text{def}}{=} E(x, y_1) \circ \cdots \circ E(x, y_n), \quad x \in \{0,1\}^k,\ y \in \{0,1\}^t$$

is a $1/\text{poly}(n)$-PRG for (w,n,r)-ROBPs (assuming $r < m = \Theta(k)$).

Then, using the same construction as in Theorem 4.1.6, there exists a $1/poly(n)$-PRG G for (w,n,r)-ROBPs with seed length

$$\ell = O\left(\frac{k \log n}{\max\{1, \log(k/t)\}} + t\right).$$

Proof. The correctness proof is the same as Theorem 4.1.6. For the seed length, to use the same notation as Theorem 4.1.6, $h = \Theta(\frac{\log n}{\log(k/t)})$ levels of recursion is needed, each requiring a seed of k bits, and 1 seed of length t is required at the end. The total seed length is therefore $O(hk + t) = O\left(\frac{k \log n}{\max\{1, \log(k/t)\}} + t\right)$. □

Since we want to apply good PRGs resulting from this construction to the Saks-Zhou framework, it is appropriate to operate under the following assumption on n and w:

Assumption 4.2.2.

$$\log n = \Omega(\sqrt{\log w}).$$

We note that when E is an extractor, then $t = \Theta(\log n)$ suffices for $\hat{G}$ to be a $1/\text{poly}(n)$-PRG. It is reasonable to assume that this is the best t can get for any E, since the extractor already achieves the best possible dependence on k and ε. Therefore,

Assumption 4.2.3.

$$t = \Theta(\log n).$$

Under these assumptions, we must have

$$k \geq t \geq \Theta(\sqrt{\log w}),$$

as we require $t = O(k)$ for the recursion to make sense (recall that we are cutting length $m = \Theta(k)$ blocks into many length t blocks).

We note that if $t \ll k$, then $\log(k/t)$ becomes $\log k$ in the denominator.

Lemma 4.2.4. *If $t = O(k^\alpha)$ for some $\alpha < 1$, then $\ell = O\left(\frac{k \log n}{\log k}\right)$.*

Proof.

$$\ell = O\left(\frac{k \log n}{\max\{1, \log k - \log k^\alpha\}} + k^\alpha\right) = O\left(\frac{k \log n}{\log k}\right).$$

□

Finally, we present calculations that relate k to the derandomization of **BPL**.

Theorem 4.2.5. *Let $k(w, w) = \Theta(m)$. If $m = \Omega((\log w)^{2/3+\alpha})$ for some $\alpha > 0$ and $m = O(\log w)$, then*

$$\mathbf{BPL} \subseteq \mathbf{DSPACE}\left(\frac{\log N \sqrt{m(N)}}{\sqrt{\log \log N}}\right).$$

Else, if $\Theta((\log w)^{1/2+\alpha}) \leq m \leq \Theta((\log w)^{2/3})$ for some $\alpha > 0$, then

$$\mathbf{BPL} \subseteq \mathbf{DSPACE}\left(\frac{\log^2 N}{m(N)}\right).$$

Proof. Let $2^s = w$. Assume the seed length $\ell = O\left(\frac{k \log n}{\log k}\right)$ (i.e., $t = O(k^\alpha)$). Let $\log n = \sqrt{s} \cdot f$ for some f, then by Theorem 3.4.2, a randomized log-space Turing machine with input length poly(w) can be derandomized in space $O\left(\frac{m\sqrt{s}}{\log m} \cdot f + \frac{s\sqrt{s}}{f}\right)$ (by taking $g = \sqrt{s}/f$). By AM-GM, this value is minimized when $f = \sqrt{\frac{s \log m}{s}}$, which yields $O\left(\frac{s\sqrt{m}}{\sqrt{\log m}}\right)$.

If $\Theta(s^{2/3+\alpha}) \leq m \leq \Theta(s)$, then let $\log n = \frac{s\sqrt{\log m}}{\sqrt{m}}$, and we have $t = \Theta(\log n) = O(m^\beta)$ for some $\beta < 1$. Therefore, out assumption on ℓ holds true. Thus, by the previous calculations, a log-space machine can be derandomized in $O\left(\frac{s\sqrt{m}}{\sqrt{\log m}}\right)$ space by applying Theorem 3.4.2. Since $\Theta(s^{2/3+\alpha}) \leq m \leq \Theta(s)$, the denominator can be further simplified asymptotically, yielding $O\left(\frac{s\sqrt{m}}{\sqrt{\log s}}\right)$.

If $\Theta((\log w)^{1/2+\alpha}) \leq m \leq \Theta((\log w)^{2/3})$, then it turns out if $\log n = \frac{s\sqrt{\log m}}{\sqrt{m}}$, then $t = \Theta(\log n) = \omega(m)$, which contradicts $t = O(k)$. Therefore, $O\left(\frac{m\sqrt{s}}{\log m} \cdot f + \frac{s\sqrt{s}}{f}\right)$ must

be optimized under the assumption that $m = \Omega(\log n) = \Omega(\sqrt{s} \cdot f)$. Thus, the maximum value is taken when $\log n = \Theta(m) \iff f = \Theta(m/\sqrt{s})$. Under these conditions, $t = \Theta(m)$, so the denominator in ℓ becomes constant: $\ell = O(k \log n)$. By Theorem 3.4.2, then, a log-space Turing machine can be derandomized in $O(m \cdot m + \frac{s\sqrt{s}}{m/\sqrt{s}}) = O(s^2/m)$ space.

Note, however, we cheated slightly in our proof: the PRG G has $1/\text{poly}(n)$ error instead of $1/\text{poly}(w)$, and the latter is required to apply Theorem 3.4.2. But, as we have mentioned before, turning a $1/\text{poly}(n)$-PRG into a $1/\text{poly}(w)$-WPRG only adds $O(\log w \cdot \log\log w)$ in seed length and space complexity. Therefore, as long as for our optimal choice of n, the seed length $\ell = \omega(\log w \cdot \log\log w)$, we can obtain a $1/\text{poly}(w)$-WPRG with the same asymptotic seed length.

When $\Theta(s^{2/3+\alpha}) \leq m \leq \Theta(s)$, then since we take $\log n = \frac{s\sqrt{\log m}}{\sqrt{m}}$, we have $\ell = \Theta\left(\frac{s\sqrt{m}}{\sqrt{\log m}}\right) = \omega(\log w \cdot \log\log w)$. When $\Theta((\log w)^{1/2+\alpha}) \leq m \leq \Theta((\log w)^{2/3})$, then since we take $\log n = m$, we have $\ell = \Theta(m^2) = \omega(\log w \cdot \log\log w)$. So in both cases, we can obtain a $1/\text{poly}(w)$-WPRG with the same asymptotic seed length, and our conclusion follows. □

We end with a few consequences of this theorem. First, we note that in Armoni's original PRG, $k = \Theta(\log wn)$, which means $m = \Theta(\log w)$. This falls in the first category, and by Theorem 4.2.5, $\mathbf{BPL} \subseteq \mathbf{DSPACE}\left(\frac{\log^{3/2} N}{\sqrt{\log\log N}}\right)$.

However, Theorem 4.2.5 also implies that, under our assumptions, the best derandomization result we may obtain by improving parameters of Armoni's PRG is $\mathbf{BPL} \subseteq \mathbf{L}^{4/3}$. New methods of PRG construction must be devised if we wish to obtain any $o(\log^{4/3} N)$ result.

Chapter 5

Conclusions

In this paper, we clarified the relationship between improvements in PRG seed length and improvements in derandomization of **BPL**. However, an immediate question is whether constructing PRGs with better seed lengths is feasible. Specifically, in Corollary 3.4.3, we saw that "separating" $\log n$ and $\log w$ terms might help us achieve better derandomization. Although the $\log(1/\varepsilon)$ term has been successfully separated using WPRG in successive works of [BCG19], [CL20] and [Hoz21a], our goal still remains elusive. Thus, some questions we can ask for future research is:

- What is the difficulty in obtaining a PRG that separates $\log n$ and $\log w$ terms in its seed length? If we do not care about the seed length being optimal, what might such a construction look like?
- Can we somehow make use of the condition $\log n \approx \sqrt{\log w}$ to construct better PRGs?
- So far all of our PRGs fool ROBPs, while derandomization via Saks-Zhou only requires our PRG to fool FSMs. Can we obtain better PRGs by restricting to FSMs?

Regarding Armoni's PRG, we note that $\hat{G}$ has the crucial property of being read-once. Naturally, it would be fruitful to investigate PRGs with this property in greater depths. Furthermore, we skirted the question of what other functions might be used

in place of extractors to achieve sublinear dependence of k on $\log w$, which is certainly worthy of further investigation.

As we continue to make progress towards proving $\mathbf{BPL} = \mathbf{L}$, it is our hope that reviewing and synthesizing Saks-Zhou and Armoni's techniques offer fresh insights and directions for future research.

Bibliography

[Arm99] Roy Armoni. On the derandomization of space-bounded computations. In *Randomization and Approximation Techniques in Computer Science: Second International Workshop, RANDOM'98 Barcelona, Spain, October 8–10, 1998 Proceedings*, pages 47–59. Springer, 1999.

[BCG19] Mark Braverman, Gil Cohen, and Sumegha Garg. Pseudorandom pseudo-distributions with near-optimal error for read-once branching programs. *SIAM Journal on Computing*, 49(5):STOC18–242, 2019.

[CDR+21] Gil Cohen, Dean Doron, Oren Renard, Ori Sberlo, and Amnon Ta-Shma. Error Reduction for Weighted PRGs Against Read Once Branching Programs. In Valentine Kabanets, editor, *36th Computational Complexity Conference (CCC 2021)*, volume 200 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:17, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[CL20] Eshan Chattopadhyay and Jyun-Jie Liao. Optimal error pseudodistributions for read-once branching programs. *arXiv preprint arXiv:2002.07208*, 2020.

[GUV09] Venkatesan Guruswami, Christopher Umans, and Salil Vadhan. Unbalanced expanders and randomness extractors from parvaresh–vardy codes. *Journal of the ACM (JACM)*, 56(4):1–34, 2009.

[Hoz21a] William M Hoza. Better pseudodistributions and derandomization for space-bounded computation. In *Approximation, Randomization, and Combinato-*

rial Optimization. Algorithms and Techniques (APPROX/RANDOM 2021). Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[Hoz21b] William Michael Hoza. *Derandomizing space-bounded computation via pseudorandom generators and their generalizations*. PhD thesis, The University of Texas at Austin, 2021.

[Hoz22] William M Hoza. Recent progress on derandomizing space-bounded computation. *Bulletin of EATCS*, 138(3), 2022.

[Ind06] Piotr Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *Journal of the ACM (JACM)*, 53(3):307–323, 2006.

[KNW08] Daniel M Kane, Jelani Nelson, and David P Woodruff. Revisiting norm estimation in data streams. *arXiv preprint arXiv:0811.3648*, 2008.

[Nis90] Noam Nisan. Pseudorandom generators for space-bounded computations. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 204–212, 1990.

[SZ99] Michael Saks and Shiyu Zhou. bphspace (s) $\subseteq$ dspace ($s^{3/2}$). *Journal of computer and system sciences*, 58(2):376–403, 1999.