Typically-Correct Derandomization for Small Time and Space

William M. Hoza
Department of Computer Science, University of Texas at Austin, USA
`https://williamhoza.com`
whoza@utexas.edu

Abstract

Suppose a language L can be decided by a bounded-error randomized algorithm that runs in space S and time $n \cdot \mathrm{poly}(S)$. We give a randomized algorithm for L that still runs in space $O(S)$ and time $n \cdot \mathrm{poly}(S)$ that uses only $O(S)$ random bits; our algorithm has a low failure probability on all but a negligible fraction of inputs of each length. As an immediate corollary, there is a deterministic algorithm for L that runs in space $O(S)$ and succeeds on all but a negligible fraction of inputs of each length. We also give several other complexity-theoretic applications of our technique.

2012 ACM Subject Classification Theory of computation → Pseudorandomness and derandomization; Theory of computation → Probabilistic computation; Theory of computation → Complexity classes

Keywords and phrases Derandomization, pseudorandomness, space complexity

Digital Object Identifier 10.4230/LIPIcs.CCC.2019.9

Funding *William M. Hoza*: Supported by the NSF GRFP under Grant DGE-1610403 and by a Harrington Fellowship from UT Austin.

Acknowledgements We thank Michael Forbes, Scott Aaronson, David Zuckerman, Adam Klivans, and Anna Gál for helpful comments on an early draft of this paper. We thank Amnon Ta-Shma, Lijie Chen, Chris Umans, David Zuckerman, Adam Klivans, Anna Gál, Gil Cohen, Shachar Lovett, Oded Goldreich, and Avi Wigderson for helpful discussions.

1 Introduction

1.1 The Power of Randomness When Time and Space Are Limited

A central goal of complexity theory is to understand the relationship between three fundamental resources: time, space, and randomness. Based on a long line of research [42, 9, 6, 29, 20, 37, 25], most complexity theorists believe that randomized decision algorithms can be made deterministic without paying too much in terms of time and space. Specifically, suppose a language L can be decided by a randomized algorithm that runs in time $T = T(n) \geq n$ and space $S = S(n) \geq \log n$. Klivans and van Melkebeek showed that assuming some language in $\mathbf{DSPACE}(n)$ has exponential circuit complexity, there is a deterministic algorithm for L that runs in time $\mathrm{poly}(T)$ and space $O(S)$ [25].[1]

Proving the hypothesized circuit lower bound seems unlikely for the foreseeable future. In the 90s and early 2000s, researchers managed to prove powerful *unconditional* derandomization theorems by focusing on the space complexity of the deterministic algorithm. For example, Nisan and Zuckerman showed that if $S \geq T^{\Omega(1)}$, there is a deterministic algorithm for

[1] More generally, Klivans and van Melkebeek constructed a pseudorandom generator that fools size-T circuits on T input bits under this assumption. The generator has seed length $O(\log T)$ and is computable in $O(\log T)$ space.

L that runs in space $O(S)$ [30].[2] Alas, in the past couple of decades, progress on such general, unconditional derandomization has stalled. Nobody has managed to extend the Nisan-Zuckerman theorem to a larger regime of pairs (T, S), and researchers have been forced to focus on more restricted models of computation.

In this paper, we focus on *highly efficient* randomized algorithms. That is, we consider the case that T and S are both small, such as $T \leq \widetilde{O}(n)$ and $S \leq O(\log n)$.

1.2 Our Results

1.2.1 Reducing the Amount of Randomness to $O(S)$

Suppose $T \leq n \cdot \mathrm{poly}(S)$. For our main result, we give a randomized algorithm for L that still runs in time $n \cdot \mathrm{poly}(S)$ and space $O(S)$ that uses only $O(S)$ random bits. The catch is that our algorithm is only guaranteed to succeed on *most* inputs. The fraction of "bad" inputs of length n is at most 2^{-S^c}, where $c \in \mathbb{N}$ is an arbitrarily large constant. On "good" inputs, our algorithm's failure probability is at most $2^{-S^{1-\alpha}}$, where $\alpha > 0$ is an arbitrarily small constant.

1.2.2 Eliminating Randomness Entirely

From the result described in the preceding paragraph, a deterministic algorithm that runs in space $O(S)$ follows immediately by iterating over all $O(S)$-bit random strings. We can express this theorem in terms of complexity classes using terminology introduced by Kinne et al. for typically-correct algorithms [24]. Suppose L is a language, $\mathbf{C}$ is a complexity class, and $\varepsilon(n)$ is a function. We say that L *is within* ε *of* $\mathbf{C}$ if there is some $L' \in \mathbf{C}$ such that for every n,

$$\Pr_{x \in \{0,1\}^n}[x \in L \Delta L'] \leq \varepsilon(n). \tag{1}$$

If $\mathbf{C}$ and $\mathbf{C}'$ are complexity classes, we say that $\mathbf{C}$ *is within* ε *of* $\mathbf{C}'$ if every language in $\mathbf{C}$ is within ε of $\mathbf{C}'$. In these terms, our result is that

$$\mathbf{BPTISP}(n \cdot \mathrm{poly}(S), S) \text{ is within } 2^{-S^c} \text{ of } \mathbf{DSPACE}(S). \tag{2}$$

Here, $\mathbf{BPTISP}(T, S)$ is the class of languages that can be decided by a bounded-error randomized algorithm that runs in time $O(T(n))$ and space $O(S(n))$, and $\mathbf{DSPACE}(S)$ is the class of languages that can be decided by a deterministic algorithm that runs in space $O(S)$. Note that if $S \geq n^{\Omega(1)}$, the mistake rate in Equation (2) drops below 2^{-n}. Since there are only 2^n inputs of length n, the algorithm must in fact be correct on all inputs. Our result can therefore be viewed as a generalization of the Nisan-Zuckerman theorem $\mathbf{BPTISP}(\mathrm{poly}(S), S) \subseteq \mathbf{DSPACE}(S)$ [30].

1.2.3 Derandomization with Advice

Adleman's argument [1] shows that $\mathbf{BPL} \subseteq \mathbf{L}/\mathrm{poly}$. We study the problem of derandomizing $\mathbf{BPL}$ with as little advice as possible. Goldreich and Wigderson discovered a critical threshold: roughly, if an algorithm can be derandomized with fewer than n bits of advice, then there is a *typically-correct* derandomization of the algorithm with *no* advice [15].[3]

[2] More generally, the Nisan-Zuckerman theorem applies as long as the original randomized algorithm for L uses only $\mathrm{poly}(S)$ random bits, regardless of how much time it takes.

[3] This result also requires that (a) most advice strings are "good", and (b) there is an appropriate efficient extractor.

Motivated by this phenomenon, Fortnow and Klivans proved that $\mathbf{BPL} \subseteq \mathbf{L}/O(n)$ [12]. We refine their argument and show that $\mathbf{BPL} \subseteq \mathbf{L}/(n + O(\log^2 n))$, getting very near the critical threshold of n bits of advice. More interestingly, we show that the connection identified by Goldreich and Wigderson [15] works the other way: in the space-bounded setting, typically-correct derandomizations imply derandomizations with just a little advice. Combining with our main result gives that for every constant $c \in \mathbb{N}$,

$$\mathbf{BPTISP}(\widetilde{O}(n), \log n) \subseteq \mathbf{L}/(n - \log^c n). \tag{3}$$

1.2.4 Derandomizing Turing Machines

All algorithms in the results mentioned so far are formulated in a general *random-access* model, i.e., the algorithm can read any specified bit of its input in a single step. (See Section 2.2 for details.) We also study the weaker *multitape Turing machine* model. The main weakness of the Turing machine model is that if its read head is at position i of its input and it wishes to read bit j of its input, it must spend $|i - j|$ steps moving its read head to the appropriate location. Let $\mathbf{BPTISP}_{\mathrm{TM}}(T, S)$ denote the class of languages that can be decided by a bounded-error randomized Turing machine that runs in time $O(T(n))$ and space $O(S(n))$.

1.2.4.1 Beyond Linear Advice

We give a typically-correct derandomization for $\mathbf{BPTISP}_{\mathrm{TM}}$ analogous to our main result but with a lower mistake rate. In terms of advice, our derandomization implies that for every constant $c \in \mathbb{N}$,

$$\mathbf{BPTISP}_{\mathrm{TM}}(\widetilde{O}(n), \log n) \subseteq \mathbf{L}/O\left(\frac{n}{\log^c n}\right). \tag{4}$$

Equation (4) gives an interesting example of a class of $\mathbf{BPL}$ algorithms that can be derandomized with $o(n)$ bits of advice.

1.2.4.2 Beyond Quasilinear Time

Using different techniques, we also show how to derandomize log-space Turing machines that use almost a *quadratic* amount of time. In particular, we show that if $TS^2 \leq o(n^2/\log n)$, then

$$\mathbf{BPTISP}_{\mathrm{TM}}(T, S) \text{ is within } o(1) \text{ of } \mathbf{DTISP}(\mathrm{poly}(n), S). \tag{5}$$

1.2.5 Disambiguating Nondeterministic Algorithms

For some of our derandomization results, we give analogous theorems regarding *unambiguous* simulations of *nondeterministic* algorithms. We defer a discussion of these results to Section 6.

1.3 Techniques

1.3.1 "Out of Sight, out of Mind"

Our typically-correct derandomizations work by treating the *input as a source of randomness*. This idea was pioneered by Goldreich and Wigderson [15]. For the sake of discussion, let $\mathcal{A}$ be a randomized algorithm that uses n random bits. A naïve strategy for derandomizing $\mathcal{A}$

is to run $\mathcal{A}(x, x)$. Most random strings of $\mathcal{A}$ lead to the right answer, so it is tempting to think that for most x, $\mathcal{A}(x, x)$ will give the right answer. This reasoning is flawed, because $\mathcal{A}$ might behave poorly when its input is *correlated* with its random bits.

In this work, we avoid these troublesome correlations using a simple idea embodied by the adage "out of sight, out of mind." We use *part* of the input as a source of randomness while $\mathcal{A}$ is processing *the rest* of the input.

To go into more detail, suppose $\mathcal{A}$ runs in time $\widetilde{O}(n)$ and space $O(\log n)$. Our randomness-efficient simulation of $\mathcal{A}$ operates in $\operatorname{polylog}(n)$ *phases*. At the beginning of a new phase, we pick a random $\operatorname{polylog}(n)$-bit block $x|_I$ of the input x. We apply a seeded extractor to $x|_I$, giving a string of length $\Theta(\log^2 n)$. We apply Nisan's pseudorandom generator for space-bounded computation [26], giving a pseudorandom string of length $\widetilde{O}(n)$. We use the pseudorandom string to run the simulation of $\mathcal{A}$ forward until it tries to read from $x|_I$, at which time we pause the simulation of $\mathcal{A}$ and move on to the next phase.

The key point is that the output of the extractor is processed without ever looking at $x|_I$, the input to the extractor. Extractors are good *samplers* [44], and $\mathcal{A}$ only has polynomially many possible configurations, so for most x, the output of the extractor is essentially as good as a uniform random seed to Nisan's generator. Therefore, in each phase, with high probability, we successfully simulate $n/\operatorname{polylog}(n)$ steps of $\mathcal{A}$ before it reads from $x|_I$ and we have to move on to the next phase. Thus, with high probability, after $\operatorname{polylog}(n)$ phases, the simulation of $\mathcal{A}$ is complete.

Each bit of the output of Nisan's generator can be computed in time[4] $\operatorname{polylog}(n)$ and space $O(\log n)$. Therefore, our simulation of $\mathcal{A}$ still runs in time $\widetilde{O}(n)$ and space $O(\log n)$, but now it uses just $\operatorname{polylog}(n)$ random bits ($O(\log n)$ random bits per phase to pick the random block I and to pick a seed for the extractor).

The reader may wonder whether we could have achieved the same effect by simply directly applying Nisan's generator from the start – its seed length is $\operatorname{polylog}(n)$, after all. The point is that Nisan's generator requires *two-way* access to its seed, whereas our simulation only uses one-way access to its random bits. During our simulation, we are able to give Nisan's generator two-way access to its seed, because we have two-way access to the *input* x from which we extract that seed.

Finally, because our simulation reads its $\operatorname{polylog}(n)$ random bits from left to right, we can further reduce the number of random bits to just $O(\log n)$ by applying the Nisan-Zuckerman pseudorandom generator [30].

1.3.2 Other Techniques

Our derandomizations with advice are based on Fortnow and Klivans' technique for proving $\mathbf{BPL} \subseteq \mathbf{L}/O(n)$ [12] and Nisan's technique for proving $\mathbf{RL} \subseteq \mathbf{SC}$ [28]. Our derandomization of $\mathbf{BPTISP}_{\mathrm{TM}}$ with a low mistake rate uses a similar "out of sight, out of mind" technique as our main result. The lower mistake rate is achieved by exploiting knowledge of the region of the input that will be processed in the near future, based on the locality of the Turing machine's read head. Our derandomization of $\mathbf{BPTISP}_{\mathrm{TM}}(T, S)$ for $T(n) \approx n^2$ is based on a seed-extending pseudorandom generator for multiparty communication protocols by Kinne et al. [24].

[4] See work by Diehl and van Melkebeek [11] for an even faster implementation of Nisan's generator.

1.4 Related Work

We will only mention some highlights of the large body of research on unconditional derandomization of time- and space-bounded computation. Fix $L \in \mathbf{BPTISP}(T, S)$. Nisan gave a randomized algorithm for L that runs in time $\mathrm{poly}(T)$ and space $O(S \log T)$ that uses only $O(S \log T)$ random bits [26]. Nisan also gave a deterministic algorithm for L that runs in time $2^{O(S)}$ and space $O(S \log T)$ [28]. Nisan and Zuckerman gave a randomized algorithm for L that runs in time $\mathrm{poly}(T)$ and space $O(S + T^{\varepsilon})$ that uses only $O(S + T^{\varepsilon})$ random bits, where $\varepsilon > 0$ is an arbitrarily small constant [30] (this is a generalization of the result mentioned in Section 1.1). Saks and Zhou gave a deterministic algorithm for L that runs in space $O(S\sqrt{\log T})$ [32]. Combining the techniques from several of these works, Armoni [4] gave a deterministic algorithm for L that runs in space[5]

$$O\left(S \cdot \sqrt{\frac{\log T}{\max\{1, \log S - \log\log T\}}}\right). \tag{6}$$

Armoni's algorithm remains the most space-efficient derandomization known for all T and S. When $T = \widetilde{\Theta}(n)$ and $S = \Theta(\log n)$, Armoni's algorithm runs in space $\Theta(\log^{3/2} n)$, just like the earlier Saks-Zhou algorithm [32]. Cai et al. gave a time-space tradeoff [10] interpolating between Nisan's deterministic algorithm [28] and the Saks-Zhou algorithm [32].

All of the preceding results apply, *mutatis mutandis*, to derandomizing algorithms that use at most T random bits, regardless of how much time they take. In contrast, our proofs crucially rely on the fact that a time-T algorithm queries its *input* at most T times. This aspect of our work is shared by work by Beame et al. [8] on time-space lower bounds.

Goldreich and Wigderson's idea of using the input as a source of randomness for a typically-correct derandomization [15] has been applied and developed by several researchers [5, 41, 23, 43, 35, 24, 33, 3]; see related survey articles by Shaltiel [34] and by Hemaspaandra and Williams [19]. Researchers have proven unconditional typically-correct derandomization results for several restricted models, including sublinear-time algorithms [43, 35], communication protocols [35, 24], constant-depth circuits [35, 24], and streaming algorithms [35]. On the other hand, Kinne et al. proved that any typically-correct derandomization of **BPP** with a sufficiently low mistake rate would imply strong circuit lower bounds [24]. We are the first to study typically-correct derandomization for algorithms with simultaneous bounds on time and space.

1.5 Outline of This Paper

In Section 2, we discuss random-access models of computation and extractors. In Section 3, we give our derandomization of $\mathbf{BPTISP}(n \cdot \mathrm{poly}(S), S)$. In Section 4, we give our two derandomizations of $\mathbf{BPTISP}_{\mathrm{TM}}(T, S)$. In Section 5, we discuss derandomization with advice. Section 6 concerns disambiguation of nondeterministic algorithms, and we conclude in Section 7 with some suggested directions for further research.

[5] Actually, the space bound given in Equation (6) is achieved by using better extractors than were known when Armoni wrote his paper [4, 22].

2 Preliminaries

2.1 General Notation

Strings

For strings x, y, let $x \circ y$ denote the concatenation of x with y. For a natural number n, let $[n] = \{1, 2, \ldots, n\}$. For a string $x \in \{0,1\}^n$ and a set $I = \{i_1 < i_2 < \cdots < i_\ell\} \subseteq [n]$, let $x|_I = x_{i_1} x_{i_2} \ldots x_{i_\ell} \in \{0,1\}^\ell$.

Sets

For a finite set X, we will use the notations $\#X$ and $|X|$ interchangeably to refer to the number of elements of X. For $X \subseteq \{0,1\}^n$, let $\text{density}(X) = |X|/2^n$. We will sometimes omit the parentheses, e.g., $\text{density}\{000, 111\} = 0.25$. We identify a language $L \subseteq \{0,1\}^*$ with its indicator function $L : \{0,1\}^* \to \{0,1\}$, i.e.,

$$L(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{if } x \notin L. \end{cases} \tag{7}$$

Probability

If X and Y are probability distributions on the same space, we write $X \sim_\varepsilon Y$ to indicate that X and Y are ε-close in total variation distance. For $T \in \mathbb{N}$, let U_T denote the uniform distribution over $\{0,1\}^T$.

2.2 Random-Access Algorithms

Our main theorems govern general *random-access algorithms.* Our results are not sensitive to the specific choice of model of random-access computation. For concreteness, following Fortnow and van Melkebeek [13], we will work with the *random-access Turing machine* model. This model is defined like the standard multitape Turing machine model, except that each ordinary tape is supplemented with an "index tape" that can be used to move the ordinary tape's head to an arbitrary specified location in a single step. See the paper by Fortnow and van Melkebeek [13] for details.

A *randomized random-access Turing machine* is a random-access Turing machine equipped with an additional read-only tape, initialized with random bits, that can only be read from *left to right.* Thus, if the algorithm wishes to reread old random bits, it needs to have copied them to a work tape, which counts toward the algorithm's space usage. The random tape does not have a corresponding index tape.

For functions $T : \mathbb{N} \to \mathbb{N}$ and $S : \mathbb{N} \to \mathbb{N}$, we define $\mathbf{BPTISP}(T, S)$ to be the class of languages L such that there is a randomized random-access Turing machine $\mathcal{A}$ such that on input $x \in \{0,1\}^n$, $\mathcal{A}(x)$ always halts in time $O(T(n))$, $\mathcal{A}(x)$ always touches $O(S(n))$ cells on all of its read-write tapes, and $\Pr[\mathcal{A}(x) = L(x)] \geq 2/3$.

2.3 Randomized Branching Programs

Our algorithms are most naturally formulated in terms of branching programs, a standard *nonuniform* model of time- and space-bounded computation. Recall that in a digraph, a *terminal vertex* is a vertex with no outgoing edges. In the following definition, n is the number of input bits and m is the number of random bits.

▶ **Definition 1.** *A* randomized branching program *on* $\{0,1\}^n \times \{0,1\}^m$ *is a directed acyclic graph, where each nonterminal vertex* v *is labeled with two indices* $i(v) \in [n], j(v) \in [m]$ *and has four outgoing edges labeled with the four two-bit strings. If* $\mathcal{P}$ *is a randomized branching program, we let* $V(\mathcal{P})$ *be the set of vertices of* $\mathcal{P}$*.*

The interpretation is that from vertex v, the program follows the edge labeled $x_{i(v)}y_{j(v)}$, where x is the input and y is the random string. This interpretation is formalized by the following definition, which sets $\mathcal{P}(v; x, y)$ to be the vertex reached from v on input x using randomness y.

▶ **Definition 2.** *Suppose* $\mathcal{P}$ *is a randomized branching program on* $\{0,1\}^n \times \{0,1\}^m$*. We identify* $\mathcal{P}$ *with a function* $\mathcal{P} : V(\mathcal{P}) \times \{0,1\}^n \times \{0,1\}^m \to V(\mathcal{P})$ *defined as follows. Fix* $v \in V(\mathcal{P}), x \in \{0,1\}^n, y \in \{0,1\}^m$*. Take a walk through* $\mathcal{P}$ *by starting at* v *and, having reached vertex* u*, following the edge labeled* $x_{i(u)}y_{j(u)}$*. Then* $\mathcal{P}(v; x, y)$ *is the terminal vertex reached by this walk.*

As previously discussed, random-access Turing machines can only access their random bits from left to right. This corresponds to an *R-OW randomized branching program.*

▶ **Definition 3.** *An* R-OW randomized branching program *is a randomized branching program* $\mathcal{P}$ *such that for every edge* (v, v') *between two nonterminal vertices,* $j(v') \in \{j(v), j(v) + 1\}$*.*

The term "R-OW" indicates that the branching program has "random access" to its input bits and "one-way access" to its random bits.

The *size* of a branching program is defined as $\text{size}(\mathcal{P}) = |V(\mathcal{P})|$. The *length* of the program, $\text{length}(\mathcal{P})$, is defined to be the length of the longest path through the program. Observe that $\mathbf{BPTISP}(T, S)$ corresponds to R-OW randomized branching programs of size $2^{O(S)}$ and length $O(T)$.

Many of our algorithms will use a *restriction* operation that we now introduce.

▶ **Definition 4.** *Suppose* $\mathcal{P}$ *is a randomized branching program on* $\{0,1\}^n$ *and* $I \subseteq [n]$*. Let* $\mathcal{P}|_I$ *be the program obtained from* $\mathcal{P}$ *by deleting all outgoing edges from vertices* v *such that* $i(v) \notin I$*.*

So in $\mathcal{P}|_I$, there are two types of terminal vertices: vertices that were terminal in $\mathcal{P}$, and vertices v that are now terminal because $i(v) \notin I$. The computation $\mathcal{P}|_I(v; x, y)$ halts when it reaches either type of terminal vertex. Thus, $\mathcal{P}|_I(v; x, y)$ does not depend on $x|_{[n]\setminus I}$, because $\mathcal{P}|_I(v; x, y)$ outputs the vertex reached by running the computation $\mathcal{P}(v; x, y)$ until it finishes or it tries to read from $x|_{[n]\setminus I}$.

2.4 Extractors

Recall that a (k, ε)*-extractor* is a function $\mathsf{Ext} : \{0,1\}^\ell \times \{0,1\}^d \to \{0,1\}^s$ such that if X has "min-entropy" at least k and $Y \sim U_d$ is independent of X, then $\mathsf{Ext}(X, Y) \sim_\varepsilon U_s$. It can be shown nonconstructively that for every ℓ, k, ε, there exists Ext with $d \leq \log(\ell - k) + 2\log(1/\varepsilon) + O(1)$ and $s \geq k + d - 2\log(1/\varepsilon) - O(1)$ (see, e.g., Vadhan's monograph [38]).

We will need a computationally efficient extractor. The extractor literature has mainly focused on the time complexity of computing extractors, but we are concerned with space complexity, too. This paper is not meant to be about extractor constructions, so we encourage the reader to simply pretend that optimal extractors can be computed in a single step with no space overhead. In actuality, we will use two incomparable non-optimal extractors.

To prove our main results, we will use an extractor by Shaltiel and Umans [36]. The benefit of the Shaltiel-Umans extractor is that it allows for small error ε.

▶ **Theorem 5** ([36]). *Fix a constant $\alpha > 0$. For every $\ell, k \in \mathbb{N}, \varepsilon > 0$ such that $k \geq \log^{4/\alpha} \ell$ and $k \geq \log^{4/\alpha}(1/\varepsilon)$, there is a (k, ε)-extractor* $\mathsf{SUExt} : \{0,1\}^\ell \times \{0,1\}^d \to \{0,1\}^s$ *where* $d \leq O\left(\log \ell + \frac{\log \ell \log(1/\varepsilon)}{\log k}\right)$ *and $s \geq k^{1-\alpha}$. Given x, y, k, and ε,* $\mathsf{SUExt}(x, y)$ *can be computed in time* $\mathrm{poly}(\ell)$ *and space $O(d)$.*

To derandomize **BPL** with as little advice as possible, we will use an extractor by Guruswami, Umans, and Vadhan [16] (not the most famous extractor from their work, but a slight variant). The benefit of the GUV extractor is that it outputs a constant fraction of the entropy.

▶ **Theorem 6** ([16]). *Let $\alpha, \varepsilon > 0$ be constant. For every $\ell, k \in \mathbb{N}$, there is a (k, ε)-extractor* $\mathsf{GUVExt} : \{0,1\}^\ell \times \{0,1\}^d \to \{0,1\}^s$ *with $s \geq (1-\alpha)k$ and $d \leq O(\log \ell)$ such that given x and y,* $\mathsf{GUVExt}(x, y)$ *can be computed in $O(\log \ell)$ space.*

In both cases, the original authors [36, 16] did not explicitly analyze the space complexity of their extractors, so we explain in Appendices A and B why these extractors can be implemented in small space. (We remark that Hartman and Raz also constructed small-space extractors [18], but the seed lengths of their extractors are too large for us.)

2.4.1 Extractors as Samplers

We will actually only be using extractors for their *sampling* properties. The connection between extractors and samplers was first discovered by Zuckerman [44]. The following standard proposition expresses this connection for non-Boolean functions.

▶ **Proposition 7** ([44]). *Suppose* $\mathsf{Ext} : \{0,1\}^\ell \times \{0,1\}^d \to \{0,1\}^s$ *is a (k, ε)-extractor and $f : \{0,1\}^s \to V$ is a function. Let $\delta = \varepsilon |V|/2$. Then*

$$\#\{x \in \{0,1\}^\ell : f(U_s) \not\sim_\delta f(\mathsf{Ext}(x, U_d))\} \leq 2^{k+1}|V|. \tag{8}$$

For completeness, we include a proof of Proposition 7 in Appendix C, since the specific statement of Proposition 7 does not appear in Zuckerman's paper [44].

2.5 Constructibility

We say that $f : \mathbb{N} \to \mathbb{N}$ is *constructible* in space $S(n)$, time $T(n)$, etc. if there is a deterministic random-access Turing machine $\mathcal{A}$ that runs in the specified resource bounds with $\mathcal{A}(1^n) = f(n)$, written in binary. As usual, we say that f is *space constructible* if f is constructible in space $O(f(n))$. We say that $\delta : \mathbb{N} \to [0,1]$ is constructible in specified resource bounds if δ can be written as $\delta(n) = \frac{\delta_1(n)}{\delta_2(n)}$, where $\delta_1, \delta_2 : \mathbb{N} \to \mathbb{N}$ are both constructible in the specified resource bounds.

3 Derandomizing Efficient Random-Access Algorithms

3.1 Main Technical Algorithm: Low-Randomness Simulation of Branching Programs

Suppose $\mathcal{P}$ is an R-OW randomized branching program on $\{0,1\}^n \times \{0,1\}^T$ of length T and size 2^S. (As a reminder, such a program models $\mathbf{BPTISP}(T, S)$.) Given $\mathcal{P}$, v_0, and x, the distribution $\mathcal{P}(v_0; x, U_T)$ can trivially be sampled in time $T \cdot \mathrm{poly}(S)$ and space $O(S)$ using T random bits. Our main technical result is an efficient typically-correct algorithm for approximately sampling $\mathcal{P}(v_0; x, U_T)$ using roughly T/n random bits.

▶ **Theorem 8.** *For each constant $c \in \mathbb{N}$, there is a randomized algorithm* A *with the following properties. Suppose $\mathcal{P}$ is an R-OW randomized branching program on $\{0,1\}^n \times \{0,1\}^T$ with $S \geq \log n$, where $S \overset{\text{def}}{=} \lceil \log \operatorname{size}(\mathcal{P}) \rceil$. Suppose $v_0 \in V(\mathcal{P})$, $T \geq \operatorname{length}(\mathcal{P})$, and $x \in \{0,1\}^n$. Then $\mathsf{A}(\mathcal{P}, v_0, x, T)$ outputs a vertex $v \in V(\mathcal{P})$ in time*[6] *$T \cdot \operatorname{poly}(S)$ and space $O(S)$ using $\lceil T/n \rceil \cdot \operatorname{poly}(S)$ random bits. Finally, for every such $\mathcal{P}, v_0, T$,*

$$\operatorname{density}\{x \in \{0,1\}^n : \mathsf{A}(\mathcal{P}, v_0, x, T) \not\sim_{\exp(-cS)} \mathcal{P}(v_0; x, U_T)\} \leq 2^{-S^c}. \tag{9}$$

The algorithm of Theorem 8 relies on Nisan's pseudorandom generator [26]. The seed length of Nisan's generator is not $O(S)$, but Nisan's generator does *run* in space $O(S)$, given two-way access to the seed.

▶ **Theorem 9** ([26]). *For every $S, T \in \mathbb{N}, \varepsilon > 0$ with $T \leq 2^S$, there is a generator* NisGen : *$\{0,1\}^s \to \{0,1\}^T$ with seed length $s \leq O((S + \log(1/\varepsilon)) \cdot \log T)$, such that if $\mathcal{P}$ is an R-OW randomized branching program of size 2^S, v is a vertex, and x is an input, then*

$$\mathcal{P}(v; x, \mathsf{NisGen}(U_s)) \sim_\varepsilon \mathcal{P}(v; x, U_T). \tag{10}$$

Given S, T, ε, z, i, the bit $\mathsf{NisGen}(z)_i$ can be computed in time $\operatorname{poly}(S, \log(1/\varepsilon))$ and space $O(S + \log(1/\varepsilon))$.

Algorithm 1: The algorithm A of Theorem 8.

```
if $S^{c+1} > \lfloor n/9 \rfloor$ then
    Directly simulate $P(v_0; x, U_T)$ using $T$ random bits
else
    Let $I_1, I_2, \ldots, I_B \subseteq [n]$ be disjoint sets of size $S^{c+1}$ with $B$ as large as possible
    Initialize $v \leftarrow v_0$
    repeat $r$ times                          /* $r$ is given by Equation (11) */
        Pick $b \in [B]$ uniformly at random and let $I \leftarrow I_b$
        Pick $y \in \{0,1\}^{O(S)}$ uniformly at random
        Update $v \leftarrow \mathcal{P}|_{[n]\setminus I}(v; x, \mathsf{NisGen}(\mathsf{SUExt}(x|_I, y)))$
    end
    return $v$
end
```

For Theorem 8, we can replace T with $\min\{T, 2^S\}$ without loss of generality, so we will assume that $T \leq 2^S$. The algorithm A is given in Algorithm 1.

Parameters

Set

$$r \overset{\text{def}}{=} \max\left\{\left\lceil \frac{8T}{B-8} \right\rceil, 8(cS+1)\right\} = \lceil T/n \rceil \cdot \operatorname{poly}(S). \tag{11}$$

The parameter r is the number of "phases" of A as outlined in Section 1.3.1. Note that if $S^{c+1} \leq \lfloor n/9 \rfloor$, then $B \geq 9$, so Equation (11) makes sense. Naturally, Nisan's generator

[6] The graph of $\mathcal{P}$ should be encoded in adjacency list format, so that the neighborhood of a vertex v can be computed in $\operatorname{poly}(S)$ time.

NisGen is instantiated with the parameters S, T from the statement of Theorem 8. The error of NisGen is set at

$$\varepsilon \overset{\text{def}}{=} \frac{e^{-cS}}{4r} = 2^{-\Theta(S)}. \tag{12}$$

That way, the seed length of NisGen is $s \leq O(S \log T) \leq O(S^2)$. The algorithm A also relies on the Shaltiel-Umans extractor SUExt of Theorem 5. This extractor is instantiated with source length $\ell \overset{\text{def}}{=} S^{c+1}$, $\alpha \overset{\text{def}}{=} 2/3$, error

$$\varepsilon' \overset{\text{def}}{=} \frac{e^{-cS}}{2r \cdot 2^S} = 2^{-\Theta(S)}, \tag{13}$$

and entropy

$$k \overset{\text{def}}{=} \max\{s^3, \log^6 \ell, \log^6(1/\varepsilon)\} = \Theta(S^6). \tag{14}$$

Our choice of k explicitly meets the hypotheses of Theorem 5, and by construction, $k^{1-\alpha} \geq s$, so we can think of SUExt as outputting s bits.

Efficiency

We now analyze the computational efficiency of A. First, we bound the running time. If $S^{c+1} > \lfloor n/9 \rfloor$, then A clearly runs in time $T \cdot \text{poly}(S)$. Otherwise, A repeatedly replaces v with one of its neighbors a total of at most T times, since $T \geq \text{length}(\mathcal{P})$. Each such step requires computing a bit of Nisan's generator, which takes time $\text{poly}(S)$, times $\text{poly}(S)$ steps to compute each bit of the seed of Nisan's generator by running SUExt. Thus, overall, A runs in time $T \cdot \text{poly}(S)$.

Next, we bound the space complexity of A. If $S^{c+1} > \lfloor n/9 \rfloor$, then A clearly runs in space $O(S + \log T) = O(S)$. Otherwise, space is required to store a loop index ($O(\log r)$ bits), the vertex v ($O(S)$ bits), the index b ($O(\log n)$ bits), and the seed y ($O(S)$ bits). These terms are all bounded by $O(S)$. Running SUExt takes $O(\log \ell + \frac{\log \ell \log(1/\varepsilon')}{\log k})$ bits of space. Since $k \geq S^{\Omega(1)}$, $\frac{\log \ell}{\log k} \leq O(1)$, and hence the space used for SUExt is only $O(\log S + \log(1/\varepsilon')) = O(S)$. Finally, running NisGen takes $O(S + \log(1/\varepsilon)) = O(S)$ bits of space. Therefore, overall, A runs in space $O(S)$.

Finally, we bound the number of random bits used by A. If $S^{c+1} > \lfloor n/9 \rfloor$, then A uses T random bits, which is at most $\frac{9T(1+S^{c+1})}{n}$ in this case. Otherwise, in each iteration of the loop, A uses $O(\log n)$ random bits for b, plus $O(S)$ random bits for y. Therefore, overall, the number of random bits used by A is $O(rS)$, which is $\lceil T/n \rceil \cdot \text{poly}(S)$.

Correctness

We now turn to proving Equation (9). If $S^{c+1} > \lfloor n/9 \rfloor$, then obviously $\mathsf{A}(\mathcal{P}, v_0, x, T) \sim \mathcal{P}(v_0; x, U_T)$. Assume, therefore, that $S^{c+1} \leq \lfloor n/9 \rfloor$. The proof will be by a hybrid argument with three hybrid distributions. The first hybrid distribution is defined by the algorithm H_1 given by Algorithm 2.

We need a standard fact about Markov chains. Suppose M and M' are stochastic matrices (i.e., each row is a probability vector) of the same size. We write $M \sim_\gamma M'$ to mean that for each row index i, the probability distributions M_i and M'_i are γ-close in total variation distance.

▶ **Lemma 10.** *If $M \sim_\gamma M'$, then $M^r \sim_{\gamma r} (M')^r$.*

Algorithm 2: The algorithm H_1 defining the first hybrid distribution used to prove Equation (9). The only difference between A and H_1 is that H_1 picks a uniform random seed for NisGen, instead of extracting the seed from the input.

```
Initialize $v \leftarrow v_0$
repeat $r$ times
    Pick $b \in [B]$ uniformly at random and let $I \leftarrow I_b$
    Pick $y' \in \{0,1\}^s$ uniformly at random
    Update $v \leftarrow \mathcal{P}|_{[n]\setminus I}(v; x, \mathsf{NisGen}(y'))$
end
return $v$
```

For a proof of Lemma 10, see, e.g., work by Saks and Zhou [32, Proposition 2.3]. We are now ready to prove that for most x, the behavior of A is statistically similar to the behavior of H_1.

▷ Claim 11 ($\mathsf{A} \approx \mathsf{H}_1$). Let $\delta = \varepsilon' \cdot r \cdot 2^{S-1} = 2^{-\Theta(S)}$. Then

$$\text{density}\{x \in \{0,1\}^n : \mathsf{A}(\mathcal{P}, v_0, x, T) \not\sim_\delta \mathsf{H}_1(\mathcal{P}, v_0, x, T)\} \le 2^{-S^c}. \tag{15}$$

Proof. Fix any $b \in [B]$ and $v \in V(\mathcal{P})$. Let $I = I_b$, and fix any $x' \in \{0,1\}^n$ with $x'|_I = 0^{|I|}$. Define $f : \{0,1\}^s \to V$ by

$$f(y') = \mathcal{P}|_{[n]\setminus I}(v; x', \mathsf{NisGen}(y')). \tag{16}$$

By Proposition 7,

$$\#\{x|_I \in \{0,1\}^\ell : f(\mathsf{SUExt}(x|_I, U_d)) \not\sim_{\varepsilon' 2^{S-1}} f(U_s)\} \le 2^{k+S+1}. \tag{17}$$

Therefore,

$$\#\{x \in \{0,1\}^n : x|_{[n]\setminus I} = x'|_{[n]\setminus I} \text{ and } f(\mathsf{SUExt}(x|_I, U_d)) \not\sim_{\varepsilon' 2^{S-1}} f(U_s)\} \le 2^{k+S+1}. \tag{18}$$

Now, let $M[x]$ be the $\text{size}(\mathcal{P}) \times \text{size}(\mathcal{P})$ stochastic matrix defined by

$$M[x]_{uv} = \Pr_{b,y}[\mathcal{P}|_{[n]\setminus I}(u; x, \mathsf{NisGen}(\mathsf{SUExt}(x|_I, y))) = v \text{ where } I = I_b]. \tag{19}$$

Let $M'[x]$ be the stochastic matrix defined by

$$M'[x]_{uv} = \Pr_{b,y'}[\mathcal{P}|_{[n]\setminus I}(u; x, \mathsf{NisGen}(y')) = v \text{ where } I = I_b]. \tag{20}$$

By summing over all b, v, x', we find that

$$\#\{x \in \{0,1\}^n : M[x] \not\sim_{\varepsilon' 2^{S-1}} M'[x]\} \le B \cdot 2^S \cdot 2^{n-\ell} \cdot 2^{k+S+1} \tag{21}$$
$$\le 2^{n - S^{c+1} + O(S^6)} \tag{22}$$
$$\le 2^{n - S^c}, \tag{23}$$

assuming $c \ge 6$ and n is sufficiently large. If $M[x] \sim_{\varepsilon' 2^{S-1}} M'[x]$, then by Lemma 10, $M[x]^r \sim_\delta M'[x]^r$. The output of A is a sample from $(M[x]^r)_{v_0}$ and the output of H_1 is a sample from $(M'[x]^r)_{v_0}$, completing the proof. ◁

The second hybrid distribution is defined by the algorithm H_2 given by Algorithm 3.

Algorithm 3: The algorithm H_2 defining the second hybrid distribution used to prove Equation (9). The only difference between H_1 and H_2 is that H_2 feeds true randomness to $\mathcal{P}|_{[n]\setminus I}$, instead of feeding it a pseudorandom string from Nisan's generator.

```
Initialize $v \leftarrow v_0$
repeat $r$ times
    Pick $b \in [B]$ uniformly at random and let $I \leftarrow I_b$
    Pick $y'' \in \{0,1\}^T$ uniformly at random
    Update $v \leftarrow \mathcal{P}|_{[n]\setminus I}(v; x, y'')$
end
return $v$
```

Algorithm 4: The algorithm H_3 defining the third hybrid distribution used to prove Equation (9). The only difference between H_2 and H_3 is that H_2 terminates after r iterations, whereas H_3 waits until it reaches a terminal vertex of $\mathcal{P}$.

```
Initialize $v \leftarrow v_0$
while $v$ is not a terminal vertex of $\mathcal{P}$ do
    Pick $b \in [B]$ uniformly at random and let $I \leftarrow I_b$
    Pick $y'' \in \{0,1\}^T$ uniformly at random
    Update $v \leftarrow \mathcal{P}|_{[n]\setminus I}(v; x, y'')$
end
return $v$
```

▷ Claim 12 ($\mathsf{H}_1 \approx \mathsf{H}_2$). For every x,

$$\mathsf{H}_1(\mathcal{P}, v_0, x, T) \sim_{\varepsilon r} \mathsf{H}_2(\mathcal{P}, v_0, x, T). \tag{24}$$

Proof. This follows immediately from the correctness of NisGen and an application of Lemma 10 that is perfectly analogous to the reasoning used to prove Claim 11. ◁

Next, we must show that the output of H_2 is statistically close to the output of H_3. The idea is that in each iteration, with high probability, H_2 progresses by roughly B steps before running into a vertex v with $i(v) \in I$. (Recall that $i(v)$ is the index of the input queried by vertex v.) Therefore, in total, with high probability, H_2 progresses roughly rB steps, which is at least T by our choice of r. We now give the detailed statement and proof.

▷ Claim 13 ($\mathsf{H}_2 \approx \mathsf{H}_3$). For every x,

$$\mathsf{H}_2(\mathcal{P}, v_0, x, T) \sim_{\exp(-r/8)} \mathsf{H}_3(\mathcal{P}, v_0, x, T). \tag{25}$$

Proof. Consider iteration t of the loop in H_2, where $1 \le t \le r$. Let T_t be the number of steps through $\mathcal{P}|_{[n]\setminus I}$ that are taken in iteration t when updating $v = \mathcal{P}|_{[n]\setminus I}(v; x, y'')$ before reaching a vertex that tries to query from I. (If we never reach such a vertex, i.e., we reach a terminal vertex of $\mathcal{P}$, then let $T_t = T$.) We claim that

$$\Pr\left[\sum_{t=1}^{r} T_t < T\right] \le e^{-r/8}. \tag{26}$$

Proof: For $t \in [r]$, consider the value of v at the beginning of iteration t and the string $y'' \in \{0,1\}^T$ chosen in iteration t. As a thought experiment, consider computing $\mathcal{P}(v; x, y'')$, i.e., taking a walk through the *unrestricted* program. Let $v = u_0, u_1, u_2, \dots, u_{T'}$ be the vertices visited in this walk, $T' \leq T$. Let S_t be the set of blocks $b' \in [B]$ that are queried by the first $B/2$ steps of this walk. That is,

$$S_t = \{b' \in [B] : \exists h < \lfloor B/2 \rfloor \text{ such that } i(u_h) \in I_{b'}\}, \tag{27}$$

so that $|S_t| \leq \lfloor B/2 \rfloor$. Let $S'_t = S_t \cup [B']$, where B' is chosen so that $|S'_t| = \lfloor B/2 \rfloor$. Let E_t be the event that $b \in S'_t$, where b is the value chosen by H_2 in iteration t of the loop.

Since b and y'' are chosen *independently* at random, the events E_t are independent, and $\Pr[E_t] = \frac{\lfloor B/2 \rfloor}{B} \leq 1/2$. Therefore, by Hoeffding's inequality,

$$\Pr[\#E_t \text{ that occur} > (3/4)r] \leq e^{-r/8}. \tag{28}$$

Now, suppose that E_t does not occur. Then $b \notin S'_t$, so $b \notin S_t$. This implies that when updating $v = \mathcal{P}|_{[n]\setminus I}(v; x, y'')$ (taking a walk through the *restricted* program), we either reach a terminal vertex of $\mathcal{P}$ or we take at least $\lfloor B/2 \rfloor$ steps before reaching a vertex that tries to query I. Therefore, $T_t \geq \min\{\lfloor B/2 \rfloor, T\}$. By Equation (11),

$$\frac{r}{4} \cdot \left\lfloor \frac{B}{2} \right\rfloor \geq r \cdot \left(\frac{B}{8} - 1 \right) \geq T. \tag{29}$$

Equation (26) follows. Since $T \geq \text{length}(\mathcal{P})$, $\sum_{t=1}^{r} T_t \geq T$ implies that H_2 outputs a terminal vertex of $\mathcal{P}$. Therefore, any random string that gives $\sum_{t=1}^{r} T_t \geq T$ also causes H_2 and H_3 to output the same vertex. ◁

Finally, we argue that H_3 perfectly simulates $\mathcal{P}$ (with zero error).

▷ Claim 14 ($\mathsf{H}_3 \sim \mathcal{P}$). For every x,

$$\mathsf{H}_3(\mathcal{P}, v_0, x, T) \sim \mathcal{P}(v_0; x, U_T). \tag{30}$$

Proof. For any path $v_0, v_1, \dots, v_{T'}$ through $\mathcal{P}$ ending at a terminal vertex, both computations, $\mathsf{H}_3(\mathcal{P}, v_0, x, T)$ and $\mathcal{P}(v_0; x, U_T)$, have exactly a $2^{-T'}$ chance of following that path. ◁

Proof of Theorem 8. By Claims 11 to 14 and the triangle inequality,

$$\text{density}\{x \in \{0,1\}^n : \mathsf{A}(\mathcal{P}, v_0, x, T) \not\sim_\delta \mathcal{P}(v_0; x, U_T)\} \leq 2^{-S^c}, \tag{31}$$

where $\delta = \varepsilon r + \varepsilon' r \cdot 2^{S-1} + e^{-r/8}$. By our choice of ε (Equation (12)), the first term is at most $e^{-cS}/4$. By our choice of ε' (Equation (13)), the second term is also at most $e^{-cS}/4$. By our choice of r (Equation (11)), the third term is at most $e^{-cS}/2$. Therefore, $\delta \leq e^{-cS}$. ◀

3.2 Main Result: Derandomizing Uniform Random-Access Algorithms

Theorem 8 immediately implies $\mathbf{BPTISP}(n \cdot \text{poly}(S), S)$ can be simulated by a typically-correct algorithm that runs in time $n \cdot \text{poly}(S)$ and space $O(S)$ that uses only $\text{poly}(S)$ random bits.

▶ **Corollary 15.** *Fix a function $S(n) \geq \log n$ that is constructible in time $n \cdot \text{poly}(S)$ and space $O(S)$, and fix a constant $c \in \mathbb{N}$. For every language $L \in \mathbf{BPTISP}(n \cdot \text{poly}(S), S)$, there is a randomized algorithm $\mathcal{A}$ running in time $n \cdot \text{poly}(S)$ and space $O(S)$ that uses $\text{poly}(S)$ random bits such that*

$$\text{density}\{x \in \{0,1\}^n : \Pr[\mathcal{A}(x) \neq L(x)] > 2^{-S^c}\} \leq 2^{-S^c}. \tag{32}$$

Proof. Let $\mathcal{B}$ be the algorithm witnessing $L \in \mathbf{BPTISP}(n \cdot \mathrm{poly}(S), S)$. Let c' be a constant so that $\mathcal{B}$ runs in time $n \cdot S^{c'}$. For $n \in \mathbb{N}$, let $\mathcal{P}_n$ be a randomized branching program, where each vertex in $V(\mathcal{P}_n)$ describes a configuration of $\mathcal{B}$ with at most $S(n)$ symbols written on each tape. For each vertex $v \in V(\mathcal{P}_n)$, let $i(v)$ be the location of the input tape read head in the configuration described by v, and let $j(v)$ be the location of the random tape read head in the configuration described by v. The transitions of $\mathcal{P}_n$ correspond to the transitions of $\mathcal{B}$ in the obvious way.

By construction, $\mathcal{P}_n$ is an R-OW branching program with size $2^{O(S)}$ and length at most $n \cdot S^{c'}$. Furthermore, given a vertex v, the neighborhood of v can be computed in $\mathrm{poly}(S)$ time and $O(S)$ space, simply by consulting the transition function for $\mathcal{B}$.

Given $x \in \{0,1\}^n$, the algorithm $\mathcal{A}_0$ runs the algorithm of Theorem 8 on input $(\mathcal{P}_n, v_0, x, n \cdot S^{c'})$, where v_0 encodes the starting configuration of $\mathcal{B}$. This gives a vertex $v \in V(\mathcal{P}_n)$. The algorithm $\mathcal{A}_0$ accepts if and only if v encodes an accepting configuration of $\mathcal{B}$. That way,

$$\mathrm{density}\{x \in \{0,1\}^n : \Pr[\mathcal{A}_0(x) \neq L(x)] > 1/3 + e^{-cS}\} \leq 2^{-S^c}. \tag{33}$$

The algorithm $\mathcal{A}(x)$ runs $O(S^c)$ repetitions of $\mathcal{A}_0(x)$ and takes a majority vote, driving the failure probability down to 2^{-S^c}.

Clearly, $\mathcal{A}$ runs in time $n \cdot S^{c'} \cdot \mathrm{poly}(S) \cdot S^c = n \cdot \mathrm{poly}(S)$ and space $O(S)$. The number of random bits used by $\mathcal{A}$ is $O(\frac{n \cdot S^{c'}}{n} \cdot \mathrm{poly}(S) \cdot S^c) = \mathrm{poly}(S)$. ◀

We can further reduce the randomness complexity by using a pseudorandom generator by Nisan and Zuckerman [30].

▶ **Theorem 16** ([30]). *Fix constants $c \in \mathbb{N}, \alpha > 0$. For every $S \in \mathbb{N}$, there is a generator $\mathsf{NZGen} : \{0,1\}^s \to \{0,1\}^{S^c}$ with seed length $s \leq O(S)$ such that if $\mathcal{P}$ is an R-OW randomized branching program of size 2^S, v is a vertex, and x is an input, then*

$$\mathcal{P}(v; x, \mathsf{NZGen}(U_s)) \sim_\varepsilon \mathcal{P}(v; x, U_{S^c}), \tag{34}$$

where $\varepsilon = 2^{-S^{1-\alpha}}$. Given S and z, $\mathsf{NZGen}(z)$ can be computed in $O(S)$ space and $\mathrm{poly}(S)$ time.

▶ **Corollary 17** (Main result). *Fix a function $S(n) \geq \log n$ that is constructible in time $n \cdot \mathrm{poly}(S)$ and space $O(S)$, and fix constants $c \in \mathbb{N}, \alpha > 0$. For every language $L \in \mathbf{BPTISP}(n \cdot \mathrm{poly}(S), S)$, there is a randomized algorithm $\mathcal{A}$ running in time $n \cdot \mathrm{poly}(S)$ and space $O(S)$ that uses $O(S)$ random bits such that*

$$\mathrm{density}\{x \in \{0,1\}^n : \Pr[\mathcal{A}(x) \neq L(x)] > 2^{-S^{1-\alpha}}\} \leq 2^{-S^c}. \tag{35}$$

Proof sketch. Compose the algorithm of Corollary 15 with the Nisan-Zuckerman generator (Theorem 16). The algorithm of Corollary 15 can be implemented as a randomized branching program as in the proof of Corollary 15. ◀

Finally, we can eliminate the random bits entirely at the expense of time.

▶ **Corollary 18.** *For every space-constructible function $S(n) \geq \log n$, for every constant $c \in \mathbb{N}$,*

$$\mathbf{BPTISP}(n \cdot \mathrm{poly}(S), S) \text{ is within } 2^{-S^c} \text{ of } \mathbf{DSPACE}(S). \tag{36}$$

Proof. Run the algorithm of Corollary 17 on all possible random strings and take a majority vote. ◀

4 Derandomizing Turing Machines

In this section, we give our improved typically-correct derandomizations for Turing machines. Sections 4.1 and 4.2 concern derandomization with a low mistake rate, and Sections 4.3 to 4.5 concern derandomization of Turing machines with runtime $T \approx n^2$.

4.1 Low-Randomness Simulation of Sequential-Access Branching Programs with a Low Mistake Rate

Recall that for a nonterminal vertex v in a branching program, $i(v)$ is the index of the input queried by v, and $j(v)$ is the index of the random string queried by v.

▶ **Definition 19.** *An* S-OW randomized branching program *is a randomized branching program $\mathcal{P}$ such that for every edge (v, v') between two nonterminal vertices, $|i(v) - i(v')| \leq 1$ and $j(v') \in \{j(v), j(v) + 1\}$.*

In words, an S-OW randomized branching program has *sequential* access to its input and one-way access to its random bits. By "sequential access", we mean that after reading bit i, it reads bit $i - 1$, bit i, or bit $i + 1$, like a head of a Turing machine. For S-OW branching programs, we give an algorithm analogous to Theorem 8 but with a much lower rate of mistakes.

▶ **Theorem 20.** *For each constant $c \in \mathbb{N}$, there is a randomized random-access algorithm A with the following properties. Suppose $\mathcal{P}$ is an S-OW randomized branching program on $\{0,1\}^n \times \{0,1\}^T$ with $S \geq \log n$, where $S \overset{\text{def}}{=} \lceil \log \operatorname{size}(\mathcal{P}) \rceil$. Suppose $v_0 \in V(\mathcal{P})$, $T \geq \operatorname{length}(\mathcal{P})$, and $x \in \{0,1\}^n$. Then $\mathsf{A}(\mathcal{P}, v_0, x, T)$ outputs a vertex $v \in V(\mathcal{P})$. The number of random bits used by A is $\lceil T/n \rceil \cdot \operatorname{poly}(S)$, and A runs in time[7] $T \cdot \operatorname{poly}(n, S)$ and space $O(S)$. Finally, for every such $\mathcal{P}, v_0, T$,*

$$\#\{x \in \{0,1\}^n : \mathsf{A}(\mathcal{P}, v_0, x, T) \not\sim_{\exp(-cS)} \mathcal{P}(v_0; x, U_T)\} \leq 2^{n/S^c}. \tag{37}$$

The proof of Theorem 20 is very similar to the proof of Theorem 8. The main difference is that instead of using a *small* part of the input as the source of randomness, we use *most* of the input as a source of randomness. The only part of the input that is *not* used as a source of randomness is the region near the bit that the branching program was processing at the beginning of the current phase.

Because the proof of Theorem 20 does not introduce any significantly new techniques, we defer the proof to Appendix D.

4.2 Derandomizing Turing Machines with a Low Mistake Rate

A *randomized Turing machine* is defined like a randomized random-access Turing machine except that there are no index tapes. Thus, moving a read head from position i to position j takes $|i - j|$ steps. For functions $T, S : \mathbb{N} \to \mathbb{N}$, let $\mathbf{BPTISP}_{\mathrm{TM}}(T, S)$ denote the class of languages L such that there is a randomized Turing machine $\mathcal{A}$ that always runs in time $O(T(n))$ and space $O(S(n))$ such that for every $x \in \{0,1\}^*$,

$$\Pr[\mathcal{A}(x) = L(x)] \geq 2/3. \tag{38}$$

[7] Like in Theorem 8, the graph of $\mathcal{P}$ should be encoded in adjacency list format. We also stress that A is a random-access simulation of sequential-access branching programs.

Trivially, a randomized Turing machine can be simulated by a randomized random-access Turing machine without loss in efficiency. Conversely, a single step of a randomized $O(S)$-space random-access Turing machine can be simulated in $O(n + S)$ steps by a randomized Turing machine. This proves the following elementary containments.

▶ **Proposition 21.** *For any functions $T, S : \mathbb{N} \to \mathbb{N}$ with $S(n) \geq \log n$,*

$$\mathbf{BPTISP}_{\mathrm{TM}}(T, S) \subseteq \mathbf{BPTISP}(T, S) \subseteq \mathbf{BPTISP}_{\mathrm{TM}}(T \cdot (n + S), S). \tag{39}$$

Theorem 20 combined with the Nisan-Zuckerman generator [30] immediately implies a derandomization theorem for Turing machines analogous to Corollary 17.

▶ **Corollary 22.** *Fix a function $S : \mathbb{N} \to \mathbb{N}$ with $S(n) \geq \log n$ that is constructible in time $\mathrm{poly}(n, S)$ and space $O(S)$, and fix constants $c \in \mathbb{N}, \alpha > 0$. For every language $L \in \mathbf{BPTISP}_{\mathrm{TM}}(n \cdot \mathrm{poly}(S), S)$, there is a randomized algorithm $\mathcal{A}$ running in time $\mathrm{poly}(n, S)$ and space $O(S)$ that uses $O(S)$ random bits such that*

$$\#\{x \in \{0, 1\}^n : \Pr[\mathcal{A}(x) \neq L(x)] > 2^{-S^{1-\alpha}}\} \leq 2^{n/S^c}. \tag{40}$$

Proof sketch. A randomized Turing machine obviously gives rise to an S-OW randomized branching program. Like in the proof of Corollary 15 (but with Theorem 20 in place of Theorem 8), we first obtain an algorithm that uses $\mathrm{poly}(S)$ random bits. Composing with the Nisan-Zuckerman generator (Theorem 16) completes the proof. ◀

▶ **Corollary 23.** *For every space-constructible function $S(n) \geq \log n$, for every constant $c \in \mathbb{N}$,*

$$\mathbf{BPTISP}_{\mathrm{TM}}(n \cdot \mathrm{poly}(S), S) \text{ is within } 2^{-n+n/S^c} \text{ of } \mathbf{DSPACE}(S). \tag{41}$$

Proof. Simulate the algorithm of Corollary 22 on all possible random strings and take a majority vote. ◀

4.3 Simulating Branching Programs with Random Access to Random Bits

We now move on to our second derandomization of Turing machines, as outlined in Section 1.2.4. Recall that for a nonterminal vertex v in a branching program, $i(v)$ is the index of the input that is queried by v.

▶ **Definition 24.** *An* S-R *randomized branching program is a randomized branching program $\mathcal{P}$ such that for every edge (v, v') between two nonterminal vertices, $|i(v) - i(v')| \leq 1$.*

In words, an S-R randomized branching program has sequential access to its input and random access to its random bits. This model is *more general* than the S-OW model; the S-OW model corresponds more directly to the randomized Turing machine model. But studying the more general S-R model will help us derandomize Turing machines.

We will give a randomness-efficient algorithm for simulating S-R randomized branching programs, roughly analogous to Theorems 8 and 20. The simulation will only work well if the branching program has small length *and* uses few random bits.

Our simulation of S-R randomized branching programs is a fairly straightforward application of work by Kinne et al. [24]; this section is not technically novel. But it is useful to be able to compare the work by Kinne et al. [24] to our algorithms based on the "out of sight, out of mind" technique.

Unlike Theorems 8 and 20, our simulation of S-R branching programs will not work on a step-by-step basis, generating a distribution on vertices that approximates the behavior of the branching program. Instead, our simulation of S-R branching programs will only work for S-R branching programs that *compute a Boolean function.* We now give the relevant definition.

▶ **Definition 25.** *Let $\mathcal{P}$ be a randomized branching program on $\{0,1\}^n \times \{0,1\}^m$. Suppose some vertex $v_0 \in V(\mathcal{P})$ is labeled as the* start vertex*, and every terminal vertex of $\mathcal{P}$ is labeled with an* output bit *$b \in \{0,1\}$. In this case, we identify $\mathcal{P}$ with a function $\mathcal{P} : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ defined by*

$$\mathcal{P}(x,y) = \textit{the output bit labeling } \mathcal{P}(v_0; x, y). \tag{42}$$

We say that $\mathcal{P}$ computes $f : \{0,1\}^n \to \{0,1\}$ with failure probability δ *if for every $x \in \{0,1\}^n$,*

$$\Pr[\mathcal{P}(x, U_m) = f(x)] \geq 1 - \delta. \tag{43}$$

Instead of assuming a time bound, it will be useful to assume a bound on the *query complexity* of the branching program.

▶ **Definition 26.** *Let $\mathcal{P}$ be randomized branching program. The* query complexity *of $\mathcal{P}$, denoted* queries$(\mathcal{P})$*, is the maximum, over all paths $v_1, v_2, \dots, v_T$ through $\mathcal{P}$ consisting entirely of nonterminal vertices, of*

$$1 + \#\{t \in \{2,3,\dots,T\} : i(v_t) \neq i(v_{t-1})\}. \tag{44}$$

In words, queries$(\mathcal{P})$ is the number of steps that $\mathcal{P}$ takes in which it queries a *new* bit of its input, i.e., not the bit that it queried in the previous step. Trivially, queries$(\mathcal{P}) \leq$ length$(\mathcal{P})$. The reader is encouraged to think of the distinction between queries$(\mathcal{P})$ and length$(\mathcal{P})$ as being a technicality that can be ignored on the first reading.

We can now state our deterministic simulation theorem for S-R randomized branching programs. It consists of a method of deterministically generating coins for the branching program from its input.

▶ **Theorem 27.** *There is a constant $\alpha > 0$ so that for every n, m with $m \leq n/3$, there is a function $\mathsf{R} : \{0,1\}^n \to \{0,1\}^m$ with the following properties. Suppose $\mathcal{P}$ is an S-R randomized branching program on $\{0,1\}^n \times \{0,1\}^m$ that computes a function f with failure probability δ. Suppose $TSm \leq \alpha n^2$, where $T \overset{\text{def}}{=}$ queries$(\mathcal{P})$ and $S \overset{\text{def}}{=} \lceil \log \text{size}(\mathcal{P}) \rceil$. Then*

$$\text{density}\{x \in \{0,1\}^n : \mathcal{P}(x, \mathsf{R}(x)) \neq f(x)\} \leq 3\delta + m \cdot 2^{-\alpha n/m}. \tag{45}$$

Furthermore, given x and m, $\mathsf{R}(x)$ can be computed in space $O(\log n)$.

The function R is based on a pseudorandom generator by Kinne et al. [24] for *multiparty communication protocols.* In a *public-coin randomized* 3*-party NOF protocol* Π, there are three parties, three inputs x_1, x_2, x_3, and one random string y. Party i knows x_j for $j \neq i$, and all three parties know y. All parties have access to a blackboard. The protocol specifies who should write next as a function of what has been written on the blackboard so far and y. Eventually, the protocol specifies the output $\Pi(x_1, x_2, x_3, y)$, which should be a function of what has been written on the blackboard and y. The communication complexity of Π is the maximum number of bits written on the blackboard over all x_1, x_2, x_3, y. A *deterministic* 3*-party NOF protocol* is just the case $|y| = 0$.

Following Kinne et al. [24], we rely on a 3-party communication complexity lower bound by Babai et al. [7]. For an integer $\ell \in \mathbb{N}$, define $\mathrm{GIP}_\ell : (\{0,1\}^\ell)^3 \to \{0,1\}$ to be the generalized inner product function, i.e.,

$$\mathrm{GIP}_\ell(x, y, z) = \sum_{i=1}^{\ell} x_i y_i z_i \bmod 2. \tag{46}$$

Babai et al. showed that the trivial communication protocol for GIP_ℓ is essentially optimal, even in the average-case setting.

▶ **Theorem 28** ([7]). *There is a constant $\beta > 0$ so that for every $\ell \in \mathbb{N}, \varepsilon > 0$, if Π is a deterministic 3-party NOF protocol with*

$$\Pr_{x,y,z}[\Pi(x, y, z) = \mathrm{GIP}_\ell(x, y, z)] \geq \frac{1}{2} + \varepsilon, \tag{47}$$

then the communication complexity of Π is at least $\beta \cdot (\ell - \log(1/\varepsilon))$.

To define R, let $x \in \{0,1\}^n$. Partition $n = n_1 + n_2 + n_3$, where $n_i \geq \lfloor n/3 \rfloor$ for each i. Correspondingly partition $x = x_1 \circ x_2 \circ x_3$, where $|x_i| = n_i$. Define

$$\ell = \left\lfloor \frac{\lfloor n/3 \rfloor}{m} \right\rfloor, \tag{48}$$

so that $\ell \geq 1$. For $i \in [3]$ and $j \in [m]$, let x_{ij} be the jth ℓ-bit substring of x_i. (Note that due to roundoff errors, for some values of n, some bits of x are not represented in any x_{ij}.) Then we define

$$\mathsf{R}(x) = \mathrm{GIP}_\ell(x_{11}, x_{21}, x_{31}) \circ \cdots \circ \mathrm{GIP}_\ell(x_{1m}, x_{2m}, x_{3m}) \in \{0,1\}^m. \tag{49}$$

Kinne et al. observed that $x \mapsto (x, \mathsf{R}(x))$ is a pseudorandom generator that fools 3-party NOF protocols [24]. For clarity, we reproduce the argument here.

▶ **Lemma 29.** *Suppose $\Pi : \{0,1\}^{n_1} \times \{0,1\}^{n_2} \times \{0,1\}^{n_3} \times \{0,1\}^m \to \{0,1\}$ is a public-coin randomized 3-party NOF protocol. Suppose that for some $\varepsilon > 0$, Π uses less than $\beta \cdot (\ell - \log(1/\varepsilon))$ bits of communication, where β is the constant of Theorem 28. Then*

$$\left| \Pr_{x_1,x_2,x_3,y}[\Pi(x_1, x_2, x_3, y) = 1] - \Pr_{x_1,x_2,x_3}[\Pi(x_1, x_2, x_3, \mathsf{R}(x_1, x_2, x_3)) = 1] \right| < \varepsilon m. \tag{50}$$

Proof. Let

$$\delta = \left| \Pr_{x_1,x_2,x_3,y}[\Pi(x_1, x_2, x_3, y) = 1] - \Pr_{x_1,x_2,x_3}[\Pi(x_1, x_2, x_3, \mathsf{R}(x_1, x_2, x_3)) = 1] \right|. \tag{51}$$

By Yao's distinguisher-to-predictor argument [42], there is some index $i \in [m]$ and a protocol $\Pi' : \{0,1\}^{n_1} \times \{0,1\}^{n_2} \times \{0,1\}^{n_3} \times \{0,1\}^{i-1} \to \{0,1\}$ so that

$$\Pr_{x_1,x_2,x_3}[\Pi'(x_1, x_2, x_3, \mathsf{R}(x_1, x_2, x_3)|_{[i-1]}) = \mathsf{R}(x_1, x_2, x_3)_i] \geq \frac{1}{2} + \frac{\delta}{m}. \tag{52}$$

The protocol Π' is a public-coin randomized 3-party NOF protocol that still uses less than $\beta \cdot (\ell - \log(1/\varepsilon))$ bits of communication, since it merely involves simulating Π with certain input/coin bits fixed to certain values and possibly negating the output. This immediately implies a protocol for GIP_ℓ with the same parameters with advantage δ/m. There is some way to fix the randomness to preserve advantage, so by Theorem 28, $\delta/m < \varepsilon$. ◀

The connection between S-R randomized branching programs and 3-party communication protocols is given by the following lemma.

▶ **Lemma 30.** *There is a public-coin randomized 3-party NOF protocol* $\Pi : \{0,1\}^{n_1} \times \{0,1\}^{n_2} \times \{0,1\}^{n_3} \times \{0,1\}^m \to \{0,1\}$ *such that*

$$\Pi(x_1, x_2, x_3, y) = \mathcal{P}(x_1 \circ x_2 \circ x_3, y), \tag{53}$$

and Π *uses only* $O(\frac{TS}{n})$ *bits of communication.*

Proof. Parties 1 and 3 alternate simulating the operation of $\mathcal{P}$. If party 1 is simulating and the program reads from the first n_1 bits of the input, party 1 sends the state to party 3. Similarly, if party 3 is simulating and the program reads from the last n_3 bits of the input, party 3 sends the state to party 1. Each such transition indicates that the program must have spent at least n_2 steps traversing the middle n_2 bits of the input. Therefore, the total number of such transitions is at most $\frac{T}{n_2}$. ◀

Given Lemmas 29 and 30, Theorem 27 follows by a lemma by Kinne et al. [24, Lemma 1]. For clarity, we reproduce the argument here.

Proof of Theorem 27. The best case is at least as good as the average case, so there is some string $y_* \in \{0,1\}^m$ such that

$$\Pr_{x \in \{0,1\}^n}[\mathcal{P}(x, y_*) \neq f(x)] \leq \delta. \tag{54}$$

Define $g : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ by

$$g(x, y) = \begin{cases} 1 & \text{if } \mathcal{P}(x, y) = \mathcal{P}(x, y_*) \\ 0 & \text{otherwise.} \end{cases} \tag{55}$$

Think of $x \in \{0,1\}^n$ as $x = x_1 \circ x_2 \circ x_3$, like in the definition of R. Then by Lemma 30, g can be computed by a 3-party NOF protocol using $O(\frac{TS}{n})$ bits of communication. By choosing α small enough and setting $\varepsilon = 2^{-\alpha n/m}$, this protocol for f will use fewer than $\beta(\ell - \log(1/\varepsilon))$ bits of communication. Therefore, by Lemma 29,

$$\Pr_x[\mathcal{P}(x, \mathsf{R}(x)) \neq \mathcal{P}(x, y_*)] \leq \Pr_{x,y}[\mathcal{P}(x, y) \neq \mathcal{P}(x, y_*)] + \varepsilon m. \tag{56}$$

Therefore,

$$\begin{aligned} \Pr_x[\mathcal{P}(x, \mathsf{R}(x)) \neq f(x)] &\leq \Pr_x[\mathcal{P}(x, y_*) \neq f(x)] + \Pr_x[\mathcal{P}(x, \mathsf{R}(x)) \neq \mathcal{P}(x, y_*)] && (57)\\ &\leq \delta + \Pr_{x,y}[\mathcal{P}(x, y) \neq \mathcal{P}(x, y_*)] + \varepsilon m && (58)\\ &\leq \delta + \Pr_{x,y}[\mathcal{P}(x, y) \neq f(x)] + \Pr_x[\mathcal{P}(x, y_*) \neq f(x)] + \varepsilon m && (59)\\ &\leq \delta + \delta + \delta + \varepsilon m. && (60) \end{aligned}$$

Obviously, $\mathsf{R}(x)$ can be computed in $O(\log n)$ space. ◀

4.4 Randomness-Efficient Amplification for Branching Programs

We will use a space-efficient expander walk algorithm by Gutfreund and Viola [17].

▶ **Theorem 31** ([17]). *For every $s \in \mathbb{N}$, there is a constant-degree expander graph G on vertex set $\{0,1\}^s$. Furthermore, there is an algorithm* GVWalk *such that if $y \in \{0,1\}^s$ is a vertex and $e_1, e_2, \ldots, e_r \in \{0,1\}^{O(1)}$ are edge labels, then* $\mathsf{GVWalk}(y, e_1, e_2, \ldots, e_r)$ *outputs the vertex reached by starting at y and taking a walk by following the edge labels $e_1, e_2, \ldots, e_r$. The algorithm* GVWalk *runs in space $O(\log s + \log r)$.*

Recall that we are working toward derandomizing the class $\mathbf{BPTISP}_{\mathrm{TM}}(T, S)$ for all $TS^2 \le o(n^2/\log n)$. This class corresponds to branching programs on $\{0,1\}^n \times \{0,1\}^T$ that compute some function with failure probability 1/3. But Theorem 27 requires that the branching program use at most $\frac{\alpha n^2}{TS}$ random bits. Furthermore, the failure probability of the branching program governs the mistake rate of the derandomization.

We can overcome these two difficulties because randomized Turing machines correspond to *S-OW* randomized branching programs (i.e., programs that have sequential access to the input and one-way access to the random bits), whereas Theorem 27 applies to the more powerful S-R model (i.e., programs that have sequential access to the input and *random* access to the random bits). An S-OW branching program can be simulated by an S-R branching program using very few random bits by applying Nisan's generator. The following lemma combines this idea with a random walk on an expander graph (Theorem 31) for amplification. This is the same technique that Fortnow and Klivans used to prove that $\mathbf{BPL} \subseteq \mathbf{L}/O(n)$ [12].

▶ **Lemma 32.** *Suppose $\mathcal{P}$ is an S-OW randomized branching program on $\{0,1\}^n \times \{0,1\}^T$ that computes a function $f : \{0,1\}^n \to \{0,1\}$ with failure probability 1/3. Let $S = \log \operatorname{size}(\mathcal{P})$. For every $\delta > 0$, there is an S-R branching program $\mathcal{P}'$ on $\{0,1\}^n \times \{0,1\}^m$ that computes f with failure probability δ such that*

$$\operatorname{queries}(\mathcal{P}') \le O((\operatorname{queries}(\mathcal{P}) + n)\log(1/\delta)), \tag{61}$$

$$\log \operatorname{size}(\mathcal{P}') \le O(S + \log\log(1/\delta)), \tag{62}$$

$$m \le O(S \log T + \log(1/\delta)). \tag{63}$$

Furthermore, given $\mathcal{P}$, δ, and a vertex $v \in V(\mathcal{P}')$, the neighborhood of v can be computed in time[8] $\operatorname{poly}(S, \log(1/\delta))$ *and space $O(S + \log\log(1/\delta))$.*

Proof. Let $\mathsf{NisGen} : \{0,1\}^s \to \{0,1\}^T$ be Nisan's generator with error 0.1 for randomized branching programs of size size($\mathcal{P}$). Let G be the expander of Theorem 31 on vertex set $\{0,1\}^s$. We will interpret a string $y \in \{0,1\}^m$ as describing a walk through G from an arbitrary initial vertex of length $r-1$, so that $m = s + O(r)$. Let $y_1, \ldots, y_r \in \{0,1\}^s$ be the vertices visited by this walk. The program $\mathcal{P}'(x, y)$ runs $\mathcal{P}(x, \mathsf{NisGen}(y_t))$ for every $y \in [r]$ and takes a majority vote of the answers; it finds the vertices y_t by running the algorithm GVWalk of Theorem 31. By the expander walk Chernoff bound [14], for an appropriate choice of $r = \Theta(\log(1/\delta))$, the failure probability of $\mathcal{P}'$ is at most δ.

Clearly, $\operatorname{queries}(\mathcal{P}') \le r \cdot (\operatorname{queries}(\mathcal{P}) + n)$, where the $+n$ term takes care of the steps needed to get from the final position of x read in one iteration of $\mathcal{P}$ to the first position of x read in the next iteration of $\mathcal{P}$ (recall that $\mathcal{P}'$ is an S-R branching program).

[8] As usual, we assume that the graph of $\mathcal{P}$ is encoded in adjacency list format. We also assume that the start vertex v_0 is designated in a way that allows it to be computed in the specified time and space.

The space needed by $\mathcal{P}'$ consists of the S bits of space needed for $\mathcal{P}$, plus $O(S)$ bits of space for computing NisGen, plus $O(\log r)$ bits of space to keep track of the answers generated by the iterations, plus $O(\log S + \log r)$ bits of space for GVWalk. Finally, computing the neighborhood of v merely requires inspecting the transition functions for the algorithms NisGen and GVWalk, inspecting $\mathcal{P}$, and doing arithmetic. ◀

4.5 Derandomizing Turing Machines with Runtime Near n^2

Finally, we are ready to state and prove our typically-correct derandomization of Turing machines based on Theorem 27.

▶ **Corollary 33.** *Suppose $T, S : \mathbb{N} \to \mathbb{N}$ are both constructible in time $\mathrm{poly}(n)$ and space $O(S)$ and $TS^2 \leq o\left(\frac{n^2}{\log n}\right)$. For every language $L \in \mathbf{BPTISP}_{\mathrm{TM}}(T, S)$, there is a constant $\gamma > 0$ so that*

$$L \text{ is within } \exp\left(-\frac{\gamma n}{\sqrt{TS}}\right) + \exp\left(-\frac{\gamma n^2}{TS^2 \log n}\right) \text{ of } \mathbf{DTISP}(\mathrm{poly}(n), S). \tag{64}$$

The rate of mistakes in Corollary 33 is always $o(1)$. The rate of mistakes gets smaller (i.e., the simulation quality gets higher) when T and S are smaller. For example, if $S = \log n$ and $T = n^2/\log^4 n$, the rate of mistakes in Equation (64) is $n^{-\Omega(1)}$. For another example, if $S = \mathrm{polylog}\, n$ and $T = n\, \mathrm{polylog}\, n$, the rate of mistakes in Equation (64) is $\exp\left(-\widetilde{\Omega}(\sqrt{n})\right)$.

As a reminder, Corollary 33 is incomparable to Corollary 18: the randomized classes in the two results are incomparable; the deterministic algorithm in Corollary 33 is faster; the mistake rate in Corollary 33 is lower when S and T are not too big. Similarly, Corollary 33 is incomparable to Corollary 23: the randomized class in Corollary 33 is more powerful and the deterministic algorithm in Corollary 33 is faster, but the mistake rate in Corollary 33 is much higher. Finally, even when $S \geq n^{\Omega(1)}$, Corollary 33 is incomparable to derandomizing via the Nisan-Zuckerman generator [30], because the deterministic algorithm of Corollary 33 runs in polynomial time, although it makes some mistakes.

Conceptually, the proof of Corollary 33 merely consists of combining Lemma 32 and Theorem 27. The only work to be done is in appropriately choosing δ and verifying parameters.

Proof of Corollary 33. Let $\mathcal{A}$ be the algorithm witnessing $L \in \mathbf{BPTISP}_{\mathrm{TM}}(T, S)$. Let $\mathcal{P}_n$ be the S-OW branching program on $\{0,1\}^n \times \{0,1\}^T$ describing the behavior of $\mathcal{A}$ on inputs of length n.

We consider two cases. First, suppose $TS^3 > n^2/\log^2 n$. Then let

$$\delta = \exp\left(-\frac{\gamma_0 n^2}{TS^2 \log n}\right), \tag{65}$$

where the constant γ_0 will be specified later. Let $\mathcal{P}'_n$ be the S-R branching program on $\{0,1\}^n \times \{0,1\}^m$ given by Lemma 32. There is a constant c that does not depend on γ so that

$$\begin{aligned} \mathrm{queries}(\mathcal{P}'_n) \cdot \log \mathrm{size}(\mathcal{P}'_n) \cdot m &\leq cTS^2 \log n \ln(1/\delta) + cTS \ln^2(1/\delta) && (66)\\ &= c\gamma_0 n^2 + \frac{c\gamma_0^2 n^4}{TS^3 \log^2 n} && (67)\\ &\leq c\gamma_0 n^2 + c\gamma_0^2 n^2. && (68)\end{aligned}$$

Choose γ_0 so that $c\gamma_0 + c\gamma_0^2 \leq \alpha$, where α is the value in Theorem 27. Since $TS^2 \leq o(n^2/\log n)$ and $T \geq n$, we must have $S \leq o(\sqrt{n/\log n})$. Therefore,

$$m \leq O\left(S\log n + \frac{n^2}{TS^2\log n}\right) \leq O\left(S\log n + \frac{TS^3\log n}{TS^2}\right) \leq o(\sqrt{n\log n}) \leq n/3. \tag{69}$$

Therefore, the hypotheses of Theorem 27 are satisfied.

The deterministic algorithm, naturally, outputs $\mathcal{P}'_n(x, \mathsf{R}(x))$, where R is the function of Theorem 27. It is immediate that this runs in $\mathrm{poly}(n)$ time and $O(S)$ space. Finally, to compute the rate of mistakes, observe that

$$m \cdot 2^{-\alpha n/m} \leq \exp\left(-\Omega\left(-\frac{n}{S\log n}\right)\right), \tag{70}$$

whereas

$$\delta \geq \exp\left(-O\left(\frac{n}{S^2\log n}\right)\right). \tag{71}$$

Therefore, when n is sufficiently large, $m \cdot 2^{-\alpha n/m} < \delta$. Therefore,

$$\mathrm{density}\{x \in \{0,1\}^n : \mathcal{P}'_n(x, \mathsf{R}(x)) \neq L(x)\} \leq 4\delta. \tag{72}$$

For the second case, suppose $TS^3 \leq n^2/\log^2 n$. Then let

$$\delta = \exp\left(-\frac{\gamma_0 n}{\sqrt{TS}}\right). \tag{73}$$

Again, let $\mathcal{P}'_n$ be the S-R branching program on $\{0,1\}^n \times \{0,1\}^m$ given by Lemma 32. Then

$$\begin{aligned} \mathrm{queries}(\mathcal{P}'_n) \cdot \log \mathrm{size}(\mathcal{P}'_n) \cdot m &\leq cTS^2 \log n \ln(1/\delta) + cTS\ln^2(1/\delta) && (74)\\ &= c\gamma_1\sqrt{TS^3} n\log n + c\gamma_1^2 n^2 && (75)\\ &\leq c\gamma_1 n^2 + c\gamma_1^2 n^2 && (76)\\ &\leq \alpha n^2. && (77) \end{aligned}$$

Furthermore, since $TS^3 \leq n^2/\log^2 n$, taking a square root gives $S\sqrt{TS} \leq n/\log n$, and hence

$$m \leq O\left(S\log n + \frac{n}{\sqrt{TS}}\right) \leq O\left(\frac{n}{\sqrt{TS}}\right) < n/3. \tag{78}$$

Therefore, again, the hypotheses of Theorem 27. In this case as well, the deterministic algorithm outputs $\mathcal{P}'_n(x, \mathsf{R}(x))$. We now compute the rate of mistakes again. We have

$$m \cdot 2^{-\alpha n/m} \leq \exp(-\Omega(\sqrt{TS})) < \delta \tag{79}$$

for sufficiently large n, because $\sqrt{TS} \geq \sqrt{n\log n}$. Therefore, once again,

$$\mathrm{density}\{x \in \{0,1\}^n : \mathcal{P}'_n(x, \mathsf{R}(x)) \neq L(x)\} \leq 4\delta. \tag{80}$$

Choosing $\gamma < \gamma_0$ completes the proof. ◀

5 Derandomization with Advice

As previously mentioned, Fortnow and Klivans showed that $\mathbf{BPL} \subseteq \mathbf{L}/O(n)$ [12]. We now explain how to refine their ideas and slightly improve their result. Fortnow and Klivans' argument relied on the Gutfreund-Viola space-efficient expander walk (Theorem 31). They only used this expander for its sampling properties. Extractors also have good sampling properties. Our improvement will come from simply replacing the expander-based sampler in Fortnow and Klivans' argument with the GUV-based extractor of Theorem 6.

▶ **Theorem 34.** $\mathbf{BPL} \subseteq \mathbf{L}/(n + O(\log^2 n))$.

Proof. Let $\mathcal{A}$ be an algorithm witnessing $L \in \mathbf{BPL}$, and assume $\mathcal{A}$ has failure probability at most 0.1. Let $\mathsf{NisGen} : \{0,1\}^s \to \{0,1\}^{\mathrm{poly}(n)}$ be Nisan's generator (Theorem 9) with error 0.1 and space bound sufficient to fool $\mathcal{A}$, so that $s \leq O(\log^2 n)$. Let $\mathsf{GUVExt} : \{0,1\}^{n+2s+3} \times \{0,1\}^d \to \{0,1\}^s$ be the $(2s, 0.1)$-extractor of Theorem 6, so that $d \leq O(\log n)$.

Given input $x \in \{0,1\}^n$ and advice $a \in \{0,1\}^{n+2s+3}$, run $\mathcal{A}(x, \mathsf{NisGen}(\mathsf{GUVExt}(a, z)))$ for all z and take a majority vote.

This algorithm clearly runs in space $O(\log n)$. By Proposition 7, for each fixed x, the number of advice strings a causing the algorithm to give the wrong answer is at most 2^{2s+2}. Therefore, the total number of advice strings a that cause the algorithm to give the wrong answer for any x is at most $2^{n+2s+2} < 2^{|a|}$. Therefore, there is some choice of a such that the algorithm succeeds on all inputs. ◀

We now generalize Theorem 34, showing that the amount of advice can be reduced to below n in certain cases. We will rely on a special feature of Nisan's generator that Nisan used to prove $\mathbf{RL} \subseteq \mathbf{SC}$. The seed to Nisan's generator is naturally divided into two parts, $s = s_1 + s_2$, where $s_2 \leq O(S + \log(1/\varepsilon))$.[9] Nisan showed that there is an efficient procedure to *check* that the first part of the seed is "good" for a particular randomized log-space algorithm and a particular input to that algorithm.

▶ **Lemma 35** ([28]). *For every $S \in \mathbb{N}$, there is a function* $\mathsf{NisGen} : \{0,1\}^{s_1} \times \{0,1\}^{s_2} \to \{0,1\}^{2^S}$*, with $s_1 \leq O(S^2)$ and $s_2 \leq O(S)$, and an algorithm* Check*, so that*

- *For any R-OW randomized branching program $\mathcal{P}$ with $\log \mathrm{size}(\mathcal{P}) \leq S$ and any input $x \in \{0,1\}^n$,*

$$\Pr_{y_1 \in \{0,1\}^{s_1}}[\mathsf{Check}(\mathcal{P}, x, y_1) = 1] \geq 1/2. \tag{81}$$

- *If $\mathsf{Check}(\mathcal{P}, x, y_1) = 1$, then for any vertex $v_0 \in V(\mathcal{P})$,*

$$\mathcal{P}(v_0; x, \mathsf{NisGen}(y_1, U_{s_2})) \sim_{0.1} \mathcal{P}(v_0; x, U_{2^S}). \tag{82}$$

Furthermore, Check *runs in space $O(S)$, and given S, y_1, and y_2,* $\mathsf{NisGen}(y_1, y_2)$ *can be computed in space $O(S)$.*

A $\mathbf{ZP} \cdot \mathbf{SPACE}(S)$ *algorithm* for a language L with failure probability δ is a randomized Turing machine $\mathcal{A}$ with *two-way* access to its random bits such that $\mathcal{A}$ runs in space $O(S)$, $\Pr[\mathcal{A}(x) \in \{L(x), \perp\}] = 1$, and $\Pr[\mathcal{A}(x) = \perp] \leq \delta$. The following lemma refines a theorem by Nisan that says that $\mathbf{BPL} \subseteq \mathbf{ZP} \cdot \mathbf{L}$ [27]; the improvement is that our algorithm has a low failure probability relative to the number of random bits it uses.

[9] The first s_1 bits specify the hash functions, and the last s_2 bits specify the input to those hash functions.

▶ **Lemma 36.** *Fix $S : \mathbb{N} \to \mathbb{N}$ with $S(n) \geq \log n$ and $\delta : \mathbb{N} \to [0,1]$, both constructible in space $O(S)$. For every $L \in \mathbf{BPSPACE}(S)$, there is a $\mathbf{ZP} \cdot \mathbf{SPACE}(S)$ algorithm $\mathcal{A}$ that decides L with failure probability δ and uses $\log_2(1/\delta) + O(S^2)$ random bits.*

Proof. Let $\mathcal{B}$ be the algorithm witnessing $L \in \mathbf{BPSPACE}(S)$, and assume $\mathcal{B}$ has failure probability at most 0.1. Let $\mathcal{P}$ be the corresponding R-OW branching program for inputs of length n. Let $\mathsf{NisGen} : \{0,1\}^{s_1} \times \{0,1\}^{s_2} \to \{0,1\}^{\mathrm{poly}(n)}$ be the generator of Lemma 35 with space bound $\lceil \log \mathrm{size}(\mathcal{P}) \rceil$, so that $s_1 \leq O(S^2)$.

Let $\ell = \lceil \log_2(1/\delta) \rceil + 2s_1 + 2$, and let $\mathsf{GUVExt} : \{0,1\}^\ell \times \{0,1\}^d \to \{0,1\}^{s_1}$ be the $(2s_1, 0.1)$-extractor of Theorem 6, so that $d \leq O(\log\log(1/\delta) + \log S)$. On input $x \in \{0,1\}^n$ and random string $y \in \{0,1\}^\ell$, run Algorithm 5.

Algorithm 5: The algorithm used to prove Lemma 36.

```
for $z \in \{0,1\}^d$ do
    Let $y_1 \leftarrow \mathsf{GUVExt}(y, z)$
    if $\mathsf{Check}(\mathcal{P}, x, y_1)$ accepts then  /* Check is the algorithm of Lemma 35 */
        Run $\mathcal{B}(x, \mathsf{NisGen}(y_1, y_2))$ for every $y_2$, take a majority vote, and output the
          answer
    end
end
Output $\perp$
```

Clearly, Algorithm 5 runs in space $O(S + d)$. Since δ is constructible in space $O(S)$, its denominator must have at most $2^{O(S)}$ digits. Therefore, $\delta \geq 2^{-2^{O(S)}}$ and $d \leq O(S)$, so the algorithm runs in space $O(S)$. Furthermore, the algorithm is clearly zero-error. Finally, by Proposition 7, the number of y such that $\mathsf{Check}(\mathcal{P}, x, y_1)$ rejects for every z is at most 2^{2s_1+2}, and hence the failure probability of the algorithm is at most $\frac{2^{2s_1+2}}{2^\ell} \leq \delta$. ◀

We now give our generalization of Theorem 34. From the work of Goldreich and Wigderson [15], it follows that if a language $L \in \mathbf{BPSPACE}(S)$ is in $\mathbf{DPSPACE}(S)/a$ for $a \ll n$ via an algorithm where most advice strings are "good", then L is close to being in $\mathbf{DPSPACE}(S)$. Our theorem is a *converse*[10] to this result, showing that in the space-bounded setting, there is a very tight connection between typically-correct derandomizations and simulations with small amounts of advice.

▶ **Theorem 37.** *Fix functions $S : \mathbb{N} \to \mathbb{N}$ with $S(n) \geq \log n$ and $\varepsilon : \mathbb{N} \to [0,1]$ that are constructible in space $O(S)$. Suppose a language $L \in \mathbf{BPSPACE}(S)$ is within ε of $\mathbf{DSPACE}(S)$. Then*

$$L \in \mathbf{DSPACE}(S)/(n - \log_2(1/\varepsilon(n)) + O(S^2)). \tag{83}$$

Proof. Let $\mathcal{A}$ be the algorithm of Lemma 36 with $\delta < 2^{-n}/\varepsilon$. Let $m = m(n)$ be the number of random bits used by $\mathcal{A}$. Let $\mathcal{B}$ be the algorithm witnessing the fact that L is within ε of $\mathbf{DSPACE}(S)$.

The algorithm with advice is very simple. Given input $x \in \{0,1\}^n$ and advice $a \in \{0,1\}^m$, output $\mathcal{A}(x, a)$, unless $\mathcal{A}(x, a) = \perp$, in which case output $\mathcal{B}(x)$. This algorithm clearly runs in $O(S)$ space and uses $n - \log_2(1/\varepsilon(n)) + O(S^2)$ bits of advice.

[10] The statement of Theorem 37 doesn't mention it, but indeed, in the proof of Theorem 37, most advice strings are "good".

Now we argue that there is some advice string such that the algorithm succeeds on all inputs. Let $S \subseteq \{0,1\}^n$ be the set of inputs on which $\mathcal{B}$ fails. Consider picking an advice string a uniformly at random. For each string $x \in S$, $\Pr_a[\mathcal{A}(x,a) = \perp] \leq \delta$. Therefore, by the union bound, the probability that there is some $x \in S$ such that $\mathcal{A}(x,a) = \perp$ is at most $|S|\delta = \varepsilon \cdot 2^n \cdot \delta < 1$. Therefore, there is *some* advice string such that the algorithm succeeds on all inputs in S. Finally, for *any* advice string, the algorithm succeeds on all inputs in $\{0,1\}^n \setminus S$, because $\mathcal{A}$ is zero-error. ◀

Combining Theorem 37 with our typically-correct derandomizations gives unconditional simulations with fewer than n bits of advice:

▶ **Corollary 38.** *For every constant $c \in \mathbb{N}$,*

$$\mathbf{BPTISP}(n \operatorname{polylog} n, \log n) \subseteq \mathbf{L}/(n - \log^c n). \tag{84}$$

Proof. Combine Corollary 18 and Theorem 37. ◀

▶ **Corollary 39.** *For every constant $c \in \mathbb{N}$,*

$$\mathbf{BPTISP}_{\mathrm{TM}}(n \operatorname{polylog} n, \log n) \subseteq \mathbf{L}/\left(\frac{n}{\log^c n}\right). \tag{85}$$

Proof. Combine Corollary 23 and Theorem 37. ◀

▶ **Corollary 40.**

$$\mathbf{BPTISP}_{\mathrm{TM}}(n^{1.99}, \log n) \subseteq \mathbf{L}/(n - n^{\Omega(1)}). \tag{86}$$

Proof. Combine Corollary 33 and Theorem 37. ◀

6 Disambiguating Efficient Nondeterministic Algorithms

6.1 Overview

Recall that a nondeterministic algorithm is *unambiguous* if on every input, there is at most one accepting computation. Suppose a language L can be decided by a nondeterministic algorithm that runs in time $T = T(n) \geq n$ and space $S = S(n) \geq \log n$. Allender, Reinhardt, and Zhou showed that if SAT has exponential circuit complexity, there is an unambiguous algorithm for L that runs in space $O(S)$ [2]. Unconditionally, van Melkebeek and Prakriya recently gave an unambiguous algorithm for L that runs in time $2^{O(S)}$ and space $O(S\sqrt{\log T})$ [40].

For some of our results on derandomizing efficient algorithms, we give a corresponding theorem for disambiguating efficient nondeterministic algorithms, albeit with slightly worse parameters.

6.1.1 Our Results

Let $\mathbf{NTISP}(T, S)$ denote the class of languages that can be decided by a nondeterministic random-access Turing machines that runs in time T and space S. Define $\mathbf{UTISP}(T, S)$ the same way, but with the additional requirement that the algorithm is unambiguous. In Sections 6.4 and 6.5, we show that for every S and every constant $c \in \mathbb{N}$,

$$\mathbf{NTISP}(n \cdot \operatorname{poly}(S), S) \text{ is within } 2^{-S^c} \text{ of } \mathbf{UTISP}(2^{O(S)}, S\sqrt{\log S}). \tag{87}$$

Equation (87) is analogous to Corollary 18.

Reinhardt and Allender showed that $\mathbf{NL} \subseteq \mathbf{UL}/\operatorname{poly}$ [31]. In Section 6.6, we improve the Reinhardt-Allender theorem by showing that $\mathbf{NL} \subseteq \mathbf{UL}/(n + O(\log^2 n))$. More generally, we show that if a language $L \in \mathbf{NSPACE}(S)$ is within $\varepsilon(n)$ of being in $\mathbf{USPACE}(S)$, then $L \in \mathbf{USPACE}(S)/(n - \log_2(1/\varepsilon(n)) + O(S^2))$. This result is analogous to Theorem 37.

6.1.2 Techniques

Our disambiguation theorems are proven using the same "out of sight, out of mind" technique that we used in Sections 3 and 4.2 for derandomization. Roughly, this is possible because of prior work [31, 40] that reduces the problem of disambiguating algorithms to certain derandomization problems. We review the necessary background in Section 6.3.

Our disambiguation algorithms do not really introduce any additional novel techniques, beyond what we already used in Sections 3 and 4.2. Rather, our contribution in this section is to identify another setting where our techniques are helpful, thereby illustrating the generality of our techniques.

6.2 Preliminaries

Unambiguous algorithms can be *composed* as long as the inner algorithm is "single-valued", which we now define. This notion corresponds to classes such as $\mathbf{UL} \cap \mathbf{coUL}$.

▶ **Definition 41.** *$\mathcal{A}$* single-valued unambiguous algorithm *$\mathcal{A}$ is a nondeterministic algorithm such that for every input x, all but one computation path outputs a special symbol $\perp_{\mathrm{n}}$ (indicating that the nondeterministic choices were "bad"). We let $\mathcal{A}(x)$ denote the output of the one remaining computation path.*

When describing unambiguous algorithms, we will often include steps such as "Compute $a \leftarrow \mathcal{A}(x)$", where $\mathcal{A}$ is a single-valued unambiguous algorithm. Such a step should be understood as saying to run $\mathcal{A}$ on input x. If $\mathcal{A}$ outputs $\perp_{\mathrm{n}}$, immediately halt and output $\perp_{\mathrm{n}}$. Otherwise, let a be the output of $\mathcal{A}$.

6.3 Unambiguous Algorithms for Connectivity by van Melkebeek and Prakriya

Recall that the *s-t connectivity problem* is defined by

$$\mathrm{STConn} = \{(G, s, t) : \text{there is a directed path from } s \text{ to } t\}, \tag{88}$$

where G is a digraph and $s, t \in V(G)$. STConn is a classic example of an $\mathbf{NL}$-complete language [21]. Using an "inductive counting" technique, Reinhardt and Allender gave a single-valued unambiguous algorithm for testing whether a given digraph is "min-unique", as well as a single-valued unambiguous algorithm for solving STConn in min-unique digraphs [31]. Using the isolation lemma, Reinhardt and Allender showed that assigning random weights to a digraph makes it "min-unique" [31]. These two results are the main ingredients in the proof that $\mathbf{NL} \subseteq \mathbf{UL}/\operatorname{poly}$ [31].

Recently, van Melkebeek and Prakriya gave a "pseudorandom weight generator" with seed length $O(\log^2 n)$ [40].[11] Just like uniform random weights, the weights produced by this generator make a digraph "min-unique" with high probability.[12]

[11] In the terminology of van Melkebeek and Prakriya [40], here we refer to the "hashing only" approach.
[12] The van Melkebeek-Prakriya generator only works for *layered* digraphs, but this technicality does not matter for us.

Roughly, this pseudorandom weight generator by van Melkebeek and Prakriya will play a role in our disambiguation results that is analogous to the role that Nisan's generator played in our derandomization results.

For our purposes, it is not necessary to give a precise account of min-uniqueness. What matters is that STConn can be decided in unambiguous log-space given two-way access to an $O(\log^2 n)$-bit random string. Furthermore, "bad" random strings can be unambiguously detected. We now state this result more carefully.

▶ **Theorem 42** ([40]). *There is a single-valued unambiguous algorithm* vMPSeededAlg *so that for every $x \in \{0,1\}^n$,*

$$\Pr_{y \in \{0,1\}^\infty}[\mathsf{vMPSeededAlg}(x, y) \in \{\mathrm{STConn}(x), \bot_\mathrm{r}\}] = 1, \tag{89}$$

$$\Pr_{y \in \{0,1\}^\infty}[\mathsf{vMPSeededAlg}(x, y) = \bot_\mathrm{r}] \leq 1/2. \tag{90}$$

Furthermore, vMPSeededAlg(x, y) *only reads the first $O(\log^2 n)$ bits of y (the "seed") and runs in space $O(\log n)$.*

Proof sketch. We assume that the reader is familiar with the paper by van Melkebeek and Prakriya [40]. Given an instance x of STConn, the algorithm vMPSeededAlg first applies a reduction, giving a *layered* digraph G on which to test connectivity. Then, the first $O(\log^2 n)$ bits of y are interpreted as specifying $O(\log n)$ hash functions, which are used to assign weights to the vertices in G. An algorithm by Reinhardt and Allender [31] is run to determine whether the resulting weighted digraph is min-unique. If it is not, vMPSeededAlg outputs $\bot_\mathrm{r}$. If it is, another closely related algorithm by Reinhardt and Allender [31] is run to decide connectivity in the resulting weighted digraph. ◀

Notice that vMPSeededAlg can be thought of as having *three* read-only inputs: the "real" input $x \in \{0,1\}^n$; the random seed $y \in \{0,1\}^{O(\log^2 n)}$; and the nondeterministic bits $z \in \{0,1\}^{\mathrm{poly}(n)}$. The algorithm has two-way access to x and y and one-way access to z. Notice also that a computation path of vMPSeededAlg has *four* possible outputs: 0, indicating that $x \notin \mathrm{STConn}$; 1, indicating that $x \in \mathrm{STConn}$; $\bot_\mathrm{n}$, indicating bad nondeterministic bits z; and $\bot_\mathrm{r}$, indicating bad random bits y.

Iterating over all y in Theorem 42 would take $\Theta(\log^2 n)$ space. By modifying their "pseudorandom weight generator", van Melkebeek and Prakriya gave an unambiguous algorithm for STConn that runs in $O(\log^{3/2} n)$ space. The performance of their algorithm is improved if we only need to search for *short* paths; the precise details are given by the following theorem.

▶ **Theorem 43** ([40]). *There is a single-valued unambiguous algorithm* vMPShortPathsAlg *such that if G is a digraph, $s, t \in V(G)$, and $r \in \mathbb{N}$, then* vMPShortPathsAlg$(G, s, t, r) = 1$ *if and only if there is a directed path from s to t in G of length at most r. Furthermore,* vMPShortPathsAlg *runs in time* $\mathrm{poly}(n)$ *and space $O(\log n \sqrt{\log r})$.*

Proof sketch. Again, we assume that the reader is familiar with the paper by van Melkebeek and Prakriya [40]. Again, we first apply a reduction, giving a layered digraph G' of width $|V(G)|$ and length r, so that the question is whether there is a path from the first vertex in the first layer to the first vertex in the last layer.

We rely on the "combined hashing and shifting" generator by van Melkebeek and Prakriya [40, Theorem 1]. The seed of this generator specifies $O(\sqrt{\log r})$ hash functions (each is specified with $O(\log n)$ bits). We find these hash functions by exhaustive search one at a time, maintaining the invariant that portions of G' that have weights assigned are min-unique. We test for min-uniqueness using a slight variant of the algorithm by Reinhardt and Allender [31] described by van Melkebeek and Prakriya [40, Lemma 1]. ◀

Roughly speaking, Theorem 43 plays a role in our disambiguation results that is analogous to the role that the Nisan-Zuckerman generator played in our derandomization results.

6.4 Disambiguating Branching Programs

For us, a *nondeterministic branching program* $\mathcal{P}$ on $\{0,1\}^n \times \{0,1\}^m$ is a randomized branching program (but we think of the second input to the program as nondeterministic bits instead of random bits) such that some vertex $v_0 \in V(\mathcal{P})$ is labeled as the *start vertex* and some vertex $v_{\text{accept}} \in V(\mathcal{P})$ is labeled as the *accepting vertex.* We identify $\mathcal{P}$ with a function $\mathcal{P} : \{0,1\}^n \times \{0,1\}^m \to \{0,1\}$ defined by

$$\mathcal{P}(x,y) = \begin{cases} 1 & \text{if } \mathcal{P}(v_0; x, y) = v_{\text{accept}}, \\ 0 & \text{otherwise,} \end{cases} \tag{91}$$

and we *also* identify $\mathcal{P}$ with a function $\mathcal{P} : \{0,1\}^n \to \{0,1\}$ defined by

$$\mathcal{P}(x) = 1 \iff \exists y \, \mathcal{P}(x,y) = 1. \tag{92}$$

(Equation (92) expresses the fact that $\mathcal{P}$ is a *nondeterministic* branching program.) Finally, an *R-OW* nondeterministic branching program is just a nondeterministic branching program that is R-OW when thought of as a randomized branching program, i.e., it reads its nondeterministic bits from left to right.

▶ **Theorem 44.** *For every constant $c \in \mathbb{N}$, there is a single-valued unambiguous algorithm* A *with the following properties. Suppose $\mathcal{P}$ is an R-OW nondeterministic branching program on $\{0,1\}^n \times \{0,1\}^T$. Suppose $S \geq \log n$, where $S \overset{\text{def}}{=} \lceil \log \operatorname{size}(\mathcal{P}) \rceil$, and $T \geq \operatorname{length}(\mathcal{P})$. Then*

$$\operatorname{density}\{x \in \{0,1\}^n : \mathsf{A}(\mathcal{P}, x, T) \neq \mathcal{P}(x)\} \leq 2^{-S^c}. \tag{93}$$

Furthermore, $\mathsf{A}(\mathcal{P}, x, T)$ runs in time $2^{O(S)}$ and space $O(S\sqrt{\log\lceil T/n \rceil + \log S})$.

Toward proving Theorem 44, we introduce some notation. The computation of $\mathcal{P}(x)$ naturally reduces to STConn. Let $\mathcal{P}[x]$ be the digraph (V, E), where $V = V(\mathcal{P})$ and E is the set of edges (u, v) in $\mathcal{P}$ labeled with $x_{i(u)}0$ or $x_{i(u)}1$. (So every nonterminal vertex in $\mathcal{P}[x]$ has outdegree 2.) That way, $\mathcal{P}(x) = 1$ if and only if $(\mathcal{P}[x], v_0, v_{\text{accept}}) \in \text{STConn}$.

The algorithm A of Theorem 44 is given in Algorithm 6. The algorithm relies on a subroutine B given in Algorithm 7.

Parameters

Let s be the number of random bits used by $\mathsf{vMPSeededAlg}$, so that $s \leq O(S^2)$. The subroutine B relies on the extractor GUVExt of Theorem 6. This extractor is instantiated with source length $\ell \overset{\text{def}}{=} S^{c+1}$, error 0.1, entropy $k \overset{\text{def}}{=} 2s$, and output length s. The seed length of GUVExt is $d \leq O(\log \ell) = O(\log S)$.

Efficiency

First, we bound the space complexity of A. If $S^{c+1} > n$, then A runs in space

$$O(\log \operatorname{size}(\mathcal{P})\sqrt{\log T}) = O(S\sqrt{\log T}) \leq O\left(S\sqrt{\log \frac{TS^{c+1}}{n}}\right) \tag{94}$$

$$= O(S\sqrt{\log(T/n) + \log S}). \tag{95}$$

Algorithm 6: The algorithm A of Theorem 44.

```
if $S^{c+1} > n$ then
    return vMPShortPathsAlg($\mathcal{P}[x], v_0, v_{accept}, T$)
else
    Let $I_1, I_2, \dots, I_B \subseteq [n]$ be disjoint sets of size $S^{c+1}$ with $B$ as large as possible
    for $b = 1$ to $B$ do
        Let $I \leftarrow I_b$
        Let $V_b \leftarrow \{v \in V(\mathcal{P}) : i(v) \in I\} \cup \{v_0, v_{\text{accept}}\}$
        Let $E_b$ be the set of pairs $(u, v) \in V_b^2$ such that there is a directed path from
          $u$ to $v$ in $\mathcal{P}|_{[n]\setminus I}[x]$
        Let $H_b$ be the digraph $(V_b, E_b)$
        Compute $a \leftarrow$ vMPShortPathsAlg($H_b, v_0, v_{\text{accept}}, \lfloor S^{c+1}T/n \rfloor + 1$). Whenever
          vMPShortPathsAlg asks whether some pair $(u, v)$ is in $E_b$, run B($\mathcal{P}, x, b, u, v$),
          where B is given in Algorithm 7
        if $a = 1$ then return 1
    end
    return 0
end
```

Algorithm 7: The algorithm B used by A to decide whether $(u, v) \in E_b$. The block I is the same block I_b used by A.

```
for $y \in \{0,1\}^{O(\log S)}$ do
    Let $a' \leftarrow$ vMPSeededAlg($\mathcal{P}|_{[n]\setminus I}[x], u, v$, GUVExt($x|_I, y$))
    if $a' \neq \bot_{\mathrm{r}}$ then return $a'$
end
return $\bot_{\mathrm{i}}$
```

Suppose now that $S^{c+1} \leq n$. The extractor GUVExt runs in space $O(\log S)$, and the algorithm vMPSeededAlg runs in space $O(S)$, so B runs in space $O(S)$. The algorithm vMPShortPathsAlg runs in space

$$O(\log |V_b| \sqrt{\log(\lfloor S^{c+1}T/n \rfloor + 1)}) \leq O(S\sqrt{\log\lceil T/n \rceil + \log S}). \tag{96}$$

Therefore, overall, A runs in space $O(S\sqrt{\log\lceil T/n \rceil + \log S})$.

Next, we bound the running time of A. If $S^{c+1} > n$, then A runs in time $\text{poly}(\text{size}(\mathcal{P})) = 2^{O(S)}$ as claimed. Suppose now that $S^{c+1} \leq n$. Because B runs in space $O(S)$, it must run in time $2^{O(S)}$. Therefore, vMPShortPathsAlg runs in time $2^{O(S)} \cdot 2^{O(S)} = 2^{O(S)}$. Therefore, overall, A runs in time $2^{O(S)}$.

Correctness

Since vMPSeededAlg and vMPShortPathsAlg are single-valued unambiguous algorithms, A is a single-valued unambiguous algorithm. All that remains is to show that for most x, $\mathsf{A}(\mathcal{P}, x, T) = \mathcal{P}(x)$. First, we show that for most x, the subroutine B is correct, i.e., the one computation path that does not output $\bot_{\mathrm{n}}$ outputs a bit indicating whether $(u, v) \in E_b$. Clearly, the only way that B can be incorrect is if it outputs $\bot_{\mathrm{i}}$, indicating a "hard" input x.

▷ Claim 45. For every $\mathcal{P}$,

$$\text{density}\{x \in \{0,1\}^n : \exists b, u, v \text{ such that } \mathsf{B}(\mathcal{P}, x, b, u, v) = \bot_{\mathrm{i}}\} \leq 2^{-S^c}. \tag{97}$$

Proof. The graph $\mathcal{P}|_{[n]\setminus I}[x]$ does not depend on $x|_I$. Therefore, for each fixed b, each fixed $z \in \{0,1\}^{n-|I_b|}$, and each fixed $u, v \in V(\mathcal{P})$, by Proposition 7,

$$\#\{x : x|_{[n]\setminus I} = z \text{ and } \mathsf{B}(\mathcal{P}, x, b, u, v) = \bot_{\mathrm{i}}\} \leq 2^{k+2} \leq 2^{O(S^4)}. \tag{98}$$

Therefore, by summing over all b, z, u, v,

$$\begin{aligned} \#\{x \in \{0,1\}^n : \exists b, u, v \text{ such that } \mathsf{B}(\mathcal{P}, x, b, u, v) = \bot_{\mathrm{i}}\} &\leq 2^{n - S^{c+1} + \log n + 2S + O(S^4)} && (99)\\ &= 2^{n - S^{c+1} + O(S^4)} && (100)\\ &\leq 2^{n - S^c} && (101) \end{aligned}$$

for sufficiently large n. ◁

Next, we show that as long as B does not make any mistakes, A is correct.

▷ Claim 46. If $\mathcal{P}(x) = 1$, there is some $b \in [B]$ so that there is a path from v_0 to v_{accept} through H_b of length at most $\lfloor S^{c+1}T/n \rfloor + 1$.

Proof. Since $\mathcal{P}(x) = 1$, there is a path from v_0 to v_{accept} through $\mathcal{P}[x]$ of length at most T. Let $v_0, v_1, v_2, \ldots, v_{T'} = v_{\text{accept}}$ be the vertices visited by that path, so that $T' \leq T$. Consider picking $b \in [B]$ uniformly at random. Then for each $t < T'$, $\Pr[i(v_t) \in I_b] \leq S^{c+1}/n$. Therefore, by linearity of expectation,

$$\mathrm{E}[\#\{t : i(v_t) \in I_b\}] \leq S^{c+1}T/n. \tag{102}$$

The best case is at least as good as the average case, so there is some $b \in [B]$ such that $\#\{t : i(v_t) \in I_b\} \leq S^{c+1}T/n$. Let $t_1, t_2, \ldots, t_r$ be the indices t such that $i(v_t) \in I_b$. Then by the definition of E_b, the edges $(v_0, t_1), (t_1, t_2), \ldots, (t_{r-1}, t_r), (t_r, v_{\text{accept}})$ are all present in H_b. Therefore, there is a path from v_0 to v_{accept} through H_b of length at most $r + 1$. ◁

Combining Claims 45 and 46 completes the proof of Theorem 44.

6.5 Disambiguating Uniform Random-Access Algorithms

▶ **Corollary 47.** *For every space-constructible function $S(n) \geq \log n$, for every constant $c \in \mathbb{N}$,*

$$\mathbf{NTISP}(n \cdot \mathrm{poly}(S), S) \textit{ is within } 2^{-S^c} \textit{ of } \mathbf{UTISP}(2^{O(S)}, S\sqrt{\log S}). \tag{103}$$

Proof sketch. The class $\mathbf{NTISP}(n \cdot \mathrm{poly}(S), S)$ corresponds to R-OW nondeterministic branching programs of size $2^{O(S)}$ and length $T = n \cdot \mathrm{poly}(S)$. For these parameters, the algorithm of Theorem 44 runs in time $2^{O(S)}$ and space $O(S\sqrt{\log S})$. ◀

6.6 Disambiguation with Advice

We now show how to disambiguate $\mathbf{NL}$ with only $n + O(\log^2 n)$ bits of advice. The proof is very similar to the proof of Theorem 34.

▶ **Theorem 48.** $\mathbf{NL} \subseteq \mathbf{UL}/(n + O(\log^2 n))$.

Proof. Let $\mathcal{R}$ be a log-space reduction from $L \in \mathbf{NL}$ to STConn. Let s be the number of random bits used by $\mathsf{vMPSeededAlg}$ on inputs of length n^c, where n^c is the length of outputs of $\mathcal{R}$ on inputs of length n. Let $\mathsf{GUVExt} : \{0,1\}^{n+2s+3} \times \{0,1\}^d \to \{0,1\}^s$ be the $(2s, 0.1)$-extractor of Theorem 6, so that $d \leq O(\log n)$.

Given input $x \in \{0,1\}^n$ and advice $a \in \{0,1\}^{n+2s+3}$, compute

$$a_z \overset{\text{def}}{=} \mathsf{vMPSeededAlg}(\mathcal{R}(x), \mathsf{GUVExt}(a, z)) \tag{104}$$

for all z and accept if there is some z so that $a_z = 1$.

This algorithm clearly runs in space $O(\log n)$ and is unambiguous. By Proposition 7, for each fixed x, the number of advice strings a causing the algorithm to give the wrong answer is at most 2^{2s+2}. Therefore, the total number of advice strings a that cause the algorithm to give the wrong answer for any x is at most $2^{n+2s+2} < 2^{|a|}$. Therefore, there is some choice of a such that the algorithm succeeds on all inputs. ◀

Just like we did with Theorem 34, we now generalize Theorem 48, showing that the amount of advice can be reduced to below n if we start with a language that has a typically-correct disambiguation.

▶ **Theorem 49.** *Fix functions $S : \mathbb{N} \to \mathbb{N}$ with $S(n) \geq \log n$ and $\varepsilon : \mathbb{N} \to [0,1]$ that are constructible in $O(S)$ space. Suppose a language $L \in \mathbf{NSPACE}(S)$ is within ε of $\mathbf{USPACE}(S)$. Then*

$$L \in \mathbf{USPACE}(S)/(n - \log_2(1/\varepsilon(n)) + O(S^2)). \tag{105}$$

The proof of Theorem 49 is very similar to the proof of Theorem 37. Because the proof of Theorem 49 does not introduce any significantly new techniques, we defer the proof to Appendix E.

▶ **Corollary 50.** *For every constant $c \in \mathbb{N}$,*

$$\mathbf{NTISP}(n \operatorname{polylog} n, \log n) \subseteq \mathbf{USPACE}(\log n \sqrt{\log \log n})/(n - \log^c n). \tag{106}$$

Proof. For any $L \in \mathbf{NTISP}(n \operatorname{polylog} n, \log n)$, obviously $L \in \mathbf{NSPACE}(\log n \sqrt{\log \log n})$, and by Corollary 47, L is within $2^{-\log^c n}$ of $\mathbf{USPACE}(\log n \sqrt{\log \log n})$. Applying Theorem 49 completes the proof. ◀

7 Directions for Further Research

The main open problem in this area is to prove that $\mathbf{BPL}$ is within $o(1)$ of $\mathbf{L}$. Corollary 18 implies that $\mathbf{BPTISP}(n \operatorname{polylog} n, \log n)$ is within $o(1)$ of $\mathbf{L}$, and Corollary 33 implies that $\mathbf{BPTISP}_{\mathrm{TM}}(n^{1.99}, \log n)$ is within $o(1)$ of $\mathbf{L}$, but $\mathbf{BPL}$ allows time n^c where c is an arbitrarily large constant. At present, for a generic language $L \in \mathbf{BPL}$, we do not even know a deterministic log-space algorithm that succeeds on at least *one* input of each length.

This work also provides some additional motivation for studying small-space extractors. The two extractors we used in this paper (Theorems 5 and 6) were sufficient for our applications, but it would be nice to have a single log-space extractor that is optimal up to constants for the full range of parameters.

References

1 Leonard Adleman. Two theorems on random polynomial time. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science (FOCS '78)*, pages 75–83. IEEE, 1978.

2 Eric Allender, Klaus Reinhardt, and Shiyu Zhou. Isolation, matching, and counting uniform and nonuniform upper bounds. *Journal of Computer and System Sciences*, 59(2):164–181, 1999. doi:10.1006/jcss.1999.1646.

3 Josh Alman. An illuminating algorithm for the light bulb problem. In *2nd Symposium on Simplicity in Algorithms*, volume 69 of *OASIcs OpenAccess Ser. Inform.*, pages Art. No. 2, 11. Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2019.

4 Roy Armoni. On the derandomization of space-bounded computations. In *Proceedings of the 2nd International Workshop on Randomization and Computation (RANDOM '98)*, volume 1518 of *Lecture Notes in Computer Science*, pages 47–59. Springer, Berlin, 1998. doi:10.1007/3-540-49543-6_5.

5 Vikraman Arvind and Jacobo Toran. Solvable group isomorphism is (almost) in NP ∩ coNP. In *Proceedings of the 19th Annual Conference on Computational Complexity (CCC '04)*, pages 91–103. IEEE, 2004.

6 László Babai, Lance Fortnow, Noam Nisan, and Avi Wigderson. BPP has subexponential time simulations unless EXPTIME has publishable proofs. *Computational Complexity*, 3(4):307–318, 1993. doi:10.1007/BF01275486.

7 László Babai, Noam Nisan, and Márió Szegedy. Multiparty protocols, pseudorandom generators for logspace, and time-space trade-offs. *Journal of Computer and System Sciences*, 45(2):204–232, 1992. doi:10.1016/0022-0000(92)90047-M.

8 Paul Beame, Michael Saks, Xiaodong Sun, and Erik Vee. Time-space trade-off lower bounds for randomized computation of decision problems. *Journal of the ACM*, 50(2):154–195, 2003. doi:10.1145/636865.636867.

9 Manuel Blum and Silvio Micali. How to generate cryptographically strong sequences of pseudorandom bits. *SIAM Journal on Computing*, 13(4):850–864, 1984. doi:10.1137/0213053.

10 Jin-Yi Cai, Venkatesan T. Chakaravarthy, and Dieter van Melkebeek. Time-space tradeoff in derandomizing probabilistic logspace. *Theory of Computing Systems*, 39(1):189–208, 2006. doi:10.1007/s00224-005-1264-9.

11 Scott Diehl and Dieter van Melkebeek. Time-space lower bounds for the polynomial-time hierarchy on randomized machines. *SIAM Journal on Computing*, 36(3):563–594, 2006. doi:10.1137/050642228.

12 Lance Fortnow and Adam R. Klivans. Linear advice for randomized logarithmic space. In *Proceedings of the 23rd Annual Symposium on Theoretical Aspects of Computer Science (STACS '06)*, volume 3884 of *Lecture Notes in Computer Science*, pages 469–476. Springer, Berlin, 2006. doi:10.1007/11672142_38.

13 Lance Fortnow and Dieter van Melkebeek. Time-space tradeoffs for nondeterministic computation. In *Proceedings of the 15th Annual Conference on Computational Complexity (CCC '00)*, pages 2–13. IEEE, 2000. doi:10.1109/CCC.2000.856730.

14 David Gillman. A Chernoff bound for random walks on expander graphs. *SIAM Journal on Computing*, 27(4):1203–1220, 1998. doi:10.1137/S0097539794268765.

15 Oded Goldreich and Avi Wigderson. Derandomization that is rarely wrong from short advice that is typically good. In *Randomization and approximation techniques in computer science (RANDOM '02)*, volume 2483 of *Lecture Notes in Computer Science*, pages 209–223. Springer, Berlin, 2002. doi:10.1007/3-540-45726-7_17.

16 Venkatesan Guruswami, Christopher Umans, and Salil Vadhan. Unbalanced expanders and randomness extractors from Parvaresh-Vardy codes. *Journal of the ACM*, 56(4):Art. 20, 34, 2009. doi:10.1145/1538902.1538904.

17 Dan Gutfreund and Emanuele Viola. Fooling parity tests with parity gates. In *Proceedings of the 8th International Workshop on Randomization and Computation (RANDOM '04)*, volume 3122 of *Lecture Notes in Computer Science*, pages 381–392. Springer, 2004.

18 Tzvika Hartman and Ran Raz. On the distribution of the number of roots of polynomials and explicit weak designs. *Random Structures & Algorithms*, 23(3):235–263, 2003. doi:10.1002/rsa.10095.

19 Lane A Hemaspaandra and Ryan Williams. SIGACT news complexity theory column 76: an atypical survey of typical-case heuristic algorithms. *ACM SIGACT News*, 43(4):70–89, 2012.

20 Russell Impagliazzo and Avi Wigderson. P = BPP if E requires exponential circuits: Derandomizing the XOR lemma. In *Proceedings of the 29th Annual Symposium on Theory of Computing (STOC '97)*, pages 220–229, New York, NY, USA, 1997. ACM. doi:10.1145/258533.258590.

21 Neil D. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Sciences*, 11(1):68–85, 1975. doi:10.1016/S0022-0000(75)80050-X.

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

23 Neeraj Kayal and Nitin Saxena. On the ring isomorphism & automorphism problems. In *Proceedings of the 20th Annual Conference on Computational Complexity (CCC '05)*, pages 2–12. IEEE, 2005.

24 Jeff Kinne, Dieter van Melkebeek, and Ronen Shaltiel. Pseudorandom generators, typically-correct derandomization, and circuit lower bounds. *Computational Complexity*, 21(1):3–61, 2012. doi:10.1007/s00037-011-0019-z.

25 Adam R. Klivans and Dieter van Melkebeek. Graph nonisomorphism has subexponential size proofs unless the polynomial-time hierarchy collapses. *SIAM Journal on Computing*, 31(5):1501–1526, 2002. doi:10.1137/S0097539700389652.

26 Noam Nisan. Pseudorandom generators for space-bounded computation. *Combinatorica*, 12(4):449–461, 1992. doi:10.1007/BF01305237.

27 Noam Nisan. On read-once vs. multiple access to randomness in logspace. *Theoretical Computer Science*, 107(1):135–144, 1993. doi:10.1016/0304-3975(93)90258-U.

28 Noam Nisan. RL $\subseteq$ SC. *Computational Complexity*, 4(1):1–11, 1994. doi:10.1007/BF01205052.

29 Noam Nisan and Avi Wigderson. Hardness vs. randomness. *Journal of Computer and System Sciences*, 49(2):149–167, 1994. doi:10.1016/S0022-0000(05)80043-1.

30 Noam Nisan and David Zuckerman. Randomness is linear in space. *Journal of Computer and System Sciences*, 52(1):43–52, 1996. doi:10.1006/jcss.1996.0004.

31 Klaus Reinhardt and Eric Allender. Making nondeterminism unambiguous. *SIAM Journal on Computing*, 29(4):1118–1131, 2000. doi:10.1137/S0097539798339041.

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

33 Rahul Santhanam and Ryan Williams. On uniformity and circuit lower bounds. *Computational Complexity*, 23(2):177–205, 2014. doi:10.1007/s00037-014-0087-y.

34 Ronen Shaltiel. Typically-correct derandomization. *ACM SIGACT News*, 41(2):57–72, 2010.

35 Ronen Shaltiel. Weak derandomization of weak algorithms: explicit versions of Yao's lemma. *Computational Complexity*, 20(1):87–143, 2011. doi:10.1007/s00037-011-0006-4.

36 Ronen Shaltiel and Christopher Umans. Simple extractors for all min-entropies and a new pseudorandom generator. *Journal of the ACM*, 52(2):172–216, 2005. doi:10.1145/1059513.1059516.

37 Madhu Sudan, Luca Trevisan, and Salil Vadhan. Pseudorandom generators without the XOR lemma. *Journal of Computer and System Sciences*, 62(2):236–266, 2001. Special issue on the 14th Annual Conference on Computational Complexity (CCC '99). doi:10.1006/jcss.2000.1730.

38 Salil P. Vadhan. Pseudorandomness. *Foundations and Trends® in Theoretical Computer Science*, 7(1–3):1–336, 2012. doi:10.1561/0400000010.

39 J. H. van Lint. *Introduction to coding theory*, volume 86 of *Graduate Texts in Mathematics*. Springer-Verlag, Berlin, third edition, 1999. `doi:10.1007/978-3-642-58575-3`.

40 Dieter van Melkebeek and Gautam Prakriya. Derandomizing Isolation in Space-Bounded Settings. In *32nd Annual Conference on Computational Complexity (CCC '17)*, volume 79 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:32, Dagstuhl, Germany, 2017. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.CCC.2017.5`.

41 Dieter van Melkebeek and Rahul Santhanam. Holographic proofs and derandomization. *SIAM Journal on Computing*, 35(1):59–90, 2005. `doi:10.1137/S0097539703438629`.

42 Andrew C. Yao. Theory and applications of trapdoor functions. In *23rd Annual Symposium on Foundations of Computer Science (FOCS '82)*, pages 80–91. IEEE, New York, 1982.

43 Marius Zimand. Exposure-resilient extractors and the derandomization of probabilistic sublinear time. *Computational Complexity*, 17(2):220–253, 2008. `doi:10.1007/s00037-008-0243-3`.

44 David Zuckerman. Randomness-optimal oblivious sampling. *Random Structures & Algorithms*, 11(4):345–367, 1997. `doi:10.1002/(SICI)1098-2418(199712)11:4<345::AID-RSA4>3.0.CO;2-Z`.

A Proof of Theorem 5: The Shaltiel-Umans Extractor

In this section, we discuss the proof of Theorem 5. The extractor follows the same basic construction that Shaltiel and Umans used for a "low error" extractor [36, Corollary 4.21]. We will assume that the reader is familiar with the paper by Shaltiel and Umans [36]. We will also switch to the parameter names by Shaltiel and Umans, so the source length of the extractor is n rather than ℓ, and the seed length is t rather than d. In these terms, we are shooting for time $\mathrm{poly}(n)$ and space $O(t)$.

The only change to the *construction* that we make is that we will use a different instantiation of the "base field" $\mathbb{F}_q$. Shaltiel and Umans [36] used a deterministic algorithm by Shoup that finds an irreducible polynomial of degree $\log q$ over $\mathbb{F}_2$ in time $\mathrm{poly}(\log q)$. Unfortunately, Shoup's algorithm is not sufficiently space-efficient for our purposes. To get around this issue, we use an extremely explicit family of irreducible polynomials:

▶ **Lemma 51** ([39, Theorem 1.1.28]). *For every $a \in \mathbb{N}$, the polynomial $x^{2\cdot 3^a} + x^{3^a} + 1$ is irreducible over $\mathbb{F}_2$.*

Therefore, by replacing q by some power of two between q and q^3, we can easily, deterministically construct an irreducible polynomial of degree $\log q$ in time $\mathrm{poly}(\log q)$ and space $O(\log q)$. This only affects the bit length of field elements, $\log q$, by at most a factor of 3. Therefore, the hypotheses of Shaltiel and Umans' main technical theorem [36, Theorem 4.5] are still met, so the extractor is still correct.

Now we turn to analyzing the efficiency of the extractor. The parameters h, d, m, ρ, q used by Shaltiel and Umans (with the described modification to q) can all easily be computed in time $\mathrm{poly}(n)$ and space $O(t)$. Next, we inspect the construction of the matrix B used by Shaltiel and Umans [36, Proof of Lemma 4.18]. The exhaustive search used to find the irreducible polynomial $p(z)$ takes space $O(d \log q) \leq O(t)$. The exhaustive search used to find the generator g for $(H^d)^\times$ also takes space $O(d \log q) = O(t)$. Finally, multiplication by g takes space $O(d \log q) = O(t)$.

It follows immediately that the "q-ary extractor" E' given by Shaltiel and Umans [36, Equation 8] runs in space $O(t)$, because we only need to store the vector $B^i\vec{v}$. Finally, to get from E' to the final extractor, a simple Hadamard code is applied, which can trivially be computed in time $\mathrm{poly}(n)$ and space $O(t)$.

B Proof of Theorem 6: The GUV Extractor

In this section, we discuss the proof of Theorem 6. We will assume that the reader is familiar with the paper by Guruswami, Umans, and Vadhan. Recall that a *condenser* is like an extractor, except that the output is merely guaranteed to be close to having high entropy instead of being guaranteed to be close to uniform.

▶ **Definition 52.** *A function* $\mathsf{Con} : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^{n'}$ *is a* $k \to_\varepsilon k'$ condenser *if for every random variable* X *with* $H_\infty(X) \geq k$*, there exists a distribution* Z *with* $H_\infty(Z) \geq k'$ *such that if we let* $Y \sim U_d$ *be independent of* X*, then* $\mathsf{Con}(X, Y) \sim_\varepsilon Z$*.*

Guruswami, Umans, and Vadhan constructed a lossy condenser based on folded Reed-Solomon codes [16, Theorem 6.2]. To ensure space efficiency, we will slightly modify their construction to get the following condenser. We will follow the parameter names by Guruswami, Umans, and Vadhan.

▶ **Theorem 53** (Based on [16, Theorem 6.2])**.** *Let* $\alpha > 0$ *be a constant. Consider any* $n \in \mathbb{N}, \ell \leq n$ *such that* 2^ℓ *is an integer and any* $\varepsilon > 0$*. There is a parameter* $t = \Theta(\log(n\ell/\varepsilon))$ *and a*

$$(1 + 1/\alpha)\ell t + \log(1/\varepsilon) \to_{3\varepsilon} \ell t + d - 2$$

condenser $\mathsf{GUVCon} : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^{n'}$*, computable in space* $O(d)$*, with seed length* $d \leq (1 + 1/\alpha)t$ *and output length* $n' \leq (1 + 1/\alpha)\ell t + d$*, provided* $\ell t \geq \log(1/\varepsilon)$*.*

Proof sketch. We need to use a base field $\mathbb{F}_q$ based on Lemma 51, so we slightly modify the parameters of the GUV construction as follows. Choose q to be the smallest power of two of the form $2^{2\cdot 3^a}$ such that $q \geq (2^{2+1/\alpha} \cdot n\ell/\varepsilon)^{1+\alpha}$. This q satisfies $q \leq (2^{2+1/\alpha} \cdot n\ell/\varepsilon)^{3+3\alpha}$. Next, define $t = \lceil \frac{\alpha \log q}{1+\alpha} \rceil$ and $h = 2^t$, so that $q \in ((h/2)^{1+1/\alpha}, h^{1+1/\alpha}]$. Therefore, we still have

$$q > h \cdot h^{1/\alpha}/2^{1+1/\alpha} \tag{107}$$
$$\geq h \cdot q^{1/(1+\alpha)}/2^{1+1/\alpha} \tag{108}$$
$$\geq 2hn\ell/\varepsilon, \tag{109}$$

and hence $A \geq \varepsilon q/2$. The rest of the argument is as in the original paper [16]. ◀

There is a standard extractor based on expander walks that works well for constant error and constant entropy rate. Using the Gutfreund-Viola expander walk (Theorem 31), this extractor runs in logarithmic space:

▶ **Lemma 54.** *Let* $\alpha, \varepsilon > 0$ *be constants. There is some constant* $\beta \in (0, 1)$ *so that for all* n*, there is a* $(\beta n, \varepsilon)$*-extractor* $\mathsf{GVExt} : \{0,1\}^n \times \{0,1\}^d \to \{0,1\}^m$ *with* $t \leq \log(\alpha n)$ *and* $m \geq (1 - \alpha)n$ *so that given* x *and* y*,* $\mathsf{GVExt}(x, y)$ *can be computed in* $O(\log n)$ *space.*

Proof sketch. This construction of an extractor from an expander is standard; see, e.g., an exposition by Guruswami et al. [16, Theorem 4.6]. The space bound follows from Theorem 31. ◀

Finally, Theorem 6 follows by composing Theorem 53 and Lemma 54, just as is explained in the paper by Guruswami et al. [16, Theorem 4.7].

C Proof of Proposition 7: Extractors Are Good Samplers

Let $X \subseteq \{0,1\}^\ell$ be the set on the left-hand side of Equation (8). Since total variation distance is half ℓ_1 distance, for each $x \in X$,

$$\sum_{v \in V} |\Pr[f(U_s) = v] - \Pr[f(\mathsf{Ext}(x, U_d)) = v]| > \varepsilon |V|. \tag{110}$$

Therefore, by the triangle inequality, for each $x \in X$, there is some $v_x \in V$ such that

$$|\Pr[f(U_s) = v_x] - \Pr[f(\mathsf{Ext}(x, U_d)) = v_x]| > \varepsilon. \tag{111}$$

Partition $X = X_1 \cup \cdots \cup X_{|V|}$, where $\mathcal{X}_v = \{x \in X : v_x = v\}$. For each v, we can further partition X_v into $X_v^+ \cup X_v^-$, based on which term of the left hand side of Equation (111) is bigger.

Identify X_v^+ with a random variable that is uniformly distributed over the set X_v^+, and let $Y \sim U_d$ be independent of X_v^+. Then

$$\Pr[\mathsf{Ext}(X_v^+, Y) \in f^{-1}(v_x)] > \Pr[U_s \in f^{-1}(v_x)] + \varepsilon. \tag{112}$$

Therefore, by the extractor condition, $|X_v^+| \leq 2^k$. Similarly, $|X_v^-| \leq 2^k$, and hence $|X_v| \leq 2^{k+1}$. By summing over all v, we conclude that $|X| \leq 2^{k+1}|V|$ as claimed.

D Proof of Theorem 20: Derandomizing S-OW Branching Programs

The algorithm A of Theorem 20 is given in Algorithm 8. The analysis is similar to the proof of Theorem 8. The main difference is when we argue that the second hybrid distribution, H_2, simulates $\mathcal{P}$. (This argument has just two hybrid distributions.) Details follow.

Parameters

Just like in the proof of Theorem 8, we can assume without loss of generality that $T \leq 2^S$. The block size h in Algorithm 8 is

$$h \overset{\text{def}}{=} \left\lfloor \frac{n}{3S^{c+1}} \right\rfloor. \tag{113}$$

Note that this time, the number of phases, r, is $\lceil T/h \rceil$, where h is the *block size*, in contrast to the proof of Theorem 8, where the number of phases was roughly T/B, where B is the *number of blocks*.

The algorithm A relies on Nisan's generator NisGen (Theorem 9). Naturally, the generator is instantiated with parameters S, T from the statement of Theorem 20. The error of NisGen is set at $\varepsilon \overset{\text{def}}{=} \frac{\exp(-cS)}{2r}$, just like in the proof of Theorem 8. Again, the seed length of NisGen is $s \leq O(S \log T) \leq O(S^2)$.

The algorithm A also relies on the Shaltiel-Umans extractor SUExt of Theorem 5. This extractor is instantiated with source length $\ell \overset{\text{def}}{=} n - 3h$, $\alpha \overset{\text{def}}{=} 1/2$, error

$$\varepsilon' \overset{\text{def}}{=} \frac{\exp(-cS)}{r \cdot 2^S}, \tag{114}$$

and entropy $k \overset{\text{def}}{=} \sqrt{n}$. This choice of k meets the hypotheses of Theorem 5, because $\log^{4/\alpha} \ell \leq \log^8 n \leq k$, and $S^{c+1} \leq \sqrt{n}$, so $\log^{4/\alpha}(1/\varepsilon) \leq \operatorname{polylog} n \leq k$. Furthermore, by construction, $k^{1-\alpha} = n^{1/4} \geq s$ as long as $c \geq 4$ and n is sufficiently large, so we can think of SUExt_2 as outputting s bits.

Algorithm 8: The algorithm A of Theorem 20.

```
if $S^{c+1} > \sqrt{n}$ then
    Directly simulate $\mathcal{P}(v_0; x, U_T)$ using $T$ random bits
else
    Partition $[n]$ into disjoint blocks, $[n] = I_1 \cup I_2 \cup \cdots \cup I_B$, where $|I_b| \approx h$. More
      precisely, let $B = \lceil n/h \rceil$, and let
      $I_b \leftarrow \{h \cdot (b-1)+1, h \cdot (b-1)+2, \dots, \min\{h \cdot b, n\}\}$
    Let $I_0, I_{B+1} \leftarrow \varnothing$
    for $b \in [B]$ do
        Let $I'_b \leftarrow [n] \setminus (I_{b-1} \cup I_b \cup I_{b+1})$, with the largest elements removed so that
          $|I'_b| = n - 3h$
    end
    Initialize $v \leftarrow v_0$
    repeat $r$ times                                        /* Here $r \overset{\text{def}}{=} \lceil T/h \rceil$ */
        Let $b \in [B]$ be such that $i(v) \in I_b$
        Let $I \leftarrow I'_b$
        Pick $y \in \{0,1\}^{O(S)}$ uniformly at random
        Update $v \leftarrow \mathcal{P}|_{[n]\setminus I}(v; x, \mathsf{NisGen}(\mathsf{SUExt}(x|_I, y)))$
    end
    return $v$
end
```

Efficiency

The runtime analysis of A is essentially the same as in the proof of Theorem 8; the only substantial difference is that the input to SUExt has length $\Theta(n)$, so SUExt takes $\mathrm{poly}(n)$ time instead of $\mathrm{poly}(S)$ time. Thus, overall, A runs in time $T \cdot \mathrm{poly}(n, S)$. The space complexity and randomness complexity analyses are essentially the same as in the proof of Theorem 8.

Correctness

The proof of Equation (37) has the same structure as the proof of Equation (9). Assume without loss of generality that $S^{c+1} \leq \sqrt{n}$. The first hybrid distribution is defined by Algorithm 9. The number of "bad" inputs in Claim 55 is much lower than the number of "bad" inputs in Claim 11; intuitively, this is because A uses a *much larger* portion of the input as a source of randomness compared to the algorithm of Theorem 8.

▷ Claim 55 ($\mathsf{A} \approx \mathsf{H}_1$). Recall that ε' is the error of SUExt. Then

$$\#\{x \in \{0,1\}^n : \mathsf{A}(\mathcal{P}, v_0, x, T) \not\sim_{\varepsilon' r \cdot 2^{S-1}} \mathsf{H}_1(\mathcal{P}, v_0, x, T)\} \leq 2^{n/S^c}. \tag{115}$$

Proof sketch. The proof follows exactly the same reasoning as the proof of Claim 11. The number of bad x values is bounded by

$$\begin{aligned}
\#\text{ bad } x &\leq B \cdot 2^S \cdot 2^{n-|I'_b|} \cdot 2^{k+S+1} && (116)\\
&\leq 2^{3h+\sqrt{n}+O(S)} && (117)\\
&\leq 2^{n/S^{c+1}+\sqrt{n}+O(S)} && (118)\\
&\leq 2^{3n/S^{c+1}} && (119)\\
&\leq 2^{n/S^c} && (120)
\end{aligned}$$

for sufficiently large n. ◁

Algorithm 9: The algorithm H_1 defining the first hybrid distribution used to prove Equation (37). The only difference between A and H_1 is that H_1 picks a uniform random seed for NisGen, instead of extracting the seed from the input.

```
Initialize $v \leftarrow v_0$
repeat $r$ times
    Let $b \in [B]$ be such that $i(v) \in I_b$
    Let $I \leftarrow I'_b$
    Pick $y' \in \{0,1\}^s$ uniformly at random
    Update $v \leftarrow \mathcal{P}|_{[n]\setminus I}(v; x, \mathsf{NisGen}(y'))$
end
return $v$
```

Algorithm 10: The algorithm H_2 defining the second hybrid distribution used to prove Equation (37). The only difference between H_1 and H_2 is that H_2 feeds true randomness to $\mathcal{P}|_{[n]\setminus I}$, instead of feeding it a pseudorandom string from Nisan's generator.

```
Initialize $v \leftarrow v_0$
repeat $r$ times
    Let $b \in [B]$ be such that $i(v) \in I_b$
    Let $I \leftarrow I'_b$
    Pick $y'' \in \{0,1\}^T$ uniformly at random
    Update $v \leftarrow \mathcal{P}|_{[n]\setminus I}(v; x, y'')$
end
return $v$
```

The second hybrid distribution is defined by Algorithm 10.

▷ Claim 56 ($\mathsf{H}_1 \approx \mathsf{H}_2$). For every x,

$$\mathsf{H}_1(\mathcal{P}, v_0, x, T) \sim_{\varepsilon r} \mathsf{H}_2(\mathcal{P}, v_0, x, T), \tag{121}$$

where ε is the error of NisGen.

Proof sketch. The proof is the same as that of Claim 12. ◁

All that remains is the final step of the hybrid argument. In this case, H_2 actually simulates $\mathcal{P}$ with *no* error. This argument is where we finally use the fact that $\mathcal{P}$ only has sequential access to its input.

▷ Claim 57 ($\mathsf{H}_2 \sim \mathcal{P}$). For every x,

$$\mathsf{H}_2(\mathcal{P}, v_0, x, T) \sim \mathcal{P}(v_0; x, U_T). \tag{122}$$

Proof sketch. The set I'_b chosen by H_2 excludes every index in $[n]$ that is within h of $i(v)$. Therefore, each iteration of the loop in H_2 simulates at least h steps of $\mathcal{P}$. Since $r \geq T/h$, overall, H_2 simulates at least T steps of $\mathcal{P}$. But $T \geq \text{length}(\mathcal{P})$, so we are done, just like in the proof of Claim 14. ◁

Proof of Theorem 20. By Claims 55 to 57 and the triangle inequality,

$$\#\{x \in \{0,1\}^n : \mathsf{A}(\mathcal{P}, v_0, x, t) \not\sim_\delta \mathcal{P}(v_0; x, U_T)\} \leq 2^{n/S^c}, \tag{123}$$

where $\delta = \varepsilon r + \varepsilon' r \cdot 2^{S-1}$. By our choice of ε, the first term is at most $e^{-cS}/2$. By our choice of ε', the second term is also at most $e^{-cS}/2$. Therefore, $\delta \leq e^{-cS}$. ◀

E Proof of Theorem 49: Disambiguation with Advice

We begin with randomness-efficient amplification of Theorem 42; Lemma 58 is analogous to Lemma 36, and its proof follows the same reasoning. The details are included only for completeness.

▶ **Lemma 58.** *Fix $S : \mathbb{N} \to \mathbb{N}$ with $S(n) \geq \log n$ and $\delta : \mathbb{N} \to [0,1]$, both constructible in space $O(S)$. For every $L \in \mathbf{NSPACE}(S)$, there is a single-valued unambiguous algorithm $\mathcal{A}$ so that for every $x \in \{0,1\}^n$,*

$$\Pr_{y \in \{0,1\}^\infty}[\mathcal{A}(x,y) \in \{L(x), \perp_{\mathrm{r}}\}] = 1, \tag{124}$$

$$\Pr_{y \in \{0,1\}^\infty}[\mathcal{A}(x,y) = \perp_{\mathrm{r}}] \leq \delta(n). \tag{125}$$

Furthermore, $\mathcal{A}$ only reads the first $\log_2(1/\delta(n)) + O(S^2)$ bits of y and runs in space $O(S)$.

Proof. Let $\mathcal{R}$ be an $O(S)$-space reduction from L to STConn. For $x \in \{0,1\}^n$, $\mathcal{R}(x) \in \{0,1\}^{\overline{n}}$, where $\overline{n} = 2^{O(S)}$, and without loss of generality, $\overline{n}$ depends only on n. Let s be the number of random bits used by vMPSeededAlg on inputs of length $\overline{n}$, so that $s \leq O(\log^2 \overline{n}) = O(S^2)$.

Let $\ell = \lceil \log_2(1/\delta) \rceil + 2s + 2$, and let $\mathsf{GUVExt} : \{0,1\}^\ell \times \{0,1\}^d \to \{0,1\}^s$ be the $(2s, 0.1)$-extractor of Theorem 6, so that $d \leq O(\log\log(1/\delta) + \log S)$. On input $x \in \{0,1\}^n, y \in \{0,1\}^\ell$, run Algorithm 11.

Algorithm 11: The algorithm used to prove Lemma 58.

```
for $z \in \{0,1\}^d$ do
    Let $a \leftarrow \mathsf{vMPSeededAlg}(\mathcal{R}(x), \mathsf{GUVExt}(y,z))$
    if $a \neq \perp_{\mathrm{r}}$ then return $a$
end
return $\perp_{\mathrm{r}}$
```

Clearly, Algorithm 11 runs in space $O(S + d)$. Since δ is constructible in space $O(S)$, its denominator must have at most $2^{O(S)}$ digits. Therefore, $\delta \geq 2^{-2^{O(S)}}$ and $d \leq O(S)$, so the algorithm runs in space $O(S)$. Furthermore, it is clearly single-valued unambiguous, and it is "zero-error", i.e., Equation (124) holds. Finally, by Proposition 7, the number of y such that $\mathsf{vMPSeededAlg}(\mathcal{R}(x), \mathsf{GUVExt}(y,z)) = \perp_{\mathrm{r}}$ for every z is at most 2^{2s+2}, and hence the probability that the algorithm outputs $\perp_{\mathrm{r}}$ is at most $\frac{2^{2s+2}}{2^\ell} \leq \delta$. ◀

Proof of Theorem 49. Let $\mathcal{A}$ be the algorithm of Lemma 58 with $\delta < 2^{-n}/\varepsilon$. Let $m = m(n)$ be the number of random bits used by $\mathcal{A}$. Let $\mathcal{B}$ be the algorithm witnessing the fact that L is within ε of $\mathbf{USPACE}(S)$.

Given input $x \in \{0,1\}^n$ and advice $a \in \{0,1\}^m$, compute $a = \mathcal{A}(x,a)$. If $a \neq \perp_{\mathrm{r}}$, output a. If $a = \perp_{\mathrm{r}}$, output $\mathcal{B}(x)$. This algorithm clearly runs in $O(S)$ space, uses $n - \log_2(1/\varepsilon(n)) + O(S^2)$ bits of advice, and is unambiguous (in fact, single-valued unambiguous).

Now we argue that there is some advice string such that the algorithm succeeds on all inputs. Let $S \subseteq \{0,1\}^n$ be the set of inputs on which $\mathcal{B}$ fails. Consider picking an advice string a uniformly at random. For each string $x \in S$, $\Pr_a[\mathcal{A}(x,a) = \perp_{\mathrm{r}}] \leq \delta$. Therefore, by the union bound, the probability that there is some $x \in S$ such that $\mathcal{A}(x,a) = \perp_{\mathrm{r}}$ is at most $|S|\delta = \varepsilon \cdot 2^n \cdot \delta < 1$. Therefore, there is *some* advice string such that the algorithm succeeds on all inputs in S. Finally, for *any* advice string, the algorithm succeeds on all inputs in $\{0,1\}^n \setminus S$ by Equation (124). ◀