On Recycling the Randomness of States in Space Bounded Computation

Ran Raz [*] Omer Reingold [†]

Abstract

Let M be a logarithmic space Turing machine (or a polynomial width branching program) that uses up to $k \approx 2^{\sqrt{\log n}}$ (read once) random bits. For a fixed input, let $P_i(S)$ be the probability (over the random string) that at time i the machine M is in state S, and assume that some weak estimation of the probabilities $P_i(S)$ is known or given or can be easily computed. We construct a logarithmic space pseudo-random generator that uses only logarithmic number of truly random bits and outputs a sequence of k bits that looks random to M.

This means that a very weak estimation of the state probabilities of M is sufficient for a full derandomization of M and for constructing pseudo-random sequences for M. We have several applications of the main theorem, as stated within.

To prove our theorem, we introduce the idea of recycling the state S of the machine M at time i as part of the random string for the same machine at later time. That is, we use the entropy of the random variable S in order to save truly random bits later on.

Our techniques and results can both be generalized to larger size of space.

1 Introduction

A sequence of results places **RL** (Randomized-Logspace) and **SL** (Symmetric-Logspace) lower and lower in the deterministic space hierarchy: **RL** $\subset$ **DSPACE**$(\log^2 n)$ [12] (and by pseudo-random generators in [6]). **SL** $\subset$ **DSPACE**$(\log^{3/2} n)$ [7]. And more recently, **RL** $\subset$ **DSPACE**$(\log^{3/2} n)$ [11], and **SL** $\subset$ **DSPACE**$(\log^{4/3} n)$ [3]. These results derandomize up to a polynomial number of (read once) random bits[1], which is the maximal number that can be used by a logarithmic space machine. The price is that the space used increases from $O(\log n)$ to $O(\log^{3/2} n)$ (for **RL**) or $O(\log^{4/3} n)$ (for **SL**). Can one do better by restricting oneself to machines that use a smaller number of random bits ?

In this paper we try to make progress in answering the following outstanding problem: How many random bits can be fully derandomized for **RL** or **SL** ? Let M be a randomized machine (in **RL** or **SL**) that uses up to k (read once) random bits (e.g. $k = 2^{\sqrt{\log n}}$). For what values of k can one simulate the machine M in **L** (Logspace) ? We say that k bits can be fully derandomized for **RL** (correspondingly, for **SL**) if every such machine that uses up to k random bits can be simulated in **L**. A trivial argument shows that $O(\log n)$ bits can be fully derandomized for **RL** (and hence also for **SL**). The only non-trivial results were proved in [1, 8]. In particular, Nisan and Zuckerman showed that any poly-logarithmic number of random bits can be fully derandomized [8] (see also [2]).

One direction to attack the problem of derandomization is the concept of *pseudo-random generator*, which is also very interesting in its own right. Given a machine M, as above, that uses k random bits, the idea is to use a smaller number of truly random bits in order to generate k so called *pseudo-random bits* $x_1, ..., x_k$ that the machine M confuses to be truly random (up to a small probability of error). We can then use $x_1, ..., x_k$ as the random string for M.

Nisan's original pseudo-random generator for **RL** [6] uses $O(\log^2 n)$ truly random bits in order to generate a polynomial number of output bits $x_1, ..., x_k$ that look random to **every RL** machine. In that sense the sequence $x_1, ..., x_k$ constructed by Nisan's pseudo-random generator is *universal*. Here we suggest that in some cases it may be helpful to construct pseudo-random bits that are **specific for the machine** M (and for it's input I). In our construction, the states of the machine M itself are used by the pseudo-random generator in order to generate the pseudo-random bits for M.

An alternative construction for Nisan's pseudo-random generator for **RL** was given by [5]. The following is a short description of our interpretation for their construction

[*]Department of Applied Mathematics and Computer Science, Weizmann Institute, Rehovot, 76100 Israel. E-mail: ranraz@wisdom.weizmann.ac.il Work supported by an American-Israeli BSF grant 95-00238 and by ESPRIT working group RAND2.

[†]Dept. of Applied Mathematics and Computer Science, Weizmann Institute of Science, Rehovot 76100, Israel. Research supported by a Clore Scholars award and an Eshkol Fellowship of the Israeli Ministry of Science. E-mail: reingold@wisdom.weizmann.ac.il.

[1]The standard definition of **RL** restricts the machine to read each random bit only once.

(the original construction is presented differently and uses expanders rather than extractors): The construction is recursive, and (very briefly) can be described as follows. First use r_0 truly random bits $z_1, ..., z_{r_0}$ to generate (recursively) k_0 pseudo-random bits $x_1, ..., x_{k_0}$. Then use r' additional truly random bits to "recycle" the bits $z_1, ..., z_{r_0}$ (using extractors). More precisely, use the r' new truly random bits to extract from $z_1, ..., z_{r_0}$ additional r_1 random bits, $z'_1, ..., z'_{r_1}$, that are (almost) independent of the state S reached by the machine M after reading the sequence $x_1, ..., x_{k_0}$. Finally, use the r_1 new random bits $z'_1, ..., z'_{r_1}$ to generate (recursively) k_1 additional pseudo-random bits $x_{k_0+1}, ..., x_{k_0+k_1}$. As in Nisan's generator, this construction doesn't depend on the machine M and hence the output is a universal sequence.

The state S reached by the machine M after reading the sequence $x_1, ..., x_{k_0}$ is a random variable with entropy of the order of up to $\log n$. Since the new random bits $z'_1, ..., z'_{r_1}$ are required to be independent of the state S, the entropy of S is lost forever, which means that we have lost an order of $\log n$ random bits. Since this happens at each level of the recursion, the total number of truly random bits needed in order to generate k pseudo-random bits (by this method) is at least $\Omega(\log k \cdot \log n)$.

In this paper we suggest that since the state S is a random variable that has some entropy, it may be possible to "recycle" that entropy as well, and to reuse it later on as random bits. This, however, is not trivial and requires a careful analysis of the dependencies between the random bits and the states reached by the machine. Our main idea gives a specific method to reuse the entropy of the state S. Unfortunately, our method only works in cases where some (very weak) estimation of the probability of the actual state S, reached by the machine, is known. Loosely speaking, that's because otherwise we don't know how many random bits can be extracted out of the state S. If we overestimate that number we create nontrivial dependencies that are hard to analyze. If we underestimate we loose random bits.

Nevertheless, in cases where the probabilities of the states are known our method works well and we are able to generate up to $k \approx 2^{\sqrt{\log n}}$ pseudo-random bits using only a logarithmic number of truly random bits (and only logarithmic space). The machine M doesn't distinguish between these pseudo-random bits and truly random bits with advantage better than $\approx O(2^{-\sqrt{\log n}})$ (i.e., the probability of error is at most $\approx O(2^{-\sqrt{\log n}})$). Note that a logarithmic number of truly random bits can always be fully derandomized. Hence, our main theorem implies that if an estimation of the states probabilities is given or can be easily computed, (as in the case of a stationary random walk on a graph), then up to $\approx 2^{\sqrt{\log n}}$ random bits can be fully derandomized.

We give several interpretations and applications of the main theorem. For example:

1. We show how to derandomize up to $\approx 2^{\sqrt{\log n}}$ random bits for **Symmetric-Logspace** on any regular graph, and more generally on any graph such that the ratio between the degrees of the two special vertices s and t is between $\approx 2^{\sqrt{\log n}}$ and $\approx 2^{-\sqrt{\log n}}$.

2. Let G be a graph with n vertices. We give a **Logspace** Turing machine that uses only $O(\log n)$ random bits and (on input G) outputs a length $k \approx 2^{\sqrt{\log n}}$ "pseudo-random walk" (on G), that looks random to any machine M with space $O(\log k)$ (and to any branching program of size $poly(k)$).

3. Say that we need random variables $x_1, ..., x_k \in \{1, ..., n\}$ with some specific individual distributions x_i and some (consistent) conditional distributions $x_{i+1}|x_i$. We show that for $k \approx 2^{\sqrt{\log n}}$, such $x_1, ..., x_k$ can be constructed (up to error $\approx 2^{-\sqrt{\log n}}$) by a **Logspace** Turing machine that uses only $O(\log n)$ random bits.

2 The Main Theorem

Since the main theorem applies for the non-uniform case as well, we state it in the most general form for branching programs rather than for Turing machines. Since we think of the input as fixed, the only input for the branching program will be the variables $x_1, ..., x_k$ that should be interpreted as a random string. The branching program is layered, and each one of the variables $x_1, ..., x_k$ is read only once, where x_i is read only when entering the i^{th} layer.

Formally, we define an (n, k)-LBP (layered branching program) in the following way.

Definition 2.1 *An (n, k)-LBP is a directed graph with $n \cdot (k+1)$ vertices, partitioned into $k+1$ layers with n vertices in each layer. For $0 \le i \le k$ and $1 \le S \le n$, we denote by (i, S) the S^{th} vertex in the i^{th} layer. For $0 \le i \le k-1$ the out-degree of every vertex (i, S) is exactly 2, where both edges go into vertices in layer $i+1$, and one of these edges is labeled 0 and the other 1.*

Let M be an (n, k)-LBP, and let $\bar{x} = (x_1, ..., x_k)$ be a string of k bits. The state $M_i(\bar{x})$ is defined by induction in the following way.

1. $M_0(\bar{x}) = 1$. That is, we think of the vertex $(0, 1)$ as the starting state.

2. For $i \ge 1$, $M_i(\bar{x})$ is defined to be the unique S such that $((i-1, M_{i-1}(\bar{x})), (i, S))$ is a directed edge labeled by x_i.

(Intuitively, we start from the vertex $(0, 1)$ and branch at the i^{th} layer according to x_{i+1}.)

The state probability $P_i(S)$ is defined by

$$P_i(S) = \mathbf{PROB}_{\bar{x}}[M_i(\bar{x}) = S],$$

where $\bar{x}$ is chosen according to the uniform distribution. Note that for every i, $P_i(S)$ is a probability distribution over $\{1, ..., n\}$. Intuitively, a pseudo-random sequence for the machine M is a sequence of random variables $\bar{x} = (x_1, ..., x_k)$ that are not necessarily uniformly distributed (i.e., not necessarily truly random), but still simulates the probability distributions $P_i(S)$ with a very small error.

Formally, we define an ϵ-error pseudo-random sequence in the following way:

Definition 2.2 *Let M be an (n,k)-LBP. An ϵ-error pseudo-random sequence for M is a sequence of random variables $\bar{x} = (x_1, ..., x_k)$, such that for every i,*

$$\frac{1}{2} \sum_{S} |\mathbf{PROB}_{\bar{x}}[M_i(\bar{x}) = S] - P_i(S)| \le \epsilon.$$

In other words, these two probability distributions of M's state at layer i are of statistical difference at most ϵ (see Section 4.1 for the definition of statistical difference).

As mentioned above, our main theorem applies only for cases where a weak estimation for the state probabilities $P_i(S)$ is available.

Definition 2.3 *Let M be an (n,k)-LBP. A function $A : \{0,..,k\} \times \{1,..,n\} \to R^+$ is a (δ, r)-estimator for M if for every $i \in \{0,..,k\}$ there exists $\mathbf{B}_i \subset \{1,..,n\}$, such that for every i:*

1. *$\sum_{S \in \mathbf{B}_i} P_i(S) \le \delta$.*
2. *For every $S \in \{1,..,n\} \setminus \mathbf{B}_i$,*

$$2^{-r} P_i(S) \le A(i,S) \le 2^r P_i(S).$$

That is, for every i and a state S in the good set $\{1,..,n\} \setminus \mathbf{B}_i$, the value $A(i,S)$ estimates $P_i(S)$ up to a factor of 2^r. For states in the bad set $\mathbf{B}_i$, the estimation $A(i,S)$ can be arbitrary. However, the total probability of states in $\mathbf{B}_i$ is $\le \delta$.

It may be claimed that if the parameters r and δ are both 0 then for every (i,S), $A(i,S) = P_i(S)$, and hence if the estimations $A(i,S)$ are known then a full derandomization is trivial. In most applications, however, r will be of the order of $\approx \sqrt{\log n}$, and hence $A(i,S)$ will not tell the difference between probabilities such as $1/3$ and $2/3$, and even between $\approx 2^{-\sqrt{\log n}}$ and 1. Also, if δ is larger than 0 then $A(i,S)$ can give an arbitrary answer whenever $P_i(S) \in \mathbf{B}_i$, and hence it cannot be used for a full derandomization. Moreover, even if for every (i,S), $A(i,S) = P_i(S)$, the main theorem is still interesting because it gives a construction of pseudo-random sequences for M (which is stronger than derandomization).

In our construction we use extractors which are combinatorial objects that were introduced by Nisan and Zuckerman [8] (a formal definition appears in Section 4.1). For simplicity, we state most of the theorems and proofs of this paper under the assumption that linear-space "optimal extractors" (defined in Definition 4.2) exist. Using the same proofs, one can obtain the corresponding theorems for specific constructions of extractors and for general extractors (we mention these theorems within). We note that the concrete results obtained by currently known extractors are only slightly worse than the ones obtained by optimal extractors. We will now state the main theorem for general parameters and then restate it for the parameters we consider most interesting.

Theorem 2.1 *Let M be an (n,k)-LBP, and let A be an $(\epsilon/(2k^2), r)$-estimator for M. Denote*

$$s = (\log k) \cdot \left[r + \log(k/\epsilon) + \log\log n\right] + \log n.$$

If linear-space optimal extractors (as in Definition 4.2) exist then there exists a randomized Turing machine T such that:

1. *The inputs for T are M and A (as well as the parameters ϵ, r).*
2. *T uses space $O(s)$ and $O(s)$ random bits.*
3. *The output of T is an ϵ-error pseudo-random sequence for M (of length k).*

We find the most interesting parameters for Theorem 2.1 to be:

$$k = 2^{\sqrt{\log n}} \ \ , \ \ r = \sqrt{\log n} \ \ , \ \ \epsilon = 2^{-\sqrt{\log n}}.$$

In this case $s = O(\log n)$, and we can restate the theorem as follows.

Corollary 2.1 *Let $r = \sqrt{\log n}$. Let M be an $(n, 2^r)$-LBP, and let A be a $(2^{-3r-1}, r)$-estimator for M. If linear-space optimal extractors (as in Definition 4.2) exist then there exists an* **RL** *Turing machine T such that:*

1. *The inputs for T are M and A.*
2. *T uses $O(\log n)$ random bits (and $O(\log n)$ space).*
3. *The output of T is a 2^{-r}-error pseudo-random sequence for M.*

Using the extractors of Ta-Shma [13] (rather than optimal ones) it is possible to prove an unconditional version of Corollary 2.1 for $r = \sqrt{\log n}/(poly \log\log n)$ (instead of $r = \sqrt{\log n}$). This is almost as good as the result obtained by optimal extractors and it can be somewhat improved using new constructions of extractors [10].

Theorem 2.2 *There exists a positive constant c_{ext}, such that for $r = \sqrt{\log n}/(\log\log n)^{c_{ext}}$, if M is an $(n, 2^r)$-LBP and A is a $(2^{-3r-1}, r)$-estimator for M then there exists an* **RL** *Turing machine T such that:*

1. *The inputs for T are M and A.*
2. *T uses $O(\log n)$ random bits (and $O(\log n)$ space).*
3. *The output of T is a 2^{-r}-error pseudo-random sequence for M.*

The proof of Theorem 2.2 is essentially the same as the one of Corollary 2.1.

3 Applications

In this section, we give several applications and interpretations of our main theorem. Some of these applications will follow immediately, while others will require a proof. Each application highlights a different feature of the main theorem.

Fix $r = \sqrt{\log n}/(\log\log n)^{c_{ext}}$, where c_{ext} is the constant from Theorem 2.2, and fix $k = 2^r$ and $\epsilon = 2^{-r}$. If linear-space optimal extractors exist (and hence Theorem 2.1 is true without conditions) then the results below can all be proved for the slightly better parameters $k = 2^{\sqrt{\log n}}$ and $\epsilon = 2^{-\sqrt{\log n}}$.

3.1 Random Walks on Graphs

One situation where the probabilities of states are known is the stationary random walk on a graph. In general, to generate such a walk $w_1, ..., w_k$, one needs up to $k \log n$ random bits. Our first application will be a logspace construction of "pseudo-random walks" (of length k), using only $O(\log n)$ random bits. The pseudo-random walk will look, in many ways, the same as a truly random walk.

Let G be an undirected graph with set of vertices $\{1, ..., n\}$ and with D edges. For every vertex S, let $d(S)$ be the degree of S, and denote $\sigma(S) = d(S)/(2D)$. Obviously, $\sigma(S)$ is a probability distribution over the set of vertices $\{1, ..., n\}$.

Formally, the stationary random walk of length k on G is a sequence of random variables $w_1, ..., w_k \in \{1, ..., n\}$, defined in the following way: w_1 is distributed according to σ, that is, for every S, $\mathbf{PROB}[w_1 = S] = \sigma(S)$. For every $1 \le i \le k-1$, w_{i+1} is uniformly distributed on the neighbors of w_i. It is well known (and easy to prove) that for every $1 \le i \le k$, the random variable w_i is distributed according to σ. In that sense the random walk is stationary.

Given G, the random walk $w_1, ..., w_k$ can be easily simulated by an (n, k')-LBP M, where k' is slightly larger than k (say, $k' = k \log n$). The state probabilities for M can be easily computed. Hence, by Theorem 2.1, there exists an **RL** Turing machine T that uses only $O(\log n)$ random bits and outputs an ϵ-error pseudo-random sequence for M. That sequence can now be translated into a sequences of vertices $x_1, ..., x_k \in \{1, ..., n\}$, such that for every $1 \le i \le k-1$, x_{i+1} is a neighbor of x_i. The sequence $x_1, ..., x_k$ is our pseudo-random walk.

First, by Theorem 2.1, we have the following property:

Corollary 3.1 *For every i,*

$$\sum_S |\mathbf{PROB}_{\bar{x}}[x_i = S] - \mathbf{PROB}_{\bar{w}}[w_i = S]| \le 2\epsilon.$$

Moreover, the sequence $x_1, ..., x_k$ looks the same as $w_1, ..., w_k$ in a much stronger sense. We will show that no logarithmic space Turing machine, and more generally, no polynomial size branching program is able to distinguish between the two sequences (note that by definition, the branching program has read-once access to the sequence).

Lemma 3.1 *Let L be a branching program of size $poly(k)$, then $|\mathbf{PROB}_{\bar{x}}[L(\bar{x}) = 0] - \mathbf{PROB}_{\bar{w}}[L(\bar{w}) = 0]| \le \epsilon$.*

To prove this lemma, we use the fact that for our generator we only need estimates of the state-probabilities (rather than exact values).

Proof:(Idea) We consider a machine $\hat{L}$ that combines the machine L and the machine M that produces the random walk on G. Each state of $\hat{L}$ is essentially a pair (S_0, S_1), where S_0 is a vertex of G and S_1 is a state of L. A rather good estimate for the probability of (S_0, S_1) is the stationary probability of S_0. The reason that we can ignore S_1 is that it only contains $\log k = O(\sqrt{\log n})$ bits and that we only go through $k \log n$ states of L. This means that we do not need good estimates of states (S_0, S_1) such that the conditional probability of S_1 given S_0 is much smaller then $\epsilon/poly(k) = 2^{-O(\sqrt{\log n})}$.

Therefore, by Theorem 2.1 we can generate a pseudo-random sequence for $\hat{L}$. Unfortunately, this still does not prove the lemma, as the pseudo-random walk generated by this procedure will not be our original one: The generator of Theorem 2.1 uses the states of $\hat{L}$ and in particular the states of L to construct the pseudo-random sequence (that defines the pseudo-random walk). To prove the lemma, we have to generalize Theorem 2.1, and to show that whenever the generator recycles the state (S_0, S_1), it can ignore S_1 and only use S_0. This is a loss of $O(\sqrt{\log n})$ bits at each level of the recursion, and hence sums up to a loss of only $O(\log n)$ random bits. □

In fact, we can show that the two sequences are indistinguishable for any *n-way $poly(k)$*-size branching program. That is, a program that reads at each level $\log n$ bits (i.e., one vertex of the walk) and branches accordingly to one of $poly(k)$ states at the next level:

Lemma 3.2 *Let L be an n-way branching program of size $poly(k)$, then $|\mathbf{PROB}_{\bar{x}}[L(\bar{x}) = 0] - \mathbf{PROB}_{\bar{w}}[L(\bar{w}) = 0]| \le \epsilon$.*

Note that 2-way $poly(n)$-size branching programs can simulate n-way $poly(n)$-size branching programs (to simulate one transition of the n-way branching program, read $\log n$ bits "into your state" (which multiplies the size of the program by n) and then perform the original transition). Therefore, in Theorem 2.1 there is no need to consider n-way branching programs.

Lemma 3.2, implies the following corollary:

Corollary 3.2 *Let $f : \{1, ..., n\} \to \{1, ..., N\}$ be any function (where N is any number), and let L be a branching program of size $poly(k)$, then*

$$|\mathbf{PROB}_{\bar{x}}[L(f(x_1), ..., f(x_k)) = 0] -$$

$$\mathbf{PROB}_{\bar{w}}[L(f(w_1), ..., f(w_k)) = 0]| \le \epsilon.$$

The last corollary can be applied, for example, in cases where each vertex has a color (and not just a name). The corollary shows that no such L can distinguish between the sequence of colors generated by the pseudo-random walk and the sequence of colors generated by the truly random walk.

3.2 Small Random Spaces

The construction of small random spaces with certain properties is one of the main directions of research in derandomization. In general, we would like to generate m random variables $x_1, ..., x_m$ that satisfy certain properties. The question is how many random bits (and other resources) are needed. Particularly interesting is the case where $x_1, ..., x_m$ can be generated using only $O(\log n)$ random bits. In that case, the constructed random space is of polynomial size and can hence be searched by a polynomial time algorithm. If, in addition, that random space can be constructed by a logspace machine then it can be searched by logspace algorithms.

A famous example is the case of pairwise independence (or k-wise independence). In this case, we want to generate $x_1, ..., x_m$, say in $\{1, ..., n\}$, such that each x_i is uniformly distributed and every two of them are independent. It is known [4] that pairwise independent random variables $x_1, ..., x_m \in \{1, ..., n\}$ can be constructed using only $O(\log n + \log m)$ random bits. This was applied for the derandomization of many randomized algorithms.

Here, we consider the case where the individual distributions of x_i are specified and for every $i > 1$ the conditional distribution of $x_i | x_{i-1}$ is also specified. More precisely, for every i we are given a distribution ρ_i on the set $\{1, ..., n\}$, and for every $i > 1$ and $S \in \{1, ..., n\}$, we are given a distribution $\rho_{i|S}$ on $\{1, ..., n\}$ as well. We need to generate $x_1, ..., x_k \in \{1, ..., n\}$, such that for every i the joint distribution of x_i and x_{i+1} is ϵ-close (in the L_1 norm) to $\rho_{i,i+1}$, where $\rho_{i,i+1}$ is the distribution on the set $\{1, ..., n\} \times \{1, ..., n\}$, defined by $\rho_{i,i+1}(S, S') = \rho_i(S) \cdot \rho_{i+1|S}(S')$. We will say, in this case, that the conditional distributions of $x_1, ..., x_k$ are ϵ-close to $\{\rho_i\}, \{\rho_{i|S}\}$.

Note that the distributions ρ_1 and $\{\rho_{i|S}\}$ define a Markov process. The generated variables $x_1, ..., x_k$ are required to have the same conditional distributions as that Markov process (up to a small error). The main theorem implies that such $x_1, ..., x_k$ can be constructed by a logspace machine that uses only $O(\log n)$ random bits.

Corollary 3.3 *Let ϵ, k be as above. There exists an* **RL** *Turing machine T, that uses only $O(\log n)$ random bits, such that: 1) The inputs for T is the distributions $\{\rho_i\}$ and $\{\rho_{i|S}\}$ (for all i, S). 2) T outputs a sequence $x_1, ..., x_k$, with conditional distributions ϵ-close to $\{\rho_i\}, \{\rho_{i|S}\}$.*

3.3 Derandomization of k Random Bits for SL

Let G be an undirected graph with n vertices and with two special vertices s and t. Denote by $d(s)$ the degree of s in G and by $d(t)$ the degree of t in G. For any two vertices u and v and any integer i denote by $\alpha_{u,i}(v)$ the probability that a random walk of length i, starting from u, gets (at some point) to the vertex v. Assume that we are promised that one of the following two cases occurs.

1. $\alpha_{s,k}(t) > 1/2$.
2. s and t are not connected in G.

How hard is it to determine which of the two cases occurs ?

We prove that if the graph G is regular then the problem can be solved by a deterministic logspace Turing machine. (We interpret this as a derandomization of k random bits for **SL** algorithms on regular graphs). Moreover, our algorithm works for any graph G such that the ratio between $d(s)$ and $d(t)$ is at least ϵ (which means that the probability $\alpha_{t,k}(s)$ is at least $\epsilon \cdot \alpha_{s,k}(t)$) and the algorithm separates between the following two cases: (1) $\alpha_{s,k}(t) > \epsilon$. (2) s and t are not connected in G.

Lemma 3.3 *There is a deterministic logspace Turing machine T that gets as input a graph G with two special vertices s, t, and such that the output $T(G)$ satisfies the following:*

1. *If s and t are not connected in G then $T(G) = 0$.*
2. *If $\alpha_{s,k}(t) > \epsilon$ and $\epsilon < d(s)/d(t)$ then $T(G) = 1$.*

The proof of this corollary is based on ideas from [3]. First let us introduce a few more notation: Let v be any vertex, let W be any subset of vertices. Then

- Denote by $\Gamma(v)$ the set of v's neighbors, and by $\Gamma(W)$ the set of W's neighbors: $\left(\bigcup_{w \in W} \Gamma(w)\right) \setminus W$.
- For any $w, u \in W$ and any integer i denote by $\alpha^W_{w,i}(u)$ the probability that a random walk that starts at w reaches u in at most i steps and that until the walk reaches u it stays in W.
- For any $w, u, v \in W$ and any integer i denote by $\alpha^W_{w,i}(u, v)$ the probability that a random walk that starts at w reaches both u and v in at most i steps and that until the walk reaches both vertices it stays in W.

Proof: We start by showing a property of graphs that will help us define the machine T:

Lemma 3.4 *Let G be a graph and let s and t be any two vertices such that $\alpha_{s,k}(t) > \epsilon$ and $\epsilon < d(s)/d(t)$. Then there exists a subset of vertices W such that for any vertex $w \in W$*

$$\alpha^W_{w,2k}(s, t) = \Omega(\epsilon^4/k^2).$$

Given the lemma we describe a machine T that outputs 1 if it finds a path from s to t and outputs 0 otherwise. By definition, if s and t are not connected then $T(G) = 0$. We therefore assume that s and t are connected and that $\alpha_{s,k}(t) > \epsilon$ and $\epsilon < d(s)/d(t)$. We show that in this case, T indeed finds a path from s to t.

Let W be as in Lemma 3.4 and define the subgraph, H, of G as follows: The vertices of H are $W \cup \Gamma(W)$ and its edges are all the edges of G that contain at least one vertex in W. Consider a stationary random walk in H. By definition, the weight of W under the stationary distribution is at least $1/2$. Therefore, the random walk has probability at least $1/2$ to start at a vertex $w \in W$. By Lemma 3.4, we get that the random walk in H has probability at least $1/2 \cdot \alpha^W_{w,2k}(s,t) = \Omega(\epsilon^2/k^4) = 2^{-O(r)}$ to start at W and to reach s and t in at most $2k$ steps before leaving W. Using Lemma 3.2 it can be shown that there exists an **RL** Turing machine $\tilde{T}$ that uses only $O(\log n)$ random bits and outputs a pseudo random walk in H that has *non-zero probability* to start at W and to reach both s and t before leaving W.

Assume now that T gets the set W as input and can therefore compute the subgraph H. In such a case, T can enumerate over all possible random strings of $\tilde{T}$ and see if at least one of the corresponding walks reaches both s and t. Unfortunately, T is *not given* the set W. It turns out however that $\tilde{T}$ only needs to know *the number of edges D_H in H* in order to produce a walk that has non-zero probability to reach both s and t. Therefore T can still find the required path by enumerating over all possible random strings of $\tilde{T}$ *and all possible values of D_H*. We describe this idea in some more details:

Assume that $\tilde{T}$ gets as input the graph G and the number of edges D_H in H and tries to produce a pseudo-random walk in H. We also assume for now that $\tilde{T}$ gets the initial vertex v_0 (sampled according to the stationary distribution in H). Now $\tilde{T}$ simultaneously produces a pseudo-random sequence and the corresponding walk in H. By looking into the proof of Lemma 3.2, we observe that the only additional information $\tilde{T}$ needs is the set of neighbors of v_i in H for every vertex v_i reached by the walk. This set is needed for two reasons: (1) To compute the stationary probability of v_i in H (which is $d_H(v)/D_H$). This probability is needed to evaluate the rest of the pseudo-random sequence. (2) To choose the next vertex v_{i+1} of the walk. However, since we are only interested in walks until they leave W, $\tilde{T}$ can assume for each v_i that all its neighbors are in H (in case this assumption is wrong, it will only affect the rest of the walk (which is not interesting)). The only problem that still remains is how to choose the first vertex v_i. However, if $\tilde{T}$ has non-zero probability to produce a good walk when v_i is chosen under some distribution, then there exists at least one vertex w such that $\tilde{T}$ has non-zero probability to produce a good walk when v_i is always set to w. This vertex w can be found by enumerating over all the vertices (of G) as well.

It remains to prove Lemma 3.4:

Proof: It is well known that $\alpha_{t,k}(s) = \alpha_{s,k}(t) \cdot (d(s)/d(t))$ (to verify this fact consider any path between s and t and compare the probability of taking this path from s to t and the probability of taking this path from t to s). Therefore we have that $\alpha_{t,k}(s) > \epsilon^2$. From now on our only assumption on s and t would be that $\alpha_{t,k}(s), \alpha_{s,k}(t) > \epsilon^2$.

The set W is defined as the union $W = W_s \cup W_t$. We now define W_t. The definition of W_s is symmetric (i.e., it is obtained by replacing the roles of s and t). The set W_t is defined as the union: $W_t \stackrel{\text{def}}{=} \bigcup_{i=0}^{k} W_t^i$ where

$$W_t^i \stackrel{\text{def}}{=} \{w \mid \alpha_{w,k-i}(t) \geq \epsilon^2 \cdot (1-1/k)^i\}$$

Observe that $s \in W_t^0 \subset W_t$ and that $W_t^k = \{t\}$. Let $\epsilon_i \stackrel{\text{def}}{=} \epsilon^2 \cdot (1-1/k)^i$. Let w be any vertex in W_t^i. Let q_1 be the probability that a random walk from w leaves W_t before it reaches t. Let q_2 be the probability that a random walk from w reaches t in at most $k-i$ steps given that the walk leaves W_t before it reaches t. It is not hard to show that $q_2 \geq \epsilon_i \cdot (1-1/k)$ and therefore that

$$\epsilon_i \leq \alpha_{w,k-i}(t) = \alpha^{W_t}_{w,k-i}(t) + q_1 \cdot q_2 < \alpha^{W_t}_{w,k-i}(t) + 1 \cdot \epsilon_i \cdot (1-1/k)$$

which implies that $\alpha^{W_t}_{w,k}(t) \geq \alpha^{W_t}_{w,k-i}(t) > \epsilon_i/k = \Omega(\epsilon^2/k)$. To conclude, we got that for any vertex $w \in W_t$, $\alpha^{W_t}_{w,k}(t) = \Omega(\epsilon^2/k)$. By symmetry we get that for any vertex $w \in W_s$, $\alpha^{W_s}_{w,k}(s) = \Omega(\epsilon^2/k)$. By the definition of W it is easy to show that for any vertex $w \in W$ $\alpha^W_{w,2k}(s,t) = \Omega(\epsilon^4/k^2)$.
□

□

3.4 Other Applications

We have several other applications and interpretations of the main theorem. Let us give two more examples.
Strongly connected graphs:
Let G be a graph with n vertices, and assume for simplicity that G is regular. We say that G is *strongly connected* if it has the following property: For any vertex i, if we start a random walk of length k from i then the distribution of the last vertex is very close to the stationary distribution (say, that the L_1 distance is smaller than ϵ). Strongly connected graphs are very common (e.g., expanders, the cube $\{0,1\}^n$ and many other graphs).

Our main theorem implies several algorithms for strongly connected graphs. For example, if we know that G is either strongly connected or a union of two strongly connected components of roughly the same number of vertices, we are able to distinguish between the two cases by a **Logspace** algorithm.
RL machines that use Markov process random bits:
Let M be an arbitrary **RL** machine (or branching program), and assume that M works well only if its random string is given by some specific stationary Markov process $x_1, ..., x_{poly(n)} \in \{1, ..., n^{\log n}\}$ (and assume that the stationary distribution for that Markov process is easy to compute, e.g., the uniform distribution). We show that a pseudo-random sequence for M can be constructed using $O(\log^2 n)$ space and $O(\log^2 n)$ random bits (note that the usual results of pseudo-random generators for **RL** do not apply for this case).

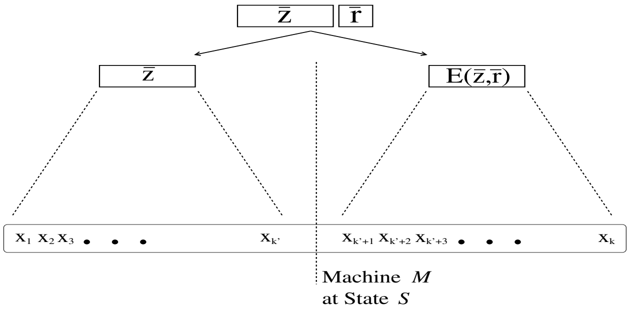

Figure 1: The INW-generator using an extractor E

4 Main Idea and Intuitions

In this section, we present the main idea behind the proof of Theorem 2.1: recycling the states reached in space-bounded computations and reusing their entropy may help to derandomize such computations. This leads to a construction of pseudo-random sequences for space-bounded machines that uses the number of truly random bits claimed in Theorem 2.1. However, computing these sequences requires too much space. For the full proof of Theorem 2.1 we will need a much more complicated construction that uses much less space. To demonstrate our main idea and to give the main intuitions, we first describe in this section the simpler construction. The full construction is given in Section 5 (along with a sketch for the proof of Theorem 2.1).

As mentioned in the introduction, our starting point is the pseudo-random generator of Impagliazzo, Nisan and Wigderson [5]. To get our construction we introduce a slight modification of the INW-generator using extractors instead of expanders. Let M be the logspace machine (or the branching program) that we want to "fool". Let $\bar{z}, \bar{r}$ be the truly random input for the INW-generator. The output of the generator $\bar{x} = x_1, \ldots, x_k$ is roughly defined as follows (see Figure 1 for an illustration): Use the first part of the input, i.e., $\bar{z}$, to recursively generate the first k' bits, $x_1, \ldots, x_{k'}$, of the pseudo-random output. Let S be the state of M after reading these k' bits (i.e., $S = M_{k'}(\bar{x})$). Use the second part of the input, i.e., $\bar{r}$, to extract from $\bar{z}$ a new string $\bar{z}' = E(\bar{z}, \bar{r})$ such that (with high probability) the conditional distribution of $\bar{z}'$ given S is almost uniform. Finally, recursively apply the generator on $\bar{z}'$ to obtain the rest of the pseudo-random sequence, $x_{k'+1}, \ldots, x_k$.

The function E used to define $\bar{z}'$ is called an extractor. This object, introduced by Nisan and Zuckerman [8], enables to extract an almost uniformly distributed string of length m from *an arbitrary* random variable that has sufficiently large min-entropy (i.e., at least m). To do so, E also uses a few (much less than m) additional truly random bits. In our setting, the "defected source" is $\bar{z}$. Given the state S, the string $\bar{z}$ is not uniformly distributed (since S depends on $\bar{z}$). Nevertheless, unless S is very rare, $\bar{z}$ still has sufficiently large min-entropy. To extract this entropy E uses the string $\bar{r}$ which is uniformly distributed and independent of $\bar{z}$ and S. Since extractors are the main technical tool of our construction we start by defining these objects more formally and considering some of the relevant constructions and properties:

4.1 Extractors

Let X and Y be random variables over a set D. The **min-entropy** of X is defined to be[2] $H_\infty(X) \stackrel{\text{def}}{=} -\log(\max_{a\in D} \Pr[X = a])$. The statistical difference between X and Y is defined by $\Delta(X,Y) \stackrel{\text{def}}{=} \frac{1}{2}\sum_{a\in D} |\Pr[X = a] - \Pr[Y = a]|$.

For any integer n, denote by I^n the set of all n-bit strings, $\{0,1\}^n$. Denote by U_n the uniform distribution over I^n. For simplicity, we also denote by U_n a random variable uniformly distributed over I^n (we identify random variables and their distributions in a few additional places in this paper). We use the definition of extractors given in [13] (which is a variant of the original definition given in [8]).

Definition 4.1 *(Extractors) A function $E : I^\ell \times I^t \mapsto I^{m'}$ is an (m,ϵ)-*extractor *if for any random variable Z over I^ℓ with min-entropy at least m, $\Delta(E(Z,U_t), U_{m'}) \le \epsilon$. That is, the distribution of $E(Z,U_t)$ is of statistical difference at most ϵ from $U_{m'}$.*

A primary criteria in determining the quality of an extractor is the number of truly random bits it uses. That is, how large is t as a function of ℓ, m, m' and ϵ. Nisan and Zuckerman [8] proved a lower bound (later adapted in [13] to the definition of extractors used here) of $t = \Omega(\log 1/\epsilon + \log(\ell - m))$ even when extracting a single bit (i.e., for $m' = t+1$). It can also be shown that (up to a constant factor) this lower bound can non-constructively be matched: For any $\ell \ge m$ and ϵ there exists an (m,ϵ)-extractor $E : I^\ell \times I^t \mapsto I^{m'}$, where $t = O(\log 1/\epsilon + \log(\ell - m))$. We call such extractors that use $O(\log 1/\epsilon + \log \ell)$ truly random bits and extract the entire m bits of entropy **optimal extractors**.

Usually, however, one would be interested in **explicit extractors**, that is, a *family* of extractors[3] computable in polynomial time (in the input length $n + t$). Although there is no proven explicit construction of optimal extractors (for the full range of parameters) there has been substantial progress in this direction. In fact, for limited range of parameters explicit optimal extractors do exist (e.g., [14, 15]).

For the sake of our construction, extractors that are *not explicit* might still be interesting. However it is essential that the extractor can be *evaluated in linear space*. Using linear-space extractors ensures that the pseudo-random sequence we construct against space s machines could also be

[2] by log we denote the logarithm base 2

[3] Most applications require an (m,ϵ)-extractor $E : I^\ell \times I^t \mapsto I^{m'}$ *for a range of possible parameters*, $\langle \ell, m, m', \epsilon\rangle$. That is, a family of extractors parameterized by $\langle \ell, m, m', \epsilon\rangle$. In fact, whenever an extractor will be considered in this paper (as in most other discussions on extractors) it will implicitly refer to such a family (which also justifies the use of asymptotic-notation).

generated in space s (therefore, the derandomization does not increase the space). Most explicit extractors that have been constructed so far are indeed linear space (in fact most of these extractors are computable in polylogarithmic space).

Definition 4.2 *(Linear Space Optimal Extractors) An (m, ϵ)-extractor $E : I^\ell \times I^t \mapsto I^m$ (i.e., a family of extractors $\{E_{\ell,m,\epsilon} : I^\ell \times I^t \mapsto I^m\}_{\ell,m,\epsilon}$) is a* linear-space optimal extractor *if: (1) $t = t(\ell, m, \epsilon) = O(\log 1/\epsilon + \log \ell)$ and (2) E is computable in space $O(n + t)$.*

For simplicity, we describe our construction under the assumption that linear space optimal extractors do exist. The construction can easily be extended to general extractors (though the statement of such a general construction would be somewhat cumbersome). As indicated by Theorem 2.2, the extractors of Ta-Shma [13] are almost as good as optimal extractors for our needs. These extractors use $polylog(\ell) \cdot \log(1/\epsilon)$ truly random bits and extract the entire m bits of entropy. Furthermore, the extractors can be evaluated in linear space[4]. We note that extractors that use even fewer random bits were recently constructed in [10]. To conclude, Theorem 2.2 can be proven using the following theorem:

Theorem 4.1 *(implicit in [13]) For every constant $\gamma < 1$, every $\epsilon \geq 2^{-\ell^\gamma}$, and every $m = m(\ell)$ there is an (m, ϵ)-extractor $E : I^\ell \times I^t \mapsto I^m$, where $t = O(polylog(\ell) \cdot \log(1/\epsilon))$. Furthermore E can be evaluated in $O(\ell)$ space.*

4.2 How to Recycle the State

Consider any intermediate stage of the INW-construction. The machine M at time i is at some state S_0, and it has strings $\bar{z}, \bar{r}$ that are almost uniformly distributed given the state S_0. The strings $\bar{z}, \bar{r}$ are used to generate the next portion $x_{i+1}, \ldots, x_{i'}$ of the pseudo-random sequence. The first step in doing so is to recursively generate $x_{i+1}, \ldots, x_j$ from $\bar{z}$. Let S be the state reached by M from S_0 after reading the sequence $x_{i+1}, \ldots, x_j$ (i.e., the state of M at time j). Now, a new string $\bar{z}' = E(\bar{z}, \bar{r})$ is used to (recursively) generate $x_{j+1}, \ldots, x_{i'}$. For the construction to work, $\bar{z}'$ should be almost uniformly distributed given the state S (ensuring that $x_{j+1}, \ldots, x_{i'}$ is almost independent of S). However, since S depends on $\bar{z}$, the entropy of $\bar{z}$ conditioned on S may decrease by roughly $-\log P_j(S)$ which is an order of $\log n$ bits even if S is not too rare. Therefore, $\bar{z}'$ is shorter by $\Omega(\log n)$ bits than $\bar{z}$. Avoiding this loss of entropy due to the state S is exactly the goal of our construction. Note that, since this loss of $\log n$ bits occurs at every level of the recursion, the INW-Construction requires at least $\Omega(\log k \cdot \log n)$ truly random bits in order to generate k pseudo-random bits.

[4] In fact, in [13] the space requirements of the extractor are not analyzed (since the important claim there is that the extractor is explicit). However, it can be shown that the extractor can indeed be evaluated in linear-space (though it is not clear if it can be evaluated simultaneously in polynomial time and linear space).

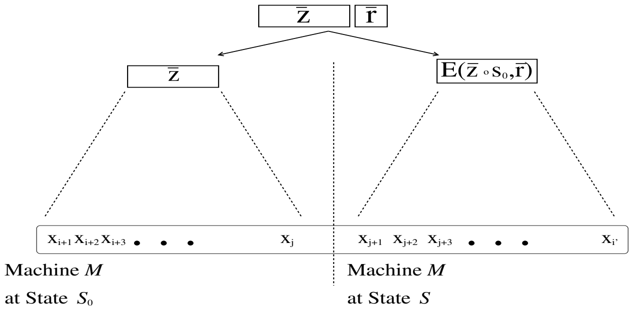

Figure 2: Recycling the state S_0 when M is at state S.

Our main idea is that even though the entropy of a state is lost at some intermediate stage of the INW-construction it can be recovered at a later (or more accurately, at an inner) stage: Consider the intermediate stage which is described in the previous paragraph (also see Figure 2 for an illustration). When recycling the randomness of $\bar{z}$ in order to generate the sequence $x_{j+1}, \ldots, x_{i'}$, the entropy of the state S is indeed lost. However, at this point *the entropy of S_0 can be recovered*. Note that S_0 is a random variable that has some entropy and although $x_{j+1}, \ldots, x_{i'}$ should be independent of the current state S *it may depend on* S_0. Therefore, more bits can be extracted by defining $\bar{z}' = E(\bar{z} \circ S_0, \bar{r})$ (rather than $\bar{z}' = E(\bar{z}, \bar{r})$) (where the concatenation of any two bit strings $\bar{z}$ and $\bar{y}$ is denoted by $\bar{z} \circ \bar{y}$).

4.3 Some More Details

Our construction shows that (in the setting described above) if $\bar{z}$ has ℓ bits then roughly $\ell - \log P_i(S_0) + \log P_j(S)$ bits can be extracted from $\bar{z} \circ S_0$. Therefore, in order to exhaust the entropy in $\bar{z} \circ S_0$, the construction requires some estimation of the states' probabilities. If we underestimate $P_j(S)$ or overestimate $P_i(S_0)$ we might extract from $\bar{z} \circ S_0$ too many bits creating nontrivial dependencies (of $\bar{z}'$ in S). On the other hand, if we overestimate $P_j(S)$ or underestimate $P_i(S_0)$ we loose random bits. However, only a very weak estimation of these probabilities is needed: If $P_i(S_0)$ and $P_j(S)$ are estimated up to a multiplicative factor of 2^r we only loose $O(r)$ bits in every level of the recursion (since $\ell - \log P_i(S_0) + \log P_j(S) - O(r)$ bits can be safely extracted from $\bar{z} \circ S_0$). Moreover, we *only need to estimate the probabilities of states that are actually reached*. Therefore, the estimates of small probabilities can be arbitrarily wrong.

The natural question at this point is whether the method described above completely recycles the entropy of states. In other words, whether the entropy loss in the INW-Construction due to the entropy of states is completely eliminated. Our analysis gives a positive answer to this question: At each one of the $\log k$ levels of our construction there are only two kinds of entropy losses: (1) The true randomness used by the extractor (i.e., $\bar{r}$ in the description above). When using optimal extractors, this amounts to $O(\log(k/\epsilon) + \log \log n)$ bits. (2) $O(r)$ bits from underestimating the entropy in $\bar{z} \circ S_0$ (to compensate for of the estimator's inaccuracies). Altogether, the number of bits needed for this construction (using optimal extractors) is of order

$$(\log k) \cdot \left[r + \log(k/\epsilon) + \log\log n\right] + \log n,$$

as claimed in Theorem 2.1.

To see how the recycling method described above can save such a significant amount of randomness let us first consider a simplified scenario: Assume that for every state $\tilde{S}$ and every time t the probability $P_t(\tilde{S})$ is exactly $1/n$. In such a case, the amount of randomness that can be extracted from $\bar{z} \circ S_0$ (as above) is exactly ℓ (the length of $\bar{z}$). That is, the entropy in S_0 completely compensates for the entropy loss due to S. However, given an *exact* estimator to the probabilities $P_t(\tilde{S})$ every machine M can easily be transformed into an "equivalent" machine M' that (almost) has this special property. Assume for simplicity that for every state $\tilde{S}$ and every time t the probability $P_t(\tilde{S})$ is at least $1/n^2$ (smaller probabilities can be ignored). M' simulates M and at time t if M enters a state $\tilde{S}$ then the machine M' (after reading $P_t(\tilde{S})$) enters into a state $\langle\tilde{S}, \bar{v}\rangle$ where $\bar{v}$ is a random string of $\lceil 2\log n + \log P_t(\tilde{S})\rceil$ bits. It is easy to verify that the probability of every such state $\langle\tilde{S}, \bar{v}\rangle$ is $\Theta(1/n^2)$. To conclude, using exact estimates of states' probabilities the entropy of states can be completely recycled. As mentioned above, having estimates that are only accurate up to a multiplicative factor of 2^r implies an $O(r)$ loss of entropy in each level.

5 The Full Construction

In this section we describe the full construction of pseudo-random sequences for layered branching programs (and logspace machines) that satisfies Theorem 2.1. This construction is essentially very similar to the one described in Section 4. However, computing the pseudo-random sequences for the construction of Section 4 requires too much space. We therefore introduce in this section a few additional ideas in order to reduce the space requirements of the construction. We also introduce a new combinatorial object which we call a *condenser* that might have additional applications elsewhere.

5.1 Reducing the Space

Consider any intermediate node in the recursion tree of the construction that is described in Section 4. At any such node, the generator applies the extractor E on a string $\bar{z}, S_0$, where S_0 is the state of the machine at some previous time. In order to do so, the generator has to remember $\bar{z}, S_0$, and in particular the state S_0. Since the depth of the recursion is $\Omega(\log k)$, the generator has to remember $\Omega(\log k)$ states, and hence it uses a space of $\Omega(\log k \cdot \log n)$. In this section we show a way to achieve the same results using much less space.

The main idea for reducing the space is that the generator *does not really need to apply E on the string $\bar{z}, S_0$ itself.* Instead, the generator can apply E on any function $F(\bar{z}, S_0)$ as long as it contains more or less the same information as $\bar{z}, S_0$ (that is, it has more or less the same entropy). Say that we need to remember $\bar{z}, S_0$ in some node in the recursion tree. We will allow the two "children" of that node to work on the same space where $\bar{z}, S_0$ is written. However, when they finish their job they must return in that space a function $F(\bar{z}, S_0)$ as above. That is, what is written in that space after they finish their job must be *a function of what was there before*, and it must *contain more or less the same information* (if it is a function of other bits as well, we will get hard to analyze dependencies). If $F(\bar{z}, S_0)$ contains less information than $\bar{z}, S_0$ we loose entropy.

To make things easier, we will allow $F(\bar{z}, S_0)$ to contain a little bit more bits than the original string $\bar{z}, S_0$ (hence it will be easier to guarantee that it contains more or less the same information). Furthermore, since the children have to return the current state, S_1, of the machine as well (which in itself is a function of $\bar{z}, S_0$) the actual requirement is that $F(\bar{z}, S_0)$ *together with* S_1 contain the same information as $\bar{z}, S_0$.

To describe the (recursive) way such a function F can be defined, let us consider any node in the recursion tree. The node has as its input the string $\bar{z}, \bar{r}, S_0$ and it calls the left-hand child with the string $\bar{z}, S_0$ as input. After constructing the next portion of the pseudo-random sequence (which takes the machine to state S_1), the child returns the string $S_1, F(\bar{z}, S_0)$. The node then computes $\bar{z}' = E(F(\bar{z}, S_0), \bar{r})$ and calls the right-hand child with the string $\bar{z}', S_1$ as input. After finishing its job, the child returns $S_2, F'(\bar{z}', S_1)$. We know by the requirements that $S_1, F(\bar{z}, S_0)$ contains more or less the same information as $\bar{z}, S_0$, and that $S_2, F'(\bar{z}', S_1)$ contains more or less the same information as $\bar{z}', S_1$. But we have to ensure that our node will also return to **its parent** more or less the same information as $\bar{z}, \bar{r}, S_0$. This seems to be a problem because $\bar{z}', S_1 = E(F(\bar{z}, S_0), \bar{r}), S_1$ contains less information than $F(\bar{z}, S_0), \bar{r}, S_1$ and hence less information than $\bar{z}, \bar{r}, S_0$ (see [9] for tight lower bounds on the entropy loss of extractors). It turns out however that it is not hard to produce an additional buffer (of a relatively small size) that together with $E(F(\bar{z}, S_0), \bar{r})$ will contain more or less the same information as $F(\bar{z}, S_0), \bar{r}$. Hence, the additional buffer together with $E(F(\bar{z}, S_0), \bar{r}), S_1$ will contain more or less the same information as $F(\bar{z}, S_0), \bar{r}, S_1$ and hence more or less the same information as $\bar{z}, \bar{r}, S_0$. This buffer is produced by the so called condenser. Since the depth of the recursion is $O(\log k)$, the generator will have to remember $O(\log k)$ such buffers. Nevertheless, since the buffer will be relatively short this will not be a problem. We will now define extractor-condenser pairs and shortly discuss their constructions.

5.2 Condensers

As described above, each time a node of the recursion tree applies an extractor it also produces an additional buffer using a condenser. Since the condenser and the extractor are applied at the same time, we can analyze their operation together: The extractor extracts the pure random bits from a string $\bar{z}$ and the condenser produces an additional buffer that makes sure we "do not loose much entropy" (e.g. the entropy of the random string $\bar{r}$ used by the extractor). There are a few ways to formalize the phrase "do not loose much entropy". The way we use here is not the most general one but it seems to be the most convenient for our needs: we simply define the extractor-condenser pair to be 1-to-1 on "most of its inputs". Assume that the min-entropy of a source Z is at least m. We are interested in an (m,ϵ)-extractor $E : I^\ell \times I^t \mapsto I^{m'}$ and a condenser $C : I^\ell \times I^t \mapsto I^b$ such that the combined function $\langle E, C\rangle$ is 1-to-1 on all but an ϵ fraction of its inputs when the distribution of these inputs is Z, U_t. However, such a definition would require the buffer-size b to be very large in the case that for many strings $\bar{z}$ the probability $\Pr[Z = \bar{z}]$ is non-zero but very small Therefore, in our definition we exclude such strings (where "very small" is determined by the slackness parameter a). This will not set a problem in the definition of the generator, since there the set of "bad strings" will only have a small probability.

For any random variable Z define its support $\mathbf{SUP}(Z)$ to be the set of strings with non-zero probability:

$$\mathbf{SUP}(Z) \stackrel{\text{def}}{=} \{z \in I^\ell \mid \Pr[Z = z] > 0\}$$

Definition 5.1 *(Extractor-Condenser Pairs) A pair of functions* $\langle E, C\rangle : I^\ell \times I^t \mapsto I^{m'} \times I^b$ *is an* (m, ϵ, a)*-ECP if* E *is an* (m,ϵ)*-extractor and for any random variable* Z *over* I^ℓ *if*

$$\forall z \in \mathbf{SUP}(Z) \;\; 2^{-(m+a)} \leq \Pr[Z = z] \leq 2^{-m}$$

then there exists a set of inputs $\mathbf{B}_Z \subseteq I^\ell \times I^t$ *such that the following holds:*

1. $\Pr[\langle Z, U_t\rangle \in \mathbf{B}_Z] \leq \epsilon$
2. $E \times C$ *is 1-to-1 on* $\left(I^\ell \times I^t\right) \setminus \mathbf{B}_Z$.

It turns out that given any extractor that extracts all the min-entropy of the source, it is very easy to define an extractor-condenser pair (ECP) with comparable parameters:

Theorem 5.1 *Let* $E' : I^\ell \times I^{t'} \mapsto I^m$ *be any* $(m, \epsilon/3)$*-extractor and let* a *be any positive integer. Then there exist two integers* $t, b = O(t' + a)$ *and a function* $C : I^\ell \times I^t \mapsto I^b$ *that is computable in linear-space (and polynomial time) such that* $\langle E, C\rangle$ *is an* (m, ϵ, a)*-ECP where* $E : I^\ell \times I^t \mapsto I^m$ *is defined by* $\forall \bar{z} \in I^\ell, \bar{r} \in I^{t'}, \bar{h} \in I^{t-t'}, \quad E(\bar{z}, \bar{r} \circ \bar{h}) = E'(\bar{z}, \bar{r})$.

The definition of the condenser C that satisfies Theorem 5.1 is very simple: $C(\bar{z}, \bar{r} \circ \bar{h}) \stackrel{\text{def}}{=} h(\bar{z}, \bar{r}) \circ \bar{h}$ where h is an almost 2-universal function represented by $\bar{h}$. Almost 2-universal functions (as well as 2-universal functions and other related notions) were introduced by Carter and Wegman [4]:

Definition 5.2 *Let* ℓ *and* m *be any two positive integers and let* δ *be any positive constant. A distribution* H *of functions from* I^ℓ *to* I^m *is* δ*-almost* 2*-universal if for any* $x \neq y \in I^\ell$,* $\Pr_{h \in H}[h(x) = h(y)] \leq \delta$

Acknowledgments

We thank Moni Naor Amnon Ta-Shma and Avi Wigderson for many helpful discussions.

References

[1] M. Ajtai, J. Komlos and E. Szemeredi, Deterministic simulations of Logspace, *Proc. 19th Ann. ACM Symp. on Theory of Computing*, pp. 132-140, 1987.

[2] R. Armoni, On the de-randomization of space-bounded computations, Random'98, 1998.

[3] R. Armoni, A. Ta-Shma, A. Wigderson and S. Zhou, $SL \subseteq L^{\frac{4}{3}}$, *Proc. 29th Ann. ACM Symp. on Theory of Computing*, 1997, pp. 230-239.

[4] L. Carter and M. Wegman, Universal hash functions, *J. of Computer and System Sciences*, vol. 18, 1979, pp. 143-154.

[5] R. Impagliazzo, N. Nisan and A. Wigderson, Pseudorandomness for network algorithms, *Proc. 26th Ann. ACM Symp. on Theory of Computing*, 1994, pp. 356–364.

[6] N. Nisan, Pseudorandom generators for space-bounded computation, *Combinatorica*, vol. 12(4), 1992, pp. 449-461.

[7] N. Nisan, E. Szemeredi and A. Wigderson, Undirected connectivity in $O(\log^{1.5} n)$ space, *Proc. 33rd IEEE Symp. on Foundations of Computer Science*, 1992, pp. 24–29.

[8] N. Nisan and D. Zuckerman, Randomness is linear in space, *J. Comput. System Sci.*, vol. 52, 1996, pp. 43–52. Preliminary version: More deterministic simulations in logspace *Proc. 25th Ann. ACM Symp. on Theory of Computing*, 1993, pp. 235-244.

[9] J. Radhakrishnan and A. Ta-Shma, Tight bounds for depth-two superconcentrators, *Proc. 38rd IEEE Symp. on Foundations of Computer Science*, 1997, pp. 585-594.

[10] R. Raz, O. Reingold and S. Vadhan, Extracting all the Randomness and Reducing the Error in Trevisan's Extractors, *Proc. 31st Ann. ACM Symp. on Theory of Computing*, 1999 (this proceeding).

[11] M. Saks and S. Zhou, RSPACE(S) $\subseteq$ DSPACE($S^{3/2}$), *Proc. 36th IEEE Symp. on Foundations of Computer Science*, 1995, pp. 23-25.

[12] W. J. Savitch, Relationship between nondeterministic and deterministic tape complexities, *J. Comput. System Sci.*, vol. 4(2), 1970, pp. 177–192.

[13] A. Ta-Shma, On extracting randomness from weak random sources, *Proc. 28th Ann. ACM Symp. on Theory of Computing*, pp. 276-285, 1996.

[14] L. Trevisan, Constructions of near-optimal extractors using pseudo-random generators, To appear in: *Proc. 31th Ann. ACM Symp. on Theory of Computing*, 1999.

[15] D. Zuckerman, Randomness-optimal oblivious sampling, *Random Structures & Algorithms*, vol. 11(4), pp. 345-367,1997.