An $O(\log(n)^{4/3})$ Space Algorithm for (s,t) Connectivity in Undirected Graphs

Roy Armoni* Amnon Ta-Shma† Avi Wigderson‡ Shiyu Zhou§

July 30, 1999

Abstract

We present a deterministic algorithm that computes *st*-connectivity in undirected graphs using $O(\log^{4/3} n)$ space. This improves the previous $O(\log^{3/2} n)$ bound of Nisan, Szemerédi and Wigderson [NSW92].

1 Introduction

Undirected *st*-connectivity (*USTCON*) is a fundamental computational problem, and algorithms for it serve as basic subroutines for more complex graph problems. It is complete for the class SL of symmetric nondeterministic log-space computations [LP82], and is a subproblem of directed *st*-connectivity, which captures the class NL of general nondeterministic computation. The combinatorics of *USTCON*, as well as its time complexity, are extremely well understood. However, its space complexity is still a mystery, which was a source of some beautiful discoveries in complexity theory. We adopt NC-style notations and let $L^\alpha = DSPACE((\log n)^\alpha)$.

Savitch's result [Sav70] from 1970, $NL \subseteq L^2$ implies a deterministic $(\log n)^2$ space bound for SL directly. Remarkably, since then, all progress went via probabilistic algorithms for *USTCON* and their derandomization.

*Institute of Computer Science, The Hebrew University of Jerusalem, Israel. E-mail: *aroy@cs.huji.ac.il*

†ICSI, Berkeley. E-mail: *amnon@icsi.berkeley.edu*

‡Institute of Computer Science, The Hebrew University of Jerusalem, Israel. E-mail: *avi@cs.huji.ac.il.*
This research was supported by grant number 69/96 of the Isael Science Foundation, founded by the Israel Academy for Sciences and Humanities

§Bell Laboratories, Murray Hill, New Jersey, USA. The work was mainly done while visiting the Institute of Computer Science, The Hebrew University of Jerusalem, Israel. E-mail: *shiyu@bell-labs.com*

In the late 70's Cook suggested universal traversal sequences (UTS) as a basis for log-space algorithms for $USTCON$. A traversal sequence is a (deterministic) instruction sequence for a pebble moving on the vertices of a graph, in much the same way as the random coin provides such instructions in a random walk. Such a sequence is universal if it eventually leads the pebble to visit all nodes in every connected graph (of a given size).

Aleliunas et al [AKL+79] proved not only the *existence* of such a UTS of polynomial length, but did it via the probabilistic method, giving in particular a probabilistic log-space (RL) algorithm for $USTCON$. Thus they established $SL \subseteq RL$. Unfortunately, it did not provide deterministic space-efficient algorithms to generate such short UTS.

In a seminal paper, Nisan [Nis92] proved that a UTS can be constructed in L^2. This construction was based on a pseudo-random generator that fools RL machines. In particular, it can be used to derandomize the above probabilistic algorithm. This hierarchical generator requires $\log n$ universal hash functions of $O(\log n)$ bits each. While not directly improving the deterministic space bound for $USTCON$, Nisan's techniques were the basis of all subsequent progress, starting with his own paper [Nis94] proving $RL \subseteq SC$ (which in particular gives a $(\log n)^2$-space, polynomial time algorithm for $USTCON$).

The first reduction in space for this problem was achieved by Nisan, Szemerédi and Wigderson [NSW92] who proved $SL \subseteq L^{3/2}$. The key idea is to scale down Nisan's UTS. They use short ($O(\log^2 k)$ bits) UTS's from every vertex of the graph to visit large (size k) neighborhoods. Then a pairwise independent sample of the vertices (which is easily derandomized) is used to create a new graph which is much smaller (by a factor of k), but still captures the connectivity essence of the original. Iterating this process eventually leads to solving $USTCON$ on a 2-node graph. The bottleneck for improving this bound was that log-space UTS can only guarantee neighborhoods of size $\exp(\sqrt{\log n})$, implying a similar shrinking in size per iteration, which implies $\sqrt{\log n}$ iterations.

While it seems that the symmetric structure was essential for the above improvement, Saks and Zhou [SZ95] found a completely different way to obtain the stronger result $RL \subseteq L^{3/2}$. They showed that Nisan's generator can be replaced by a weaker process, an *off-line randomized algorithm*, which uses only $\sqrt{\log n}$ hash functions, that are used repeatedly in $\sqrt{\log n}$ iterations. This required a mechanism for removing the dependencies of the outcome of each iteration on the choice of hash functions, which was achieved by (easily derandomized) perturbations and rounding.

In this paper we show:

Theorem 1: $SL \subseteq L^{4/3}$

The bound of this paper, $SL \subseteq L^{4/3}$, requires a careful combination of the ingredients of both papers above, with some new ideas. We will follow the shrinking scheme of [NSW92]. However, we replace the UTS of [NSW92] by a pseudo-random walk using Nisan's generator with *short* hash functions. We show that such a walk of length $k^{O(1)}$, just like a random one,

will visit (the neighbors of) $O(k)$ vertices with constant probability on every graph of any size! Note that since only $(\log k)^2$ bits are used, Nisan's analysis does not apply, and the analysis we provide is one of the technical contributions of the paper.

Next we would like to repeatedly use the same hash functions of this pseudo-random walk in many iterations, in [SZ95] style. To remove dependencies we approximate the average behavior of this set of functions (an object independent of any particular one function) with sufficient accuracy and high probability. This is achieved without space and random bit penalty via deterministic sampling of either [BGG93] or the recent extractor constructions of [Zuc96]. This technique for removing dependencies, stated as Theorem 6, is the other technical contribution of this paper, and may become a useful derandomization tool.

The rest of the paper is organized as follows. We start with some preliminaries (Section 2), continue with an informal overview (Section 3) followed by a high level description of our algorithm (Section 4). We then describe the two results upon which the construction is based: the sampler that (w.h.p) does not depend on its random coins (Section 5), and the derandomized short random walks (Section 6).

2 Preliminaries

2.1 Random Computations

The usual definition of a randomized space bounded computation allows the machine to flip a random coin at any point of the computation. It does not allow the machine to recall the outcome of previous coin tosses. This is equivalent to giving the machine a *read-once* access to a long-enough random tape.

A different variant to this model is the one where the machine is allowed *multiple access* to the random tape. We call such algorithms *off-line randomized algorithms* [SZ95]. They are sometimes also called algorithms with 2-way random input. An *off-line randomized* algorithm A is a randomized algorithm such that upon receipt of an input x, it first computes the total number $R(|x|)$ of random bits it will need for the computation and then requests a random string $y \in \{0,1\}^{R(|x|)}$ from the random source and stores it in a read-only section of memory; given x and y, the computation is then completely deterministic. We use the notation $(x;y)$ to separate the "true" input x from the "off-line random input" y.

An off-line randomized algorithm A is said to have *off-line random bit complexity* $R(\cdot)$ and *processing space complexity* $S(\cdot)$ if on any input of length l, the algorithm requests $R(l)$ random bits and given these bits it runs in space $S(l)$.

We say that A accepts a language with *one-sided error* if for a given input x in the language, the probability over a randomly chosen $y \in \{0,1\}^{R(|x|)}$ that $A(x;y)$ outputs 1 is at least $1/2$; and for a given x not in the language, it always outputs 0.

There is a trivial way to derandomize an off-line randomized algorithm A with one-sided error: given an input x, we simply enumerate over all the choices of y and run $A(x;y)$, and output 1 if and only if any of the runs outputs 1. The enumeration over y clearly takes space $O(R(|x|))$ and for each y the computation of $A(x;y)$ takes processing space $O(S(|x|))$ by definition. Observing that the same processing space can be re-used for each y, the space needed for the trivial derandomization is $O(R(|x|) + S(|x|))$.

2.2 Matrix Algorithms

A *matrix algorithm* A is an algorithm that gets a primary input x and in addition two indices $i, j \in [d]$; the output of A is interpreted as the (i,j)-th entry of a $d \times d$ matrix over the reals, denoted $A(x)$. Since the entire matrix $A(x)$ can be obtained by running the algorithm over all the indices $i, j \in [d]$, in a sense, a matrix algorithm is a function that maps inputs x to square matrices over the reals, given in some finite representation. Understood that the computation operates in the way described, we will say that a matrix algorithm A on input x *computes* a matrix $A(x)$.

Let $M_0 = M, M_1, \ldots, M_p$ be a sequence of square matrices and $z_1, \ldots, z_p$ be a sequence of additional parameters. For typographical simplicity, we will denote a sequence $[x_1, \ldots, x_p]$ by $[x_i]_p$, e.g., $[M_i]_p$ stands for $(M_1, \ldots, M_p)$ and $[z_i]_p$ for $(z_1, \ldots, z_p)$. We also denote the dimension of M by $dim(M)$, i.e., if M is a $d \times d$ matrix then $dim(M) = d$. The following fact is well known:

Proposition 2: Suppose there is a matrix algorithm A such that for any $1 \leq i \leq p$, A on input (M_{i-1}, z_i, i) computes M_i and runs in space $S_i \geq \log(dim(M_{i-1}))$. Then there is a matrix algorithm F such that F on input $(M = M_1, [z_i]_p)$ computes M_p and runs in space $O(\sum_{i=1}^{p} S_i)$.

Such an algorithm F will be called a *recursive matrix algorithm* and we say that F on input $(M, [z_i]_p)$ *(recursively) computes* a sequence of matrices $[M_i]_p$.

3 Informal Overview

One common point of the algorithms presented in [Sav70] and [NSW92] for connectivity is that they both can be viewed as a recursive matrix algorithm such that on an input instance (G, s, t) of undirected st-connectivity, it computes a sequence of graphs $[G_i]_p$ that satisfy the following properties which we call *recursive connectivity properties* (RCP):

1. For each $1 \leq i \leq p$, s, t are vertices in G_i and s, t are connected in G_i if and only if they are connected in G_{i-1}, where $G_0 = G$.

2. The sizes of G_i are non-increasing.

3. G_p is a transitive closure graph (union of cliques).

The output of the algorithm is "yes" iff s is connected to t in G_p. Clearly such an algorithm recognizes the language undirected st-connectivity.

3.1 Savitch's Algorithm

We will identify a graph G with its adjacency matrix, denoted also by G, in which $G[i,j]$ is 1 if (i,j) is an edge in G and is 0 otherwise.

In [Sav70], the algorithm computes the sequence of graphs $[G_i]_p$ such that $G_i = G_{i-1}^2 = G^{2^i}$. In words, vertices u, v are connected in G_i if and only if there is a path of length at most 2^i from u to v in G. Clearly, $[G_i]_p$ satisfy RCP. We can see that the space used at every recursive level (to compute G_i given G_{i-1}) is $O(\log n)$ and p is $\log n$. So the total space required is $O(\log^2 n)$ (by Proposition 2).

3.2 The Algorithm of Nisan, Szemerédi and Wigderson

The algorithm of [NSW92] computes a sequence of matrices $[G_i, N_i]_p$ depending on a parameter k which we call the *shrinking parameter*. The sequence $[G_i]_p$ satisfies RCP, and has the property that $dim(G_i) \leq dim(G_{i-1})/k$, i.e., the algorithm "shrinks" the graph G by a factor of k at every recursive level and at the same time preserves RCP.

The computation at each recursive level consists of computing two matrices G_i and N_i. N_i is a k-rich *neighborhood matrix*:

Definition 1: [neighborhood matrix] Let G be a graph. A *neighborhood matrix* N of G is a real matrix of dimension $dim(G)$ such that all the entries are in the range $[0,1]$ and if $N[i,j] \neq 0$ then vertex i is connected (by a path) to vertex j in G. A *Boolean* neighborhood matrix of G is a neighborhood matrix of G with *Boolean* entries.

Definition 2: [rich Boolean neighborhood matrix] A Boolean neighborhood matrix N of G is k-rich if any row of N is k-rich. A row i is k-rich if it contains at least $\min\{k, \#CC(i)\}$ "1" entries, where $\#CC(i)$ is the number of the vertices in the connected component of i in G.

N_i is obtained by taking a walk on the graph G_i according to the universal traversal sequence constructed in [Nis92], which takes space $O(\log^2 k + \log n)$. Using the UTS property, it can be easily proved that N_i is indeed k-rich. The computation of G_i is based on a matrix

algorithm which we denote by SRNK for *graph shrinking procedure*. This algorithm will be the basic building-block of the construction of our algorithm and we summarize its properties as follows.

Lemma 3: Given as input (G, s, t, N), where G is an undirected graph of size n, s, t are vertices in G and N is a k-rich neighborhood matrix of G, the algorithm SRNK runs in space $O(\log(n))$ and computes a graph G' such that

1. s, t are distinct vertices in G' and s is connected to t in G iff it is connected in G'.
2. $dim(G') \leq dim(G)/k$.

In the [NSW92] algorithm, G_i is computed as SRNK (G_{i-1}, s, t, N_{i-1}). We see that the space required at every recursive level of the algorithm is $O(\log^2 k + \log n)$ and the number of recursive levels is $O(\log n / \log k)$, thus the total space needed for the computation is $O(\log^2 n / \log k + \log n \log k)$. By choosing $\log k = \log^{1/2} n$ the algorithm has space complexity $O(\log^{3/2} n)$.

3.3 Our Variant - Intuition

In [NSW92] the rich neighborhood matrix N_i, at each recursive level i, was computed using a UTS, constructed by Nisan using his pseudo-random generator. As we will see the NSW algorithm can be modified so that it directly uses the generator. At each recursive level, we use a random string y'_i of length $O(\log^2 k)$ and additional space $O(\log(n))$ to compute a neighborhood matrix $N_i(y'_i)$ which is k–rich with high probability. Summing over the recursive levels, the naive derandomization of this algorithm uses, as before, space $O(\log(n)\log(k) + \log^2 n / \log(k))$.

The advantage of this modification is that if we can find a way to reduce the number of random bits, we can improve the space complexity. As described, we generate a new set of $O(\log^2 k)$ bits at each recursive level. Suppose that we could reuse the same bits at every recursive level. Then the overall space requirements would be $O(\log^2 k + \log^2 n / \log(k))$, and choosing $\log(k) = \log^{2/3} n$ would give $O(\log^{4/3} n)$ space complexity.

Reusing the random bits introduces dependencies at the various recursive levels, which must be controlled. A technique for this was developed in [SZ95]. However, their technique requires that at each recursive level there is a "target" matrix such that with very high probability the randomized algorithm produces a matrix that is very close to the target. This is not the case for us, because the neighborhood matrix $N_i(y_i)$ produced at level i heavily depends on the random string y_i, and different strings y_i produce very different neighborhood matrices.

To overcome this we notice that if we average over all the possible neighborhood matrices N_y that our randomized algorithm can give, i.e., taking $\mathcal{N} = \mathbf{E}_y[N_y]$, we also get a rich

neighborhood matrix. The crucial point is that the average does not depend on y. Here we also see another advantage of replacing the UTS by a randomized algorithm: since most y's are good the average is also good.

We are left with the problem of approximating $\mathcal{N}$ using only small space. One way to compute $\mathcal{N}$ is by computing N_y for every possible y, which results in space complexity that is essentially the length of the random string y, which is too expensive for us. A more efficient way is by using efficient samplers [BGG93, Zuc96].

4 The New Algorithm

We now formally describe how to implement the ideas presented in Section 3.3.

4.1 An algorithm for finding rich neighborhoods

We apply Nisan's pseudo-random generator for space-bounded computation [Nis92] to simulate short random walks on graphs, and construct an off-line randomized algorithm, which we call WALK for *pseudo-random walks*, that gives the following:

Lemma 4: Given as input a graph G of size n and an integer m, the algorithm WALK $(G, m; y)$ takes an off-line random input $y \in \{0,1\}^r$ of length $|y| = r = O(m^2)$, runs in space $O(\log n)$ and computes a Boolean neighborhood matrix N_y of G such that for each $1 \leq v \leq n$,

$$\Pr_{y \in \{0,1\}^r}[\text{the } v\text{-th row of } N_y \text{ is } 2^m\text{-rich}] \geq \frac{2}{3}$$

We prove this lemma in Section 6.

4.2 The average is also rich

The significance of procedure WALK can be seen if we consider the expectation over y

$$\mathcal{N}(G, m) = \mathbf{E}_y[N_y] = \mathbf{E}_{y \in \{0,1\}^{O(m^2)}}[\text{WALK } (G, m; y)]$$

We first extend the notion of a rich neighborhood matrix and then we show that $\mathcal{N}(G, m)$ is $(k, \frac{1}{n})$-rich.

Definition 3: [rich neighborhood matrix] A neighborhood matrix N of G is (k, α)-rich if any row of N is (k, α)-rich. A row i is (k, α)-rich if it contains at least $max\{k, \#CC(i)\}$ entries j with $N[i, j] > \alpha$, where $\#CC(i)$ is the number of vertices in the connected component of i in G.

Lemma 5: For any graph G of size n and any m, $\mathcal{N}(G,m)$ is a $(2^{m-1}, \frac{1}{n})$-rich neighborhood matrix of G.

Proof: If $\mathcal{N}(G,m)[i,j] > 0$ then there is at least one y s.t. WALK $(G,m;y)[i,j] > 0$. Thus, i must be connected to j in G, and $\mathcal{N}(G,m)$ is a neighborhood matrix of G. Now we prove it is rich. Recall that

$$\mathcal{N}(G,m) = 2^{-r} \sum_{y \in \{0,1\}^r} \text{WALK } (G,m;y)$$

where r is the number of random bits requested by algorithm WALK on input (G,m). Let $i \in [n]$ be arbitrary and let w be the number of entries of $\mathcal{N}(G,m)[i,\cdot]$ that are bigger than $\frac{1}{n}$. We want to show that $w \geq 2^{m-1}$.

By the definition of w, we know that

$$\sum_{j \in [n]} \mathcal{N}(G,m)[i,j] \leq w \cdot 1 + (n-w) \cdot \frac{1}{n}$$

On the other hand,

$$\sum_{j \in [n]} \mathcal{N}(G,m)[i,j] = 2^{-r} \sum_{y \in \{0,1\}^r} \sum_{j \in [n]} \text{WALK } (G,m;y)[i,j] \geq \frac{2}{3} 2^m$$

where the inequality follows from Lemma 4, because for at least $\frac{2}{3}$ of the y's, WALK $(G,m;y)[i,\cdot]$ contains at least 2^m entries with value 1. Thus $w \geq \frac{2}{3}2^m - 1 \geq 2^{m-1}$ and the proof is complete. □

Notice that the averaging argument allows us relaxed requirements from the algorithm WALK ; Instead of requiring that most y's are "good" for every vertex v, we only require from algorithm WALK that for every vertex v most y's are "good" (for different v's, different y's are good).

4.3 Approximating the average

We now want to approximate the average $\mathcal{N}(G,m)$, using an off-line randomized algorithm. For reasons that will become apparent soon, we want the approximation to be (almost) independent of the off-line random string that we use. Employing ideas from [SZ95] this can be done without any space penalty. We start with a definition:

Definition 4: Let $A(x;y,q)$ be a randomized off-line algorithm, whose off line random input is partitioned into two parts y and q. For each x and q we define $M_A(x;q)$ to be the output value $A(x;y,q)$ that occurs with maximum probability over y (with ties broken arbitrarily).

We say A and q are γ*–oblivious to* y, for $\gamma \leq 1$, if

$$\forall x \; Prob_y(A(x;y,q) = M_A(x;q)) \geq 1 - \gamma$$

and in that case we say q is γ*–good.* We say A is γ*–oblivious to* y *with probability at least* p if

$$\forall x \; Prob(q \text{ is } \gamma\text{–good}) \geq p$$

We have the following general theorem:

Theorem 6: (Approximating an average) Let A be an off-line randomized algorithm with random bit complexity $R(\cdot)$ and processing space complexity $S(\cdot)$; the output of A is in $K = [0,1]^d$ on which we define the l_∞ norm $\|\cdot\|$.

Let $\beta < 1$ and $b < R(\cdot)^\beta$. There is an algorithm $B(x; y, q)$ such that

B **is nearly independent of** y : B is γ–oblivious to y with probability at least $1 - \frac{2d}{2^b}$, where $\gamma = \frac{d}{2^{R(|x|)}}$.

Correctness : For all x and γ–good $q \in \{0,1\}^b$, $\|M_B(x; q) - \mathcal{A}(x)\| < 2^{-b}$, where $\mathcal{A}(x) = \mathbf{E}_{y' \in \{0,1\}^{R(|x|)}}[A(x; y')]$.

Complexity : B has $O(S(\cdot) + b) + \text{poly} \log R(\cdot)$ processing space complexity. Also, $|y| \leq 3R(|x|)$ and $|q| \leq b$.

We prove this Theorem in Section 5.

4.4 Repeated Averaging

We can use Theorem 6 to get a recursive matrix algorithm that repeatedly computes the average of an algorithm using the same off-line random string in all levels.

Theorem 7: (Repeated averaging of p levels) Let $K = [0,1]^d$ be a space with l_∞ norm $\|\cdot\|$. Let A_i, $i = 1, \ldots, p$, be a sequence of off-line randomized algorithms with input and output in K, processing space complexity $S(\cdot)$ and off-line random bit complexity $R(\cdot)$. Define $\mathcal{A}_i(x) = \mathbf{E}_{y \in \{0,1\}^{R(|x|)}} A_i(x; y)$.

Let $\beta < 1$. Then for every integer $b < R(|x|)^\beta$ there exists a randomized algorithm AVER with input $x_0 \in K$ that outputs a sequence AVER $(x_0; y, q) = (x_1, \ldots, x_p)$ of p elements in K such that

- For every $x_0 \in K$,

$$Prob_{y,q}(\exists_i \|x_i - \mathcal{A}_i(x_{i-1})\| \geq 2^{-b}) \leq pd(2^{-R(|x_0|)} + 2^{-b+1})$$

- AVER has $O(p(S(|x_0|)+b+\text{poly}\log R(|x_0|)))$ processing space complexity and $O(R(|x_0|)+pb)$ off-line random bit complexity. *

Proof: [of Theorem 7]

We first define the algorithm AVER :

ALGORITHM 4.1 (AVER) *: On input $x_0 \in K$, and parameter b, toss off-line random input $y \in \{0,1\}^{3R(d)}$ and $\vec{q} = (q_1, \ldots, q_p) \in (\{0,1\}^b)^p$. The algorithm outputs the sequence $[x_i]_p$, where*

$$x_{i+1} = B_i(x_i; y, q_i)$$

for $i = 1, \ldots, p$ where B_i is the algorithm that approximates the average of A_i by Theorem 6.

Notice that we use the same y in all different levels. The off-line random bit complexity of the algorithm is immediate. The processing space complexity is also simple. By Proposition 2 the space complexity is the sum of the space complexities of all the different recursive levels, and at each such level the space complexity is at most $O(S(|x_0|) + b) + \text{poly}\log R(|x_0|)$. We are left to prove that:

$$\|x_i - \mathcal{A}_i(x_{i-1})\| \leq 2^{-b}$$

for $i = 1, \ldots, p$.

For every $\vec{q} = (q_1, \ldots, q_p)$, the sequence we see is some $x_1, \ldots, x_p$. The sequence we would like to see (and in fact we almost always see!) is $z_0 = x_0$ and $z_i = M_{B_i}(z_{i-1}, q_i)$ for $i = 1, \ldots p$. Notice that the sequence z is independent of y. We have:

$$\begin{aligned}
\Pr_{y,\vec{q}}[\exists i \in [p],\ x_i \neq z_i] &= \Pr_{y,\vec{q}}[\bigcup_{i=1}^{p}(\ x_i \neq z_i \wedge [x]_{i-1} = [z]_{i-1})] \\
&= \Sigma_{i=1}^{p}\ \Pr_{y,\vec{q}}[B_i(z_{i-1}; y, q_i) \neq M_{B_i}(z_{i-1}; q_i) \wedge [x]_{i-1} = [z]_{i-1}] \\
&\leq \Sigma_{i=1}^{p}\ \Pr_{y,\vec{q}}[B(z_{i-1}; y, q_i) \neq M_{B_i}(z_{i-1}; q_i)]
\end{aligned}$$

But since y and q_i are independent of z_{i-1} it follows by Theorem 6 that $\Pr_{y,\vec{q}}[\exists i \in [p],\ x_i \neq z_i] \leq p \cdot (d2^{-R(|x_0|)} + 2d2^{-b})$. Hence, with probability at least $1 - p \cdot (d2^{-R(|x_0|)} + 2d2^{-b})$ over y and $\vec{q}$ the sequence $x_1, \ldots, x_p$ is exactly the sequence $z_1, \ldots, z_p$. Since $z_i = M_{B_i}(z_{i-1}, q_i)$ we know by Theorem 6 that $\|z_i - \mathcal{A}_i(z_{i-1})\| \leq 2^{-b}$. In particular $\|x_i - \mathcal{A}_i(x_{i-1})\| \leq 2^{-b}$ for any $i = 1, \ldots, p$. □

*We repeat here, at the risk of boring the reader, that in the off-line random bit complexity, p - the number of recursive levels - multiplies only the b, and not the original $R(\cdot)$.

4.5 The connectivity algorithm

The connectivity algorithm is now a simple application of Theorem 7. We choose the domain K to be the set $\{(G,N)\}$ of all $n \times n$ matrices G, N s.t. G is a Boolean adjacency matrix of an undirected graph, and N is a real neighborhood matrix of G. Given $k_1, k_2 \in K$ we view k_1 and k_2 as elements of $[0,1]^d$ (with $d = 2n^2$) and we let $\|k_1 - k_2\|$ be their maximal difference in any coordinate, i.e., we use the l_∞ norm.

Given $x = (G,N)$ we let $A(x;y)$ be (G',N') where $G' = \text{SRNK}\ (G,s,t,N)$ and $N' = \text{WALK}\ (G',m;y)$. I.e., the randomized algorithm A first shrinks G according to N and then computes a new neighborhood matrix N'. A has $O(\log(n))$ processing space complexity and $O(m^2)$ off-line random bit complexity. The shrinked graph G' has fewer vertices, and hence can be represented using fewer dimensions. However, to avoid using different domains K_i, we represent G' as if it still has n vertices with non-active vertices having zero rows and columns. The average is $\mathcal{A}(x) = \mathbf{E}_{y \in \{0,1\}^{R(|x|)}}[A(x;y)]$. On input (G,N) we have $\mathcal{A}(G,N) = (\mathcal{G}', \mathcal{N}')$ where $\mathcal{G}' = \mathbf{E}_y \text{SRNK}\ (G,s,t,N)$ and $\mathcal{N}' = \mathbf{E}_y[\text{WALK}\ (G',m;y)]$. However, SRNK (G,s,t,N) does not depend on y and therefore $\mathcal{G}' = \text{SRNK}\ (G,s,t,N)$.

Given an undirected graph G, and two distinguished vertices s and t, we form the input $x_0 = (G,I)$, where I is the identity neighborhood matrix. We define $A_1, \ldots, A_p$ to be the algorithm A with $m = \log^{2/3} n$. Applying Theorem 7 with the parameters $\beta = \frac{5}{6}$,$d = n^2$,$b = 4 \log n$ and $p = \Theta(\frac{\log n}{m}) = \Theta(\log^{1/3} n)$, we get an algorithm AVER that outputs a sequence $(x_1, \ldots, x_p)$ s.t.

- With probability at least $1 - 1/n$, $\|x_i - \mathcal{A}(x_{i-1})\| \leq \frac{1}{n^2}$ for every $i = 1, \ldots, p$.
- AVER has $O(\log^{4/3} n)$ processing space complexity and $O(m^2 + \log^{4/3} n) = O(\log^{4/3} n)$ off-line random bit complexity.

We complete the proof of Theorem 1 by proving

Lemma 8: Let $x_0 = (G,I) \in K$ be an input, where G is an undirected graph with two distinguished vertices s and t, and I is the identity neighborhood matrix. Let $p = \Theta(\frac{\log n}{m})$. For any sequence $x_1, \ldots, x_p \in K$ s.t. $\|x_i - \mathcal{A}(x_{i-1})\| < \frac{1}{n}$, x_p represents a graph with only two vertices s and t, and they are connected in x_p iff they are connected in G.

Proof: Suppose $x_i = (G_i, N_i)$. We first prove by induction on i that for every i s is connected to t in G iff it is connected in G_i. The base case $i = 0$ is trivial. Assume for i and let us prove for $i+1$. We know that $\mathcal{A}(x_i) = (\mathcal{G}_{i+1}, \mathcal{N}_{i+1})$ with $\mathcal{G}_{i+1} = \text{SRNK}\ (G_i,s,t,N_i)$, and we are given an element $x_{i+1} = (G_{i+1}, N_{i+1})$ such that $\|x_{i+1} - \mathcal{A}(x_i)\| < \frac{1}{n}$. In particular $\|G_{i+1} - \text{SRNK}\ (G_i,s,t,N_i)\| < \frac{1}{n}$. Since both G_{i+1} and SRNK (G_i,s,t,N_i) are boolean matrices we conclude that $G_{i+1} = \text{SRNK}\ (G_i,s,t,N_i)$. Finally, as N_i is a neighborhood matrix of G_i (as $x_i \in K$) if follows that s is connected to t in G_{i+1} iff it is connected in G_i which by induction is iff s is connected to t in G.

The second thing we show is that the graph G_{i+1} represents a shrink by a factor of 2^{m-1} of G_i. To see this look at $\mathcal{A}(x_{i-1}) = (\mathcal{G}_i, \mathcal{N}_i)$ $(i \geq 1)$. By Lemma 5, $\mathcal{N}_i$ is a $(2^{m-1}, \frac{1}{n})$ rich neighborhood matrix of $\mathcal{G}_i$. Since $\|x_i - \mathcal{A}(x_{i-1})\| = \delta < \frac{1}{n}$ it follows that $G_i = \mathcal{G}_i$ and N_i is a $(2^{m-1}, \frac{1}{n} - \delta)$ rich neighborhood matrix of $G_i = \mathcal{G}_i$. Hence, G_{i+1} shrinks by a factor of at least 2^{m-1}.

Hence, taking $p = \Theta(\frac{\log n}{m})$ we end up with a graph with only two vertices s and t that are connected iff they are connected in x_0. □

5 The Proof of the Approximated-Averaging Theorem

In this section we prove Theorem 6. Suppose we are given a randomized algorithm A that outputs a value in $[0,1]$. Using *samplers* we can *estimate* the *expected value* of A with polynomial accuracy. Clearly, there is no way we can avoid the error in the estimate and get perfect accuracy. Thus, the sampler's answer depends on the random bits it uses.

We would like to break this dependency. A natural way to do this, suggested by [SZ95], is by truncating the approximated value to the desired accuracy level. One can then hope that the resulting (truncated) value is *exactly* the truncation of the correct value. Indeed, this is almost true. To make it true with high probability [SZ95] use first random perturbations. We use this technique to prove Theorem 6.

5.1 Deterministic Sampling

Given a function $f : \{0,1\}^r \to [0,1]$ we want to efficiently approximate the expectation over a randomly chosen $y' \in \{0,1\}^r$ of $f(y')$, i.e., to estimate

$$\mathbf{E}[f] \;=\; \mathbf{E}_{y' \in \{0,1\}^r}[f(y')] \;=\; \frac{1}{2^r}\Sigma_{y' \in \{0,1\}^r} f(y')$$

to a given accuracy, minimizing the number of random bits used.

Definition 5: [BR94, Zuc96] An (ϵ, γ) *non-adaptive sampler* $\mathrm{SMPL}_f : \{0,1\}^l \to [0,1]$ is a deterministic algorithm that on an input l-bit string y, computes p queries to an oracle function $f : \{0,1\}^r \to [0,1]$, and based on the answers outputs a value v s.t.

$$\Pr_{y \in \{0,1\}^l}[|v - \mathbf{E}[f]| > \epsilon] \leq \gamma$$

In other words, for all but γ fraction of the $y \in \{0,1\}^l$ $|v - \mathbf{E}[f]| \leq \epsilon$.

Clearly a natural way to obtain an (ϵ, γ) non-adaptive sampler is to randomly choose independent sample points. In fact, a simple counting argument shows that choosing $p =$

$O(\frac{1}{\epsilon^2}\log(\frac{1}{\gamma}))$ points is enough. However, this way we use too many random bits: $l = pr$. We would like to reduce l to only $O(r)$ while keeping the number of sample points small with $p = poly(r, \frac{1}{\epsilon})$. Two optimal methods for doing that are known, [BGG93] and [Zuc96].

The processing space complexity of a non-adaptive sampler is the processing space needed to compute the queries plus the processing space needed to compute the output given the answers to the queries. We remind the reader that the processing space complexity can be much smaller than the size of the output itself. We use a special case of [Zuc96, Theorem 5.5] by setting $m = r, d = p, \alpha = \frac{1}{2}$ and $\gamma = \frac{1}{2^r}$ to get Lemma 9:

Lemma 9: [Zuc96] For any constant $\beta < 1$, any positive integer r and any function $f : \{0,1\}^r \to [0,1]$ computable in space s and any $\epsilon > 2^{-r^\beta}$, there is an $(\epsilon, \frac{1}{2^r})$ sampler $\text{SMPL}_f : \{0,1\}^{3r} \to [0,1]$ computable in space $s + \log^{O(1)} r$.

We extend this Lemma to $f : \{0,1\}^r \to K$ (where $K = [0,1]^d$ with the l_∞ norm $\|\cdot\|$). We first need to extend the definition of the sampler to work on norm spaces.

Definition 6: An (ϵ, γ) *non-adaptive sampler* $\text{SMPL}_f : \{0,1\}^l \to K$ is a deterministic algorithm that on an input l-bit string y, computes p queries to an oracle function $f : \{0,1\}^r \to K$, and based on the answers outputs a value $v \in K$ such that

$$\Pr_{y \in \{0,1\}^l}[\|v - \mathbf{E}[f]\| > \epsilon] \leq \gamma$$

Lemma 10: For any constant $\beta < 1$, any positive integer r, any function $f : \{0,1\}^r \to K$ computable in space s and any $a < r^\beta$, there is a $(2^{-a}, \frac{d}{2^r})$ sampler $\text{SMPL}_f : \{0,1\}^{3r} \to K$ computable in space $O(s) + \log^{O(1)} r$.

Proof: Remember that $K = [0,1]^d$. For every $j \in [d]$ let f_j be the jth coordinate of f, let SMPL_{f_j} be the $(2^{-a}, 2^{-r})$ sampler of Lemma 9 and define

$$\text{SMPL}_f = (\text{SMPL}_{f_1}, \ldots, \text{SMPL}_{f_d})$$

Let v_j be the jth entry of v. Then,

$$\begin{aligned} \Pr_{y \in \{0,1\}^{3r}}[\|v - \mathbf{E}[f]\| > 2^{-a}] &= \Pr_{y \in \{0,1\}^{3r}}[\vee_{j \in [d]} |v_j - \mathbf{E}[f_j]| > 2^{-a}] \\ &\leq \sum_{j \in [d]} \Pr_{y \in \{0,1\}^{3r}}[|v_j - \mathbf{E}[f_j]| > 2^{-a}] \leq d2^{-r} \end{aligned}$$

By Lemma 9, for every $j \in [d]$, the sampler SMPL_{f_j} runs in space $s + \log^{O(1)} r$ (since f_j is definitely computable in space s) and while computing SMPL_f we need only extra $O(\log d)$ space to point to the jth entry in the output of f, but $s \geq \log d$ since the output is of length at least d. □

5.2 Exact Truncated Sampling

Now we construct the off-line randomized algorithm B that approximates the average of A in a way that (w.h.p.) does not depend on the random bits we use. The construction uses the perturbation and truncation scheme of [SZ95]. We start with a definition:

Definition 7: [Perturbations and Truncations]: Let x be a real value. The *perturbed* value of x, $x_{-\delta}$, is defined to be $\max\{x-\delta, 0\}$. The *perturbed and truncated* value of x, $\lfloor x_{-\delta} \rfloor_t$, is obtained by truncating the binary expansion of $x_{-\delta}$ after t binary digits. These operators are extended to vectors in $K = [0,1]^d$ ($v_{-\delta}$ and $\lfloor v_{-\delta} \rfloor_t$) by simply applying them entry by entry to the vector v.

We set $a = 2b+1$ and set a truncation parameter $t = a - b$. For every input x, let $y \in \{0,1\}^{3R(|x|)}$ and $q \in \{0,1\}^b$ and let $\delta = q2^{-a}$. Define B as

$$B(x; y, q) = \lfloor \text{SMPL}_{A(x,\cdot)}(y)_{-\delta} \rfloor_t$$

and we prove:

Lemma 11: For every input x, if $a = 2b+1 < \mathrm{R}(|x|)^\beta$ for some constant $\beta < 1$, then

$$\Pr_{q \in \{0,1\}^b}[\Pr_{y \in \{0,1\}^{3R(|x|)}}[B(x;y,q) = \lfloor \mathcal{A}(x)_{-\delta} \rfloor_t] > 1 - d2^{-r}] > 1 - 2d2^{-b}$$

Proof: A nonnegative real number x is said to be (a,t)-*dangerous* for positive integers $a > t$ if x can be written in the form $2^{-t}I + \rho$ where I is a positive integer and $\rho \in [-2^{-a}, 2^{-a})$, and is said to be (a,t)-*safe* otherwise. E.g., $0.76 = 3/4 + 0.01$ is $(6,2)$-dangerous and $(7,2)$-safe. A vector $v \in K$ is (a,t)-*dangerous* if one of its entries is (a,t)-dangerous and it is (a,t)-*safe* if all its entries are (a,t)-safe.

For two positive integers $a \geq b$, we define $\Delta(a,b)$ to be the set of 2^b real numbers $\{q2^{-a} \mid \text{integer } q \in [0, 2^b - 1]\}$. The proofs of the next two lemmas can be found in [SZ95].

Lemma 12: [SZ95, Lemma 5.5] Suppose $v \in K$ and $a \geq b$ are positive integers. Let $t = a - b$. Then,

$$\Pr_{\delta \in \Delta(a,b)}[v_{-\delta} \text{ is } (a,t)\text{-dangerous}] \leq 2d2^{-b}$$

Lemma 13: [SZ95, Lemma 5.2] Suppose $v \in K$ is (a,t)-safe. Then for any $v' \in K$ such that $\|v - v'\| \leq 2^{-a}$ it holds that $\lfloor v \rfloor_t = \lfloor v' \rfloor_t$.

Thus, by Lemma 12 for all but $2d2^{-b}$ fraction of the choices of δ, $\mathcal{A}(x)_{-\delta}$ is (a,t)-safe. Also, by Lemma 10, for all but $\frac{d}{2^{R(|x|)}}$ fraction of the choices of y, $\text{SMPL}_{A(x,\cdot)}(y)$ is within

2^{-a} to the real average $\mathcal{A}(x)$. By Lemma 13 whenever one vector is (a,t)-safe and a second vector is within 2^{-a} distance of the first, then their truncation at position t is equal. This completes the proof of Lemma 11. □

Note now that if $\Pr_{y\in\{0,1\}^{3R(|x|)}}[B(x;y,q) = \lfloor\mathcal{A}(x)_{-\delta}\rfloor_t] > 1 - d2^{-r}$, that means that $\lfloor\mathcal{A}(x)_{-\delta}\rfloor_t$ is the value of $B(x;y,q)$ that appears maximal number of times over the random input $y \in \{0,1\}^{3R(|x|)}$, which implies that $M_B(x;q) = \lfloor\mathcal{A}(x)_{-\delta}\rfloor_t$, which in turn implies that q is γ–good for x, where $\gamma = d2^{-r}$. Then, by Lemma 11

$$\Pr_{q\in\{0,1\}^b}[q \text{ is } \gamma\text{–good for } x] > 1 - 2d2^{-b}$$

which proves the first item of Theorem 6. To complete the proof of the Theorem one should note that $\|\lfloor\mathcal{A}(x)_{-\delta}\rfloor_t - \mathcal{A}(x)\| \leq 2 \cdot 2^{-t}$ because each of the perturbation and truncation operation introduces an error of at most 2^{-t} which proves the second item of the Theorem, and since B is simply applying perturbation and truncation to the sampler that runs in space $O(s) + \log^{O(1)} r$, everything can be computed in space $O(s+b) + \text{poly}\log R(|x|)$, which completes the proof of Theorem 6.

6 Pseudo-random Walks

In this section we present algorithm WALK (G,m;h) and prove Lemma 4. The algorithm receives a graph G and a number m as inputs, a random string h as an off-line random input, and since it is a matrix algorithm it also receives two more indices, s, u. It should output the value of the (s,u)th entry of the output matrix. We want to show that for any graph $G = (V,E)$ and any vertex $s \in V$, for at least $\frac{2}{3}$ of the off-line random inputs h, there are at least 2^m vertices u s.t. WALK $(G,m;h)[s,u] = 1$.

6.1 Preliminaries

6.1.1 Random walks

Given an undirected regular graph $G = (V,E)$ of degree D and a vertex $s \in V$ a sequence $\sigma_1,\ldots,\sigma_l \in [1..D]$ defines a "walk" $v_1,\ldots,v_{l+1}$ of length l in the following way: $v_1 = s$, and v_{i+1} is the σ_i'th neighbor of v_i. I.e., we start at the vertex s, and each σ_i moves us to the σ_i'th neighbor of our current vertex. A random walk of length l on an undirected graph $G = (V,E)$ is the walk obtained by a sequence $\sigma_1,\ldots,\sigma_l$ picked uniformly at random from $[1..D]^l$.

We are interested on random walks on *irregular* undirected graphs $G = (V,E)$ with degree *at most* D. When the graph is irregular we need to specify how an instruction σ_i translates to a move. For example, how does the instruction $\sigma \in [1..16]$ translate to a move from a

vertex v that has 3 neighbors. There are several standard ways to address this (e.g., by first converting the graph to a regular graph) and we choose the following. We express D as $[1, jd] \cup [jd+1, D]$ where j is the biggest integer s.t. $jd \leq D$. If $\sigma_i \in [1, jd]$ we let v_{i+1} be the σ_i mod d neighbor of v_i, otherwise $v_{i+1} = v_i$, i.e., we do nothing. Using this solution, at least half of the moves are "real" moves, and moreover, each neighbor of v has equal probability of being reached from v.

We need the following lemma of Barnes and Feige [BF93]:

Lemma 14: [BF93] Suppose G is a connected graph and s is an arbitrary vertex in G. Let k be an integer. Then with probability at least $3/4$ a random walk of length $l = l(k) = O(k^3)$ on G from s visits at least k distinct vertices or the whole connected component of s.

We could equally use a slightly weaker Lemma with $l = O(k^4)$ of [Lin92]. For the rest of this section, let us fix the graph G, the starting vertex (the row of the output matrix) s, the parameter m, the number of vertices that we should visit $k = 2^m$ and the length $l = O(k^3)$ given by Lemma 14. We will analyze random walks on G, that start from s.

6.1.2 Nisan's generator

We remind the reader that an automaton Q is a tuple $\langle S, \Sigma, q_{start}, q_{acc}, \delta \rangle$, with states S, a starting state $q_{start} \in S$, an accepting state $q_{acc} \in S$, an alphabet Σ and a transition function $\delta : S \times \Sigma \mapsto S$. If the automaton is in state q and is given an input $x \in \Sigma$ the automaton moves to the new state $\delta(q, x)$. For a distribution $\mathcal{D}$ over Σ^l, we denote by $Q(\mathcal{D})$ the distribution over S obtained by starting in q_{start} and running the automaton Q over a sequence drawn at random from $\mathcal{D}$.

Our main tool is Nisan's generator [Nis92] that can be described as a function $F_{q,r}(x, h) : \{0,1\}^q \times \{0,1\}^{2qr} \mapsto \{0,1\}^{q2^r}$ where q and r are parameters that determine the quality of the generator. The key property of $F_{q,r}$ is given in the following lemma, which is derived from [Nis92, Lemma 2] by plugging some specific constants that suit our needs.

Lemma 15: Let S be a set with $|S| = O(2^{7m})$, $k = 2^m$ and $l = O(2^{3m})$. There exists a function $\nu = O(m)$ such that for every automaton $Q = \langle S, \Sigma = \{0,1\}^\nu, q_{start}, q_{acc}, \delta \rangle$, for all but k^{-5} of the choices of $h \in (\{0,1\}^{2\nu})^{\log l}$

$$\|Q(U_l) - Q(F_{\nu,\log\ l}(U_x, h)\| \leq k^{-5}$$

where U_l is the uniform distribution over Σ^l, U_x is the uniform distribution over Σ and $\|\cdot\|$ is the l_∞ norm.

6.2 The Algorithm WALK

Our algorithm is:

Algorithm WALK
Let $l = O(k^3)$ be as in Lemma 14 and $\nu = O(m)$ as in Lemma 15.
Input: A graph G of n vertices and a parameter m; two vertices s, u.
Initialization: Set $k = 2^m$.
Off-line Random Input: $h \in (\{0,1\}^{2\nu})^{\log l}$.
Output: Try every $x \in \{0,1\}^{\nu}$. If for one of the x's, a walk on G according to $F(x,h)$ from s passes through u or through a neighbor v of u, output 1. Otherwise, output 0.

6.3 The proof

We want to prove Lemma 4, i.e., we want to show that for any given vertex s, if we choose and fix h at random then with high probability (over the choice of h) WALK $((G,s,u);h) = 1$ for at least k different vertices u. This says, in a sense, that h is good for s : using the same h in all the n different applications of WALK $(G,s,u);h)$ yields at least k distinct neighbors of s. The lemma claims that for any s, most h's are good for s.

If the Lemma fails then there is a graph $G = (V,E)$ and a vertex $s \in V$ such that with probability at least $\frac{1}{3}$ (over the choice of h) fewer than $k = 2^m$ vertices are found. Given such a "bad" instance we would like to build an automaton Q, with only $O(2^{7m}) = poly(k)$ states, for which $Q(U_l)$ and $Q(F_{\nu,\log l})$ are very different. Since Lemma 15 guarantees that no such Q exists we conclude that no bad instance exists. The key property that we need from Q is that it has only $poly(k)$ states. To achieve that we identify a set H of only $poly(k)$ vertices in V and prove that it is enough to focus on these vertices alone.

6.3.1 The Set H

Let $P_{v,w}$ be the probability a random step from v moves to w (and so if $(v,w) \notin E$, $P_{v,w} = 0$). We define $\rho_0(s) = 1$ and $\forall v \neq s$, $\rho_0(v) = 0$. For every $i \geq 0$ define

$$\begin{aligned} H_i &= \{v : \rho_i(v) \geq \frac{1}{2lk}\} \\ L_i &= \{v : 0 < \rho_i(v) < \frac{1}{2lk}\} \\ \rho_{i+1}(v) &= \sum_{w \in H_i} \rho_i(w) \cdot P_{w,v} \end{aligned}$$

H_i is the set of vertices with "high" reaching probability in the ith step, L_i are the vertices with "low" reaching probability and $\rho_i(v)$ is the probability of reaching v in the ith step when walking only through vertices in $H_1, \ldots, H_{i-1}$. Clearly, $|H_i| \leq 2lk \leq poly(k)$.

t will be the number of steps that we want to analyze. It is at most l, but if at time $i < l$ the probability of entering L_{i+1} is too large, or, we discover a vertex with at least k neighbors, then we stop. I.e., denote by $\Gamma(v) = \{w : (u,w) \text{ is an edge of } G\}$ the set of neighbors of v and for a subset U of vertices, let $\Gamma(U)$ denote the set of neighbors of U. Define:

$$t = \min\{\{l\} \cup \{i : \exists v \in H_i, |\Gamma(v)| \geq k\} \cup \{i : \sum_{v \in L_{i+1}} \rho_{i+1}(v) > \frac{1}{2l}\}\}$$

Denote by $H = \bigcup_{i=0}^{t} H_i$.

Lemma 16: $|H \cup \Gamma(H)| \geq k$

Proof: If $t = l$, the probability that a random walk leaves H is at most $l \cdot \frac{1}{2l} = \frac{1}{2}$ and the probability that it does not visit k distinct vertices is at most $\frac{1}{4}$ (by Lemma 14). Thus, with probability at least $\frac{1}{4}$, the random walk both stays in H and visits k distinct vertices, thus $|H| \geq k$.

If $t < l$, then there are two possible cases: (1) for some $v \in H_t$, $|\Gamma(v)| \geq k$, or (2) $\sum_{v \in L_{t+1}} \rho_{t+1}(v) > \frac{1}{2l}$. If the first case happens then clearly $|\Gamma(H)| \geq k$. If the second case happens, then since for every $v \in L_{t+1}$, $\rho_{t+1}(v) < \frac{1}{2lk}$, it must be that the size of L_{t+1} is at least k. Moreover, $L_{t+1} \subseteq \Gamma(H_t) \subseteq \Gamma(H)$. We see that in either case $|\Gamma(H)| \geq k$. □

6.3.2 A pseudo-random walk traverses H

Lemma 17: For every $v \in H$, for all but k^{-5} of the choices of $h \in (\{0,1\}^{2\nu})^{\log l}$, for at least one of the $x \in \{0,1\}^{\nu}$ the walk according to the output of $F(x,h)$ in G will pass through the vertex v.

Proof:

Given the graph $G = (V,E)$ and the vertex $s \in V$, we construct an automaton $M = \langle S, \Sigma, q_{start}, q_{acc}, \delta \rangle$ of size polynomial in k. The states S of the automaton M are

$$S = \{(i,u) : 0 \leq i \leq t, u \in H_i \cup \{0\}\ \}$$

It is easy to see that $|S| = O(l^2 k) = O(k^7) = O(2^{7m})$. The starting state of M is $q_{start} = (s,0)$. The alphabet Σ of the automaton is $\Sigma = \{0,1\}^{\nu}$, where ν is a parameter of the generator ($\nu = O(m) = O(\log k)$).

We now define the transition function δ. For $a = 0$, $\delta((i, a = 0), \sigma) = (i+1, 0)$. For $a \neq 0$ we have $a \in H_i$, and therefore it has $d \leq k < 2^{\nu}$ neighbors. We again express $[1..2^{\nu}]$ as $[1..jd] \cup [jd+1..2^{\nu}]$ with j being the largest integer s.t. $jd \leq 2^{\nu}$. If $\sigma > jd$ we define $\delta((i,a),\sigma) = (i+1,a)$, i.e., we do nothing. Otherwise, let b be the σ mod d neighbor of a. If $b \in H_{i+1}$ then $\delta((i,a),\sigma) = (i+1,b)$ otherwise $\delta((i,a),\sigma) = (i+1,0)$.

We now claim that for every $i \leq t$ and for every $w \in H_i$ the probability over $\sigma \in (\{0,1\}^\nu)^i$ that a walk on M according to σ reaches (i,w) is at least $\rho_i(w)$, i.e., in $M(U_i)$ the state (i,w) has probability at least $\rho_i(w)$. To see that notice that any walk in G of length i that passes only through vertices in $H_1, \ldots, H_i$ and ends up at w, has a unique corresponding walk on the automaton M that ends up in (i,w).

Therefore, It follows by Lemma 15 that for all but k^{-5} of the choices of h the state (i,w) has probability at least $\rho_i(w) - 2k^{-5}$ in $M(F_{\nu,\log l})(U_x, h)$. We conclude:

Lemma 18: For all but k^{-5} of the choices of $h \in (\{0,1\}^{2\nu})^{\log l}$, for every $i \leq t$ and for every $w \in H_i$,

$$\Pr_{x \in \{0,1\}^\nu}[M(F(x,h)) \text{ passes through } (i,w)] \geq \rho_i(w) - 2k^{-5}$$

Finally, we note that by the construction of the automaton M for every string $\sigma \in (\{0,1\}^\nu)^i$, where $i \leq t$, if σ takes M to some state (i,w) with $w \neq 0$, then a walk on G according to σ will end up at vertex w as well. Since $\rho_i(w) \geq \frac{1}{2lk} \geq \Omega(k^{-4})$, it follows that for at least one of the $x \in \{0,1\}^\nu$, the walk according to the output of F in G will pass through the vertex w. □

6.3.3 Putting the pieces together

Lemma 19: For at least 3/4 of the h, algorithm WALK using h will see at least k distinct vertices or the whole connected component of s.

Proof:

If the connected component of s is smaller than k, then the walks according to the output of F will traverse it completely for almost every h since F is a pseudo-random generator for automata of such size. In the other case where the connected component of s is at least of size k, we know that for all but k^{-5} of the possible h's, the probability over the x's that a walk on G according to $F(x,h)$ passes through a vertex v in H is positive. If $|H| \geq k$, then for all but k^{-4} of the possible h's, when we try all possible x's, we see k elements of H. If $|H| \leq k$, then for all but k^{-4} of the possible h's, when we try all possible x's, we see all the elements of H. However, by Lemma 16, $|\Gamma(H)| \geq k$, and WALK outputs 1 for every u that it either traverses or a neighbor of a traversed vertex v, thus WALK will output 1 for every vertex of $\Gamma(H)$. □

Finally, notice that the length of the off-line random input is $2\nu \cdot \log l = O(m^2)$. As for the space complexity, $\nu = O(m)$ and since $m \leq \log n$, the space needed to store an $x \in \{0,1\}^\nu$ or an output block of F is certainly $O(\log n)$. The space complexity of computing a block of $F(x,h)$ is also $O(\log n)$ and the space required to walk on a graph is of course $O(\log n)$ as well. We have proved Lemma 4. □

Acknowledgment

The authors are grateful to Mike Saks for many helpful discussions on the subject, and to the anonymous referee for many helpful comments that greatly improved the paper's presentation.

References

[AKL+79] Romas Aleliunas, Richard M. Karp, Richard J. Lipton, Laszló Lovász, and Charles Rackoff. Random walks, universal traversal sequences and the complexity of maze problems. In *20th Annual Symposium on Foundation of Computer Science*, pages 218–223. IEEE, 1979.

[BF93] Greg Barnes and Uriel Feige. Short random walks on graphs. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 728–737. ACM, 1993.

[BGG93] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Randomness in interactive proofs. *Computational Complexity*, 3(4):319–354, 1993.

[BR94] Mihir Bellare and John Rompel. Randomness-efficient oblivious sampling. In *Proceedings of the 32nd Symposium on Foundation of Computer Science*, pages 276–287. IEEE, 1994.

[Lin92] Nati Linial. Private communication. *Unpublished manuscript*, 1992.

[LP82] Harry R. Lewis and Cristos H. Papadimitriou. Symmetric space-bounded computation. *Theoretical Computer Science*, 19:161–187, 1982.

[Nis92] Noam Nisan. Pseudorandom generators for space-bounded computation. *Combinatorica*, 12(4):449–461, June 1992.

[Nis94] Noam Nisan. RL $\subseteq$ SC. *Computational Complexity*, 4, 1994.

[NSW92] Noam Nisan, Endre Szemeredi, and Avi Wigderson. Undirected connectivity in $O(\log^{1.5} n)$ space. In *Proc. 33th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 24–29, 1992.

[Sak96] Michael Saks. Randomization and derandomization in space-bounded computation. *11th Annual Conference on Computational Complexity*, 1996.

[Sav70] Walter J. Savitch. Relationships between nondeterministic and deterministic space complexities. *Journal of Computer and System Sciences*, 4(2):177–192, 1970.

[SZ95] Michael Saks and Shiyu Zhou. $RSPACE(S) \subset DSPACE(S^{\frac{3}{2}})$. In *Proc. 36th Annual IEEE Conference on Foundations of Computer Science (FOCS)*, pages 344–353, Milwaukee, Wisconsin, October 1995.

[Wig92] Avi Wigderson. The complexity of graph connectivity. In *Mathematical Foundations of Computer Science: Proceedings, 17th Symposium, Lecture Notes in Computer Science*, volume 629, pages 112–132, 1992.

[Zuc96] David Zuckerman. Randomness-optimal sampling, extractors, and constructive leader election. In *ACM Symposium on Theory of Computing (STOC)*, pages 286–295, Philadelphia, Pennsylvania, May 1996.