Time-Space Tradeoff In Derandomizing Probabilistic Logspace *

Jin-Yi Cai   Venkatesan T. Chakaravarthy   Dieter van Melkebeek

Computer Sciences Department
University of Wisconsin, Madison.
Email: {jyc,venkat,dieter}@cs.wisc.edu.

Abstract

Nisan [5] showed that any randomized logarithmic space algorithm (running in polynomial time and with two-sided error) can be simulated by a deterministic algorithm that runs simultaneously in polynomial time and $\Theta(\log^2 n)$ space. Subsequently Saks and Zhou [8] improved the space complexity and showed that a deterministic simulation can be carried out in space $\Theta(\log^{1.5} n)$. However, their simulation runs in time $n^{\Theta(\log^{0.5} n)}$. We prove a time-space tradeoff that interpolates these two simulations. Specifically, we prove that, for any $0 \leq \alpha \leq 0.5$, any randomized logarithmic space algorithm (running in polynomial time and with two-sided error) can be simulated deterministically in time $n^{O(\log^{0.5-\alpha} n)}$ and space $O(\log^{1.5+\alpha} n)$. That is, we prove that $\mathsf{BPL} \subseteq \mathsf{DTISP}[n^{O(\log^{0.5-\alpha} n)}, O(\log^{1.5+\alpha} n)]$.

1 Introduction

Given an undirected graph and two vertices s and t, the undirected st–connectivity problem consists in checking whether there exists a path from s to t. A *randomized* logarithmic space algorithm for this problem was known for a long time, but a *deterministic* one remained elusive until very recently. Aleliunas et al. [1] showed in 1979 how to use a random walk to solve the problem in randomized logarithmic space (with one-sided error). Several attempts were made to obtain a deterministic logarithmic space algorithm via derandomizing the random walk, and undirected st–connectivity grew into the prime example of a computational problem where randomness seemed to reduce the space complexity. However, earlier this year, Reingold [6] eventually managed to construct a deterministic logarithmic space algorithm for undirected st–connectivity.

This paper deals with the fundamental question whether *every* computational problem that has a randomized logarithmic space algorithm, also has a deterministic logarithmic space algorithm. For any randomized logspace computation, a straightforward dynamic program yields a deterministic simulation that runs in polynomial space and time, but better results are known. Savitch's theorem [9] gives an algorithm that uses only $O(\log^2 n)$ space but runs in time $n^{O(\log n)}$. In a major breakthrough, Nisan [4] constructed a pseudorandom generator that stretches a seed of length $O(\log^2 n)$ into a string of polynomial length. He showed that the output of the generator (on a

*A preliminary version was presented at STACS 2004. Research supported in part by NSF grants CCR-0208013 and CCR-0133693.

randomly chosen seed) looks almost truly random for any randomized logspace computation (with two-sided error). Then, if we run through all possible seeds, we get a deterministic simulation. This naive simulation using Nisan's generator needs $O(\log^2 n)$ space and $n^{O(\log n)}$ time, and hence fails to beat the bounds from Savitch's theorem. However, Nisan [5] showed that, in fact, one can perform the simulation in polynomial time, while using only $O(\log^2 n)$ space. This settled an important problem of whether randomized logspace algorithms can be simulated in simultaneous polynomial time and polylog space. Saks and Zhou [8] subsequently proved that any randomized logspace algorithm can be simulated deterministically in $O(\log^{1.5} n)$ space, but the time complexity of their simulation is $n^{O(\log^{0.5} n)}$. In this paper, we generalize the deterministic simulations of Nisan [5], and of Saks and Zhou [8], and exhibit a time-space tradeoff interpolating the two. The question whether every randomized logspace computation can be simulated by a deterministic logspace computation remains open, although we have strong evidence that this ultimate goal can be realized [3]. See Figure 1 for an overview.

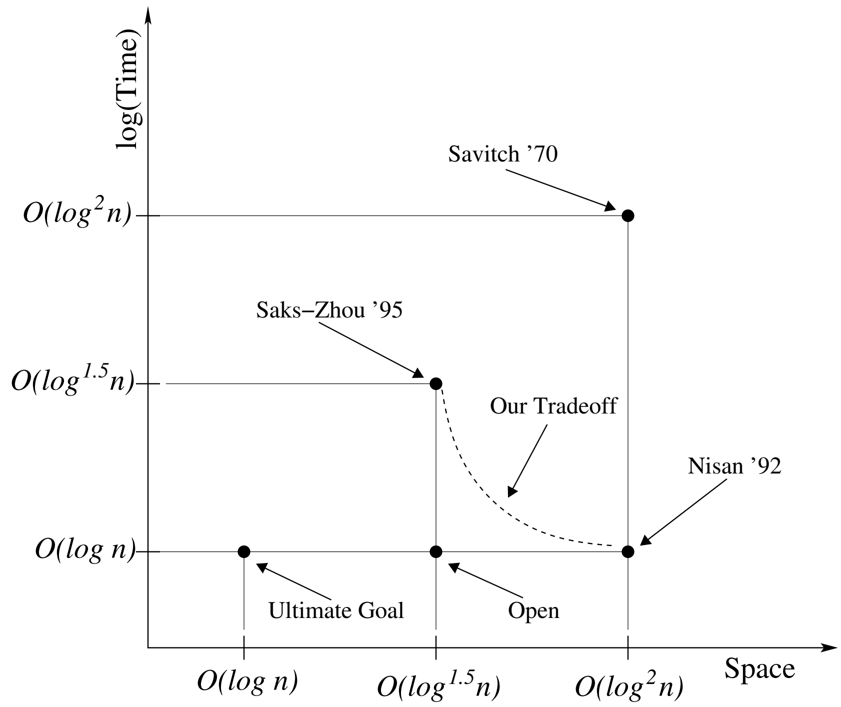

Figure 1: Overview of deterministic simulations of randomized logspace computations.

We denote by BPL the class of all languages that can be recognized by a randomized algorithm with two-sided error that uses logarithmic space and runs in polynomial time (for any input and for any random coin tosses). It contains RL, the one-sided error class. We refer to the excellent survey on probabilistic space bounded complexity classes by Saks [7] for more background. For a time bound $t(n)$ and space bound $s(n)$, let $\mathsf{DTISP}[t, s]$ denote the class of languages that can be recognized by a deterministic algorithm that runs in time $t(n)$ and uses space $s(n)$. With this notation, we can restate the previous results as:

- $\mathsf{BPL} \subseteq \mathsf{DTISP}[n^{O(1)}, O(\log^2 n)]$ (Nisan [5]).

- $\mathsf{BPL} \subseteq \mathsf{DTISP}[n^{O(\log^{0.5} n)}, O(\log^{1.5} n)]$ (Saks and Zhou [8]).

In this paper, we establish a time-space tradeoff that generalizes the above two results:

Theorem 1 *For any rational number* $0 \leq \alpha \leq 0.5$*,* $\mathsf{BPL} \subseteq \mathsf{DTISP}[n^{O(\log^{0.5-\alpha} n)}, O(\log^{1.5+\alpha} n)]$.

Our Technique. The proof of Theorem 1 builds on the work of Savitch [9], Nisan [4, 5], and Saks and Zhou [8]. We describe their techniques and then present the main idea of the proof. We start with the standard way of reducing the issue to a question of approximate matrix powering.

The computation of a randomized logspace Turing machine on a fixed input w can be modeled as a Markov chain with some n states, where n is polynomial in $|w|$. The acceptance probability is simply an entry in M^p, where M is the $n \times n$ transition probability matrix of the chain and p is the running time of the machine on input w. Thus, checking the acceptance of w reduces to computing M^p approximately. Since the machine runs in polynomial time, we have $p = n^{O(1)}$.

Savitch's algorithm [9] computes M^p exactly via repeated squaring in a recursive fashion. It implements the recursion in a space efficient way by recomputing the intermediate matrices whenever needed instead of storing them. The algorithm consists of $\log p$ recursive levels using $O(\log n)$ space for each level, resulting in an overall space complexity of $O(\log p \cdot \log n)$ and running time of $n^{O(\log p)}$.

Nisan designed a randomized algorithm for approximate matrix powering (implicit in [4], see [8]). Apart from M and p, the algorithm also takes as input a seed s of length $O(\log p \cdot \log n)$ and outputs an $n \times n$ matrix denoted $\mathcal{N}_s(M, p)$. We call a seed *good* if the matrix $\mathcal{N}_s(M, p)$ approximates M^p well. The main property of the algorithm is that *most* seeds of the appropriate length are good [4]. Given 2-way access to the seed s, the algorithm uses space $O(\log n)$ and runs in time $n^{O(1)}$. A straightforward approach to derandomize the above algorithm is to cycle through all possible seeds and take the average of the output matrices. Such a naive derandomization has space usage $O(\log p \cdot \log n)$ and running time $n^{O(\log p)}$, which are the same as that of Savitch's algorithm. Later, Nisan [5] showed that a good seed can be found deterministically in time $n^{O(1)}$ and space $O(\log p \cdot \log n)$ and thereby proved that $\mathsf{BPL} \subseteq \mathsf{DTISP}[n^{O(1)}, O(\log^2 n)]$.

Notice that, in Nisan's algorithm, there is an imbalance between the processing space of $O(\log n)$ and the seed length of $O(\log p \cdot \log n)$, making the seed length the dominating term in the overall space usage of the naive derandomization. One way to obtain a more space efficient deterministic algorithm is to reduce the seed length before the naive derandomization, probably at the expense of increasing the processing space. A tempting idea for achieving this goal is to combine the algorithms of Nisan and Savitch, by substituting each step of the latter algorithm (which does exact powering) with Nisan's algorithm (which only does approximate powering) and using the *same* seed for all the applications of Nisan's algorithm. The use of same seed at each level of recursion makes sense since Nisan's analysis implies that most seeds are good for not just a single matrix but (via a union bound) simultaneously for any fixed set of matrices. More specifically, consider the following attempt at a space efficient algorithm. Write $r \doteq \log p$ as $r = r_1 \cdot r_2$ and run $\mathcal{N}_s(\cdot, 2^{r_1})$ for r_2 times recursively with the same seed s and without storing the intermediate matrices. That is, compute the sequence of matrices $M_0, M_1, \ldots, M_{r_2}$ in a space efficient recursive fashion where $M_0 = M$ and $M_{i+1} = \mathcal{N}_s(M_i, 2^{r_1})$, and output M_{r_2}. The algorithm uses a single seed of length $O(r_1 \cdot \log n)$. It needs $O(\log n)$ processing space per recursive level leading to a total of $O(r_2 \cdot \log n)$ space for the r_2 levels. We can derandomize it naively and obtain a deterministic algorithm using space $O((r_1 + r_2) \log n)$. By setting $r_1 = r_2 = \sqrt{r}$, we get a deterministic algorithm with space complexity $O(\log^{1.5} n)$.

The problem with the above approach is that we don't know how to guarantee that most seeds are good for *all* the r_2 applications of Nisan's algorithm simultaneously. We cannot use the union bound as above due to dependencies: the matrices $M_1, M_2, \ldots, M_{r_2}$ are not fixed a priori but depend on the seed s. We refer the reader to [8] for a more detailed discussion of this issue.

Saks and Zhou [8] circumvent the dependency issue by using a randomized rounding scheme. We know that for a good seed s, the entries in $\mathcal{N}_s(M, 2^{r_1})$, although dependent on s, all lie very near to values independent of s, namely the corresponding entries in the exact power $M^{2^{r_1}}$. Truncating the entries after t bits, where t is slightly less than the accuracy a good seed s is guaranteed to produce, results in an approximation that is *independent* of the (good) seed s and is only slightly less accurate. This approach works unless an exact value happens to lie very close to a multiple of 2^{-t}. In the latter case, we can add or subtract a small quantity before truncation and obtain a good approximation which is the same for all good seeds s. Saks and Zhou argued that most perturbations from a set of $n^{O(1)}$ possibilities work. Thus, choosing such a perturbation independently at random at every level of recursion circumvents the dependency hurdle. Since each perturbation involves $O(\log n)$ bits, the method induces an additional cost $O(\log n)$ random bits per level of recursion. The processing space remains $O(\log n)$ per level. The algorithm is then derandomized by cycling through all possible seeds s and all perturbations, and averaging the output matrices M_{r_2}. Storing the seed and perturbations requires space $O(r_1 \cdot \log n)$ and $O(r_2 \cdot \log n)$, respectively. Since the processing space of the randomized algorithm is $O(r_2 \cdot \log n)$, the derandomized algorithm runs in space $O((r_1 + r_2) \cdot \log n)$ and therefore in time $n^{O(r_1+r_2)}$. For future reference, we dissect the running time a bit further: There are $n^{O(r_1)}$ choices for the random seed to cycle through, and $n^{O(r_2)}$ choices for the perturbations; it takes $n^{O(r_2)}$ time to process each combination, mainly due to the recomputation of the sequence of matrices in a space efficient recursive fashion. Setting $r_1 = r_2 = \sqrt{r}$, Saks and Zhou proved that $\mathsf{BPL} \subseteq \mathsf{DTISP}[n^{O(\log^{0.5} n)}, O(\log^{1.5} n)]$.

We prove our main result by modifying the Saks-Zhou algorithm. Essentially, instead of picking the seed and the perturbations at random as Saks and Zhou do, we pick them more judiciously in a time and space efficient way. This effectively eliminates the term $n^{O(r_1)}$ in the running time analysis above and leads to our time-space tradeoff. Our algorithm runs in r_2 phases, one for each level in the Saks-Zhou algorithm. In the kth phase, we use [5] to determine a single seed that is good for *each* of the first k levels of recursion, and a brute force search to fix a perturbation that works at the kth level. The critical property of our scheme is that once we have determined a seed that is good for the first k levels, we can forget the seed that was good for the first $k-1$ levels while maintaining the perturbations from the previous phases. Note that for $k = 1$, our approach coincides with Nisan's [5].

More specifically, our algorithm goes as follows. In the first phase, we deterministically find in time $n^{O(1)}$ a good seed and a perturbation that works for the first level, and obtain a good approximation to $M^{2^{r_1}}$. In the second phase, we use the above seed and the perturbation to find in time $n^{O(1)}$ a new seed that is good for the first two levels and obtain approximations to both $M^{2^{r_1}}$ and $M^{2^{2 \cdot r_1}}$. We set up the parameters such that the perturbation from the first phase also works with the new seed. We can now discard the old seed. We then find a perturbation that works for the second level. In general, at the end of the $(k-1)$st phase, we have a good seed and $k-1$ perturbations that work for $k-1$ levels using which we can compute good approximations to $M^{2^{1 \cdot r_1}}$, $M^{2^{2 \cdot r_1}}$, …, $M^{2^{(k-1) \cdot r_1}}$. In the kth phase, we use the seed and the $k-1$ perturbations from the $(k-1)$st phase to find a new seed and an additional perturbation that work for k levels using which we can compute good approximations to $M^{2^{1 \cdot r_1}}$, $M^{2^{2 \cdot r_1}}$, …, $M^{2^{k \cdot r_1}}$. We then discard the seed from the $(k-1)$st phase but keep the perturbations from that phase. Proceeding like this for r_2 phases we compute a good approximation to $M^{2^{r_1 \cdot r_2}} = M^p$. The running time of the kth phase is $n^{O(k)}$ since it needs to compute the k levels of matrices recursively in a space efficient way. Thus the total running time of our algorithm is $n^{O(r_2)}$. Storing the seeds takes $O(r_1 \cdot \log n)$ space. The kth phase uses $O(k \cdot \log n)$ processing space; the space required for storing (old and new) seeds is $O(r_1 \cdot \log n)$. Hence, the total space usage for the derandomized algorithm amounts to $O((r_1 + r_2) \cdot \log n))$. Notice that our algorithm uses the same amount of space as the Saks-Zhou algorithm but avoids a factor of $n^{O(r_1)}$ in the running time. We obtain our time-space tradeoff by varying the choice of r_1 and r_2.

Organization. The paper is organized as follows. Section 2 describes the overall approach and casts previous work in the way we will use it. In particular, we discuss how Nisan's randomized algorithm [4] works. In Section 3, we discuss our time-space tradeoff result in detail and present rigorous correctness proofs. Section 4 ends the paper with some directions for further research.

2 Setup and Previous Work

As is standard in sublinear space bounded computation, we use an off-line Turing machine model to measure space complexity. Thus, the input is considered given on a separate read-only tape and does not count towards the space complexity. Randomness is provided by a one-way read-only tape containing unbiased and uncorrelated random bits. The random bit tape does not count towards the space complexity either.

As is common in prior works, we model the computation of a randomized logspace polynomial-time Turing machine on an input w as a finite state automaton, and analyze the automaton's behavior using its transition matrix. This way, the issue of deterministic simulation reduces to approximate matrix powering. The following definitions and facts will be used for that purpose.

2.1 Notation

Definition 1 *For positive integers d and m, a (d, m)-*automaton *Q is a mapping*

$$Q : \{0, 1, \ldots, d\} \times \{0, 1\}^m \longrightarrow \{0, 1, \ldots, d\},$$

such that, for all $\alpha \in \{0, 1\}^m$, $Q(0, \alpha) = 0$. We say that Q has $d + 1$ states numbered $0, 1, 2, \ldots, d$. State 0 is a special state called the dead state *of Q. We define a mapping*

$$\widehat{Q} : \{0, 1, \ldots, d\} \times (\{0, 1\}^m)^* \longrightarrow \{0, 1, \ldots, d\},$$

recursively as follows. For $i \in \{0, 1, \ldots, d\}$, we define $\widehat{Q}(i, \epsilon) = i$, where ϵ denotes the empty string. Let $i \in \{0, 1, \ldots, d\}$ and $\alpha_1\alpha_2 \ldots \alpha_t \in (\{0, 1\}^m)^+$, where each $\alpha_i \in \{0, 1\}^m$, and t is a positive integer. Then $\widehat{Q}(i, \alpha_1\alpha_2 \ldots \alpha_t) = Q(\widehat{Q}(i, \alpha_1\alpha_2 \ldots \alpha_{t-1}), \alpha_t)$.

Definition 2 *For a $d \times d$ matrix M over the real numbers, define the* norm *of M to be*

$$||M|| = \max_{1 \le i \le d} \left(\sum_{j=1}^{d} |M[i, j]| \right),$$

where $M[i,j]$ is the (i,j) entry of M. The matrix M is said to be sub-stochastic *if $||M|| \le 1$ and $M[i,j] \ge 0$ for any $i,j \in \{1,2,\ldots,d\}$, i.e., M has non-negative entries with each row sum at most 1. Such a matrix is said to be a (d,m)-*matrix *if for any $i,j \in \{1,2,\ldots,d\}$, $2^m M[i,j]$ is an integer.*

Definition 3 *Let Q be a (d,m)-automaton. The transition matrix of Q, denoted by $\mathcal{M}(Q)$, is the (d,m)-matrix M with, for $i,j \in \{1,2,\ldots,d\}$,*

$$M[i,j] = \Pr_{\alpha \in \{0,1\}^m}[Q(i,\alpha) = j].$$

We will exploit the following connection between powers of the transition matrix $M = \mathcal{M}(Q)$ and "powers" of Q: For any nonnegative integer k,

$$M^k[i,j] = \Pr_{s \in (\{0,1\}^m)^k}[\widehat{Q}(i,s) = j], \tag{1}$$

i.e., the (i,j) entry of M^k equals the probability that a random string of length k over the alphabet $\{0,1\}^m$ "moves" Q from i to j.

Note that different (d,m)-automata Q may have the same transition matrix $\mathcal{M}(Q)$. We will need to invert the operator $\mathcal{M}$ in a canonical way.

Definition 4 *Let M be a (d,m)-matrix. The* canonical automaton *corresponding to M, denoted by $\mathcal{Q}(M)$, is a (d,m)-automaton Q defined as follows. Let $i \in \{1,2,\ldots,d\}$ and $\alpha \in \{0,1\}^m$. Let $\underline{\alpha}$ denote the number with binary representation $.\alpha$. If $\underline{\alpha} > \sum_{j=1}^{d} M[i,j]$, then $Q(i,\alpha) = 0$; otherwise, let $k \in \{1,2,\ldots,d\}$ be the least integer such that $\underline{\alpha} \le \sum_{j=1}^{k} M[i,j]$, then we let $Q(i,\alpha) = k$. An automaton Q is called* canonical *if it is the canonical automaton corresponding to some M.*

Note that $\mathcal{M}(\mathcal{Q}(M)) = M$ for any (d,m)-matrix M. By applying $\mathcal{Q}$ once more it follows that, for a canonical Q, $\mathcal{Q}(\mathcal{M}(Q)) = Q$.

2.2 Reduction to Matrix Powering

Given a randomized logspace, polynomial-time, two-sided error Turing machine, and an input w of length n, we need to distinguish between the case where the acceptance probability is at least 2/3, and the case where it is at most 1/3. We do so by approximating the acceptance probability to within 1/6 in a deterministic way within the space and time bounds specified in the tradeoff. The problem reduces to matrix powering in the following standard fashion.

Consider the configurations of the machine on input w, where each configuration consists of the state of the machine, position of the tape heads and the contents of work tape. Let the total number of configurations be d. We can label them $1,2,\ldots,d$. As the machine runs in logspace and polynomial time, $d = n^{\Theta(1)}$. We can assume that the machine reads a fresh random bit b in every step of the computation. The machine then behaves like a $(d,1)$-automaton. More formally, the function Q that maps (i,b) to the state of the machine after reading random bit b in state i, defines a $(d,1)$-automaton. Note that Q as well as its transition matrix $M = \mathcal{M}(Q)$ can be computed in logspace given w. Without loss of generality, the initial configuration of the machine is labeled 1, and the machine has a unique accepting configuration labeled d, which is absorbing (i.e., $M[d,d] = 1$). Suppose the machine runs for p steps, where p is polynomial in n. By (1), the acceptance probability of the machine on input w is given by the $(1,d)$ entry of the matrix M^p, which is the same as the $(1,d)$ entry of M^{2^r} for any integer $r \ge \log p$. In Section 3, we prove the following approximation result.

Theorem 2 *Given a $(d,1)$-matrix M, positive integers r, r_1 and r_2, where $r_1 \geq r_2$ and $r = r_1 r_2 = O(\log d)$, and a positive integer $a = O(\log d)$, there is a deterministic algorithm that computes a matrix M' such that $||M^{2^r} - M'|| < 2^{-a}$. The algorithm uses space $O(r_1 \log d)$ and runs in time $d^{O(r_2)}$.*

Setting $r_1 = \Theta(\log^{0.5+\alpha} d)$, $r_2 = \Theta(\log^{0.5-\alpha} d)$, and $a = 3$ in Theorem 2 yields Theorem 1.

Hence, given a $(d,1)$-matrix M and an integer $r = O(\log d)$, our goal is to approximate M^{2^r} within certain time and space bounds. Instead of working with $(d,1)$-matrices, we consider the problem for (d,m)-matrices. Clearly, any $(d,1)$-matrix is also a (d,m)-matrix for any $m \geq 1$. A specific value for $m = O(\log d)$ will be fixed later.

2.3 Nisan's Pseudorandom Generator Construction

A standard way of computing M^{2^r} (either exact or approximately) is by repeated squaring. A recursive implementation as in Savitch's Theorem yields a procedure that runs in $\Theta(\log^2 d)$ space and $d^{\Theta(\log d)}$ time: Each of the $\Theta(\log d)$ levels of recursion induces an additive term of $\Theta(\log d)$ to the space complexity, and a multiplicative term of $d^{\Theta(1)}$ to the time complexity for storing and (re)computing intermediate results.

Nisan's breakthrough [4, 5] can be viewed as repeated approximate squaring in a way that avoids the need for storing and recomputing intermediate results[1]. The basic ingredient is the following procedure to approximately "square" an underlying automaton using a hash function h.

Definition 5 *Let Q be a (d,m)-automaton, $h : \{0,1\}^m \to \{0,1\}^m$, and $\epsilon > 0$. We define Q_h, also denoted by $\mathcal{N}_h(Q)$, to be the following (d,m)-automaton: For any $i \in \{0,1,\ldots,d\}$ and $\alpha \in \{0,1\}^m$,*

$$Q_h(i,\alpha) = \widehat{Q}(i, \alpha \circ h(\alpha)),$$

where $\circ$ denotes concatenation. We say that h is ϵ-pseudorandom for Q if $||\mathcal{M}(Q_h) - \mathcal{M}(Q)^2|| < \epsilon$.

By (1), $\mathcal{M}(Q)^2$ corresponds to picking $\alpha, \beta \in \{0,1\}^m$ independently at random and running Q on $\alpha \circ \beta$. Similarly, $\mathcal{M}(Q_h)$ corresponds to picking only $\alpha \in \{0,1\}^m$ uniformly at random, and running Q on $\alpha \circ h(\alpha)$. This motivates the use of the term "pseudorandom."

A crucial question is how to find the ϵ-pseudorandom hash functions h we need. Nisan argues that ϵ-pseudorandom hash functions for a (d,m) automaton abound in a universal family of hash functions $\mathcal{H}_m$ from $\{0,1\}^m$ to $\{0,1\}^m$ with $m = \Omega(\log d + \log 1/\epsilon)$. We will fix a universal family [2] such that we can uniformly sample from $\mathcal{H}_m$ in $O(m)$ space, as well as evaluate any $h \in \mathcal{H}_m$ on a given input. An example of such a family is the set of all linear functions over a finite field with 2^m elements.

Lemma 1 (Nisan [4]) *Let Q be a (d,m)-automaton, $\mathcal{H}_m$ a universal family of hash functions, and $\epsilon > 0$.*

$$\Pr_{h \in \mathcal{H}_m}[h \textit{ is not } \epsilon\textit{-pseudorandom for } Q] \leq \frac{d^5}{\epsilon^2 2^m}.$$

Moreover, given Q and ϵ, an exhaustive search for a hash function $h \in \mathcal{H}_m$ that is ϵ-pseudorandom for Q can be performed in space $O(m + \log d)$.

[1] We develop Nisan's approach in a somewhat nonstandard way in order to highlight this point.

Using Lemma 1, we can approximately square a (d, m)-matrix M as follows: Construct the canonical (d, m)-automaton $Q = \mathcal{Q}(M)$, find a hash function $h \in \mathcal{H}_m$ that is ϵ-pseudorandom for Q, construct Q_h, and transform Q_h into its transition matrix $\mathcal{M}(Q_h)$. The result is a (d, m)-matrix that is no more than ϵ away from M^2 in norm. When we apply this procedure r times successively to approximate M^{2^r}, we can, of course, eliminate the intermediate automaton-to-matrix and matrix-to-automaton transformations. We introduce the following simplified notation for the intermediate automata.

Definition 6 *Let Q be a (d, m)-automaton and $h_1, h_2, \ldots, h_r$ be hash functions over $\{0,1\}^m$. Then we define $Q_{h_1,h_2,\ldots,h_k}$, also denoted by $\mathcal{N}_{h_1,h_2,\ldots,h_k}(Q)$, as*

$$Q_{h_1,h_2,\ldots,h_k} = (\ldots((Q_{h_1})_{h_2})\ldots)_{h_k}.$$

Another important question is how the errors accumulate. The accuracy of the final result is governed by Lemma 2, which uses the following definition [4].

Definition 7 *Let Q be a (d, m)-automaton and $h_1, h_2, \ldots, h_r$ be hash functions over $\{0,1\}^m$. We say that the sequence $h_1, h_2, \ldots, h_r$ is* ϵ-pseudorandom *for Q, if*

$$||\mathcal{M}(Q_{h_1,h_2,\ldots,h_r}) - (\mathcal{M}(Q))^{2^r}|| < \epsilon.$$

Lemma 2 (Nisan [4]) *Let Q be a (d, m)-automaton, and $\epsilon > 0$. Let $h_1, h_2, \ldots, h_r$ be hash functions over $\{0,1\}^m$ such that, for each $1 \le i \le r$, h_i is ϵ-pseudorandom for $Q_{h_1,h_2,\ldots,h_{i-1}}$. Then the sequence $h_1, h_2, \ldots, h_r$ is $2^r\epsilon$-pseudorandom for Q.*

It follows from Lemmata 1 and 2 that a choice of $m = O(\log d)$ is sufficient to obtain a good enough approximation to M^{2^r} for our purposes.

A straightforward recursive implementation of Nisan's algorithm uses roughly the same amount of resources as the trivial recursive squaring approach, namely $\Theta(\log^2 d)$ space and $d^{\Theta(\log d)}$ time. However, the crucial property in Proposition 1 below allowed Nisan to do better. The following definition plays a central role.

Definition 8 *Let Q be a (d, m) automaton, and $h_1, h_2, \ldots, h_r$ be hash functions over $\{0,1\}^m$. Recursively define $G_{h_1,h_2,\ldots,h_r}$ as a mapping from $\{0,1\}^m$ to $(\{0,1\}^m)^{2^r}$ by*

$$\begin{aligned} G(\alpha) &= \alpha \\ G_{h_1,h_2,\ldots,h_i}(\alpha) &= G_{h_1,h_2,\ldots,h_{i-1}}(\alpha) \circ G_{h_1,h_2,\ldots,h_{i-1}}(h_i(\alpha)). \end{aligned}$$

Note that, given hash functions $h_1, h_2, \ldots, h_r \in \mathcal{H}_m$, a bit index, and $\alpha \in \{0,1\}^m$, any bit of $G_{h_1,h_2,\ldots,h_r}(\alpha)$ can be computed in $O(m + \log r)$ space.

Proposition 1 *Let Q be a (d, m)-automaton and $h_1, h_2, \ldots, h_r$ be hash functions over $\{0,1\}^m$. Then for any $i \in \{0, 1, \ldots, d\}$ and $\alpha \in \{0,1\}^m$,*

$$Q_{h_1,h_2,\ldots,h_r}(i, \alpha) = \widehat{Q}(i, G_{h_1,h_2,\ldots,h_r}(\alpha)).$$

Although Proposition 1 looks innocuous, it has far reaching consequences. First, it shows that we obtain our approximations to the probabilities defined by (1) for $k = 2^r$ by subsampling, i.e., by using a pseudorandom generator. The function mapping $(h_1, h_2, \ldots, h_r, \alpha)$ to $G_{h_1,h_2,\ldots,h_r}(\alpha)$ defines a pseudorandom generator stretching $O(rm)$ random bits to $2^r m$ pseudorandom bits that "fool" the automaton Q. We will not explicitly use this fact but it was Nisan's original motivation [4].

Second, and more important for us, Proposition 1 provides a shortcut for storing the intermediate results in the recursive computation of $Q_{h_1,h_2,\ldots,h_r}$. Given $h_1, h_2, \ldots, h_r \in \mathcal{H}_m$, $i \in \{0, 1, \ldots, d\}$, and $\alpha \in \{0,1\}^m$, we can now compute $Q_{h_1,h_2,\ldots,h_r}(i, \alpha)$ in space $O(m + r + \log d)$ instead of $O(r(m + \log d))$. This allows us to deterministically find pseudorandom hash functions h_i for every level of recursion as well as compute $\mathcal{M}(Q_{h_1,h_2,\ldots,h_r})$ from $h_1, h_2, \ldots, h_r$ using only $O(m + r + \log d)$ work space, and therefore in time $2^{O(m+r+\log d)}$. The only additional space needed is $O(rm)$ bits to store the hash functions. This is how Nisan established his result that $\mathsf{BPL} \subseteq \mathsf{DTISP}[n^{O(1)}, O(\log^2 n)]$ [5].

2.4 Saks-Zhou Rounding

The only reason why Nisan's algorithm needs more than logarithmic space is to store the $r = O(\log d)$ hash functions of $O(m) = O(\log d)$ bits each. A natural approach to save space then, is to use the same hash functions at multiple levels of recursion.

Given any *fixed* collection of (d, m)-automata $Q_1, Q_2, \ldots, Q_r$, we can efficiently find a hash function h that is pseudorandom for all of them. This follows from Lemma 1.

Corollary 1 *For any (d, m)-automata $Q_1, Q_2, \ldots, Q_r$,*

$$\Pr_{h \in \mathcal{H}_m}[h \text{ is not } \epsilon\text{-pseudorandom for at least one of } Q_1, Q_2, \ldots, Q_r] \leq \frac{rd^5}{\epsilon^2 2^m}.$$

Moreover, given $Q_1, Q_2, \ldots, Q_r$ and ϵ, an exhaustive search for a hash function $h \in \mathcal{H}_m$ that is ϵ-pseudorandom for all of $Q_1, Q_2, \ldots, Q_r$ can be performed in space $O(m + \log d + \log r)$.

However, the sequence $Q_1, Q_2, \ldots, Q_r$ in Nisan's algorithm is not fixed but depends on the choice of the hash functions: $Q_{i+1} = (Q_i)_{h_i}$. If Lemma 1 guaranteed the existence of a fixed automaton Q^* such that $Q_h = Q^*$ for most choices of h, then we could apply Corollary 1 to the sequence $Q_1, Q_2, \ldots, Q_r$ with $Q_1 = Q$ and $Q_{i+1} = Q_i^*$, and safely use the same hash function h at every level of recursion. Unfortunately, Lemma 1 only tells us that Q_h is "close" to some fixed Q^* for most h and not that it coincides with Q^* for most h.

Saks and Zhou [8] devised a randomized rounding scheme to make the automata at every level of recursion independent of the hash functions at the previous levels (for most choices of these hash functions), namely by randomly perturbing and then truncating the entries of the transition matrices. We will review the details below and see that the rounding requires $O(\log d)$ bits to be stored for a $d \times d$ matrix.

We point out one unfortunate side-effect of the rounding scheme: the breakdown of the analogue of Proposition 1. We can no longer circumvent the recursion because we need to act on the matrices, and therefore have to transform between automata and matrices at every level of recursion. As a consequence, the processing space increases from $O(m + r + \log d)$ to $O(r(m + \log d))$ again.

The storage space consists of two components: $O(m)$ bits for the hash function h, and $O(r \log d)$ bits for all the perturbations. In order to balance the two components of the space complexity, Saks and Zhou considered a hybrid algorithm that uses r_1 hash functions $\vec{h} = (h_1, h_2, \ldots, h_{r_1})$ cyclically r_2 times, where $r_1 r_2 = r$, and applies the randomized rounding between cycles. The processing space becomes $O(r_2(m + \log d))$ and the storage space $O(r_1 m + r_2 \log d)$. By setting $r_1 = r_2 = \sqrt{r}$, Saks and Zhou obtained their result that $\mathsf{BPL} \subseteq \mathsf{DTISP}[n^{O(\log^{0.5} n)}, O(\log^{1.5} n)]$ [8].

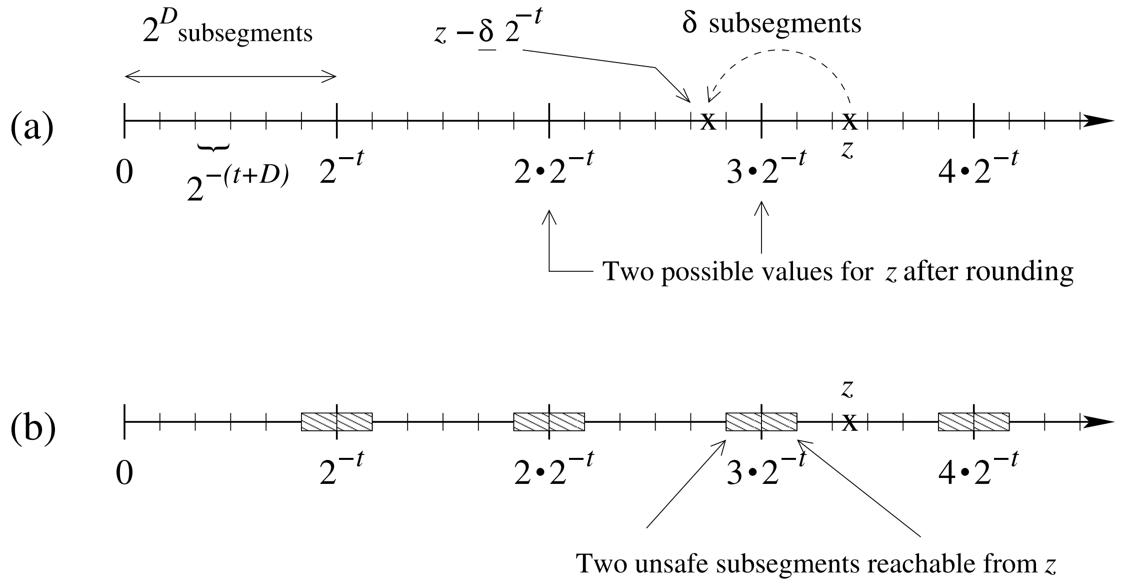

Figure 2: Saks-Zhou Rounding

We now discuss the Saks-Zhou rounding scheme in more detail for the hybrid setting, and adapt it to our needs.

Definition 9 *Let t, D be positive integers, and $\delta \in \{0,1\}^D$. We define the rounding operator $\mathcal{R}_{\delta,t}$ as the function mapping $z \in [0,1]$ to $\lfloor \max(z - \underline{\delta}2^{-t}, 0) \rfloor_t$, where $\underline{\delta}$ denotes the number with binary representation $.\delta$, and $\lfloor z \rfloor_t = \lfloor 2^t z \rfloor 2^{-t}$. The operator extends to real-valued matrices with each entry in $[0,1]$ by entry-wise application.*

The effect of $\mathcal{R}_{\delta,t}$ is to perturb the binary expansion of every number after the t-th bit by δ (obtaining a smaller number but never going negative), and then truncate it after the t-th bit. Figure 2(a) illustrates the rounding operation. We divide the interval $[0,1]$ into 2^t segments of length 2^{-t} each and subdivide each segment into 2^D subsegments of length $2^{-(t+D)}$ each. The number z lies in one of the subsegments. Treating δ as a D-bit number, we see that subtracting $\underline{\delta}2^{-t}$ moves z by δ subsegments to the left and the truncation moves it further to the nearest segment boundary (a multiple of 2^{-t}) on the left. Notice that there are just two possible values for $\mathcal{R}_{\delta,t}(z)$ when δ ranges over $\{0,1\}^D$. We note that if M is a (d,m)-matrix and $t \leq m$, then $\mathcal{R}_{\delta,t}(M)$ also is a (d,m)-matrix.

Definition 10 *Let M be a $d \times d$-matrix over $[0,1]$, and t, D, r_1, k be positive integers. Let $\vec{\delta} = (\delta_1, \delta_2, \ldots, \delta_k) \in (\{0,1\}^D)^k$. We define the* rounded sequence $\mathrm{R}(M, \vec{\delta})$ *with parameters t, r_1, and k as the following sequence of matrices:*

$$N_0, \widetilde{N}_1, N_1, \widetilde{N}_2, N_2, \ldots, \widetilde{N}_k, N_k$$

where, $N_0 = M$ and for $1 \leq i \leq k$,

$$\widetilde{N}_i = N_{i-1}^{2^{r_1}}, \quad N_i = \mathcal{R}_{\delta_i,t}(\widetilde{N}_i).$$

Note that if M is a (d,m)-matrix and $t \leq m$, then every matrix N_i is a (d,m)-matrix. Their canonical automata $\mathcal{Q}(N_i)$ are the "fixed" automata to which Corollary 1 will be applied.

Rounding does not affect the final result by much.

Lemma 3 (Saks-Zhou [8]) *Let M be a $d \times d$-matrix over $[0,1]$, and t, D, r_1, k be positive integers. Let $\vec{\delta} = (\delta_1, \delta_2, \ldots, \delta_k) \in (\{0,1\}^D)^k$. Then, the matrix N_k defined in the rounded sequence $\mathrm{R}(M, \vec{\delta})$ with parameters t, r_1, and k satisfies*

$$||N_k - M^{2^{r_1 k}}|| < 2^{-t + r_1 k + \log d + 1}.$$

The crucial property of the rounding operator is that for any $z \in [0,1]$, unless δ is exceptional, $\mathcal{R}_{\delta,t}$ maps any number that is about $2^{-(t+D)}$ close to z to the same value as z. Saks and Zhou formalized this property using the notion of safety. We will distinguish between two levels of safety. We define them right away for matrices.

Definition 11 *Let M be a $d \times d$-matrix over $[0,1]$, t, D positive integers, and $\delta \in \{0,1\}^D$. We denote δ as* t-safe *for M if for any $d \times d$-matrix Z over $[0,1]$ satisfying $||Z - M|| < 2^{-(t+D)}$,*

$$\mathcal{R}_{\delta,t}(Z) = \mathcal{R}_{\delta,t}(M). \tag{2}$$

If (2) holds whenever Z satisfies $||Z - M|| < 2^{-(t+D+1)}$, we call δ t-weakly-safe *for M.*

By definition, t-safeness implies t-weakly-safeness. We will use the following stronger relationship between the two levels of safety.

Lemma 4 *Let M and N be $d \times d$-matrices over $[0,1]$, t, D positive integers, and $\delta \in \{0,1\}^D$. If δ is t-safe for M and $||M - N|| \leq 2^{-(t+D+1)}$, then δ is t-weakly-safe for N.*

For any $z \in [0,1]$, there are at most two values of $\delta \in \{0,1\}^D$ that are not t-safe for z (viewed as a 1×1 matrix), and these values are easy to find. This observation can be seen from Figure 2 as follows. Fix a number $z \in [0,1]$. Any number z' with $|z - z'| < 2^{-(t+D)}$ lies either in the same subsegment as z or in one of the two adjacent subsegments. Subtracting $\underline{\delta} 2^{-t}$ moves both z and z' by δ subsegments to the left. The truncation that follows will map both points to the same multiple of 2^{-t} unless z and z' lie in different subsegments that are separated by an intersegment boundary after the shift by $\underline{\delta} 2^{-t}$. Observe that the latter is only possible if $z - \underline{\delta} 2^{-t}$ lies in a subsegment that is to the left or to the right of an intersegment boundary (the shaded subsegments in Figure 2(b)). In other words, a δ that moves z to an unshaded subsegment is safe for z. Since the subtraction operation can move z at most one segment, there are at most two shaded subsegments that are reachable from z (as shown in Figure 2(b)). It follows that there are at most two values of δ that move z to a shaded subsegment and hence that all but two values of $\delta \in \{0,1\}^D$ are safe for z. Generalizing the observation, we have that for any $d \times d$ matrix, at most $2d^2$ values of $\delta \in \{0,1\}^D$ are not t-safe. We record the observation as the following lemma.

Lemma 5 *Let M be a $d \times d$-matrix over $[0,1]$, and t, D positive integers. For all but at most $2d^2$ values of $\delta \in \{0,1\}^D$, the operator $\mathcal{R}_{\delta,t}$ is t-safe for M. Moreover, given M, t, and D, an exhaustive search for a $\delta \in \{0,1\}^D$ that is t-safe for M can be performed in space $O(D + \log d + \log t)$.*

Definition 12 *Let M be a (d,m)-matrix, and t, D, r_1, k positive integers with $t \le m$. Let $\vec{\delta} = (\delta_1, \delta_2, \dots \delta_k) \in (\{0,1\}^D)^k$. Let $\vec{h} = (h_1, h_2, \dots, h_{r_1})$ be a sequence of hash functions from $\mathcal{H}_m$. We define the* Saks-Zhou sequence $\mathrm{SZ}(M, \vec{h}, \vec{\delta})$ *with parameters t, r_1, and k as the following sequence of matrices and automata:*

$$M_0, Q_0, \widetilde{Q}_1, \widetilde{M}_1, M_1, Q_1, \widetilde{Q}_2, \widetilde{M}_2, M_2, \dots, Q_{r_2-1}, \widetilde{Q}_k, \widetilde{M}_k, M_k$$

where, $M_0 = M$, and for $1 \le i \le k$,

$$Q_{i-1} = \mathcal{Q}(M_{i-1}), \qquad \widetilde{Q}_i = \mathcal{N}_{h_1,h_2,\dots,h_{r_1}}(Q_{i-1}), \qquad \widetilde{M}_i = \mathcal{M}(\widetilde{Q}_i), \qquad M_i = \mathcal{R}_{\delta_i,t}(\widetilde{M}_i)$$

See Figure 3 for a schema of the construction of the Saks-Zhou sequence $\mathrm{SZ}(M, \vec{h}, \vec{\delta})$, the rounded sequence $\mathrm{R}(M, \vec{\delta})$, and their intended relationships (for $k = r_2$). In the figure, the operator $\mathcal{P}$ denotes raising a matrix to the power 2^{r_1}.

Saks and Zhou defined $\vec{\delta}$ to be t-safe for the sequence $\mathrm{R}(M, \vec{\delta})$ if δ_i is t-safe for $\widetilde{N}_i$, for each i. As can be seen from Lemma 5, for a large enough choice of $D = O(\log d)$, a random $\vec{\delta}$ will be safe for $\mathrm{R}(M, \vec{\delta})$. When $\vec{\delta}$ is safe for $\mathrm{R}(M, \vec{\delta})$, one can argue that for most $\vec{h}$, $M_i = N_i$ for $0 \le i \le r_2$, where M_i and N_i are matrices defined in $\mathrm{SZ}(M, \vec{h}, \vec{\delta})$ and $\mathrm{R}(M, \vec{\delta})$, respectively. In particular, $M_{r_2} = N_{r_2}$ is a good approximation to M^{2^r}.

The fact that M_i and N_i coincide for every $1 \le i \le r_2$ can be seen as follows. Let $P_0, P_1, \dots, P_{r_2}$ be the canonical automata corresponding to the matrices $N_0, N_1, \dots, N_{r_2}$, respectively. If we choose $\vec{h}$ at random from $\mathcal{H}_m^{r_1} = \mathcal{H}_m \times \mathcal{H}_m \times \cdots \times \mathcal{H}_m$, then, with high probability, $\vec{h}$ is ϵ-pseudorandom for all the automata $P_0, P_1, \dots, P_{r_2}$, for suitably small ϵ. This is guaranteed by Corollary 1 because these automata are defined using only M and $\vec{\delta}$, and in particular they do not depend on $\vec{h}$. By definition, $M_0 = N_0 = M$ and thus $Q_0 = P_0$. Therefore $\vec{h}$ is ϵ-pseudorandom for Q_0. It follows that $\widetilde{M}_1$ is close to $\widetilde{N}_1$. Then $M_1 = N_1$ by δ_1 being safe for $\widetilde{N}_1$. Continuing this way for r_2 steps, we get $M_{r_2} = N_{r_2}$.

Using these properties, the Saks-Zhou [8] algorithm approximates M^{2^r} as the average of the matrices M_{r_2} over all $\vec{\delta}$ and $\vec{h}$.

3 Our Algorithm

In this section, we prove Theorem 2. We first present an informal description of the proof and then describe the algorithm claimed in Theorem 2 formally. Proof of correctness and time/space analysis of the algorithm are given subsequently.

3.1 An Informal Description

Given a (d,m)-matrix M and integers r_1, r_2 with $r = r_1 r_2$ and $r_1 \ge r_2$, the Saks-Zhou algorithm described in Section 2.4 approximates M^{2^r} in space $O((r_1 + r_2)\log d)$ but uses time $d^{O(r_1+r_2)}$. The factor $d^{O(r_1)}$ comes from trying all possible sequences of r_1 hash functions. Our goal is to avoid this exhaustive search and approximate M^{2^r} using space $O((r_1 + r_2)\log d)$ and time $d^{O(r_2)}$.

For any sequence $\vec{\delta} = (\delta_1, \delta_2, \dots, \delta_{r_2}) \in (\{0,1\}^D)^{r_2}$, the matrix N_{r_2} defined in the rounded sequence $\mathrm{R}(M, \vec{\delta})$ is a close approximation to M^{2^r}. We want to find a $\vec{\delta}$ and a sequence of hash functions $\vec{h} = (h_1, h_2, \dots, h_{r_1})$ such that we can efficiently compute the matrices $N_0, N_1, \dots, N_{r_2}$ defined in the sequence $\mathrm{R}(M, \vec{\delta})$ by computing the corresponding matrices $M_0, M_1, \dots, M_{r_2}$ in the Saks-Zhou sequence $\mathrm{SZ}(M, \vec{h}, \vec{\delta})$. See Figure 3 for the correspondences.

$$
\begin{tikzcd}
N_0 \arrow[r, "\mathcal{P}"] \arrow[d, equal] & \widetilde{N}_1 \arrow[r, "\mathcal{R}_{\delta_1,t}"] \arrow[d, phantom, "\wr"] & N_1 \arrow[r, "\mathcal{P}"] \arrow[d, equal] & \widetilde{N}_2 \arrow[r, "\mathcal{R}_{\delta_2,t}"] \arrow[d, phantom, "\wr"] & N_2 \arrow[r] \arrow[d, equal] & \cdots\cdots \arrow[r] & N_{r_2-1} \arrow[r, "\mathcal{P}"] \arrow[d, equal] & \widetilde{N}_{r_2} \arrow[r, "\mathcal{R}_{\delta_{r_2},t}"] \arrow[d, phantom, "\wr"] & N_{r_2} \arrow[d, equal] \\
M_0 \arrow[d, "\mathcal{Q}"'] & \widetilde{M}_1 \arrow[r, "\mathcal{R}_{\delta_1,t}"] & M_1 \arrow[d, "\mathcal{Q}"'] & \widetilde{M}_2 \arrow[r, "\mathcal{R}_{\delta_2,t}"] & M_2 \arrow[d, "\mathcal{Q}"'] & \cdots\cdots \arrow[r] & M_{r_2-1} \arrow[d, "\mathcal{Q}"'] & \widetilde{M}_{r_2} \arrow[r, "\mathcal{R}_{\delta_{r_2},t}"] & M_{r_2} \\
Q_0 \arrow[r, "\mathcal{N}_{\vec{h}}"] & \widetilde{Q}_1 \arrow[u, "\mathcal{M}"] & Q_1 \arrow[r, "\mathcal{N}_{\vec{h}}"] & \widetilde{Q}_2 \arrow[u, "\mathcal{M}"] & Q_2 \arrow[r] & \cdots\cdots & Q_{r_2-1} \arrow[r, "\mathcal{N}_{\vec{h}}"] & \widetilde{Q}_{r_2} \arrow[u, "\mathcal{M}"] &
\end{tikzcd}
$$

Figure 3: Rounded and Saks-Zhou sequences

A natural idea is to find the hash functions h_i, one by one, based on Lemma 1. Thus, for the automaton $Q_0 = \mathcal{Q}(M)$, we can find a sequence of hash functions, $\vec{h} = (h_1, h_2, \ldots, h_{r_1})$, one by one, such that $\widetilde{M}_1 = \mathcal{M}(\widetilde{Q}_1) = \mathcal{M}(\mathcal{N}_{\vec{h}}(Q_0))$ is very close to $\widetilde{N}_1 = M^{2^{r_1}}$. This step is essentially Nisan's algorithm for raising M to the power 2^{r_1}. Each step to find the next h_i is carried out by an exhaustive search over $\mathcal{H}_m$, and thus can be done in polynomial time and space $O(r_1 m)$.

Then we can deterministically find a $\delta_1 \in \{0,1\}^D$ which is safe for $\widetilde{M}_1$. By Lemma 5, we can guarantee the existence of δ_1 by setting $D = O(\log d)$. Since $\widetilde{M}_1$ is very close to $\widetilde{N}_1$, δ_1 will in fact be weakly-safe for $\widetilde{N}_1$, even though we never really compute $\widetilde{N}_1$ exactly. With $\vec{h}$ already written, the task of finding δ_1 is in logspace (and thus in polynomial time). Since $\widetilde{M}_1$ and $\widetilde{N}_1$ are close, the rounding operator $\mathcal{R}_{\delta_1,t}$ on $\widetilde{M}_1$ and $\widetilde{N}_1$ yields identical matrices $M_1 = N_1$. At this point, we have computed $\vec{h} = \vec{h}^{(1)}$ and δ_1, which can be viewed as *a succinct representation* of N_1, which is an approximation to $M^{2^{r_1}}$.

Now we would like to extend this to N_2. Let $Q_1 = \mathcal{Q}(M_1) = \mathcal{Q}(N_1)$. By Corollary 1, it is true that most sequences of r_1 hash functions $\vec{h'}$ will be simultaneously pseudorandom for Q_0 and Q_1. If we had chosen our $\vec{h}$ randomly, then with high probability it would be pseudorandom for both Q_0 and Q_1. But we found our $\vec{h}$ deterministically; there is no guarantee that $\vec{h}$ will be pseudorandom for Q_1.

Our idea is to find a *new* sequence of hash functions $\vec{h'}$. With the *succinct representation* $\vec{h}$ and δ_1 in hand, we can compute Q_1 and find a new sequence of r_1 hash functions $\vec{h'}$ which is pseudorandom for both Q_0 and Q_1. Of course, due to the space bound we cannot keep the old $\vec{h}$ around. So, at this point we want to discard $\vec{h}$ and replace it by $\vec{h'}$.

However, after discarding the old $\vec{h}$, we no longer have access to $\widetilde{M}_1 = \mathcal{M}(\mathcal{N}_{\vec{h}}(Q_0))$; and it is from $\widetilde{M}_1$ and δ_1 that we defined Q_1, for which the new $\vec{h'}$ was found to be pseudorandom. If at this point the new $\widetilde{M}'_1 = \mathcal{M}(\mathcal{N}_{\vec{h'}}(Q_0))$ were to lead to an automaton different from Q_1, we would have made no progress.

However, note that δ_1 is weakly-safe for $\widetilde{N}_1$. While $\widetilde{M}'_1 = \mathcal{M}(\mathcal{N}_{\vec{h'}}(Q_0))$ may be different from $\widetilde{M}_1 = \mathcal{M}(\mathcal{N}_{\vec{h}}(Q_0))$, both matrices are very close. In fact, the new $\widetilde{M}'_1 = \mathcal{M}(\mathcal{N}_{\vec{h'}}(Q_0))$ will be close to $\widetilde{N}_1$. It follows that when we apply the rounding operator $\mathcal{R}_{\delta_1,t}$, the matrices are the same again, namely $M_1 = N_1$. See the left-hand side of Figure 4. Therefore, we do get back to the same automaton $Q_1 = \mathcal{Q}(M_1) = \mathcal{Q}(N_1)$.

We then proceed to find a δ_2 which is weakly-safe for $\widetilde{N}_2 = N_1^{2^{r_1}}$. Note that $\widetilde{N}_2$ is never computed exactly but we choose δ_2 as being safe for the matrix $\widetilde{M}'_2 = \mathcal{M}(\mathcal{N}_{\vec{h'}}(Q_1))$, which is very close to the matrix $\widetilde{N}_2$. Then applying $\mathcal{R}_{\delta_2,t}$ to either matrix yields N_2. We have arrived at a

$$
\begin{tikzcd}
Q_0 \arrow[r, "\mathcal{N}_{\vec{h}}"] & \widetilde{Q}_1 \arrow[r, "\mathcal{M}"] & \widetilde{M}_1 \arrow[ddr, "\mathcal{R}_{\delta_1,t}"] & Q_1 \arrow[r, "\mathcal{N}_{\vec{h}}"] & \widetilde{Q}_2 \arrow[r, "\mathcal{M}"] & \widetilde{M}_2 \arrow[ddr, "\mathcal{R}_{\delta_k,t}"] & & \cdots\cdots \\
 & & \wr & & & \wr & & \\
M_0 \arrow[uu, "\mathcal{Q}"] \arrow[dd, "\mathcal{Q}"'] \arrow[rr, "\mathcal{P}"] & & \widetilde{N}_1 \arrow[r, "\mathcal{R}_{\delta_1,t}"] & M_1 = N_1 \arrow[uu, "\mathcal{Q}"] \arrow[dd, "\mathcal{Q}"'] \arrow[rr, "\mathcal{P}"] & & \widetilde{N}_2 \arrow[r, "\mathcal{R}_{\delta_k,t}"] & M_2 = N_2 & \cdots\cdots \\
 & & \wr & & & \wr & & \\
Q_0 \arrow[r, "\mathcal{N}_{\vec{h'}}"'] & \widetilde{Q}'_1 \arrow[r, "\mathcal{M}"'] & \widetilde{M}'_1 \arrow[uur, "\mathcal{R}_{\delta_1,t}"'] & Q_1 \arrow[r, "\mathcal{N}_{\vec{h'}}"'] & \widetilde{Q}'_2 \arrow[r, "\mathcal{M}"'] & \widetilde{M}'_2 \arrow[uur, "\mathcal{R}_{\delta_k,t}"'] & & \cdots\cdots
\end{tikzcd}
$$

$$
\begin{tikzcd}
Q_{k-1} \arrow[r, "\mathcal{N}_{\vec{h}}"] & \widetilde{Q}_k \arrow[r, "\mathcal{M}"] & \widetilde{M}_k \arrow[ddr, "\mathcal{R}_{\delta_k,t}"] & & & & \\
 & & \wr & & & & \\
M_{k-1} = N_{k-1} \arrow[uu, "\mathcal{Q}"] \arrow[dd, "\mathcal{Q}"'] \arrow[rr, "\mathcal{P}"] & & \widetilde{N}_k \arrow[r, "\mathcal{R}_{\delta_k,t}"] & M_k = N_k \arrow[dd, "\mathcal{Q}"] \arrow[rr, "\mathcal{P}"] & & \widetilde{N}_{k+1} \arrow[r, "\mathcal{R}_{\delta_{k+1},t}"] & M_{k+1} = N_{k+1} \\
 & & \wr & & & \wr & \\
Q_{k-1} \arrow[r, "\mathcal{N}_{\vec{h'}}"'] & \widetilde{Q}'_k \arrow[r, "\mathcal{M}"'] & \widetilde{M}'_k \arrow[uur, "\mathcal{R}_{\delta_k,t}"'] & Q_k \arrow[r, "\mathcal{N}_{\vec{h'}}"'] & \widetilde{Q}'_{k+1} \arrow[r, "\mathcal{M}"'] & \widetilde{M}'_{k+1} \arrow[uur, "\mathcal{R}_{\delta_{k+1},t}"'] &
\end{tikzcd}
$$

Figure 4: Finding new sequence of hash functions

succinct representation for N_1 and N_2, namely $\vec{h'} = \vec{h}^{(2)}$ and (δ_1, δ_2).

Generalizing, we accomplish the computation of our final approximation N_{r_2} to M^{2^r} in r_2 stages as follows (see Figure 4). At the end of stage k, we will have computed a sequence of r_1 hash functions $\vec{h}$ and a sequence $\vec{\delta} = (\delta_1, \delta_2, \ldots, \delta_k) \in (\{0,1\}^D)^k$. They constitute a *succinct representation* of the matrices $N_0, N_1, \ldots, N_k$ of the rounded sequence; we can compute these matrices N_i as the corresponding matrices M_i in the Saks-Zhou sequence $\mathrm{SZ}(Q, \vec{h}, \vec{\delta})$. Moreover, each δ_i in $\vec{\delta}$ is weakly-safe for $\widetilde{N}_i$ in $\mathrm{R}(M, \vec{\delta})$, $1 \le i \le k$.

In the $(k+1)$st stage, we use $\vec{h}$ and $\vec{\delta}$ to reconstruct the matrices $N_0 = M_0$, $N_1 = M_1$, $\ldots$, $N_k = M_k$. We then compute a new sequence of r_1 hash functions $\vec{h'}$, one by one, which is simultaneously pseudorandom for $Q_0 = \mathcal{Q}(N_0), Q_1 = \mathcal{Q}(N_1), \ldots, Q_k = \mathcal{Q}(N_k)$. Thus, while both $\vec{h}$ and $\vec{h'}$ are pseudorandom for $Q_0, Q_1, \ldots, Q_{k-1}$, $\vec{h'}$ is also pseudorandom for Q_k. Now each $\widetilde{M}'_1 = \mathcal{M}(\mathcal{N}_{\vec{h'}}(Q_0)), \widetilde{M}'_2 = \mathcal{M}(\mathcal{N}_{\vec{h'}}(Q_1)), \ldots, \widetilde{M}'_k = \mathcal{M}(\mathcal{N}_{\vec{h'}}(Q_{k-1}))$ are sufficiently close to $\widetilde{N}_1, \widetilde{N}_2, \ldots, \widetilde{N}_k$, respectively, that when we apply the rounding operator $\mathcal{R}$ based on $\vec{\delta}$ and t, we get the matrices $N_1, \ldots, N_k$ of the rounded sequence $\mathrm{R}(M, \vec{\delta})$ back. This is because δ_i is weakly safe for $\widetilde{N}_i$, $1 \le i \le k$. In addition, we have obtained $\widetilde{M}'_{k+1} = \mathcal{M}(\mathcal{N}_{\vec{h'}}(Q_k))$, which is close to $\widetilde{N}_{k+1}$. We then find a $\delta_{k+1} \in \{0,1\}^D$ which is guaranteed to be weakly-safe for the matrix $\widetilde{N}_{k+1}$. Again, even though we do not really compute this $\widetilde{N}_{k+1}$ exactly, we find δ_{k+1} as being safe for its close approximation $\widetilde{M}'_{k+1}$. At this point, $\vec{h'}$ and $\vec{\delta'} = (\delta_1, \delta_2, \ldots, \delta_k, \delta_{k+1})$ constitute a succinct representation of the matrices $N_0, N_1, \ldots, N_k, N_{k+1}$ in $\mathrm{R}(M, \vec{\delta'})$, as shown in Figure 4. Now, we discard the old $\vec{h}$ and replace it with the newly found $\vec{h'}$, and we append δ_{k+1} to $\vec{\delta}$.

At the end of r_2 stages, we have found $\vec{h}$ and $\vec{\delta} = (\delta_1, \delta_2, \ldots, \delta_{r_2})$ such that the rounded sequence $\mathrm{R}(M, \vec{\delta})$ and the Saks-Zhou sequence $\mathrm{SZ}(M, \vec{h}, \vec{\delta})$ agree as in Figure 3.

In terms of time and space complexity, the most time consuming task is to recursively reconstruct the automata and matrices in the Saks-Zhou sequences with a maximum of r_2 levels of recursion, each level involving a polynomial time computation. The most space consuming task is to store the r_1 hash functions $\vec{h}$.

3.2 The Algorithm

In this section, we present the algorithm claimed in Theorem 2. Our input is a $(d,1)$-matrix M, integers $r_1 \geq r_2$, $r = r_1 r_2$, and an accuracy parameter a. We first compute the following parameters used throughout the algorithm. We set $t = \lceil a + r + \log d \rceil + 1$, $D = \lceil 2\log d \rceil + 1$, $\epsilon = 2^{-(t+D+r_1+1)}$, $m = \lceil 5\log d + \log r_2 + 2\log\frac{1}{\epsilon}\rceil + 1$. Note that, since we assume $a = O(\log d)$ and $r = r_1 r_2 = O(\log d)$, the parameters $t, D, \log\frac{1}{\epsilon}$ and m are all $O(\log d)$. We will treat M as a (d,m)-matrix. We present a procedure MatrixPower that outputs a matrix M' satisfying $||M' - M^{2^r}|| < 2^{-a}$. We assume that the input matrix M and various parameters $r, r_1, r_2, t, D, \epsilon$ and m are stored globally and are accessible throughout the execution of MatrixPower.

Procedure MatrixPower
Input: Matrix M and the parameters.
Output: A matrix M' satisfying $||M' - M^{2^r}|| < 2^{-a}$.

1. For $k \longleftarrow 1$ to r_2 do

   (a) For $i \longleftarrow 1$ to r_1 do
   Using exhaustive search find $h'_i \in \mathcal{H}_m$ such that, h'_i is ϵ-pseudorandom for the k automata $\mathcal{N}_{h'_1,h'_2,\ldots,h'_{i-1}}(Q_0)$, $\mathcal{N}_{h'_1,h'_2,\ldots,h'_{i-1}}(Q_1)$, $\ldots$, $\mathcal{N}_{h'_1,h'_2,\ldots,h'_{i-1}}(Q_{k-1})$, where $Q_0, Q_1, \ldots, Q_{k-1}$ are as defined in the sequence $\mathrm{SZ}(M, \vec{h}, (\delta_1, \delta_1, \ldots \delta_{k-1}))$.

   (b) Using exhaustive search, find a $\delta_k \in \{0,1\}^D$ that is t-safe for $\mathcal{M}(\mathcal{N}_{h'_1,h'_2,\ldots,h'_{r_1}}(Q_{k-1}))$.

   (c) $\vec{h} \longleftarrow (h'_1, h'_2, \ldots, h'_{r_1})$.

2. Output M_{r_2}, where M_{r_2} is the matrix as defined in $\mathrm{SZ}(M, \vec{h}, (\delta_1, \delta_2, \ldots, \delta_{r_2}))$.

3.3 Proof of Correctness

Our goal is to argue that the algorithm produces a good approximation to M^{2^r}. Let N_0, $\widetilde{N}_1$, N_1, $\ldots$, $\widetilde{N}_{r_2}$, N_{r_2} be the matrices defined in $\mathrm{R}(M, (\delta_1, \delta_2, \ldots, \delta_{r_2}))$. Observe that these matrices are good approximations to the exact powers of M (i.e, $\widetilde{N}_k \simeq N_k \simeq M^{2^{k \cdot r_1}}$); in particular, $N_{r_2} \simeq M^{2^{r_1 \cdot r_2}} = M^{2^r}$. Our algorithm works in r_2 iterations. We will show that the matrices $N_1, N_2, \ldots, N_k$ are contained in the Saks-Zhou sequence obtained using the sequence of hash functions found in the kth iteration and the k perturbation from the first k iterations. This would imply that $N_{r_2} \simeq M^{2^r}$ is contained in the the Saks-Zhou sequence obtained at the end of r_2 iterations. Finally, we will conclude that the algorithm outputs N_{r_2}.

We formally state and prove the above claim in Lemma 6. The following notations are used for that purpose. For $1 \leq k \leq r_2$, we denote by $\vec{h}^{(k)}$ the sequence of r_1 hash functions $(h'_1, h'_2, \ldots, h'_{r_1})$ found in the kth iteration, We define $\vec{\delta}^{(k)}$ to be the sequence $(\delta_1, \delta_2, \ldots, \delta_k)$. Finally, let $\vec{\delta}^{(0)}$ and $\vec{h}^{(0)}$ be the empty sequences.

Lemma 6 *Fix any $0 \leq k \leq r_2$. Let $M_0, M_1, M_2, \ldots, M_k$ be the matrices as defined in $\mathrm{SZ}(M, \vec{h}^{(k)}, \vec{\delta}^{(k)})$. Then, for $0 \leq i \leq k$, $M_i = N_i$. Moreover, if $k \geq 1$, δ_k is t-weakly-safe for $\widetilde{N}_k$.*

Proof The proof is by induction on k. The lemma is true for $k = 0$, since $N_0 = M_0 = M$ by definition. Assuming that the lemma holds up to some k, we will prove it for $(k+1)$.

We have to first argue that the r_1 exhaustive searches in the $(k+1)$st iteration (in Step 1a) succeed and produce good hash functions. In any ith loop there, we try to find a hash function h'_i that is ϵ-pseudorandom for some k automata. By Corollary 1, for any fixed r_2 automata, the probability that a randomly chosen h' is ϵ-pseudorandom for all the automata is at least $1 - r_2 d^5/(\epsilon^2 2^m)$. The above probability is non-zero by our choice of parameters. Since $k < r_2$, all r_1 the exhaustive searches will be successful.

Now consider the exhaustive search for δ_{k+1} (in Step 1b of $(k+1)st$ iteration), where we try to find a D-bit perturbation that is t-safe for some matrix. Lemma 5 shows that all but at most $2d^2$ perturbations are t-safe. Since $2^D > 2d^2$, we conclude that the search will successfully produce a t-safe perturbation δ_{k+1}.

We proceed to prove the induction step. For ease of reference, we first give names to the canonical automata corresponding to $N_0, N_1, \ldots, N_k$ and the matrices and automata obtained using $\vec{h}^{(k)}$ and $\vec{h}^{(k+1)}$.

- Let $M_0, \widetilde{M}_1, M_1, \ldots, \widetilde{M}_k, M_k$ be the matrices defined in $\mathrm{SZ}(M, \vec{h}^{(k)}, \vec{\delta}^{(k)})$.
  Let $Q_0, Q_1, \ldots, Q_k$ be the canonical automata corresponding to the matrices $M_0, M_1, \ldots, M_k$.
- Let $M'_0, \widetilde{M}'_1, M'_1, \ldots, \widetilde{M}'_{k+1}, M'_{k+1}$ be the matrices defined in $\mathrm{SZ}(M, \vec{h}^{(k+1)}, \vec{\delta}^{(k+1)})$.
  Let $Q'_0, Q'_1, \ldots, Q'_k$ be the canonical automata corresponding to the matrices $M'_0, M'_1, \ldots, M'_k$.
- Let $P_0, P_1, \ldots P_{r_2}$ be the canonical automata of $N_0, N_1, \ldots, N_{r_2}$, respectively.

Figure 5 illustrates the relationships among these matrices and automata. We will prove these relationships and argue that for $0 \le j \le k+1$, $M'_j = N_j$ and for $1 \le j \le k+1$, δ_{k+1} is t-weakly-safe for $\widetilde{N}_j$. This would prove the two parts of the the induction step.

$$
\begin{tikzcd}
Q_0 \arrow[r, "\mathcal{N}_{\vec{h}^{(k)}}"] & \widetilde{Q}_1 \arrow[d, "\mathcal{M}"] & Q_1 \arrow[r, "\mathcal{N}_{\vec{h}^{(k)}}"] & \widetilde{Q}_2 \arrow[d, "\mathcal{M}"] & Q_2 \arrow[r] & \cdots\cdots & Q_k & & \\
M_0 \arrow[u, "\mathcal{Q}"] & \widetilde{M}_1 \arrow[r, "\mathcal{R}_{\delta_1,t}"] & M_1 \arrow[u, "\mathcal{Q}"] & \widetilde{M}_2 \arrow[r, "\mathcal{R}_{\delta_2,t}"] & M_2 \arrow[u, "\mathcal{Q}"] & \cdots\cdots \arrow[r] & M_k \arrow[u, "\mathcal{Q}"] & & \\
\| & \wr & \| & \wr & \| & & \| & & \\
N_0 \arrow[r, "\mathcal{P}"] & \widetilde{N}_1 \arrow[r, "\mathcal{R}_{\delta_1,t}"] & N_1 \arrow[r, "\mathcal{P}"] & \widetilde{N}_2 \arrow[r, "\mathcal{R}_{\delta_2,t}"] & N_2 \arrow[r] & \cdots\cdots \arrow[r] & N_k \arrow[r, "\mathcal{P}"] & \widetilde{N}_{k+1} \arrow[r, "\mathcal{R}_{\delta_{k+1},t}"] & N_{k+1} \\
\| & \wr & \| & \wr & \| & & \| & \wr & \| \\
M'_0 \arrow[d, "\mathcal{Q}"] & \widetilde{M}'_1 \arrow[r, "\mathcal{R}_{\delta_1,t}"] & M'_1 \arrow[d, "\mathcal{Q}"] & \widetilde{M}'_2 \arrow[r, "\mathcal{R}_{\delta_2,t}"] & M'_2 \arrow[d, "\mathcal{Q}"] & \cdots\cdots \arrow[r] & M'_k \arrow[d, "\mathcal{Q}"] & \widetilde{M}'_{k+1} \arrow[r, "\mathcal{R}_{\delta_{k+1},t}"] & M'_{k+1} \\
Q'_0 \arrow[r, "\mathcal{N}_{\vec{h}^{(k+1)}}"] & \widetilde{Q}'_1 \arrow[u, "\mathcal{M}"] & Q'_1 \arrow[r, "\mathcal{N}_{\vec{h}^{(k+1)}}"] & \widetilde{Q}'_2 \arrow[u, "\mathcal{M}"] & Q'_2 \arrow[r] & \cdots\cdots & Q'_k \arrow[r, "\mathcal{N}_{\vec{h}^{(k+1)}}"] & \widetilde{Q}'_{k+1} \arrow[u, "\mathcal{M}"] &
\end{tikzcd}
$$

Figure 5: Induction step of Lemma 6.

We first claim that $\vec{h}^{(k+1)}$ is $2^{-(t+D+1)}$-pseudorandom for $P_0, P_1, \ldots, P_k$. Write $\vec{h}^{(k+1)} = (h'_1, h'_2, \ldots, h'_{r_1})$. By the construction of the algorithm, for $1 \le i \le r_1$, h'_i is ϵ-pseudorandom for $\mathcal{N}_{h'_1,h'_2,\ldots,h'_{i-1}}(Q_0)$, $\mathcal{N}_{h'_1,h'_2,\ldots,h'_{i-1}}(Q_1)$, $\ldots$, $\mathcal{N}_{h'_1,h'_2,\ldots,h'_{i-1}}(Q_k)$. Therefore, by Lemma 2, $\vec{h}^{(k+1)}$ is $2^{r_1}\epsilon$-pseudorandom for $Q_0, Q_1, \ldots, Q_k$. By the induction hypothesis, for $0 \le j \le k$, $M_j = N_j$ and hence $Q_j = P_j$. By our choice of parameters, $2^{r_1}\epsilon = 2^{-(t+D+1)}$. It follows that $\vec{h}^{(k+1)}$ is $2^{-(t+D+1)}$-pseudorandom for $P_0, P_1, \ldots, P_k$. (Notice that $\vec{h}^k$ is is $2^{-(t+D+1)}$-pseudorandom for $P_0, P_1, \ldots, P_{k-1}$, but not necessarily for P_k. The advantage of $\vec{h}^{(k+1)}$ over $\vec{h}^k$ is that the former is $2^{-(t+D+1)}$-pseudorandom for P_k also.) We have proved the claim.

We next analyze the safety properties of the perturbations $\delta_1, \delta_2, \ldots, \delta_{k+1}$. By the induction hypothesis, δ_j is t-weakly-safe for $\widetilde{N}_j$, for $1 \le j \le k$. Now we consider δ_{k+1}. By the construction of the algorithm, δ_{k+1} is t-safe for $\mathcal{M}(\mathcal{N}_{\vec{h}^{(k+1)}}(Q_k))$. Since $\vec{h}^{(k+1)}$ is $2^{-(t+D+1)}$-pseudorandom for P_k and P_k is the canonical automaton of N_k, we have $||\mathcal{M}(\mathcal{N}_{\vec{h}^{(k+1)}}(Q_k)) - N_k^{2^{r_1}}|| < 2^{-(t+D+1)}$. By our notation $\widetilde{N}_{k+1} = N_k^{2^{r_1}}$. It follows by Lemma 4 that δ_{k+1} is t-weakly-safe for $\widetilde{N}_{k+1}$. This proves the second part of induction step.

Finally, we prove the first part of induction step using the two fact shown already: (i) $\vec{h}^{(k+1)}$ is $2^{-(t+D+1)}$-pseudorandom for $P_0, P_1, \ldots, P_k$; (ii) for $1 \le j \le k+1$, δ_j is t-weakly-safe for $\widetilde{N}_j$. By definition $M'_0 = N_0 = M$. This implies that their canonical Q'_0 and P_0 are the same. So $\vec{h}^{(k+1)}$ is $2^{-(t+D+1)}$-pseudorandom for Q'_0. Hence the two matrices $\widetilde{M}'_1 = \mathcal{M}(\mathcal{N}_{\vec{h}^{(k+1)}}(Q'_0))$ and $\widetilde{N}_1 = M^{2^{r_1}}$ satisfy $||\widetilde{M}'_1 - \widetilde{N}_1|| < 2^{-(t+D+1)}$. Since δ_1 is t-weakly-safe for $\widetilde{N}_1$, by definition of t-weakly-safety, we see that $M'_1 = \mathcal{R}_{\delta_{k+1},t}(\widetilde{M}'_1) = \mathcal{R}_{\delta_{k+1},t}(\widetilde{N}_1) = N_1$. We use the argument inductively and conclude that for $0 \le j \le k+1$, $M'_j = N_j$. □

The following theorem proves the correctness of the algorithm.

Theorem 3 *Let M' be the matrix output by the procedure* MatrixPower*. Then* $||M' - M^{2^r}|| < 2^{-a}$.

Proof Consider the matrix M_{r_2} defined in $\mathrm{SZ}(M, \vec{h}^{(r_2)}, \vec{\delta}^{(r_2)})$. Lemma 6 shows that $M_{r_2} = N_{r_2}$. From Lemma 3 we know that N_{r_2} is a good approximation to M^{2^r}, namely $||N_{r_2} - M^{2^r}|| < 2^{1-t+r+\log d}$. From the choice of our parameters, we get that $||M_{r_2} - M^{2^r}|| < 2^{-a}$. Finally, notice that M_{r_2} is precisely the matrix M' output by the algorithm. □

3.4 Time/Space Analysis

Let us first analyze some subroutines. Given a (d, m)-automaton Q, and hash functions $h_1, h_2, \ldots, h_i \in \mathcal{H}_m$ for some $i \le r_1$, Proposition 1 allows us to compute $\mathcal{N}_{h_1,h_2,\ldots,h_i}(Q)$ in space $O(\log d)$ and thus time $d^{O(1)}$. The same resource bounds hold for computing $\mathcal{M}(\mathcal{N}_{h_1,h_2,\ldots,h_i}(Q))$, as we can evaluate the (u, v) entry by counting the number of $\alpha \in \{0,1\}^m$ that satisfy the condition, $Q_{h_1,h_2,\ldots,h_i}(u, \alpha) = v$.

Regarding the computation of the Saks-Zhou sequences, given $\vec{h} = (h_1, h_2, \ldots, h_{r_1})$ and $\vec{\delta} = (\delta_1, \delta_2, \ldots, \delta_{k-1})$, computing $\mathrm{SZ}(M, \vec{h}, \vec{\delta})$ can be done recursively in space $O(k \log d)$ and time $d^{O(k)}$. As indicated in Section 2.4, because of the breakdown of Proposition 1, we do not know of a more time efficient way for evaluating the Saks-Zhou sequence in $O(k \log d)$ space than the trivial $d^{O(k)}$.

Given the Saks-Zhou sequence $\mathrm{SZ}(M, \vec{h}, \vec{\delta})$, and $h'_1, h'_2, \ldots, h'_{i-1}$, it follows from the above and Corollary 1 that the exhaustive search for h'_i in step 1a can be performed in space $O(\log d)$ and time $d^{O(1)}$. Similarly, Lemma 5 shows that the exhaustive search for δ_k in step 1b can be performed within the same resource bounds.

The overall space consumption consists of work space for the subroutines that is constantly being reused, and storage for the global information. The latter consists of the "old" and a "new" sequence of r_1 hash functions, which requires $O(r_1 \log d)$ bits, and the $k \leq r_2$ many δ values of $O(\log d)$ bits each, for a total of $O((r_1 + r_2) \log d)$ bits. The need for temporary work space is dominated by the computation of the Saks-Zhou sequences, which requires no more than $O(r_2 \log d)$ bits. Thus, the overall space usage is $O((r_1 + r_2) \log d) = O(r_1 \log d)$.

As for the running time, given a blackbox for the Saks-Zhou sequences, everything runs in time $d^{O(1)}$. Since the computation of a Saks-Zhou sequence of length k takes time $d^{O(k)}$ and $k \leq r_2$, the overall running time is $d^{O(r_2)}$.

4 Further Research

In this paper, we established a tradeoff between Nisan's result that $\mathsf{BPL} \subseteq \mathsf{DTISP}[n^{O(1)}, O(\log^2 n)]$, and the Saks-Zhou result that $\mathsf{BPL} \subseteq \mathsf{DTISP}[n^{O(\log^{0.5} n)}, O(\log^{1.5} n)]$. It would be nice to get the best of both worlds at the same time, i.e., a deterministic simulation of BPL that simultaneously runs in polynomial time and space $O(\log^{1.5} n)$. The breakdown of Proposition 1 seems to be the bottleneck. This relates to the fact that Saks and Zhou obtain a deterministic simulation that runs in space $O(\log^{1.5} n)$ but not a pseudorandom generator with seed length $O(\log^{1.5} n)$ [8]; the construction of a pseudorandom generator for logspace with seed length $o(\log^2 n)$ remains open.

The big open question, of course, is whether BPL can be simulated deterministically in logarithmic space. We know that this is the case if there exists a language in deterministic linear space that requires branching programs of size $2^{\epsilon n}$ for some positive constant ϵ [3].

Acknowledgments

We thank the anonymous referees for their comments and suggestions.

References

[1] R. Aleliunas, R. Karp, R. Lipton, L. Lovasz, and C. Rackoff. Random walks, universal travel sequences and the complexity of maze problems. In *Proceedings of the 20th IEEE Symposium on Foundations of Computer Science*, pages 218–223, 1979.

[2] L. Carter and M. Wegman. Universal Hash Functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[3] A. Klivans and D. van Melkebeek. Graph nonisomorphism has subexponential size proofs unless the polynomial hierarchy collapses. *SIAM Journal on Computing*, 31(5):1501–1526, 2002.

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

[5] N. Nisan. RL$\subseteq$ SC. *Computational Complexity*, 4:1–11, 1994.

[6] O. Reingold. Undirected st-connectivity in log-space. To appear in *Proceedings of the 37th ACM Symposium on Theory of Computing*, 2005.

[7] M. Saks. Randomization and derandomization in space bounded computation. In *Proceedings of the 11th Annual IEEE Conference on Computational Complexity*, 1996.

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

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