Pseudo-determinism

by

Ofer Grossman

S. B., Massachusetts Institute of Technology (2017)
S. M., Massachusetts Institute of Technology (2020)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored by: Ofer Grossman
Department of Electrical Engineering and Computer Science
May 18, 2023

Certified by: Shafi Goldwasser
RSA Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Pseudo-determinism

by

Ofer Grossman

Submitted to the Department of Electrical Engineering and Computer Science
on June 2023, in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Abstract

A curious property of randomized algorithms for search problems is that on different executions on the same input, the algorithm may return different outputs due to differences in the internal randomness used by the algorithm. We would like to understand how we can construct randomized algorithms which while still using the power of randomness can ensure that when run multiple time on the same input, with high probability result in the same output on each of the executions.

We first show a pseudo-deterministic NC algorithm for finding matchings in bipartite graphs. As a corollary, we also show a pseudo-deterministic NC algorithm for constructing DFS trees in graphs.

We then show a reproducible algorithm for problems in search-RL. That is, we show an algorithm for problems in search-RL such that the output depends only on $O(\log n)$ many random bits used by the algorithm. We also show a pseudo-deterministic log-space low time algorithm for finding paths in undirected graphs. The algorithm is much faster than deterministic logspace algorithms for the problem.

Next, we investigate pseudo-determinism in the context of streaming algorithms. We show both lower and upper bounds for some classic streaming problems. Most notably, we show that the problem of approximate counting in a stream (for which the well known algorithm of Morris gives a $O(\log \log n)$ space algorithm), has no pseudo-deterministic algorithms using space $o(\log n)$.

Finally, we examine an extension of pseudo-determinism to the context of interactive proofs.

Thesis Supervisor: Shafi Goldwasser
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

Thank you to Shafi being such a supportive advisor and for giving me so much freedom to work on whatever I wanted.

Contents

Chapter 1

Introduction

In [GG11], Gat and Goldwasser initiated the study of probabilistic (polynomial-time) search algorithms that, with high probability, output the same solution on different executions. That is, for all inputs x, the randomized algorithm A satisfies $Pr_{r_1,r_2}(A(x,r_1) = A(x,r_2)) \geq 1 - 1/poly(n)$.

Another way of viewing such algorithms is that for a fixed binary relation R, for every x the algorithm associates a canonical solution $s(x)$ satisfying $(x, s(x)) \in R$, and on input x the algorithm outputs $s(x)$ with overwhelmingly high probability. Algorithms that satisfy this condition are called *pseudo-deterministic*, because they essentially offer the same functionality as deterministic algorithms; that is, they produce a canonical output for each possible input (except with small error probability)[1]. In contrast, arbitrary probabilistic algorithms that solve search problems may output different solutions when presented with the same input (but using different internal coin tosses); that is, on input x, the output may arbitrarily distributed among all valid solutions for x (e.g. it may be uniformly distributed).

Many questions come to mind, the main one being: in what instances can randomized algorithms be made pseudo-deterministic?

1.1 Some Motivation for Pseudo-determinism

Randomization is a powerful tool in algorithm design, and it is accepted that usually a deterministic algorithm for a problem is "better" than a randomized one, assuming the computational resources

[1] In fact, by amplifying the success probability, one can ensure that as black boxes, pseudo-deterministic algorithms are indistinguishable from deterministic algorithms by a polynomial time machine.

for the algorithms are the same.

Why are deterministic algorithms preferred over randomized ones? We believe the main downsides of randomized algorithms (compared to deterministic algorithms) are the following:

- They have a nonzero probability of error, failure, or large run-time.
- Random bits must be sampled, which may be infeasible or computationally expensive.
- The output is not predictable. That is, multiple executions of the algorithm on the same input may result in different outputs.

The first two points above have both significantly informed research on randomized algorithms. For example, the first point motivates the distinction between Monte Carlo and Las Vegas algorithms as well as RP versus BPP, and the need for improvements over naive amplification [IZ89, GM20]. The second point has led to work on pseudorandom generators [HILL99], how to recycle random bits [IZ89], and efficient extractors [Tre01], to mention a few.

Notice that unlike the first two points above, which are relevant both in the context of search and decision problems, the third point mainly applies to search problems. An algorithm for a decision problem, if its failure probability is small enough, with high probability will result in the same output when run twice on the same input (using different randomness). However, in the case of algorithms for search problems, it is possible for algorithms even with zero error probability to result in different outputs when executed twice on the same input.

The study of *pseudo-determinism* addresses the third point directly. Pseudo-deterministic algorithms also help address the two other problems, as we discuss below.

Lowering error probability. Consider the first item above (having a nonzero probability of error). For a standard search algorithm, it may not be clear how to reduce the probability of error. The most standard way of reducing error probability for algorithms is via repetition – if you have an algorithm with some error probability, and you run it 100 times and take the output that was most common, you will end up with a lower probability of error. While this strategy is very effective for decision problems (it can give an exponentially low probability of error after just polynomially many repetitions), note that this strategy does not work for general search algorithms. For example, if a search algorithm gives a different answer on each of the 100 repetitions, we will not

be able to reduce our error probability. However, for a pseudo-deterministic algorithm, we know that with high probability many of the 100 repetitions of the algorithm will result in the same output, allowing us to use this standard amplification technique. Other methods of amplification, such as those in [IZ89, GM20] also do not work for general search problems, but do for pseudo-deterministic algorithms.

Saving random bits. We now view the second item (random bits must be sampled, which may be infeasible or computationally expensive). Suppose you have 10 instances of a problem, and wish to solve all of them. Something you can do to save random bits is to simply sample random bits once, and then use the same random bits 10 times to solve all 10 instances. By a union bound, the probability of error in any of the 10 instances is at most a factor of 10 more than for an individual instance. This, of course, wouldn't work of the 10 instances were chosen adaptively. That is, suppose you first solve the first instance, and then the second instance may depend on the output of the output you gave for the first problem, and so on. In this case, one cannot use the union bound. However, if the algorithm used was pseudo-deterministic, since the output to each of the inputs does not strongly depend on the random bits chosen, the union bound would still work, allowing us to save random bits. In this sense, one can think of a pseudo-deterministic algorithm as one which uses "zero amortized random bits". This is since whatever random bits used to solve a pseudo-deterministic algorithm can with high probability later be safely reused.

This idea is used in this thesis in section 2.6 to save random bits for finding matchings and DFS trees in graphs in RNC.

Predictable outputs. As for the third problem (the output is not predictable. That is, multiple executions of the algorithm on the same input may result in different outputs), pseudo-deterministic algorithms address it directly. A pseudo-deterministic algorithm is exactly one which on the same input will with high probability result in the same output. Notice that this means if a pseudo-deterministic algorithm is viewed as a black box, it is indistinguishable from a deterministic algorithm. Whenever using it on a certain input, it results in the same output (with high probability). So, the black box exhibits a deterministic behaviour – whenever using the black box multiple times on the same input, it results in the same output on those multiple executions.

1.2 Outline.

This thesis is based on joint work with Shafi Goldwasser, Meghal Gupta, Dhiraj Holden, Yang Liu, Sidhanth Mohanty, Mark Sellke, and David Woodruff [GG15, GL19, GGMW20, GGS23, GGH17].

In Chapter 2, based on joint work with Shafi Goldwasser [GG15], we discuss a pseudo-deterministic parallel algorithm for finding a matching in a bipartite graph. In Chapter 3, based on joint work with Yang Liu [GL19] we study an extension of pseudo-determinism which we call *reproducibility*, and show that all problems in search-RL have reproducible algorithms. In Chapter 4, based on joint work with Shafi Goldwasser, Sidhanth Mohanty, and David Woodruff [GGMW20], we study pseudo-determinism in the context of streaming algorithms, showing both lower and upper bounds for various classic streaming problems. In Chapter 5, based on joint work with Meghal Gupta and Mark Sellke [GGS23], we continue the discussion of pseudo-determinism in the context of streaming algorithms, proving a lower bound for the pseudo-deterministic complexity of approximate counting. In Chapter 6, based on joint work with Shafi Goldwasser and Dhiraj Holden [GGH17], we study an extension of the notion of pseudo-deterministic algorithms to to pseudo-deterministic interactive proofs.

Chapter 2

Matching in pseudo-deterministic NC

The work in this Chapter is based on joint work with Shafi Goldwasser [GG15].

2.1 Introduction

Computing a maximum matching in a graph is a paradigm-setting algorithmic problem whose understanding has paved the way to formulating some of the central themes of theoretical computer science.

In particular, Edmonds [Edm65] proposed the definition of tractable polynomial-time solvable problems versus intractable non-polynomial time solvable problems following the study of the graph matching problem versus the graph clique problem. In the context of parallel algorithms, computing a maximum (or possibly perfect) matching is the problem standing at the center of the RNC versus NC question. It is known both in the general and in the bipartite case to be solvable by randomized RNC algorithms but it is unknown if deterministic NC algorithm exist.

Brief History of the Parallel Complexity of Matching:

One can distinguish between the *decision* version of the perfect matching problem (which asks whether a perfect matching exists) and the *search* version (which asks to return a perfect matching if any exist).

The decision problem is equivalent to testing whether the determinant of the Tutte matrix (or a simplified version of it in the bipartite case as shown by Edmonds [Edm65]) is identically 0. Lovász

[Lov79] realized that for the polynomial corresponding to the determinant of the Tutte matrix, this can be determined in a manner similar to the randomized identity testing of polynomials [Sch80]. The Tutte matrix's determinant is a polynomial whose degree n is the number of nodes in the graph. By assigning the indeterminates of the Tutte matrix values in the field $\mathbb{F}_p$ where p is a prime greater than $2n$, we create an integer matrix which with high probability has nonzero determinant if the Tutte matrix has nonzero determinant. Testing that a given integer matrix is non-singular can be done in NC, so an RNC algorithm for the decision problem follows.

The search version was subsequently shown to be in RNC by Karp, Upfal, and Wigderson [KUW85] [1]. The next breakthrough was the RNC algorithm of Mulmuley, Vazirani, and Vazirani [MVV87] which introduced the well-known isolation lemma. They assigned random weights from the set $\{1, 2, \ldots, 2|E|\}$ to the edges of the graph and proved the *isolation lemma* which states that with high probability the assignment induces (isolates) a unique min-weight perfect matching (if at least one perfect matching exists). Given an isolating weight assignment for a graph G, a perfect matching in G can be constructed in NC in a straightforward manner: for each edge we create a process whose task is to determine if this edge participates in the unique min-weight matching. Checking if an edge is in the unique min-weight perfect matching can be done by removing the edge from the graph and computing the determinant of the corresponding matrix.

A significant step forward has been made by Fenner, Gurjar, and Thierauf [FGT15] who showed how to remove randomization and obtain a quasi-NC algorithm for computing perfect matchings in bipartite graphs. Namely, they obtain a deterministic poly$(\log n)$ depth algorithm which uses more than a polynomial number of processors to compute a perfect matching in bipartite graphs.

Essentially, [FGT15] constructs an isolating weight assignments for bipartite graphs with quasi-polynomially large weights. More precisely, they first construct a weight function which ensures nonzero circulation (alternating sum of weights) for all small (length at most $4 \log n$) cycles in the graph. Next, they argue non-constructively that the union of all min-weight perfect matchings in the original graph would result in a smaller graph which has no small cycles. By iterating this argument for the new graph (after contracting degree 2 nodes with their neighbors), they show that after $k = O(\log n)$ iterations all cycles disappear and a single perfect matching remains. The final weight assignment is a combination of the weight assignments in each of the k iterations

[1] The algorithm probabilistically prunes out a constant fraction of edges from the graph in $O(\log |V|)$ stages such that after each stage, the remaining graph still has a perfect matching with high probability.

$w = \sum_{i=1}^{k} w_i B^{k-i}$ which can be proved to induce a unique min-weight matching for the original graph. Note that this entire procedure is not constructive since taking the union of all min-weight matchings takes too much time; however, it suffices to show that the weights involved in any iteration can be bounded by $O(n^{c \log n})$, and thus one can deterministically cycle through $O(n^{ck \log n})$ possible weight assignments where each weight can be described with $\text{poly}(\log n)$ bits.

This state of affairs leaves intriguing open problems:

- Can the decision problem "does a perfect matching exist" be solved in NC without the use of randomization?
- Can a perfect matching be found in NC without the use of a randomization, assuming that at least one (and possibly many) perfect matching exists?
- Can an isolating weight assignment be constructed in NC without resorting to randomization?
- Can an RNC algorithm be constructed such that with high probability it always finds the same perfect matching? That is, can an algorithm output with exponentially high probability a canonical matching for each input graph? Note that for the algorithm in [MVV87], each random weight assignment induces a unique min-weight perfect matching, but different random weight assignments for the same input graph may induce different matchings.

2.1.1 Our Results

In this work we settle this problem for the case of bipartite graphs.

We present a pseudo-deterministic RNC algorithm for the bipartite perfect matching search problem. On a bipartite graph G, the algorithm uses polynomially many processors, runs in $\text{poly}(\log n)$ time, and uses $\text{poly}(\log n)$ random bits to output a canonical perfect matching of G (or to state that no perfect matching exists). This is the first pseudo-deterministic RNC algorithm for bipartite perfect matching. In other words, previous RNC algorithms would output different matchings on different executions whereas our algorithm outputs the same perfect matching on each execution (with high probability).

We note that the best improvements in [CRS95] and in [FGT15] of the RNC algorithm of [MVV87] to find a perfect (though not necessarily unique) bipartite matching via the isolation lemma use $O(n \log \frac{m}{n})$ random bits.

Aggarwal, Anderson, and Kao [AAK89] found an RNC algorithm for constructing a depth first search tree for directed graphs. Their algorithm's only use of randomization is to solve bipartite min-weight maximum matching as a subroutine. Our results can be adapted to solve bipartite maximum matching. Hence, our results imply a pseudo-deterministic RNC algorithm for computing depth first search (DFS) in general directed graphs.

2.2 Solution Outline

In this section, we give an overview of the proof. It is intended to provide the reader with the basic ideas without going into detail. Sections 3-6 contain the rigorous analysis.

Let G be a graph and w be a weight assignment to the edges of G. We will later explain how we construct w, but for now we treat it as given.

Our first observation, Lemma 2.4.3, is that randomization allows us to find the union of all min-weight perfect matchings of G (deterministically, this is not known to be possible). By taking the union of min-weight perfect matchings, we are effectively throwing out the edges which do not participate in any perfect matching while maintaining the property that the graph has a perfect matching. The idea behind computing the union of min-weight perfect matchings is that for each edge e_i, we create a process whose goal is to figure out if e_i participates in some min-weight perfect matching. Each such process will create a new weight assignment which lowers the weight of e_i by a small amount and then uses randomization to find a min-weight perfect matching with the new weight assignment. We pick the weights so that if e_i is in a min-weight matching with respect to the original weight assignment, then it must be in all min-weight matching with respect to the new assignment. Also, if e_i is not in a min-weight matching with respect to the original weight assignment, then it must be in none of the min-weight matchings with respect to the new assignment. Now, by finding a min-weight perfect matching (which we can do in RNC using techniques in [MVV87]), we can determine whether e_i is in the union of min-weight matchings.

We now construct weight assignments such that the union of all edges in min-weight matchings has many vertices of degree at most 2. Degree 2 vertices are very easy to deal with since we can contract them with their neighbors to get a smaller graph. If we find a perfect matching on the smaller graph, it is easy to extend it to the original graph.

By a theorem in [AHL02], we learn that if the girth (length of the shortest cycle) of G is at least

$4 \log n$, then at least $\frac{1}{10}$ of the vertices have degree at most 2. Therefore, if our weight assignment can make all small cycles disappear, we will be able to reduce our problem to a smaller graph and be done.

We also show that given a weight assignment, every even cycle with nonzero circulation (alternating sum of weights) disappears when we look at the union of min-weight matchings. We then construct a weight function such that small (containing fewer than $4 \log n$ vertices) cycles disappear:

Lemma 2.2.1. *Let G be a bipartite graph on n vertices. Then, for any number s, one can construct a set of $O(s \log n)$ weight assignments with weights bounded by $O(s \log n)$ such that every cycle of length up to s has nonzero circulation for at least one of the weight assignments.*

The proof for this Lemma appears in section 4.

We now set $s = 4 \log n$, and in series we update our graph for each of the $O(s \log n)$ weight assignments by looking at the union of min-weight matchings. When we are done, we have a graph with high girth, so we can contract many vertices of degree up to 2. We then recurse, completing the proof.

2.3 Preliminaries

We begin with some lemmas from previous work.

Lemma 2.3.1 (Theorem 2 in [MVV87])**.** *Given a graph G with a weight function $w : E \to \mathbb{Z}$, with polynomially bounded weights, it is possible to construct a w-minimal perfect matching of G in RNC.*

Definition 2.3.2 (Circulation)**.** Let $G(V, E)$ be a graph with weight function w. Let C be a cycle in the graph. The *circulation* $c_w(C)$ of an even length cycle $C = (v_1, v_2, \ldots, v_k)$ is defined as the alternating sum of the edge weights of C,

$$c_w(C) = |w(v_1, v_2) - w(v_2, v_3) + w(v_3, v_4) - \cdots - w(v_k, v_1)|.$$

Circulation has been used for an NC algorithm for perfect planer bipartite matching [DKR10] and for a quasi-NC algorithm for bipartite matching [FGT15].

Lemma 2.3.3 (Lemma 3.4 in [FGT15]). *Let G be a bipartite graph. Let w be a weight function such that the cycles $C_1, C_2, \ldots, C_n$ have nonzero circulations. Then the graph G_1 obtained by taking the union of all min-weight perfect matchings on G will not have any of the cycles $C_1, C_2, \ldots, C_n$.*

The following lemma originates in [AHL02], and is presented in this form in [FGT15].

Lemma 2.3.4 (Corollary 3.6 in [FGT15]). *Let H be a graph with girth (length of shortest cycle) $g \geq 4 \log n - 2$. Then H has average degree < 2.5. In particular, at least $\frac{1}{10}$ (a constant fraction) of the vertices have degree at most 2.*

2.4 Key Lemmas

Recall that in [MVV87] a weight assignment is chosen at random such that with high probability there is a unique min-weight perfect matching. Our goal will be to deterministically construct weight assignments with similar properties.

Lemma 2.4.1 (Uniquifying assignment for small sets.). *Let S be a set with $|S| = n$. For any number k, one can construct (in NC) a weight assignment $w : S \to \mathbb{Z}$ with weights bounded by $2^{O(s \log n)}$ such that no two distinct subsets $S_1, S_2 \subset S$ satisfying $|S_1|, |S_2| \leq k$ have the same sum of weights.*

We can think about the Lemma as an assignment which isolates all small subsets of S. We will later use this Lemma to construct a weight assignment for the graph G.

Proof. Let $S = \{s_1, \ldots, s_n\}$. Consider the following weight assignment, where we write $w(m)$ as shorthand for $w(s_m)$:

$$w(m) = p^{2k}m + p^{2(k-1)}[m^2]_p + p^{2(k-2)}[m^3]_p + \cdots + k^0 p^0 [m^{k+1}]_p$$

where $[x]_p$ means the number between 1 and p which is equal to x modulo p and where p is an arbitrary prime greater than n^2. We can find such a prime by having n^2 processes each check a different number between n^2 and $2n^2$. Each of these processes initiate $2n^2$ processes which each test divisibility by an integer up to $2n^2$. (Note that this has no implications regarding generating primes in NC since our input is of size n instead of $\log n$).

Suppose there exist two distinct subsets of size up to k with equal sums of weights. We can add zeroes to both subsets such that the sizes of the sets are exactly k. Suppose that the sums of the

weights of two subsets $A = \{a_1, a_2, \ldots, a_k\}$ and $B = \{b_1, b_2, \ldots, b_k\}$ are the same. We note that this would imply that the sums of the $p^{2k}m$ terms for both A and B must be equal because the $p^{2k}m$ term is much larger than all other terms (or it is 0). Similarly, the sums of the $k^{2(k-1)}p^{2(k-1)}[m^2]_p$ terms must equal and so on for $k^{2(k-i)}p^{2(k-i)}[m^{i+1}]_p$ for all i. Therefore, we have the following equivalences modulo p :

$$\begin{aligned} a_1 + a_2 + \cdots + a_k &\equiv b_1 + b_2 + \cdots + b_k \pmod{p} \\ a_1^2 + a_2^2 + \cdots + a_k^2 &\equiv b_1^2 + b_2^2 + \cdots + b_k^2 \pmod{p} \\ &\cdots \\ a_1^{k+1} + a_2^{k+1} + \cdots + a_k^{k+1} &\equiv b_1^{k+1} + b_2^{k+1} + \cdots + b_k^{k+1} \pmod{p}. \end{aligned}$$

We claim that this implies that $A = B$. We note that if $a_i \equiv b_j$ modulo p, then $a_i = b_j$ because p is larger than n^2 which is the maximal size of a_i or b_j. Therefore, it will suffice to show that the set A and the set B are equivalent in $\mathbb{F}_p$.

Newton's identities, given the sums of the ith powers of the a_j for i between 1 and k, uniquely determine the values of the fundamental symmetric polynomials in the a_j. Therefore, Newton's identities also uniquely determine the minimal polynomial which has as roots all of the a_j (with multiplicity). We know that this polynomial will be of degree k and therefore since the b_j share this polynomial, the set of the a_i and the set of the b_j must be equal (they are both the set of roots of the same polynomial), completing the proof that the weight assignment has no two subsets of size up to k with the same sum of weights.

We note that the weights are bounded by $p^{2k+2} = 2^{O(k \log n)}$. □

If a cycle of length s has circulation 0, then there are two distinct subsets of size $s/2$ that have the same sum of weights (namely, the sum of the weights of the cycle's odd edges equals the sum of the weights of the cycle's even edges). Therefore, the above Lemma implies that we can construct weight assignment for G with weights bounded by $2^{O(s \log n)}$ such that all cycles of length up to s have nonzero circulation.

Lemma 2.4.2 (Nonzero circulation for small cycles.)**.** *Let G be a graph on n vertices. Then for any s, one can construct a set of $O(s \log n)$ weight assignments with weights bounded by $O(s \log n)$ such that every cycle of length up to s has nonzero circulation for at least one of the weight assignments.*

Proof. We begin with our weight assignment from Lemma 2.4.1 with $k = \lfloor s/2 \rfloor$. We consider the weight assignment modulo small numbers, i.e., the weight functions $\{w \pmod j | 2 \leq j \leq t\}$ for some appropriately chosen t. (The idea here is to pick t so that if a cycle has nonzero circulation in w, then it will have nonzero circulation in $w \pmod j$, for some $j \leq t$.)

We note that if the lemma does not hold then there exists a cycle C such that $c_w(C) \equiv 0 \pmod j$, for all j between 1 and t. Therefore,

$$\text{lcm}(2, 3, \ldots, t) | c_w(C_i).$$

The right is bounded above by $2^{O(s \log n)}$. In [Nai82] we learn that $\text{lcm}(2, 3, \ldots, t) > 2^t$ for sufficiently large t, so letting $t = O(s \log n)$ makes it so a cycle with nonzero circulation with respect to w is guaranteed to have nonzero circulation with respect to $w \pmod j$ for some $2 \leq j \leq t$.

Therefore, we have $O(s \log n)$ total weight assignments with weights bounded by $O(s \log n)$ such that every cycle of length up to s has nonzero circulation in at least one weight assignment. □

The following lemma shows that in RNC we can find the union of min-weight perfect matchings of a graph G with a weight assignment w.

Lemma 2.4.3 (Union of min-weight perfect matchings)**.** *Let $G(V, E)$ be a bipartite graph with weight function w. Let E_1 be the union of all min-weight perfect matchings in G. There exists an RNC algorithm for finding the set E_1.*

The idea behind the proof is that for each edge e_i, we run a process whose goal is to tell whether e_i is part of a min-weight perfect matching. To do so, the process creates a new weight function which lowers e_i so that if e_i was in a min-weight perfect matching, under the new weight assignment e_i is in every min-weight perfect matching (but if e_i was not in a min-weight perfect matching, it should still not be in a min-weight matching). Then, we use Lemma 2.3.1 to find a min-weight perfect matching, and we check if e_i is in the matching. e_i will be in the matching if and only if it is part of a min-weight matching with respect to the original weight function.

Proof. For each edge $e_i \in E$, consider the weight function w_i defined by

$$w_i(e_j) = \begin{cases} 2w(e_j) - 1 & \text{if } i = j \\ 2w(e_j) & \text{if } i \neq j. \end{cases}$$

Suppose that M is the minimal weight for a matching with respect to w. Then with respect to w_i, the min-weight matching will have weight $2M$ if e_i is in no w-minimal matching. Otherwise, the min-weight matching will have weight $2M - 1$. By finding a w_i-minimal, which we can do with Lemma 2.3.1, and checking whether e_i participates in the matching, we can determine whether e_i is in a w-minimal matching.

Note that this is highly parallelizable: we can run the above for each edge in parallel. Then, we return the set of all e_i which are part of some w-minimal matching. □

2.5 The Algorithm

We now put everything together to construct an algorithm (Fig. 1).

```
PERFECT-MATCHING(G)
 1  If |E(G)| ≤ 100 :
 2        Find and return a perfect matching of G using brute force.
 3  Construct the weight assignments defined in Lemma 2.4.2 with s = 4 log n.
 4  For each weight assignment w:
 5        In parallel for each edge e_i in G :
 6              Construct the weight w_i defined in Lemma 2.4.3.
 7              If e_i is not in the w_i-minimal matching:
 8                    Remove e_i from G.
 9  Contract all edges adjacent to vertices of degree 2 in G to create G'.
10  Run PERFECT MATCHING(G').
11  Remove from G all vertices part of the perfect matching of G'.
12  In parallel for each connected component of G :
13        Find a perfect matching of the connected component (which is a cycle or a path).
14  Return the union of matchings of connected components and the matching of G'.
```

Figure 2-1: A pseudo-deterministic *RNC* algorithm finding a perfect matching in a given bipartite graph G.

We note that randomization is only used in line 7. We use the randomization in the following

context: given a weight assignment and an edge, output whether the edge is part of some min-weight perfect matching. Since this is a Yes/No question, correctness implies uniqueness so our algorithm returns the same output with high probability, and is therefore pseudo-deterministic.

We will now analyze the algorithm and show it lies in RNC.

We note that line 2 takes $O(1)$ time. Line 3 is in RNC, as shown in Lemma 2.4.2.

Line 7 can be done in RNC by Lemma 2.4.3. Line 7 is in RNC since it is finding a min-weight perfect matching when the weight function is isolating. After line 8, the graph G still has a perfect matching.

By Lemma 2.4.2 and Lemma 2.3.3, we see that after the loop in line 4, G has no cycles of length less than $4 \log n$. By Lemma 2.3.4, G' will have a constant fraction of the number of vertices of G. Therefore, the depth of the recursions line 10 gives us is $\log n$.

In line 13, we know that each connected component only has vertices of degree 1 or 2. Therefore, each connected component is either a path or a cycle. We can find a perfect matching here deterministically by letting the weight of one of the edges be 1 and the weight of all other edges be 0. In this case, if we are a path we have one perfect matching which we will find, and if we are a cycle, one of the perfect matchings will have weight 1 and the other will have weight 0. Therefore, the weight assignment is isolating, so lines 12 and 13 are in NC.

We see that every vertex is either part of a connected component from line 12 or is an element of the perfect matching of G'. Hence, every vertex is matched, proving the correctness of the algorithm.

This completes the algorithm's analysis.

We note that contracting edges in NC takes some care. To contract, we first create the subgraph consisting of all edges adjacent to a vertex of degree at most 2. We can do this by running a process for each edge which checks the degree of its vertices. Then, we will contract all connected components. We do this by checking for each pair of vertices if they are in the same connected component (we can do this because the ST connectivity problem is in NL which is a subset of NC). Now, for each vertex v we find the minimal (in lexicographic order) vertex v_m in its connected component and delete v while attaching all edges adjacent to v to v_m. This completes the analysis for contraction in NC.

2.6 Using Fewer Random Bits

In this section, we will construct a pseudo-deterministic RNC algorithm for the bipartite perfect matching problem which uses only $\text{poly}(\log n)$ random bits.

Our algorithm is based on our previous pseudo-deterministic NC algorithm. We note that in our previous algorithm, the only use of randomization was to solve the following subproblem: given a graph G, a weight assignment w with polynomially bounded weights, and an edge e, output whether the edge e is part of a max-weight perfect matching (we note that we can talk about max-weight matchings even though earlier we talked about min-weight matchings because we can define a new weight function $w'(e_i) = \max_{e_j} w(e_j) - w(e_i)$ such that all w-minimal matchings are w'-maximal). We will show how to solve this with $\text{polylog}(n)$ random bits.

Let $M = \max_{x \in E} w(x)$. Consider the weight assignment w_e defined by

$$w_e(e') = \begin{cases} w(e') + nM & \text{if } e' = e \\ w(e') & \text{otherwise.} \end{cases}$$

If there exists a perfect matching containing e, then all max-weight perfect matchings with respect to w_e will contain e. We note that if e is part of a max-weight matching with respect to w, then the max-weight matching with respect to w_e will have weight $W + nM$ where W is the weight of the maximum perfect matching with w. On the other hand, if e is not a max-weight matching with respect to w, then the max-weight matching with respect to w_e will have weight at most $(W - 1) + nM = (W + nM) - 1$. We will detect this difference by constructing a matrix and calculating its determinant.

Consider the following matrix, where the a_{ij} will be defined later, and z is an indeterminate.

$$A_e(i,j) = \begin{cases} z^{w_e(v_i,u_j)} a_{ij} & \text{if } (u_i, v_j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

We can set z to be much larger than the a_{ij}. For example, we can set $z = n^{n^2} \max_{i,j} |a_{ij}|^n$. We can write the determinant as

$$\det(A_e) = \sum_{S \text{ a perfect matching in } G} sgn(S) z^{w_e(S)} \prod_{e \in S} a_e.$$

We see that because we picked z to be so large, each term where S a max-weight matching will be larger than the sum of all terms with non-max-weight matchings. Then, assuming that the terms with z^{W_e} (where W_e is the weight of a max-weight matching with respect to w_e) do not cancel, we can recover W_e from the determinant by finding the largest n such that $z^n \leq 2|\det(A_e)|$.

Now that we know W_e for every edge e, we can find the maximum of the set $\{W_e : e \in E\}$. The e_i such that W_{e_i} is maximal are the edges which are part of a max-weight perfect matching with w. This set is the union of max-weight perfect matchings, as we wished.

Therefore, it will suffice to find a_{ij} so that the terms with z^{W_e} do not cancel. This is exactly the same as finding a_{ij} such that the matrix

$$A'_e(i,j) = \begin{cases} a_{ij} & \text{if } (u_i, v_j) \text{ in a max-weight matching with } w_e \\ 0 & \text{otherwise} \end{cases}$$

has nonzero determinant.

In section 5 of [FGT15], there is a randomized construction for the a_{ij} such that for each graph G' which has a perfect matching, the matrix

$$A'_{G'}(i,j) = \begin{cases} a_{ij} & \text{if } (u_i, v_j) \in E(G') \\ 0 & \text{otherwise} \end{cases}$$

has nonzero determinant with high probability. (Note that the a_{ij} do not depend on G'. This is important because we don't actually know that the set of edges in a max-weight matching with w_e.)

Because the construction uses $\text{poly}(\log n)$ bits and can achieve $\frac{1}{n^3}$ probability of failure, we can use the the same values of a_{ij} for all e, and by the union bound the probability that any failures occur is still small: at most $\frac{1}{n}$. Therefore, we can solve the subproblem using $\text{polylog}(n)$ bits, which completes the proof.

2.7 Discussion

The above implies a pseudo-deterministic RNC algorithm for depth first search, another problem in RNC that is not known to be in NC. This result follows immediately from [AAK89], where an RNC algorithm for DFS is presented, and the only use of randomization is in a subroutine for finding a min-weight perfect matching in a bipartite graph.

The above also implies RNC algorithms for some network flow problems such as max-flow approximation, which was shown in [SS91] to be NC-reducible to maximum bipartite matching.

Followup work. After the work from this chapter was first published, an extension to general (nonbipartite) graphs was shown in [AV19]. In [GG21] a pseudo-deterministic reduction from search to weighted-decision was shown for the matroid intersection problem.

Chapter 3

Reproducibility and Low Space Computation

The work in this Chapter is based on joint work with Yang Liu [GL19].

3.1 Introduction

3.1.1 Reproducible Outputs

When using a log-space machine to perform a randomized search algorithm with a polynomial-sized output, the output cannot be fully stored. Running the algorithm again with new random bits may result in a new (and potentially different) output. Hence, after running the computation, we lose access to the outputted answer, and are unable to reproduce it.

Consider, for example, the following simple computational problem: Given a (directed) graph G and two vertices s and t such that a random walk from s hits t quickly with high probability, output two copies of the same path from s to t. That is, the goal is to output some path P, and then output the same path P again. It's not clear how to perform the above in randomized log-space, since after outputting some path P, it's not clear how to reproduce P and be able to output it again. So, although outputting a single path is easy, or two potentially different paths, it's not clear how to output the same path twice.

Another example of this phenomenon in play is that it is known that there is a randomized reduction from NL to UL (in fact, NL is reducible to UL $\cap$ coUL) [RA00, GW96]. It follows that if

UL $\subseteq$ RL, then NL can be solved by randomized log-space algorithms with *two-way access* to the random bits (that is, there is a randomized disambiguation of NL which uses two-way access to the random bits). However, when assuming UL $\subseteq$ RL, it is not known whether NL can be solved by a randomized log-space algorithm with *one-way* access to the random bits. The two-way access to the random bits is needed so that the output of the reduction (which is an instance of a problem in UL) can be accessed in a two-way fashion. If the output of the reduction was reproducible, then one-way access to the random bits would suffice. However, it is not known whether there exists such a reduction which is reproducible.

One way to achieve reproducibility is through *pseudo-determinism*. Pseudo-deterministic algorithms are randomized search algorithms which, when run on the same input multiple times, with high probability output the same result on all executions. Given such an algorithm, it is possible to reproduce outputs: simply run the algorithm again using new randomness. We manage to achieve reproducibility using a different and novel approach which does not involve finding a pseudo-deterministic algorithm for the problem.

3.1.2 Our Contribution

Our contribution falls into two parts: contributions to reproducibility in the context of log-space, and contributions to pseudo-determinism in the context of log-space.

Reproducibility: We introduce the notion of reproducibility and provide a definition in Section 3.3. Our main result shows that every problem in search-RL (see Section 3.2 for a definition of search-RL) can be solved so that its output is reproducible. Essentially, a problem has reproducible solutions if for every input x we can generate a short string t_x so that given both x and t_x we can keep reproducing copies of the same y satisfying $(x, y) \in R$. That is, by memorizing only the short string t_x, we can continue to produce more copies of the same output. Given such an algorithm, it is now possible to simulate running any logspace algorithm on y in logspace by simply memorizing t_x and using it to reconstruct y whenever needed. Otherwise, we could only run a log-space algorithm A on y if A required one-way access to their input. If some algorithm A required two-way access to its input, we wouldn't be able to to use composition to run that algorithm on y.

In order to achieve reproducibility, we show that for every problem in search-RL there is some randomized log-space algorithm A such that with high probability, the output of A only depends

on the first $O(\log n)$ random bits A samples. That is, after sampling the first $O(\log n)$ random bits, with high probability for most choices of the rest of the random bits used by the algorithm, the same result will be outputted. This property makes it possible for an algorithm to output a pair (y, y), where y is a valid output to the original problem, as it can store the first $O(\log n)$ random bits it sampled in memory, and use them to recreate y twice. Since the algorithm can find and store the information needed to reproduce the answer y, we say that the output is *reproducible*.

Our first result is that every problem in search-RL (as defined in [RTV06]) has a randomized log-space algorithm whose output, with high probability, only depends on its first $O(\log n)$ random bits. This implies that every problem in search-RL has reproducible solutions.

Informal Theorem 3.1.1. Every problem in search-RL has a randomized log-space algorithm whose output, with high probability, only depends on its first $O(\log n)$ random bits.

A more precise statement is given in Section 3.3 as Theorem 3.3.5. The algorithm we present has several other noteworthy properties, which we discuss in Subsection 3.3.6. This includes that the output of the algorithm, when viewed as a random variable depending on the random choices used by the algorithm, has entropy $O(\log n)$. This is significantly lower than a standard search-RL algorithm, which may have polynomial entropy.

Pseudo-determinism: In later sections, we show faster pseudo-deterministic algorithms for finding paths in undirected and Eulerian graphs. These algorithms are reproducible even without storing $O(\log n)$ bits in memory.

For undirected graphs, a deterministic log-space algorithm has been shown by Reingold [Rei08]. One of the drawbacks of this algorithm is that its runtime, while polynomial, has a very large exponent, since it requires going over all paths of length $O(\log n)$ on a certain graph, with a large constant hidden in the O (for certain expositions of the algorithm, the polynomial runtime is larger than $O(n^{10^9})$). This can likely be improved, but we imagine it would be difficult to lower it to a "reasonable" polynomial time complexity. We show a pseudo-deterministic algorithm for the problem which runs in the more reasonable time of $\tilde{O}(mn^3)$:

Theorem 3.1.2 (Pseudo-deterministic Path Finding in Undirected graphs in $\tilde{O}(mn^3)$ time, $O(\log n)$ space)**.** Let G be a given undirected graph with n vertices and m edges. Given two vertices s and t

of G which are connected, there is a pseudo-deterministic log-space algorithm which outputs a path from s to t. Furthermore, the algorithm runs in time $\tilde{O}(mn^3)$.

We then generalize the theorem to Eulerian graphs (directed graphs where each vertex has indegree equal to its outdegree). Finding paths in such graphs deterministically has been shown in [RTV06]. Once again, the algorithm given in [RTV06] suffers from a very large polynomial runtime.

Theorem 3.1.3 (Pseudo-deterministic Path Finding in Eulerian graphs in $\tilde{O}(m^5n^3)$ time, $O(\log n)$ space)**.** Let G be a given Eulerian graph with n vertices and m edges. Given two vertices s and t of G such that there is a directed path from s to t, there is a pseudo-deterministic log-space algorithm which outputs a path from s to t. Furthermore, the algorithm runs in time $\tilde{O}(m^5n^3)$.

3.1.3 Related Work

RL vs L: Related to our result on pseudo-deterministic undirected connectivity is the work of Reingold, which showed that undirected connectivity can be solved *deterministically* with logarithmic space [Rei08]. Later, this result was extended to find pseudo-random walks on Eulerian graphs by Reingold, Trevisan, and Vadhan [RTV06].

One of our techniques may remind some readers of the work of Saks and Zhou that show that problems in BPL can be solved deterministically using $O(\log^{3/2}(n))$ space [SZ99]. In [SZ99], the authors add random noise to certain computed matrices, in order to be able to reuse certain random bits. In this work, we pick a certain 'threshold' at random, and this allows us to reuse randomness (more accurately, it makes our output be pseudo-deterministic with respect to certain random bits). The two ideas are similar in that they use randomization in an unconventional way in order to make the output not depend on certain random bits (for Saks and Zhou, this was helpful since it allowed those random bits to be reused). While the approach of Saks and Zhou was surely an inspiration for some of the high level ideas of this paper, the details of this paper turned out quite different from those in [SZ99].

3.2 Preliminaries

In this section we establish some definitions and lemmas that will be useful in later parts of the paper. Many of our definitions, especially those related to search problems in the context of log-space, follow closely to the definitions in [RTV06].

We begin by defining a search problem.

Definition 3.2.1 (Search Problem)**.** A *search problem* is a relation R consisting of pairs (x, y). We define $L_R = \{x | \exists y \text{ s.t. } (x, y) \in R\}$, and $R(x) = \{y | (x, y) \in R\}$.

The computational task associated with a search problem R is: given x, find a y such that $(x, y) \in R$. From this point of view, L_R corresponds to the set of valid inputs, and $R(x)$ corresponds to the set of valid outputs on input x.

We now define classes of search problems in the context of log-space. Our definitions follow closely to those of [RTV06].

Definition 3.2.2 (Log-space search problem)**.** A search problem R is *log-space* if there is a polynomial p such that if $y \in R(x)$ then $|y| \leq p(|x|)$ and there is a deterministic log-space machine that can decide if $(x, y) \in R$ with two-way access to x and one-way access to y.

We now define the class search-L. We remind the reader that a transducer is a Turing machine with a read-only input tape, a work tape (in our case, of logarithmic size), and a write-only output tape.

Definition 3.2.3 (search-L)**.** A search problem R is in *search-L* if it is log-space and if there is a logarithmic space transducer A such that $A(x) \in R(x)$ for all x in L_R.

Definition 3.2.4 (search-RL)**.** A search problem R is in *search-RL* if it is log-space and if there is a randomized logarithmic space transducer A and polynomial p such that $\Pr_r[A(x, r) \in R(x)] \geq \frac{1}{p(|x|)}$ for all $x \in L_R$.

The following computational problem is complete for search-RL:

Definition 3.2.5 (Short-Walk Find Path)**.** Let R be the search problem whose valid inputs are $x = (G, s, t, 1^k)$ where G is a directed graph, s and t are two vertices of G, and a random walk of length k from s reaches t with probability at least $1 - 1/|x|$ (where $|x|$ represents the length of the input x). On such an x, a valid output is a path of length up to k from s to t.

Lemma 3.2.6. Short-Walk Find Path *is complete for search-RL.*

We prove the above lemma in 3.7, via a reduction from Poly-Mixing Find Path, a problem which was shown to be complete for search-RL in [RTV06]. Intuitively speaking, the input to this problem is a graph with low enough mixing time, and the goal is to find a path in this graph between two input vertices s and t. For a formal definition of Poly-Mixing Find Path, we refer the reader to [RTV06].

Before going on to the algorithm in Section 3.3, we make a definition to simplify the explanations.

Definition 3.2.7. For a graph G with vertices s and t, and a positive integer k, let $p_k(s,t)$ denote the probability that a random walk of length k starting from s goes through t.

One of the key lemmas used by the algorithm is that one can estimate the value of $p_k(s,t)$ up to some polynomial additive error in search-RL. To do so, we simulate polynomially many random walks starting at s and count the fraction that pass through t. This is made precise in the following lemma:

Lemma 3.2.8. *Consider a graph G with n vertices, two of which are s and t. Let k be a positive integer. Then there exists a randomized log-space algorithm that on input $(G, s, t, 1^k)$ outputs an estimate μ for $p_k(s,t)$ satisfying $|\mu - p_k(s,t)| \leq \frac{1}{k^5 n^5}$ with probability at least $1 - 2e^{-2kn}$.*

Proof. To find μ, we simulate many random walks from s of length k, and then output the fraction which reach t. More precisely, we use the following algorithm: simulate $k^{11}n^{11}$ random walks of length k starting at s, and count how many end at t. Say that C of them do. Then output $\frac{C}{k^{11}n^{11}}$. To show that this works, it suffices to note that

$$\Pr\left[\left|\frac{C}{k^{11}n^{11}} - p_k(s,t)\right| \leq \frac{1}{k^5 n^5}\right] \geq 1 - 2e^{-2kn}$$

by Hoeffding's inequality. □

3.3 An Algorithm for Search-RL with Reproducible Outputs

3.3.1 Reproducibility

We begin with a formal definition of a problem with reproducible outputs. Essentially, a problem has reproducible solutions if for every input x we can generate a short string t_x so that given both x

and t_x we can keep reproducing copies of the same y satisfying $(x, y) \in R$. That is, by memorizing only the short string t_x, we can continue to produce more copies of the same output.

Definition 3.3.1 (Reproducible)**.** We say that a search problem R has *log-space reproducible* solutions if there exist randomized log-space algorithms A and B satisfying the following properties:

- On input x, with probability at least $\frac{2}{3}$, A outputs a string t_x of length $O(\log n)$ such that the second bullet holds.[1]
- There exists some y satisfying $(x, y) \in R$ such that with probability at least $\frac{2}{3}$, B outputs y when running on input (x, t_x).

A justification for the definition of reproducibility: One may argue that the definition proposed for reproducibility is highly structured, and that there may be ways to construct algorithms which capture the notion of reproducibility without adhering to the structure of Definition 3.3.1. We note that the "weakest" possible notion for reproducibility is that an algorithm has reproducible solutions if there is some randomized logspace algorithm C which outputs (y, y) where y is a valid output for the original problem. This is since if such an algorithm doesn't exist, then there is no hope to achieve any sort of reproducibility, since we essentially can't even reproduce the output a single time. Lemma 3.3.2 below shows that this weak notion is equivalent to Definition 3.3.1, showing that the seemingly too-strict definition (Definition 3.3.1) is equivalent to the seemingly too weak definition (that a problem has reproducible outputs if there exists an algorithm C which outputs (y, y) where y is a valid output to the original problem), demonstrating that the correct notion of reproducibility is captured by Definition 3.3.1.

Reproducibility is closely related to pseudo-determinism. In the case where t_x is of size 0, the algorithm B is a pseudo-deterministic algorithm for the search problem R.

An alternate way to view reproducibility is that a search problem R has log-space reproducible solutions if there exists some randomized log-space algorithm C such that algorithm C outputs (y, y) for some y satisfying $(x, y) \in R$. Essentially, this alternate view captures the fact that a problem

[1] Via exhaustive search, it can be shown that if for all inputs x there exists a t_x such that the second bullet holds, there also exists a log-space algorithm A that for all x with high probability will output some t_x satisfying the second bullet.

has log-space reproducible solutions if and only if we can produce some output, and then produce it again, ensuring that the output was not lost after the first time we computed it.

Lemma 3.3.2. *A search problem R has log-space reproducible solutions if and only if there exists some randomized log-space algorithm C such that for all valid inputs x (i.e., $x \in L_R$), with high probability $C(x)$ outputs two copies of an output y satisfying $(x, y) \in R$. That is, with high probability C outputs the tuple (y, y), where $(x, y) \in R$.*

Proof. First we show that every search problem R with log-space reproducible solutions has a randomized log-space algorithm C that given some x, with high probability outputs two copies of an output y satisfying $(x, y) \in R$. Let A and B be the algorithms for problem R from Definition 3.3.1. We now show how to amplify algorithm B so that the probability it outputs y is $1 - \frac{1}{4n^2}$. We do this by determining the i-th bit of the output for all $1 \leq i \leq |y|$, where $|y|$ denotes the length of the output y. More specifically, consider the algorithm B' that loops through all i such that $1 \leq i \leq |y|$, and for each index i, runs B at least $\Omega(\log(2n|y|))$ times to determine the most common bit in that position. Because there exists an input y such that $\Pr_r[B(x, t_x, r) = y] \geq \frac{2}{3}$, the most common bit in each position will be the same as the bit of y in that position. Therefore, (after choosing a large enough constant in the Ω) by a Chernoff bound and a union bound over all bits in y, B' will output y with probability at least $1 - \frac{|y|}{8n^3|y|^3}$. Now, an algorithm C for R can do the following: first run A to get t_x, and then run algorithm B' two times. By a union bound, with high probability, the output will be y both times, as the failure probability is bounded by $\frac{2|y|}{8n^3|y|^3}$, so the success probability is high.

Now we show the reverse direction. Consider an algorithm C such that with high probability on input x, C will output two copies of an output y satisfying $(x, y) \in R$. Now we construct algorithms A and B satisfying the conditions of Definition 3.3.1. First, let algorithm A simulate algorithm C, and output the configuration of the Turing machine corresponding to algorithm C after C has outputted one copy of y (that is, after it has outputted the comma between the two y's in (y, y)). This will be our string t_x. The length of t_x will be of size $O(\log n)$ as C is a log-space algorithm. Now, algorithm B will continue simulating algorithm C, starting from configuration t_x. With high probability, C will output another copy of y after reaching configuration t_x, since we know which high probability C outputs the pair same output twice. Therefore, algorithm B will output y with high probability, as desired. □

We note that if a problem R has reproducible solutions, then for any polynomially bounded ℓ it has a randomized logspace algorithm D which on valid input x outputs ℓ copies of a valid output y. That is, it outputs $(y, y, \ldots, y)$, where $(x, y) \in R$. This can be done by first running the algorithm A (from Definition 3.3.1) to create an advice string t_x, and then running algorithm B (from Definition 3.3.1) ℓ times using the same advice string s on all those ℓ executions.

3.3.2 Algorithms with few influential bits

To construct a log-space algorithm whose output is reproducible, we will design an algorithm A whose output with high probability only depends on $O(\log n)$ of the random bits A samples. Then, the algorithm can store those $O(\log n)$ influential random bits, and using those it can reproduce its output by running again using the same $O(\log n)$ influential random bits. Below we give a precise definition of what we mean by "influential random bits".

Definition 3.3.3 (Influential bits)**.** [2] Let $k(n)$ be a polynomial-time computable function. Say that a randomized log-space search algorithm A has $k(n)$ *influential bits* if for all valid inputs x, with probability at least $\frac{1}{2}$ over random strings r_1 of length $k(n)$, we have that there exists an output y such that y is valid for input x and $\Pr_{r_2}[A(x, r_1, r_2) = y] \geq \frac{2}{3}$. Here, r_2 denotes the remaining randomness (after r_1) used by A and $A(x, r_1, r_2)$ denotes the output of A on input x with randomness r_1 and r_2.

We note that a concept closely related to algorithms with few influential bits is k-pseudo-deterministic algorithms, which were introduced by Oded Goldreich [Gol19] after the paper this chapter is based on was first published. A k-pseudo-deterministic algorithm is an algorithm such that for every valid input, there is a set of valid outputs of size up to k such that with high probability the algorithm outputs one a value in this set. An algorithm with b influential bits can be used to construct a 2^b-pseudo-deterministic algorithm. Similarly, k-pseudo-deterministic algorithms can be turned into algorithms with few influential bits, as shown in Theorem 8 in [Gol19].

We now prove that if a randomized log-space algorithm A has $O(\log n)$ influential bits, then its output is reproducible. Essentially, the idea is that the algorithm A can store its $O(\log n)$ influential random bits in memory and then use these bits to recompute its previous output.

[2] An algorithm is pseudo-deterministic if it has zero influential bits. In this sense, the above definition is an extension of pseudo-determinism.

Lemma 3.3.4. *If a search problem R can be solved by a randomized log-space algorithm with $O(\log n)$ influential random bits, then it has log-space reproducible solutions.*

Proof. Let C be an algorithm for the search problem R with $b = O(\log n)$ influential random bits. Let $m = \text{poly}(n)$ be an upper bound on the output size. We will construct algorithms A and B that satisfy the conditions of Definition 3.3.1, i.e., for an input x, A outputs a string t_x of length $O(\log n)$, such that with high probability algorithm B running on input (x, t_x) will pseudo-deterministically produce an output y satisfying $(x, y) \in R$.

First, we will show how to amplify the $\frac{2}{3}$ from Definition 3.3.3. We randomly generate b bits (recall that b is the number of influential random bits used by algorithm C). With probability at least $\frac{1}{2}$, fixing these b bits will cause algorithm C to produce the same output for at least $\frac{2}{3}$ of the choices for the remaining random bits. We can amplify the $\frac{2}{3}$ via repetition. That is, we can create a new algorithm C' where the kth output bit is 0 if after running C a total of cn times (for some constant c), the majority of times the kth output bit was a 0. Otherwise, we set the kth output bit to be a 1. If we let c be sufficiently large, then by a Chernoff bound and a union bound over the coordinates of the output, we have that the whole output of C' will be y with probability at least $1 - \frac{1}{2^n}$.

Now, we describe the algorithms A and B. Algorithm A begins by sampling a random string s_1 of length b bits. Next, algorithm A will test whether s_1 is a "good" string. That is, we test whether with high probability there is some y such that the probability $\Pr_{r_2}[C'(x, s_1, r_2) = y]$ is large (at least $1 - \frac{1}{n^2}$). This can be done by, for each output bit i, running the algorithm C' a total of $\Theta(n^2)$ times, and checking if the i-th output bit was the same in all executions (we remark here that A knows which bits of C' are influential because those are the bits that are sampled first). If $\Pr_{r_2}[C'(x, s_1, r_2) = y]$ is at least $1 - \frac{1}{2^n}$, then s_1 passes this test with probability at least $1 - \frac{\Theta(n^2)}{2^n}$. If the string s_1 passes, we know that with high probability $\Pr_{r_2}[C'(x, s_1, r_2) = y] \geq 1 - \frac{1}{n}$. If it is the case that for each coordinate, C' outputted the same bit on each of the executions, algorithm A can output s_1 as its string t_x. Otherwise, if one of the output bits was not the same on all executions (i.e., s_1 did not pass the test for having a high value of $\Pr_{r_2}[C'(x, s_1, r_2) = y]$), we sample a new string s_1 and repeat. After $O(\log n)$ tries for the string s_1, with high probability we will find a good string s_1, where C' outputs a certain y with high probability. Now, algorithm B can simply simulate algorithm C' on the input (t_x, x), where t_x is the good string that A outputted. Since

with high probability $\Pr_{r_2}[C'(x, s_1, r_2) = y] \geq 1 - \frac{1}{n}$, we know that algorithm B, when run multiple times on (t_x, x), will output the same y with high probability. □

Note that the reduction in 3.3.4 from randomized log-space algorithms with $O(\log n)$ influential bits to reproducible solutions is non-uniform. We needed uniformity in order to let algorithm A know which bits of the original algorithm C were influential.

In the rest of the section, we prove that every problem in search-RL has an algorithm with $O(\log n)$ influential bits:

Theorem 3.3.5. Every problem in search-RL has a randomized log-space algorithm that only has $O(\log n)$ influential bits.

As an immediate corollary of Theorem 3.3.5 and Lemma 3.3.4 we have:

Corollary 3.3.6. *Every problem in search-RL has log-space reproducible solutions.*

3.3.3 High Level Proof Idea for Theorem 3.3.5

At the high level, the idea for the algorithm for Theorem 3.3.5 is as follows. First, we consider the problem Short-Walk Find Path from Definition 3.2.5, which we know is complete for search-RL by Lemma 3.2.6. Now, suppose that we wish to find a path from s to t, and we know that $p_k(s, t) \geq \frac{1}{2}$ (see Definition 3.2.7 for a definition of p_k). This implies that there must exist an outneighbor v of s such that $p_{k-1}(v, t) \geq \frac{1}{2}$. Therefore, if we could estimate $p_{k-1}(v, t)$ for all outneighbors v of s, we could pick the lexicographically first neighbor satisfying $p_{k-1}(v, t) \geq \frac{1}{2}$, and continue recursively from there. Since v is uniquely determined (it is the lexicographically first outneighbor of s satisfying $p_{k-1}(v, t) \geq \frac{1}{2}$), if such an algorithm worked, it would be fully pseudo-deterministic (and hence would have no influential random bits).

Unfortunately, this proposed algorithm of finding an outneighbor will not work. To see why, consider the situation where for the first outneighbor v of s that we check, $p_{k-1}(v, t)$ is exactly equal to $1/2$. Then no matter how accurately we estimate $p_{k-1}(v, t)$, much of the time our estimate will be less than $1/2$, and other times it will be greater than $1/2$. This makes our algorithm not pseudo-deterministic, as in some runs we will use vertex v in the path, and in other runs we will not.

Instead, we construct an algorithm that has logarithmically many influential bits in the following way. We will generate a *threshold* c (from some distribution) and find the first outneighbor v satisfying $p_{k-1}(v,t) \geq 1/2 - c$, and use that vertex v as part of our path. Then, we recurse to find the next vertex in the path. Of course, this still fails if $p_{k-1}(v,t) = 1/2 - c$ (or if $p_{k-1}(v,t)$ is close to $1/2 - c$). However, if the value of c is far away from all values of $1/2 - p_i(u,t)$ for all u and all $1 \leq i \leq k$, then our algorithm, for this fixed value of c, will always give the same output. Hence, to get an algorithm with logarithmically many influential bits for Short-Walk Find Path, we just need a way to use logarithmically many bits to select a value of c such that for all $1 \leq i \leq k$, and vertices v, $|1/2 - p_{k-1}(v,t) - c|$ is large (at least $1/n^5k^5$).

We are able to find such a value of c by sampling it at random from some set of polynomial size. Note that there are kn possible values for an expression of the form $1/2 - p_i(u,t)$ (with $i \leq k$), since there are n options for u, and k options for i, and we need c to be far from all kn of these options. If we were to randomly sample c from the set $\{\frac{1}{k^4n^4}, \frac{2}{k^4n^4}, \ldots, \frac{k^2n^2}{k^4n^4}\}$, then with high probability our chosen value of c would be far away from all of the expressions of the form $1/2 - p_i(u,t)$ (with $i \leq k$), and hence once we fix such a c, we get the same output with high probability. Because we can sample c using $O(\log nk)$ bits, our output will only depend on the first $O(\log nk)$ bits sampled.

3.3.4 Algorithm and Analysis

Here we will state the algorithm for Theorem 3.3.5 more precisely and provide a detailed analysis.

Algorithm 1 Randomized algorithm with $O(\log n)$ influential bits for Short-Walk Find Path on input $(G, s, t, 1^k)$

1: Initialize $u = s$. u is the current vertex.
2: Choose a threshold c from the set $\{\frac{1}{k^4n^4}, \frac{2}{k^4n^4}, \ldots, \frac{k^2n^2}{k^4n^4}\}$ uniformly at random.
3: For $d = k, k-1, \ldots, 1$:
4: Print u (on the output tape).
5: For each outneighbor v of u (in lexicographic order):
6: Estimate $1/2 - p_{d-1}(v,t)$, up to additive error $\frac{1}{k^5n^5}$ (use Lemma 3.2.8). Call the estimate μ.
7: If $\mu \leq c$ then set $u \leftarrow v$, and continue (i.e., return to line 3).

Lemma 3.3.7. *Algorithm 1 runs in randomized log-space, has $O(\log nk)$ influential bits, and with high probability it outputs a path from s to t in expected polynomial time.*

Proof. We first show the algorithm runs in randomized log-space. Then we show the algorithm outputs a path with high probability in polynomial time, and then we show the output has $O(\log nk)$ influential bits.

Runs in randomized log-space: At every point in the algorithm, we must store in memory the value of c (which requires $\log(\text{poly}(n,k)) = O(\log nk)$ bits), the current value of d, which requires $\log k$ bits, and the current vertex u, which requires $\log n$ bits. In addition, in line 6 we estimate the value of $p_{d-1}(v,t)$, which can be done in log-space by Lemma 3.2.8. Hence, the total number of bits needed is $O(\log nk)$, which is logarithmic in the input size.

With high probability outputs a path from s to t in polynomial time: Out of the possible values for c in the set $\{\frac{1}{k^4n^4}, \frac{2}{k^4n^4}, \ldots, \frac{k^2n^2}{k^4n^4}\}$, at most kn of them could satisfy $|1/2 - p_i(v,t) - c| \leq \frac{1}{n^5k^5}$ for some value of $1 \leq i \leq k$ and vertex v (since there are kn possible values of $p_i(v,t)$). We choose such a value with probability at most $\frac{1}{kn}$ (since there are at most kn such "bad" choices, out of k^2n^2 total choices for c). Now, consider the other values of c, which do not satisfy $|1/2 - p_i(v,t) - c| \leq \frac{1}{n^5k^5}$ for any i and v. We now show that with high probability if the if statement in line 7 is satisfied, it is the case that with high probability $1/2 - p_{d-1}(v,t) \leq c$. This is since $1/2 - p_{d-1}(v,t)$ is more than $\frac{1}{k^5n^5}$ away from c, and by Lemma 3.2.8, the estimate for $p_{d-1}(v,t)$ is within $\frac{1}{k^5n^5}$ of the true value of $p_{d-1}(v,t)$ with high probability. Since with high probability $p_d(u,t) \geq 1/2 - c$, vertex u must have an outneighbor v satisfying $p_{d-1}(v,t) \geq 1/2 - c$. Since $p_{d-1}(v,t)$ is further than $\frac{1}{n^5k^5}$ from c, by Lemma 3.2.8, with high probability when reaching the vertex v in the for loop of line 5, in line 7 the if statement will be satisfied. Hence, with high probability, u will change on each iteration of the for loop of line 3, and we maintain that with high probability throughout the algorithm the values of d and u satisfy $p_d(u,t) \geq 1/2 - c$.

Now we show that the algorithm succeeds with high probability. Once again, the probability we choose a c satisfying $|1/2 - p_i(v,t) - c| \leq \frac{1}{n^5k^5}$ for some $1 \leq i \leq k$ and vertex v is at most $\frac{1}{nk}$. The remaining probabilistic parts of the algorithm come from estimating $1/2 - p_i(u,t)$ to an additive error of $\frac{1}{n^5k^5}$. By Lemma 3.2.8, this has error probability at most $2e^{-2nk}$ per estimate. As we make at most nk estimates, the error probability here is bounded by $2nke^{-2nk}$, which is low.

Output has $O(\log nk)$ influential bits: We claim the influential bits used by the algorithm are the bits used to pick c. Out of the values c in the set $\{\frac{1}{k^4n^4}, \frac{2}{k^4n^4}, \ldots, \frac{k^2n^2}{k^4n^4}\}$, at most kn of them could satisfy $|1/2 - p_i(v,t) - c| \leq \frac{1}{n^5k^5}$ for some values of $1 \leq i \leq k$ and vertices v. The probability that we pick such a c is $\frac{nk}{n^2k^2} = \frac{1}{nk}$, so with high probability we do not pick such a c.

Now, for the remaining values of c, the algorithm will have the same output with high probability over the remaining random bits. We note that the only other place where randomness is used is in line 6 to estimate μ. Note that if $1/2 - p_{d-1}(v,t) \leq c$, we also know that $1/2 - p_{d-1}(v,t) \leq c - \frac{1}{n^5k^5}$. Hence, by Lemma 3.2.8, the probability that in this case the if statement in line 7 is not satisfied is at most $2e^{-2nk}$. Similarly, if $1/2 - p_{d-1}(v,t) \geq c$, we know that $1/2 - p_{d-1}(v,t) \geq c + \frac{1}{n^5k^5}$, and so the if statement in line 7 is satisfied with probability at most $2e^{-2nk}$. Hence, with high probability (at least $1 - 2nke^{-2nk}$) the if statement in line 7 is satisfied if and only if $1/2 - p_{d-1}(v,t) \leq c$, and so the output is the same for almost all choices of the remaining random bits. □

Lemma 3.3.7 immediately implies Theorem 3.3.5, completing the proof.

3.3.5 Why we cannot try all possible thresholds

One idea to make the algorithm pseudo-deterministic would be to try every possible value of c (of which there are polynomially many, and therefore can be enumerated), thus removing the randomization required to sample c. This idea will not immediately provide a pseudo-deterministic algorithm, as we explain below.

Consider the approach of going over all possible values of c in some set and choosing the first "good one", i.e. the first value of c which is far from all values of $1/2 - p_i(v,t)$. The problem with such an algorithm is that for a fixed value of c, it may be hard to tell whether it is a "good" value of c. Suppose, for example, that we call a value "good" if it is at distance at least $1/n^2k^2$ from any value of $1/2 - p_i(v,t)$. Then, if the distance is exactly $1/n^2k^2$, it is not clear how one can check if the value of c is good or not. If we simply estimate the values of $1/2 - p_i(v,t)$ and see if out estimates are at distance at least $1/n^2k^2$, we will sometimes choose c, and sometimes we will not (depending on the randomness we use to test whether c is good). Hence, the algorithm will not be pseudo-deterministic, since this value of c will sometimes be chosen, and sometimes a different value of c will be chosen.

3.3.6 Discussion of Algorithm 1

Algorithm 1 has the property that its output, when viewed as a distribution depending on the random bits chosen by the algorithm, has entropy $O(\log n)$. This essentially follows from the fact that the output of Algorithm 1 with very high probability depends on only its first $O(\log n)$ random bits. Hence, after amplifying the success probability, one can show the entropy of the output would be $O(\log n)$. We note that for a pseudo-deterministic algorithm, the output has entropy less than 1. An arbitrary search-RL algorithm can have polynomial entropy.

Another way to view Algorithm 1 is that with high probability, the output will be one of polynomially many options (as opposed to a unique option, which would be achieved by a pseudo-deterministic algorithm). That is, for each input x, there exists a list L_x of polynomial size such that with high probability, the output is in L_x. This follows from the fact that with high probability, the output of Algorithm 1 only depends on its first $O(\log n)$ random bits. Therefore, with high probability, the output will be one of $2^{O(\log n)} = \text{poly}(n)$ different paths. Another way to see this is that with high probability the outputted path depends only on the choice of c, and there are polynomially many (n^2k^2) possible values for c.

3.4 Improved Pseudo-deterministic Algorithms for Path Finding

In this section, we show faster pseudo-deterministic algorithms for both undirected path finding and directed path finding in Eulerian graphs. While both of these problems have been shown to be in deterministic log-space [Rei08, RTV06], our algorithms here have a much lower run-time than those in [Rei08, RTV06].

We note that throughout this section, to compute runtime, instead of dealing with Turing machines, we assume that we can make the following queries in $O(1)$ time: for a vertex v we can query the degree of vertex v, and given a vertex v and an integer i we can query the i-th neighbor of v (if v has fewer than i neighbors, such a query returns $\perp$). In the case of an Eulerian graph, we assume that for a vertex v, we can query the degree of v, its i-th in-neighbor, and its i-th out-neighbor.

3.4.1 Undirected Graphs

In this section, we present the algorithm for undirected graphs. Throughout we assume that we have a graph G (possibly with multiedges or self-loops) with n vertices and m edges, and we wish to find a path from vertex s to vertex t (assuming such a path exists). We will number the vertices from 1 to n, and refer to the k^{th} vertex as "vertex k". The idea for the algorithm is as follows. First, note that checking connectivity *using randomness* in undirected graphs is possible [AKL$^+$79]: if vertices s and t are connected, then a random walk starting from s of length $\tilde{O}(mn)$ will reach reach t with high probability:

Lemma 3.4.1. *Given an undirected or Eulerian graph G and two vertices s and t, there exists a randomized algorithm running in time $\tilde{O}(mn)$ that checks whether there exists a path from s to t, and succeeds with probability $1 - \frac{1}{n^{10}}$.*

A version of Lemma 3.4.1 has been shown in [AKL$^+$79]. For completeness, we include a proof in 3.6.

Now, we proceed to prove Theorem 3.1.2, restated here for convenience.

Theorem (Pseudo-deterministic Path Finding in Undirected graphs in $\tilde{O}(mn^3)$ time, $O(\log n)$ space)**.** Let G be a given undirected graph with n vertices and m edges. Given two vertices s and t of G which are connected, there is a pseudo-deterministic log-space algorithm which outputs a path from s to t. Furthermore, the algorithm runs in time $\tilde{O}(mn^3)$.

For our pseudo-deterministic algorithm for undirected path finding, we use the following approach. We delete vertices of small ID from the graph one at a time (excluding s and t), and check if vertex s is still connected to t. Now, suppose that after deleting vertices $1, 2, 3, \ldots, k-1$, excluding s and t, (recall that we number the vertices from 1 to n, and refer to the ith vertex as "vertex i"), s is still connected to t. However, suppose that when we delete vertices $1, 2, \ldots, k$, vertex s is no longer connected to t. Then we will recursively find paths $s \to k$ and $k \to t$. Repeating this process, we will get a path from $s \to t$.

Of course, as described, this algorithm will not run in log-space (since, for example, one must store in memory which vertices have been removed, as well as the recursion tree, both of which may require large space). With a few modifications though, one can adapt the algorithm to run in log-space. For a complete description of the algorithm, see Algorithm 2.

We use the variables v_{cur} and v_{dest} to denote the vertex our walk is currently on and the vertex which is the destination (at the current level of the recursion).

Algorithm 2 Pseudo-deterministic log-space algorithm for undirected path finding.

1: Use Lemma 3.4.1 to test if s and t are connected. If they are not, return "not connected"
2: Set $v_{cur} = s$.
3: While $v_{cur} \neq t$:
4: Set $v_{dest} = t$
5: For $k = 1, 2, \ldots, n$:
6: If v_{cur} is adjacent to v_{dest} then set $v_{cur} \leftarrow v_{dest}$, print v_{cur} (on the output tape), and go to line 3.
7: If v_{cur} is not connected to v_{dest} in the graph with vertices v_{cur}, v_{dest}, and $k+1, k+2, \ldots, n$ (for a detailed description of the implementation of this step, see the proof of Lemma 3.4.2) then set $v_{dest} \leftarrow k$.

Lemma 3.4.2. *Given a graph G, Algorithm 2 outputs a path from vertex s to t with high probability (if such a path exists), and runs in pseudo-deterministic log-space and time $\tilde{O}(mn^3)$.*

Proof. We begin by providing a more detailed description of the implementation of line 7. Then, we will analyze the algorithm in detail. Specifically, we will show that the algorithm returns a path from s to t with high probability, uses logarithmic space, runs in time $\tilde{O}(mn^3)$, and is pseudo-deterministic.

Description of line 7: In order to check if v_{cur} and v_{dest} are connected, we run a random walk on the graph H with vertices v_{cur}, v_{dest}, and $k+1, k+2, \ldots, n$. To do so, in each step of the random walk, if the walk is currently on v, we pick a random edge (v, u) adjacent to v, and test if the other endpoint u of the edge is in H (this can be done by testing if the ID of u is larger or smaller than k). If it is, the random walk proceeds to u. Otherwise, the random walk remains at v. To analyze the runtime of this walk, we note that such a walk is identical to a random walk on the graph H which is the graph induced by G on the vertices v_{cur}, v_{dest}, and $k+1, k+2, \ldots, n$, along with self loops, where every edge (v, u) where $v \in H$ and $u \notin H$ is replaced by a self loop at v. Since this graph has fewer than n vertices, and at most m edges, by Lemma 3.4.1 Line 7 takes time $\tilde{O}(mn)$.

Returns a path from s to t with high probability: The key claim is that the variable v_{cur} never returns to the same vertex twice, and changes in each iteration of the while loop in line 3.

This implies the success of the algorithm since then after at most n iterations of line 3, v_{cur} must have achieved the value of t at some point.

To prove that v_{cur} never returns to the same vertex twice, and changes in each iteration of the while loop, we consider the "destination sequence" of the vertex $v = v_{cur}$. We define the destination sequence of v to be the sequence of values the v_{dest} variable takes during the period when $v = v_{cur}$. That is, the destination sequence of some vertex $v = v_{cur}$ is $(t, c_1, c_2, \ldots, c_i)$ where c_j is the value of k on the jth time that the if statement in line 7 evaluated to True. Note that the destination sequence is a function of a vertex (i.e., the sequence of values the v_{dest} variable takes during the period when $v = v_{cur}$ depends only on v, assuming line 7 was implemented successfully). It's worth noting that since during the period that $v_{cur} = v$, the value of k only increases, so for a destination sequence $(t, c_1, c_2, \ldots, c_i)$ we have $c_1 < c_2 < \ldots < c_i$.

We note that c_{j+1} is the smallest integer such that on the graph G induces on the vertices v_{cur}, c_j, and $c_{j+1} + 1, c_{j+1} + 2, c_{j+1}, \ldots, n$, the vertex v_{cur} is not connected to c_j

Consider the following total ordering on sequences. A sequence $C = (t, c_1, c_2, \ldots, c_i)$ is larger than $D = (t, d_1, d_2, \ldots, d_i, \ldots, d_j)$ if either $c_\ell = d_\ell$ for all $1 \leq \ell \leq i$ (and $i < j$), or for the first value of ℓ for which $c_\ell \neq d_\ell$, we have $c_\ell > d_\ell$. Otherwise, if $C \neq D$, we say that $C < D$.

We claim that during the algorithm, the destination sequences strictly increase according to the above ordering whenever the value of v_{cur} changes, which happens every time Algorithm 2 returns to Line 3. Note that this would imply that v_{cur} never achieves the same vertex v twice (if it does, that contradicts the fact that the destination sequence of v_{cur} must have increased). Hence, it will suffice to show that the destination sequence of v_{cur} increases according to the above ordering whenever Algorithm 2 returns to Line 3.

Suppose that $u = v_{cur}$. Let the destination sequence of u be $(t, c_1, c_2, \ldots, c_i, u')$, where the next value achieved by v_{cur} is u'. By construction, we have that $c_1 < c_2 < \cdots < c_i < u'$. Also, let the destination sequence of u' be $(t, d_1, d_2, \ldots, d_i, \ldots, d_j)$, where we similarly have that $d_1 < d_2 < \cdots < d_j$. We will show that the destination sequence of u' is larger than that of u under the ordering defined above.

First, we claim that $c_\ell = d_\ell$ for $1 \leq \ell \leq i$. We can show this by induction. We first show that $c_1 = d_1$. Indeed, we have that $d_1 \leq c_1$ because deleting vertices $1, 2, \ldots, c_1$ (which doesn't include u') from the graph will disconnect u' from t, as u and u' are adjacent. To show that $d_1 \geq c_1$, we

show that there is a path from u' to t that doesn't use any vertices (other than maybe t) with labels less than c_1. Indeed, deleting vertices $1, 2, \ldots, u'$ disconnected u from c_i, but deleting $1, 2, \ldots, u'-1$ didn't, so there is a path from u' to c_i that doesn't use any vertices (other than c_i) with labels less that u'. Similarly, for any $2 \leq j \leq i$, deleting vertices $1, 2, \ldots, c_j$ disconnected u from c_{j-1}, but deleting $1, 2, \ldots, c_j - 1$ didn't. Therefore, there is a path from c_j to c_{j-1} that doesn't use any vertices (other than c_{j-1}) with labels less than c_j. Finally, there is a path from c_1 to t that doesn't use any vertices (other than maybe t) with labels less than c_1. By merging all these paths together at the endpoints, we get a path from u' to t doesn't use any vertices with labels less than c_1, as desired. This implies $c_1 \leq d_1$. Combining this with $c_1 \geq d_1$ which we showed above, we now have $c_1 = d_1$. Now, for some integer $p < i$, assume by induction that $c_\ell = d_\ell$ for all $1 \leq \ell \leq p$. We want to show that $c_{p+1} = d_{p+1}$. This can be shown using the exact same argument for showing that $c_1 = d_1$, with t replaced by c_p. Therefore, we have that $d_{p+1} = c_{p+1}$. Now, by the same argument again (with t replaced by c_i), we can show that either $d_{i+1} > u'$, or d_{i+1} doesn't exist. The former occurs when u' and c_i aren't adjacent, and the latter happens when they are. Thus, the destination sequence of u' is larger than that of u under the ordering: all the first i entries stay the same, and we either delete the last entry, or make it larger. Thus, v_{cur} never returns to the same vertex. Hence, it must eventually reach t, proving that the algorithm returns a path from s to t.

To show that the algorithm succeeds with high probability, note that the only lines which are probabilistic are lines 1 and 7. For each execution of these lines, it has failure probability at most $\frac{1}{n^{10}}$ by Lemma 3.4.1. Each of these lines is run at most n^3 times, so the total failure probability is bounded by $O\left(\frac{1}{n^7}\right)$.

Uses $O(\log n)$ space: At every point in the algorithm, the following is stored: v_{cur}, v_{dest}, and k (all of which require space $O(\log n)$). In addition, in line 7 the algorithm will run a random walk, which will require storing the id of the current vertex, as well as a counter storing how many steps of the random walk have been executed. Both of these can be stored using logarithmic space.

Runs in time $\tilde{O}(mn^3)$: v_{cur} can take at most n different values and with high probability does not take the same value more than once (see the paragraph above on why the algorithm returns a path with high probability for a proof of this fact). Since v_{cur} changes its value in each iteration of

the while loop, line 3 executes at most n times. Line 5 runs at most n times. Line 6 is checkable in $O(\log n)$ time, and line 7 runs in time $\tilde{O}(mn)$ as shown earlier in the proof (as part of the description of line 7). Therefore, the total runtime is $\tilde{O}(n \times n \times mn) = \tilde{O}(mn^3)$, as desired.

Is pseudo-deterministic: Randomness is only used in lines 1 and 7, and this is only for checking connectivity. Testing connectivity is a pseudo-deterministic protocol since given two vertices, with high probability when testing for connectivity twice, the same result will be output (namely, if the two vertices are connected, with high probability the algorithm will output that they are connected in both runs. If the two vertices are not connected, with high probability the algorithm will output that they are not connected in both runs). Since all uses of randomization is for checking connectivity in a pseudo-deterministic fashion, the algorithm as a whole is pseudo-deterministic. □

3.4.2 Eulerian Graphs

In this section, we show an efficient pseudo-deterministic log-space algorithm for finding paths in Eulerian graphs (directed graphs such that for every vertex v, the indegree and outdegree of v are equal). Recall that in our model of computation, for a vertex v, we can query either the degree of v, the i-th in-neighbor of v, or the i-th out-neighbor of v in $O(1)$ time. We will prove Theorem 3.1.3, repeated below for convenience:

Theorem (Pseudo-deterministic Path Finding in Eulerian graphs in $\tilde{O}(m^5n^3)$ time, $O(\log n)$ space)**.** Given an Eulerian graph G and two vertices s and t where there exists a path from s to t, there is a pseudo-deterministic log-space algorithm which outputs a path from s to t. Furthermore, the algorithm runs in time $\tilde{O}(m^5n^3)$.

The algorithm will be a variation on the algorithm for undirected graphs of Subsection 3.4.1.

First, note that as in the case with undirected graphs, checking connectivity in Eulerian graphs can be done efficiently using a randomized algorithm (see Lemma 3.4.1). We first would like to note that the algorithm for undirected graphs doesn't immediately generalize to the Eulerian case. This is because the algorithm for undirected graphs involves checking for connectivity on the graph G with some vertices (and their adjacent edges) removed. The reason it is possible to check connectivity in this modified graph is that after removing vertices, the resulting graph is still undirected. However, in the Eulerian case, removing vertices along with their edges may result in a non-Eulerian graph.

So, instead of removing vertices from the graph, we instead remove directed cycles in the graph. One of the key observations is that when deleting a cycle, the resulting graph is still Eulerian, so we can apply Lemma 3.4.1 to test for connectivity.

At the high level, the algorithm proceeds as follows: we remove directed cycles from the graph and check (using randomization) whether vertex t can be reached from vertex s. If it can, we continue removing cycles. If not, then we recursively try to go from vertex s to some vertex on the cycle whose deletion disconnects vertex s and t (call this cycle C). After we find a path to some vertex on the cycle C, note that there exists a vertex v on C such that deleting C does not disconnect v from t. So we then walk on the cycle to v, and then recursively apply the algorithm to find a path from v to t.

As described, this algorithm will not work in log-space, because it is not clear how the algorithm can store in memory a description of which cycles have been deleted. The following lemma provides us with a way to delete cycles in a specified way, so we can compute in log-space whether an edge is part of a deleted cycle or not.

Lemma 3.4.3. *Let E be the set of edges of G. There exists a log-space-computable permutation $f : E \to E$ that satisfies the following property: if e is an inedge of vertex v, then $f(e)$ is an outedge of v. In particular, this condition implies for any edge e, we have that $e, f(e), f^2(e), \ldots$ forms a cycle in G.*

Proof. Take a vertex v of indegree d, and take some ordering of its inedges $e_1^{\text{in}}, e_2^{\text{in}}, \ldots, e_d^{\text{in}}$ (say, in lexicographic order), and some ordering of the outedges $e_1^{\text{out}}, e_2^{\text{out}}, \ldots, e_d^{\text{out}}$ (say again, in lexicographic order). Then simply set $f(e_i^{\text{in}}) = e_i^{\text{out}}$. □

Note that in our model, computing $f(e)$ takes $O(n)$ time for each edge e. In particular, for a cycle C_i formed by repeatedly applying f to some edges, and an edge e in the cycle, computing the next edge in the cycle takes time $O(n)$.

The importance of the lemma is that it provides us with a way to delete cycles in some order: we begin from the "smallest" edge (in whatever ordering) and delete the cycle associated with that edge. Then, we pick the second smallest edge, and delete its cycle, etc. When executing this algorithm, we may try to delete a cycle multiple times, since multiple edges correspond to the same cycle, but this will not be an issue.

See Algorithm 3 for a precise description of the algorithm. As in the undirected case, we use the variables v_{cur} and v_{dest} to denote the vertex our walk is currently on and the current destination. We let the set of e_i be the edges in G (we denote the set of edges of G as E), and let C_i be the cycle $(e_i, f(e_i), f^2(e_i), \ldots, e_i)$ in G. We note that it is possible for C_i and C_j to have the same set of edges for $i \neq j$ (this will not affect the correctness of the algorithm).

Algorithm 3 Pseudo-deterministic log-space algorithm for path finding in Eulerian digraphs.

1: Set $v_{cur} = s$. Write v_{cur} on the output tape.
2: While $v_{cur} \neq t$:
3: Set $v_{dest} = t$.
4: For $k = 1 \ldots m$:
5: If v_{cur} and v_{dest} are not connected using only edges in $E \setminus \{C_1, \ldots, C_k\}$ (more details of the implementation of this step are in the body of the paper in the proof of Lemma 3.4.4) then
6: If $v_{cur} \in C_k$:
7: Find a vertex $v \in C_k$ such that v and v_{dest} are connected using edges only in $E \setminus \{C_1, \ldots, C_k\}$.
8: Walk on C_k from v_{cur} until you reach v, print the vertices on this path (on the output tape).
9: Set $v_{cur} \leftarrow v$, return to top of while loop.
10: Else:
11: Find a vertex $v \in C_k$ such that v_{cur} can get to v in $G \setminus \{C_1, \ldots, C_k\}$
12: Set $v_{dest} \leftarrow v$.

Lemma 3.4.4. *Algorithm 3 runs in time $\tilde{O}(m^5 n^3)$, is pseudo-deterministic, uses logarithmic space, and outputs a path from s to t with high probability.*

Proof. We first give a more detailed description of the implementation of line 5. Then, we analyze the algorithm in steps, first showing that it returns a path with high probability, and then showing that it runs in pseudo-deterministic log-space with runtime $\tilde{O}(m^5 n^3)$. The proof closely follows the approach of the proof in the undirected case.

Implementation of line 5: Below, we describe the details of the implementation of step 5. As done in the proof of Lemma 3.4.1 in Appendix 3.6, we check connectivity by making every edge in the graph G undirected, and then perform a random walk on the undirected graph G. The key difficulty is to ensure that we can check if an edge is in one of the deleted cycles efficiently in log-space.

Say that we are on vertex u, and the randomly chosen neighbor which is next in the random

walk is v. Let e be the edge between u and v. We wish to check whether e is in any of the cycles $C_1, C_2, \ldots, C_k$. To check whether edge e is on the cycle C_i, we can check whether e is any of the edges $e_i, f(e_i), f^2(e_i), \ldots e_i$, which can all be computed in log-space. To see if e is on any of the cycles $C_1, \ldots, C_k$, we check if e is in each of the C_i. Each such check takes time $O(mn)$ (since the cycle is of length $O(m)$, and given an edge e, computing the next edge in the cycle takes time $O(n)$). Hence, in total it takes time $O(kmn)$.

Now, if e is on one of the cycles, the random walk stays at u, and otherwise the random walk proceeds to v. It is clear that this is equivalent to taking a random walk on the graph G', where G' is the graph G but with all edges (u', v') in at least one of the cycles $C_1, \ldots, C_k$ replaced with a self-loop at u' (since, if such an edge is chosen, the random walk stays at u'. As G' still has at most m edges and n vertices, checking connectivity takes $\tilde{O}(mn)$ time by Lemma 3.4.1.

Returns a path with high probability: As in the undirected case, the main claim is that v_{cur} never repeats a vertex on two different iterations of the while loop of line 2. To prove that v_{cur} is never repeated, we will use the notion of the "destination sequence", similar to the undirected case. We note that our definition here of a destination sequence is different from the definition in the undirected case. We say that the associated destination sequence to v_{cur} is the sequence of cycles $(C_{i_1}, C_{i_2}, \ldots, C_{i_k})$, where we add C_{i_j} to the sequence if the if statement of step 5 of the algorithm was true when $k = i_j$. Note that this implies that $i_1 \leq i_2 \leq \cdots \leq i_k$.

Now, as in the proof of Lemma 3.4.2, we give a total ordering on all destination sequences. Consider two destination sequences $i = (C_{i_1}, C_{i_2}, \ldots, C_{i_k})$ and $j = (C_{j_1}, C_{j_2}, \ldots, C_{j_k}, \ldots, C_{j_m})$. Say that i is greater than j if either $i_\ell = j_\ell$ for all $1 \leq \ell \leq k$ (and $k < m$), or for the smallest value of ℓ such that $i_\ell \neq j_\ell$, we have that $i_\ell > j_\ell$. Otherwise, if $i \neq j$, then say that $i < j$.

Now, we proceed to prove that v_{cur} never repeats a value. Say that v_{cur} is set to v'_{cur} after one loop of line 2. Let the destination sequence of v_{cur} be $i = (C_{i_1}, C_{i_2}, \ldots, C_{i_k})$ and let the corresponding destination sequence of v'_{cur} be $(C_{j_1}, C_{j_2}, \ldots, C_{j_m})$. First, we claim that $i_p = j_p$ for all $1 \leq p \leq k-1$. This is because v_{cur} is connected to v'_{cur} via cycle C_{i_k}, so deleting C_{i_p} disconnects v_{cur} and v_{dest} if and only if it disconnects v'_{cur} and v_{dest}. As in the proof of Lemma 3.4.2, we have two situations now. One case is that $v'_{cur} \in C_{i_{k-1}}$, and therefore, $m = k - 1$ (the destination sequence for v'_{cur} is one shorter than that of v_{cur}). The other is that $v'_{cur} \notin C_{i_{k-1}}$, and therefore, $j_k \geq i_k$, as

deleting C_{i_k} doesn't disconnect v'_{cur} and v_{dest} by the condition of line 7 of the algorithm. So the destination sequence of v'_{cur} is greater than that of v_{cur} under the total ordering described above, which implies that v_{cur} can never repeat a vertex.

To see that the algorithm succeeds with high probability, note that the only randomness is in lines 5, 7, and 11 for checking connectivity between two vertices. We will check connectivity at most $O(nm^2)$ times, so the failure probability is bounded by $\frac{nm^2}{n^{10}}$ by Lemma 3.4.1, as desired.

Uses $O(\log n)$ space: The information our algorithm needs to store is: s, t, v_{cur}, v_{dest}, and k. After that, by Lemma 3.4.3, we can compute whether an edge e is part of a cycle C_k is log-space, and testing whether two vertices are connected in an Eulerian graph can be done in (randomized) log-space by Lemma 3.4.1. Therefore, everything can be implemented in log-space.

Runs in time $\tilde{O}(m^5n^3)$: Line 2 repeats at most n times since v_{cur} never repeats, and Line 4 repeats at most m times since there are m possible values for k. Due to Lemma 3.4.1, each execution of Line 5 takes $\tilde{O}(mn) \times O(m^2n)$ time, where the $O(m^2n)$ comes from having to check whether each edge we try to use comes from one of the cycles $C_1, \ldots, C_k$ (a factor m from the fact that there are up to m cycles C_i, a factor m from the fact that the size of each C_i is at most m, and a factor $O(n)$ because given some edge e in C_i, computing the next edge in the cycle takes time $O(n)$). Line 7 and 11 take time $O(m) \times \tilde{O}(mn) \times O(m^2n)$, for the same reason as above, except with the extra $O(m)$ factor for having to check all vertices on the cycle C_k. Therefore, our runtime bound is $O(n) \times O(m) \times (\tilde{O}(mn) \times O(m^2) + O(m) \times \tilde{O}(mn) \times O(m^2n)) = \tilde{O}(m^5n^3)$, as desired.

Is pseudo-deterministic: The only randomness is used to check connectivity between pairs of vertices. As each of these checks succeeds with high probability, this clearly implies that our randomness used will not affect the output of the algorithm, since if the two vertices tested are connected, with high probability the same result (of "accept") will be outputted, and if the two vertices tested are not connected, with high probability the same result (of "reject") will be outputted. □

3.5 Discussion

The main problem left open is that of search-RL vs pseudo-deterministic-L:

Problem 3.5.1. Can every problem in search-RL be solved pseudo-deterministically in RL?

A notable open problem in complexity is whether NL equals UL. It is known that under randomized reductions, with *two way access* to the random bits, NL is reducible to UL (in fact, it is reducible to UL $\cap$ coUL) [RA00]. It is not known whether NL is reducible to UL when given one-way access to the random bits. A reproducible reduction from NL to UL would imply such a result, giving us the following problem:

Problem 3.5.2. Does there exist a reproducible log-space reduction from NL to UL?

Another interesting problem would be to fully derandomize the pseudo-deterministic algorithms we present for undirected and Eulerian connectivity, in order to get deterministic log-space algorithms which work in low polynomial time.

Problem 3.5.3. Does there exists a deterministic log-space algorithm for undirected connectivity (or connectivity in Eulerian graphs) using low time complexity?

There are several natural extensions of the notion of reproducibility to the time-bounded setting, some which may be worth exploring. A noteworthy extension is that of low-entropy output algorithms. Our algorithm for search-RL has the property that its output, when viewed as a random variable depending on the random choices of the algorithm, has $O(\log n)$ entropy. It may be interesting to understand such algorithms in the context of time-bounded computation.

Problem 3.5.4. Let search-BPP($\log n$) be the set of problems solvable by randomized polynomial time machines, whose outputs (when viewed as random variables depending on the random choices of the algorithms) have $O(\log n)$ entropy. What is relationship between search-BPP($\log n$) and search-BPP? What is the relationship between search-BPP($\log n$) and pseudo-deterministic-BPP?

3.6 Testing Connectivity for Undirected and Eulerian graphs in RL

In this section, we prove Lemma 3.4.1, repeated below for convenience:

Lemma. *Given an undirected or Eulerian graph G and two vertices s and t, there exists a randomized algorithm running in time $\tilde{O}(mn)$ that checks whether there exists a path from s to t, and succeeds with probability $1 - \frac{1}{n^{10}}$.*

Proof. We begin by showing that for an Eulerian graph G, if there is an edge from vertex u to vertex v, then there is also a path from vertex v to vertex u. Let V_v be the set of vertices reachable from v. Note that the number of edges incoming to V_v must be the same as the number of edges going out of V_v. However, by the definition of V_v, there cannot be edges leaving the set (if there is a an edge (v', u') where $v' \in V_v$ and $u' \notin V_v$, then u' can be reached from v, and hence $u' \in V_v$, a contradiction). Hence, since there are no outgoing edges, there are also no incoming edges. Hence, since (u, v) has one endpoint in V_v, both endpoints must be in V_v, so $u \in V_v$ is reachable from v.

Hence, in order to test reachability in Eulerian graphs, it is enough to test reachability in the undirected graph defined by making all edges of the Eulerian graph undirected. Hence, it suffices to prove the lemma for undirected graphs.

The expected number of steps needed to get to vertex t after starting a random walk at vertex s where there is a path from s to t is bounded by $2mn$ [AKL$^+$79]. By Markov's inequality, the probability that a random walk of length $4mn$ starting at vertex s doesn't reach vertex t is at most $\frac{1}{2}$. Therefore, starting at vertex s and repeating $O(\log n)$ random walks of length $4mn$ provides the result. It is easy to check that in our model, taking one step in a random walk takes time $O(\log n)$. □

3.7 Short-Walk Find Path is complete for search-RL

In this section, we prove Lemma 3.2.6, which states that Short-Walk Find Path is complete for search-RL. We repeat the definition of Short-Walk Find Path below for convenience, and then we proceed to prove that it is complete for search-RL.

Definition 3.7.1 (Short-Walk Find Path)**.** Let R be the search problem whose valid inputs are $x = (G, s, t, 1^k)$ where G is a directed graph, s and t are two vertices of G, and a random walk of length k from s reaches t with probability at least $1 - 1/|x|$. On such an x, a valid output is a path of length up to poly(k) from s to t.

We now prove that Short-Walk Find Path is complete for search-RL. The definition of reductions in the context of search-RL is given in [RTV06].

Proof. First, it is easy to see that Short-Walk Find Path is in search-RL, as we can just take a random walk starting from s of length k.

Now we show that Short-Walk Find Path is search-RL-hard via a reduction from Poly-Mixing Find Path. In [RTV06] Section A.3 (proof of Theorem 3.1), it is shown that Poly-Mixing Find Path with input $(G, s, t, 1^k)$ is complete for search-RL (we refer the reader to [RTV06] for a formal definition of Poly-Mixing Find Path. Intuitively speaking, the input to the problem is a graph with low enough mixing time, two vertices s and t, and 1^k, and the goal is to find a path of length k from s to t). They also state that a path of length $m = 2k \log k$ starting from s reaches t with probability at least $\frac{1}{2k}$ in the problem Poly-Mixing Find Path. Now, we amplify this probability of $\frac{1}{2k}$ by constructing a new graph. To do this, consider a graph G' which is made as follows: it has $(m+1)|V(G)|$ vertices, each of which is a pair (i, v) for $0 \leq i \leq m$ and vertex $v \in G$. If the edge $u \to v$ is in G then add edges $(i, u) \to (i+1, v)$ for $0 \leq i \leq m-1$ in G'. Finally, create edges $(m, v) \to (0, s)$ for all $v \neq t$, and add only a self-loop to the vertex (m, t) (so if a random walk reaches (m, t), the random walk will stay there forever). Then, it is easy to see that a random walk of length $\ell = m + 2(m+1)k \log x$ starting at $(0, s)$ will end at (m, t) with probability at least $1 - \left(1 - \frac{1}{2k}\right)^{2k \log x} \geq 1 - \frac{1}{x}$. Choosing x larger than the length of the input gives the desired reduction. That is, when choosing such an x, given a solution to the Short-Walk Find Path, we can output a polynomially long list $y_1, y_2, \ldots, y_p$ such that at least one of the y_i is a solution to the Poly-Mixing Find Path instance. Therefore, Short-Walk Find Path is complete for search-RL. □

Chapter 4

Psuedo-deterministic Streaming

The work in this Chapter is based on joint work with Shafi Goldwasser, Sidhanth Mohanty, and David Woodruff. [GGMW20].

4.1 Introduction

Consider some classic streaming problems: heavy hitters, approximate counting, ℓ_p approximation, finding a nonzero entry in a vector (for turnstile algorithms), counting the number of distinct elements in a stream. These problems were shown to have low-space randomized algorithms in [CCFC04, Mor78, Fla85, AMS99, IW05, MW10], respectively. All of these algorithms exhibit the property that when running the algorithm multiple times on the same stream, different outputs may result on the different executions.

For the sake of concreteness, let's consider the problem of ℓ_2 approximation: given a stream of poly(n) updates to a vector (the vector begins as the zero vector, and updates are of the form "increase the i^{th} entry by 1" or "decrease the j^{th} entry by 1"), output an approximation of the ℓ_2 norm of the vector. There exists a celebrated randomized algorithm for this problem [AMS99]. This algorithm has the curious property that running the same algorithm multiple times on the same stream may result in different approximations. That is, if Alice runs the algorithm on the same stream as Bob (but using different randomness), Alice may get some approximation of the ℓ_2 norm (such as 27839.8), and Bob (running the same algorithm, but with your own randomness) may get a different approximation (such as 27840.2). The randomized algorithm has the guarantee that both

of the approximations will be close to the true value. However, interestingly, Alice and Bob end up with slightly different approximations. Is this behavior inherent? That is, could there exist an algorithm which, while being randomized, for all streams with high probability both Alice and Bob will end up with the *same* approximation for the ℓ_2 norm?

Such an algorithm, which when run on the same stream multiple times outputs the same output with high probability is called *pseudo-deterministic*. The main question we tackle in this paper is:

What streaming problems have low-memory pseudo-deterministic algorithms?

4.1.1 Our Contributions

This paper is the first to investigate pseudo-determinism in the context of streaming algorithms. We show certain problems have pseudo-deterministic algorithms substantially faster than the optimal deterministic algorithm, while other problems do not.

Lower Bounds

Find-Support-Elem: We show pseudo-deterministic lower bounds for finding a nonzero entry in a vector in the turnstile model. Specifically, consider the problem Find-Support-Elem of finding a nonzero entry in a vector for a turnstile algorithm (the input is a stream of updates of the form "increase entry i by 1" or "decrease entry j by 1", and we wish to find a nonzero entry in the final vector). We show this problem does not have a low-memory pseudo-deterministic algorithm:

Theorem 4.1.1. There is no pseudo-deterministic algorithm for Find-Support-Elem which uses $\tilde{o}(n)$ memory.

This is in contrast with the work of [MW10], which shows a randomized algorithm for the problem using polylogarithmic space.

Theorem 4.1.1 can be viewed as showing that any low-memory algorithm A for Find-Support-Elem must have an input x where the output $A(x)$ (viewed as a random variable depending on the randomness used by A) must have at least a little bit of entropy. The algorithms we know for Find-Support-Elem have a very high amount of entropy in their outputs (the standard algorithms, for an input which is the all 1s vector, will find a uniformly random entry). Is this inherent, or can the entropy of the output be reduced? We show that this is inherent: for every low memory algorithm there is an input x such that $A(x)$ has high entropy.

Theorem 4.1.2. Every randomized algorithm for FIND-SUPPORT-ELEM using $o(s)$ space must have an input x such that $A(x)$ has entropy at least $\log\left(\frac{n}{s \log n}\right)$.

So, in particular, an algorithm using $n^{1-\varepsilon}$ space must have outputs with entropy $\Omega(\log n)$, which is maximal up to constant factors.

We also show analogous lower bounds for the problem FIND-DUPLICATEin which the input is a stream of $3n/2$ integers between 1 and n, and the goal is to output a number k which appears at least twice in the stream:

Theorem 4.1.3. Every randomized algorithm for FIND-DUPLICATE using $o(s)$ space must have an input x such that $A(x)$ has entropy at least $\log\left(\frac{n}{s \log n}\right)$.

Techniques To prove a pseudo-deterministic lower bound for FIND-SUPPORT-ELEM, the idea is to show that if a pseudo-deterministic algorithm existed for FIND-SUPPORT-ELEM, then there would also exist a pseudo-deterministic one-way communication protocol for the problem ONE-WAY-FIND-DUPLICATE, where Alice has a subset of $[n]$ of size $3n/4$, and so does Bob, and they wish to find an element which they share.

To prove a lower bound on the one-way communication problem ONE-WAY-FIND-DUPLICATE, we show that if such a pseudo-deterministic protocol existed, then Bob can use Alice's message to recover many ($n/10$) elements of her input (which contains much more information than one short message). The idea is that using Alice's message, Bob can find an element they have in common. Then, he can remove the element he found that they have in common from his input, and repeat to find another element they have in common (using the original message Alice sent, so Alice does not have to send another message). After repeating $n/10$ times, he will have found many elements which Alice has.

It may not be immediately obvious where pseudo-determinism is being used in this proof. The idea is that because the algorithm is pseudo-deterministic, the element which Bob finds as the intersection with high probability does not depend on the randomness used by Alice. That is, let $b_1, b_2, \ldots$ be the sequence of elements which Bob finds. Because the algorithm is pseudo-deterministic, there exists a specific sequence $b_1, b_2, \ldots$ such that with high probability this will be the sequence of elements which Bob finds. Notice that a randomized (but not pseudo-deterministic) algorithm for ONE-WAY-FIND-DUPLICATEwould result in different sequences on different executions.

When the sequence $b_1, b_2, \ldots$ is determined in advance, we can use a union bound and argue that with high probability, one of Alice's messages will likely work on all of Bob's inputs. If $b_1, b_2, \ldots$ is not determined in advance, then it's not possible to use a union bound.

Proving a lower bound on the entropy of the output of an algorithm for FIND-SUPPORT-ELEM uses a similar idea, but is more technically involved. It is harder to ensure that Bob's later inputs will be able to succeed with Alice's original message. The idea, at a very high level, is to have Alice send many messages (but not too many), so that Bob's new inputs will not strongly depend on any part of Alice's randomness, and also to have Alice send additional messages to keep Bob from going down a path where Alice's messages will no longer work.

This lower bound technique may seem similar to the way one would show a deterministic lower bound. It's worth noting that for certain problems, deterministic lower bounds do not generalize to pseudo-deterministic lower bounds; see our results on pseudo-deterministic upper bounds for some examples and intuition for why certain problems remain hard in the pseudo-deterministic setting while others do not.

Sketching lower bounds for pseudo-deterministic ℓ_2 norm estimation: The known randomized algorithms (such as [AMS99]) for approximating the ℓ_2 norm of a vector x in a stream rely on *sketching*, i.e., storing $\boldsymbol{S}x$ where $\boldsymbol{S}$ is a $d \times n$ random matrix where $d \ll n$ and outputting the ℓ_2 norm of $\boldsymbol{S}x$. More generally, an abstraction of this framework is the setting where one has a distribution over matrices $\mathcal{D}$ and a function f. One then stores a *sketch* of the input vector $\boldsymbol{S}x$ where $\boldsymbol{S} \sim \mathcal{D}$ and outputs $f(\boldsymbol{S}x)$. By far, most streaming algorithms fall into this framework and in fact some work [LNW14, AHLW16] proves under some caveats and assumptions that low-space turnstile streaming algorithms imply algorithms based on low-dimensional sketches. Since sketching-based streaming algorithms are provably optimal in many settings, it motivates studying whether there are low-dimensional sketches of x from which the ℓ_2 norm can be estimated pseudo-deterministically.

We prove a lower bound on the dimension of sketches from which the ℓ_2 norm can be estimated pseudo-deterministically:

Theorem 4.1.4. Suppose $\mathcal{D}$ is a distribution over $d \times n$ matrices and f is a function from $\mathbb{R}^d$ to $\mathbb{R}$ such that for all $x \in \mathbb{R}^n$, when $\boldsymbol{S} \sim \mathcal{D}$:

- $f(\boldsymbol{S}x)$ approximates the ℓ_2 norm of x to within a constant factor with high probability,

- $f(\boldsymbol{S}x)$ takes a unique value with high probability.

Then d must be $\Omega(n)$.

As an extension, we also show that

Theorem 4.1.5. For every constant $\varepsilon, \delta > 0$, every randomized sketching algorithm A for ℓ_2 norm estimation using a $O(n^{1-\delta})$-dimensional sketch, there is a vector x such that the output entropy of $A(x)$ is at least $1 - \varepsilon$. Furthermore, there is a randomized algorithm using a $O(\operatorname{poly}\log n)$-dimensional sketch with output entropy at most $1 + \varepsilon$ on all input vectors.

Techniques The first insight in our lower bound is that if there is a pseudo-deterministic streaming algorithm A for ℓ_2 norm estimation in k space, then that means there is a fixed function g such that $g(x)$ approximates $\|x\|_2$ and A is a randomized algorithm to compute $g(x)$ with high probability. The next step uses a result in the work of [HW13] to illustrate a (randomized) sequence of vectors $\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}$ only depending on g such that any linear sketching-based algorithm that uses sublinear dimensional sketches outputs an incorrect approximation to the ℓ_2 norm of some vector in that sequence with constant probability, thereby implying a dimension lower bound.

Upper Bounds

On the one hand, all the problems considered so far were such that

1. There were "low-space" randomized algorithms.
2. The pseudo-deterministic and deterministic space complexity were "high" and equal up to logarithmic factors.

This raises the question if there are natural problems where pseudo-deterministic algorithms outperform deterministic algorithms (by more than logarithmic factors). We answer this question in the affirmative.

We illustrate several natural problems where the pseudo-deterministic space complexity is strictly smaller than the deterministic space complexity.

The first problem is that of finding a nonzero row in a matrix given as input in a turnstile stream. Our result for this problem has the bonus of giving a natural problem where the pseudo-

deterministic streaming space complexity is strictly sandwiched between the deterministic and randomized streaming space complexity.

In the problem Find-Nonzero-Row, the input is an $n \times d$ matrix A streamed in the turnstile model, and the goal is to output an i such that the i^{th} row of the matrix A is nonzero.

Theorem 4.1.6. The randomized space complexity for Find-Nonzero-Row is $\widetilde{\Theta}(1)$, the pseudo-deterministic space complexity for Find-Nonzero-Row is $\widetilde{\Theta}(n)$, and the deterministic space complexity for Find-Nonzero-Row is $\widetilde{\Theta}(nd)$.

The idea behind the proof of Theorem 4.1.6 is to sample a random vector x, and then deterministically find a nonzero entry of Ax. With high probability, if a row of A is nonzero, then the corresponding entry of Ax will be nonzero as well.

Discussion: Roughly speaking, in this problem there is a certain structure that allows us to use randomness to "hash" pieces of the input together, and then apply a deterministic algorithm on the hashed pieces. The other upper bounds we show for pseudo-deterministic algorithms also have a structure which allows us to hash, and then use a deterministic algorithm. It is interesting to ask if there are natural problems which have faster pseudo-deterministic algorithms than the best deterministic algorithms, but for which the pseudo-deterministic algorithms follow a different structure.

The next problems we show upper bounds for are estimating frequencies in a length-m stream of elements from a large universe $[n]$ up to error εm, and that of estimating the inner product of two vectors x and y in an insertion-only stream of length-m up to error $\varepsilon \cdot \|x\|_1 \cdot \|y\|_1$. We show a separation between the deterministic and (weak) pseudo-deterministic space complexity in the regime where $m \ll n$.

Theorem 4.1.7. There is a pseudo-deterministic algorithm for point query estimation and inner product estimation that uses $O\left(\frac{\log m}{\varepsilon} + \log n\right)$ bits of space. On the other hand, any deterministic algorithm needs $\Omega\left(\frac{\log n}{\varepsilon}\right)$ bits of space.

4.1.2 Related work

The problem of finding duplicates in a stream of integers between 1 and n was first considered by [GR09], where an $O(\log^3 n)$ bits of space algorithm is given, later improved by [JST11] to

$O(\log^2 n)$ bits. We show that in contrast to these low space randomized algorithms, a pseudo-deterministic algorithm needs significantly more space in the regime where the length of the stream is, say, $3n/2$. [KNP+17] shows optimal lower bounds for randomized algorithms solving the problem.

The method of ℓ_p-sampling to sample an index of a turnstile vector with probability proportional to its ℓ_p mass, whose study was initiated in [MW10], is one way of outputting an element from the support of a turnstile stream. A line of work [FIS08, MW10, JST11, AKO10], ultimately leading to an optimal algorithm in [JW18] and tight lower bounds in [KNP+17], characterizes the space complexity of randomized algorithms to output an element from the support of a turnstile vector as $\Theta(\text{poly} \log n)$, in contrast with the space lower bounds we show for algorithms constrained to a low entropy output.

4.1.3 Open Problems

ℓ_2-norm estimation: In this work, we show that there are no low-dimensional pseudo-deterministic sketching algorithms for estimating the ℓ_2-norm of a vector. However, we do not show a turnstile streaming lower bound for pseudo-deterministic algorithms, which motivates the following question. Does there exist a $O(\text{poly} \log n)$ space pseudo-deterministic algorithm for ℓ_2-norm estimation?

Multi-pass streaming lower bounds: All the streaming lower bounds we prove are in the single pass model, i.e., where the algorithm receives the stream exactly once. How do these lower bounds extend to the multi-pass model, where the algorithm receives the stream multiple times? All of the pseudo-deterministic streaming lower bounds in this paper do not even extend to 2-pass streaming algorithms.

4.1.4 Table of complexities

In the below table, we outline the known space complexity of various problems considered in our work.

4.2 Preliminaries

For the purposes of this work, we define a simple notion that we call a *k-concentrated algorithm.*

Problem	Randomized	Deterministic	Pseudo-deterministic
Morris Counters	$\Theta(\log\log n)$	$\Theta(\log n)$	$O(\log n)$, $\Omega(\log\log n)$
Find-Duplicate	$\Theta(\log n)$	$\Theta(n)$	$\widetilde{\Theta}(n)$
ℓ_2-approximation (streaming)	$\Theta(\log n)$	$\widetilde{\Theta}(n)$	$\widetilde{\Theta}(n)$
ℓ_2-approximation (sketching)			$\widetilde{O}(n)$, $\widetilde{\Omega}(\log n)$
Find-Nonzero-Row	$\widetilde{\Theta}(1)$	$\widetilde{\Theta}(nd)$	$\widetilde{\Theta}(n)$

Table 4.1: Table of space complexities.

Definition 4.2.1. We say that an algorithm A is *k-concentrated* if for all valid inputs x, there is some output $F(x)$ such that $\Pr_r[A(x,r) = F(x)] \geq \frac{1}{k}$.

The reason for making this definition is that any $\log k$-entropy randomized algorithm, and any $(k+2)$-pseudo-deterministic algorithm is k-concentrated. Thus, showing an impossibility result for k-concentrated algorithms also shows an impossibility result for $\log k$-entropy and $(k+2)$-pseudo-deterministic algorithms. Indeed, in this work, we use space lower bounds against k-concentrated algorithms to simultaneously conclude space lower bounds against low entropy and multi-pseudo-deterministic algorithms.

Definition 4.2.2. A turnstile streaming algorithm is one where there is a vector v, and the input is a stream of updates of the form "increase the i^{th} coordinate of v by r" or "decrease the i^{th} coordinate of v by r'". The goal is to compute something about the final vector, after all of the updates.

We use a pseudorandom generator for space-bounded computation due to Nisan [Nis92], which we recap below.

Theorem 4.2.3. There is a function $G : \{0,1\}^{s \log r} \to \{0,1\}^r$ such that

1. Any bit of $G(x)$ for any input x can be computed in $O(s \log r)$ space.
2. For all functions f from $\{0,1\}^r$ to some set A such that f is computable by a finite state machine on 2^s states, the total variation distance between the random variables $f(\boldsymbol{x})$ and $f(G(\boldsymbol{y}))$ where $\boldsymbol{x}$ is uniformly drawn from $\{0,1\}^r$ and $\boldsymbol{y}$ is uniformly drawn from $\{0,1\}^{s \log r}$ is at most 2^{-s}.

4.3 Find-Duplicate: Pseudo-deterministic lower bounds

Consider the following problem: the input is a stream of $3n/2$ integers between 1 and n. The goal is to output a number k which appears at least twice in the stream. Call this problem Find-Duplicate. Recall that this problem has been considered in the past literature, specifically in [GR09,JST11,KNP$^+$17], where upper and lower bounds for randomized algorithms have been shown.

Indeed, we know the following is true from [GR09, JST11].

Theorem 4.3.1. Find-Duplicate has an algorithm which uses $O(\text{poly}\log n)$ memory and succeeds with all but probability $\frac{1}{\text{poly}(n)}$.

We formally define a *pseudo-deterministic streaming algorithm* and show a pseudo-deterministic lower bound for Find-Duplicate to contrast with the randomized algorithm from Theorem 4.3.1.

Definition 4.3.2 (Pseudo-deterministic Streaming Algorithm)**.** A pseudo-deterministic streaming algorithm is a (randomized) streaming algorithm A such that for all valid input streams $s = \langle x_1, \ldots, x_m \rangle$, the algorithm A satisfies $\Pr_{r_1,r_2}[(A(x, r_1) = A(x, r_2)] \geq 2/3$.

One can also think of a pseudo-deterministic streaming algorithm as an algorithm A such that for every valid input stream s, there exists some valid output $f(s)$ such that the algorithm A outputs $f(s)$ with probability at least $2/3$ (one would have to amplify the success probability using repetition to see that this alternate notion is the same as the definition above).

Definition 4.3.3 (Find-Duplicate)**.** Define Find-Duplicate to be the streaming problem where the input is a stream of length $3n/2$ consisting of up to n, and the output must be an integer which has occured at least twice in the string.

Theorem 4.3.4. Find-Duplicate has no pseudo-deterministic algorithm with memory $o(n)$.

Proof Overview: In order to prove Theorem 4.3.4, we introduce two communication complexity problems — One-Way-Find-Duplicate and One-Way-Partial-Recovery:

In the One-Way-Find-Duplicate problem, Alice has a list of $3n/4$ integers between 1 and n, and so does Bob. Alice sends a message to Bob, after which Bob must output an integer which is in both Alice's and Bob's list. Formally:

Definition 4.3.5 (One-Way-Find-Duplicate)**.** Define One-Way-Find-Duplicate to be the one-way communication complexity problem where Alice has input $S_A \subseteq [n]$ and Bob has input $S_B \subseteq [n]$, where $|S_A|, |S_B| \geq 3n/4$. The goal is for Bob to output an element in $S_A \cap S_B$.

The idea is that one can reduce One-Way-Find-Duplicate to Find-Duplicate. So, our new goal will be to show that One-Way-Find-Duplicate requires high communication. To do so, we will show that it is possible to reduce a different problem, denoted One-Way-Partial-Recovery(defined below), to One-Way-Find-Duplicate. Informally, in the One-Way-Partial-Recovery problem, Alice has a list of $3n/4$ integers between 1 and n. Bob does not have an input. Alice sends a message to Bob, after which Bob must output $n/10$ distinct elements which are all in Alice's list. Formally:

Definition 4.3.6 (One-Way-Partial-Recovery)**.** Define One-Way-Partial-Recovery to be the one-way communication complexity problem where Alice has input $S_A \subseteq [n]$ and Bob has no input. The goal is for Bob to output a set S satisfying $S \subseteq S_A$ and $|S| \geq n/10$.

We will show in Claim 1 that a low memory pseudo-deterministic algorithm for Find-Duplicate implies a low-communication pseudo-deterministic algorithm for One-Way-Find-Duplicate, and in Claim 2 that a low-communication pseudo-deterministic algorithm for One-Way-Find-Duplicate implies a low communication algorithm for One-Way-Partial-Recovery. Finally, in Claim 3, we show that One-Way-Partial-Recovery cannot be solved with low communication. Combining the claims yields Theorem 4.3.4.

Proof of Theorem 4.3.4.

Claim 1. *A pseudo-deterministic algorithm for* Find-Duplicate *with space S and success probability p implies a pseudo-deterministic communication protocol for* One-Way-Find-Duplicate *with communication S and success probability at least p.*

Proof. To prove the above claim, we construct a protocol for One-Way-Find-Duplicate from a streaming algorithm for Find-Duplicate. Given an instance of One-Way-Find-Duplicate, Alice can stream her input set of integers in increasing order, and simulate the streaming algorithm for Find-Duplicate. Then, she sends the current state of the algorithm (which is at most S bits) to Bob, who continues the execution of the streaming algorithm. At the end, the streaming

algorithm outputs a repetition with probability p, which means the element showed up in both Alice and Bob's lists. Note that for a given input to Alice and Bob, Bob outputs a unique element with high probability because the streaming algorithm is pseudo-deterministic. □

Claim 2. *A pseudo-deterministic one-way communication protocol for* One-Way-Find-Duplicate *with S communication and failure probability $O\left(\frac{1}{n^2}\right)$ implies a pseudo-deterministic communication protocol for* One-Way-Partial-Recovery *with S communication and $O\left(\frac{1}{n}\right)$ failure probability.*

Proof. We will show how to use a protocol for One-Way-Find-Duplicate to solve the instance of One-Way-Partial-Recovery.

Suppose we have an instance of One-Way-Partial-Recovery. Alice sends the same message to Bob as if the input was an instance of One-Way-Find-Duplicate, which is valid since in both of these problems, Alice's input is a list of length $3n/4$ of integers between 1 and n.

Now, Bob's goal is to use the message sent by Alice to recover $n/10$ elements of Alice. Let X be the (initially empty) set of elements of Alice's input that Bob knows and let B be a set of $3n/4$ elements in $\{1, \ldots, n\}$ disjoint from X, where we initially set B to $\{1, 2, \ldots, n\}$. While the size of X is less than $n/10$, Bob simulates the protocol of One-Way-Find-Duplicate with Alice's message and input B. This will result in Bob finding a single element x in Alice's input that is (i) in B, and (ii) not in X. Bob adds x to X, and deletes x from B. Once the size of X is $n/10$, Bob outputs X.

If Alice has the set A as her input, define $f_A(B)$ to be the output which the pseudo-deterministic algorithm for One-Way-Find-Duplicate outputs with high probability when Alice's input is A and Bob's input is B. Now, set $B_0 = \{1, 2, \ldots, n\}$, and $B_i = B_{i-1} \setminus \{f_A(B)\}$. Note that these B_i (for $i = 0$ through $n/10$) are the sets which, assuming the pseudo-deterministic algorithm never errs during the reduction (where we say the algorithm errs if it does not output the unique element which is guaranteed to be output with high probability), Bob will use as his inputs for the simulated executions of One-Way-Find-Duplicate. The pseudo-deterministic algorithm does not err on any of the B_i except with probability at most $1/n$, by a union bound. If Bob succeeds on all of the B_i, that means that the sequence of inputs which will be his inputs for the simulated executions of One-Way-Find-Duplicate are indeed $B_0, B_1, \ldots, B_{n/10}$. So, since we have shown with high probability the algorithms succeeds on all of the B_i, and therefore with high probability the B_i are also Bob's inputs for the simulated executions of One-Way-Find-Duplicate, we see that

with high probability Bob will succeed on all of the $n/10$ inputs he tries to simulate executions of One-Way-Find-Duplicate with.

Note that we used the union bound over all the B_i for $i = 1$ through $n/10$. All of these B_i are a function of A. In particular, notice that by definition, the B_i do not depend on the randomness chosen by Alice. □

Claim 3. *Every pseudo-deterministic* One-Way-Partial-Recovery *protocol which succeeds with probability at least* $\frac{2}{3}$ *requires* $\Omega(n)$ *bits of communication.*

Proof. We prove this lower bound by showing that a protocol for One-Way-Partial-Recovery can be used to obtain a protocol with exactly the same communication for the problem where Alice is given a string x in $\{0,1\}^{Cn}$ as input, she sends a message to Bob, and Bob must exactly recover x from Alice's message with probability at least $2/3$. This problem has a lower bound of $\Omega(n)$ bits of communication.

Suppose there exists a pseudo-deterministic algorithm for One-Way-Partial-Recovery. Given such a pseudo-deterministic protocol that succeeds with probability at least $2/3$, there is a function F such that $F(S)$ (a set with $n/10$ elements) is Bob's output after the protocol with probability at least $2/3$ when Alice is given S as input.

We will construct sets $S_1, \ldots, S_t$ to be subsets of $[n]$ of size $3n/4$ such that for any $i \neq j$, $F(S_i)$ is not a subset of S_j. To do so, we use the probabilistic method: set $S_1, \ldots, S_t$ be random subsets of $[n]$ of size $3n/4$. The probability that $F(S_i)$ is contained S_j for fixed $i \neq j$ is at most $\left(\frac{3}{4}\right)^{n/10}$. Thus, by a union bound, the probability that for any $i \neq j$, $F(S_i)$ is contained S_j is at most $t^2 \left(\frac{3}{4}\right)^{n/10}$, a quantity which is strictly less than 1 when t is $\left(\frac{4}{3}\right)^{n/100}$, so $S_1, \ldots, S_t$ satisfying the desired guarantee exist.

Alice and Bob can (ahead of time) agree on an encoding of $\lfloor \log t \rfloor$-bit strings that is an injective function G from $\{0,1\}^{\lfloor \log t \rfloor}$ to $\{S_1, \ldots, S_t\}$. Now, if Alice is given a $\lfloor \log t \rfloor$-bit string x as input, she can send a message to Bob according to the pseudo-deterministic protocol for One-Way-Partial-Recovery by treating her input as $G(x)$. Bob then recovers $F(G(x))$ with probability at least $2/3$, and can use it to recover $G(x)$ since there is unique S_i in which $F(G(x))$ is contained. Since G is injective, Bob can also recover x with probability $2/3$.

This reduction establishes a lower bound of $\Omega(\lfloor \log t \rfloor)$ on the pseudo-deterministic communication complexity of One-Way-Partial-Recovery, which is an $\Omega(n)$ lower bound. □

Combining Claim 1, Claim 2 and Claim 3 completes the proof of Theorem 4.3.4. □

It is worth noting that the problem has pseudo-deterministic algorithms with sublinear space if one allows multiple passes through the input. Informally, a p-pass streaming algorithm is a streaming algorithm which, instead of seeing the stream only once, gets to see the stream p times.

Claim 4. *There is a p-pass deterministic streaming algorithm that uses $\widetilde{O}(n^{1/p})$ memory for the* Find-Duplicate *problem.*

Proof. At the start of t-th pass, the algorithm maintains a candidate interval I of width $n^{1-(t-1)/p}$ from which it seeks to find a repeated element. At the very beginning, this candidate interval is $[1, n]$. In the t-th pass, first partition the interval into $n^{1/p}$ equal sized intervals $I'_1, \ldots, I'_{n^{1/p}}$, each of whose width (the width of an interval $[a, b]$ is $b - a$) is $n^{1-t/p}$ and count the number of elements of the stream that lie in each such subinterval – this count must exceed the width of at least one subinterval I'_t. Update I to I'_t and proceed to the next pass. After p passes, this interval will contain at most 1 integer. □

4.4 Entropy Lower Bound for Find-Duplicate

Theorem 4.4.1. Every zero-error randomized algorithm for Find-Duplicate that is $\frac{n}{s}$-concentrated must use $\Omega\left(\frac{s}{\log n}\right)$ space.

By *zero error*, we mean that the algorithm never outputs a number k which is not repeated. With probability one it either outputs a valid output, or $\perp$.

Proof. We use a reduction similar to that of the pseudo-deterministic case (cf. Proof of Claim 2). Using the exact same reduction from the proof of Claim 1, we get that a $\frac{n}{s}$-concentrated streaming algorithm for Find-Duplicate using T space must give us a $\frac{n}{s}$-concentrated protocol for One-Way-Find-Duplicate with communication complexity T. If we can give a way to convert such a protocol for One-Way-Find-Duplicate into an $O\left(\frac{Tn \log n}{s}\right)$-communication protocol for One-Way-Partial-Recovery, the desired lower bound on T follows from the lower bound on communication complexity of One-Way-Partial-Recovery from Claim 3. We will now show how to make such a conversion by describing a protocol for One-Way-Partial-Recovery.

Alice sends Bob $\Theta(n \log n/s)$ messages according to the protocol for One-Way-Find-Duplicate (that is, she simulates the protocol for One-Way-Find-Duplicate a total of $\Theta(n \log n/s)$ times). Bob's goal is to use these $\Theta(n \log n/s)$ messages to recover at least $n/10$ input elements of Alice. Towards this goal, he maintains a set of elements recovered so far, X (initially empty), and a family of 'active sets' $\mathcal{B}$ (initially containing the set $\{1, 2, \ldots, n\}$). While the size of X is smaller than $n/10$, Bob simulates the remainder of the One-Way-Find-Duplicate protocol on every possible pair (B, M) where B is a set in $\mathcal{B}$ and M is one of the messages of Alice. For each such pair (B, M) where the protocol is successful in finding a duplicate element x, Bob adds x to X, removes B from $\mathcal{B}$ and adds $B \setminus \{x\}$ to $\mathcal{B}$.

We now wish to prove that this protocol indeed lets Bob recover $n/10$ elements of Alice. Suppose Alice has input A. For each set S, define $f_A(S)$ be an element of $A \cap S$ that has probability at least s/n of being outputted by Bob on input S at the end of a One-Way-Find-Duplicate protocol. Let $S_0 := \{1, 2, \ldots, n\}$ and $S_i := S_{i-1} \setminus \{f_A(S_i)\}$ be defined for $0 \leq i \leq n/10$. Note that S_i are predetermined: it is a function of Alice's input (and, in particular, not a function of the randomness she uses when choosing her messages). For a fixed i, the probability of failure to recover $f_A(S_i)$ from any of Alice's messages is at most $1/n^2$. A failure to fill in X with $n/10$ elements implies that for some i, Bob failed to recover $f_A(S_i)$ from all of Alice's messages. The probability that such a failure happens for a specific i is at most $(1 - s/n)^{\Theta(n \log n/s)}$. By setting the constant in the Θ to be large enough, we can have this be at most $\frac{1}{n^2}$, and so by a union bound the probability that there is an i such that $f_A(S_i)$ is not recovered by Bob is at most $1/n$.

Thus, we obtain a protocol for One-Way-Partial-Recovery with communication complexity $O(Tn \log n/s)$, and so $T \leq s/\log n$, completing the proof. □

We obtain the following as immediate corollaries:

Corollary 4.4.2. *Any zero-error* $\log\binom{n}{s}$*-entropy randomized algorithm for* Find-Duplicate *must use* $\Omega\left(\frac{s}{\log n}\right)$ *space.*

Corollary 4.4.3. *Any zero-error* $O\binom{n}{s}$*-pseudo-deterministic algorithm for* Find-Duplicate *must use* $\Omega\left(\frac{s}{\log n}\right)$ *space.*

Below we show that the above lower bound is tight up to log factors.

Theorem 4.4.4. For all s, there exists a zero-error randomized algorithm for FIND-DUPLICATE using $\widetilde{O}(s)$ space (where $\widetilde{O}$ hides factors polylogarithmic in n) that is $O\left(\frac{n}{s}\right)$-concentrated.

Proof. Define the following algorithm A for FIND-DUPLICATE: pick a random number i in $[3n/2]$, then remember the i^{th} element a of the stream, and see if a appears again later in the stream. If it does, return x. Otherwise return $\perp$.

The $O\left(\frac{n}{s}\right)$-concentrated algorithm algorithm is as follows: Run $s \log n$ copies of Algorithm A independently (in parallel), and then output the minimum of the outputs.

We are left to show that this algorithm is indeed $O\left(\frac{n}{s}\right)$-concentrated.

Define f to be a function where $f(i)$ is the total number of times which i shows up in the stream, and define $g(i) = \max((f(i) - 1), 0)$. Note that then, the probability that i is outputted by algorithm A is $g(i)/(3n/2)$, since i will be outputted if A chooses to remember one of the first $i - 1$.

Consider the smallest a such that $\sum_{i=1}^{a} g(i) \geq n/(2s)$. We will show that the probability that the output is less than a with high probability. It will follow that the algorithm is s-concentrated, since of the $a-1$ smallest elements, at most $\sum_{i=1}^{a-1} g(i)$ outputs are possible (since if $g(i) = 0$, then i is not a possible output). So, we will see that with high probability, one of at most $\sum_{i=1}^{a-1} g(i)+1 \leq n/(2s)$ outputs (namely, the valid outputs less than or equal to a) will be outputted with high probability. And hence, at least one of them will be outputted with probability at least $\frac{s}{n}$.

The probability that the output is at most a in a single run of algorithm A is $\frac{3n}{2} \sum_{i=1}^{a} g(i) \geq 3/(4s)$. So, the probability that in $s \log n$ runs of algorithm, in at least one of them an element which is at most a is outputted is $1 - (1 - \frac{3}{4s})^{s \log n}$, which is polynomially small in n. Hence, with high probability an element which is at most a (and there are $n/(2s)$ valid outputs less than a) will be outputted.

□

4.4.1 Getting Rid of the Zero Error Requirement

A downside of Theorem 4.4.1 is that it shows a lower bound only for zero-error algorithms. In this section, we strengthen the theorem by getting rid of that requirement:

Theorem 4.4.5. Every randomized algorithm for FIND-DUPLICATE that is $\frac{n}{s}$-concentrated and errs with probability at most $\frac{1}{n^2}$ must use $\tilde{\Omega}\left(s^{1-\epsilon}\right)$ space (for all $\epsilon > 0$).

Proof overview: We begin by outlining why the approach of Theorem 4.4.1 does not work without the zero-error requirement. Recall that the idea in the proof was to have Alice send many messages (for One-Way-Find-Duplicate) to Bob, and Bob simulates the One-Way-Find-Duplicate algorithm (using simulated inputs he creates for himself) using these messages to find elements in Alice's input.

The problem is that the elements we end up removing from Bob's simulated input[1] depend on Alice's messages, and therefore we can't use a union bound to bound the probability that the protocol failed for a certain simulated input. So, we want the elements we remove from Bob's fake input not to depend on the inputs Alice sent. One idea to achieve this is to have Alice send a bunch of messages (for finding a shared element), and then Bob will remove the element that gets output the largest number of times (by simulating the protocol with each of the many messages Alice sent). The issue with this is that if the two most common outputs have very similar probability, the outputted element depends not only on Alice's input, but also on the randomness she uses when choosing what messages to send to Bob. This makes it again not possible to use a union bound.

There are two new ideas to fix this issue. The first is to use the following "Threshold" technique: Bob will pick a random "threshold" T between $ks/(2n)$ and $ks/(4n)$ (where we wish to show a lower bound on n/s-concentrated algorithms, and k is the total number of messages Alice sends to Bob). He simulates the algorithm for One-Way-Find-Duplicate with all k messages Alice sent him, and gets a list L of k outputs. Then, he will consider the "real" output to be the lexicographically first output $y \in L$ where there are more than T copies of y in the list L (note that since the algorithm is n/s-concentrated, its very unlikely for no such element to exist).

Now, it follows that with high probability, the shared element does not really depend on the messages. This is because with all but probability approximately $1/\sqrt{ks/n}$, the threshold is far (more than $\sqrt{ks/n}\log^2 ks/n$ away) from the the frequency of every element in L. We note that we pick $\sqrt{ks/n}\log^2 ks/n$ since from noise we would expect to have the frequencies of elements in L change by up to $\sqrt{ks/n}\log^2 ks/n$, depending on the randomness of A. We want the threshold to be further than that from the expected frequencies, so that with high probability there will be no element which sometimes has frequency more than T and sometimes has frequency less than T, depending on Alice's messages (recall that the goal is to make the outputs depend as little as

[1] recall that Bob simulates an input to the One-Way-Find-Duplicate problem, and then he repeatedly finds elements he shares with Alice, removes them from the "fake" input, and reconstructs a large fraction Alice's inputs

possible on Alice's messages, but to only depend on shared randomness and on Alice's input).

This is still not enough for us: we still cannot use a union bound, as $1/\sqrt{ks/n}$ fraction of the time Bob's output will depends on Alice's message (and not just her input). The next idea resolves this. What Alice will do is send $n/\sqrt{ks/n}$ additional pieces of information: telling Bob where the chosen thresholds are bad, and what threshold to use instead. We assume that we have shared randomness so Alice knows all of the thresholds that will be chosen by Bob (note heavy-recovery is hard, even in the presence of shared randomness, so the lower bound is sufficient with shared randomness). Now, Alice can tell for which executions there the threshold chosen will be too close to the likelihood of an element. So, Alice will send approximately $n/\sqrt{ks/n}$ additional pieces of information: telling Bob where the chosen thresholds are bad, and what threshold to use instead. By doing so, Alice has guaranteed that a path independent of her messages will be taken.

To recap, idea 1 is to use the threshold technique so that with probability $1 - 1/\sqrt{ks/n}$ what Bob does doesn't depend on Alice's messages (only on her input). Idea 2 is to have Alice tell Bob where these $1/\sqrt{ks/n}$ bad situations are, and how to fix them.

The total amount of information Alice sends (ignoring logs) is $\tilde{\Theta}(kb + n/\sqrt{ks/n})$, (where b is the message size we are assuming exists for pseudo-deterministically finding a shared element, and k is the number of messages Alice sends). The factor $n/\sqrt{ks/n}$ follows since $1/\sqrt{ks/n}$ of the times, short messages will be sent to Bob due to a different threshold. A threshold requires $\log n$ bits to describe, which can be dropped since we are ignoring log factors. Setting $n/s \ll k \ll n/b$, we conclude that Alice sends a total of $\tilde{o}(n)$ bits. This establishes a contradiction, since we need $\tilde{\Theta}(n)$ bits to solve One-Way-Partial-Recovery. So, whenever $s = \tilde{\omega}(b)$, we can pick a k such that we get a contradiction.

Proof. Below we write the full reduction written out as an algorithm for One-Way-Find-Duplicate.

- Alice Creates $k = n/\sqrt{sb}$ messages for One-Way-Find-Duplicate, and sends them to Bob (Call these messages of type A).
- Additionally, Alice looks at the thresholds in the shared randomness. every time there is a threshold that is close (within $\sqrt{ks/n}\log^2 ks/n$) of the expected number of times a certain y will be outputted on the corresponding input (that is, for each fake input Bob will try, Alice checks if the probability of outputting some y is close to T – to be precise, say she checks if

its probability of being outputted, assuming a randomly chosen message by Alice, is close to T), she sends a message to Bob informing him about the bad threshold, and suggests a good threshold to be used instead (call these messages of type B). Notice that these messages do not depend on the messages of type A that Alice sends, and that each such message is of size $O(\log n)$.

- Bob sets B to be the simulated input $\{1, ..., n\}$
- Bob uses each of the messages of type A that Alice sent, along with B, to construct a list of outputs.
- Bob looks at the shared randomness to find a threshold T (if Alice has informed him it is a bad threshold, use the threshold Alice suggests instead), and consider the lexicographically minimal output y that is contained in the multiset more than T times.
- Bob removes y from the fake input and repeat the last three steps of the algorithm (this time using a new threshold).

Claim 5. *The above protocol solves* One-Way-Partial-Recovery *with high probability using* $\tilde{o}(n)$ *bits.*

Proof. First we show that the total number of bits communicated is $\tilde{o}(n)$. Notice that the total number of messages of type A that are sent is $n/\sqrt{sb}$. We assume that each of these is of size at most b, giving us a total of $n\sqrt{b}/\sqrt{s}$ bits sent in messages of type A. Under the assumption that $b = \tilde{o}(n)$, we see that this is $\tilde{o}(n)$ total bits for messages of type A.

We now count the total number of bits communicated in messages of type B. Each message of type B is of size $O(\log n)$ (it is describing a single element, and a number corresponding to which execution the message is relevant for, each requiring $O(\log n)$ bits). So, we wish to show that with high probability the total number of messages of type B is $\tilde{o}(n)$. The total number of messages of type B that will be sent is $O(\frac{n}{\sqrt{ks/n}})$, since for every input, the probability that the randomly chosen threshold (which is sampled using public randomness) is more than $\sqrt{ks/n}\log^2 ks/n$ away from the frequency of every output is $O(\frac{1}{\sqrt{ks/n}})$. Note that $\frac{n}{\sqrt{ks/n}} = \tilde{o}(n)$ since $ks = n\sqrt{\frac{s}{b}}$, and we assume $b = \tilde{o}(s)$.

We are now left to show the protocol correctly solves One-Way-Partial-Recovery with high probability. We will first show that, after fixing Alice's input and the public randomness, with high probability there will be a single sequence of inputs that Bob will try that will occur with high probability (that is, there is a sequence of y's that Bob goes through with high probability). To do this, consider a certain input that Bob tries. We will bound the probability that there are two values y and y' such that both y and y' have probability at least $\frac{1}{n}$ of being outputted. Suppose there exists two such y and y' that means that at least one of them (say y, without loss of generality) has to be the output of more than T of the k executions with probability more than $\frac{1}{n}$, but less than $\frac{n-1}{n}$. Additionally, we know that the expected number of times that y will be outputted of the k times is more than $\sqrt{ks/n}\log^2 ks/n$ away from T (otherwise Alice will pick a different value of T such that this will be true, and send that value to Bob in a message of type B). However, the probability of being more than $\sqrt{ks/n}\log^2 ks/n = \Theta(\left(\frac{s}{b}\right)^{1/4}\log^2 s/b) = \Theta(n^{\varepsilon/4}\log^2 n)$ away from the expectation, by a Chernoff bound, is (asymptotically) less than $\frac{1}{n}$.

Notice also, that by the assumption that the algorithm in n/s-concentrated, there will always be an output $y_{\max}$ which is expected to appear at least $\frac{s}{n}$ of the time. Also, since the threshold T is at most $ks/2n$, the probability that $y_{\max}$ appeared fewer than T times is exponentially low in $ks/n = \sqrt{s/b} = \tilde{\Theta}(n^{\epsilon/2})$, and so with high probability there will always exist a y which was outputted on more than T of the executions, so in the second to last step, the multiset will always have an element that appears at least T_1 times.

Hence, by a union bound over all inputs that Bob tries, with high probability there will be a single sequence of inputs which Bob goes through (which depends only only on the public thresholds and Alice's input).

We will show that each y generated by Bob is an element in Alice's input with high probability. Notice that the y that Bob picks has appeared more than T times out of k, where T is at least $ks/(4n)$. If y is not a valid output then its probability of being outputted is $\frac{1}{n^2}$. The probability it is outputted at least once is at most $\frac{k}{n^2} \leq \frac{1}{n}$. Taking a union bound over the inputs that Bob tries (of which there are $n/10$), we get that the probability that there is an invalid y at any point is at most $1/10$. So, with probability $9/10$, no invalid y is ever outputted. □

□

4.5 Entropy lower bounds for finding a support element

Consider the turnstile model of streaming, where a vector $z \in \mathbb{R}^n$ starts out as 0 and receives updates of the form 'increment z_i by 1' or 'decrement z_i by 1', and the goal of outputting a nonzero coordinate of z. This is a well studied problem and a common randomized algorithm to solve this problem in a small amount of space is known as ℓ_0 sampling [FIS08]. ℓ_0 sampling uses polylogarithmic space and outputs a uniformly random coordinate from the support of z. A natural question one could ask is whether the output of any low space randomized algorithm is necessarily close to uniform, i.e., has high entropy. We answer this affirmatively and show a nearly tight tradeoff between the space needed to solve this problem and the entropy of the output of a randomized algorithm under the assumption that the algorithm is not allowed to output anything outside the support[2]

Theorem 4.5.1. Every zero-error randomized algorithm for Find-Support-Elem that is $\frac{n}{s}$-concentrated must use $\Omega\left(\frac{s}{\log n}\right)$ space.

We only provide a sketch of the proof and omit details since they are nearly identical to the proof of Theorem 4.4.1.

Proof Sketch. Let $\mathcal{A}$ be such an algorithm that uses T space. Just like the proof of Theorem 4.4.1, the way we show this lower bound is by illustrating that $\mathcal{A}$ can be used to obtain an $O\left(\frac{Tn\log n}{s}\right)$-communication protocol for One-Way-Partial-Recovery, which combined with Claim 3 yields the desired result.

For every element a in Alice's input set A, she streams 'increment z_a by 1' and runs $\Theta\left(\frac{n}{s}\log n\right)$ independent copies of $\mathcal{A}$ on the input. She then sends the states of each these independent runs of $\mathcal{A}$ to Bob, which is at most $\frac{Tn\log n}{s}$ bits, to Bob. Bob maintains a set of states $\mathcal{M}$, initially filled with all of Alice's messages. While he has not yet recovered $n/10$ elements, Bob picks a message $M \in \mathcal{M}$ and recovers x in A using algorithm $\mathcal{A}$. And for each $M \in \mathcal{M}$, Bob resumes $\mathcal{A}$ on state M and streams 'decrement z_x by 1' and adds the new state to $\mathcal{M}$, and deletes M from $\mathcal{M}$.

The proof of correctness for why Bob indeed eventually recovers $n/10$ elements of A is identical to that in the proof of Theorem 4.4.1, thus giving a protocol for One-Way-Partial-Recovery and proving the statement. □

[2]We note that using similar ideas to those in Subsection 4.4.1, the zero error requirement could be removed. We omit this adaptation since it is very similar to that of Subsection 4.4.1.

We can immediately conclude the following.

Corollary 4.5.2. *Any zero-error* $\log\left(\frac{n}{s}\right)$*-entropy randomized algorithm for* Find-Support-Elem *must use* $\Omega\left(\frac{s}{\log n}\right)$ *space.*

Corollary 4.5.3. *Any zero-error* $O\left(\frac{n}{s}\right)$*-pseudo-deterministic algorithm for* Find-Support-Elem *must use* $\Omega\left(\frac{s}{\log n}\right)$ *space.*

This lower bound is also tight up to polylogarithmic factors due to an algorithm nearly identical to the one from Theorem 4.4.4. In particular, we have:

Theorem 4.5.4. For all s, there exists a zero-error randomized algorithm for Find-Duplicate using $O(s)$ space that is $O\left(\frac{n}{s}\right)$-concentrated.

4.6 Space complexity of pseudo-deterministic ℓ_2-norm estimation

In this section, we once again consider the pseudo-deterministic complexity of ℓ_2 norm estimation in the *sketching model*. The algorithmic question here is to design a distribution $\mathcal{D}$ over $s \times n$ matrices along with a function $f : \mathbb{R}^s \to \mathbb{R}$ so that for any $x \in \mathbb{R}^n$:

$$\Pr_{\boldsymbol{S}\sim\mathcal{D}}[f(\boldsymbol{S}x) \notin \left[\frac{1}{\alpha}\|x\|_2, \alpha\|x\|_2\right] \leq \frac{1}{\text{poly}(n)}.$$

Further, we want $f(\boldsymbol{S}x)$ to be a pseudo-deterministic function; i.e., we want $f(\boldsymbol{S}x)$ to be a unique number with high probability.

Theorem 4.6.1. The pseudo-deterministic sketching complexity of ℓ_2 norm estimation is $\Omega(n)$.

The following query problem is key to our lower bound.

Definition 4.6.2 (ℓ_2 adaptive attack)**.** Let $\alpha > 0$ be some constant. Let S be an $s \times n$ matrix with real-valued entries and $f : \mathbb{R}^s \to \mathbb{R}$ be some function. Now, consider the query model where an algorithm is allowed to specify a vector $x \in \mathbb{R}^n$ as a query and is given $f(Sx)$ as a response. The goal of the algorithm is to output y such that

$$f(Sy) \notin \left[\frac{1}{\alpha}\|y\|_2, \alpha\|y\|_2\right]$$

in as few queries as possible. We call this algorithmic problem the ℓ_2*-adaptive attack problem.*

We use a theorem on adaptive attacks on ℓ_2 sketches proved in [HW13].

Theorem 4.6.3. There is a poly(n)-query protocol to solve the ℓ_2 adaptive attack problem with probability at least 9/10, i.e., the problem in Definition 4.6.2 when $s = o(n)$.

We remark that there is an analogous theorem to Theorem 4.6.3 that applies to the ℓ_2 heavy hitters (i.e., compressed sensing) problem, so it is likely that one can also prove a pseudo-deterministic lower bound for the heavy hitters problem in an analogous way.

Proof of Theorem 4.6.1. Suppose $\mathcal{D}$ is a distribution over $s \times n$ sketching matrices and f is a function mapping $\mathbb{R}^s$ to $\mathbb{R}$ with the property that the pair $(\mathcal{D}, f)$ gives a pseudo-deterministic sketching algorithm for ℓ_2 norm estimation. Henceforth, we use $\boldsymbol{S}$ to denote a random matrix sampled from $\mathcal{D}$. Then there is a function $g : \mathbb{R}^n \to \mathbb{R}$ such that:

1. g is an α-approximation of the ℓ_2 norm.
2. On every input x, $f(\boldsymbol{S}x) = g(x)$ with probability at least $1 - \frac{1}{n^c}$ for some constant c.

We will show that s must be $\Omega(n)$ by deducing a contradiction when $k = o(n)$. Let r be a parameter to be chosen later. Let $\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(r)}$ be the (random) sequence of vectors in $\mathbb{R}^n$ obtained by the adaptive query protocol from Theorem 4.6.3 based on responses $g(\boldsymbol{x}^{(0)}), \ldots, g(\boldsymbol{x}^{(r)})$ where $r = \text{poly}(n)$, and let $\boldsymbol{y}$ be the (random) output of the protocol. Note that the guarantee that $r = \text{poly}(n)$ hinges on assuming $s = o(n)$. From the guarantees of Theorem 4.6.3, for any fixed matrix B and function h such that $h(B\boldsymbol{x}^{(i)}) = g(\boldsymbol{x}^{(i)})$ for all i, it is true with probability at least 9/10 that $h(B\boldsymbol{y}) \neq g(\boldsymbol{y})$. On the other hand, for any sequence of $r + 2$ fixed vectors $v_0, \ldots, v_{r+1}$, $f(\boldsymbol{S}v_i) = g(v_i)$ for all i with probability at least $1 - \frac{1}{\text{poly}(n)}$. Call the event $\{f(\boldsymbol{S}\boldsymbol{x}^{(0)}) = g(\boldsymbol{x}^{(0)}), \ldots, f(\boldsymbol{S}\boldsymbol{x}^{(r)}) = g(\boldsymbol{x}^{(r)}), f(\boldsymbol{S}\boldsymbol{y}) = g(\boldsymbol{y})\}$ as $\mathcal{E}$. Let p_S be the probability density function of $\boldsymbol{S}$ and let p_T be the probability density function of $(\boldsymbol{x}^{(0)}, \ldots, \boldsymbol{x}^{(r)}, \boldsymbol{y})$. This results in the following two estimates of $\Pr[\mathcal{E}]$.

On the one hand,

$$\begin{aligned}\Pr[\mathcal{E}] &= \int_{\boldsymbol{S}} \Pr[\mathcal{E}|\boldsymbol{S}] p_S(\boldsymbol{S}) \\ &\leq \int_{\boldsymbol{S}} \frac{1}{10} p_S(\boldsymbol{S}) \\ &= \frac{1}{10},\end{aligned}$$

and on the other hand,

$$\begin{aligned}\Pr[\mathcal{E}] &= \int_{\boldsymbol{x}^{(0)},\ldots,\boldsymbol{x}^{(r)},\boldsymbol{y}} \Pr[\mathcal{E}|\boldsymbol{x}^{(0)},\ldots,\boldsymbol{x}^{(r)},\boldsymbol{y}] p_T(\boldsymbol{x}^{(0)},\ldots,\boldsymbol{x}^{(r)},\boldsymbol{y}) \\ &\geq \int_{\boldsymbol{x}^{(0)},\ldots,\boldsymbol{x}^{(r)},\boldsymbol{y}} \left(1 - \frac{1}{\text{poly}(n)}\right) p_T(\boldsymbol{x}^{(0)},\ldots,\boldsymbol{x}^{(r)},\boldsymbol{y}) \\ &= 1 - \frac{1}{\text{poly}(n)}.\end{aligned}$$

The contradiction arises since $\Pr[\mathcal{E}]$ cannot simultaneously be at least $1 - \frac{1}{\text{poly}(n)}$ and at most $\frac{1}{10}$, and hence s cannot be $o(n)$.

□

Corollary 4.6.4. *For any constant $\delta > 0$, any $(2-\delta)$-concentrated sketching algorithm that where the sketching matrix is $s \times n$ can be turned into a pseudo-deterministic one by running $\log n$ independent copies of the sketch and outputting the majority answer. Thus, as an upshot of Theorem 4.6.1 we obtain a lower bound of $\Omega\left(\frac{n}{\log n}\right)$ on $(2-\delta)$-concentrated algorithms for pseudo-deterministic ℓ_2-norm estimation in the sketching model.*

In contrast to Corollary 4.6.4 which says that $(2-\delta)$-concentrated algorithms for ℓ_2 estimation in the sketching model need near linear dimension, we show that there is an $O(\text{poly}\log n)$-dimension $(2+\delta)$-concentrated sketching algorithm to solve the problem, thus exhibiting a 'phase transition'.

Theorem 4.6.5. There is a distribution $\mathcal{D}$ over $s \times n$ matrices and a function $f : \mathbb{R}^s \to \mathbb{R}$ when $s = O(\text{poly}\log n)$ For every constant $\delta > 0$, there is an $O(\text{poly}(\log n, \log m))$-space $(2+\delta)$-concentrated sketching algorithm for ℓ_2-norm estimation.

Proof. Let the true ℓ_2 norm of the input vector be r. Run the classic sketching algorithm of [AMS99] for randomized ℓ_2 norm estimation with error $\min\{1/2^{20}, \varepsilon^4\}$ and failure probability $\frac{1}{\text{poly}(n)}$ where $(1+\varepsilon)$ is the desired approximation ratio. This uses a sketch of dimension $O(\text{poly}\log n)$. Now, we describe the function f we use. Take the output of the sketching algorithm of [AMS99] and return the number obtained by zeroing out all its bits beyond the first $\max\{2\log\left(\frac{1}{\varepsilon}\right), 5\}$ significant bits.[3] First, the outputted number is a $(1+\varepsilon)$ approximation. Further, for each input, the output is one of two candidates with probability $1 - \frac{1}{\text{poly}(n)} > 1 - \delta$ for every constant δ. This is because [AMS99]

[3]The parameters $1/2^{20}, 5$ and ε^4 are chosen purely for *safety* reasons

produces a $(1+\varepsilon^4)$-approximation to r, and there are only two candidates for the $2\log\left(\frac{1}{\varepsilon}\right)$ most significant bits of any real number that lies in an interval $[(1-\varepsilon^4)r, (1+\varepsilon^4)r]$. □

4.7 Pseudo-deterministic Upper Bounds

4.7.1 Finding a nonzero row

Given updates to an $n \times d$ matrix A (where we assume $d \leq n$) that is initially 0 in a turnstile stream such that all entries of A are always in range $[-n^3, n^3]$, the problem Find-Nonzero-Row is to either output an index i such that the ith row of A is nonzero, or output `none` if A is the zero matrix.

Theorem 4.7.1. The randomized space complexity for Find-Nonzero-Row is $\widetilde{\Theta}(1)$, the pseudo-deterministic space complexity for Find-Nonzero-Row is $\widetilde{\Theta}(n)$, and the deterministic space complexity for Find-Nonzero-Row is $\widetilde{\Theta}(nd)$.

Proof. We first will show a randomized $\widetilde{\Theta}(1)$ space algorithm for the problem, then we will show pseudo-deterministic upper and lower bounds, and then show the deterministic lower bound.

Randomized algorithm for Find-Nonzero-Row. A randomized algorithm for this problem is given below. Note that the version of the algorithm as stated below does not have the desirable $\widetilde{O}(1)$ space guarantee, but we will show how to use a pseudorandom generator of Nisan [Nis92] to convert the below algorithm to one that uses low space.

1. Sample a random d-dimensional vector $\boldsymbol{x}$ where each entry is an independently drawn integer in $[-n^3, n^3]$ and store it.

2. Simulate a turnstile stream which maintains $A\boldsymbol{x}$. In particular, consider the n-dimensional vector y, which is initially 0, and for each update to A of the form "add Δ to A_{ij}", add $\Delta \boldsymbol{x}_j$ to y_i. We run an ℓ_0-sampling algorithm [FIS08] on this simulated stream updating y, and return the output of the ℓ_0-sampler, which is close in total variation distance to a uniformly random element in the support of y.

In the above algorithm, step 1 is not low-space as stated. Before we give a way to perform step 1 in $\widetilde{O}(1)$ space, we prove the correctness of the above algorithm. Suppose A_i is a nonzero row of

A, then let j be an index where A_i is nonzero. Suppose all coordinates of $\boldsymbol{x}$ except for the j-th coordinate have been sampled, there is at most one value C for $\boldsymbol{x}_j$ for which $\langle A_i, \boldsymbol{x}\rangle$ is 0, and there is at most a $1/n^3$ probability that $\boldsymbol{x}_j$ equals C, which means if i is a nonzero row, then $(A\boldsymbol{x})_i$ is nonzero except with probability at most $1/n^3$. In fact, by taking a union bound over all nonzero rows we can conclude that the set of nonzero rows and the set of nonzero indices of $A\boldsymbol{x}$ are exactly the same, except with probability bounded by $1/n^2$.

Now we turn our attention to implementing step 1 in low space. Towards doing so we use Nisan's pseudorandom generator for space bounded computation in a very similar manner to [Ind06].

Instead of sampling $3d \log n + 1$ bits to store $\boldsymbol{x}$, we sample and store a uniformly random seed $\boldsymbol{w}$ of length $O(\text{poly}\log(n, d))$ and add $\Delta G(\boldsymbol{w})_j$ to y_i when an update "add Δ to A_{ij}" is received, where G is the function from Theorem 4.2.3 that maps the random seed to a sequence $3d \log n + 1$ bits. To prove the algorithm is still correct if we use the pseudorandom vector $G(\boldsymbol{w})$ instead of the uniformly random vector $\boldsymbol{x}$, we must show that when A_i is nonzero, then $\langle A_i, G(\boldsymbol{w})\rangle$ is nonzero with probability at least $1 - O(1/n^3)$. Towards this, for a fixed d-dimensional vector q, consider the following finite state machine. The states are labeled by pairs (i, a) where i is in $\{0, 1, \ldots, d\}$ and a is in $[-n^6 d, n^6 d]$. The FSM takes a d-dimensional vector r as input, starts at state $(0, 0)$, and transitions from state (i, a) to $(i + 1, a + q_{i+1} \cdot r_{i+1})$ until it reaches a state (d, ℓ). The FSM then outputs ℓ. This establishes that for a fixed q, the function $f(x) \coloneqq \langle q, x\rangle$ is computable by an FSM on $\text{poly}(d, n)$ states, and hence from Theorem 4.2.3, $f(\boldsymbol{x})$ and $f(G(\boldsymbol{w}))$ are $1/dn^6$ close in total variation distance, which means when A_i is nonzero, then $\langle A_i, G(\boldsymbol{w})\rangle$ is nonzero except with probability bounded by $O(1/n^3)$.

A pseudo-deterministic algorithm and lower bound for FIND-NONZERO-ROW The pseudo-deterministic algorithm is very similar to the randomized algorithm from the previous section.

1. Sample a random d-dimensional vector $\boldsymbol{x}$ where each entry is an independently drawn integer in $[-n^3, n^3]$. Store $\boldsymbol{x}$ and maintain $A\boldsymbol{x}$.

2. Output the smallest index i such that $(A\boldsymbol{x})_i$ is nonzero.

Storing $\boldsymbol{x}$ takes $O(d \log n)$ space, and maintaining $A\boldsymbol{x}$ takes $O(n \log n)$ space. Recall from the discussion surrounding the randomized algorithm that the set of nonzero indices of $A\boldsymbol{x}$ and the set of nonzero rows were equal with probability $1 - 1/n^2$, which establishes correctness of the above

pseudo-deterministic algorithm. The space complexity is thus $O((d+n)\log n)$, which is equal to $O(n \log n)$ from the assumption that $d \leq n$.

A pseudo-deterministic lower bound of $\widetilde{\Omega}(n)$ follows immediately from Corollary 4.5.3 since FIND-NONZERO-ROW specialized to the $d = 1$ case is the same as FIND-SUPPORT-ELEM.

Lower Bound for deterministic algorithms. An $\Omega(nd \log n)$ bit space lower bound for deterministic algorithms follows from a reduction to the communication complexity problem of EQUALITY. Alice and Bob are each given $nd \log n$ bit strings as input, which they interpret as $n \times d$ matrices, A and B respectively, where each entry is a chunk of length $\log n$. Suppose a deterministic algorithm $\mathcal{A}$ takes S bits of space to solve this problem. We will show that this can be converted to a S-bit communication protocol to solve EQUALITY. Alice runs $\mathcal{A}$ on a turnstile stream updating matrix X initialized at 0 by adding A_{ij} to X_{ij} for all (i, j) in $[n] \times [d]$. Alice then sends the S bits corresponding to the state of the algorithm to Bob and he continues running $\mathcal{A}$ on the updates 'add $-B_{ij}$ to X_{ij}'. $\mathcal{A}$ outputs `none` if and only if $A = B$ and thus Bob outputs the answer to EQUALITY depending on the output of $\mathcal{A}$. Due to a communication complexity lower bound of $\Omega(nd \log n)$ on EQUALITY, S must be $\Omega(nd \log n)$.

□

4.7.2 Point Query Estimation and Inner Product Estimation

In this section, we give pseudo-deterministic algorithms that beat the deterministic lower bounds for two closely related streaming problems — point query estimation and inner product estimation.

Point Query Estimation. Given a parameter ε and a stream of m elements where each element comes from a universe $[n]$, followed by a query $i \in [n]$, output f_i' such that $|f_i - f_i'| \leq \varepsilon m$ where f_i is the frequency of element i in the stream.

Inner Product Estimation. Given a parameter ε and a stream of m updates to (initially 0-valued) n-dimensional vectors x and y in an insertion-only stream[4], output estimate e satisfying $|e - \langle x, y \rangle| < \varepsilon \cdot \|x\|_1 \cdot \|y\|_1$.
In the above problems, we will be interested in the regime where $m \ll n$.

[4]A stream where only increments by positive numbers are promised.

Our main result regarding a pseudo-deterministic algorithm for point query estimation is:

Theorem 4.7.2. There is an $O\left(\frac{\log m}{\varepsilon} + \log n\right)$-space pseudo-deterministic algorithm $\mathcal{A}$ for point query estimation with the following precise guarantees. For every sequence $s_1, \ldots, s_m$ in $[n]^m$, there is a sequence $f'_1, \ldots, f'_n$ such that

1. For all i, $|f'_i - f_i| \leq \varepsilon m$ where f_i is the frequency of i in the stream.
2. Except with probability $1/m$, for all $i \in [n]$ $\mathcal{A}$ outputs f'_i on query i.

We remark that the deterministic complexity of the problem is $\Omega(\frac{\log n}{\varepsilon}$ (see Theorem 4.7.7).

Towards establishing Theorem 4.7.2, we recall two facts.

Theorem 4.7.3 (Misra–Gries algorithm [MG82])**.** Given a parameter ε and a length-m stream of elements in $\{1, \ldots, d\}$, there is a deterministic $O\left(\frac{\log d + \log m}{\varepsilon}\right)$-space algorithm that given any query $s \in [d]$, outputs f'_s such that $|f'_s - f_s| \leq \varepsilon m$ where f_s is the number of occurrences of s in the stream. An additional guarantee that the algorithm satisfies is the following, which we call *permutation invariance*. Consider the stream

$$s_1, s_2, \ldots, s_m$$

and for any permutation $\pi : [d] \to [d]$, consider the stream

$$\pi(s_1), \pi(s_2), \ldots, \pi(s_m).$$

When the algorithm is given the first stream as input, let f'_s denote its output on query s, and when the algorithm is given the second stream as input, let $g'_{\pi(s)}$ denote its output on query $\pi(s)$. The algorithm has the guarantee that $f'_s = g'_{\pi(s)}$.

Theorem 4.7.4 (Pairwise independent hashing, [V$^+$12, Corollary 3.34])**.** Assume $d \ll n$. There is a pairwise independent hash function $h : [n] \to [d]$, which can be sampled using $O(\log n)$ random bits and also can be stored in $O(\log n)$ bits.

Proof of Theorem 4.7.2. The algorithm is as follows.

- Sample a random pairwise independent hash function $h : [n] \to [m^3]$, which can be sampled and stored in $O(\log n)$ bits.

- Run the Misra–Gries algorithm with the following simulated stream as input: for each s streamed as input, stream $h(s)$ to the simulation.
- Given any query s, perform query $h(s)$ to the Misra–Gries algorithm running on the simulated stream, and return its output.

Let S be the collection of elements of $[n]$ that occur in the input stream $s_1, \ldots, s_m$. Assuming h maps S into $[m^3]$ without any collisions[5], it follows from the permutation invariance property of the Misra–Gries algorithm from Theorem 4.7.3 the output of the above algorithm on any query q is equal to $F(s_1, \ldots, s_m, q)$ for a *fixed* function F. Thus if we show that h indeed maps S into $[m^3]$ injectively pseudo-determinism of the given algorithm would follow.

Given $i, j \in S$, due to pairwise independence of h, the probability that $h(i) = h(j)$ is equal to $1/m^3$. A union bound over all pairs of elements in S tells us that h is collision-free except with probability at most $1/m$, which implies that the above algorithm is indeed pseudo-deterministic. □

Theorem 4.7.5. There is a (weakly) pseudo-deterministic algorithm for inner product estimation that uses $O\left(\frac{\log m}{\varepsilon} + \log n\right)$ space.

The algorithm for inner product estimation is based on point query estimation, and towards stating the algorithm we first state a known result that helps relate the two problems.

Lemma 4.7.6 (Easily extracted from the proof of [NNW14, Theorem 1])**.** *Let x, y, x', y' be vectors such that $\|x - x'\|_\infty \leq \varepsilon \|x\|_1$ and $\|y - y'\|_\infty \leq \varepsilon \|y\|_1$. Now, let x'' (and respectively y'') denote x' with everything except the maximum $1/\varepsilon$ entries zeroed out. Then the following holds:*

$$|\langle x'', y'' \rangle - \langle x, y \rangle| \leq \varepsilon \cdot \|x\|_1 \cdot \|y\|_1.$$

Proof of Theorem 4.7.5. Given a stream of updates to x and y, run two instances of the point query estimation algorithm from Theorem 4.7.2 — one for updates to x and one for updates to y. There are x' and y' that only depend on the stream such that

$$\|x - x'\|_\infty \leq \varepsilon \cdot \|x\|_1 \quad \text{and} \quad \|y - y'\|_\infty \leq \varepsilon \cdot \|y\|_1$$

[5]I.e. the restriction of h to domain S is an injective function.

and except with probability $O(1/m)$ both point query algorithms respond to any query i with x'_i (and y'_i respectively). Maintaining these two instances takes $O\left(\frac{\log m}{\varepsilon} + \log n\right)$ space.

Next, enumerate over elements of $[n]$ and for each $i \in [n]$ query both instances with i, and store the running max-$1/\varepsilon$ answers to queries to each instance along with the hashed identities of the indices of entries that are part of the running max. Storing the running max takes $O\left(\frac{\log m}{\varepsilon}\right)$ space, and storing a counter to enumerate over $[n]$ takes $\log n$ space. Thus, at the end of this routine, except with probability $O(1/m)$ our two lists are equal to $(x'_{i_1}, h(i_1)), \ldots, (x'_{i_{1/\varepsilon}}, h(i_{1/\varepsilon}))$ and $(y'_{j_1}, h(j_1)), \ldots, (y'_{j_{1/\varepsilon}}, h(j_{1/\varepsilon}))$ respectively where $x'_{i_1}, \ldots, x_{i_{1/\varepsilon}}$ are the max-$1/\varepsilon$ entries of x' and $y'_{j_1}, \ldots, y_{j_{1/\varepsilon}}$ are the max-$1/\varepsilon$ entries of y'.

Finally, if there is t, u such that $h(i_t) = h(i_u)$ or $h(j_t) = h(j_u)$, return 'fail'; otherwise output

$$\sum_{\ell \in \{h(i_t)\}_{t=1,\ldots,1/\varepsilon} \cap \{h(j_t)\}_{t=1,\ldots,1/\varepsilon}} x'_\ell y'_\ell.$$

With probability at least $1 - 2/m$, the above quantity is equal to $\langle x'', y'' \rangle$ from Lemma 4.7.6, which lets us conclude via Lemma 4.7.6 that the output is within $\varepsilon \cdot \|x\|_1 \cdot \|y\|_1$ of the true inner product. □

Finally, we remark that the following lower bounds can be proved for deterministic algorithms.

Theorem 4.7.7. Any deterministic algorithm for point query estimation and inner product estimation needs $\Omega\left(\frac{\log n}{\varepsilon}\right)$ space.

Proof. We prove a lower bound for point query estimation via a reduction from EQUALITYin communication complexity. Alice encodes a $\log \binom{n}{1/(3\varepsilon)}$ bit string as a subset S of $[n]$ of size $1/(3\varepsilon)$ and runs the point query streaming algorithm on the input where she streams each element of this subset $3\varepsilon m$ times. She then sends the state of the algorithm to Bob, who can query every index in the universe and learn S (the element corresponding to the query is in S if and only if the response to the query is at least $2\varepsilon \cdot m$), decode S back to a $\log \binom{n}{1/(3\varepsilon)}$ and check if it is equal to his own input. The space lower bound from the theorem statement then follows since $\log \binom{n}{1/(3\varepsilon)} = \Omega\left(\frac{\log n}{\varepsilon}\right)$.

A space lower bound for inner product estimation follows from the lower bound for point query estimation since the latter is a special case of the former when x is the vector of frequencies and y is a standard unit vector e_i corresponding to query i. □

4.7.3 Retrieving a Basis of a Row-space

We now work in a 'mixed' model, where an input $n \times d$ matrix A of rank-$\leq k$ is given to us via a sequence of updates in a turnstile stream, and each entry at all times in the stream can be represented by an $O(\log n)$-bit word. During this phase, there is an upper bound T on the number of bits of space an algorithm is allowed to use. In the "second phase", we are allowed to perform arbitrary computation and the goal is to output a basis for the row-span of A

We show a lower bound on T of $\widetilde{\Omega}(nd)$ for deterministic algorithms, and a pseudo-deterministic algorithm that uses $\widetilde{O}(\text{poly}(k) \cdot d)$ space in the streaming phase.

Theorem 4.7.8. Any deterministic streaming algorithm for RecoverBasis needs $\widetilde{\Omega}(nd)$ space.

Proof. Suppose the matrix A is 0, then the algorithm would have to output the empty set. A T space streaming algorithm for this problem could be used to solve the communication complexity problem of equality Equality using T bits of communication. In particular, Alice and Bob could encode their respective inputs x and y as matrices M_x and M_y. Alice can then run the T-space algorithm on adding M_x in a turnstile stream, and send Bob the state of the algorithm. Bob can then resume running the algorithm from Alice's state on updates that subtract M_y. If Bob outputs the empty set, then $x = y$ and Bob outputs 'yes'. Otherwise, Bob outputs 'no'. □

While the deterministic complexity is $\widetilde{\Omega}(nd)$, there is a pseudo-deterministic streaming algorithm which uses only $\widetilde{O}(\text{poly}(k) + k \cdot d)$ in its streaming phase:

Theorem 4.7.9. There is a pseudo-deterministic algorithm for RecoverBasis that uses $\widetilde{O}(\text{poly}(k) + k \cdot d)$ space in its streaming phase, where the $\widetilde{O}(\cdot)$ hides factors of $\text{poly} \log n$.

Towards giving a pseudo-deterministic algorithm, we first state a result about pseudorandom matrices that is a special case of [CW09, Lemma 3.4].

Theorem 4.7.10. There is a distribution $\mathcal{D}$ over $m \times n$ matrices where $m = O(k \log n)$ with ± 1 entries such that for any $n \times m$ matrix U with orthonormal columns and $\boldsymbol{S} \sim \mathcal{D}$, the following holds with probability $1 - 1/\text{poly}(n)$:

$$\|U^T \boldsymbol{S}\boldsymbol{S}^T U - I\|_2 \leq 1/2.$$

Further, the rows of $\boldsymbol{S}$ are independent and each row can be generated by a $(k + \log n)$-wise independent hash family.

Theorem 4.7.11 (t-wise independent hash families [V$^+$12, Corollary 3.34])**.** There is a t-wise independent hash family $\mathcal{H}$ of functions from $[n] \to \{\pm 1\}$ such that sampling a uniformly random h from $\mathcal{H}$ can be done using a $\text{poly}(\log n, t)$-length random seed, and $h(x)$ for any $x \in [n]$ can be computed in $\text{poly}(\log n, t)$ time and space from the random seed used to sample it.

As a consequence we have:

Corollary 4.7.12. *Let A be a $n \times d$ matrix of rank k and let $\mathcal{D}$ be the distribution over $O(k \log n) \times n$ matrices from the statement of Theorem 4.7.10. Then, for $\boldsymbol{S} \sim \mathcal{D}$, $\boldsymbol{S}A$ has rank k with probability $1 - 1/\text{poly}(n)$.*

Proof. We start by writing A in its singular value decomposition $U\Sigma V^T$. Since A has rank k, U is a $n \times k$ matrix with orthonormal columns and ΣV^T surjectively maps $\mathbb{R}^d$ to $\mathbb{R}^k$. From Theorem 4.7.10, $\boldsymbol{S}A$ is also full rank, which means the collection of vectors

$$\{\boldsymbol{S}Ax : x \in \mathbb{R}^d\} = \{\boldsymbol{S}U\Sigma V^T x : x \in \mathbb{R}^d\} = \{\boldsymbol{S}Ux : x \in \mathbb{R}^k\}$$

is a k-dimensional space, and hence $\boldsymbol{S}A$ has rank k. □

Proof of Theorem 4.7.9. Begin by sampling $\boldsymbol{S} \sim \mathcal{D}$ via a seed $\boldsymbol{s}$ of length $O(\text{poly}(k) \cdot \text{poly}\log(n))$ from which entries of $\boldsymbol{S}$ can be efficiently computed where $\mathcal{D}$ is the distribution over matrices given by Corollary 4.7.12, and maintain the sketch $\boldsymbol{S}A$ in the stream.

The row-span of $\boldsymbol{S}A$ is exactly the same as that of A assuming the two matrices have equal rank, which happens with probability $1 - 1/\text{poly}(n)$.

$\boldsymbol{S}A$ is an $O(k) \times d$ matrix and each entry is a signed combination of at most n entries of A and hence there is a bit complexity bound of $\widetilde{O}(kd)$ on the space used to store $\boldsymbol{S}A$.

In the second phase (i.e., after the stream is over) of the algorithm, we first find an orthonormal basis Q for the row-span of $\boldsymbol{S}A$ and compute $\widetilde{\Pi}_A = QQ^T$. And finally, use a deterministic algorithm to compute the singular value decomposition $\widetilde{U}\Sigma\widetilde{V}^T$ of $\widetilde{\Pi}_A$ and output the rows of $\widetilde{V}^T$.

The row-span of $\boldsymbol{S}A$ and A are equal except with probability $1/\text{poly}(n)$; assuming this happens, $\widetilde{\Pi}_A$ is exactly equal to Π_A, the unique projection matrix onto the row-span of A. Write Π_A in its singular value decomposition $U\Sigma V^T$. If $\widetilde{\Pi}_A = \Pi_A$, $\widetilde{V}^T$ is exactly equal to V^T. Since V^T is given by a deterministic function of A, and the output of the algorithm $\widetilde{V}$ is equal to V^T with high probability, our algorithm is pseudo-deterministic. □

Chapter 5

Lower Bound for Pseudo-Deterministic Approximate Counting

This section is based on joint work with Meghal Gupta and Mark Sellke [GGS23].

5.1 Introduction

The study of streaming algorithms originated with the seminal paper of Morris [Mor78], which gave a low-memory randomized algorithm to approximately count a number of elements which arrive online in a stream. Roughly speaking, the idea is to have a counter which approximates $\log n$, where n is the number of elements seen in the stream so far. Each time the algorithm encounters an element from the stream, it increases the counter with probability about $\frac{1}{n}$. As later proved by [Fla85], Morris's algorithm achieves a constant-factor approximation error for streams of length at most N in space $O(\log \log N)$.

Morris's algorithm has the property that running it multiple times on the same stream may result in different approximations. That is, if Alice runs the algorithm on the same stream as Bob (but using different randomness), Alice may get some approximation (such as 2^{30}), and Bob (running the same algorithm but with independent randomness) may get a different approximation (such as 2^{29}). Is this behavior inherent? That is, could there exist a low-space algorithm which, while being randomized, for all streams with high probability both Alice and Bob will end up with the *same* approximation for the length? Such algorithms, namely those which output the same output with

high probability when run multiple times on the same input, are called *pseudo-deterministic*.

This question of the pseudo-deterministic space complexity of approximate counting in a stream was first posed in [GGMW20]. Our main result fully resolves the problem by giving a tight $\Omega(\log N)$ lower bound. In fact, our lower bound applies for an easier threshold version of the problem asking to distinguish between streams of length at most N versus at least M for any integers $N < M$. Moreover, it depends only on N, i.e. the problem is hard even when M is arbitrarily larger than N.

Theorem 5.1.1 (Informal)**.** For any $N < M$, a pseudo-deterministic streaming algorithm to distinguish between streams of length at most N and at least M must use $\Omega(\log N)$ space.

Concurrently, [BKKS23] showed a non-trivial $\Omega(\sqrt{\log n / \log\log n})$ space lower bound for this problem using different techniques.

5.1.1 Related Work

Prior Work on Approximate Counting in a Stream. The study of streaming algorithms began with the work Morris [Mor78] on approximate counting; a rigorous analysis was given later in [Fla85]. As explained earlier, the (randomized) Morris counter requires logarithmically fewer states than a (deterministic) exact counter. The approximate counter has been useful as a theoretical primitive for other streaming algorithms [AJKS02, GS09, KNW10, BDW18, JW19] and its performance has been evaluated extensively [Cve07, Csü10, XKNS21].

The optimal dependence on the error level and probability in approximate counting was determined recently in [NY22]. Moreover, [AAHNY22] studied the amortized complexity of maintaining several approximate counters rather than just one.

As this work was being finalized, [BKKS23] independently showed a $\Omega(\sqrt{\log n / \log\log n})$ lower bound. Their proof proceeds via reduction to one-way communication complexity. Our proof, on the other hand, takes a completely different approach and analyzes the complexity directly by modeling the streaming algorithm as a Markov chain and showing it behaves as an ensemble of cyclic parts in a suitable sense.

5.1.2 Main Result

We consider the problem of pseudo-deterministic approximate counting in a stream. We first recall the definition of a pseudo-deterministic streaming algorithm:

Definition 5.1.2. We define a streaming algorithm $\mathcal{A}$ to be *pseudo-deterministic* if there exists some function F such that for all valid input streams x,

$$\mathbb{P}_r[\mathcal{A}(x, r) = F(x)] \geq 2/3$$

where r is the randomness sampled and used by the algorithm $\mathcal{A}$.

Our main result gives a tight bound of $\Omega(\log N)$ bits of memory. In fact, we prove that pseudo-deterministically distinguishing between streams of length $T \leq N$ and $T \geq f(N)$ requires $\Omega(\log N)$ bits of memory for *any* function $f : \mathbb{N} \to \mathbb{N}$.

Definition 5.1.3 (Pseudo-Deterministic Approximate Threshold Problem)**.** Fix $N, M \in \mathbb{N}$. Suppose $\mathcal{A}$ is a pseudo-deterministic streaming algorithm such that with T the stream length:

1. $F(T) = 1$ for $T \leq N$,
2. $F(T) = 0$ for $T \geq M$.

Then we say $\mathcal{A}$ solves the (N, M)-pseudo-deterministic approximate threshold problem.

Theorem 5.1.4. For any $f : \mathbb{N} \to \mathbb{N}$, any sequence of pseudo-deterministic algorithms $\mathcal{A}_N$ which solve the $(N, f(N))$-approximate threshold problem requires $\Omega(\log N)$ bits of space.

Since counting is harder than thresholding, the next corollary is an immediate consequence.

Corollary 5.1.5. *Any pseudo-deterministic streaming algorithm which solves approximate counting up to a constant multiplicative factor for stream lengths at most N requires $\Omega(\log N)$ bits of space.*

Remark. Our result also implies a space lower bound for solving a variant of the heavy hitters problem pseudo-deterministically. Consider a version of this problem where given a $\{0, 1\}$-valued stream, we aim to output a bit that appeared at least 10% of the time. It follows from Corollary 5.1.5 that any pseudo-deterministic streaming algorithm solving this problem requires $\Omega(\log N)$ bits of space. Indeed, if it is public information that the first $9N$ bits are 0 and the rest are 1, then we are reduced to the $(N, 81N)$-approximate threshold problem by counting the 1's. Randomized algorithms again can solve the problem with $O(\log \log N)$ bits as it suffices to maintain a pair of Morris counters for the 0's and 1's separately. (Note that while heavy hitters is often considered

to be solvable deterministically in constant space, such solutions typically assume a model where exact counting uses constant space.)

5.1.3 Markov Chain Formulation

It will be helpful to reframe the approximate threshold problem and pseudo-determinism in the language of Markov chains. Note that any b-bit pseudo-deterministic streaming algorithm can be described as a Markov chain on 2^b states. Throughout the rest of the paper, we use this formulation instead, which we make precise below.

Let $\mathcal{M}_n$ be an arbitrary Markov chain on state space $V = \{v_1, \ldots, v_n\}$ with starting state $x_0 = v_1$, and let $x_1, x_2, \ldots$ be the random states at each subsequent time. Moreover, let $U \subseteq V$ be a distinguished subset of V.

Definition 5.1.6 (Markov Chain Solution to Pseudo-Deterministic Approximate Thresholding)**.** We say $(\mathcal{M}_n, U)$ is a $(N, f(N))$-solution to the approximate threshold problem if:

1. $\mathbb{P}[x_t \in U] \geq 2/3$ for all $t \leq N$,
2. $\mathbb{P}[x_t \in U] \leq 1/3$ for all $t \geq f(N)$.

We say $\mathcal{M}_n$ is pseudo-deterministic if $\mathbb{P}[x_t \in U] \in [0, 1/3] \cup [2/3, 1]$ for all t.

We will show the following:

Theorem 5.1.7. For any pseudo-deterministic $(\mathcal{M}_n, U)$ that $(N, f(N))$-solves the approximate threshold problem for some $f(N) > N$, it holds that $n \geq N^{\Omega(1)}$.

It is easy to see that Theorem 5.1.7 implies Theorem 5.1.4 by viewing a streaming algorithm using b bits of memory as a Markov chain on 2^b states. Here U is the set of algorithm states on which the algorithm outputs 1.

5.2 Technical Overview

5.2.1 Illustrative Examples

We begin by describing two examples of Markov chains that fail to solve the pseudo-deterministic approximate threshold problem, but end up being surprisingly illustrative of the general case.

Example 1: Threshold Morris Counter. We first describe how to solve approximate thresholding in the usual randomized (non-pseudo-deterministic) setting, where $M = 10N$ and for $t \in [N, 10N]$, it is *not* required that $\mathbb{P}[x_t \in U] \in [0, 1/3] \cup [2/3, 1]$. We will use a version of a Morris counter with a simple two-state implementation (since we aim to threshold rather than count).

Our Markov chain will have states (v_1, v_2) where v_2 is terminal, and the transition from v_1 to v_2 occurs with probability $\frac{1}{3N}$. Formally, $\mathbb{P}[x_{t+1} = v_2 \mid x_t = v_2] = 1$, and $\mathbb{P}[x_{t+1} = v_2 \mid x_t = v_1] = \frac{1}{3N}$. Setting $U = \{v_1\}$, this chain gives a randomized algorithm for $(N, 10N)$ approximate thresholding. However, this example is not pseudo-deterministic because the function $t \mapsto \mathbb{P}[x_t \in U]$ decays gradually. In particular it is easy to see that $\mathbb{P}[x_{t+1} \in U] \geq 0.99 \cdot \mathbb{P}[x_t \in U]$, so by the discrete-time intermediate value theorem there exists t such that $\mathbb{P}[x_t \in U] \notin [0, 1/3] \cup [2/3, 1]$.

Example 2: Prime Cycles. In our second example, we discuss a futile attempt to solve a simpler version of the approximate threshold problem where it is given that the stream's length is at most $10M = 100N$. Informally speaking, on the first step the Markov chain picks a prime p_i. On all subsequent steps, it simply records the current time modulo p_i. A main idea in our proof, outlined in the next subsection, will be to show that any Markov chain $\mathcal{M}_n$ behaves similarly to this example.

Let $k = [n^{1/4}, 2n^{1/4}]$ and choose k distinct primes $p_1, \ldots, p_k \in [n^{1/3}, 2n^{1/3}]$. Noting that $\sum_{i=1}^{k} p_i \leq n$, we can construct a Markov chain $\mathcal{M}_n$ which contains an initial state v_0 as well as a deterministic p_i-cycle for each $1 \leq i \leq k$. Here a p-cycle consists of vertices $(v_0^{(p)}, \ldots, v_{p-1}^{(p)})$ with dynamics deterministically incrementing the subscript by 1 modulo p each time. At the first timestep, $\mathcal{M}_n$ moves to $x_1 = v_1^{(p_i)}$ for a uniformly random p_i. One could try to solve the pseudo-deterministic approximate threshold problem (with restricted stream length at most $100N$) by choosing a special subset $U \subseteq \bigcup_{1 \leq i \leq k} \{v_0^{(p_i)}, \ldots, v_{p_i-1}^{(p_i)}\}$ of the state space of $\mathcal{M}_n$.

We argue below that this is not possible, which will serve as a useful warmup for the main proof. Note that by the discrete-time intermediate value theorem, for any such solution there must exist $0 \leq T \leq 9N$ so that $\mathbb{P}[x_t \in U] \geq 2/3$ holds for roughly half of the values $t \in [T, T+1, \ldots, T+N]$. To contradict pseudo-determinism, it suffices to show an upper bound

$$\sum_{t \in [T, T+1, \ldots, T+N]} (\mathbb{P}[x_t \in U] - 1/2)^2 \leq o(N). \tag{5.1}$$

Since it involves squares of probabilities, the left-hand side can be rewritten as a sum over all pairs

p_i, p_j of cycle lengths. Moreover $p_i p_j \leq O(n^{2/3}) \ll N$, so the Chinese remainder theorem implies that every pair of distinct primes behaves almost independently on $[T, T+1, \ldots, T+N]$. Combining, it is not difficult to conclude (5.1), thus contradicting pseudo-determinism.

5.2.2 Proof Outline

Suppose for sake of contradiction that the Markov chain $(\mathcal{M}_n, U)$ is pseudo-deterministic and solves the approximate threshold problem. The main idea of our proof is to extract from $\mathcal{M}_n$ a subsystem behaving roughly like Example 2 above, in the sense that it is an ensemble of cycles with different period lengths. We leverage this behavior around a time T where $\mathbb{P}[x_s \in U] \geq 2/3$ holds for roughly half of the values $s \in [T, T + \text{poly}(n)]$, which exists by the discrete-time intermediate value theorem. Finally, we use Fourier analysis to prove there exists some $s \in [T, T + \text{poly}(n)]$ where $\mathbb{P}[x_s \in U]$ is not too biased, contradicting the pseudo-determinism of $(\mathcal{M}_n, U)$.

Recurrent Behavior on Moderate Time-Scales. The first step is to bypass the issue of permanent irreversible transitions as in Example 1 by considering a random time. We choose a random time $1 \leq t \leq \text{poly}(n)$ and show that with high probability the chain *behaves recurrently* at vertex x_t. More precisely, let $t + r$ be the next time at which $x_{t+r} = x_t$ (where $r = \infty$ if the chain never returns). We show in Lemma 5.3.1 that with high probability,

$$\mathbb{P}[r \leq 10n] \geq 1/2, \tag{5.2}$$

$$\mathbb{P}[r \leq n^{18}] \geq 1 - n^{-16}. \tag{5.3}$$

Using this guarantee, we can view the Markov chain from the perspective of the (random) state x_t by considering the sequence of cycle lengths. The bound (5.3) allows us to define this process up to large $\text{poly}(n)$ number of time steps, with all cycle lengths at most n^{18} with high probability. This circumvents the behavior in Example 1: although the chain might eventually transition to a terminal state and never return to x_t, it tends not to do so during a certain $\text{poly}(n)$ time window.

Decomposition into Periodic Parts. Next, using (5.2), it follows that there exists an integer $1 \leq m_v \leq 10n$ such that conditioned on $x_t = v$, each cycle returning to v has probability $\Omega(1/n)$ to have length exactly m_v. We call cycles of length exactly m_v *special*, and condition also on a *masked*

cycle length process $\vec{\ell}$ which hides the special cycles but reveals all other cycle lengths in order, and stops after a moderately large fixed poly(n) number of non-special cycles.[1]

As an explicit example, suppose that at vertex v, there is a $1/4$ probability of arriving back at v after 3 steps, $1/4$ probability of arriving back at v after 5 steps, $1/3$ probability of arriving back at v after 7 steps, and $1/6$ probability of arriving back at v after 11 steps, and let $m_v = 7$. Suppose that on a specific run of the Markov chain starting at v, the sequence of cycles taken are of lengths $(3, 5, 7, 11, 7, 5, 3, 7)$. Then $\vec{\ell} = (3, 5, 11, 5, 3)$.

After this point, the number of hidden m_v cycles is approximately Gaussian with poly(n) standard deviation. In particular, its probability mass function is almost constant on short scales, which lets us approximate the function

$$\mathbb{P}[x_s \in U \mid x_t = v]$$

by an m_v-periodic function for large s. Averaging over the randomness of x_t, we have approximated $\mathbb{P}[x_s \in U]$ by an average of periodic functions with periods $m \leq 10n$.[2] This completes the structural phase of the proof. We now turn to finding a time s such that $\mathbb{P}[x_s \in U] \in [1/3, 2/3]$.

Analysis of Periodic Decomposition. The last step is to analyze this mixture-of-cyclic behavior at a time-region T on which $\mathbb{P}[x_s \in U] \geq 2/3$ holds for roughly half of the values $s \in [T, T+\text{poly}(n)]$; such T exists by the discrete-time intermediate value theorem. If the different periods m_v were relatively prime, intuitively each pair of cyclic behaviors would be approximately independent over the range $[T, T + \text{poly}(n)]$ since $m_v m_v' \leq 100n^2 \ll \text{poly}(n)$. This pairwise almost independence would allow us to upper bound $\sum_{s \in [T, T+\text{poly}(n)]} (\mathbb{P}[x_s \in U] - 1/2)^2$ similarly to Example 2, thus contradicting pseudo-determinism.

To handle general period lengths, the key idea is to divide out common prime factors and restrict to a corresponding arithmetic progression. Precisely, for each prime p we let $e_p \geq 0$ be the largest integer such that m_v has at least some constant probability to be a multiple of p^{e_p}. Then it can be shown that $\beta = \prod_{p \text{ prime}} p^{e_p} \leq \text{poly}(n)$, so we restrict attention to a fixed arithmetic progression with difference β.

Restricting to this arithmetic progression causes the different cycles to behave almost as if

[1] Technically, we will assign length m_v cycles to be non-special with some positive probability. This is necessary if the cycle length *always* equals m_v, for example.

[2] We formalize this approximation using a notion of comparison we call c-covering, see Definition 5.3.4.

they are relatively prime. Precisely, the "reduced period lengths" $M_v = m_v/\gcd(m_v, \beta)$ have the property that any fixed prime p has low probability of dividing M_v. This property does *not* imply pairwise independence of different cycles: it could still be true that $\gcd(M_v, M_v') > 1$ holds with high probability for independent M_v, M_v'. However using Fourier analysis, we were still able to show that it implies a non-trivial upper bound on $\sum_{s\in[T,T+\mathrm{poly}(n)]}(\mathbb{P}[x_s \in U] - 1/2)^2$. As mentioned above, such an upper bound contradicts pseudo-determinism which completes the proof.

5.3 Proof of Theorem 5.1.7

Assume $n \geq n_0$ is sufficiently large and let the threshold $N > n^{34}$. We will show such a Markov chain cannot (N, M)-solve the approximate counting problem for any M. We will replace the intervals $[0, 1/3]$ and $[2/3, 1]$ by $[0, 10^{-4}]$ and $[1 - 10^{-4}, 1]$ in the statement of Theorem 5.1.7, which is equivalent by amplification.

5.3.1 Recurrent Behavior on Moderate Time-Scales

Given any $v \in V$, let μ_v be the *return-time distribution* from v, i.e. the distribution for the smallest $r \geq 1$ such that $x_r = v$, starting from $x_0 = v$. (We let $r = \infty$ if this never holds again.)

Lemma 5.3.1. *There exists $1 \leq t \leq N/100$ and a subset $S \subseteq V$ such that:*

1. $\mathbb{P}[x_t \in S] \geq 1/2$.
2. *For all $v \in S$:*

$$\mathbb{P}_{r\sim\mu_v}(r \in [0, 10n]) \geq 1/2,$$
$$\mathbb{P}_{r\sim\mu_v}(r \in [0, n^{18}]) \geq 1 - n^{-16}.$$

Proof. We first show that at least 0.85-fraction of the values of $1 \leq t \leq N/100$ satisfy:

$$\mathbb{E}_{x_t}\mathbb{P}_{r\sim\mu_{x_t}}(r \in [0, 10n]) \geq 0.85 \tag{5.4}$$

Partition the interval $[1, N/100]$ into intervals $I_j = [1 + (j-1)10n, 10jn]$ of length $10n$ for $1 \leq j \leq N/1000n$. Consider a fixed run of the Markov chain $x_1 \ldots x_{N/100}$. For each interval I_j, at most n

values of z_i for $i \in I_j$ will not return to themselves at some later time in I_j. Thus, in a fixed run of the Markov chain, the return time back to x_t starting from time t is at most $10n$ steps for at least 0.9 fraction of times $1 \le t \le N/100$. Averaging over all runs of the Markov chain, we get that

$$\mathbb{E}_{t\in[1,N/100]}\mathbb{E}_{x_t}\mathbb{P}_{r\sim\mu_{x_t}}(r \in [1, 10n]) \ge 0.9$$

and therefore at least 0.85 fraction of $0 < t < N/100$ satisfy (5.4) by Markov's inequality. By an identical argument, for at least 0.85 fraction of $1 \le t \le N/100$,

$$\mathbb{E}_{x_t}\mathbb{P}_{r\sim\mu_{x_t}}(r \in [0, n^{18}]) \ge 1 - 10n^{-17}$$

In particular, there exists a value of t satisfying both conditions, which will be the value of t in the lemma statement.

Next applying Markov's inequality to (5.4) shows for some $S_1 \subseteq V$ with $\mathbb{P}[x_t \in S_1] \ge 0.6$,

$$\mathbb{P}_{r\sim\mu_v}(r \in [1, 10n]) \ge 1/2.$$

Similarly for some $S_2 \subseteq V$ such that $\mathbb{P}[x_t \in S_2] > 0.9$,

$$\mathbb{P}_{r\sim\mu_v}(r \in [0, n^{18}]) \ge 1 - 100n^{-17} \ge 1 - n^{-16}.$$

Taking $S = S_1 \cap S_2$ completes the proof. □

5.3.2 Decomposition into Periodic Parts

From now on we fix t, S as in the previous subsection. For each $v \in S$, define the period length

$$m_v \equiv \operatorname*{argmax}_{1\le m\le 10n} \mu_v(m).$$

Starting from $x_t = v$, we define a sequence of steps as follows. Let $(C_1, C_2, \dots)$ be the sequence of cycle lengths returning back to location x_t, i.e. $x_{t+C_1+\cdots+C_j}$ for $j \ge 0$ are exactly the times $s \ge t$

with $x_s = x_t$. When $C_i = m_v$, we assign C_i to be *special* with probability

$$\mathbb{P}[C_i \text{ special} \mid C_i = m_v] = \frac{n^{-2}\mu_v(\{1, 2, \dots, n^{18}\})}{\mu_v(m_v)}. \tag{5.5}$$

This choice of probability results in $\mathbb{P}[C_i \text{ special} \mid C_i \le n^{18}] = n^{-2}$, which will be useful later. In all other cases where $C_i \le n^{18}$, we assign C_i to be *typical*. We define the *special-steps sequence* $(s_1, s_2, \dots) = (m_v, m_v, \dots)$ to be the subsequence of special cycle lengths, and the *typical-steps sequence* $(\ell_1, \ell_2, \dots)$ to consist of the typical cycle lengths. Note that cycles of length greater than n^{18} are neither special nor typical.

If $x_t = v$, let τ_v be the first time that n^{14} typical steps have been completed. (We let τ_v be undefined or infinite for all other states v.) Let Y be the number of special cycles until time τ_{x_t}, and let $s_{\text{tot}} = Y m_v$ be the total duration of these special cycles.

For each $v \in S$, let E_v be the event that $x_t = v$, and that all cycles until time τ_v are special or typical (in particular $\tau_v < \infty$ on this event).

Lemma 5.3.2. *For any $v \in S$, we have $\mathbb{P}[E_v \mid x_t = v] \ge 0.99$.*

Proof. Note that each cycle length C_i is independent and has probability at least $1 - n^{-16}$ of being at most n^{18} by Lemma 5.3.1. We find that with high probability the first n^{15} cycle lengths are special or typical:

$$\mathbb{P}\left[\max_{1 \le i \le n^{15}} C_i \le n^{17}\right] \ge 1 - n^{-1}. \tag{5.6}$$

Conditioned on the event in (5.6), each C_i is labeled typical independently with probability at least $1/2$. Hence the number of typical steps among the first n^{15} cycles stochastically dominates a binomial $Bin(n^{15}, 1/2)$ variable, and in particular is at least n^{14} with probability at least $1 - n^{-1}$. On this event, τ_v occurs during the first n^{15} cycles and so E_v holds with overall probability at least $(1 - n^{-1})(1 - n^{-1}) \ge 0.99$. □

Next, we use our concept of special and typical cycles to describe the distribution of τ_v, which we recall is the first time that n^{14} typical steps have been completed conditioned on $x_t = v$.

Definition 5.3.3. A probability distribution ν on $\mathbb{Z}$ is **m-interval-periodic** if it is supported on a discrete interval $\{I, I+1, \dots, J\}$ and within this interval, $\nu(j) = \nu(j+m)$ depends only on j mod m. The **range** of ν is $J - I$.

Similarly a sequence $\vec{\nu} = (\nu_I, \nu_{I+1}, \ldots, \nu_J)$ of probability distributions on V is m-interval-periodic if $\nu_j = \nu_{j+m}$ depends only on $j \mod m$.

Definition 5.3.4. Let μ, ν be probability distributions on a countable set $\mathcal{I}$ and $0 < c < 1$ be a constant. We say μ is a c-cover for ν if $\mu(i) \geq c\nu(i)$ for all $i \in \mathcal{I}$.

Let $\mu_{\text{tot},v,\vec{\ell}}$ be the distribution of τ_v conditionally on the event E_v and the sequence $\vec{\ell}$ of n^{14} typical steps. In fact, it is easy to see from (5.5) that conditionally on (v, E_v) the number z_i of special steps between each adjacent pair (ℓ_{i-1}, ℓ_i) of typical steps is exactly the geometric distribution $\text{Geom}(1 - n^{-2})$. Thus $\mathbb{P}[z_i = j] = (1 - n^{-2})n^{-2j}$ for each $j \in \mathbb{Z}_{\geq 0}$, and the mean and variance are known to be

$$\mathbb{E}[z_i] = \frac{1}{n^2 - 1},$$
$$\text{Var}[z_i] = \frac{n^2}{(n^2 - 1)^2}.$$

Lemma 5.3.5. *For each $v \in S$ and $\vec{\ell}$, there exists an m_v-interval-periodic distribution $\nu_{v,\vec{\ell}}$ with range $n^6 < w_v < n^8$ such that the distribution $\mu_{tot,v,\vec{\ell}}$ is a 0.06-cover of $\nu_{v,\vec{\ell}}$.*

Proof. Let $Z = \sum_{i=1}^{n^{14}} z_i$ be the total number of special cycles until time τ_v. We first show that Z obeys a central limit theorem as $n \to \infty$. Note that $w_i = z_i - \mathbb{E}[z_i]$ satisfies $\mathbb{E}[w_i] = 0$, $\mathbb{E}[w_i^2] = \Theta(n^{-2})$ and $\mathbb{E}[|w_i|^3] = \Theta(n^{-2})$. Hence it follows from the Berry-Esseen theorem that with $\Phi(A) = \mathbb{P}^{y \sim \mathcal{N}(0,1)}[y \leq A]$ the cumulative distribution function of the standard Gaussian, and μ_Z, σ_Z the mean and standard deviation of Z:

$$\sup_{A \in \mathbb{R}} |\mathbb{P}\left[\frac{Z - \mu_Z}{\sigma_Z} \leq A\right] - \Phi(A)| \leq \frac{Cn^{-2}}{n^{-3}\sqrt{n^{14}}} \leq O(1/n^2).$$

In particular, for n large enough,

$$\mathbb{P}[Z \in [\mu_Z - 2\sigma_Z, \mu_Z - \sigma_Z]] \geq \Phi(-1) - \Phi(-2) - 0.001 \geq 0.13,$$
$$\mathbb{P}[Z \in [\mu_Z + \sigma_Z, \mu_Z + 2\sigma_Z]] \geq \Phi(2) - \Phi(1) - 0.001 \geq 0.13.$$

By averaging, there exist integers $a_1 \in [\mu_Z - 2\sigma_Z, \mu_Z - \sigma_Z]$ and $a_2 \in [\mu_Z + \sigma_Z, \mu_Z + 2\sigma_Z]$ with

$$\mathbb{P}[Z = a_1], \mathbb{P}[Z = a_2] \geq \frac{1}{8\sigma_Z}.$$

Next, we claim the probability mass function of Z is log-concave; this implies that

$$\mathbb{P}[Z = a] \geq \frac{1}{8\sigma_Z}, \quad \forall a \in [\mu_Z - \sigma_Z, \mu_Z + \sigma_Z]. \tag{5.7}$$

Since $\mu_{\mathrm{tot},v,\vec{\ell}}$ is exactly the law of $m_v Z$ and $\sigma_Z = n^7\sqrt{\mathrm{Var}[z_1]} = n^6(1 \pm O(1/n))$, the statement (5.7) readily implies the desired result. It remains only to prove that Z has log-concave probability mass function.

For this, first note that Z is a negative binomial distribution and its probability mass function takes the exact form

$$\mathbb{P}[Z = k] = \binom{k + n^{14} - 1}{n^{14} - 1} n^{-14k}(1 - n^{-2})^{n^{14}}.$$

(This description is well-known and follows from an elementary stars and bars argument.) Hence it suffices to show that $k \mapsto \binom{k+n^{14}-1}{n^{14}-1}$ is log-concave. For this we note that $k \mapsto k + c$ is log-concave for each $c \geq 0$ on $k \in \mathbb{Z}_{\geq 0}$ and

$$\binom{k + n^{14} - 1}{n^{14} - 1} = \frac{(k + n^{14} - 1)(k + n^{14} - 2) \cdots (k + 1)}{(n^{14} - 1)!}$$

is proportional to a product of such sequences. □

We remark that the conditioning on $\vec{\ell}$ played no role above as it just shifts $\mu_{tot,v,\vec{\ell}}$. However, in the next subsection it will be important to apply Lemma 5.3.5 after conditioning on $\vec{\ell}$.

For each $v \in S$, typical-steps sequence $\vec{\ell}$, and time $s > t$, define the probability distribution

$$\widetilde{\mu}_s = \widetilde{\mu}_s(v, \vec{\ell})$$

on V to be the conditional law of x_s given $(E_v, \vec{\ell})$.

Lemma 5.3.6. *For each $v \in S$ and $\vec{\ell}$ and $T' > n^{33}$, there exists an m_v-interval-periodic sequence $\vec{\omega} = \vec{\omega}_v = (\omega_{T'}, \ldots, \omega_{T'+n^5})$ of distributions that is 0.03-covered by the sequence $\widetilde{\mu}_{T'} \ldots \widetilde{\mu}_{T'+n^5}$.*

Proof. For any vertex u and time r, define $P^r(u)$ to be the distribution of the Markov chain state

after r steps starting from vertex u. Let

$$\mathrm{supp}(\nu_{v,\vec{\ell}}) = \{I, I+1, \ldots, J\} = \{I(\nu_{v,\vec{\ell}}), I(\nu_{v,\vec{\ell}})+1, \ldots, J(\nu_{v,\vec{\ell}})\}$$

be the support of $\nu_{v,\vec{\ell}}$. Moreover let $\overline{\nu}_{v,\vec{\ell}}(0), \ldots, \overline{\nu}_{v,\vec{\ell}}(m_v - 1)$ be such that $\nu_{v,\vec{\ell}}(j) = \overline{\nu}_{v,\vec{\ell}}(j \mod m)$ for $I \le j \le J$.

We explicitly compute $\widetilde{\mu}_s$ for $T' \le s \le T' + n^5$. With inequalities of positive measures indicating that the difference is a positive measure, we have

$$\begin{aligned}
\widetilde{\mu}_s(v, \vec{\ell}) &\ge \sum_{t=1}^{s} \mathbb{P}[\tau_v = t | (x_t = v, \vec{\ell})] \cdot P^{s-t}(v) \\
&\ge 0.06 \sum_{t=1}^{s} \nu_{v,\vec{\ell}}(t) \cdot P^{s-t}(v) \\
&= 0.06 \sum_{u=I-T'}^{J-T'-n^5} \nu_{v,\vec{\ell}}(s-u) \cdot P^u(v) \\
&\ge 0.06 \left(\frac{J - I - n^5 - 2m}{m} \right) \cdot \sum_{i=1}^{m} \overline{\nu}_{v,\vec{\ell}}(s - i \mod m) \sum_{\substack{I-T' \le u \le J-T'-n^5 \\ u \equiv i \mod m}} P^u(v).
\end{aligned}$$

Note that this lower bound is exactly m-periodic for s in the stated range. We now show it has a significant total mass. Note that

$$\begin{aligned}
&\left(\frac{J - I + n^5 + 2m}{m} \right) \cdot \sum_{i=1}^{m} \overline{\nu}_{v,\vec{\ell}}(s - i \mod m) \sum_{\substack{I-T' \le u \le J-T'-n^5 \\ u \equiv i \mod m}} P^u(v) \\
&\ge \sum_{t=1}^{s} \nu_{v,\vec{\ell}}(t) \cdot P^{s-t}(v)
\end{aligned}$$

has total mass at least 0.99 since clearly $\mathbb{P}[\tau_v < s \mid E_v] \ge 0.99$ (by Markov's inequality on s_{tot}).

Since $J - I \ge n^6$ we have

$$\frac{J - I - n^5 - 2m}{J - I + n^5 + 2m} \ge 0.99$$

so we conclude that the lower bound above has a total mass of at least 0.03. □

5.3.3 Analysis of Periodic Decomposition

We fix the values m_v and distributions $\nu_{v,\vec{\ell}}$ from Lemma 5.3.5. Next, we construct a "global period length" β as follows. For each prime p, let e_p be the largest value such that

$$\mathbb{P}[p^{e_p} \text{ divides } m_v \mid v \in S] \geq 0.55$$

and define $\beta = \prod_p p^{e_p}$.

Lemma 5.3.7. *It holds that $\beta \leq n^2$.*

Proof. It suffices to observe that

$$\log(10n) \geq \mathbb{E}[\log(m_v) \mid v \in S] \geq \log(\beta^{0.55}).$$

The former holds since $m_v \leq 10n$ while the latter holds by Markov's inequality. Hence $\beta \leq (10n)^{20/11} \leq n^2$. □

Let $M_v = m_v / \gcd(m_v, \beta)$. By definition, for any prime p we have

$$\mathbb{P}[p \text{ divides } M_{x_t} \mid x_t \in S] \leq 0.55. \tag{5.8}$$

Recalling Subsection 5.1.3, for each time s let $F(s) = 0$ if $\mathbb{P}[x_s \in U] \in [0, 1/3]$ and $F(s) = 1$ otherwise. Moreover define

$$\overline{F}_a = \frac{1}{\lfloor n^5/\beta \rfloor} \sum_{i=1}^{\lfloor n^5/\beta \rfloor} F((a+i)\beta).$$

Lemma 5.3.8. *There exists $a \in \mathbb{N}$ such that $|\overline{F}_a - 1/2| \leq 0.01$.*

Proof. This follows by the discrete-time intermediate value theorem. Indeed since $N \geq n^5$ we have $\overline{F}_0 = 1$, while $\overline{F}_a = 0$ for $a \geq f(N)$. Clearly $|\overline{F}_{a+1} - \overline{F}_a| \leq 0.01$ for all a. □

Throughout the rest of the proof, let $T = a\beta$ for a as in Lemma 5.3.8. For each time $T \leq s \leq T + n^5$, let

$$\overline{\omega}_s = \mathbb{E}^{x_t, \vec{\ell}} \left[\mathbb{P}^{v \sim \omega_{x_t, \ell, s}} [v \in U \,] \mid x_t \in S \text{ and } E_{x_t} \right].$$

(Note that $\{x_t \in S \text{ and } E_{x_t}\}$ is just an event.)

Lemma 5.3.9. *$\overline{\omega}_s \geq 0.99$ if $F(s) = 1$ and $\overline{\omega}_s \leq 0.01$ if $F(s) = 0$.*

Proof. We focus on the latter statement since they are symmetric.

Given x_t, generate $w \sim P^{s-t}(x_t)$ to be the state of the Markov chain at time s drawn from the original trajectory distribution. By Bayes' rule for expectations, since the event $\{x_t \in S \text{ and } E_{x_t}\}$ has probability at least $1/3$, conditioning on it increases the expectation of any non-negative random variable by a factor at most 3. In particular:

$$\begin{aligned}\mathbb{E}^{x_t,\vec{\ell}}[\mathbb{P}[w \in U\ ] \mid x_t \in S \text{ and } E_{x_t}] &\leq \frac{\mathbb{E}^{x_t,\vec{\ell}}[\mathbb{P}[w \in U\ ]]}{\mathbb{P}[x_t \in S \text{ and } E_{x_t}]} \\ &= \frac{\mathbb{P}[x_s \in U]}{\mathbb{P}[x_t \in S \text{ and } E_{x_t}]} \\ &\leq 3 \cdot 10^{-4}.\end{aligned}$$

Given x_t and $\vec{\ell}$, we also let $\widetilde{w} \sim \omega_{x_t,\vec{\ell},s}$ for $\omega_{x_t,\vec{\ell},s}$ as in Lemma 5.3.6. Then the 0.03-covering guarantee in Lemma 5.3.6 directly implies

$$\begin{aligned}&\mathbb{E}^{x_t,\vec{\ell}}[\mathbb{P}[w \in U\ ] \mid x_t \in S \text{ and } E_{x_t}] \\ &\geq 0.03 \cdot \mathbb{E}^{x_t,\vec{\ell}}[\mathbb{P}[\widetilde{w} \in U\ ] \mid x_t \in S \text{ and } E_{x_t}] \\ &= 0.03 \cdot \overline{\omega}_s.\end{aligned}$$

Combining completes the proof. □

Lemma 5.3.10. *It holds that*

$$\left| \frac{1}{\lfloor n^5/\beta \rfloor} \sum_{i=1}^{\lfloor n^5/\beta \rfloor} \overline{\omega}_{(a+i)\beta} - \frac{1}{2} \right| \leq 0.02.$$

Proof. By Lemma 5.3.9, $|\overline{\omega}_s - F(s)| \leq 0.01$ for each s. Hence

$$\left| \frac{1}{\lfloor n^5/\beta \rfloor} \sum_{i=1}^{\lfloor n^5/\beta \rfloor} \overline{\omega}_{(a+i)\beta} - \frac{1}{2} \right| \leq |\overline{F}_a - 1/2| + 0.01 \leq 0.02.$$

□

We will use the following lemma which is proved in Section 5.4.

Lemma 5.3.11. *The following holds with $L = n^3$. Fix n and let $S = (S_1, S_2, \dots, S_L) \in [-1,1]^L$ be a random sequence which is m-periodic for some random $m = m(S) \le 10n$. Suppose that*

$$-L/10 \le \mathbb{E} \sum_{t=1}^{L} S_t \le L/10 \tag{5.9}$$

and that for each prime p,

$$\mathbb{P}[p \text{ divides } m] \le 0.55.$$

Then there exists an index $1 \le t \le L$ such that

$$|\mathbb{E}[S_t]| \le 0.9.$$

Combining the above, we can finally prove Theorem 5.1.7.

Proof of Theorem 5.1.7. Given a putative $(\mathcal{M}_n, U)$, we apply Lemma 5.3.11 with

$$S = S_{x_t,\vec{\ell}} = \mathbb{P}^{v \sim \omega_{x_t,\ell,s}}[v \in U].$$

where we map $[0,1] \to [-1,1]$ via $a \mapsto 2a - 1$. The first condition holds by Lemma 5.3.10 since $\mathbb{E}[S_t] = \overline{\omega}_{(a+t)\beta}$. The second condition holds by (5.8). Hence Lemma 5.3.11 implies that there exists s such that $\overline{\omega}_s \in [0.02, 0.98]$. This contradicts Lemma 5.3.9 above, giving a contradiction and completing the proof. □

5.4 Proof of Lemma 5.3.11

Here we prove Lemma 5.3.11 using Fourier analysis. For each $m \ge 1$, let

$$Z_m = \{0, 1/m, \dots, (m-1)/m\}$$

and set $Z_m^* = Z_m \backslash \{0\}$. Given a m-periodic sequence $S = (S_1, S_2, \dots)$ let $\widehat{S} : [0,1) \to \mathbb{C}$ be its Fourier transform

$$\widehat{S}(\alpha) = \begin{cases} \frac{1}{m} \sum_{j=1}^{m} e^{2\pi i j \alpha} S_j, & \alpha \in Z_m, \\ 0, & \alpha \notin Z_m \,. \end{cases} \tag{5.10}$$

The definition (5.10) agrees with the Fourier transform of S viewed as a periodic function on $\mathbb{Z}$ and hence is independent of the period m — we could view S as being km periodic for any $k \geq 1$ and $\widehat{S}$ would remain consistent. This allows us to use Fourier analysis on pairs S, S' of sequences with different period lengths m, m'. Indeed with $M = mm'$ we can define:

$$\langle S, S' \rangle = \frac{1}{M} \sum_{i=1}^{M} S_i S'_i \tag{5.11}$$

$$\|S\|_{L^2}^2 = \frac{1}{M} \sum_{i=1}^{M} S_i^2 \tag{5.12}$$

$$\langle \widehat{S}, \widehat{S}' \rangle = \sum_{\alpha \in Z_M} \widehat{S}(\alpha) \widehat{S}'(\alpha) \tag{5.13}$$

$$\|\widehat{S}\|_{\ell^2}^2 = \sum_{\alpha \in Z_m} |\widehat{S}(\alpha)|^2. \tag{5.14}$$

Parseval identity's modulo M implies that $\langle S, S' \rangle = \langle \widehat{S}, \widehat{S}' \rangle$ and $\|S\|_{L^2} = \|\widehat{S}\|_{L^2}$.

The next proposition approximates averages on $1, 2, \ldots, L$ by Fourier averages. Here and below we let μ be the law of the random sequence S, and recall that $S \sim \mu$ is $m(S)$ periodic for $m \leq 10n$ almost surely.

Lemma 5.4.1. *Then*

$$\left| \frac{1}{L} \sum_{t=1}^{L} \mathbb{P}^{S \sim \mu}[S_t = 1]^2 - \mathbb{E}^{S, S' \sim \mu} \langle \widehat{S}, \widehat{S}' \rangle \right| \leq O(n^2/L), \tag{5.15}$$

$$\left| \left(\frac{1}{L} \mathbb{E}^{S \sim \mu} \sum_{t=1}^{L} S_t \right)^2 - \mathbb{E}^{S, S' \sim \mu} [\widehat{S}(0) \widehat{S}'(0)] \right| \leq O(n^2/L). \tag{5.16}$$

Proof. Notice that by definition,

$$\frac{1}{L} \sum_{t=1}^{L} \mathbb{P}^{S \sim \mu}[S_t = 1]^2 = \frac{1}{L} \mathbb{E}^{S, S' \sim \mu} \big\langle (S_1 \ldots, S_L), (S'_1, \ldots, S'_T) \big\rangle$$

where $\langle (S_1 \ldots, S_L), (S'_1, \ldots, S'_T) \rangle = \frac{1}{L} \sum_{t=1}^{L} S_t S'_t$. Since $M = mm' \leq 100n^2$,

$$\left| \frac{1}{L} \mathbb{E}^{S, S' \sim \mu} \big\langle (S_1 \ldots, S_L), (S'_1, \ldots, S'_T) \big\rangle - \langle S, S' \rangle \right| \leq O(n^2/L).$$

Combining and using $\langle S, S' \rangle = \langle \widehat{S}, \widehat{S}' \rangle$ by Parseval completes the proof of (5.15). The proof of (5.16) is similar since

$$\left| \frac{1}{L} \mathbb{E}^{S \sim \mu} \sum_{t=1}^{L} S_t - \mathbb{E}^{S \sim \mu}[\hat{S}(0)] \right| = \left| \frac{1}{L} \mathbb{E}^{S \sim \mu} \sum_{t=1}^{L} S_t - \frac{1}{m(S)} \mathbb{E}^{S \sim \mu} \sum_{i=1}^{m(S)} S_i \right| \leq O(n/L).$$

Indeed $(\mathbb{E}^{S \sim \mu}[\widehat{S}(0)])^2 = \mathbb{E}^{S, S' \sim \mu}[\widehat{S}(0)\widehat{S}'(0)]$, and the function $x \mapsto x^2$ is Lipschitz on $x \in [-1, 1]$. □

The main idea to prove Lemma 5.3.11 is that S, S' are unlikely to have similar periods. In carrying out this argument we have to handle the bias $\widehat{S}(0)$ separately from the $\alpha \in (0, 2\pi)$ contributions. The next proposition gives an estimate to handle the latter contributions; note it is important that the last term below mixes S, S'.

Lemma 5.4.2. *For m and m'-periodic functions $S, S' : \mathbb{Z} \to [-1, 1]$,*

$$\langle \widehat{S}, \widehat{S}' \rangle \leq \widehat{S}(0)\widehat{S}'(0) + \| \widehat{S}|_{Z^*_{m'}} \|_{\ell^2}$$

where $\widehat{S}|_{Z^*_{m'}}$ denotes the restriction of $\widehat{S}$ to $Z^*_{m'}$.

Proof. The summand $\widehat{S}(\alpha)\widehat{S}'(\alpha)$ in (5.13) is non-zero only when $\alpha \in Z_g$ for $g = \gcd(m, m')$. This yields an upper bound of

$$\widehat{S}(0)\widehat{S}'(0) + \| \widehat{S}|_{Z^*_g} \|_{\ell^2} \cdot \| \widehat{S}'|_{Z^*_g} \|_{\ell^2}.$$

This immediately implies the claim since

$$\| \widehat{S}|_{Z^*_g} \|_{\ell^2} \leq \| \widehat{S}|_{Z^*_{m'}} \|_{\ell^2}$$

(in fact, equality holds) while

$$\| \widehat{S}'|_{Z^*_g} \|_{\ell^2} \leq \| \widehat{S}'|_{Z_M} \|_{\ell^2} = \| S' \|_{L^2} \leq 1.$$

□

Lemma 5.4.3. *Suppose $S, S' \overset{i.i.d.}{\sim} \mu$ and $\mathbb{P}^{S \sim \mu}[p \text{ divides } m(S)] \leq \varepsilon$ for any prime p. Then*

$$\mathbb{E}^{S \sim \mu}\big[\| \widehat{S}|_{Z^*_{m'}} \|_{\ell^2} \big] \leq \sqrt{\varepsilon}$$

Proof. Directly by the assumption, for any fixed $\alpha \neq 0$, we have $\mathbb{P}[\alpha \in Z^*_{m'}] \leq \varepsilon$. Thus linearity of expectation implies

$$\mathbb{E}^{S\sim\mu}\big[\| \widehat{S}|_{Z^*_{m'}}\|^2_{\ell^2}\big] \leq \varepsilon.$$

Applying Jensen's inequality completes the proof. □

Proof of Lemma 5.3.11. By applying Markov's inequality to (5.15) and noting $n^2/L \leq 1/n$ it suffices to show that

$$\mathbb{E}^{S,S'\sim\mu}\langle \widehat{S}, \widehat{S}'\rangle \overset{?}{\leq} 0.8 < 0.9^2. \tag{5.17}$$

We first deal with the bias term of the left-hand side. Combining (5.16) and (5.9) yields

$$\mathbb{E}^{S,S'\sim\mu}[\widehat{S}(0)\widehat{S}'(0)] \leq 0.02.$$

For the remaining terms $\alpha \neq 0$, combining Proposition 5.4.2 with Lemma 5.4.3 (which holds with $\varepsilon = 0.55$ by assumption) yields

$$\mathbb{E}^{S,S'\sim\mu} \sum_{\alpha\in Z^*_M} \widehat{S}(\alpha)\widehat{S}'(\alpha) \leq \sqrt{0.55} \leq 0.75.$$

Summing the contributions establishes (5.17) and thus finishes the proof. □

Chapter 6

Pseudo-deterministic Proofs

The work in this Chapter is based on joint work with Shafi Goldwasser, and Dhiraj Holden [GGH17].

6.1 Introduction

In this chapter we extend the study of pseudo-determinism in the context of probabilistic algorithms to the context of interactive proofs and non-determinism. We view pseudo-deterministic interactive proofs as a natural extension of pseudo-deterministic randomized polynomial time algorithms: the goal of the latter is to *find* canonical solutions to search problems whereas the goal of the former is to *prove* that a solution to a search problem is canonical to a probabilistic polynomial time verifier. This naturally models the cryptographic setting when an authority generates system-wide parameters (e.g. an elliptic curve for all to use or a generator of a finite group) and it must prove that the parameters were chosen properly.

6.1.1 Our Contribution

Consider the search problem of finding a large clique in a graph. A nondeterministic efficient algorithm for this problem exists: simply guess a set of vertices C, confirm in polynomial time that the set of vertices forms a clique, and either output C or reject if C is not a clique. Interestingly, in addition to being nondeterministic, there is another feature of this algorithm; on the same input graph there may be many possible solutions to the search problem and any one of them may be produced as output. Namely, on different executions of the algorithm, on the same input graph G,

one execution may guess clique C and another execution may guess clique $C' \neq C$, and both are valid accepting executions.

A natural question is whether for each graph with a large clique, there exists a unique canonical large clique C which can be verified by a polynomial time verifier: that is, can the verifier V be convinced that the clique C is the canonical one for the input graph? Natural candidates which come to mind, such as being the lexicographically smallest large clique, are not known to be verifiable in polynomial time (but seem to require the power of Σ_2 computation).

In this paper, we consider this question in the setting of interactive proofs and ask whether the interactive proof mechanism enables provers to convince a probabilistic verifier of the "uniqueness of their answer" with high probability.

Pseudo-deterministic Interactive Proofs: We define *pseudo-deterministic interactive proofs* for a search problem R (consisting of pairs $(instance, solution)$) with associated function s as a pair of interacting algorithms: a probabilistic polynomial time verifier and a computationally unbounded prover which on a common input instance x engage in rounds of interaction at the end of which with high probability the verifier output a canonical solution $y = s(x)$ for x if any solution exists and otherwise rejects x. Analogously to the case of completeness in interactive proofs for languages, we require *Canonical Completeness*: that for every input x, there exists an honest prover which can send the correct solution $s(x)$ to the verifier when one exists. Analogously to the case of soundness, we require *Canonical Soundness*: no dishonest prover can cause the verifier to output a solution other than $s(x)$ (the canonical one) (except with very low probability).

One may think of the powerful prover as aiding the probabilistic polynomial time verifier to find canonical solutions to search problems, with high probability over the randomness of the verifier. The challenge is that pseudo-determinism should hold not only with respect to the randomness, but also with respect to the prover: a malicious prover should not be able to cause the verifier to output a solution other than the canonical unique one. In addition to the intrinsic complexity theoretic interest in this problem, *consistency* or *predictability* of different executions on the same input are natural requirements from protocols.

We define *pseudo-deterministic IP* (psdIP) to be the class of search problems R (relation on inputs and solutions) for which there exists a pseudo-deterministic polynomial round interactive proof for R.

Theorem: For any problem L in NP, there is a pseudo-deterministic polynomial round interactive proof for the search problem R consisting of all pairs (x, w) where $x \in L$ and w is a witness for x.

One can prove the above theorem by noting that finding the lexicographically first witness w for x is a problem in PSPACE. Then, since IP = PSPACE [Sha92], we know an interactive proof for finding the lexicographically first witness w exists. More formally we have:

Proof: Let us consider the function $f(x)$ which outputs the lexicographically first witness that $x \in L$ if $x \in L$ or $\perp$ otherwise. It is easy to see that determining whether $f(x) = y$ is in PSPACE. As a result, there is an polynomial-round IP protocol to determine whether $f(x) = y$. Then, the psdIP protocol is as follows; the prover gives the verifier y and then they run the protocol, and the verifier accepts and outputs y if the protocol accepts. This satisfies the conditions for pseudo-determinism because of the completeness and soundness properties of the IP protocol.

In light of the above, we ask: do ***c****onstant-round pseudo-deterministic interactive proofs exist for hard problems in* NP *for which many witnesses exist?* We let psdAM refer to those pseudo-deterministic interactive proofs in which a constant number of rounds is used. [1].

Graph Isomorphism is in pseudo-deterministic AM**:** Theorem 6.3.1: *There exists a pseudo-deterministic constant-round Arthur-Merlin protocol for finding an isomorphism between two given graphs.*

Recall that the first protocol showing graph non-isomorphism is in constant round IP was shown by [GMW91] and later shown to be possible using public coins via the general transformation of private to public coins [GS86]. Our algorithm finds a unique isomorphism by producing the lexicographically first isomorphism. In order to prove that a particular isomorphism between input graph pairs is lexicographically smallest, the prover will prove in a sequence of sub-protocols to the verifier that a sequence of graphs suitably defined are non-isomorphic. In an alternative construction, we exhibit an interactive protocol that computes the automorphism group of a graph in a verifiable fashion.

SAT is not in pseudo-deterministic AM**:** Theorem 6.4.2: *if any* NP*-complete problem has a a pseudo-deterministic constant round* AM *protocol, then,* NP $\subseteq$ coNP*/poly and the polynomial hierarchy collapses to the third level, showing that it is unlikely that NP complete problems have*

[1]We note that historically, the class AM referred to protocols in which the verifiers' messages consisted of his coin tosses, namely public-coin protocols. In this work, we use AM to refer to constant round interactive proofs

pseudo-deterministic constant round AM *protocols.*

This result extends the work of [HNOS96] which shows that if there are polynomial time unique verifiable proofs for SAT, then the polynomial hierarchy collapses. Essentially, their result held for deterministic interactive proofs (i.e., NP), and we extend their result to probabilistic interactive proofs with constant number of rounds (i.e., AM).

Every problem in search-BPP **is in subexponential-time pseudo-deterministic** MA**:** Theorem 6.5.4: *For every problem in search-*BPP*, there exists a pseudo-deterministic* MA *protocol where the verifier takes subexponential time on infinitely many input lengths.*

The idea of the result is to use known circuit lower bounds to get pseudo-deterministic subexponential time MA protocols for problems in search-BPP for infinitely many input lengths. We remark that Oliveira and Santhanam [OS16] showed a subexponential time pseudo-deterministic algorithm for infinitely many input lengths for all properties which have inverse polynomial density and are testable in probabilistic polynomial time. (An example of such a property is the property of being prime, as the set of primes has polynomial density.) In their construction, the condition of high density is required in order for the property to intersect with their subexponential-size hitting set. (Subsequent work in [Hol17] also drops this requirement but only results in an average-case pseudo-deterministic algorithm.) In the case of MA, unconditional circuit lower bounds for MA with a verifier which runs in exponential time have been shown by Miltersen et al [MVW99], which allows us to no longer require inverse polynomial density. Hence, we can obtain a pseudo-deterministic MA algorithm from circuit lower bounds. Thus, compared to [OS16], our result shows a pseudo-derandomization (for a subexponential verifier and infinitely many input sizes n) for all problems in search-BPP (and not just those with high density), but requires a prover.

Pseudo-deterministic NL equals search-NL: Theorem 6.6.2: *For every search problem in search-*NL*, there exists a pseudo-deterministic* NL *protocol.*

We define *pseudo-deterministic NL* to be the class of search problems R (a relation on inputs and solutions) for which there exists log-space non-deterministic algorithm M (Turing machines) such that for every input x, there exists a unique $s(x)$ such that $R(x, s(x)) = 1$ and $M(x)$ outputs $s(x)$ or rejects x. Namely, there are no two accepting paths for input x that result in different outputs.

To prove the above theorem, we look at the problem of directed connectivity (that is, given a

directed graph G with two vertices s and t, we find a path from s to t), and we show that it is possible to find the lexicographically first path of shortest length in NL. To do so, we first find the length d of the shortest path, which can be done in NL. Then, we find the lexicographically first outneighbor u of s such that there is a path of length $d-1$ from u to t. This can be done by going in order over all outneighbors of s, and for each of them checking if there is a path of length $d-1$ to t (if there is not such a path, that can be demonstrated in NL since NL = coNL [Imm88, Sze88]). By recursively applying this protocol to find a path from u to t, we end up obtaining the lexicographically first path of shortest length, which is unique.

Structural Results: We show a few structural results regarding pseudo-deterministic interactive proofs In Section 6.7. Specifically, we show that psdAM equals to the class search$-\mathrm{P}^{\mathrm{promise}-(\mathrm{AM}\cap\mathrm{coAM})}$, where for valid inputs x, all queries to the oracle must be in the promise. We show similar results in the case of pseudo-deterministic MA and pseudo-deterministic NP.

6.1.2 Other Related Work

In their seminal paper on NP with unique solutions, Valiant and Vazirani asked the following question: is the inherent intractability of NP-complete problems caused by the fact that NP-complete problems have many solutions? They show this is not the case by exhibiting a problem – SAT with unique solutions – which is NP-hard under randomized reductions. They then showed how their result enables to show the NP-hardness under randomized reductions for a few related problems such as parity-SAT. We point out that our question is different. We are not restricting our study to problems (e.g. satisfiable formulas) with unique solutions. Rather, we consider hard problems for which there may be exponentially many solutions, and ask if one can focus on one of them and verify it in polynomial time. In the language of satisfiability, ϕ can be any satisfiable formula with exponentially many satisfying assignments; set $s(\phi)$ to be a unique valued function which outputs a satisfying assignment for ϕ. We study whether there exists an s which can be efficiently computed, or which has an efficient interactive proof.

The question of computing canonical labellings of graphs was considered by Babai and Luks [BL83] in the early eighties. Clearly graph isomorphism is polynomial time reducible to computing canonical labellings of graphs (compute the canonical labeling for your graphs and compare), however it is unknown whether the two problems are equivalent (although finding canonical labellings

in polynomial time seems to be known for all classes of graphs for which isomorphism can be computed in polynomial time). The problem of computing a set of generators (of size $O(\log n)$) of the automorphism group of a graph G was shown by Mathon [Mat79] (among other results) to be polynomial-time reducible to the problem of computing the isomorphism of a graph. We use this in our proof that graph isomorphism is in psdAM.

A line of work on search vs decision and hierarchy collapses, some in the flavor of our result of Section 6.4, have appeared in [HNOS96, HN17, HHM13, CCHO05].

Finally, we mention that another notion of uniqueness has been studied in the context of interactive proofs by Reingold et al [RRR16], called *unambiguous interactive proofs* where the prover has a unique successful strategy. This again differs from pseudo-deterministic interactive proofs, in that we don't assume (nor guarantee) a unique strategy by the successful prover, we only require that the prover proves that the solution (or witness) the verifier receives is unique (with high probability).

6.1.3 Subsequent Work

In [Hol17], inspired by this work, Holden shows that for every BPP search problem there exists an algorithm A which for infinitely many input lengths n and for every polynomial-time samplable distribution over inputs of length n runs in subexponential time and produces a unique answer with high probability on inputs drawn from the distribution and over A's random coins.

[Hol17] expands on the work [OS16] of Oliveira and Santhanam in several ways. Recall that the work of Oliveira and Santhanam shows a subexponential time pseudo-deterministic algorithm for infinitely many input lengths for all properties which have inverse polynomial density and are testable in probabilistic polynomial time. Whereas [OS16] give a pseudo-deterministic algorithm for estimating the acceptance probability of a circuit on inputs of a given length, [Hol17] applies to general search-BPP problems, where the input is a string of a given length over some alphabet and algorithm's A goal is to output a solution that satisfies a BPP testable relation with the input string. Holden [Hol17] shows that for infinitely many input lengths, average-case (over the input distribution) pseudo-deterministic algorithms are possible for problems in search-BPP.

6.2 Definitions of Pseudo-deterministic Interactive Proofs

In this section, we define pseudo-determinism in the context of nondeterminism and interactive proofs. We begin by defining a search problem.

Definition 6.2.1 (Search Problem)**.** A *search problem* is a relation R consisting of pairs (x, y). We define L_R to be the set of x's such that there exists a y satisfying $(x, y) \in R$. An algorithm solving the search problem is an algorithm that, when given $x \in L_R$, finds a y such that $(x, y) \in R$. When L_R contains all strings, we say that R is a *total* search problem. Otherwise, we say R is a *promise* search problem.

We now define pseudo-determinism in the context of interactive proofs for search problems. Intuitively speaking, we say that an interactive proof is pseudo-deterministic if an honest prover causes the verifier to output the same unique solution with high probability (canonical completeness), and dishonest provers can only cause the verifier to output either the unique solution or $\perp$ with high probability (canonical soundness). In other words, dishonest provers cannot cause the verifier to output an answer which is not the unique answer. Additionally, we have the condition that for an input x with no solutions, for all provers the verifier will output $\perp$ with high probability (standard soundness). We note that we use psdIP, psdAM, psdNP, psdMA, and so on, to refer to a class of promise problems, unless otherwise stated.

Definition 6.2.2 (Pseudo-deterministic IP)**.** A search problem R is in *pseudo-deterministic* IP (often denoted psdIP) if there exists a function s mapping inputs to the search problem to solutions (i.e., all $x \in L_R$ satisfy $(x, s(x)) \in R$), and there is an interactive protocol between a probabilistic polynomial time verifier algorithm V and a prover (unbounded algorithm) P such that for every $x \in L_R$:

1. (Canonical Completeness) There exists a P such that $\Pr_r[(P, V)(x, r) = s(x)] \geq \frac{2}{3}$. (We use $(P, V)(x, r)$ to denote the output of the verifier V when interacting with prover P on input x using randomness r).

2. (Canonical Soundness) For all P', $\Pr_r[(P', V)(x, r) = s(x) \text{ or } \perp] \geq \frac{2}{3}$.

And (Standard Soundness) for every $x \notin L_R$, for all provers P', $\Pr_r[(P', V)(x, r) \neq \perp] \leq \frac{1}{3}$.

One can similarly define pseudo-deterministic MA, and pseudo-deterministic AM, where MA is a 1-round protocol, and AM is a 2-round protocol. One can show that any constant-round interactive protocol can be reduced to a 2-round interactive protocol [Bab85]. Hence, the definition of pseudo-deterministic AM captures the set of all search problems solvable in a constant number of rounds of interaction.

Historical Note: Historically, AM referred to public coin protocols, whereas IP referred to private coin protocols. In this work, we use AM to refer to constant round protocols, and IP to refer to polynomial round protocols (unless explicitly stated otherwise). By the result of [GS86], we know that when the prover is all-powerful, a private coin protocol can be simulated using private coins, so in this setting the distinction between private and public coins does not matter.

Definition 6.2.3 (Pseudo-deterministic AM)**.** A search problem R is in *pseudo-deterministic* AM (often denoted psdAM) if there exists a function s where all $x \in L_R$ satisfy $(x, s(x)) \in R$, a probabilistic polynomial time verifier algorithm V, and polynomials p and q, such that for every $x \in L_R$:

1. (Canonical Completeness) $\Pr_{r \in \{0,1\}^{p(n)}}(\exists z \in \{0,1\}^{q(n)}\, V(x, r, z) = s(x)) \geq \frac{2}{3}$
2. (Canonical Soundness) $\Pr_{r \in \{0,1\}^{p(n)}}(\forall z \in \{0,1\}^{q(n)}\, V(x, r, z) \in \{s(x), \perp\}) \geq \frac{2}{3}$.

And (Standard Soundness) for every $x \notin L_R$, we have $\Pr_{r \in \{0,1\}^{p(n)}}(\forall z \in \{0,1\}^{q(n)}\, V(x, r, z) = \{\perp\}) \geq \frac{2}{3}$.

Definition 6.2.4 (Pseudo-deterministic MA)**.** A search problem R is in *pseudo-deterministic* MA (often denoted psdMA) if there exists a function s where all $x \in L_R$ satisfy $(x, s(x)) \in R$ and $|s(x)| \leq poly(x)$, a probabilistic polynomial time verifier V such that for every $x \in L_R$[2]:

1. (Canonical Completeness) There exists a message M of polynomial size such that $\Pr_r[V(x, M, r) = s(x)] \geq \frac{2}{3}$.
2. (Canonical Soundness) For all M', $\Pr_r[V(x, M', r) = s(x) \text{ or } \perp] > \frac{2}{3}$.

And (Standard Soundness) for every $x \notin L_R$, for all M', $\Pr_r[V(x, M', r) \neq \perp] \leq \frac{1}{3}$.

[2]We remark that we use M to denote the proof sent by the prover Merlin, and not the algorithm implemented by the prover.

Pseudo-determinism can similarly be defined in the context of NP (which can be viewed as a specific case of an interactive proof):

Definition 6.2.5 (Pseudo-deterministic NP)**.** A search problem R is in *pseudo-deterministic* NP (often denoted psdNP) if there exists a function s where all $x \in L_R$ satisfy $(x, s(x)) \in R$ and $|s(x)| \leq poly(x)$, and there is a deterministic polynomial time verifier V such that for every $x \in L_R$:

1. There exists a message M of polynomial size such that $V(x, M) = s(x)$.
2. For all M', $V(x, M') = s(x)$ or $V(x, M') = \perp$.

And for every $x \notin L_R$, for all M', we have $V(x, M') = \perp$.

A similar definition for pseudo-deterministic NL follows naturally:

Definition 6.2.6 (Pseudo-deterministic NL)**.** A search problem R is in *pseudo-deterministic* NL (often denoted psdNL) if there exists a function s where all $x \in L_R$ satisfy $(x, s(x)) \in R$ and $|s(x)| \leq poly(x)$, there is a nondeterministic log-space machine V such that for every $x \in L_R$:

1. There exist nondeterministic choices N for the machine such that such that $V(x, N) = s(x)$.
2. For all possible nondeterministic choices N', $V(x, N') = s(x)$ or $V(x, N') = \perp$.

And for every $x \notin L_R$, for all nondeterministic choices N', $V(x, N') = \perp$.

6.3 Pseudo-deterministic-AM algorithm for graph isomorphism

In this section we give an algorithm for finding an isomorphism between two graphs in AM that outputs the same answer with high probability. The way this algorithm works is that the prover will send the lexicographically first isomorphism to the verifier and then prove that it is the lexicographically first isomorphism. To prove that the isomorphism is the lexicographically first isomorphism, we label the graph and run a sequence of graph non-isomorphism protocols to show no lexicographically smaller isomorphism exists. We present an alternate proof of the same result in the appendix (the proof in the appendix is more group theoretic, whereas the proof below is more combinatorial).

Theorem 6.3.1. Finding an isomorphism between graphs can be done in psdAM.

Proof. Let the vertices of G_1 be $v_1, v_2, \ldots, v_n$, and the vertices of G_2 be $u_1, u_2, \ldots, u_n$. We will show an AM algorithm which outputs a unique isomorphism ϕ. Our algorithm will proceed in n stages (which we will later show can be parallelized). After the kth stage, the values $\phi(v_1), \phi(v_2), \ldots, \phi(v_k)$ will be determined.

Suppose that the values $\phi(v_1), \phi(v_2), \ldots, \phi(v_k)$ have been determined. Then we will determine the smallest r such that there exists an isomorphism ϕ^* such that for $1 \leq i \leq k$, we have $\phi^*(v_i) = \phi(v_i)$, and in addition, $\phi^*(v_{k+1}) = u_r$. If we find r, we can set $\phi(v_{k+1}) = \phi^*(v_{k+1})$ and continue to the $k + 1^{th}$ stage.

To find the correct value of r, the (honest) prover will tell the verifier the value of r and ϕ. Then, to show that the prover is not lying, for each $r' < r$, the prover will prove that there exists no isomorphism ϕ' such that for $1 \leq i \leq k$, we have $\phi'(v_i) = \phi(v_i)$, and in addition, $\phi'(v_{k+1}) = u_{r'}$. To prove this, the verifier will pick G_1 or G_2, each with probability 1/2. If the verifier picked G_1, he will randomly shuffle the vertices $v_{k+2}, \ldots, v_n$, and send the shuffled graph to the prover. If the verifier picked G_2, he will set $u'_i = \phi(v_i)$ for $1 \leq i \leq k$, and $u'_{k+1} = u_{r'}$, and shuffle the rest of the vertices. If the prover can distinguish between whether the verifier initially picked G_1 or G_2, then that implies there is no isomorphism sending v_i to $\phi(v_i)$ for $1 \leq i \leq k$, and sending v_{k+1} to $u_{r'}$. The prover now can show this for all $r' \leq r$ (in parallel), as well as exhibit the isomorphism ϕ, thus proving that r is the minimum value such that there is an isomorphism sending v_i to $\phi(v_i)$ for $1 \leq i \leq k$, and sending v_{k+1} to u_r.

The above n stages can be done in parallel in order to achieve a constant round protocol. To do so, in the first stage, the prover sends the isomorphism ϕ to the verifier. Then, the verifier can test (in parallel) for each k whether under the assumption that $\phi(v_1), \phi(v_2), \ldots, \phi(v_k)$ are correct, $\phi(v_{k+1})$ is the lexicographically minimal vertex which v_{k+1} can be sent to. The correctness of this protocol follows from the fact that multiple AM interactive proofs can be performed in parallel while maintaining soundness and completeness for all of the interactive proofs performed (as shown in appendix C.1 of [Gol98]).

We note that in the above protocol, the prover only needs to have the power to solve graph isomorphism (and graph non-isomorphism). Also, we note that the above protocol uses private coins. While the protocol can be simulated with a public coin protocol [GS86], the simulation requires the prover to be very powerful. □

6.4 Lower bound on pseudo-deterministic AM algorithms

In this section, we establish that if any NP-complete problem has an AM protocol that outputs a unique witness with high probability, then the polynomial hierarchy collapses. To do this we show the analog of AM $\subseteq$ NP/*poly* for the pseudo-deterministic setting, and then use this fact to get a NP/*poly* algorithm with a unique witness. We can then use [HNOS96] to show that NP $\subseteq$ coNP/*poly*, which obtains the hierarchy collapse.

We begin by proving that psdAM $\subseteq$ psdNP/*poly*:

Lemma 6.4.1. *Suppose that there is a* psdAM *protocol for a search problem R, which on input $x \in L_R$, outputs $s(x)$. Then, the search problem R has a* psdNP/*poly algorithm which, on input x, outputs $s(x)$.*

Proof. Consider a psdAM protocol, and suppose that on input $x \in L_R$, it outputs $s(x)$.

Since we are guaranteed that when the verifier of the the psdAM accepts, it will output $s(x)$ with high probability, we can use standard amplification techniques to show that the verifier will output $s(x)$ with probability $1 - o(\exp(-n))$, assuming an honest prover, and will output anything other than $s(x)$ with probability $o(\exp(-n))$, even with a malicious prover. Then, by a union bound, there exists a choice of random string r that makes the verifier output $s(x)$ for all inputs $x \in \{0,1\}^n$ of size n with an honest prover, and that for malicious provers, the verifier will either reject or output $s(x)$. We encode this string r as the advice string for the NP/*poly* machine.

The NP/*poly* machine computing s can read r off the advice tape and then guess the prover's message, and whenever the verifier accepts, $s(x)$ will be output by that nondeterministic branch. Thus $s(x)$ can be computed by an NP/*poly* machine. □

Next, we show that if an NP-complete problem has a pseudo-deterministic-NP/*poly* algorithm, then the polynomial hierarchy collapses.

Theorem 6.4.2. Let $L \in$ NP be an NP-complete problem. Let R be a polynomial time algorithm such that there exists a polynomial p so that $x \in L$ if and only if $\exists y \in \{0,1\}^{p(|x|)} R(x,y)$. Suppose that there is a psdAM protocol that when given some $x \in L$, outputs a unique $s(x) \in \{0,1\}^{p(|x|)}$ such that $R(x, s(x)) = 1$. Then, NP $\subseteq$ coNP/*poly* and the polynomial hierarchy collapses to the third level.

Proof. Assume that there is a psdAM protocol that when given some $\phi \in L$, outputs a unique $s(\phi) \in \{0,1\}^{p(|\phi|)}$ such that $R(\phi, s(\phi)) = 1$. From Lemma 6.4.1, we have that there exists psdNP/*poly* algorithm A that given $\phi \in L$, outputs a unique witness $s(\phi)$ for ϕ. Given such an algorithm A, we can construct a function g computable in psdNP/*poly* that on two inputs ϕ_1 and ϕ_2, $g(\phi_1, \phi_2)$ is one of either ϕ_1 or ϕ_2 with the condition that if either ϕ_1 or ϕ_2 is in L, then $g(\phi_1, \phi_2)$ is satisfiable. If neither ϕ_1 nor ϕ_2 are in L, then $g(\phi_1, \phi_2) = \perp$.

To construct such a g, define a function g' where $g'(\phi_1, \phi_2) = \{\phi_1, \phi_2\} \cap L$ (i.e., $g'(\phi_1, \phi_2)$ is the subset of $\{\phi_1, \phi_2\}$ consisting of satisfiable formulas). We construct g by reducing the language $L' = \{(\phi_1, \phi_2) | g'(\phi_1, \phi_2) \neq \emptyset\}$ (which is in NP, and hence reducible to L, since L is NP-complete) to L and running A to find a unique witness for g, which we can then turn into a witness for L'. Note that a witness for L' is either a witness for ϕ_1 or for ϕ_2. We can then check whether this unique witness is a witness for ϕ_1 or ϕ_2, and output the ϕ_i for which it is a witness (in the case that the witness works for both of the ϕ_i, we output the lexicographically first ϕ_i).

We note that we view g as a function on the set $\{\phi_1, \phi_2\}$. That is, we set $g(\phi_1, \phi_2) = g(\phi_2, \phi_1)$ (if a function g does not satisfy this property, we can create a g^* satisfying this property by setting $g^*(\phi_1, \phi_2) = g(min(\phi_1, \phi_2), max(\phi_2, \phi_1)))$.

Now, our goal is to use g, which we know is computable in psdNP/*poly* to construct an NP/*poly* algorithm for $\bar{L}$ (the complement of L).

We construct the advice string for L for length n as follows. Our advice string will be a set S consisting of strings ϕ_i. Start out with $S = \emptyset$. We know that there exists a $\phi_1 \in \{0,1\}^n \cap L$ such that $g(\phi, \phi_1) = x$ for at least half of the set $\{\phi \in \{0,1\}^n \cap L | g(\phi, s) = s \forall s \in S\}$. Such an s exists since in expectation, when picking a random s, half of the ϕ's will satisfy $g(\phi, s) = x$. If we keep doing this, we get a set S with $|S| \leq poly(n)$ such that for every $\phi \in L$ of length n, there exists an $s \in S$ such that $g(\phi, s) = x$.

Now, to check that $\phi \in \bar{L}$ in NP/*poly* (where S as defined above is the advice), we compute $g(\phi, s)$ for every $s \in S$, and check that $g(\phi, s) = s$ for every $s \in S$ which is possible because $|S|$ is polynomial in n. It is clear that this algorithm accepts if $\phi \notin L$ and rejects if $\phi \in L$, so therefore $L \in$ coNP/*poly*, which implies that NP $\subseteq$ coNP/*poly*. Furthermore, NP $\subseteq$ coNP/*poly* implies that the polynomial hierarchy collapses to the third level. □

6.5 Pseudo-deterministic derandomization for BPP in subexponential time MA

In this section, we prove the existence of pseudo-deterministic subexponential time (time $O(2^{n^\epsilon})$ for every ϵ) MA protocols for problems in search-BPP for infinitely many input lengths.

In this section, we prove that every problem R in search-BPP has an MA proof where the verifier takes subexponential time (and the prover is unbounded). For completeness, we define search-BPP below:

Definition 6.5.1 (Search-BPP)**.** A binary relation R is in *search-BPP* if there exist probabilistic polynomial-time algorithms A, B such that

1. Given $x \in R_L$, A outputs a y such that with probability at least $2/3$, $(x, y) \in R$.
2. If y is output by A when run on x, and $(x, y) \notin R$, then B rejects on (x, y) with probability at least $2/3$. Furthermore, for all $x \in L_R$, with probability at least $1/2$ B accepts on (x, y) with probability at least $1/2$.

When $x \notin R_L$, A outputs $\perp$ with probability at least $2/3$.

The intuition of the above definition is that A is used to find an output y, and then B can be used to verify y, and amplify the success probability.

A main component of our proof will be the Nisan-Wigderson pseudo-random generator, which shows a way to construct pseudorandom strings given access to an oracle solving a problem of high circuit complexity.

To obtain the best running time for our pseudo-deterministic algorithm, we will need the iterated exponential functions first used in complexity theory by [MVW99]. We will be considering functions with half-exponential growth, i.e. functions f such that $f(f(n)) \in O(2^{n^k})$ for some k.

Definition 6.5.2 (Fractional exponentials [MVW99])**.** The fractional exponential function $e_\alpha(x)$ will be defined as $A^{-1}(A(x) + \alpha)$, where A is the solution to the functional equation $A(e^x - 1) = A(x) + 1$. In addition, we can construct such functions so that $e_\alpha(e_\beta(x)) = e_{\alpha+\beta}(x)$. It is clear from this definition that $e_1(n) = O(2^n)$, and that $e_{1/2}(e_{1/2}(x)) = O(2^n)$. We call a function f satisfying $f(x) = \Theta(e_{1/2}(x))$ a *half-exponential* function.

Definition 6.5.3 (Half-Exponential Time MA)**.** We define a *half-exponential time* MA *proof* to be an interactive MA proof in which the verifier runs in half-exponential time.

Theorem 6.5.4. Given a problem R in search-BPP, it is possible to obtain a pseudo-deterministic MA algorithm for R where the verifier takes subexponential time for infinitely many input lengths.

Proof. From [MVW99], we see that $\mathrm{MA} \cap \mathrm{coMA}$ where the verifier runs in half-exponential time cannot be approximated by polynomial-sized circuits. By Nisan-Wigderson [NW94], it follows that in half-exponential time MA, one can construct a pseudorandom generator with half-exponential stretch which is secure against any given polynomial-size circuit for infinitely many input lengths. We provide more details below.

Let T be the truth-table of a hard function in $\mathrm{MA} \cap \mathrm{coMA}$. Then, let R be a relation in search-BPP. Recall from Definition 6.5.1 that there is an algorithm A that given x, produces y such that $(x, y) \in R$ with high probability if $x \in R_L$.

We will now describe the MA protocol. First, the prover sends T to the verifier and proves that it is indeed the truth table of the hard function in half-exponential time MA (which can be done in half-exponential time). With T in hand, the verifier can then compute the output of the Nisan-Wigderson pseudorandom generator. The verifier loops through the seeds in lexicographic order and uses the pseudorandom generator on each seed to create pseudo-random strings, which the verifier then uses as the randomness for A. Each time, the verifier tests whether $(x, A(x, r)) \in R$ (which can be done in BPP, and hence also in MA) and returns the first such valid output.

This will output the same solution whenever the verifier both gets the correct truth-table for the PRG, and succeeds in testing for each PRG output whether the output it provides is valid. Both of these happen with high probability, and thus this is a pseudo-deterministic subexponential-time MA algorithm for any problem in search-BPP which succeeds on infinitely many input lengths. □

6.6 Uniqueness in NL

In this section, we prove that every problem in search-NL can be made pseudo-deterministic. For completeness we include a definition of search-NL:

Definition 6.6.1 (search-NL)**.** A search problem R is in *search*-NL if there is a nondeterministic log-space machine V such that for every $x \in L_R$,

1. There exist nondeterministic choices N for the machine such that such that $V(x, N) = y$, and $(x, y) \in R$.

2. For all possible nondeterministic choices N', $(x, V(x, N')) \in R$, or $V(x, N') = \perp$.

And for every $x \notin L_R$, for all nondeterministic choices N', $V(x, N') = \perp$.

Theorem 6.6.2 (Pseudo-determinism NL)**.** Every search problem in search-NL is in psdNL.

One can think of the complete search problem for NL as: given a directed graph G, and two vertices s and t such that there is a path from s to t, find a path from s to t. Note that the standard nondeterministic algorithm of simply guessing a path will result in different paths for different nondeterministic guesses. Our goal will be to find a unique path, so that on different nondeterministic choices, we will not end up with a path which is not the unique one.

The idea will be to find the lexicographically first shortest path (i.e, if the min-length path from s to t is of length d, we will output the lexicographically first path of length d from s to t). To do so, first we will determine the length d of the min-length path from s to t. Then, for each neighbor of s, we will check if it has a path of length $d - 1$ to t, and move to the first such neighbor. Now, we have reduced the problem to finding a unique path of length $d - 1$, which we can do recursively.

The full proof is given below:

Proof. Given a problem in search-NL, consider the set of all min-length computation histories. We will find the lexicographically first successful computation history in this set.

To do so, we first (nondeterministically) compute the length of the min-length computation history. This can be done because coNL = NL (so if the shortest computation history is of size T, one can show a history of size T. Also, because it is coNL to show that there is no history of size up to $T - 1$, we can show that there is no history of size less than T in NL).

In general, using the same technique, given a state S of the NL machine, we can tell what is the shortest possible length for a successful computation history starting at S.

Our algorithm will proceed as follows. Given a state S (which we initially set to be the initial configuration of the NL machine), we will compute T, the length of the shortest successful computation path starting at S. Then, for each possible nondeterministic choice, we will check (in NL) whether there exists a computation history of length $T - 1$ given that nondeterministic choice. Then, we will choose the lexicographically first such nondeterministic choice, and recurse.

This algorithm finds the lexicographically first computation path of minimal length which is unique. Hence, the algorithm will always output the same solution (or reject), so the algorithm is pseudo-deterministic. □

6.7 Structural Results

In [GGR13], Goldreich et al showed that the set of total search problems solved by pseudo-deterministic polynomial time randomized algorithms equals the set of total search problems solved by deterministic polynomial time algorithms, with access to an oracle to decision problems in BPP. In [GG15], this result was extended to the context of RNC. We show analogous theorems here. In the context of MA, we show that for total search problems, psdMA = search$-\mathrm{P}^{\mathrm{MA}\cap\mathrm{coMA}}$.[3] In other words, any pseudo-deterministic MA algorithm can be simulated by a polynomial time search algorithm with an oracle solving decision problems in MA $\cap$ coMA, and vice versa.

In the case of search problems that are not total, we show that psdMA equals to the class search$-\mathrm{P}^{\mathrm{promise}-(\mathrm{MA}\cap\mathrm{coMA})}$, where when the input x is in L_R, all queries to the oracle must be in the promise. We note that generally, when having an oracle to a promise problem, one is allowed to query the oracle on inputs not in the promise, as long as the output of the algorithm as a whole is correct for all possible answers the oracle gives to such queries. In our case, we simply do not allow queries to the oracle to be in the promise. Such reductions have been called *smart* reductions [GS88].

We show similar theorems for AM, and NP. Specifically, we show psdAM = search$-\mathrm{P}^{\mathrm{promise}-(\mathrm{AM}\cap\mathrm{coAM})}$ and psdNP = search$-\mathrm{P}^{\mathrm{promise}-(\mathrm{NP}\cap\mathrm{coNP})}$, where the reductions to the oracles are smart reductions.

In the case of total problems, one can use a similar technique to show psdAM = search$-\mathrm{P}^{\mathrm{AM}\cap\mathrm{coAM}}$ and psdNP = search$-\mathrm{P}^{\mathrm{NP}\cap\mathrm{coNP}}$, where the oracles can only return answers to total decision problems.

Theorem 6.7.1. The class psdMA equals the class search$-\mathrm{P}^{\mathrm{promise}-(\mathrm{MA}\cap\mathrm{coMA})}$, where on any input $x \in L_R$, the all queries to the oracle are in the promise.

Proof. The proof is similar to the proofs in [GGR13] and [GG15] which show similar reductions to decision problems in the context of pseudo-deterministic polynomial time algorithms and pseudo-deterministic NC algorithms.

[3] What we call search$-$P is often denoted as FP.

First, we show that a polynomial time algorithm with an oracle for promise$-(\mathrm{MA} \cap \mathrm{coMA})$ decision problems which only asks queries in the promise has a corresponding pseudo-deterministic MA algorithm. Consider a polynomial time algorithm A which uses an oracle for promise$-(\mathrm{MA} \cap \mathrm{coMA})$. We can simulate A by an MA protocol where the prover sends the verifier the proof for every question which A asks the oracle. Then, the verifier can simply run the algorithm from A, and whenever he accesses the oracle, he instead verifies the proof sent to him by the prover.

We note that the condition of a smart reduction is required in order for the prover to be able to send to the verifier the list of all queries A will make to the oracle. If A can ask the oracle queries not in the promise, it may be that on different executions of A, different queries will be made to the oracle (since A is a adaptive, and the queries A makes may depend on the answers returned by the oracle for queries not in the promise), so the prover is unable to predict what queries A will need answered.

We now show that a pseudo-deterministic MA algorithm B has a corresponding polynomial time algorithm A that uses a promise$-(\mathrm{MA} \cap \mathrm{coMA})$ oracle while only querying on inputs in the promise. On input $x \in L_R$, the polynomial time algorithm can ask the promise$-(\mathrm{MA} \cap \mathrm{coMA})$ oracle for the first bit of the unique answer given by B. This is a decision problem in promise$-(\mathrm{MA} \cap \mathrm{coMA})$ since it has a constant round interactive proof (namely, run B and then output the first bit). Similarly, the algorithm A can figure out every other bit of the unique answer, and then concatenate those bits to obtain the full output.

Note that it is required that the oracle is for promise$-(\mathrm{MA} \cap \mathrm{coMA})$, and not just for promise$-$MA, since if one of the bits of the output is 0, the verifier must be able to convince the prover of that (and this would require a promise$-$coMA protocol). □

A very similar proof shows the following:

Theorem 6.7.2. The class psdNP equals the class search$-\mathrm{P}^{\mathrm{promise}-(\mathrm{NP} \cap \mathrm{coNP})}$, where on any input $x \in L_R$, all queries to the oracle are in the promise.

We now prove a similar theorem for the case of AM protocols. We note that this is slightly more subtle, since it's not clear how to simulate a search$-\mathrm{P}^{\mathrm{promise}-(\mathrm{AM} \cap \mathrm{coAM})}$ protocol using only a constant number of rounds of interaction, since the search-P algorithm may ask polynomial many queries in an adaptive fashion.

Theorem 6.7.3. The class psdAM equals the class search$-\mathrm{P}^{\mathrm{promise}-(\mathrm{AM}\cap\mathrm{coAM})}$, where on any input $x \in L_R$, the all queries to the oracle are in the promise.

Proof. First, we show that a polynomial time algorithm with an oracle for promise$-(\mathrm{AM} \cap \mathrm{coAM})$ decision problems where the queries are all in the promise has a corresponding pseudo-deterministic AM algorithm. We proceed similarly to the proof of Theorem 6.7.1. Consider a polynomial time algorithm A which uses an oracle for promise$-(\mathrm{AM}\cap\mathrm{coAM})$. The prover will internally simulate that algorithm A, and then send to the verifier a list of all queries that A makes to the promise$-(\mathrm{AM} \cap$ coAM) oracle. Then, the prover can prove the answer (in parallel), to all of those queries.

To prove correctness, suppose that the prover lies about at least one of the oracle queries. Then, consider the first oracle query to which the prover lied. Then, by a standard simulation argument, one can show that it can be made overwhelmingly likely that the verifier will discover that the prover lied on that query.

Once all queries have been answered by the verifier the algorithm B can run like A, but instead of querying the oracle, it already knows the answer since the prover has proved it to him.

The proof that a pseudo-deterministic MA algorithm B has a corresponding polynomial time algorithm A that uses an promise$-(\mathrm{AM}\cap\mathrm{coAM})$ oracle is identical to the proof of Theorem 6.7.1 □

As a corollary of the above, we learn that private coins are no more powerful than public coins in the pseudo-deterministic setting:

Corollary 6.7.4. *A pseudo-deterministic constant round interactive proof using private coins can be simulated by a pseudo-deterministic constant round interactive proof using public coins.*

Proof. By Theorem 6.7.3, we can view the algorithm as an algorithm in search$-\mathrm{P}^{\mathrm{AM}\cap\mathrm{coAM}}$.

By a similar argument to that in Theorem 6.7.3, one can show that $\mathrm{psdIP} = \mathrm{search}-\mathrm{P}^{\mathrm{IP}\cap\mathrm{coIP}}$, where in this context IP refers to *constant* round interactive proofs using *private coins*, and AM refers to constant round interactive proofs using *public coins*. Since promise$-(\mathrm{AM} \cap \mathrm{coAM}) =$ promise$-(\mathrm{IP}\cap\mathrm{coIP})$, since every constant round *private coin* interactive proof for decision problems can be simulated by a constant round interactive proof using *public coins* [GS86], we have:

$$\mathrm{psdAM} = \mathrm{search}-\mathrm{P}^{\mathrm{promise}-(\mathrm{AM}\cap\mathrm{coAM})} = \mathrm{search}-\mathrm{P}^{\mathrm{promise}-(\mathrm{IP}\cap\mathrm{coIP})} = \mathrm{psdIP}.$$

□

6.8 Discussion and Open Problems

Pseudo-determinism and TFNP: The class of total search problems solvable by pseudo-deterministic NP algorithms is a very natural subset of TFNP, the set of all total NP search problems. It is interesting to understand how the set of total psdNP problems fits in TFNP. For example, it is not known whether TFNP = psdNP. It would be interesting either to show that every problem in TFNP has a pseudo-deterministic NP algorithm, or to show that under plausible assumptions there is a problem in TFNP which does not have a pseudo-deterministic NP algorithm.

Similarly, it is interesting to understand the relationship of psdNP to other subclasses of TFNP. For example, one can ask whether every problem in PPAD has a pseudo-deterministic NP algorithm (i.e., given a game, does there exists a pseudo-deterministic NP or AM algorithm which outputs a Nash Equilibrium), or whether under plausible assumptions this is not the case. Similar questions can be asked for CLS, PPP, and so on.

Pseudo-determinism in Lattice problems: There are several problems in the context of lattices which have NP (and often also NP $\cap$ coNP) algorithms [AR05]. Notable examples include gap-SVP and gap-CVP, for certain gap sizes. It would be interesting to show pseudo-deterministic interactive proofs for those problems. In other words, one could ask: does there exists an AM protocol for gap-SVP so that when a short vector exists, the *same* short vector is output every time. Perhaps more interesting would be to show, under plausible cryptographic assumptions, that certain such problems *do not* have psdAM protocols.

Pseudo-determinism and Number Theoretic Problems: The problem of generating primes (given a number n, output a prime greater than n), and the problem of finding primitive roots (given a prime p, find a primitive root mod p) have efficient randomized algorithms, and have been studied in the context of pseudo-determinism [Gro15, GG11, OS16], though no polynomial time pseudo-deterministic algorithms have been found. It is interesting to ask whether these problems have polynomial time psdAM protocols.

The Relationship between psdAM **and** search$-$BPP**:** One of the main open problems in pseudo-determinism is to determine whether every problem in search$-$BPP also has a polynomial time pseudo-deterministic algorithm. This remains unsolved. As a step in that direction (and as an interesting problem on its own), it is interesting to determine whether search$-$BPP $\subseteq$ psdAM. In this paper, we proved a partial result in this direction, namely that search$-$BPP $\subseteq$

$i.o.\text{psdMA}_{\text{SUBEXP}}$.

Zero Knowledge Proofs of Uniqueness: The definition of pseudo-deterministic interactive proofs can be extended to the context of Zero Knowledge. In other words, the verifier gets no information other than the answer, and knowing that it is the unique/canonical answer. It is interesting to examine this notion and understand its relationship to psdAM.

The Power of the Prover in pseudo-deterministic interactive proofs: Consider a search problem which can be solved in IP where the prover, instead of being all-powerful, is computationally limited. We know that such a problem can be solved in psdIP if the prover has unlimited computational power (in fact, one can show it is enough for the prover to be in PSPACE). In general, if the prover can be computationally limited for some IP protocol, can it also be computationally limited for a psdIP protocol for the same problem? It is also interesting in general to compare the power needed for the psdIP protocol compared to the power needed to solve the search problem non-pseudo-deterministically. Similar questions can be asked in the context of AM.

The Power of the Prover in pseudo-deterministic private vs public coins proofs: In our psdAM protocol for Graph Isomorphism, the verifier uses private coins, and the prover is weak (it can be simulated by a polynomial time machine with an oracle for graph isomorphism). If using public coins, what power would the prover need? In general, it is interesting to compare the power needed by the prover when using private coins vs public coins in psdAM and psdIP protocols.

Pseudo-deterministic interactive proofs for setting cryptographic global system parameters: Suppose an authority must come up with global parameters for a cryptographic protocol (for instance, a prime p and a primitive root g of p, which would be needed for a Diffie-Hellman key exchange). It may be important that other parties in the protocol know that the authority did not come up with these parameters because he happens to have a trapdoor to them. If the authority proves to the other parties that the parameters chosen are canonical, the other parties now know that the authority did not just pick these parameters because of a trapdoor (instead, the authority had to pick those parameters, since those are the canonical ones). It would be interesting to come up with a specific example of a protocol along with global parameters for which there is a pseudo-deterministic interactive proof showing the parameters are unique.

6.9 Alternate Algorithm for Graph Isomorphism in pseudo-deterministic AM

In this section, we present another psdAM algorithm for Graph Isomorphism, this one more group theoretic (as opposed to the more combinatorial approach of the algorithm in Section 6.3). The method we use to do this involves finding the lexicographically first isomorphism using group theory. In particular, the verifier will obtain the automorphism group of one of the graphs from the prover and verify that it is indeed the automorphism group, and then the verifier will convert an isomorphism obtained from the prover into the lexicographically first isomorphism between the two graphs. We will define the group-theoretic terms used below.

Definition 6.9.1 (Automorphism Group)**.** The *automorphism group* $Aut(G)$ of a graph is the set of permutations $\phi : G \to G$ such that for every $u, v \in V(G)$, $(u, v) \in E(G) \iff (\phi(u), \phi(v)) \in E(G)$ (i.e., ϕ is an automorphism of G).

Definition 6.9.2 (Stabilize)**.** Given a set S and elements $\alpha_1, \alpha_2, ..., \alpha_i \in S$, we say that a permutation $\phi : S \to S$ *stabilizes* $\{\alpha_1, \alpha_2, ..., \alpha_k\}$ iff $\phi(\alpha_i) = \alpha_i$ for $i \in \{1, ..., k\}$. We also say that a group G *stabilizes* $\{\alpha_1, \alpha_2, ..., \alpha_k\}$ when every $\phi \in G$ stabilizes $\{\alpha_1, \alpha_2, ..., \alpha_k\}$.

Definition 6.9.3 (Stabilizer)**.** The *stabilizer* of an element s in S for a group G acting on S is the set of elements of G that stabilize s.

Lemma 6.9.4. *Suppose that we are given a tuple (G_1, G_2, H, ϕ) where G_1 and G_2 are graphs, $H = Aut(G_1)$ is represented as a set of generators, and ϕ an isomorphism between G_1 and G_2. Then, in polynomial time, we can compute a unique isomorphism ϕ^* from G_1 to G_2 independent of the choice of ϕ and the representation of H.*

Proof. We use the algorithm given in [Can73] to compute a canonical coset representative, observing that the set of isomorphisms between G_1 and G_2 is a coset of the automorphism group of G_1. Let $\alpha_1, ..., \alpha_t$ be a basis of H, i.e., a set such that any $h \in H$ fixing $\alpha_1, ..., \alpha_t$ is the identity. Let H_i be the subgroup of H that stabilizes $\alpha_1, ..., \alpha_{i-1}$. Now, let U_i be a set of coset representatives of H_{i+1} in H_i. Given the generators of H_i, we can calculate U_i, and by Schreier's theorem we can calculate the generators for H_{i+1}. In this fashion, we can get generators and coset representatives for all the H_i. To produce ϕ^*, we do the following.

FIND-FIRST-ISOMORPHISM

1 $\phi^* = \phi$

2 For $i = 1, ..., t$

3 Let $P_i = \{\phi^* u | u \in U_i\}$.

4 Set $\phi^* = \arg\min_{\phi \in P_i}(\phi(\alpha_i))$.

To see that this produces a unique isomorphism that does not depend on ϕ, observe that $\phi^*(\alpha_1)$ is the minimum possible value of $\phi(\alpha_1)$ over all isomorphisms of G_1 to G_2 as U_1 is a set of coset representatives for the stabilizer of α_1 over H. Also, if $\phi^*(\alpha_i)$ is fixed for $i \in \{1, ..., k\}$, then $\phi^*(\alpha_{k+1})$ is the minimum possible value of $\phi(\alpha_{k+1})$ over all isomorphisms which take α_1 to $\phi^*(\alpha_1)$, α_2 to $\phi^*(\alpha_2)$,..., and α_k to $\phi^*(\alpha_k)$, as U_{i+1} stabilizes $\alpha_1, ..., \alpha_k$, so everything in P_{i+1} takes α_1 to $\phi^*(\alpha_1)$, α_2 to $\phi^*(\alpha_2)$,..., and α_k to $\phi^*(\alpha_k)$. This implies that ϕ^* does not depend on ϕ and is unique. □

Given this result, this means that it suffices to show a protocol that lets the verifier obtain a set of generators for the automorphism group of G_1 and an isomorphism that are correct with high probability, as by the above lemma this can be used to obtain a unique isomorphism between G_1 and G_2 independent of the isomorphism or the generators.

Theorem 6.9.5. There exists an interactive protocol for graph isomorphism such that with high probability, the isomorphism that is output by the verifier is unique, where in the case of a cheating prover the verifier fails instead of outputting a non-unique isomorphism. In other words, finding an isomorphism between graphs can be done in psdAM.

Proof. From Lemma 6.9.4, it suffices to show an interactive protocol that computes the automorphism group of a graph in a verifiable fashion. [Mat79] reduces the problem of computing the generators of the automorphism group to the problem of finding isomorphisms. Using this reduction, we can make a constant-round interactive protocol to determine the automorphism group by finding the isomorphisms in parallel. The reason we can do this in parallel is that [Mat79] implies that there are $O(n^4)$ different pairs of graphs to check and for each pair of graphs we either run the graph isomorphism protocol or the graph non-isomorphism protocol. In the case of the graph isomorphism protocol, the verifier need only accept with an isomorphism in hand; for graph non-isomorphism, the messages sent to the prover are indistinguishable between the two graphs when

they are isomorphic, so since the graphs and permutations are chosen independently, there is no way for the prover to correlate their answers to gain a higher acceptance probability for isomorphic graphs. Thus this means that the verifier can determine the automorphism group of a graph and verify that it is indeed the entire automorphism group. Using Lemma 6.9.4 we then see that the prover just has to give the verifier an isomorphism, and verifier can compute a unique isomorphism using the automorphism group. □

Bibliography

[AAHNY22] Ishaq Aden-Ali, Yanjun Han, Jelani Nelson, and Huacheng Yu. On the amortized complexity of approximate counting. *arXiv preprint arXiv:2211.03917*, 2022.

[AAK89] Alok Aggarwal, Richard J Anderson, and M-Y Kao. Parallel depth-first search in general directed graphs. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pages 297–308. ACM, 1989.

[AHL02] Noga Alon, Shlomo Hoory, and Nathan Linial. The moore bound for irregular graphs. *Graphs and Combinatorics*, 18(1):53–57, 2002.

[AHLW16] Yuqing Ai, Wei Hu, Yi Li, and David P. Woodruff. New characterizations in turnstile streams with applications. In *31st Conference on Computational Complexity, CCC 2016, May 29 to June 1, 2016, Tokyo, Japan*, pages 20:1–20:22, 2016.

[AJKS02] Miklós Ajtai, TS Jayram, Ravi Kumar, and D Sivakumar. Approximate counting of inversions in a data stream. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of Computing*, pages 370–379, 2002.

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

[AKO10] Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Streaming algorithms from precision sampling. *arXiv preprint arXiv:1011.1263*, 2010.

[AMS99] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and system sciences*, 58(1):137–147, 1999.

[AR05] Dorit Aharonov and Oded Regev. Lattice problems in NP ∩ coNP. *Journal of the ACM (JACM)*, 52(5):749–765, 2005.

[AV19] Nima Anari and Vijay V Vazirani. A pseudo-deterministic rnc algorithm for general graph perfect matching. *CoRR*, 2019.

[Bab85] László Babai. Trading group theory for randomness. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 421–429. ACM, 1985.

[BDW18] Arnab Bhattacharyya, Palash Dey, and David P Woodruff. An optimal algorithm for ℓ^1-heavy hitters in insertion streams and related problems. *ACM Transactions on Algorithms (TALG)*, 15(1):1–27, 2018.

[BKKS23] Vladimir Braverman, Robert Krauthgamer, Adithya Krishnan, and Shay Sapir. Lower bounds for pseudo-deterministic counting in a stream. *arXiv preprint arXiv:2303.16287*, 2023.

[BL83] László Babai and Eugene M Luks. Canonical labeling of graphs. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 171–183. ACM, 1983.

[Can73] John J Cannon. Construction of defining relators for finite groups. *Discrete Mathematics*, 5(2):105–129, 1973.

[CCFC04] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theoretical Computer Science*, 312(1):3–15, 2004.

[CCHO05] J. Cai, V. Chakaravarthy, L. Hemaspaandra, and M. Ogihara. Competing provers yield improved Karp–Lipton collapse results. *Information and Computation*, 198(1):1–23, 2005.

[CRS95] Suresh Chari, Pankaj Rohatgi, and Aravind Srinivasan. Randomness-optimal unique element isolation with applications to perfect matching and related problems. *SIAM Journal on Computing*, 24(5):1036–1050, 1995.

[Csü10] Miklós Csürös. Approximate counting with a floating-point counter. In *COCOON*, volume 6196, pages 358–367. Springer, 2010.

[Cve07] Andrej Cvetkovski. An algorithm for approximate counting using limited memory resources. *ACM SIGMETRICS Performance Evaluation Review*, 35(1):181–190, 2007.

[CW09] Kenneth L Clarkson and David P Woodruff. Numerical linear algebra in the streaming model. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 205–214. ACM, 2009.

[DKR10] Samir Datta, Raghav Kulkarni, and Sambuddha Roy. Deterministically isolating a perfect matching in bipartite planar graphs. *Theory of Computing Systems*, 47(3):737–757, 2010.

[Edm65] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17(3):449–467, 1965.

[FGT15] Stephen Fenner, Rohit Gurjar, and Thomas Thierauf. Bipartite perfect matching is in quasi-*nc*. *ECCC*, 9th November 2015. http://eccc.hpi-web.de/report/2015/177/.

[FIS08] Gereon Frahling, Piotr Indyk, and Christian Sohler. Sampling in dynamic data streams and applications. *International Journal of Computational Geometry & Applications*, 18(01n02):3–28, 2008.

[Fla85] Philippe Flajolet. Approximate counting: a detailed analysis. *BIT Numerical Mathematics*, 25(1):113–134, 1985.

[GG11] Eran Gat and Shafi Goldwasser. Probabilistic search algorithms with unique answers and their cryptographic applications. In *Electronic Colloquium on Computational Complexity (ECCC)*, volume 18, page 136, 2011.

[GG15] Shafi Goldwasser and Ofer Grossman. Perfect bipartite matching in pseudo-deterministic RNC. In *Electronic Colloquium on Computational Complexity (ECCC)*, volume 22, page 208, 2015.

[GG21] Sumanta Ghosh and Rohit Gurjar. Matroid intersection: A pseudo-deterministic parallel reduction from search to weighted-decision. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2021)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021.

[GGH17] Shafi Goldwasser, Ofer Grossman, and Dhiraj Holden. Pseudo-deterministic proofs. In *Electronic Colloquium on Computational Complexity (ECCC)*, volume 24, page 105, 2017.

[GGMW20] Shafi Goldwasser, Ofer Grossman, Sidhanth Mohanty, and David P Woodruff. Pseudo-deterministic streaming. In *11th Innovations in Theoretical Computer Science Conference (ITCS 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[GGR13] Oded Goldreich, Shafi Goldwasser, and Dana Ron. On the possibilities and limitations of pseudodeterministic algorithms. In *Proceedings of the 4th conference on Innovations in Theoretical Computer Science*, pages 127–138. ACM, 2013.

[GGS23] Ofer Grossman, Meghal Gupta, and Mark Sellke. Tight space lower bound for pseudo-deterministic approximate counting. *arXiv preprint arXiv:2304.01438*, 2023.

[GL19] Ofer Grossman and Yang P Liu. Reproducibility and pseudo-determinism in log-space. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 606–620. SIAM, 2019.

[GM20] Ofer Grossman and Dana Moshkovitz. Amplification and Derandomization without Slowdown. *SIAM Journal on Computing*, 49(5):959–998, 2020.

[GMW91] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in NP have zero-knowledge proof systems. *Journal of the ACM (JACM)*, 38(3):690–728, 1991.

[Gol98] Oded Goldreich. *Modern cryptography, probabilistic proofs and pseudorandomness*, volume 17. Springer Science & Business Media, 1998.

[Gol19] Oded Goldreich. Multi-pseudodeterministic algorithms. In *Electronic Colloquium on Computational Complexity (ECCC)*, 2019.

[GR09] Parikshit Gopalan and Jaikumar Radhakrishnan. Finding duplicates in a data stream. In *Proceedings of the twentieth annual ACM-SIAM symposium on Discrete algorithms*, pages 402–411. Society for Industrial and Applied Mathematics, 2009.

[Gro15] Ofer Grossman. Finding primitive roots pseudo-deterministically. In *Electronic Colloquium on Computational Complexity (ECCC)*, volume 22, page 207, 2015.

[GS86] Shafi Goldwasser and Michael Sipser. Private coins versus public coins in interactive proof systems. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 59–68. ACM, 1986.

[GS88] Joachim Grollmann and Alan L Selman. Complexity measures for public-key cryptosystems. *SIAM Journal on Computing*, 17(2):309–335, 1988.

[GS09] André Gronemeier and Martin Sauerhoff. Applying approximate counting for computing the frequency moments of long data streams. *Theory of Computing Systems*, 44:332–348, 2009.

[GW96] Anna Gál and Avi Wigderson. Boolean complexity classes vs. their arithmetic analogs. *Random Structures and Algorithms*, 9(1-2):99–111, 1996.

[HHM13] E. Hemaspaandra, L. Hemaspaandra, and C. Menton. Search versus decision for election manipulation problems. In *Proceedings of the 30th Annual Symposium on Theoretical Aspects of Computer Science*, pages 377–388. Leibniz International Proceedings in Informatics (LIPIcs), February/March 2013.

[HILL99] Johan Håstad, Russell Impagliazzo, Leonid A Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.

[HN17] L. Hemaspaandra and D. Narváez. The opacity of backbones. In *AAAI-2017*, pages 3900–3906. AAAI Press, February 2017.

[HNOS96] Lane A Hemaspaandra, Ashish V Naik, Mitsunori Ogihara, and Alan L Selman. Computing solutions uniquely collapses the polynomial hierarchy. *SIAM Journal on Computing*, 25(4):697–708, 1996.

[Hol17] Dhiraj Holden. A note on unconditional subexponential-time pseudo-deterministic algorithms for BPP search problems. *arXiv preprint arXiv:1707.05808*, 2017.

[HW13] Moritz Hardt and David P Woodruff. How robust are linear sketches to adaptive inputs? In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*, pages 121–130. ACM, 2013.

[Imm88] Neil Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on computing*, 17(5):935–938, 1988.

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

[IW05] Piotr Indyk and David Woodruff. Optimal approximations of the frequency moments of data streams. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 202–208. ACM, 2005.

[IZ89] Russell Impagliazzo and David Zuckerman. How to recycle random bits. In *FOCS*, volume 30, pages 248–253, 1989.

[JST11] Hossein Jowhari, Mert Sağlam, and Gábor Tardos. Tight bounds for lp samplers, finding duplicates in streams, and related problems. In *Proceedings of the thirtieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 49–58. ACM, 2011.

[JW18] Rajesh Jayaram and David P Woodruff. Perfect lp sampling in a data stream. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 544–555. IEEE, 2018.

[JW19] Rajesh Jayaram and David P Woodruff. Towards optimal moment estimation in streaming and distributed models. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

[KNP+17] Michael Kapralov, Jelani Nelson, Jakub Pachocki, Zhengyu Wang, David P Woodruff, and Mobin Yahyazadeh. Optimal lower bounds for universal relation, and for samplers and finding duplicates in streams. In *Foundations of Computer Science (FOCS), 2017 IEEE 58th Annual Symposium on*, pages 475–486. Ieee, 2017.

[KNW10] Daniel M Kane, Jelani Nelson, and David P Woodruff. On the exact space complexity of sketching and streaming small norms. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 1161–1178. SIAM, 2010.

[KUW85] Richard M Karp, Eli Upfal, and Avi Wigderson. Constructing a perfect matching is in random nc. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 22–32. ACM, 1985.

[LNW14] Yi Li, Huy L Nguyen, and David P Woodruff. Turnstile streaming algorithms might as well be linear sketches. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 174–183. ACM, 2014.

[Lov79] László Lovász. On determinants, matchings, and random algorithms. In *FCT*, volume 79, pages 565–574, 1979.

[Mat79] Rudolf Mathon. A note on the graph isomorphism counting problem. *Information Processing Letters*, 8(3):131–136, 1979.

[MG82] Jayadev Misra and David Gries. Finding repeated elements. *Science of computer programming*, 2(2):143–152, 1982.

[Mor78] Robert Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978.

[MVV87] Ketan Mulmuley, Umesh V Vazirani, and Vijay V Vazirani. Matching is as easy as matrix inversion. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 345–354. ACM, 1987.

[MVW99] Peter Bro Miltersen, N Variyam Vinodchandran, and Osamu Watanabe. Super-polynomial versus half-exponential circuit size in the exponential hierarchy. In *International Computing and Combinatorics Conference*, pages 210–220. Springer, 1999.

[MW10] Morteza Monemizadeh and David P Woodruff. 1-pass relative-error l p-sampling with applications. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 1143–1160. Society for Industrial and Applied Mathematics, 2010.

[Nai82] Mohan Nair. On chebyshev-type inequalities for primes. *American Mathematical Monthly*, pages 126–129, 1982.

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

[NNW14] Jelani Nelson, Huy L Nguyen, and David P Woodruff. On deterministic sketching and streaming for sparse recovery and norm estimation. *Linear Algebra and its Applications*, 441:152–167, 2014.

[NW94] Noam Nisan and Avi Wigderson. Hardness vs randomness. *Journal of computer and System Sciences*, 49(2):149–167, 1994.

[NY22] Jelani Nelson and Huacheng Yu. Optimal Bounds for Approximate Counting. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, 2022.

[OS16] Igor C Oliveira and Rahul Santhanam. Pseudodeterministic constructions in subexponential time. *arXiv preprint arXiv:1612.01817*, 2016.

[RA00] Klaus Reinhardt and Eric Allender. Making nondeterminism unambiguous. *SIAM Journal on Computing*, 29(4):1118–1131, 2000.

[Rei08] Omer Reingold. Undirected connectivity in log-space. *Journal of the ACM (JACM)*, 55(4):17, 2008.

[RRR16] Omer Reingold, Guy N Rothblum, and Ron D Rothblum. Constant-round interactive proofs for delegating computation. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing*, pages 49–62. ACM, 2016.

[RTV06] Omer Reingold, Luca Trevisan, and Salil Vadhan. Pseudorandom walks on regular digraphs and the RL vs. L problem. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, pages 457–466. ACM, 2006.

[Sch80] Jacob T Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *Journal of the ACM (JACM)*, 27(4):701–717, 1980.

[Sha92] Adi Shamir. IP=PSPACE. *Journal of the ACM (JACM)*, 39(4):869–877, 1992.

[SS91] Maria Serna and Paul Spirakis. Tight rnc approximations to max flow. In *STACS 91*, pages 118–126. Springer, 1991.

[SZ99] Michael Saks and Shiyu Zhou. $BPHSPACE(S) \subseteq DSPACE(S^{3/2})$. *Journal of Computer and System Sciences*, 58(2):376–403, 1999.

[Sze88] Róbert Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26(3):279–284, 1988.

[Tre01] Luca Trevisan. Extractors and pseudorandom generators. *Journal of the ACM*, 48(4):860–879, 2001.

[V+12] Salil P Vadhan et al. Pseudorandomness. *Foundations and Trends® in Theoretical Computer Science*, 7(1–3):1–336, 2012.

[XKNS21] Jingyi Xu, Sehoon Kim, Borivoje Nikolic, and Yakun Sophia Shao. Memory-efficient hardware performance counters with approximate-counting algorithms. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 226–228. IEEE, 2021.