Truly Low-Space Element Distinctness and Subset Sum via Pseudorandom Hash Functions*

Lijie Chen† MIT
Ce Jin‡ MIT
R. Ryan Williams§ MIT
Hongxun Wu ¶ Tsinghua University

Abstract

We consider low-space algorithms for the classic Element Distinctness problem: given an array of n input integers with $O(\log n)$ bit-length, decide whether or not all elements are pairwise distinct. Beame, Clifford, and Machmouchi [FOCS 2013] gave an $\tilde{O}(n^{1.5})$-time randomized algorithm for Element Distinctness using only $O(\log n)$ bits of working space. However, their algorithm assumes a random oracle (in particular, read-only random access to polynomially many random bits), and it was asked as an open question whether this assumption can be removed.

In this paper, we positively answer this question by giving an $\tilde{O}(n^{1.5})$-time randomized algorithm using $O(\log^3 n \log\log n)$ bits of space, *with one-way access to random bits*. As a corollary, we also obtain a $\mathrm{poly}(n)$-space $O^*(2^{0.86n})$-time randomized algorithm for the *Subset Sum* problem, removing the random oracles required in the algorithm of Bansal, Garg, Nederlof, and Vyas [STOC 2017].

The main technique underlying our results is a pseudorandom hash family based on iterative restrictions, which can fool the cycle-finding procedure in the algorithms of Beame et al. and Bansal et al.

1 Introduction

What problems can be solved simultaneously in low time and low space? When we restrict the space usage for solving a problem, how does this affect the possible running time of algorithms? The area of time-space tradeoffs has studied such questions for decades, beginning with Cobham [Cob66]. A central problem studied in time-space tradeoffs is Element Distinctness:

> Element Distinctness: Given an array of n positive integers $a_1, a_2, \ldots, a_n$ with $a_i \leq \mathrm{poly}(n)$ for all i, decide whether all a_i's are distinct.

The problem is extremely basic and useful: thinking of the array as describing a function from $[n]$ to $[\mathrm{poly}(n)]$, we are asking if the function is injective. The obvious algorithm that checks all pairs of elements takes $O(n^2)$ time and uses $O(\log n)$ bits of workspace. If we allow $\widetilde{O}(n)$ bits of workspace, Element Distinctness can be solved in near-linear time by sorting the input array. Applying low-space sorting algorithms directly [MP80, PR98], one can interpolate between these two algorithms and solve Element Distinctness in time $T(n)$ and space $S(n)$ for all $T(n), S(n)$ such that $T(n) \cdot S(n) \leq \widetilde{O}(n^2)$. For *comparison-based* algorithms, in which the only operation on elements allowed are pairwise comparisons, this time-space tradeoff was shown to be near-optimal in the 1980s [BFM+87, Yao88].

In 2013, Beame, Clifford, and Machmouchi [BCM13] surprisingly bypassed this longstanding lower bound, by giving a non-comparison-based algorithm for Element Distinctness with the time-space tradeoff $T(n) \leq \widetilde{O}(n^{3/2}/S(n)^{1/2})$. In particular, their algorithm can run in $\widetilde{O}(n^{1.5})$ time using only $O(\log n)$ bits of space. For brevity, we call this the *BCM algorithm*. A major disadvantage of the BCM algorithm is that it requires a *random oracle*: read-only random access to polynomially many uniform random bits (which do not count towards the

*Supported by NSF CCF-1909429 and NSF CCF-2127597. Lijie Chen is also supported by an IBM Fellowship. The full version of this paper [CJWW21] is at https://arxiv.org/abs/2111.01759.

†lijieche@mit.edu
‡cejin@mit.edu
§rrw@mit.edu
¶wuhx18@mails.tsinghua.edu.cn

space complexity). In the BCM algorithm, these random bits are used to specify the outgoing edges of a random 1-out digraph, on which Floyd's cycle-finding algorithm [Knu69] is performed to look for a pair of equal elements. Due to complicated dependencies on the paths in this random digraph, it looks difficult to reduce the number of random bits using pseudorandomness. It was asked as an open question [BCM13, BGNV18] whether the BCM algorithm can be modified to work with only "one-way access" to random bits, where we may toss up to $O(t)$ coins in time t, but cannot randomly access arbitrary coins tossed in the past. In particular, [BCM13] stated it "seems plausible" that the random oracle in the BCM algorithm could be replaced by some family of $\operatorname{poly}\log(n)$-wise independent hash functions in the analysis.

1.1 Our Results Our main result in this paper proves that one-way access to randomness is sufficient for implementing the BCM algorithm. We design a pseudorandom hash family with $O(\log^3 n \log\log n)$-bit seed length based on iterative restrictions of $O(\log n \log\log n)$-wise independent generators, and show that the analysis of the BCM algorithm still works when the random oracle is replaced by our pseudorandom generator. In fact, our proofs use a careful coupling-based analysis of an infinite tree generated from our pseudorandom generator. Hence we have the following result.

THEOREM 1.1. ELEMENT DISTINCTNESS *can be decided by a Monte Carlo algorithm in* $\widetilde{O}(n^{1.5})$ *time, with* $O(\log^3 n \log\log n)$ *bits of workspace and no random oracle. Moreover, when there is a colliding pair, the algorithm reports one.*

A closely related problem is the LIST DISJOINTNESS problem (which is equivalent to the 2-SUM problem).

> LIST DISJOINTNESS: Given two integer arrays $(a_1, a_2, \ldots, a_n)$ and $(b_1, b_2, \ldots, b_n)$ with entries in $[\operatorname{poly}(n)]$, decide whether there are $i, j \in [n]$ such that $a_i = b_j$.

This problem is harder than ELEMENT DISTINCTNESS, since the latter problem can be easily reduced to the former with only $O(\log n)$-factor overhead. The BCM algorithm for ELEMENT DISTINCTNESS does not straightforwardly extend to LIST DISJOINTNESS, and it is still open whether LIST DISJOINTNESS can be solved in $n^{o(1)}$-space and $n^{2-\Omega(1)}$ time, even allowing random oracles. Recently, Bansal, Garg, Nederlof, and Vyas [BGNV18] showed that a variant of the BCM algorithm can be applied to solve LIST DISJOINTNESS with an improved running time, provided that the input arrays have small second frequency moment (*i.e.*, there are few collision pairs within each arrays). Formally, define

$$F_2(a) = \sum_{i=1}^{n} \sum_{j=1}^{n} \mathbf{1}[a_i = a_j],$$

and assume an upper bound p on $F_2(a) + F_2(b)$ is known. Then their algorithm solves the LIST DISJOINTNESS problem in $\widetilde{O}(n\sqrt{p/s})$ time and $O(s \log n)$ space (with random oracle), for any $s \le n^2/p$. In this paper, we show that our pseudorandom family designed for the BCM algorithm also applies to this setting for $s = 1$.

THEOREM 1.2. *There is a Monte Carlo algorithm for* LIST DISJOINTNESS *such that, given input arrays* $a = (a_1, \ldots, a_n), b = (b_1, \ldots, b_n)$ *and an upper bound* $p \ge F_2(a) + F_2(b)$*, runs in* $\widetilde{O}(n\sqrt{p})$ *time and uses* $O(\log^3 n \log\log n)$ *bits of workspace and no random oracle.*

Combining the above LIST DISJOINTNESS algorithm with additive-combinatorial techniques, Bansal et al. gave a $\operatorname{poly}(n)$-space $O^*(2^{0.86n})$-time algorithm for *Subset Sum*: Given positive input integers $a_1, a_2 \ldots, a_n$ and a target integer t, find a subset of the input integers that sums to exactly t. They also solved the harder *Knapsack* problem with essentially the same time and space complexity. Replacing their LIST DISJOINTNESS subroutine with our Theorem 1.2, we immediately remove the assumption of random oracles in these algorithms as well.

THEOREM 1.3. (FOLLOWS FROM THEOREM 1.2 AND [BGNV18]) *Subset Sum and Knapsack can be solved by a Monte Carlo algorithm in* $O^*(2^{0.86n})$ *time, with* $O(\operatorname{poly}(n))$ *working space and no random oracle.*

In our ELEMENT DISTINCTNESS algorithm (Theorem 1.1), the 1.5 exponent in the time complexity seems hard to improve using current techniques. However, it is also difficult to prove a matching lower bound for such a decision problem. Hence we are motivated to look at a closely related multi-output problem for which our techniques still apply, and for which stronger time-space lower bounds are known. We consider the *Set Intersection* problem:

SET INTERSECTION: Given two integer sets A, B represented as two (not necessarily sorted) input arrays $(a_1, \ldots, a_n), (b_1, \ldots, b_n)$ which are promised to not contain duplicates, print all the elements in their intersection $A \cap B$.

Patt-Shamir and Peleg [PP93] showed that any $\operatorname{poly}\log(n)$-space algorithm for this problem must have time complexity $\widetilde{\Omega}(n^{1.5})$, even if the printed elements can be in any order, and each element in $A \cap B$ is allowed to be printed multiple times. (The recent work of Dinur [Din20] also implies the same lower bound.) We observe that our techniques imply a *nearly-matching time upper bound* for this problem, up to polylogarithmic factors.

THEOREM 1.4. (SET INTERSECTION) *There is a randomized algorithm that, given input arrays* $A = (a_1, \ldots, a_n), B = (b_1, \ldots, b_n)$ *where* A *and* B *are both YES instances of* ELEMENT DISTINCTNESS*, prints all elements in* $\{a_1, \ldots, a_n\} \cap \{b_1, \ldots, b_n\}$ *in* $\widetilde{O}(n^{1.5})$ *time, with* $O(\log^3 n \log\log n)$ *bits of workspace and no random oracle. The algorithm prints elements in no particular order, and the same element may be printed multiple times.*

1.2 Related Work In the following, we discuss several related works from various areas.

Element Distinctness and Collision Finding. In cryptography there has been intensive study on finding collisions in *random-like* functions using attacks based on the birthday paradox. Floyd's cycle-finding algorithm [Knu69, Pol75] has been used in memoryless birthday attacks [vOW99], which can be seen as low-space algorithms for ELEMENT DISTINCTNESS (or LIST DISJOINTNESS) with *random-like* input. In contrast, we consider worst-case input and do not rely on any heuristic assumptions.

Ambainis [Amb07] gave a quantum algorithm for ELEMENT DISTINCTNESS (as well as LIST DISJOINTNESS) with optimal $O(n^{2/3})$ query complexity [AS04]. The space complexity of Ambainis' algorithm is $\widetilde{O}(n^{2/3})$. In the $\operatorname{poly}\log(n)$-space setting, there are no known quantum algorithms that can significantly beat the simple $O(n)$-query algorithm obtainable from Grover Search [HM21].

Time-Space Tradeoff Lower Bounds. Borodin and Cook [BC82] proved nearly-optimal time-space tradeoff lower bounds for the sorting problem against (multi-way) branching programs. Their techniques were extended to prove time-space lower bounds for many other multi-output functions [Yes84, Abr87, Abr91, Bea91, MNT93, PP93]. Recently, McKay and Williams [MW19] generalized techniques of Beame [Bea91] to show quadratic time-space product lower bounds against branching programs armed with random oracles. However, these techniques cannot prove nontrivial time-space lower bounds for *decision* problems such as ELEMENT DISTINCTNESS. For decision problems, the current best known time-space lower bound states that SAT cannot be solved in $n^{1.801}$ time and $n^{o(1)}$ space ([Wil08, BW15], building on [FLvMV05]). For ELEMENT DISTINCTNESS, Ajtai [Ajt05] proved that for every $k \geq 1$, there exists an $\varepsilon > 0$ such that it cannot be solved by kn-time εn-space algorithms in the RAM model. Other time-space tradeoff lower bounds for decision problems are proved in [Kar86, Ajt02, BV02, BSSV03].

Random oracles. In the usual notion of randomized space-bounded computation, the outcomes of previous coin tosses cannot be recalled unless they are stored in working memory: this is typically called *one-way* access to randomness. The stronger model where all previous coin tosses can be recalled (*i.e.*, *two-way* access to randomness) has also been studied in the computational complexity literature. For example, Nisan [Nis93] showed that bounded two-sided error log-space machines with one-way access to randomness can be simulated by *zero-error* randomized log-space machines with two-way access to randomness ($\mathsf{BPL} \subseteq \mathsf{2wayZPL}$).

In the streaming literature, it is common to first design an streaming algorithm assuming access to a random oracle, then to use pseudorandom generators to remove this assumption, sometimes incurring a blowup in space complexity. Nisan's pseudorandom generator [Nis92] offers a generic way to derandomize many streaming algorithms (*e.g.*, [Ind06]). In our case, it is entirely unclear whether any off-the-shelf pseudorandom generators (such as [Nis92] or [FK18]) can be directly applied to replace the random oracle, since the queries made to the random oracle by the cycle detection algorithm are highly adaptive, dependent on the outcomes of previous queries.

(Pseudo-)random graphs. The ELEMENT DISTINCTNESS algorithm of Beame et al. [BCM13] (and related work) uses versions of the following basic fact about random mappings (a.k.a. random 1-out digraphs): starting from any vertex, the expected number of reachable vertices is $\Theta(\sqrt{n})$. The statistical properties (such as cycle lengths and component sizes) of random mappings have been extensively studied, see *e.g.*, [FK16, Chapter 16] and the references therein. However, most of these studies crucially assume the random graphs are generated

with full independence, and generally do not imply useful results about pseudorandomly generated graphs. One exception is the work of Alon and Nussboim [AN08] on k-wise independent Erdős-Rényi graphs, but it is very different from our setting of 1-out digraphs.

Subset Sum and Related Problems. The best known time complexity for Subset Sum is $O^*(2^{n/2})$ based on a meet-in-middle approach, first given by Horowitz and Sahni [HS74] in 1974. The space complexity of this algorithm was later improved from $O^*(2^{n/2})$ to $O^*(2^{n/4})$ by Schroeppel and Shamir [SS81]. Very recently, Nederlof and Węgrzycki gave an $O^*(2^{n/2})$-time $O^*(2^{0.249999n})$-space algorithm [NW21]. This algorithm (as well as the $O^*(2^{0.86n})$-time poly(n)-space algorithm [BGNV18]) used the techniques developed in [AKKN15, AKKN16], which were inspired by advances on *average-case* Subset Sum algorithms [HJ10].

The low-space List Disjointness algorithm of [BGNV18] also has implications for average-case k-Sum algorithms in low space [BGNV18, GLP18]. See also [Wan14, LVWW16].

There is also a long line of research on low-space *pseudopolynomial-time* algorithms (*i.e.*, with running time poly(n, t)) for Subset Sum [LN10, EJT10, Kan10, Bri17, JVW21], culminating in an $\widetilde{O}(nt)$-time $O(\log n \log\log n + \log t)$-space algorithm [JVW21].

1.3 Open Questions

We conclude by discussing several interesting questions left open by our work.

Time-space Tradeoffs? Beame et al. [BCM13] (and Bansal et al. [BGNV18]) not only gave efficient log-space algorithms for Element Distinctness (and List Disjointness), but also provided a smooth time-space trade-off interpolating between the log-space algorithms and the linear-space algorithms. These algorithms, when given S memory, perform the cycle-finding procedure from S starting vertices, and use a redirection idea (which requires S space to store the redirected edges) to nicely handle the collisions among all these S walks. Our analysis of the pseudorandom family only considers the case with a single starting vertex, corresponding to the poly $\log(n)$-space algorithm. It would be interesting to see whether the analysis can be generalized to the case of multiple starting vertices, and hence remove the random oracle assumption for these time-space trade-off algorithms as well.

Shorter Seed Length? Our algorithm needs $O(\log^3 n \log\log n)$ bits of space to store the "seed": the description of the pseudorandom mapping. An interesting question is whether we can reduce this seed length to $O(\log n)$. It seems plausible that our k-wise generators could be replaced by almost k-wise generators (*e.g.*, [AGHP90]) which have shorter seed length. However, to get $O(\log n)$ seed length, one might need to significantly modify our $O(\log n)$-level iterative restriction approach, which already incurs an $O(\log n)$ multiplicative factor.

Faster List Disjointness Algorithm? We reiterate the question raised by Bansal et al. [BGNV18]: can List Disjointness be decided in $n^{2-\Omega(1)}$ time and $n^{o(1)}$ space, even allowing random oracles? In hard instances for the current algorithms, there is only one "real" collision between the two arrays, but many "pseudo-collisions" coming from the same array, and it is not clear how to filter these pseudo-collisions without affecting the real collision. As to the question of whether List Disjointness does not have such an algorithm, the current lower bound techniques do not seem to distinguish between Element Distinctness and List Disjointness, and it is entirely unclear how to prove an $n^{1.5+\Omega(1)}$-time lower bound for $n^{o(1)}$-space algorithms solving such decision problems (for example, the best known time lower bound for Element Distinctness in the small-space setting is barely superlinear [Ajt05]).

1.4 Organization

In Section 2, we provide an overview of the intuitions behind the proof of Theorem 1.1. In Section 3 we give useful definitions and notations. In Section 4 we give the construction of our pseudorandom family, formally state the properties satisfied by the pseudorandom family, and show how to use them to obtain algorithms for Element Distinctness and List Disjointness. The properties stated in Section 4 are proved in the full version of this paper [CJWW21].

2 Overview of Techniques

Now we give an informal overview of the techniques behind the proof of Theorem 1.1.

Notation. Let $a = (a_1, \ldots, a_n) \in [m]^n$ be the input array to Element Distinctness. Throughout this overview, we will assume our instances are NO instances (note the YES case is simply the absence of a collision pair), and for simplicity we assume our NO instances have at most one collision pair $a_u = a_v$ where $u \neq v$. (It

turns out that the hardest NO instances are those with exactly one collision pair.) We will always use (u, v) to denote the unique collision pair in the NO instance that our algorithm needs to find.

Let $\mathcal{H}_{\mathsf{full}}$ be the collection of all functions from $[m]$ to $[n]$, and $h \in_{\mathsf{R}} \mathcal{H}_{\mathsf{full}}$ be a truly random function (implemented using a random oracle in the BCM algorithm). We define a 1-out digraph (*i.e.*, each node has at most one outgoing edge) $G_{a,h}$ on the vertex set $[n]$ with the edge set $\{(x, h(a_x)) \mid x \in [n]\} \subseteq [n] \times [n]$. For a collision pair (u, v), note that vertices $u, v \in [n]$ point to the same vertex $h(a_u) = h(a_v)$ since $a_u = a_v$. We also use $f^*_{a,h}(s)$ to denote the set of all vertices reachable from s in the digraph $G_{a,h}$.

In the following, we often use bold letters (*e.g.*, $\mathbf{x}$ and $\mathbf{y}$) to denote random variables.

2.1 Review of the BCM Algorithm It is instructive to first review the $O(\log n)$-space $\widetilde{O}(n^{1.5})$-time BCM algorithm for Element Distinctness, and understand why it requires a random oracle.

The BCM algorithm. The BCM algorithm first chooses a random vertex $s \in [n]$ and performs Floyd's cycle-finding algorithm on digraph $G_{a,h}$ starting from s. This will successfully detect u, v if both u and v are reachable from s, since u and v point to the same vertex. To bound the running time, the following two properties are established, using a birthday-paradox-style argument.

$$\mathop{\mathbf{E}}_{\boldsymbol{h} \in_{\mathsf{R}} \mathcal{H}_{\mathsf{full}}, \boldsymbol{s} \in_{\mathsf{R}} [n]} [|f^*_{a,\boldsymbol{h}}(\boldsymbol{s})|] \le O(\sqrt{n}), \tag{2.1}$$

$$\mathop{\mathbf{Pr}}_{\boldsymbol{h} \in_{\mathsf{R}} \mathcal{H}_{\mathsf{full}}, \boldsymbol{s} \in_{\mathsf{R}} [n]} [u, v \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})] \ge \Omega(1/n). \tag{2.2}$$

Condition (2.2) says the probability that both u and v are reachable from s is at least $\Omega(1/n)$. Thus, running $\widetilde{O}(n)$ independent trials of cycle detection (each using a different h) will lead to at least one trial with u and v reachable, with high probability. Condition (2.1) says we expect $O(\sqrt{n})$ vertices to be reachable from s. Together, these imply the running time can be bounded by $\widetilde{O}(n^{1.5})$. See Section 4 for a more formal description.

Why the BCM algorithm needs a high degree of independence. Let us see why the birthday argument mentioned above apparently needs the values of h to be fully independent (or close to that). For simplicity, we consider how one proves that the probability of reaching v from a random starting vertex $\boldsymbol{s}$ is $\Theta(1/\sqrt{n})$ (the probability of reaching both u and v can be analyzed similarly). Let $\boldsymbol{s}_0 = \boldsymbol{s}, \boldsymbol{s}_1, \boldsymbol{s}_2, \ldots$ be the vertices on the walk starting from $\boldsymbol{s}$. Conditioning on $\boldsymbol{s}_0 = s_0, \boldsymbol{s}_1 = s_1, \ldots, \boldsymbol{s}_k = s_k$, where $a_{s_0}, a_{s_1}, \ldots, a_{s_k}$ are distinct, the distribution of the next vertex $\boldsymbol{s}_{k+1}$ is uniform over $[n]$, due to the full independence of h. Once the elements are not distinct (a collision has occurred), the walk will follow the formed cycle (which is completely determined by the walk history) and no new vertices will be reached. From there, a standard birthday argument can be applied, yielding the desired $\Theta(1/\sqrt{n})$ probability bound of reaching v, and $\Theta(1/n)$ of reaching both u and v.

Note in the argument above we have to condition on *all* previous $(k+1)$ random choices, because determining the value of $\boldsymbol{s}_{k+1}$ involves the $(k+1)$ compositions of the h function. Since k is typically as large as $\sqrt{n}$, it appears that one needs at least $\Omega(\sqrt{n})$-wise independence of the values of h.

2.2 Overcoming the $\Omega(\sqrt{n})$-wise Independence Barrier We first show how to overcome the need of $\Omega(\sqrt{n})$-wise independence with a toy pseudorandom hash function family $\mathcal{H}_{\mathsf{toy}}$ based on a simple two-level iterative restriction. In particular, $\mathcal{H}_{\mathsf{toy}}$ is constructed from three $\widetilde{\Theta}(n^{1/4})$-wise independent hash functions, so that $\boldsymbol{h} \in_{\mathsf{R}} \mathcal{H}_{\mathsf{toy}}$ can be sampled using $\widetilde{\Theta}(n^{1/4})$ random bits.

Drawing a sample $\boldsymbol{h}$ from the toy pseudorandom hash function family $\mathcal{H}_{\mathsf{toy}}$

- Set a parameter $\tau = \Theta(n^{1/4} \log n)$, and independently draw two τ-wise independent uniform hash functions $\boldsymbol{r}_1, \boldsymbol{r}_2 \colon [m] \to [n]$. Independently draw a τ-wise independent hash function $\boldsymbol{g}_1 \colon [m] \to \{0, 1\}$ such that for every $x \in [m]$, $\boldsymbol{g}_1(x) = \begin{cases} 0 & \text{with probability } n^{-1/4}, \\ 1 & \text{otherwise.} \end{cases}$
- Finally, $\boldsymbol{h} \colon [m] \to [n]$ is defined by $\boldsymbol{h}(x) = \begin{cases} \boldsymbol{r}_1(x) & \text{when } \boldsymbol{g}_1(x) = 1, \\ \boldsymbol{r}_2(x) & \text{otherwise.} \end{cases}$

Now we instantiate the BCM algorithm with the hash function $\boldsymbol{h} \in_{\mathsf{R}} \mathcal{H}_{\mathsf{toy}}$. We can also view $f^*_{a,\boldsymbol{h}}(\boldsymbol{s})$ as the following random walk on the vertex set $[n]$.

The random walk corresponding to $f^*_{a,\boldsymbol{h}}(\boldsymbol{s})$ for $\boldsymbol{h} \in_{\mathsf{R}} \mathcal{H}_{\mathsf{toy}}$

$f^*_{a,\boldsymbol{h}}(\boldsymbol{s})$ contains the vertices on the following random walk:

- $\boldsymbol{w}_1 = \boldsymbol{s}$.
- For each integer $j \geq 2$, set $\boldsymbol{w}_j = \boldsymbol{h}(a_{\boldsymbol{w}_{j-1}})$ if there is no $k \in \{2, \ldots, j-1\}$ satisfying $a_{\boldsymbol{w}_{k-1}} = a_{\boldsymbol{w}_{j-1}}$; otherwise the walk is terminated.

Since $\boldsymbol{h} \in_{\mathsf{R}} \mathcal{H}_{\mathsf{toy}}$, the following is an equivalent view of the walk above, in terms of $\boldsymbol{g}_1, \boldsymbol{r}_1$ and $\boldsymbol{r}_2$:

- Initially, $\boldsymbol{s}_0 = \boldsymbol{s}$ and $\boldsymbol{w}$ is empty.
- For each integer $i \geq 1$:
  1. We start the i-th *subwalk* from $\boldsymbol{s}_{i-1}$ following the edges defined by $x \mapsto \boldsymbol{r}_1(a_x)$.
  2. Each time we visit a new vertex x (including $\boldsymbol{s}_{i-1}$), suppose there are already $j-1$ vertices in $\boldsymbol{w}$. We set $\boldsymbol{w}_j = x$ if there is no $k \in \{2, \ldots, j-1\}$ satisfying $a_{\boldsymbol{w}_{k-1}} = a_{\boldsymbol{w}_{j-1}}$; otherwise we terminate the whole walk.
  3. Then we check whether $\boldsymbol{g}_1(a_x) = 0$. If this happens (with probability $n^{-1/4}$), we stop this subwalk, and let $\boldsymbol{s}_i = \boldsymbol{r}_2(a_x)$. Namely, we follow the edge $x \mapsto \boldsymbol{r}_2(a_x)$ for one step. Then we move to Step (1) to continue with the $(i+1)$-th subwalk, starting from $\boldsymbol{s}_i$.

Roughly speaking, when $\boldsymbol{h} \in_{\mathsf{R}} \mathcal{H}_{\mathsf{toy}}$, the random walk generated above alternates between subwalks of typical length $O(n^{1/4})$ defined by $\boldsymbol{r}_1$, and single steps defined by $\boldsymbol{r}_2$. In the below, we provide some intuition about why such a random walk suffices for analyzing the BCM algorithm. For simplicity, we will make a unrealistic assumption, which we will mark by underlining it. Later, we will explain how to remove the assumption.

Intuition. We first argue that each subwalk has length less than $\tau/2$ with high probability. Fix an integer $i \geq 1$, and $\boldsymbol{s}_{i-1} = s_{i-1}$ for some $s_{i-1} \in [n]$. From s_{i-1}, suppose the subwalk has visited $t+1$ vertices $x_1, x_2, \ldots, x_{t+1}$ before termination. From the definition of our subwalk, we have $\boldsymbol{g}_1(a_{x_k}) = 1$ for every $k \in [t]$. Assuming the walk does not stop before $\boldsymbol{s}_i$, the elements $a_{x_1}, \ldots, a_{x_{t+1}}$ must be distinct. By the τ-wise independence of $\boldsymbol{g}_1$, such an event happens with probability at most $(1-n^{-1/4})^{-\min(t,\tau)}$. Applying a union bound over all possible $s_{i-1} \in [n]$, we can conclude that all subwalks have length at most $\tau/2$, with at probability at least

$$1 - n(1-n^{-1/4})^{-\tau/2} = 1 - n^{-\Theta(1)}.$$

From now on, we will condition on the event that all subwalks have length at most $\tau/2$.

In each subwalk, we follow the edges defined by $\boldsymbol{r}_1$ for at most $\tau/2$ steps. By the τ-wise independence of $\boldsymbol{r}_1$, each subwalk has the same distribution as a truly random walk with the same length, as long as its starting point $\boldsymbol{s}_{i-1}$ is independent of $\boldsymbol{r}_1$. However, we also note that different subwalks are *not* independent. Therefore, our analysis has to overcome the following two challenges:

(i) Remove the dependency of $\boldsymbol{s}_{i-1}$ on $\boldsymbol{r}_1$.

(ii) Handle correlations between subwalks.

First, we show how to handle challenge (i). If $\boldsymbol{s}_{i-1}$ is the random starting point $\boldsymbol{s}$, it is independent of $\boldsymbol{r}_1$. Otherwise, $\boldsymbol{s}_{i-1}$ is the vertex reached by a subwalk started from $\boldsymbol{s}_{i-2}$ (which depends on $\boldsymbol{r}_1$) together with a single step defined by $\boldsymbol{r}_2$. We wish to remove this dependency on $\boldsymbol{r}_1$ using the single step following $\boldsymbol{r}_2$.

The key observation is the following. A truly random walk has typical length $\Theta(\sqrt{n})$, while each subwalk has a typical length $\Theta(n^{1/4})$. So to mimic a truly random walk, our analysis only needs to handle $O(n^{1/4})$ queries to the hash function $\boldsymbol{r}_2$ (each query represents one step following $\boldsymbol{r}_2$). Assuming that the walk does not stop before $\boldsymbol{s}_i$ for all $i \in [n^{1/4}]$, these $O(n^{1/4})$ queries are distinct. Then by the τ-wise independence of $\boldsymbol{r}_2$ and the fact that $n^{1/4} \ll \tau$, each $\boldsymbol{s}_i$ can indeed be replaced by a truly uniformly random variable over $[n]$ without changing the distribution of the generated random walk. Therefore, $\boldsymbol{s}_{i-1}$ is independent of $\boldsymbol{r}_1$ and $\boldsymbol{g}_1$ as desired.

To handle challenge (ii) (*i.e.*, the correlation across subwalks), the key idea is that in a standard birthday paradox argument, we do not require complete independence of all items; in fact, pairwise independence already suffices. Since each subwalk has length at most $\tau/2$ and $\boldsymbol{r}_1$ is τ-wise independent, such subwalks are also pairwise independent, which enables us to perform a birthday-paradox-style analysis. Of course, this is an oversimplification, and our actual analysis framework will be clarified in Section 2.3.

Here we made the (unrealistic) assumption that the walk does not stop before reaching each $\boldsymbol{s}_i$. (In reality, the walk *has to* stop during some subwalk.) Note that whether the walk stops at the j-th step is equivalent to whether j is no greater than the length of the walk $|\boldsymbol{w}|$. Since $|\boldsymbol{w}|$ is a random variable depending on all of $\boldsymbol{r}_1, \boldsymbol{r}_2, \boldsymbol{g}_1$, we have to carefully ensure that our analysis does not involve $|\boldsymbol{w}|$, to keep $\boldsymbol{s}_{i-1}$ and $\boldsymbol{r}_1$ independent. We will explain how we overcome such difficulty in Section 2.3, and in Section 2.4 we will extend the two-level structure above into a $O(\log n)$-level tree (using signficantly less randomness in our hash functions).

2.3 An Alternative Analysis of the BCM Algorithm The starting point of our work is a coupling-based proof of Condition (2.2), based on what we call *extended* random walks.[1] This proof will introduce the key strategy of our later analysis, when we replace the random oracle by a pseudorandom hash function.

The random walk corresponding to $f^*_{a,\boldsymbol{h}}(\boldsymbol{s})$. Note for $\boldsymbol{h} \in_{\mathsf{R}} \mathcal{H}_{\mathsf{full}}$ and $\boldsymbol{s} \in_{\mathsf{R}} [n]$, $f^*_{a,\boldsymbol{h}}(\boldsymbol{s})$ can be seen as a random walk on the vertex set $[n]$ in a straightforward way.

The random walk $\boldsymbol{w}$ corresponding to $f^*_{a,\boldsymbol{h}}(\boldsymbol{s})$

- $\boldsymbol{w}_1 = \boldsymbol{s}$.
- For each integer $j \geq 2$, set $\boldsymbol{w}_j = \boldsymbol{h}(a_{\boldsymbol{w}_{j-1}})$ if there is no $2 \leq k \leq j-1$ such that $a_{\boldsymbol{w}_{j-1}} = a_{\boldsymbol{w}_{k-1}}$; otherwise stop the the walk.

Since the walk stops immediately after a collision occurs, one can see that $f^*_{a,\boldsymbol{h}}(\boldsymbol{s})$ is exactly the set of all vertices in the walk $\boldsymbol{w} = (\boldsymbol{w}_1, \ldots, \boldsymbol{w}_{|\boldsymbol{w}|})$.[2] Recall (u, v) is the unique collision pair. In order to prove Condition (2.2), our goal now is to lower bound the probability

$$\mathbf{Pr}[(\exists\ (i,j) \in [|\boldsymbol{w}|]^2)[\boldsymbol{w}_i = u \wedge \boldsymbol{w}_j = v]] \tag{2.3}$$

$$= \sum_{(i,j)\in\mathbb{N}^2} \mathbf{Pr}[(i \leq |\boldsymbol{w}| \wedge \boldsymbol{w}_i = u) \wedge (j \leq |\boldsymbol{w}| \wedge \boldsymbol{w}_j = v)]. \tag{2.4}$$

The equality of (2.3) and (2.4) holds since, by definition, if there is an (i, j) such that $(\boldsymbol{w}_i = u) \wedge (\boldsymbol{w}_j = v)$, then the walk would immediately stop at step $\max(i, j) + 1$ (*i.e.*, $|\boldsymbol{w}| = \max(i, j)$). So $\boldsymbol{w}$ contains at most one pair (i, j) such that $(\boldsymbol{w}_i, \boldsymbol{w}_j) = (u, v)$, and hence we can decompose (2.3) into (2.4).

Our initial hope is that (2.4) may be simpler to analyze, as it is a sum of many simpler terms, each of which only depends on two entries $\boldsymbol{w}_i$ and $\boldsymbol{w}_j$. However, the condition $(i \leq |\boldsymbol{w}| \wedge \boldsymbol{w}_i = u)$ is still difficult to analyze, as it depends on the length $|\boldsymbol{w}|$.

Coupling with the basic extended walk. To move forward, we wish to find a way to lower bound (2.3) by a sum of many simpler probabilities that do not involve $|\boldsymbol{w}|$. The first idea is to extend the random walk $\boldsymbol{w}$ to an *infinite* extended random walk $\bar{\boldsymbol{w}}$. We stress that the walk $\bar{\boldsymbol{w}}$ defined below is only used in the analysis, and not in the algorithm.

Basic extended walk $\bar{\boldsymbol{w}}$

- Extend the domain of $\boldsymbol{h}$ from $[m]$ to $[m] \cup \{\star_0, \star_1, \ldots\}$ as follows: for each $t \in \mathbb{N}$, sample $\boldsymbol{h}(\star_t) \in_{\mathsf{R}} [n]$, where all samples are independent.
- Perform the random walk $\boldsymbol{w}$. After $\boldsymbol{w}$ ends, set $\bar{\boldsymbol{w}} = \boldsymbol{w}$ and for every $t \in \mathbb{N}$ append $\boldsymbol{h}(\star_t)$ to the end of $\bar{\boldsymbol{w}}$.

[1] Condition (2.1) is easier to establish. We will focus on Condition (2.2) since it is more difficult.

[2] Note it is possible that for some $j \geq 2$, $a_{w_{j-1}}$ is distinct from all $a_{w_{k-1}}$ for $k < j$, but $\boldsymbol{h}(a_{w_{j-1}})$ has a collision with a previous $\boldsymbol{h}(a_{w_{k-1}})$. In this case, the walk moves to $\boldsymbol{w}_j = \boldsymbol{w}_k$ (which was already visited before) and stops at step $j + 1$.

Note that $\bar{\boldsymbol{w}}$ and $\boldsymbol{w}$ are both defined over the joint probability space $(\boldsymbol{h}, \boldsymbol{s})$ (for the extended $\boldsymbol{h}$), and $\boldsymbol{w}$ is always a prefix of $\bar{\boldsymbol{w}}$. From the definition of $\bar{\boldsymbol{w}}$, we have the following nice properties:

(2.5) • All entries of $\bar{\boldsymbol{w}}$ are i.i.d. samples from $[n]$.

(2.6) • For all i, if $a_{\bar{\boldsymbol{w}}_j} \neq a_{\bar{\boldsymbol{w}}_k}$ for all $1 \leq j < k < i$, then $\boldsymbol{w}_i = \bar{\boldsymbol{w}}_i$.

Proof strategy: subtracting the overcount. By (2.6), we know that $u, v \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})$ if there are $i, j \in \mathbb{N}$ such that (1) $\bar{\boldsymbol{w}}_i = u$ and $\bar{\boldsymbol{w}}_j = v$, and (2) for all $1 \leq t < q < \max(i, j)$, $a_{\bar{\boldsymbol{w}}_t} \neq a_{\bar{\boldsymbol{w}}_q}$. In this way, we have reformulated the success condition $u, v \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})$ as a statement that does **not** involve the length $|\boldsymbol{w}|$ of the original random walk $\boldsymbol{w}$, and can be analyzed more easily. Fixing a length parameter $L = c\sqrt{n}$ for some small constant $c > 0$ to be determined later, we have

$$\begin{aligned}\mathbf{Pr}[u, v \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})] &\geq \mathbf{Pr}[\exists i, j \in [L] \text{ s.t. } (\bar{\boldsymbol{w}}_i, \bar{\boldsymbol{w}}_j) = (u, v) \text{ and for all } 1 \leq t < q \leq L,\ a_{\bar{\boldsymbol{w}}_t} \neq a_{\bar{\boldsymbol{w}}_q}] \\ &= \sum_{i,j \in [L]} \mathbf{Pr}[(\bar{\boldsymbol{w}}_i, \bar{\boldsymbol{w}}_j) = (u, v) \text{ and for all } 1 \leq t < q \leq L,\ a_{\bar{\boldsymbol{w}}_t} \neq a_{\bar{\boldsymbol{w}}_q}]. \end{aligned} \tag{2.7}$$

The last equality above holds because if for all $1 \leq t < q \leq L$, we have $a_{\bar{\boldsymbol{w}}_t} \neq a_{\bar{\boldsymbol{w}}_q}$, then there can only be one pair $(i, j) \in [L]^2$ satisfying $(\bar{\boldsymbol{w}}_i, \bar{\boldsymbol{w}}_j) = (u, v)$.

To further lower bound (2.7), we define the following two quantities:

$$E_{\mathsf{total}} = \sum_{(i,j) \in [L]^2} \mathbf{Pr}[\bar{\boldsymbol{w}}_i = u \wedge \bar{\boldsymbol{w}}_j = v] \quad \text{and} \quad E_{\mathsf{bad}} = \sum_{\substack{(i,j) \in [L]^2 \\ 1 \leq t < q \leq L}} \mathbf{Pr}[\bar{\boldsymbol{w}}_i = u \wedge \bar{\boldsymbol{w}}_j = v \wedge a_{\bar{\boldsymbol{w}}_t} = a_{\bar{\boldsymbol{w}}_q}].$$

We claim that $\mathbf{Pr}[u, v \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})] \geq E_{\mathsf{total}} - E_{\mathsf{bad}}$: note that E_{total} counts the total expected number of pairs (i, j) with $(\bar{\boldsymbol{w}}_i, \bar{\boldsymbol{w}}_j) = (u, v)$, and $E_{\mathsf{total}} - E_{\mathsf{bad}}$ subtracts all the "bad pairs" from the total count.[3]

The rest of the analysis is a straightforward calculation using the property (2.5). We can see that $E_{\mathsf{total}} = \Theta(L^2/n^2) = \Theta(c^2/n)$, and $E_{\mathsf{bad}} = \Theta(L^4/n^3) = \Theta(c^4/n)$. Setting c to be small enough, we have $E_{\mathsf{total}} - E_{\mathsf{bad}} \geq \Omega(1/n)$, which concludes the proof.

Remark. Setting

$$E_{\mathsf{total}} = \sum_{i \in [L]} \mathbf{Pr}[\bar{\boldsymbol{w}}_i = u] \quad \text{and} \quad E_{\mathsf{bad}} = \sum_{\substack{j \in [L] \\ 1 \leq t < q \leq L}} \mathbf{Pr}[\bar{\boldsymbol{w}}_i = u \wedge a_{\bar{\boldsymbol{w}}_{t-1}} = a_{\bar{\boldsymbol{w}}_{q-1}}], \tag{2.8}$$

one can also show $\mathbf{Pr}[v \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})] \geq \Omega(1/\sqrt{n})$ for all possible vertices v, by showing $E_{\mathsf{total}} - E_{\mathsf{bad}} \geq \Omega(1/\sqrt{n})$ (for $L = c\sqrt{n}$ and appropriately small $c > 0$). Later in this overview, we will explain how to get an $\Omega(1/\sqrt{n})$ lower bound for this single-vertex case when we replace the random oracle h by a pseudorandom function, and discuss additional challenges that arise for the two-vertex case (with u, v).

2.4 Pseudorandom Hash Functions, the Dependency Tree, and the Indexing Scheme Next we describe our construction of pseudorandom hash functions $\boldsymbol{h}$ based on *iterative restrictions*. In particular, we use a small number of independent partial functions defined by random restrictions to form a full hash function. By considering how the hash values of vertices on the random walk are determined by the iterative restriction, we can naturally organize these vertices into a hierarchical structure we call the *dependency tree*, which will play an crucial role in our later analysis.

Pseudorandom hashing by iterative restrictions. Instead of using full randomness, we will implement the hash function $h\colon [m] \to [n]$ by the following iterative pseudorandom restriction process, using only $\operatorname{poly}\log(n)$ seed length. Initially, all values of $h(x)$ are undefined. The values are defined over $\ell \leq \log n$ iterations. In

[3] We call such an (i, j) a "bad pair" because it should not be counted in (2.7), and has to be subtracted from the total count. Also, we remark that is possible that a bad pair is subtracted more than once in E_{bad}. This is not an issue for us, as we are trying to lower bound $\mathbf{Pr}[u, v \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})]$.

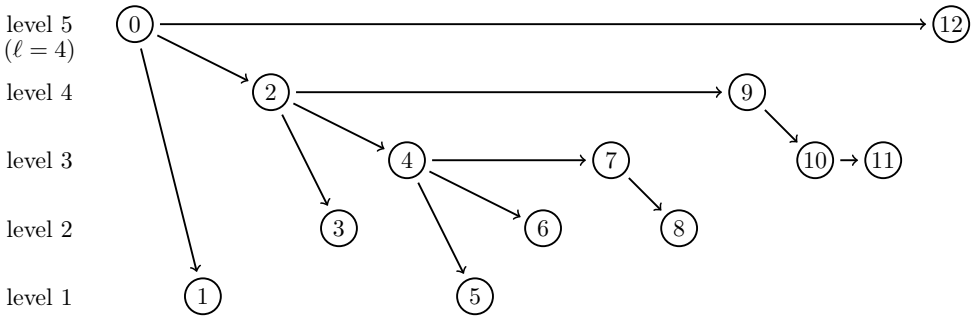

Figure 1: An example of a dependency tree $\boldsymbol{T}$. For example, the index of 7 is $(0, 0, 2, 1)$, since the path $0 \leftarrow 2 \leftarrow 4 \leftarrow 7$ has two level-3 nodes (node 4 and node 7), and one level-4 node (node 2).

the i-th iteration, we sample $O(\log n \log\log n)$-wise random functions $g_i \colon [m] \to \{0,1\}, r_i \colon [m] \to [n]$, and for every $x \in [m]$ such that $g_i(x) = 1$ and $h(x)$ is still undefined, we define $h(x)$ to be $r_i(x)$. See Section 4.1 for details. Informally, in each iteration we independently use $O(\log n \log\log n)$-wise generators to fix about half of the remaining undefined values in h: the g_i selects which half, and the r_i selects the values. (It is possible that a tiny number of hash values $h(x)$ may still be undefined after $\log(n)$ iterations, but this is not a significant issue for us and we ignore it in this overview.)

Let $\mathcal{H}$ denote the above family of pseudorandom functions. In the following, $\boldsymbol{h}$ will denote the random variable for a function randomly drawn from $\mathcal{H}$. Analogously to Section 2.3, one can define a random walk $\boldsymbol{w}$ on the random graph $G_{a,\boldsymbol{h}}$.

Tree structure of pseudorandom walks. We now describe a *dependency tree* $\boldsymbol{T}$ for a walk $\boldsymbol{w}$ on $G_{a,\boldsymbol{h}}$. We use non-negative integers to denote the nodes of $\boldsymbol{T}$: node 0 is a "dummy" node representing the root, and for $\mu \geq 1$, node μ corresponds to the μ-th node of walk $\boldsymbol{w}$ if it exists (*i.e.*, node μ is associated with vertex $\boldsymbol{w}_\mu$). We will use Greek letters $\alpha, \beta, \mu, \ldots$ to refer to nodes in the dependency tree $\boldsymbol{T}$.

The tree $\boldsymbol{T}$ has one "level" for each iteration $1, \ldots, \ell$ of the process defining $\boldsymbol{h}$. For each node μ of $\boldsymbol{T}$, we define $\mathsf{level}(\mu)$ (the "level of μ") to be the smallest integer j such that $g_j(a_{\boldsymbol{w}_\mu}) = 1$ (note that this j corresponds to the iteration in which the hash value of $a_{\boldsymbol{w}_\mu}$ is defined). If no such j exists, then we set $\mathsf{level}(\mu) = \ell + 1$. We also set $\mathsf{level}(0) = \ell + 1$, and define $\mathsf{next}(\mu) = \boldsymbol{h}(a_{\boldsymbol{w}_\mu}) = r_{\mathsf{level}(\mu)}(a_{\boldsymbol{w}_\mu})$. Informally, $\mathsf{next}(\mu)$ corresponds to the "next" vertex on the walk after $\boldsymbol{w}_\mu$.

Dependency tree $\boldsymbol{T}$ based on $\boldsymbol{w}$

- Node 0 is the root of $\boldsymbol{T}$.
- For each node μ of $\boldsymbol{T}$, its parent $\mathsf{par}(\mu)$ is defined as the largest node $\nu < \mu$ with level at least $\mathsf{level}(\mu)$.

Observe that the walk $\boldsymbol{w}$ is simply the pre-order traversal of $\boldsymbol{T}$. Also, observe that every root-to-node path of $\boldsymbol{T}$ has non-increasing node levels.

Indexing a tree node. Recall $\ell \leq \log n$ is the number of iterations, which bounds the number of levels of $\boldsymbol{T}$. Each node x of $\boldsymbol{T}$ can be assigned a unique "index" in a natural way, via a sequence $\vec{k} = (k_1, k_2, \ldots, k_\ell)$ of non-negative integers, where k_i specifies the number of level-i nodes on the path from the root to the node x. See Figure 1 for an illustration of a tree and the index scheme. We will explain why such indexing scheme helps our analysis at the end of the next subsection.

2.5 A Coupling-based Approach Based on the Dependency Tree We wish to mimic the strategy of the coupling-based proof in Section 2.3. Instead of proving an $\Omega(1/n)$ lower bound for $\mathbf{Pr}[u, v \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})]$, we will first consider how to prove an $\Omega(1/\sqrt{n})$ lower bound for $\mathbf{Pr}[u \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})]$, which already contains all the important ideas. Then, we will briefly discuss additional technical challenges that arise for the analysis of the two-vertex case (computing $\mathbf{Pr}[u, v \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})]$).

As in Section 2.3, our strategy is again to carefully design an extended random walk $\bar{\boldsymbol{w}}$ which is coupled with

$\boldsymbol{w}$, so that $\boldsymbol{w}$ is always a prefix of $\bar{\boldsymbol{w}}$. We will also build a corresponding extended dependency tree ("extended tree" for short) $\bar{\boldsymbol{T}}$ on $\bar{\boldsymbol{w}}$. Note that $\boldsymbol{T}$ would be a subtree of $\bar{\boldsymbol{T}}$ as $\boldsymbol{w}$ is a prefix of $\bar{\boldsymbol{w}}$. We will similarly define next and level values for nodes on extended tree $\bar{\boldsymbol{T}}$, and these values would be consistent with $\boldsymbol{T}$ on the corresponding subtree. We will sometimes use $\mathsf{next}_{\boldsymbol{T}}$ or $\mathsf{next}_{\bar{\boldsymbol{T}}}$ when there is a chance of confusion on which tree next is referring to.

We hope to define an extended walk $\bar{\boldsymbol{w}}$ that maintains Condition (2.6) as before. For notational convenience, we slightly change Condition (2.6) to

$$\text{For all } i, \text{ if } a_{\mathsf{next}_{\bar{\boldsymbol{T}}}(\alpha)} \neq a_{\mathsf{next}_{\bar{\boldsymbol{T}}}(\beta)} \text{ for all } 0 \leq \alpha < \beta < i-1, \text{ then } \boldsymbol{w}_i = \bar{\boldsymbol{w}}_i. \tag{2.9}$$

Note that since $\mathsf{next}_{\bar{\boldsymbol{T}}}(\alpha) = h(a_{\bar{\boldsymbol{w}}_\alpha}) = \bar{\boldsymbol{w}}_{\alpha+1}$, the above is equivalent to (2.6).

For an index $\vec{k} \in \mathbb{N}^\ell$, we also let $\mu^{\vec{k}}$ denote the node indexed by $\vec{k}$ in the dependency tree $\bar{\boldsymbol{T}}$. Note that such a node may not exist in the tree; we use $\mathcal{F}^{\vec{k}}$ to denote the event that $\mu^{\vec{k}}$ exists in $\bar{\boldsymbol{T}}$. To lower bound $\mathbf{Pr}[u \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})]$, we define the following two quantities analogous to (2.8):

$$E_{\mathsf{total}} = \sum_{\vec{k} \in \mathbb{N}^\ell} \mathbf{Pr}\left[\mathcal{F}^{\vec{k}} \wedge \mathsf{next}(\mu^{\vec{k}}) = u\right], \tag{2.10}$$

and

$$E_{\mathsf{bad}} = \sum_{\substack{\vec{k} \in \mathbb{N}^\ell \\ \vec{k}^1 < \vec{k}^2 \in \mathbb{N}^\ell}} \mathbf{Pr}[\mathcal{F}^{\vec{k}} \wedge \mathsf{next}(\mu^{\vec{k}}) = u \wedge \mathcal{F}^{\vec{k}^1} \wedge \mathcal{F}^{\vec{k}^2} \wedge a_{\mathsf{next}(\mu^{\vec{k}^1})} = a_{\mathsf{next}(\mu^{\vec{k}^2})}]. \tag{2.11}$$

Note that our choice of E_{bad} in (2.11) is a bit different from that in Section 2.3, as we consider a "bad occurrence" to happen whenever there is a collision in $\bar{\boldsymbol{w}}$ (while in (2.8) we restricted t, q to the interval $[\ell]$). This will not be a problem if we choose ℓ carefully.

By an argument similar to that of Section 2.3, we have that $\mathbf{Pr}[u \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})] \geq E_{\mathsf{total}} - E_{\mathsf{bad}}$. Hence, the goal is to design $\bar{\boldsymbol{w}}$ and $\bar{\boldsymbol{T}}$ such that (2.9) holds and the summands in E_{total} and E_{bad} can be bounded.

Quick estimate: a sanity check. To better understand the summands in E_{total} and E_{bad}, let us first calculate these summands under the unrealistic assumption that all involved events are independent. Note that $\mathcal{F}^{\vec{k}}$ asserts the existence of node $\mu^{\vec{k}}$ in the tree $\bar{\boldsymbol{T}}$, which requires that there is a tree path starting from the root, and extending down the levels in a way that is consistent with the vector $\vec{k}$, which specifies the number of level-i nodes on this path for every $i \in [\ell]$. Observe that, for every node β of level i on this path, we must have $g_i(a_{\boldsymbol{w}_\beta}) = 1$, since otherwise β would not have been on level i, and the path would not extend to reach β. Hence, the event $(\mathcal{F}^{\vec{k}} \wedge \mathsf{next}(\mu^{\vec{k}}) = u)$ is equivalent to the conjunction of the two conditions:

(1) Let $\alpha = \mu^{\vec{k}}$. For all the k_i level-i nodes β on the path from root to node α, we have $g_i(a_{\boldsymbol{w}_\beta}) = 1$, and
(2) $r_{\mathsf{level}(\alpha)}(a_{\boldsymbol{w}_\alpha}) = u$,

where Item (2) directly follows from our definition of $\mathsf{next}(\cdot)$. Observe that the event in Item (2) happens with $1/n$ probability, and for each β the event in Item (1) happens with $1/2$ probability. Pretending that all these events are independent, we would have

$$\mathbf{Pr}\left[\mathcal{F}^{\vec{k}} \wedge \mathsf{next}(\mu^{\vec{k}}) = u\right] = \left(2^{|\vec{k}|_1} \cdot n\right)^{-1}, \tag{2.12}$$

where $|\vec{k}|_1$ is the ℓ_1-norm of $\vec{k}$. Similarly, pretending all events are independent, we would have

$$\mathbf{Pr}\left[\mathcal{F}^{\vec{k}} \wedge \mathsf{next}(\mu^{\vec{k}}) = u \wedge \mathcal{F}^{\vec{k}^1} \wedge \mathcal{F}^{\vec{k}^2} \wedge a_{\mathsf{next}(\mu^{\vec{k}^1})} = a_{\mathsf{next}(\mu^{\vec{k}^2})}\right] = \left(2^{|\vec{k}|_1 + |\vec{k}^1|_1 + |\vec{k}^2|_1} \cdot n^2\right)^{-1}. \tag{2.13}$$

Observe that $\sum_{\vec{k} \in \mathbb{N}^\ell} 2^{-|\vec{k}|_1} = \sum_{\vec{k} \in \mathbb{N}^\ell} 2^{-k_1} \cdot 2^{-k_2} \cdot \dots \cdot 2^{-k_\ell} = (\sum_{i \in \mathbb{N}} 2^{-i})^\ell = 2^\ell$. Then, plugging (2.12) and (2.13) into (2.10) and (2.11), we would have $E_{\mathsf{total}} = \Omega(2^\ell/n)$, and $E_{\mathsf{bad}} = O(2^{3\ell}/n^2)$. Setting $\ell = \frac{1}{2} \cdot \log(n) - c$ for a large enough constant c, we would have

$$E_{\mathsf{total}} - E_{\mathsf{bad}} = \Omega\left(\frac{1}{2^c\sqrt{n}}\right) - O\left(\frac{1}{2^{3c}\sqrt{n}}\right) \geq \Omega(1/\sqrt{n}). \tag{2.14}$$

Now we can explain why we chose such an indexing scheme: the existence of $\mu^{\vec{k}}$ and the value of $\mathsf{next}(\mu^{\vec{k}})$ only depends on the ancestors of $\mu^{\vec{k}}$ in the dependency tree. Since typically there are at most $\operatorname{poly}\log(n)$ many ancestors, we can use the τ-wise independence of $\boldsymbol{g}_i$ and $\boldsymbol{r}_i$ to analyze the event $\mathcal{F}^{\vec{k}} \wedge \mathsf{next}(\mu^{\vec{k}}) = u$.

2.6 Designing the Extended Random Walk Finally we explain how to design the extended random walk $\bar{\boldsymbol{w}}$, by constructing an extended tree $\bar{\boldsymbol{T}}$. We first aim to ensure Condition (2.12) holds, leading to a desired lower bound on E_{total}. Handling E_{bad} is more challenging; we will discuss that later.

Specifically, we will ensure that (2.12) holds for all "short" vectors $\vec{k} \in [\tau/4]^{\ell}$ and $u \in [n]$, where $\tau = O(\log n \log\log n)$ is the independence parameter of our pseudorandom hash function.[4]

Establishing (2.12) by induction. To show (2.12), we wish to prove the following claim.

CLAIM 2.1. *Fix an index $\vec{k}$ corresponding to a level-i node ($\vec{k} = (0, \ldots, k_i, k_{i+1}, \ldots, k_\ell)$ and $k_i > 0$). Conditioned on the event $\mathcal{F}^{\vec{k}}$, with $1/2$ probability $\mu^{\vec{k}}$ has a level-i child ν (i.e., for $\vec{k}' = (0, \ldots, k_i + 1, k_{i+1}, \ldots, k_\ell)$, $\mathcal{F}^{\vec{k}'}$ holds) and $\mathsf{next}(\nu)$ is distributed uniformly in $[n]$.*

Assuming that Claim 2.1 holds, then (2.12) follows by a simple induction.[5] However, it is not hard to see that Claim 2.1 does not hold for the original tree $\boldsymbol{T}$. To understand the issue, let $\vec{k}, \vec{k}'$ be as in Claim 2.1 and assume $\mu^{\vec{k}}$ exists (*i.e.*, $\mathcal{F}^{\vec{k}}$ holds). We wish to better understand the conditions under which $\mu^{\vec{k}'}$ exists. Letting $r_{<i}$ and $g_{<i}$ denote $(r_1, \ldots, r_{i-1})$ and $(g_1, \ldots, g_{i-1})$ respectively, we additionally fix $(\boldsymbol{r}_{<i}, \boldsymbol{g}_{<i}) = (r_{<i}, g_{<i})$ (we use $r_{<i} \wedge g_{<i}$ to denote this event for simplicity).

The existence condition of $\mu^{\vec{k}'}$ in $\boldsymbol{T}$. Let α be the smallest-numbered node such that $\alpha > \mu^{\vec{k}}$ and the level of α is greater than $i-1$. Then $\mu^{\vec{k}'}$ exists if and only if α exists and $\mathsf{level}(\alpha) = i$. Hence, our goal is to determine α. By definition, to move from $\mu^{\vec{k}}$ to α in the random walk $\boldsymbol{w}$, one first move to the node corresponding to vertex $\mathsf{next}(\mu^{\vec{k}})$, and then keep going to the next node, until reaching a node with level at least i. The following algorithm implements this procedure and returns the simulated random walk, and we observe that it only uses the values of $(r_{\le i}, g_{\le i})$. Note that we use $(\cdots)$ to denote a sequence of vertices, and use $\circ$ to denote the concatenation of two sequences.

Algorithm 1: Simulating the random walk from s' until reaching a level greater than i

```
Function sim(s', i)
    if i = 0 then
        return (s')                          // stop here since all nodes have levels at least 1
    s_0 ← s', j ← 0, w ← ()                                          // start from s_0 = s'
    repeat
        w ← w ∘ sim(s_j, i − 1)   // simulate from s_j until hitting a node with level at least i
        x_{j+1} ← w_{|w|}        // vertex x_{j+1} corresponds to the next node after s_j with level ≥ i
        if g_i(a_{x_{j+1}}) = 1 then
            s_{j+1} ← r_i(a_{x_{j+1}}) // move to the next node since the node corresponding to x_{j+1}
                has level i
            j ← j + 1
    until g_i(a_{x_j}) = 0
    return x_j// stop here since the node corresponding to x_j has level > i
Function Find(s', i)
    return the last vertex in the sequence returned by sim(s', i)
```

[4] This is already enough for lower bounding E_{total}, as the contribution of "long" (non-short) $\vec{k}$ is negligible. Intuitively this is true because for a "long" $\vec{k}$, we have $|\vec{k}|_1 \ge \max_{i\in[\ell]} k_i > \tau/4$, the probability that $\mu^{\vec{k}}$ exists in the tree is quite small ($2^{-|\vec{k}|_1}$) assuming (2.12).

[5] One also needs to show that with probability $1/2$, μ has a level-j child with a uniformly random next-value, for all $j < i$. We ignore this part in the technical overview.

One can see that $\mathsf{sim}(\mathsf{next}(\mu), i-1)$ generates the entire *sub-walk* after μ until reaching the next node with level at least i. Now, the hope is to argue that, conditioning on $\mathcal{F}^{\vec{k}} \wedge r_{<i} \wedge g_{<i}$, we have

$$\boldsymbol{g}_i(\mathsf{Find}(\mathsf{next}(\mu), i-1)) = 1$$

with probability 1/2.

Two issues with the original random walk $\boldsymbol{w}$. There are two important issues with the argument above:

1. We need to argue $\boldsymbol{g}_i(\mathsf{Find}(\mathsf{next}(\mu), i-1))$ is independent from the event $\mathcal{F}^{\vec{k}} \wedge r_{<i} \wedge g_{<i}$.
2. Even if $\boldsymbol{g}_i(\mathsf{Find}(\mathsf{next}(\mu), i-1)) = 1$, it could be the case that $\boldsymbol{w}$ stops during the simulation of $\mathsf{sim}(\mathsf{next}(\mu), i-1)$ due to a collision, and in that case $\mu^{\vec{k}'}$ also does not exist.[6]

The second issue is fundamental, as it reveals the "global dependency nature" of the original random walk $\boldsymbol{w}$: the event that $\boldsymbol{w}$ stops depends on *all* entries in $\boldsymbol{w}$.

A locally simulatable extended random walk. To circumvent the second issue, we wish for our extended random walk $\bar{\boldsymbol{w}}$ to be *locally simulatable*. That is, knowing that node μ exists and knowing the value of $\mathsf{next}(\mu)$, together with fixed $r_{<i}$ and $g_{<i}$, one should be able to simulate the extended random walk $\bar{\boldsymbol{w}}$ after μ until reaching a node with level at least i. The second issue above amounts to the fact that $\mathsf{sim}(\mu, i)$ fails to locally simulate the walk $\boldsymbol{w}$, since it does not have enough information to determine whether $\boldsymbol{w}$ has already terminated during its simulation (it cannot determine whether there is a collision between the encountered node and the nodes before in $\boldsymbol{w}$).

Similar to the basic extended random walk in Section 2.3, for each $i \in [\ell]$, we extend the domain of $\boldsymbol{g}_i$ and $\boldsymbol{r}_i$ from $[m]$ to $[m] \cup \{\star_0, \star_1, \dots\}$ as follows: for each $t \in \mathbb{N}$, we sample $\boldsymbol{g}_i(\star_t) \in_{\mathsf{R}} \{0,1\}$ and $\boldsymbol{r}_i(\star_t) \in_{\mathsf{R}} [n]$, where all samples are independent.

Since the "local" simulation with respect to node 0, $\mathsf{next}(0) = \boldsymbol{s}$ and fixed $r_{\le \ell}$ and $g_{\le \ell}$ is just the entire random walk, we will define our extended random walk by giving its local simulation in Algorithm 2, and we set $\bar{\boldsymbol{w}} \leftarrow \mathsf{walk}(\boldsymbol{s}, \ell, 0)$.[7] Note that $\mathsf{walk}(\boldsymbol{s}, \ell, 0)$ also gives the extended tree $\bar{\boldsymbol{T}}$ by specifying level and next.

Establishing Claim 2.1 for $\bar{\boldsymbol{T}}$. One can inspect that the algorithm walk behaves the same as sim until a collision occurs at Line 8 (that is, there is a collision in $\{a_{x_1}, a_{x_2}, \dots, a_{x_{j+1}}\}$). That is, $\mathsf{sim}(\boldsymbol{s}, \ell)$ and $\mathsf{walk}(\boldsymbol{s}, \ell, 0)$ behave the same until reaching a collision $a_{\boldsymbol{w}_j} = a_{\boldsymbol{w}_k}$ for $j \neq k$. This implies that (2.9) holds.

To show Claim 2.1 holds for $\bar{\boldsymbol{w}}$ and $\bar{\boldsymbol{T}}$, we still have to argue that $\boldsymbol{g}_i(\mathsf{ExtFind}(\mathsf{next}(\mu), i-1))$ is independent from the event $\mathcal{F}^{\vec{k}} \wedge r_{<i} \wedge g_{<i}$. Formally proving this requires a delicate induction, but the intuition is that $\mathcal{F}^{\vec{k}}$ depends on at most k_i values in $\boldsymbol{g}_i$ and $\boldsymbol{r}_i$, and the procedure walk carefully ensures that $\boldsymbol{g}_i(\mathsf{ExtFind}(\mathsf{next}(\mu), i-1))$ is never one of them. Hence, since $k_i \le \tau/4$ and $\boldsymbol{g}_i$ is τ-wise independent, we have the desired independence.

Handling E_{bad} and the two-vertex case. We have just established Condition (2.12) which gives a lower bound for E_{total}; now we briefly discuss how to obtain an upper bound on E_{bad} sufficient for proving the desired lower bound on $\mathbf{Pr}[u \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})]$ using (2.14). One can first observe that (2.13) cannot hold for all possible $\vec{k}, \vec{k}^1, \vec{k}^2$, as there could be a collision between these three paths. In fact, let K be the total number of nodes in the union of the paths corresponding to $\vec{k}, \vec{k}^1, \vec{k}^2$. Then a revised estimate for $\mathbf{Pr}[\mathcal{F}^{\vec{k}} \wedge \mathsf{next}(\mu^{\vec{k}}) = u \wedge \mathcal{F}^{\vec{k}^1} \wedge \mathcal{F}^{\vec{k}^2} \wedge a_{\mathsf{next}(\mu^{\vec{k}^1})} = a_{\mathsf{next}(\mu^{\vec{k}^2})}]$ should be $\left(2^K \cdot n^2\right)^{-1}$. By a careful calculation, one can show that this revised estimate is still enough to show E_{bad} is upper bounded by $O(2^{3\ell}/n^2)$, which is good enough for our purposes.

However, even establishing this revised estimate is quite challenging. Recall that $\mathcal{F}^{\vec{k}} \wedge \mathcal{F}^{\vec{k}^1} \wedge \mathcal{F}^{\vec{k}^2}$ is equivalent to the condition that, for every level-i node β on the paths from root to $\mu^{\vec{k}}, \mu^{\vec{k}^1}$ or $\mu^{\vec{k}^2}$, it holds that $\boldsymbol{g}_i(a_{\boldsymbol{w}_\beta}) = 1$. This amounts to K events and we hope to show they are all independent. However, this is not true in general, as there can be a collision of $a_{\boldsymbol{w}_\beta}$ between two different paths among these three paths. We overcome this issue by showing that for each "bad node" $\mu^{\vec{k}}$, there must exist a "bad" collision pair $\vec{k}^1$ and $\vec{k}^2$ on the extended walk without this issue. In such case one can establish a revised estimate; subtracting all these revised estimates from E_{good} would still yield a good lower bound on $\mathbf{Pr}[u \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})]$.

[6] Indeed, if the simulation $\mathsf{sim}(\mathsf{next}(\mu), i-1)$ detects a pair of collision (two nodes α, β such that $a_{\boldsymbol{w}_\alpha} = a_{\boldsymbol{w}_\beta}$), it would loop forever.

[7] See the full version of the paper [CJWW21, Section 5.1] for a detailed explanation of Algorithm 2.

Algorithm 2: Algorithm for extended walk

Function $\mathsf{walk}(s', i, \mu_0)$ *(where $s' \in [n], 0 \le i \le \ell$)*
  if $i = 0$ **then return** (s')
  $C_0 \leftarrow \emptyset$, $\mathsf{star} \leftarrow$ false
  $j \leftarrow 0, s_0 \leftarrow s', w \leftarrow ()$
  repeat
    $w \leftarrow w \circ \mathsf{walk}(s_j, i-1, \mu_0 + |w|)$
    $x_{j+1} \leftarrow w_{|w|}$
    $y, \mathsf{star} \leftarrow \begin{cases} a_{x_{j+1}}, \text{false} & \text{if } a_{x_{j+1}} \notin C_j \wedge \neg\mathsf{star} \\ \star_t, \text{true} & \text{otherwise (where } t := \min\{t \in \mathbb{N} \mid \star_t \notin C_j\}) \end{cases}$
    $\mu_{j+1} \leftarrow \mu_0 + |w|$
    if $g_i(y) = 1$ **then**
      $C_{j+1} \leftarrow C_j \cup \{y\}$, $s_{j+1} \leftarrow r_i(y)$
      $\mathsf{level}(\mu_{j+1}) \leftarrow i, \mathsf{next}(\mu_{j+1}) \leftarrow r_i(y)$
      $j \leftarrow j + 1$
  until $g_i(y) = 0$
  return w
Function $\mathsf{ExtFind}(s', i)$
  return the last vertex in the sequence returned by $\mathsf{walk}(s', i, 0)$

Our proof for lower-bounding $\mathbf{Pr}[u, v \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})]$ follows the same template above, while using a more involved analysis to handle the dependency issues across the paths (we have to consider four paths now: two corresponding to u and v, and the other two corresponding to the "bad" collision pair).

3 Preliminaries

Let $[n]$ denote $\{1, 2, \ldots, n\}$. We use $\mathbb{N}$ to denote the set of non-negative integers. We use $\widetilde{O}(f)$ to denote $O(f \cdot \operatorname{poly}\log f)$ in the usual way; $\widetilde{\Omega}, \widetilde{\Theta}$ are defined similarly.

We measure the space complexity of an algorithm by the maximum number of bits in its working memory: the read-only input is not counted. We measure the time complexity by the number of word operations (with word length $\Theta(\log n)$) in the word RAM model.

For Element Distinctness and List Disjointness, we always assume the input arrays of length n consist of positive integers bounded from above by $m = n^c + c$, where c is a fixed constant independent of n. (We often abbrievate this by saying $m = \operatorname{poly}(n)$.) For an array $a \in [m]^n$, define the second frequency moment $F_2(a) = \sum_{i=1}^n \sum_{j=1}^n \mathbf{1}[a_i = a_j]$ as the number of colliding pairs (i, j) (including the case where $i = j$). Note that $n \le F_2(a) \le n^2$.

We will use the following standard pseudorandomness construction.

Theorem 3.1. (Explicit k-wise independent hash family, [CW79]; see also [Vad12, Corollary 3.34]) *For n, m, k, there is a family of k-wise independent functions $\mathcal{H} \subseteq \{h \mid h\colon \{0,1\}^n \to \{0,1\}^m\}$ such that every function from $\mathcal{H}$ can be described in $k \cdot \max\{n, m\}$ random bits, and evaluating a function from $\mathcal{H}$ (given its description, and given an input $x \in \{0,1\}^n$) takes time* $\operatorname{poly}(n, m, k)$.

We often use bold font letters (*e.g.*, $\boldsymbol{X}$) to denote random variables. We also use $\operatorname{supp}(\boldsymbol{X})$ to denote the support of random variable $\boldsymbol{X}$.

For a set U, we often use $x \in_{\mathsf{R}} U$ to denote the process of selecting an element x from U uniformly at random.

4 Properties of the Pseudorandom Family and their Implications

We will first define our pseudorandom hash family in Section 4.1, and then give the proofs of our main theorems in Section 4.2, assuming some key technical lemmas that will be proved in the full version of the paper [CJWW21].

4.1 Construction of the Pseudorandom Family We first introduce some handy notation. For two functions $a, b\colon [m] \to ([n] \cup \{\star\})$, we naturally view them as "restrictions" (where $\star$ means "unrestricted"), and define their *composition* as

$$(a \bullet b)(x) := \begin{cases} b(x) & b(x) \neq \star, \\ a(x) & \text{otherwise.} \end{cases}$$

Observe that $(a \bullet b) \bullet c = a \bullet (b \bullet c)$.

Let $\ell \leq \log n$ and $\tau = O(\log n \log\log n)$ be two positive integer parameters to be determined later. A sample $\boldsymbol{h}\colon [m] \to ([n] \cup \{\star\})$ from $\mathcal{H}_{\ell,m,n}$ is generated by an ℓ-level iterative restriction process, defined as follows.

Drawing a sample $\boldsymbol{h}$ from the pseudorandom hash function family $\mathcal{H}_{\ell,m,n}$

1. For each $i \in [\ell]$, independently draw two random functions $\boldsymbol{g}_i\colon [m] \to \{0,1\}$ and $\boldsymbol{r}_i\colon [m] \to [n]$ from τ-wise independent hash families (Theorem 3.1). Define $\boldsymbol{h}_i\colon [m] \to [n] \cup \{\star\}$ to be
$$\boldsymbol{h}_i(x) := \begin{cases} \star & \text{if } \boldsymbol{g}_i(x) = 0, \\ \boldsymbol{r}_i(x) & \text{if } \boldsymbol{g}_i(x) = 1. \end{cases}$$
2. Define $\boldsymbol{h}$ to be $\boldsymbol{h}_\ell \bullet \cdots \bullet \boldsymbol{h}_2 \bullet \boldsymbol{h}_1$.

Intuitively, the functions $g_i\colon [m] \to \{0,1\}$ control whether the value of $h(x)$ should be restricted at the i-th level, while the functions $r_i\colon [m] \to [n]$ determine the value that $h(x)$ is restricted to, at the i-th level. Note that $h(x) = \star$ if $g_1(x) = \cdots = g_\ell(x) = 0$, and $h(x) = r_j(x)$ if $g_1(x) = \cdots = g_{j-1}(x) = 0$ and $g_j(x) = 1$.

Since $m = \mathrm{poly}(n)$, the seed length for each $i \in [\ell]$ is $O(\log^2 n \log\log n)$ bits (Theorem 3.1), and hence the total seed length for describing the hash function h is $O(\ell \log^2 n \log\log n) = O(\log^3 n \log\log n)$. Slightly abusing notation, we also use $\boldsymbol{h} \in_{\mathsf{R}} \mathcal{H}_{\ell,m,n}$ to denote that $\boldsymbol{h}$ is a hash function generated as above.

Digraph $G_{a,h}$ and reachable set $f^*_{a,h}(s)$. Next we set up some notation. Recall that $a \in [m]^n$ is the input array. For a hash function $h\colon [m] \to [n]$, we define a mapping $f_{a,h}\colon [n] \to ([n] \cup \{\star\})$ by $f_{a,h}(x) := h(a_x)$. This mapping naturally defines a n-vertex digraph $G_{a,h}$, where each vertex $x \in [n]$ has one outgoing edge $x \mapsto h(a_x)$ if $h(a_x) \neq \star$, and no outgoing edge if $h(a_x) = \star$.

We use $f^*_{a,h}(s)$ to denote the set of vertices reachable in $G_{a,h}$ from s. When a and h are clear from context, we will simply write $f^*_{a,h}(s)$ as $f^*(s)$. Since each vertex in $G_{a,h}$ has at most one outgoing edge, note that the vertices in $f^*(s)$ form either a path or a "rho-shaped" component.

4.2 Proofs of the Main Results Let $a = (a_1, \ldots, a_n) \in [m]^n$ be the read-only input array. The BCM Element Distinctness algorithm [BCM13] uses the following version of Floyd's cycle-finding algorithm performed on the digraph specified by $f_{a,h}$.

LEMMA 4.1. ([BCM13, THEOREM 2.1]) *Assuming oracle access to $f_{a,h}\colon [n] \to ([n] \cup \{\star\})$, there is a deterministic algorithm $\mathsf{COLLIDE}(s)$ which finds the pair $(u, v) \in [n] \times [n]$ (if it exists) such that $u, v \in f^*_{a,h}(s), u \neq v$ and $a_u = a_v$, in $O(|f^*_{a,h}(s)|)$ time and $O(\log n)$ space.*[8]

In the BCM algorithm, h was chosen from a truly random hash family. Our goal is to show that sampling h from our pseudorandom hash family $\mathcal{H}_{\ell,m,n}$ also suffices. To do this, we need the following two properties of our hash family $\mathcal{H}_{\ell,m,n}$.

LEMMA 4.2. (BOUNDING THE VISIT PROBABILITY FOR A SINGLE VERTEX) *Suppose $\ell = \log n - \frac{\log F_2(a)}{2} - 10$.*[9]

[8] The original BCM algorithm works for $f_{a,h}\colon [n] \to [n]$. But it works equally well when some vertices v may have no outgoing edges (*i.e.*, $f_{a,h}(v) = \star$).

[9] We ignore all floors and ceilings for simplicity.

For every vertex $v \in [n]$, we have

$$\Pr_{\boldsymbol{h}\in_R\mathcal{H}_{\ell,m,n},\boldsymbol{s}\in_R[n]}[v \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})] = \Theta\left(\frac{1}{\sqrt{F_2(a)}}\right).$$

LEMMA 4.3. (LOWER BOUND FOR COLLISION PROBABILITY) *Suppose $\ell = \log n - \frac{\log F_2(a)}{2} - 10$. For every $u, v \in [n]$ such that $u \neq v$ and $a_u = a_v$, we have*

$$\Pr_{\boldsymbol{h}\in_R\mathcal{H}_{\ell,m,n},\boldsymbol{s}\in_R[n]}[u, v \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})] \geq \Omega\left(\frac{1}{F_2(a)}\right).$$

Lemma 4.2 and Lemma 4.3 are proved in the full version of this paper [CJWW21].

REMARK 4.4. *In Lemma 4.2, we obtain both a lower bound and an upper bound for $\mathbf{Pr}_{\boldsymbol{h},\boldsymbol{s}}[v \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})]$, and we will see shortly that only the upper bound will be useful in the proof of Theorem 1.1; the lower bound part of Lemma 4.2 can be seen as a warm-up for the proof of Lemma 4.3, which requires to prove a lower bound for the more involved two-vertex case (see the full version of the paper [CJWW21, Section 7]).*

Since $\ell \leq \log n$, each hash function h from our hash family $\mathcal{H}_{\ell,m,n}$ can be described with a seed of $O(\log^3 n \log\log n)$ bits and can be evaluated in $\operatorname{poly}\log(n)$ time and $O(\log^3 n \log\log n)$ space. Armed with the two lemmas above, we can prove our main theorems.

Reminder of Theorem 1.1. ELEMENT DISTINCTNESS *can be decided by a Monte Carlo algorithm in $\widetilde{O}(n^{1.5})$ time, with $O(\log^3 n \log\log n)$ bits of workspace and no random oracle. Moreover, when there is a colliding pair, the algorithm reports one.*

Proof. Given input $a \in [m]^n$, we first assume that we know the correct parameter $1 \leq \ell \leq \log n$ required in Lemma 4.2 and Lemma 4.3, and let $\mathcal{H}$ be the pseudorandom hash family $\mathcal{H}_{\ell,m,n}$. We run $O(n \log n)$ trials of the $\mathsf{COLLIDE}(s)$ algorithm (Lemma 4.1) on $f_{a,h}$, where each trial uses a fresh random $h \in \mathcal{H}$. We return YES if no collisions are found, and return NO otherwise. It is evident that this algorithm only requires one-way access to randomness, and the description of each h can be stored in low space.

We first analyze the running time of this algorithm. By Lemma 4.1, the running time of each trial is $O(|f^*_{a,h}(s)|)$. By Lemma 4.2, the expected running time of each trial is

$$\mathop{\mathbf{E}}_{\boldsymbol{h}\in\mathcal{H},\boldsymbol{s}\in[n]}[|f^*_{a,\boldsymbol{h}}(\boldsymbol{s})|] \cdot \operatorname{poly}\log(n) = \sum_{v\in[n]} \Pr_{\boldsymbol{h}\in\mathcal{H},\boldsymbol{s}\in[n]}[v \in f^*_{a,\boldsymbol{h}}(\boldsymbol{s})] \cdot \operatorname{poly}\log(n) \leq \frac{n \cdot \operatorname{poly}\log n}{\sqrt{F_2(a)}},$$

where the $\operatorname{poly}\log(n)$ factor comes from the time complexity of evaluating $h(\cdot)$. Hence, the expected total running time of $O(n \log n)$ trials is $\widetilde{O}(n^2/\sqrt{F_2(a)}) \leq \widetilde{O}(n^{1.5})$. By Markov's inequality, with at least $1 - o(1)$ probability, the total running time is bounded by $\widetilde{O}(n^{1.5})$.

To analyze the success probability, note that in a "NO" instance (*i.e.*, the elements are not distinct) there are $F_2(a) - n > 0$ pairs of $u, v \in [n]$ such that $u \neq v$ and $a_u = a_v$. By linearity of expectation, Lemma 4.3 implies that the success probability of each trial is

$$\Omega\left(\frac{F_2(a) - n}{F_2(a)}\right) \geq \Omega\left(1/n\right).$$

Since the samples of $h \in \mathcal{H}$ are independent across the trials, the probability of not finding any collisions is at most $(1 - \Omega(1/n))^{n \log n} \leq n^{-\Omega(1)}$. The proof then follows from a simple union bound.

Recall at the beginning of the proof, we assumed ℓ was known. To remove this assumption, our actual algorithm simply tries all possible $\ell \in \{1, 2, \ldots, \log n\}$ one by one (and terminates a trial if the running time is already too long for a specific ℓ), which only increases the overall running time by an $O(\log n)$ multiplicative factor. ∎

Now we similarly prove the performance of the List Disjointness algorithm.

Reminder of Theorem 1.2. *There is a Monte Carlo algorithm for* List Disjointness *such that, given input arrays $a = (a_1, \ldots, a_n), b = (b_1, \ldots, b_n)$ and an upper bound $p \geq F_2(a) + F_2(b)$, runs in $\widetilde{O}(n\sqrt{p})$ time and uses $O(\log^3 n \log\log n)$ bits of workspace and no random oracle.*

Proof. Similar to the proof of Theorem 1.1, we can assume that the correct ℓ required in Lemma 4.2 and Lemma 4.3 is known. Let array c be the concatenation of a and b, which must satisfy $F_2(c) \leq 2(F_2(a) + F_2(b)) \leq 2p$. We run $2p \log n$ trials of the $\mathsf{COLLIDE}(s)$ algorithm (Lemma 4.1) on $f_{c,h}$, each time using a fresh random $h \in \mathcal{H}$. We return NO if we find a collision in c where the two items come from a and b respectively. We return YES if the total time spent by the algorithm exceeds $\widetilde{O}(n\sqrt{p})$ while no such collisions have been found.

To analyze the running time, we focus on the first $F_2(c) \log n$ trials executed by the algorithm. By a similar argument in the previous proof, with at least $1 - o(1)$ probability, the total running time of these $F_2(c) \log n$ trials is at most

$$\widetilde{O}\left(F_2(c) \cdot \frac{n}{\sqrt{F_2(c)}}\right) \leq \widetilde{O}(n \cdot \sqrt{F_2(c)}).$$

By Lemma 4.3, the success probability of each trial is $\Omega(1/F_2(c))$ (note that in the previous proof we had $F_2(a) - n$ pairs of "good" collisions (u, v), while here it is possible that we have only one "good" pair, along with many "bad" pairs coming from the same input array). Then, the probability of finding a collision during the first $F_2(c) \log n$ trials is at least $1 - n^{\Omega(1)}$.

By a union bound, we can show that, on a "NO" input, with at least $1 - o(1)$ probability the algorithm will terminate in one of the first $F_2(c) \log n$ trials, without exceeding the time limit $\widetilde{O}(n\sqrt{p})$. ∎

Now we similarly give a low-space algorithm for Set Intersection, with near-optimal time complexity.

Reminder of Theorem 1.4. *There is a randomized algorithm that, given input arrays $A = (a_1, \ldots, a_n), B = (b_1, \ldots, b_n)$ where A and B are both YES instances of* Element Distinctness*, prints all elements in $\{a_1, \ldots, a_n\} \cap \{b_1, \ldots, b_n\}$ in $\widetilde{O}(n^{1.5})$ time, with $O(\log^3 n \log\log n)$ bits of workspace and no random oracle. The algorithm prints elements in no particular order, and the same element may be printed multiple times.*

Proof. Similar to the proof of Theorem 1.1, we can assume that the correct ℓ required in Lemma 4.2 and Lemma 4.3 is known.

As before, we define c to be the concatenation of a and b. We run $n \log^2 n$ trials of the $\mathsf{COLLIDE}(s)$ algorithm (Lemma 4.1) on $f_{c,h}$, each using a fresh random $h \in \mathcal{H}$. We print all the collisions found. Note these must be elements in $\{a_1, \ldots, a_n\} \cap \{b_1, \ldots, b_n\}$, by our assumption on the input: since A and B are YES instances of Element Distinctness, all colliding pairs must have one element from A and one element from B.

By a similar argument as in the proof of Theorem 1.1, with $1 - o(1)$ probability the total running time is bounded by $\widetilde{O}(n^{1.5})$. And for every element in the intersection, the probability that it is never printed is at most

$$\left(1 - \Omega\left(\frac{1}{F_2(c)}\right)\right)^{n \log^2 n} \leq n^{-\omega(1)},$$

where we used $F_2(c) = \Theta(n)$ implied by the input assumption. The proof then follows from a simple union bound. ∎

References

[Abr87] Karl R. Abrahamson. Generalized string matching. *SIAM J. Comput.*, 16(6):1039–1051, 1987.

[Abr91] Karl R. Abrahamson. Time-space tradeoffs for algebraic problems on general sequential machines. *J. Comput. Syst. Sci.*, 43(2):269–289, 1991.

[AGHP90] Noga Alon, Oded Goldreich, Johan Håstad, and René Peralta. Simple constructions of almost k-wise independent random variables. In *31st Annual Symposium on Foundations of Computer Science*, pages 544–553, 1990.

[Ajt02] Miklós Ajtai. Determinism versus nondeterminism for linear time RAMs with memory restrictions. *J. Comput. Syst. Sci.*, 65(1):2–37, 2002.

[Ajt05] Miklós Ajtai. A non-linear time lower bound for boolean branching programs. *Theory Comput.*, 1(1):149–176, 2005.

[AKKN15] Per Austrin, Petteri Kaski, Mikko Koivisto, and Jesper Nederlof. Subset sum in the absence of concentration. In *32nd International Symposium on Theoretical Aspects of Computer Science, STACS 2015*, pages 48–61, 2015.

[AKKN16] Per Austrin, Petteri Kaski, Mikko Koivisto, and Jesper Nederlof. Dense Subset Sum may be the hardest. In *Proceedings of the 33rd Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 13:1–13:14, 2016.

[Amb07] Andris Ambainis. Quantum walk algorithm for Element Distinctness. *SIAM Journal on Computing*, 37(1):210–239, 2007.

[AN08] Noga Alon and Asaf Nussboim. k-wise independent random graphs. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008*, pages 813–822. IEEE Computer Society, 2008.

[AS04] Scott Aaronson and Yaoyun Shi. Quantum lower bounds for the collision and the element distinctness problems. *J. ACM*, 51(4):595–605, 2004.

[BC82] Allan Borodin and Stephen A. Cook. A time-space tradeoff for sorting on a general sequential model of computation. *SIAM J. Comput.*, 11(2):287–297, 1982.

[BCM13] Paul Beame, Raphaël Clifford, and Widad Machmouchi. Element distinctness, frequency moments, and sliding windows. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 290–299. IEEE, 2013.

[Bea91] Paul Beame. A general sequential time-space tradeoff for finding unique elements. *SIAM J. Comput.*, 20(2):270–277, 1991.

[BFM+87] Allan Borodin, Faith E. Fich, Friedhelm Meyer auf der Heide, Eli Upfal, and Avi Wigderson. A time-space tradeoff for Element Distinctness. *SIAM J. Comput.*, 16(1):97–99, 1987.

[BGNV18] Nikhil Bansal, Shashwat Garg, Jesper Nederlof, and Nikhil Vyas. Faster space-efficient algorithms for Subset Sum, k-Sum, and related problems. *SIAM J. Comput.*, 47(5):1755–1777, 2018.

[Bri17] Karl Bringmann. A near-linear pseudopolynomial time algorithm for Subset Sum. In *Proceedings of the 28th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1073–1084, 2017.

[BSSV03] Paul Beame, Michael E. Saks, Xiaodong Sun, and Erik Vee. Time-space trade-off lower bounds for randomized computation of decision problems. *J. ACM*, 50(2):154–195, 2003.

[BV02] Paul Beame and Erik Vee. Time-space tradeoffs, multiparty communication complexity, and nearest-neighbor problems. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing*, pages 688–697. ACM, 2002.

[BW15] Samuel R. Buss and Ryan Williams. Limits on alternation trading proofs for time-space lower bounds. *Comput. Complex.*, 24(3):533–600, 2015.

[CJWW21] Lijie Chen, Ce Jin, R. Ryan Williams, and Hongxun Wu. Truly low-space element distinctness and subset sum via pseudorandom hash functions. *CoRR*, abs/2111.01759, 2021.

[Cob66] Alan Cobham. The recognition problem for the set of perfect squares. In *7th Annual Symposium on Switching and Automata Theory*, pages 78–87. IEEE Computer Society, 1966.

[CW79] J. Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[Din20] Itai Dinur. Tight time-space lower bounds for finding multiple collision pairs and their applications. In *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 405–434. Springer, 2020.

[EJT10] Michael Elberfeld, Andreas Jakoby, and Till Tantau. Logspace versions of the theorems of Bodlaender and Courcelle. In *Proceedings of the 51st IEEE Symposium on Foundations of Computer Scienc (FOCS)*, pages 143–152, 2010.

[FK16] Alan Frieze and Michał Karoński. *Introduction to random graphs*. Cambridge University Press, 2016.

[FK18] Michael A. Forbes and Zander Kelley. Pseudorandom generators for read-once branching programs, in any order. In *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018*, pages 946–955. IEEE Computer Society, 2018.

[FLvMV05] Lance Fortnow, Richard J. Lipton, Dieter van Melkebeek, and Anastasios Viglas. Time-space lower bounds for satisfiability. *J. ACM*, 52(6):835–865, 2005.

[GLP18] Isaac Goldstein, Moshe Lewenstein, and Ely Porat. Improved space-time tradeoffs for kSUM. In *26th Annual European Symposium on Algorithms, ESA 2018*, pages 37:1–37:14, 2018.

[HJ10] Nick Howgrave-Graham and Antoine Joux. New generic algorithms for hard knapsacks. In *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques Proceedings*, pages 235–256. Springer, 2010.

[HM21] Yassine Hamoudi and Frédéric Magniez. Quantum time-space tradeoff for finding multiple collision pairs. In *16th Conference on the Theory of Quantum Computation, Communication and Cryptography, TQC 2021*, pages 1:1–1:21, 2021.

[HS74] Ellis Horowitz and Sartaj Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM*, 21(2):277–292, 1974.

[Ind06] Piotr Indyk. Stable distributions, pseudorandom generators, embeddings, and data stream computation. *J. ACM*, 53(3):307–323, 2006.

[JVW21] Ce Jin, Nikhil Vyas, and Ryan Williams. Fast low-space algorithms for Subset Sum. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021*, pages 1757–1776. SIAM, 2021.

[Kan10] Daniel M. Kane. Unary subset-sum is in logspace. *CoRR*, 2010.

[Kar86] Mauricio Karchmer. Two time-space tradeoffs for element distinctness. *Theor. Comput. Sci.*, 47(3):237–246, 1986.

[Knu69] Donald E. Knuth. The art of computer programming, vol. 2: Seminumerical algorithms, 1969.

[LN10] Daniel Lokshtanov and Jesper Nederlof. Saving space by algebraization. In *Proceedings of the 42nd ACM Symposium on Theory of Computing (STOC)*, pages 321–330, 2010.

[LVWW16] Andrea Lincoln, Virginia Vassilevska Williams, Joshua R. Wang, and R. Ryan Williams. Deterministic time-space trade-offs for k-SUM. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016*, pages 58:1–58:14, 2016.

[MNT93] Yishay Mansour, Noam Nisan, and Prasoon Tiwari. The computational complexity of universal hashing. *Theor. Comput. Sci.*, 107(1):121–133, 1993.

[MP80] J. Ian Munro and Mike Paterson. Selection and sorting with limited storage. *Theor. Comput. Sci.*, pages 315–323, 1980.

[MW19] Dylan M. McKay and Richard Ryan Williams. Quadratic time-space lower bounds for computing natural functions with a random oracle. In *10th Innovations in Theoretical Computer Science Conference, ITCS 2019*, pages 56:1–56:20, 2019.

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

[Nis93] Noam Nisan. On read-once vs. multiple access to randomness in logspace. *Theor. Comput. Sci.*, 107(1):135–144, 1993.

[NW21] Jesper Nederlof and Karol Wegrzycki. Improving schroeppel and shamir's algorithm for subset sum via orthogonal vectors. In *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing*, pages 1670–1683. ACM, 2021.

[Pol75] John M. Pollard. A Monte Carlo method for factorization. *BIT*, 15:331–334, 1975.

[PP93] Boaz Patt-Shamir and David Peleg. Time-space tradeoffs for set operations. *Theor. Comput. Sci.*, 110(1):99–129, 1993.

[PR98] Jakob Pagter and Theis Rauhe. Optimal time-space trade-offs for sorting. In *39th Annual Symposium on Foundations of Computer Science, FOCS '98*, pages 264–268. IEEE Computer Society, 1998.

[SS81] Richard Schroeppel and Adi Shamir. A $T = O(2^{n/2})$, $S = O(2^{n/4})$ algorithm for certain NP-complete problems. *SIAM Journal on Computing*, 10(3):456–464, 1981.

[Vad12] Salil P. Vadhan. Pseudorandomness. *Found. Trends Theor. Comput. Sci.*, 7(1-3):1–336, 2012.

[vOW99] Paul C. van Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptol.*, 12(1):1–28, 1999.

[Wan14] Joshua R. Wang. Space-efficient randomized algorithms for K-SUM. In *Algorithms - ESA 2014 - 22th Annual European Symposium*, pages 810–829. Springer, 2014.

[Wil08] R. Ryan Williams. Time-space tradeoffs for counting NP solutions modulo integers. *Comput. Complex.*, 17(2):179–219, 2008.

[Yao88] Andrew Chi-Chih Yao. Near-optimal time-space tradeoff for Element Distinctness. In *29th Annual Symposium on Foundations of Computer Science*, pages 91–97. IEEE Computer Society, 1988.

[Yes84] Yaacov Yesha. Time-space tradeoffs for matrix multiplication and the discrete Fourier transform on any general sequential random-access computer. *J. Comput. Syst. Sci.*, 29(2):183–197, 1984.