

DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY


Streaming Algorithms and Graph Connectivity

Citation

Wang, Zhengyu. 2019. Streaming Algorithms and Graph Connectivity. Doctoral dissertation, Harvard University, Graduate School of Arts & Sciences.

Link

http://nrs.harvard.edu/urn-3:HUL.InstRepos:42029816

Terms of use

This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material (LAA), as set forth at

https://harvardwiki.atlassian.net/wiki/external/NGY5NDE4ZjgzNTc5NDQzMGIzZWZhMGFlOWI2M2EwYTg

Accessibility

https://accessibility.huit.harvard.edu/digital-accessibility-policy

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story

Streaming Algorithms and Graph Connectivity

A dissertation presented

by

Zhengyu Wang

to

Harvard John A. Paulson School of Engineering and Applied Sciences

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in the subject of

Computer Science

Harvard University

Cambridge, Massachusetts

April 2019

© 2019 Zhengyu Wang

All rights reserved.

Dissertation Advisor:
Professor Jelani Nelson

Author:
Zhengyu Wang

Streaming Algorithms and Graph Connectivity

Abstract

The streaming model treats the input as a sequence of items that can be read in one single pass using only limited storage, preferably poly-logarithmic in the input size. Streaming algorithms have numerous applications such as Internet monitoring, databases, and trend detection. In the first part of the thesis, I describe several new contributions to streaming algorithms, including:

- A space and time optimal algorithm (taking $O(1)$ words, $O(1)$ update and query time) that finds ℓ_2-heavy hitters in insertion streams. ℓ_2-heavy hitter is the strongest notion of heavy hitters (or frequent items) achievable in poly-logarithmic space.
- An optimal space lower bound for samplers. Samplers are used as the building blocks for streaming graph algorithms such as graph connectivity.

Graph connectivity is one of the most fundamental graph problems. In the second part of the thesis, I present new algorithms for graph connectivity in the dynamic streaming setting and parallel computing setting:

- A randomized algorithm for dynamic graph connectivity using $O(n \log^3 n)$ bits with improved update time: With $1/\mathrm{poly}(n)$ failure probability, the algorithm has worst-case running time $O(\log^3 n)$ per edge insertion, $O(\log^4 n)$ per edge deletion, and $O(\log n / \log\log n)$ per query, where n is the number of vertices.
- A randomized graph connectivity algorithm that runs in $O(\log \mathrm{diam}(G) \cdot \log\log_{M/n} n)$ parallel time under the Massive Parallel Computing (MPC) model for undirected graph G with n vertices and total memory constraint M, where $\mathrm{diam}(G)$ refers to the largest diameter among all its connected components.

Contents

Abstract . . . iii
Acknowledgments . . . viii

1 Introduction **1**
1.1 A Puzzle on Streaming Algorithms . . . 3
1.2 Heavy Hitters . . . 6
1.3 Samplers . . . 7
1.4 Dynamic Graph Connectivity . . . 9
1.5 Parallel Graph Connectivity . . . 13

2 ℓ_2 Heavy Hitters in Insertion Streams **15**
2.1 Introduction . . . 16
2.1.1 Problem formulation . . . 16
2.1.2 Related works . . . 17
2.1.3 Summary of our main results . . . 22
2.1.4 Our approach . . . 23
2.2 Preliminaries . . . 26
2.3 Overview of the Algorithm . . . 27
2.4 Bernoulli Process with 4-wise Independence . . . 30
2.5 Identifying a Single Heavy Hitter Given Approximation to F_2 . . . 32
2.5.1 Randomizing the labels . . . 32
2.5.2 Learning the bits of the randomized label . . . 33
2.6 F_2 Tracking . . . 37
2.7 The Complete Heavy Hitters Algorithm . . . 40
2.8 Conclusion . . . 43

3 Sampler Lower Bounds **45**
3.1 Introduction . . . 47
3.1.1 Problem formulation . . . 47
3.1.2 Related works . . . 49
3.1.3 Summary of our main results . . . 52

3.1.4 Our approach . . . 53
3.2 Preliminaries and Our Results . . . 58
3.3 Overview of techniques . . . 62
3.4 Communication Lower Bound for $\mathbf{UR}^{\subset}$ . . . 66
3.4.1 Encoding/decoding scheme . . . 66
3.4.2 Analysis . . . 67
3.5 Communication Lower Bound for $\mathbf{UR}_k^{\subset}$ . . . 73
3.5.1 Encoding/decoding scheme . . . 74
3.5.2 Analysis . . . 74
3.6 A tight upper bound for $\mathbf{R}_{\delta}^{\rightarrow,pub}(\mathbf{UR}_k)$ . . . 75
3.7 Another variant of samplers, with applications . . . 78
3.8 Conclusion . . . 81

4 Dynamic Graph Connectivity **83**
4.1 Introduction . . . 84
4.1.1 Problem formulation . . . 84
4.1.2 Related works . . . 84
4.1.3 Summary of our main results . . . 89
4.2 Cutset Data Structure and Invaiants . . . 89
4.2.1 Our techniques . . . 96
4.3 Our Dynamic Graph Connectivity Algorithm . . . 96

5 Parallel Graph Connectivity **102**
5.1 Introduction . . . 103
5.1.1 Problem formulation . . . 103
5.1.2 Related works . . . 105
5.1.3 Summary of our main results . . . 106
5.1.4 Our approach . . . 107
5.2 Neighbor Increment Operation . . . 108
5.3 Random Leader Selection . . . 112
5.4 Tree Contraction Operation . . . 114
5.5 Connectivity Algorithm . . . 117
5.6 MPC Implementation . . . 124

References **127**

List of Tables

2.1 Related results on heavy hitters 18

3.1 Guarantees needed by various works using samplers as subroutines 51

4.1 Related results on fully dynamic graph connectivity 85

5.1 Related results on parallel graph connectivity 105

List of Figures

2.1 Illustration of an insertion stream and its frequency vector f . . . 16
2.2 Illustration of CountSketch . . . 20
2.3 The supremum of a Bernoulli process . . . 23
2.4 The reduction from the heavy hitter problem to the single heavy hitter case . . . 24
2.5 Illustration of the chaining method . . . 31
2.6 Example execution of HH1 . . . 33

3.1 An instance of $\mathbf{UR}^{\subset}$ . . . 48
3.2 An example of the sub-sampling procedure used by samplers . . . 53
3.3 Decoding procedure for a sampler that encodes a uniform random set S. . . . 54
3.4 Illustration of an encoding/decoding procedure for a uniform random set S using a sampler, which gives a nearly optimal sampler lower bound. . . . 55
3.5 Illustration of an encoding/decoding procedure for a uniform random set S using a sampler, which gives an optimal sampler lower bound. . . . 57

4.1 Illustration of replacement edges for dynamic graph connectivity. . . . 91
4.2 An example execution of [KKM13] . . . 95

5.1 Illustration of the MPC model . . . 104

Acknowledgments

The five years of my Ph.D. career have left me with a thank-you list impossible to exhaust in a short acknowledgement. First, I would like to express my deepest gratitude for my advisor Professor Jelani Nelson who has always been a most reliable source of inspirations and encouragement. I would also like to thank Professors Boaz Barak and Madhu Sudan, and my colleague Yi-Hsiu Chen, for invaluable advice on my dissertation drafts. My academic journey could not have been so enjoyable without the support from my colleagues and friends. I learned a lot from cooperation with my co-authors including Alexandr Andoni, Vladimir Braverman, Stephen Chestnut, Nikita Ivkin, Michael Kapralov, Vasileios Nakos, Jelani Nelson, Jakub Pachocki, Zhao Song, Clifford Stein, David Woodruff, Mobin Yahyazadeh, Peilin Zhong (in alphabetical order). The Theory of Computation Group at Harvard University offered me a great platform to exchange ideas with other scholars and keep up with the state-of-the-art. I would like to thank Rohit Agrawal, Jaroslaw Blasiok, Anudhyan Boral, Mark Bun, Yi-Hsiu Chen, Chi-Ning Chou, Rasmus Kyng, Zhixian Lei, Yi Li, Kyle Luh, Tom Morgan, Preetum Nakkiran, Zhao Song, Thomas Steinke, Prayaag Venkat, Huacheng Yu, Fred Zhang (in alphabetical order), and other group members, for their help along the way. Sketching reading group, organized by Professor Jelani Nelson, was another space where new ideas sparkled. I am grateful to my family for the great support. I would like to thank my wife Weichu Wang, who is also my best friend and ideal companion, for her numerous help, encouragement, and love.

Chapter 1

Introduction

The streaming model was first formulated and popularized by [AMS96] (journal version [AMS99]) in 1996. The model addresses the challenge that there are much more input data than the capacity of storage can hold. For example, nowadays, a typical web server of a high traffic website receives packets at the speed of $0.01 \sim 0.1$ GB/s, while it only has up to several dozen GB of memory and several hundred GB of hard disks. In less data-intensive applications like network monitoring where we may only get interested in the headers of packets in order to compute some statistics of the traffic, simply storing all the headers is still infeasible or undesirable.

Most streaming problems can be formulated [Mut05] as data structure problems that maintain a vector $x \in \mathbb{Z}^{\mathcal{U}}$ under a sequence (or *stream*) of updates in the form of "$x_i \leftarrow x_i + \Delta$", where $\mathcal{U}$ is referred as the *universe*, $i \in \mathcal{U}$ and $\Delta \in \mathbb{Z}$. When we are not interested in the underlying structures of elements in the universe, we usually use $[n] := \{1, \ldots, n\}$ where $n = |\mathcal{U}|$ to refer to the universe and consider x to be an n-dimensional vector. Let m denote the length of the stream, i.e., the number of updates in the stream. For each $t \in [m]$, we say at time t, the t-th update/item in the stream arrives. Unless otherwise specified, we assume that $m = \text{poly}(n)$, in each update $|\Delta| \leq \text{poly}(n)$, and each machine word contains $\Theta(\log n)$ bits. At the end of the stream (i.e. at time m), the data structure is required to answer a query related to x of a pre-defined type. Namely, we know the type of query before reading the stream, and different types of queries define different streaming problems. The ultimate goal of designing a streaming algorithm is

to use space as small as possible, and meanwhile try to optimize both update time and query time. Ideally, a streaming algorithm should spend only $\text{poly}(\log n)$ space, and read the stream in a single pass.

The small space goal usually comes at a cost. First, we might no longer be able to get an exact solution. Instead, we might want to get an approximate solution, and we usually use parameter ε to characterize the fineness or precision of the approximation, and the exact meaning of ε varies from problem to problem. Second, designing streaming algorithms often needs randomness so that the algorithm may fail with probability δ over its coin tosses. Fortunately, for many applications, such compromise is satisfying. For example, if a server wants to know how many distinct incoming IPs, an approximation with a multiplicative error of 1% and failure probability 0.1% is usually good enough.

If Δ in an update can be either positive or negative, we call the stream a *turnstile stream*. If at any time $\forall i \in [n], x_i \geq 0$, we call it a *strict turnstile* stream. As an example of strict turnstile streams, consider the graph stream of a social network representing some kinds of relations like friendship. $\Delta = 1$ corresponds to a relation is established; $\Delta = -1$ corresponds to a relation is removed. x_i is always non-negative since a relation cannot be removed before being established. If Δ is required to be positive, we call it an *insertion-only* stream (or for short *insertion* stream). A notable special case where $\Delta = 1$ is called *cash register model*. For example, in some network traffic monitoring applications, let vector x be a counter on the number of packets from each IP where the universe consists of all the IPs. Δ is always 1 because every update corresponds to an incoming packet from an IP address.

Some history on streaming algorithms. Alon, Matias and Szegedy [AMS96] popularized the study of streaming algorithms. [AMS96] studied the problem of estimating frequency moments in an insertion stream, where the k-th moment is defined as $F_k = \sum_{i \in [n]} f_i^k$. Even before [AMS96], there are works in streaming algorithms. Morris [Mor78] designed a probabilistic algorithm that approximately maintains a counter and consumes space of only $O(\log \log n)$ bits, and the result can be seen as approximating F_1. Flajolet and Martin [FM85] studied the problem of

estimating F_0. Munro and Paterson [MP80] studied selection and sorting in the streaming setting. In particular, they show that finding the median of a stream of n numbers with constant failure probability requires $\widetilde{\Theta}(\sqrt{n})$ space. [MP80] also showed that sorting in the streaming model is intrinsically hard, by proving that in order to verify the ordering of a stream is of the form "$r_1, r_3, \ldots, r_{2\lceil n/2\rceil - 1}, r_2, r_4, \ldots, r_{2\lfloor n/2\rfloor}$" (where r_t means that the rank of the item is t), a streaming algorithm scanning P passes of the stream requires $\Theta(n/P)$ space.

Other streaming models. The *sliding window* model is a variant of the stream model proposed by [DGIM02]. The sliding window model has a parameter N, and at the end of the stream, we want to answer a query only related to the last N items (or equivalently, updates) in the stream. Namely, after N time steps, an update will become stale and will be discarded. The formulation of the sliding window model is motivated by applications such as network traffic monitoring where we usually only get interested in the most recent time span.

Semi-streaming model was proposed by [FKM+05]. They mainly wanted to process graph (such as call graph and web graph) streams where each item is an update to an edge, and they observed that most interesting graph problems cannot be solved in poly-logarithmic space. Therefore, in the semi-streaming model, the target space complexity is $O(n \cdot \mathrm{poly}(\log n))$, where n is the number of vertices. Moreover, they relaxed the number of passes to be constant or logarithmic. This characterizes the situation that the vertices of a large graph can be fit into the memory, but not all the edges.

1.1 A Puzzle on Streaming Algorithms

We start with discussing a well-known puzzle and some of its variants. The puzzle is highly relevant to samplers (Chapter 3) and dynamic graph connectivity (Chapter 4). Moreover, the discussion will cover some basic techniques for designing streaming algorithms.

Problem 1.1.1. *We are given a sequence (or equivalently, stream) of integers where all but one are duplicated. Find the integer that is not duplicated.*

For example, for input $(2, 3, 6, 4, 6, 4, 2)$*, the answer is* 3 *since only the number* 3 *appears once and all the rest appear exactly twice.*

The puzzle has a one-sentence solution: maintain the "xor" of the numbers in the sequence and it only takes space of one machine word. The correctness of the solution follows by the commutative and associative properties of the xor operation, as well as the fact that a number xor itself is 0, which is the *identity* of the xor operation in $\mathbb{N}$.

One standard follow-up question is: what if there are two numbers (instead of one number) in the stream that are not duplicated? A canonical solution takes $O(1)$ words of space but requires 2 passes over the input stream. In the first pass, we compute the xor of all numbers in the stream and denote the result as K. Let a and b denote the two numbers that are not duplicated, then we have $K = a \oplus b$. Let K_i denote the i-th bit of K. We find i^* such that $K_{i^*} = 1$. Such i^* must exist because $K = a \oplus b \neq 0$. Note that the i^*-th bit of a must be different from the i^* bit of b. Without loss of generosity, assume the i^*-th bit of a equal to 0, and the i^*-th bit of b equal to 1. In the second pass, we divide the stream into two sub-streams S_0, S_1 where S_j contains all number q with $q_{i^*} = j$. We maintain the xor for each sub-stream, which will be equal to a and b respectively. This is because a is the only number that is not duplicated in the first sub-stream, and b is the only number that is not duplicated in the second sub-stream. By getting prepared for all possible values of i^*, we can get a streaming algorithm with a single pass. As a trade-off, the space complexity blowing up by a factor of ω, where ω is the number of bits of a machine word.

More generally, we can ask what if there are k numbers in the stream that are not duplicated. Let n denote the length of the stream. Then we have $(n - k)/2$ distinct duplicated numbers. We assume that $k \ll n$, otherwise we can store the entire stream, which takes $O(n)$ words, while on the other hand, the space lower bound of the problem is $\Omega(k)$ because simply recording the answer needs $\Omega(k)$ words.

Recall our approaches for $k = 2$ mentioned above. We managed to isolate each non-duplicated integer in a separate sub-stream. Here is another way to view this problem. We consider each item in the stream a (copyable) ball and each sub-stream a bucket. Note that same item in the universe goes to the same bucket. Basically, the 2-pass algorithm maintains 2 buckets and it

learns how to distribute the balls into buckets in its first pass. In contrast, the 1-pass algorithm maintains 2ω buckets (i.e. ω pairs of buckets), and each ball goes to ω buckets. For general k, suppose we have B buckets, and each element in the universe goes to a uniform random bucket. Then the probability is at least $1 \cdot (1 - \frac{1}{B}) \cdot \cdots \cdot (1 - \frac{k-1}{B}) \geq 1 - \frac{k^2}{B}$ that all the non-duplicated integers are isolated. We can represent the mapping from balls to buckets as a hash function $h : [2^\omega] \to [B]$. Note that as a streaming algorithm we cannot afford storing h because it takes $2^\omega \log B$ bits. Fortunately, we can replace the full-wise independence to be 2-wise independence, which only takes $O(\omega + \log B)$ bits. Thus by taking $B = O(k^2)$ we have obtained a streaming algorithm with space complexity $O(k^2)$ words and constant failure probability.

Using $\Theta(k^2)$ buckets is wasteful in the sense that $1 - \frac{1}{k}$ fraction of buckets will be "empty". Here is another approach. For each bucket, we put each item into the bucket with probability $\frac{1}{k}$, then with probability $k \cdot \frac{1}{k} \cdot (1 - \frac{1}{k})^{k-1} \approx \frac{1}{e}$, the bucket will contain exactly one (uniform random) non-duplicated item. By the coupon collecting problem, we need $O(k \log k)$ buckets in order to collect all k non-duplicated items. Before concluding that we obtain a streaming algorithm with space complexity $O(k \log k)$ words and failure probability $1/\text{poly}(k)$, we need to find a way to efficiently identify whether a bucket contains exact one non-duplicated item or not. The task is equivalent to identifying whether a bucket is "empty", where by empty, we mean that it does not contain any non-duplicated item. We can achieve that by first assigning each item a label by a random hashing and then maintaining the xor of the labels. We say the bucket is empty if and only if the xor value is 0. Then the failure probability is exponentially small in the length bit length of a label. Choosing the bit length of a label to be $\Theta(\log k)$ bits and using a k-wise independent hash function for assigning elements to buckets give us $O(k \log k)$ words algorithm with $O(1)$ failure probability.

The discussion of the puzzle reveals some useful techniques in designing streaming algorithms. For example, partitioning the stream into B sub-streams (by partitioning the universe into B parts) can reduce the complexity of the problem. In the solution to the puzzle, by sub-streaming, the problem for general k is reduced to the problem for $k = 1$, which is much more approachable. In the $O(k \log k)$ words solution, by setting $B = k$, a non-duplicated element is identified with

only constant probability. We can repeat $O(\log k)$ times to boost the success probability. Note that in the $O(k^2)$ and $O(k \log k)$ words solutions, we use 2-wise and k-wise instead of full-wise independent hash functions, respectively, to save space.

1.2 Heavy Hitters

Instead of non-duplicated items, now we want to find all frequent items in a data stream, we call the task heavy hitter problem. More specifically, in an insertion stream $p_1, \ldots, p_m \in [n]$, *frequent items* or *heavy hitters* refers to those item i such that its frequency f_i exceeds a threshold τ. By letting $\tau = \varepsilon \|f\|_p$ where $\|f\|_p$ is the ℓ_p norm of the frequency vector, we can guarantee that there are at most ε^{-p} heavy hitters. In that case, we call them (ε, ℓ_p)-heavy hitters, or ℓ_p-heavy hitters when ε is fixed. In this introduction, we fix ε to be a small constant (say $\varepsilon = 10^{-3}$).

It is known[1] that finding ℓ_p-heavy hitters is harder for larger p. It worth pointing out that $p = 2$ is a sharp threshold in the space complexity for finding ℓ_p-heavy hitters. For $p < 2$, it takes poly-logarithmic space [CCF04], while for $p > 2$, it takes polynomial space [CKS03]. Therefore, ℓ_2-heavy hitter is the strongest notion of heavy hitters achievable in poly-logarithmic space.

The first algorithm for finding ℓ_2 heavy hitters is CountSketch [CCF04], which achieves $O(\log n)$ words of space, $O(\log n)$ update time, and $\widetilde{O}(n)$ query time. [BCIW16] developed CountSieve that achieves $O(\log \log n)$ words of space, $O\big(2^{(\log \log n)^2}\big)$ update time, and $\widetilde{O}(n)$ query time.

Our contributions. In Chapter 2, we present a new algorithm called BPTree for finding ℓ_2-heavy hitters in insertion streams with space complexity $O(1)$ words and success probability 0.99. The algorithm takes $O(1)$ process time per update and query. As part of our heavy hitter algorithm, we also develop a new streaming algorithm tracking ℓ_2 norm using $\Theta(1)$ words with constant success probability. Our result on derandomizing Bernoulli processes [BCIW16] with 4-wise independent random signs is also of independent interest.

Similar to the techniques used in the puzzle, by sub-streaming and repeating, BPTree first

[1] See Table 2.1 for (almost) tight space bounds for finding ℓ_p-heavy hitters. Moreover, [JST11] showed a blackbox reduction for a slight different definition of heavy hitters.

reduces the task of finding all the ℓ_2-heavy hitters to the single ℓ_2-heavy hitter case, where the ratio between the frequency of the heavy hitter and the ℓ_2 norm of the frequency vector excluding the heavy hitter exceeds a sufficiently large constant (i.e., heavy hitter H has the property that $f_H / \sqrt{\sum_{i \neq H} f_i} \geq C$ where C is a sufficiently large constant). It is useful to regard other items in the stream as noise, and f_H is at least C times the ℓ_2 norm of the noise.

Then, BPTree assumes the knowledge of a constant approximation of $\|f\|_2$ (we manage to remove the assumption by the end). BPTree maintains a Bernoulli process (BP) to learn one-bit information of the heavy hitter H. The Bernoulli process needs to consume $\frac{1}{10} f_H$ occurrences of H in the stream in order to succeed with a constant probability. The key observation is that using this one-bit information of H, we can on average remove half of the noise so that in the next round, we can bring down both the consumption of H and the failure probability by a constant fraction. Therefore, both the consumption of H and the failure probability decay geometrically, and by a union bound, BPTree is able to learn $\Theta(\log n)$ bits information of H with constant failure probability.

Bibliographic remarks: Chapter 2 is based on joint work with Vladimir Braverman, Stephen Chestnut, Nikita Ivkin, Jelani Nelson, and David Woodruff [BCI+17].

1.3 Samplers

A natural variant of the puzzle in Section 1.1 is to find an arbitrary non-duplicated item (or equivalently, an item appearing an odd number of times) instead of all of them. If k is known, similar to our $O(k \log k)$ words algorithm for the puzzle (but we just maintain one bucket instead of $O(k \log k)$ buckets), we sub-sample the items in the universe with probability $1/k$, and with constant probability exact one desirable item has been sub-sampled so that we can successfully recover it. Therefore, it takes $O(1)$ words of space to solve the problem with constant success probability. By repeating $O\big(\log(1/\delta)\big)$ times with independent randomness, we can boost the success probability to $1 - \delta$. If we do not know k, we can try $(1 + \log n)$ different sub-sampling rates, namely 2^{-i} for $i \in \{0, \ldots, \log n\}$. Thus we have obtained a data structure that takes

$O\big(\log(1/\delta)\log^2 n\big)$ bits of space and with probability $1-\delta$ outputs a (uniform random) item from the items that appear an odd number of times in the stream.

In a turnstile stream, a vector $x \in \mathbb{Z}^n$ starts as 0^n and receives a stream of updates in the form of $x_i \leftarrow x_i + \Delta$. By the end of the stream, a sampler is required to output an element in $\mathrm{supp}(x) = \{i : x_i \neq 0\}$. We call the task support-finding if any element in the support is a valid output. An ℓ_0-sampler additionally requires the output to be uniformly distributed in $\mathrm{supp}(x)$. More generally, an ℓ_p-sampler outputs i with probability proportional to $|x_i|^p$. Clearly, ℓ_p-sampler solves the support-finding problem, so lower bound on support-finding implies lower bound on ℓ_p-sampler.

It was shown in [JST11] that the space complexity of ℓ_0-sampler with constant failure probability is $\Theta(\log^2 n)$ bits. [JST11] also implemented an ℓ_0-sampler with failure probability at most δ using $O\big(\log(1/\delta)\log^2 n\big)$ bits. Prior to our work, no non-trivial lower bounds are known in terms of δ. On the other hand, graph stream applications often require $\delta = O(1/\mathrm{poly}(n))$. Therefore, it is interesting to bridge the gap.

Our contributions. In Chapter 3, we prove an optimal space lower bound of $\Omega\big(\log(1/\delta)\log^2 n\big)$ bits for any sampler (or support-finding algorithm) with failure probability $\delta > 2^{-n^{99}}$. Our sampler lower bound matches the best upper bound in [JST11]. Note that [JST11] already gave a lower bound of $\Omega(\log^2 n)$ bits for constant δ, and our lower bound is for small δ. For example, graph streaming applications usually require $\delta = \Omega(1/\mathrm{poly}(n))$. We also consider a variant of sampler that outputs k samples and we give an optimal lower bound.

In addition to sampler lower bounds, we decide the one-way randomized communication complexity of universal relation with failure probability δ. Let $\mathbf{UR}_k^{\subset}$ denote two party communication problem called *universal relation* in which Alice and Bob receive $x, y \in \{0,1\}^n$ respectively with the promise that $\mathrm{supp}(y) \subset \mathrm{supp}(x)$ and they want to find $\min\{k, |\mathrm{supp}(x-y)|\}$ elements in $\{i \in [n] : x_i \neq y_i\}$. We prove that its one-way public-coin communication complexity $\mathbf{R}_\delta^{\rightarrow,pub}(\mathbf{UR}_k^{\subset}) = \Theta(\min\{n, t\log^2(n/t)\})$ where $t = \max\{\log(1/\delta), k\}$.

Our results also imply an optimal space lower bound of $\Omega\big(\log(1/\delta)\log^2 n\big)$ bits for finding

duplicates with failure probability δ. In $\mathsf{FindDuplicate}(n)$, one is given a length-$(n+1)$ stream of integers in $[n]$, and the algorithm must output some element $i \in [n]$ which appeared at least twice in the stream.

Moreover, we show that if the updates and queries to a randomized data structure with failure probability δ are not independent of the randomness used in the data structure, then the failure probability can be approximately bounded by δ plus a "loss" which equals to their mutual information divided by $\log(1/\delta)$. This bound is essentially tight and is of independent interest in dynamic data structure designs.

Bibliographic remarks: Chapter 3 is based on joint work with Michael Kapralov, Jelani Nelson, Jakub Pachocki, David Woodruff and Mobin Yahyazadeh [KNP+17].

1.4 Dynamic Graph Connectivity

Dynamic graph connectivity is a fundamental problem in data structure and graph algorithm design. Let graph $G = (V, E)$, where V is the set of vertices, and E is the set of edges. We usually denote $n = |V|$ and $m = |E|$. A data structure for *fully dynamic* graph connectivity maintains updates consisting of edge insertions and edge deletions, and answers queries in the form of "are vertex s and t connected in the graph?" There is also a variant of the problem where query simply asks whether the graph is connected. Since the queries on whether two vertices are connected appear more frequently in applications, and techniques that solve the two vertices case usually can be adapted to answer whether the entire graph is connected, most literature focus on the two vertices case, and we will as well.

In the thesis, we consider G to be an undirected graph. The problem where G is a directed graph is called *graph reachability*, which is considered as a harder problem. If updates only consist of edge insertions, we call the case *incremental* or *insertion-only*; similarly if updates only consist of edge deletions, we call the case *decremental* or *deletion-only*. We will focus on the fully dynamic case, where both edge insertion and deletion are allowed.

Our goal is to minimum update and query time. When measuring time complexity for

dynamic data structures, two most popular measures are worst-case running time and amortized running time (i.e. average running time per operation). Depending on whether the algorithm uses randomness, an algorithm is either randomized or deterministic. If a randomized algorithm never outputs a wrong answer (but has a running time bound that only holds in expectation), we call it a *Las Vegas* algorithm, otherwise, we call it a *Monte Carlo* algorithm.

If we impose some special properties of the graph, for example, assuming the graph is a planar graph or a forest, we have a chance to design algorithms that achieve better performance over the general case. Notably, if the graph is a forest, [ST83] design a deterministic *dynamic tree* data structure with worst update and query time $O(\log n)$. It remains a long-standing open problem for general graphs, and even deterministic algorithm with a worst-case process time that is better than $\widetilde{O}(\sqrt{n})$ is not known.

Dynamic graph connectivity is a useful primitive for designing dynamic graph algorithms. For example, it can be used to maintain a $(1+\varepsilon)$ approximation of spanning forest of a weighted graph. Another notable example is that it can be used to maintain the number of connected components of a dynamic graph. This immediately solves dynamic graph bipartiteness, since there is a simple reduction that reduces the bipartiteness of a graph to the parity of the number of connected components of a new graph whose vertex set and edge set both double in size. Dynamic graph connectivity is interesting also because its solution can usually shed lights on other dynamic graph problems.

Most dynamic connectivity algorithms maintain a dynamic spanning forest of the graph. When an edge $e = \{u, v\}$ is inserted, the algorithm will first check if u and v are connected in the spanning forest. If they are not connected, the algorithm will connect the trees containing u and v together in the spanning forest. This step can be implemented efficiently in $O(\log n)$ worst-case time by using the dynamic tree data structure introduced in [ST83]. When an edge $e = \{u, v\}$ is deleted where e is an edge in the maintained spanning forest, the algorithm has to try to find a replacement edge in G reconnecting u and v in the spanning forest, or decide that u and v are disconnected in G. In order to find the replacement edge efficiently, samplers are used in [KKM13] (see also [AGM12a] which make the use of samplers explicit, but the focus of

[AGM12a] is space complexity and it is not a dynamic data structure).

We formulate the main idea of [KKM13] in terms of streaming algorithms. Let the universe $\mathcal{U}$ where $|\mathcal{U}| = \binom{n}{2}$ denote all possible edges. We assume that at any time, there is no multi-edge in the graph. We regard the updates as n insertion-only streams of edges $\{s_v\}_{v\in V}$, where stream s_v contains all the edges in the updates (both insertion and deletion) that are incident to v. Namely, when edge $e = \{u, v\}$ is inserted into or deleted from the graph, e will be inserted to both s_u and s_v. One nice property of $\{s_v\}_{v\in V}$ is that for any $W \subseteq V$, the edges appearing odd number of times in the stream $\cup_{v\in W} s_v$ (obtained by concatenating all the s_v for $v \in W$) are exactly the edges in the cut $\{\{u, v\} : u \in W, v \in V \setminus W\} \cap E$.

[KKM13] maintain a sketch $\textsc{sk}_\sigma(s_v) \in \mathbb{Z}^d$ for each $v \in V$, where σ refers to the random seed used in computing the sketch. They require the sketch function $\textsc{sk}$ have the following two properties. First, for any insertion-only stream s, with success probability $1 - 1/\text{poly}(n)$ under the randomness of σ, given $\textsc{sk}_\sigma(s)$ one can query if there exists an item appearing odd number of times in s, and find one if exist. Second, they require the sktech to be *linear* in the sense that for any two streams s_1 and s_2, $\textsc{sk}_\sigma(s_1 \cup s_2) = \textsc{sk}_\sigma(s_1) + \textsc{sk}_\sigma(s_2)$, where $s_1 \cup s_2$ denotes the stream obtained by concatenating s_1 and s_2. [KKM13] implement such a sketch function $\textsc{sk}$ for $d = O(\log^2 n)$ (the value in each dimension is bounded by $\text{poly}(n)$), and it takes $O(\log^2 n)$ worst-case time to update or query any sketch.

They then construct a data structure called *cutset* maintaining a dynamic forest using Euler tour representation [HK95] of trees, so that for each tree $T = (V_T, E_T)$ in the forest, $\sum_{v\in V_T} \textsc{sk}_\sigma(s_v)$ can be computed efficiently. The sketch $\sum_{v\in V_T} \textsc{sk}_\sigma(s_v)$ is used to find an outgoing edge of V_T, i.e., an edge in the cut $\{\{u, v\} : u \in V_T, v \in V \setminus V_T\} \cap E$. We call an edge tree edge if it belongs to a tree in the forest which the data structure maintains. In summary, the cutset data structure supports updates including tree edge insertion, tree edge deletion, (graph) edge insertion, (graph) edge deletion in worst-case time $O(\log^3 n)$, and finds an outgoing edge of a tree in the forest in worst-case time $O(\log^2 n)$. In addition, as a dynamic tree data structure, for any $v \in V$ cutset can return an identifier to the tree vertex v belongs to in worst-case time $O(\log n / \log\log n)$.

Note that the query of finding outgoing edge fails with probability $1/\text{poly}(n)$ under the

randomness of σ, and we require that the updates to cutset independent from σ. This requirement prohibits us from directly maintaining the spanning forest of G with a cutset, since after a tree edge got deleted and we find an outgoing edge with the cutset, we should not insert the outgoing edge into the cutset because the outgoing edge depends on σ, and by doing so we will no longer have success guarantees for future queries of outgoing edges.

Instead, [KKM13] maintain $\ell + 1$ levels where $\ell = \lceil \log n \rceil$, and on level $i \in \{0, \ldots, \ell - 1\}$, $cutset_i$ is maintained to provide additional tree edges for $cutset_{i+1}$. The tree edges of $cutset_0$ is always $\emptyset$, and $cutset_0$ is used to find an outgoing edge for each $v \in V$. These outgoing edges will then be inserted as tree edges to all $cutset_i$ where $i > 0$. Similarly, for $i < \ell$, $cutset_i$ is used to find an outgoing edge for each tree in the forest maintained by $cutset_i$, and these outgoing edges will be inserted as tree edges to all $cutset_{i'}$ where $i' > i$. As a result, $cutset_\ell$ will maintain the spanning forest of G, because, on layer i, the minimum size of a non-maximal tree maintained by $cutset_i$ is 2^i.

Whenever an edge is inserted or deleted from the graph, every $cutset_i$ will update the graph edge accordingly (to update the sketches). This takes $O(\log^4 n)$ worst-case time. In addition, when an edge $e = \{u, v\}$ is inserted into the graph, if u and v are not in the same tree in $cutset_\ell$, we know that u and v were not connected before, and will insert e as a tree edge to all the $cutset_i$, which takes $O(\log^4 n)$ worst-case time; when an edge $e = \{u, v\}$ is deleted from the graph, the algorithm will first find the smallest i^* so that e is a tree edge in $cutset_{i^*}$, and then iterate i from i^* to ℓ to do the following: (1) remove e from the tree edges in $cutset_i$, (2) use $cutset_{i-1}$ to find replacement edges for trees containing u and v respectively, (3) insert the replacement edges as tree edges in $cutset_i, \ldots, cutset_\ell$, and note that this step might result in cycles but fortunately we can maintain a separate efficient data structure to handle this. The worst-case time for edge deletion is $O(\log^5 n)$. Each connectivity query can be done in $O(\log n / \log\log n)$ worst-case time by $cutset_\ell$.

Our contributions. In Chapter 4, we improve the update time of [KKM13] by a factor of $\log n$. Namely, we have obtained a randomized data structure for dynamic graph connectivity with

failure probability $1/\text{poly}(n)$ where each edge insertion takes $O(\log^3 n)$ worst-case time, each edge deletion takes $O(\log^4 n)$ worst-case time, and each query takes $O(\log n/\log\log n)$ worst-case time. Moreover, similar to [AGM12a], our data structure takes $O(n\log^2 n)$ words of memory. The improvement is made by observing that we can reduce the dimension of sketch $\text{sk}_\sigma(\cdot)$ to $O(\log n)$, so that the sketch has a weaker guarantee that if the input stream s does not contain any element appearing odd number of times, then querying sketch $\text{sk}_\sigma(s)$ will always tell "s does not contain any element appearing odd number of times"; otherwise it tells "s contains an element appearing odd number of times" with probability $9/10$, and outputs one such element correctly with probability $1-1/\text{poly}(n)$. We argue with a new invariant, and show that $O(\log n)$ levels are sufficient to prove that the top level is a spanning forest of the graph with probability $1-1/\text{poly}(n)$.

Bibliographic remarks: The result presented in this chapter is based on [Wan15].

1.5 Parallel Graph Connectivity

The increasing popularity of parallel computing systems such as MapReduce [DG04, DG08], Hadoop [Whi12], Spark [ZCF+10] necessitates a theoretical formulation of these parallel systems. The works of [FMS+10, KSV10, GSZ11, BKS13, ANOY14] have lead to the *massive parallel computing* (MPC) model, which is a variant of the *bulk synchronous parallel* (BSP) model [Val90].

The MPC model can be specified by two parameters γ and δ (we call it (γ,δ)-MPC model), where $\gamma \geq 0$ captures the total space constraint and $\delta > 0$ captures the local memory size on each processor (or machine, we use them interchangeably in the thesis). In particluar, the (γ,δ)-MPC model has p processors, and each processor has $s = \Theta(N^\delta)$ space where N denote the input size, and the total memory constraint is $M = p\cdot s = \Theta(N^{1+\gamma})$. Note that δ is a constant in $(0,1)$, and γ is either 0 which corresponds to the most restrictive case or a positive constant where an algorithm is allowed to use $\Theta(N^\gamma)$ times more memory than the input size. The input is distributed on the local memory of $\Theta(N/s)$ input machines. The computation proceeds in rounds. In each round, each machine performs computation on the data in its local memory and sends messages to other

machines at the end of the round. The total size of messages sent or received by a machine in a round is bounded by s. In the next round, each machine only holds the received messages in its local memory. At the end of the computation, the output is distributed on the output machines. The parallel time of an MPC algorithm is defined as the number of rounds the algorithm takes.

An MPC algorithm can be equivalently seen as a small circuit, with arbitrary, N^δ fan-in gates, the depth of the circuit is the parallel time. Any PRAM algorithm can be simulated on MPC in the same parallel time [KSV10, GSZ11]. In fact, the MPC model is more powerful than the classic *parallel random-access machine* (PRAM) [Wyl79]. For example, it is shown in [BH89] that computing the xor of n bits requires $\Theta(\log n / \log\log n)$ parallel time in CRCW (concurrent read concurrent write) PRAM, the strongest PRAM model; while it takes $\Theta(\delta^{-1})$ parallel time in the (γ, δ)-MPC model.

The main algorithmic question of this area is then: for which problems can we design MPC algorithms that are faster than the best PRAM algorithms? Graph problems have been particularly well studied and one fundamental problem is connectivity in a graph. While this problem has a standard logarithmic time PRAM algorithm [SV82], we do not know whether we can solve it faster in the MPC model. There are strong indications that such algorithms do not exist: [BKS13] showed logarithmic lower bounds for restricted algorithms.

Our contribution to parallel graph connectivity. In Chapter 5, we present a graph connectivity algorithm that runs in $O(\log \mathrm{diam}(G) \cdot \log\log_{M/n} n)$ parallel time under the Massive Parallel Computing (MPC) model for undirected graph G with n vertices and total memory constraint $\Theta(M)$. If G is disconnected, $\mathrm{diam}(G)$ refers to the largest diameter among all its connected components.

Bibliographic remarks: The results described in Chapter 5 were developed jointly with Alexandr Andoni, Clifford Stein, Zhao Song, and Peilin Zhong [ASS+18].

Chapter 2

ℓ_2 Heavy Hitters in Insertion Streams

In this chapter, we present a new one-pass streaming algorithm called BPTree for finding ℓ_2 heavy hitters in insertion streams. En route to BPTree we give a new supremum bound for Bernoulli processes with bounded independence, as well as a more advanced analysis of the AMS sketch [AMS99] for tracking the ℓ_2 norm of the frequency vector at the cost of using 8-wise independent random signs rather than 4-wise independent signs.

This chapter is organized as follows. Section 2.1 consists of 4 parts: Section 2.1.1 defines the heavy hitter problem, Section 2.1.2 reviews some of the streaming literature on heavy hitters, Section 2.1.3 summarizes our main results, Section 2.1.4 gives an overview of our approach. Section 2.2 gives preliminaries and defines some notations. Section 2.3 presents our main subroutine HH1 which assumes a constant approximation of the ℓ_2 norm of the frequency vector. Section 2.4 proves an upper bound on the expected supremum of the Bernoulli processes with bounded independence used by the subroutine. Section 2.5 uses the supremum bound to prove the guarantees of the main subroutine HH1. Section 2.6 shows how to track the ℓ_2 norm of the frequency vector. Section 2.7 presents subroutine HH2 which makes use of the ℓ_2 tracker. It also describes and analyzes the complete algorithm BPTree. Section 2.8 concludes this chapter.

Bibliographic remarks: The results described in this chapter were developed jointly with Vladimir Braverman, Stephen Chestnut, Nikita Ivkin, Jelani Nelson, and David Woodruff [BCI+17].

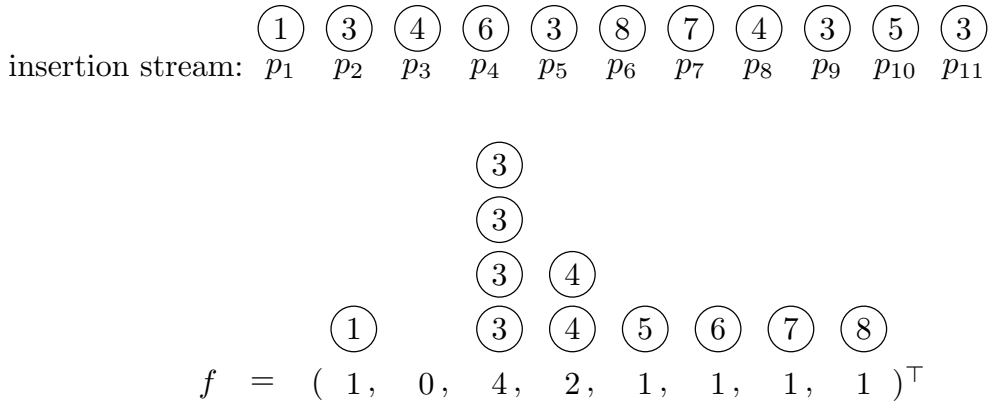


Figure 2.1: *Illustration of an insertion stream $p_1, \ldots, p_m$ and its frequency vector f. In this example the universe size $n = 8$ and the length of the stream $m = 11$.*

2.1 Introduction

2.1.1 Problem formulation

In an insertion stream $p_1, \ldots, p_m \in [n]$, the notion of *heavy hitters* or *frequent items* refers to those i such that its frequency f_i exceeds a threshold τ. By letting $\tau = \varepsilon \|f\|_p$ where f is the frequency vector, we can guarantee that there are at most ε^{-p} heavy hitters. In that case, we call them (ε, ℓ_p)-heavy hitters, and most commonly $p = 1$ or $p = 2$. We give a formal definition of (ε, ℓ_p)-heavy hitters as follows.

Definition 2.1.1 ((ε, ℓ_p)-heavy hitters in an insertion stream)**.** *Let $p > 0$. Let $p_1, \ldots, p_m \in [n]$ be an insertion stream, and for any $i \in [n]$ let $f_i = |\{t \in [m] : p_t = i\}|$. Namely, $f \in \mathbb{N}^n$ is the frequency vector for the stream. For any $i \in [n]$, we say i is an (ε, ℓ_p)-heavy hitter if $f_i \geq \varepsilon \|f\|_p$.*

The task of finding (ε, ℓ_p)-heavy hitters is to output a list containing all (ε, ℓ_p)-heavy hitters but not containing any element that is not $(\frac{\varepsilon}{2}, \ell_p)$-heavy hitter.

Problem 2.1.2 ((ε, ℓ_p)-heavy hitter problem in insertion streams)**.** *Let $p > 0$ and $\varepsilon \in (0, 1)$. For an insertion stream, the task of (ε, ℓ_p)-heavy hitter problem is to output $L \subseteq [n]$ such that*

- *For any $i \in [n]$, if i is an (ε, ℓ_p)-heavy hitter, then $i \in L$;*
- *For any $i \in L$, i is a $(\frac{\varepsilon}{2}, \ell_p)$-heavy hitter.*

For example, in the insertion stream illustrated in Figure 2.1, item 3 is a $(\frac{1}{2}, \ell_2)$-heavy hitter, because $\|f\|_2 = 5$ and $f_3 = 4 \geq \frac{1}{2}\|f\|_2$. Also note that item 4 is not a $(\frac{1}{2}, \ell_2)$-heavy hitter, but it is a $(\frac{1}{4}, \ell_2)$-heavy hitter. Thus in this example, $(\frac{1}{2}, \ell_2)$-heavy hitter problem requires to output either $\{3\}$ or $\{3, 4\}$.

Most heavy hitter algorithms not only give the list of heavy hitters but also give an estimate to the frequency of each heavy hitters, as observed by [BCIS09]. The guarantee of an estimate is measured as follows.

Definition 2.1.3 ((ε, k, ℓ_p)-tail guarantee)**.** *Let $f \in \mathbb{R}^n$, and $\varepsilon, k \geq 0$. For any $i \in [n]$, we say that $\widetilde{f_i}$ satisfies (ε, k, ℓ_p)-tail guarantee if $|\widetilde{f_i} - f_i| \leq \varepsilon \cdot \|f_{tail(k)}\|_p$, where $f_{tail(k)}$ refers to the vector f with the largest k entries replaced by zeros.*

The definition of heavy hitters can be generalized to turnstile streams, where f_i can be either positive or negative, and in that case item $i \in [n]$ is an (ε, ℓ_p)-heavy hitter if $|f_i| \geq \varepsilon \|f\|_p$.

2.1.2 Related works

We summarize some related works on heavy hitters in Table 2.1.

The earliest heavy hitter algorithm MJRTY [BM81] identifies the majority item in an insertion stream where its frequency is strictly greater than half of $\|f\|_1$. MJRTY maintains an active item i and a counter c recording the number of i's appearances minus appearances of the rest since i becomes active. In the beginning, there is no active item. If there is no active item, the newly arrived item becomes the active item, and the counter is assigned to 1; otherwise, if the newly arrived item equals to the active item, the counter is increased by 1, if the newly arrived item does not equal to the active item, the counter is decreased by 1; whenever the counter becomes 0, i will no longer be active. If there is a majority item whose frequency is greater than $\|f\|_1$, then by a pairing argument, in the end, the active item will be the majority item.

Algorithm MisraGries [MG82] generalizes MJRTY to arbitrary ε, and finds all (ε, ℓ_1)-heavy hitters in insertion stream with $O(1/\varepsilon)$ words of space, which is optimal because there can be as many as $\lfloor 1/\varepsilon \rfloor$ (ε, ℓ_1)-heavy hitters. MisraGries maintains a list of up to $\Theta(1/\varepsilon)$ active items each associate with a counter. When the active item list reaches its capacity and an inactive item arrives,

Algorithm	Reference	p	Model	Space (words)	D/R	Comment
MJRTY	[BM81, BM91]	1	I	$O(1)$	D	require $\varepsilon > 0.5$
MisraGries	[MG82]	1	I	$O(1/\varepsilon)$	D	
Frequent	[DLM02, KSP03]	1	I	$O(1/\varepsilon)$	D	similar to MisraGries [a]
StickySampling	[MM02]	1	I	$O(\varepsilon^{-1}\log\varepsilon^{-1})$	R	
LossyCounting	[MM02]	1	I	$O(\varepsilon^{-1}\log(\varepsilon\|f\|_1))$	D	
SpaceSaving	[MAE05]	1	I	$O(1/\varepsilon)$	D	
GK	[GK01]	1	I	$O(\varepsilon^{-1}\log(\varepsilon\|f\|_1))$	D	quantile algorithms
QDigest	[SBAS04]	1	I	$O(\varepsilon^{-1}\log\|\mathcal{U}\|)$	D	
KLL	[KLL16]	1	I	$O(\varepsilon^{-1}\log\log\varepsilon^{-1})$	R	
Dyadic-CountMin	[CM05]	1	ST	$O(\varepsilon^{-1}\log^2 n)$	R	fast update & query [b]
CountMin	[CM05]	1	ST	$O(\varepsilon^{-1}\log n)$	R	update time $\log n$ query time $n\log n$
CountSketch	[CCF02]	2	T	$O(\varepsilon^{-2}\log n)$	R	
	[JST11]	$(0,2]$	T	$O(\varepsilon^{-p}\log n)$	R	
	[JST11]	$(0,2]$	ST	$\Omega(\varepsilon^{-p}\log n)$	R	hold for $\delta = 1/3$
F_pHH	[KNPW11]	$(0,2]$	T	$O(\varepsilon^{-p}\log^2 n)$	R	update time $\log^2 n$ query time $\varepsilon^{-p}\log^2 n$
CountSketch-with-codes	[Pag13]	$(0,2]$	T	$O(\varepsilon^{-p}\log^2 n)$	R	
	[LW13]	$(2,+\infty)$	T	$\Omega(\varepsilon^{-2}n^{1-2/p})$	R	
TP-Estimator [c]	[Gan15]	$(2,+\infty)$	T	$O(\varepsilon^{-2}n^{1-2/p})$	R	$\varepsilon < 1/\log^{O(1)} n$
Pick-and-Drop	[BO13]	$[3,+\infty)$	I	$O(n^{1-2/p})$	R	constant ε
	[CKS03]	$(2,+\infty)$	I	$\Omega(\frac{n^{1-2/p}}{\log n})$	R	
BKSV	[BKSV14]	$(3,+\infty)$	I	$O(\frac{\varepsilon^{-C}n^{1-2/p}}{\log n})$	R	absolute constant C
CountSieve	[BCIW16]	2	I	$O(\frac{\log\varepsilon^{-1}}{\varepsilon^2}\log\log n)$	R	
ExpanderSketch	[LNNT16]	$(0,2]$	T	$O(\varepsilon^{-p}\log n)$	R	fast update & query [d]
BPTree (**this thesis**)	[BCI+17]	2	I	$O(\varepsilon^{-2}\log\varepsilon^{-1})$	R	
BDW	[BDW18]	1	I	$O(\frac{1}{\varepsilon}+\frac{\log_n\varepsilon^{-1}}{\varepsilon'})$	R	stronger guarantee [e]

Table 2.1: *Some results on* 1*-pass streaming algorithms (and space lower bounds) for heavy hitters. Space is measured in words, where each word has* $\Theta(\log n)$ *bits. Unless explicitly specified, we assume that both* $|\mathcal{U}|$ *(universe size) and* $\|f\|_1$ *are* $O(\mathrm{poly}(n))$*. In the "model" column, "I" refers to insertion stream, "T" refers to general turnstile stream, and "ST" refers to strict turnstile stream. In the "D/R" column, "D" refers to deterministic, and "R" refers to randomized. (a) [DLM02, KSP03] essentially rediscovered* MisraGries*, and the algorithm also becomes known as* Frequent*. (b) update time* $O(\log^2 n)$*, query time* $O(\varepsilon^{-1}\log^2 n)$*. (c) abbreviation of "Taylor polynomial estimator". (d) update time* $O(\log n)$*, query time* $\varepsilon^{-p}\mathrm{poly}(\log n)$*. (e)* $\varepsilon' < \varepsilon$*. The algorithm outputs all* i *such that* $f_i \geq \varepsilon\|f\|_1$*, no* i *such that* $f_i < (\varepsilon-\varepsilon')\|f\|_1$*. Moreover, it estimates the frequency of all the returned heavy hitters with error at most* $\varepsilon'\|f\|_1$*.*

every counter is decreased by 1. A similar algorithm was later rediscovered by [DLM02] and [KSP03] separately, and becomes known as Frequent. Other counter-based approaches for (ε, ℓ_1)-heavy hitters in insertion streams including LossyCounting [MM02] and SpaceSaving [MAE05] achieve a similar performance.

Another type of approach for finding ℓ_1 heavy hitters in insertion streams is based on quantile algorithms. In an insertion stream, let $rank(j) = \sum_{i<j} f_i$, an ε-approximate ϕ-th quantile (where $\phi \in [0,1]$) requires to find j such that $rank(j) \leq (\phi+\varepsilon)\|f\|_1$ and $rank(j+1) \geq \phi\|f\|_1$. A quantile algorithm that answers ε-approximate ϕ-quantile for all $\phi \in \{\varepsilon q : q \in \{0, \ldots, \lfloor \varepsilon^{-1} \rfloor\}\}$ can be used to obtain $(4\varepsilon, \ell_1)$-heavy hitters. GK [GK01] and QDigest [SBAS04] are two popular deterministic quantile algorithms with space complexity $O(\varepsilon^{-1}\log(\varepsilon\|f\|_1))$ and $O(\varepsilon^{-1}\log|\mathcal{U}|)$ words respectively. [KLL16] propose a quantile algorithm using $O(\varepsilon^{-1}\log\log\delta^{-1})$ words of space to answer a quantile query with failure probability δ. It is sufficient to let $\delta = O(\varepsilon)$ to make it an ℓ_1 heavy hitter algorithm with constant failure probability.

CountSketch [CCF02, CCF04] is a sketch-based algorithm that finds (ε, ℓ_2)-heavy hitters in a turnstile stream using $O(\varepsilon^{-2}\log n)$ words of space with $(\varepsilon, 1/\varepsilon^2, \ell_2)$-tail guarantee. It has two parameters B and R, which are referred as "bucket size" and "number of repetitions". In each repetition $r \in [R]$, CountSketch is specified by a pairwise hash function $h_r : [n] \to [B]$ and a pairwise random sign function $\sigma_r : [n] \to \{-1, 1\}$, and for each $b \in [B]$, $y_{r,b} := \sum_{i:h_r(i)=b} \sigma_r(i) f_i$ is maintained (or measured in the context of compressed sensing). We illustrate CountSketch in Figure 2.2. Let $B = O(1/\varepsilon^2)$, we can prove that with constant probability $|\sigma_r(i) y_{r,h_r(i)} - f_i| = O(\varepsilon\|x_{tail(1/\varepsilon^2)}\|_2)$. Let $R = O(\log n)$ and take $\widetilde{f_i} = \text{median}_{r\in[R]}\{\sigma_r(i) y_{r,h_r(i)}\}$ as estimation of f_i for each $i \in [n]$. It is guaranteed that with probability $1 - 1/\text{poly}(n)$ every estimate has an error of at most $O(\varepsilon\|x_{tail(1/\varepsilon^2)}\|_2)$.

Also as a sketch-based algorithm, CountMin [CM05] finds (ε, ℓ_1)-heavy hitters in strict turnstile streams with $O(\varepsilon^{-1}\log n)$ words of space. Compared to CountSketch, the number of "buckets" used by CountMin is $O(\varepsilon^{-1})$ instead of $O(\varepsilon^{-2})$, and CountMin drops the random signs by maintaining $y_{r,b} := \sum_{i:h_r(i)=b} f_i$ and takes eastimate to be $\widetilde{f_i} = \min_{r\in[R]}\{y_{r,h_r(i)}\}$. CountMin has $(\varepsilon, 1/\varepsilon, \ell_1)$-tail guarantee.

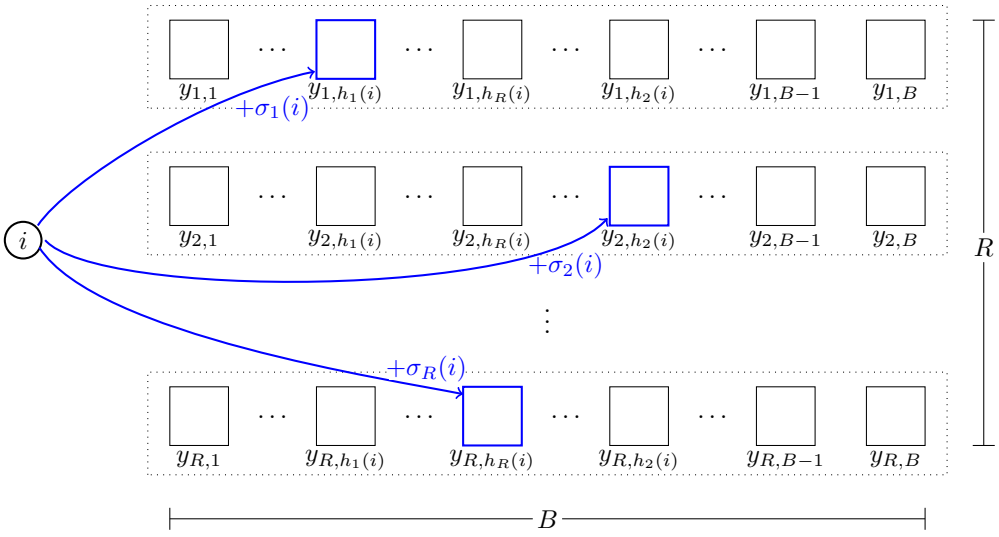


Figure 2.2: *Illustration of CountSketch. A CountSketch data structure has R rows (or repetitions) and each row $r \in [R]$ has B buckets, where bucket $b \in [B]$ maintains a "count sketch" $y_{r,b}$. For an insertion stream, when item i arrives (or equivalently, we receive an update of i), CountSketch will update $y_{r,h_r(i)} \leftarrow y_{r,h_r(i)} + \sigma_r(i)$ for every $r \in [R]$ (as illustrated in blue). For general turnstile update $f_i \leftarrow f_i + \Delta$, CountSketch will update $y_{r,h_r(i)} \leftarrow y_{r,h_r(i)} + \Delta \cdot \sigma_r(i)$ for every $r \in [R]$. Usually we set $R = \Theta(\log n)$ and $B = \Theta(\varepsilon^{-2})$.*

Given a constant estimation of $\|f\|_p$ (which has been extensively studied since [AMS96]), CountSketch can be used to find ℓ_p heavy hitters for any $p > 0$ in turnstile streams. For $p \in (0, 2]$, by setting bucket size $B = O(\varepsilon^{-p})$, CountSketch solves (ε, ℓ_p)-heavy hitter problem with $O(\varepsilon^{-p} \log n)$ words of space. This is first observed by [JST11] by showing $\frac{1}{\sqrt{k}}\|f_{tail(k)}\|_2 \le \frac{1}{k^{1/p}}\|f\|_p$ and choosing $k = \varepsilon^{-p}$. [JST11] also gave a matching space lower bound of $\Omega(\varepsilon^{-p} \log n)$ words for turnstile streams. The lower bound even holds for strict turnstile streams. Intuitively, finding all (ε, ℓ_p)-heavy hitters in a strict turnstile stream requires $\Omega(\varepsilon^{-p} \log n)$ words because there can be as many as $\Theta(\varepsilon^{-p})$ (ε, ℓ_p)-heavy hitters, and if a heavy hitter algorithm can identify all of them, then we can remove all the heavy hitters and subsequently identify heavy hitters for the remaining stream (we can set their frequencies to be one quarter of the previously removed heavy hitters), and this process can be repeated for $\Theta(\log n)$ times, so that overall the heavy hitter algorithm can be used to encode $\Theta(\varepsilon^{-p} \log n)$ items. For $p \in (2, \infty)$, by Hölder's inequality $\frac{\varepsilon}{n^{1/2-1/p}}\|f\|_2 \le \varepsilon\|f\|_p$. By setting bucket size $B = \varepsilon^{-2} n^{1-2/p}$, CountSketch takses $O(\varepsilon^{-2} n^{1-2/p} \log n)$ words of space and gives point estimation for each $i \in [n]$ with good enough precision to identify all the (ε, ℓ_p)-heavy

hitters (given a good estimation of $\|f\|_p$ as the threshold).

For $p > 2$, the best space upper bound for finding (ε, ℓ_p)-heavy hitters in turnstile streams is $O(\varepsilon^{-2} n^{1-2/p})$ words for $\varepsilon < 1/\log^{O(1)} n$ in [Gan15], which matches the lower bound of $\Omega(\varepsilon^{-2} n^{1-2/p})$ words proved in [LW13]. Note that both papers focus on estimating $\|f\|_p$ within a multiplicative factor of ε in turnstile streams for $p > 2$, but their results imply space bounds for finding heavy hitters, since en route to estimating $\|f\|_p$ [Gan15] first finds (ε, ℓ_p)-heavy hitters, and the communication problem studied by [LW13] also implies heavy hitter lower bound.

It worth pointing out that $p = 2$ is a sharp threshold in the space complexity for finding (ε, ℓ_p)-heavy hitters. For $p < 2$, it takes poly-logarithmic space, while for $p > 2$, it takes polynomial space. This is also true for insertion streams. For $p > 2$, the optimal space lower bound for finding $(\frac{1}{2}, \ell_p)$-heavy hitters in insertion stream is $\Omega(n^{1-2/p})$ bits by [CKS03], obtained by improving the communication complexity of multi-party set disjointness first proposed and studied in [AMS96]. A matching upper bound of $O(n^{1-2/p})$ bits is given by [BKSV14] for $p > 3$ and constant ε, which is obtained by storing the IDs of "samples" of Pick-and-Drop [BO13] in $O(1)$ amortized bits.

For $p = 2$, CountSieve [BCIW16] uses $O(\varepsilon^2 \log \varepsilon^{-1} \log \log n)$ words of space to find ℓ_2 heavy hitters in insertion streams. In the single heavy hitter case, CountSieve uses Bernoulli processes to extract information of the heavy hitter, and maintains $\Theta(\log \log n)$ Bernoulli processes *in parallel* to filter out the noise so that the heavy hitter is $\Omega(\operatorname{poly} \log n)$-heavy (we say i is α-heavy if $f_i \geq \alpha \sqrt{\sum_{j \neq i} f_j^2}$). Once the heavy hitter is $\Omega(\operatorname{poly} \log n)$-heavy, CountSieve can maintain $\Theta(\log n)$ Bernoulli processes *sequentially* (so that at any time there is only one active Bernoulli process) and each Bernoulli process approximately extracts one-bit information of the heavy hitter.

So far we have been focusing on the space complexity of finding heavy hitters, but update and query time are also important. The main drawback of CountSketch and CountMin is that the query time is $O(n \log n)$. In the paper [CM05] that introduces CountMin, the dyadic trick is used to bring down the query time to $O(\varepsilon^{-1} \log^2 n)$, while both its space and update time are worse by a factor of $O(\log n)$. [KNPW11] extended the dyadic trick result to $p \in (0, 2]$ for turnstile streams with fast update and query time, but space is a $\log n$ factor worse than the optimal. [Pag13] obtained a similar guarantee using CountSketch with error-correcting codes. [LNNT16] showed that the

trade-off between space and query time is not necessary by proposing ExpanderSketch with optimal space and fast update & query for finding (ε, ℓ_p)-heavy hitters in turnstile streams where $p \in (0, 2]$. In terms of time lower bounds, [LNN15] showed that any "non-adaptive" turnstile algorithm using $O(\text{poly}\log(n))$ space must have $\Omega(\sqrt{\log n / \log\log n})$ update time, where "non-adaptive" captures almost all existing turnstile algorithms.

In the formulation of the heavy hitter problem, we can add an additional parameter $\varepsilon' < \varepsilon$ and require to output all (ε, ℓ_p)-heavy hitters and all the items in the output must be $(\varepsilon - \varepsilon', \ell_p)$-heavy hitters. [BDW18] gave a space optimal algorithm for finding ℓ_1 heavy hitters in insertion streams for this stronger guarantee.

2.1.3 Summary of our main results

We design BPTree [BCI+17] using $O(\varepsilon^{-2}\log\varepsilon^{-1})$ words of space to find (ε, ℓ_2)-heavy hitters in an insertion stream, which is optimal for constant ε. Its guarantee is formulated in the following theorem.

Theorem 2.1.4 (restatement of Theorem 2.7.2)**.** *For any $\varepsilon > 0$ there is 1-pass streaming algorithm BPTree that takes an insertion stream as input, and with probability at least $(1-\delta)$ returns a set of $(\frac{\varepsilon}{2}, \ell_2)$-heavy hitters containing every (ε, ℓ_2)-heavy hitter and an approximate frequency for every item returned satisfying the $(\varepsilon, 1/\varepsilon^2, \ell_2)$-tail guarantee. The algorithm uses $O(\frac{1}{\varepsilon^2}\log\frac{1}{\varepsilon\delta})$ words of space and has $O(\log\frac{1}{\varepsilon\delta})$ update and $O(\varepsilon^{-2}\log\frac{1}{\varepsilon\delta})$ retrieval time.*

Bernoulli process is a crucial building block in both CountSieve and BPTree. We prove that using 4-wise independent random signs will still make the supremum bounded. We illustrate the supremum of a Bernoulli process in Figure 2.3.

Theorem 2.1.5 (restatement of Theorem 2.3.1)**.** *Let $p_1, \ldots, p_m \in [n]$ be an insertion stream. For any $t \in \{0, \cdots, m\}$ and any $i \in [n]$, let $f_i^{(t)} = |\{j \in [t] : p_j = i\}|$. If $Z \in \{-1, 1\}^n$ is drawn from a 4-wise independent family, $\mathbb{E}\sup_t |\langle f^{(t)}, Z\rangle| < 23 \cdot \|f^{(m)}\|_2$.*

As a component of BPTree, we design an algorithm to track at any time the second moment of the frequency vector with an error up to εF_2 where F_2 is the second moment at the end of the

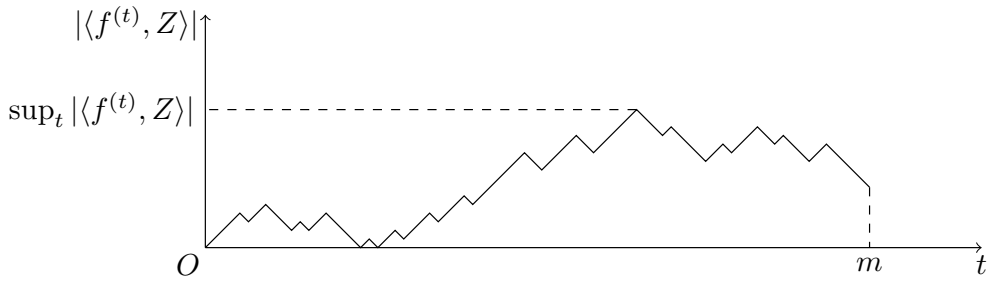


Figure 2.3: *The supremum of a Bernoulli process.*

stream. Note that for our BPTree, we only need ε to be a small constant.

Theorem 2.1.6 (restatement of Theorem 2.6.1)**.** *Let $p_1, \ldots, p_m \in [n]$ be an insertion stream. For any $t \in \{0, \cdots, m\}$ and any $i \in [n]$, let $f_i^{(t)} = |\{j \in [t] : p_j = i\}|$. for any $t \in \{0, \ldots, m\}$, let $F_2^{(t)} = \sum_{i\in[n]} \left(f_i^{(t)}\right)^2$. Let $F_2 = F_2^{(m)}$. Let $0 < \varepsilon, \delta < 1$. There is a 1-pass streaming algorithm that outputs at each time t a value $\widehat{F}_2^{(t)}$ such that $\mathbb{P}(|\widehat{F}_2^{(t)} - F_2^{(t)}| \le \varepsilon F_2, \forall t : 0 \le t \le m) \ge 1 - \delta$. The algorithm use $O(\frac{1}{\varepsilon^2} \log \frac{1}{\delta})$ words of space and has $O(\frac{1}{\varepsilon^2} \log \frac{1}{\delta})$ update time.*

2.1.4 Our approach

In the remainder of this introduction, we will go through some of the key technical insights of our BPTree algorithm, and highlight its contributions by comparing it with previous CountSketch and CountSieve approaches.

Similar to CountSketch and CountSieve, our BPTree first reduces to the problem of identifying the single heavy hitter H in a stream with the guarantee that H is very heavy — namely $f_H \ge K\|f_{[n]\setminus\{H\}}\|_2$ (in that case, we say H is K-heavy) where K is a sufficiently large constant. Any solution that takes $O(s)$ words of space and identifies the single heavy hitter H with constant success probability will lead to a solution of finding all the heavy hitters using $O(s\varepsilon^{-2} \log \varepsilon^{-1})$ words of space.

The reduction follows two simple observations. First, we can use CountSketch to decide whether an item is a heavy hitter. In particular, if we can find a set of heavy hitter candidates $\widehat{\mathcal{H}}$ that contains all the heavy hitters, then it takes CountSketch $O(\varepsilon^{-2} \log \|\widehat{\mathcal{H}}\|)$ words (we maintain CountSketch in parallel to the subroutine that obtains $\widehat{\mathcal{H}}$) to give a good estimation (with additive

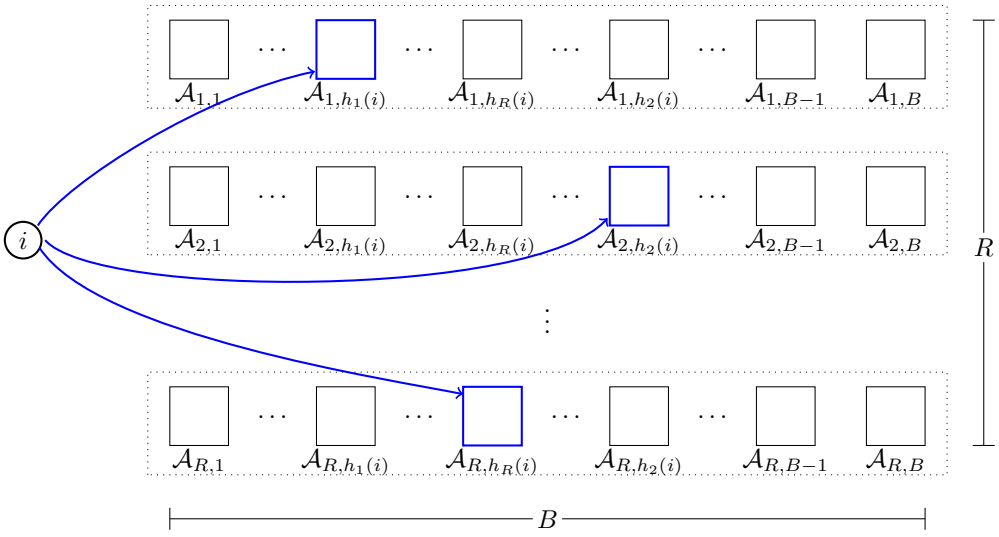


Figure 2.4: *Illustration of the reduction from the heavy hitter problem to identifying a single heavy hitter. We have $R = \Theta(\log \varepsilon^{-1})$ and $B = \Theta(\varepsilon^{-2})$. Each box $\mathcal{A}_{r,b}$ where $r \in [R]$ and $b \in [B]$ represents an independent instance of the streaming algorithm that identifies the single heavy hitter. When item i arrives, for every $r \in [R]$, $\mathcal{A}_{r,h_r(i)}$ (i.e. the $h_r(i)$-th sub-stream/bucket) receives the update, where $h_r : [n] \to [B]$ is a pairwise hash function.*

error $\frac{\varepsilon}{4}\|f\|_2$) to each f_j for $j \in \widehat{\mathcal{H}}$. If we simply choose $\widehat{\mathcal{H}}$ to be $[n]$, CountSketch takes $O(\varepsilon^{-2} \log n)$ words of space to solve the heavy hitter problem. In order to beat CountSketch, BPTree, like CountSieve, makes $|\widehat{\mathcal{H}}| = O(\varepsilon^{-2} \log \varepsilon^{-1})$. Second, given an algorithm $\mathcal{A}$ that solves the single heavy hitter case, we can obtain such heavy hitter candidates $\widehat{\mathcal{H}}$ of small size by sub-streaming and repetition. We illustrate the process in Figure 2.4, which is similar to CountSketch, but instead of maintaining a (signed) counter for each sub-stream, we maintain an instance of the streaming algorithm that solves the single heavy hitter case. We first discuss one repetition. Let $B = O(\frac{K^2}{\varepsilon^2})$. We partition the original stream $p_1, \ldots, p_m$ into B disjoint sub-streams by choosing a pairwise independent hash function $h : [n] \to [B]$ and item i belongs to the $h(i)$-th sub-stream. For each heavy hitter H, we can prove that with constant probability, H is at least K-heavy in the sub-stream it resides in. If we have an algorithm that takes $O(s)$ words of space and identifies H with constant success probability, then we can make sure that H is chosen as a heavy hitter candidate with constant probability. If we repeat the above sub-streaming procedure $R = O(\log \varepsilon^{-1})$ times with different random seeds, we can make sure that with good probability, all the heavy hitters will be chosen as candidates. The space required by finding $\widehat{\mathcal{H}}$ is $O(s\varepsilon^{-2} \log \varepsilon^{-1})$ words.

Now, the task is to identify the heavy hitter H which is K-heavy for sufficiently large constant K. It is useful to see other items in the stream as noise, and f_H is at least K times the ℓ_2 norm of the noise. CountSieve introduces an important building block that extracts one bit of information about H. The one-bit subroutine randomly partitions $[n]$ into S_0 and S_1 (via a pairwise hash function), and maintains $X_0^{(t)} = \sum_{i \in S_0} Z_i f_i^{(t)}$ and $X_1^{(t)} = \sum_{i \in S_1} Z_i f_i^{(t)}$, where $Z \in \{-1, 1\}^n$ is a Rademacher random vector, and $f^{(t)}$ refers to the frequency vector at time t. Then H is more likely to appear in the S_b with larger $|X_b^{(m)}|$. Let t_0 so that $f_H^{(t_0)} = \frac{1}{10} f_H^{(m)}$. Let $H \in S_b$ where $b \in \{0, 1\}$. Moreover, we can prove that the probability is at least 0.9 (for large enough constant K) that $|X_b^{(t)}| > |X_{1-b}^{(t)}|$ for all $t \in [t_0, m]$. Namely, with good probability, after consuming a constant fraction of H in the stream, $\arg\max_{b \in \{0,1\}} |X_b^{(t)}|$ will always tell whether H is in S_0 or in S_1.

Note that a Rademacher random vector $Z \in \{-1, 1\}^n$ requires n bits of space, which is impractical for streaming algorithms. CountSieve addresses this issue by first applying a Johnson-Lindenstrauss (JL) transformation of dimension $O(\log n)$ to the frequency vector, so that the required dimension of the Rademacher random vector to inner-product with is $O(\log n)$. However, the JL transformation itself requires $O(\log n \log \log n)$ bits of space to store. In contrast, our BPTree uses 4-wise independent hash functions to generate Z, and a chaining method to show that the expected supremum of the Bernoulli process is still bounded in a desirable way.

CountSieve uses the one-bit subroutine in a two-part strategy to identify the ℓ_2 heavy hitter with $O(\log \log n)$ memory. The two parts are: (1) Amplify the heavy hitter so that H is $\text{poly}(\log n)$-heavy; (2) Identify H with independent repetitions of the one-bit subroutine. Part (1) winnows the stream from, potentially, n distinct elements to at most $n/\text{poly}(\log n)$ elements. The heavy hitter remains and we get $f_H \geq (1 - \frac{1}{\text{poly}(\log n)})\|f\|_2$ because many of the other elements have been removed. CountSieve accomplishes this by running $\Theta(\log \log n)$ independent copies of the one-bit subroutine in parallel, and discarding elements that do not pass a super-majority of the tests. A standard Chernoff bound implies that only $n/2^{O(\log \log n)} = n/\text{poly}(\log n)$ items survive. Part (2) of the strategy identifies $\Theta(\log n)$ "break-points" where $\|f\|_2$ of the winnowed stream increases by approximately a $(1 + 1/\log n)$ factor from one break-point to the next. Because H already accounts for nearly all of the value of $\|f\|_2$ it is still a heavy hitter within each of the $\Theta(\log n)$

intervals. CountSieve learns one bit of the identity of H on each interval by running the one-bit subroutine. After all $\Theta(\log n)$ intervals have completed, the identity of H is known.

BPTree merges the two parts of the above strategy. Similar to CountSieve, BPTree runs a series of $\Theta(\log n)$ rounds where the goal of each round is to learn one bit of the identity of H. As compared to CountSieve, BPTree discards more items after each round, then recurses on learning the remaining bits. As the algorithm proceeds, it discards more and more items and H becomes heavier and heavier in the stream. This is reminiscent of a work on adaptive compressed sensing [IPW11], but here we are able to run all the rounds in a single pass given the insertion-only property of the stream.

2.2 Preliminaries

An insertion stream is a list of items $p_1, \ldots, p_m \in [n]$. The frequency of j at time t is $f_j^{(t)} := |\{i \leq t \mid p_i = j\}|$, and $f^{(t)} \in \mathbb{N}^n$ is called the *frequency vector*. We denote $f := f^{(m)}$, $F_2^{(t)} := \sum_{j=1}^n (f_j^{(t)})^2$, $F_2 := \sum_{j=1}^n f_j^2$, and $F_0 := |\{j \in [n] : f_j > 0\}|$. An item $H \in [n]$ is α-heavy if $f_H^2 \geq \alpha^2 \sum_{j \neq H} f_j^2 = \alpha^2(F_2 - f_H^2)$. In the example illustrated in Figure 2.1, item 3 is $\frac{4}{3}$-heavy, because $f_3 = 4$ and $\sum_{j \neq 3} f_j^2 = 9$.

For $W \subseteq [n]$, denote by $f^{(t)}(W) \in \mathbb{N}^n$ the frequency vector at time t of the stream restricted to the items in W, That is, a copy of $f^{(t)}$ with the ith coordinate replaced by 0 for every $i \notin W$. We also define $f^{(s:t)}(W) := f^{(t)}(W) - f^{(s)}(W)$ and $F_2(W) := \sum_{j \in W} f_j^2$.

Our algorithms make use of 2-universal (pairwise independent), 4-wise independent, and 8-wise independent hash functions. We will denote such a function $h : [n] \to [p]$ where p is a prime larger than n, or we may use $h : [n] \to \{0,1\}^R$, which may be taken to mean a function of the first type for some prime $p \in [2^{R-1}, 2^R)$. We use $h(x)_i$ to denote the ith bit, with the first bit most significant. A crucial step in our algorithm involves comparing the bits of two values $a, b \in [p]$. Notice that, for any $0 \leq r \leq \lceil \log_2 p \rceil$, we have $a_i = b_i$, for all $1 \leq i \leq r$, if and only if $|a - b| < 2^{\lceil \log_2 p \rceil - r}$. Therefore, the test $a_i = b_i$, for all $1 \leq i \leq r$, can be performed with a constant number of operations.

We will use CountSketch [CCF02] as a subroutine, and also compare our algorithm against

it. CountSketch has two parameters B and R, which determine the number of "buckets" and "repetitions" (or "rows") in the table it stores, respectively.

Definition 2.2.1 (CountSketch)**.** *A CountSketch data structure with bucket size B and repetitions R corresponds to a measurement matrix $\Phi \in R^{BR\times n}$. Matrix Φ is a concatenation of $\Phi^{(1)}, \ldots, \Phi^{(R)}$, where each $\Phi^{(r)} \in \mathbb{R}^{B\times n}$ ($r \in [R]$) is defined by a pairwise hash function $h_i : [n] \to [B]$ and a pairwise sign function $\sigma_i : [n] \to \{-1, 1\}$, such that for each $j \in [B]$ and $i \in [n]$,*

$$\Phi^{(r)}_{j,i} = \begin{cases} \sigma_r(i) & \text{if } h_r(i) = j, \\ 0 & \text{otherwise.} \end{cases}$$

The data structure maintains the value of Φx and for each $i \in [n]$, it gives

$$x'_i = \mathit{median}\left(\sigma_1(i) \cdot (\Phi^{(1)}x)_{h_1(i)}, \ldots, \sigma_R(i) \cdot (\Phi^{(R)}x)_{h_R(i)}\right)$$

as estimation to x_i.

Let $x' \in \mathbb{R}^n$ denote the approximation given by the count sketch data structure. We have the following guarantee for count-sketch with bucket size B and repetitions R.

Lemma 2.2.2 ([CCF02])**.** *Let $x \in \mathbb{R}^n$, and let $x' \in \mathbb{R}^n$ be the output of a CountSketch data structure (Definition 2.2.1) with bucket size B and repetitions R. For each $i \in [n]$, with probability at least $1 - 2^{-\Omega(R)}$,*

$$|x_i - {x'}_i| \leq \frac{2}{\sqrt{B}}\|x\|_2.$$

In our algorithm we use the notation $\mathbf{1}(A)$ denote the indicator function of the event A. Namely, $\mathbf{1}(A) = 1$ if A is true and 0 otherwise. We use $x \lesssim y$ to denote $x = O(y)$.

2.3 Overview of the Algorithm

Our heavy hitter algorithm consists of several subroutines. The most important subroutine is HH1 in Algorithm 1, which finds a single $O(1)$-heavy hitter assuming we have an estimate σ of $\sqrt{F_2}$ such that $\sqrt{F_2} \leq \sigma \leq 2\sqrt{F_2}$. HH2 (Algorithm 2) in Section 2.7 removes the assumption on σ by repeatedly "guessing" values for σ and restarting HH1 as more items arrive. The F_2 guessing in

HH2 is where we need F_2 tracking. Finally, a standard reduction from finding ε-heavy hitters to finding a single $O(1)$-heavy hitter leads us to the main heavy hitters algorithm BPTree, which is formally described in Theorem 2.7.2.

Algorithm 1 Identify a heavy hitter.

procedure HH1(σ, p_1, $p_2, \ldots, p_m$) ▷ $\sqrt{F_2} \le \sigma \le 2\sqrt{F_2}$
  $R \leftarrow 3\lfloor \log_2(\min\{n, \sigma^2\} + 1) \rfloor$
  Initialize $b = b_1 b_2 \cdots b_R = 0 \in [2^R]$
  Sample $h : [n] \to \{0,1\}^R \sim$ 2-wise independent family
  Sample $Z \in \{-1, 1\}^n$ with 4-wise independence
  $X_0, X_1 \leftarrow 0$
  $r \leftarrow 1$, $H \leftarrow -1$
  $\beta \leftarrow 3/4$, $c \leftarrow 1/32$
  for $t = 1, 2, \ldots, m$ and $r < R$ **do**
    if $h(p_t)_i = b_i$, for all $i \in [r-1]$ **then**
      $H \leftarrow p_t$
      $X_{h(p_t)_r} \leftarrow X_{h(p_t)_r} + Z_{p_t}$
      if $|X_0 + X_1| \ge c\sigma\beta^r$ **then**
        Record one bit $b_r \leftarrow \mathbf{1}(|X_1| > |X_0|)$
        Refresh $(Z_i)_{i=1}^n$, $X_0, X_1 \leftarrow 0$
        $r \leftarrow r + 1$
      end if
    end if
  end for
  return H
end procedure

HH1 in Algorithm 1 begins with randomizing the item labels by replacing them with pairwise independent values on $R = \Theta(\log \min\{n, \sigma^2\})$ bits via the hash function h. Since n and $\sigma^2 \ge F_2$ are both upper bounds for the number of distinct items in the stream, R is chosen so that every item receives a distinct hash value.

Once the labels are randomized, HH1 proceeds in rounds wherein one bit of the randomized label of the heavy hitter is determined during each round. It completes all of the rounds and outputs the heavy hitter's identity within one pass over the stream. As the rounds proceed, items are discarded from the stream. The remaining items are called *active*. When the algorithm discards an item it will never reconsider it (unless the algorithm is restarted). In each round, it creates two Bernoulli processes X_0 and X_1. In the rth round, X_0 will be determined by the active

items whose randomized labels have their rth bit equal to 0, and X_1 determined by those with rth bit 1. Let $g_0^{(t)}, g_1^{(t)} \in \mathbb{N}^n$ denote the frequency vectors of the active items in each category, respectively, initialized to 0 at the beginning of the round. Then we have Bernoulli processes that $X_0^{(t)} = \langle Z, g_0^{(t)} \rangle$ and $X_1^{(t)} = \langle Z, g_1^{(t)} \rangle$, where Z is a vector of 4-wise independent Rademacher random variables.

The rth round ends when $|X_0 + X_1| > c\sigma\beta^{r-1}$, for constants $c = 1/32$ and $\beta = 3/4$. At this point, the algorithm compares the values $|X_0|$ and $|X_1|$ and records the identity of the larger one as the rth bit of the candidate heavy hitter. All those items with rth bit corresponding to the smaller counter are discarded (made inactive), and the next round is started.

After R rounds are completed, if there is a heavy hitter then its randomized label will be known with good probability. The identity of the item can be determined by selecting an item in the stream that passes all of the R bit-wise tests, or by inverting the hash function used for the label. If it is a K-heavy hitter, for a sufficiently large $K = O(1)$, then the algorithm will find it with probability at least $2/3$.

The most important technical component of the analysis is the following Theorem 2.3.1, which is proved in Section 2.4. It gives us control of the evolution of $|X_0|$ and $|X_1|$ so we can be sure that the larger of the two identifies a bit of H.

Theorem 2.3.1. *If $Z \in \{-1, 1\}^n$ is drawn from a 4-wise independent family, $\mathbb{E} \sup_t |\langle f^{(t)}, Z \rangle| < 23 \cdot \|f^{(m)}\|_2$.*

We will use $C_* < 23$ to denote the optimal constant in this theorem.

The key idea behind the algorithm is that as we learn bits of the heavy hitter and discard other items, it becomes easier to learn additional bits of the heavy hitter's identity. With fewer items in the stream as the algorithm proceeds, the heavy hitter accounts for a larger and larger fraction of the remaining stream as time goes on. As the heavy hitter gets heavier the discovery of the bits of its identity can be sped up. When the stream does not contain a heavy hitter this acceleration of the rounds might not happen, though that is not a problem because when there is no heavy hitter the algorithm is not required to return any output. Early rounds will each use a constant fraction of the updates to the heavy hitter, but the algorithm will be able to finish all $R = \Theta(\log n)$

rounds because of the speed-up. The parameter β controls the speed-up of the rounds. Any value of $\beta \in (\frac{1}{2}, 1)$ can be made to work (possibly with an adjustment to c), but the precise value affects the heaviness requirement and the failure probability.

2.4 Bernoulli Process with 4-wise Independence

Let $Z \in \{-1, 1\}^n$ be a random vector. We are interested in upper bounding $\mathbb{E} \sup_t |\langle f^{(t)}, Z \rangle|$. It was shown in [BCIW16] that if each entry in Z is drawn independently and uniformly from $\{-1, 1\}$, then $\mathbb{E} \sup_t |\langle f^{(t)}, Z \rangle| \lesssim \|f^{(m)}\|_2$. In this section, we show that this inequality still holds if the entries of Z are drawn from a 4-wise independent family. The result is used both in our analysis of HH1 in Section 2.5 and our F_2 tracking algorithm in Section 2.6.

The following lemma is implied by [Haa82].

Lemma 2.4.1 (Khintchine's inequality)**.** *Let $Z \in \{-1, 1\}^n$ be chosen uniformly at random, and $x \in \mathbb{R}^n$ a fixed vector. Then for any even integer p, $\mathbb{E} \langle Z, x \rangle^p \leq \sqrt{p}^p \cdot \|x\|_2^p$.*

Note that Khintchine's inequality $\mathbb{E} \langle Z, x \rangle^p \leq \sqrt{p}^p \cdot \|x\|_2^p$ still holds if Z is drawn from a p-wise independent family, because $\mathbb{E} \langle Z, x \rangle^p$ will remain unchanged. Now we are ready to give the proof for Theorem 2.3.1.

Proof of Theorem 2.3.1. To simplify notation, we first normalize the vectors in $\{f^{(0)} = 0, f^{(1)}, \ldots, f^{(m)}\}$ (i.e., divide by $\|f^{(m)}\|_2$). Denote the set of these normalized vectors by $T = \{v_0, \ldots, v_m\}$, where $\|v_m\|_2 = 1$. Recall that an ε-net of a set of points T under a metric d is a set of points $T' \subseteq T$ such that for each $t \in T$, there exists $t' \in T'$ such that $d(t, t') \leq \varepsilon$. For every $k \in \mathbb{N}$, we can find a $1/2^k$-net of T in ℓ_2 with size $|T_k| \leq 2^{2k}$ by a greedy construction as follows.

To construct an ε-net for T, we first take v_0, then choose the smallest i such that $\|v_i - v_0\|_2 > \varepsilon$, and so on. To prove the number of elements selected is upper bounded by $1/\varepsilon^2$, let $u_0, u_1, u_2, \ldots, u_t$ denote the vectors we selected accordingly, and note that the second moments of $u_1 - u_0, u_2 - u_1, \ldots, u_t - u_{t-1}$ are greater than ε^2. Because the vectors $u_i - u_{i-1}$ have non-negative coordinates, $\|u_t\|_2^2$ is lower bounded by the summation of these moments, while on the other hand $\|u_t\|_2^2 \leq 1$. Hence the net is of size at most $1/\varepsilon^2$.

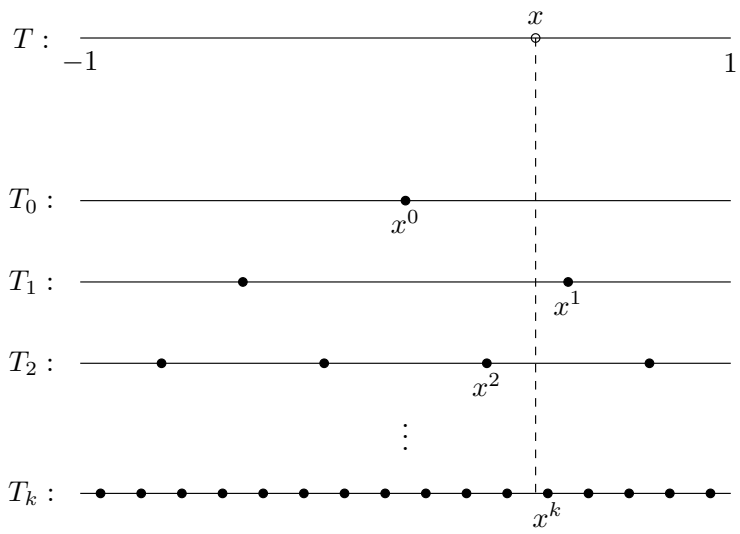


Figure 2.5: *Illustration of the chaining method. In this example, $T = [-1, 1]$, and for $k \in \mathbb{N}$, T_k is a $(1/2^k)$-net of T. Consider $x \in T$, and let x^k denote its closest element in T_k. We can decompose $x = x^0 + (x^1 - x^0) + (x^2 - x^1) + \cdots$.*

Let S be a set of vectors. Let $Z \in \{-1, 1\}^n$ be drawn from a p-wise independent family, where p is an even integer. Let $x \in \mathbb{R}^n$. By Markov and Khintchine's inequality,

$$\mathbb{P}(|\langle x, Z\rangle| > \lambda \cdot |S|^{1/p} \cdot \|x\|_2) < \frac{\mathbb{E}\,|\langle x, Z\rangle|^p}{\lambda^p \cdot |S| \cdot \|x\|_2^p} \leq \frac{1}{|S|} \cdot \left(\frac{\sqrt{p}}{\lambda}\right)^p.$$

Hence,

$$\begin{aligned}
\mathbb{E} \sup_{x \in S} |\langle x, Z\rangle| &= \int_0^\infty \mathbb{P}(\sup_{x \in S} |\langle x, Z\rangle| > u) du \\
&= |S|^{1/p} \cdot \sup_{x \in S} \|x\|_2 \cdot \int_0^\infty \mathbb{P}(\sup_{x \in S} |\langle x, Z\rangle| > \lambda \cdot |S|^{1/p} \cdot \sup_{x \in S} \|x\|_2) d\lambda \\
&< |S|^{1/p} \cdot \sup_{x \in S} \|x\|_2 \cdot \left(\sqrt{p} + \int_{\sqrt{p}}^\infty \left(\frac{\sqrt{p}}{\lambda}\right)^p d\lambda\right) \qquad \text{(union bound)} \\
&= |S|^{1/p} \cdot \sup_{x \in S} \|x\|_2 \cdot \sqrt{p} \cdot \left(1 + \frac{1}{p-1}\right).
\end{aligned}$$

Now we apply a similar chaining argument as in the proof of Dudley's inequality (cf. [Dud67]). For $x \in T$, let x^k denote the closest point to x in T_k. We illustrate this in Figure 2.5. We have $\|x^k - x^{k-1}\|_2 \leq \|x^k - x\|_2 + \|x - x^{k-1}\|_2 \leq (1/2^k) + (1/2^{k-1}) = 3/2^k$. Note that for any $x \in T$, let $v_t = x^k$ denote the closest vector to x in T_k (under ℓ_2), then the closest vector x_{k-1} to x in T_{k-1} must either be the frequency vector $v_{t'}$ in T_{k-1} such that t' is the smallest timestamp after t of a

vector in T_{k-1}, or the largest timestamp before t in T_{k-1}. Thus the size of $\{x^k - x^{k-1} | x \in T\}$ is upper bounded by $2|S_k| \leq 2^{2k+1}$, implying for $p = 4$,

$$\begin{aligned}
\mathbb{E} \sup_{x \in T} |\langle x, Z \rangle| &\leq \sum_{k=1}^{\infty} \mathbb{E} \sup_{x \in T} |\langle x^k - x^{k-1}, Z \rangle| \\
&< \sqrt{p} \left(1 + \frac{1}{p-1}\right) \sum_{k=1}^{\infty} (2^{2k+1})^{1/p} \cdot (3/2^k) \\
&< 23.
\end{aligned}$$

□

2.5 Identifying a Single Heavy Hitter Given Approximation to F_2

This section analyzes the subroutine HH1, which is formally presented in Algorithm 1. The goal of this section is to prove the correctness of HH1 in Lemma 2.5.4. We use $H \in [n]$ to denote the identity of the most frequent item in the stream. It is not assumed to be a heavy hitter unless explicitly stated.

2.5.1 Randomizing the labels

The first step of HH1 is to choose a hash function $h : [n] \to \{0,1\}^R$ where $R = O(\log n)$, that relabels the universe of items $[n]$. For each $r \geq 0$, let

$$\mathcal{H}_r := \{i \in [n] \setminus \{H\} \mid h(i)_k = h(H)_k \text{ for all } 1 \leq k \leq r\},$$

and let $\bar{\mathcal{H}}_r := \mathcal{H}_{r-1} \setminus \mathcal{H}_r$ for $r \geq 1$, and $\bar{\mathcal{H}}_0 = \emptyset$. By definition, $\mathcal{H}_R \subseteq \mathcal{H}_{R-1} \subseteq \cdots \subseteq \mathcal{H}_0 = [n] \setminus \{H\}$. In round $r \in [R]$, we hope that the active items are $\mathcal{H}_{r-1} \cup \{H\}$.

The purpose of randomizing the labels is a sort of "load balancing" among the item labels. The idea is that each bit of $h(H)$ partitions the active items into two roughly equal sized parts, i.e. $|\mathcal{H}_r| \approx |\bar{\mathcal{H}}_r|$ in every round r. This leads HH1 to discard roughly half of the active items after each round, allowing us to make progress on learning the (hashed) identity of the heavy hitter H. We will make use of the randomized labels in the next section, within the proof of Lemma 2.5.1.

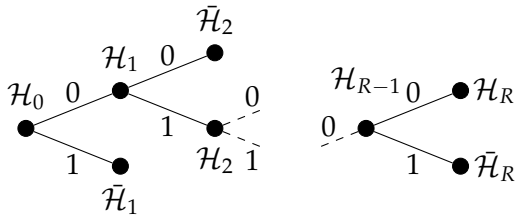


Figure 2.6: *In this example of the execution of HH1, the randomized label $h(H)$ of the heavy hitter H begins with* 01 *and ends with* 00*. Each node in the tree corresponds to a round of HH1, which must follow the path from $\mathcal{H}_0$ to $\mathcal{H}_R$ for the output to be correct.*

For h, we recommend choosing a prime $p \approx \min\{n, F_2\}^2$ and assigning the labels $h(i) = a_0 + a_1 i \bmod p$, for a_0 and a_1 randomly chosen in $\{0, 1, \ldots, p-1\}$ and $a_1 \neq 0$. We can always achieve this with $R = 3\log_2(\min\{n, F_2\} + 1)$, which is convenient for the upcoming analysis. This distribution on h is known to be a 2-wise independent family [CW79]. Note that computing $h(i)$ for any i takes $O(1)$ time. It is also simple to invert: namely $x = a_1^{-1}(h(x) - a_0) \bmod p$, so x can be computed quickly from $h(x)$ when $p > n$. Inverting requires computing the inverse of a_1 modulo p, which takes $O(\log \min\{n, F_2\})$ time via repeated squaring, however this computation can be done once, for example during initialization of the algorithm, and the result stored for all subsequent queries. Thus, the time to compute $a_1^{-1} \bmod p$ is negligible in comparison to reading the stream.

2.5.2 Learning the bits of the randomized label

After randomizing the labels, HH1 proceeds with a series of R rounds to identify the (randomized) label $h(H)$ of the heavy hitter H. The sequence of rounds is depicted in Figure 2.6. Each node in the tree corresponds to one round of HH1. The algorithm traverses the tree from left to right as the rounds progress. HH1 will successfully output H if (1) it traverses the path from $\mathcal{H}_0$ to $\mathcal{H}_R$ with the 0/1 labels on the path exactly the bits of $h(H)$; (2) $\mathcal{H}_R = \emptyset$. When $R = O(\log n)$ is sufficiently large we get $\mathcal{H}_R = \emptyset$ with high probability.

Now let us focus on one round and suppose that H is a K-heavy hitter, for some large constant K. Suppose the algorithm is in round $r \geq 1$, and recall that the goal of the round is to learn the

rth bit of $h(H)$. Our hope is that the active items are those in $\mathcal{H}_{r-1}$ (otherwise the algorithm will fail), which means that the algorithm has correctly discovered the first $r-1$ bits of $h(H)$. The main idea is that HH1 partitions $\mathcal{H}_{r-1} \cup \{H\}$ into $\mathcal{H}_r \cup \{H\}$ and $\bar{\mathcal{H}}_r$, creates a Bernoulli process for each partition, and compares the values of the two Bernoulli processes to discern $h(H)_r$. Suppose that the active items are indeed $\mathcal{H}_{r-1}$ and, for the sake of discussion, that the rth bit of the heavy hitter's label is $h(H)_r = 0$. Then the Bernoulli processes X_0 and X_1, defined in Algorithm 1, have the following form

$$X_0(t) = Z_H f_H^{(s:t)} + \sum_{i \in \mathcal{H}_r} Z_i f_i^{(s:t)}, \quad X_1(t) = \sum_{i \in \bar{\mathcal{H}}_r} Z_i f_i^{(s:t)},$$

where $s \leq m$ is the time of the last update to round $r-1$ and t is the current time. To simplify presentations we adopt the notation $f(S)$ for the frequency vector restricted to only items in S. For example, in the equations above become $X_0 = Z_H f_H^{(s:t)} + \langle Z, f^{(s:t)}(\mathcal{H}_r) \rangle$ and $X_1 = \langle Z, f^{(s:t)}(\bar{\mathcal{H}}_r) \rangle$.

The round is successful if $|X_0| > |X_1|$ (because we assumed $h(H)_r = 0$) at the first time when $|X_0 + X_1| > c\sigma\beta^r$. When that threshold is crossed, we must have either $|X_0| \geq c\sigma\beta^r/2$ or $|X_1| \geq c\sigma\beta^r/2$, or both. The way we will ensure that the round is a success is by establishing the following bound on the Bernoulli process X_1: $|X_1| = |\langle Z, f^{(s:t)}(\bar{\mathcal{H}}_r) \rangle| < c\sigma\beta^r/2$, for all times $t \geq s$. Of course, the round does not end until the threshold is crossed, so we will also establish a bound on the complementary Bernoulli process $|X_0 - Z_H f_H^{(s:t)}| = |\langle Z, f^{(s:t)}(\bar{\mathcal{H}}_r) \rangle| < c\sigma\beta^r/2$, at all times $t \geq s$. When this holds we must have $|X_0 + X_1| > c\sigma\beta^r$ no later than the first time t where $f_H^{(s:t)} \geq 2c\sigma\beta^r$, so the round ends after at most $2c\sigma\beta^r$ updates to H. In total over all of the rounds this uses up no more than $\sum_{r \geq 0} 2c\sigma\beta^r < f_H$ updates to H, where we have used $\sigma < 2\sqrt{F_2} < 3f_H$ by our assumption that H is a heavy hitter. In fact, both of those inequalities fail to hold with some probability, but the failure probability is $O(1/\beta^r 2^{r/2})$ so the probability that the algorithm succeeds will be $1 - \sum_{r \geq 0} O(1/\beta^r 2^{r/2}) > 2/3$.

The next lemma establishes the control on the Bernoulli processes that we have just described.

Lemma 2.5.1. *For any $r \in \{0, 1, \ldots, R\}$ and $K > 0$, the events*

$$E_{2r-1} := \left\{ \max_{s,t \leq m} |\left\langle Z, f^{(s:t)}(\bar{\mathcal{H}}_r) \right\rangle| \leq K F_2(\mathcal{H}_0)^{1/2} \right\}$$

and

$$E_{2r} := \left\{ \max_{s,t \le m} |\left\langle Z, f^{(s:t)}(\mathcal{H}_r) \right\rangle| \le KF_2(\mathcal{H}_0)^{1/2} \right\}$$

happen with probabilities at least $1 - \frac{4C_*}{K2^{r/2}}$ *respectively, where* $C_* < 23$ *is the constant from Theorem 2.3.1.*

Proof. By the Law of Total Probability and Theorem 2.3.1 with Markov's Inequality we have

$$\begin{aligned}
&\mathbb{P}\left(\max_{t \le m} |\langle Z, f^{(t)}(\mathcal{H}_r)\rangle| \ge \frac{1}{2} KF_2(\mathcal{H}_0)^{1/2} \right) \\
&= \mathbb{E}\left\{ \mathbb{P}\left(\max_{t \le m} |\langle Z, f^{(t)}(\mathcal{H}_r)\rangle| \ge \frac{1}{2} KF_2(\mathcal{H}_0)^{1/2} \middle| \mathcal{H}_r \right) \right\} \\
&\le \mathbb{E}\left\{ \frac{2C_* F_2(\mathcal{H}_r)^{1/2}}{KF_2(\mathcal{H}_0)^{1/2}} \right\} \\
&= \frac{2C_* F_2(\mathcal{H}_0)^{1/2}}{KF_2(\mathcal{H}_0)^{1/2} 2^{r/2}} \\
&= \frac{2C_*}{K2^{r/2}}.
\end{aligned}$$

The same argument holds if $\mathcal{H}_r$ is replaced by $\bar{\mathcal{H}}_r$.

By triangle inequality, $|\langle Z, f^{(s:t)}(\mathcal{H}_r)\rangle| \le |\langle Z, f^{(s)}(\mathcal{H}_r)\rangle| + |\langle Z, f^{(t)}(\mathcal{H}_r)\rangle|$. Therefore, $\mathbb{P}(E_{2r}) \ge 1 - \frac{4C_*}{K2^{r/2}}$. A similar argument proves $\mathbb{P}(E_{2r-1}) \ge 1 - \frac{4C_*}{K2^{r/2}}$. □

We will prove the correctness of HH1 by inductively applying Lemma 2.5.1 to bound the success probability of each round.

Let U be the event $\{h(j) \ne h(H) \text{ for all } j \ne H, f_j > 0\}$ which has probability $\mathbb{P}(U) \ge 1 - F_0 2^{-R} \ge 1 - \frac{1}{\min\{n, F_2\}^2}$, recalling that $F_0 \le \min\{n, F_2\}$ is the number of distinct items appearing before time m. The next lemma is the main proof of correctness for our algorithm.

Lemma 2.5.2. *Let* $K' \ge 128$, $c = 1/32$, *and* $\beta = 3/4$. *If* $2K'C_*\sqrt{F_2(\mathcal{H}_0)} \le \sigma \le 2\sqrt{2} f_H$ *and* $f_H > 2K'C_*\sqrt{F_2(\mathcal{H}_0)}$ *then, with probability at least* $1 - \frac{1}{\min\{F_2, n\}^2} - \frac{8}{K'c(\sqrt{2}\beta - 1)}$ *the algorithm HH1 returns* H.

Proof. Recall that H is *active* during round r if $h(H)_i = b_i$ for all $1 \le i \le r-1$, which means that updates for H in the stream are not discarded by the algorithm during round r. For each $r \in [R]$, we define events E_{2r-1} and E_{2r} by taking the value K to be $K_r = K'cC_*\beta^r$ in Lemma 2.5.1. Let E denote the event that U and $\cap_{r=1}^{2R} E_r$ both happen. We will prove by induction on r that if E

happens then either $b_r = h(H)_r$ for all $r \in [R]$, or H is the only item appearing in the stream. In either case, the algorithm correctly outputs H.

Let $r \geq 1$ be such that H is still active in round r, i.e. $b_i = h(H)_i$ for all $1 \leq i \leq r-1$. Note that all items are active in round 1. Since H is active, the rest of active items in round r are exactly $\mathcal{H}_{r-1} = \mathcal{H}_r \cup \bar{\mathcal{H}}_r$. Let t_r denote the time of the last update received during the rth round, and define $t_0 = 0$. At time $t_{r-1} \leq t < t_r$ we have

$$\begin{aligned} c\sigma\beta^r &> |X_0 + X_1| \\ &= |\langle Z, f^{(t_{r-1}:t)}(\mathcal{H}_r \cup \bar{\mathcal{H}}_r)\rangle + Z_H f_H^{(t_{r-1}:t)}| \\ &= |\langle Z, f^{(t_{r-1}:t)}(\mathcal{H}_{r-1})\rangle + Z_H f_H^{(t_{r-1}:t)}| \\ &\geq f_H^{(t_{r-1}:t)} - K_{r-1} F_2(\mathcal{H}_0)^{1/2}, \end{aligned}$$

where the last inequality follows because of event $E_{2(r-1)}$. Rearranging and using the assumed lower bound on σ, we get the bound

$$K_{r-1} F_2(\mathcal{H}_0)^{1/2} \leq \frac{K_{r-1}}{2K'C_*}\sigma = \frac{1}{2}c\sigma\beta^{r-1}. \tag{2.1}$$

Therefore, by rearranging we have $f_H^{(t_{r-1}:t_r)} \leq 1 + f_H^{(t_{r-1}:t_r-1)} < 1 + (\frac{1}{2}+\beta)c\sigma\beta^{r-1}$. That implies $f_H^{(t_r)} = \sum_{k=1}^r f_H^{(t_{k-1}:t_k)} < r + (\frac{1}{2}+\beta)c\sigma\sum_{k=1}^r \beta^{k-1} \leq r + \frac{(1+2\beta)\sqrt{2}c}{1-\beta} f_H$. Thus, if $f_H - \frac{3\sqrt{2}c}{1-\beta} f_H > R$ then round $r \leq R$ is guaranteed to be completed and a further update to H appears after the round. Suppose, that is not the case, and rather $R \geq f_H - \frac{(1+2\beta)\sqrt{2}c}{1-\beta} f_H \geq \frac{1}{2} f_H$, where the last inequality follows from our choices $\beta = 3/4$ and $c = 1/32$. Then, by the definition of R, $9(1+\log_2 8f_H^2)^2 \geq R^2 \geq \frac{1}{4} f_H^2$. One can check that this inequality implies that $f_H \leq 104$. However, $K' \geq 128$ and the heaviness requirement of H implies that $F_2(\mathcal{H}_0) = 0$. Therefore, H is the only item in the stream, and, in that case the algorithm will always correctly output H.

Furthermore, at the end of round r, $|X_0 + X_1| \geq c\sigma\beta^r$, so we have must have either $|X_0| \geq c\sigma\beta^r/2$ or $|X_1| \geq c\sigma\beta^r/2$. Note that both E_{2r-1} and E_{2r} occur. Recall these two events govern the non-heavy items contributions to X_0 and X_1, together with inequality (2.1), we have

$$|\langle Z, f^{(t_{r-1}:t_r)}(\mathcal{H}_r)\rangle| \leq K_r F_2(\mathcal{H}_0)^{1/2} < \frac{1}{2}c\sigma\beta^r$$

and the same holds for $\bar{\mathcal{H}}_r$. Therefore, the Bernoulli process not including H has its value smaller than $c\sigma\beta^r/2$, and the other larger process identifies the bit $h(H)_r$. By induction, the algorithm completes every round $r = 1, 2, \ldots, R$ and there is at least one update to H after round R. This proves the correctness of the algorithm assuming the event E happens.

It remains to compute the probability of E. Lemma 2.5.1 provides the bound

$$\begin{aligned}
\mathbb{P}(U \text{ and } \cap_{i=1}^{2R} E_i) &\geq 1 - \frac{1}{\min\{n, F_2\}^2} - \sum_{r=0}^{R} \frac{8C_*}{K_r 2^{r/2}} \\
&= 1 - \frac{1}{\min\{n, F_2\}^2} - \sum_{r=0}^{R} \frac{8}{K' c \beta^r 2^{r/2}} \\
&> 1 - \frac{1}{\min\{n, F_2\}^2} - \frac{8}{K' c(\sqrt{2}\beta - 1)}.
\end{aligned}$$

□

Proposition 2.5.3. *Let $\alpha \geq 1$. If $F_2^{1/2} \leq \sigma \leq 2F_2^{1/2}$ and $f_H \geq \alpha\sqrt{F_2(\mathcal{H}_0)}$, then $\alpha\sqrt{F_2(\mathcal{H}_0)} \leq \sigma \leq 2\sqrt{2}f_H$.*

Proof. It follows from $\alpha\sqrt{F_2(\mathcal{H}_0)} \leq f_H \leq \sigma$ and $f_H^2 = F_2 - F_2(\mathcal{H}_0) \geq (1 - \frac{1}{1+\alpha^2})F_2 \geq \frac{1}{8}\sigma^2$. □

Lemma 2.5.4 (HH1 Correctness)**.** *There is a constant K such that if H is a K-heavy hitter and $\sqrt{F_2} \leq \sigma \leq 2\sqrt{F_2}$, then with probability at least $2/3$ algorithm HH1 returns H. HH1 uses $O(1)$ words of storage.*

Proof. The Lemma follows immediately from Proposition 2.5.3 and Lemma 2.5.2 by setting $K' = 2^{13}$, so that it is sufficient to set $K = 2^{14}C_* \leq 380,000$. □

2.6 F_2 Tracking

This section proves that the AMS algorithm with 8-wise, rather than 4-wise, independent random signs has an additive εF_2 approximation guarantee at all points in the stream. We will use the tracking algorithm to "guess" a good value of σ for input to HH1. Because the AMS algorithm is a fundamental streaming primitive, it is of independent interest from the BPTree algorithm. The following theorem is a direct consequence of Lemma 2.6.3 and [AMS99].

Theorem 2.6.1. *Let $0 < \varepsilon < 1$. There is a streaming algorithm that outputs at each time t a value $\widehat{F}_2^{(t)}$ such that $\mathbb{P}(|\widehat{F}_2^{(t)} - F_2^{(t)}| \leq \varepsilon F_2, \forall t : 0 \leq t \leq m) \geq 1 - \delta$. The algorithm use $O(\frac{1}{\varepsilon^2} \log \frac{1}{\delta})$ words of storage and has $O(\frac{1}{\varepsilon^2} \log \frac{1}{\delta})$ update time.*

Let us remark that it follows from Theorem 2.6.1 and a union bound that one can obtain a $(1 \pm \varepsilon)$ *multiplicative* approximation to F_2 at all points in the stream using $O(\varepsilon^{-2} \log \log m)$ words. This can be done by breaking the stream into $O(\log m)$ intervals where the breaking points are those t so that $F_2^{(t)}$ doubles.

In its original form [AMS99], the AMS sketch is a product of the form Πf, where Π is a random $k \times n$ matrix chosen with each row independently composed of 4-wise independent $\pm \frac{1}{\sqrt{k}}$ random variables. Note that $\mathbb{E} \|\Pi f\|_2^2 = \|f\|_2^2$. The sketch uses $k = \Theta(1/\varepsilon^2)$ rows to achieve a $(1 \pm \varepsilon)$-approximation with constant probability. We show that the AMS sketch with $k \simeq 1/\varepsilon^2$ rows and 8-wise independent entries provides ℓ_2-tracking with additive error $\varepsilon \|f\|_2$ at all times. We use $\|A\|$ to denote the spectral norm of A, which is equal to its largest singular value, and $\|A\|_F$ for the Frobenius norm, which is the Euclidean length of A when viewed as a vector. Our proof makes use of the following moment bound for quadratic forms. Recall that given a metric space (X, d) and $\varepsilon > 0$, an *ε-net* of X is a subset $N \subseteq X$ such that $d(x, N) = \inf_{y \in N} d(x, y) \leq \varepsilon$ for all $x \in X$.

Theorem 2.6.2 (Hanson-Wright [HW71])**.** *For $B \in \mathbb{R}^{n \times n}$ symmetric with (Z_i) uniformly random in $\{-1, 1\}^n$, for all $p \geq 1$, $\|Z^T B Z - \mathbb{E}\, Z^T B Z\|_p \lesssim \sqrt{p} \|B\|_F + p\|B\|$.*

In the following, we prove our F_2 tracking lemma by a chaining argument. The proof is similar to the supremum bound proof of Bernoulli process with bounded independence given in Section 2.4.

Lemma 2.6.3 (F_2 tracking)**.** *If $k \gtrsim 1/\varepsilon^2$ and $Z \in \{-1, +1\}^{kn}$ are 8-wise independent then*

$$\mathbb{E} \sup_t \left| \|\Pi f^{(t)}\|_2^2 - \|f^{(t)}\|_2^2 \right| \leq \varepsilon \|f\|_2^2.$$

Proof. Let $v_t = f^{(t)} / \|f^{(m)}\|_2$ and $T = \{v_0, v_1, \ldots, v_m\}$. Note that $\|v_0\|_2 = 0$, $\|v_m\|_2 = 1$, and $\|v_t\|_2$ is monotonically increasing in t. For $\ell \in \mathbb{N}$, let T_ℓ be the $2^{-\ell}$-net of T under ℓ_2 by the greedy

construction in Section 2.4. We have $|T_\ell| \le 4^\ell$ and $v_0 \in T_\ell$.

Note that the AMS sketch can be written as $\Pi x = A_x Z$, where $A_x \in \mathbb{R}^{k \times kn}$ so that

$$A_x := \frac{1}{\sqrt{k}} \begin{bmatrix} x^T & 0_{1\times n} & \cdots & 0_{1\times n} \\ 0_{1\times n} & x^T & \cdots & 0_{1\times n} \\ \vdots & \vdots & \ddots & \vdots \\ 0_{1\times n} & 0_{1\times n} & \cdots & x^T \end{bmatrix},$$

and $Z \in \{-1, 1\}^{kn}$ is $2p$-wise independent (we will finally let $p = 4$, but the following argument holds as long as p is an even integer). Let $B_x = A_x^T A_x$. We are interested in bounding $\mathbb{E}_Z \sup_{x\in T} |Z^T B_x Z - \mathbb{E}\, Z^T B_x Z|$.

Note that for any $x, y \in \mathbb{R}^n$ such that $\|x\|_2, \|y\|_2 \le 1$,

$$\begin{aligned} \|xx^T - yy^T\|_F &= \|x(x-y)^T + (x-y)y^T\|_F \\ &\le \|x(x-y)^T\|_F + \|(x-y)y^T\|_F \\ &= \|x\|_2\|x-y\|_2 + \|x-y\|_2\|y\|_2 \\ &\le 2\|x-y\|_2. \end{aligned}$$

Thus $\|B_x - B_y\|_F = \frac{1}{\sqrt{k}}\|xx^T - yy^T\|_F \le \frac{2}{\sqrt{k}}\|x-y\|_2$, for all $x, y \in T$. Moreover, $\sup_{x\in T}\|B_x\| \le \sup_{x\in T}\|B_x\|_F = \|B_{v_m}\|_F = 1/\sqrt{k}$. Therefore $\mathcal{B}_\ell := \{B_x : x \in T_\ell\}$ is a $2^{1-\ell}/\sqrt{k}$-net under both $\|\cdot\|$ and $\|\cdot\|_F$. For $x \in T$, let $x^\ell \in T_\ell$ denote the closest element in T_ℓ sunder ℓ_2. Then we can write $B_x = B_{x^0} + \sum_{\ell=1}^\infty \Delta_{x,\ell} = \sum_{\ell=1}^\infty \Delta_{x,\ell}$, where $\Delta_{x,\ell} = B_{x^\ell} - B_{x^{\ell-1}}$. For brevity, we also define $\gamma(A) := |Z^T A Z^T - \mathbb{E}\, Z^T A Z^T|$. We have

$$\mathbb{E}\sup_{x\in T}\gamma(B_x) \le \mathbb{E}\sup_{x\in T}\sum_{\ell=1}^\infty \gamma(\Delta_{x,\ell}) \le \sum_{\ell=1}^\infty \mathbb{E}\sup_{x\in T}\gamma(\Delta_{x,\ell}). \tag{2.2}$$

For symmetric matrix $A \in \mathbb{R}^{n\times n}$, by Hanson-Wright Inequality

$$\mathbb{P}(\gamma(A) > \lambda \cdot S^{1/p}) < \frac{1}{S} \cdot \left(\frac{C\sqrt{p}\|A\|_F + Cp\|A\|}{\lambda}\right)^p$$

for some constant $C > 0$. Thus if $\mathcal{A}$ is a collection of such matrices, $|\mathcal{A}| = S$, choosing

$u^* = C(\sqrt{p} \cdot \sup_{A\in\mathcal{A}} \|A\|_F + p \cdot \sup_{A\in\mathcal{A}} \|A\|)$,

$$\begin{aligned}
\mathbb{E} \sup_{A\in\mathcal{A}} \gamma(A) &= \int_0^\infty \mathbb{P}(\sup_{A\in\mathcal{A}} \gamma(A) > u) du \\
&= S^{1/p} \cdot \int_0^\infty \mathbb{P}(\sup_{A\in\mathcal{A}} \gamma(A) > \lambda \cdot S^{1/p}) d\lambda \\
&= S^{1/p}(u^* + \int_{u^*}^\infty \mathbb{P}(\sup_{A\in\mathcal{A}} \gamma(A) > \lambda \cdot S^{1/p}) d\lambda) \\
&\lesssim S^{1/p}(\sqrt{p} \cdot \sup_{A\in\mathcal{A}} \|A\|_F + p \cdot \sup_{A\in\mathcal{A}} \|A\|). \qquad (2.3)
\end{aligned}$$

Now by applying (2.3) to (2.2) repeatedly with $\mathcal{A} = \mathcal{A}_\ell = \{B_{x^\ell} - B_{x^{\ell-1}} : x \in T\}$, noting $\sup_{A\in\mathcal{A}_\ell} \|A\| \le \sup_{A\in\mathcal{A}_\ell} \|A\|_F \le 2^{1-\ell}/\sqrt{k}$ and $|\mathcal{A}_\ell| \le 2|T_\ell| \le 2^{2\ell+1}$,

$$\mathbb{E} \sup_{x\in T} \gamma(B_x) \lesssim \sum_{\ell=1}^\infty \frac{p 2^{2\frac{\ell}{p}-\ell}}{\sqrt{k}} \lesssim \frac{p}{\sqrt{k}}$$

for $p \ge 4$. Thus it suffices for the entries of Z to be 8-wise independent. □

2.7 The Complete Heavy Hitters Algorithm

In this section, we first describe HH2, formally presented in Algorithm 2. The algorithm removes the assumption on σ required by HH1. The complete algorithm BPTree follows.

The step in HH2 that guesses an approximation σ for $\sqrt{F_2}$ works as follows. HH2 constructs the estimator $\widehat{F}_2$ of the previous section to (approximately) track F_2. HH2 starts a new instance of HH1 each time the estimate $\widehat{F}_2$ crosses a power of 2. Each new instance is initialized with the current estimate of $\sqrt{F_2}$ as the value for σ, but HH2 maintains only the two most recent copies of HH1. Thus even though it may instantiate overall $\Omega(\log n)$ copies of HH1, at most two will be running concurrently and the total storage remains $O(1)$ words. At least one of the thresholds will be the "right" one, in the sense that HH1 gets initialized with σ in the interval $[\sqrt{F_2}, 2\sqrt{F_2}]$, so we expect the corresponding instance of HH1 to identify the heavy hitter, if one exists.

The scheme could fail if $\widehat{F}_2$ is wildly inaccurate at some points in the stream, for example if $\widehat{F}_2$ ever grows too large then the algorithm could discard every instance of HH1 that was correctly initialized. But Theorem 2.6.1 guarantees that it fails only with small probability.

We begin by proving the correctness of HH2 in Lemma 2.7.1 and then complete the description and correctness of BPTree in Theorem 2.7.2.

Algorithm 2 Identify a heavy hitter by guessing σ.

procedure HH2($p_1, p_2, \ldots, p_m$)
  Run $\widehat{F}_2$ from Theorem 2.6.1 with $\varepsilon = 1/100$ and $\delta = 1/20$
  Start HH1 $(1, p_1, \ldots, p_m)$
  Let $t_0 = 1$ and $t_k = \min\{t \mid \widehat{F}_2^{(t)} \geq 2^k\}$, for $k \geq 1$.
  for each time t_k **do**
    Start HH1$((\widehat{F}_2^{(t_k)})^{1/2}, p_{t_k}, p_{t_k+1}, \ldots p_m)$
    Let H_k denote its output if it completes
    Discard H_{k-2} and the copy of HH1 started at t_{k-2}
  end for
  return H_{k-1}
end procedure

Lemma 2.7.1. *There exists a constant $K > 0$ and a 1-pass streaming algorithm HH2 (in Algorithm 2), such that if the stream contains a K-heavy hitter, then with probability at least 0.6 HH2 returns the identity of the heavy hitter. The algorithm uses $O(1)$ words of memory and $O(1)$ update time.*

Proof. The space and update time bounds are immediate from the description of the algorithm. The success probability follows by a union bound over the failure probabilities in Lemma 2.5.4 and Theorem 2.6.1, which are $1/3$ and $\delta = 0.05$ respectively. It remains to prove that there is a constant K such that conditionally given the success of the F_2 estimator, the hypotheses of Lemma 2.5.4 are satisfied by the penultimate instance of HH1 by HH2.

Let K' denote the constant from Lemma 2.5.4 and set $K = 12K'$, so if H is a K-heavy hitter then for any $\alpha > 2/K$ and in any interval $(t, t']$ where $(F_2^{(t')})^{1/2} - (F_2^{(t)})^{1/2} \geq \alpha\sqrt{F_2}$ we will have

$$f_H^{(t:t')} + F_2(\mathcal{H}_0)^{1/2} \geq \|f^{(t:t')}\|_2 \geq \|f^{(t')}\|_2 - \|f^{(t)}\|_2 \geq \alpha\sqrt{F_2}.$$

If follows with in the stream $p_t, p_{t+1}, \ldots, p_{t'}$ the heaviness of H is at least

$$\frac{f_H^{(t:t')}}{F_2^{(t:t')}(\mathcal{H}_0)^{1/2}} \geq \frac{\alpha\sqrt{F_2} - \sqrt{F_2(\mathcal{H}_0)}}{\sqrt{F_2(\mathcal{H}_0)}} \geq K\alpha - 1 \geq \frac{K\alpha}{2}. \tag{2.4}$$

Of course, if $F_2^{(t:t')}(\mathcal{H}_0) = 0$ the same heaviness holds.

Let k be the last iteration of HH2. By the definition of t_k, we have $(\widehat{F}_2^{(t_{k-1})})^{1/2} \geq \frac{1}{4}(\widehat{F}_2)^{1/2} \geq \frac{1}{4}\sqrt{(1-\varepsilon)F_2}$. Similar calculations show that there exists a time $t_* > t_{k-1}$ such that $(F_2^{(t_*)})^{1/2} - (F_2^{(t_{k-1})})^{1/2} \geq \frac{1}{6}F_2$ and $\|f^{(t_{k-1}:t_*)}\|_2 \leq (\widehat{F}_2^{(t_{k-1})})^{1/2} \leq 2\|f^{(t_{k-1}:t_*)}\|_2$. In particular, the second pair of inequalities implies that $\widehat{F}_2^{(t_k-1)}$ is a good "guess" for σ on the interval $(t_{k-1}, t^*]$. We claim H is a K' heavy hitter on that interval, too. Indeed, because of (2.4), with $\alpha = 1/6$, we get that H is a K'-heavy hitter on the interval $(t_{k-1}, t^*]$. This proves that the hypotheses of Lemma 2.5.4 are satisfied for the stream $p_{t_{k-1}+1}, \ldots, p_{t^*}$. It follows that from Lemma 2.5.4 that HH1 correctly identifies $H_{k-1} = H$ on that substream and the remaining updates in the interval $(t_*, t_m]$ do not affect the outcome. □

Now a standard reduction from ε-heavy hitters to $O(1)$-heavy hitters gives the following theorem.

Theorem 2.7.2. *For any $\varepsilon > 0$ there is 1-pass streaming algorithm BPTree that, with probability at least $(1-\delta)$, returns a set of $\frac{\varepsilon}{2}$-heavy hitters containing every ε-heavy hitter and an approximate frequency for every item returned satisfying the $(\varepsilon, 1/\varepsilon^2)$-tail guarantee. The algorithm uses $O(\frac{1}{\varepsilon^2}\log\frac{1}{\varepsilon\delta})$ words of space and has $O(\log\frac{1}{\varepsilon\delta})$ update and $O(\varepsilon^{-2}\log\frac{1}{\varepsilon\delta})$ retrieval time.*

Proof. The algorithm BPTree constructs a hash table in the same manner as CountSketch where the items are hashed into $B = O(1/\varepsilon^2)$ buckets for $R = O(\log 1/\varepsilon\delta)$ repetitions. On the stream fed into each bucket we run an independent copy of HH2. A standard $R \times B$ CountSketch is also constructed. The constants are chosen so that when an ε-heavy hitter in the stream is hashed into a bucket it becomes a K-heavy hitter with probability at least 0.95. Thus, in any bucket with a the ε-heavy hitter, the heavy hitter is identified with probability at least 0.55 by Lemma 2.7.1 and the aforementioned hashing success probability.

At the end of the stream, all of the items returned by instances of HH2 are collected and their frequencies checked using the CountSketch. Any items that cannot be ε-heavy hitters are discarded. The correctness of this algorithm, the bound on its success probability, and the $(\varepsilon, 1/\varepsilon^2)$-tail guarantee follow directly from the correctness of CountSketch and the fact that no more than $O(\varepsilon^{-2}\log(1/\delta\varepsilon))$ items are identified as potential heavy hitters. □

We can amplify the success probability of HH2 to any $1-\delta$ by running $O(\log \delta^{-1})$ copies in parallel and taking a majority vote for the heavy hitter. This allows one to track $O(1)$-heavy hitters at all points in the stream with an additional $O(\log \log m)$ factor in space and update time. The reason is because there can be a succession of at most $O(\log m)$ 2-heavy hitters in the stream, since their frequencies must increase geometrically, so setting $\delta = \Theta(1/\log m)$ is sufficient. The same scheme works for BPTree tree, as well, and if one replaces each of the counters in the attached CountSketch with an F_2-at-all-times estimator of [BCIW16] then one can track the frequencies of all ε-heavy hitters at all times as well. The total storage becomes $O(\frac{1}{\varepsilon^2}(\log \log n + \log \varepsilon^{-1}))$ words and the update time is $O(\log \log n + \log \varepsilon^{-1})$.

2.8 Conclusion

We have obtained a new algorithm called BPTree for finding ℓ_2 heavy hitters in insertion streams. It has space complexity $O(\varepsilon^{-2} \log \varepsilon^{-1})$ words. When ε is a constant, which is one of the most common settings in application, our space complexity is optimal. It is still open and interesting to see if it is possible to achieve $O(\varepsilon^{-2})$ words, since the current best space lower bound is $\Omega(\varepsilon^{-2})$ words, which comes from the fact that there can be as many as $\Theta(\varepsilon^{-2})$ (ε, ℓ_2)-heavy hitters, and each requires $\Theta(1)$ words to store.

Our BPTree has an extra $O(\log \varepsilon^{-1})$ factor in space complexity mainly because of the following two reasons: (1) In the the process of identifying heavy hitters, we pay an extra factor of $\log \varepsilon^{-1}$, because intuitively we identify heavy hitters in a coupon collecting fashion, and we need $\Theta(\varepsilon^{-2} \log \varepsilon^{-1})$ trials to collect all the $\Theta(\varepsilon^{-2})$ heavy hitters. Namely, for each heavy hitter, we approximately run $\Theta(\log \varepsilon^{-1})$ instances of HH1 to make sure the heavy hitter is identified with a good enough success probability (so that we can finally afford to apply a union bound over all heavy hitters). (2) CountSketch takes $\Theta(\varepsilon^{-2} \log \varepsilon^{-1})$ words of space to estimate correctly the frequencies of all $\text{poly}(\varepsilon^{-1})$ heavy hitter candidates.

Actually, the main obstacle is part (1) of the above discussion, i.e., the identification part. Because we can replace CountSketch by maintaining a counter in HH1 recording the number of times the current heavy hitter candidate appears in the stream. The counter then serves as a

lower bound on the frequency of the heavy hitter candidate. And by setting parameters properly, we can make sure that with constant probability, we obtain the heavy hitter as candidate before consuming half of its occurrences in the stream (given the condition that the heavy hitter is at least constant heavy).

CountSieve in [BCIW16] has an extra $O(\log\log n)$ factor in space complexity mainly because their derandomization of Bernoulli process (BP) requires $O(\log\log n)$ words to store a random seed, as well as because their subroutine called Amplifier needs to instantiate $O(\log\log n)$ copies of BPs in parallel in order to make the heavy hitter from $O(1)$-heavy to $\text{poly}(\log n)$-heavy. Our BPTree resolves these two issues by using a 4-wise independent family to derandomize *BP*, and a procedure similar to walking from the root of a *tree* to identify the heavy hitter.

It is worth to point out that CountSieve does not necessarily need to spend $O(\log\log n)$ words in Amplifier. It is possible to use $O(\log\log\log n)$ BPs in parallel in Amplifier to boost the heavy hitter to be $\text{poly}(\log\log n)$-heavy. Pipe-lined with the output of Amplifier, it takes $O(\log\log n)$ sequential BPs to boost heaviness to $\text{poly}(\log n)$, and then another $O(\log n)$ sequential BPs to identify the heavy hitter. Applying the sequential BP steps multiple times, one can show that Amplifier and subsequent subroutines only take $O(\log^{(c)} n)$ words (if we ignore the cost of derandomizing BPs), where c is an arbitrary constant, and $\log^{(c)}$ refers to applying log operation c times.

We have shown that a Bernoulli process with 4-wise independent Rademacher random vector instead of full-wise independence still has its expected supremum bounded by a constant of the final ℓ_2 norm of its frequency vector. It is interesting to see if pairwise independent Rademacher suffices.

Chapter 3

Sampler Lower Bounds

The sampler is a fundamental primitive in streaming literature, especially useful in dynamic graph streaming problems. In a turnstile stream, a vector $x \in \mathbb{Z}^n$ starts as 0 and receives a stream of updates in the form of $x_i \leftarrow x_i + \Delta$. By the end of the stream, a sampler is required to output an element in $\text{supp}(x) = \{i : x_i \neq 0\}$. We call the task support-finding if we are satisfied with outputting an arbitrary element in the support. An ℓ_0-sampler additionally requires the output to be uniformly distributed in $\text{supp}(x)$. More generally, an ℓ_p-sampler outputs i with probability proportional to $|x_i|^p$. Note that a space lower bound on support-finding implies the same lower bound on ℓ_p-sampler, so it is sufficient to prove lower bound on support-finding.

It is known that the space complexity of ℓ_0-sampler with constant failure probability is $\Theta(\log^2 n)$ bits [JST11]. [JST11] also implement an ℓ_0-sampler with failure probability at most δ using $O(\log \delta^{-1} \log^2 n)$ bits. Prior to our work, no non-trivial lower bounds are known in terms of δ. On the other hand, graph stream applications often require $\delta = O(1/\text{poly}(n))$. Therefore, it is interesting to bridge the gap. In this chapter, we show that $\Omega(\min(n, \log \delta^{-1} \log^2 \frac{n}{\log \delta^{-1}}))$ bits is required for ℓ_0-sampling and support-finding with failure probability δ. For $\delta > 2^{-n^{.99}}$, namely when δ is not extremely small, the space lower bound becomes $\Omega(\log \delta^{-1} \log^2 n)$, matching the upper bound in [JST11].

We also consider a variant of the sampler which outputs k (instead of 1) elements in the support with constant failure probability, which we call support-finding$_k$. ℓ_0-sampling$_k$ requires

the k outputs to be uniform. We obtain a lower bound of $\Omega(k \log^2(n/k))$ bits for support-finding$_k$ (and of course also ℓ_0-sampling$_k$).

In this chapter, we prove lower bounds for support-finding with failure probability δ and support-finding$_k$ by first proving a lower bound on $\mathbf{R}_\delta^{\rightarrow,pub}(\mathbf{UR}_k^{\subset})$, where $\mathbf{R}_\delta^{\rightarrow,pub}$ refers to randomized one-way communication complexity under public coin with failure probability δ and $\mathbf{UR}_k^{\subset}$ refers to two party communication problem called *universal relation* in which Alice and Bob receive $x, y \in \{0,1\}^n$ respectively with the promise that $\text{supp}(y) \subset \text{supp}(x)$ and they want to find k elements in $\{i \in [n] : x_i \neq y_i\}$. In this chapter, our main result is $\mathbf{R}_\delta^{\rightarrow,pub}(\mathbf{UR}_k^{\subset}) = \Theta(\min\{n, t\log^2(n/t)\})$ where $t = \max\{\log \delta^{-1}, k\}$. Our sampler lower bounds follow directly from the universal relation bound. In addition, we also prove a new space lower bound on finding duplicates by reducing to universal relation.

Organization. This chapter is organized as follows. Section 3.1 is the introduction section which consists of 4 parts: Section 3.1.1 gives a formal definition of samplers and related problems, Section 3.1.2 reviews some of the related works, Section 3.1.3 summarizes our main results, Section 3.1.4 presents our intuition of the sampler lower bounds. Section 3.2 presents notations, reductions and main results. Section 3.3 gives an overview of the techniques. Section 3.4 provides the proof that $\mathbf{R}_\delta^{\rightarrow,pub}(\mathbf{UR}^{\subset}) = \Omega(\min\{n, \log^2(\frac{n}{\log(1/\delta)})\log\frac{1}{\delta}\})$. Section 3.5 extends the result to $\mathbf{UR}_k^{\subset}$ for $k \geq 1$, and proves a lower bound of $\Omega(k\log^2(n/k))$ communication for constant failure probability. Section 3.6 proves a matching upper bound for $\mathbf{R}_\delta^{\rightarrow,pub}(\mathbf{UR}_k^{\subset})$. Section 3.7 discusses a variant of samplers that has applications in graph streams. Section 3.8 concludes this chapter with some remarks.

Bibliographic remarks: The results described in this chapter were developed jointly with Michael Kapralov, Jelani Nelson, Jakub Pachocki, David Woodruff and Mobin Yahyazadeh [KNP+17].

3.1 Introduction

3.1.1 Problem formulation

In turnstile ℓ_0-sampling, a vector $z \in \mathbb{R}^n$ starts as the zero vector and receives coordinate-wise updates of the form "$z_i \leftarrow z_i + \Delta$" for $\Delta \in \{-M, -M+1, \ldots, M\}$. During a query, one must return a uniformly random element from $\mathrm{supp}(x) = \{i : z_i \neq 0\}$. In its most general form, the sampling problem is specified by the following four parameters, and a data structure solving the problem is called a *sampler*.

- parameter p, where $p \geq 0$. We call the problem ℓ_p-sampling, and one must return index $i \in [n]$ with probability approximately proportional to $|z_i|^p$.
- parameter δ, where $\delta \in (0,1)$ denotes the failure probability. Namely, the sampler can report "fail" with probability at most δ.
- parameter ε, where $\varepsilon \in [0,1)$. We require that conditioned on the sampler does not report "fail", it returns index i with probability $(1 \pm \varepsilon)\frac{|z_i|^p}{\|z\|_p^p} + O(\delta n^{-c})$, where c is a sufficiently large constant (for example, $c = 2$) and the $O(\delta n^{-c})$ term in some literature is written as $O(n^{-c})$ but in the thesis δ can be as small as 2^{-n} so it is reasonable to have $O(\delta n^{-c})$ here. If $\varepsilon = 0$, we say the sampler is *perfect*. Unless explicitly specified, we assume that ε is a small constant.
- parameter k, where $1 \leq k \leq n$. Here k refers to the number of samples we require from the sampler. In the case that $k = 1$, we can drop the subscript in ℓ_p-sampling$_k$ and denote as ℓ_p-sampling.

Problem 3.1.1 (ℓ_p-sampling$_k$ and support-finding$_k$)**.** *In the ℓ_p-sampling$_k$ problem, we have a vector $z \in \mathbb{Z}^n$ receiving turnstile updates. We want to maintain a sampler, such that at the end of the stream, the sampler outputs "fail" with probability at most δ, and otherwise if z is not a zero vector, it returns k independent samples, where for each sample, i is chosen with probability $(1 \pm \varepsilon)\frac{|z_i|^p}{\|z\|_p^p} + O(\delta n^{-c})$, where ε is a small constant and c is a sufficiently large constant. The support-finding$_k$ problem drops the requirement on distribition, and only requires to output $\min\{k, \|z\|_0\}$ elements in $\mathrm{supp}(z)$.*

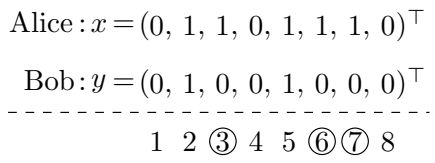


Figure 3.1: *An instance of* $\mathbf{UR}^{\subset}$. *The universe size* $n = 8$, *Alice receives* x *and Bob receives* y *with* $\mathrm{supp}(x) \supset \mathrm{supp}(y)$. *In this figure, the items in circle represent the elements of* $\mathrm{supp}(x - y) = \{3, 6, 7\}$, *and the goal of* $\mathbf{UR}^{\subset}$ *is to output any of them.*

We define the communication problem **UR** (universal relations), whose one-way public-coin communication complexity implies space lower bounds on 1-pass streaming algorithms for both ℓ_p-sampling$_k$ and support-finding$_k$.

Problem 3.1.2 (Universal relations)**.** **UR** *(universal relations) is a two-party communication problem where Alice and Bob receive* $x, y \in \{0,1\}^n$ *respectively with the promise that* $x \neq y$. *The goal of the communication is to decide an index* $i \in [n]$ *such that* $x_i \neq y_i$. *We also consider a generalized problem* $\mathbf{UR}_k$, *where the output must be* $\min\{k, \|x - y\|_0\}$ *distinct indices on which* x, y *differ. We also use* $\mathbf{UR}^{\subset}, \mathbf{UR}_k^{\subset}$ *to denote the variants when promised* $\mathrm{supp}(y) \subset \mathrm{supp}(x)$, *and also Bob knows* $\|x\|_0$. *Clearly* $\mathbf{UR}, \mathbf{UR}_k$ *can only be harder than* $\mathbf{UR}^{\subset}, \mathbf{UR}_k^{\subset}$, *respectively.*

Figure 3.1 illustrates an instance of $\mathbf{UR}^{\subset}$ where the universe size $n = 8$, Alice receives x and Bob receives y with $\mathrm{supp}(x) \supset \mathrm{supp}(y)$. The goal of $\mathbf{UR}^{\subset}$ is to output an element from $\mathrm{supp}(x - y) = \{3, 6, 7\}$. For the same input, $\mathbf{UR}_k^{\subset}$ for $k = 2$ requires they output any $\min\{k, \|x - y\|_0\} = 2$ elements from $\mathrm{supp}(x - y) = \{3, 6, 7\}$, namely, $\{3, 6\}$, $\{3, 7\}$ or $\{6, 7\}$.

Our main result for this chapter is a tight bound on $\mathbf{R}_\delta^{\rightarrow, pub}(\mathbf{UR}_k^{\subset})$, which denotes the randomized one-way communication complexity of $\mathbf{UR}_k^{\subset}$ with failure probability at most δ under the public coin model. Our lower bound on $\mathbf{R}_\delta^{\rightarrow, pub}(\mathbf{UR}_k^{\subset})$ also implies a space lower bound on finding duplicates in an insertion stream, which is defined as follows.

Problem 3.1.3 (Finding duplicates)**.** *In* FindDuplicate(n), *one is given a length-*$(n+1)$ *stream of integers in* $[n]$, *and the algorithm must output some element* $i \in [n]$ *which appeared at least twice in the stream.*

3.1.2 Related works

The question of whether ℓ_0-sampling is possible in low memory in turnstile streams was first asked in [CMR05, FIS08]. The work [FIS08] apply ℓ_0-sampling as a subroutine in approximating the cost of the Euclidean minimum spanning tree of a subset of a discrete geometric space subject to insertions and deletions. The algorithm given there used space $O(\log^3 n)$ bits to achieve failure probability $1/\text{poly}(n)$ (though it is likely that the space could be improved to $O(\log^2 n \log\log n)$ with a worse failure probability, by replacing a subroutine used there with a more recent ℓ_0-estimation algorithm of [KNW10]). The currently best known upper bound solves ℓ_0-sampling$_k$ using $O(t \log^2 n)$ bits [JST11] where $t = \max\{k, \log \delta^{-1}\}$.

Sketches for ℓ_0-sampling were also used to solve various dynamic graph streaming problems in [AGM12a] and since then have been crucially used in almost all known dynamic graph streaming algorithms[2], such as for: connectivity, k-connectivity, bipartiteness, and minimum spanning tree [AGM12a], subgraph counting, minimum cut, and cut-sparsifier and spanner computation [AGM12b], spectral sparsifiers [AGM13], maximal matching [CCHM15], maximum matching [AGM12a, BS15, Kon15, AKLY16, CCE+16, AKL17], vertex cover [CCHM15, CCE+16], hitting set, b-matching, disjoint paths, k-colorable subgraph, and several other maximum subgraph problems [CCE+16], densest subgraph [BHNT15, MTVV15, EHW16], vertex and hyperedge connectivity [GMT15], and graph degeneracy [FT16]. For an introduction to the power of ℓ_0-sketches in designing dynamic graph stream algorithms, see the recent survey of McGregor [McG14, Section 3]. Such sketches have also been used outside streaming, such as in distributed algorithms [HPP+15, PRS16] and data structures for dynamic connectivity [KKM13, Wan15, GKKT15].

Given the rising importance of ℓ_0-sampling in algorithm design, a clear task is to understand the exact complexity of this problem. The work [JST11] gave an $\Omega(\log^2 n)$-bit space lower bound for data structures solving even the case $M = 1$ which fail with constant probability, and otherwise whose query responses are $(1/3)$-close to uniform in the statistical distance. They also gave an upper bound for $M \leq \text{poly}(n)$ with failure probability δ, which in fact gave $\min\{\|z\|_0, \Theta(\log(1/\delta))\}$ uniform samples from the support of z, using space $O(\log^2 n \log(1/\delta))$

[2]The spectral sparsification algorithm of [KLM+14] is a notable exception.

(here $\|z\|_0$ denotes $|\operatorname{supp}(z)|$). Thus we say their data structure actually solves the harder problem of ℓ_0-sampling$_k$ for $k = \Theta(\log(1/\delta))$ with failure probability δ, where in ℓ_0-sampling$_k$ the goal is to recover $\min\{\|z\|_0, k\}$ uniformly random elements, without replacement, from $\operatorname{supp}(z)$. The upper and lower bounds in [JST11] thus match up to a constant factor for $k = 1$ and δ a constant. We note though in many settings, even if the final application desires constant failure probability, ℓ_0-sampling$_k$ with either failure probability $o(1)$ or $k > 1$ (or both) is needed as a subroutine (see Table 3.1).

For ℓ_p-sampling as defined above, the first work to realize its importance came even earlier than for ℓ_0-sampling: [CK04] showed that an ℓ_2-sampler using small memory would lead to a nearly space-optimal streaming algorithm for multiplicatively estimating $\|x\|_3$ in the turnstile model, but did not know how to implement such a data structure. The first implementation was given in [MW10], where they achieved space $\operatorname{poly}(\varepsilon^{-1}\log n)$ for failure probability $1/\operatorname{poly}(n)$. For $1 \le p \le 2$ the space was improved to $O(\varepsilon^{-p}\log^3 n)$ bits for constant failure probability [AKO11]. In [JST11] this bound was improved to $O(\varepsilon^{-\max\{1,p\}}\log(1/\delta)\log^2 n)$ bits for failure probability δ when $0 < p < 2$ and $p \neq 1$. For $p = 1$ the space bound achieved by [JST11] was a $\log(1/\varepsilon)$ factor worse: $O(\varepsilon^{-1}\log(1/\varepsilon)\log(1/\delta)\log^2 n)$ bits.

Recently, [JW18] showed how to design perfect samplers where $\varepsilon = 0$. For $p \in (0,2)$, they achieve $O(\log(1/\delta)\log^2 n(\log\log n)^2)$ bits; for $p = 2$, they achieve $O(\log(1/\delta)\log^3 n)$ bits. When $\delta = 1/\operatorname{poly}(n)$, they achieve $O(\log^3 n)$ bits. In addition, in the random oracle model, the space can be improved to $O(\log(1/\delta)\log^2 n)$ bits for $p \in (0,2)$.

For finding a duplicate item in a stream, the question of whether a space-efficient randomized algorithm exists was asked in [Mut05, Tar07]. The question was positively resolved in [GR09], which gave an $O(\log^3 n)$-space algorithm with constant failure probability. An improved algorithm was given in [JST11], using $O(\log(1/\delta)\log^2 n)$ bits of space with failure probability δ.

[3] [AGM12a] describe these algorithms as only needing δ a constant, but for a different definition of support-finding: when the data structure fails, it should output Fail instead of behaving arbitrarily. They then cite [JST11] as providing the sampler they use, but unfortunately, [JST11] does not solve this variant of this problem. This issue can be avoided by using [JST11] with $\delta < 1/\operatorname{poly}(n)$ so that with high probability no failures occur throughout their algorithm, very slightly worsening their final space bound in the main application by a $\lg n$ factor. Alternatively, one could use another variant of sampling which we define in Section 3.7 and which can be solved via a minor modification of [JST11], which allows retaining the bounds of [AGM12a] for connectivity, k-connectivity, and bipartiteness testing.

reference	problem	distribution	$k > 1$?	$\delta = o(1)$?
[FIS08]	Euclidean minimum spanning tree	ℓ_0	yes	
[AGM12a]	connectivity[3]	any		yes
[AGM12a]	k-connectivity[3]	any		yes
[AGM12a]	bipartiteness[3]	any		yes
[AGM12a]	minimum spanning tree	any		yes
[AGM12b]	subgraph counting	ℓ_0	yes	
[AGM12b]	minimum cut	any		yes
[AGM12b]	cut sparsifiers	any		yes
[AGM12b]	spanners	any	yes	yes
[AGM12b]	spectral sparsifiers	any		yes
[CCHM15]	maximal matching	ℓ_0	yes	yes
[BS15]	maximum matching (unweighted) maximum matching (weighted)	ℓ_0 ℓ_0	yes yes	 yes
[Kon15]	maximum matching	any	yes	yes
[AKLY16]	maximum matching	ℓ_0		yes
[AKL17]	maximum matching	ℓ_0		yes
[CCE+16]	maximum matching vertex cover hitting set b-matching disjoint paths k-colorable subgraph	ℓ_0		yes
[BHNT15]	densest subgraph	ℓ_0		yes
[MTVV15]	densest subgraph	ℓ_0	yes	yes
[EHW16]	densest subgraph	ℓ_0	yes	
[GMT15]	vertex connectivity hyperedge connectivity	any		yes
[FT16]	graph degeneracy	ℓ_0	yes	

Table 3.1: *Guarantees needed by various works using samplers as subroutines. The last two columns indicate whether the work needs to use a sampler that returns k samples at a time when queried for some $k > 1$, or for some subconstant failure probability δ even to achieve failure probability $1/3$ in the main application. The "distribution" column indicates the output distribution needed from the sampler for the application ("any" means a support-finding subroutine is sufficient, i.e. it suffices for a query to return any index i for which $z_i \neq 0$).*

3.1.3 Summary of our main results

Our main result is the following tight bound of $\mathbf{R}_{\delta}^{\rightarrow,pub}(\mathbf{UR}_k^{\subset})$, where $\mathbf{R}_{\delta}^{\rightarrow,pub}(\mathbf{UR}_k^{\subset})$ refers to the randomized one-way communication problem of $\mathbf{UR}_k^{\subset}$ with failure probability at most δ under public randomness.

Theorem 3.1.4 (Restatement of Theorem 3.2.4)**.** *For any δ bounded away from 1 and $1 \leq k \leq n$, $\mathbf{R}_{\delta}^{\rightarrow,pub}(\mathbf{UR}_k^{\subset}) = \Theta\big(\min\{n, t\log^2(n/t)\}\big)$, where $t = \max\{k, \log(1/\delta)\}$.*

By the following claim, our lower bound on $\mathbf{R}_{\delta}^{\rightarrow,pub}(\mathbf{UR}_k^{\subset})$ implies an optimal space lower bound of $\Omega(t\log^2 n)$ bits for any support-finding$_k$ algorithm (or any ℓ_p-sampling$_k$ algorithm) with failure probability δ, where $t = \max\{k, \log(1/\delta)\} > 2^{-n^{.99}}$. This lower bound matches the upper bound in [JST11].

Claim 3.1.5 (Restatement of Claim 3.2.2)**.** *Any one-pass streaming algorithm for support-finding$_k$ with failure probability at most δ in the strict turnstile model must use $\mathbf{R}_{\delta}^{\rightarrow,pub}(\mathbf{UR}_k^{\subset})$ bits of space, even if promised that $z \in \{0,1\}^n$ at all points in the stream.*

By the following claim, our universal relations lower bound also implies an optimal space lower bound on finding duplicates of $\Omega\big(\log(1/\delta)\log^2 n\big)$ bits for $\delta > 2^{-n^{0.99}}$.

Claim 3.1.6 (Restatement of Claim 3.2.1)**.** *Any one-pass streaming algorithm for $\mathsf{FindDuplicate}(n)$ with failure probability at most δ must use $\mathbf{R}_{\delta}^{\rightarrow,pub}(\mathbf{UR}^{\subset})$ space.*

Moreover, in proving the lower bound of $\mathbf{UR}^{\subset}$, we show that if the updates and queries to a randomized data structure with failure probability δ are not independent of the randomness used in the data structure, then the failure probability can be approximately bounded by δ plus a "loss" which equals to their mutual information divided by $\log(1/\delta)$. This bound is essentially tight and is of independent interest in dynamic data structure designs.

Lemma 3.1.7 (Restatement of Lemma 3.3.1)**.** *Consider $f\colon \{0,1\}^b \times \{0,1\}^q \to \{0,1\}$ and $X \in \{0,1\}^b$ uniformly random. If $\forall y \in \{0,1\}^q$, $\mathbb{P}(f(X,y)=1) \leq \delta$ where $0 < \delta < 1$, then for any random variable Y supported on $\{0,1\}^q$,*

$$\mathbb{P}(f(X,Y)=1) \leq \frac{I(X;Y) + H_2(\delta)}{\log(1/\delta)},$$

level	sub-sample probability	sub-sampled items	sub-sampled stream
0	1	1 2 3 4 5 6 7 8	2 3 5 6 7 −2 −5
1	1/2	2 3 6 8	2 3 6 −2
2	1/4	3 5	3 5 −5
3	1/8	5	5 −5

Figure 3.2: *An example of the sub-sampling procedure used by samplers. The universe size $n = 8$. We use the notation of $-i$ to denote the frequency of i is decreased by 1. The original turnstile stream is the same as the sub-sampled stream in level 0.*

where $I(X;Y)$ is the mutual information between X and Y, and H_2 is the binary entropy function.

3.1.4 Our approach

Historically, we first considered and proved space lower bounds for samplers, and then adapted the proof to $\mathbf{R}_{\delta}^{\rightarrow,pub}(\mathbf{UR}_k^{\subset})$ lower bound. In the following, we discuss some insight on how we proved lower bounds on support-finding with failure probability δ and support-finding$_k$ with constant failure probability respectively.

Support-finding with failure probability δ. We first recall the upper bound for support-finding and ℓ_0-sampler. The data structure in [JST11] maintains $1 + \log n$ levels, where level $\ell \in \{0, \ldots, \log n\}$ sub-samples each element in the universe $[n]$ with probability $2^{-\ell}$. Figure 3.2 illustrates an example of this sub-sampling procedure. Each level maintains a sparse recovery data structure for those sub-sampled items, with the guarantee that if the sparsity is $O\big(\log(1/\delta)\big)$, then the data structure outputs an element in the support with probability 1 (actually in that case the whole support is discovered), otherwise with high probability it outputs "dense". The data structure on each level takes $O\big(\log(1/\delta)\log n\big)$ bits of space, so the overall space is $O\big(\log(1/\delta)\log^2 n\big)$ bits.

Let $x \in \mathbb{Z}^n$ denote the vector at the end of the stream. If the size of $\mathrm{supp}(x)$ is as small as $O\big(\log(1/\delta)\big)$, then a sparse recovery data structure of $O\big(\log(1/\delta)\log n\big)$ bits can be used to get $\mathrm{supp}(x)$ exactly. The purpose of $\log n$ levels in the above approach is that if $|\mathrm{supp}(x)|$ is large, down-sampling in each level in expectation shrinks the size of the support by a half, and finally

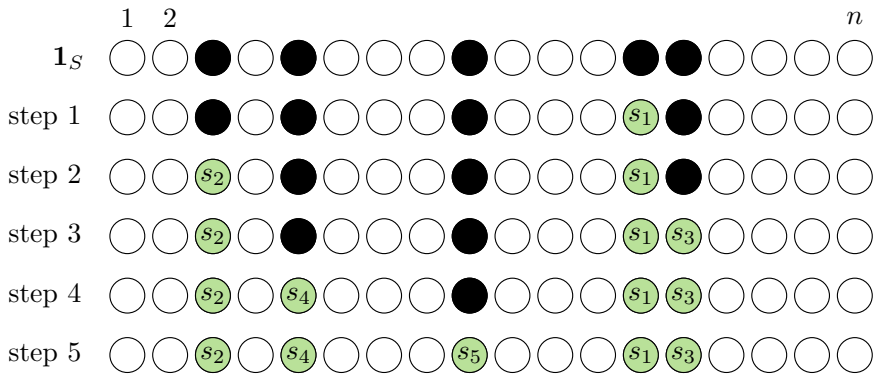


Figure 3.3: *Decoding procedure for a sampler that encodes a uniform random set S. In this figure, each row represents the underlying frequency vector of the sampler that we are going to query. The circles in black (●) represent 1 and otherwise (either ○ or ●) represent 0. In particular, green balls (●) represent elements in S that are recovered by the sampler. In this example, $\delta = 2^{-10}$ and $|S| = \frac{1}{2}\log \delta^{-1} = 5$. The decoding procedure proceeds in $|S| = 5$ steps, where in step i, $s_i \in S$ is recovered by the sampler and then s_i is removed from the sampler to facilitate recovering the rest of elements in S.*

with probability at least δ there is a level with support size between 1 and $\log(1/\delta)$.

In order to see whether it is possible to improve the space complexity of the previous approach, two natural questions arise. Given the promise that the sparsity (i.e. $|\operatorname{supp}(x)|$) is $O\big(\log(1/\delta)\big)$, is $O\big(\log(1/\delta)\log n\big)$ bits optimal for a support-finding data structure with failure probability δ? Are $\log n$ levels really necessary? We give affirmative answers to both questions. To ease our discussion in this section we assume that $\delta > 2^{-n^{.99}}$ so that $\log(\frac{n}{\log \delta^{-1}}) = \Theta\big(\log(1/\delta)\log n\big)$.

For the first question, let us assume that there is a data structure $\mathcal{DS}$ that takes a turnstile stream as input and outputs an element in $\operatorname{supp}(x)$ with probability $1-\delta$ as long as $|\operatorname{supp}(x)| \leq \log(1/\delta)$. Then the data structure $\mathcal{DS}$ can be used to encode a random subset of $[n]$ of size $\frac{1}{2}\log(1/\delta)$ and succeeds with constant probability as follows. Let S be a random subset of $[n]$ and $|S| = \log(1/\delta)$. As an encoder we run an instance of $\mathcal{DS}$ and feed it with a stream of updates $x_i \leftarrow x_i + 1$ for $i \in S$, then the memory footprint of $\mathcal{DS}$, denoted by $\mathcal{DS}(S)$, is an encoding for S. The decoder runs in $\frac{1}{2}\log(1/\delta)$ rounds, initially let $T \leftarrow \emptyset$, and in each round the decoder first makes a copy of $\mathcal{DS}(S)$, makes updates $x_i \leftarrow x_i - 1$ for $i \in T$ (from the smallest to largest elements in T) to the copy, and then makes a query to (hopefully) obtain $s \in S \backslash T$, and finally updates $T \leftarrow T \cup \{s\}$. We illustrate this decoding procedure in Figure 3.3. Now we analyze the failure probability. Let $E_{T'}$ where $T' \subsetneq S$ denote the event that the query to $\mathcal{DS}(S)$ with

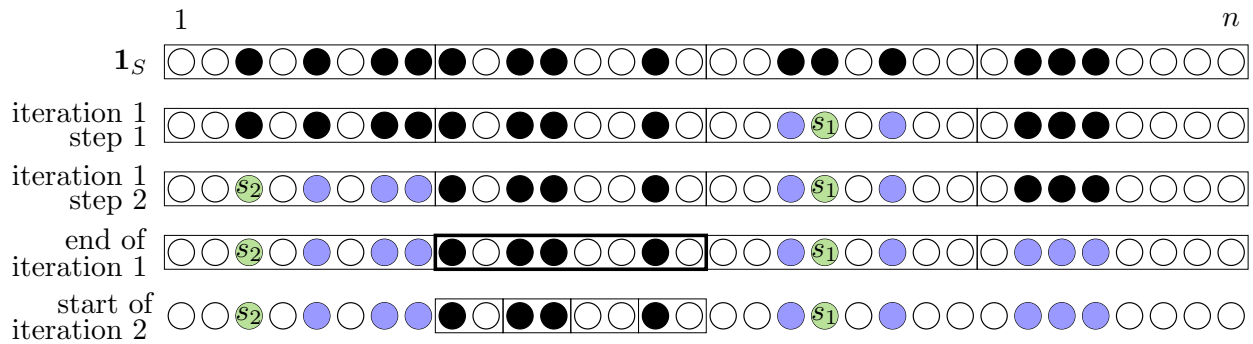


Figure 3.4: *Illustration of an encoding/decoding procedure for a uniform random set S ($|S| = n^{0.999}$) using a sampler, which gives a nearly optimal sampler lower bound. In this figure, each row represents the underlying frequency vector of the sampler that we are going to query. The circles in black (●) represent 1 and otherwise (○, ● or ●) represent 0. In particular, green balls (●) represent elements in S that are recovered by the sampler; blue balls (●) represent elements in S that the sampler does not need to recover (these elements will be included in the set A, and finally A, the memory footprint of the sample, together with some negligible information, will be the encoding of S). In the instance illustrated by the figure, $D = 4$ (in general $D = \Theta\big(\log(1/\delta)\log n\big)$) and in the first iteration the universe $[n]$ is evenly divided into D intervals $I_1, \ldots, I_D$ (see the rectangles on the first row, where the black circles in the j-th rectangle correspond to $S \cap I_j$). In the first iteration, the procedure proceeds in 2 (in general $\Theta(\frac{\log(1/\delta)}{\log D})$) steps: in step 1 the sampler finds $s_1 \in S \cap I_3$ and then $S \cap I_3$ are removed from the sampler, in step 2 the sampler finds $s_2 \in S \cap I_1$ and then $S \cap I_1$ are removed from the sampler. At the end of the first iteration, the procedure samples a uniform index $i \in [D]$, and in this instance fortunately $i = 2 \notin \{1, 3\}$ (so the procedure will not report "fail"), $S \cap ([n] \setminus I_i)$ will be removed from the sampler. Then the procedure will start the second iteration (which recurses on $S \cap I_i$). In total, the procedure will have $H = \Theta(\frac{\log n}{\log D})$ iterations.*

further updates $x_i \leftarrow x_i - 1$ for $i \in T'$ gives an element in $S \setminus T'$. By a union bound, we have $\mathbb{P}[T \neq S] \leq \mathbb{P}[\bigcup_{T' \subsetneq S} \overline{E_{T'}}] \leq \delta 2^{\frac{1}{2}\log(1/\delta)} = \sqrt{\delta}$. By source coding theorem, the space of $\mathcal{DS}$ is at least $\Omega\big(\binom{n}{\frac{1}{2}\log(1/\delta)}\big) = \Omega\big(\log(1/\delta)\log(\frac{n}{\log(1/\delta)})\big) = \Omega\big(\log(1/\delta)\log n\big)$ bits.

For the second question, it is possible to prove that, intuitively, we can extract $\Theta(\log(1/\delta)\log n)$ bits of extra information out of the sampler from each of $\Theta(\log n)$ levels. But before presenting the optimal lower bound, it is worth presenting a lower bound that is $(\log\log n + \log\log(1/\delta))^2$ worse than the optimal $\log(1/\delta)\log^2 n$, but introduces three core ideas and is a precursor of our optimal lower bound. First, instead of trying to encode a random set S only by $\mathcal{DS}(S)$, the encoder will identify $A \subseteq S$ containing elements that cannot be decoded by $\mathcal{DS}(S)$, and use both A and $\mathcal{DS}(S)$ (and maybe some negligible amount of auxiliary information) as the encoding of S. Our ultimate goal is to have in expectation $|S| - |A| = \Omega\big(\log(1/\delta)\log n\big)$, because in that case $\mathcal{DS}(S)$ must contain $\Omega\big(\log(1/\delta)\log n\big)$ words of information. Second, let $|S| = n^{0.999}$

and let $D = \Theta\big(\log(1/\delta)\log n\big)$, we divide the universe $[n]$ into a D-ary tree, and its height is $H = \Theta(\frac{\log n}{\log\log n + \log\log(1/\delta)})$. Let $I_1, \ldots, I_D$ correspond to the first layer of the tree (the root corresponds to $[n]$ and is located on layer 0). Note that $I_1, \ldots, I_D$ is a partition of $[n]$. In the very beginning $A \leftarrow S$. Both encoding and decoding scheme make query to the sampler in order to (hopefully) get $s \in S$, and let $A \leftarrow A \backslash \{s\}$ (the encoding/decoding scheme also needs an auxiliary bit recording whether $s \in S$, but it is negligible compared to $\Theta(\log n)$ bits saved by identifying an element in S). After that, we find j so that $s \in I_j$, and remove all the elements in $S \cap I_j$ from the sampler. We call this technique "remove in blocks", and it enables us to make $\Theta(\frac{\log(1/\delta)}{\log D}) = \Theta(\frac{\log(1/\delta)}{\log\log n + \log\log(1/\delta)})$ queries with good success probability. Third, after obtaining $\Theta(\frac{\log(1/\delta)}{\log\log n + \log\log(1/\delta)})$ samples, we can uniformly at random choose $i \in [D]$, and recursively try to recover elements in $S \cap I_i$. If the interval I_i has been removed because of the answer to queries, then the encoding/decoding scheme reports fail. The probability that I_i gets removed is $\Theta(\frac{\log(1/\delta)}{D\log D}) < \frac{1}{\log n}$, which affords a union bound over $O(H)$ layers. We recurse the process on I_i in the same way until we dive into layer $\Theta(\frac{\log n}{\log\log n + \log\log(1/\delta)})$ of the D-ary tree, and overall we have recovered $\Theta(\frac{\log(1/\delta)\log n}{(\log\log n + \log\log(1/\delta))^2})$ elements in S from the sampler $\mathcal{DS}(S)$. We illustrate the encoding/decoding procedure in Figure 3.4.

In order to shave these two $\log\log n + \log\log(1/\delta)$ factors, each time after we obtain a sample from the sampler (initially with elements of S inserted), instead of "removing in blocks", we remove some random elements in S so that future queries still succeed with good probability. Intuitively, we are trying to hide the samples returned by the sampler by injecting some noise, in the hope that the resulting set (to be removed from the sampler) has small mutual information with the internal randomness of the sampler. In fact, we can prove that by setting $|S| = \sqrt{n\log(1/\delta)}$ and each iteration we shrink the size of S by $2^{-\frac{16}{\log(1/\delta)}}$ (remove the sample returned by the sampler and then remove some uniform random elements), then in each iteration the mutual information between S and the internal randomness of the sampler is bounded by $\frac{3}{8}\log(1/\delta)$. The final piece of the puzzle is a lemma proved in this chapter bounding the failure probability of a data structure in terms of the mutual information between updates to the data structure and the internal randomness of the data structure. By the lemma, the query succeeds with constant

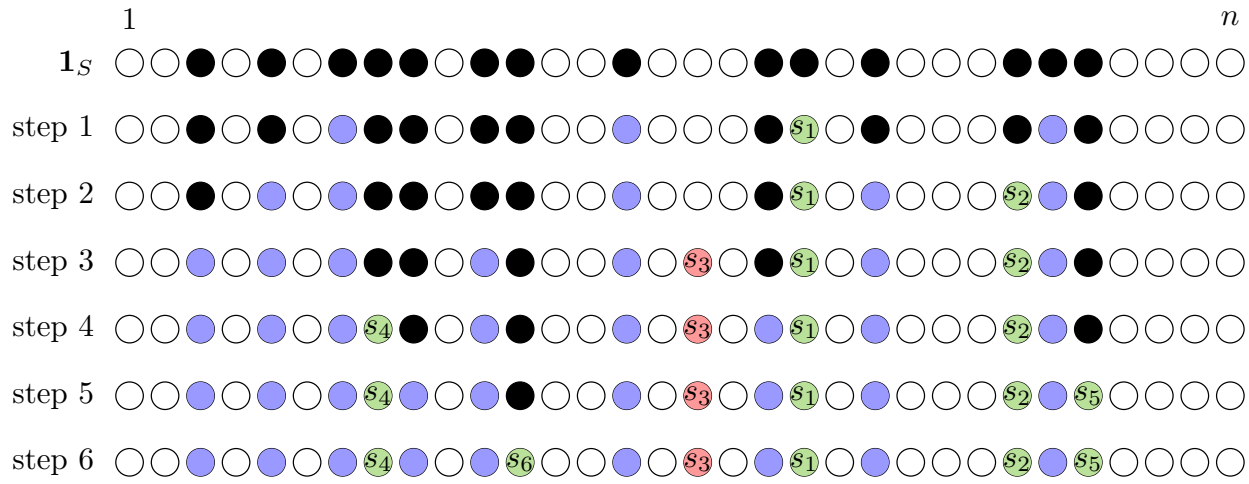


Figure 3.5: *Illustration of an encoding/decoding procedure for a uniform random set S ($|S| = \sqrt{n \log(1/\delta)}$) using a sampler, which gives an optimal sampler lower bound. In this figure, each row represents the underlying frequency vector of the sampler that we are going to query. The circles in black (●) represent 1 and otherwise (○, ●, ● or ●) represent 0. In particular, green balls (●) represent elements in S that are recovered by the sampler; blue balls (●) represent elements in S that the sampler does not need to recover; red balls (●) represent the wrong samples (i.e. not in S) given by the sampler. In step 3 of this execution instance, the sampler outputs $s_3 \notin S$. This would not be a trouble, because we can afford 1 bit of information for each round recording whether the sampler works correctly.*

probability. Altogether we have $\Theta\big(\log(1/\delta)\log(\frac{n}{\log(1/\delta)})\big)$ iterations and the sample obtained from each iteration contains $\Theta\big(\log(\frac{n}{\log(1/\delta)})\big)$ bits of information. Therefore the space lower bound of the sampler/support-finding problem is $\Omega\big(\log(1/\delta)\log^2(\frac{n}{\log(1/\delta)})\big)$ bits. When $\delta > 2^{-n^{.99}}$ the bound becomes $\Omega\big(\log(1/\delta)\log^2 n\big)$.

support-finding$_k$ with constant failure probability. The proof is a similar but simpler encoding argument. Let S be a random subset of $[n]$ and $|S| = \sqrt{nk}$. As before, initially all the elements in S are inserted into the sampler, and that memory footprint is part of the encoding. The encoder/decoder proceed in $\Theta(\log \frac{n}{k})$ rounds, and each round they randomly partition the universe into two even halves where they use the first half intersect with the k samples output by the sampler as elements identified by the sampler (and then remove S intersect with the first half from the sampler), and use the second half as the new universe for the next round. The expected number of elements identified by the sampler is $\Theta(k \log \frac{n}{k})$, and each identified element contains $\Theta(\log \frac{n}{k})$ bits of information. Thus the space lower bound is $\Omega(k \log^2 \frac{n}{k})$ bits.

3.2 Preliminaries and Our Results

The work [JST11] gave an $\Omega(\log^2 n)$-bit space lower bound for data structures solving the case $M = 1$ which fail with constant probability, and otherwise whose query responses are $(1/3)$-close to uniform in statistical distance. They also gave an upper bound for $M \leq \text{poly}(n)$ with failure probability δ, which in fact gave $\min\{\|z\|_0, \Theta(\log(1/\delta))\}$ uniform samples from the support of z, using space $O(\log^2 n \log(1/\delta))$ bits (here $\|z\|_0$ denotes $|\text{supp}(z)|$). Thus we say their data structure actually solves the harder problem of ℓ_0-sampling$_{\Theta(\log(1/\delta))}$ with failure probability δ, where in ℓ_0-sampling$_k$ the goal is to recover $\min\{\|z\|_0, k\}$ uniformly random elements, without replacement, from $\text{supp}(z)$. The upper and lower bounds in [JST11] thus match up to a constant factor for $k = 1$ and δ a constant.

Universal relation. The work of [JST11] obtains its lower bound for ℓ_0-sampling (and some other problems) via reductions from *universal relation* (**UR**). The problem **UR** was first defined in [KRW95] and arose in connection with work of Karchmer and Wigderson on circuit depth lower bounds [KW90]. For $f : \{0,1\}^n \to \{0,1\}$, $D(f)$ is the minimum depth of a fan-in 2 circuit over the basis $\{\neg, \vee, \wedge\}$ computing f. Meanwhile, the (deterministic) communication complexity $C(f)$ is defined as the minimum number of bits that need to be communicated in a correct protocol for Alice and Bob to solve the following communication problem: Alice receives $x \in f^{-1}(0)$ and Bob receives $y \in f^{-1}(1)$ (and hence in particular $x \neq y$), and they must both agree on an index $i \in [n]$ such that $x_i \neq y_i$. It is shown in [KW90] that $D(f) = C(f)$, where they then used this correspondence to show a tight $\Omega(\log^2 n)$ depth lower bound on monotone circuits solving undirected s-t connectivity. The work of [KRW95] then proposed a strategy to separate the complexity classes $\mathbf{NC}^1$ and $\mathbf{P}$: start with a function f on $\log n$ bits requiring depth $\Omega(\log n)$, then "compose" it with itself $k = \log n / \log\log n$ times (see [KW90] for a precise definition of composition). If one could prove a strong enough direct sum theorem for communication complexity after composition, such a k-fold composition would yield a function that is provably in $\mathbf{P}$ (and in fact, even in $\mathbf{NC}^2$), but not in $\mathbf{NC}^1$. Proving such a direct sum theorem is still wide open, and the statement that it is true is known as the "KRW conjecture"; see for example the

recent works [GMWW14, DM16] toward resolving this conjecture. As a toy problem en route to resolving it, [KRW95] suggested proving a direct sum theorem for k-fold composition of a particular function $\mathbf{UR}$ that they defined; that task was positively resolved in [EIRS91] (see also [HW90]).

The problem $\mathbf{UR}$ abstracts away the function f and requires Alice and Bob to agree on the index i only knowing that $x, y \in \{0,1\}^n$ are unequal. The deterministic communication complexity of $\mathbf{UR}$ is nearly completely understood, with upper and lower bounds that match up to an additive 3 bits, even if one requires an upper bound on the number of rounds [TZ97]. Henceforth we also consider a generalized problem $\mathbf{UR}_k$, where the output must be $\min\{k, \|x-y\|_0\}$ distinct indices on which x, y differ. We also use $\mathbf{UR}^{\subset}, \mathbf{UR}^{\subset}_k$ to denote the variants when promised $\mathrm{supp}(y) \subset \mathrm{supp}(x)$, and also Bob knows $\|x\|_0$. Clearly $\mathbf{UR}, \mathbf{UR}_k$ can only be harder than $\mathbf{UR}^{\subset}, \mathbf{UR}^{\subset}_k$, respectively.

More than twenty years after its initial introduction in connection with circuit depth lower bounds, Jowhari et al. in [JST11] demonstrated the relevance of $\mathbf{UR}$ in the randomized one-way communication model for obtaining space lower bounds for certain streaming problems, such as various sampling problems and finding duplicates in streams. In particular, if $\mathbf{R}^{\rightarrow,pub}_{\delta}(f)$ denotes the randomized one-way communication complexity of f in the public coin model with failure probability δ, [JST11] showed that the space complexity of $\mathsf{FindDuplicate}(n)$ with failure probability δ is at least $\mathbf{R}^{\rightarrow,pub}_{\frac{7}{8}+\frac{\delta}{8}}(\mathbf{UR})$. In $\mathsf{FindDuplicate}(n)$, one is given a length-$(n+1)$ stream of integers in $[n]$, and the algorithm must output some element $i \in [n]$ which appeared at least twice in the stream (note that at least one such element must exist, by the pigeonhole principle). The work [JST11] then showed a reduction demonstrating that any solution to ℓ_0-sampling with failure probability δ in turnstile streams immediately implies a solution to $\mathsf{FindDuplicate}(n)$ with failure probability at most $(1+\delta)/2$ in the same space (and thus the space must be at least $\mathbf{R}^{\rightarrow,pub}_{\frac{15}{16}+\frac{\delta}{16}}(\mathbf{UR})$). The same result is shown for ℓ_p-sampling for any $p > 0$, in which the output index should equal i with probability $|x_i|^p/(\sum_j |x_j|^p)$, and a similar result is shown even if the distribution on i only has to be close to this ℓ_p-distribution in variational distance (namely, the distance should be bounded away from 1). It is then shown in [JST11] that $\mathbf{R}^{\rightarrow,pub}_{\delta}(\mathbf{UR}) = \Omega(\log^2 n)$ for any δ bounded away from 1. The approach used though unfortunately does not provide an improved lower bound for

$\delta \downarrow 0$.

Seemingly unnoticed in [JST11], we first point out here that the lower bound proof for **UR** in that work actually proves the same lower bound for the promise problem $\mathbf{UR}^{\subset}$. This observation has several advantages. First, it makes the reductions to the streaming problems trivial (they were already quite simple when reducing from **UR**, but now they are even simpler). Second, a simple reduction from $\mathbf{UR}^{\subset}$ to sampling problems provides space lower bounds even in the strict turnstile model, and even for the simpler *support-finding* streaming problem for which when queried is allowed to return *any* element of $\mathrm{supp}(z)$, without any requirement on the distribution of the index output. Both of these differences are important for the meaningfulness of the lower bound. This is because in dynamic graph streaming applications, typically z is indexed by $\binom{n}{2}$ for some graph on n vertices, and z_e is the number of copies of edge e in some underlying multigraph. Edges then are never deleted unless they had previously been inserted, thus only requiring sampler subroutines that are correct with the strict turnstile promise. Also, for every single application mentioned in the first paragraph of Section 3.2 (except for the two applications in [FIS08]), the known algorithmic solutions which we cited as using ℓ_0-sampling as a subroutine actually only need a subroutine for the easier support-finding problem. Finally, third and most relevant to our current work's main focus, the straightforward reductions from $\mathbf{UR}^{\subset}$ to the streaming problems we are considering here do not suffer any increase in failure probability, allowing us to transfer lower bounds on $\mathbf{R}_{\delta}^{\rightarrow,pub}(\mathbf{UR}^{\subset})$ for small δ to lower bounds on various streaming problems for small δ. The work [JST11] could not provide lower bounds for the streaming problems considered there in terms of δ for small δ.

We now show simple reductions from $\mathbf{UR}^{\subset}$ to $\mathsf{FindDuplicate}(n)$ and from $\mathbf{UR}_k^{\subset}$ to support-finding$_k$. In support-finding$_k$ we must report $\min\{k, \|z\|_0\}$ elements in $\mathrm{supp}(z)$. In the claims below, δ is the failure probability for the considered streaming problem.

Claim 3.2.1. *Any one-pass streaming algorithm for* $\mathsf{FindDuplicate}(n)$ *must use* $\mathbf{R}_{\delta}^{\rightarrow,pub}(\mathbf{UR}^{\subset})$ *space.*

Proof. We reduce from $\mathbf{UR}^{\subset}$. Suppose there were a space-S algorithm $\mathcal{A}$ for $\mathsf{FindDuplicate}(n)$. Alice creates a stream consisting of all elements of $\mathrm{supp}(x)$ and runs $\mathcal{A}$ on those elements, then sends the memory contents of $\mathcal{A}$ to Bob. Bob then continues running $\mathcal{A}$ on $n+1-\|x\|_0$ arbitrarily

chosen elements of $[n] \setminus \mathrm{supp}(y)$. Then there must be a duplicate in the resulting concatenated stream, and all duplicates i satisfy $x_i \neq y_i$. □

Claim 3.2.2. *Any one-pass streaming algorithm for support-finding$_k$ in the strict turnstile model must use $\mathbf{R}_\delta^{\rightarrow,pub}(\mathbf{UR}_k^{\subset})$ bits of space, even if promised that $z \in \{0,1\}^n$ at all points in the stream.*

Proof. This is again via reduction from $\mathbf{UR}_k^{\subset}$. Let $\mathcal{A}$ be a space-S algorithm for support-finding$_k$ in the strict turnstile model. For each $i \in \mathrm{supp}(x)$, Alice sends the update $z_i \leftarrow z_i + 1$ to $\mathcal{A}$. Alice then sends the memory contents of $\mathcal{A}$ to Bob. Bob then for each $i \in \mathrm{supp}(y)$ sends the update $z_i \leftarrow z_i - 1$ to $\mathcal{A}$. Now note that z is exactly the indicator vector of the set $\{i : x_i \neq y_i\}$. □

Claim 3.2.3. *Any one-pass streaming algorithm for ℓ_p-sampling for any $p \geq 0$ in the strict turnstile model must use $\mathbf{R}_\delta^{\rightarrow,pub}(\mathbf{UR}_k^{\subset})$ bits of space, even if promised $z \in \{0,1\}^n$ at all points in the stream.*

Proof. This is via straightforward reduction from support-finding$_k$, since reporting $\min\{k, \|z\|_0\}$ elements of $\mathrm{supp}(z)$ satisfying some distributional requirements is only a harder problem than finding *any* $\min\{k, \|z\|_0\}$ elements of $\mathrm{supp}(z)$. □

The reductions above thus raise the question: what is the asymptotic behavior of $\mathbf{R}_\delta^{\rightarrow,pub}(\mathbf{UR}_k^{\subset})$?

Our main contribution: We prove for any δ bounded away from 1 and $k \in [n]$, $\mathbf{R}_\delta^{\rightarrow,pub}(\mathbf{UR}_k^{\subset}) = \Theta(\min\{n, t\log^2(n/t)\})$ where $t = \max\{k, \log(1/\delta)\}$. Given known upper bounds in [JST11], our lower bounds are optimal for $\mathsf{FindDuplicate}(n)$, support-finding, and ℓ_p-sampling for any $0 \leq p < 2$ for nearly the full range of n, δ (namely, for $\delta > 2^{-n^{.99}}$). Also given an upper bound of [JST11], our lower bound is optimal for ℓ_0-sampling$_k$ for nearly the full range of parameters n, k, δ (namely, for $t < n^{.99}$). Previously no lower bounds were known in terms of δ (or k). Our main theorem:

Theorem 3.2.4. *For any δ bounded away from 1 and $1 \leq k \leq n$, $\mathbf{R}_\delta^{\rightarrow,pub}(\mathbf{UR}_k^{\subset}) = \Theta(\min\{n, t\log^2(n/t)\})$.*

Our upper bound is also new, though follows by minor modifications of the upper bound in [JST11] and thus we describe it in the appendix. The previous upper bound was $O(\min\{n, t\log^2 n\})$. We also mention here that it is known that the upper bound for both $\mathbf{UR}_k$ and ℓ_0-sampling$_k$ in

two rounds (respectively, two passes) is only $O(t \log n)$ [JST11]. Thus, one cannot hope to extend our new lower bound to two or more passes, since it simply is not true.

3.3 Overview of techniques

We describe our proof of Theorem 3.2.4. For the upper bound, [JST11] achieved $O(t \log^2 n)$, but in the appendix we show that slight modifications to their approach yield $O(\min\{n, t \log^2(n/t)\})$ bits. Our main contribution is in proving an improved lower bound. Assume $t < cn$ for some sufficiently small constant c (since otherwise we already obtain an $\Omega(n)$ lower bound). Our lower bound proof in this regime is split into two parts: we show $\mathbf{R}_{\delta}^{\rightarrow,pub}(\mathbf{UR}^{\subset}) = \Omega(\log \frac{1}{\delta} \log^2 \frac{n}{\log \frac{1}{\delta}})$ and $\mathbf{R}_{\frac{1}{2}}^{\rightarrow,pub}(\mathbf{UR}_k^{\subset}) = \Omega(k \log^2 \frac{n}{k})$ separately. We give an overview the former here, which is the more technically challenging half. Our proof of the latter can be found in Section 3.5.

We prove the lower bound via an encoding argument. Fix m. A randomized encoder is given a set $S \subset [n]$ with $|S| = m$ and must output an encoding $\mathsf{ENC}(S)$, and a decoder sharing public randomness with the encoder must be able to recover S given only $\mathsf{ENC}(S)$. We consider such schemes in which the decoder must succeed with probability 1, and the encoding length is a random variable. Any such encoding must use $\Omega(\log \binom{n}{m}) = \Omega(m \log \frac{n}{m})$ bits in expectation for some S.

There is a natural, but sub-optimal approach to using a public-coin one-way protocol $\mathcal{P}$ for $\mathbf{UR}^{\subset}$ to devise such an encoding/decoding scheme. The encoder pretends to be Alice with input x being the indicator set of S, then lets $\mathsf{ENC}(S)$ be the message M Alice would have sent to Bob. The decoder attempts to recover S by iteratively pretending to be Bob m times, initially pretending to have input $y = 0 \in \{0,1\}^n$, then iteratively adding elements found in S to y's support. Henceforth let $\mathbf{1}_T \in \{0,1\}^n$ denote the indicator vector of a set $T \subset [n]$.

One might hope to say that if the original failure probability were $\delta < 1/m$, then by a union bound, with constant probability every iteration succeeds in finding a new element of S (or one could even first apply some error-correction to x so that the decoder could recover S even if only a constant fraction of iterations succeeded). The problem with such thinking though is that this

Algorithm 3 Simple Decoder.

```
procedure DEC(M)
    T ← ∅
    for r = 1, ..., m do
        Let i be Bob's output upon receiving message M from Alice when Bob's input is 1_T
        T ← T ∪ {i}
    end for
    return T
end procedure
```

decoder chooses y's adaptively! To be specific, $\mathcal{P}$ being a correct protocol means

$$\forall x, y \in \{0,1\}^n, \ \mathop{\mathbb{P}}_{s}(\mathcal{P} \text{ is correct on inputs } x, y) \geq 1 - \delta, \tag{3.1}$$

where s is the public random string that both Alice and Bob have access to. The issue is that even in the second iteration (when $r = 2$), Bob's "input" $\mathbf{1}_T$ *depends on* s, since T depends on the outcome of the first iteration! Thus the guarantee of (3.1) does not apply.

One way around the above issue is to realize that as long as every iteration succeeds, T is always a subset of S. Thus it suffices for the following event $\mathcal{E}$ to occur: $\forall T \subset S$, $\mathcal{P}$ is correct on inputs $\mathbf{1}_S, \mathbf{1}_T$. Then $\mathbb{P}_s(\neg\mathcal{E}) \leq 2^m\delta$ by a union bound, which is at most $1/2$ for $m = \lfloor \log_2(1/\delta) \rfloor - 1$. We have thus just shown that $\mathbf{R}_{\delta}^{\rightarrow,pub}(\mathbf{UR}^{\subset}) = \Omega(\min\{n, \log(\binom{n}{m})\}) = \Omega(\min\{n, \log\frac{1}{\delta}\log\frac{n}{\log(1/\delta)}\})$.

Our improvement is as follows. Our new decoder again iteratively tries to recover elements of S as before. We will give up though on having m iterations and hoping for all (or even most) of them to succeed. Instead, we will only have $R = \Theta(\log\frac{1}{\delta}\log\frac{n}{\log\frac{1}{\delta}})$ iterations, and our aim is for the decoder to succeed in finding a new element in S for at least a constant fraction of these R iterations. Simplifying things for a moment, let us pretend for now that all R iterations do succeed in finding a new element. $\mathsf{ENC}(S)$ will then be Alice's message M, together with the set $B \subset S$ of size $m - R$ not recovered during the R rounds, explicitly written using $\lceil \log(\binom{n}{|B|}) \rceil$ bits. If the decoder can then recover these R remaining elements, this then implies the decoder has recovered S, and thus we must have $|M| = \Omega(\log(\binom{n}{m}) - \log(\binom{n}{|B|})) = \Omega(R\log\frac{n}{m})$. The decoder proceeds as follows. Just as before, initially the decoder starts with $T = \emptyset$ and lets i be the output of Bob on $\mathbf{1}_T$ and adds it to T. Then in iteration r, before proceeding to the next iteration, the decoder

randomly picks some elements from B and adds them into T, so that the number of elements left to be uncovered is some fixed number n_r. These extra elements being added to T should be viewed as "random noise" to mask information about the random string s used by $\mathcal{P}$, an idea very loosely inspired by ideas in differential privacy. For intuition, as an example suppose the iteration $r = 1$ succeeds in finding some $i \in S$. If the decoder were then to add i to T, as well as $\approx m/2$ random elements from B to T, then the resulting T reveals only ≈ 1 bit of information about i (and hence about s). This is as opposed to the $\log n$ bits T would have revealed if the masking were not performed. Thus the next query in round $r = 2$, although correlated with s, has very weak correlation after masking and we thus might hope for it to succeed. This intuition is captured in the following lemma, which we prove in Section 3.4:

Lemma 3.3.1. *Consider $f\colon \{0,1\}^b \times \{0,1\}^q \to \{0,1\}$ and $X \in \{0,1\}^b$ uniformly random. If $\forall y \in \{0,1\}^q$, $\mathbb{P}(f(X,y) = 1) \leq \delta$ where $0 < \delta < 1$, then for any random variable Y supported on $\{0,1\}^q$,*

$$\mathbb{P}(f(X,Y) = 1) \leq \frac{I(X;Y) + H_2(\delta)}{\log(1/\delta)}, \tag{3.2}$$

where $I(X;Y)$ is the mutual information between X and Y, and H_2 is the binary entropy function.

Fix some $x \in \{0,1\}^n$. One should imagine here that $f(X,y)$ is 1 iff $\mathcal{P}$ fails when Alice has input x and Bob has input y in a $\mathbf{UR}^{\subset}$ instance, and the public random string is $X = s$. Then the lemma states that if $y = Y$ is not arbitrary, but rather random (and correlated with X), then the failure probability of the protocol is still bounded as long as the mutual information between X and Y is bounded. It is also not hard to see that this lemma is sharp up to small additive terms. Consider the case $x, y \in [n]$, and $f(x,y) = 1$ iff $x = y$. Then if X is uniform, for all y we have $\mathbb{P}(f(X,y) = 1) = 1/n$. Now consider the case where Y is random and equal to X with probability $t/\log n$ and is uniform in $[n]$ with probability $1 - t/\log n$. Then in expectation Y reveals t bits of X, so that $I(X;Y) = t$. It is also not hard to see that $\mathbb{P}(f(X,Y) = 1) \approx t/\log n + 1/n$.

In light of the strategy stated so far and Lemma 3.3.1, the path forward is clear: at each iteration r, we should add enough random masking elements to T to keep the mutual information between T and all previously added elements below, say, $\frac{1}{2}\log(1/\delta)$. Then we expect a constant fraction of iterations to succeed. The encoder knows which iterations do not succeed since it shares public

randomness with the decoder (and can thus simulate it), so it can simply tell the decoder which rounds are the failed ones, then explicitly include in M correct new elements of S for the decoder to use in the place of Bob's wrong output in those rounds. A calculation shows that if one adds a $(1-1/K) \approx 2^{-1/K}$ fraction of the remaining items in S to T after drawing one more support element from Bob, the mutual information between the next query to Bob and the randomness used by $\mathcal{P}$ will be $O(K)$ (see Lemma 3.4.4). Thus we do this for K a sufficiently small constant times $\log(1/\delta)$. We will then have $n_r \approx (1-1/K)^r m$. Note that we cannot continue in this way once $n_r < K$ (since the number of "random noise" elements we inject should at least be one). Thus we are forced to stop after $R = \Theta(K \log(m/K)) = \Theta\big(\log(1/\delta)\log(\frac{n}{\log(1/\delta)})\big)$ iterations. We then set $m = \sqrt{n\log(1/\delta)}$, so that $\mathbf{R}_\delta^{\rightarrow,pub}(\mathbf{UR}^{\subset}) = \Omega(|R|\log\frac{n}{m}) = \Omega\big(\min\{n, \log(1/\delta)\log^2(\frac{n}{\log(1/\delta)})\}\big)$ as desired.

The argument for lower bounding $\mathbf{R}_\delta^{\rightarrow,pub}(\mathbf{UR}_k^{\subset})$ is a bit simpler, and in particular does not need rely on Lemma 3.3.1. Both the idea and rigorous argument can be found in Section 3.5, but again the idea is to use a protocol for this problem to encode appropriately sized subsets of $[n]$.

As mentioned above, our lower bounds use protocols for $\mathbf{UR}^{\subset}$ and $\mathbf{UR}_k^{\subset}$ to establish protocols for encoding subsets of $[n]$ of some fixed size m. These encoders always consist of some message M Alice would have sent in a $\mathbf{UR}^{\subset}$ or $\mathbf{UR}_k^{\subset}$ protocol, together with a random subset $B \subset S$ (using $\lceil\log_2|B|\rceil + \lceil\log\binom{n}{|B|}\rceil$ bits, to represent both $|B|$ and the set B itself). Here $|B|$ is a random variable. These encoders are thus *Las Vegas*: the length of the encoding is a random variable, but the encoder/decoder always succeed in compressing and recovering the subset. The final lower bounds then come from the following simple lemma, which follows from the source coding theorem.

Lemma 3.3.2. *Let s denote the number of bits used by the $\mathbf{UR}^{\subset}$ or $\mathbf{UR}_k^{\subset}$ protocol, and let s' denote the expected number of bits to represent B. Then $(1+s+s') \geq \log\binom{n}{m}$. In particular, $s \geq \log\binom{n}{m} - s' - 1$.*

Section 3.4 provides the full details of the proof that

$$\mathbf{R}_\delta^{\rightarrow,pub}(\mathbf{UR}^{\subset}) = \Omega\big(\min\{n, \log^2(\frac{n}{\log(1/\delta)})\log(1/\delta)\}\big).$$

We extend our results in Section 3.5 to $\mathbf{UR}_k^{\subset}$ for $k \geq 1$, proving a lower bound of $\Omega(k\log^2(n/k))$

communication even for constant failure probability.

3.4 Communication Lower Bound for $\mathbf{UR}^{\subset}$

Consider a protocol $\mathcal{P}$ for $\mathbf{UR}^{\subset}$ with failure probability δ, operating in the one-way public coin model. When Alice's input is x and Bob's is y, Alice sends $\mathsf{Alice}(x)$ to Bob, and Bob outputs $\mathsf{Bob}(\mathsf{Alice}(x), y)$, which with probability at least $1-\delta$ is in $\mathrm{supp}(x-y)$. As mentioned in Section 3.3, we use $\mathcal{P}$ as a subroutine in a scheme for encoding/decoding elements of $\binom{[n]}{m}$ for $m = \lfloor \sqrt{n \log(1/\delta)} \rfloor$. We assume $\log(1/\delta) \leq n/64$, since for larger n we have an $\Omega(n)$ lower bound.

3.4.1 Encoding/decoding scheme

We now describe our encoding/decoding scheme $(\mathsf{ENC}, \mathsf{DEC})$ for elements in $\binom{[n]}{m}$, which uses $\mathcal{P}$ in a black-box way. The parameters shared by ENC and DEC are given in Algorithm 4.

As discussed in Section 3.3, on input $S \in \binom{[n]}{m}$, ENC computes $M \leftarrow \mathsf{Alice}(\mathbf{1}_S)$ as part of its output. Moreover, ENC also outputs a subset $B \subseteq S$ computed as follows. Initially $B = S$ and $S_0 = S$. ENC proceeds in R rounds. In round $r \in [R]$, ENC computes $s_r \leftarrow \mathsf{Bob}(M, \mathbf{1}_{S \setminus S_{r-1}})$. Let b denote a binary string of length R, where b_r records whether Bob succeeds in round r. ENC also outputs b. If $s_r \in S_{r-1}$, i.e. $\mathsf{Bob}(M, \mathbf{1}_{S \setminus S_{r-1}})$ succeeds, ENC sets $b_r = 1$ and removes s_r from B (since the decoder can recover s_r from the $\mathbf{UR}^{\subset}$-protocol, ENC does not need to include it in B); otherwise ENC sets $b_r = 0$. At the end of round r, ENC picks a uniformly random set S_r in $\binom{S_{r-1} \setminus \{s_r\}}{n_r}$. In particular, ENC uses its shared randomness with DEC to generate S_r in such a way that ENC, DEC agree on the sets S_r (DEC will actually iteratively construct $C_r = S \setminus S_r$). We present ENC in Algorithm 5.

The decoding process is symmetric. Let $C_0 = \emptyset$ and $A = \emptyset$. DEC proceeds in R rounds. On round $r \in [R]$, DEC obtains $s_r \in S \setminus C_{r-1}$ by invoking $\mathsf{Bob}(M, \mathbf{1}_{C_{r-1}})$. By construction of C_{r-1} (to be described later), it is guaranteed that $S_{r-1} = S \setminus C_{r-1}$. Therefore, DEC recovers exactly the same s_r as ENC. DEC initially assigns $C_r \leftarrow C_{r-1}$. If $b_r = 1$, DEC adds s_r to both A and C_r. At the end

of round r, DEC inserts many random items from B into C_r so that $C_r = S \backslash S_r$. DEC can achieve this because of the shared random permutation π when constructing S_r. In the end, DEC outputs $B \cup A$. We present DEC in Algorithm 6.

Algorithm 4 Variables shared by encoder ENC and decoder DEC.

$m \leftarrow \lfloor \sqrt{n \log(1/\delta)} \rfloor$
$K \leftarrow \lfloor \frac{1}{16} \log(1/\delta) \rfloor$
$R \leftarrow \lfloor K \log(m/4K) \rfloor$
for $r = 0, \ldots, R$ **do**
    $n_r \leftarrow \lfloor m \cdot 2^{-r/K} \rfloor$ ▷ $|S_r| = n_r$, and $\forall r\ n_r - n_{r+1} \geq 2$
end for
π is a random permutation on $[n]$ ▷ Used to generate S_r and C_r

Algorithm 5 Encoder ENC.

procedure $\mathrm{ENC}(S)$
    $M \leftarrow \mathsf{Alice}(\mathbf{1}_S)$
    $A \leftarrow \emptyset$ ▷ the set DEC recovers just from M
    $S_0 \leftarrow S$ ▷ at end of round r, DEC still needs to recover S_r
    for $r = 1, \ldots, R$ **do**
        $s_r \leftarrow \mathsf{Bob}(M, \mathbf{1}_{S \backslash S_{r-1}})$ ▷ $s_r \stackrel{?}{\in} S_{r-1}$ found in round r
        $S_r \leftarrow S_{r-1}$
        if $s_r \in S_{r-1}$ **then** ▷ i.e. if s_r is a valid sample
            $b_r \leftarrow 1$ ▷ $b \in \{0,1\}^R$ indicating which rounds succeed
            $A \leftarrow A \cup \{s_r\}$
            $S_r \leftarrow S_r \backslash \{s_r\}$
        else
            $b_r \leftarrow 0$
        end if
        Remove $|S_r| - n_r$ elements from S_r with smallest π_a's among $a \in S_r$ ▷ now $|S_r| = n_r$
    end for
    return $(M, S \backslash A, b)$
end procedure

3.4.2 Analysis

We have two random objects in our encoding/decoding scheme: (1) the random source used by $\mathcal{P}$, denoted by X, and (2) the random permutation π. These are independent.

First, we can prove that $\mathsf{DEC}(\mathsf{ENC}(S)) = S$. That is, for any fixing of the randomness in X and

Algorithm 6 Decoder DEC.

procedure $\mathrm{DEC}(M, B, b)$
  ▷ M is $\mathsf{Alice}(\mathbf{1}_S)$
  ▷ $b \in \{0,1\}^R$ indicates rounds in which Bob succeeds
  ▷ B contains all elements of S that DEC doesn't recover via M
  $A \leftarrow \emptyset$ ▷ the subset of S DEC recovers just from M
  $C_0 \leftarrow \emptyset$ ▷ subset of S we have built up so far
  for $r = 1, \ldots, R$ **do** ▷ each iteration tries to recover 1 element via M
    $C_r \leftarrow C_{r-1}$
    if $b_r = 1$ **then** ▷ this means Bob succeeds in round r
      $s_r \leftarrow \mathsf{Bob}(M, \mathbf{1}_{C_{r-1}})$ ▷ Invariant: $C_r = S \backslash S_r$ (S_r is defined in ENC)
      $A \leftarrow A \cup \{s_r\}$
      $C_r \leftarrow C_r \cup \{s_r\}$
    end if
    Insert $m - n_r - |C_r|$ items into C_r with smallest π_a's among $a \in B \backslash C_r$
      ▷ Random masking "Differential Privacy" step. Still n_r elements left to recover.
  end for
  return $B \cup A$
end procedure

π, DEC will always decode S successfully. It is because ENC and DEC share X and π, so that DEC essentially simulates ENC. We formally prove this by induction in Lemma 3.4.1.

Now our goal is to prove that by using the $\mathbf{UR}^{\subset}$-protocol, the number of bits that ENC saves in expectation over the naive $\lceil \log \binom{n}{m} \rceil$-bit encoding is $\Omega\big(\log(1/\delta) \log^2(\frac{n}{\log(1/\delta)})\big)$ bits. Intuitively, it is equivalent to prove the number of elements that ENC saves is $\Omega\big(\log(1/\delta) \log(\frac{n}{\log(1/\delta)})\big)$. We formalize this in Lemma 3.4.2. Note that ENC also needs to output b (i.e., whether the Bob succeeds on R rounds), which takes R bits. By our setting of parameters, we can afford the loss of R bits. Thus it is sufficient to prove $\mathbb{E}\,|B| = |S| - \Omega\big(\log(1/\delta) \log \frac{n}{\log(1/\delta)}\big)$.

We have $|S| - |B| = \sum_{r=1}^{R} b_r$. In Lemma 3.3.1, we prove the probability that Bob fails on round r is upper bounded by $\frac{I(X;S_{r-1})+1}{\log(1/\delta)}$, where $I(X; S_{r-1})$ is the mutual information between X and S_{r-1}. Furthermore, we will show in Lemma 3.4.4 that $I(X; S_{r-1})$ is upper bounded by $O(K)$. By our setting of parameters, we have $\mathbb{E}\, b_r = \Omega(1)$ and thus $\mathbb{E}(|S| - |B|) = \Omega(R) = \Omega\big(\log(1/\delta) \log \frac{n}{\log(1/\delta)}\big)$.

Lemma 3.4.1. *$DEC(ENC(S)) = S$.*

Proof. We claim that for $r = 0, \ldots, R$, $\{S_r, C_r\}$ is a partition of S (S_r is defined in Algorithm 5, and

C_r in Algorithm 6). We prove the claim by induction on r. Our base case is $r = 0$, for which the claim holds since $S_0 = S$, $C_0 = \emptyset$.

Assume the claim holds for $r-1$ ($1 \le r \le R$), and we consider round r. On round r, by induction $S \backslash S_{r-1} = C_{r-1}$, the index s_r obtained by both ENC and DEC are the same. Initially $S_r = S_{r-1}$ and $C_r = C_{r-1}$, and so $\{S_r, C_r\}$ is a partition of S. If s_r is a valid sample (i.e. $s_r \in S_{r-1}$), then $b_r = 1$, and ENC removes s_r from S_r and in the meanwhile DEC inserts s_r into C_r, so that $\{S_r, C_r\}$ remains a partition of S. Next, ENC repeats removing the a from S_r with the smallest π_a value until $|S_r| = n_r$. Symmetrically, DEC repeats inserting the a into C_r with the smallest π_a value among $a \in B \backslash C_r$, until $|C_r| = |S| - n_r$. In the end we have $|S_r| + |C_r| = |S|$, so ENC and DEC execute repetition the same number of times. Moreover, we can prove that during the same iteration of this repeated insertion, the element removed from S_r is exactly the same element inserted to C_r. This is because in the beginning of a repetition $\{S_r, C_r\}$ is a partition of S. We have $B \backslash C_r \subseteq S \backslash C_r = S_r$. Let a^* denote $a \in S_r$ that minimizes π_a. Then $a^* \in B \backslash C_r \subseteq S_r$ (since a^* will be removed from S_r, it has no chance to be included in S in ENC, so that B contains a^*), and π_{a^*} is also the smallest among $\{\pi_a : a \in B \backslash C_r\}$. Thus both ENC and DEC will take a^* (for ENC, to remove from S_r, and for DEC, to insert into C_r). Therefore, $\{S_r, C_r\}$ remains a partition of S.

Given the fact that $\{S_r, C_r\}$ is a partition of S, the s_r are the same in ENC and DEC. Furthermore, $A = \{s_r : b_r = 1, r = 1, \ldots, R\}$ are the same in ENC and DEC. We know $A \subseteq S$. Since ENC outputs $S \backslash A$, and DEC outputs $(S \backslash A) \cup A$, we have $\mathsf{DEC}(\mathsf{ENC}(S)) = S$. □

Lemma 3.4.2. *Let $W \in \mathbb{N}$ be a random variable with $W \le m$ and $\mathbb{E}\, W \le m - d$. Then $\mathbb{E}(\log \binom{n}{m} - \log \binom{n}{W}) \ge d \log(\frac{n}{m} - 1)$.*

Proof.

$$\begin{aligned}
\log \binom{n}{m} - \log \binom{n}{W} &= \log \frac{n!/(m!(n-m)!)}{n!/(W!(n-W)!)} \\
&= \sum_{i=1}^{m-W} \log \frac{n - W - i + 1}{m - i + 1} \\
&\ge (m - W) \cdot \log \frac{n - W}{m} \\
&\ge (m - W) \cdot \log \frac{n - m}{m}.
\end{aligned}$$

Taking expectation on both sides, we have $\mathbb{E}(\log \binom{n}{m} - \log \binom{n}{W}) \geq d \log(\frac{n}{m} - 1)$. □

Lemma 3.3.1 (restated). Consider f: $\{0,1\}^b \times \{0,1\}^q \to \{0,1\}$ and $X \in \{0,1\}^b$ uniformly random. If $\forall y \in \{0,1\}^q$, $\mathbb{P}(f(X,y) = 1) \leq \delta$ where $0 < \delta < 1$, then for any r.v. Y supported on $\{0,1\}^q$,

$$\mathbb{P}(f(X,Y) = 1) \leq \frac{I(X;Y) + H_2(\delta)}{\log(1/\delta)},$$

where $I(X;Y)$ is the mutual information between X and Y, and H_2 is the binary entropy function.

Proof. It is equivalent to prove

$$I(X;Y) \geq \mathbb{E}(f(X,Y)) \cdot \log(1/\delta) - H_2(\delta).$$

By definition of mutual entropy $I(X;Y) = H(X) - H(X|Y)$, where $H(X) = b$ and we must show

$$\begin{aligned} H(X|Y) &\leq H_2(\delta) + (1 - \mathbb{E}(f(X,Y))) \cdot b + \mathbb{E}(f(X,Y)) \cdot (b - \log(1/\delta)) \\ &= b + H_2(\delta) - \mathbb{E}(f(X,Y)) \cdot \log(1/\delta). \end{aligned}$$

The upper bound for $H(X|Y)$ is obtained by considering the following one-way communication problem: Alice knows both X and Y while Bob only knows Y, and Alice must send a single message to Bob so that Bob can recover X. The expected message length in an optimal protocol is exactly $H(X|Y)$. Thus, any protocol gives an upper bound for $H(X|Y)$, and we simply take the following protocol: Alice prepends a 1 bit to her message iff $f(X,Y) = 1$ (taking $H_2(\delta)$ bits in expectation). Then if $f(X,Y) = 0$, Alice sends X directly (taking b bits). Otherwise, when $f(X,Y) = 1$, Alice sends the index of X in $\{x | f(x,Y) = 1\}$ (taking $\log(\delta 2^b) = b - \log(1/\delta)$ bits). □

Corollary 3.4.3. *Let X denote the random source used by the $\mathbf{UR}^{\subset}$-protocol with failure probability at most δ. If S is a fixed set and $T \subset S$, $\mathbb{P}(\mathsf{Bob}(\mathsf{Alice}(\mathbf{1}_S), \mathbf{1}_T) \notin S \backslash T) \leq \frac{I(X;T)+H_2(\delta)}{\log(1/\delta)}$.*

Lemma 3.4.4. *$I(X;S_r) \leq 6K$, for $r = 1, \ldots, R$.*

Proof. Note that $I(X;S_r) = H(S_r) - H(S_r|X)$. Since $|S_r| = n_r$ and $S_r \subseteq S$, $H(S_r) \leq \log \binom{m}{n_r}$. Here is the main idea to lower bound $H(S_r|X)$: By definition of conditional entropy, $H(S_r|X) =$

$\sum_x p_x \cdot H(S_r|X=x)$. We fix an arbitrary x. If we can prove that for any $T \subseteq S$ where $|T| = n_r$, $\mathbb{P}(S_r = T|X=x) \le p$, then by definition of entropy we have $H(S_r|X=x) \ge \log \frac{1}{p}$.

First we can prove for any fixed T,

$$\mathbb{P}(S_r = T|X=x) \le \prod_{i=1}^{r} \frac{\binom{n_{i-1}-n_r-1}{n_{i-1}-n_i-1}}{\binom{n_{i-1}-1}{n_{i-1}-n_i-1}}. \tag{3.3}$$

We have $\mathbb{P}(S_r = T|X=x) = \Pi_{i=1}^{r}\mathbb{P}(T \subseteq S_i|T \subseteq S_{i-1})$. On round i ($1 \le i \le r$), ENC removes $n_{i-1} - n_i$ elements (at least $n_{i-1} - n_i - 1$ of which are chosen all at random) from S_{i-1} to obtain S_i. Conditioned on the event that $T \subseteq S_{i-1}$, the probability that $T \subseteq S_i$ is at most $\binom{n_{i-1}-n_r-1}{n_{i-1}-n_i-1}/\binom{n_{i-1}-1}{n_{i-1}-n_i-1}$, where the equation achieves when $s_i \in S_{i-1}\backslash T$, and ENC takes a uniformly random subset of $S_{i-1}\backslash\{s_i\}$ of size $n_{i-1} - n_i - 1$, so that the subset does not intersect with T.

Next we will prove

$$\prod_{i=1}^{r} \frac{\binom{n_{i-1}-n_r-1}{n_{i-1}-n_i-1}}{\binom{n_{i-1}-1}{n_{i-1}-n_i-1}} \le \frac{2^{6K}}{\binom{m}{n_r}}. \tag{3.4}$$

For notational simplicity, let $n^{\underline{k}}$ denote $n \cdot (n-1)\ldots(n-k+1)$. We have

$$\begin{aligned}
\prod_{i=1}^{r} \frac{\binom{n_{i-1}-n_r-1}{n_{i-1}-n_i-1}}{\binom{n_{i-1}-1}{n_{i-1}-n_i-1}} &= \prod_{i=1}^{r} \frac{(n_{i-1}-n_r-1)!n_i!}{(n_{i-1}-1)!(n_i-n_r)!} \\
&= \prod_{i=1}^{r} \frac{n_i^{\underline{n_r}}}{(n_{i-1}-1)^{\underline{n_r}}} \\
&= \prod_{i=1}^{r} \left(\frac{n_i^{\underline{n_r}}}{n_{i-1}^{\underline{n_r}}} \cdot \frac{n_{i-1}}{n_{i-1}-n_r} \right).
\end{aligned}$$

By telescoping,

$$\prod_{i=1}^{r} \frac{n_i^{\underline{n_r}}}{n_{i-1}^{\underline{n_r}}} = \frac{n_r^{\underline{n_r}}}{n_0^{\underline{n_r}}} = \frac{n_r!(n_0-n_r)!}{n_0!} = \frac{1}{\binom{n_0}{n_r}} = \frac{1}{\binom{m}{n_r}}. \tag{3.5}$$

Moreover,

$$\begin{aligned}\prod_{i=1}^{r}\frac{n_{i-1}}{n_{i-1}-n_r} &\le \prod_{i=1}^{r}\frac{1}{1-\frac{m\cdot 2^{-r/K}}{m\cdot 2^{-(i-1)/K}-1}}\\ &\le \prod_{i=1}^{r}\frac{1}{1-\frac{m\cdot 2^{-r/K}+1}{m\cdot 2^{-(i-1)/K}}}\\ &= \prod_{j=1}^{r}\frac{1}{1-2^{-j/K}-\frac{2^{\frac{r-j}{K}}}{m}}. \end{aligned} \tag{3.6}$$

By our setting of parameters

$$\frac{2^{\frac{r}{K}}}{m} \le \frac{2^{\frac{R}{K}}}{m} \le \frac{1}{4K}.$$

Therefore, for $j \in \{1,\dots,r\}$,

$$\frac{1}{1-2^{-\frac{j}{K}}-\frac{2^{\frac{r-j}{K}}}{m}} \le \frac{1}{1-(1+\frac{1}{4K})2^{-\frac{j}{K}}}.$$

By Taylor series $2^{1/K}=\sum_{n=0}^{\infty}\frac{(\ln 2)^n}{n!K^n} > 1+\frac{\ln 2}{K} > 1+\frac{1}{4K}$, and thus $\frac{1}{1-(1+\frac{1}{4K})2^{-j/K}} \le \frac{1}{1-2^{(1-j)/K}}$, for $j=2,\dots,r$. For $j=1$, we have $\frac{1}{1-(1+\frac{1}{4K})2^{-\frac{1}{K}}} \le 2^K$.

By Lemma 3.4.5, we have $\prod_{j=1}^{\infty}\frac{1}{1-2^{-j/K}} \le 2^{5K}$. Therefore, the right hand side of (3.6) is upper bounded by 2^{6K}. Together with (3.5), we prove (3.4) holds.

Finally, let $p = 2^{6K}/\binom{m}{n_r}$, we have $\mathbb{P}(S_r = T|X=x) \le p$ and thus $H(S_r|X=x) \ge \log\frac{1}{p} = \log\binom{m}{n_r} - 6K$. Therefore, $H(S_r|X) \ge \log\binom{m}{n_r} - 6K$ and so $I(X;S_r) = H(S_r) - H(S_r|X) \le 6K$. □

Lemma 3.4.5. *Let $K \in \mathbb{N}$ and $K \ge 1$. We have $\prod_{j=1}^{\infty}\frac{1}{1-2^{-j/K}} \le 2^{5K}$.*

Proof. First, we bound the product of first $2K$ terms. Note that $\frac{1}{1-2^{-x}} \le \frac{8}{3x}$ for $0 < x \le 2$. Therefore,

$$\begin{aligned}\prod_{j=1}^{2K}\frac{1}{1-2^{-j/K}} &\le (8/3)^{2K}\cdot\frac{K^{2K}}{(2K)!}\\ &\le (8/3)^{2K}\cdot\frac{K^{2K}}{(2K/e)^{2K}}\\ &= (4e/3)^{2K}\\ &< 2^{4K}.\end{aligned}$$

Then, we bound the product of the rest terms

$$\begin{aligned}\prod_{j=2K+1}^{\infty}\frac{1}{1-2^{-j/K}} &\le \prod_{j=2K+1}^{\infty}\frac{1}{1-2^{-\lfloor j/K\rfloor}}\\ &\le \prod_{i=2}^{\infty}\left(\frac{1}{1-2^{-i}}\right)^{K}\\ &\le \left(\frac{1}{1-\sum_{i=2}^{\infty}2^{-i}}\right)^{K}\\ &= 2^{K}.\end{aligned}$$

Multiplying two parts proves the lemma. □

Theorem 3.4.6. $\mathbf{R}_{\delta}^{\rightarrow,pub}(\mathbf{UR}^{\subset}) = \Omega\big(\log(1/\delta)\log^2(\frac{n}{\log(1/\delta)})\big)$, *given that* $64 \le \log(1/\delta) \le \frac{n}{64}$.

Proof. By Lemma 3.4.1, the success probability of protocol $(\mathsf{ENC}, \mathsf{DEC})$ is 1. By Lemma 3.3.2, we have $\mathsf{s} \ge \log\binom{n}{m} - \mathsf{s}' - 1$, where $\mathsf{s}' = \log n + R + \mathbb{E}(\log\binom{n}{|B|})$. The size of B is $|B| = |S| - \sum_{r=1}^{R} b_r$. By Corollary 3.4.3, conditioned on S, $\mathbb{P}(b_r = 0) \le \frac{I(X;S_{r-1})+1}{\log(1/\delta)}$. By Lemma 3.4.4, $I(X;S_{r-1}) \le 6K$ (Note that when $r = 1$, $I(X;S_0) = 0 \le 6K$). Therefore, $\mathbb{E}(b_r) \ge 1 - \frac{6K+1}{\log\frac{1}{\delta}}$. By the setting of parameters (see Algorithm 4) we have $\mathbb{E}(b_r) \ge \frac{39}{64}$. Therefore, $\mathbb{E}(|B|) \le |S| - \frac{39}{64}R$. By Lemma 3.4.2,

$$\log\binom{n}{m} - \mathbb{E}\left(\log\binom{n}{|B|}\right) \ge \frac{39}{64}R\cdot\log\left(\frac{n}{m}-1\right) \ge \frac{1}{2}R\log\left(\frac{n}{\log(1/\delta)}\right).$$

Furthermore, $\frac{1}{6}R\log(\frac{n}{\log(1/\delta)}) \ge R$. Thus we obtain

$$\mathsf{s} \ge \frac{R}{3}\log\left(\frac{n}{\log(1/\delta)}\right) - (\log n + 1) = \Omega\left(\log(1/\delta)\log^2\left(\frac{n}{\log(1/\delta)}\right)\right).$$

□

3.5 Communication Lower Bound for $\mathbf{UR}_k^{\subset}$

In this section, we prove the lower bound $\mathbf{R}_{1/2}^{\rightarrow,pub}(\mathbf{UR}_k^{\subset}) = \Omega(\min\{n, k\log^2\frac{n}{k}\})$. In fact, our lower bound holds for any failure probability δ bounded away from 1. Let $\mathcal{P}$ denote a $\mathbf{UR}_k^{\subset}$-protocol where Alice sends $\mathsf{Alice}_k(x)$ to Bob, and Bob outputs $\mathsf{Bob}_k(\mathsf{Alice}_k(x), y)$. We consider the following encoding/decoding scheme $(\mathsf{ENC}_k, \mathsf{DEC}_k)$ for $S \in \binom{[n]}{m}$. ENC_k computes $M \leftarrow \mathsf{Alice}_k(\mathbf{1}_S)$ as part of

its message. In addition, ENC_k includes $B \subseteq S$ constructed as follows, spending $\lceil \log \binom{n}{|B|} \rceil$ bits. Initially $B = S$, and ENC_k proceeds in $R = \Theta(\log(n/k))$ rounds. Let $S_0 = S \supseteq S_1 \supseteq \ldots \supseteq S_R$ where S_r is generated by sub-sampling each element in S_{r-1} with probability $\frac{1}{2}$. In round r ($r = 1, \ldots, R$), ENC_k tries to obtain k elements from S_{r-1} by invoking $\mathsf{Bob}_k(M, \mathbf{1}_{S \setminus S_{r-1}})$, denoted by A_k, and removes $A_k \cap (S_{r-1} \setminus S_r)$ (whose expected size is $\frac{k}{2}$) from B. Note that DEC_k is able to recover the elements in $A_k \cap (S_{r-1} \setminus S_r)$. For each round the failure probability of Bob_k is at most δ. Thus we have $\mathbb{E}(|S| - |B|) \geq \frac{k}{2} \cdot (1 - \delta) \cdot R = \Omega(k \log \frac{n}{k})$. Furthermore, each element contains $\Theta(\log \frac{n}{k})$ bits of information, thus yielding a lower bound of $\Omega(k \log^2 \frac{n}{k})$ bits.

In this section we assume $k \leq n/2^{10}$, since for larger n we have an $\Omega(n)$ lower bound.

3.5.1 Encoding/decoding scheme

We present our encoder in Algorithm 8 and decoder in Algorithm 9. The variables shared by the encoder and decoder are presented in Algorithm 7.

Algorithm 7 Variables Shared by Encoder ENC_k and Decoder DEC_k.

$m \leftarrow \lfloor \sqrt{nk} \rfloor$
$R \leftarrow \lfloor \frac{1}{2} \log(n/k) - 2 \rfloor$ ▷ Note that $R \geq 3$ because $k \leq \frac{n}{2^{10}}$
$T_0 \leftarrow [n]$
for $r = 1, \ldots, R$ **do**
    $T_r \leftarrow \emptyset$
    For each $a \in T_{r-1}$, $T_r \leftarrow T_r \cup \{a\}$ with probability $\frac{1}{2}$ ▷ We have $S_r = S \cap T_r$
end for

3.5.2 Analysis

Theorem 3.5.1. $\mathbf{R}_{\delta}^{\rightarrow, pub}(\mathbf{UR}_k^{\subset}) = \Omega(k \log^2 \frac{n}{k})$, *given that* $1 \leq k \leq \frac{n}{2^{10}}$ *and* $\delta \leq \frac{1}{2}$.

Proof. Let $S_r = S \cap T_r$. Let SUCC denote the event that $|S \cap T_R| = |S_R| \geq k$. Note that $\mathbb{E}\,|S_R| = \frac{1}{2^R} m = 4k$. By the Chernoff bound, $\mathbb{P}(\mathsf{SUCC}) \geq \frac{1}{2}$. In the following, we argue conditioned on SUCC. Namely, in each round r, there are at least k items in S_r.

Similar to Lemma 3.4.1, we can prove the protocol $(\mathsf{ENC}_k, \mathsf{DEC}_k)$ always succeeds. By Lemma 3.3.2, we have $\mathsf{s} \geq \log \binom{n}{m} - \mathsf{s}' - 2$, where $\mathsf{s}' = \log n + R + \mathbb{E} \log \binom{n}{|B|}$. The size of B

Algorithm 8 Encoder ENC_k.

procedure $\mathrm{ENC}_k(S)$
  $M \leftarrow \mathsf{Alice}_k(\mathbf{1}_S)$
  $A \leftarrow \emptyset$
  for $r = 1, \ldots, R$ **do**
    $A_r \leftarrow \mathsf{Bob}_k(M, \mathbf{1}_{S\setminus(S\cap T_{r-1})})$
    if $A_r \subseteq S \cap T_{r-1}$ **then** ▷ i.e. if A_r is valid
      $b_r \leftarrow 1$ ▷ b is a binary string of length R, indicating if Bob_k succeeds in round r
      $A \leftarrow A \cup (A_r \cap (T_{r-1}\setminus T_r))$
    else
      $b_r \leftarrow 0$
    end if
  end for
  return $(M, S\setminus A, b)$
end procedure

is $|B| = |S| - \sum_{r=1}^{R} (b_r \cdot |A_r \cap (S_{r-1}\setminus S_r)|)$. The randomness used by $\mathcal{P}$ is independent from $S\setminus S_{r-1}$ for every $r \in [R]$. Therefore, $\mathbb{E}\, b_r \geq 1 - \delta \geq \frac{1}{2}$, and b_r is independent from $|A_r \cap (S_{r-1}\setminus S_r)|$. We have $\mathbb{E}\,|A_r \cap (S_{r-1}\setminus S_r)| = \frac{k}{2}$, and thus $\mathbb{E}(|S| - |B|) \geq \frac{kR}{4}$. By Lemma 3.4.2, $\log\binom{n}{m} - \mathbb{E}\log\binom{n}{|B|} \geq \frac{kR}{4} \cdot \log(\frac{n}{m} - 1) \geq \frac{kR}{9}\log(\frac{n}{k})$. Moreover, $\frac{kR}{10}\log\frac{n}{k} \geq R$. Thus we have $\mathsf{s} = \Omega(kR\log\frac{n}{k}) = \Omega(k\log^2\frac{n}{k})$. □

3.6 A tight upper bound for $\mathbf{R}_\delta^{\rightarrow,pub}(\mathbf{UR}_k)$

In [JST11, Proposition 1] it is shown that $\mathbf{R}_\delta^{\rightarrow,pub}(\mathbf{UR}_k) = O(\min\{n, t\log^2 n\})$ for $t = \max\{k, \log(1/\delta)\}$. Here we show that a minor modification of their protocol in fact shows the correct complexity $\mathbf{R}_\delta^{\rightarrow,pub}(\mathbf{UR}_k) = O(\min\{n, t\log^2(n/t)\})$, which given our new lower bound, is optimal up to a constant factor for the full range of n, k, δ as long as δ is bounded away from 1.

Recall Alice and Bob receive $x, y \in \{0,1\}^n$, respectively, and share a public random string. Alice must send a single message M to Bob, from which Bob must recover $\min\{k, \|x - y\|_0\}$ indices $i \in [n]$ for which $x_i \neq y_i$. Bob is allowed to fail with probability δ. The fact that $\mathbf{R}_\delta^{\rightarrow,pub}(\mathbf{UR}_k) \leq n$ is obvious: Alice can simply send the message $M = x$, and Bob can then succeed with failure probability 0. We thus now show $\mathbf{R}_{e^{-ck}}^{\rightarrow,pub}(\mathbf{UR}_k) \leq k\log^2(n/k)$ for some constant $c > 0$, which completes the proof of the upper bound. We assume $k \leq n/2$ (otherwise, Alice sends x explicitly).

Algorithm 9 Decoder DEC_k.

procedure $\mathrm{DEC}_k(M, B, b)$
  $A \leftarrow \emptyset$
  $C_0 \leftarrow \emptyset$
  for $r = 1, \ldots, R$ **do**
    $C_r \leftarrow C_{r-1}$
    if $b_r = 1$ **then**
      $A_r \leftarrow \mathsf{Bob}_k(M, \mathbf{1}_{C_{r-1}})$ ▷ Invariant: $C_r = S \backslash (S \cap T_r)$
      $A \leftarrow A \cup (A_r \cap (T_{r-1} \backslash T_r))$
      $C_r \leftarrow C_r \cup (A_r \cap (T_{r-1} \backslash T_r))$
    end if
    $C_r \leftarrow C_r \cup (B \cap (T_{r-1} \backslash T_r))$
  end for
  return $B \cup A$
end procedure

As mentioned, the protocol we describe is nearly identical to one in [JST11] (see also [CF14]). We will describe the new protocol here, then point out the two minor modifications that improve the $O(k \log^2 n)$ bound to $O(k \log^2(n/k))$ in Remark 3.6.3. We first need the following lemma.

Lemma 3.6.1. *Let $\mathbb{F}_q$ be a finite field and $n > 1$ an integer. Then for any $1 \leq k \leq \frac{n}{2}$, there exists $\Pi_k \in \mathbb{F}_q^{m \times n}$ for $m = O(k \log_q(qn/k))$ s.t. for any $w \neq w' \in \mathbb{F}_q^n$ with $\|w\|_0, \|w'\|_0 \leq k$, $\Pi_k w \neq \Pi_k w'$.*

Proof. The proof is via the probabilistic method. $\Pi_k w = \Pi_k w'$ iff $\Pi_k(w - w') = 0$. Note $v = w - w'$ has $\|v\|_0 \leq 2k$. Thus it suffices to show that such a Π_k exists with no $(2k)$-sparse vector in its kernel. The number of vectors $v \in \mathbb{F}_q^n$ with $\|v_0\| \leq 2k$ is at most $\binom{n}{2k} \cdot q^{2k}$. For any fixed v, $\mathbb{P}(\Pi_k v = 0) = q^{-m}$. Thus

$$\mathbb{P}(\exists v, \|v\|_0 \leq 2k : \Pi_k v = 0) \leq \binom{n}{2k} \cdot q^{2k} \cdot q^{-m}$$

by a union bound. The above is strictly less than 1 for $m > 2k + \log_q \binom{n}{2k}$, yielding the claim. □

Corollary 3.6.2. *Let $\mathbb{F}_q$ be a finite field and $n > 1$ an integer. Then for any $1 \leq k \leq \frac{n}{2}$, there exists $\Pi_k \in \mathbb{F}_q^{m \times n}$ for $m = O(k \log_q(qn/k))$ together with an algorithm $\mathcal{R}$ such that for any $w \in \mathbb{F}_q^n$ with $\|w\|_0 \leq k$, $\mathcal{R}(\Pi_k w) = w$.*

Proof. Given Lemma 3.6.1, a simple such $\mathcal{R}$ is as follows. Given some $y = \Pi_k w^*$ with $\|w^*\|_0 \leq k$, $\mathcal{R}$ loops over all w in $\mathbb{F}_q^n$ with $\|w\|_0 \leq k$ and outputs the first one it finds for which $\Pi_k w = y$. □

The protocol for $\mathbf{UR}_k$ is now as follows. Alice and Bob use public randomness to pick commonly known random functions $h_0, \ldots, h_L : [n] \to \{0,1\}$ for $L = \lfloor \log_2(n/k) \rfloor$, such that for any $i \in [n]$ and for any j, $\mathbb{P}(h_j(i) = 1) = 2^{-j}$. They also agree on a matrix Π_{16k} and $\mathcal{R}$ as described in Corollary 3.6.2 for a sufficiently large constant $C > 0$ to be determined later, with $q = 3$. Thus Π_{16k} has $m = O(k \log(n/k))$ rows. Alice then computes $v_j = \Pi_{16k} x|_{h_j^{-1}(1)}$ for $j = 0, \ldots, L$ where $v_j \in \mathbb{F}_q^m$, and her message to Bob is $M = (v_0, \ldots, v_L)$. For $S \subseteq [n]$ and x an n-dimensional vector, $x|_S$ denotes the n-dimensional vector with $(x|_S)_i = x_i$ for $i \in S$, and $(x|_S)_i = 0$ for $i \notin S$. Note Alice's message M is $O(k \log^2(n/k))$ bits, as desired. Bob then executes the following algorithm and outputs the returned values.

Algorithm 10 Bob's algorithm in the $\mathbf{UR}_k$ protocol.

```
procedure Bob(v_0, ..., v_L)
    for j = L, L-1, ..., 0 do
        v_j ← v_j − Π_{16k} y|_{h_j^{-1}(1)}
        w_j ← R(v_j)
        if ||w_j||_0 ≥ k or j = 0 then
            return an arbitrary min{k, ||w_j||_0} elements from supp(w_j)
        end if
    end for
end procedure
```

The correctness analysis is then as follows, which is nearly the same as the ℓ_0-sampler of [JST11]. If Alice's input is x and Bob's is y, let $a = x - y \in \{-1, 0, 1\}^n$, so that a can be viewed as an element of $\mathbb{F}_3^n$. Also let $a_j = a|_{h_j^{-1}(1)}$. Then $\mathbb{E} \|v_j\|_0 = \|a\|_0 \cdot 2^{-j}$, and since $0 \leq \|a\|_0 \leq n$, there either (1) exists a unique $0 \leq j^* \leq L$ such that $2k \leq \mathbb{E} \|a_j\|_0 \cdot 2^{-j^*} < 4k$, or (2) $\|a\|_0 < 2k$ (in which case we define $j^* = 0$). Let $\mathcal{E}$ be the event that $\|a_j\|_0 \leq 16k$ simultaneously for all $j \leq j^*$. Let $\mathcal{F}$ be the event that *either* we are in case (2), or we are in case (1) and $\|a_{j^*}\|_0 \geq k$ holds. Note that conditioned on $\mathcal{E}, \mathcal{F}$ both occurring, Bob succeeds by Corollary 3.6.2.

We now just need to show $\mathbb{P}(\neg\mathcal{E} \wedge \neg\mathcal{F}) < e^{-\Omega(k)}$. We use the union bound. First, consider $\mathcal{F}$. If $j^* = 0$, then $\mathbb{P}(\neg\mathcal{F}) = 0$. If $j^* \neq 0$, then $\mathbb{P}(\neg\mathcal{F}) \leq \mathbb{P}(\|a_{j^*}\|_0 < \frac{1}{2} \cdot \mathbb{E} \|a_{j^*}\|_0)$, which is $e^{-\Omega(k)}$ by the Chernoff bound since $\mathbb{E} \|a_{j^*}\|_0 = \Theta(k)$. Next we bound $\mathbb{P}(\neg\mathcal{E})$. For $j \geq j^*$, we know

$\mathbb{E}\,\|a_j\|_0 \le 4k/2^{j-j^*}$. Thus, letting μ denote $\mathbb{E}\,\|a_j\|_0$,

$$\mathbb{P}(\|a_j\|_0 > 16k) < \left(\frac{e^{\frac{16k}{\mu}-1}}{(\frac{16k}{\mu})^{\frac{16k}{\mu}}}\right)^{\mu} < \left(\frac{16k}{\mu}\right)^{-\Omega(k)} < (e^{-Ck})^{j-j^*} \tag{3.7}$$

for some constant $C > 0$ by the Chernoff bound and the fact that $16k/\mu \ge 4 > e$. Recall that the Chernoff bound states that for X a sum of independent Bernoullis,

$$\forall \delta > 0,\ \mathbb{P}(X > (1+\delta)\,\mathbb{E}\,X) < \left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mathbb{E}\,X}.$$

Then by a union bound over $j \ge j^*$ and applying (3.7),

$$\mathbb{P}(\neg\mathcal{E}) = \mathbb{P}(\exists j \ge j^* : \|a_j\|_0 > 16k) < \sum_{j=j^*}^{\infty} (e^{-Ck})^{j-j^*} = O(e^{-Ck}).$$

Remark 3.6.3. As already mentioned, the protocol given above and the one described in [JST11] using $O(k\log^2 n)$ bits differ in minor points. First: the protocol there used $\lfloor \log_2 n \rfloor$ different hash functions h_j, but as seen above, only $\lfloor \log_2(n/k) \rfloor$ are needed. This already improves one $\log n$ factor to $\log(n/k)$. The other improvement comes from replacing the k-sparse recovery structure with $2k$ rows used in [JST11] with our Corollary 3.6.2. Note the matrix Π_k in our corollary has even *more* rows, but the key point is that the bit complexity is improved. Whereas using a k-sparse recovery scheme as described in [JST11] would use $2k$ linear measurements of a k-sparse vector $w \in \{-1,0,1\}^n$ with $\log n$ bits per measurement (for a total of $O(k\log n)$ bits), we use $O(k\log(n/k))$ measurements with only $O(1)$ bits per measurement. The key insight is that we can work over $\mathbb{F}_3^n$ instead of $\mathbb{R}^n$ when the entries of w are in $\{-1,0,1\}$, which leads to our slight improvement.

3.7 Another variant of samplers, with applications

In this section we define another variant of the ℓ_0-sampling problem and provide a solution for it. The solution is a minor modification of the ℓ_0-sampling algorithm of [JST11].

Definition 3.7.1. *In the turnstile streaming problem ℓ_0-sampling$_k(\delta_1,\delta_2)$, there is a vector $z \in \mathbb{R}^n$ receiving turnstile streaming updates, and the answer to* `query()` *must behave as follows:*

- *With probability at most δ_1, the output can be "Fail".*
- *With probability at most δ_2, the output can be arbitrary.*
- *Otherwise, the output should be a uniformly random subset of size* $\min\{k, \|z\|_0\}$*, without replacement, from* $\mathrm{supp}(z)$.

One can define the support-finding$_k(\delta_1, \delta_2)$ problem analogously, as well as similar variants for other desired output distributions. The important distinction is that there are two types of failures, which are allowed to happen with different probabilities.

The description of algorithm given in [AGM12a] for outputting connected components is based on a support-finding subroutine that always knows when it fails and outputs Fail at those times, and [AGM12a] cites [JST11] for implementing this subroutine. Unfortunately, the algorithm of [JST11] does not provide this behavior and can behave arbitrarily when it fails. One can avoid this issue by simply conditioning on never failing, which would require [AGM12a] to call the [JST11] subroutine with $\delta < 1/\mathrm{poly}(n)$. This would worsen their claimed space for finding connected components from $O(n \log^3 n)$ bits to $O(n \log^4 n)$ bits. As mentioned in Footnote 3 though, it is apparent from the correctness analysis of the algorithm in [AGM12a] that their algorithm actually only needs a support-finding$_1(\delta_1, \delta_2)$ data structure for δ_1 a small but universal constant, and $\delta_2 = 1/\mathrm{poly}(n)$. As we sketch below, ℓ_0-sampling$_k(\delta_1, \delta_2)$ can be solved in the general turnstile model in $O((t \log n + \log(n/\delta_2)) \log(n/t))$ bits of space for $t = \max\{k, \log(1/\delta_1)\}$, based on modifying the algorithm of [JST11]. This leads to an implementation of the connectivity algorithm of [AGM12a] using space $O(n \log^3 n)$ bits as they claim.

We will need the following standard lemma.

Lemma 3.7.2. *There is a turnstile streaming algorithm for testing whether a vector $z \in \mathbb{R}^n$ updated in a stream is identically zero. If $z = 0$, the algorithm outputs $z = 0$. If $z \neq 0$, the algorithm outputs $z = 0$ with probability at most δ. The space consumption is $O(\log(nT/\delta))$ bits, assuming that the entries of z are always integers bounded by T in magnitude.*

Proof. We pick a prime p larger than $T + n/\delta$. We then pick a random $r \in \mathbb{F}_p$ at the beginning of the stream, and throughout the stream we maintain a single number in memory: $q(r) = \sum_{i=1}^n z_i r^i$

mod p. That is, during an update "$z_i \leftarrow z_i + \Delta$", we add $\Delta \cdot r^i$ to our counter, where the addition, multiplication, and exponentiation are all in $\mathbb{F}_p$. Note if $z \neq 0$, q is a non-zero degree-n polynomial (since $p > T$) and thus has at most n zeroes in $\mathbb{F}_p$, implying that $\mathbb{P}_r(q(r) = 0) \leq n/p < \delta$. □

We now sketch the modification to the algorithm of [JST11] which solves the above variant of ℓ_0-sampling with the desired space complexity. We assume that the vector z always has integer entries and $\|z\|_\infty \leq T = \text{poly}(n)$ throughout the stream. First we describe the algorithm without regard for the space needed to store hash functions. For fixed k and δ_2, we give an algorithm which achieves failure probability $\delta_1 = \exp(-\Theta(k))$ so that $t = \Theta(k)$. The algorithm of [JST11] is then similar to the upper bound for **UR** in Section 3.6 (see also [CF14]). Here we set $L = \Theta(\log(n/k))$. We then at each level, as in Section 3.6, use a Ck-sparse recovery scheme for vectors in $\mathbb{F}_p^n$ for some prime $p > \max\{n, T\}$ and constant $C > 0$ to arise in the analysis later (so that taking each entry of the recovered vector modulo p still returns the same vector). Such a measurement matrix Π_{Ck} only needs $2Ck$ measurements over $\mathbb{F}_p$ and thus the recovery scheme uses $O(k \log p) = O(k \log n)$ bits. For example, one can use Π being a $2Ck \times n$ Vandermonde matrix, so that no Ck-sparse vector is in its kernel. Recovery can be done by brute force. Alternatively one can use syndrome decoding, which uses the same space but allows recovery in time $\text{poly}(k \log p)$ time (see [DORS08, Section E]). Now, suppose level j^* is the level we return our output from, as per the scheme in Algorithm 10. If w_j denotes the output of the recovery scheme, before returning w_j we first would like to check that $w_j = a_j$ with a_j denoting $z|_{h_j^{-1}(1)}$, i.e. that $d_j = w_j - a_j$ is equal to zero. For this we use the zero-tester algorithm of Lemma 3.7.2 as a subroutine, with failure probability parameter δ_2. Thus overall we obtain the desired guarantee with space $O((k \log n + \log(n/\delta_2)) \log(n/k))$, as desired. The analysis of correctness is the same as in Section 3.6, based on the same events $\mathcal{E}, \mathcal{F}$. The difference is that $\mathcal{E}$ or $\mathcal{F}$ fail to hold, which happens with probability at most δ_1, then if the level j we base our output on does not provide us with a k-sparse vector, then we catch this with the zero-tester and say Fail (except with probability δ_2).

Now we must take into account the space to store the hash functions $h_0, \ldots, h_L$. We implement these hash functions by a single hash function $h : [n] \to [n]$ where we then interpret the event "$h_j(i) = 1$" as occurring when the least significant bit of $h(i)$ is equal to j. We now just need to

derandomized the selection of this single hash function. For this, we need to make sure that h is still chosen in such a way that $\mathbb{P}_h(\neg\mathcal{E} \vee \neg\mathcal{F}) < \delta_2$. For the purposes of this algorithm, we modify the definition of $\mathcal{E}$ to be stronger: here we consider it to be the event that $\sum_{j=j^*}^{L} \|a_j\|_0 \leq Ck$ for some constant $C > 0$, i.e. the *total* number of indices from $\mathrm{supp}(z)$ that hash to levels $j \geq j^*$ is $O(k)$. Then $\mathbb{P}(\neg\mathcal{E}) < \exp(-\Omega(k))$ is still true by a union bound and Chernoff bound as in Section 3.6, and $\mathcal{F}$ is not modified from that section, so we also have $\mathbb{P}(\neg\mathcal{F}) < \exp(-\Omega(k))$, as desired. Now, we cannot afford to store a truly random $h : [n] \to [n]$, so we instead determine h using Nisan's pseudorandom generator (PRG) [Nis92]. More specifically, it suffices to fool the following two branching programs up to statistical distance δ_2, one branching program for each of $\mathcal{E}, \mathcal{F}$. For $\mathcal{E}$, we have a branching program that sees the values $h(1), \ldots, h(n)$ in order and maintains a single word of memory which simply counts the number of items $i \in [n]$ that then hash to a level $j \geq j^*$. The space of this branching program is then $\log n$ bits which is certainly at most $S = \log(n/\delta_2)$, and the randomness needed is $R = n \log n$. Thus when using Nisan's PRG, the probability of $\mathcal{E}$ occurring changes by at most an additive $\exp(-\Omega(S)) < \delta_2$, and the required seed length for Nisan's PRG is $O(S \log(R/S)) = O(\log(n/\delta_2) \log n)$ bits. For the next branching program, recall that $\mathcal{F}$ is the event that $\|a_{j^*}\|_0 \geq k$. We thus set up a similar branching program, which sees $h(1), \ldots, h(n)$ in order and counts the number of $i \in [n]$ hashing to level j^*. The seed length and error are the same as for the first branching program. We have thus established the following theorem.

Theorem 3.7.3. *For any $k \geq 1$ and $0 < \delta_1, \delta_2 < 1$, there is a solution to the ℓ_0-sampling$_k(\delta_1, \delta_2)$ in turnstile streams with space complexity $O((t \log n + \log(n/\delta_2)) \log(n/t))$ bits, where $t = \max\{k, \log(1/\delta_1)\}$.*

3.8 Conclusion

We have decided the asymptotic behavior of $\mathbf{R}_{\delta}^{\rightarrow,pub}(\mathbf{UR}_k^{\subset})$. For $\delta > 2^{-n^{.99}}$, our sampler/support-finding and duplicate-finding lower bounds are optimal. For other regimes of δ, our sampler lower bound is not necessarily optimal. For example, when $\delta < 2^{-n}$, our result implies a lower bound of $\Omega(n)$ bits. However, if δ is extremely small (for example $\delta = 0$), the lower bound of

support-finding is $\Omega(n \log n)$ bits, because a data structure needs to record each x_i.

One motivation that we studied sampler is because it is used as a building block for graph connectivity. Almost all existing graph connectivity data structure explicitly maintains a spanning forest. When an edge is removed, causing two components no longer connected, samplers are used to find an outgoing edge of a connected component.

[AGM12a] designed a data structure taking $O(n \log^3 n)$ bits of space to maintain a spanning forest in a turnstile graph stream, where n is the number of vertices. Intuitively, a data structure outputting a spanning forest can be viewed as support-finding$_{\Theta(n)}$, and it is possible to adapt the proof in Section 3.5 to give a space lower bound of $\Omega(n \log^2 n)$ bits. [NY19] obtained the optimal lower bound of $\Omega(n \log^3 n)$ bits by reducing the task of solving $\frac{n}{2}$ independent copies of $\mathbf{UR}^{\subset}$ to finding a spanning forest of a bipartite graph. Then they prove a direct product theorem that a one-way communication protocol for n-fold $\mathbf{UR}^{\subset}$ with cost C and failure probability δ gives a new protocol for $\mathbf{UR}^{\subset}$ with cost (under a weaker notion) $\frac{C}{n}$ and failure probability $\sqrt{\frac{\delta}{n}}$.

The space complexity for graph connectivity in turnstile streams remains open: the current best lower bound is $\Omega(n \log n)$ bits while the current best upper bound is $O(n \log^3 n)$ bits.

Chapter 4

Dynamic Graph Connectivity

In this chapter, we present a randomized algorithm for dynamic graph connectivity with failure probability $1/\text{poly}(n)$, which has worst-case running time $O(\log^3 n)$ per edge insertion, $O(\log^4 n)$ per edge deletion, and $O(\log n/\log\log n)$ per query.

For the amortized case, the current best algorithm [HHKP17] achieves expected amortized time $O(\log n(\log\log n)^2)$ per update, and worst-case time $O(\log n/\log\log\log n)$ per query. For the worst case, the current best deterministic algorithm [KKPT15] achieves $O(\sqrt{\frac{n(\log\log n)^2}{\log n}})$ worst-case update time and constant query time. Using the power of randomness, [KKM13] propose a randomized algorithm with failure probability $1/\text{poly}(n)$ which has worst case running time $O(\log^4 n)$ for edge insertion, $O(\log^5 n)$ for edge deletion, and $O(\log n/\log\log n)$ for each query. In this chapter, we show how to improve its update time by a factor of $\log n$.

Organization. This chapter is organized as follows. Section 4.1 is the introduction section which consists of 3 parts: Section 4.1.1 formulates the dynamic graph connectivity problem, Section 4.1.2 reviews some of the related works, Section 4.1.3 summarizes our main results. Section 4.2 desribes the cutset data structure, which is a building block for both [KKM13] and our algorithm. Section 4.2 also compares the invariants maintained by [KKM13] and our algorithm. Section 4.3 presents our new algorithm.

Bibliographic remarks: The result presented in this chapter is based on [Wan15].

4.1 Introduction

4.1.1 Problem formulation

Fully dynamic graph connectivity is a fundamental problem in data structure and graph algorithm design. We formulate the problem as follows.

Problem 4.1.1 (fully dynamic graph connectivity)**.** *Let graph $G = (V, E)$, where V is the set of vertices, and E is the set of edges. We start with $E = \emptyset$, and receive a sequence of operations of the following three forms.*

- INSERT*(e): insert edge e into the graph.*
- DELETE*(e): delete edge e from the graph.*
- QUERY*(u, v): return whether vertex u and v are connected in G.*

In particular, we assume the sequence of operations comes in a online fashion, so that whenever we receive QUERY *we have to answer immediately before receiving any further operations. Both* INSERT *and* DELETE *are called update.*

The goal is to design a data structure minimizing the time to process each operation. Time can be measured in either amortized time or worst-case time. And depending on whether a data structure uses randomness, we classify the data structure as deterministic or randomized one.

4.1.2 Related works

Sleator and Tarjan's ST-tree [ST83], also known as *link-cut tree*, solves the fully dynamic graph connectivity problem for the special case that at any time the graph is a forest. ST-tree is a deterministic data structure that achieves $O(\log n)$ worst-case time for both update and query. If the data structure is allowed to spend more update time, let us say $O(t(n) \log n)$ where $t(n) > 1$, then it can answer the query in $O(\log n / \log t(n))$ worst-case time. Actually, ST-tree is much more powerful: it allows users to associate a weight to each edge, and supports path aggregation queries such as min or sum of the edge weights on the path between two vertices. The main

Reference	Update time	Update type	Query time
[Fre85]	$O(\sqrt{m})$	deterministic worst-case	$O(1)$
[EGIN97]	$O(\sqrt{n})$	deterministic worst-case	$O(1)$
[HK95]	$O(\log^3 n)$	randomized amortized	$O(\log n / \log\log n)$
[HT96]	$O(\log^2 n)$	randomized amortized	$O(\log n / \log\log n)$
[HDT98]	$O(\log^2 n)$	deterministic amortized	$O(\log n / \log\log n)$
[Tho00]	$O(\log n(\log\log n)^3)$	randomized amortized	$O(\log n / \log\log\log n)$
[Wul13]	$O(\log^2 n / \log\log n)$	deterministic amortized	$O(\log n / \log\log n)$
[KKM13]	$O(\log^5 n)$	randomized worst-case	$O(\log n / \log\log n)$
[Wan15] (**this thesis**)	$O(\log^4 n)$	randomized worst-case	$O(\log n / \log\log n)$
[GKKT15]	$O(\log^4 n)$	randomized worst-case	$O(\log n / \log\log n)$
[KKPT15]	$O(\sqrt{\frac{n(\log\log n)^2}{\log n}})$	deterministic worst-case	$O(1)$
[HHKP17]	$O(\log n(\log\log n)^2)$	randomized amortized	$O(\log n / \log\log\log n)$

Table 4.1: *Related results on fully dynamic graph connectivity. Query time in all rows refers to worst-case time.*

algorithmic idea of ST-tree is to decompose tree into paths, maintain them in *globally biased search trees*, and perform *expose* operations (similar to the splay operation in [ST85]) to lift the most recently updated or queried vertices to the root of the tree.

Worst case. [Fre85] designed a deterministic data structure that solves the fully dynamic graph connectivity problem with $O(\sqrt{m})$ worst-case time per update and $O(1)$ worst-case time per query. The data structure of [Fre85] is actually designed to maintain minimum spanning forest for a fully dynamic weighted graph, which is a more general problem than dynamic connectivity. [Fre85] make a transformation called *ternarization* to the graph so that the maximum degree of the vertices is at most 3 (and the number of vertices will become $O(m)$). Then they partition vertices into hierarchical clusters based on the topology of the minimum spanning trees, and maintain the clusters in a data structure called *topology trees*. The goal of hierarchical clustering is to facilitate the lookup of a replacement edge when an edge of the minimum spanning forest is deleted (or the weight of an edge in the minimum spanning forest increases). Their algorithm is involved: they design topology trees to make the update time $O(\sqrt{n\log n})$, and then improve to $O(\sqrt{n})$ by 2-dimensional topology trees. To illustrate the effectiveness of partition into clusters, [Fre85] start with a simple algorithm that takes $O(m^{2/3})$ worst-case update time, where the algorithm

partitions the vertices into $O(m/z)$ clusters and each cluster is of size $O(z)$ so that the operation to one cluster takes $O(z)$ time and enumerating cluster pairs takes $O(m^2/z^2)$ time. By choosing $z = m^{2/3}$, the simple algorithm takes $O(m^{2/3})$ worst-case time per update.

[EGIN97] improved the worst-case update time for dynamic connectivity and minimum spanning forest to $O(\sqrt{n})$ by introducing a technique called *sparsification*. Whenever sparsification is applicable, the problem is reduced to a sparse graph where the number of edges is $O(n)$ instead of $O(m)$. Intuitively, this can be done by partitioning the edges into $O(m/n)$ subgraphs where each subgraph has $O(n)$ edges. For the dynamic graph connectivity problem, a binary tree of height $O(\log(m/n))$ is maintained, where every subgraph of $O(n)$ edges corresponds to a leaf and every parent node merges the spanning forests of its child nodes and obtains a new spanning forest (so that finally the root of the binary tree has the spanning forest of the original graph). Note that the aforementioned sparsification technique gives $O(\sqrt{n}\log(m/n))$ worst-case update time for dynamic connectivity and minimum spanning forest, and a more involved sparsification will improve it by a factor of $\log(m/n)$.

The current best deterministic algorithm [KKPT15] for the worst-case running time achieves $O(\sqrt{\frac{n(\log\log n)^2}{\log n}})$ update time and $O(1)$ query time. Their speedup mainly comes from word-level parallelism. It is a longstanding open question whether there exists a deterministic $\operatorname{poly}\log(n)$-time algorithm per operation in the worst case.

Using the power of randomness, [KKM13] achieved worst-case time $\operatorname{poly}\log(n)$ per operation. They propose a randomized algorithm which has worst-case running time $O(\log^4 n)$ for edge insertion, $O(\log^5 n)$ for edge deletion, and $O(\log n/\log\log n)$ for each query. However, the algorithm is Monte Carlo and has one-sided error. That is, with probability less than $1/n^c$, the algorithm will declare that u and v are not connected even though they are. The key algorithmic idea in [KKM13] is to use ET-trees (introduced in [HK95]) to maintain linear sketches of cut-edge samplers for a dynamic set of vertex-disjoint trees. They call such a data structure *cutset*, which is used for finding an outgoing edge of a dynamic tree. They maintain $\log n + 1$ instances of such data structure, denoted as $cutset_0, \ldots, cutset_{\log n}$, where $cutset_i$ is used to find additional tree edges to insert into $cutset_{i+1}, \ldots, cutset_{\log n}$.

Amortized case. For the amortized case, [HK95] proposed a randomized algorithm with $O(\log^3 n)$ amortized expected time per update and $O(\log n / \log\log n)$ worst-case time per query. Their algorithm is Las Vegas and always gives correct answer to the queries. The algorithm partitions the edges into $O(\log n)$ levels, and denote by $E_1, E_2, \ldots$, where E_1 is the lower level, and they call the subgraph on level i to be $G_i = (V, \cup_{j \le i} E_j)$. The algorithm maintains a spanning forest F of the graph such that for every i, $F_i := F \cap \cup_{j \le i} E_j$ is a spanning forest of G_i. They call edges in F tree edges and edges not in F non-tree edges. They design a data structure called ET-tree (base on Euler tour) to maintain a dynamic set of trees and support efficient information aggregation for subtrees. In contrast, ST-trees [ST83] support efficient information aggregation for paths. The algorithm in [HK95] maintains each F_i in an ET-tree and each vertex in F_i attaches a list of non-tree edges in E_i that are incident to the vertex. By using ET-trees to aggregate those non-tree edges (i.e. candidates for replacement edges), the algorithm can sample from or enumerate them efficiently. When a level-i edge $e \in F$ is deleted, causing the tree containing e in F_i partitioned into T_1 and T_2, the algorithm tries to find a replacement edge reconnecting T_1 and T_2. Without loss of generosity, assume that T_1 has fewer incident non-tree edges. The algorithm then samples $O(\log^2 n)$ uniformly random edges from the incident non-tree edges of T_1. If a replacement edge (i.e. also incident to T_2) is sampled, the algorithm reconnects T_1 and T_2 in F_i with the replacement edge; otherwise the algorithm concludes that the cut of T_1 on level i is sparse (i.e., let A denote the non-tree edges that are incident to F_1, and let B denote the non-tree edges that are incident to both F_1 and F_2, then $|B|/|A| = O(1/\log n)$) and by an amortization argument, the algorithm can afford enumerating A and put all edges in B to level $i+1$.

[HT96] improved the expected amortized update time of [HK95] to $O(\log^2 n)$ per update. The improvement is made by reducing the sample size of [HK95].

[HDT98] proposed a deterministic algorithm that achieves $O(\log^2 n)$ amortized update time and $O(\log n / \log\log n)$ worst-case query time. The techniques used by [HDT98] rely on some intuitions used by [HK95], but the algorithm of [HDT98] is simpler. The algorithm partitions edges into $\log n + 1$ levels, namely $E_0, \ldots, E_{\log n}$, and let the graph on level i be $G_i = (V, \cup_{j \ge i} E_j)$ (note that the top level is $G_0 = G$, which is the opposite in [HK95]). The algorithm also maintains

a spanning forest F such that $F_i := F \cap \cup_{j \geq i} E_j$ is a spanning forest of G_i. In addition, the algorithm maintains the invariant that for any i, the maximum size of connected components in G_i is no greater than $n/2^i$. When a level-i edge $e \in F$ is deleted, causing the tree containing e in F_i partitioned into T_1 and T_2, where without loss of generosity, assume that the number of vertices in T_1 is smaller than T_2. The algorithm first changes all level-i tree edges in T_1 into level $i+1$, and then tries to find a replacement edge reconnecting T_1 and T_2 by enumerating level i non-tree edges incident to T_1. If another endpoint of the edge is incident to T_2, then the algorithm successfully finds the replacement edge; otherwise, if another endpoint of the enumerated edge is also incident to T_1, then the algorithm change its level to $i+1$. Note that in the beginning when an edge is inserted, it will be in level 0. The level of an edge is non-decreasing, and whenever it increases we charge $O(\log n)$ running time with it. By an amortization argument, each update takes $O(\log^2 n)$ amortized time.

A further result by [Tho00] gave a nearly optimal randomized algorithm with expected amortized time $O(\log n(\log\log n)^3)$ per update, and $O(\log n/\log\log\log n)$ worst-case time per query. At a high level, the start point of [Tho00] is the algorithm in [HDT98], and [Tho00] introduces several innovative algorithmic ideas to speedup the process of finding replacement edges, so that on average it takes $O((\log\log n)^3)$ time instead of $O(\log n)$ time per failed attempt. For that purpose, [Tho00] maintain a single hierarchical structure to represent the graph and build a short-cut system to efficiently navigate between certain nodes of the structure. In order to find a replacement edge after deletion of a tree edge, [Tho00] sample in expectation $O(1)$ non-tree edges, and if no replacement edge has been found it will perform a batch test which can be done efficiently and paid by an amortization argument. To facilitate fast (approximate) uniform sampling, [Tho00] maintain a system of approximate counters.

To date, the best deterministic amortized update time is $O(\log^2 n/\log\log n)$ achieved by [Wul13], and its query time is $O(\log n/\log\log n)$ in the worst case. The best randomized amortized update is $O(\log n(\log\log n)^2)$ achieved by [HHKP17], and its query time is $O(\log n/\log\log\log n)$ in the worst case.

Lower bounds. In terms of time lower bound, the current best lower bound for fully dynamic graph connectivity is $\max(t_{\text{update}}, t_{\text{query}}) = \Omega(\log n)$ in [PD04], which improve previous lower bound of $\Omega(\log n / \log\log n)$ in [HF98]. Their lower bound works for the cell probe model, and even holds for amortized running time and the case that the graph is a forest.

4.1.3 Summary of our main results

We improve the update time of [KKM13] by a factor of $\log n$. The improvement mainly comes from reducing the space complexity of samplers used by [KKM13]. We use samplers with weaker guarantees, but we are still able to argue the correctness of the algorithm via a new invariant. In particular, the sampler used by [KKM13] takes $O(\log^2 n)$ words of space, and outputs a sample with probability at least $1 - 1/\text{poly}(n)$ if the support is not empty. In constrast, we use another type of sampler (see Section 3.7) that only takes $O(\log n)$ words of space, so that if the support is empty, the sampler will always output "fail"; if the support is not empty, with probability at most $1/2$ outputs "fail", and otherwise it gives a correct sample with probability at least $1 - 1/\text{poly}(n)$.

Theorem 4.1.2. *There exists a randomized data structure (Algorithm 11) taking $O(n \log^2 n)$ words of space for the fully dynamic graph connectivity problem that has worst-case time $O(\log^3 n)$ per edge insertion, $O(\log^4 n)$ per edge deletion, and $O(\log n / \log\log n)$ per query. For each query, the data structure answers correctly with probability at least $1 - 1/\text{poly}(n)$.*

We note that [GKKT15] also independently improves [KKM13]. As a comparison, the work [GKKT15] also improves the update time for the deletion operation from $O(\log^5 n)$ to $O(\log^4 n)$, but its insertion time is still $O(\log^4 n)$.

4.2 Cutset Data Structure and Invaiants

The framework used by most previous works goes as follows. For given $G = (V, E)$, we maintain a spanning forest for G via a dynamic tree structure supporting the following three kinds of operations efficiently.

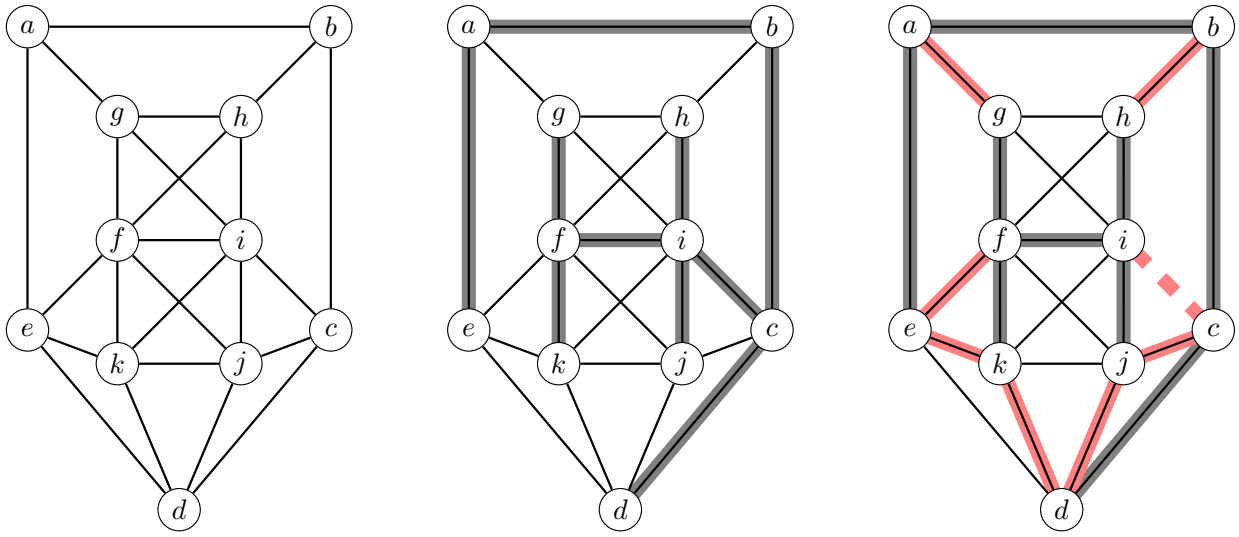
- Link two disjoint trees to form a new tree. Specifically, link(u, v) receives u, v in two different trees of the forest F, then adds the edge (u, v) to F;
- Cut a tree edge so that the tree is split into two;
- Query whether two vertices are in the same tree.

Actually, there are already many good ways to support all these three operations with deterministic worst case running time $O(\log n)$ per operation, since they are standard operations supported by dynamic trees such as the ST-tree [ST83] or ET-tree [HK99]. With a dynamic tree structure, we can deal with the operations in the dynamic connectivity problem as follows:

- When answering a query whether vertices u and v are connected, we just need to check if they are in the same tree in the dynamic tree structure;
- When inserting a new edge, if the edge is across two different trees, we link these two trees via this new edge in the dynamic tree structure so that we keep the invariant that the structure we are maintaining forms a spanning tree for the graph;
- When deleting an edge, if the edge is not a tree edge, then we simply remove the edge without changing the dynamic tree structure.

However, difficulty arises when we insert an edge which connects two vertices in the same tree, or when we delete a tree edge in the dynamic tree structure. Indeed, we can disregard new edges connecting vertices in the same tree, without affecting the invariant that we are maintaining a spanning forest. However, when we delete a tree edge, cutting the tree into two parts which we call A and B, we need to check whether these two parts can reconnect, in order to maintain a spanning forest of G. More specifically, we want to find an edge across A and B, i.e. an edge in the cut $E \cap (A \times B)$[4] (or $E \cap (A \times (V \backslash A))$ because of the invariant) if one exists. Figure 4.1 illustrates such a situation where a tree edge $\{i, c\}$ (the dashed red segment in Figure 4.1 (c)) is deleted, and the algorithm has to find one of 7 possible replacement edges in

[4]Here we denote $A \times B = \{\{a, b\} : a \in A, b \in B\}$, which is a slight abuse of notation.



(a) an undirected graph (b) spanning forest (c) edge $\{i, c\}$ is deleted

Figure 4.1: *Illustration of replacement edges for dynamic graph connectivity. As illustrated in (a), we have an undirected graph G with $n = 11$ vertices and $m = 24$ edges. In (b), we illustrate the spanning forest of G (tree edges are highlighted in grey; in this example, it's a tree since the graph is connected) maintained by a specific dynamic connectivity algorithm. In (c), we illustrate the possible replacement edges (highlighted in red) when tree edge $\{i, c\}$ (the dashed red segment) is deleted.*

$\{\{a, g\}, \{b, h\}, \{e, f\}, \{e, k\}, \{k, d\}, \{j, d\}, \{j, c\}\}$ (highlighted in red in Figure 4.1 (c)). Thus, if we follow this natural framework, we need to find a clever method to find an edge in the cut. Also, we cannot afford doing nothing during insertions of edges whose endpoints are already in the same tree, since such edges may become important after later deletions of tree edges.

One naive way is to undergo a brute force search of all possible edges. For example, iterate over all edges adjacent to A, and check whether there is one such edge with the other end in B. One breakthrough made in [HT96] is by observing that we can organize the edges in a clever way so that every edge is checked at most $O(\log n)$ times. Since every time an edge is checked one dynamic tree operation is performed, they get an amortized time $O(\log^2 n)$ per update. A further improvement was made to amortized time $O(\log n(\log \log n)^3)$ in [Tho00] via a more delicate data structure and randomized sampling technique. [KKM13] proposed a data structure to find such an edge in $\operatorname{poly} \log(n)$ worst case time with the guarantee that if there is at least one edge in the cut, the data structure will find one with probability at least $1 - 1/\operatorname{poly}(n)$. It is mainly based

on the following two observations.

1. First, if there is exactly one edge in the cut $E \cap (A \times B)$, then the edge can be found in $O(\log n)$ worst time by a so-called XOR trick. It goes as follows: assign each edge a unique name (from 1 to $\binom{n}{2}$, for example), and for each vertex v, maintain the xor value of the names of edges adjacent to v, which we call $xor(v)$. Moreover, let $xor(A) = \bigoplus_{v \in A} xor(v)$ where A is a set. One important property of the value is that for a vertex set $S \subseteq V$, $\bigoplus_{v \in S} xor(v)$ equals the XOR of the edges in the cut $E \cap (S \times (V \backslash S))$. Therefore, if in the dynamic tree structure we also maintain the XOR value of $xor(v)$ for all vertices v in a tree, then we can get the name of the only edge in the cut, which is $xor(A)$. The ET (Euler tour) tree introduced in [HK99] can be used to maintain these XOR values (that is, we maintain a XOR value for each tree) in $O(\log n)$ time per operation in the worst case.

2. Second, if there is more than one edge in the cut then we can use a sampling method as follows. For our graph $G = (V, E)$, we keep $2 \log n + 1$ subgraphs of G, namely $G_0, \ldots, G_{2 \log n}$, where $G_i = (V, E_i)$ is subsampled from G by selecting each edge in E independently with probability $1/2^i$, i.e., $\forall e \in E, \mathbb{P}(e \in E_i) = 1/2^i$. Now instead of keeping track of the *xor* values only on the graph $G = G_0$, we also maintain the *xor* values for G_i, for $i = 1, \ldots, 2 \log n$. It is proven in [KKM13] that if $E \cap (A \times B) \neq \emptyset$ and A, B are independent from $E_1, \ldots, E_{2 \log n}$, with probability more than a positive constant, there is exactly one edge in $E_i \cap (A \times B)$ for some i. So with probability greater than a positive constant, we can find an edge in a non-empty cut. If we duplicate $O(\log n)$ copies with independent randomness, we can boost the success possibility to $1 - 1/n^c$. In order to know whether a level succeeded, it is sufficient to store all edges E in a hash table, so we can check whether the XOR is actually a real edge by looking in the hash table in $O(1)$ time.

Note that the above observations can be also formulated as linear sketches of samplers as in [AGM12a]. Lemma 4.2.1 below generalizes the cutset data structure implemented in [KKM13].

Lemma 4.2.1 (Cutset data structure with failure probability δ)**.** *There is a data structure maintaining a* dynamic forest $F = \{T_i = \{V_{T_i}, E_{T_i}\}, i = 1, \ldots\}$ *for a dynamic graph* $G = (V, E)$*, where* $\{V_{T_i}\}$ *is*

a partition of V, and denote $E_F = \cup_i E_{T_i}$ as tree edges. It supports the following operations, with the requirement that when invoking function insertTreeEdge *or* deleteTreeEdge*, the input edge e must be independent from the (random) outputs obtained by all previous calls to the function* outgoingEdge*.*

- insertTreeEdge$(e = \{u, v\})$*: insert edge e into the forest F, combining the tree containing u and the one containing v together to be a single tree via e;*
- deleteTreeEdge$(e = \{u, v\})$*: remove edge e from the tree containing it, splitting the tree into two new trees;*
- isTreeEdge(e)*: return whether e is a tree edge;*
- insertEdge(e)*: insert edge e into E;*
- deleteEdge(e)*: remove edge e from E;*
- tree(v)*: return the name of the tree containing vertex v (each tree has a unique name);*
- outgoingEdge(T)*: for input $T \in F$ (T is specified by its name), let $C_T = E \cap (V_T \times (V \backslash V_T))$. (1) If $C_T = \emptyset$, return $\emptyset$; (2) If $C_T \neq \emptyset$, return an edge $e \in C_T$ with probability at least $1 - \delta$ or return $\emptyset$ otherwise.*

insertTreeEdge, deleteTreeEdge, insertEdge *and* deleteEdge *have worst-case time $O(\log^2 n \log \delta^{-1})$ per invocation;* outgoingEdge *has worst-case time $O(\log n \log \delta^{-1})$ per invocation;* tree *has worst-case time $O(\log n / \log \log n)$ per invocation. The data structure support function* isTreeEdge *by maintaining a hash table, which takes $O(1)$ worst-case time.*

Moreover, the data structure can be implemented in $O(n \log n \log \delta^{-1})$ words, but the guarantee of outgoingEdge *becomes worse if $C_T \neq \emptyset$:* outgoingEdge(T) *returns an edge in C_T with probability at least $1 - \delta$, returns an edge not in C_T with probability at most $\delta / \text{poly}(n)$, returns $\emptyset$ otherwise.*

From now on, we will use the cutset data structure as a black box to construct dynamic connectivity algorithms. The cutset data structure cannot directly be applied to solve the dynamic connectivity problem for the following reason. Suppose we use it to maintain the spanning forest of the graph, after deleting a tree edge (cutting the tree into two parts which we call A

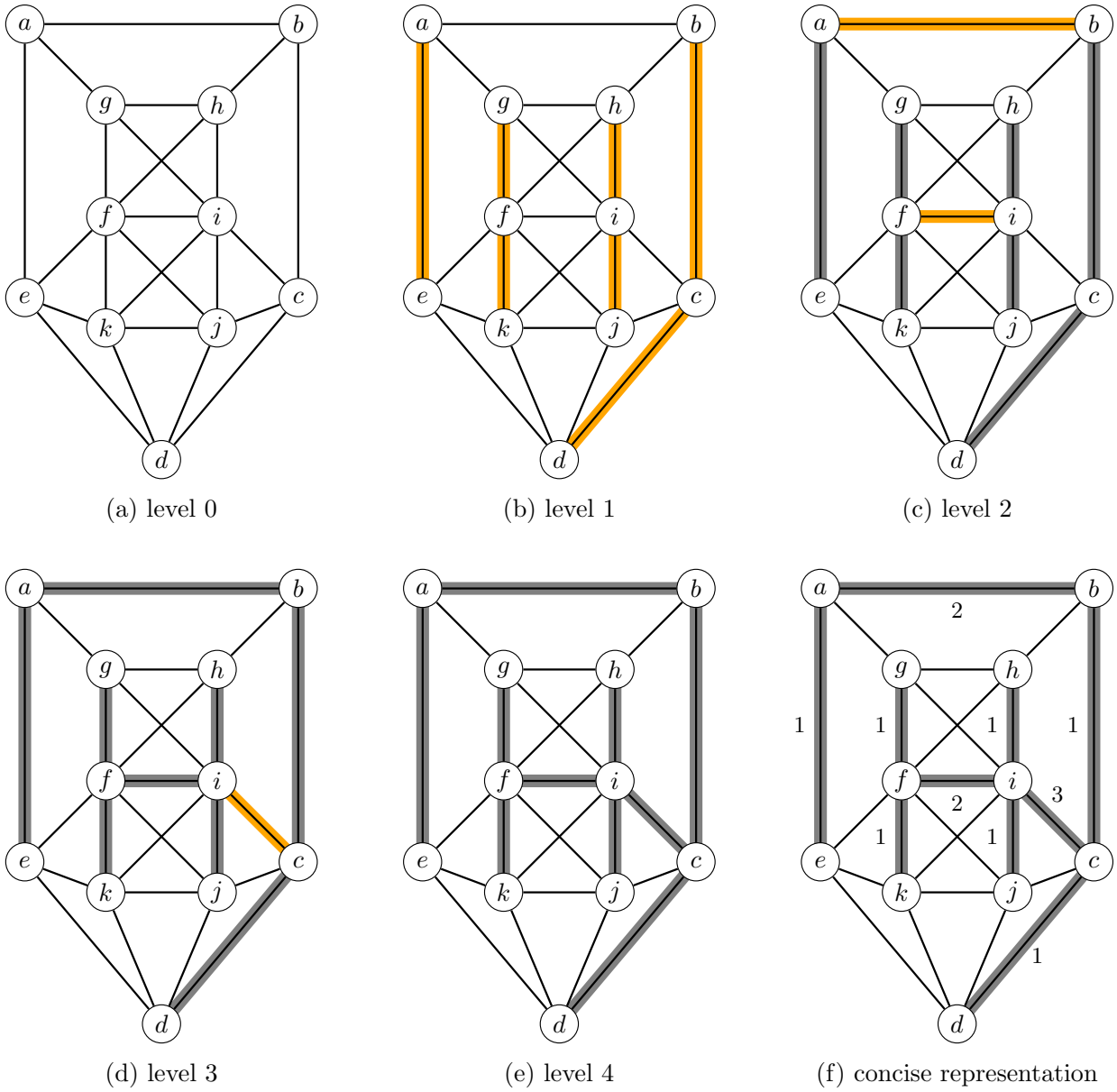
and B). Even though we can find an edge $e \in E \cap (A \times B) \neq \emptyset$ by invoking OUTGOINGEDGE, we cannot insert e as a new tree edge into the cutset data structure because of the requirement in Lemma 4.2.1 that inserted tree edges be independent of all previous calls to OUTGOINGEDGE. [KKM13] circumvent this issue by implicitly maintaining the structure of a Boruvka tree. A Boruvka tree records the process of how the vertices merge together to form larger and larger connected components. Starting from isolated nodes on layer 0, i.e. single vertices of G, a Boruvka tree is built by coupling connected nodes[5] on the i-th layer to form the nodes on the $(i+1)$-st layer, and we stop the process until we cannot merge any two nodes on some layer.

Specifically, [KKM13] maintain $\ell + 1$ forests $F_0, \ldots, F_\ell$, where $\ell = \lceil \log n \rceil$. We refer to F_i as the forest at layer i for $i = 0, \ldots, \ell$. They use a cutset data structure $cutset_i$ with failure probability $1/\text{poly}(n)$ for each F_i. For each $i > 0$, F_i contains all the edges in F_{i-1}, and $cutset_{i-1}$ is used to provide additional edges to F_i (that is, $E_{F_0} \subseteq \ldots \subseteq E_{F_\ell}$). For a tree T on any layer i where $0 \leq i < \ell$, we say it is *merged* (or *matched*) with a partner iff the tree containing T on layer $i+1$ is strictly larger than T. [KKM13] maintains the invariant (with probability $1 - 1/\text{poly}(n)$) that any non-maximal tree on any layer i is merged with a partner. That is to say,

Invariant 4.2.2. *For any layer $i = 0, \ldots, \ell - 1$, for any tree $T \in F_i$, if T is not a maximal tree in G, then there is a tree $T' \in F_{i+1}$ such that $V_T \subsetneq V_{T'}$ (i.e. non-maximal tree T on layer i is merged with a partner).*

Figure 4.2 illustrates an example execution of [KKM13] on the graph depicted in Figure 4.1. If Invariant 4.2.2 holds, then F_ℓ is a spanning forest for G, because every non-maximal tree on layer i has size at least 2^i, so that every tree on layer ℓ is maximal. To answer QUERY(u, v), it is sufficient to check whether u and v are contained in the same tree in F_ℓ by deciding whether $cutset_\ell$.TREE$(u) = cutset_\ell$.TREE(v), which takes time $O(\log n / \log\log n)$. To maintain Invariant 4.2.2, [KKM13] perform $O(\log n)$ operations (other than TREE) of the cutset data structure for each insertion, and $O(\log^2 n)$ operations (other than TREE) of the cutset data structure for each deletion. Because the cutset data structure with failure probability $1/\text{poly}(n)$ has worst-case time

[5] A node in a Boruvka tree corresponds to a set of vertices connected to each other. We say two nodes in a Boruvka tree are connected if there is an edge across the vertex sets corresponding these two nodes. If on the i-th layer one node is not matched with another node, then we just make an identical node on the $(i+1)$-st layer, so that on each layer the nodes give a partition of V.



(a) level 0 (b) level 1 (c) level 2

(d) level 3 (e) level 4 (f) concise representation

Figure 4.2: *An example execution of [KKM13] on the graph illustrated in Figure 4.1. In this example,* $n = 11$ *and* $\ell = \lceil \log n \rceil = 4$*. The algorithm maintains* $\ell + 1 = 5$ *levels (namely level* $0, \ldots, 4$*) depicted by (a) – (e). For level* i*, where* $i \in \{0, \ldots, 4\}$*, the algorithm maintains dynamic forest* F_i*, whose edges are highlighted in either grey or orange, where orange represents the edges found by* $cutset_{i-1}$*. In (f), we give a concise representation of (a) – (e), where the label to each tree edge represents the smallest level in which it becomes a tree edge.*

$O(\log^3 n)$ per operation, the worst-case running time of [KKM13] is $O(\log^4 n)$ per insertion and $O(\log^5 n)$ per deletion.

4.2.1 Our techniques

We observe that it is possible to use the cutset data structure with failure probability $1/2$ instead of $1/\text{poly}(n)$, at the cost of only a constant factor of blowup in ℓ. That is, let $\ell = C \log n$ where C is a large constant. We can prove that these ℓ layers are sufficient to ensure that F_ℓ is a spanning forest with high probability, by "almost" maintaining Invariant 4.2.3 in our new algorithm.

Invariant 4.2.3. *For any layer $i = 0, \ldots, \ell - 1$, conditioned on the randomness of layer $0, \ldots, i-1$ (that is, the randomness used in $cutset_0, \ldots, cutset_{i-1}$), for any tree $T \in F_i$ such that T is not a maximal tree in G, the probability (over the randomness of $cutset_i$) that there is a tree $T' \in F_{i+1}$ such that $V_T \subsetneq V_{T'}$ (i.e. non-maximal tree T on layer i is merged with a partner) is at least $1/2$.*

More precisely, because there is a subtle dependence issue prohibiting us from arguing correctness directly for our new algorithm, we adapt a coupling method and instead prove that there is another algorithm (whose output is exactly identical to our new algorithm with probability $1 - 1/\text{poly}(n)$) that maintains Invariant 4.2.3. Furthermore, we show that Invariant 4.2.3 is sufficient to prove that F_ℓ is a spanning forest with probability $1 - 1/\text{poly}(n)$.

4.3 Our Dynamic Graph Connectivity Algorithm

In this section, we present our new algorithm with worst-case time $O(\log^3 n)$ per insertion and $O(\log^4 n)$ per deletion. Our algorithm maintains $\ell + 1$ layers where $\ell = C \log n$. For each layer i ($0 \le i \le \ell$), we maintain a dynamic forest F_i of the dynamic graph $G = (V, E)$ by using a cutset data structure with failure probability $1/2$, denoted as $cutset_i$. At all times we have for each $i > 0$, F_i contains all the edges in F_{i-1}, so $E_{F_0} \subseteq \ldots \subseteq E_{F_\ell}$. The data structure $cutset_{i-1}$ is used to provide additional edges to F_i, for $1 \le i \le \ell$. That is, when a tree edge is deleted from layer i splitting some tree T into T_1, T_2, we attempt to find new matches for T_1 and T_2 using $cutset_{i-1}$. Our goal is to make F_ℓ a spanning forest of G, so that queries can be answered efficiently.

We implement INSERT and DELETE in Algorithm 11 for our dynamic graph $G = (V, E)$, where initially $E = \emptyset$. UPDATE is an auxiliary function for DELETE. For QUERY(u, v), we simply output $cutset_\ell.\text{TREE}(u) = cutset_\ell.\text{TREE}(v)$. In our implementation, we also need to maintain the size of the

Algorithm 11 Dynamic connectivity with $O(\log^3 n)$ insertion time and $O(\log^4 n)$ deletion time

```
1: procedure INSERT(e = {u, v})
2:     for i = 0, ..., ℓ do
3:         cutset_i.INSERTEDGE(e)
4:     end for
5:     if cutset_ℓ.TREE(u) ≠ cutset_ℓ.TREE(v) then
6:         for i = 0, ..., ℓ do
7:             cutset_i.INSERTTREEEDGE(e)
8:         end for
9:     end if
10: end procedure
11: procedure DELETE(e = {u, v})
12:     for i = 0, ..., ℓ do
13:         cutset_i.DELETEEDGE(e)
14:     end for
15:     for i = 0, ..., ℓ do
16:         if e ∈ E_{F_i} then          ▷ membership can be tested by maintaining a Boolean vector
17:             cutset_i.DELETETREEEDGE(e)
18:         end if
19:     end for
20:     for i = 0, ..., ℓ − 1 do
21:         UPDATE(i, u)                 ▷ Algorithm 12
22:         UPDATE(i, v)
23:     end for
24: end procedure
```

trees on each layer in order to check whether a tree is merged with a partner. It can be supported by the ET tree [HK99], for which the running time is $O(\log n)$ in the worst case per operation. Initially any F_i has no tree edges.

In Algorithm 11, when inserting an edge $e = \{u, v\}$, (1) we first insert e into $cutset_i$ for every i, because now e belongs to G; (2) If u and v were not connected in F_ℓ before inserting e, then we add e as tree edges for every F_i. When deleting an edge $e = \{u, v\}$, (1) we first remove e from $cutset_i$ for every i (note that at this point it might be possible that e is a tree edge, but at the end of the delete operation e will not be in any F_i); (2) For every layer i, if e is an edge in F_i, then remove it from F_i; (3) Start with $i = 0$ and go up to $\ell - 1$, we perform Update for both u and v on layer i. An update operation for u on layer i ensures that if the tree containing u in F_i (denoted by T_u) is non-maximal, then with probability at least $1/2$, T_u is merged with a partner. In UPDATE(i, u),

Algorithm 12 Axillary procedure for Algorithm 11

```
1: procedure UPDATE(i, u)
2:     if size(cutset_i.TREE(u)) < size(cutset_{i+1}.TREE(u)) then
3:         return
4:     end if
5:     {v, w} = e' ← cutset_i.OUTGOINGEDGE(cutset_i.TREE(u))
6:     if e' = null then
7:         return
8:     end if
9:     for k = i + 1, ..., ℓ do
10:        if cutset_k.TREE(v) = cutset_k.TREE(w) then
11:            Find edge e'' connecting cutset_{k-1}.TREE(v) and cutset_{k-1}.TREE(w) in E_{F_k}
12:            for j = k, ..., ℓ do
13:                cutset_j.DELETETREEEDGE(e'')
14:            end for
15:            break
16:            for j = i + 1, ..., ℓ do
17:                cutset_i.INSERTTREEEDGE(e')
18:            end for
19:        end if
20:    end for
21: end procedure
```

we first check whether T_u is merged with a partner. If it is not, we do as follows: try to find an outgoing edge e' for T_u using $cutset_i$, if we find one, then insert e' into $F_{i+1}, \ldots, F_\ell$. Note that we might need to first remove an edge e'' from $F_k, \ldots, F_\ell$ for some $k > i$ to ensure that no cycle is formed. As shown in [KKM13, Section 4.3], the cycle edge e'' can be found in $O(\log n)$ time in the worst case, by maintaining a dynamic data structure whose update time is bounded by $O(\log^2 n)$ for each edge insertion or deletion to the dynamic graph G.

Because only $O(\log n)$ cutset data structure operations are performed in INSERT, and $O(\log^2 n)$ cutset data structure operations are performed in DELETE (including ones performed in UPDATE), the time complexity for insertion is $O(\log^3 n)$; the time complexity for deletion is $O(\log^4 n)$.

In order to prove its correctness, we use a coupling argument. We design Algorithm 11′ identical to Algorithm 11 (using exactly the same random bits) except for the implementation of DELETE. The only change is that we replace "$cutset_\ell.\text{TREE}(u) \neq cutset_\ell.\text{TREE}(v)$" in Algorithm 11 to be "u and v are not connected in G". For that purpose, Algorithm 11′ will record the whole

graph G. We don't care about the efficiency of Algorithm 11′ since it is only a tool used in the analysis. We first prove that Algorithm 11′ maintains Invariant 4.2.3 in Lemma 4.3.1, and then in Theorem 1 we prove that Invariant 4.2.3 ensures that F_ℓ in Algorithm 11′ is a spanning forest of G with overwhelming probability. F_ℓ being a spanning forest implies the algorithm answers correctly. Finally in Lemma 4.3.3, we prove that with overwhelming probability the outputs of Algorithm 11 and Algorithm 11′ are identical, and thus prove the correctness of Algorithm 11.

Lemma 4.3.1. *Algorithm 11′ maintains Invariant 4.2.3.*

Proof. We can prove the lemma by induction on the number of updates to the graph. Assume it holds for the first t updates. For the inductive step, we consider $(t+1)$-th update:

Case 1: (we insert an edge $\{u, v\}$ where u and v were already connected in G) In this case, we do not change any tree. So the probability remains the same that any non-maximal tree in any layer i is merged with a partner. Moreover, this insertion does not change any maximal tree into being non-maximal.

Case 2: (we insert an edge $\{u, v\}$ where u and v were not previously connected in G) Let T_u and T_v be the trees containing u and v respectively on layer i (here we consider every $i = 0, \ldots, \ell - 1$). If both T_u and T_v are maximal, then the combined tree (denoted by T_{uv}) is also maximal, which does not affect the invariant. If either T_u or T_v is not maximal (say T_u is not maximal), then T_{uv} is also non-maximal. But now the probability that T_{uv} is merged with a partner is at least the probability that T_u is merged with a partner after the first t updates, making the probability no less than $1/2$ by induction.

For non-maximal trees that do not contain u or v, the probability related to the invariant is unchanged.

Case 3: (we delete an edge $e = \{u, v\}$) We consider every layer i ($i = 0, \ldots, \ell - 1$) separately. For non-maximal trees that do not contain u or v, the probability related to the invariant is unchanged; for trees containing u or v, we analyze the following two cases separately.

(a) In the first case we consider, e is a tree edge in layer i. That is, T_{uv} is split into T_u and T_v after removing e. Then the deletion algorithm will try to find a partner for T_u and

T_v separately. If T_u is not maximal, then over the randomness on layer i (that is, the randomness in $cutset_i$), the probability that it is merged with a partner is at least $1/2$ via invoking $\textsc{update}(i, u)$ (and similarly for T_v).

(b) In the second case, e is not a tree edge in layer i. Then Algorithm 11′ tries to merge T_u (and also T_v) with a partner via invoking $\textsc{update}(i, u)$. This ensures both T_u and T_v satisfy the invariant.

□

We assume that m (where m is the number of operations to the dynamic graph) is upper bounded by n^{c_1} where c_1 is a constant. Let $c = 10c_1 + 10$, and let $C = 50c$ where C is the constant appearing in $\ell = C \log n$.

Theorem 4.3.2. *If Algorithm 11′ maintains Invariant 4.2.3, then with probability greater than $1 - 1/n^c$, F_ℓ is a spanning forest for G.*

Proof. Let X_i denote the number of non-maximal trees on layer i, for $0 \le i \le \ell$. It is sufficient to prove that $\mathbb{P}[X_\ell > 0] \le n^{-c}$. We have $X_0 \le n$, and X_i is non-increasing. Let Y_i ($1 \le i \le \ell$) denote the indictor random variable for the event that $X_{i-1} = 0$ or $X_i / X_{i-1} \le 7/8$. Because of the invariant, $\mathbb{E}_{cutset_i}[X_i | X_0, \ldots, X_{i-1}] \le \frac{3}{4} X_{i-1}$. By Markov inequality, we have $\mathbb{P}(Y_i = 1) \ge 1/7$ conditioned on $Y_1, \ldots, Y_{i-1}$. By coupling with n independently distributed Bernoulli$(1/7)$ and then applying Chernoff bound, we have

$$\mathbb{P}(X_\ell > 0) \le \mathbb{P}(X_\ell \ge 1) \le \mathbb{P}(\sum_{i=1}^{\ell} Y_i \le 8 \log n) \le e^{-\frac{C \log n}{21}(1 - 56/C)^2} < n^{-c}.$$

□

Lemma 4.3.3. *For any dynamic graph operations of length m, over the randomness used in the cutset data structure, the outputs of Algorithm 11 and Algorithm 11′ are identical for all the queries with probability at least $1 - n^{-c/2}$.*

Proof. As long as F_ℓ is a spanning forest, the cutset data structures in Algorithm 11 and Algorithm 11′ are exactly the same. So the probability is upper bounded by $n^{-c} \cdot m < n^{-c/2}$ by taking a union bound and then applying Theorem 4.3.2. □

Chapter 5

Parallel Graph Connectivity

In this chapter, we present a graph connectivity algorithm that runs in $O(\log \mathrm{diam}(G) \cdot \log \log_{M/n} n)$ parallel time under the Massive Parallel Computing (MPC) model for undirected graph G with n vertices and total memory constraint $\Theta(M)$. If G is disconnected, $\mathrm{diam}(G)$ refers to the largest diameter among all its connected components.

Organization. This chapter is organized as follows. Section 5.1 is the introduction section which consists of 4 parts: Section 5.1.1 introduces the Massive Parallel Computing (MPC) model and formulates the graph connectivity problem in the MPC model, Section 5.1.2 reviews some of the related works on parallel graph connectivity, Section 5.1.3 summarizes our main results covered in this chapter. Section 5.1.4 gives a high-level description of our approach. Our new algorithm consists of $O(\log \log_{M/n} n)$ reduction steps, where each reduction step first increases the number of neighbors for every vertex by a *truncated broadcasting* scheme (in Section 5.2), and then reduces the problem size by a *random leader contraction* step (in Section 5.3). Section 5.4 presents an axillary procedure that contracts trees. Section 5.5 presents our new graph connectivity algorithm in the batch fashion, and Section 5.6 discusses its MPC implementation.

Bibliographic remarks: The results described in this chapter were developed jointly with Alexandr Andoni, Clifford Stein, Zhao Song, and Peilin Zhong [ASS+18].

5.1 Introduction

5.1.1 Problem formulation

The MPC model [BKS13] gives an elegant theoretical characterization for modern parallel computing architectures such as map-reduce. The model can be specified by two parameters γ and δ (we call it (γ, δ)-MPC model), where $\gamma \geq 0$ captures the total space constraint and $\delta > 0$ captures the local memory size on each processor (or machine, we use them interchangeably in the thesis).

Definition 5.1.1 ($(\gamma, \delta) -$ MPC model)**.** *Fix parameters $\gamma \geq 0, \delta > 0$, and suppose $N \geq 1$ is the input size. There are $p \geq 1$ processors each with local memory size $s = \Theta(N^{\delta})$, such that total memory $M = p \cdot s = O(N^{1+\gamma})$. The space size is measured by words, each of $\Theta(\log(s \cdot p))$ bits. The input is distributed on the local memory of $\Theta(N/s)$ input machines. The computation proceeds in rounds. In each round, each machine performs computation on the data in its local memory, and sends messages to other machines at the end of the round. The total size of messages sent or received by a machine in a round is bounded by s. In the next round, each machine only holds the received messages in its local memory. At the end of the computation, the output is distributed on the output machines. Input/output machines and other machines are identical except that input/output machine can hold a part of the input/output. The parallel time of an algorithm is the number of rounds needed to finish the computation.*

In this model, the space per machine is sublinear in N, and the total space is only an $O(N^{\gamma})$ factor more than the input size N. In this chapter, we consider the case when δ is an arbitrary constant in $(0, 1)$. Our results are for both the most restrictive case of $\gamma = 0$ (total space is linear in the input size), as well as $\gamma > 0$ (for which our algorithms are a bit faster). The model from Definition 5.1.1 matches the model MPC(ε) from [BKS13] with $\varepsilon = \gamma/(1 + \gamma - \delta)$ and the number of machines $p = O(N^{1+\gamma-\delta})$. The most restricted case is when $\gamma = 0$. In this case $(0, \delta) -$ MPC model is exactly the same as MPC(0) model. Since in this chapter, the total space used by the algorithm is more involved, we use (γ, δ) to characterize the model, while in [BKS13] they use parameter ε to characterize the repetition of the data.

In the graph connectivity problem, the goal is to output the connected components of an input graph G, i.e. at the end of the computation, $\forall v \in V$, there is a unique tuple (x, y) with $x = v$

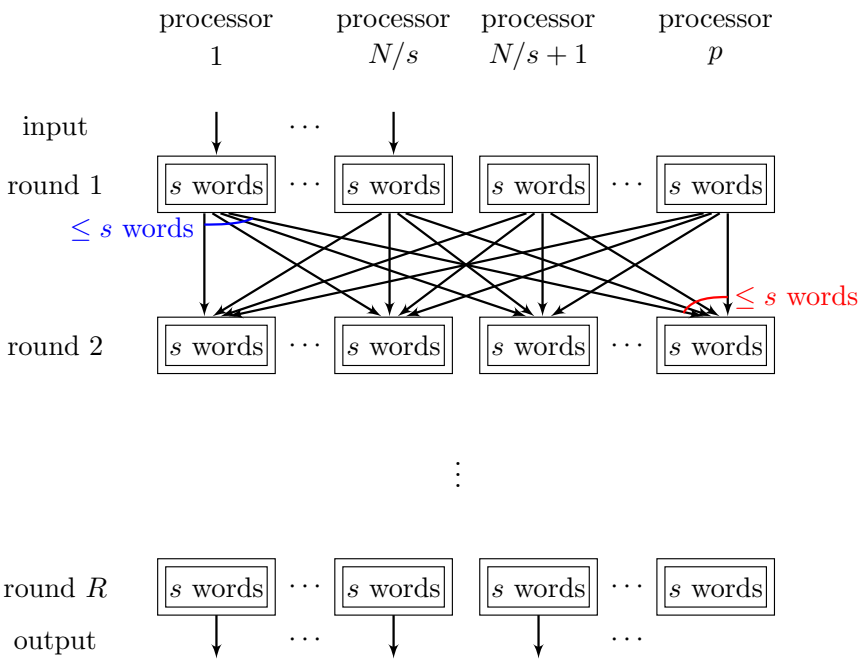


Figure 5.1: *Illustration of the* MPC *model. In the* (γ, δ)-MPC *model, there are* p *processes, each of which has* $s = \Theta(N^\delta)$ *words of space. The total memory of this* MPC *model is* $M = p \cdot s = N^{1+\gamma}$*. The input size is* N*, and in the beginning of round* 1*, the first* N/s *processors will receive a chunk of the input. In each round, each process can send at most* s *words of message, and receive at most* s *words of message. After* R *rounds, all the processors decide to finish and some processors provide output. We call* R *the parallel time of this* MPC *model.*

stored on an output machine, where y is called the color of v. Any two vertices u, v have the same color if and only if they are in the same connected component.

Problem 5.1.2 (Parallel graph connectivity / connected components)**.** *We are given an undirected graph* $G = (V, E)$ *where* $|V| = n$ *and* $|E| = m$*. Let* $N = m$ *denote the input size. The inputs to the* (γ, δ)-MPC *model is the edge list* E*, where in the first round, the first processor receives the first* N^δ *edges in* E *as its input, and the second processor receives the next* N^δ *edges in* E*, . . ., and the* $N^{1-\delta}$*-th processor receives the last* N^δ *edges in* E*. At the end of the computation, the machines altogether are required to output* n *tuples in the form of* (x, y)*, where every* $v \in V$ *appears as* x *exactly once and we write its corresponding* y *as* $c(v)$*, so that* $c(v) = c(v')$ *iff* v *and* v' *are connected in* G*.*

Reference	Model	Parallel time	Total space	Comment
[SV82]	CRCW PRAM	$\log n$	m	m processors
[KNP92]	EREW PRAM	$\log n$	m	m processors
[ASS+18] (**this thesis**)	MPC	$\log \mathrm{diam}(G) \cdot \log \log_{M/n} n$	M	
[ASW18]	MPC	$\log \log n + \log(1/\lambda)$	$\widetilde{O}(n/\lambda^2)$	λ is spectral gap[6]

Table 5.1: *Some related results on parallel graph connectivity. We drop $O(\cdot)$ for simplicity.*

5.1.2 Related works

Graph connectivity has been extensively studied in the PRAM model. The strongest model of the PRAM family is CRCW (concurrent read concurrent write) PRAM, where if multiple processors concurrently write to the same memory, one of them will succeed. [SV82] design a CRCW PRAM algorithm for graph connectivity which takes $O(\log n)$ parallel time. [KNP92] design a $O(\log n)$ parallel time algorithm for graph connectivity under the EREW (exclusive read exclusive write) PRAM model, the weakest model in the PRAM family. It is known (implied by [CDR86] and [DKR94]) that $\Omega(\log n)$ is a time lower bound for graph connectivity in CREW (concurrent read exclusive write) PRAM.

While we would like *fully scalable* algorithms—which work for any value of $\delta > 0$—there have been graph algorithms that use space close to the number of vertices n of the graph. In particular, the result of [LMSV11] showed a faster algorithm for the setting when the space per machine is polynomially larger than the number of vertices, i.e., $s \geq n^{1+\Omega(1)}$, and hence the number of edges is necessarily $m \geq n^{1+\Omega(1)}$. In fact, similar space restrictions are pervasive for all known sub-logarithmic time graph algorithms, which require $s = \Omega(\frac{n}{\log^{O(1)} n})$ [LMSV11, AG18, AK17, CLM+18] (the only exception is [ANOY14] who consider geometric graphs). We highlight the work of [CLM+18], who manage to obtain *slightly sublinear* space of $n/\log^{\Omega(1)} n$ in $\log^{O(1)} \log n$ parallel time, for the approximate matching problem and [ABB+17] who obtain *slightly sublinear* space of $n/\log^{\Omega(1)} n$ in $O(\log \log n)$ parallel time. We note that the space of $\sim n$ also coincides with the space barrier of the semi-streaming model: essentially no graph problems are solvable in less than n space in the streaming model, unless we have many more passes; see for example the survey [McG09].

In fact, some previous MPC algorithms for connectivity in the applied communities have been *conjectured* to obtain $O(\log \mathrm{diam}(G))$ time [RMCS13]; alas, we show in [ASS+18] the algorithm of [RMCS13] has a lower bound of $\Omega(\log n)$ time.

[ASW18] also give a connectivity algorithm in the MPC model but with different guarantees. In particular, their runtime is parameterized as a function of λ, which is a lower bound on the spectral gap[6] of the connected components of G. For a graph G with n vertices and $m = \widetilde{O}(n)$ edges, their algorithm runs in $O(\log\log n + \log(1/\lambda))$ parallel time and uses $\widetilde{O}(n/\lambda^2)$ total space. In contrast, our algorithm has a runtime of $O(\log \mathrm{diam}(G) \cdot \log\log_{M/n} n)$, and $M = \Omega(m)$ is the total space available. To compare the two runtimes, we note that: (1) $\mathrm{diam}(G) \leq O(\frac{\log n}{\lambda})$ for any undirected graph G; and (2) there exist sparse graphs[7] G with n vertices and $O(n)$ edges such that $\frac{1}{\lambda} \geq \mathrm{diam}(G) \cdot n^{\Omega(1)}$ and $\mathrm{diam}(G) \leq O(\log n)$. Thus, our results subsume [ASW18] in the case when total space is $N = n^{1+\Omega(1)}$, but are incomparable otherwise.

5.1.3 Summary of our main results

Theorem 5.1.3 (Connectivity in MPC)**.** *For any $\gamma \in [0,2]$ and any constant $\delta \in (0,1)$, there is a randomized $(\gamma,\delta) -$* MPC *algorithm (see Algorithm 15) which outputs the connected components of the graph G in $O(\min(\log \mathrm{diam}(G) \cdot \log \frac{\log n}{\log(N^{1+\gamma}/n)}, \log n))$ parallel time. The success probability is at least* 0.98*. In addition, if the algorithm fails, then it returns* FAIL*.*

Notice that in the most restrictive case of $\gamma = 0$ and $m = \Theta(n)$, we obtain $O(\min(\log \mathrm{diam}(G) \cdot \log\log n, \log n))$ time. When the total space is slightly larger or the graph is slightly denser (i.e. $\gamma > c$ or $\log_n m > 1 + c$, where $c > 0$ is an arbitrarily small constant) then we obtain $O(\log \mathrm{diam}(G))$ time. Parameterizing as a function of $\mathrm{diam}(G)$ is standard, say, in the distributed computing literature [PRS16, HHW18].

[6]The spectral gap of a graph G is the second smallest eigenvalue of the normalized Laplacian of G.

[7]We can construct G as the following: a bridge connects two 3-regular expanders where each expander has $n/2$ vertices.

5.1.4 Our approach

Our parallel graph connectivity algorithm follows from a general technique called *double exponential speed problem size reduction*, where the reduction is implemented as a *truncated broadcasting scheme* followed by a *random leader contraction* step.

The basic idea of reducing a problem size double exponentially is: if we can reduce a problem of size N to size $\frac{N}{(M/N)^{\Theta(1)}}$ in one single round, then we can reduce the problem of size N to size $O(1)$ in $O(\log \log_{M/N} N)$ rounds. For graph connectivity problem, we consider n, the number of vertices, to be the size of the problem.

In order to implement such a reduction step, we first find a graph transformation to increase vertex degrees, while at the same time keep the new graph as sparse as possible and preserve the connectivity information. Consider the classical neighbor broadcasting scheme, where in each round, each vertex sends its neighbor set to all its neighbors, and then each vertex merges all sets received to its neighbor set. It is easy to see the scheme finishes in $O(\log \mathrm{diam}(G))$ rounds, and each neighbor set ends up to be all the vertices in its connected component. However, the total number of communications can be as much as $O(n^3)$. Our key observation is that if in each round we truncate the size of each neighbor set to be d, then the total number of communications is bounded by $O(nd^2)$. We choose $d = \sqrt{m/n}$ so that in each round the total communications are bounded by m. This new scheme will also finish in $O(\log \mathrm{diam}(G))$ rounds (each round can be implemented in $O(1)$ parallel time), and each neighbor set either contains exact d vertices, or it contains all the vertices in its connected components. We also include all edges in the original graph to the new graph so that the new graph has the same connectivity information with the original one. We present and analyze this truncated neighbor broadcasting scheme Section 5.2.

After the truncated neighbor broadcasting step, we have already obtained those small cliques whose sizes are at most d. The remaining vertices have degrees at least d. Let $p = \min\{\frac{\log d}{d}, 1/2\}$. We first choose each vertex to be a leader independently with probability p, and then contract each non-leader vertex to a neighbor who is a leader if it has at least one. With good probability the fraction of vertices that are not contracted to any leader is $O(p)$, so the size of the graph is reduced to $O(\frac{\log d}{d} n) = \frac{n}{(M/n)^{\Theta(1)}}$. We discuss random leader selection in more details in Section 5.3.

Moreover, as shown in Section 5.4, the contraction can be implemented in $O(1)$ parallel time.

Therefore, it takes $O(\log \mathrm{diam}(G))$ parallel time to reduce the problem size from n to $O(\frac{n}{(M/n)^{\Theta(1)}})$. Since the reduction follows the double exponential problem size reduction pattern, it takes $O(\log \log_{m/n} n)$ reduction steps to reduce to $O(1)$. Therefore, our parallel time complexity is $O(\log \mathrm{diam}(G) \cdot \log \log_{M/n} n)$. We describe our full graph connectivity algorithm in Section 5.5, and discuss its MPC implementation in Section 5.6.

5.2 Neighbor Increment Operation

In this section, we describe a procedure which can increase the number of neighbors of every vertex and preserve the connectivity at the same time. The input of the procedure is an undirected graph $G = (V, E)$ and a parameter M which is larger than $|V|$. The output is a graph $G' = (V, E')$ such that for each vertex v, either the connected component which contains v is a clique or v has at least $\lceil (M/|V|)^{1/2} \rceil - 1$ neighbors. Furthermore, $|E'| \leq |E| + M$. We use $\Gamma_G(v)$ to denote the neighbors of v in graph G, i.e. $\Gamma_G(v) = \{u \in V \mid (u, v) \in E\}$.

Lemma 5.2.1. *Let $G = (V, E)$ be an undirected graph, $M \in \mathbb{Z}_{\geq 0}$ which has $M \geq 4|V|$. Let $n = |V|$. Let r be the value at the end of the procedure* NeighborIncrement(M, G) *(Algorithm 13). Then $\forall i \in \{0, 1, \cdots, r\}, v \in V$, $S_v^{(i)}$ satisfies the following properties:*

1. $v \in S_v^{(i)}$.
2. $\forall u \in S_v^{(i)}, \mathrm{dist}_G(u, v) < \infty$.
3. $|S_v^{(i)}| < \lceil (M/n)^{1/2} \rceil \Rightarrow S_v^{(i)} = \{u \in V \mid \mathrm{dist}_G(u, v) \leq 2^i\}$.
4. $|S_v^{(i)}| \leq M/n$.

Proof. For property 1, we can prove it by induction. When $i = 0$, due to line 2, we know $v \in S_v^{(0)}$. Suppose property 1 holds for $S_v^{(i-1)}$ for all $v \in V$. If $S_v^{(i)}$ is updated by line 16, there are two cases: 1. if $u = v$, then $v \in S_u^{(i-1)}$, and the condition of line 16 does not hold, thus $v \in S_v^{(i)}$; 2. if $u \neq v$, then after implementing line 16, v will not be removed, thus $v \in S_v^{(i)}$. If $S_v^{(i)}$ is updated by line 19, then since $v \in S_v^{(i-1)}$, v is also in the set $S_v^{(i)}$. Thus, property 1 holds for every $S_v^{(i)}$.

For property 2, we can prove it by induction. When $i = 0$, it is easy to see $S_v^{(0)} \subseteq \Gamma_G(v) \cup \{v\}$, thus property 2 holds for it. Suppose property 2 holds for $S_v^{(i-1)}$ for all $v \in V$. If $S_v^{(i)}$ is updated by line 16, then since $u \in S_v^{(i-1)}$ and $S_v^{(i)} \subseteq S_u^{(i-1)} \cup \{v\}$, all the vertices from $S_v^{(i)}$ are in the same connected component as u and u is in the same connected component as v. Thus, property 2 holds in this case. If $S_v^{(i)}$ is updated by the line 19, then $\forall p \in S_v^{(i)}$, there exists $u \in S_v^{(i-1)}$ such that $p \in S_u^{(i-1)}$. We have p is in the same connected component as u, and u is in the same connected component as v. Thus, property 2 also holds in this case.

For property 3, we can prove it by induction. When $i = 0$, due to line 6, we have $|S_v^{(0)}| < \lceil (M/n)^{1/2} \rceil \to S_v^{(0)} = \Gamma_G(v) \cup \{v\} = \{u \in V \mid \mathrm{dist}_G(u, v) \leq 1\}$. Suppose property 3 holds for $S_v^{(i-1)}$ for all $v \in V$. Since if $|S_v^{(i)}| < \lceil (M/n)^{1/2} \rceil$, then $S_v^{(i)}$ can only be updated by line 19, and $\forall u \in S_v^{(i-1)}$, it has $|S_u^{(i-1)}| < \lceil (M/n)^{1/2} \rceil$. Thus, $S_v^{(i)} = \bigcup_{u \in S_v^{(i-1)}} S_u^{(i-1)} = \bigcup_{u \in V, \mathrm{dist}_G(u,v) \leq 2^{i-1}} \{p \in V \mid \mathrm{dist}_G(p, u) \leq 2^{i-1}\} = \{u \in V \mid \mathrm{dist}_G(u, v) \leq 2^i\}$. Thus, property 3 holds.

For property 4, we can prove it by induction. When $i = 0$, due to line 6, $\forall v \in V$, we have $|S_v^{(0)}| \leq \lceil (M/n)^{1/2} \rceil \leq M/n$, where the last inequality follows by $M/n \geq 4$. Now suppose property 4 holds for $S_v^{(i-1)}$ for all $v \in V$. If $S_v^{(i)}$ is updated by line 16, then $|S_v^{(i)}| = |S_u^{(i-1)}| \leq M/n$. If $S_v^{(i)}$ is updated by line 19, we know $\forall u \in S_v^{(i-1)}, |S_u^{(i-1)}| < \lceil (M/n)^{1/2} \rceil$. Notice that by property 1, $v \in S_v^{(i-1)}$, so $|S_v^{(i-1)}| < \lceil (M/n)^{1/2} \rceil$. Thus, $|S_v^{(i)}| = |\bigcup_{u \in S_v^{(i-1)}} S_u^{(i-1)}| \leq (M/n)^{1/2} \cdot (M/n)^{1/2} \leq M/n$. □

The following definition defines the number of iterations of Algorithm 13.

Definition 5.2.2. *Given an undirected graph $G = (V, E)$ and a parameter $M \in \mathbb{Z}_{\geq 0}, M \geq 4|V|$, the number of iterations of* NeighborIncrement(M, G) *(Algorithm 13) is the value of r at the end of the procedure.*

In the following lemma, we characterize the properties of Algorithm 13.

Lemma 5.2.3. *Let $G' = (V, E')$ be the output of* NeighborIncrement(M, G). *We have:*

1. *The number of iterations (Definition 5.2.2), $r \leq \min(\lceil \log(\mathrm{diam}(G)) \rceil, \lceil \log(M/n) \rceil) + 1$.*
2. *For all $u, v \in V$, $\mathrm{dist}_G(u, v) < \infty \Leftrightarrow \mathrm{dist}_{G'}(u, v) < \infty$.*

Algorithm 13 Neighbor Increment Operation

procedure NeighborIncrement($M, G = (V, E)$) ▷ Lemma 5.2.1, Lemma 5.2.3
▷ Output: $G' = (V, E')$
  Initially, $n = |V|, E' = \emptyset$ and let $S_v^{(0)} = \{v\}$ for all $v \in V$.
  for $v \in V$ **do** ▷ Initially, let $S_v^{(0)}$ be the set (or subset) of direct neighbors
    for $u \in \Gamma_G(v)$ **do**
      if $|S_v^{(0)}| < \lceil (M/n)^{1/2} \rceil$ **then**
        $S_v^{(0)} \leftarrow S_v^{(0)} \cup \{u\}$.
      end if
    end for
  end for
  $r \leftarrow 1$.
  for true do
    for $v \in V$ **do**
      if $\exists u \in S_v^{(r-1)}, |S_u^{(r-1)}| \geq \lceil (M/n)^{1/2} \rceil$ **then** ▷ neighbor u has many neighbors
        $S_v^{(r)} = S_u^{(r-1)} \cup \{v\}$.
        if $|S_v^{(r)}| > |S_u^{(r-1)}|$ **then**
          $S_v^{(r)} \leftarrow S_v^{(r)} \setminus \{u\}$.
        end if
      else
        $S_v^{(r)} = \bigcup_{u \in S_v^{(r-1)}} S_u^{(r-1)}$. ▷ neighbors of v's neighbors are v's new neighbors.
      end if
    end for
    ▷ $S_v^{(r)}$ is large or is a component
    if $\forall v \in V$, either $|S_v^{(r)}| \geq \lceil (M/n)^{1/2} \rceil$ or $|S_v^{(r)}| = |S_v^{(r-1)}|$ **then**
      Let $E' = E \cup \bigcup_{v \in V} \{(v, u) \mid v \in S_u^{(r)}$ or $u \in S_v^{(r)}, u \neq v\}$.
      return $G' = (V, E')$
    else
      $r \leftarrow r + 1$.
    end if
  end for
end procedure

3. $\forall v \in V$, *if* $|\Gamma_{G'}(v)| < \lceil (M/n)^{1/2} \rceil - 1$, *then the connected component in* G' *which contains* v *is a clique. It also implies that* $\forall u, v \in V$, *if* $|\Gamma_{G'}(v)| < \lceil (M/n)^{1/2} \rceil - 1$ *and* $|\Gamma_{G'}(u)| \geq \lceil (M/n)^{1/2} \rceil - 1$, *then* $\mathrm{dist}_{G'}(u, v) = \infty$.

4. $E \subseteq E', |E'| \leq |E| + M$.

Proof. For property 1, if $r > \lceil \log(\mathrm{diam}(G)) \rceil + 1$, then let $i = \lceil \log(\mathrm{diam}(G)) \rceil + 1$. Let $v \in V$.

By property 3 of Lemma 5.2.1, if $|S_v^{(i)}| < \lceil (M/n)^{1/2} \rceil$, then $S_v^{(i)} = \{u \in V \mid \text{dist}_G(u,v) \leq 2^i\} = \{u \in V \mid \text{dist}_G(u,v) \leq 2 \cdot 2^{\lceil \log(\text{diam}(G)) \rceil}\} = \{u \in V \mid \text{dist}_G(u,v) < \infty\}$. Furthermore, if $|S_v^{(i)}| < \lceil (M/n)^{1/2} \rceil$, then $|S_v^{(i-1)}| < \lceil (M/n)^{1/2} \rceil$, which means that $S_v^{(i-1)} = \{u \in V \mid \text{dist}_G(u,v) \leq 2^{i-1}\} = \{u \in V \mid \text{dist}_G(u,v) \leq 2^{\lceil \log(\text{diam}(G)) \rceil}\} = \{u \in V \mid \text{dist}_G(u,v) < \infty\} = S_v^{(i)}$. Then due to the condition in line 22, it will end the procedure in this round, which contradicts to $r > \lceil \log(\text{diam}(G)) \rceil + 1 = i$. Similarly, if $r > \lceil \log(M/n) \rceil + 1$, then let $i = \lceil \log(M/n) \rceil + 1$. Furthermore, if $|S_v^{(i)}| < \lceil (M/n)^{1/2} \rceil$, then $|S_v^{(i-1)}| < \lceil (M/n)^{1/2} \rceil$, which means that $S_v^{(i-1)} = \{u \in V \mid \text{dist}_G(u,v) \leq 2^{i-1}\} \neq \{u \in V \mid \text{dist}_G(u,v) \leq 2^i\} = S_v^{(i)}$. Thus, there exists $u \in S_v^{(i)}$ such that $|S_v^{(i)}| > \text{dist}_G(u,v) > 2^{i-1} > M/n \geq \lceil (M/n)^{1/2} \rceil$ which leads to a contradiction.

For property 2, if u, v are in the same connected component in G, then since $E \subseteq E'$, u, v are in the same connected component in G'. If u, v are in the same connected component in G', then there should be a path $u = u_1 \to u_2 \to \cdots \to u_p = v$ in G', i.e. $\forall j \in [p-1], (u_j, u_{j+1}) \in E'$. $(u_j, u_{j+1}) \in E'$ implies that either $(u_j, u_{j+1}) \in E$ or $u_j \in S_{u_{j+1}}^{(r)}$ or $u_{j+1} \in S_{u_j}^{(r)}$. By property 2 of Lemma 5.2.1, we know that $\forall j \in [p-1]$, u_j and u_{j+1} are in the same connected component in G. Thus, u and v are in the same connected component in G.

For property 3, due to line 23, if $|\Gamma_{G'}(v)| < \lceil (M/n)^{1/2} \rceil - 1$, then we have $|S_v^{(r)}| < \lceil (M/n)^{1/2} \rceil$. By property 3 of Lemma 5.2.1, and the condition in line 22, we know $\{u \in V \mid \text{dist}_G(u,v) \leq 2^r\} = S_v^{(r)} = S_v^{(r-1)} = \{u \in V \mid \text{dist}_G(u,v) \leq 2^{r-1}\}$. Thus, $S_v^{(r)} = \{u \in V \mid \text{dist}_G(u,v) < \infty\}$. Due to property 2, we have $\Gamma_{G'}(v) \cup \{v\} \subseteq \{u \in V \mid \text{dist}_G(u,v) < \infty\}$. Notice that $S_v^{(r)} \subseteq \Gamma_{G'}(v) \cup \{v\}$, thus, we have $\Gamma_{G'}(v) \cup \{v\} = \{u \in V \mid \text{dist}_G(u,v) < \infty\}$. Let $v' \in \{u \in V \mid \text{dist}_G(u,v) < \infty\}$, then due to property 2, $\Gamma_{G'}(v') \cup \{v'\} \subseteq \{u \in V \mid \text{dist}_G(u,v') < \infty\} = \{u \in V \mid \text{dist}_G(u,v) < \infty\}$, then we have $|\Gamma_{G'}(v') \cup \{v'\}| < \lceil (M/n)^{1/2} \rceil$. Thus, $|S_{v'}^{(r)}| < \lceil (M/n)^{1/2} \rceil$. By property 3 of Lemma 5.2.1, and the condition in line 22, we know $\{u \in V \mid \text{dist}_G(u,v') \leq 2^r\} = S_{v'}^{(r)} = S_{v'}^{(r-1)} = \{u \in V \mid \text{dist}_G(u,v') \leq 2^{r-1}\}$. Thus, $S_{v'}^{(r)} = \{u \in V \mid \text{dist}_G(u,v') < \infty\}$. Thus, $\Gamma_{G'}(v') \cup \{v'\} = \{u \in V \mid \text{dist}_G(u,v') < \infty\} = \{u \in V \mid \text{dist}_G(u,v) < \infty\}$. Thus, $\forall p, q \in \{u \in V \mid \text{dist}_G(u,v) < \infty\}$, we have $(p,q) \in E'$, which means that $\{u \in V \mid \text{dist}_G(u,v) < \infty\}$ is a clique in G'.

Now consider two vertices $u, v \in V$. Suppose $|\Gamma_{G'}(v)| < \lceil (M/n)^{1/2} \rceil - 1$, then we have that $\{p \in V \mid \text{dist}_G(p,v) < \infty\}$ is a clique in G'. Thus, $\forall q \in \{p \in V \mid \text{dist}_G(p,v) < \infty\}$, we have

$|\Gamma_{G'}(q)| = |\Gamma_{G'}(v)| < \lceil (M/n)^{1/2} \rceil - 1$. If $|\Gamma_{G'}(u)| \geq \lceil (M/n)^{1/2} \rceil - 1$, then $\text{dist}_{G'}(u, v) = \infty$.

For property 4, by line 23, we have $E \subseteq E'$ and $|E'| \leq |E| + \sum_{v \in V} |S_v^{(r)}| \leq |E| + n \cdot M/n = |E| + M$ where the last inequality follows by the property 4 of Lemma 5.2.1. □

5.3 Random Leader Selection

Given an undirected graph $G = (V, E)$, to design a connected component algorithm, a natural way is constantly contracting the vertices in the same component. One way to do the contraction is that we randomly choose some vertices as leaders, then contract non-leader vertices to the neighbor leader vertices.

In this section, we show that if $\forall v \in V$, the number of neighbors of v is large enough, then we can just sample a small number of leaders such that for each non-leader vertex $v \in V$, there is at least one neighbor of v which is chosen as a leader. A more generalized statement is stated in the following lemma.

Lemma 5.3.1. *Let V be a vertex set with n vertices. Let $0 < \gamma \leq n, \delta \in (0, 1)$. For each $v \in V$, let S_v be a subset of $V \setminus \{v\}$ with size at least $\gamma - 1$. Let $l : V \to \{0, 1\}$ be a random hash function such that $\forall v \in V, l(v)$ are i.i.d. Bernoulli random variables, i.e.*

$$l(v) = \begin{cases} 1 & \text{with probability } p; \\ 0 & \text{otherwise.} \end{cases}$$

If $p \geq \min((10 \log(2n/\delta))/\gamma, 1)$, then, with probability at least $1 - \delta$,

1. $\sum_{v \in V} l(v) \leq \frac{3}{2} pn$;
2. $\forall v \in V, \exists u \in S_v \cup \{v\}$ *such that* $l(u) = 1$.

Proof. For a fixed vertex $v \in V$, we have

$$\begin{aligned}
&\mathbb{P}\left(\sum_{u\in S_v\cup\{v\}}(\mathbb{E}(l(u))-l(u)) > \frac{1}{2}\sum_{u\in S_v\cup\{v\}}\mathbb{E}(l(u))\right)\\
\leq\ &\exp\left(-\frac{\frac{1}{2}\left(\frac{1}{2}\sum_{u\in S_v\cup\{v\}}\mathbb{E}(l(u))\right)^2}{\sum_{u\in S_v\cup\{v\}}\mathrm{Var}(l(u))+\frac{1}{3}\cdot 1\cdot\frac{1}{2}\sum_{u\in S_v\cup\{v\}}\mathbb{E}(l(u))}\right)\\
\leq\ &\exp\left(-\frac{\frac{1}{2}\left(\frac{1}{2}\sum_{u\in S_v\cup\{v\}}\mathbb{E}(l(u))\right)^2}{\sum_{u\in S_v\cup\{v\}}\mathbb{E}(l(u))+\frac{1}{3}\cdot 1\cdot\frac{1}{2}\sum_{u\in S_v\cup\{v\}}\mathbb{E}(l(u))}\right)\\
=\ &\exp\left(-\frac{3}{28}\cdot\sum_{u\in S_v\cup\{v\}}\mathbb{E}(l(u))\right)=\exp\left(-\frac{3}{28}\cdot p\cdot|S_v\cup\{v\}|\right)\leq\frac{\delta}{2n},
\end{aligned}$$

where the first inequality follows by Bernstein inequality and $|l(u)-E(l(u))|\leq 1$, the second inequality follows by $\mathrm{Var}(l(u))\leq\mathbb{E}(l^2(u))=\mathbb{E}(l(u))$. The last inequality follows by $|S_v\cup\{v\}|\geq\gamma$, and $p\geq\min((10\log(2n/\delta))/\gamma,1)$. Since $\frac{1}{2}\sum_{u\in S_v\cup\{v\}}\mathbb{E}(l(u))\geq 1$, with probability at least $1-\delta/(2n)$, $\sum_{u\in S_v\cup\{v\}}l(v)\geq 1$. By taking union bound over all S_v, with probability at least $1-\delta/2$, $\forall v\in V,\exists u\in S_v\cup\{v\},l(u)=1$.

Similarly, we have

$$\begin{aligned}
&\mathbb{P}\left(\sum_{u\in V}(l(u)-\mathbb{E}(l(u))) > \frac{1}{2}\sum_{u\in V}\mathbb{E}(l(u))\right)\\
\leq\ &\exp\left(-\frac{\frac{1}{2}\left(\frac{1}{2}\sum_{u\in V}\mathbb{E}(l(u))\right)^2}{\sum_{u\in V}\mathrm{Var}(l(u))+\frac{1}{3}\cdot 1\cdot\frac{1}{2}\sum_{u\in V}\mathbb{E}(l(u))}\right)\\
\leq\ &\exp\left(-\frac{\frac{1}{2}\left(\frac{1}{2}\sum_{u\in V}\mathbb{E}(l(u))\right)^2}{\sum_{u\in V}\mathbb{E}(l(u))+\frac{1}{3}\cdot 1\cdot\frac{1}{2}\sum_{u\in V}\mathbb{E}(l(u))}\right)\\
=\ &\exp\left(-\frac{3}{28}\cdot\sum_{u\in V}\mathbb{E}(l(u))\right)\\
=\ &\exp\left(-\frac{3}{28}\cdot p\cdot|V|\right)\leq\frac{\delta}{2n}\leq\frac{\delta}{2}.
\end{aligned}$$

Since $\sum_{u\in V}\mathbb{E}(l(u))=p\cdot n$, with probability at least $1-\delta/2$, $\sum_{u\in V}l(u)\leq 1.5pn$.

By taking union bound, with probability at least $1-\delta,\sum_{u\in V}l(u)\leq 1.5pn$ and $\forall v\in V,\exists u\in S_v\cup\{v\},l(u)=1$. □

If the number of neighbors of each vertex is not large, then we can still have a constant fraction

of vertices which can contract to a leader.

Lemma 5.3.2. *Let V be a vertex set with n vertices. Let S_v be a subset of $V \setminus \{v\}$ with size at least 1. Let $l : V \to \{0,1\}$ be a random hash function such that $\forall v \in V, l(v)$ are i.i.d. Bernoulli random variables, i.e.*

$$l(v) = \begin{cases} 1 & \text{with probability } \frac{1}{2}; \\ 0 & \text{otherwise.} \end{cases}$$

Let $L = \{v \in V \mid l(v) = 1\} \cup \{v \in V \mid \forall u \in S_v \cup \{v\}, l(u) = 0\}$. $\mathbb{E}(L) \leq 0.75n$.

Proof. For $v \in V$, $\mathbb{P}(l(v) = 1) = \frac{1}{2}$. Let $u \in S_v$. Then $\mathbb{P}(\forall x \in S_v \cup \{v\}, l(x) = 0) \leq \mathbb{P}(l(v) = 0, l(u) = 0) = 0.25$. $\mathbb{E}(|L|) = \sum_{v \in V} \mathbb{P}(v \in L) \leq 0.75n$. □

5.4 Tree Contraction Operation

In this section, we introduce the contraction operation. Firstly, let us introduce the concept of the parent pointers which can define a rooted forest.

Definition 5.4.1. *Given a set of vertices V, let $\mathrm{par} : V \to V$ satisfy that $\forall v \in V, \exists i > 0$ such that $\mathrm{par}^{(i)}(v) = \mathrm{par}^{(i+1)}(v)$, where $\forall v \in V, j > 0, \mathrm{par}^{(j)}(v)$ is defined as $\mathrm{par}(\mathrm{par}^{(j-1)}(v))$, and $\mathrm{par}^{(0)}(v) = v$. Then, we call such par a set of parent pointers on V. For $v \in V$, if $\mathrm{par}(v) = v$, then we say v is a root of par. par can have more than one root. The depth of $v \in V$, $\mathrm{dep}_{\mathrm{par}}(v)$ is the smallest $i \in \mathbb{Z}_{\geq 0}$ such that $\mathrm{par}^{(i)}(v) = \mathrm{par}^{(i+1)}(v)$. The root of $v \in V$, $\mathrm{par}^{(\infty)}(v)$ is defined as $\mathrm{par}^{(\mathrm{dep}_{\mathrm{par}}(v))}(v)$. The depth of par, $\mathrm{dep}(\mathrm{par})$ is defined as $\max_{v \in V} \mathrm{dep}_{\mathrm{par}}(v)$.*

It is easy to see that a set of parent pointers par on V formed a rooted forest on V. For a vertex $v \in V$, if $\mathrm{par}(v) = v$, then v is a root in the forest. Otherwise $\mathrm{par}(v)$ is the parent of v in the forest.

In the following, we define the union operation of several sets of parent pointers.

Definition 5.4.2. *Let $\mathrm{par}_1 : V_1 \to V_1, \mathrm{par}_2 : V_2 \to V_2, \cdots, \mathrm{par}_k : V_k \to V_k$ be k sets of parent pointers on vertex sets $V_1, V_2, \cdots, V_k$ respectively, where $\forall i \neq j \in [k], V_i \cap V_j = \emptyset$. Then $\mathrm{par} = \mathrm{par}_1 \cup \mathrm{par}_2 \cup \cdots \cup \mathrm{par}_k$ is a set of parent pointers on the vertex set $V_1 \cup V_2 \cup \cdots \cup V_k$ such that $\forall i \in [k], v \in V_i, \mathrm{par}(v) = \mathrm{par}_i(v)$.*

Now we focus on the parent pointers which can preserve the connectivity of the graph.

Definition 5.4.3. *Given a graph $G = (V, E)$ and a set of parent pointers* par *on V, if $\forall v \in V$, we have* $\mathrm{dist}_G(v, \mathrm{par}(v)) < \infty$, *then* par *is compatible with G.*

It is easy to show the following fact:

Fact 5.4.4. *Given a graph $G = (V, E)$ and a set of parent pointers* par *which is compatible with G, then $\forall u, v \in V$ with $\mathrm{par}^{(\infty)}(u) = \mathrm{par}^{(\infty)}(v)$, we have $\mathrm{dist}_G(u, v) < \infty$.*

Proof. By the definition of compatible, $\forall v \in V, \mathrm{dist}_G(v, \mathrm{par}(v)) < \infty$. By induction, $\forall l \in \mathbb{Z}_{>0}, v \in V$, we have $\mathrm{dist}_G(v, \mathrm{par}^{(l)}(v)) \leq \mathrm{dist}_G(v, \mathrm{par}^{(l-1)}(v)) + \mathrm{dist}_G(\mathrm{par}^{(l-1)}(v), \mathrm{par}^{(l)}(v)) < \infty$. Thus, for any pair of vertices $u, v \in V$, if $\mathrm{par}^{(\infty)}(u) = \mathrm{par}^{(\infty)}(v)$, then $\mathrm{dist}_G(u, v) \leq \mathrm{dist}_G(u, \mathrm{par}^{(\infty)}(u)) + \mathrm{dist}_G(\mathrm{par}^{(\infty)}(v), v) < \infty$. □

In this section, we describe a procedure which can be used to reduce the number of vertices. The input of the procedure is an undirected graph $G = (V, E)$ and a set of parent pointers $\mathrm{par} : V \to V$, where par is compatible with G. The output of the procedure will be the root of each vertex in V and an undirected graph $G' = (V', E')$ which satisfies $V' = \{v \in V \mid \mathrm{par}(v) = v\}, E' = \{(u, v) \in V' \times V' \mid u \neq v, \exists (p, q) \in E, \mathrm{par}^{(\infty)}(p) = u, \mathrm{par}^{(\infty)}(q) = v\}$. Notice that V' only contains all the roots in the forest induced by par, and $|E'| \leq |E|$.

Lemma 5.4.5. *Let $G = (V, E)$ be an undirected graph,* $\mathrm{par} : V \to V$ *be a set of parent pointers (See Definition 5.4.1). Then* TreeContraction(G, par) *(See Algorithm 14) will output $(G', g^{(r)})$ with $r \leq \lceil \log \mathrm{dep}(\mathrm{par}) \rceil$ satisfies the following properties:*

1. $\forall v \in V, g^{(r)}(v) = \mathrm{par}^{(\infty)}(v)$.
2. $V' = \{v \in V \mid \mathrm{par}(v) = v\}$.
3. $E' = \{(u, v) \in V' \times V' \mid u \neq v, \exists (p, q) \in E, \mathrm{par}^{(\infty)}(p) = u, \mathrm{par}^{(\infty)}(q) = v\}$.

Proof. One crucial observation is the following claim.

Claim 5.4.6. *$\forall l \in \{0, 1, \cdots, r\}, v \in V$, we have $g^{(l)}(v) = \mathrm{par}^{(2^l)}(v)$.*

Algorithm 14 Tree Contraction Operation

1: **procedure** TreeContraction($G = (V, E)$, par : $V \to V$) ▷ Lemma 5.4.5, Corollary 5.4.8
▷ Output: $G' = (V', E')$, $\text{par}^{(\infty)}(v)$ for all $v \in V$
2: Initially, for each $v \in V$ let $g^{(0)}(v) \leftarrow \text{par}(v)$. Let $V' = \emptyset, E' = \emptyset$.
3: $l \leftarrow 0$.
4: **for** $\exists v \in V, \text{par}(g^{(l)}(v)) \neq g^{(l)}(v)$ **do**
5: $l \leftarrow l + 1$.
6: For each $v \in V$, compute $g^{(l)}(v) = g^{(l-1)}(g^{(l-1)}(v))$. ▷ $g^{(l)}$ is $\text{par}^{(2^l)}$
7: **end for**
8: $r \leftarrow l$. ▷ r is the number of iterations, and is used in the analysis.
9: For $v \in V$, if $\text{par}(v) = v$, let $V' \leftarrow V' \cup \{v\}$.
10: For $(u, v) \in E$, if $g^{(r)}(u) \neq g^{(r)}(v)$, let $E' \leftarrow E' \cup \{(g^{(r)}(u), g^{(r)}(v))\}$.
▷ $\forall v \in V$, contract v to $\text{par}^{(\infty)}(v)$
11: **return** $g^{(r)}(v)$ as $\text{par}^{(\infty)}(v)$ for all $v \in V$, and $G' = (V', E')$
12: **end procedure**

Proof. The proof is by induction. When $l = 0$, $\forall v \in V, g^{(0)}(v) = \text{par}(v) = \text{par}^{(1)}(v)$, the claim is true. Suppose for $l - 1$, we have $\forall v \in V, g^{(l-1)}(v) = \text{par}^{(2^{l-1})}(v)$, then $\forall v \in V, g^{(l)}(v) = g^{(l-1)}(g^{(l-1)}(v)) = \text{par}^{(2^{l-1})}(\text{par}^{(2^{l-1})}(v)) = \text{par}^{(2^l)}(v)$. So the claim is true. □

If $r > \lceil \log \text{dep}(\text{par}) \rceil$, then $r - 1 \geq \lceil \log \text{dep}(\text{par}) \rceil$. Due to claim 5.4.6, we have $\forall v \in V, g^{(r-1)}(v) = \text{par}^{(2^{r-1})}(v) = \text{par}^{(\infty)}(v)$. Due to the condition in line 4, the loop will stop when $l \leq r - 1$ which leads to a contradiction to line 8. Thus, at the end of the algorithm, r should be at most $\lceil \log \text{dep}(\text{par}) \rceil$.

Since we have $\forall v \in V, \text{par}(g^{(r)}(v)) = g^{(r)}(v)$ at the end of the Algorithm 14, $\forall v \in V, g^{(r)}(v)$ must be $\text{par}^{(\infty)}(v)$. Then due to line 9 and line 10, we have $V' = \{v \in V \mid \text{par}(v) = v\}, E' = \{(u, v) \in V' \times V' \mid u \neq v, \exists (p, q) \in E, \text{par}^{(\infty)}(p) = u, \text{par}^{(\infty)}(q) = v\}$. □

Definition 5.4.7. *Let $G = (V, E)$ be an undirected graph, par : $V \to V$ be a set of parent pointers (See Definition 5.4.1). Then the number of iteration of* TreeContraction(G, par) *is defined as the value of r at the end of the procedure.*

Corollary 5.4.8 (Preserved connectivity and diameter)**.** *Let $G = (V, E)$ be an undirected graph, par : $V \to V$ be a set of parent pointers (See Definition 5.4.1) which is compatible (See Definition 5.4.3) with G. Then at the end of the Algorithm 14, $r \leq \lceil \log \text{dep}(\text{par}) \rceil$ and the output $(G', g^{(r)})$ will satisfy the following properties:*

1. $\mathrm{diam}(G') \leq \mathrm{diam}(G)$.

2. $\forall u, v \in V, \mathrm{dist}_G(u,v) < \infty \Rightarrow \mathrm{dist}_{G'}(\mathrm{par}^{(\infty)}(u), \mathrm{par}^{(\infty)}(v)) < \infty$.

3. $\forall u, v \in V, \mathrm{dist}_G(u,v) < \infty \Leftarrow \mathrm{dist}_{G'}(\mathrm{par}^{(\infty)}(u), \mathrm{par}^{(\infty)}(v)) < \infty$.

Proof. By Lemma 5.4.5, we have $r \leq \lceil \log \mathrm{dep}(\mathrm{par}) \rceil$, $V' = \{v \in V \mid \mathrm{par}(v) = v\}$ and $E' = \{(u,v) \in V' \times V' \mid u \neq v, \exists (p,q) \in E, \mathrm{par}^{(\infty)}(p) = u, \mathrm{par}^{(\infty)}(q) = v\}$.

For any two vertices $u, v \in V$ which are in the same connected component in G, then there should be a path $u = u_1 \to u_2 \to \cdots \to u_p = v$ in graph G. So $\forall i \in [p-1], (u_i, u_{i+1}) \in E$ which means that either $\mathrm{par}^{(\infty)}(u_i) = \mathrm{par}^{(\infty)}(u_{i+1})$ or $(\mathrm{par}^{(\infty)}(u_i), \mathrm{par}^{(\infty)}(u_{i+1})) \in E'$. Thus, $\mathrm{par}^{(\infty)}(u_1) \to \mathrm{par}^{(\infty)}(u_2) \to \cdots \to \mathrm{par}^{(\infty)}(u_p)$ is a valid path in G', and the length of this path in G' is at most p. Thus, the properties 1 and 2 are true.

For any two vertices $u, v \in V$ which are not in the same connected component in G, but there is a path $\mathrm{par}^{(\infty)}(u) = u'_1 \to u'_2 \to \cdots \to u'_p = \mathrm{par}^{(\infty)}(v)$ in G', then it means that there exists vertices $u_{1,1}, u_{1,2}, u_{2,1}, u_{2,2}, \cdots, u_{p,1}, u_{p,2} \in V$ which satisfies

(a) $\forall i \in [p-1], (u_{i,2}, u_{i+1,1}) \in E, \mathrm{par}^{(\infty)}(u_{i,2}) = u'_i, \mathrm{par}^{(\infty)}(u_{i+1,1}) = u'_{i+1}$.

(b) $u_{1,1} = u, u_{p,2} = v$.

(c) $\forall i \in [p], \mathrm{par}^{(\infty)}(u_{i,1}) = \mathrm{par}^{(\infty)}(u_{i,2})$. By Fact 5.4.4, we have $\mathrm{dist}_G(u_{i,1}, u_{i,2}) < \infty$.

Thus, there exists a path from u to v. This contradicts to that u, v are not in the same connected component. Therefore, property 3 is also true. □

5.5 Connectivity Algorithm

In this section, we described a batch algorithm for graph connectivity problem. The input is an undirected graph $G = (V, E)$, a space/rounds trade-off parameter M, and the rounds parameter r. The output is a function $\mathrm{col} : V \to V$ such that $\forall u, v \in V, \mathrm{dist}_G(u,v) < \infty \Leftrightarrow \mathrm{col}(u) = \mathrm{col}(v)$.

The algorithm is described in Algorithm 15. The following theorem shows the correctness of Algorithm 15.

Algorithm 15 Graph Connectivity

1: **procedure** CONNECTIVITY($G = (V, E), M, r$) ▷ Theorem 5.5.1, Theorem 5.5.8
2: Output: FAIL or col : $V \to V$.
3: $n \leftarrow |V|$
4: $\forall v \in V, h_0(v) \leftarrow$ null.
5: $G_0 = (V_0, E_0) = G$, i.e. $V_0 = V, E_0 = E$.
6: $n_0 = n$.
7: **for** $i = 1 \to r$ **do**
8: $\forall v \in V, h_i(v) \leftarrow$ null. ▷ $h_i(v)$ is the vertex that v contracts to
9: $G_i' = (V_i', E_i') = \text{NEIGHBORINCREMENT}(M, G_{i-1})$. ▷ Algorithm 13
10: Compute $V_i'' = \{v \in V_i' \mid |\Gamma_{G_i'}(v)| \geq \lceil (M/n_{i-1})^{1/2} \rceil - 1\}$.
11: Compute $E_i'' = \{(u, v) \in E_{i-1} \mid u \in V_i'', v \in V_i''\}$.
12: $G_i'' = (V_i'', E_i'')$. ▷ G_i'' is obtained by removing all the small components of G_i
13: Let $\gamma_i = \lceil (M/n_{i-1})^{1/2} \rceil$, $p_i = \min((30 \log(n) + 100)/\gamma_i, 1/2)$.
14: Let $l_i : V_i'' \to \{0, 1\}$ be a random hash function such that $\forall v \in V_i'', l_i(v)$ are i.i.d. Bernoulli random variables, and $\mathbb{P}(l_i(v) = 1) = p_i$.
15: Let $L_i = \{v \in V_i'' \mid l_i(v) = 1\} \cup \{v \in V_i'' \mid \forall u \in \Gamma_{G_i'}(v) \cup \{v\}, l_i(u) = 0\}$. ▷ L_i are leaders
16: $\forall v \in V_i''$ with $v \in L_i$, let $\text{par}_i(v) = v$.
17: $\forall v \in V_i''$ with $v \notin L_i$, let $\text{par}_i(v) = \min_{u \in L_i \cap (\Gamma_{G_i'}(v) \cup \{v\})} u$. ▷ Non-leader finds a leader.
18: Let $((V_i, E_i), g_i^{(r_i')}) = \text{TREECONTRACTION}(G_i'', \text{par}_i)$. ▷ Algorithm 14
19: $G_i = (V_i, E_i)$.
20: $n_i = |V_i|$.
21: For each $v \in V_i' \setminus V_i''$, let $h_i(v) \leftarrow \min_{u \in \Gamma_{G_i'}(v) \cup \{v\}} u$. ▷ Contract small component to one vertex
22: For each $v \in V_i'' \setminus V_i$, let $h_i(v) \leftarrow g_i^{(r_i')}(v)$. ▷ Contract non-leader to leader
23: For each $v \in V$, if $h_{i-1}(v) \neq$ null, then let $h_i(v) = h_{i-1}(v)$.
24: **end for**
25: If $n_r \neq 0$, return FAIL.
26: $((\widehat{V}, \widehat{E}), \text{col}) = \text{TREECONTRACTION}(G, h_r)$. ▷ Algorithm 14
27: **return** col .
28: **end procedure**

Theorem 5.5.1 (Correctness of Algorithm 15)**.** *Let $G = (V, E)$ be an undirected graph, $M \geq 4|V|$, and $r \leq |V|$ be the rounds parameter. If* CONNECTIVITY(G, M, r) *(Algorithm 15) does not output FAIL, then $\forall u, v \in V$, we have $\text{dist}_G(u, v) < \infty \Leftrightarrow \text{col}(u) = \text{col}(v)$.*

Firstly, we show that the input of Line 18 is valid.

Claim 5.5.2. *$\forall i \in [r]$, par_i is a set of parent pointers on V_i'', (See Definition 5.4.1) and is compatible (See Definition 5.4.3) with G_i''.*

Proof. $\forall v \in V_i''$, if $v \in L_i$, then $\text{par}_i(v) = v$. For $v \in V_i'' \setminus L_i$, due to property 3 of Lemma 5.2.3, we have $\text{par}_i(v) \in V_i''$. Since $\text{par}_i(v) \in L_i$, we have $\text{par}_i(\text{par}_i(v)) = \text{par}_i(v)$. Thus, $\text{par}_i : V_i'' \to V_i''$ is

a set of parent pointers on V_i''. Due to property 2 of Lemma 5.2.3 and $\mathrm{dist}_{G_i'}(\mathrm{par}_i(v), v) < \infty$, we know that $\mathrm{dist}_{G_{i-1}}(\mathrm{par}_i(v), v) < \infty$. Thus, $\mathrm{dist}_{G_i''}(\mathrm{par}_i(v), v) < \infty$. It implies that par_i is compatible with G_i''. □

The following claim shows that the number of the remaining vertices cannot increase after each round.

Claim 5.5.3. *If* Connectivity(G, M, r) *does not output FAIL, then* $\forall i \in [r], V_i \subseteq V_i'' \subseteq V_i' = V_{i-1}$.

Proof. Let $i \in [r]$. Due to Claim 5.5.2, the input of Line 18 is valid. Then, we can apply property 2 of Lemma 5.4.5 to get $V_i \subseteq V_i''$. By the construction of V_i'' we have $V_i'' \subseteq V_i'$. Since the procedure NeighborIncrement(M, G_{i-1}) (Algorithm 13) does not change the vertex set, we have $V_i' = V_{i-1}$. □

Now, we show that $\forall u, v \in V_i, \mathrm{dist}_{G_i}(u, v) < \infty \Leftrightarrow \mathrm{dist}_G(u, v) < \infty$.

Claim 5.5.4. *If* Connectivity(G, M, r) *does not output* FAIL, *then* $\forall i \in [r], \forall u, v \in V_i$, *we have* $\mathrm{dist}_{G_i}(u, v) < \infty \Leftrightarrow \mathrm{dist}_G(u, v) < \infty$.

Proof. The proof is by induction. Suppose $\forall u, v \in V_{i-1}, \mathrm{dist}_{G_{i-1}}(u, v) < \infty \Leftrightarrow \mathrm{dist}_G(u, v) < \infty$. $\forall w, z \in V_i$, according to Claim 5.5.3, $w, z \in V_i''$. By property 2,3 of Lemma 5.4.8, and property 2 of Lemma 5.4.5, $\mathrm{dist}_{G_i}(w, z) < \infty \Leftrightarrow \mathrm{dist}_{G_i''}(w, z) < \infty$. Due to property 2,3 of Lemma 5.2.3, there is no edge in E_{i-1} between V_i'' and $V_i' \setminus V_i''$. According to Claim 5.5.3, $w, z \in V_{i-1}$. Thus, $\mathrm{dist}_{G_i''}(w, z) < \infty \Leftrightarrow \mathrm{dist}_{G_{i-1}}(w, z) < \infty$. By induction hypothesis, we have $\forall w, z \in V_i, \mathrm{dist}_{G_i}(w, z) < \infty \Leftrightarrow \mathrm{dist}_G(w, z)$. □

The following claim states that once a vertex $v \in V$ is contracted to an another vertex, it will never be operated.

Claim 5.5.5. *Suppose* Connectivity(G, M, r) *does not output* FAIL. $\forall i \in \{0, 1, \cdots, r\}$, $v \in V$, *we have* $h_i(v) = \mathrm{null} \Leftrightarrow v \in V_i$. *Furthermore,* $\forall v \in V, \exists j \in [r]$ *such that* $h_0(v) = h_1(v) = \cdots = h_{j-1}(v) = \mathrm{null}$ *and* $h_j(v) = h_{j+1}(v) = \cdots = h_r(v) \neq \mathrm{null}, \mathrm{dist}_G(v, h_r(v)) < \infty$.

Proof. When $i = 0$, $\forall v \in V$, $h_0(v) = \text{null}, v \in V_0 = V$. Suppose it is true that $\forall v \in V, h_{i-1}(v) = \text{null} \Leftrightarrow v \in V_{i-1}$. If $v \notin V_i$, according to Claim 5.5.3, there are three cases: $v \in V_i'' \setminus V_i, v \in V_i' \setminus V_i'', v \notin V_{i-1}$. In the first case, due to Line 22, $h_i(v) \neq \text{null}$. In the second case, due to Line 21, $h_i(v) \neq \text{null}$, In the third case, due to Line 23, $h_i(v) \neq \text{null}$. If $h_i(v) = \text{null}$, then $h_i(v)$ cannot be updated by Line 21, Line 22 or Line 23 which implies that $v \in V_{i-1}, v \notin V_i' \setminus V_i'', v \notin V_i'' \setminus V_i$. Thus, $v \in V_i$.

Since the procedure does not FAIL, we have $n_r = 0$ which means that $\forall v \in V, h_r(v) \neq \text{null}$. Notice that by Line 23, if $h_{i-1}(v) \neq \text{null}$, then $h_i(v) = h_{i-1}(v)$. Thus, $\forall v \in V, \exists j \in [r]$ such that $h_0(v) = h_1(v) = \cdots = h_{j-1}(v) = \text{null}$ and $h_j(v) = h_{j+1}(v) = \cdots = h_r(v) \neq \text{null}$.

For $v \in V$, if $h_j(v) \neq \text{null}$ and $h_{j-1}(v) = \text{null}$, then $h_j(v)$ can only be updated by 21 or Line 22. In both cases, $\text{dist}_{G_{j-1}}(v, h_j(v)) < \infty$. By Claim 5.5.4, we have that $\text{dist}_G(v, h_j(v)) < \infty$. □

In the following, we show that h_r is a rooted tree such that $\text{dist}_G(u, v) < \infty \Leftrightarrow u, v$ have the same root. Due to Claim 5.5.5, if $\textsc{Connectivity}(G, M, r)$ does not output FAIL, then $n_r = 0$ which implies that $\forall v \in V, h_r(v) \neq \text{null}$. Thus, we can define $h_r^{(k)}(v)$ for $k \in \mathbb{Z}_{>0}$ as applying h_r on v k times. $\forall v \in V$, by Claim 5.5.5, let $j \in [r]$ satisfy that $h_j(v) \neq \text{null}$ and $h_{j-1}(v) = \text{null}$. If $h_j(v)$ is updated by Line 22, then $h_j(h_j(v)) = \text{null}$. If $h_j(v)$ is updated by Line 21, then $h_j(h_j(v)) = h_j(v)$. In both cases, h_j cannot create a cycle. Thus, we can define $h_r^{(\infty)}(v) = h_r^{(k)}(v)$ for some k which satisfies $h_r(h_r^{(k)}(v)) = h_r^{(k)}(v)$.

Claim 5.5.6. *Suppose* $\textsc{Connectivity}(G, M, r)$ *does not output* FAIL. *Then* $\forall u, v \in V$, *we have* $\text{dist}_G(u, v) < \infty \Leftrightarrow h_r^{(\infty)}(u) = h_r^{(\infty)}(v)$.

Proof. Let $u, v \in V$. By Claim 5.5.5, if $h_r^{\infty}(u) = h_r^{\infty}(v)$ we have $\text{dist}_G(u, v) < \infty$.

If $\text{dist}_G(u, v) < \infty$, then let $u' = h_r^{(\infty)}(u), v' = h_r^{(\infty)}(v)$. By Claim 5.5.5, $\text{dist}_G(u', v') \leq \text{dist}_G(u, u') + \text{dist}_G(u, v) + \text{dist}_G(v, v') < \infty$, and we can find $j \in [r]$ such that $h_j(u') \neq \text{null}, h_{j-1}(u') = \text{null}$. Without loss of generality, we can assume $h_{j-1}(v') = \text{null}$ (otherwise we can swap u' and v'). Due to Claim 5.5.5, $u', v' \in V_{i-1}$. Since $h_j(u') = h_r(u') = u'$, $h_j(u')$ can be only updated by Line 21, and $u' \in V_j' \setminus V_j''$. Then due to property 3 of Lemma 5.4.5, v' should be in $\Gamma_{G_i'}(u) \cup \{u\}$. Since $h_j(v') = h_r(v') = v'$, we can conclude that $u' = v'$.

If Connectivity(G, M, r) does not output FAIL, then in Line 26, col is exactly $h_r^{(\infty)}$. By Claim 5.5.6, we have $\forall u, v \in V, \mathrm{dist}_G(u, v) < \infty \Leftrightarrow \mathrm{col}(u) = \mathrm{col}(v)$.

□

Now let us consider the number of iterations of Algorithm 15 and the success probability.

Definition 5.5.7 (Total iterations)**.** *Let $G = (V, E)$ be an undirected graph, $\mathrm{poly}(n) \geq M > 4n$, and $r \leq n$ be the rounds parameter where n is the number of vertices in G. The total number of iterations of* Connectivity(G, M, r) *(Algorithm 15) is defined as $\sum_{i=1}^{r}(k_i + r'_i)$, where k_i denotes the number of iterations (See Definition 5.2.2) of* NeighborIncrement(M, G_{i-1}) *(see Line 9), and r'_i denotes the number of iterations (See Definition 5.4.7) of* TreeContraction(G''_i, par_i) *(see Line 18).*

Theorem 5.5.8 (Success probability and total iterations)**.** *Let $G = (V, E)$ be an undirected graph, $\mathrm{poly}(n) \geq M > 4n$, and $r \leq n$ be the rounds parameter where $n = |V|$. Let $c > 0$ be a sufficiently large constant. If $r \geq c \log\log_{M/n}(n)$, then with probability at least 0.98,* Connectivity(G, M, r) *(Algorithm 15) will not return* FAIL*. If* Connectivity(G, M, r) *succeeds, let k_i denote the number of iterations (See Definition 5.2.2) of* NeighborIncrement(M, G_{i-1}) *(see Line 9), and let r'_i denote the number of iterations of (See Definition 5.4.7) of* TreeContraction(G''_i, par_i) *(see Line 18), then*

1. *$\forall i \in [r]$, $r'_i = 0$.*
2. *$\forall i \in [r]$, k_i is at most $\lceil \log(\mathrm{diam}(G)) \rceil + 1$.*
3. *The number of iterations of Line 26 is at most $\lceil \log r \rceil$.*
4. *$\sum_{i=1}^{r} k_i \leq O(r \log(\mathrm{diam}(G)))$.*

Let $c_1 > 0$ be a sufficiently large constant. If $M \geq c_1 n \log^4 n$, then with probability at least 0.99, $\sum_{i=1}^{r} k_i \leq O(\log(\mathrm{diam}(G)) \log\log_{\mathrm{diam}(G)}(n))$. If $M < c_1 n \log^4 n$, then with probability at least 0.98, $\sum_{i=1}^{r} k_i \leq O(\log(\mathrm{diam}(G)) \log\log_{\mathrm{diam}(G)}(n) + (\log\log(n))^2)$.

Proof. Suppose Connectivity(G, M, r) succeeds. Property 1 follows by $\forall v \in V''_i, \mathrm{par}_i(\mathrm{par}_i(v)) = \mathrm{par}_i(v)$ and Lemma 5.4.5. Property 2 follows by $\mathrm{diam}(G_r) \leq \mathrm{diam}(G''_r) \leq \mathrm{diam}(G'_r) \leq \mathrm{diam}(G_{r-1}) \leq$

$\operatorname{diam}(G''_{r-1}) \le \operatorname{diam}(G'_{r-1}) \le \cdots \le \operatorname{diam}(G_0) = \operatorname{diam}(G)$ and Property 1 of Lemma 5.2.3. Property 3 follows by the depth of h_r is at most r and Lemma 5.4.5. Property 4 follows by property 2.

Now let us prove the success probability. Let $i \in [r]$. If $p_i < 0.5$, then we can apply Lemma 5.3.1 on vertex set V''_i, parameter γ_i, and hash function l_i. Notice that the set S_v in the statement of Lemma 5.3.1 is $\Gamma_{G'_i}(v)$ in the algorithm. Notice that $|V''_i| \le n$. Then in the i^{th} round, if $p_i < 0.5$, then with probability at most $1/(100n^2)$, L_i will be $\{v \in V''_i \mid l_i(v) = 1\}$, and $n_i = |L_i| \le 1.5 p_i n_{i-1}$. By taking union bound over all $i \in [r]$, we have that with probability at least 0.99, event $\mathcal{E}$ happens: for all $i \in [r]$, if $p_i < 0.5$, then $n_i \le 1.5 p_i n_{i-1} \le 0.75 n_{i-1}$. Suppose $\mathcal{E}$ happens. For $i \in [r]$, $p_i = 0.5$, if we apply Lemma 5.3.2, then condition on n_{i-1}, we have $\mathbb{E}(n_i) \le 0.75 n_{i-1}$. Thus, we know $\forall i \in [r], \mathbb{E}(n_i) \le 0.75\,\mathbb{E}(n_{i-1}) \le 0.75^i n$.

Next, we discuss the case for $p_0 = 0.5$ and the case for $p_0 < 0.5$ separately.

If $p_0 = 0.5$, then $M \le n \cdot (600 \log n)^4$. By Markov's inequality, when $i^* \ge 4 \log_{4/3}(6000 \log n)$, with probability at least 0.99, $n_{i^*} \le n/(600 \log n)^4$ and thus $p_{i^*} < 0.5$. Condition on this event and $\mathcal{E}$, we have

$$
\begin{aligned}
n_r &\le \frac{\left(\frac{\left(\frac{n_{i^*}^{1.5}}{m^{0.5}}(45\log n+150)\right)^{1.5}}{M^{0.5}}(45\log n+150)\right)^{\cdots}}{\cdots} && \text{(Apply } r' = r - i^* \text{ times)} \\
&= \frac{n_{i^*}^{1.5^{r'}}}{M^{1.5^{r'}-1}}(45\log n+150)^{2\cdot(1.5^{r'}-1)} \\
&= n_{i^*}/(M/n_{i^*})^{1.5^{r'}-1} \cdot (45\log n+150)^{2\cdot(1.5^{r'}-1)} \\
&\le n/\left(M/\left(n_{i^*}(45\log n+150)^2\right)\right)^{1.5^{r'}-1} \\
&\le n/\left(M/\left(n_{i^*}(45\log n+150)^2\right)\right)^{1.5^{r'/2}} \\
&\le n/\left(M/n\right)^{1.5^{r'/2}} \le \frac{1}{2},
\end{aligned}
$$

where the second inequality follows by $n_{i^*} \le n$, the third inequality follows by $r' \ge 5$, the forth inequality follows by $n_{i^*} \le n/(600 \log n)^4$, and the last inequality follows by $r' \ge \frac{2}{\log 1.5} \log\log_{M/n}(2n)$. Since $4n \le M \le n \cdot (600 \log n)^4$, $\log\log_{M/n} n = \Theta(\log\log n)$. Let $c > 0$ be a sufficiently large constant. Thus, when $r \ge c \log\log_{M/n} n \ge i^* + r' = 4\log(6000 \log n)/\log(4/3) + \frac{2}{\log 1.5} \log\log_{M/n}(2n)$, with probability at least 0.98, $\textsc{Connectivity}(G, M, r)$ will not fail.

Since property 1 of Lemma 5.2.3, we have $k_i \le O(\log(\min(M/n_{i-1}, \mathrm{diam}(G))))$. Thus,

$$\begin{aligned}
\sum_{i=1}^{r} k_i &= \sum_{i=1}^{i^*} k_i + \sum_{i=i^*+1}^{r} k_i \le O\left((\log\log n)^2\right) + \sum_{i=i^*+1}^{r} k_i \\
&\le O\left((\log\log n)^2\right) + \sum_{i:i\ge i^*+1, M/n_{i-1}\le \mathrm{diam}(G)} k_i + \sum_{i:i\le r, M/n_{i-1}>\mathrm{diam}(G)} k_i \\
&\le O\left((\log\log n)^2\right) + O\left(\sum_{i=0}^{\lceil \log_{1.25}\log_2(\mathrm{diam}(G))\rceil} \log(2^{1.25^i})\right) + O\left(\sum_{i=0}^{\lceil \log_{1.25}\log_{\mathrm{diam}(G)}(M)\rceil} \log(\mathrm{diam}(G))\right) \\
&\le O\left((\log\log n)^2\right) + O(\log(\mathrm{diam}(G))) + O(\log(\mathrm{diam}(G))\log\log_{\mathrm{diam}(G)}(n)) \\
&\le O(\log(\mathrm{diam}(G))\log\log_{\mathrm{diam}(G)}(n) + (\log\log(n))^2),
\end{aligned}$$

where the first inequality follows by $i^* = O(\log\log n)$ and $\forall i \le [i^*], M/n_{i-1} \le \mathrm{poly}(\log n)$, the third inequality follows by $M/n_{i+1} \ge (M/n_i)^{1.5}/(45\log n + 150) \ge (M/n_i)^{1.25}$.

If $M > n\cdot(600\log n)^4$, then $\forall i \in \{0\}\cup[r-1]$, we have $p_i < 0.5$. Since $\mathcal{E}$ happens, we have:

$$\begin{aligned}
n_r &\le \frac{\left(\frac{\left(\frac{n^{1.5}}{M^{0.5}}(45\log n+150)\right)^{1.5}}{M^{0.5}}(45\log n+150)\right)^{\cdots}}{\cdots} && \text{(Apply } r \text{ times)} \\
&= \frac{n^{1.5^r}}{M^{1.5^r-1}}(45\log n+150)^{2\cdot(1.5^r-1)} \\
&= n/(M/n)^{1.5^r-1}\cdot(45\log n+150)^{2\cdot(1.5^r-1)} \\
&= n/\left(M/\left(n(45\log n+150)^2\right)\right)^{1.5^r-1} \\
&\le n/\left(M/\left(n(45\log n+150)^2\right)\right)^{1.5^{r/2}} \\
&\le n/\left(M/\left(n(200\log n)^2\right)\right)^{1.5^{r/2}} \\
&\le \frac{1}{2},
\end{aligned}$$

where the second inequality follows by $r \ge 5$, the third inequality follows by $45\log n + 150 \le 200\log n$, and the last inequality follows by

$$r \ge c\log\log_{M/n} n \ge 2\log_{1.5}\log_{(M/n)^{1/2}} 2n \ge 2\log_{1.5}\log_{M/(n(200\log n)^2)} 2n$$

for a sufficiently large constant $c > 0$.

By property 1 of Lemma 5.2.3, we have $k_i \leq O(\log(\min(M/n_{i-1}, \mathrm{diam}(G))))$. Thus,

$$\begin{aligned}\sum_{i=1}^{r} k_i &\leq \sum_{M/n_{i-1} \leq \mathrm{diam}(G)} k_i + \sum_{M/n_{i-1} > \mathrm{diam}(G)} k_i \\ &\leq O\left(\sum_{i=0}^{\lceil \log_{1.25} \log_2(\mathrm{diam}(G)) \rceil} \log(2^{1.25^i})\right) + O\left(\sum_{i=0}^{\lceil \log_{1.25} \log_{\mathrm{diam}(G)}(M) \rceil} \log(\mathrm{diam}(G))\right) \\ &\leq O(\log(\mathrm{diam}(G))) + O(\log(\mathrm{diam}(G)) \log \log_{\mathrm{diam}(G)}(n)),\end{aligned}$$

where the first inequality follows by $M/n_{i+1} \geq (M/n_i)^{1.5}/(45 \log n + 150) \geq (M/n_i)^{1.25}$.

Since n_r is an integer, n_r must be 0 when $n_r \leq 1/2$. Let $c > 0$ be a sufficiently large constant. For all $M \geq 4n$, if $r \geq c \log \log_{M/n} n$ then Connectivity(G, M, r) will succeed with probability at least 0.98. □

5.6 MPC Implementation

In this section, we discuss how to implement our graph connectivity algorithm in the MPC model. Readers can refer to [ASS+18] for further implementation details. We will describe the MPC implementations of neighbor increment operation, random leader selection, and tree contraction operation respectively. Finally, we discuss how to implement Algorithm 15, our parallel graph connectivity algorithm.

It is important to know that sorting can be done in $O(\delta^{-1}) = O(1)$ parallel time in the (γ, δ)-MPC model [GSZ11, Goo99]. Using sorting as a primitive, we can also remove duplicates (for a list of $O(M)$ integers distributed in multiple processors) in $O(1)$ parallel time.

Neighbor increment operation. If $M/n > s$, then we need to store each $S_v^{(r)}$ in Algorithm 13 in multiple processors. For ease of exposition, in this section, we replace all the occurances of M/n in Algorithm 13 to be $\min\{M/n, s\}$. By doing so, we have an easier MPC implementation because each $S_v^{(r)}$ will fit into one processor; on the other hand, the new algorithm has the same asymptotic guarantee because if $M/n > s$ then $\log(M/n) = \Theta(\log s)$ (i.e. M/n and s are both $\mathrm{poly}(n)$). Let $\tau = \min\{\sqrt{M/n}, \sqrt{s}\}$ be the new threshold for the size of $S_v^{(r)}$ (i.e. after $|S_v^{(r)}| \geq \tau$, we stop updating $S_v^{(r)}$).

We are going to use the first $\frac{n}{s/\tau^2} = \frac{\tau^2 n}{s}$ processors to store $\{S_v^{(r)}\}_{v\in V}$. In particular, the first processor stores the first $\frac{s}{\tau^2}$ sets, the second processor stores the next $\frac{s}{\tau^2}$ sets, etc. We say that the first processor takes charge of $S_1^{(r)}, \ldots, S_{s/\tau^2}^{(r)}$, the second processor takes charge of $S_{s/\tau^2+1}^{(r)}, \ldots, S_{2s/\tau^2}^{(r)}$, etc. In the beginning, we can obtain $\{S_v^{(0)}\}_{v\in V}$ by first sorting E (the set of edges) where each edge $\{a, b\}$ appears twice as ordered pairs: once as (a, b) and once as (b, a). For each $v \in V$, all the neighbors of v will appear sequencially as $(v, x_1), \ldots, (v, x_k)$ in the sorted list; we can take the first τ neighbors and assign them to $S_v^{(0)}$, which can be implemented in $O(1)$ parallel time.

Now we describe how to compute $\{S_v^{(r+1)}\}_{v\in V}$ given $\{S_v^{(r)}\}_{v\in V}$. We fix v and let i denote the index of the processor which takes charge of v. If $|S_v^{(r)}| < \tau$, then process i will issue $|S_v^{(r)}|$ queries for v, namely, for every $u \in S_v^{(r)}$, processor i wants to get $S_u^{(r)}$ in order to compute $S_v^{(r+1)} = \cup_{u\in S_v^{(r)}} S_u^{(r)}$. The total number of queries issued by processor i is at most $\tau \cdot \frac{s}{\tau^2} = \frac{s}{\tau}$. Because each response to the query is at most τ words, the total input received by process i is bound by s words. However, when processor i wants to get $S_u^{(r)}$, it cannot directly ask the processor who takes charge of u (that processor holds $S_u^{(r)}$), because it might be possible that $S_u^{(r)}$ is on high demand so that all the responses of $S_u^{(r)}$ will exceed the output limit of one single processor. Fortunately, it is possible to implement a load-balancing scheme to respond to all the queries in $O(1)$ parallel time.

By a straightforward aggregation, it takes $O(\delta^{-1}) = O(1)$ parallel time to compute the Boolean value of $\forall v \in V, (|S_v^{(r+1)}| \geq \tau) \vee (|S_v^{(r+1)}| = |S_v^{(r)}|)$.

Random leader selection. Sampling each vertex independently can be done in $O(1)$ parallel time. Moreover, for each non-leader vertex v, we need to find a leader vertex in $S_v^{(R)}$ (where R refers to the last round of the neighbor increment operation). Similar to computing $S_v^{(0)}$, this can be done in $O(1)$ parallel time.

Tree contraction operation. Similar to the queries of $S_u^{(r)}$ in the neighbor increment operation, by a load-balancing scheme, we can implement each iteration of $g^{(l)}(v) = g^{(l-1)}(g^{(l-1)}(v))$ in $O(1)$ parallel time. We can compute the edges of the merged graph by computing $\{g^{(R)}(u), g^{(R)}(v)\}$ (R

is the last round of the iteration) for $\{u, v\}$ in the original graph and removing duplicates.

MPC **Implementation of Algorithm 15.** We store the tree edges for all the invocations of the tree contraction operation and then invoke tree contraction operation on the union of these tree edges. This can be performed efficiently in $O(\log R)$ rounds, where R is the number of reduction steps in the parallel connectivity algorithm.

References

[ABB+17] Sepehr Assadi, MohammadHossein Bateni, Aaron Bernstein, Vahab S. Mirrokni, and Cliff Stein. Coresets meet EDCS: algorithms for matching and vertex cover on massive graphs. In *arXiv preprint*. http://arxiv.org/pdf/1711.03076, 2017.

[AG18] Kook Jin Ahn and Sudipto Guha. Access to data and number of iterations: Dual primal algorithms for maximum matching under resource constraints. *ACM Transactions on Parallel Computing (TOPC)*, 4(4):17, 2018.

[AGM12a] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proceedings of the 23rd ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 459–467, 2012.

[AGM12b] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 5–14, 2012.

[AGM13] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Spectral sparsification in dynamic graph streams. In *Proceedings of the 16th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*, pages 1–10, 2013.

[AK17] Sepehr Assadi and Sanjeev Khanna. Randomized composable coresets for matching and vertex cover. In *SPAA*. https://arxiv.org/pdf/1705.08242, 2017.

[AKL17] Sepehr Assadi, Sanjeev Khanna, and Yang Li. On estimating maximum matching size in graph streams. In *Proceedings of the 28th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1723–1742, 2017.

[AKLY16] Sepehr Assadi, Sanjeev Khanna, Yang Li, and Grigory Yaroslavtsev. Maximum matchings in dynamic graph streams and the simultaneous communication model. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1345–1364, 2016.

[AKO11] Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. Streaming algorithms via precision sampling. In *Proceedings of the 52nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 363–372, 2011.

[AMS96] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *Computing (STOC)*, 1996.

[AMS99] Noga Alon, Yossi Matias, and Mario Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[ANOY14] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. Parallel algorithms for geometric graph problems. In *Proceedings of the Symposium on Theory of Computing (STOC)*. http://arxiv.org/pdf/1401.0042, 2014.

[ASS+18] Alexandr Andoni, Zhao Song, Clifford Stein, Zhengyu Wang, and Peilin Zhong. Parallel graph connectivity in log diameter rounds. In *FOCS*. https://arxiv.org/pdf/1805.03055, 2018.

[ASW18] Sepehr Assadi, Xiaorui Sun, and Omri Weinstein. Massively parallel algorithms for finding well-connected components in sparse graphs. In *ArXiv preprint*. https://arxiv.org/pdf/1805.02974, 2018.

[BCI+17] Vladimir Braverman, Stephen R Chestnut, Nikita Ivkin, Jelani Nelson, Zhengyu Wang, and David P Woodruff. Bptree: An ℓ_2 heavy hitters algorithm using constant memory. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 361–376. ACM, 2017.

[BCIS09] Radu Berinde, Graham Cormode, Piotr Indyk, and Martin J. Strauss. Space-optimal heavy hitters with strong error bounds. In *Proceedings of the Twenty-Eigth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 157–166, 2009.

[BCIW16] Vladimir Braverman, Stephen R. Chestnut, Nikita Ivkin, and David P. Woodruff. Beating countsketch for heavy hitters in insertion streams. In *Proceedings of the Symposium on Theory of Computing (STOC)*, pages 740–753, 2016.

[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):2, 2018.

[BH89] Paul Beame and Johan Håstad. Optimal bounds for decision problems on the CRCW PRAM. *J. ACM*, 36(3):643–670, 1989.

[BHNT15] Sayan Bhattacharya, Monika Henzinger, Danupon Nanongkai, and Charalampos E. Tsourakakis. Space- and time-efficient algorithm for maintaining dense subgraphs on one-pass dynamic streams. In *Proceedings of the 47^{th} Annual ACM on Symposium on Theory of Computing (STOC)*, pages 173–182, 2015.

[BKS13] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication steps for parallel query processing. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*, pages 273–284. ACM, 2013.

[BKSV14] Vladimir Braverman, Jonathan Katzman, Charles Seidell, and Gregory Vorsanger. An optimal algorithm for large frequency moments using o (nˆ(1-2/k)) bits. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques (APPROX/RANDOM 2014)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.

[BM81] Robert S. Boyer and J. Strother Moore. A fast majority vote algorithm. Technical Report Technical Report ICSCA-CMP-32, Institute for Computer Science, University of Texas, 1981.

[BM91] Robert S. Boyer and J. Strother Moore. MJRTY: A fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 105–118, 1991.

[BO13] Vladimir Braverman and Rafail Ostrovsky. Approximating large frequency moments with pick-and-drop sampling. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 42–57. Springer, 2013.

[BS15] Marc Bury and Chris Schwiegelshohn. Sublinear estimation of weighted matchings in dynamic data streams. In *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA)*, pages 263–274, 2015.

[CCE+16] Rajesh Chitnis, Graham Cormode, Hossein Esfandiari, MohammadTaghi Hajiaghayi, Andrew McGregor, Morteza Monemizadeh, and Sofya Vorotnikova. Kernelization via sampling with applications to finding matchings and related problems in dynamic graph streams. In *Proceedings of the 27th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1326–1344, 2016.

[CCF02] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. In *Automata, Languages and Programming*, pages 693–703. Springer, 2002.

[CCF04] Moses Charikar, Kevin C. Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312(1):3–15, 2004.

[CCHM15] Rajesh Hemant Chitnis, Graham Cormode, Mohammad Taghi Hajiaghayi, and Morteza Monemizadeh. Parameterized streaming: Maximal matching and vertex cover. In *Proceedings of the 26th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1234–1251, 2015.

[CDR86] Stephen Cook, Cynthia Dwork, and Rüdiger Reischuk. Upper and lower time bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15(1):87–97, 1986.

[CF14] Graham Cormode and Donatella Firmani. A unifying framework for ℓ_0-sampling algorithms. *Distributed and Parallel Databases*, 32(3):315–335, 2014. Preliminary version in ALENEX 2013.

[CK04] Don Coppersmith and Ravi Kumar. An improved data stream algorithm for frequency moments. In *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 151–156, 2004.

[CKS03] Amit Chakrabarti, Subhash Khot, and Xiaodong Sun. Near-optimal lower bounds on the multi-party communication complexity of set disjointness. In *Proceedings of the IEEE Conference on Computational Complexity*, pages 107–117, 2003.

[CLM+18] Artur Czumaj, Jakub Lacki, Aleksander Madry, Slobodan Mitrovic, Krzysztof Onak, and Piotr Sankowski. Round compression for parallel matching algorithms. In *Proceedings of the Symposium on Theory of Computing (STOC)*. https://arxiv.org/pdf/1707.03478, 2018.

[CM05] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, 2005.

[CMR05] Graham Cormode, S. Muthukrishnan, and Irina Rozenbaum. Summarizing and mining inverse distributions on data streams via dynamic inverse sampling. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 25–36, 2005.

[CW79] Larry Carter and Mark N. Wegman. Universal classes of hash functions. *J. Comput. Syst. Sci.*, 18(2):143–154, 1979.

[DG04] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[DG08] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DGIM02] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *SIAM journal on computing*, 31(6):1794–1813, 2002.

[DKR94] Martin Dietzfelbinger, Mirosław Kutyłowski, and Rüdiger Reischuk. Exact lower time bounds for computing boolean functions on crew prams. *Journal of Computer and System Sciences*, 48(2):231–254, 1994.

[DLM02] Erik D. Demaine, Alejandro López-Ortiz, and J. Ian Munro. Frequency estimation of Internet packet streams with limited space. In *Proceedings of the 10th Annual European Symposium on Algorithms (ESA)*, pages 348–360, 2002.

[DM16] Irit Dinur and Or Meir. Toward the KRW composition conjecture: Cubic formula lower bounds via communication complexity. In *Proceedings of the 31st Conference on Computational Complexity (CCC)*, pages 3:1–3:51, 2016.

[DORS08] Yevgeniy Dodis, Rafail Ostrovsky, Leonid Reyzin, and Adam D. Smith. Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. *SIAM J. Comput.*, 38(1):97–139, 2008.

[Dud67] Richard M. Dudley. The sizes of compact subsets of Hilbert space and continuity of Gaussian processes. *J. Functional Analysis*, 1:290–330, 1967.

[EGIN97] David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification – a technique for speeding up dynamic graph algorithms. *Journal of the ACM (JACM)*, 44(5):669–696, 1997.

[EHW16] Hossein Esfandiari, MohammadTaghi Hajiaghayi, and David P. Woodruff. Brief announcement: Applications of uniform sampling: Densest subgraph and beyond.

In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 397–399, 2016.

[EIRS91] Jack Edmonds, Russell Impagliazzo, Steven Rudich, and Jiri Sgall. Communication complexity towards lower bounds on circuit depth. In *Proceedings of the 32nd Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 249–257, 1991.

[FIS08] Gereon Frahling, Piotr Indyk, and Christian Sohler. Sampling in dynamic data streams and applications. *Int. J. Comput. Geometry Appl.*, 18(1/2):3–28, 2008. Preliminary version in SOCG 2005.

[FKM+05] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2-3):207–216, 2005.

[FM85] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[FMS+10] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Clifford Stein, and Zoya Svitkina. On distributing symmetric streaming computations. *ACM Transactions on Algorithms*, 6(4), 2010. Previously in SODA'08.

[Fre85] Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM Journal on Computing*, 14(4):781–798, 1985.

[FT16] Martin Farach-Colton and Meng-Tsung Tsai. Tight approximations of degeneracy in large graphs. In *Proceedings of the 12th Latin American Symposium on Theoretical Informatics (LATIN)*, pages 429–440, 2016.

[Gan15] Sumit Ganguly. Taylor polynomial estimator for estimating frequency moments. In *International Colloquium on Automata, Languages, and Programming*, pages 542–553. Springer, 2015.

[GK01] Michael Greenwald and Sanjeev Khanna. Space-efficient online computation of quantile summaries. *ACM SIGMOD Record*, 30(2):58–66, 2001.

[GKKT15] David Gibb, Bruce M. Kapron, Valerie King, and Nolan Thorn. Dynamic graph connectivity with improved worst case update time and sublinear space. *CoRR*, abs/1509.06464, 2015.

[GMT15] Sudipto Guha, Andrew McGregor, and David Tench. Vertex and hyperedge connectivity in dynamic graph streams. In *Proceedings of the 34th ACM Symposium on Principles of Database Systems (PODS)*, pages 241–247, 2015.

[GMWW14] Dmitry Gavinsky, Or Meir, Omri Weinstein, and Avi Wigderson. Toward better formula lower bounds: an information complexity approach to the KRW composition conjecture. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing (STOC)*, pages 213–222, 2014.

[Goo99] Michael T Goodrich. Communication-efficient parallel sorting. *SIAM Journal on Computing*, 29(2):416–432, 1999.

[GR09] Parikshit Gopalan and Jaikumar Radhakrishnan. Finding duplicates in a data stream. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 402–411, 2009.

[GSZ11] Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. Sorting, searching, and simulation in the mapreduce framework. In *ISAAC*, volume 7074, pages 374–383. Springer, 2011.

[Haa82] Uffe Haagerup. The best constants in the Khintchine inequality. *Studia Math.*, 70(3):231–283, 1982.

[HDT98] Jacob Holm, Kristian De Lichtenberg, and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. In *Proceedings of the Symposium on Theory of Computing (STOC)*, pages 79–89, 1998.

[HF98] Monika Rauch Henzinger and Michael L Fredman. Lower bounds for fully dynamic connectivity problems in graphs. *Algorithmica*, 22(3):351–362, 1998.

[HHKP17] Shang-En Huang, Dawei Huang, Tsvi Kopelowitz, and Seth Pettie. Fully dynamic connectivity in o (log n (log log n) 2) amortized expected time. In *Proceedings of the twenty-eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 510–520. Society for Industrial and Applied Mathematics, 2017.

[HHW18] Bernhard Haeupler, D. Ellis Hershkowitz, and David Wajc. Round- and message-optimal distributed graph algorithms. In *arXiv preprint*. http://arxiv.org/pdf/1801.05127, 2018.

[HK95] Monika Rauch Henzinger and Valerie King. Randomized dynamic graph algorithms with polylogarithmic time per operation. In *Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*, pages 519–527. Citeseer, 1995.

[HK99] Monika R. Henzinger and Valerie King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM (JACM)*, 46(4):502–516, 1999.

[HPP+15] James W. Hegeman, Gopal Pandurangan, Sriram V. Pemmaraju, Vivek B. Sardeshmukh, and Michele Scquizzato. Toward optimal bounds in the congested clique: Graph connectivity and MST. In *Proceedings of the 34th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 91–100, 2015.

[HT96] Monika R. Henzinger and Mikkel Thorup. *Improved sampling with applications to dynamic graph algorithms*. Springer, 1996.

[HW71] David Lee Hanson and Farroll Tim Wright. A bound on tail probabilities for quadratic forms in independent random variables. *The Annals of Mathematical Statistics*, 42(3):1079–1083, 1971.

[HW90] Johan Håstad and Avi Wigderson. Composition of the universal relation. In *Proceedings of a DIMACS Workshop on Advances In Computational Complexity Theory*, pages 119–134, 1990.

[IPW11] Piotr Indyk, Eric Price, and David P. Woodruff. On the power of adaptivity in sparse recovery. In *IEEE 52nd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 285–294, 2011.

[JST11] Hossein Jowhari, Mert Sağlam, and Gábor Tardos. Tight bounds for Lp samplers, finding duplicates in streams, and related problems. In *Proceedings of the 30th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 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.

[KKM13] Bruce M. Kapron, Valerie King, and Ben Mountjoy. Dynamic graph connectivity in polylogarithmic worst case time. In *Proceedings of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1131–1142, 2013.

[KKPT15] Kejlberg-Rasmussen, Casper, Tsvi Kopelowitz, Seth Pettie, and Mikkel Thorup. Faster worst case deterministic dynamic connectivity. *arXiv preprint arXiv:1507.05944*, 2015.

[KLL16] Zohar Karnin, Kevin Lang, and Edo Liberty. Optimal quantile approximation in streams. In *2016 IEEE 57th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 71–78. IEEE, 2016.

[KLM$^+$14] Michael Kapralov, Yin Tat Lee, Cameron Musco, Christopher Musco, and Aaron Sidford. Single pass spectral sparsification in dynamic streams. In *Proceedings of the 55th IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 561–570, 2014.

[KNP92] David R Karger, Noam Nisan, and Michal Parnas. Fast connected components algorithms for the erew pram. In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures*, pages 373–381. ACM, 1992.

[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.

[KNPW11] Daniel M Kane, Jelani Nelson, Ely Porat, and David P Woodruff. Fast moment estimation in data streams in optimal space. In *Proceedings of the forty-third annual ACM symposium on Theory of computing*, pages 745–754. ACM, 2011.

[KNW10] Daniel M. Kane, Jelani Nelson, and David P. Woodruff. An optimal algorithm for the distinct elements problem. In *Proceedings of the 29th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 41–52, 2010.

[Kon15] Christian Konrad. Maximum matching in turnstile streams. In *Proceedings of the 23rd Annual European Symposium on Algorithms (ESA)*, pages 840–852, 2015.

[KRW95] Mauricio Karchmer, Ran Raz, and Avi Wigderson. Super-logarithmic depth lower bounds via the direct sum in communication complexity. *Computational Complexity*, 5(3-4):191–204, 1995.

[KSP03] Richard M. Karp, Scott Shenker, and Christos H. Papadimitriou. A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.*, 28:51–55, 2003.

[KSV10] Howard J. Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for MapReduce. In *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 938–948, 2010.

[KW90] Mauricio Karchmer and Avi Wigderson. Monotone circuits for connectivity require super-logarithmic depth. *SIAM J. Discrete Math.*, 3(2):255–265, 1990.

[LMSV11] Silvio Lattanzi, Benjamin Moseley, Siddharth Suri, and Sergei Vassilvitskii. Filtering: a method for solving graph problems in MapReduce. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architectures*, pages 85–94. ACM, 2011.

[LNN15] Kasper Green Larsen, Jelani Nelson, and Huy L Nguyên. Time lower bounds for nonadaptive turnstile streaming algorithms. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 803–812. ACM, 2015.

[LNNT16] Kasper Green Larsen, Jelani Nelson, Huy L Nguyen, and Mikkel Thorup. Heavy hitters via cluster-preserving clustering. In *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on*, pages 61–70. IEEE, https://arxiv.org/pdf/1604.01357, 2016.

[LW13] Yi Li and David P Woodruff. A tight lower bound for high frequency moment estimation with small error. In *Proceedings of the International Workshop on Randomization and Computation (RANDOM)*, pages 623–638, 2013.

[MAE05] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. Efficient computation of frequent and top-k elements in data streams. In *Proceedings of the 10th International Conference on Database Theory (ICDT)*, pages 398–412, 2005.

[McG09] Andrew McGregor. Graph mining on streams. *Encyclopedia of Database Systems*, pages 1271–1275, 2009.

[McG14] Andrew McGregor. Graph stream algorithms: a survey. *SIGMOD Record*, 43(1):9–20, 2014.

[MG82] Jayadev Misra and David Gries. Finding repeated elements. *Sci. Comput. Program.*, 2(2):143–152, 1982.

[MM02] Gurmeet Singh Manku and Rajeev Motwani. Approximate frequency counts over data streams. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 346–357. Elsevier, 2002.

[Mor78] Robert Morris. Counting large numbers of events in small registers. *Communications of the ACM*, 21(10):840–842, 1978.

[MP80] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. *TCS*, 12, 1980.

[MTVV15] Andrew McGregor, David Tench, Sofya Vorotnikova, and Hoa T. Vu. Densest subgraph in dynamic graph streams. In *Proceedings of the 40^{th} International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 472–482, 2015.

[Mut05] S. Muthukrishnan. Data Streams: Algorithms and Applications. *Foundations and Trends in Theoretical Computer Science*, 1(2):117–236, 2005.

[MW10] Morteza Monemizadeh and David P. Woodruff. 1-pass relative-error l_p-sampling with applications. In *Proceedings of the 21^{st} Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1143–1160, 2010.

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

[NY19] Jelani Nelson and Huacheng Yu. Optimal lower bounds for distributed and streaming spanning forest computation. In *Proceedings of the Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1844–1860. SIAM, 2019.

[Pag13] Rasmus Pagh. Compressed matrix multiplication. *ACM Transactions on Computation Theory (TOCT)*, 5(3):9, 2013.

[PD04] Mihai Pătraşcu and Erik D Demaine. Lower bounds for dynamic connectivity. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 546–553. ACM, 2004.

[PRS16] Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. Fast distributed algorithms for connectivity and mst in large graphs. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 429–438. ACM, 2016.

[RMCS13] Vibhor Rastogi, Ashwin Machanavajjhala, Laukik Chitnis, and Anish Das Sarma. Finding connected components in map-reduce in logarithmic rounds. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 50–61. IEEE, 2013.

[SBAS04] Nisheeth Shrivastava, Chiranjeeb Buragohain, Divyakant Agrawal, and Subhash Suri. Medians and beyond: new aggregation techniques for sensor networks. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 239–249. ACM, 2004.

[ST83] Daniel D Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3):362–391, 1983.

[ST85] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM (JACM)*, 32(3):652–686, 1985.

[SV82] Yossi Shiloach and Uzi Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algorithms*, 3(1):57–67, 1982.

[Tar07] Jun Tarui. Finding a duplicate and a missing item in a stream. In *Proceedings of the 4th Annual Conference on Theory and Applications of Models of Computation (TAMC)*, pages 128–135, 2007.

[Tho00] Mikkel Thorup. Near-optimal fully-dynamic graph connectivity. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 343–350. ACM, 2000.

[TZ97] Gábor Tardos and Uri Zwick. The communication complexity of the universal relation. In *Proceedings of the 12th Annual IEEE Conference on Computational Complexity (CCC)*, pages 247–259, 1997.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[Wan15] Zhengyu Wang. An improved randomized data structure for dynamic graph connectivity. *CoRR*, abs/1510.04590, 2015.

[Whi12] Tom White. *Hadoop: the definitive guide*. O'Reilly, 2012.

[Wul13] Wulff-Nilsen, Christian. Faster deterministic fully-dynamic graph connectivity. In *Proceedings of the twenty-fourth annual ACM-SIAM symposium on Discrete algorithms*, pages 1757–1769. Society for Industrial and Applied Mathematics, 2013.

[Wyl79] James C Wyllie. The complexity of parallel computations. Technical report, Cornell University, 1979.

[ZCF$^+$10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.