Data Stream Algorithms

Lecture Notes

Amit Chakrabarti
Dartmouth College

Latest Update: July 2, 2020

Preface

This book grew out of lecture notes for offerings of a course on data stream algorithms at Dartmouth, beginning with a first offering in Fall 2009. It's primary goal is to be a resource for students and other teachers of this material. The choice of topics reflects this: the focus is on foundational algorithms for space-efficient processing of massive data streams. I have emphasized algorithms that are simple enough to permit a clean and fairly complete presentation in a classroom, assuming not much more background than an advanced undergraduate student would have. Where appropriate, I have provided pointers to more advanced research results that achieve improved bounds using heavier technical machinery.

I would like to thank the many Dartmouth students who took various editions of my course, as well as researchers around the world who told me that they had found my course notes useful. Their feedback inspired me to make a book out of this material. I thank Suman Kalyan Bera, Sagar Kale, and Jelani Nelson for numerous comments and contributions. Of special note are the 2009 students whose scribing efforts got this book started: Radhika Bhasin, Andrew Cherne, Robin Chhetri, Joe Cooley, Jon Denning, Alina Djamankulova, Ryan Kingston, Ranganath Kondapally, Adrian Kostrubiak, Konstantin Kutzkow, Aarathi Prasad, Priya Natarajan, and Zhenghui Wang.

Amit Chakrabarti
April 2020

Contents

Unit 0

Preliminaries: The Data Stream Model

0.1 The Basic Setup

In this course, we are concerned with algorithms that compute some function of a massively long input stream σ. In the most basic model (which we shall call the *vanilla streaming model*), this is formalized as a sequence $\sigma = \langle a_1, a_2, \ldots, a_m \rangle$, where the elements of the sequence (called *tokens*) are drawn from the universe $[n] := \{1, 2, \ldots, n\}$. Note the two important size parameters: the stream length, m, and the universe size, n. If you read the literature in the area, you will notice that some authors interchange these two symbols. In this course, we shall consistently use m and n as we have just defined them.

Our central goal will be to process the input stream using a small amount of *space* s, i.e., to use s bits of random-access working memory. Since m and n are to be thought of as "huge," we want to make s much smaller than these; specifically, we want s to be *sublinear* in both m and n. In symbols, we want

$$s = o(\min\{m, n\}) . \tag{1}$$

The holy grail is to achieve

$$s = O(\log m + \log n) , \tag{2}$$

because this amount of space is what we need to store a constant number of tokens from the stream and a constant number of counters that can count up to the length of the stream. Sometimes we can only come close and achieve a space bound of the form $s = \mathrm{polylog}(\min\{m, n\})$, where $f(n) = \mathrm{polylog}(g(n))$ means that there exists a constant $c > 0$ such that $f(n) = O((\log g(n))^c)$.

The reason for calling the input a stream is that we are only allowed to access the input in "streaming fashion." That is, we do not have random access to the tokens and we can only scan the sequence in the given order. We *do* consider algorithms that make p *passes* over the stream, for some "small" integer p, keeping in mind that the holy grail is to achieve $p = 1$. As we shall see, in our first few algorithms, we will be able to do quite a bit in just one pass.

0.2 The Quality of an Algorithm's Answer

The function we wish to compute—$\phi(\sigma)$, say—is often real-valued. We shall typically seek to compute only an *estimate* or *approximation* of the true value of $\phi(\sigma)$, because many basic functions can provably not be computed exactly using sublinear space. For the same reason, we shall often allow *randomized* algorithms than may err with some small, but controllable, probability. This motivates the following basic definition.

Definition 0.2.1. Let $\mathscr{A}(\sigma)$ denote the output of a randomized streaming algorithm $\mathscr{A}$ on input σ; note that this is a random variable. Let ϕ be the function that $\mathscr{A}$ is supposed to compute. We say that the algorithm (ε, δ)-estimates ϕ if we have

$$\Pr\left[\left|\frac{\mathscr{A}(\sigma)}{\phi(\sigma)} - 1\right| > \varepsilon\right] \le \delta\,.$$

Notice that the above definition insists on a multiplicative approximation. This is sometimes too strong a condition when the value of $\phi(\sigma)$ can be close to, or equal to, zero. Therefore, for some problems, we might instead seek an additive approximation, as defined below.

Definition 0.2.2. In the above setup, the algorithm $\mathscr{A}$ is said to $(\varepsilon,\delta)^+$-estimate ϕ if we have

$$\Pr\left[|\mathscr{A}(\sigma) - \phi(\sigma)| > \varepsilon\right] \le \delta\,.$$

We have mentioned that certain things are *provably* impossible in sublinear space. Later in the course, we shall study how to prove such impossibility results. Such impossibility results, also called *lower bounds*, are a rich field of study in their own right.

0.3 Variations of the Basic Setup

Quite often, the function we are interested in computing is some statistical property of the *multiset* of items in the input stream σ. This multiset can be represented by a frequency vector $\boldsymbol{f} = (f_1, f_2, \ldots, f_n)$, where

$$f_j \;=\; |\{i : a_i = j\}| \;=\; \text{number of occurrences of } j \text{ in } \sigma\,.$$

In other words, σ implicitly defines this vector $\boldsymbol{f}$, and we are then interested in computing some function of the form $\Phi(\boldsymbol{f})$. While processing the stream, when we scan a token $j \in [n]$, the effect is to increment the frequency f_j. Thus, σ can be thought of as a sequence of *update instructions*, updating the vector $\boldsymbol{f}$.

With this in mind, it is interesting to consider more general updates to $\boldsymbol{f}$: for instance, what if items could both "arrive" and "depart" from our multiset, i.e., if the frequencies f_j could be both incremented *and* decremented, and by variable amounts? This leads us to the *turnstile model*, in which the tokens in σ belong to $[n] \times \{-L, \ldots, L\}$, interpreted as follows:

$$\text{Upon receiving token } a_i = (j,c)\,, \text{ update } f_j \leftarrow f_j + c\,.$$

Naturally, the vector $\boldsymbol{f}$ is assumed to start out at $\mathbf{0}$. In this generalized model, it is natural to change the role of the parameter m: instead of the stream's length, it will denote the maximum number of items in the multiset at any point of time. More formally, we require that, at all times, we have

$$\|\boldsymbol{f}\|_1 = |f_1| + \cdots + |f_n| \le m\,.$$

A special case of the turnstile model, that is sometimes important to consider, is the *strict turnstile model*, in which we assume that $\boldsymbol{f} \ge 0$ at all times. A further special case is the *cash register model*, where we only allow positive updates: i.e., we require that every update (j,c) have $c > 0$.

Unit 1

Finding Frequent Items Deterministically

Our study of data stream algorithms begins with *statistical* problems: the input stream describes a multiset of items and we would like to be able to estimate certain statistics of the empirical frequency distribution (see Section 0.3). We shall study a number of such problems over the next several units. In this unit, we consider one such problem that admits a particularly simple and deterministic (yet subtle and powerful) algorithm.

1.1 The Problem

We are in the vanilla streaming model. We have a stream $\sigma = \langle a_1, \ldots, a_n \rangle$, with each $a_i \in [n]$, and this implicitly defines a frequency vector $\boldsymbol{f} = (f_1, \ldots, f_n)$. Note that $f_1 + \cdots + f_n = m$.

In the MAJORITY problem, our task is as follows. If $\exists\, j : f_j > m/2$, then output j, otherwise, output "$\perp$".

This can be generalized to the FREQUENT problem, with parameter k, as follows: output the set $\{j : f_j > m/k\}$.

In this unit, we shall limit ourselves to deterministic algorithms for this problem. If we further limit ourselves to one-pass algorithms, even the simpler problem, MAJORITY, provably requires $\Omega(\min\{m,n\})$ space. However, we shall soon give a one-pass algorithm—the Misra–Gries Algorithm [MG82]—that solves the related problem of estimating the frequencies f_j. As we shall see,

1. the properties of Misra–Gries are interesting in and of themselves, and
2. it is easy to extend Misra–Gries, using a second pass, to then solve the FREQUENT problem.

Thus, we now turn to the FREQUENCY-ESTIMATION problem. The task is to process σ to produce a data structure that can provide an estimate $\hat{f}_a$ for the frequency f_a of a given token $a \in [n]$. Note that the query a is given to us only *after* we have processed σ. This problem can also be thought of as querying the frequency vector $\boldsymbol{f}$ at the point a, so it is also known as the POINT-QUERY problem.

1.2 Frequency Estimation: The Misra–Gries Algorithm

As with all one-pass data stream algorithms, we shall have an *initialization* section, executed before we see the stream, a *processing* section, executed each time we see a token, and an *output* section, where we answer question(s) about the stream, perhaps in response to a given *query*.

This algorithm (see Algorithm 1) uses a parameter k that controls the quality of the answers it gives. (Looking ahead: to solve the FREQUENT problem with parameter k, we shall run the Misra–Gries algorithm with parameter k.) It maintains an associative array, A, whose keys are tokens seen in the stream, and whose values are counters associated with these tokens. We keep at most $k - 1$ counters at any time.

Algorithm 1 The Misra–Gries frequency estimation algorithm

```
Initialize:
 1: $A \leftarrow$ (empty associative array)

Process (token $j$):
 2: if $j \in \text{keys}(A)$ then
 3:     $A[j] \leftarrow A[j] + 1$
 4: else if $|\text{keys}(A)| < k - 1$ then
 5:     $A[j] \leftarrow 1$
 6: else
 7:     foreach $\ell \in \text{keys}(A)$ do
 8:         $A[\ell] \leftarrow A[\ell] - 1$
 9:         if $A[\ell] = 0$ then remove $\ell$ from $A$

Output (query $a$):
10: if $a \in \text{keys}(A)$ then report $\hat{f}_a = A[a]$ else report $\hat{f}_a = 0$
```

1.3 Analysis of the Algorithm

To process each token quickly, we could maintain the associative array A using a balanced binary search tree. Each key requires $\lceil \log n \rceil$ bits to store and each value requires at most $\lceil \log m \rceil$ bits. Since there are at most $k-1$ key/value pairs in A at any time, the total space required is $O(k(\log m + \log n))$.

We now analyze the quality of the algorithm's output.

Pretend that A consists of n key/value pairs, with $A[j] = 0$ whenever j is not actually stored in A by the algorithm. Consider the increments and decrements to $A[j]$s as the stream comes in. For bookkeeping, pretend that upon entering the loop at line 7, $A[j]$ is incremented from 0 to 1, and then immediately decremented back to 0. Further, noting that each counter $A[j]$ corresponds to a set of occurrences of j in the input stream, consider a variant of the algorithm (see Algorithm 2) that explicitly maintains this set. Of course, actually doing so is horribly space-inefficient!

Algorithm 2 Thought-experiment algorithm for analysis of Misra–Gries

```
Initialize:
 1: $B \leftarrow$ (empty associative array)

Process (stream-position $i$, token $j$):
 2: if $j \in \text{keys}(B)$ then
 3:     $B[j] \leftarrow B[j] \cup \{i\}$
 4: else
 5:     $B[j] \leftarrow \{i\}$
 6: if $|\text{keys}(B)| = k$ then
 7:     foreach $\ell \in \text{keys}(B)$ do
 8:         $B[\ell] \leftarrow B[\ell] \setminus \min(B[\ell])$        ▷ forget earliest occurrence of $\ell$
 9:         if $B[\ell] = \varnothing$ then remove $\ell$ from $B$

Output (query $a$):
10: if $a \in \text{keys}(B)$ then report $\hat{f}_a = |B[a]|$ else report $\hat{f}_a = 0$
```

Notice that after each token is processed, each $A[j]$ is precisely the cardinality of the corresponding set $B[j]$. The increments to $A[j]$ (including the aforementioned pretend ones) correspond precisely to the occurrences of j in the stream. Thus, $\hat{f}_j \leq f_j$.

On the other hand, decrements to counters (including the pretend ones) occur in sets of k. To be precise, referring to Algorithm 2, whenever the sets $B[\ell]$ are shrunk, exactly k of the sets shrink by one element each and the removed elements are k distinct stream positions. Focusing on a particular item j, each decrement of $A[j]$— is "witnessed" by a collection of k distinct stream positions. Since the stream length is m, there can be at most m/k such decrements. Therefore, $\hat{f}_j \geq f_j - m/k$. Putting these together we have the following theorem.

Theorem 1.3.1. *The Misra–Gries algorithm with parameter k uses one pass and $O(k(\log m + \log n))$ bits of space, and provides, for any token j, an estimate $\hat{f}_j$ satisfying*

$$f_j - \frac{m}{k} \leq \hat{f}_j \leq f_j\,.$$

1.4 Finding the Frequent Items

Using the Misra–Gries algorithm, we can now easily solve the FREQUENT problem in one additional pass. By the above theorem, if some token j has $f_j > m/k$, then its corresponding counter $A[j]$ will be positive at the end of the Misra–Gries pass over the stream, i.e., j will be in $\text{keys}(A)$. Thus, we can make a second pass over the input stream, counting exactly the frequencies f_j for all $j \in \text{keys}(A)$, and then output the desired set of items.

Alternatively, if limited to a single pass, we can solve FREQUENT in an approximate sense: we may end up outputting items that are below (but not too far below) the frequency threshold. We explore this in the exercises.

Exercises

1-1 Let $\hat{m}$ be the sum of all counters maintained by the Misra–Gries algorithm after it has processed an input stream, i.e., $\hat{m} = \sum_{\ell \in \text{keys}(A)} A[\ell]$. Prove that the bound in Theorem 1.3.1 can be sharpened to

$$f_j - \frac{m - \hat{m}}{k} \leq \hat{f}_j \leq f_j\,. \tag{1.1}$$

1-2 Items that occur with high frequency in a dataset are sometimes called *heavy hitters*. Accordingly, let us defined the HEAVY-HITTERS problem, with real parameter $\varepsilon > 0$, as follows. The input is a stream σ. Let $m, n, \boldsymbol{f}$ have their usual meanings. Let

$$\text{HH}_\varepsilon(\sigma) = \{j \in [n] : f_j \geq \varepsilon m\}$$

be the set of ε-heavy hitters in σ. Modify Misra–Gries to obtain a one-pass streaming algorithm that outputs this set "approximately" in the following sense: the set H it outputs should satisfy

$$\text{HH}_\varepsilon(\sigma) \subseteq H \subseteq \text{HH}_{\varepsilon/2}(\sigma)\,.$$

Your algorithm should use $O(\varepsilon^{-1}(\log m + \log n))$ bits of space.

1-3 Suppose we have run the (one-pass) Misra–Gries algorithm on two streams σ_1 and σ_2, thereby obtaining a summary for each stream consisting of k counters. Consider the following algorithm for merging these two summaries to produce a single k-counter summary.

```
1:  Combine the two sets of counters, adding up counts for any common items.
2:  If more than $k$ counters remain:
    2.1:  $c \leftarrow$ value of $(k+1)$th counter, based on decreasing order of value.
    2.2:  Reduce each counter by $c$ and delete all keys with non-positive counters.
```

Prove that the resulting summary is good for the combined stream $\sigma_1 \circ \sigma_2$ (here "$\circ$" denotes concatenation of streams) in the sense that frequency estimates obtained from it satisfy the bounds given in Eq. (1.1).

Unit 2

Estimating the Number of Distinct Elements

We continue with our study of data streaming algorithms for statistical problems. Previously, we focused on identifying items that are particularly dominant in a stream, appearing with especially high frequency. Intuitively, we could solve this in sublinear space because only a few items can be dominant and we can afford to throw away information about non-dominant items. Now, we consider a very different statistic: namely, *how many* distinct tokens (elements) appear in the stream. This is a measure of how "spread out" the stream is. It is not intuitively clear that we *can* estimate this quantity well in sublinear space, because we can't afford to ignore rare items. In particular, merely sampling some tokens from the stream will mislead us, since a sample will tend to pick up frequent items rather than rare ones.

2.1 The Problem

As in Unit 1, we are in the vanilla streaming model. We have a stream $\sigma = \langle a_1, \ldots, a_n \rangle$, with each $a_i \in [n]$, and this implicitly defines a frequency vector $\boldsymbol{f} = (f_1, \ldots, f_n)$. Let $d = |\{j : f_j > 0\}|$ be the number of distinct elements that appear in σ.

In the DISTINCT-ELEMENTS problem, our task is to output an (ε, δ)-estimate (as in Definition 0.2.1) to d.

It is provably impossible to solve this problem in sublinear space if one is restricted to either deterministic algorithms (i.e., $\delta = 0$), or exact algorithms (i.e., $\varepsilon = 0$). Thus, we shall seek a randomized approximation algorithm. In this unit, we give a simple algorithm for this problem that has interesting, but not optimal, quality guarantees. Despite being sub-optimal, it is worth studying because

- the algorithm is *especially* simple;
- it introduces us to two ingredients used in tons of randomized algorithms, namely, *universal hashing* and the *median trick*;
- it introduces us to probability tail bounds, a basic technique for the analysis of randomized algorithms.

2.2 The Tidemark Algorithm

The idea behind the algorithm is originally due to Flajolet and Martin [FM85]. We give a slightly modified presentation, due to Alon, Matias and Szegedy [AMS99]. Since that paper designs several other algorithms as well (for other problems), it's good to give this particular algorithm a name more evocative than "AMS algorithm." I call it the *tidemark algorithm* because of how it remembers information about the input stream. Metaphorically speaking, each token has an opportunity to raise the "water level" and the algorithm simply keeps track of the high-water mark, just as a tidemark records the high-water mark left by tidal water.

For an integer $p > 0$, let $\mathrm{zeros}(p)$ denote the number of zeros that the binary representation of p ends with. Formally,

$$\mathrm{zeros}(p) = \max\{i : 2^i \text{ divides } p\}\,.$$

Our algorithm's key ingredient is a 2-*universal hash family*, a very important concept that will come up repeatedly. If you are unfamiliar with the concept, working through Exercises 2-1 and 2-2 is strongly recommended. Once we have this key ingredient, our algorithm is very simple.

Algorithm 3 The tidemark algorithm for the number of distinct elements

```
Initialize:
 1: Choose a random hash function $h : [n] \to [n]$ from a 2-universal family
 2: $z \leftarrow 0$

Process (token $j$) :
 3: if $\mathrm{zeros}(h(j)) > z$ then $z \leftarrow \mathrm{zeros}(h(j))$

Output: $2^{z+\frac{1}{2}}$
```

The basic intuition here is that we expect 1 out of the d distinct tokens to hit $\mathrm{zeros}(h(j)) \ge \log d$, and we don't expect any tokens to hit $\mathrm{zeros}(h(j)) \gg \log d$. Thus, the maximum value of $\mathrm{zeros}(h(j))$ over the stream—which is what we maintain in z—should give us a good approximation to $\log d$. We now analyze this.

2.3 The Quality of the Algorithm's Estimate

Formally, for each $j \in [n]$ and each integer $r \ge 0$, let $X_{r,j}$ be an indicator random variable for the event "$\mathrm{zeros}(h(j)) \ge r$," and let $Y_r = \sum_{j:f_j>0} X_{r,j}$. Let T denote the value of z when the algorithm finishes processing the stream. Clearly,

$$Y_r > 0 \iff T \ge r\,. \tag{2.1}$$

We can restate the above fact as follows (this will be useful later):

$$Y_r = 0 \iff T \le r-1\,. \tag{2.2}$$

Since $h(j)$ is uniformly distributed over the $(\log n)$-bit strings, we have

$$\mathbb{E}\,X_{r,j} = \mathbb{P}\{\mathrm{zeros}(h(j)) \ge r\} = \mathbb{P}\{2^r \text{ divides } h(j)\} = \frac{1}{2^r}\,.$$

We now estimate the expectation and variance of Y_r as follows. The first step of Eq. (2.3) below uses the pairwise independence of the random variables $\{X_{r,j}\}_{j\in[n]}$, which follows from the 2-universality of the hash family from which h is drawn.

$$\mathbb{E}\,Y_r = \sum_{j:f_j>0} \mathbb{E}\,X_{r,j} = \frac{d}{2^r}\,.$$

$$\mathrm{Var}\,Y_r = \sum_{j:f_j>0} \mathrm{Var}\,X_{r,j} \le \sum_{j:f_j>0} \mathbb{E}(X_{r,j}^2) = \sum_{j:f_j>0} \mathbb{E}\,X_{r,j} = \frac{d}{2^r}\,. \tag{2.3}$$

Thus, using Markov's and Chebyshev's inequalities respectively, we have

$$\mathbb{P}\{Y_r > 0\} = \mathbb{P}\{Y_r \ge 1\} \le \frac{\mathbb{E}\,Y_r}{1} = \frac{d}{2^r}\,, \text{ and} \tag{2.4}$$

$$\mathbb{P}\{Y_r = 0\} \le \mathbb{P}\{|Y_r - \mathbb{E}\,Y_r| \ge d/2^r\} \le \frac{\mathrm{Var}\,Y_r}{(d/2^r)^2} \le \frac{2^r}{d}\,. \tag{2.5}$$

Let $\hat{d}$ be the estimate of d that the algorithm outputs. Then $\hat{d} = 2^{T+\frac{1}{2}}$. Let a be the smallest integer such that $2^{a+\frac{1}{2}} \geq 3d$. Using eqs. (2.1) and (2.4), we have

$$\mathbb{P}\left\{\hat{d} \geq 3d\right\} = \mathbb{P}\{T \geq a\} = \mathbb{P}\{Y_a > 0\} \leq \frac{d}{2^a} \leq \frac{\sqrt{2}}{3}. \tag{2.6}$$

Similarly, let b be the largest integer such that $2^{b+\frac{1}{2}} \leq d/3$. Using Eqs. (2.2) and (2.5), we have

$$\mathbb{P}\left\{\hat{d} \leq d/3\right\} = \mathbb{P}\{T \leq b\} = \mathbb{P}\{Y_{b+1} = 0\} \leq \frac{2^{b+1}}{d} \leq \frac{\sqrt{2}}{3}. \tag{2.7}$$

These guarantees are weak in two ways. Firstly, the estimate $\hat{d}$ is only of the "same order of magnitude" as d, and is not an arbitrarily good approximation. Secondly, these failure probabilities in eqs. (2.6) and (2.7) are only bounded by the rather large $\sqrt{2}/3 \approx 47\%$. Of course, we could make the probabilities smaller by replacing the constant "3" above with a larger constant. But a better idea, that does not further degrade the quality of the estimate $\hat{d}$, is to use a standard "median trick" which will come up again and again.

2.4 The Median Trick

Imagine running k copies of this algorithm in parallel, using mutually independent random hash functions, and outputting the median of the k answers. If this median exceeds $3d$, then at least $k/2$ of the individual answers must exceed $3d$, whereas we only expect $k\sqrt{2}/3$ of them to exceed $3d$. By a standard Chernoff bound, this event has probability $2^{-\Omega(k)}$. Similarly, the probability that the median is below $d/3$ is also $2^{-\Omega(k)}$.

Choosing $k = \Theta(\log(1/\delta))$, we can make the sum of these two probabilities work out to at most δ. This gives us an $(O(1), \delta)$-estimate for d. Later, we shall give a different algorithm that will provide an (ε, δ)-estimate with $\varepsilon \to 0$.

The original algorithm requires $O(\log n)$ bits to store (and compute) a suitable hash function, and $O(\log \log n)$ more bits to store z. Therefore, the space used by this final algorithm is $O(\log(1/\delta) \cdot \log n)$. When we reattack this problem with a new algorithm, we will also improve this space bound.

Exercises

These exercises are designed to get you familiar with the very important concept of a 2-universal hash family, as well as give you constructive examples of such families.

Let X and Y be finite sets and let Y^X denote the set of all functions from X to Y. We will think of these functions as "hash" functions. [The term "hash function" has no formal meaning; strictly speaking, one should say "family of hash functions" or "hash family" as we do here.] A family $\mathscr{H} \subseteq Y^X$ is said to be 2-universal if the following property holds, with $h \in_R \mathscr{H}$ picked uniformly at random:

$$\forall x, x' \in X \; \forall y, y' \in Y \left(x \neq x' \;\Rightarrow\; \mathbb{P}_h\left\{h(x) = y \wedge h(x') = y'\right\} = \frac{1}{|Y|^2} \right).$$

We shall give two examples of 2-universal hash families from the set $X = \{0,1\}^n$ to the set $Y = \{0,1\}^k$ (with $k \leq n$).

2-1 Treat the elements of X and Y as column vectors with 0/1 entries. For a matrix $A \in \{0,1\}^{k \times n}$ and vector $b \in \{0,1\}^k$, define the function $h_{A,b} : X \to Y$ by $h_{A,b}(x) = Ax + b$, where all additions and multiplications are performed mod 2.

Prove that the family of functions $\mathscr{H} = \{h_{A,b} : A \in \{0,1\}^{k \times n}, b \in \{0,1\}^k\}$ is 2-universal.

2-2 Identify X with the finite field $\mathbb{F}_{2^n}$ using an arbitrary bijection—truly arbitrary: e.g., the bijection need not map the string 0^n to the zero element of $\mathbb{F}_{2^n}$. For elements $a, b \in X$, define the function $g_{a,b} : X \to Y$ as follows:

$$\begin{aligned} g_{a,b}(x) &= \text{rightmost } k \text{ bits of } f_{a,b}(x)\,, \quad \text{where} \\ f_{a,b}(x) &= ax + b\,, \quad \text{with addition and multiplication performed in } \mathbb{F}_{2^n}\,. \end{aligned}$$

Prove that the family of functions $\mathscr{G} = \{g_{a,b} : a, b \in \mathbb{F}_{2^n}\}$ is 2-universal. Is the family $\mathscr{G}$ better or worse than $\mathscr{H}$ in any sense? Why?

Unit 3

A Better Estimate for Distinct Elements

3.1 The Problem

We revisit the DISTINCT-ELEMENTS problem from Unit 2, giving a better solution, in terms of both approximation guarantee and space usage. We also seek good time complexity. Thus, we are again in the *vanilla streaming model*. We have a stream $\sigma = \langle a_1, a_2, a_3, \ldots, a_m \rangle$, with each $a_i \in [n]$, and this implicitly defines a frequency vector $\boldsymbol{f} = (f_1, \ldots, f_n)$. Let $d = |\{j : f_j > 0\}|$ be the number of distinct elements that appear in σ. We want an (ε, δ)-approximation (as in Definition 0.2.1) to d.

3.2 The BJKST Algorithm

In this section we present the algorithm dubbed BJKST, after the names of the authors: Bar-Yossef, Jayram, Kumar, Sivakumar and Trevisan [BJK+02]. The original paper in which this algorithm is presented actually gives three algorithms, the third (and, in a sense, "best") of which we are presenting. The "zeros" notation below is the same as in Section 2.2. The values b and c are universal constants that will be determined later, based on the desired guarantees on the algorithm's estimate.

Algorithm 4 The BJKST algorithm for DISTINCT-ELEMENTS

```
Initialize:
 1: Choose a random hash function $h : [n] \to [n]$ from a 2-universal family
 2: Choose a random hash function $g : [n] \to [b\varepsilon^{-4} \log^2 n]$ from a 2-universal family
 3: $z \leftarrow 0$
 4: $B \leftarrow \varnothing$

Process (token $j$) :
 5: if zeros$(h(j)) \geq z$ then
 6:     $B \leftarrow B \cup \{(g(j), \text{zeros}(h(j)))\}$
 7:     while $|B| \geq c/\varepsilon^2$ do
 8:         $z \leftarrow z + 1$
 9:         shrink $B$ by removing all $(\alpha, \beta)$ with $\beta < z$

Output: $|B|2^z$
```

Intuitively, this algorithm is a refined version of the tidemark algorithm from Section 2.2. This time, rather than simply tracking the maximum value of zeros$(h(j))$ in the stream, we try to determine the size of the bucket B consisting of all tokens j with $\text{zeros}(h(j)) \geq z$. Of the d distinct tokens in the stream, we expect $d/2^z$ to fall into this bucket. Therefore $|B|2^z$ should be a good estimate for d.

We want B to be small so that we can store enough information (remember, we are trying to save space) to track $|B|$ accurately. At the same time, we want B to be large so that the estimate we produce is accurate enough. It turns out that letting B grow to about $\Theta(1/\varepsilon^2)$ in size is the right tradeoff. Finally, as a space-saving trick, the algorithm does not store the actual tokens in B but only their hash values under g, together with the value of $\text{zeros}(h(j))$ that is needed to remove the appropriate elements from B when B must be shrunk.

We now analyze the algorithm in detail.

3.3 Analysis: Space Complexity

We assume that $1/\varepsilon^2 = o(m)$: otherwise, there is no point to this algorithm! The algorithm has to store h, g, z, and B. Clearly, h and B dominate the space requirement. Using the finite-field-arithmetic hash family from Exercise **2-2** for our hash functions, we see that h requires $O(\log n)$ bits of storage. The bucket B has its size capped at $O(1/\varepsilon^2)$. Each tuple (α, β) in the bucket requires $\log(b\varepsilon^{-4}\log^2 n) = O(\log(1/\varepsilon) + \log\log n)$ bits to store the hash value α, which dominates the $\lceil \log\log n \rceil$ bits required to store the number of zeros β.

Overall, this leads to a space requirement of $O(\log n + (1/\varepsilon^2)(\log(1/\varepsilon) + \log\log n))$.

3.4 Analysis: The Quality of the Estimate

The entire analysis proceeds under the assumption that storing hash values (under g) in B, instead of the tokens themselves, does not change $|B|$. This is true whenever g does not have collisions on the set of tokens to which it is applied. By choosing the constant b large enough, we can ensure that the probability of this happening is at most $1/6$, for each choice of h (you are asked to flesh this out in Exercise **3-1**). Thus, making this assumption adds at most $1/6$ to the error probability. We now analyze the rest of the error, under this no-collision assumption.

The basic setup is the same as in Section 2.3. For each $j \in [n]$ and each integer $r \geq 0$, let $X_{r,j}$ be an indicator random variable for the event "$\text{zeros}(h(j)) \geq r$," and let $Y_r = \sum_{j: f_j > 0} X_{r,j}$. Let T denote the value of z when the algorithm finishes processing the stream, and let $\hat{d}$ denote the estimate output by the algorithm. Then we have

$$\begin{aligned} Y_T &= \text{value of } |B| \text{ when the algorithm finishes, and} \\ \hat{d} &= Y_T 2^T . \end{aligned}$$

Proceeding as in Section 2.3, we obtain

$$\forall r: \quad \mathbb{E} Y_r = \frac{d}{2^r}; \quad \operatorname{Var} Y_r \leq \frac{d}{2^r} . \tag{3.1}$$

Notice that if $T = 0$, then the algorithm never incremented z, which means that $d < c/\varepsilon^2$ and $\hat{d} = |B| = d$. In short, the algorithm computes d exactly in this case.

Otherwise ($T \geq 1$), we say that a FAIL event occurs if $\hat{d}$ is not a $(1 \pm \varepsilon)$-approximation to d. That is,

$$\text{FAIL} \iff |Y_T 2^T - d| \geq \varepsilon d \iff \left| Y_T - \frac{d}{2^T} \right| \geq \frac{\varepsilon d}{2^T} .$$

We can estimate this probability by summing over all possible values $r \in \{1, 2, \ldots, \log n\}$ of T. For the small values of r, a failure will be unlikely when $T = r$, because failure requires a large deviation of Y_r from its mean. For the large values of r, simply having $T = r$ is unlikely. This is the intuition for splitting the summation into two parts below. We need to choose the threshold that separates "small" values of r from "large" ones and we do it as follows.

Let s be the unique integer such that

$$\frac{12}{\varepsilon^2} \leq \frac{d}{2^s} < \frac{24}{\varepsilon^2} . \tag{3.2}$$

Then we calculate

$$\begin{aligned}
\mathbb{P}(\text{FAIL}) &= \sum_{r=1}^{\log n} \mathbb{P}\left\{\left|Y_r - \frac{d}{2^r}\right| \ge \frac{\varepsilon d}{2^r} \wedge T = r\right\} \\
&\le \sum_{r=1}^{s-1} \mathbb{P}\left\{\left|Y_r - \frac{d}{2^r}\right| \ge \frac{\varepsilon d}{2^r}\right\} + \sum_{r=s}^{\log n} \mathbb{P}\{T = r\} \\
&= \sum_{r=1}^{s-1} \mathbb{P}\left\{|Y_r - \mathbb{E}Y_r| \ge \frac{\varepsilon d}{2^r}\right\} + \mathbb{P}\{T \ge s\} && \triangleright \text{ by eq. (3.1)} \\
&= \sum_{r=1}^{s-1} \mathbb{P}\left\{|Y_r - \mathbb{E}Y_r| \ge \frac{\varepsilon d}{2^r}\right\} + \mathbb{P}\{Y_{s-1} \ge c/\varepsilon^2\}. && (3.3)
\end{aligned}$$

Bounding the terms in (3.3) using Chebyshev's inequality and Markov's inequality, respectively, we continue:

$$\begin{aligned}
\mathbb{P}(\text{FAIL}) &\le \sum_{r=1}^{s-1} \frac{\operatorname{Var} Y_r}{(\varepsilon d/2^r)^2} + \frac{\mathbb{E}Y_{s-1}}{c/\varepsilon^2} \\
&\le \sum_{r=1}^{s-1} \frac{2^r}{\varepsilon^2 d} + \frac{\varepsilon^2 d}{c 2^{s-1}} \\
&\le \frac{2^s}{\varepsilon^2 d} + \frac{2\varepsilon^2 d}{c 2^s} \\
&\le \frac{1}{\varepsilon^2} \cdot \frac{\varepsilon^2}{12} + \frac{2\varepsilon^2}{c} \cdot \frac{24}{\varepsilon^2} && \triangleright \text{ by eq. (3.2)} \\
&\le \frac{1}{6}, && (3.4)
\end{aligned}$$

where the final bound is achieved by choosing a large enough constant c.

Recalling that we had started with a no-collision assumption for g, the final probability of error is at most $1/6 + 1/6 = 1/3$. Thus, the above algorithm $(\varepsilon, \frac{1}{3})$-approximates d. As before, by using the median trick, we can improve this to an (ε, δ)-approximation for any $0 < \delta \le 1/3$, at a cost of an $O(\log(1/\delta))$-factor increase in the space usage.

3.5 Optimality

This algorithm is very close to optimal in its space usage. Later in this course, when we study lower bounds, we shall show both an $\Omega(\log n)$ and an $\Omega(\varepsilon^{-2})$ bound on the space required by an algorithm that $(\varepsilon, \frac{1}{3})$-approximates the number of distinct elements. The small gap between these lower bounds and the above upper bound was subsequently closed by Kane, Nelson, and Woodruff [KNW10]: using *considerably* more advanced ideas, they achieved a space bound of $O(\varepsilon^{-2} + \log n)$.

Exercises

3-1 Let $\mathscr{H} \subseteq Y^X$ be a 2-universal hash family, with $|Y| = cM^2$, for some constant $c > 0$. Suppose we use a random function $h \in_R \mathscr{H}$ to hash a stream σ of elements of X, and suppose that σ contains at most M distinct elements. Prove that the probability of a collision (i.e., the event that two distinct elements of σ hash to the same location) is at most $1/(2c)$.

3-2 Recall that we said in class that the buffer B in the BJKST Algorithm for DISTINCT-ELEMENTS can be implemented cleverly by not directly storing the elements of the input stream in B, but instead, storing the hash values of these elements under a secondary hash function whose range is of size cM^2, for a suitable M.

Using the above result, flesh out the details of this clever implementation. (One important detail that you must describe: how do you implement the buffer-shrinking step?) Plug in $c = 3$, for a target collision probability bound of $1/(2c) = 1/6$, and figure out what M should be. Compute the resulting upper bound on the space usage of the algorithm. It should work out to

$$O\left(\log n + \frac{1}{\varepsilon^2}\left(\log\frac{1}{\varepsilon} + \log\log n\right)\right).$$

Unit 4

Approximate Counting

While observing a data stream, we would like to know how *many* tokens we have seen. Just maintain a counter and increment it upon seeing a token: how simple! For a stream of length m, the counter would use $O(\log m)$ space which we said (in eq. 2) was the holy grail.

Now suppose, just for this one unit, that even this amount of space is too much and we would like to solve this problem using $o(\log m)$ space: maybe m is *really* huge or maybe we have to maintain so many counters that we want each one to use less than the trivial amount of space. Since the quantity we wish to maintain—the number of tokens seen—has $m+1$ possible values, maintaining it exactly necessarily requires $\geq \log(m+1)$ bits. Therefore, our solution will have to be approximate.

4.1 The Problem

We are, of course, in the vanilla streaming model. In fact, the identities of the tokens are now immaterial. Therefore, we can think of the input as a stream $(1,1,1,\ldots)$. Our goal is to maintain an approximate count of n, the number of tokens so far, using $o(\log m)$ space, where m is some promised upper bound on n. Note that we're overriding our usual meaning of n: we shall do so for just this unit.

4.2 The Algorithm

The idea behind the algorithm is originally due to Morris [Sr.78], so the approximate counter provided by it is called a "Morris counter." We now present it in a more general form, using modern notation.

Algorithm 5 The Morris counter

```
Initialize:
 1: $x \leftarrow 0$

Process (token) :
 2: with probability $2^{-x}$ do $x \leftarrow x+1$

Output: $2^x - 1$
```

As stated, this algorithm requires $\lceil \log m \rceil$ space after all, because the counter x could grow up to m. However, as we shall soon see, x is *extremely* unlikely to grow beyond $2\log m$ (say), so we can stay within a space bound of $\lceil \log\log m \rceil + O(1)$ by aborting the algorithm if x does grow larger than that.

4.3 The Quality of the Estimate

The interesting part of the analysis is understanding the quality of the algorithm's estimate. Let C_n denote the (random) value of 2^x after n tokens have been processed. Notice that $\hat{n}$, the algorithm's estimate for n, equals $C_n - 1$. We shall prove that $\mathbb{E}C_n = n+1$, which shows that $\hat{n}$ is an *unbiased* estimator for n. This by itself is not enough: we would like to prove a *concentration* result stating that $\hat{n}$ is unlikely to be too far from n. Let's see how well we can do.

To aid the analysis, it will help to rewrite Algorithm 5 in the numerically equivalent (though space-inefficient) way shown below.

Algorithm 6 Thought-experiment algorithm for analysis of Morris counter

```
Initialize:
1: $c \leftarrow 1$

Process (token) :
2: with probability $1/c$ do $c \leftarrow 2c$

Output: $c - 1$
```

Clearly, the variable c in this version of the algorithm is just 2^x in Algorithm 5, so C_n equals c after n tokens. Here is the key lemma in the analysis.

Lemma 4.3.1. *For all* $n \ge 0$, $\mathbb{E}C_n = n+1$ *and* $\operatorname{Var} C_n = n(n-1)/2$.

Proof. Let Z_i be an indicator random variable for the event that c is increased in line 2 upon processing the $(i+1)$th token. Then, $Z_i \sim \operatorname{Bern}(1/C_i)$ and $C_{i+1} = (1+Z_i)C_i$. Using the law of total expectation and the simple formula for expectation of a Bernoulli random variable,

$$\mathbb{E}C_{i+1} = \mathbb{E}\,\mathbb{E}[(1+Z_i)C_i \mid C_i] = \mathbb{E}\left(1+\frac{1}{C_i}\right)C_i = 1 + \mathbb{E}C_i\,.$$

Since $\mathbb{E}C_0 = 1$, this gives us $\mathbb{E}C_n = n+1$ for all n. Similarly,

$$\begin{aligned}
\mathbb{E}C_{i+1}^2 &= \mathbb{E}\,\mathbb{E}[(1+2Z_i+Z_i^2)C_i^2 \mid C_i] \\
&= \mathbb{E}\,\mathbb{E}[(1+3Z_i)C_i^2 \mid C_i] && \triangleright \text{ since } Z_i^2 = Z_i \\
&= \mathbb{E}\left(1+\frac{3}{C_i}\right)C_i^2 \\
&= \mathbb{E}C_i^2 + 3(i+1)\,. && \triangleright \text{ by our above formula for } \mathbb{E}C_i
\end{aligned}$$

Since $\mathbb{E}C_0^2 = 1$, this gives us $\mathbb{E}C_n^2 = 1 + \sum_{i=1}^n 3i = 1 + 3n(n+1)/2$. Therefore,

$$\operatorname{Var} C_n = \mathbb{E}C_n^2 - (\mathbb{E}C_n)^2 = 1 + \frac{3n(n+1)}{2} - (n+1)^2 = \frac{n(n-1)}{2}\,,$$

upon simplification. □

Unfortunately, this variance is too large for us to directly apply the median trick introduced in Section 2.4. (Exercise: try it out! See what happens if you try to bound $\mathbb{P}\{\hat{n} < n/100\}$ using Chebyshev's inequality and Lemma 4.3.1). However, we can still get something good out of Algorithm 5, in two different ways.

- We could tune a parameter in the algorithm, leading to a lower variance at the cost of higher space complexity. This technique, specific to this algorithm, is explored further in the exercises.
- We could plug the algorithm into a powerful, widely-applicable template that we describe below.

4.4 The Median-of-Means Improvement

As noted above, we have an estimator ($\hat{n}$, in this case) for our quantity of interest (n, in this case) that is unbiased but with variance so large that we are unable to get an (ε,δ)-estimate (as in Definition 0.2.1). This situation arises often. A generic way to deal with it is to take our original algorithm (Algorithm 5, in this case) to be a *basic estimator* and then build from it a *final estimator* that runs several independent copies of the basic estimator in parallel and combines their output. The key idea is to first bring the variance down by *averaging* a number of independent copies of the basic estimator, and *then* apply the median trick.

Lemma 4.4.1. *There is a universal positive constant c such that the following holds. Let random variable X be an unbiased estimator for a real quantity Q. Let $\{X_{ij}\}_{i\in[t],j\in[k]}$ be a collection of independent random variables with each X_{ij} distributed identically to X, where*

$$t = c\log\frac{1}{\delta}\,, \quad \textit{and}$$
$$k = \frac{3\operatorname{Var}X}{\varepsilon^2(\mathbb{E}X)^2}\,.$$

Let $Z = \operatorname{median}_{i\in[t]}\left(\frac{1}{k}\sum_{j=1}^{k}X_{ij}\right)$. Then, we have $\mathbb{P}\{|Z-Q|\ge\varepsilon Q\}\le\delta$, i.e., Z is an (ε,δ)-estimate for Q.

Thus, if an algorithm can produce X using s bits of space, then there is an (ε,δ)-estimation algorithm using

$$O\left(s\cdot\frac{\operatorname{Var}X}{(\mathbb{E}X)^2}\cdot\frac{1}{\varepsilon^2}\log\frac{1}{\delta}\right)$$

bits of space.

Proof. For each $i\in[t]$, let $Y_i = k^{-1}\sum_{j=1}^{k}X_{ij}$. Then, by linearity of expectation, we have $\mathbb{E}Y_i = Q$. Since the variables X_{ij} are (at least) pairwise independent, we have

$$\operatorname{Var}Y_i = \frac{1}{k^2}\sum_{j=1}^{k}\operatorname{Var}X_{ij} = \frac{\operatorname{Var}X}{k}\,.$$

Applying Chebyshev's inequality, we obtain

$$\mathbb{P}\{|Y_i-Q|\ge\varepsilon Q\}\le\frac{\operatorname{Var}Y_i}{(\varepsilon Q)^2} = \frac{\operatorname{Var}X}{k\varepsilon^2(\mathbb{E}X)^2} = \frac{1}{3}\,.$$

Now an application of a Chernoff bound (exactly as in the median trick from Section 2.4) tells us that for an appropriate choice of c, we have $\mathbb{P}\{|Z-Q|\ge\varepsilon Q\}\le\delta$. □

Applying the above lemma to the basic estimator given by Algorithm 5 gives us the following result.

Theorem 4.4.2. *For a stream of length at most m, the problem of approximately counting the number of tokens admits an (ε,δ)-estimation in $O(\log\log m\cdot\varepsilon^{-2}\log\delta^{-1})$ space.*

Proof. If we run the basic estimator as is (according to Algorithm 5), using s bits of space, the analysis in Lemma 4.3.1 together with the median-of-means improvement (Lemma 4.4.1) gives us a final (ε,δ)-estimator using $O(s\varepsilon^{-2}\log\delta^{-1})$ space.

Now suppose we tweak the basic estimator to ensure $s = O(\log\log m)$ by aborting if the stored variable x exceeds $2\log m$. An abortion would imply that $C_n\ge m^2\ge n^2$. By Lemma 4.3.1 and Markov's inequality,

$$\mathbb{P}\{C_n\ge n^2\}\le\frac{\mathbb{E}C_n}{n^2} = \frac{n+1}{n^2}\,.$$

Then, by a simple union bound, the probability that any one of the $\Theta(\varepsilon^{-2}\log\delta^{-1})$ parallel runs of the basic estimator aborts is at most $o(1)$. Thus, a final estimator based on this tweaked basic estimator produces an $(\varepsilon,\delta+o(1))$-estimate within the desired space bound. □

Exercises

4-1 Here is a different way to improve the accuracy of the basic estimator in Algorithm 5. Observe that the given algorithm roughly tracks the logarithm (to base 2) of the stream length. Let us change the base from 2 to $1+\beta$ instead, where β is a small positive value, to be determined. Update the pseudocode using this idea, ensuring that the output value is still an unbiased estimator for n.

Analyze the variance of this new estimator and show that, for a suitable setting of β, it directly provides an (ε,δ)-estimate, using only $\log\log m + O(\log\varepsilon^{-1} + \log\delta^{-1})$ bits of space.

4-2 Show how to further generalize the version of the Morris counter given by the previous exercise to solve a more general version of the approximate counting problem where the stream tokens are positive integers and a token j is to be interpreted as "add j to the counter." As usual, the counter starts at zero. Provide pseudocode and rigorously analyze the algorithm's output quality and space complexity.

4-3 Instead of using a median of means for improving the accuracy of an estimator, what if we use a mean of medians? Will it work just as well, or at all?

Unit 5

Finding Frequent Items via (Linear) Sketching

5.1 The Problem

We return to the FREQUENT problem that we studied in Unit 1: given a parameter k, we seek the set of tokens with frequency $> m/k$. The Misra–Gries algorithm, in a single pass, gave us enough information to solve FREQUENT with a second pass: namely, in one pass it computed a data structure which could be queried at any token $j \in [n]$ to obtain a sufficiently accurate estimate $\hat{f}_j$ to its frequency f_j. We shall now give two other one-pass algorithms for this same problem, that we can call FREQUENCY-ESTIMATION.

5.2 Sketches and Linear Sketches

Let MG(σ) denote the data structure computed by Misra–Gries upon processing the stream σ. In Exercise 1-3, we saw a procedure for combining two instances of this data structure that would let us space-efficiently compute MG$(\sigma_1 \circ \sigma_2)$ from MG(σ_1) and MG(σ_2), where "$\circ$" denotes concatenation of streams. Clearly, it would be desirable to be able to combine two data structures in this way, and when it can be done, such a data structure is called a *sketch*.

Definition 5.2.1. A data structure DS(σ) computed in streaming fashion by processing a stream σ is called a *sketch* if there is a space-efficient combining algorithm COMB such that, for every two streams σ_1 and σ_2, we have

$$\text{COMB}(\text{DS}(\sigma_1), \text{DS}(\sigma_2)) = \text{DS}(\sigma_1 \circ \sigma_2)\,.$$

However, the Misra–Gries has the drawback that it does not seem to extend to the turnstile (or even strict turnstile) model. In this unit, we shall design two different solutions to the FREQUENT problem that *do* generalize to turnstile streams. Each algorithm computes a sketch of the input stream in the above sense, but these sketches have an additional important property that we now explain.

Since algorithms for FREQUENT are computing functions of the frequency vector $\boldsymbol{f}(\sigma)$ determined by σ, their sketches will naturally be functions of $\boldsymbol{f}(\sigma)$. It turns out that for the two algorithms in this unit, the sketches will be *linear* functions. That's special enough to call out in another definition.

Definition 5.2.2. A sketching algorithm "sk" is called a *linear sketch* if, for each stream σ over a token universe $[n]$, sk(σ) takes values in a vector space of dimension $\ell = \ell(n)$, and sk(σ) is a linear function of $\boldsymbol{f}(\sigma)$. In this case, ℓ is called the *dimension* of the linear sketch.

Notice that the combining algorithm for linear sketches is to simply add the sketches (in the appropriate vector space). A data stream algorithm based on a linear sketch naturally generalizes from the vanilla to the turnstile model. If the arrival of a token j in the vanilla model causes us to add a vector $\boldsymbol{v}_j$ to the sketch, then an update (j, c) in the turnstile model is handled by adding $c\boldsymbol{v}_j$ to the sketch: this handles both cases $c \geq 0$ and $c < 0$.

We can make things more explicit. Put $\boldsymbol{f} = \boldsymbol{f}(\sigma) \in \mathbb{R}^n$ and $\boldsymbol{y} = \mathrm{sk}(\sigma) \in \mathbb{R}^\ell$, where "sk" is a linear sketch. Upon choosing a basis for $\mathbb{R}^n$ and one for $\mathbb{R}^\ell$, we can express the algorithm "sk" as left multiplication by a suitable sketch matrix $\boldsymbol{S} \in \mathbb{R}^{\ell \times n}$: i.e., $\boldsymbol{y} = \boldsymbol{S}\boldsymbol{f}$. The vector $\boldsymbol{v}_j$ defined above is simply the jth column of $\boldsymbol{S}$. Importantly, a sketching algorithm should not be storing $\boldsymbol{S}$: that would defeat the purpose, which is to save space! Instead, the algorithm should be performing this multiplication implicitly.

All of the above will be easier to grasp upon seeing the concrete examples in this unit, so let us get on with it.

5.3 CountSketch

We now describe the first of our sketching algorithms, called CountSketch, which was introduced by Charikar, Chen and Farach-Colton [CCFC04]. We start with a *basic sketch* that already has most of the required ideas in it. This sketch takes an accuracy parameter ε which should be thought of as small and positive.

Algorithm 7 CountSketch: basic estimator

```
Initialize:
1: $C[1 \ldots k] \leftarrow \vec{0}$, where $k := 3/\varepsilon^2$
2: Choose a random hash function $h : [n] \to [k]$ from a 2-universal family
3: Choose a random hash function $g : [n] \to \{-1, 1\}$ from a 2-universal family

Process (token $(j, c)$):
4: $C[h(j)] \leftarrow C[h(j)] + c\,g(j)$

Output (query $a$):
5: report $\hat{f}_a = g(a)\,C[h(a)]$
```

The sketch computed by this algorithm is the array of counters C, which can be thought of as a vector in $\mathbb{Z}^k$. Note that for two such sketches to be combinable, they must be based on the same hash functions h and g.

5.3.1 The Quality of the Basic Sketch's Estimate

Fix an arbitrary token a and consider the output $X = \hat{f}_a$ on query a. For each token $j \in [n]$, let Y_j be the indicator for the event "$h(j) = h(a)$". Examining the algorithm's workings we see that a token j contributes to the counter $C[h(a)]$ iff $h(j) = h(a)$, and the amount of the contribution is its frequency f_j times the random sign $g(j)$. Thus,

$$X = g(a) \sum_{j=1}^{n} f_j g(j) Y_j = f_a + \sum_{j \in [n] \setminus \{a\}} f_j g(a) g(j) Y_j \,.$$

Since g and h are independent and g is drawn from a 2-universal family, for each $j \neq a$, we have

$$\mathbb{E}[g(a)g(j)Y_j] = \mathbb{E}\,g(a) \cdot \mathbb{E}\,g(j) \cdot \mathbb{E}\,Y_j = 0 \cdot 0 \cdot \mathbb{E}\,Y_j = 0\,. \tag{5.1}$$

Therefore, by linearity of expectation, we have

$$\mathbb{E}X = f_a + \sum_{j \in [n] \setminus \{a\}} f_j\, \mathbb{E}[g(a)g(j)Y_j] = f_a\,. \tag{5.2}$$

Thus, the output $X = \hat{f}_a$ is an *unbiased estimator* for the desired frequency f_a.

We still need to show that X is unlikely to deviate too much from its mean. For this, we analyze its variance. By 2-universality of the family from which h is drawn, we see that for each $j \in [n] \setminus \{a\}$, we have

$$\mathbb{E}\,Y_j^2 = \mathbb{E}\,Y_j = \mathbb{P}\{h(j) = h(a)\} = \frac{1}{k}\,. \tag{5.3}$$

Next, we use 2-universality of the family from which g is drawn, and independence of g and h, to conclude that for all $i, j \in [n]$ with $i \neq j$, we have

$$\mathbb{E}[g(i)g(j)Y_iY_j] = \mathbb{E}\,g(i) \cdot \mathbb{E}\,g(j) \cdot \mathbb{E}[Y_iY_j] = 0 \cdot 0 \cdot \mathbb{E}[Y_iY_j] = 0\,. \tag{5.4}$$

Thus, we calculate

$$\begin{aligned}
\operatorname{Var} X &= 0 + \operatorname{Var}\left[g(a) \sum_{j \in [n]\setminus\{a\}} f_j g(j) Y_j\right] \\
&= \mathbb{E}\left[g(a)^2 \sum_{j \in [n]\setminus\{a\}} f_j^2 Y_j^2 \;+\; g(a)^2 \sum_{\substack{i,j \in [n]\setminus\{a\} \\ i \neq j}} f_i f_j g(i) g(j) Y_i Y_j\right] - \left(\sum_{j \in [n]\setminus\{a\}} f_j\, \mathbb{E}[g(a)g(j)Y_j]\right)^2 \\
&= \sum_{j \in [n]\setminus\{a\}} \frac{f_j^2}{k} \;+\; 0 \;-\; 0 \qquad\qquad \triangleright \text{ using } g(a)^2 = 1, (5.3), (5.4), \text{ and } (5.1) \\
&= \frac{\|\boldsymbol{f}\|_2^2 - f_a^2}{k}\,,
\end{aligned} \tag{5.5}$$

where $\boldsymbol{f} = \boldsymbol{f}(\sigma)$ is the frequency distribution determined by σ. From (5.2) and (5.5), using Chebyshev's inequality, we obtain

$$\begin{aligned}
\mathbb{P}\left\{ |\hat{f}_a - f_a| \geq \varepsilon\sqrt{\|\boldsymbol{f}\|_2^2 - f_a^2} \right\} &= \mathbb{P}\left\{ |X - \mathbb{E}X| \geq \varepsilon\sqrt{\|\boldsymbol{f}\|_2^2 - f_a^2} \right\} \\
&\leq \frac{\operatorname{Var}[X]}{\varepsilon^2(\|\boldsymbol{f}\|_2^2 - f_a^2)} \\
&= \frac{1}{k\varepsilon^2} \\
&= \frac{1}{3}\,.
\end{aligned}$$

For $j \in [n]$, let us define $\boldsymbol{f}_{-j}$ to be the $(n-1)$-dimensional vector obtained by dropping the jth entry of $\boldsymbol{f}$. Then $\left\|\boldsymbol{f}_{-j}\right\|_2^2 = \|\boldsymbol{f}\|_2^2 - f_j^2$. Therefore, we can rewrite the above statement in the following more memorable form.

$$\Pr\left\{ |\hat{f}_a - f_a| \geq \varepsilon \left\|\boldsymbol{f}_{-a}\right\|_2 \right\} \leq \frac{1}{3}\,. \tag{5.6}$$

5.3.2 The Final Sketch

The sketch that is commonly referred to as "Count Sketch" is in fact the sketch obtained by applying the median trick (see Section 2.4) to the above basic sketch, bringing its probability of error down to δ, for a given small $\delta > 0$. Thus, the Count Sketch can be visualized as a two-dimensional array of counters, with each token in the stream causing several counter updates. For the sake of completeness, we spell out this final algorithm in full below.

As in Section 2.4, a standard Chernoff bound argument proves that this estimate $\hat{f}_a$ satisfies

$$\mathbb{P}\left\{ |\hat{f}_a - f_a| \geq \varepsilon \left\|\boldsymbol{f}_{-a}\right\|_2 \right\} \leq \delta\,. \tag{5.7}$$

With a suitable choice of hash family, we can store the hash functions above in $O(t \log n)$ space. Each of the tk counters in the sketch uses $O(\log m)$ space. This gives us an overall space bound of $O(t \log n + tk \log m)$, which is

$$O\left(\frac{1}{\varepsilon^2} \log \frac{1}{\delta} \cdot (\log m + \log n)\right).$$

Algorithm 8 CountSketch: final estimator

```
Initialize:
1: $C[1\ldots t][1\ldots k] \leftarrow \vec{0}$, where $k := 3/\varepsilon^2$ and $t := O(\log(1/\delta))$
2: Choose $t$ independent hash functions $h_1,\ldots h_t : [n] \to [k]$, each from a 2-universal family
3: Choose $t$ independent hash functions $g_1,\ldots g_t : [n] \to \{-1,1\}$, each from a 2-universal family

Process (token $(j,c)$) :
4: for $i \leftarrow 1$ to $t$ do
5:     $C[i][h_i(j)] \leftarrow C[i][h_i(j)] + c\,g_i(j)$

Output (query $a$) :
6: report $\hat{f}_a = \text{median}_{1\le i\le t}\; g_i(a)\,C[i][h_i(a)]$
```

5.4 The Count-Min Sketch

Another solution to FREQUENCY-ESTIMATION is the so-called "Count-Min Sketch", which was introduced by Cormode and Muthukrishnan [CM05]. As with the Count Sketch, this sketch too takes an accuracy parameter ε and an error probability parameter δ. And as before, the sketch consists of a two-dimensional $t \times k$ array of counters, which are updated in a very similar manner, based on hash functions. The values of t and k are set, based on ε and δ, as shown below.

Algorithm 9 Count-Min Sketch

```
Initialize:
1: $C[1\ldots t][1\ldots k] \leftarrow \vec{0}$, where $k := 2/\varepsilon$ and $t := \lceil \log(1/\delta) \rceil$
2: Choose $t$ independent hash functions $h_1,\ldots h_t : [n] \to [k]$, each from a 2-universal family

Process (token $(j,c)$) :
3: for $i \leftarrow 1$ to $t$ do
4:     $C[i][h_i(j)] \leftarrow C[i][h_i(j)] + c$

Output (query $a$) :
5: report $\hat{f}_a = \min_{1\le i\le t} C[i][h_i(a)]$
```

Note how much simpler this algorithm is, as compared to Count Sketch! Also, note that its space usage is

$$O\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}\cdot(\log m + \log n)\right),$$

which is better than that of Count Sketch by a $1/\varepsilon$ factor. The place where Count-Min Sketch is weaker is in its approximation guarantee, which we now analyze.

5.4.1 The Quality of the Algorithm's Estimate

We focus on the case when each token (j,c) in the stream satisfies $c > 0$, i.e., the cash register model. Clearly, in this case, every counter $C[i][h_i(a)]$, corresponding to a token a, is an overestimate of f_a. Thus, we always have

$$f_a \;\le\; \hat{f}_a\,,$$

where $\hat{f}_a$ is the estimate of f_a output by the algorithm.

For a fixed a, we now analyze the excess in one such counter, say in $C[i][h_i(a)]$. Let the random variable X_i denote this excess. For $j \in [n] \setminus \{a\}$, let $Y_{i,j}$ be the indicator of the event "$h_i(j) = h_i(a)$". Notice that j makes a contribution to the counter iff $Y_{i,j} = 1$, and when it does contribute, it causes f_j to be added to this counter. Thus,

$$X_i = \sum_{j \in [n] \setminus \{a\}} f_j Y_{i,j} .$$

By 2-universality of the family from which h_i is drawn, we compute that $\mathbb{E} Y_{i,j} = 1/k$. Thus, by linearity of expectation,

$$\mathbb{E} X_i = \sum_{j \in [n] \setminus \{a\}} \frac{f_j}{k} = \frac{\|\boldsymbol{f}\|_1 - f_a}{k} = \frac{\|\boldsymbol{f}_{-a}\|_1}{k} .$$

Since each $f_j \geq 0$, we have $X_i \geq 0$, and we can apply Markov's inequality to get

$$\mathbb{P}\{X_i \geq \varepsilon \|\boldsymbol{f}_{-a}\|_1\} \leq \frac{\|\boldsymbol{f}_{-a}\|_1}{k\varepsilon \|\boldsymbol{f}_{-a}\|_1} = \frac{1}{2} ,$$

by our choice of k.

The above probability is for one counter. We have t such counters, mutually independent. The excess in the output, $\hat{f}_a - f_a$, is the minimum of the excesses X_i, over all $i \in [t]$. Thus,

$$\begin{aligned} \mathbb{P}\left\{\hat{f}_a - f_a \geq \varepsilon \|\boldsymbol{f}_{-a}\|_1\right\} &= \mathbb{P}\left\{\min\{X_1, \ldots, X_t\} \geq \varepsilon \|\boldsymbol{f}_{-a}\|_1\right\} \\ &= \mathbb{P}\left\{\bigwedge_{i=1}^{t} \left(X_i \geq \varepsilon \|\boldsymbol{f}_{-a}\|_1\right)\right\} \\ &= \prod_{i=1}^{t} \Pr\{X_i \geq \varepsilon \|\boldsymbol{f}_{-a}\|_1\} \\ &\leq \frac{1}{2^t} , \end{aligned}$$

and using our choice of t, this probability is at most δ. Thus, we have shown that, with high probability,

$$f_a \leq \hat{f}_a \leq f_a + \varepsilon \|\boldsymbol{f}_{-a}\|_1 ,$$

where the left inequality always holds, and the right inequality fails with probability at most δ.

The reason this estimate is weaker than that of Count Sketch is that its deviation is bounded by $\varepsilon \|\boldsymbol{f}_{-a}\|_1$, rather than $\varepsilon \|\boldsymbol{f}_{-a}\|_2$. For all vectors $\boldsymbol{z} \in \mathbb{R}^n$, we have $\|\boldsymbol{z}\|_1 \geq \|\boldsymbol{z}\|_2$. The inequality is tight when $\boldsymbol{z}$ has a single nonzero entry. It is at its weakest when all entries of $\boldsymbol{z}$ are equal in absolute value: the two norms are then off by a factor of $\sqrt{n}$ from each other. Thus, the quality of the estimate of Count Sketch gets better (in comparison to Count-Min Sketch) as the stream's frequency vector gets more "spread out".

5.5 Comparison of Frequency Estimation Methods

At this point, we have studied three methods to estimate frequencies of tokens in a stream. The following table throws in a fourth method, and compares these methods by summarizing their key features.

Method	$\hat{f}_a - f_a \in \cdots$	**Space,** $O(\cdot)$	**Error Probability**	**Model**
Misra–Gries	$\left[-\varepsilon \|\boldsymbol{f}_{-a}\|_1, 0\right]$	$\frac{1}{\varepsilon}(\log m + \log n)$	0 (deterministic)	Cash register
CountSketch	$\left[-\varepsilon \|\boldsymbol{f}_{-a}\|_2, \varepsilon \|\boldsymbol{f}_{-a}\|_2\right]$	$\frac{1}{\varepsilon^2} \log \frac{1}{\delta} \cdot (\log m + \log n)$	δ (overall)	Turnstile
Count-Min Sketch	$\left[0, \varepsilon \|\boldsymbol{f}_{-a}\|_1\right]$	$\frac{1}{\varepsilon} \log \frac{1}{\delta} \cdot (\log m + \log n)$	δ (upper bound)	Cash register
Count/Median	$\left[-\varepsilon \|\boldsymbol{f}_{-a}\|_1, \varepsilon \|\boldsymbol{f}_{-a}\|_1\right]$	$\frac{1}{\varepsilon} \log \frac{1}{\delta} \cdot (\log m + \log n)$	δ (overall)	Turnstile

The claims in the first row can be proved by analyzing the Misra–Gries algorithm from Lecture 1 slightly differently. The last row refers to an algorithm that maintains the same data structure as the Count-Min Sketch, but answers queries by reporting the median of the values of the relevant counters, rather than the minimum. It is a simple (and instructive) exercise to analyze this algorithm and prove the claims in the last row.

Exercises

5-1 Prove that for every integer $n \geq 1$ and every vector $\boldsymbol{z} \in \mathbb{R}^n$, we have $\|\boldsymbol{z}\|_1 / \sqrt{n} \leq \|\boldsymbol{z}\|_2 \leq \|\boldsymbol{z}\|_1$. For both inequalities, determine when equality holds.

5-2 Write out the Count/Median algorithm formally and prove that it satisfies the properties claimed in the last row of the above table.

5-3 Our estimate of $\varepsilon \left\|\boldsymbol{f}_{-a}\right\|_2$ on the absolute error of CountSketch is too pessimistic in many practical situations where the data are highly skewed, i.e., where most of the weight of the vector $\boldsymbol{f}$ is supported on a small "constant" number of elements. To make this precise, we define $\boldsymbol{f}_{-a}^{\mathrm{res}(\ell)}$ to be the $(n-1)$-dimensional vector obtained by dropping the ath entry of $\boldsymbol{f}$ and then setting the ℓ largest (by absolute value) entries to zero.

Now consider the CountSketch estimate $\hat{f}_a$ as computed by the algorithm in Sec 5.3.2 with the only change being that k is set to $6/\varepsilon^2$. Prove that

$$\mathbb{P}\left\{|\hat{f}_a - f_a| \geq \varepsilon \left\|\boldsymbol{f}_{-a}^{\mathrm{res}(\ell)}\right\|_2\right\} \leq \delta\,,$$

where $\ell = 1/\varepsilon^2$.

5-4 Consider a stream σ in the turnstile model, defining a frequency vector $\boldsymbol{f} \geq \mathbf{0}$. The Count-Min Sketch solves the problem of estimating f_j, given j, but does not directly give us a *quick* way to identify, e.g., the set of elements with frequency greater than some threshold. Fix this.

In greater detail: Let α be a constant with $0 < \alpha < 1$. We would like to maintain a suitable summary of the stream (some enhanced version of Count-Min Sketch, perhaps?) so that we can, on demand, quickly produce a set $S \subseteq [n]$ satisfying the following properties w.h.p.: (1) S contains every j such that $f_j \geq \alpha F_1$; (2) S does not contain any j such that $f_j < (\alpha - \varepsilon)F_1$. Here, $F_1 = F_1(\sigma) = \|\boldsymbol{f}\|_1$. Design a data stream algorithm that achieves this. Your space usage, as well as the time taken to process each token and to produce the set S, should be polynomial in the usual parameters, $\log m$, $\log n$, and $1/\varepsilon$, and may depend arbitrarily on α.

Unit 6

Estimating Frequency Moments

6.1 Background and Motivation

We are in the vanilla streaming model. We have a stream $\sigma = \langle a_1, \ldots, a_m \rangle$, with each $a_j \in [n]$, and this implicitly defines a frequency vector $\boldsymbol{f} = \boldsymbol{f}(\sigma) = (f_1, \ldots, f_n)$. Note that $f_1 + \cdots + f_n = m$. The kth frequency moment of the stream, denoted $F_k(\sigma)$ or simply F_k, is defined as follows:

$$F_k := \sum_{j=1}^{n} f_j^k = \|\boldsymbol{f}\|_k^k . \tag{6.1}$$

Using the terminology "kth" suggests that k is a positive integer. But in fact the definition above makes sense for every real $k > 0$. And we can even give it a meaning for $k = 0$, if we first rewrite the definition of F_k slightly: $F_k = \sum_{j:f_j>0} f_j^k$. With this definition, we get

$$F_0 = \sum_{j:f_j>0} f_j^0 = |\{j : f_j > 0\}| ,$$

which is the number of distinct tokens in σ. (We could have arrived at the same result by sticking with the original definition of F_k and adopting the convention $0^0 = 0$.)

We have seen that F_0 can be (ε, δ)-approximated using space logarithmic in m and n. And $F_1 = m$ is trivial to compute exactly. Can we say anything for general F_k? We shall investigate this problem in this and the next few lectures.

By way of motivation, note that F_2 represents the size of the self join $r \bowtie r$, where r is a relation in a database, with f_j denoting the frequency of the value j of the join attribute. Imagine that we are in a situation where r is a huge relation and n, the size of the domain of the join attribute, is also huge; the tuples can only be accessed in streaming fashion (or perhaps it is much cheaper to access them this way than to use random access). Can we, with one pass over the relation r, compute a good estimate of the self join size? Estimation of join sizes is a crucial step in database query optimization.

The solution we shall eventually see for this F_2 estimation problem will in fact allow us to estimate arbitrary equi-join sizes (not just self joins). For now, though, we give an (ε, δ)-approximation for arbitrary F_k, provided $k \geq 2$, using space sublinear in m and n. The algorithm we present is due to Alon, Matias and Szegedy [AMS99], and is *not* the best algorithm for the problem, though it is the easiest to understand and analyze.

6.2 The (Basic) AMS Estimator for F_k

We first describe a surprisingly simple basic estimator that gets the answer right in expectation, i.e., it is an *unbiased estimator*. Eventually, we shall run many independent copies of this basic estimator in parallel and combine the results to get our final estimator, which will have good error guarantees.

The estimator works as follows. Pick a token from the stream σ uniformly at random, i.e., pick a position $J \in_R [m]$. Count the length, m, of the stream and the number, r, of occurrences of our picked token a_J in the stream from that point on: $r = |\{j \geq J : a_j = a_J\}|$. The basic estimator is then defined to be $m(r^k - (r-1)^k)$.

The catch is that we don't know m beforehand, and picking a token uniformly at random requires a little cleverness, as seen in the pseudocode below. This clever way of picking a uniformly random stream element is called *reservoir* sampling.

Algorithm 10 AMS basic estimator for frequency moments

```
Initialize:
1: $(m, r, a) \leftarrow (0, 0, 0)$

Process (token $j$) :
2: $m \leftarrow m + 1$
3: with probability $1/m$ do
4:     $a \leftarrow j$
5:     $r \leftarrow 0$
6: if $j = a$ then
7:     $r \leftarrow r + 1$

Output: $m(r^k - (r-1)^k)$
```

This algorithm uses $O(\log m)$ bits to store m and r, plus $\lceil \log n \rceil$ bits to store the token a, for a total space usage of $O(\log m + \log n)$. Although stated for the vanilla streaming model, it has a natural generalization to the cash register model. It is a good homework exercise to figure this out.

6.3 Analysis of the Basic Estimator

First of all, let us agree that the algorithm does indeed compute r as described above. For the analysis, it will be convenient to think of the algorithm as picking a random token from σ in two steps, as follows.

1. Pick a random token *value*, $a \in [n]$, with $\mathbb{P}\{a = j\} = f_j/m$ for each $j \in [n]$.
2. Pick one of the f_a occurrences of a in σ uniformly at random.

Let A and R denote the (random) values of a and r after the algorithm has processed σ, and let X denote its output. Taking the above viewpoint, let us condition on the event $A = j$, for some particular $j \in [n]$. Under this condition, R is equally likely to be any of the values $\{1, \ldots, f_j\}$, depending on which of the f_j occurrences of j was picked by the algorithm. Therefore,

$$\mathbb{E}[X \mid A = j] = \mathbb{E}[m(R^k - (R-1)^k) \mid A = j] = \sum_{i=1}^{f_j} \frac{1}{f_j} \cdot m(i^k - (i-1)^k) = \frac{m}{f_j}(f_j^k - 0^k).$$

By the law of total expectation,

$$\mathbb{E}X = \sum_{j=1}^{n} \mathbb{P}\{A = j\}\, \mathbb{E}[X \mid A = j] = \sum_{j=1}^{n} \frac{f_j}{m} \cdot \frac{m}{f_j} \cdot f_j^k = F_k.$$

This shows that X is indeed an unbiased estimator for F_k.

We shall now bound $\operatorname{Var} X$ from above. Calculating the expectation as before, we have

$$\operatorname{Var} X \leq \mathbb{E}X^2 = \sum_{j=1}^{n} \frac{f_j}{m} \sum_{i=1}^{f_j} \frac{1}{f_j} \cdot m^2 (i^k - (i-1)^k)^2 = m \sum_{j=1}^{n} \sum_{i=1}^{f_j} \left(i^k - (i-1)^k\right)^2. \tag{6.2}$$

By the mean value theorem (from basic calculus), for all $x \geq 1$, there exists $\xi(x) \in [x-1,x]$ such that

$$x^k - (x-1)^k = k\xi(x)^{k-1} \leq kx^{k-1}\,,$$

where the last step uses $k \geq 1$. Using this bound in (6.2), we get

$$\begin{aligned}
\operatorname{Var} X &\leq m \sum_{j=1}^{n} \sum_{i=1}^{f_j} ki^{k-1}\left(i^k - (i-1)^k\right) \\
&\leq m \sum_{j=1}^{n} kf_j^{k-1} \sum_{i=1}^{f_j} \left(i^k - (i-1)^k\right) \\
&= m \sum_{j=1}^{n} kf_j^{k-1} f_j^k \\
&= kF_1F_{2k-1}\,. \qquad (6.3)
\end{aligned}$$

For reasons we shall soon see, it will be convenient to bound $\operatorname{Var} X$ by a multiple of $(\mathbb{E}X)^2$, i.e., F_k^2. To do so, we shall use the following lemma.

Lemma 6.3.1. *Let $n > 0$ be an integer and let $x_1, \ldots, x_n \geq 0$ and $k \geq 1$ be reals. Then*

$$\left(\sum x_i\right)\left(\sum x_i^{2k-1}\right) \leq n^{1-1/k}\left(\sum x_i^k\right)^2,$$

where all the summations range over $i \in [n]$.

Proof. We continue to use the convention that summations range over $i \in [n]$. Let $x_* = \max_{i\in[n]} x_i$. Then, we have

$$x_*^{k-1} = \left(x_*^k\right)^{(k-1)/k} \leq \left(\sum x_i^k\right)^{(k-1)/k}. \qquad (6.4)$$

Since $k \geq 1$, by the power mean inequality (or directly, by the convexity of the function $x \mapsto x^k$), we have

$$\frac{1}{n}\sum x_i \leq \left(\frac{1}{n}\sum x_i^k\right)^{1/k}. \qquad (6.5)$$

Using (6.4) and (6.5) in the second and third steps (respectively) below, we compute

$$\begin{aligned}
\left(\sum x_i\right)\left(\sum x_i^{2k-1}\right) &\leq \left(\sum x_i\right)\left(x_*^{k-1}\sum x_i^k\right) \\
&\leq \left(\sum x_i\right)\left(\sum x_i^k\right)^{(k-1)/k}\left(\sum x_i^k\right) \\
&\leq n^{1-1/k}\left(\sum x_i^k\right)^{1/k}\left(\sum x_i^k\right)^{(k-1)/k}\left(\sum x_i^k\right) \\
&= n^{1-1/k}\left(\sum x_i^k\right)^2,
\end{aligned}$$

which completes the proof. □

Using the above lemma in (6.3), with $x_j = f_j$, we get

$$\operatorname{Var} X \leq kF_1F_{2k-1} \leq kn^{1-1/k}F_k^2\,. \qquad (6.6)$$

6.4 The Final Estimator and Space Bound

Our final F_k estimator, which gives us good accuracy guarantees, is obtained by combining several independent basic estimators and using the median-of-means improvement (Lemma 4.4.1 in Section 4.4). By that lemma, we can obtain an (ε,δ)-estimator for F_k by combining $O(t\varepsilon^{-2}\log\delta^{-1})$ independent copies of the basic estimator X, where

$$t = \frac{\operatorname{Var} X}{(\mathbb{E}X)^2} \le \frac{kn^{1-1/k}F_k^2}{F_k^2} = kn^{1-1/k}\,.$$

As noted above, the space used to compute each copy of X, using Algorithm 10, is $O(\log m + \log n)$, leading to a final space bound of

$$O\left(\frac{1}{\varepsilon^2}\log\frac{1}{\delta}\cdot kn^{1-1/k}(\log m + \log n)\right) = \widetilde{O}(\varepsilon^{-2}n^{1-1/k})\,. \tag{6.7}$$

6.4.1 The Soft-O Notation

For the first time in these notes, we have a space bound that is sublinear, but not polylogarithmic, in n (or m). In such cases it is convenient to adopt an $\widetilde{O}$-notation, also known as a "soft-O" notation, which suppresses factors polynomial in $\log m$, $\log n$, $\log\varepsilon^{-1}$, and $\log\delta^{-1}$. We have adopted this notation in eq. (6.7), leading to the memorable form $\widetilde{O}(\varepsilon^{-2}n^{1-1/k})$. Note that we are also treating k as a constant here.

The above bound is good, but not optimal, as we shall soon see. The optimal bound (upto polylogarithmic factors) is $\widetilde{O}(\varepsilon^{-2}n^{1-2/k})$ instead; there are known lower bounds of $\Omega(n^{1-2/k})$ and $\Omega(\varepsilon^{-2})$. We shall see how to achieve this better upper bound in a subsequent unit.

Unit 7

The Tug-of-War Sketch

At this point, we have seen a sublinear-space algorithm — the AMS estimator — for estimating the kth frequency moment, $F_k = f_1^k + \cdots + f_n^k$, of a stream σ. This algorithm works for $k \geq 2$, and its space usage depends on n as $\widetilde{O}(n^{1-1/k})$. This fails to be polylogarithmic even in the important case $k = 2$, which we used as our motivating example when introducing frequency moments in the previous lecture. Also, the algorithm does *not* produce a sketch in the sense of Section 5.2.

But Alon, Matias and Szegedy [AMS99] also gave an *amazing* algorithm that *does* produce a sketch—a linear sketch of merely logarithmic size—which allows one to estimate F_2. What is amazing about the algorithm is that seems to do almost nothing.

7.1 The Basic Sketch

We describe the algorithm in the turnstile model.

Algorithm 11 Tug-of-War Sketch for F_2

```
Initialize:
  1: Choose a random hash function $h : [n] \to \{-1, 1\}$ from a 4-universal family
  2: $z \leftarrow 0$

Process (token $(j, c)$) :
  3: $z \leftarrow z + c\,h(j)$

Output: $z^2$
```

The sketch is simply the random variable z. It is pulled in the positive direction by those tokens j that have $h(j) = 1$ and is pulled in the negative direction by the rest of the tokens; hence the name Tug-of-War Sketch. Clearly, the absolute value of z never exceeds $f_1 + \cdots + f_k = m$, so it takes $O(\log m)$ bits to store this sketch. It also takes $O(\log n)$ bits to store the hash function h, for an appropriate 4-universal family.

7.1.1 The Quality of the Estimate

Let Z denote the value of z after the algorithm has processed σ. For convenience, define $Y_j = h(j)$ for each $j \in [n]$. Then $Z = \sum_{j=1}^n f_j Y_j$. Note that $Y_j^2 = 1$ and $\mathbb{E} Y_j = 0$, for each j. Therefore,

$$\mathbb{E} Z^2 = \mathbb{E}\left[\sum_{j=1}^n f_j^2 Y_j^2 + \sum_{i=1}^n \sum_{\substack{j=1 \\ j\neq i}}^n f_i f_j Y_i Y_j\right] = \sum_{j=1}^n f_j^2 + \sum_{i=1}^n \sum_{\substack{j=1 \\ j\neq i}}^n f_i f_j \mathbb{E} Y_i \mathbb{E} Y_j = F_2 ,$$

where we used the fact that $\{Y_j\}_{j\in[n]}$ are pairwise independent (in fact, they are 4-wise independent, because h was picked from a 4-universal family). This shows that the algorithm's output, Z^2, is indeed an unbiased estimator for F_2.

The variance of the estimator is $\operatorname{Var} Z^2 = \mathbb{E} Z^4 - (\mathbb{E} Z^2)^2 = \mathbb{E} Z^4 - F_2^2$. We bound this as follows. By linearity of expectation, we have

$$\mathbb{E} Z^4 = \sum_{i=1}^n \sum_{j=1}^n \sum_{k=1}^n \sum_{\ell=1}^n f_i f_j f_k f_\ell \, \mathbb{E}[Y_i Y_j Y_k Y_\ell] .$$

Suppose one of the indices in (i,j,k,ℓ) appears exactly once in that 4-tuple. Without loss of generality, we have $i \notin \{j,k,\ell\}$. By 4-wise independence, we then have $\mathbb{E}[Y_i Y_j Y_k Y_\ell] = \mathbb{E} Y_i \cdot \mathbb{E}[Y_j Y_k Y_\ell] = 0$, because $\mathbb{E} Y_i = 0$. It follows that the only potentially nonzero terms in the above sum correspond to those 4-tuples (i,j,k,ℓ) that consist either of one index occurring four times, or else two distinct indices occurring twice each. Therefore we have

$$\mathbb{E} Z^4 = \sum_{j=1}^n f_j^4 \mathbb{E} Y_j^4 + 6 \sum_{i=1}^n \sum_{j=i+1}^n f_i^2 f_j^2 \, \mathbb{E}[Y_i^2 Y_j^2] = F_4 + 6 \sum_{i=1}^n \sum_{j=i+1}^n f_i^2 f_j^2 ,$$

where the coefficient "6" corresponds to the $\binom{4}{2} = 6$ permutations of (i,i,j,j) with $i \neq j$. Thus,

$$\begin{aligned} \operatorname{Var} Z^2 &= F_4 - F_2^2 + 6 \sum_{i=1}^n \sum_{j=i+1}^n f_i^2 f_j^2 \\ &= F_4 - F_2^2 + 3\left(\left(\sum_{j=1}^n f_j^2\right)^2 - \sum_{j=1}^n f_j^4\right) \\ &= F_4 - F_2^2 + 3(F_2^2 - F_4) \le 2F_2^2 . \end{aligned}$$

7.2 The Final Sketch

As before, having bounded the variance, we can design a final sketch from the above basic sketch by a median-of-means improvement. By Lemma 4.4.1, this will blow up the space usage by a factor of

$$\frac{\operatorname{Var} Z^2}{(\mathbb{E} Z^2)^2} \cdot O\left(\frac{1}{\varepsilon^2} \log \frac{1}{\delta}\right) \le \frac{2F_2^2}{F_2^2} \cdot O\left(\frac{1}{\varepsilon^2} \log \frac{1}{\delta}\right) = O\left(\frac{1}{\varepsilon^2} \log \frac{1}{\delta}\right)$$

in order to give an (ε,δ)-estimate. Thus, we have estimated F_2 using space $O(\varepsilon^{-2} \log(\delta^{-1})(\log m + \log n))$, with a sketching algorithm that in fact computes a *linear* sketch.

7.2.1 A Geometric Interpretation

The AMS Tug-of-War Sketch has a nice geometric interpretation. Consider a final sketch that consists of t independent copies of the basic sketch. Let $\boldsymbol{M} \in \mathbb{R}^{t\times n}$ be the matrix that "transforms" the frequency vector $\boldsymbol{f}$ into the t-dimensional sketch vector $\boldsymbol{y}$. Note that $\boldsymbol{M}$ is not a fixed matrix but a random matrix with ± 1 entries: it is drawn from a certain distribution described implicitly by the hash family. Specifically, if M_{ij} denotes the (i,j)-entry of $\boldsymbol{M}$, then $M_{ij} = h_i(j)$, where h_i is the hash function used by the ith basic sketch.

Let $t = 6/\varepsilon^2$. By stopping the analysis in Lemma 4.4.1 after the Chebyshev step (and before the "median trick" Chernoff step), we obtain that

$$\mathbb{P}\left\{\left|\frac{1}{t}\sum_{i=1}^{t} y_i^2 - F_2\right| \geq \varepsilon F_2\right\} \leq \frac{1}{3},$$

where the probability is taken with respect to the above distribution of $\boldsymbol{M}$, resulting in a random sketch vector $\boldsymbol{y} = (y_1, \ldots, y_t)$. Thus, with probability at least $2/3$, we have

$$\left\|\frac{1}{\sqrt{t}}\boldsymbol{Mf}\right\|_2 = \frac{1}{\sqrt{t}}\|\boldsymbol{y}\|_2 \in \left[\sqrt{1-\varepsilon}\,\|\boldsymbol{f}\|_2, \sqrt{1+\varepsilon}\,\|\boldsymbol{f}\|_2\right] \subseteq \left[(1-\varepsilon)\,\|\boldsymbol{f}\|_2, (1+\varepsilon)\,\|\boldsymbol{f}\|_2\right]. \tag{7.1}$$

This can be interpreted as follows. The (random) matrix $\boldsymbol{M}/\sqrt{t}$ performs a *dimension reduction*, simplifying an n-dimensional vector $\boldsymbol{f}$ to a t-dimensional sketch $\boldsymbol{y}$—with $t = O(1/\varepsilon^2)$—while preserving ℓ_2-norm within a $(1 \pm \varepsilon)$ factor. Of course, this is only guaranteed to happen with probability at least $2/3$. But clearly this correctness probability can be boosted to an arbitrary constant less than 1, while keeping $t = O(1/\varepsilon^2)$.

The "amazing" AMS sketch now feels quite natural, under this geometric interpretation. We are using dimension reduction to maintain a low-dimensional image of the frequency vector. This image, by design, has the property that its ℓ_2-length approximates that of the frequency vector very well. Which of course is what we're after, because the second frequency moment, F_2, is just the square of the ℓ_2-length.

Since the sketch is linear, we now also have an algorithm to estimate the ℓ_2-difference $\|\boldsymbol{f}(\sigma) - \boldsymbol{f}(\sigma')\|_2$ between two streams σ and σ'.

Exercises

7-1 In Section 6.1, we noted that F_2 represents the size of a self join in a relational database and remarked that our F_2 estimation algorithm would allow us to estimate arbitrary equi-join sizes (not just self joins). Justify this by designing a sketch that can scan a relation in one streaming pass such that, based on the sketches of two *different* relations, we can estimate the size of their join. Explain how to compute the estimate.

Recall that for two relations (i.e., tables in a database) $r(A,B)$ and $s(A,C)$, with a common attribute (i.e., column) A, we define the join $r \bowtie s$ to be a relation consisting of all tuples (a,b,c) such that $(a,b) \in r$ and $(a,c) \in s$. Therefore, if $f_{r,j}$ and $f_{s,j}$ denote the frequencies of j in the first columns (i.e., "A"-columns) of r and s, respectively, and j can take values in $[n]$, then the size of the join is $\sum_{j=1}^{n} f_{r,j} f_{s,j}$.

7-2 As noted in Section 7.2.1, the $t \times n$ matrix that realizes the tug-of-war sketch has every entry in $\{-1/\sqrt{t}, 1/\sqrt{t}\}$: in particular, every entry is nonzero. Therefore, in a streaming setting, updating the sketch in response to a token arrival takes $\Theta(t)$ time, under the reasonable assumption that the processor can perform arithmetic on $\Theta(\log n)$-bit integers in constant time.

Consider the matrix $\boldsymbol{P} \in \mathbb{R}^{t \times n}$ given by

$$P_{ij} = \begin{cases} g(j), & \text{if } i = h(j), \\ 0, & \text{otherwise,} \end{cases}$$

where hash functions $g\colon [n] \to \{-1, 1\}$ and $h\colon [n] \to [t]$ are drawn from a k_1-universal and a k_2-universal family, respectively. Show that using $\boldsymbol{P}$ as a sketch matrix (for some choice of constants k_1 and k_2) leads to dimension reduction guarantees similar to eq. (7.1). What is the per-token update time achievable using $\boldsymbol{P}$ as the sketch matrix?

Unit 8

Estimating Norms Using Stable Distributions

As noted at the end of Unit 7, the AMS Tug-of-War sketch allows us to estimate the ℓ_2-difference between two data streams. Estimating similarity metrics between streams is an important class of problems, so it is nice to have such a clean solution for this specific metric.

However, this raises a burning question: Can we do the same for other ℓ_p norms, especially the ℓ_1 norm? The ℓ_1-difference between two streams can be interpreted (modulo appropriate scaling) as the total variation distance (a.k.a., statistical distance) between two probability distributions: a fundamental and important metric. Unfortunately, although our log-space F_2 algorithm automatically gave us a log-space ℓ_2 algorithm, the trivial log-space F_1 algorithm works only in the cash register model and does not give an ℓ_1 algorithm at all.

It turns out that thinking harder about the geometric interpretation of the AMS Tug-of-War Sketch leads us on a path to polylogarithmic space ℓ_p norm estimation algorithms, for all $p \in (0,2]$. Such algorithms were given by Indyk [Ind06], and we shall study them now. For the first time in this course, it will be necessary to gloss over several technical details of the algorithms, so as to have a clear picture of the important ideas.

8.1 A Different ℓ_2 Algorithm

The length-preserving dimension reduction achieved by the Tug-of-War Sketch is reminiscent of the famous Johnson-Lindenstrauss Lemma [JL84, FM88]. One high-level way of stating the JL Lemma is that the random linear map given by a $t \times n$ matrix whose entries are independently drawn from the standard normal distribution $\mathcal{N}(0,1)$ is length-preserving (up to a scaling factor) with high probability. To achieve $1 \pm \varepsilon$ error, it suffices to take $t = O(1/\varepsilon^2)$. Let us call such a matrix a JL Sketch matrix. Notice that the sketch matrix for the Tug-of-War sketch is a very similar object, except that

1. its entries are uniformly distributed in $\{-1,1\}$: a much simpler distribution;
2. its entries do not have to be fully independent: 4-wise independence in each row suffices; and
3. it has a succinct description: it suffices to describe the hash functions that generate the rows.

The above properties make the Tug-of-War Sketch "data stream friendly". But as a thought experiment one can consider an algorithm that uses a JL Sketch matrix instead. It would give a correct algorithm for ℓ_2 estimation, except that its space usage would be very large, as we would have to store the entire sketch matrix. In fact, since this hypothetical algorithm calls for arithmetic with real numbers, it is unimplementable as stated.

Nevertheless, this algorithm has something to teach us, and will generalize to give (admittedly unimplementable) ℓ_p algorithms for each $p \in (0,2]$. Later we shall make these algorithms realistic and space-efficient. For now, we consider the basic sketch version of this algorithm, i.e., we maintain just one entry of $\boldsymbol{Mf}$, where $\boldsymbol{M}$ is a JL Sketch matrix. The pseudocode below shows the operations involved.

Algorithm 12 Sketch for ℓ_2 based on normal distribution

```
Initialize:
 1: Choose $Y_1, \ldots, Y_n$ independently, each from $\mathcal{N}(0,1)$
 2: $z \leftarrow 0$

Process (token $(j,c)$):
 3: $z \leftarrow z + cY_j$

Output: $z^2$
```

Let Z denote the value of x when this algorithm finishes processing σ. Then $Z = \sum_{j=1}^{n} f_j Y_j$. From basic statistics, using the independence of the collection $\{Y_j\}_{j \in [n]}$, we know that Z has the same distribution as $\|\boldsymbol{f}\|_2 Y$, where $Y \sim \mathcal{N}(0,1)$. This is a fundamental property of the normal distribution.[1] Therefore, we have $\mathbb{E} Z^2 = \|\boldsymbol{f}\|_2^2 = F_2$, which gives us our unbiased estimator for F_2.

8.2 Stable Distributions

The fundamental property of the normal distribution that was used above has a generalization, which is the key to generalizing this algorithm. The next definition captures the general property.

Definition 8.2.1. Let $p > 0$ be a real number. A probability distribution $\mathcal{D}_p$ over the reals is said to be *p-stable* if for all integers $n \geq 1$ and all $\mathbf{c} = (c_1, \ldots, c_n) \in \mathbb{R}^n$, the following property holds. If $X_1, \ldots, X_n$ are independent and each $X_i \sim \mathcal{D}_p$, then $c_1 X_1 + \cdots + c_n X_n$ has the same distribution as $\bar{c}X$, where $X \sim \mathcal{D}_p$ and

$$\bar{c} = \left(c_1^p + \cdots + c_n^p\right)^{1/p} = \|\mathbf{c}\|_p \, .$$

The concept of stable distributions dates back to Lévy [Lév54] and is more general than what we need here. It is known that p-stable distributions exist for all $p \in (0,2]$, and do not exist for any $p > 2$. The fundamental property above can be stated simply as: "The standard normal distribution is 2-stable."

Another important example of a stable distribution is the Cauchy distribution, which can be shown to be 1-stable. Just as the standard normal distribution has density function

$$\phi(x) = \frac{1}{\sqrt{2\pi}} e^{-x^2/2} \, ,$$

the Cauchy distribution also has a density function expressible in closed form as

$$c(x) = \frac{1}{\pi(1+x^2)} \, .$$

But what is really important to us is not so much that the density function of $\mathcal{D}_p$ be expressible in closed form, but that it be easy to generate random samples drawn from $\mathcal{D}_p$. The Chambers-Mallows-Stuck method [CMS76] gives us the following simple algorithm. Let

$$X = \frac{\sin(p\theta)}{(\cos\theta)^{1/p}} \left(\frac{\cos((1-p)\theta)}{\ln(1/r)}\right)^{(1-p)/p} \, ,$$

where $(\theta, r) \in_R [-\pi/2, \pi/2] \times [0,1]$. Then the distribution of X is p-stable.

Replacing $\mathcal{N}(0,1)$ with $\mathcal{D}_p$ in the above pseudocode, where $\mathcal{D}_p$ is p-stable, allows us to generate a random variable distributed according to $\mathcal{D}_p$ "scaled" by $\|\boldsymbol{f}\|_p$. Note that the scaling factor $\|\boldsymbol{f}\|_p$ is the quantity we want to estimate. To estimate it, we shall simply take the median of a number of samples from the scaled distribution, i.e., we shall maintain a sketch consisting of several copies of the basic sketch and output the median of (the absolute values of) the entries of the sketch vector. Here is our final "idealized" sketch.

[1] The proof of this fact is a nice exercise in calculus.

Algorithm 13 Indyk's sketch for ℓ_p estimation

```
Initialize:
1: $M[1\ldots t][1\ldots n] \leftarrow tn$ independent samples from $\mathscr{D}_p$, where $t = O(\varepsilon^{-2}\log(\delta^{-1}))$
2: $z[1\ldots t] \leftarrow \vec{0}$

Process (token $(j,c)$) :
3: for $i = 1$ to $t$ do
4:     $z[i] \leftarrow z[i] + cM[i][j]$

Output: $\text{median}_{1\le i\le t} |z_i| \,/\, \text{median}(|\mathscr{D}_p|)$
```

8.3 The Median of a Distribution and its Estimation

To analyze this algorithm, we need the concept of the median of a probability distribution over the reals. Let $\mathscr{D}$ be an absolutely continuous distribution, let ϕ be its density function, and let $X \sim \mathscr{D}$. A median of $\mathscr{D}$ is a real number μ that satisfies

$$\frac{1}{2} = \mathbb{P}\{X \le \mu\} = \int_{-\infty}^{\mu} \phi(x)\,dx\,.$$

The distributions that we are concerned with here are nice enough to have uniquely defined medians; we will simply speak of *the* median of a distribution. For such a distribution $\mathscr{D}$, we will denote this unique median as $\text{median}(\mathscr{D})$.

For a distribution $\mathscr{D}$, with density function ϕ, we denote by $|\mathscr{D}|$ the distribution of the absolute value of a random variable drawn from $\mathscr{D}$. It is easy to show that the density function of $|\mathscr{D}|$ is ψ, where

$$\psi(x) = \begin{cases} 2\phi(x)\,, & \text{if } x \ge 0 \\ 0 & \text{if } x < 0\,. \end{cases}$$

For $p \in (0,2]$ and $c \in \mathbb{R}$, let $\phi_{p,c}$ denote the density function of the distribution of $c|X|$, where $X \sim \mathscr{D}_p$, and let $\mu_{p,c}$ denote the median of this distribution. Note that

$$\phi_{p,c}(x) = \frac{1}{c}\phi_{p,1}\left(\frac{x}{c}\right)\,, \quad \text{and} \quad \mu_{p,c} = c\mu_{p,1}\,.$$

Let Z_i denote the final value of z_i after Algorithm 13 has processed σ. By the earlier discussion, and the defintion of p-stability, we see that $Z_i \equiv \|\boldsymbol{f}\|_p Z$, where $Z \sim \mathscr{D}_p$. Therefore, $|Z_i|/\,\text{median}(|\mathscr{D}_p|)$ has a distribution whose density function is $\phi_{p,\lambda}$, where $\lambda = \|\boldsymbol{f}\|_p/\,\text{median}(|\mathscr{D}_p|) = \|\boldsymbol{f}\|_p/\mu_{p,1}$. Thus, the median of this distribution is $\mu_{p,\lambda} = \lambda\mu_{p,1} = \|\boldsymbol{f}\|_p$.

The algorithm—which seeks to estimate $\|\boldsymbol{f}\|_p$—can thus be seen as attempting to estimate the median of an appropriate distribution by drawing $t = O(\varepsilon^{-2}\log\delta^{-1})$ samples from it and outputting the *sample median*. We now show that this does give a fairly accurate estimate.

8.4 The Accuracy of the Estimate

Lemma 8.4.1. *Let $\varepsilon > 0$, and let $\mathscr{D}$ be a distribution over $\mathbb{R}$ with density function ϕ, and with a unique median $\mu > 0$. Suppose that ϕ is absolutely continuous on $[(1-\varepsilon)\mu, (1+\varepsilon)\mu]$ and let $\phi_* = \min\{\phi(z) : z \in [(1-\varepsilon)\mu, (1+\varepsilon)\mu]\}$. Let $Y = \text{median}_{1\le i\le t} Z_i$, where $Z_1,\ldots,Z_t$ are independent samples from $\mathscr{D}$. Then*

$$\mathbb{P}\{|Y-\mu| \ge \varepsilon\mu\} \le 2\exp\left(-\frac{2}{3}\varepsilon^2\mu^2\phi_*^2 t\right)\,.$$

Proof. We bound $\mathbb{P}\{Y < (1-\varepsilon)\mu\}$ from above. A similar argument bounds $\mathbb{P}\{Y > (1+\varepsilon)\mu\}$ and to complete the proof we just add the two bounds.

Let $\Phi(y) = \int_{-\infty}^{y} \phi(z)\,dz$ be the cumulative distribution function of $\mathscr{D}$. Then, for each $i \in [t]$, we have

$$\begin{aligned}\mathbb{P}\{Z_i < (1-\varepsilon)\mu\} &= \int_{-\infty}^{\mu} \phi(z)\,dz - \int_{(1-\varepsilon)\mu}^{\mu} \phi(z)\,dz \\ &= \frac{1}{2} - \Phi(\mu) + \Phi((1-\varepsilon)\mu) \\ &= \frac{1}{2} - \varepsilon\mu\phi(\xi)\,,\end{aligned}$$

for some $\xi \in [(1-\varepsilon)\mu, \mu]$, where the last step uses the mean value theorem and the fundamental theorem of calculus: $\Phi' = \phi$. Let α be defined by

$$\left(\frac{1}{2} - \varepsilon\mu\phi(\xi)\right)(1+\alpha) = \frac{1}{2}\,. \tag{8.1}$$

Let $N = |\{i \in [t] : Z_i < (1-\varepsilon)\mu\}|$. By linearity of expectation, we have $\mathbb{E}N = (\frac{1}{2} - \varepsilon\mu\phi(\xi))t$. If the sample median, Y, falls below a limit λ, then at least half the Z_is must fall below λ. Therefore

$$\mathbb{P}\{Y < (1-\varepsilon)\mu\} \le \mathbb{P}\{N \ge t/2\} = \mathbb{P}\{N \ge (1+\alpha)\mathbb{E}N\} \le \exp(-\mathbb{E}N\alpha^2/3)\,,$$

by a standard Chernoff bound. Now, from (8.1), we derive $\mathbb{E}N\alpha = \varepsilon\mu\phi(\xi)t$ and $\alpha \ge 2\varepsilon\mu\phi(\xi)$. Therefore

$$\mathbb{P}\{Y < (1-\varepsilon)\mu\} \le \exp\left(-\frac{2}{3}\varepsilon^2\mu^2\phi(\xi)^2 t\right) \le \exp\left(-\frac{2}{3}\varepsilon^2\mu^2\phi_*^2 t\right)\,. \qquad \square$$

To apply the above lemma to our situation we need an estimate for ϕ_*. We will be using the lemma with $\phi = \phi_{p,\lambda}$ and $\mu = \mu_{p,\lambda} = \|\boldsymbol{f}\|_p$, where $\lambda = \|\boldsymbol{f}\|_p / \mu_{p,1}$. Therefore,

$$\begin{aligned}\mu\phi_* &= \mu_{p,\lambda} \cdot \min\{\phi_{p,\lambda}(z) : z \in [(1-\varepsilon)\mu_{p,\lambda}, (1+\varepsilon)\mu_{p,\lambda}]\} \\ &= \lambda\mu_{p,1} \cdot \min\left\{\frac{1}{\lambda}\phi_{p,1}\left(\frac{z}{\lambda}\right) : z \in [(1-\varepsilon)\lambda\mu_{p,1}, (1+\varepsilon)\lambda\mu_{p,1}]\right\} \\ &= \mu_{p,1} \cdot \min\{\phi_{p,1}(y) : y \in [(1-\varepsilon)\mu_{p,1}, (1+\varepsilon)\mu_{p,1}]\}\,,\end{aligned}$$

which is a constant depending only on p: call it c_p. Thus, by Lemma 8.4.1, the output Y of the algorithm satisfies

$$\mathbb{P}\left\{\left|Y - \|\boldsymbol{f}\|_p\right| \ge \varepsilon\|\boldsymbol{f}\|_p\right\} \le \exp\left(-\frac{2}{3}\varepsilon^2 c_p^2 t\right) \le \delta\,,$$

for the setting $t = (3/(2c_p^2))\varepsilon^{-2}\log(\delta^{-1})$.

8.5 Annoying Technical Details

There are two glaring issues with the "idealized" sketch we have just discussed and proven correct. As stated, we do not have a proper algorithm to implement the sketch, because

- the sketch uses real numbers, and algorithms can only do bounded-precision arithmetic; and
- the sketch depends on a huge matrix — with n columns — that does not have a convenient implicit representation.

We will not go into the details of how these matters are resolved, but here is an outline.

We can approximate all real numbers involved by rational numbers with sufficient precision, while affecting the output by only a small amount. The number of bits required per entry of the matrix M is only logarithmic in n, $1/\varepsilon$ and $1/\delta$.

We can avoid storing the matrix M explicitly by using a pseudorandom generator (PRG) designed to work with space-bounded algorithms. One such generator is given by a theorem of Nisan [Nis90]. Upon reading an update to a token j, we use the PRG (seeded with j plus the initial random seed) to generate the jth column of M. This transformation blows up the space usage by a factor logarithmic in n and adds $1/n$ to the error probability.

Unit 9

Sparse Recovery

We have seen several algorithms that maintain a small linear sketch of an evolving vector $\boldsymbol{f}$ (that undergoes turnstile updates) and then answer questions about $\boldsymbol{f}$ using this sketch alone. When $\boldsymbol{f}$ is arbitrary, such sketches must necessarily throw away most of the information in $\boldsymbol{f}$ due to size constraints. However, when $\boldsymbol{f}$ itself holds only a small amount of information—for instance, by having at most s many nonzero coordinates—it is reasonable to ask whether such a linear sketch can allow a complete reconstruction of $\boldsymbol{f}$ on demand. This is the problem of sparse recovery.

Sparse recovery is an important area of research first studied extensively in the world of signal processing (especially for the problem of image acquisition), where it also goes by such names as compressed sensing and compressive sensing. It has many connections to topics throughout computer science and mathematics [GI10]. We will focus on a clean version of the problem, rather than the most general version.

9.1 The Problem

We are in the turnstile streaming model. Stream tokens are of the form (j,c), where $j \in [n]$ and $c \in \mathbb{Z}$ and have the usual meaning "$f_j \leftarrow f_j + c$." The net result is to build a frequency vector $\boldsymbol{f} \in \mathbb{Z}^n$. We will need a bound on how large the individual entries of $\boldsymbol{f}$ can get: we assume that $\|\boldsymbol{f}\|_\infty := \max_{j\in[n]} |f_j| \le M$ at all times. One often assumes that $M = \operatorname{poly}(n)$, so that $\log M = \Theta(\log n)$.

The *support* of a vector $\boldsymbol{v} = (v_1, \dots, v_n)$ is defined to be the set of indices where the vector is nonzero:

$$\operatorname{supp} \boldsymbol{v} = \{ j \in [n] : v_j \neq 0 \} .$$

We say that $\boldsymbol{v}$ *is s-sparse* if $|\operatorname{supp} \boldsymbol{v}| \le s$, and the cardinality $|\operatorname{supp} \boldsymbol{v}|$ is called the *sparsity* of $\boldsymbol{v}$.

The *streaming s-sparse recovery problem* is to maintain a sketch $\boldsymbol{y}$ of the vector $\boldsymbol{f}$, as the stream comes in, so that if $\boldsymbol{f}$ ends up being s-sparse, it can be recovered from $\boldsymbol{y}$, and otherwise we can detect the non-sparsity. Think $s \ll n$. We would like our sketch size to be "not much bigger" than s, with the dependence on the ambient dimension n being only polylogarithmic.

9.2 Special case: 1-sparse recovery

An important special case is $s = 1$, i.e., the problem of 1-sparse recovery. Let us first consider a *promise* version of the problem, where it is guaranteed that the final frequency vector $\boldsymbol{f}$ resulting from the stream is indeed 1-sparse.

For this, consider the following very simple deterministic algorithm: maintain the population size ℓ (the net number of tokens seen) and the frequency-weighted sum z of all token *values* seen. In other words,

$$\ell = \sum_{j=1}^{n} f_j\,; \quad z = \sum_{j=1}^{n} j f_j\,.$$

Since $\boldsymbol{f}$ is 1-sparse, $\boldsymbol{f} = \lambda \boldsymbol{e}_i$ for some $i \in [n]$ and $\lambda \in \mathbb{Z}$, where $\boldsymbol{e}_i$ is the standard basis vector that has a 1 entry at index i and 0 entries elsewhere. Therefore, $\ell = \lambda$ and $z = i f_i = i\lambda = i\ell$. It follows that we can recover $\boldsymbol{f}$ using

$$\boldsymbol{f} = \begin{cases} \boldsymbol{0}\,, & \text{if } \ell = 0\,, \\ \ell \boldsymbol{e}_{z/\ell}\,, & \text{otherwise.} \end{cases}$$

Of course, in the absence of the promise, the above technique cannot detect whether $\boldsymbol{f}$ is indeed 1-sparse.

To solve the more general problem of 1-*sparse detection and recovery*, we employ the important idea of *fingerprinting*. Informally speaking, a fingerprint is a randomized mapping from a large object to a much smaller sketch with the property that two distinct large objects will very likely have distinct sketches, just as two distinct humans will very likely have distinct fingerprints. In computer science, an often-used technique is to treat the large object as a polynomial and use a random evaluation of that polynomial as its fingerprint.

These ideas are incorporated into the algorithm below, which is parametrized by a finite field $\mathbb{F}$, the size of which affects the eventual error probability. As we shall see, a good choice for $\mathbb{F}$ is one with $n^3 < |\mathbb{F}| \le 2n^3$. We know that we can always find a finite field (in fact a prime field) with size in this range.

Algorithm 14 Streaming 1-sparse detection and recovery

```
Initialize:
 1: $(\ell, z, p) \leftarrow (0,0,0)$                          ▷ (population, sum, fingerprint)
 2: $r \leftarrow$ uniform random element of finite field $\mathbb{F}$    ▷ $|\mathbb{F}|$ controls the error probability

Process (token $(j, c)$) :
 3: $\ell \leftarrow \ell + c$
 4: $z \leftarrow z + c\,j$
 5: $p \leftarrow p + c\,r^j$

Output:
 6: if $\ell = z = p = 0$ then declare $\boldsymbol{f} = \boldsymbol{0}$              ▷ very likely correct
 7: else if $z/\ell \notin [n]$ then declare $\|\boldsymbol{f}\|_0 > 1$           ▷ definitely correct
 8: else if $p \ne \ell\, r^{z/\ell}$ then declare $\|\boldsymbol{f}\|_0 > 1$        ▷ definitely correct
 9: else
10:     declare $\boldsymbol{f} = \ell \boldsymbol{e}_{z/\ell}$                      ▷ very likely correct
```

9.3 Analysis: Correctness, Space, and Time

Let R be the random value of r picked in the initialization. The final values of ℓ, z, and p computed by Algorithm 14 are

$$\ell = \sum_{j=1}^{n} f_j\,, \tag{9.1}$$

$$z = \sum_{j=1}^{n} j f_j\,, \tag{9.2}$$

$$p = \sum_{j=1}^{n} f_j R^j = q(R)\,, \tag{9.3}$$

where $q(X) \in \mathbb{F}[X]$ is the polynomial $q(X) = \sum_{j=1}^{n} f_j X^j$. This polynomial $q(X)$ is naturally in 1-to-1 correspondence with the vector $\boldsymbol{f}$: coefficients of the polynomial correspond to entries in the vector.

Consider the case when $\boldsymbol{f}$ is indeed 1-sparse: say $\boldsymbol{f} = \lambda \boldsymbol{e}_i$. As before, we have $\ell = \lambda$ and $z = i\ell$. So, $z/\ell = i \in [n]$ and $p = f_i R^i = \ell R^{z/\ell}$. In the subcase $\boldsymbol{f} = \boldsymbol{0}$, this makes $p = 0$ and the algorithm gives a correct output, by the logic of line 6. In the subcase $\|\boldsymbol{f}\|_0 = 1$, the algorithm reaches line 10 and thus gives a correct output. Overall, we see that there are no false negatives.

Consider the case when $\boldsymbol{f}$ is not 1-sparse. Then, for certain "bad" choices of R, Algorithm 14 might err by giving a false positive. To be precise, an error happens when $\ell = z = p = 0$ in line 6 or z/ℓ happens to lie in $[n]$ and then $p = \ell R^{z/\ell}$ in line 8. These cases can be "cleverly" combined by defining

$$i = \begin{cases} 0, & \text{if } \ell = z = 0 \text{ or } z/\ell \notin [n], \\ z/\ell, & \text{otherwise.} \end{cases}$$

A false positive happens only if $q(R) - \ell R^i = 0$, i.e., only if R is a root of the polynomial $q(X) - \ell X^i$. This latter polynomial has degree at most n and is not the zero polynomial, since $q(X)$ has at least two nonzero coefficients. Recall the following basis theorem from algebra.

Theorem 9.3.1. *Over any field, a nonzero polynomial of degree d has at most d roots.* □

By Theorem 9.3.1, $q(X) - \ell X^i$ has at most n roots. Since R is drawn uniformly from $\mathbb{F}$,

$$\mathbb{P}\{\text{false positive}\} \le \frac{n}{|\mathbb{F}|} = O\left(\frac{1}{n^2}\right), \tag{9.4}$$

by choosing $\mathbb{F}$ such that $n^3 < |\mathbb{F}| \le 2n^3$.

For the space and time cost analysis, suppose that $|\mathbb{F}| = \Theta(n^3)$. Algorithm 14 maintains the values ℓ, z, p, and r. Recall that $\|\boldsymbol{f}\|_\infty \le M$ at all times. By eqs. (9.1) and (9.2), $|\ell| \le nM$ and $|z| \le n^2 M$; so ℓ and z take up at most $O(\log n + \log M)$ bits of space. Since p and r lie in $\mathbb{F}$, they take up at most $\lceil \log |\mathbb{F}| \rceil = O(\log n)$ bits of space. Overall, the space usage is $O(\log n + \log M)$, which is $\widetilde{O}(1)$ under the standard (and reasonable) assumption that $M = \text{poly}(n)$.

Assume, further, that the processor has a word size of $\Theta(\log n)$. Then, by using repeated squaring for the computation of r^j, the processing of each token takes $O(\log n)$ arithmetic operations and so does the post-processing to produce the output. We have thus proved the following result.

Theorem 9.3.2. *Given turnstile updates to a vector $\boldsymbol{f} \in \mathbb{Z}^n$ where $\|\boldsymbol{f}\|_\infty = \text{poly}(n)$, the 1-sparse detection and recovery problem for $\boldsymbol{f}$ can be solved with error probability $O(1/n)$ using a linear sketch of size $\widetilde{O}(1)$. The sketch can be updated in $\widetilde{O}(1)$ time per token and the final output can be computed in $\widetilde{O}(1)$ time from the sketch.* □

9.4 General Case: s-sparse Recovery

We turn to the general case. Let $s \ll n$ be a parameter. We would like to recover the vector $\boldsymbol{f}$ exactly, provided $\|\boldsymbol{f}\|_0 \le s$, and detect that $\|\boldsymbol{f}\|_0 > s$ if not. Our solution will make use of the above 1-sparse recovery sketch as a black-box data structure, denoted $\mathscr{D}$. When an instance of $\mathscr{D}$ produces an actual 1-sparse vector as output, let's say that it reports *positive*; otherwise, let's say that it reports *negative*. Remember that there are no false negatives and that a false positive occurs with probability $O(1/n)$.

The basic idea behind the algorithm is to hash the universe $[n]$ down to the range $[2s]$, thereby splitting the input stream into $2s$ sub-streams, each consisting of items that hash to a particular target. This range size is large enough to single out any particular item in $\operatorname{supp} \boldsymbol{f}$, causing the sub-stream containing that item to have a 1-sparse frequency vector. By repeating this idea some number of times, using independent hashes, we get the desired outcome. The precise logic is given in Algorithm 15 below.

Clearly, the space usage of this algorithm is dominated by the $2st$ copies of $\mathscr{D}$. Plugging in the value of t and our previous space bound on $\mathscr{D}$, the overall space usage is

$$O\left(s(\log s + \log \delta^{-1})(\log n + \log M)\right).$$

Algorithm 15 Streaming s-sparse recovery

```
Initialize:
 1: $t \leftarrow \lceil \log(s/\delta) \rceil$                         ▷ $\delta$ controls the failure probability
 2: $D[1\ldots t][1\ldots 2s] \leftarrow \vec{0}$                      ▷ each $D[i][k]$ is a copy of $\mathscr{D}$
 3: Choose independent hash functions $h_1,\ldots h_t : [n] \to [2s]$, each from a 2-universal family

Process (token $(j,c)$):
 4: foreach $i \in [t]$ do
 5:     update $D[i][h_i(j)]$ with token $(j,c)$

Output:
 6: $A \leftarrow$ (empty associative array)
 7: foreach $i \in [t]$, $k \in [2s]$ do                             ▷ query all copies of $\mathscr{D}$ and collate outputs
 8:     if $D[i][k]$ reports positive and outputs $\lambda \boldsymbol{e}_a$ where $\lambda \neq 0$ then
 9:         if $a \in \text{keys}(A)$ and $A[a] \neq \lambda$ then abort
10:         $A[a] \leftarrow \lambda$
11:         if $|\text{keys}(A)| > s$ then abort
12: declare $\boldsymbol{f} = \sum_{a \in \text{keys}(A)} A[a]\boldsymbol{e}_a$
```

Turning to the error analysis, suppose that $\boldsymbol{f}$ is indeed s-sparse and that $\operatorname{supp}\boldsymbol{f} \subseteq \{j_1,\ldots,j_s\}$. Notice that the data structure $D[i][k]$ sketches the sub-stream consisting of those tokens (j,c) for which $h_i(j) = k$. This sub-stream is 1-sparse if there is at most one $j \in \operatorname{supp}\boldsymbol{f}$ such that $h_i(j) = k$. Therefore, Algorithm 15 correctly outputs $\boldsymbol{f}$ if both of the following happen.

[SR$_1$] For each $j \in \operatorname{supp}\boldsymbol{f}$, there exists $i \in [t]$ such that $h_i^{-1}(h_i(j)) \cap \operatorname{supp}\boldsymbol{f} = \{j\}$.

[SR$_2$] None of the $2st$ copies of $\mathscr{D}$ gives a false positive.

Consider a particular item $j \in \operatorname{supp}\boldsymbol{f}$. For each $i \in [t]$,

$$\begin{aligned}\mathbb{P}\left\{h_i^{-1}(h_i(j)) \cap \operatorname{supp}\boldsymbol{f} \neq \{j\}\right\} &= \mathbb{P}\left\{\exists\, j' \in \operatorname{supp}\boldsymbol{f} :\ j' \neq j \wedge h_i(j') = h_i(j)\right\} \\ &\leq \sum_{\substack{j' \in \operatorname{supp}\boldsymbol{f} \\ j' \neq j}} \mathbb{P}\left\{h_i(j') = h_i(j)\right\} \\ &\leq \sum_{\substack{j' \in \operatorname{supp}\boldsymbol{f} \\ j' \neq j}} \frac{1}{2s} \leq \frac{1}{2},\end{aligned}$$

where the penultimate step uses the 2-universality property. By the mutual independence of the t hash functions,

$$\mathbb{P}\{\text{SR}_1 \text{ fails for item } j\} = \prod_{i=1}^{t} \mathbb{P}\left\{h_i^{-1}(h_i(j)) \cap \operatorname{supp}\boldsymbol{f} \neq \{j\}\right\} \leq \left(\frac{1}{2}\right)^t \leq \frac{\delta}{s}.$$

By a union bound over the items in $\operatorname{supp}\boldsymbol{f}$,

$$\mathbb{P}(\neg\,\text{SR}_1) \leq |\operatorname{supp}\boldsymbol{f}| \cdot \frac{\delta}{s} \leq \delta.$$

By another union bound over the copies of $\mathscr{D}$, using the false positive rate bound from eq. (9.4),

$$\mathbb{P}(\neg\,\text{SR}_2) \leq 2st \cdot O\left(\frac{1}{n^2}\right) = o(1),$$

since $s \leq n$. Overall, the probability that the recovery fails is at most $\delta + o(1)$.

Theorem 9.4.1. *Given turnstile updates to a vector $\boldsymbol{f} \in \mathbb{Z}^n$ where $\|\boldsymbol{f}\|_\infty = \mathrm{poly}(n)$, the s-sparse recovery problem for $\boldsymbol{f}$ can be solved with error probability δ using a linear sketch of size $O(s\log(s/\delta)\log n)$.* □

With just a little more work (and analysis), we can solve the s-sparse *detection* and recovery problem as well. This is left as an exercise.

Exercises

9-1 Suppose that in Algorithm 14, we modified line 6 to declare $\boldsymbol{f} = \mathbf{0}$ whenever $\ell = z = 0$, without using the value of p. Would this still be correct? Why?

9-2 As described and analyzed, Algorithm 15 requires the vector $\boldsymbol{f}$ to be s-sparse. Explain clearly why it does not already solve the problem of detecting whether this s-sparsity condition holds. Make a simple enhancement to the algorithm to enable this detection and analyze your modified algorithm to prove that it works.

Unit 10

Weight-Based Sampling

The problem of sampling a coordinate at random from a high-dimensional vector is important both in its own right and as a key primitive for other, more advanced, data stream algorithms [JW18]. For instance, consider a stream of visits by customers to the busy website of some business or organization. An analyst might want to sample one or a few customers according to some distribution defined by the frequencies of various customers' visits. Letting $\boldsymbol{f} \in \mathbb{Z}^n$ be the vector of frequencies (as usual), we would like to pick a coordinate $i \in [n]$ according to some distribution $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_n) \in [0,1]^n$. Here are three specific examples of distributions we may want to sample from.

- Sample a customer uniformly from the set of all distinct customers who visited the website. In our formalism, $\pi_i = 1/\|\boldsymbol{f}\|_0$ for each $i \in \operatorname{supp} \boldsymbol{f}$ and $\pi_i = 0$ for all other $i \in [n]$. This is called ℓ_0-sampling.
- Sample a customer with probability proportional to their visit frequency. In our formalism, $\pi_i = |f_i|/\|\boldsymbol{f}\|_1$. This is called ℓ_1-sampling.
- Sample a customer with frequent visitor weighted disproportionately highly, for instance, proportional to the square of their visit frequency. In our formalism, $\pi_i = f_i^2/\|\boldsymbol{f}\|_2^2$. More generally, one can consider $\pi_i = |f_i|^p/\|\boldsymbol{f}\|_p^p$, for a parameter $p > 0$. This is called ℓ_p-sampling.

10.1 The Problem

We shall study an algorithm for *approximate* ℓ_0 sampling and then one for *approximate* ℓ_2-sampling: these algorithms are "approximate" because the distribution over items (coordinates) that they induce are close to, but not identical to, the respective target distributions.

To be precise, the task is as follows. The input is a turnstile stream σ that builds up a frequency vector $\boldsymbol{f} \in \mathbb{Z}^n$. Assume that at all times we have $\|\boldsymbol{f}\|_\infty \le M$, for some bound $M = \operatorname{poly}(n)$. We must maintain a small-sized linear sketch of $\boldsymbol{f}$ from which we can produce a random index $i \in [n]$ distributed approximately according to $\boldsymbol{\pi}$, where $\boldsymbol{\pi}$ is one of the distributions listed above. With small probability, we may output FAIL instead of producing an index in $[n]$. If the output I of the algorithm satisfies

$$\mathbb{P}\{I = i\} \in [(1-\varepsilon)\pi_i - \theta, (1+\varepsilon)\pi_i + \theta], \text{ for each } i \in [n], \tag{10.1}$$
$$\mathbb{P}\{I = \text{FAIL}\} \le \delta, \tag{10.2}$$

where $\boldsymbol{\pi}$ is the distribution corresponding to ℓ_p-sampling, then we say that the algorithm is an $(\varepsilon, \theta, \delta)$-$\ell_p$-sampler.

Monemizadeh and Woodruff [MW10] gave sampling algorithms that exploited this relaxation of producing the distribution $\boldsymbol{\pi}$ only approximately. More recently, Jayaram and Woodruff [JW18] showed that space-efficient *perfect* sampling—where there is no approximation, but a small failure probability is allowed—is possible.

10.2 The ℓ_0-Sampling Problem

Our task is to design an $(\varepsilon, \theta, \delta)$-$\ell_0$-sampler as defined in eqs. (10.1) and (10.2), with $\boldsymbol{\pi}$ defined by

$$\begin{aligned} \pi_i &= \frac{1}{\|\boldsymbol{f}\|_0}\,, && \text{if } i \in \operatorname{supp} \boldsymbol{f}\,, \\ \pi_i &= 0\,, && \text{otherwise.} \end{aligned}$$

It is promised that $\boldsymbol{f} \neq \mathbf{0}$, so that this problem is well-defined.

A key observation is that, thanks to the sparse recovery sketches designed in Unit 9, this problem becomes simple if $\boldsymbol{f}$ is promised to be s-sparse, for a small s. This leads to the following idea: choose a random subset $S \subseteq [n]$ of coordinates so that if we form a sub-vector of $\boldsymbol{f}$ by zeroing out all coordinates outside S, the sub-vector is likely to be very sparse. If we filter the input stream $\boldsymbol{\sigma}$ and only retain tokens that update items in S, we can simulate the stream that builds up this sub-vector and feed this simulated stream to a sparse recovery data structure.

Let $d := \|\boldsymbol{f}\|_0$. If S is formed by picking each coordinate in $[n]$ with probability about $1/d$, independently, then the sub-vector described above has a good—i.e., $\Omega(1)$—chance of being 1-sparse. Of course, we don't know d in advance, so we have to prepare for several possibilities by using several sets S, using several different coordinate retention probabilities.

10.2.1 An Idealized Algorithm

It will be useful to assume, without loss of generality, that n is a power of 2.

We realize the idea outlined above by using hash functions to create the random sets: the set of items in $[n]$ that are hashed to a particular location form one of the sets S mentioned above. We begin with an idealized algorithm that uses *uniformly random* hash functions on the domain $[n]$. Of course, storing such a hash function requires $\Omega(n)$ bits, so this does not achieve sublinear storage. We can address this issue in one of the following ways.

- We can apply Nisan's space-bounded PRG to generate these hashes pseudorandomly, in limited space.
- We can use k-universal hash families, for a suitable not-too-large k, which is more space efficient.

Either of these approaches will introduce a small additional error in the algorithm's output.

The algorithm below uses several copies of a 1-sparse detection and recovery data structure, $\mathscr{D}$. We can plug in the data structure developed in Unit 9, which uses $O(\log n + \log M)$ space and has a false positive probability in $O(1/n^2)$.

Algorithm 16 Idealized ℓ_0-sampling algorithm using full randomness

```
Initialize:
 1: for $\ell \leftarrow 0$ to $\log n$ do
 2:     choose $h_\ell : [n] \to \{0,1\}^\ell$ uniformly at random from all such functions
 3:     $\mathscr{D}_\ell \leftarrow \vec{0}$                    ▷ 1-sparse detection and recovery data structure

Process (token $(j,c)$):
 4: for $\ell \leftarrow 0$ to $\log n$ do
 5:     if $h_\ell(j) = \mathbf{0}$ then                    ▷ happens with probability $2^{-\ell}$
 6:         feed $(j,c)$ to $\mathscr{D}_\ell$

Output:
 7: for $\ell \leftarrow 0$ to $\log n$ do
 8:     if $\mathscr{D}_\ell$ reports positive and outputs $\lambda \boldsymbol{e}_i$ with $\lambda \neq 0$ then
 9:         yield $(i, \lambda)$ and stop
10: output FAIL
```

10.2.2 The Quality of the Output

Let $d = |\operatorname{supp}(\boldsymbol{f})|$. Consider level ℓ such that $\frac{1}{4d} \le \frac{1}{2^\ell} < \frac{1}{2d}$.

$$\begin{aligned}
\mathbb{P}\{\text{vector fed to } D_\ell \text{ is 1-sparse}\} &= \Pr\left[\bigvee_{j\in\operatorname{supp}(\boldsymbol{f})}\left((h_\ell(j)=\mathbf{0}) \wedge \bigwedge_{i\in\operatorname{supp}(\boldsymbol{f}):i\neq j}(h_\ell(i)\neq\mathbf{0})\right)\right] \\
&= \sum_{j\in\operatorname{supp}(\boldsymbol{f})}\Pr\left[\left((h_\ell(j)=\mathbf{0}) \wedge \bigwedge_{i\in\operatorname{supp}(\boldsymbol{f}):i\neq j}(h_\ell(i)\neq\mathbf{0})\right)\right] \\
&= \sum_{j\in\operatorname{supp}(\boldsymbol{f})}\mathbb{P}\{h_\ell(j)=\mathbf{0}\}\Pr\left[\bigwedge_{i\in\operatorname{supp}(\boldsymbol{f}):i\neq j}(h_\ell(i)\neq\mathbf{0})\,\middle|\,(h_\ell(j)=\mathbf{0})\right] \\
&= \sum_{j\in\operatorname{supp}(\boldsymbol{f})}\frac{1}{2^\ell}\left(1-\Pr\left[\bigvee_{i\in\operatorname{supp}(\boldsymbol{f}):i\neq j}(h_\ell(i)=\mathbf{0})\,\middle|\,(h_\ell(j)=\mathbf{0})\right]\right) \\
&\ge \sum_{j\in\operatorname{supp}(\boldsymbol{f})}\frac{1}{2^\ell}\left(1-\sum_{i\in\operatorname{supp}(\boldsymbol{f}):i\neq j}\mathbb{P}\{(h_\ell(i)=\mathbf{0})\mid(h_\ell(j)=\mathbf{0})\}\right) \\
&= \frac{d}{2^\ell}\left(1-\frac{d-1}{2^\ell}\right) \ge \frac{d}{2^\ell}\left(1-\frac{d}{2^\ell}\right) \ge \frac{1}{4}\left(1-\frac{1}{2}\right) = \frac{1}{8}.
\end{aligned}$$

To achieve error probability of $O(\delta)$, the number of 1-sparse detection and recovery data structures to be used is $O(\log n \log \frac{1}{\delta})$, each of which fails with probability $O(\frac{1}{n})$. So

$$\mathbb{P}\{\text{some data structure fails}\} \le O\left(\frac{\log n \log \frac{1}{\delta}}{n}\right).$$

The rest of the notes for this unit are VERY rough and I am not yet decided on how much of this material I will actually teach. When they have been polished/finalized, this notice will be removed.

A practical algorithm.

Algorithm 17 ℓ_0-sampling

```
Initialize:
1: $h_\ell : [n] \to \{0,1\}^\ell$ for $\ell \in \{0,1,\ldots,\log n\}$        ▷ 2-universal, independent
2: $D_0, D_1, \ldots, D_{\log n}$        ▷ 1-sparse detection and recovery data structures

Process (token $(j,c)$):
3: for $0 \le \ell \le \log n$ do
4:     if $h_\ell(j) = \mathbf{0}$ then
5:         feed $(j,c)$ to $D_\ell$

Output:
6: for $0 \le \ell \le \log n$ do
7:     if $D_\ell$ reports positive and outputs $\lambda \boldsymbol{e}_i$ then
8:         yield $(i,\lambda)$ and stop
9: output "FAIL"
```

To argue that the element returned by the algorithm is indeed uniformly random in the support of $\boldsymbol{f}$, we need to use $O(\log n)$-universal hash families in the algorithm. A $O(\log n)$-universal hash family is also minwise independent.

A hash family $\mathscr{H}$ of functions $h : [n] \to [n]$ is minwise independent if $\forall X \subseteq [n] \forall x \in [n] \setminus X$ we have

$$\Pr_{h \in_R \in \mathscr{H}} \left[h(x) < \min_{y \in X} h(y) \right] = \frac{1}{|X|+1} .$$

10.3 ℓ_2 Sampling

Given a turnstile stream σ generating a frequency vector $\boldsymbol{f} = (f_1, f_2, \ldots, f_n)$, we seek to output $(J, \hat{f_j}^2)$ such that $\Pr[J = j] \approx \frac{f_j^2}{\|\boldsymbol{f}\|^2}$ and $\hat{f_j}^2$ is an approximation for f_j^2. The algorithm that we discuss here is knows as " precision sampling algorithm", which was introduced by Andoni, Krauthgamer, and Onak. In this algorithm we use count sketch data structure to estimate the "modified" frequency of elements in $[n]$. For each element $j \in [n]$, we define its "modified" frequency $g_j = \frac{f_j}{\sqrt{u_j}}$ where u_j is a random number chosen uniformly from the interval $[\frac{1}{n^2}, 1]$. Assume $\varepsilon > 0$ be an accuracy parameter for the algorithm (ε should be thought of as very small).

10.3.1 An ℓ_2-sampling Algorithm

Algorithm 18 ℓ_2-sampling or precision sampling

```
 Initialize:
 1: $u_1, u_2, \ldots, u_n \in_R [\frac{1}{n^2}, 1]$.
 2: Count sketch data structure $D$ for $\mathbf{g} = (g_1, g_2, \ldots, g_n)$.
 Process $(j, c)$:
 3: feed $(j, \frac{c}{\sqrt{u_j}})$ into $D$.
 Output:
 4: for each $j \in [n]$ do
 5:     $\hat{g}_j \leftarrow$ estimate of $g_j$ from $D$.
 6:     $\hat{f}_j \leftarrow \hat{g}_j \sqrt{u_j}$.
 7:     $X_j \leftarrow \begin{cases} 1 & \text{if } \hat{g_j}^2 = \frac{\hat{f_j}^2}{u_j} \geq \frac{4}{\varepsilon}, \\ 0 & \text{otherwise}. \end{cases}$
 8: if $\exists$ unique $j$ with $X_j = 1$ then
 9:     output $(j, \hat{f_j}^2)$.
10: else
11:     FAIL.
```

Remarks:

- As written we need to store $\mathbf{u} = \{u_1, u_2, \ldots, u_n\}$.
- But in fact $\mathbf{u}$ is just random and input independent, and the procedure is space-bounded. So we can use Nisan's PRG to reduce random seed.
- As written $\Pr[FAIL] \approx 1 - \varepsilon$, but just repeat $\Theta(\frac{1}{\varepsilon} \log \frac{1}{\delta})$ times for $\Pr[Fail] \leq \delta$.
- As written, algorithm 18 will work as analysed, assuming $1 \leq F_2 \leq 2$.

10.3.2 Analysis:

Let $F_2 = \sum_{i=1}^n f_j^2 = \|f\|_2^2$ and $F_2(\boldsymbol{g}) = \sum_{i=1}^n g_i^2$.

Claim 1. $\mathbb{E}\ F_2(\boldsymbol{g}) \le 5\log n$.

Proof.

$$\begin{aligned}
\mathbb{E}\ F_2(\boldsymbol{g}) &= \sum_{j=1}^{n} \mathbb{E}\ {g_j}^2 \\
&= \sum_{j=1}^{n} \mathbb{E}\ \frac{{f_j}^2}{u_j} \\
&= \sum_{j=1}^{n} {f_j}^2\, \mathbb{E}\ \frac{1}{u_j} \\
&= F_2 \int_{1/n^2}^{1} \frac{1}{u} du \cdot \frac{1}{1-1/n^2} && \text{since } \mathbb{E}\ u_i = \mathbb{E}\ u_j \forall i,j, \\
&= F_2 \frac{\ln n^2}{1-1/n^2} \\
&\le (4+\Theta(\frac{1}{n^2}))\cdot \ln n \\
&\le 5\log n\,.
\end{aligned}$$

□

Consider estimate $\hat{g}_j$ derived from a $1\times k$ count sketch data structure: $\hat{g}_j = g_j + Z_j$ where Z_j is the sum of contribution from $i \ne j$ that collide with j. Recall that $\mathbb{E}\ Z_j = 0$, $\mathbb{E}\ Z_j^2 = \frac{1}{k}\sum_{i\ne j} g_I^2 \le \frac{F_2(\boldsymbol{g})}{k}$. Hence, by applying Markov inequality, we get $\Pr\left[Z_j^2 \ge \frac{3F_2(\boldsymbol{g})}{k}\right] \le \frac{1}{3}$. Now consider the following two cases:

- If $|g_j| \ge \frac{2}{\varepsilon}$, then ${\hat{g}_j}^2 = |g_j + Z_j|^2 = e^{\pm\varepsilon} g_j^2$.
- Else $|g_j| < \frac{2}{\varepsilon}$. Then,

$$\begin{aligned}
|{\hat{g}_j}^2 - g_j^2| &\le (|g_j| + |Z_j|)^2 - |g_j|^2\,, \\
&= |Z_j|^2 + 2|g_j Z_j|\,, \\
&\le Z_j^2(1+\frac{4}{\varepsilon})\,.
\end{aligned}$$

So with probability at least $\frac{2}{3}$,

$$\begin{aligned}
|{\hat{g}_j}^2 - g_j^2| &\le \frac{3F_2(\boldsymbol{g})}{k}\cdot\frac{\varepsilon+4}{\varepsilon} && \text{since } \Pr\left[Z_j^2 \ge \frac{3F_2(\boldsymbol{g})}{k}\right] \le \frac{1}{3}\,, \\
&\le \frac{13F_2(\boldsymbol{g})}{\varepsilon k}\,.
\end{aligned}$$

We pick $k = \frac{650\log n}{\varepsilon} = \Theta(\frac{\log n}{\varepsilon})$. Using Claim 1, we get $\Pr[F_2(\boldsymbol{g}) \ge 50\log n] \le \frac{5\log n}{50\log n} = \frac{1}{10}$. Hence with probability $1-(1/3+1/10)$,

$$|{\hat{g}_j}^2 - g_j^2| \le 1\,.$$

So in both the cases, ${\hat{g}_j}^2 = e^{\pm\varepsilon} g_j^2 \pm 1$, $\Rightarrow {\hat{f}_j}^2 = e^{\pm\varepsilon} f_j^2 \pm u_j$. We observe that when $X_j = 1$, $u_j \le \frac{\varepsilon {\hat{f}_j}^2}{4}$. So ${\hat{f}_j}^2 = e^{\pm\varepsilon} f_j^2 \pm \frac{\varepsilon {\hat{f}_j}^2}{4} = e^{\pm\varepsilon} f_j^2 \pm \varepsilon {\hat{f}_j}^2$. Using the fact that $1+\varepsilon \approx e^{\varepsilon}$ for $\varepsilon \approx 0$, we get after rearranging, ${\hat{f}_j}^2 = e^{\pm 2\varepsilon} f_j^2$.

Now we are ready to lower bound the probability that algorithm 18 produces an output:

$$\begin{aligned}
\Pr[j \text{ is output}] &= \Pr\left[X_j = 1 \bigwedge \left(\bigwedge_{i\neq j} X_i = 0\right)\right], \\
&= \Pr[X_j = 1]\cdot\left(1 - \Pr\left[\bigvee_{i\neq j}(X_i = 1)\middle| X_j = 0\right]\right), \\
&= \Pr[X_j = 1]\cdot\left(1 - \Pr\left[\bigvee_{i\neq j}(X_i = 1)\right]\right) \text{ assuming } X_i\text{s are pairwise independent}, \\
&\geq \Pr[X_j = 1]\cdot\left(1 - \sum_{i\neq j}\Pr[X_i = 1]\right) \text{ by union bound}, \\
&= \Pr\left[u_j \leq \frac{\varepsilon \hat{f_j}^2}{4}\right]\cdot\left(1 - \sum_{i\neq j}\Pr\left[u_i \leq \frac{\varepsilon \hat{f_i}^2}{4}\right]\right), \\
&\approx \frac{\varepsilon \hat{f_j}^2}{4}\cdot\left(1 - \sum_{i\neq j}\frac{\varepsilon \hat{f_i}^2}{4}\right) \text{ pretending } u_j \in_R [0,1]\ , \\
&\geq \frac{\varepsilon \hat{f_j}^2}{4}\cdot\left(1 - \sum_{i}\frac{\varepsilon \hat{f_i}^2}{4}\right), \\
&= \frac{\varepsilon e^{\pm 2\varepsilon} f_j^2}{4}\cdot\left(1 - \sum_{i}\frac{\varepsilon e^{2\varepsilon} f_i^2}{4}\right), \\
&\geq \frac{\varepsilon e^{-2\varepsilon} f_j^2}{4}\cdot\left(1 - \sum_{i}\frac{\varepsilon e^{2\varepsilon} f_i^2}{4}\right), \\
&= \frac{\varepsilon e^{-2\varepsilon} f_j^2}{4}\cdot\left(1 - \frac{\varepsilon e^{2\varepsilon}}{2}\right) \text{ using } F_2 \leq 2\ .
\end{aligned}$$

Now we upper bound the probability that algorithm 18 produces an output:

$$\begin{aligned}
\Pr[j \text{ is output}] &\leq \mathbb{P}\{X_j = 1\}, \\
&\approx \frac{\varepsilon \hat{f_j}^2}{4} \text{pretending } u_j \in_R [0,1]\ , \\
&\leq \frac{\varepsilon e^{2\varepsilon} f_j^2}{4}, \\
&= \frac{e^{2\varepsilon}}{4}\cdot \varepsilon f_j^2 .
\end{aligned}$$

Hence, $\mathbb{P}\{j \text{ is output}\} = \frac{1}{4}e^{\pm 3\varepsilon}\cdot \varepsilon f_j^2$. So success probability of algorithm 18 is given by:

$$\mathbb{P}\{\neg\text{FAIL}\} = \sum_j \Pr[j \text{ is output}] = \frac{1}{4}e^{\pm 3\varepsilon}\cdot \varepsilon F_2 .$$

Finally, the sampling probability of j is givem by:

$$\mathbb{P}\{\text{sample } j \mid \neg \text{ FAIL}\} = \frac{f_j^2 \cdot e^{\pm 6\varepsilon}}{F_2} .$$

Unit 11

Finding the Median

Finding the mean of a stream of numbers is trivial, requiring only logarithmic space. How about finding the median? There is a deterministic linear-*time* algorithm for finding the median of an in-memory *array* of numbers—already a nontrivial and beautiful result due to Blum et al. [BFP+73]—but it requires full access to the array and is not even close to being implementable on streamed input.

The seminal paper of Munro and Paterson [MP80] provides a streaming, sublinear-space solution, provided at least two passes are allowed. In fact, their solution smoothly trades off number of passes for space. Moreover, they prove a restricted lower bound showing that their pass/space tradeoff is essentially optimal for algorithms that operate in a certain way. More recent research has shown that this lower bound actually applies without restrictions, i.e. without assumptions about the algorithm. In hindsight, the Munro–Paterson paper should be recognized as the *first* serious paper on data stream algorithms.

11.1 The Problem

We consider a vanilla stream model. Our input is a stream of numbers $\sigma = \langle a_1, a_2, \ldots, a_m \rangle$, with each $a_i \in [n]$. Our goal is to output the median of these numbers. Recall that

$$\text{median}(\sigma) = \begin{cases} w_{\lceil m/2 \rceil}\,, & \text{if } m \text{ is odd,} \\ \dfrac{w_{m/2} + w_{m/2+1}}{2}\,, & \text{if } m \text{ is even,} \end{cases}$$

where $\boldsymbol{w} = (w_1, \ldots, w_m) := \text{sort}(\sigma)$ is the result of sorting σ in nondecreasing order. Thus, finding the median is essentially a special case of the more general goal of finding w_r, the rth element of $\text{sort}(\sigma)$, where $r \in [m]$ is a given rank. This generalization is called the SELECTION problem. We shall give a multi-pass streaming algorithm for SELECTION.

To keep our presentation simple, we shall assume that the numbers in σ are *distinct*. Dealing with duplicates is messy but not conceptually deep, so this assumption helps us focus on the main algorithmic insights.

11.2 Preliminaries for an Algorithm

In preparation for designing an algorithm for SELECTION, we now set up some terminology and prove a useful combinatorial lemma.

Definition 11.2.1. Let T be a set of (necessarily distinct) elements from a totally ordered universe $\mathscr{U}$ and let $x \in \mathscr{U}$. The *rank* of x with respect to T is defined as

$$\text{rank}(x; T) = |\{w \in T : w \le x\}|\,.$$

Notice that this definition does not require $x \in T$.

The algorithm is driven by an idea very close to that of a *coreset*, seen in many other data stream algorithms.[1] In computational geometry and adjacent fields, a set Q is called a coreset of set P if $Q \subseteq P$, and Q is a good proxy for P with respect to some cost measure. Often, elements of Q need to be given multiplicities or weights in order to preserve the cost measure. The precise definition can depend on the intended application. For our purposes, we shall define a notion of the "core" of a sequence, as follows.

For a sequence $\rho = (a_1, \ldots, a_m)$, define

$$\begin{aligned} \operatorname{evens}(\rho) &= \{a_2, a_4, \ldots\} = \{a_{2i} : 1 \le i \le m/2, i \in \mathbb{Z}\}, \\ \operatorname{left}(\rho) &= (a_1, \ldots, a_{\lceil m/2 \rceil}), \\ \operatorname{right}(\rho) &= (a_{\lceil m/2 \rceil + 1}, \ldots, a_m). \end{aligned}$$

Recall that $\rho \circ \tau$ denotes concatenation of sequences: ρ followed by τ.

Definition 11.2.2. The *i-core* $\mathscr{C}_i(\tau)$ of a sequence τ of distinct integers is defined recursively, as follows.

$$\begin{aligned} \mathscr{C}_0(\tau) &= \operatorname{sort}(\tau); \\ \mathscr{C}_{i+1}(\tau) &= \operatorname{sort}(\operatorname{evens}(\mathscr{C}_i(\operatorname{left}(\tau))) \cup \operatorname{evens}(\mathscr{C}_i(\operatorname{right}(\tau)))), \quad \text{for each } i \ge 0. \end{aligned}$$

The sequence $\mathscr{C}_i(\tau)$ is typically not a subsequence of τ, though its elements do come from τ.

Notice that if m, the length of τ, is divisible by 2^{i+1}, then the length of $\mathscr{C}_i(\tau)$ is exactly $m/2^i$. Suppose that τ is in ascending order. Then the "sort" operations in Definition 11.2.2 have no effect and $\mathscr{C}_i(\tau)$ is obtained by taking every 2^ith element of τ. Thus, in this special case, if a appears in $\mathscr{C}_i(\tau)$), we have

$$\operatorname{rank}(a; \tau) = 2^i \cdot \operatorname{rank}(a; \mathscr{C}_i(\tau)), \tag{11.1}$$

where by $\operatorname{rank}(a; \tau)$ we mean the rank of a with respect to the *set* of elements of τ.

The next (and crucial) lemma shows that an approximate version of eq. (11.1) holds in the general case.

Lemma 11.2.3. *Let τ be a sequence of length $2^i s$ consisting of distinct integers. Suppose that $\mathscr{C}_i(\tau) = (x_1, \ldots, x_s)$. Then, for all $j \in [s]$.*

$$2^i j \le \operatorname{rank}(x_j; \tau) \le 2^i (i + j - 1) + 1.$$

Proof. We proceed by induction on i. The base case, $i = 0$, is immediate: $\mathscr{C}_0(\tau)$ and τ have the same set of elements.

Suppose the lemma has been proved for some $i \ge 0$. Consider a sequence $\tau_1 \circ \tau_2$, of distinct integers, where each of τ_1, τ_2 has length $2^i s$. Suppose that $\mathscr{C}_{i+1}(\tau_1 \circ \tau_2) = (x_1, \ldots, x_s)$. By definition,

$$(x_1, \ldots, x_s) = \operatorname{sort}(\operatorname{evens}(A) \cup \operatorname{evens}(B)),$$

where $A = \mathscr{C}_i(\tau_1)$ and $B = \mathscr{C}_i(\tau_2)$. Consider a particular element x_j. It must appear in either $\operatorname{evens}(A)$ or $\operatorname{evens}(B)$, but not both. Suppose WLOG that it is the kth element of $\operatorname{evens}(A)$. Then the $j - k$ elements of $(x_1, \ldots, x_j)$ that are not in $\operatorname{evens}(A)$ must instead appear in $\operatorname{evens}(B)$. Let y be the largest of these elements, i.e., the $(j-k)$th element of $\operatorname{evens}(B)$. Let z be the $(j-k+1)$th element of $\operatorname{evens}(B)$. Then $y < x_j < z$.

By the induction hypothesis,

$$\begin{aligned} 2^i \cdot 2k &\le \operatorname{rank}(x_j; \tau_1) \le 2^i(i + 2k - 1) + 1; \\ 2^i(2j - 2k) &\le \operatorname{rank}(y; \tau_2) \le 2^i(i + 2j - 2k - 1) + 1; \\ 2^i(2j - 2k + 2) &\le \operatorname{rank}(z; \tau_2) \le 2^i(i + 2j - 2k + 1) + 1. \end{aligned}$$

Therefore,

$$\begin{aligned} &\operatorname{rank}(x_j; \tau) \ge \operatorname{rank}(x_j; \tau_1) + \operatorname{rank}(y; \tau_2) \ge 2^i(2k + 2j - 2k) = 2^{i+1} j, \quad \text{and} \\ &\operatorname{rank}(x_j; \tau) \le \operatorname{rank}(x_j; \tau_1) + \operatorname{rank}(z; \tau_2) - 1 \le 2^i(i + 2k - 1 + i + 2j - 2k + 1) + 2 - 1 = 2^{i+1}(i + j) + 1, \end{aligned}$$

which establishes the induction step. □

[1]The original paper uses the term "sample," which can be confusing because there is nothing random about the construction. The term "coreset" is much more modern, originates in computational geometry and is often given a precise meaning that doesn't quite fit the concept we are defining. Nevertheless, the strong similarities with the coresets of computational geometry are important to note.

11.3 The Munro–Paterson Algorithm

We are ready to describe an algorithm for SELECTION. This algorithm uses multiple passes (two or more) over the input stream. The computation inside each pass is interesting and elegant, though it is not so easy to describe in pseudocode. The post-processing logic after each pass is also nontrivial.

11.3.1 Computing a Core

Having introduced the important idea of a core, the next key insight is that Definition 11.2.2 actually allows for a small-space streaming computation of a core in a single pass.

Theorem 11.3.1. *There is a one-pass streaming algorithm that, given an input stream σ' of length $m' = 2^t s$ (for integers t and s) consisting of distinct tokens from $[n]$, computes the t-core $\mathscr{C}_t(\sigma')$ using $O(s \log(m'/s) \log n)$ bits of space.*

Proof sketch. Buffer each contiguous chunk of s elements of σ'. When the buffer fills up, send its contents up the recursion tree and flush the buffer. We will have at most one buffer worth of elements per level of recursion and there are $t+1 = O(\log(m'/s))$ levels. □

[* * * Insert picture of binary tree showing core computation recursion * * *]

11.3.2 Utilizing a Core

The Munro–Paterson algorithm uses p such passes and operates in space $O(m^{1/p} \log^{2-2/p} m \log n)$. Each pass has a corresponding *active interval*: a possibly infinite open interval (a^-, a^+). Tokens in the input stream that lie in the active interval are the *active tokens* and they form a substream of the input stream σ called the *active substream* for this pass. The pass computes a core sequence of this active substream τ, using the algorithm given by Theorem 11.3.1, and does some post-processing to update the active interval, maintaining the invariant

$$a^- < w_r < a^+ , \tag{11.2}$$

where w_r is the rank-r element of $\boldsymbol{w} := \text{sort}(\sigma)$ that we seek. Initially, we set $a^- = -\infty$ and $a^+ = +\infty$. During a pass, we ignore inactive tokens except to compute $q := \text{rank}(a^-; \sigma)$; clearly, w_r is the $(r-q)$th element of the active substream τ. We compute the t-core of τ, for a suitable integer t. Then, based on the approximate ranks of elements in this core (as given by Lemma 11.2.3), we compute a new active interval that is a subinterval of the one we started with.

The precise logic used in a pass and for determining when to stop making passes is given in pseudocode form in Algorithm 19. This precision comes at the cost of simplicity, so for a full understanding of the pseudocode, it may be necessary to study the analysis that follows.

Suppose that the invariant in eq. (11.2) indeed holds at the start of each pass. Examining Algorithm 19, it is clear that when the algorithm returns a final answer, it does so having stored all elements of the active substream, and therefore this answer is correct. Further, the logic of lines 18 and 20, together with the rank bounds in Lemma 11.2.3, goes to show that the invariant is correctly maintained.

11.3.3 Analysis: Pass/Space Tradeoff

It remains to analyze the number of passes over σ made by Algorithm 19.

Consider the ith pass over the stream (i.e., the ith call to SELECTIONPASS) and suppose that this isn't the final pass (i.e., the condition tested in line 10 fails). Suppose this pass computes a t_i-core of a stream consisting of the active substream τ_i—which has length m_i—padded up with some number of "$+\infty$" tokens so that the core computation happens on a stream of length $2^{t_i} s$. Let $\boldsymbol{x} = (x_1, \ldots, x_s)$ be the core. Suppose that the desired answer w_r is the r_ith element of the active substream, i.e., $r_i = r - \text{rank}(a^-; \sigma)$. Notice that r_i is computed as the value $r - q$ in Algorithm 19.

Algorithm 19 Munro–Paterson multi-pass algorithm for the SELECTION problem

```
 1: function SelectionPass(stream $\sigma$, rank $r$, space parameter $s$, active interval $a^-$, $a^+$)
 2:     initialize core computation data structure $\mathscr{C}$ with space parameter $s$
 3:     $(m, m', q) \leftarrow (0, 0, 0)$                    ▷ (stream-length, active-length, num-below-active)
 4:     foreach token $j$ from $\sigma$ do
 5:         $m \leftarrow m + 1$
 6:         if $j \le a^-$ then $q \leftarrow q + 1$          ▷ token too small: use it to update rank of $a^-$
 7:         else if $a^- < j < a^+$ then                  ▷ active token
 8:             $m' \leftarrow m' + 1$
 9:             feed $j$ to $\mathscr{C}$
10:     if $m' \le \lceil s \log m \rceil$ then
11:         $\rho \leftarrow$ output of $\mathscr{C}$                ▷ the entire active substream, sorted
12:         return $\rho[r - q]$                              ▷ answer found
13:     else
14:         $t \leftarrow \lceil \log(m'/s) \rceil$                   ▷ depth of core computed
15:         feed $(2^t s - m')$ copies of "$+\infty$" to $\mathscr{C}$   ▷ pad stream so number of chunks is power of 2
16:         $\rho \leftarrow$ output of $\mathscr{C}$
17:         $\ell \leftarrow \max\{j \in [s] : 2^t(t + j - 1) + 1 < r - q\}$
18:         if $\ell$ is defined then $a^- \leftarrow \rho[\ell]$    ▷ largest element whose max possible rank is too low
19:         $h \leftarrow \min\{j \in [s] : 2^t j > r - q\}$
20:         if $h$ is defined then $a^+ \leftarrow \rho[h]$    ▷ smallest element whose min possible rank is too high
21:         return $(a^-, a^+)$                         ▷ new active interval (answer not found yet)

22: function Select(stream $\sigma$, rank $r$, space-bound $s$)
23:     $(a^-, a^+, w) \leftarrow (-\infty, +\infty, \perp)$                ▷ (active interval, answer)
24:     while $w = \perp$ do
25:         $z \leftarrow$ SelectionPass($\sigma$, $r$, $s$, $a^-$, $a^+$)
26:         if $z$ is a pair then $(a^-, a^+) \leftarrow z$ else $w \leftarrow z$
27:     return $w$
```

Let $\ell_i = \max\{j \in [s] : 2^{t_i}(t_i + j - 1) + 1 < r_i\}$. After the pass, we set a^- to the ℓ_ith element of the core ρ, i.e., to x_{ℓ_i}. Suppose that we are in the typical case, when ℓ_i is defined and $1 \le \ell_i < s$. By the maximality of ℓ_i, the maximum possible rank of x_{ℓ_i+1} (as given by Lemma 11.2.3) is *not* too low, i.e.,

$$2^{t_i}(t_i + (\ell_i + 1) - 1) + 1 \ge r_i \,.$$

Applying the lower bound in Lemma 11.2.3, we obtain

$$\operatorname{rank}(x_{\ell_i};\sigma) \ge 2^{t_i}\ell_i \ge r_i - 2^{t_i}t_i - 1 \,.$$

Analogously, let $h_i = \min\{j \in [s] : 2^{t_i} j > r_i\}$. We set $a^+ = x_{h_i}$ after the pass. By the minimality of h_i,

$$2^{t_i}(h_i - 1) \le r_i \,.$$

Applying the upper bound in Lemma 11.2.3, we obtain

$$\operatorname{rank}(x_{h_i};\sigma) \le 2^{t_i}(t_i + h_i - 1) + 1 \le r_i + 2^{t_i}t_i + 1 \,.$$

Putting these together, the length m_{i+1} of the active substream for the next pass satisfies

$$\begin{aligned} m_{i+1} &\le \operatorname{rank}(x_{h_i};\sigma) - \operatorname{rank}(x_{\ell_i};\sigma) \\ &\le 2(2^{t_i}t_i + 1) \\ &= 2\left(2^{\lceil \log(m_i/s) \rceil} \left\lceil \log \frac{m_i}{s} \right\rceil + 1\right) \\ &\le \frac{5 m_i \log m}{s} \,, \end{aligned} \tag{11.3}$$

where the final step tacitly uses $m_i \gg s$.

Initially, we have $m_1 = m$ active tokens. Iterating eq. (11.3), we obtain

$$m_i = O\left(m\left(\frac{5\log m}{s}\right)^{i-1}\right).$$

By the logic of line 10, Algorithm 19 executes its final pass—the pth, say—when $m_p \le \lceil s\log m\rceil$. For this value of p, we must have $m((5\log m)/s)^{p-1} = \Theta(s\log m)$, i.e., $s = \Theta(m^{1/p}\log^{1-2/p} m)$.

If we want to operate within a fixed number p of passes, we could give an alternative presentation of the Munro–Paterson algorithm that computes the chunk size s from $m := \text{length}(\sigma)$ and p. For such a presentation, we would need to know m in advance. Suppose we then set $s = \Theta(m^{1/p}\log^{1-2/p} m)$, as we just derived. The space used in each pass is dominated by the space requirements of $\mathscr{C}$, the core computation data structure. By Theorem 11.3.1, this space is $O(s\log(m/s)\log n)$ in every pass except the last. By the logic of line 10, the space is $O(s\log m\log n)$ in the last pass. Putting these observations together gives us the following theorem.

Theorem 11.3.2. *For each fixed $p \ge 1$, the* SELECTION *problem, for a data stream consisting of m tokens from the universe $[n]$, admits a p-pass algorithm that uses $O(m^{1/p}\log^{2-2/p} m\log n) = \widetilde{O}(m^{1/p})$ bits of space.* □

Notice how this theorem trades off passes for space.

Strictly speaking, we have proved the above theorem only for streams whose tokens are distinct. If you wish to achieve a thorough understanding, you may want to extend the above algorithm to the general case.

Exercises

11-1 Give a detailed proof of Theorem 11.3.1, complete with pseudocode for the recursive algorithm described in the proof sketch.

11-2 We have stated Theorem 11.3.2 only for fixed p. Go through the analysis closely and show that if our goal is to compute the exact median of a length-m data stream in $\widetilde{O}(1)$ space, we can do so in $O(\log m/\log\log m)$ passes.

Unit 12

Geometric Streams and Coresets

The notes for this unit are in reasonable shape but I am not done vetting them. There may be instances of slight errors or lack of clarity. I am continually polishing these notes and when I've vetted them enough, I'll remove this notice.

12.1 Extent Measures and Minimum Enclosing Ball

Up to this point, we have been interested in statistical computations. Our data streams have simply been "collections of tokens." We did not care about any structure amongst the tokens, because we were only concerned with functions of the token *frequencies*. The median and selection problems introduced some *slight* structure: the tokens were assumed to have a natural ordering.

Streams represent data sets. The data points in many data sets *do* have natural structure. In particular, many data sets are naturally geometric: they can be thought of as points in $\mathbb{R}^d$, for some dimension d. This motivates a host of computational geometry problems on data streams. We shall now study a simple problem of this sort, which is in fact a representative of a broader class of problems: namely, the estimation of *extent measures* of point sets.

Broadly speaking, an extent measure estimates the "spread" of a collection of data points, in terms of the size of the smallest object of a particular shape that contains the collection. Here are some examples.

- Minimum bounding box (two variants: axis-parallel or not)
- Minimum enclosing ball, or MEB
- Minimum enclosing shell (a shell being the difference of two concentric balls)
- Minimum width (i.e., min distance between two parallel hyperplanes that sandwich the data)

For this lecture, we focus on the MEB problem. Our solution will introduce the key concept of a *coreset*, which forms the basis of numerous approximation algorithms in computational geometry, including data stream algorithms for all of the above problems.

12.2 Coresets and Their Properties

The idea of a coreset was first formulated in Agarwal, Har-Peled and Varadarajan [AHPV04]; the term "coreset" was not used in that work, but became popular later. The same authors have a survey [AHPV05] that is a great reference on the subject, with full historical context and a plethora of applications. Our exposition will be somewhat specialized to target the problems we are interested in.

We define a "cost function" C to be a family of functions $\{C_P\}$ parametrized by point sets $P \subseteq \mathbb{R}^d$. For each P, we have a corresponding function $C_P \colon \mathbb{R}^d \to \mathbb{R}_+$. We say that C is *monotone* if

$$\forall Q \subseteq P \subseteq \mathbb{R}^d \; \forall x \in \mathbb{R}^d : \; C_Q(x) \le C_P(x) \,.$$

We are given a stream σ of points in $\mathbb{R}^d$, and we wish to compute the minimum value of the corresponding cost function C_σ. To be precise, we want to estimate $\inf_{x\in\mathbb{R}^d} C_\sigma(x)$; this will be our extent measure for σ.

The *minimum enclosing ball* (or MEB) problem consists of finding the minimum of the cost function C_σ, where

$$C_\sigma(x) := \max_{y\in\sigma} \|x-y\|_2 \,, \tag{12.1}$$

i.e., the radius of the smallest ball centered at x that encloses the points in σ.

Definition 12.2.1. Fix a real number $\alpha \ge 1$, and a cost function C parametrized by point sets in $\mathbb{R}^d$. We say Q is an α-*coreset* for $P \subseteq \mathbb{R}^d$ (with respect to C) if $Q \subseteq P$, and

$$\forall T \subseteq \mathbb{R}^d \; \forall x \in \mathbb{R}^d : \; C_{Q\cup T}(x) \le C_{P\cup T}(x) \le \alpha C_{Q\cup T}(x) \,. \tag{12.2}$$

Clearly, if C is monotone, the left inequality always holds. The cost function for MEB, given by (12.1), is easily seen to be monotone.

Our data stream algorithm for estimating MEB will work as follows. First we shall show that, for small $\varepsilon > 0$, under the MEB cost function, every point set $P \subseteq \mathbb{R}^d$ has a $(1+\varepsilon)$-coreset of size $O(1/\varepsilon^{(d-1)/2})$. The amazing thing is that the bound is independent of $|P|$. Then we shall give a data stream algorithm to compute a $(1+\varepsilon)$-coreset of the input stream σ, using small space. Clearly, we can estimate the MEB of σ by computing the exact MEB of the coreset.

We now give names to three useful properties of coresets. The first two are universal: they hold for all coresets, under all cost functions C. The third happens to hold for the specific coreset construction we shall see later (but is also true for a large number of other coreset constructions).

Merge Property: If Q is an α-coreset for P and Q' is a β-coreset for P', then $Q \cup Q'$ is an $(\alpha\beta)$-coreset for $P \cup P'$.

Reduce Property: If Q is an α-coreset for P and R is a β-coreset for Q, then R is an $(\alpha\beta)$-coreset for P.

Disjoint Union Property: If Q, Q' are α-coresets for P, P' respectively, and $P \cap P' = \varnothing$, then $Q \cup Q'$ is an α-coreset for $P \cup P'$.

To repeat: every coreset satisfies the merge and reduce properties. The proof is left as an easy exercise.

12.3 A Coreset for MEB

For nonzero vectors $u, v \in \mathbb{R}^d$, let $\operatorname{ang}(u,v)$ denote the angle between them, i.e.,

$$\operatorname{ang}(u,v) := \arccos \frac{\langle u,v\rangle}{\|u\|_2 \, \|v\|_2} \,,$$

where $\langle\cdot,\cdot\rangle$ is the standard inner product. We shall call a collection of vectors $\{u_1,\ldots,u_t\} \subseteq \mathbb{R}^d \setminus \{0\}$ a θ-*grid* if, for every nonzero vector $x \in \mathbb{R}^d$, there is a $j \in [t]$ such that $\operatorname{ang}(x,u_j) \le \theta$. We think of θ as being "close to zero."

The following geometric theorem is well known; it is obvious for $d = 2$, but requires a careful proof for higher d.

Theorem 12.3.1. *In $\mathbb{R}^d$, there is a θ-grid consisting of $O(1/\theta^{d-1})$ vectors. In particular, for $\mathbb{R}^2$, this bound is $O(1/\theta)$.*

Using this, we can construct a small coreset for MEB.

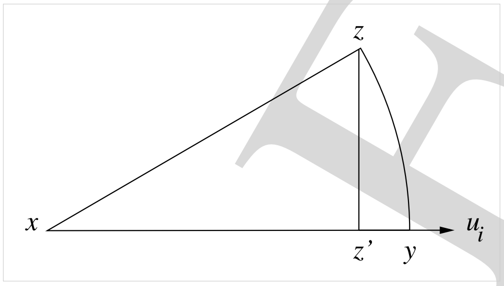
Theorem 12.3.2. *In $d \ge 2$ dimensions, the MEB cost function admits a $(1+\varepsilon)$-coreset of size $O(1/\varepsilon^{(d-1)/2})$.*

Proof. Let $\{u_1,\ldots,u_t\}$ be a θ-grid in $\mathbb{R}^d$, for a parameter $\theta = \theta(\varepsilon)$ to be chosen later. By Theorem 12.3.1, we may take $t = O(1/\theta^{d-1})$. Our proposed coreset for a point set $P \subseteq \mathbb{R}^d$ shall consist of the two extreme points of P along each of the directions $u_1,\ldots,u_t$. To be precise, let P be given. We then define

$$Q := \bigcup_{i=1}^{t} \{\arg\max_{x\in P}\langle x, v_i\rangle, \arg\min_{x\in P}\langle x, v_i\rangle\}\,. \tag{12.3}$$

We claim that Q is a $(1+\varepsilon)$-coreset of P, for a suitable choice of θ.

Since the MEB cost function is monotone, the left inequality in (12.2) always holds. We prove the right inequality "by picture" (see below); making this rigorous is left as an exercise.



Take an arbitrary $x \in \mathbb{R}^d$ and $T \subseteq \mathbb{R}^d$, and let z be the farthest point from x in the set $P \cup T$. If $z \in T$, then

$$C_{Q\cup T}(x) \geq \|x - z\|_2 = C_{P\cup T}(x)\,.$$

Otherwise, we have $z \in P$. By the grid property, there is a direction u_i that makes an angle of at most θ with $\vec{xz}$. Let y be the point such that $\vec{xy}$ is parallel to u_i and $\|x-y\|_2 = \|x-z\|_2$, and let z' be the orthogonal projection of z onto $\vec{xy}$. Then, Q contains a point from P whose orthogonal projection on $\vec{xy}$ lies to the right of z' (by construction of Q) and to the left of y (because z is farthest). Therefore

$$C_{Q\cup T}(x) \geq C_Q(x) \geq \left\|x - z'\right\|_2 \geq \|x - z\|\cos\theta = C_{P\cup T}(x)\cos\theta\,.$$

Using $\sec\theta \leq 1+\theta^2$ (which holds for θ small enough), we obtain $C_{P\cup T}(x) \leq (1+\theta^2)C_{Q\cup T}(x)$. Since this holds for all $x \in \mathbb{R}^d$, the right inequality in (12.2) holds with $\alpha = 1+\theta^2$. Since we wanted Q to be a $(1+\varepsilon)$-coreset, we may take $\theta = \sqrt{\varepsilon}$.

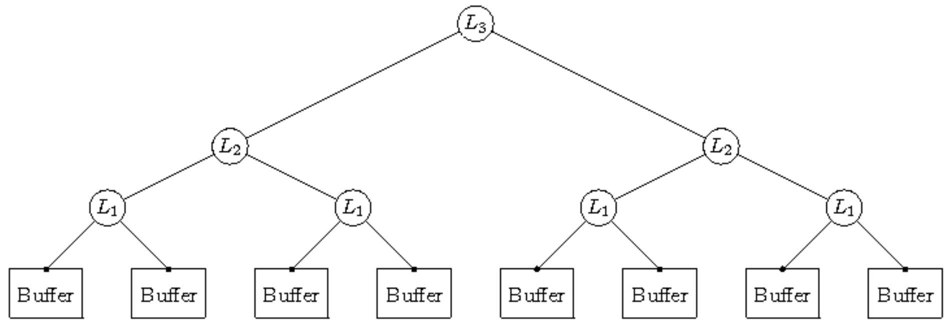
Finally, with this setting of θ, we have $|Q| \leq 2t = O(1/\varepsilon^{(d-1)/2})$, which completes the proof. □

12.4 Data Stream Algorithm for Coreset Construction

We now turn to algorithms. Fix a monotone cost function C, for point sets in $\mathbb{R}^d$, and consider the "C-minimization problem," i.e., the problem of estimating $\inf_{x\in\mathbb{R}^d} C_\sigma(x)$.

Theorem 12.4.1. *Suppose C admits $(1+\varepsilon)$-coresets of size $A(\varepsilon)$, and that these coresets have the disjoint union property. Then the C-minimization problem has a data stream algorithm that uses space $O(A(\varepsilon/\log m)\cdot\log m)$ and returns a $(1+\varepsilon)$-approximation.*

Proof. Our algorithm builds up a coreset for the input stream σ recursively from coresets for smaller and smaller substreams of σ in a way that is strongly reminiscent of the Munro-Paterson algorithm for finding the median.



[the picture is not quite right]

Set $\delta = \varepsilon/\log m$. We run a number of streaming algorithms in parallel, one at each "level": we denote the level-j algorithm by $\mathscr{A}_j$. By design, $\mathscr{A}_j$ creates a virtual stream that is fed into $\mathscr{A}_{j+1}$. Algorithm $\mathscr{A}_0$ reads σ itself, placing each incoming point into a buffer of large enough size B. When this buffer is full, it computes a $(1+\delta)$-coreset of the points in the buffer, sends this coreset to $\mathscr{A}_1$, and empties the buffer.

For $j \geq 1$, $\mathscr{A}_j$ receives a coreset at a time from $\mathscr{A}_{j-1}$. It maintains up to two such coresets. Whenever it has two of them, say P and P', it computes a $(1+\delta)$-coreset of $P \cup P'$, sends this coreset to $\mathscr{A}_{j+1}$, and discards both P and P'.

Thus, $\mathscr{A}_0$ uses space $O(B)$ and, for each $j \geq 1$, $\mathscr{A}_j$ uses space $O(A(\delta))$. The highest-numbered $\mathscr{A}_j$ that we need is at level $\lceil \log(m/B) \rceil$. This gives an overall space bound of $O(B + A(\varepsilon/\log m)\lceil \log(m/B) \rceil)$, by our choice of δ.

Finally, by repeatedly applying the reduce property and the disjoint union property, we see that the final coreset, Q, computed at the highest ("root") level is an α-coreset, where

$$\alpha = (1+\delta)^{1+\lceil \log(m/B) \rceil} \leq 1 + \delta \log m = 1 + \varepsilon\,.$$

To estimate $\inf_{x \in \mathbb{R}^d} C_\sigma(x)$, we simply output $\inf_{x \in \mathbb{R}^d} C_Q(x)$, which we can compute directly. □

As a corollary, we see that we can estimate the radius of the minimum enclosing ball (MEB), in d dimensions, up to a factor of $1+\varepsilon$, by a data stream algorithm that uses space

$$O\left(\frac{\log^{(d+1)/2} m}{\varepsilon^{(d-1)/2}}\right).$$

In two dimensions, this amounts to $O(\varepsilon^{-1/2} \log^{3/2} m)$.

Exercises

12-1 In the proof of Theorem 12.3.2, the proof that the constructed set Q satisfies the right inequality in eq. (12.2) was done "by picture." Make this proof formal, i.e., do it using only algebraic or analytic steps, without appealing to intuitive reasoning involving terms like "to the left/right of."

12-2 Prove that the coreset for MEB constructed in the proof of Theorem 12.3.2 has the disjoint union property, described in Section 12.2.

Unit 13

Metric Streams and Clustering

In Unit 12, we considered streams representing geometric data, and considered one class of computations on such data: namely, estimating extent measures. The measure we studied in detail — minimum enclosing ball (MEB) — can be thought of as follows. The center of the MEB is a crude *summary* of the data stream, and the radius of the MEB is the cost of thus summarizing the stream.

Often, our data is best summarized not by a single point, but by k points ($k \geq 1$): one imagines that the data naturally falls into k *clusters*, each of which can be summarized by a *representative* point. For the problem we study today, these representatives will be required to come from the original data. In general, one can imagine relaxing this requirement. At any rate, a particular clustering has an associated *summarization cost* which should be small if the clusters have small extent (according to some extent measure) and large otherwise.

13.1 Metric Spaces

It turns out that clustering problems are best studied in a setting more general than the geometric one. The only aspect of geometry that matters for this problem is that we have a notion of "distance" between two points. This is abstracted out in the definition of a *metric space*, which we give below.

Definition 13.1.1. A metric space is a pair (M,d), where M is a nonempty set (of "points") and $d : M \times M \to \mathbb{R}_+$ is a non-negative-valued "distance" function satisfying the following properties for all $x,y,z \in M$.

1. $d(x,y) = 0 \iff x = y$; (identity)
2. $d(x,y) = d(y,x)$; (symmetry)
3. $d(x,y) \leq d(x,z) + d(z,y)$. (triangle inequality)

Relaxing the first property to $d(x,x) = 0$ gives us a *semi-metric space* instead.

A familiar example of a metric space is $\mathbb{R}^n$, under the distance function $d(\boldsymbol{x},\boldsymbol{y}) = \|\boldsymbol{x}-\boldsymbol{y}\|\,\|_p$, where $p > 0$. In fact, the case $p = 2$ (Euclidean distance) is especially familiar. Another example should be just about as familiar to computer scientists: take an (undirected) graph $G = (V,E)$, let $M = V$, and for $u,v \in V$, define $d(u,v)$ to be the length of the shortest path in G between u and v.

At any rate, in this lecture, we shall think of the data stream as consisting of points in a metric space (M,d). The function d is made available to us through an *oracle* which, when queried with two points $x,y \in M$, returns the distance $d(x,y)$ between them. To keep things simple, we will not bother with the issues of representing, in working memory, points in M or distance values. Instead we will measure our space usage as the number of points and/or distances our algorithms store.

13.2 The Cost of a Clustering: Summarization Costs

Fix a metric space (M,d) and an integer $k \geq 1$. Given a *data set* $\sigma \subseteq M$, we wish to cluster it into at most k clusters, and summarize it by choosing a representative from each cluster. Suppose our set of chosen representatives is R.

If we think of the elements of σ as being locations of "customers" seeking some service, and elements of R as locations of "service stations," then one reasonable objective to minimize is the maximum distance that a customer has to travel to receive service. This is formalized as the *k-center* objective.

If we think of the elements of R as being locations of conference centers (for a multi-location video conference) and elements of σ as being home locations for the participants at this conference, another reasonable objective to minimize is the total fuel spent in getting all participants to the conference centers. This is formalized as the *k-median* objective. There is also another natural objective called *k-means* which is even older, is motivated by statistics and machine learning applications, and dates back to the 1960s.

To formally define these objectives, extend the function d by defining

$$d(x,S) := \min_{y\in S} d(x,y)\,,$$

for $x \in M$ and $S \subseteq M$. Then, having chosen representatives R, the best way to cluster σ is to assign each point in σ to its nearest representative in R. This then gives rise to (at least) the following three natural cost measures.

$$\begin{aligned}
\Delta_\infty(\sigma,R) &:= \max_{x\in\sigma} d(x,R)\,; && (k\text{-center})\\
\Delta_1(\sigma,R) &:= \sum_{x\in\sigma} d(x,R)\,; && (k\text{-median})\\
\Delta_2(\sigma,R) &:= \sum_{x\in\sigma} d(x,R)^2\,. && (k\text{-means})
\end{aligned}$$

Our goal is choose $R \subseteq \sigma$ with $|R| \leq k$ so as to minimize the cost $\Delta(\sigma,R)$. For the rest of this lecture we focus on the first of these costs (i.e., the k-center problem).

We shall give an efficient data stream algorithm that reads σ as an input stream and produces a summary R whose cost is at most some constant α times the cost of the best summary. Such an algorithm is called an α-approximation. In fact, we shall give two such algorithms: the first will use just $O(k)$ space and produce an 8-approximation. The second will improve the approximation ratio from 8 to $2+\varepsilon$, blowing up the space usage by about $O(1/\varepsilon)$.

As noted earlier, when we produce a set of representatives, R, we have in fact produced a clustering of the data implicitly: to form the clusters, simply assign each data point to its nearest representative, breaking ties arbitrarily.

13.3 The Doubling Algorithm

We focus on the k-center problem. The following algorithm maintains a set R consisting of at most k representatives from the input stream; these representatives will be our cluster centers. The algorithm also maintains a "threshold" τ throughout; as we shall soon see, τ approximates the summarization cost $\Delta_\infty(\sigma,R)$, which is the cost of the implied clustering.

To analyze this algorithm, we first record a basic fact about metric spaces and the cost function Δ_∞.

Lemma 13.3.1. *Suppose $x_1,\ldots,x_{k+1} \in \sigma \subseteq M$ satisfy $d(x_i,x_j) \geq t$ for all distinct $i,j \in [k+1]$. Then, for all $R \subseteq M$ with $|R| \leq k$, we have $\Delta_\infty(\sigma,R) \geq t/2$.*

Proof. Suppose, to the contrary, that there exists $R \subseteq M$ with $|R| \leq k$ and $\Delta_\infty(\sigma,R) < t/2$. Then, by the pigeonhole principle, there exist distinct $i,j \in [k+1]$, such that $\text{rep}(x_i,R) = \text{rep}(x_j,R) = r$, say. Now

$$d(x_i,x_j) \leq d(x_i,r) + d(x_j,r) < t/2 + t/2 = t\,,$$

where the first inequality is a triangle inequality and the second follows from $\Delta_\infty(\sigma,R) < t/2$. But this contradicts the given property of $\{x_1,\ldots,x_{k+1}\}$. □

Algorithm 20 Doubling algorithm for metric k-median

```
Initialize:
1: $R \leftarrow$ first $k+1$ points in stream
2: $(y,z) \leftarrow$ closest pair of points in $R$
3: $\tau \leftarrow d(y,z)$                                   ▷ Initial threshold
4: $R \leftarrow R \setminus \{z\}$          ▷ Culling step: we may have at most $k$ representatives

Process (token $x$) :
5: if $\min_{r \in R} d(x,r) > 2\tau$ then
6:     $R \leftarrow R \cup \{x\}$
7:     while $|R| > k$ do
8:         $\tau \leftarrow 2\tau$                            ▷ Raise (double) the threshold
9:         $R \leftarrow$ maximal $R' \subseteq R$ such that $\forall r \neq s \in R' : d(r,s) \geq \tau$    ▷ Culling step

Output: $R$
```

Next, we consider the algorithm's workings and establish certain invariants that it maintains.

Lemma 13.3.2. *Algorithm 20 maintains the following invariants at the start of each call to the* processing *section.*

1. *(Separation invariant) For all distinct $r,s \in R$, we have $d(r,s) \geq \tau$.*
2. *(Cost invariant) We have $\Delta_\infty(\sigma,R) \leq 2\tau$.*

Proof. At the end of initialization, Invariant 1 holds by definition of y,z, and τ. We also have $\Delta_\infty(\sigma,R) = \tau$ at this point, so Invariant 2 holds as well. Suppose the invariants hold after we have read an input stream σ, and are just about to process a new point x. Let us show that they continue to hold after processing x.

Consider the case that the condition tested in line 5 does not hold. Then R and τ do not change, so Invariant 1 continues to hold. We do change σ by adding x to it. However, noting that $d(x,R) \leq 2\tau$, we have

$$\Delta_\infty(\sigma \cup \{x\}, R) = \max\{\Delta_\infty(\sigma,R),\, d(x,R)\} \leq 2\tau\,,$$

and so, Invariant 2 also continues to hold.

Next, consider the case that the condition in line 5 holds. After line 6 executes, Invariant 1 continues to hold because the point x newly added to R satisfies its conditions. Further, Invariant 2 continues to hold since x is added to both σ and R, which means $d(x,R) = 0$ and $d(y,R)$ does not increase for any $y \in \sigma \setminus \{x\}$; therefore $\Delta_\infty(\sigma,R)$ does not change.

We shall now show that the invariants are satisfied after each iteration of the loop at line 7. Invariant 1 may be broken by line 8 but is explicitly restored in line 9. Immediately after line 8, with τ doubled, Invariant 2 is temporarily *strengthened* to $\Delta_\infty(\sigma,R) \leq \tau$. Now consider the set R' computed in line 9. To prove that Invariant 2 holds after that line, we need to prove that $\Delta_\infty(\sigma,R') \leq 2\tau$.

Let $u \in \sigma$ be an arbitrary data point. Then $d(u,R) \leq \Delta_\infty(\sigma,R) \leq \tau$. Let $r' = \operatorname{rep}(u,R) = \arg\min_{r \in R} d(u,r)$. If $r' \in R'$, then $d(u,R') \leq d(u,r') = d(u,R) \leq \tau$. Otherwise, by maximality of R', there exists a representative $s \in R'$ such that $d(r',s) < \tau$. Now

$$d(u,R') \leq d(u,s) \leq d(u,r') + d(r',s) < d(u,R) + \tau \leq 2\tau\,.$$

Thus, for all $x \in \sigma$, we have $d(x,R') \leq 2\tau$. Therefore, $\Delta_\infty(\sigma,R') \leq 2\tau$, as required. □

Having established the above properties, it is now simple to analyze the doubling algorithm.

Theorem 13.3.3. *The doubling algorithm uses $O(k)$ space and outputs a summary R whose cost is at most* 8 *times the optimum.*

Proof. The space bound is obvious. Let R^* be an optimum summary of the input stream σ, i.e., one that minimizes $\Delta_\infty(\sigma,R^*)$. Let $\hat{R}$ and $\hat{\tau}$ be the final values of R and τ after processing σ. By Lemma 13.3.2 (Invariant 2), we have $\Delta_\infty(\sigma,\hat{R}) \leq 2\hat{\tau}$.

Let $\tilde{R}$ be the value of R just before the final culling step. Then $|\tilde{R}| = k+1$ and $\tilde{R} \supseteq \hat{R}$. Further, the value of τ before it got doubled in that culling step was $\hat{\tau}/2$. By Lemma 13.3.2 (Invariant 1), every pair of distinct points in $\tilde{R}$ is at distance at least $\hat{\tau}/2$. Therefore, by Lemma 13.3.1, we have $\Delta_\infty(\sigma, R^*) \geq \hat{\tau}/4$.

Putting these together, we have $\Delta_\infty(\sigma, \hat{R}) \leq 8\Delta_\infty(\sigma, R^*)$. □

13.4 Metric Costs and Threshold Algorithms

The following two notions are key for our improvement to the above approximation factor.

Definition 13.4.1. A summarization cost function Δ is said to be *metric* if for all streams σ, π and summaries S, T, we have

$$\Delta(\sigma[S] \circ \pi, T) - \Delta(\sigma, S) \leq \Delta(\sigma \circ \pi, T) \leq \Delta(\sigma[S] \circ \pi, T) + \Delta(\sigma, S)\,. \tag{13.1}$$

Here, $\sigma[S]$ is the stream obtained by replacing each token of σ with its best representative from S.

Importantly, if we define "best representative" to be the "nearest representative," then the k-center cost function Δ_∞ is metric (an easy exercise). Also, because of the nature of the k-center cost function, we may as well replace $\sigma[S]$ by S in (13.1).

Consider the following example of a stream, with '$\circ$' representing "normal" elements in the stream and '$\otimes$' representing elements that have been chosen as representatives. Suppose a clustering/summarization algorithm is running on this stream, has currently processed σ and computed its summary S, and is about to process the rest of the stream, π.

$$\underbrace{\circ\otimes\circ\circ\otimes\circ\otimes\circ\circ\circ}_{\sigma} \,\Big|\, \underbrace{\circ\circ\circ\circ\circ\circ}_{\pi}$$

The definition of a metric cost function attempts to control the "damage" that would be done if the algorithm were to forget everything about σ at this point, except for the computed summary S. We think of T as summarizing the whole stream, $\sigma \circ \pi$.

Definition 13.4.2. Let Δ be a summarization cost function and let $\alpha \geq 1$ be a real number. An *α-threshold algorithm* for Δ is one that takes an input a *threshold* t and a data stream σ, and does one of the following two things.

1. Produces a summary S; if so, we must have $\Delta(\sigma, S) \leq \alpha t$.
2. Fails (producing no output); if so, we must have $\forall T : \Delta(\sigma, T) > t$.

The doubling algorithm contains the following simple idea for a 2-threshold algorithm for the k-center cost Δ_∞. Maintain a set S of representatives from σ that are pairwise $2t$ apart; if at any point we have $|S| > k$, then fail; otherwise, output S. Lemma 13.3.1 guarantees that this is a 2-threshold algorithm.

13.5 Guha's Cascading Algorithm

To describe Guha's algorithm, we generalize the k-center problem as follows. Our task is to summarize an input stream σ, minimizing a summarization cost given by Δ, which

- is a metric cost; and
- has an α-approximate threshold algorithm $\mathscr{A}$, for some $\alpha \geq 1$.

As we have just seen, k-center has both these properties.

The idea behind Guha's algorithm is to run multiple copies of $\mathscr{A}$ in parallel, with geometrically increasing thresholds. Occasionally a copy of $\mathscr{A}$ will fail; when it does, we start a new copy of $\mathscr{A}$ with a *much* higher threshold to take over from the failed copy, using the failed copy's summary as its initial input stream.

Here is an outline of the algorithm, which computes an $(\alpha + O(\varepsilon))$-approximation of an optimal summary, for $\varepsilon \ll \alpha$. Let S^* denote an optimal summary, i.e., one that minimizes $\Delta(\sigma, S^*)$. It should be easy to flesh this out into complete pseudocode; we leave this as an exercise.

- Perform some initial processing to determine a lower bound, c, on $\Delta(\sigma, S^*)$.
- Let $p = \lceil \log_{1+\varepsilon}(\alpha/\varepsilon) \rceil$. From now on, keep p instances of $\mathscr{A}$ running at all times, with thresholds increasing geometrically by factors of $(1+\varepsilon)$. The lowest threshold is initially set to $c(1+\varepsilon)$.
- Whenever $q \le p$ of the instances fail, start up q new instances of $\mathscr{A}$ using the summaries from the failed instances to "replay" the stream so far. When an instance fails, kill all other instances with a lower threshold. Alternatively, we can pretend that when an instance with threshold $c(1+\varepsilon)^j$ fails, its threshold is raised to $c(1+\varepsilon)^{j+p}$.
- Having processed all of σ, output the summary from the instance of $\mathscr{A}$ that has the lowest threshold.

13.5.1 Space Bounds

In the above algorithm, let s_0 denote the space required to determine the initial lower bound, c. Also, let $s_{\mathscr{A}}$ denote the space required by an instance of $\mathscr{A}$; we assume that this quantity is independent of the threshold with which $\mathscr{A}$ is run. Then the space required by the above algorithm is

$$\max\{s_0, p s_{\mathscr{A}}\} = O\left(s_0 + \frac{s_{\mathscr{A}}}{\varepsilon}\log\frac{\alpha}{\varepsilon}\right).$$

In the case of k-center, using the initialization section of the Doubling Algorithm to determine c gives us $s_0 = O(k)$. Furthermore, using the 2-threshold algorithm given at the end of Section 13.4, we get $s_{\mathscr{A}} = O(k)$ and $\alpha = 2$. Therefore, for k-center, we have an algorithm running in space $O((k/\varepsilon)\log(1/\varepsilon))$.

13.5.2 The Quality of the Summary

Consider a run of Guha's algorithm on an input stream σ. Consider the instance of $\mathscr{A}$ that had the smallest threshold (among the non-failed instances) when the input ended. Let t be the final threshold being used by this instance. Suppose this instance had its threshold raised j times overall. Let σ_i denote portion of the stream σ between the $(i-1)$th and ith raising of the threshold, and let σ_{j+1} denote the portion after the last raising of the threshold. Then

$$\sigma = \sigma_1 \circ \sigma_2 \circ \cdots \circ \sigma_{j+1},$$

Let S_i denote the summary computed by this instance of $\mathscr{A}$ after processing σ_i; then S_{j+1} is the final summary. During the processing of S_i, the instance was using threshold $t_i = t/(1+\varepsilon)^{p(j-i+1)}$. Since $p = \lceil \log_{1+\varepsilon}(\alpha/\varepsilon) \rceil$, we have $(1+\varepsilon)^p \ge \alpha/\varepsilon$, which gives $t_j \le (\varepsilon/\alpha)^{j-i+1}t$. Now, by Property 1 of an α-threshold algorithm, we have

$$\Delta(S_{i-1} \circ \sigma_i, S_i) \le \alpha(\varepsilon/\alpha)^{j-i+1}t, \quad \text{for } 1 \le i \le j+1, \tag{13.2}$$

where we put $S_0 = \varnothing$. Since Δ is metric, by (13.1), after the simplification $\sigma[S] = S$, we have

$$\Delta(\sigma_1 \circ \cdots \circ \sigma_i, S_i) \;\le\; \Delta(S_{i-1} \circ \sigma_i, S_i) + \Delta(\sigma_1 \circ \cdots \circ \sigma_{i-1}, S_{i-1}). \tag{13.3}$$

Using (13.3) repeatedly, we can bound the cost of the algorithm's final summary as follows.

$$\begin{aligned}
\Delta(\sigma, S_{j+1}) = \Delta(\sigma_1 \circ \cdots \circ \sigma_{j+1}, S_{j+1}) &\le \sum_{i=1}^{j+1} \Delta(S_{i-1} \circ \sigma_i, S_i) \\
&\le \sum_{i=1}^{j+1} \alpha(\varepsilon/\alpha)^{j-i+1}t && \text{(by (13.2))} \\
&\le \alpha t \sum_{i=0}^{\infty} \left(\frac{\varepsilon}{\alpha}\right)^i = (\alpha + O(\varepsilon))t.
\end{aligned}$$

Meanwhile, since t was the smallest threshold for a non-failed instance of $\mathscr{A}$, we know that $\mathscr{A}$ fails when run with threshold $t/(1+\varepsilon)$. By Property 2 of an α-threshold algorithm, we have

$$\Delta(\sigma, S^*) \ge \frac{t}{1+\varepsilon}.$$

Strictly speaking, the above reasoning assumed that at least one instance of $\mathscr{A}$ failed while processing σ. But notice that, by our choice of c in Guha's algorithm, the above inequality holds even if this isn't true, because in that case, we would have $t = c(1+\varepsilon)$.

Putting the last two inequalities together, we see that $\Delta(\sigma, S_{j+1})$ approximates the optimal cost $\Delta(\sigma, S^*)$ within a factor of $(1+\varepsilon)(\alpha + O(\varepsilon)) = \alpha + O(\varepsilon)$.

For k-center, since we have a 2-threshold algorithm, we get an overall approximation ratio of $2 + O(\varepsilon)$.

Exercises

13-1 Give a rigorous proof that the summarization cost function Δ_∞ (corresponding to the k-center objective) is metric, in the sense of Definition 13.4.1.
Write out the steps of reasoning explicitly, using algebra. Your proof should explain how each nontrivial step was derived. At least one of the steps will use the triangle inequality in the underlying metric space.

Unit 14

Graph Streams: Basic Algorithms

14.1 Streams that Describe Graphs

We have been considering streams that describe data with some kind of structure, such as geometric structure, or more generally, metric structure. Another very important class of structured large data sets is *large graphs*. In this and the next few units, we shall study several *streaming graph algorithms*: in each case, the input is a stream that describes a graph.

Since the terminology of graph theory is not totally uniform in the computer science and mathematics literature, it is useful to clearly define our terms.

Definition 14.1.1. A graph (respectively, digraph) is a pair (V,E) where V is a nonempty finite set of *vertices* and E is a set of *edges*. Each edge is an unordered (respectively, ordered) pair of distinct vertices. If G is a graph or digraph, we denote its vertex set by $V(G)$ and its edge set by $E(G)$.

Most streaming algorithms in the literature handle only undirected graphs (which, as our definition suggests, will simply be called "graphs"), although it makes sense to study algorithms that handle directed graphs (digraphs) too. Notice that our definition forbids parallel edges and loops in a graph.

Several interesting algorithmic questions involve graphs where each edge has a real-valued *weight* or an integer-valued *multiplicity*.

In all graph algorithmic problems we study, the vertex set $V(G)$ of the input graph G is known in advance, so we assume that $V(G) = [n]$, for some (large) integer n. The stream's tokens describe the edge set $E(G)$ in one of the following ways.

- In a *vanilla* or *insertion-only* graph stream, each token is an ordered pair $(u,v) \in [n] \times [n]$, such that the corresponding unordered pair $\{u,v\} \in E(G)$. We assume that each edge of G appears exactly once in the stream. There is no easy way to check that this holds, so we have to take this as a promise. While n is known beforehand, the stream length m, which also equals $|E(G)|$, is not.
- In a vanilla *weighted* graph stream, with weights lying in some set S, each token is a triple $(u,v,w) \in [n] \times [n] \times S$ and indicates that $\{u,v\}$ is an edge of G with weight w.
- A dynamic graph stream represents an *evolving* graph, where edges can come and go. Each token is a triple $(u,v,b) \in [n] \times [n] \times \{-1,1\}$ and indicates that an edge $\{u,v\}$ is being inserted into the graph if $b = 1$ and deleted from the graph if $b = -1$. It is promised that an edge being inserted isn't already in the graph and an edge being deleted is definitely in the graph.
- A *turnstile* graph stream represents an evolving multigraph. Each token is a triple $(u,v,\Delta) \in [n] \times [n] \times \mathbb{Z}$ and indicates that Δ is being added to the multiplicity of edge $\{u,v\}$. Edges that end up with a negative multiplicity usually don't make graph-theoretic sense, so it may be convenient to assume a promise that that doesn't happen.

In this unit, we will only study algorithms in a vanilla (i.e., insertion-only) streaming setting.

14.1.1 Semi-Streaming Space Bounds

Unfortunately, most of the interesting things we may want to compute for a graph provably require $\Omega(n)$ space in a streaming model, even allowing multiple passes over the input stream. These include such basic questions as "Is G connected?" and even "Is there a path from u to v in G?" where the vertices u and v are known beforehand. We shall prove such results when we study lower bounds, in later units.

Therefore, we have to reset our goal. Where $(\log n)^{O(1)}$ space used to be the holy grail for basic data stream algorithms, for several graph problems, the quest for a good space bound has to stop at $O(n \operatorname{polylog} n)$ space. Algorithms achieving such a space bound have come to be known as "semi-streaming" algorithms. Alternatively, we say that such an algorithm run in "semi-streaming space." Note that an algorithm that guarantees a space bound of $O(n^\alpha)$ for any constant $\alpha < 2$ is already achieving sublinear space, when the input graph is dense enough, because m could be as high as $\Omega(n^2)$.

14.2 The Connectedness Problem

Our first problem is CONNECTEDNESS: decide whether or not the input graph G, which is given by a stream of edges, is connected. This is a Boolean problem—the answer is either 0 (meaning "no") or 1 (meaning "yes")—and so we require an exact answer. We *could* consider randomized algorithms, but we won't need to.

For this problem, as well as all others in this unit, the algorithm will consist of maintaining a subgraph of G satisfying certain conditions. For CONNECTEDNESS, the idea is to maintain a spanning forest, F. As G gets updated, F might or might not become a tree at some point. Clearly G is connected iff it does.

The algorithm below maintains F as a set of edges. The vertex set is always $[n]$.

Algorithm 21 Graph connectivity

```
Initialize:
1: $F \leftarrow \varnothing$, flag $\leftarrow 0$

Process (token $\{u,v\}$):
2: if $\neg$flag and $F \cup \{\{u,v\}\}$ does not contain a cycle then
3:     $F \leftarrow F \cup \{\{u,v\}\}$
4:     if $|F| = n-1$ then flag $\leftarrow 1$

Output: flag
```

We have already argued the algorithm's correctness. Its space usage is easily seen to be $O(n \log n)$, since we always have $|F| \le n-1$, and each edge of F requires at most $2\lceil \log n \rceil = O(\log n)$ bits to describe.

The well known UNION-FIND data structure can be used to do the work in the processing section quickly. To test acyclicity of $F \cup \{\{u,v\}\}$, we simply check if root(u) and root(v) are distinct in the data structure.

14.3 The Bipartiteness Problem

A bipartite graph is one whose vertices can be partitioned into two disjoint sets—L and R, say—so that every edge is between a vertex in L and a vertex in R. Equivalently, a bipartite graph is one whose vertices can be properly colored using two colors.[1] Our next problem is BIPARTITENESS: determine whether or not the input graph G is bipartite.

[1] A coloring is proper if, for every edge e, the endpoints of e receive distinct colors.

Note that being bipartite is a *monotone* property (just as connectedness is): that is, given a non-bipartite graph, adding edges to it cannot make it bipartite. Therefore, once a streaming algorithm detects that the edges seen so far make the graph non-bipartite, it can stop doing more work. Here is our proposed algorithm.

Algorithm 22 Bipartiteness testing

```
Initialize:
 1: $F \leftarrow \varnothing$, flag $\leftarrow 1$

Process (token $\{u,v\}$) :
 2: if flag then
 3:     if $F \cup \{\{u,v\}\}$ does not contain a cycle then
 4:         $F \leftarrow F \cup \{\{u,v\}\}$
 5:     else if $F \cup \{\{u,v\}\}$ contains an odd cycle then
 6:         flag $\leftarrow 0$

Output: flag
```

Just like our CONNECTEDNESS algorithm before, this one also maintains the invariant that F is a subgraph of G and is a forest. Therefore it uses $O(n \log n)$ space. Its correctness is guaranteed by the following theorem.

Theorem 14.3.1. *Algorithm 22 outputs* 1 *(meaning "yes") iff the input graph G is bipartite.*

Proof. Suppose the algorithm outputs 0. Then G must contain an odd cycle. This odd cycle does not have a proper 2-coloring, so neither does G. Therefore G is not bipartite.

Next, suppose the algorithm outputs 1. Let $\chi : [n] \to \{0,1\}$ be a proper 2-coloring of the final forest F (such a χ clearly exists, since forests are bipartite). We claim that χ is also a proper 2-coloring of G, which would imply that G is bipartite and complete the proof.

To prove the claim, consider an edge $e = \{u,v\}$ of G. If $e \in F$, then we already have $\chi(u) \neq \chi(v)$. Otherwise, $F \cup \{e\}$ must contain an even cycle. Let π be the path in F obtained by deleting e from this cycle. Then π runs between u and v and has odd length. Since every edge on π is properly colored by χ, we again get $\chi(u) \neq \chi(v)$. □

The above algorithm description and analysis focuses on space complexity (for a reason: it is the main issue here) and does not address the time required to process each token. It is a good exercise to figure out how to make this processing time efficient as well.

14.4 Shortest Paths and Distance Estimation via Spanners

Now consider the problem of estimating distances in G. Recall that every graph naturally induces a metric on its vertex set: given two vertices $x, y \in V(G)$, the distance $d_G(x,y)$ between them is the length of the shortest x-to-y path in G.

$$d_G(x,y) := \min\{\text{length}(\pi) : \ \pi \text{ is a path in } G \text{ from } x \text{ to } y\}, \tag{14.1}$$

where the minimum of an empty set defaults to ∞ (i.e., $d_G(x,y) = \infty$ if there is no x-to-y path). In a streaming setting, our problem is to process an input (vanilla) graph stream describing G and build a data structure using which we can then answer *distance queries* of the form "what is the distance between x and y?"

The following algorithm computes an estimate $\hat{d}(x,y)$ for the distance $d_G(x,y)$. It maintains a suitable subgraph H of G which, as we shall see, satisfies the following property.

$$\forall x,y \in [n] : \ d_G(x,y) \le d_H(x,y) \le t \cdot d_G(x,y), \tag{14.2}$$

where $t \ge 1$ is an integer constant. The algorithm can then report $\hat{d}(x,y) = d_H(x,y)$ and this answer will be correct up to an approximation factor of t. A subgraph H satisfying eq. (14.2) is called a *t-spanner* of G. Note that the left inequality trivially holds for every subgraph H of G.

Algorithm 23 Distance estimation using a t-spanner

```
Initialize:
 1: $H \leftarrow \varnothing$

Process (token $\{u,v\}$) :
 2: if $d_H(u,v) \geq t+1$ then
 3:     $H \leftarrow H \cup \{\{u,v\}\}$

Output (query $(x,y)$) :
 4: report $\hat{d}(x,y) = d_H(x,y)$
```

14.4.1 The Quality of the Estimate

Theorem 14.4.1. *The final graph H constructed by Algorithm 23 is a t-spanner of G. Therefore, the estimate $\hat{d}(x,y)$ is a t-approximation to the actual distance $d_G(x,y)$: more precisely, it lies in the interval $[d_G(x,y), t \cdot d_G(x,y)]$.*

Proof. Pick any two distinct vertices $x,y \in [n]$. We shall show that eq. (14.2) holds. If $d_G(x,y) = \infty$, then G has no x-to-y path, so neither does its subgraph H, whence $d_H(x,y) = \infty$ as well. Otherwise, let π be the shortest x-to-y path in G and let $x = v_0, v_1, v_2, \ldots, v_k = y$ be the vertices on π, in order. Then $d_G(x,y) = k$.

Pick an arbitrary $i \in [k]$, and let $e = \{v_{i-1}, v_i\}$. If $e \in H$, then $d_H(v_{i-1}, v_i) = 1$. Otherwise, $e \notin H$, which means that at the time when e appeared in the input stream, we had $d_{H'}(v_{i-1}, v_i) \leq t$, where H' was the value of H at that time. Since H' is a subgraph of the final H, we have $d_H(v_{i-1}, v_i) \leq t$. Thus, in both cases, we have $d_H(v_{i-1}, v_i) \leq t$. By the triangle inequality, it now follows that

$$d_H(x,y) \leq \sum_{i=1}^{k} d_H(v_{i-1}, v_i) \leq tk = t \cdot d_G(x,y)\,,$$

which completes the proof, and hence implies the claimed quality guarantee for the algorithm. □

14.4.2 Space Complexity: High-Girth Graphs and the Size of a Spanner

How much space does Algorithm 23 use? Clearly, the answer is $O(|H| \log n)$, for the final graph H constructed by it. To estimate $|H|$, we note that, by construction, the shortest cycle in H has length at least $t+2$. We can then appeal to a result in extremal graph theory to upper bound $|H|$, the number of edges in H.

The *girth* $\gamma(G)$ of a graph G is defined to be the length of its shortest cycle; we set $\gamma(G) = \infty$ if G is acyclic. As noted above, the graph H constructed by our algorithm has $\gamma(H) \geq t+2$. The next theorem places an upper bound on the size of a graph with high girth.

Theorem 14.4.2. *For sufficiently large n, if the n-vertex graph G has m edges and $\gamma(G) \geq k$, for an integer k, then*

$$m \;\leq\; n + n^{1+1/\lfloor (k-1)/2 \rfloor}\,.$$

Proof. Let $d := 2m/n$ be the average degree of G. If $d \leq 3$, then $m \leq 3n/2$ and we are done. Otherwise, let F be the subgraph of G obtained by repeatedly deleting from G all vertices of degree less than $d/2$. Then F has minimum degree at least $d/2$, and F is nonempty, because the total number of edges deleted is less than $n \cdot d/2 = m$.

Put $\ell = \lfloor (k-1)/2 \rfloor$. Clearly, $\gamma(F) \geq \gamma(G) \geq k$. Therefore, for any vertex v of F, the ball in F centered at v and of radius ℓ is a tree (if not, F would contain a cycle of length at most $2\ell \leq k-1$). By the minimum degree property of F, when we root this tree at v, its branching factor is at least $d/2 - 1 \geq 1$. Therefore, the tree has at least $(d/2-1)^\ell$ vertices. It follows that

$$n \geq \left(\frac{d}{2} - 1\right)^\ell = \left(\frac{m}{n} - 1\right)^\ell,$$

which implies $m \leq n + n^{1+1/\ell}$, as required. □

Using $\lfloor (k-1)/2 \rfloor \geq (k-2)/2$, we can weaken the above bound to

$$m = O\left(n^{1+2/(k-2)}\right).$$

Plugging in $k = t+2$, we see that the t-spanner H constructed by Algorithm 23 has $|H| = O(n^{1+2/t})$. Therefore, the space used by the algorithm is $O(n^{1+2/t} \log n)$. In particular, we can 3-approximate all distances in a graph by a streaming algorithm in space $\widetilde{O}(n^{5/3})$.

Incidentally, more precise bounds on the size of a high-girth graph are known, though they do not lead to any asymptotic improvement in this space complexity analysis. See the paper by Alon, Hoory and Linial [AHL02] and the references therein.

Exercises

14-1 Suppose that we have a vanilla *weighted* graph stream, where each token is a triple (u, v, w_{uv}), specifying an edge $\{u, v\}$ and its weight $w_{uv} \in [W]$. This number W is an integer parameter. Distances in G are defined using weighted shortest paths, i.e.,

$$d_{G,w}(x,y) := \min\left\{ \sum_{e \in \pi} w_e : \pi \text{ is a path from } x \text{ to } y \right\}$$

Give an algorithm that processes G using space $\widetilde{O}(n^{1+2/t} \log W)$ so that, given $x, y \in V(G)$, we can then return a $(2t)$-approximation of $d_{G,w}(x,y)$. Give careful proofs of the quality and space guarantees of your algorithm.

Unit 15

Finding Maximum Matchings

In Unit 14, we considered graph problems that admitted especially simple streaming algorithms. Now we turn to *maximum matching*, where we will see a more sophisticated solution. Matchings are well-studied in the classical theory of algorithm design, and Edmonds's polynomial-time algorithm for the problem [Edm65] remains one of the greatest achievements in the field. For graphs described by streams, we cannot afford computations of the type performed by Edmonds's algorithm, but it turns out that we can achieve low (semi-streaming) space if we settle for approximation.

15.1 Preliminaries

Let $G = (V, E)$ be a graph. A *matching* in G is a set $M \subseteq E$ such that no two edges in M touch, i.e., $e \cap f = \varnothing$ for all $e \neq f \in M$. If G is weighted, with weight function $\mathrm{wt}\colon E \to \mathbb{R}$ giving a weight to each edge, then the weight of a subset $S \subseteq E$ of edges is defined by

$$\mathrm{wt}(S) = \sum_{e \in S} \mathrm{wt}(e)\,.$$

In particular, this defines the weight of every matching.

In the *maximum-cardinality matching* (MCM) problem, the goal is to find a matching in the input graph G that has maximum possible cardinality. In the *maximum-weight matching* (MWM) problem, the goal is to find a matching in the weighted input graph G that has maximum possible weight: we assume, without loss of generality, that edge weights are positive. Clearly, MCM is a special case of MWM where every edge has weight 1.

A particular matching M of G is said be an A-approximate MWM, where $A > 0$ is a real number, if

$$\frac{\mathrm{wt}(M^*)}{A} \le \mathrm{wt}(M) \le \mathrm{wt}(M^*)\,,$$

where M^* is an MWM of G. This also defines A-approximate MCMs. The right inequality is trivial, of course; it's all about the left inequality.

In this unit, the input graph will be given as a vanilla stream or a vanilla weighted stream (as defined in Section 14.1), depending on the problem at hand.

15.2 Maximum Cardinality Matching

Let us briefly consider the MCM problem, before moving on to the more interesting weighted case. It should be said that in traditional (non-streaming) algorithms, MCM is already a rich problem calling for deep algorithmic ideas. However, in a one-pass semi-streaming setting, essentially the only algorithm we know is the following extremely simple one.

Algorithm 24 Greedy algorithm for maximum-cardinality matching (MCM)

```
Initialize:
1: $M \leftarrow \varnothing$                                   ▷ invariant: $M$ will always be a matching

Process (token $(u,v)$):
2: if $M \cup \{\{u,v\}\}$ is a matching then
3:     $M \leftarrow M \cup \{u,v\}$

Output: $M$
```

A matching M of G is said to be *maximal* if no edge of G can be added to M to produce a larger matching. Note that this is *not* the same as saying that M is an MCM. For instance, if G is path of length 3, then the singleton set consisting of the middle edge is a maximal matching but obviously not an MCM.

It is easy to see that Algorithm 24 maintains a maximal matching of the input graph. This immediately implies that it uses at most $O(n \log n)$ space, since we have $|M| \le n/2$ at all times. The quality analysis is not much harder.

Theorem 15.2.1. *The output of Algorithm 24 is a 2-approximate MCM.*

Proof. Let M^* be an MCM. Suppose $|M| < |M^*|/2$. Each edge in M "blocks" (prevents from being added to M) at most two edges in M^*. Therefore, there exists an unblocked edge in M^* that could have been added to M, contradicting the maximality of M. □

We won't discuss MCM any further, but we note that if we are prepared to use more passes over the stream, it is possible to improve the approximation factor. There are various ways to do so and they all use more sophisticated ideas. Of these, one set of algorithms [**?**, **?**] uses the standard combinatorial idea of improving a matching M by finding an *augmenting path*, which is a path whose edges are alternately in M and not in M, beginning and ending in unmatched vertices. A different set of algorithms [AG13] uses linear programming duality.

A key takeaway is that for each $\varepsilon > 0$, we can produce a $(1+\varepsilon)$-approximate MCM using p_ε passes, where p_ε is a parameter depending only on ε. It remains open whether p_ε can be made to be at most $\mathrm{poly}(\varepsilon^{-1})$.

15.3 Maximum Weight Matching

We now turn to *weighted* graphs, where the goal is to produce a matching of high weight, specifically, an A-approximate MWM. Although it will be useful to think of an ideal model where edge weights can are positive reals, for space analysis purposes, we think of edge weights as lying in $[W]$, for some positive integer W. This is reasonable, because we can approximate real numbers using rationals and the clear denominators.

The algorithm uses a strategy of *eviction*: it maintains a current matching, based on edges seen so far. A newly seen edge might be *in conflict* with some edges (at most two) from the current matching. This is undesirable if the new edge is significantly better than the ones it's in conflict with, so in such a case, we take in the new edge and evict those conflicting edges from the current matching. The full logic is given in Algorithm 25.

Algorithm 25 Eviction algorithm for maximum-weight matching (MWM)

```
Initialize:
1: $M \leftarrow \varnothing$                                   ▷ invariant: $M$ will always be a matching

Process (token $(u,v,w)$):
2: $C \leftarrow \{e \in M : e \cap \{u,v\} \neq \varnothing\}$     ▷ current edge conflicts with $C$; note that $|C| \in \{1,2\}$
3: if $w > (1+\alpha)\,\mathrm{wt}(C)$ then
4:     $M \leftarrow M \cup \{u,v\} \setminus C$               ▷ current edge is too good; take it and evict $C$

Output: $M$
```

The logic of Algorithm 25 makes it clear that M is always a matching. In particular, $|M| \le n/2$. Therefore, the algorithm stores at most $O(n)$ edges and their weights, leading to a space usage in $O(n(\log n + \log W)) = \widetilde{O}(n)$, which makes it a semi-streaming algorithm.

Theorem 15.3.1. *Let $\hat{w}$ be the weight of the matching output by Algorithm 25 and let M^* be a maximum-weight matching of the input graph G. Then,*

$$\frac{\mathrm{wt}(M^*)}{c_\alpha} \le \hat{w} \le \mathrm{wt}(M^*)\,,$$

where c_α is a constant, depending only on α. Thus, the algorithm outputs a c_α-approximate MWM.

Proof. As the algorithm processes tokens, we say that an edge e is *born* when it is added to M; we say that e is *killed by* the edge f when it is removed from M upon processing f; we say that e *survives* if it is born and never killed; we say that e is *unborn* if, when it is process, it is not added to M.

We can define a parent-child relation among the born edges in $E(G)$ by defining an edge's killer (if it has one) to be its killer. This organizes the born edges into a collection of rooted trees that we call *killing tree*: each survivor is the root of such a tree, killed edges are non-root nodes in the trees, and unborn edges simply do not appear in the trees. Let S be the set of all survivors. Let $T(e)$ be the set of all strict descendants of e in its killing tree and let $T(S) = \bigcup_{e \in S} T(e)$.

The proof of the theorem is based on two key claims about these trees.

Claim 2. *The total weight of killed edges is bounded as follows:* $\mathrm{wt}(T(S)) \le \mathrm{wt}(S)/\alpha$.

Proof. By the logic of line 3 in Algorithm 25, for each edge e that is born,

$$\mathrm{wt}(e) > (1+\alpha) \sum_{f \text{ child of } e} \mathrm{wt}(f)\,.$$

Let $D_i(e) := \{f : f \text{ is a level-}i \text{ descendant of } e\}$. Applying the above fact repeatedly,

$$\mathrm{wt}(D_i(e)) < \frac{\mathrm{wt}(e)}{(1+\alpha)^i}\,.$$

Therefore,

$$\mathrm{wt}(T(S)) \le \sum_{e \in S} \sum_{i=1}^{\infty} \mathrm{wt}(D_i(e)) < \sum_{e \in S} \sum_{i=1}^{\infty} \frac{\mathrm{wt}(e)}{(1+\alpha)^i} = \sum_{e \in S} \mathrm{wt}(e) \cdot \frac{1}{\alpha} = \frac{\mathrm{wt}(S)}{\alpha}\,. \qquad \square$$

Claim 3. *Let M^* be an MWM of G. Then,* $\mathrm{wt}(M^*) \le (1+\alpha)(\mathrm{wt}(T(S)) + 2\,\mathrm{wt}(S))$.

Proof. The idea is to *charge* the weight of each edge in M^* to certain slots associated with the killing trees and then bound the amount of charge left in each slot at the end of the stream. A slot is indexed by a pair $\langle e, x\rangle$, where $e \in E(G)$ and x is an end-point of e. The charging scheme that we shall now describe will maintain the following *invariants*.

[CS1] For each vertex $x \in V(G)$, at most one slot of the form $\langle e, x\rangle$ holds a charge.

[CS2] For each slot $\langle e, x\rangle$, the charge allocated to it is at most $(1+\alpha)\,\mathrm{wt}(e)$.

Each edge $z = \{u, v\} \in M^*$ creates $\mathrm{wt}(z)$ amount of charge when it arrives in the stream. This charge is distributed among slots associated with u and v as follows.

- If z is born, a charge of $\mathrm{wt}(z)/2$ is allocated to each of $\langle z, u\rangle$ and $\langle z, v\rangle$.
- If z is not born because exactly one edge $e \in M$ touches z at u (say), then a charge of $\mathrm{wt}(z)$ is allocated to $\langle e, u\rangle$.
- If z is not born because exactly two edges $e, f \in M$ touch z at u, v (respectively), then charges of

$$\frac{\mathrm{wt}(z)\,\mathrm{wt}(e)}{\mathrm{wt}(e) + \mathrm{wt}(f)} \quad \text{and} \quad \frac{\mathrm{wt}(z)\,\mathrm{wt}(f)}{\mathrm{wt}(e) + \mathrm{wt}(f)}$$

  are allocated to $\langle e, u\rangle$ and $\langle f, v\rangle$ respectively.

Notice that, so far, invariant [CS1] is maintained because M^* is a matching. Further, [CS2] is maintained because of the logic of line 3 in Algorithm 25. Continuing with the description of the charging scheme, when an edge $e = \{u, v\} \notin M^*$ arrives in the stream, it may cause a reallocation of charges, as follows.

- If e is not born, nothing happens.
- If e is born, any charge allocated to a slot associated with u is transferred to slot $\langle e,u\rangle$, and similarly for v.

Obviously, this reallocation continues to maintain [CS1]. More importantly, when a charge allocated to some slot $\langle f,u\rangle$ is transferred to a slot $\langle e,u\rangle$, it must be the case that e kills f, implying that $\mathrm{wt}(e) \ge \mathrm{wt}(f)$, which maintains [CS2].

The total charge created during the course of the stream is $\mathrm{wt}(M^*)$. Notice that a slot that is allocated a charge must be associated with an edge that was born and is therefore in a killing tree. Each edge has exactly two slots. For an edge that was killed (i.e., an edge in $T(S)$), the charge in one of its two slots would have got transferred to a slot associated with its killer. Therefore, by invariant [CS2], the total charge allocated to the slots associated with e is at most $(1+\alpha)\,\mathrm{wt}(e)$ if $e \in T(S)$ and at most $2(1+\alpha)\,\mathrm{wt}(e)$ if $e \in S$. The claim follows. □

Using the above two claims,

$$\begin{aligned}\mathrm{wt}(M^*) &\le (1+\alpha)(\mathrm{wt}(T(S)) + 2\,\mathrm{wt}(S))\\ &\le (1+\alpha)\left(\frac{\mathrm{wt}(S)}{\alpha} + 2\,\mathrm{wt}(S)\right)\\ &= \left(\frac{1}{\alpha} + 3 + 2\alpha\right)\mathrm{wt}(S)\,. \end{aligned} \tag{15.1}$$

Since S is the matching output by Algorithm 25, Theorem 15.3.1 is proved with $c_\alpha = \alpha^{-1} + 3 + 2\alpha$. □

The best choice for α, which minimizes the expression (15.1), is $1/\sqrt{2}$. This gives us bound of $c_\alpha = 3 + 2\sqrt{2}$ on the approximation factor.

The above MWM algorithm and analysis are a fresh presentation of ideas in Feigenbaum et al. [FKM+05] and McGregor [McG05]. Since the publication of those early works, there has been a long line of further algorithms for MWM [Zel08, ELMS11, CS14], leading to improvements in the approximation factor, culminating in the algorithm of Paz and Schwartzman [PS19], which achieved a $(2+\varepsilon)$-approximation. There has also been work on multi-pass semi-streaming algorithms [AG13], where a $(1+\varepsilon)$-approximation is possible.

Exercises

15-1 Let us call Algorithm 25 an α-improving algorithm. As our analysis shows, setting $\alpha = 1/\sqrt{2}$ gives us a one-pass $(3+2\sqrt{2})$-approximation to the MWM.

Now suppose we are given $\varepsilon > 0$, and we run additional passes of an α-improving algorithm with the setting $\alpha = \varepsilon/3$ (the first pass still uses $\alpha = 1/\sqrt{2}$), using the output of the ith pass as the initial M for the $(i+1)$th pass. Let M_i denote the output of the ith pass. We stop after the $(i+1)$th pass when

$$\frac{\mathrm{wt}(M_{i+1})}{\mathrm{wt}(M_i)} \le 1 + \frac{\alpha^3}{1+2\alpha+\alpha^2-\alpha^3}\,, \quad \text{where } \alpha = \frac{\varepsilon}{3}\,.$$

Prove that this multi-pass algorithm makes $O(\varepsilon^{-3})$ passes and outputs a $(2+\varepsilon)$-approximation to the MWM.

Unit 16

Graph Sketching

The graph algorithms in Unit 14 used especially simple logic: each edge was either retained or discarded, based on some easy-to-check criterion. This simplicity came at the cost of not being able to handle edge deletions. In this unit, we consider graph problems in a *turnstile* stream setting, so a token can either insert or delete an edge or, more generally, increase or decrease its multiplicity.

Algorithms that can handle such streams use the basic observation, made back in Unit 5, that linear sketches naturally admit turnstile updates. The cleverness in these algorithms lies in constructing the right linear sketch. Such algorithms were first developed in the influential paper of Ahn, Guha, and McGregor [AGM12] and their underlying sketching technique has come to be known as the "AGM sketch." The sketch then allows us to solve CONNECTEDNESS and BIPARTITENESS (plus many other problems) in turnstile graph streams, using only semi-streaming space.

16.1 The Value of Boundary Edges

Our algorithms for CONNECTEDNESS and BIPARTITENESS will be based on the primitive of *sampling an edge from the boundary* of a given set of vertices. Later, we shall introduce a stream-friendly data structure—which will be a linear sketch—that implements this primitive.

Definition 16.1.1. Let $G = (V,E)$ be a graph. For each $S \subseteq V$, the *boundary* of S is defined to be

$$\partial S := \{e \in E : |e \cap S| = 1\},$$

i.e., the set of edges that have exactly one endpoint in S. If G' is a multigraph, we define boundaries based on the underlying simple graph consisting of the edges of G' that have nonzero multiplicity.

In the trivial cases $S = \varnothing$ and $S = V$, we have $\partial S = \varnothing$. We shall say that S is a *nontrivial* subset of vertices if $\varnothing \neq S \neq V$. Here is a simple lemma about such subsets that you can prove yourself.

Lemma 16.1.2. *Let S be a nontrivial subset of vertices of G. Then $\partial S = \varnothing$ iff S is a union of some, but not all, connected components of G. In particular if G is connected, then $\partial S \neq \varnothing$.* □

Assume, now, that we have somehow processed our turnstile graph stream to produce a data structure $\mathscr{B}$ that can be queried for boundary edges. Specifically, when $\mathscr{B}$ is queried with parameter S, it either returns some edge in ∂S (possibly a random edge, according to some distribution) or declares that $\partial S = \varnothing$. We shall call $\mathscr{B}$ a *boundary sampler* for G. Let us see how we can use $\mathscr{B}$ to solve our graph problems.

16.1.1 Testing Connectivity Using Boundary Edges

To gain some intuition for why finding boundary edges is useful, let us consider an input graph $G = (V,E)$ that *is* connected.

Suppose we have a partition of V into $k \geq 2$ nonempty subsets $C_1, \ldots, C_k$, which we shall call *clusters*. the Lemma 16.1.2 says that every one of these clusters must have nonempty boundary. Suppose we query $\mathscr{B}$ with parameter $C_1, \ldots, C_k$ (one by one) and collect the returned edges $e_1, \ldots, e_k$. Each of these edges must necessarily be inter-cluster and we can use these edges to merge clusters that are connected to one another. If (the subgraph induced by) each C_i happened to be connected to begin with, then after these mergers we again have a partition into connected clusters.

By starting with a partition into singleton clusters—which is trivially a partition into connected clusters—and iterating the above boundary query and merge procedure, we will eventually merge everything into one single cluster. At this point, we have convinced ourselves that G is indeed connected. In fact, we can say more: the edges we obtained by querying $\mathscr{B}$ suffice to connect up all the vertices in G and therefore contain a spanning tree of G.

We formalize the above intuition in Algorithm 26, which computes a spanning forest of a general (possibly disconnected) graph, given access to a boundary sampler $\mathscr{B}$ as described above. Note that this is not a streaming algorithm! It is to be called after the graph stream has been processed and $\mathscr{B}$ has been created. As usual, the vertex set of the input graph is assumed to be $[n]$.

Algorithm 26 Compute edges of spanning forest using boundary sampler

```
1: function SPANNINGFOREST(vertex set $V$, boundary sampler $\mathscr{B}$)
2:     $F \leftarrow \varnothing$
3:     $\mathscr{C} \leftarrow \{\{v\} : v \in V\}$                          ▷ initial partition into singletons
4:     repeat
5:         query $\mathscr{B}(C)$ for each $C \in \mathscr{C}$ and collect returned edges in $H$     ▷ $\mathscr{B}(C)$ yields an arbitrary edge in $\partial C$
6:         $F \leftarrow$ spanning forest of graph $(V, F \cup H)$
7:         $\mathscr{C} \leftarrow \{V(T) : T$ is a connected component of $(V, F)\}$     ▷ new partition after merging clusters
8:     until $H = \varnothing$                                        ▷ stop when no new boundary edges found
9:     return $F$
```

Consider how Algorithm 26 handles a connected input graph G. Let F_i and $\mathscr{C}_i$ denote the values of F and $\mathscr{C}$ at the start of the ith iteration of the loop at line 4. Let H_i denote the set H collected in the ith iteration. Let $k_i = |\mathscr{C}_i|$. Equivalently, k_i is the number of connected components of the forest (V, F_i). Then, k_{i+1} is exactly the number of connected components of the graph $(\mathscr{C}_i, H_i)$ that treats each cluster as a "supervertex." If $k_i > 1$, by Lemma 16.1.2, this k_i-vertex graph has no isolated vertices, so $k_{i+1} \leq k_i/2$. (It is instructive to spell out when exactly we would have $k_{i+1} = k_i/2$.)

Since $k_1 = n$, it follows that k_i reaches 1 after at most $\lfloor \log n \rfloor$ iterations of the loop at line 4. Further, when k_i becomes 1, the graph (V, F_i) becomes connected, and so, F_i is a spanner tree of G.

Finally, consider how Algorithm 26 handles a general input graph G whose connected components are $G_1, \ldots, G_\ell$. Applying the above argument to each component separately, we see that the algorithm terminates in $O(\max_{i\in[\ell]} \log |V(G_i)|) = O(\log n)$ iterations and returns a spanning forest of G.

16.1.2 Testing Bipartiteness

A neat idea now allows us to solve BIPARTITENESS as well. The key observation is that the problem reduces to CONNECTEDNESS.

The *bipartite double cover* of a graph $G = (V, E)$ is defined to be the graph $(V \times \{1, 2\}, E')$, where

$$E' = \{\{(u,1),(v,2)\} : \{u,v\} \in E\} .$$

It is important to note that $|E'| = 2|E|$, since each $\{u, v\} \in E$ in fact gives rise to both $\{\{(u,1),(v,2)\}$ and $\{\{(v,1),(u,2)\}$. More succinctly, the double cover is the tensor product $G \times K_2$. An example is shown in Figure 16.1.

Below is the important lemma we need about this construction. We skip the proof, which is a nice exercise in elementary graph theory.

Lemma 16.1.3. *A connected graph G is bipartite iff $G \times K_2$ is disconnected. More generally, for a graph G that has k connected components:*

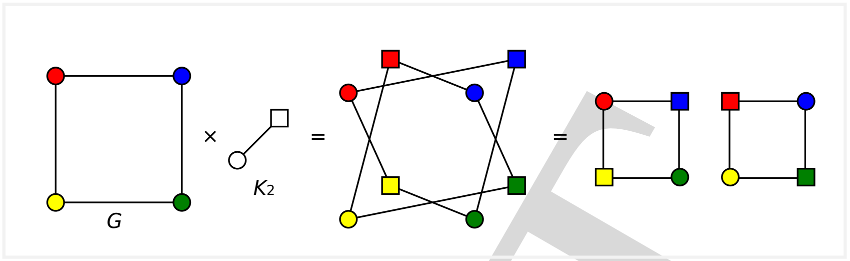


Figure 16.1: An example of a bipartite double cover. This graph G is connected and bipartite, so its double cover is disconnected. Source: Wikimedia, Miym / CC BY-SA (https://creativecommons.org/licenses/by-sa/3.0).

- *if G is bipartite, then $G \times K_2$ has exactly $2k$ connected components;*
- *else, $G \times K_2$ has fewer than $2k$ connected components.*

The lemma, combined with Algorithm 26, immediately gives us the following algorithm for bipartiteness testing, once we have a boundary sampler sketch, $\mathscr{B}$. As the stream is read, we maintain two copies of $\mathscr{B}$: one for the input graph G and one for $G \times K_2$. At the end of the stream, we run call SPANNINGFOREST on each of the copies of $\mathscr{B}$ to determine the number of connected components of G and of $G \times K_2$. Lemma 16.1.3 gives us our answer.

16.2 The AGM Sketch: Producing a Boundary Edge

We have arrived at the heart of the algorithm. It remains to implement the primitive on which the algorithms of Section 16.1.1 rest. We need to design a linear sketch of a turnstile edge stream that allows us to sample boundary edges.

As a start, consider the special-case problem of producing an edge from $\partial\{v\}$, where v is a vertex. In other words, we seek an edge incident to v. Let $\boldsymbol{A}$ be the adjacency matrix of G, i.e., the matrix whose entry A_{uv} is the multiplicity of edge $\{u,v\}$. Sampling an edge incident to u is the same as sampling an entry from the support of the uth row of A. We *do* know an efficient linear sketch for this: the ℓ_0-sampling sketch, described in Unit 10.

Therefore, if we maintain an ℓ_0-sampling sketch per row of $\boldsymbol{A}$—i.e., per vertex of G—then we use only $\widetilde{O}(n)$ space and can produce, on demand, an edge incident to any given vertex. There is a small probability that a query will fail, but this probability can be kept below $O(1/n^c)$ for any constant c.

This idea does not, however, generalize to sampling from ∂S where $|S| > 1$. We would like to sample from the support of a vector obtained by "combining the rows indexed by S" in such a way that entries corresponding to edges within S are zeroed out. The clever trick that allows such a combination is to replace the adjacency matrix $\boldsymbol{A}$ with the *signed incidence matrix* $\boldsymbol{B}$ defined as follows. Let $P = \binom{V}{2}$ be the set of unordered pairs of vertices, i.e., the set of all potential edges of G. Then $\boldsymbol{B} \in \mathbb{Z}^{V \times P}$ is the matrix whose entries are

$$B_{ue} = \begin{cases} A_{uv}\,, & \text{if } e = \{u,v\} \text{ and } u < v\,, \\ -A_{uv}\,, & \text{if } e = \{u,v\} \text{ and } u > v\,, \\ 0\,, & \text{if } u \notin e\,. \end{cases}$$

[[*** Insert example here ***]]

Consider the column of $\boldsymbol{B}$ corresponding to a pair $e = \{x,y\}$. If e is not an edge of G (equivalently, e has multiplicity zero), then the column is all-zero. Otherwise, if e appears with multiplicity c, then the column has exactly two nonzero entries—in rows x and y—and these two entries sum to zero. This immediately gives us the following.

Observation 16.2.1. *Let $\boldsymbol{b}_u$ denote the u-th row of the signed incidence matrix $\boldsymbol{B}$ of a graph G. For all $S \subseteq V(G)$,*

$$\operatorname{supp} \sum_{u \in S} \boldsymbol{b}_u = \partial S\,.$$

Let $\boldsymbol{L}$ be a (random) ℓ_0-sampling sketch matrix, as constructed in Unit 10. Then, by querying the sketch $\boldsymbol{L}\sum_{u\in S}\boldsymbol{b}_u$, we can (with high probability) produce a random edge from ∂S or detect that $\partial S = \varnothing$.[1] By linearity,

$$\boldsymbol{L}\sum_{u\in S}\boldsymbol{b}_u = \sum_{u\in S}\boldsymbol{L}\boldsymbol{b}_u\,,$$

so it suffices to maintain a sketch $\boldsymbol{L}\boldsymbol{b}_u$ corresponding to each vertex u. The construction in Unit 10 guarantees that each ℓ_0-sampling sketch uses $O(\log^2 n \log \delta^{-1})$ space to achieve a failure probability of δ, leading to an overall space usage of $\widetilde{O}(n)$ if we use one such sketch per vertex.

We almost have a complete algorithm now, but we need to address two issues.

1. In executing a "recovery procedure" such that Algorithm 26, we need to query ℓ_0-samplers multiple times. Considering this, how many independent samplers must we construct?

2. In Section 16.1.1, we assumed that the boundary samplers always work, but our ℓ_0-samplers have a certain failure probability. How do we account for this?

[[*** Wrap up analysis and provide pseudocode ***]]

[1]Moreover, the random edge we produce is nearly uniformly distributed in ∂S. The algorithms in this unit do not make use of this property.

Unit 17

Counting Triangles

When analyzing a large graph, an important class of algorithmic problems is to count how frequently some particular pattern occurs in the graph. Perhaps the most important such pattern is a *triangle*, i.e., a triple of vertices $\{u,v,w\}$ such that $\{u,v\}$, $\{v,w\}$ and $\{u,w\}$ are all edges in the graph. Counting (or estimating) the number of triangles gives us a useful summary of the "community forming" tendency of the graph: especially relevant when the graph is a social network, for instance.

We shall consider the problem of counting (or estimating) the number of triangles in a graph. It turns out that this estimation problem is a rare example of a natural graph problem where a truly low-space (as opposed semi-streaming) solution is possible. Moreover, a *sketching* algorithm is possible, which means that we can solve the problem even in a turnstile model.

17.1 A Sampling-Based Algorithm

Given a vanilla graph stream describing a graph $G = (V,E)$, where $V = [n]$, we want to estimate $T(G)$, the number of triangles in G.

Unfortunately, we *cannot* estimate $T(G)$ efficiently, in the sense of multiplicative approximation, no matter how loose the factor. The reason is that using a single pass, we require $\Omega(n^2)$ simply to distinguish the cases $T(G) = 0$ and $T(G) \geq 1$. Later in the course, we shall see how to prove such a lower bound.

In light of the above, we will need to relax our goal. One way to do so is to aim for *additive* error in our estimate, instead of multiplicative error. Another way—the one that has become standard in the literature and that we shall use here—is to multiplicatively approximate $T(G)$ *under a promise* that $T(G) \geq T$, for some lower bound $T \geq 1$. The higher this lower bound, the more restricted our space of valid inputs. Accordingly, we will aim for algorithms whose space complexity is a decreasing function of T.

Here is an algorithm (in outline form) that uses the simple primitive of *reservoir sampling* that we had seen in Section 6.2.

1. Pick an edge $\{u,v\}$ uniformly at random from the stream, using reservoir sampling.
2. Based on the above, pick a vertex w uniformly at random from $V \setminus \{u,v\}$.
3. Then, if $\{u,w\}$ and $\{v,w\}$ appear after $\{u,v\}$ in the stream, then output $m(n-2)$ else output 0.

A little thought shows that a single pass suffices to carry out all of the above steps. One can show that the above algorithm, after suitable parallel repetition, provides an (ε,δ)-estimate for $T(G)$ using space $O(\varepsilon^{-2}\log\delta^{-1}mn/T)$, under the promise that $T(G) \geq T$. The outline is as follows. First, one shows that the algorithm's output is an unbiased estimator for $T(G)$. Next, one computes a variance bound and uses Chebyshev's inequality to obtain the claimed space bound. The details are fairly straightforward given what has come before in the course.

17.2 A Sketch-Based Algorithm

The following algorithm is due to Bar-Yossef, Kumar, and Sivakumar [BKS02]. It uses space

$$\widetilde{O}\left(\frac{1}{\varepsilon^2}\log\frac{1}{\delta}\cdot\left(\frac{mn}{T}\right)^2\right), \tag{17.1}$$

to produce an (ε,δ)-estimate of $T(G)$. This is greater than the space used by the algorithm outlined in Section 17.1. However, this new algorithm has the advantage of being based on linear sketches. What it computes is not exactly a linear transformation of the stream, but nevertheless we can compose "sketches" computed by this algorithm for two different edge streams and we can handle edge deletions.

The high-level idea in the algorithm is to take the given (actual) stream σ of edges and transform it on the fly into a "virtual stream" τ of triples $\{u,v,w\}$, where $u,v,w \in V$. Specifically, upon seeing a single actual token, we pretend that we have seen $n-2$ virtual tokens as indicated below:

$$\begin{array}{lcl} \text{actual token} & & \text{virtual tokens} \\ \{u,v\} & \longrightarrow & \{u,v,w_1\},\{u,v,w_2\},\ldots,\{u,v,w_{n-2}\} \end{array}$$

where $\{w_1,w_2,\ldots,w_{n-2}\} = V \setminus \{u,v\}$. Thus, if σ consists of m edge updates, then τ consists of the corresponding $m(n-2)$ virtual tokens.

Here comes the ingenious idea: consider the frequency moments $F_k(\tau)$ of this virtual stream! Let us define

$$T_j = T_j(G) := \left|\{\{u,v,w\} : u,v,w \in V \text{ distinct and induce exactly } j \text{ edges of } G \text{ between themselves}\}\right| .$$

Then note that

$$T_0 + T_1 + T_2 + T_3 = \binom{n}{3}.$$

If we knew the frequency moments $F_k(\tau)$, we could derive further linear equations involving these T_j values. For example,

$$\begin{aligned} F_2(\tau) &= \sum_{u,v,w} (\text{number of occurrences of } \{u,v,w\} \text{ in } \tau)^2 \\ &= 1^2\cdot T_1 + 2^2\cdot T_2 + 3^2\cdot T_3 \\ &= T_1 + 4T_2 + 9T_3 \end{aligned} \tag{17.2}$$

We can derive another two linear equations using $F_1(\tau)$ and $F_0(\tau)$. It is a good exercise to derive them and then check that all four linear equations for the T_j values are linearly independent, so we can solve for the T_j values. Finally, note that $T_3 = T(G)$, the number of triangles in G.

As we know from earlier in the course, we can compute good *estimates* for $F_0(\tau), F_1(\tau)$, and $F_2(\tau)$, all using linear sketches. By carefully managing the accuracy guarantees for these sketches and analyzing how they impact the accuracy of the computed value of $T(G)$, we can arrive at the space bound given in eq. (17.1).

Exercises

17-1 Formally analyze the algorithm outlined in Section 17.1 and prove the claimed space bound.

17-2 Consider the operation of the algorithm outlined in Section 17.2 on a stream consisting of m edge insertions and no deletions, producing an n-vertex graph G that is promised to contain at least T triangles, i.e., $T(G) \geq T$. Prove that, for the derived stream τ, we have the following relations, analogous to eq. (17.2).

$$\begin{aligned} F_1(\tau) &= T_1 + 2T_2 + 3T_3\,; \\ F_0(\tau) &= T_1 + T_2 + T_3\,. \end{aligned}$$

Based on these, work out an exact formula for $T(G)$ in terms of n, m, $F_0(\tau)$, and $F_2(\tau)$. Then work out what guarantees you need on estimates for $F_0(\tau)$ and $F_2(\tau)$ so that your formula gives you a $(1\pm\varepsilon)$-approximation to $T(G)$. Finally, based on these required guarantees, prove that the space upper bound given in eq. (17.1) holds.

Unit 18

Communication Complexity and a First Look at Lower Bounds

We have been making occasional references to impossibility results saying that such-and-such in not doable in a data streaming model. Such results are called *lower bounds*: formally, one shows that in order to accomplish some well-defined task in a streaming model, at least a certain number of bits of space is required.

The standard approach to proving such lower bounds results goes through *communication complexity*, itself a rich and sometimes deep area of computer science that we will only scratch the surface of. In these notes, our focus is to develop just enough of the subject to be able to apply it to many basic lower bound problems in data streaming. Readers interested in delving deeply into communication complexity are referred to a textbook on the subject [KN97, RY20].

For now, we shall develop just enough of communication complexity to be able to prove certain space lower bounds for the MAJORITY and FREQUENCY-ESTIMATION problems, encountered in Unit 1.

18.1 Communication Games, Protocols, Complexity

A *communication game* is a cooperative game between two or more players where each player holds some portion of an overall input and their joint goal is to evaluate some function of this input. Most of the time we shall only care about two-player games, where the players—named Alice and Bob—seek to compute a function $f\colon \mathscr{X} \times \mathscr{Y} \to \mathscr{Z}$, with Alice starting out with an input fragment $x \in \mathscr{X}$ and Bob with an input fragment $y \in \mathscr{Y}$.

Obviously, in order to carry out their computation, Alice will have to tell Bob something about x, and/or Bob will have to tell Alice something about y. How they do so is given by a predetermined *protocol*, which instructs players to send message bits by evaluating appropriate "message functions." For instance, a *one-round* or *one-way* protocol would involve a single message from Alice to Bob (say), given by a function $\mathrm{msg}_A\colon \mathscr{X} \to \{0,1\}^*$, following which Bob would produce an output based on his own input fragment and the message received from Alice. A three-round protocol where each message is ℓ bits long would be specified by four functions

$$\begin{aligned}
\mathrm{msg}_{A,1}\colon & \mathscr{X} \to \{0,1\}^\ell ,\\
\mathrm{msg}_{B,1}\colon & \mathscr{Y} \times \{0,1\}^\ell \to \{0,1\}^\ell ,\\
\mathrm{msg}_{A,2}\colon & \mathscr{X} \times \{0,1\}^\ell \to \{0,1\}^\ell ,\\
\mathrm{out}_B\colon & \mathscr{Y} \times \{0,1\}^\ell \times \{0,1\}^\ell \to \mathscr{Z} .
\end{aligned}$$

On input (x,y), the above protocol would cause Alice to send $w_1 := \mathrm{msg}_{A,1}(x)$, then Bob to send $w_2 := \mathrm{msg}_{B,1}(y,w_1)$, then Alice to send $w_3 := \mathrm{msg}_{A,2}(x,w_2)$, and finally Bob to output $z := \mathrm{out}_B(y,w_1,w_3)$. Naming the above protocol Π, we would define $\mathrm{out}^\Pi(x,y)$ to be this z. We say that Π computes f if $\mathrm{out}^\Pi(x,y) = f(x,y)$ for all $(x,y) \in \mathscr{X} \times \mathscr{Y}$.

A fully formal definition that handles two-player protocols in general can be given by visualizing the protocol's execution as descent down a binary decision tree, with each node (a) belonging to the player who is supposed to send the next bit and (b) being labeled with an appropriate message function that produces the bit to be sent, thereby directing the descent either to the left or the right child. We do not need this general treatment in these notes, so we shall not give more details. The interested reader can find these details in either of the aforementioned textbooks.

The *cost* of a protocol Π is the worst-case (over all inputs) number of bits communicated in an execution of Π. The *deterministic communication complexity* $\mathrm{D}(f)$ of a function f is defined to be the minimum cost of a protocol that correctly computes f on every input.

Thus far, we have been describing *deterministic* communication protocols. More generally, we can consider *private-coin randomized* protocols, where players may compute their messages based on random coin tosses. That is, each message function takes a random bit string as an additional input (besides the usual inputs: the sending player's fragment and the history of received messages). Thus, each player generates random bits as needed by tossing a virtual coin private to them. Still more generally, we have *public-coin randomized* protocols, where a sufficiently long random string is made available to all players at the start and the players may refer to its bits at any time. In these notes, when we say "randomized protocol" without qualification, we mean this stronger, public-coin, variant.

For a randomized protocol Π, we define its cost to be the worst-case (over all inputs and all settings of the random string) number of bits communicated in an execution. For each input (x,y), the output $\mathrm{out}^{\Pi}(x,y;R)$ is a random variable, depending on the random string R. We say that Π computes f with error δ if

$$\forall (x,y) \in \mathscr{X} \times \mathscr{Y} : \; \mathbb{P}\left\{\mathrm{out}^{\Pi}(x,y;R) \neq f(x,y)\right\} \leq \delta . \tag{18.1}$$

The δ-error randomized communication complexity $\mathrm{R}_\delta(f)$ is defined to be the minimum cost of a protocol that satisfies eq. (18.1). For a Boolean function f, it will be convenient to define $\mathrm{R}(f) := \mathrm{R}_{1/3}(f)$. By a standard parallel repetition argument via Chernoff bound (analogous to the "median trick" in Section 2.4), for any positive constant $\delta \leq 1/3$,

$$\mathrm{R}_\delta(f) = \Theta(\mathrm{R}(f)) . \tag{18.2}$$

We shall use the notations $\mathrm{D}^{\rightarrow}(f)$ and $\mathrm{R}^{\rightarrow}(f)$ for deterministic and randomized communication complexities (respectively) restricted to one-way protocols where the only message goes from Alice to Bob. Similarly, we shall use the notations $\mathrm{D}^k(f)$ and $\mathrm{R}^k(f)$ when restricting to k-round protocols, where the first round sender is Alice. We shall use the notation $\mathrm{R}^{\mathrm{priv}}(f)$ when restricting to private-coin protocols.

Trivially, every two-player communication problem can be solved by having one player sending their input to the other. Keep in mind that *computation time and space do not count towards the cost*, only communication does. Further, extra rounds of communication or removing a restriction on the number of rounds can only help. We record the result of these observations below.

Observation 18.1.1. *For every two-player communication game given by a function f and integers $k > \ell > 0$, the following inequalities hold:*

- $\mathrm{R}(f) \leq \mathrm{D}(f) \leq \min\{\log|\mathscr{X}|, \log|\mathscr{Y}|\}$;
- $\mathrm{D}(f) \leq \mathrm{D}^k(f) \leq \mathrm{D}^\ell(f)$;
- $\mathrm{R}(f) \leq \mathrm{R}^k(f) \leq \mathrm{R}^\ell(f)$. □

Typically, we will be interested in the asymptotic behavior of $\mathrm{D}(f)$ or $\mathrm{R}(f)$ for functions f whose inputs can be expressed in "about N bits." By Observation 18.1.1, an upper bound of $O(N)$ would be trivial, so sublinear upper bounds would be interesting and, from a lower-bounds perspective, a bound of $\Omega(N)$ would be asymptotically the strongest possible.

18.2 Specific Two-Player Communication Games

18.2.1 Definitions

We shall now define a few two-player communication games that are canonical and often used in data stream lower bounds. Each of these is defined by a Boolean function. When an input to such a Boolean function is an N-bit string, we shall notate it like an N-dimensional vector: $\boldsymbol{x}$ for the whole string (vector) and $x_1, \ldots, x_N$ for its bits (entries).

The N-bit EQUALITY game asks the players to compute the function $\text{EQ}_N : \{0,1\}^N \times \{0,1\}^N \to \{0,1\}$, defined by

$$\text{EQ}_N(\boldsymbol{x},\boldsymbol{y}) = \begin{cases} 1, & \text{if } \boldsymbol{x} = \boldsymbol{y}, \\ 0, & \text{otherwise.} \end{cases}$$

Equivalently, using "$\oplus$" to denote the XOR operation,

$$\text{EQ}_N(\boldsymbol{x},\boldsymbol{y}) = \neg \bigvee_{i=1}^{N} x_i \oplus y_i \,. \tag{18.3}$$

For the N-bit INDEX game, Alice has an N-bit string while Bob has an integer in $[N]$ that indexes into this string. Their task is to compute the function $\text{IDX}_N : \{0,1\}^N \times [N] \to \{0,1\}$, defined by

$$\text{IDX}_N(\boldsymbol{x}, y) = x_y \,. \tag{18.4}$$

The N-bit SET-DISJOINTNESS game treats the two input fragments as characteristic vectors of subsets of $[N]$ and asks the players to decide whether these sets are disjoint. The corresponding function is $\text{DISJ}_N : \{0,1\}^N \times \{0,1\}^N \to \{0,1\}$, defined by

$$\text{DISJ}_N(\boldsymbol{x},\boldsymbol{y}) = \begin{cases} 1, & \text{if } \boldsymbol{x} \cap \boldsymbol{y} = \varnothing, \\ 0, & \text{otherwise.} \end{cases}$$

Equivalently,

$$\text{DISJ}_N(\boldsymbol{x},\boldsymbol{y}) = \neg \bigvee_{i=1}^{N} x_i \wedge y_i \,. \tag{18.5}$$

18.2.2 Results and Some Proofs: Deterministic Case

We summarize several key results about the communication complexity of the above games. We will encounter some more canonical communication games later.

Game	Model	Complexity bound	Notes
EQ_N	deterministic	$\text{D}(\text{EQ}_N) \geq N$	Theorem 18.2.4
EQ_N	one-way randomized	$\text{R}^{\text{priv},\to}(\text{EQ}_N) = O(\log N)$	Theorem 18.2.5
IDX_N	one-way deterministic	$\text{D}^{\to}(\text{IDX}_N) \geq N$	Theorem 18.2.1
IDX_N	one-way randomized	$\text{R}^{\to}(\text{IDX}_N) = \Omega(N)$	Theorem 18.2.7
IDX_N	deterministic	$\text{D}^2(\text{IDX}_N) \leq \lceil \log N \rceil$	trivial
DISJ_N	randomized	$\text{R}(\text{DISJ}_N) = \Omega(N)$	

Let's see a few things that we can prove easily.

Theorem 18.2.1. *$\text{D}^{\to}(\text{IDX}_N) \geq N$.*

Proof. Let Π be a one-way deterministic protocol for IDX_N in which $\mathscr{M}$ is the set of all possible messages from Alice, $\text{msg} : \{0,1\}^N \to \mathscr{M}$ is Alice's message function, and $\text{out} : [N] \times \mathscr{M} \to \{0,1\}$ is Bob's output function. By the correctness of Π, for all $(\boldsymbol{x}, y) \in \{0,1\}^N \times [N]$, we have

$$\text{out}(y, \text{msg}(\boldsymbol{x})) = \text{IDX}_N(\boldsymbol{x}, y) = x_y \,.$$

Given a message string $\mathfrak{m} \in \mathscr{M}$ from Alice, Bob can use it N times, pretending that his input is 1, then 2, and so on, to recover all the bits of $\boldsymbol{x}$. Formally, the function $\text{rec} : \mathscr{M} \to \{0,1\}^N$ defined by

$$\text{rec}(\mathfrak{m}) = (\text{out}(1,\mathfrak{m}), \text{out}(2,\mathfrak{m}), \ldots, \text{out}(N,\mathfrak{m})) \tag{18.6}$$

satisfies $\operatorname{rec}(\operatorname{msg}(\boldsymbol{x})) = \boldsymbol{x}$, for all $\boldsymbol{x} \in \{0,1\}^N$. Thus, rec is the inverse function of msg, which means that msg is bijective, implying $|\mathscr{M}| = |\{0,1\}^N| = 2^N$.

Suppose that the cost of Π is $\ell \le N-1$. Then every message in $\mathscr{M}$ is at most $N-1$ bits long, implying

$$|\mathscr{M}| \le 2^1 + 2^2 + \cdots + 2^{N-1} \le 2^N - 2,$$

a contradiction. □

Theorem 18.2.2. $\mathrm{D}^{\rightarrow}(\textsc{eq}_N) \ge N$ *and* $\mathrm{D}^{\rightarrow}(\textsc{disj}_N) \ge N$.

Proof sketch. Similarly to the proof of Theorem 18.2.1, Bob can use Alice's message in a correct protocol Π to recover her input entirely. As before, this implies that the cost of Π is at least N. □

To generalize the above result to non-round-restricted protocols, we need a key combinatorial fact called the *rectangle property* of deterministic communication protocols. First, let's define the *transcript* $\Pi(x,y)$ of a protocol Π on a particular input (x,y) to be the sequence of bits communicated when the protocol executes, followed by the output bits. By definition, protocols have to be self-punctuating, in the sense that any prefix of the transcript should determine which player speaks next. Thus, the cost of a protocol is the maximum length of a transcript, not counting the output bits.

Theorem 18.2.3 (Rectangle property)**.** *Let τ be a possible transcript of a deterministic two-player protocol Π on input domain $\mathscr{X} \times \mathscr{Y}$. Then the set $\{(x,y) \in \mathscr{X} \times \mathscr{Y} : \Pi(x,y) = \tau\}$ is of the form $A \times B$, where $A \subseteq \mathscr{X}$ and $B \subseteq \mathscr{Y}$.*

Thus, if $\Pi(x,y) = \Pi(x',y') = \tau$, then also $\Pi(x',y) = \Pi(x,y') = \tau$. In particular, if Π computes a function f and $\Pi(x,y) = \Pi(x',y')$ then $f(x,y) = f(x',y') = f(x',y) = f(x,y')$.

Proof sketch. Given any prefix of a possible transcript, the set of inputs consistent with that prefix is a rectangle (i.e., of the form $A \times B$), as can be proved by an easy induction on the length of the prefix. □

Theorem 18.2.4. $\mathrm{D}(\textsc{eq}_N) \ge N$ *and* $\mathrm{D}(\textsc{disj}_N) \ge N$.

Proof. We'll prove this part way, leaving the rest as an exercise.

Consider a protocol Π for $\textsc{eq}_N$. An input of the form $(\boldsymbol{x},\boldsymbol{x})$, where $\boldsymbol{x} \in \{0,1\}^N$, is called a *positive* input. Suppose $\boldsymbol{w} \ne \boldsymbol{x}$. Then $\textsc{eq}_N(\boldsymbol{w},\boldsymbol{x}) \ne \textsc{eq}_N(\boldsymbol{x},\boldsymbol{x})$, so by Theorem 18.2.3, $\Pi(\boldsymbol{w},\boldsymbol{w}) \ne \Pi(\boldsymbol{x},\boldsymbol{x})$. It follows that Π has at least 2^N different possible transcripts corresponding to the 2^N positive inputs. Then, reasoning as in the proof of Theorem 18.2.1, at least one of these transcripts must be N bits long. Subtracting off the one output bit leaves us with a communication cost of at least $N-1$. Therefore, $\mathrm{D}(\textsc{eq}_N) \ge N-1$.

The proof of $\textsc{disj}_N$ is similar, using a well-chosen subset of positive inputs. □

18.2.3 More Proofs: Randomized Case

The most remarkable thing about the EQUALITY game is that it is maximally expensive for a deterministic protocol whereas, for randomized protocols, it admits a rather cheap and one-way protocol. There are a few different ways to see this, but in the context of data stream algorithms, the most natural way to obtain the upper bound is to use the fingerprinting idea we encountered in Section 9.2, when studying 1-sparse recovery.

Theorem 18.2.5. $\mathrm{R}^{\text{priv},\rightarrow}(\textsc{eq}_N) = O(\log N)$.

Proof. The players agree on a finite field $\mathbb{F}$ with $|\mathbb{F}| = \Theta(N^2)$ and Alice picks a uniformly random element $r \in_R \mathbb{F}$. Given an input $\boldsymbol{x}$, Alice computes the fingerprint

$$p := \phi(\boldsymbol{x};r) := \sum_{i=1}^{n} x_i r^i\,,$$

which is an evaluation of the polynomial $\phi(\boldsymbol{x};Z)$ at $Z = r$. She sends (r,p) to Bob, spending $2\lceil \log|\mathbb{F}| \rceil = O(\log N)$ bits. Bob outputs 1 if $\phi(\boldsymbol{y};r) = p$ and 0 otherwise.

If $\boldsymbol{x} = \boldsymbol{y}$, Bob always correctly outputs 1. If $\boldsymbol{x} \ne \boldsymbol{y}$, Bob may output 1 wrongly, but this happens only if $\phi(\boldsymbol{x};r) = \phi(\boldsymbol{y};r)$, i.e., r is a root of the nonzero polynomial $\phi(\boldsymbol{x}-\boldsymbol{y};Z)$. Since this latter polynomial has degree at most N, by Theorem 9.3.1, it has at most N roots. So the probability of an error is at most $N/|\mathbb{F}| = O(1/N)$. □

To obtain randomized communication *lower bounds*, we need a fundamental reasoning tool for randomized algorithms, known as Yao's minimax principle. To set this up, we need to consider a different mode of computation, where the goal is not to get a correctness guarantee on *every* input, but only on an "average" input. In the context of communication games, consider a distribution μ on the domain of a function f. A deterministic protocol Π computes f up to error δ with respect to μ if

$$\mathbb{P}_{(X,Y)\sim\mu}\left\{\operatorname{out}^{\Pi}(X,Y)\neq f(X,Y)\right\}\leq\delta\,. \tag{18.7}$$

Even though we're considering random inputs when talking of Π's correctness, when it comes to cost, it will be convenient to continue to use the worst-case number of bits communicated. The *δ-error μ-distributional communication complexity* $\mathrm{D}^{\mu}_{\delta}(f)$ is defined to be the minimum cost of a protocol satisfying eq. (18.7). Incidentally, the restriction to deterministic protocols above is without loss of generality, since if a randomized protocol were to achieve a guarantee analogous to eq. (18.7), we could pick the setting of its random string that minimizes the left-hand side and produce a deterministic protocol satisfying eq. (18.7).

Theorem 18.2.6 (Yao's minimax principle for communication complexity)**.** $\mathrm{R}_{\delta}(f)=\max_{\mu}\mathrm{D}^{\mu}_{\delta}(f)$*, where the maximum is over all probability distributions μ on the domain of f.*

Proof. Given a randomized protocol Π with cost C satisfying eq. (18.1) and an input distribution μ, Π also satisfies the analog of eq. (18.7). As observed above, by suitably fixing Π's random string, we obtain a deterministic protocol Π' with cost C satisfying eq. (18.7). This proves that $\mathrm{R}_{\delta}(f)\geq\mathrm{D}^{\mu}_{\delta}(f)$ for every μ. Therefore, LHS $\geq$ RHS.

The proof that LHS $\leq$ RHS is more complicated and we won't go into it here. The standard argument, available in a textbook, goes through the von Neumann minimax principle for zero-sum games, which itself is a consequence of the linear programming duality theorem. Fortunately, the weaker statement that LHS $\geq$ RHS is often all we need for proving lower bounds. □

Armed with this simple yet powerful tool, we prove our first randomized communication lower bound.

Theorem 18.2.7. $\mathrm{R}^{\to}(\mathrm{IDX}_N)=\Omega(N)$.

Proof. Let μ be the uniform distribution on $\{0,1\}^N\times[N]$. Thanks to eq. (18.2) and Theorem 18.2.6, we can prove our lower bound for $\mathrm{D}^{\mu,\to}_{1/10}(\mathrm{IDX}_N)$ instead. To this end, let $(\boldsymbol{X},Y)\sim\mu$ and let Π be a deterministic protocol with cost C such that

$$\mathbb{P}\left\{\operatorname{out}^{\Pi}(\boldsymbol{X},Y)\neq X_Y\right\}\leq\frac{1}{10}\,.$$

By two applications of Markov's inequality, there exist sets $\mathscr{X}'\subseteq\{0,1\}^N$ and $\mathscr{Y}'\subseteq[N]$ such that

$$|\mathscr{X}'|\geq\frac{2^N}{2}\,,\ |\mathscr{Y}'|\geq\frac{8N}{10}\,,\text{ and }\forall\,(\boldsymbol{x},y)\in\mathscr{X}'\times\mathscr{Y}':\ \operatorname{out}^{\Pi}(\boldsymbol{x},y)=x_y\,. \tag{18.8}$$

Let the set $\mathscr{M}$ and the functions $\operatorname{msg}\colon\{0,1\}^N\to\mathscr{M}$ and $\operatorname{rec}\colon\mathscr{M}\to\{0,1\}^N$ be as in the proof of Theorem 18.2.1; recall that rec the input recovery function defined in eq. (18.6). By eq. (18.8),

$$\forall\boldsymbol{x}\in\mathscr{X}':\ \|\operatorname{rec}(\operatorname{msg}(\boldsymbol{x}))-\boldsymbol{x}\|_1\leq\frac{2N}{10}\,.$$

It follows that if $\boldsymbol{w},\boldsymbol{x}\in\mathscr{X}'$ are such that $\operatorname{rec}(\operatorname{msg}(\boldsymbol{w}))=\operatorname{rec}(\operatorname{msg}(\boldsymbol{x}))$, then by a triangle inequality,

$$\|\boldsymbol{w}-\boldsymbol{x}\|_1\leq\|\boldsymbol{w}-\operatorname{rec}(\operatorname{msg}(\boldsymbol{w}))\|_1+\|\operatorname{rec}(\operatorname{msg}(\boldsymbol{w}))-\operatorname{rec}(\operatorname{msg}(\boldsymbol{x}))\|_1+\|\operatorname{rec}(\operatorname{msg}(\boldsymbol{x}))-\boldsymbol{x}\|_1\leq\frac{4N}{10}\,.$$

An elementary fact in coding theory, readily proved by estimating the volume of a Hamming ball, is that a set with cardinality as large as that of $\mathscr{X}'$ contains a subset $\mathscr{X}''$ with $|\mathscr{X}''|=2^{\Omega(N)}$ that is a code with distance greater than $4N/10$, i.e., for all distinct $\boldsymbol{w},\boldsymbol{x}\in\mathscr{X}''$, we have $\|\boldsymbol{w}-\boldsymbol{x}\|_1>4N/10$.

By the above reasoning, the function $\operatorname{rec}\circ\operatorname{msg}$ is injective on such a code $\mathscr{X}''$. In particular, msg is injective on $\mathscr{X}''$. Therefore, $|\mathscr{M}|\geq|\mathscr{X}''|=2^{\Omega(N)}$ and so, one of the messages in $\mathscr{M}$ must have length $\Omega(N)$. □

We did not try to optimize constants in the above proof, but it should be clear that by a careful accounting, one can obtain $\mathrm{R}^{\to}_{\delta}(\mathrm{IDX}_N)\geq c_{\delta}N$, for some specific constant c_{δ}. The optimal constant turns out to be $c_{\delta}=1-\mathrm{H}_2(\delta)$, where $\mathrm{H}_2(x)=-x\log x-(1-x)\log(1-x)$ is the *binary entropy* function.

18.3 Data Streaming Lower Bounds

It is time to connect all of the above with the data streaming model. The starting point is the following easy observation that a streaming algorithm gives rise to some natural communication protocols.

Let $\mathscr{A}$ be a streaming algorithm that processes an input stream σ using S bits of space and p passes, producing some possibly random output $\mathscr{A}(\sigma)$. If we cut σ into t contiguous pieces $\sigma_1, \ldots, \sigma_t$, not necessarily of equal lengths, and give each piece to a distinct player (player i gets σ_i), this sets up a t-player communication game: the common goal of the players is to evaluate $\mathscr{A}(\sigma)$.

There is a natural protocol for this game that simply simulates $\mathscr{A}$. Player 1 begins by processing the tokens in σ_1 as in $\mathscr{A}$, keeping track of $\mathscr{A}$'s memory contents. When she gets to the end of her input, she sends these memory contents as a message to player 2 who continues the simulation by processing the tokens in σ_2, and so on. After player t is done, we have reached the end of the stream σ. If there are more passes to be executed, player t sends $\mathscr{A}$'s memory contents to player 1 so she can begin the next pass. Eventually, after p such passes, player t is able to produce the output $\mathscr{A}(\sigma)$. This construction is illustrated in fig. 18.1 using three players.

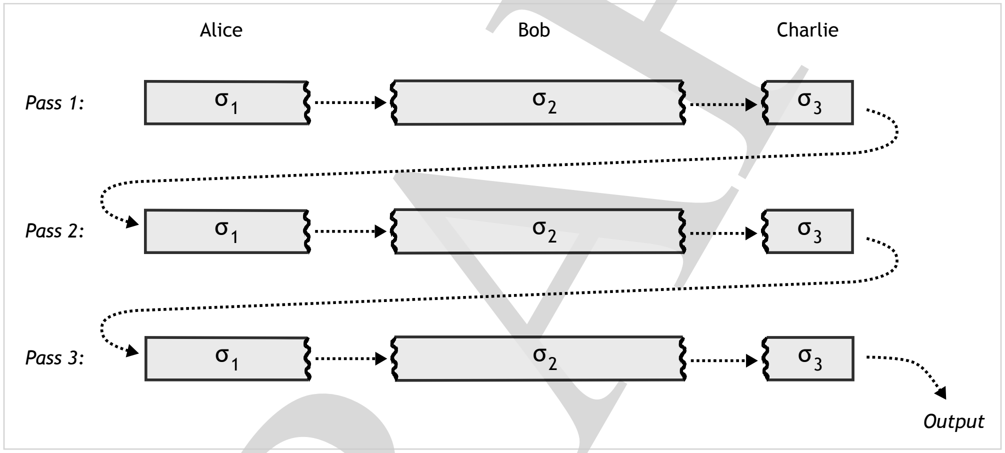


Figure 18.1: Communication protocol that simulates a streaming algorithm

Notice that this protocol causes $tp - 1$ messages to be communicated and has a total cost of $(tp-1)S$. In particular, if $t = 2$ and $p = 1$ (the most common case), the protocol is one-way.

We instantiate these abstract observations using two familiar data streaming problems.

18.3.1 Lower Bound for Majority

Recall our standard notation for data stream problems. The input σ is a stream of m tokens, each from the universe $[n]$.

Theorem 18.3.1. *A one-pass algorithm for* MAJORITY *requires* $\Omega(\min\{m,n\})$ *space.*

Proof. We shall prove the (stronger) result that even the simpler problem MAJ-DETECT, which asks *whether* the input vanilla stream contains a token of frequency greater than $m/2$, already requires this much space.

Let $\mathscr{A}$ be a one-pass S-space algorithm for MAJ-DETECT. We use the construction above to reduce from the two-player IDX_N problem.

Given an input $(\boldsymbol{x}, y)$ for IDX_N, Alice creates the stream $\sigma_1 = (a_1, \ldots, a_N)$, where $a_i = 2i - x_i$, while Bob creates the stream $(b, b, \ldots, b)$ of length N where $b = 2y - 1$. Clearly, the only possible majority element in the combined stream $\sigma := \sigma_1 \circ \sigma_2$ is b. Observe that b is in fact a majority iff $a_y = 2y - 1$, i.e., $\text{IDX}_N(\boldsymbol{x}, y) = x_y = 1$. Therefore, simulating $\mathscr{A}$ enables Alice and Bob to solve IDX_N using one S-bit message from Alice to Bob.

By Theorem 18.2.7, $S = \Omega(N)$. By construction, the stream σ has $m = n = 2N$. Therefore, we have proven a lower bound of $\Omega(N)$ for all settings where $m \geq 2N$ and $n \geq 2N$. In particular, we have shown that $S = \Omega(\min\{m,n\})$. ☐

18.3.2 Lower Bound for Frequency Estimation

Recall that the FREQUENCY-ESTIMATION problem with parameter ε asks us to process an input stream σ and answer point queries to the frequency vector $\boldsymbol{f} = \boldsymbol{f}(\sigma)$: given a query $j \in [n]$, we should output an estimate $\hat{f}_j$ that lies in the interval $[f_j - \varepsilon \|\boldsymbol{f}\|_1, f_j + \varepsilon \|\boldsymbol{f}\|_1]$ with probability at least $2/3$. The Misra-Gries algorithm from Unit 1 solves this for vanilla and cash register streams, while the Count/Median algorithm from Unit 5 solves it for turnstile streams. In each case, the space usage is $\widetilde{O}(\varepsilon^{-1})$.

Two natural question arise. First, *must* we estimate in order to achieve sublinear space? Can't we get exact answers? Second, if we wanted an accuracy of ε in the above sense, can the space usage be any better than $O(\varepsilon^{-1})$? We have the tools to answer these questions.

Theorem 18.3.2. *A one-pass algorithm for* FREQUENCY-ESTIMATION *that achieves an accuracy parameter ε must use $\Omega(\min\{m, n, \varepsilon^{-1}\})$ space. In particular, in order to get exact answers—i.e., $\varepsilon = 0$—it must use $\Omega(\min\{m, n\})$ space.*

Proof sketch. An idea similar to the above, reducing from IDX_N for an appropriate N, works. The details are left to the reader as an instructive exercise. □

Exercises

18-1 Write out a formal proof of Theorem 18.2.2 analogous to the above proof of Theorem 18.2.1.

18-2 Complete the proof of Theorem 18.2.4.

18-3 Formally prove Theorem 18.3.2.

Unit 19

Further Reductions and Lower Bounds

We have seen how space lower bounds on data streaming algorithms follow from lower bounds on communication complexity. In Section 18.3, we gave two basic streaming lower bounds, each using a reduction from the INDEX problem.

We shall now give several additional data streaming lower bounds, almost all of which are based on the communication games introduced in Unit 18.

19.1 The Importance of Randomization

Quite a large number of the data streaming algorithms we have studied are randomized. In fact, randomization is usually *crucial* and we can prove that no deterministic algorithm can achieve similar space savings. The most common way to prove such a result is by reduction from the EQUALITY communication game: recall, from Section 18.3, that this game has high deterministic communication complexity but much lower randomized communication complexity.

The most basic result of this sort is for the DISTINCT-ELEMENTS problem, i.e., F_0. It holds even in a vanilla streaming model and even if we are allowed multiple passes over the input stream.

Theorem 19.1.1. *A deterministic p-pass streaming algorithm that solves the* DISTINCT-ELEMENTS *problem exactly must use space* $\Omega(\min\{m,n\}/p)$.

Proof. Given an input $(\boldsymbol{x},\boldsymbol{y})$ for EQ_N, Alice creates the stream $\sigma = (a_1,\ldots,a_N)$, where $a_i = 2i - x_i$, while Bob creates the stream $\tau = (b_1,\ldots,b_N)$, where $b_i = 2i - y_i$. Observe that

$$F_0(\sigma\circ\tau) = N + \|\boldsymbol{x}-\boldsymbol{y}\|_1 = N + d \text{ (say)}.$$

If $\boldsymbol{x}=\boldsymbol{y}$, then $d = 0$, else $d \geq 1$. Thus, by computing $F_0(\sigma\circ\tau)$ exactly, the players can solve EQ_N. Notice that the stream $\sigma\circ\tau$ is of length $m = 2N$ and its tokens come from a universe of size $n = 2N$.

By the reasoning in Section 18.3, two players simulating a deterministic p-pass algorithm that uses S bits of space would use $(2p-1)S$ bits of communication. Thus, by Theorem 18.2.4, if DISTINCT-ELEMENTS had such an algorithm, then $(2p-1)S \geq N$, i.e., $S = \Omega(N/p) = \Omega(\min\{m,n\}/p)$. □

There is something unsatisfactory about the above result. When we designed algorithms for DISTINCT-ELEMENTS in Units 2 and 3, our solutions were randomized *and* they provided approximate answers. The above lower bound does not show that *both* relaxations are necessary. However, we can strengthen the result by a simple application of error correcting codes.

Theorem 19.1.2. *There is a positive constant ε_* such that every deterministic p-pass streaming algorithm that $(\varepsilon_*,0)$-estimates* DISTINCT-ELEMENTS *requires $\Omega(\min\{m,n\}/p)$ space.*

Proof. We reuse the elementary fact in coding theory that we encountered in the proof of Theorem 18.2.7. This time, we use it in the following form: for all large enough integers M, there exists a code $\mathscr{C} \subseteq \{0,1\}^M$ with distance greater than $3M/10$ and cardinality $|\mathscr{C}| \ge 2^{M/10}$. This enables us to solve EQ_N as follows.

Take $M = 10N$ and let $\mathscr{C}$ be the above code. Fix an injective function enc: $\{0,1\}^N \to \mathscr{C}$; it must exist, thanks to the lower bound on $|\mathscr{C}|$. Given an instance $(\boldsymbol{x},\boldsymbol{y})$ of EQ_N, Alice and Bob apply the construction in the proof of Theorem 19.1.1 to vectors $\mathrm{enc}(\boldsymbol{x})$ and $\mathrm{enc}(\boldsymbol{y})$, obtaining streams σ and τ, respectively. As above,

$$F_0(\sigma \circ \tau) = M + \|\mathrm{enc}(\boldsymbol{x}) - \mathrm{enc}(\boldsymbol{y})\|_1 = \begin{cases} = M\,, & \text{if } \boldsymbol{x} = \boldsymbol{y}\,,\\ \ge M + \dfrac{3M}{10}\,, & \text{if } \boldsymbol{x} \ne \boldsymbol{y}\,. \end{cases}$$

Given an $(\varepsilon_*, 0)$-estimate for $F_0(\sigma \circ \tau)$, where $(1+\varepsilon_*)/(1-\varepsilon_*) < 13/10$ (in particular, $\varepsilon_* = 1/8$ works), Alice and Bob can distinguish which of the two cases above has occurred and can therefore determine $\mathrm{EQ}_N(\boldsymbol{x},\boldsymbol{y})$. As before, we conclude a space lower bound of $\Omega(N/p) = \Omega(M/p) = \Omega(\min\{m,n\}/p)$, where m and n have their usual meanings. □

In fact, a still further strengthening is possible. Chakrabarti and Kale [CK16] showed that $\Omega(\min\{m,n\}/(Ap))$ space is required to estimate DISTINCT-ELEMENTS within a multiplicative factor of A, even for $A \gg 1$. This result is considerably harder to prove and necessarily so, for reasons explored in the exercises.

19.2 Multi-Pass Lower Bounds for Randomized Algorithms

Thus far, we have obtained data streaming lower bounds via reduction from either the INDEX problem (which becomes easy with two rounds of communication allowed) or the EQUALITY problem (which becomes easy with randomized communication allowed). To prove randomized *and* multi-pass lower bounds, we need a more powerful communication complexity result. The following celebrated result about the SET-DISJOINTNES game, which we defined in Section 18.2.1, is the source of most such lower bounds.

Theorem 19.2.1. $\mathrm{R}(\mathrm{DISJ}_N) = \Omega(N)$. □

Theorem 19.2.1 is one of the most important results in communication complexity, with far reaching applications (well beyond data streaming). It was first proved by Kalyanasundaram and Schnitger [KS92]. Their proof was conceptually simplified by Razborov [Raz92] and by Bar-Yossef et al. [BJKS04]. Nevertheless, even the simpler proofs of this result are much too elaborate for us to cover in these notes, so we refer the interested reader to textbooks instead [KN97, RY20].

19.2.1 Graph Problems

As a first application of Theorem 19.2.1, let us show that for the basic graph problem CONNECTEDNESS, $\Omega(n/p)$ space is required to handle n-vertex graphs using p passes, even under vanilla streaming. This shows that aiming for semi-streaming space, as we did in Unit 14, was the right thing to do.

Theorem 19.2.2. *Every (possibly randomized) p-pass streaming algorithm that solves* CONNECTEDNESS *on n-vertex graphs in the vanilla streaming model must use $\Omega(n/p)$ space.*

Proof. We reduce from DISJ_N. Given an instance $(\boldsymbol{x},\boldsymbol{y})$, Alice and Bob construct edge sets A and B for a graph on vertex set $V = \{s,t,v_1,\ldots,v_N\}$, as follows.

$$A = \{\{s,v_i\} : x_i = 0, i \in [N]\}\,,$$
$$B = \{\{v_i,t\} : y_i = 0, i \in [N]\} \cup \{\{s,t\}\}\,.$$

Let $G = (V, A \cup B)$. It is easy to check that if $\boldsymbol{x} \cap \boldsymbol{y} = \varnothing$, then G is connected. On the other hand, if there is an element $j \in \boldsymbol{x} \cap \boldsymbol{y}$, i.e., $x_j = y_j = 1$, then v_j is an isolated vertex and so G is disconnected.

It follows that Alice and Bob can simulate a streaming algorithm for CONNECTEDNESS to solve DISJ_N. Graph G has $n = N+2$ vertices. Thus, as before, we derive a space lower bound of $\Omega(N/p) = \Omega(n/p)$. □

The exercises explore some other lower bounds that are proved along similar lines.

19.2.2 The Importance of Approximation

Theorems such as theorem 19.1.2 convey the important message that randomization is crucial to certain data stream computations. Equally importantly, approximation is also crucial. We can prove this by a reduction from DISJ to the exact-computation version of the problem at hand. Here is a classic example, from the classic paper of Alon, Matias, and Szegedy [AMS99].

Theorem 19.2.3. *Every (possibly randomized) p-pass streaming algorithm that solves* DISTINCT-ELEMENTS *exactly requires* $\Omega(\min\{m,n\}/p)$ *space.*

Proof. Given an instance $(\boldsymbol{x},\boldsymbol{y})$ of DISJ_N, Alice and Bob construct streams $\sigma = (a_1,\ldots,a_N)$ and $\tau = (b_1,\ldots,b_N)$, respectively, where, for each $i \in [N]$:

$$a_i = 3i + x_i - 1\,;$$
$$b_i = 3i + 2y_i - 2\,.$$

This construction has the following easily checked property.

$$F_0(\sigma \circ \tau) = 2N - |\boldsymbol{x} \cap \boldsymbol{y}|\,.$$

Thus, an algorithm that computes $F_0(\sigma \circ \tau)$ exactly solve DISJ_N. The combined stream has length $m = 2N$ and its tokens come from a universe of size $n = 3N$. The theorem now follows along the usual lines. □

Arguing along similar lines, one can show that approximation is necessary in order to compute any frequency moment F_k in sublinear space, with the exception of F_1, which is just a count of the stream's length. It is an instructive exercise to work out the details of the proof and observe where it breaks when $k = 1$.

19.3 Space versus Approximation Quality

[*** Lower bounds via GHD, to be inserted here ***]

Exercises

19-1 Consider a graph stream describing an unweighted, undirected n-vertex graph G. Prove that $\Omega(n^2)$ space is required to determine, in one pass, whether or not G contains a triangle, even with randomization allowed.

19-2 A lower bound argument based on splitting a stream into *two* pieces and assigning each piece to a player in a communication game can never prove a result of the sort mentioned just after the proof of Theorem 19.1.2. Show that using such an argument, when $A \geq \sqrt{2}$, we cannot rule out sublinear-space solutions for producing an estimate $\hat{d} \in [A^{-1}d, Ad]$, where d is the number of distinct elements in the input stream.

19-3 Prove that computing F_2 exactly, in p streaming passes with randomization allowed, requires $\Omega(\min\{m,n\}/p)$ space. Generalize the result to the exact computation of F_k for any real constant $k > 0$ except $k = 1$.

19-4 The *diameter* of a graph $G = (V,E)$ is defined as $\text{diam}(G) = \max\{d_G(x,y) : x,y \in V\}$, i.e., the largest vertex-to-vertex distance in the graph. A real number $\hat{d}$ satisfying $\text{diam}(G) \leq \hat{d} \leq \alpha\,\text{diam}(G)$ is called an α-approximation to the diameter.

Suppose that $1 \leq \alpha < 1.5$. Prove that, in the vanilla graph streaming model, a 1-pass randomized algorithm that α-approximates the diameter of a *connected* graph must use $\Omega(n)$ space. How does the result generalize to p passes?

Bibliography

[AG13] Kook Jin Ahn and Sudipto Guha. Linear programming in the semi-streaming model with application to the maximum matching problem. *Inf. Comput.*, 222:59–79, 2013.

[AGM12] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Analyzing graph structure via linear measurements. In *Proc. 23rd Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 459–467, 2012.

[AHL02] Noga Alon, Shlomo Hoory, and Nathan Linial. The moore bound for irregular graphs. *Graphs and Combinatorics*, 18(1):53–57, 2002.

[AHPV04] Pankaj K. Agarwal, Sariel Har-Peled, and Kasturi R. Varadarajan. Approximating extent measures of points. *J. ACM*, 51(4):606–635, 2004.

[AHPV05] Pankaj K. Agarwal, Sariel Har-Peled, and Kasturi R. Varadarajan. Geometric approximation via coresets. Available online at http://valis.cs.uiuc.edu/~sariel/research/papers/04/survey/survey.pdf, 2005.

[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. Preliminary version in *Proc. 28th Annual ACM Symposium on the Theory of Computing*, pages 20–29, 1996.

[BFP$^+$73] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.

[BJK$^+$02] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, D. Sivakumar, and Luca Trevisan. Counting distinct elements in a data stream. In *Proc. 6th International Workshop on Randomization and Approximation Techniques in Computer Science*, pages 128–137, 2002.

[BJKS04] Ziv Bar-Yossef, T. S. Jayram, Ravi Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. *J. Comput. Syst. Sci.*, 68(4):702–732, 2004.

[BKS02] Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proc. 13th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 623–632, 2002.

[CCFC04] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Theor. Comput. Sci.*, 312(1):3–15, 2004.

[CK16] Amit Chakrabarti and Sagar Kale. Strong fooling sets for multi-player communication with applications to deterministic estimation of stream statistics. In *Proc. 57th Annual IEEE Symposium on Foundations of Computer Science*, pages 41–50, 2016.

[CM05] Graham Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. Alg.*, 55(1):58–75, 2005. Preliminary version in *Proc. 6th Latin American Theoretical Informatics Symposium*, pages 29–38, 2004.

[CMS76] J. M. Chambers, C. L. Mallows, and B. W. Stuck. A method for simulating stable random variables. *J. Amer. Stat. Assoc.*, 71(354):340–344, 1976.

[CS14] Michael Crouch and Daniel Stubbs. Improved streaming algorithms for weighted matching, via unweighted matching. In *Proc. 17th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, pages 96–104, 2014.

[Edm65] Jack Edmonds. Paths, trees, and flowers. *Canad. J. Math.*, 17:449–467, 1965.

[ELMS11] L. Epstein, A. Levin, J. Mestre, and D. Segev. Improved approximation guarantees for weighted matching in the semi-streaming model. *SIAM Journal on Discrete Mathematics*, 25(3):1251–1265, 2011.

[FKM$^+$05] Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. On graph problems in a semi-streaming model. *Theor. Comput. Sci.*, 348(2–3):207–216, 2005. Preliminary version in *Proc. 31st International Colloquium on Automata, Languages and Programming*, pages 531–543, 2004.

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

[FM88] Péter Frankl and Hiroshi Maehara. The Johnson-Lindenstrauss lemma and the sphericity of some graphs. *J. Combin. Theory Ser. B*, 44(3):355–362, 1988.

[GI10] Anna C. Gilbert and Piotr Indyk. Sparse recovery using sparse matrices. *Proceedings of the IEEE*, 98(6):937–947, 2010.

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

[JL84] W. B. Johnson and J. Lindenstrauss. Extensions of Lipschitz mapping into Hilbert space. *Contemp. Math.*, 26:189–206, 1984.

[JW18] Rajesh Jayaram and David P. Woodruff. Perfect lp sampling in a data stream. In *Proc. 59th Annual IEEE Symposium on Foundations of Computer Science*, pages 544–555, 2018.

[KN97] Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, Cambridge, 1997.

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

[KS92] Bala Kalyanasundaram and Georg Schnitger. The probabilistic communication complexity of set intersection. *SIAM J. Disc. Math.*, 5(4):547–557, 1992.

[Lév54] Paul Lévy. *Théorie de l'addition des variables aléatoires*. Jacques Gabay, 2e edition, 1954.

[McG05] Andrew McGregor. Finding graph matchings in data streams. In *Proc. 8th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, pages 170–181, 2005.

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

[MP80] J. Ian Munro and Mike Paterson. Selection and sorting with limited storage. *TCS*, 12:315–323, 1980. Preliminary version in *Proc. 19th Annual IEEE Symposium on Foundations of Computer Science*, pages 253–258, 1978.

[MW10] Morteza Monemizadeh and David P. Woodruff. 1-Pass relative-error l_p-sampling with applications. In *Proc. 21st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1143–1160, 2010.

[Nis90] Noam Nisan. Pseudorandom generators for space-bounded computation. In *Proc. 22nd Annual ACM Symposium on the Theory of Computing*, pages 204–212, 1990.

[PS19] Ami Paz and Gregory Schwartzman. A (2+ε)-approximation for maximum weight matching in the semi-streaming model. *ACM Trans. Alg.*, 15(2):18:1–18:15, 2019. Preliminary version in *Proc. 28th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 2153–2161, 2017.

[Raz92] Alexander Razborov. On the distributional complexity of disjointness. *Theor. Comput. Sci.*, 106(2):385–390, 1992. Preliminary version in *Proc. 17th International Colloquium on Automata, Languages and Programming*, pages 249–253, 1990.

[RY20] Anup Rao and Amir Yehudayoff. *Communication Complexity: and Applications*. Cambridge University Press, 2020.

[Sr.78] Robert H. Morris Sr. Counting large numbers of events in small registers. *Commun. ACM*, 21(10):840–842, 1978.

[Zel08] Mariano Zelke. Weighted matching in the semi-streaming model. In *Proc. 25th International Symposium on Theoretical Aspects of Computer Science*, pages 669–680, 2008.