CS49: Data Stream Algorithms

Lecture Notes, Fall 2011

Amit Chakrabarti
Dartmouth College

Latest Update: December 19, 2012

Acknowledgements

These lecture notes began as rough scribe notes for a Fall 2009 offering of the course "Data Stream Algorithms" at Dartmouth College. The initial scribe notes were prepared mostly by students enrolled in the course in 2009. Subsequently, during a Fall 2011 offering of the course, I edited the notes heavily, bringing them into presentable form, with the aim being to create a resource for students and other teachers of this material.

I would like to acknowledge the initial effort by the 2009 students that got these notes 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.

Contents

Lecture 0

Preliminaries: The Data Stream Model

0.1 The Basic Setup

In this course, we shall 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

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

The holy grail is to achieve

$$s = O(\log m + \log n) ,$$

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 $s = \text{polylog}(\min\{m, n\})$, where $f(n) = \text{polylog}(g(n))$ 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," i.e., we do not have random access to the tokens. 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 — will usually be 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 $\mathcal{A}(\sigma)$ denote the output of a randomized streaming algorithm $\mathcal{A}$ on input σ; note that this is a random variable. Let ϕ be the function that $\mathcal{A}$ is supposed to compute. We say that the algorithm (ε, δ)-approximates

ϕ if we have

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

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 $\mathcal{A}$ is said to (ε, δ)-additively-approximate ϕ if we have

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

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 $\mathbf{f} = (f_1, f_2, \ldots, f_n)$, where

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

In other words, σ implicitly defines this vector $\mathbf{f}$, and we are then interested in computing some function of the form $\Phi(\mathbf{f})$. 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 $\mathbf{f}$.

With this in mind, it is interesting to consider more general updates to $\mathbf{f}$: 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 $\mathbf{f}$ 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

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

A special case of the turnstile model, that is sometimes important to consider, is the *strict turnstile model*, in which we assume that $\mathbf{f} \ge 0$ 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 $c > 0$.

Lecture 1

Finding Frequent Items Deterministically

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 $\mathbf{f} = (f_1, \ldots, f_n)$. 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 lecture, 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 a is given to us only *after* we have processed σ.

1.2 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 uses a parameter k that controls the quality of the answers it gives. (Note: 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.

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

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

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

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

pairs in A at any time, the total space required is $O(k(\log m + \log n))$.

Now consider the quality of the algorithm's output. Let us pretend that A consists of n key/value pairs, with $A[j] = 0$ whenever j is not actually stored in A by the algorithm. Notice that the counter $A[j]$ is incremented only when we process an occurrence of j in the stream. Thus, $\hat{f}_j \le f_j$. On the other hand, whenever $A[j]$ is decremented (in lines 7-8, we pretend that $A[j]$ is incremented from 0 to 1, and then immediately decremented back to 0), we also decrement $k - 1$ other counters, corresponding to distinct tokens in the stream. Thus, each decrement of $A[j]$ is "witnessed" by a collection of k distinct tokens (one of which is a j itself) from the stream. Since the stream consists of m tokens, there can be at most m/k such decrements. Therefore, $\hat{f}_j \ge f_j - m/k$. 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*

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

Using this 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 $keys(A)$. Thus, we can make a second pass over the input stream, counting exactly the frequencies f_j for all $j \in keys(A)$, and then output the desired set of items.

Lecture 2

Estimating the Number of Distinct Elements

2.1 The Problem

As in the last lecture, 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 $\mathbf{f} = (f_1, \ldots, f_n)$. 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)-approximation (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 lecture, we give a simple algorithm for this problem that has interesting, but not optimal, quality guarantees. The idea behind the algorithm is originally due to Flajolet and Martin [FM85], and we give a slightly modified presentation, due to Alon, Matias and Szegedy [AMS99]. We shall refer to this as the "AMS Algorithm" for DISTINCT-ELEMENTS; note that there are two other (perhaps more famous) algorithms from the same paper that are also referred to as "AMS Algorithm," but those are for other problems.

2.2 The Algorithm

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

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

We use the following very simple algorithm.

```
Initialize :
1   Choose a random hash function $h : [n] \to [n]$ from a 2-universal family ;
2   $z \leftarrow 0$ ;
Process $j$:
3   if zeros$(h(j)) > z$ then $z \leftarrow$ 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 zeros$(h(j)) \geq \log d$, and we don't expect any tokens to hit zeros$(h(j)) \gg \log d$. Thus, the maximum value of 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 $r \geq 0$, 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. Clearly,

$$Y_r > 0 \iff t \geq r \,. \tag{2.1}$$

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

$$Y_r = 0 \iff t \leq r - 1 \,. \tag{2.2}$$

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

$$\mathbb{E}[X_{r,j}] = \Pr[\text{zeros}(h(j)) \geq r] = \Pr[2^r \text{ divides } h(j)] = \frac{1}{2^r} \,.$$

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 Y_r, which follows from the 2-universality of the hash family from which h is drawn.

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

$$\text{Var}[Y_r] = \sum_{j:\, f_j>0} \text{Var}[X_{r,j}] \leq \sum_{j:\, f_j>0} \mathbb{E}[X_{r,j}^2] = \sum_{j:\, f_j>0} \mathbb{E}[X_{r,j}] = \frac{d}{2^r} \,. \tag{2.3}$$

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

$$\Pr[Y_r > 0] = \Pr[Y_r \geq 1] \leq \frac{\mathbb{E}[Y_r]}{1} = \frac{d}{2^r} \,, \text{ and} \tag{2.4}$$

$$\Pr[Y_r = 0] = \Pr\left[|Y_r - \mathbb{E}[Y_r]| \geq d/2^r\right] \leq \frac{\text{Var}[Y_r]}{(d/2^r)^2} \leq \frac{2^r}{d} \,. \tag{2.5}$$

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

$$\Pr\left[\hat{d} \geq 3d\right] = \Pr[t \geq a] = \Pr[Y_a > 0] \leq \frac{d}{2^a} \leq \frac{\sqrt{2}}{3} \,.$$

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

$$\Pr\left[\hat{d} \leq d/3\right] = \Pr[t \leq b] = \Pr[Y_{b+1} = 0] \leq \frac{2^{b+1}}{d} \leq \frac{\sqrt{2}}{3} \,.$$

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 above 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 is 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 $(O(1), \delta)$-approximation to d. Later, we shall give a different algorithm that will provide an (ε, δ)-approximation 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.

Lecture 3

A Better Estimate for Distinct Elements

3.1 The Problem

We revisit the DISTINCT-ELEMENTS problem from last time, 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 $\mathbf{f} = (f_1, \ldots, f_n)$. 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+04]. 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.

```
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 $j$:
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 AMS 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 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 $O(1/\varepsilon^2)$ 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 $1/\varepsilon^2 = o(m)$: 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 Homework 1 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 Homework 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

$$\begin{aligned} Y_t &= \text{value of } |B| \text{ when algorithm finishes}, \\ \Rightarrow \quad \hat{d} &= Y_t 2^t . \end{aligned}$$

Proceeding as in Section 2.3, we obtain

$$\mathbb{E}[Y_r] = \frac{d}{2^r}\,; \quad \text{Var}[Y_r] \leq \frac{d}{2^r}\,. \tag{3.1}$$

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

$$\begin{aligned}
\Pr[\text{FAIL}] &= \sum_{r=1}^{\log n} \Pr\left[\left|Y_r - \frac{d}{2^r}\right| \ge \frac{\varepsilon d}{2^r} \bigwedge t = r\right] \\
&\le \sum_{r=1}^{s-1} \Pr\left[\left|Y_r - \frac{d}{2^r}\right| \ge \frac{\varepsilon d}{2^r}\right] + \sum_{r=s}^{\log n} \Pr[t = r] \\
&= \sum_{r=1}^{s-1} \Pr\left[\,|Y_r - \mathbb{E}[Y_r]| \ge \varepsilon d/2^r\,\right] + \Pr[t \ge s] \qquad \text{(by (3.1))} \\
&= \sum_{r=1}^{s-1} \Pr\left[\,|Y_r - \mathbb{E}[Y_r]| \ge \varepsilon d/2^r\,\right] + \Pr[Y_{s-1} \ge c/\varepsilon^2]\,. \qquad (3.3)
\end{aligned}$$

Now we bound the first term in (3.3) using Chebyshev's inequality and the second term using Markov's inequality. Then we use (3.1) to compute

$$\begin{aligned}
\Pr[\text{FAIL}] &\le \sum_{r=1}^{s-1} \frac{\operatorname{Var}[Y_r]}{(\varepsilon d/2^r)^2} + \frac{\mathbb{E}[Y_{s-1}]}{c/\varepsilon^2} \\
&\le \sum_{r=1}^{s-1} \frac{2^r}{\varepsilon^2 d} + \frac{\varepsilon^2 d}{c2^{s-1}} \\
&\le \frac{2^s}{\varepsilon^2 d} + \frac{2\varepsilon^2 d}{c2^s} \\
&\le \frac{1}{\varepsilon^2} \cdot \frac{\varepsilon^2}{12} + \frac{2\varepsilon^2}{c} \cdot \frac{24}{\varepsilon^2} \qquad \text{(by (3.2))} \\
&\le \frac{1}{6}\,,
\end{aligned}$$

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 optimal in a fairly strong sense. Later in this course, when we study lower bounds, we shall show both an $\Omega(\log n)$ and a an $\Omega(1/\varepsilon^2)$ bound on the space required by an algorithm that $(\varepsilon, \frac{1}{3})$-approximates the number of distinct elements. In fact, an even stronger lower bound that simultaneously involves n and ε is known (ref?).

Lecture 4

Finding Frequent Items via Sketching

4.1 The Problem

We return to the FREQUENT problem that we studied in Lecture 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.

4.2 Sketches and Linear Sketches

Let MG(σ) denote the data structure computed by Misra-Gries upon processing the stream σ. One drawback of this data structure is that there is no general way to 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 4.2.1. 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

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

Each of the algorithms of this lecture has the nice property that it computes a sketch of the input stream in the above sense. Since these algorithms are computing functions of the frequency vector $\mathbf{f}(\sigma)$ determined by σ, their sketches will naturally be functions of $\mathbf{f}(\sigma)$; in fact, they will be *linear* functions. That's special enough to call out in another definition.

Definition 4.2.2. 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 $\mathbf{f}(\sigma)$. 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).

Besides not producing a sketch, Misra-Gries also has the drawback that it does not seem to extend to the turnstile (or even strict turnstile) model. But the algorithms in this lecture do, because 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 $\mathbf{v}_j$ to the sketch, then an update (j, c) in the turnstile model is handled by adding $c\mathbf{v}_j$ to the sketch: this handles both cases $c \geq 0$ and $c < 0$.

4.3 The Count Sketch

We now describe the first of our sketching algorithms, called "Count Sketch", 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.

```
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 $(j, c)$:
4   $C[h(j)] \leftarrow C[h(j)] + cg(j)$ ;
Output :
5   On query $a$, 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.

4.3.1 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, we have

$$\mathbb{E}[g(j)Y_j] = \mathbb{E}[g(j)]\,\mathbb{E}[Y_j] = 0 \cdot \mathbb{E}[Y_j] = 0 \, . \tag{4.1}$$

Therefore, by linearity of expectation, we have

$$\mathbb{E}[X] = f_a + \sum_{j \in [n] \setminus \{a\}} f_j g(a)\,\mathbb{E}[g(j)Y_j] = f_a \, . \tag{4.2}$$

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

$$\mathbb{E}[Y_j^2] = \mathbb{E}[Y_j] = \Pr[h(j) = h(a)] = \frac{1}{k} \, . \tag{4.3}$$

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

$$\mathbb{E}[g(i)g(j)Y_iY_j] = \mathbb{E}[g(i)]\,\mathbb{E}[g(j)]\,\mathbb{E}[Y_iY_j] = 0 \cdot 0 \cdot \mathbb{E}[Y_iY_j] = 0 \, . \tag{4.4}$$

Thus, we calculate

$$\begin{aligned}
\mathrm{Var}[X] &= 0 + g(a)^2\,\mathrm{Var}\left[\sum_{j\in[n]\setminus\{a\}} f_j g(j) Y_j\right] \\
&= \mathbb{E}\left[\sum_{j\in[n]\setminus\{a\}} f_j^2 Y_j^2 + \sum_{\substack{i,j\in[n]\setminus\{a\}\\ i\neq j}} f_i f_j g(i) g(j) Y_i Y_j\right] - \left(\sum_{j\in[n]\setminus\{a\}} f_j\,\mathbb{E}[g(j)Y_j]\right)^2 \\
&= \sum_{j\in[n]\setminus\{a\}} \frac{f_j^2}{k} + 0 - 0 \qquad \text{(by (4.3), (4.4), and (4.1))} \\
&= \frac{\|\mathbf{f}\|_2^2 - f_a^2}{k}\,, \qquad (4.5)
\end{aligned}$$

where $\mathbf{f} = \mathbf{f}(\sigma)$ is the frequency distribution determined by σ. From (4.2) and (4.5), using Chebyshev's inequality, we obtain

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

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

$$\Pr\left[\,|\hat{f}_a - f_a| \geq \varepsilon\|\mathbf{f}_{-a}\|_2\,\right] \leq \frac{1}{3}\,. \qquad (4.6)$$

4.3.2 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.

```
  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 $g_1,\ldots g_t : [n] \to [k]$, each from a 2-universal family ;
  Process $(j,c)$:
4   for $i = 1$ to $t$ do $C[i][h_i(j)] \leftarrow C[i][h_i(j)] + c g_i(j)$ ;
  Output :
5   On query $a$, report $\hat{f}_a = \mathrm{median}_{1\leq i\leq t}\, g_i(a) C[i][h_i(a)]$ ;
```

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

$$\Pr\left[\,|\hat{f}_a - f_a| \geq \varepsilon\|\mathbf{f}_{-a}\|_2\,\right] \leq \delta\,. \qquad (4.7)$$

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

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

4.4 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.

```
  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 $(j, c)$:
3   for $i = 1$ to $t$ do $C[i][h_i(j)] \leftarrow C[i][h_i(j)] + c$ ;
  Output :
4   On query $a$, 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.

4.4.1 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,

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

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

$$\Pr[X_i \ge \varepsilon \|\mathbf{f}_{-a}\|_1] \le \frac{\|\mathbf{f}_{-a}\|_1}{k\varepsilon \|\mathbf{f}_{-a}\|_1} = \frac{1}{2} ,$$

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,

$$\begin{aligned}
\Pr\left[\hat{f}_a - f_a \ge \varepsilon\|\mathbf{f}_{-a}\|_1\right] &= \Pr\left[\min\{X_1,\ldots,X_t\} \ge \varepsilon\|\mathbf{f}_{-a}\|_1\right] \\
&= \Pr\left[\bigwedge_{i=1}^{t} (X_i \ge \varepsilon\|\mathbf{f}_{-a}\|_1)\right] \\
&= \prod_{i=1}^{t} \Pr[X_i \ge \varepsilon\|\mathbf{f}_{-a}\|_1] \\
&\le \frac{1}{2^t}\,.
\end{aligned}$$

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

$$f_a \;\le\; \hat{f}_a \;\le\; f_a + \varepsilon\|\mathbf{f}_{-a}\|_1\,,$$

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 $\varepsilon\|\mathbf{f}_{-a}\|_1$, rather than $\varepsilon\|\mathbf{f}_{-a}\|_2$. For all vectors $z \in \mathbb{R}^n$, we have $\|z\|_1 \ge \|z\|_2$. The inequality is tight when z has a single nonzero entry. It is at its weakest when all entries of 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".

4.5 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**	**Error Probability**	**Model**
Misra-Gries	$\left[-\varepsilon\|\mathbf{f}_{-a}\|_1,\, 0\right]$	$O\left(\frac{1}{\varepsilon}(\log m + \log n)\right)$	0 (deterministic)	Cash register
Count Sketch	$\left[-\varepsilon\|\mathbf{f}_{-a}\|_2,\, \varepsilon\|\mathbf{f}_{-a}\|_2\right]$	$O\left(\frac{1}{\varepsilon^2}\log\frac{1}{\delta}\cdot(\log m + \log n)\right)$	δ (overall)	Turnstile
Count-Min Sketch	$\left[0,\, \varepsilon\|\mathbf{f}_{-a}\|_1\right]$	$O\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}\cdot(\log m + \log n)\right)$	δ (upper bound only)	Cash register
Count/Median	$\left[-\varepsilon\|\mathbf{f}_{-a}\|_1,\, \varepsilon\|\mathbf{f}_{-a}\|_1\right]$	$O\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}\cdot(\log m + \log n)\right)$	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 absolute values of the relevant counters, rather than the minimun. It is a simple (and instructive) exercise to analyze this algorithm and prove the claims in the last row.

Lecture 5

Estimating Frequency Moments

5.1 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 $\mathbf{f} = \mathbf{f}(\sigma) = (f_1, \ldots, f_n)$. 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:

$$F_k := \sum_{j=1}^{n} f_j^k = \|\mathbf{f}\|_k^k . \tag{5.1}$$

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.

5.2 The 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.

```
  Initialize : $(m, r, a) \leftarrow (0, 0, 0)$ ;
  Process $j$:
1 $m \leftarrow m + 1$ ;
2 $\beta \leftarrow$ random bit with $\Pr[\beta = 1] = 1/m$ ;
3 if $\beta = 1$ then
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.

5.3 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 $\Pr[a = j] = f_j/m$ 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,

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

And

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

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

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

$$\mathrm{Var}[X] \;\le\; \mathbb{E}[X^2] \;=\; \sum_{j=1}^{n} \frac{f_j}{m} \sum_{i=1}^{f_j} \frac{1}{f_j} \cdot m^2 (i^k - (i-1)^k)^2 \;=\; m \sum_{j=1}^{n} \sum_{i=1}^{f_j} \left(i^k - (i-1)^k\right)^2 . \tag{5.2}$$

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

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

where the last step uses $k \ge 1$. Using this bound in (5.2), we get

$$\begin{aligned}
\mathrm{Var}[X] &\le m \sum_{j=1}^{n} \sum_{i=1}^{f_j} k i^{k-1} \left(i^k - (i-1)^k\right) \\
&\le m \sum_{j=1}^{n} k f_j^{k-1} \sum_{i=1}^{f_j} \left(i^k - (i-1)^k\right) \\
&= m \sum_{j=1}^{n} k f_j^{k-1} \cdot f_j^k \\
&= k F_1 F_{2k-1} .
\end{aligned} \tag{5.3}$$

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

Lemma 5.3.1. *Let $n > 0$ be an integer and let $x_1, \ldots, x_n \ge 0$ and $k \ge 1$ be reals. Then*

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

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

$$x_*^{k-1} \;=\; \left(x_*^k\right)^{(k-1)/k} \;\le\; \left(\sum x_i^k\right)^{(k-1)/k} . \tag{5.4}$$

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

$$\frac{1}{n} \sum x_i \;\le\; \left(\frac{1}{n} \sum x_i^k\right)^{1/k} . \tag{5.5}$$

Using (5.4) and (5.5) in the second and third steps (respectively) below, we compute

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

which completes the proof. □

Using the above lemma in (5.3), with $x_j = f_j$, we get

$$\mathrm{Var}[X] \;\le\; kF_1 F_{2k-1} \;\le\; kn^{1-1/k} F_k^2 . \tag{5.6}$$

We are now ready to present our final F_k estimator, which combines several independent basic estimators.

5.4 The Median-of-Means Improvement

Unfortunately, we cannot apply the median trick from Section 2.4 to our basic estimator directly. This is because its variance is so large that we are unable to bound below $\frac{1}{2}$ the probability of an ε relative deviation in the estimator. So we first bring the variance down by *averaging* a number of independent copies of the basic estimator, and *then* apply the median trick. The next theorem — a useful general-purpose theorem — quantifies this precisely.

Lemma 5.4.1. *There is a universal positive constant c such that the following holds. Let X be the distribution of 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*

$$t = c\log\frac{1}{\delta}\,, \quad \text{and} \quad k = \frac{3\,\mathrm{Var}[X]}{\varepsilon^2\,\mathbb{E}[X]^2}\,.$$

Let $Z = \mathrm{median}_{i\in[t]}\left(\frac{1}{k}\sum_{j=1}^{k} X_{ij}\right)$. Then, we have $\Pr[|Z-Q|\ge\varepsilon Q]\le\delta$.

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

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

Applying Chebyshev's inequality, we obtain

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

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 $\Pr[|Z-Q|\ge\varepsilon Q]\le\delta$. □

Using this lemma, we can build a *final estimator* from our basic AMS estimator, X, for F_k: We simply compute the median of $t = c\log(1/\delta)$ intermediate estimators, each of which is the mean of r basic estimators. To make the final estimator an (ε,δ)-approximation to F_k, we need

$$r = \frac{3\,\mathrm{Var}[X]}{\varepsilon^2\,\mathbb{E}[X]^2} \le \frac{3kn^{1-1/k}F_k^2}{\varepsilon^2 F_k^2} = \frac{3k}{\varepsilon^2}\cdot n^{1-1/k}\,,$$

where the inequality uses (5.6). This leads to a final space bound that is tr times the space usage of a basic estimator, i.e.,

$$\text{space} \le tr\cdot O(\log m+\log n) = O\left(\frac{1}{\varepsilon^2}\log\frac{1}{\delta}\cdot kn^{1-1/k}(\log m+\log n)\right)\,.$$

For the first time in the course, 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, which suppresses factors polynomial in $\log m$, $\log n$ and $\log(1/\delta)$. We can then remember this space bound as simply $\widetilde{O}(\varepsilon^{-2}n^{1-1/k})$, treating k as a constant.

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 later in the course.

Lecture 6

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 4.2.

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

6.1 The Basic Sketch

We describe the algorithm in the turnstile model.

```
Initialize :
1   Choose a random hash function $h : [n] \to \{-1, 1\}$ from a 4-universal family ;
2   $x \leftarrow 0$ ;
Process $(j, c)$:
3   $x \leftarrow x + ch(j)$ ;
Output : $x^2$
```

The sketch is simply the random variable x. It is pulled in the positive direction by those tokens j with $h(j) = 1$, 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 x 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.

6.1.1 The Quality of the Estimate

Let X denote the value of x after the algorithm has processed σ. For convenience, define $Y_j = h(j)$ for each $j \in [n]$. Then $X = \sum_{j=1}^{n} f_j Y_j$. Therefore,

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

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), and then the fact that $\mathbb{E}[Y_j] = 0$ for all $j \in [n]$. This shows that the algorithm's output, X^2, is indeed an unbiased estimator for F_2.

The variance of the estimator is $\mathrm{Var}[X^2] = \mathbb{E}[X^4] - \mathbb{E}[X^2]^2 = \mathbb{E}[X^4] - F_2^2$. We bound this as follows. By linearity of expectation, we have

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

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 $\mathbb{E}[Y_i Y_j Y_k Y_\ell] = \mathbb{E}[Y_i]\,\mathbb{E}[Y_j Y_k Y_\ell] = 0$, 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

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

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

$$\begin{aligned} \mathrm{Var}[X^2] &= F_4 - F_2^2 + 6 \sum_{i=1}^n \sum_{j=i+1}^n f_i^2 f_j^2 \\ &= F_4 - F_2^2 + 3\left(\Big(\sum_{j=1}^n f_j^2 \Big)^2 - \sum_{j=1}^n f_j^4 \right) \\ &= F_4 - F_2^2 + 3(F_2^2 - F_4) \leq 2F_2^2 . \end{aligned}$$

6.2 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 5.4.1, this will blow up the space usage by a factor of

$$\frac{O(1) \cdot \mathrm{Var}[X^2]}{\varepsilon^2 \, \mathbb{E}[X^2]^2} \cdot \log \frac{1}{\delta} \leq \frac{O(1) \cdot 2F_2^2}{\varepsilon^2 F_2^2} \cdot \log \frac{1}{\delta} = O\left(\frac{1}{\varepsilon^2} \log \frac{1}{\delta} \right)$$

in order to give an (ε, δ)-approximation. 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.

6.2.1 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 $M \in \mathbb{R}^{t \times n}$ be the matrix that "transforms" the frequency vector $\mathbf{f}$ into the t-dimensional sketch vector $\mathbf{x}$. Note that 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 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 5.4.1 after the Chebyshev step (and before the "median trick" Chernoff step), we obtain that

$$\Pr_M \left[\left| \frac{1}{t} \sum_{i=1}^t x_i^2 - F_2 \right| \geq \varepsilon F_2 \right] \leq \frac{1}{3} .$$

Thus, with probability at least $2/3$, we have

$$\left\| \frac{1}{\sqrt{t}} M\mathbf{f} \right\|_2 = \frac{1}{\sqrt{t}} \|\mathbf{x}\|_2 \in \left[\sqrt{1-\varepsilon} \cdot \|\mathbf{f}\|_2, \sqrt{1+\varepsilon} \|\mathbf{f}\|_2 \right] \subseteq [(1-\varepsilon)\|\mathbf{f}\|_2, (1+\varepsilon)\|\mathbf{f}\|_2] .$$

This can be interpreted as follows. The (random) matrix $M/\sqrt{t}$ performs a "dimension reduction", reducing an n-dimensional vector $\mathbf{f}$ to a t-dimensional sketch $\mathbf{x}$ (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 simply 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 $\|\mathbf{f}(\sigma) - \mathbf{f}(\sigma')\|_2$ between two streams σ and σ'.

Lecture 7

Estimating Norms Using Stable Distributions

As noted at the end of Lecture 6, 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 variational 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.

7.1 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 $M\mathbf{f}$, where M is a JL Sketch matrix. The pseudocode below shows the operations involved.

```
Initialize :
1   Choose $Y_1, \ldots, Y_n$ independently, each from $\mathcal{N}(0,1)$ ;
2   $x \leftarrow 0$ ;
Process $(j, c)$:
3   $x \leftarrow x + cY_j$ ;
Output : $x^2$
```

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

7.2 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 7.2.1. 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 $X \sim \mathcal{D}_p$ is independent of $X_1, \ldots, X_n$ 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 [**?**] 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 [**?**] 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 $\|\mathbf{f}\|_p$. Note that the scaling factor $\|\mathbf{f}\|_p$ 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.

```
Initialize :
1     $M[1\ldots t][1\ldots n] \leftarrow tn$ independent samples from $\mathcal{D}_p$, where $t = O(\varepsilon^{-2}\log(\delta^{-1}))$ ;
2     $x[1\ldots t] \leftarrow \vec{0}$ ;
Process $(j, c)$:
3     for $i = 1$ to $t$ do $x[i] \leftarrow x[i] + cM[i][j]$ ;
Output : $\mathrm{median}_{1\le i\le t}\, |x_i| / \mathrm{median}(|\mathcal{D}_p|)$ ;
```

7.3 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 $\mathcal{D}$ be an absolutely continuous distribution, let ϕ be its density function, and let $X \sim \mathcal{D}$. A median of $\mathcal{D}$ is a real number μ that satisfies

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

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 $\mathcal{D}$, we will denote this unique median as $\mathrm{median}(\mathcal{D})$.

For a distribution $\mathcal{D}$, with density function ϕ, we denote by $|\mathcal{D}|$ the distribution of the absolute value of a random variable drawn from $\mathcal{D}$. It is easy to show that the density function of $|\mathcal{D}|$ 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 $X \sim \mathcal{D}_p$, 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 X_i denote the final value of x_i after the above algorithm has processed σ. By the earlier discussion, and the defintion of p-stability, we see that $X_i \equiv \|\mathbf{f}\|_p X$, where $X \sim \mathcal{D}_p$. Therefore, $|X_i| / \mathrm{median}(|\mathcal{D}_p|)$ has a distribution whose density function is $\phi_{p,\lambda}$, where $\lambda = \|\mathbf{f}\|_p / \mathrm{median}(|\mathcal{D}_p|) = \|\mathbf{f}\|_p / \mu_{p,1}$. Thus, the median of this distribution is $\mu_{p,\lambda} = \lambda\mu_{p,1} = \|\mathbf{f}\|_p$.

The algorithm — which seeks to estimate $\|\mathbf{f}\|_p$ — 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.

7.4 The Accuracy of the Estimate

Lemma 7.4.1. *Let $\varepsilon > 0$, and let $\mathcal{D}$ 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 $\phi_* = \min\{\phi(x) : x \in [(1-\varepsilon)\mu, (1+\varepsilon)\mu]\}$. Let $Y = \mathrm{median}_{1\le i\le t} X_i$, where $X_1, \ldots, X_t$ are independent samples from $\mathcal{D}$. Then*

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

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

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

$$\begin{aligned}\Pr[X_i < (1-\varepsilon)\mu] &= \int_{-\infty}^{\mu} \phi(x)\,dx - \int_{(1-\varepsilon)\mu}^{\mu} \phi(x)\,dx \\ &= \frac{1}{2} - \Phi(\mu) + \Phi((1-\varepsilon)\mu) \\ &= \frac{1}{2} - \varepsilon\mu\phi(\xi)\,,\end{aligned}$$

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

$$\left(\frac{1}{2} - \varepsilon\mu\phi(\xi)\right)(1+\alpha) = \frac{1}{2}\,. \tag{7.1}$$

Let $N = |\{i \in [t] : X_i < (1-\varepsilon)\mu\}|$. 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 X_is must fall below λ. Therefore

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

by a standard Chernoff bound. Now, from (7.1), we derive $\mathbb{E}[N]\alpha = \varepsilon\mu\phi(\xi)$ and $\alpha \ge 2\varepsilon\mu\phi(\xi)$. Therefore

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

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 $\mu = \mu_{p,\lambda} = \|\mathbf{f}\|_p$, where $\lambda = \|\mathbf{f}\|_p/\mu_{p,1}$. Therefore,

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

which is a constant depending only on p: call it c_p. Thus, by Lemma 7.4.1, the output Y of the algorithm satisfies

$$\Pr\Big[\big|Y - \|\mathbf{f}\|_p\big| \ge \varepsilon\|\mathbf{f}\|_p\Big] \le \exp\left(-\frac{2}{3}\varepsilon^2 c_p^2 t\right) \le \delta\,,$$

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

7.5 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.

There is actually a third issue: there does not seem to be a theorem in the literature that proves that c_p (see the analysis above) is nonzero! We tacitly assumed this. For c_1, we can show this by explicit calculation, because we have a convenient closed-form expression for the density function of the Cauchy distribution. Indyk [Ind06] resolves this issue by changing the algorithm for ℓ_p ($p \neq 1$) slightly in a way that allows him to perform the necessary calculations. Have a look at his paper for the details on this, and the other two matters discussed above.

7.5.1 An Open Question

Life would be a lot easier if there were some way to generate entries of a suitable ℓ_p-sketching matrix M from $O(1)$-wise independent random variables, similar to the way 4-wise indepdent collections of ± 1-valued random variables worked for ℓ_2. Is there? Then we would not have to rely on the heavy hammer of Nisan's PRG, and perhaps we could make do with very simple arithmetic, as in the Tug-of-War Sketch.

Lecture 8

Estimating Norms via Precision Sampling

At this point, for each $p \in (0, 2]$, we have a polylogarithmic-sized sketch, computable in polylogarithmic space, that lets us estimate the ℓ_p-norm, $\|\mathbf{f}\|_p$, of the frequency vector $\mathbf{f}$ determined by an input stream in the general turnstile model. For $p > 2$, we have a cash register algorithm that estimates the frequency moment F_p, but it does not give a sketch, nor is its space usage optimal in terms of n.

A solution that estimates ℓ_p space-optimally via a linear sketch was given by Indyk and Woodruff [IW05], and this was simplified somewhat by Bhuvanagiri et al. [BGKS06]. Though the underlying technique in these algorithms is useful in many places, it is complicated enough that we prefer to discuss a more recent simpler solution, due to Andoni et al. [**?**]. Their solution uses an interesting new primitive, which they call *precision sampling*. For our purposes, we should think of the primitive as "sum estimation" instead.

8.1 The Basic Idea

At a very high level, the idea is to try to estimate f_j^p, for each token $j \in [n]$, using a frequency estimation sketch such as Count Sketch, which we saw in Lecture 4.

Lecture 9

Finding the Median

Scribe: Jon Denning

9.1 The Median / Selection Problem

We are working in the vanilla stream model. Given a sequence of numbers $\sigma =< a_1, a_2, \ldots, a_m >, a_i \in [n]$, assume a_i's are distinct, output the median.

Suppose that the stream is sorted. Then the median of the stream is the value in the middle position (when m is odd) or the average of the two on either side of the middle (when m is even). More concisely, the median is the average of the values at the $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$ indices. (Running two copies of our algorithm finds the mathematical median.)

This leads us to a slight generalization: finding the rth element in a sorted set.

Definition 9.1.1. Given a set S of elements from a domain D with an imposed order and an element $x \in D$, the **rank** of x is the count of elements in the set S that are not greater than x.

$$\text{rank}(x, S) = |\{y \in S : y \leq x\}|$$

This definition of rank makes sense even if x does not belong to S. Note: we are working with sets (no repeats), so assume they are all distinct. Otherwise, we'd have to consider the rank of an element in a multiset.

There are multiple ways of solving for the median. For example, sort array then pick the rth position. However, the running time is not the best, because sort is in $m \log m$ time (not linear).

There is a way to solve this in linear time. This is done by partitioning the array into 5 chunks, find median, then recurse to find median. The median of median will have rank in the whole array. So, we need to find the rank of this element. If it's $> r$ or $< r$, recurse on the right or left half. Want rank of $r - t$ if r is rank you want and t is rank of median.

Unfortunately, there's no way to solve this median problem in one pass as it is stated. There is a theorem that says, in one pass, selection requires $\Omega(\min\{m, n\})$ space (basically linear space).

Can we do something interesting in two passes? Yes! Like in the Misra-Gries algorithm, in two passes we have space usage down close to $\widetilde{O}(\sqrt{n})$ space (tilde hides some constant). It is similar but not in log. So, let's use more passes! In p passes, we can achieve $\widetilde{O}(n^{1/p})$ space. Making p large enough can "get rid of n." Let's use $p = O(\log n)$. In $O(\log n)$ passes, we're down to $\widetilde{O}(1)$ space.

We will give an algorithm that uses p passes, and uses space $O(n^{1/p})$. It will be difficult to state in pseudo code, but it is very elegant.

The concept is called i-sample ($i \in \mathbb{Z}$): from a set of numbers, construct a sample from it by selecting one in every n integers. For example, a 1-sample is about half the set, and a 2-sample is about a quarter of the set.

Definition 9.1.2. An i-sample of a population (set of numbers) of size $2^i s$ (s is some parameter) is a sorted sequence of length s defined recursively as follows, where $A \circ B$ is stream A followed by stream B:

$$\begin{array}{lcl} i = 0 & : & \text{0-sample}(A) = \text{sort}(A) \\ i > 0 & : & i\text{-sample}(A \circ B) = \\ & & = \text{sort}(\text{evens}((i-1)\text{-sample}(A)) \bigcup \text{evens}((i-1)\text{-sample}(B))) \end{array}$$

In other words, when $i > 0$, construct an $(i-1)$-sample of the first part, and $(i-1)$-sample of the second part (each sorted sequence of length s). Then combine them by picking every even positioned element in the two sets and insert in sorted order.

This algorithm says I want to run in a certain space bound, which defines what sample size can be done (can afford). Process stream and then compute sample. Making big thing into small thing takes many layers of samples.

Claim: this sample gives us a good idea of the median.
Idea: each pass shrinks the size of candidate medians

9.2 Munro-Paterson Algorithm (1980)

Given $r \in [m]$, find the element of stream with rank = r. The feature of Munro-Paterson Algorithm is it uses p passes in space $O(m^{1/p} \log^{2-2/p} m \log n)$. We have a notion of filters for each pass. At the start of the hth pass, we have filters a_h, b_h such that

$$a_h \leq \text{rank-}r \text{ element} \leq b_h.$$

Initially, we set $a_1 = -\infty, b_1 = \infty$. The main action of algorithm: as we read stream, compute i-sample. Let m_h = number of elements in stream between the filters, a_h and b_h. With a_1 and b_1 as above, $m_1 = m$. But as $a_h \rightarrow\leftarrow b_h$, m_i will shrink. During the pass, we will ignore any elements outside the interval a_h, b_h except to compute rank(a_h). Build a t-sample of size $s = S/\log m$, where S = target size, where t is such that $2^t s = m_h$. It does not matter if this equation is exactly satisfied; can always "pad" up.

Lemma 9.2.1. *let $x_1 < x_2 < \ldots < x_s$ be the i-sample (of size s) of a population P. $|P| = 2^i s$.*

To construct this sample in a streaming fashion, we'll have a working sample at each level and a current working sample s (an $(i-1)$-samples) that the new tokens go into. When the working samples become full $(i-1)$-samples, combine the two $(i-1)$-samples into an i-sample. This uses $2s$ storage. (i.e., new data goes into 0-sample. When it's full, the combined goes into 1-sample, and so on.)

At the end of the pass, what should we see in the sample? t-sample contains 2^t elements. Should be able to construct rank of elements up to $\pm 2^t$. In the final sample, we want rank r, so look at rank $r/2^t$.

Let's consider the jth element of this sample. We have some upper and lower bounds of the rank of this element. Note: with lower bounds $2^i j, 2^{i+1} j$ and upper bounds $2^i(i+j), 2^{i+1}(i+j)$, ranks can overlap.

$$2^i j \;\leq\; \text{rank}(x_j, P) \;\leq\; 2^i(i+j)$$

$$L_{ij} = \text{lower bound} = 2^i j, \qquad U_{ij} = \text{upper bound} = 2^i(i+j)$$

An example, suppose these are elements of sample with given ranks

$$\begin{array}{ccccccccc} & A & & B & & C & & D & \\ (- & - & - & - & -) & & & & \leftarrow \text{rank}(A) \\ & & (- & - & - & - & -) & & \leftarrow \text{rank}(B) \\ & & & & (- & - & - & - & -) \;\leftarrow \text{rank}(C) \\ & & & & & & (- - - - -) & & \leftarrow \text{rank}(D) \\ & & & & \uparrow & & & & \leftarrow \text{rank} = r \end{array}$$

A's upper bound is too small and D's lower bound is too large, so they will serve as the new filters. Update filters. If $a_h = b_h$, output a_h as answer.

Proof. We proceed by induction on i.

When $i = 0$, everything is trivially correct, because the sample is the whole population. $L_{ij} = U_{ij} = j$.

Consider the $(i+1)$-sample, where $i \geq 0$: $(i+1)$-sample is generated by joining two i-samples, a "blue" population and a "red" population. We made an i-sample from blue population and then picked every other element. Similarly for red population. Combined, they produce, for example,

$$i\text{-sample}_{\text{blue}} = b_1\, b_2\, b_3\, b_4\, b_5\, b_6\, b_7\, \underline{b_8}\, b_9\, b_{10}, \qquad i\text{-sample}_{\text{red}} = r_1\, r_2\, r_3\, r_4\, r_5\, r_6\, r_7\, r_8\, r_9\, r_{10}$$

$$(i+1)\text{-sample} = \text{sort}(b_2\, b_4\, b_6\, \underline{b_8}\, b_{10}\, r_2\, r_4\, r_6\, r_8\, r_{10}) = b_2\, b_4\, r_2\, b_6\, r_4\, \underline{b_8}\, r_6\, b_{10}\, r_8\, r_{10}.$$

Suppose, without loss of generality, the jth element of the $(i+1)$-sample is a b ($j = 6$, for example), so it must be the kth blue selected element from the blue i-sample ($k = 4$). This kth picked element means there are $2k$ elements up to k. Now, some must have come from the red sample ($j - k$ picked elements, or the $2(j-k)$th element in sample), because the $\cup$ is sorted from blue and red samples.

$\text{rank}(x_j, P) \geq L_{i,2k} + L_{i,2(j-k)}$ by inductive hypothesis on rank of elements in sample over their respective populations.

Combining all these lower bounds

$$L_{i+1,j} = \min_k (L_{i,2k} + L_{i,2(j-k)}) \tag{9.1}$$

Note: the kth element in the $(i+1)$-sample is the $2k$th element in the blue sample, so the lower bound of the kth element.

Upper bound: Let k be as before.

$$U_{i+1,j} = \max_k (U_{i,2k} + U_{i,2(j-k+1)-1}) \tag{9.2}$$

Have to go up two red elements to find the upper bound, because one red element up is uncertainly bigger or smaller than our blue element. (Check that L_{ij} and U_{ij} as defined by (9.1) and (9.2) satisfy lemma.) $\square$

Back to the problem, look at the upper bounds for a sample. At some point we are above the rank, $U_{t,j} < r$ but $U_{t,j+1} \geq r$. We know that all these elements (up to j) are less than the rank we're concerned with. In the final t-sample constructed, look for uth element where u is the min such that $U_{tu} \geq r$, i.e., $2^t(t+u) \geq r \Rightarrow t + u = \lceil r/2^t \rceil$. u is an integer, so $u = \lceil r/2^t \rceil - t$. Set $a \leftarrow u$th element in sample for the next pass.

Similarly, find the vth element where v is max such that $L_{tv} \leq r$, i.e., $2^t v \leq r \Rightarrow v = \lfloor r/2^t \rfloor$. Set $b \leftarrow v$th element for the next pass.

Based on the sample at the end of the pass, update filters to get a_{h+1} and b_{h+1}. Each pass reduces the number of elements under consideration by S, whittling it down by the factor $(m_h \log^2 m)/S$.

Lemma 9.2.2. *If there are n elements between filters at the start of a pass, then there are $O(n \log^2 n/S)$ elements between filters at the end of the pass.*

$$m_{h+1} = O\left(\frac{m_h \log^2 m}{S}\right) = O\left(\frac{m_h}{S/\log^2 m}\right)$$

$$m_h = O\left(\frac{m}{(S/\log^2 m)^{h-1}}\right)$$

We are done making passes when the number of candidates is down to S. At this point, we have enough space to store all elements between the filters and can find the rank element wanted.

The number of passes required p is such that, ignoring constants,

$$\begin{aligned} & \theta\left(\frac{m}{(S/\log^2 m)^{p-1}}\right) = \theta(S) \\ \Rightarrow \quad & m \log^{2p-2} m = S^p \\ \Rightarrow \quad & m^{1/p} \log^{2-2/p} m = S \end{aligned}$$

where S is the number of elements from the stream that we can store.

We're trading off the number of passes for the amount of space.

Note: this computes the exact median (or any rank element). Also, the magnitude of the elements makes no difference. We do a compare for $<$, but don't care how much less it is. This algorithm is immune to the magnitudes of the elements.

Lecture 10

Approximate Selection

Scribe: Alina Djamankulova

10.1 Two approaches

The selection problem is to find an element of a given rank r, where rank is a number of element that are less than or equal to the given element. The algorithm we looked at previously gives us the exact answer, the actual rank of an element. Now we are going to look at two different algorithms that take only one pass over data. And therefore give us an approximate answer. We'll give only rough detailes of the algorithms analysing them at very high level. Munro-Paterson algorithm we previously looked at considered this problem.

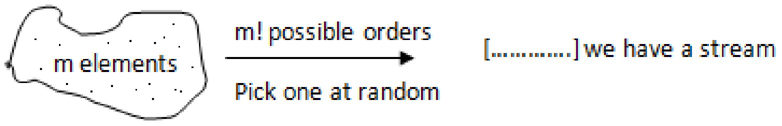
1. Given target rank r, return element of rank $\approx r$. The important note is that we are not looking for the approximate value of r element, which does not make a lot of sense.
   Let us define a notion of relative rank. Relative rank is computed the following way

   $$relrank(x, S) = \frac{rank(x, S)}{|S|}$$

   The relative rank is going to be some number between 0 and 1. For example, median is an element of relative rank 1/2. We can generalize this in terms of quantiles. Top quantile is the element of relative rank 1/4.
   Given target relative rank ϕ, return element of relative rank $\in [\phi - \varepsilon, \phi + \varepsilon]$
   (ε - approximation parameter, ϕ - quantile problem) [Greenwald - Khanna '01]

2. Random-order streams The idea is to create some really bad ordering of the elements in the stream. This does not mean that we literally mess the data. But we just suppose that we can get data in the worse order it can be. And if we can successfully solve this problem within the time and space limits, we are guaranteed to get good results, since data is very unlikely to come in that messy order.

   - Given a multiset of tokens, serialized into a sequence uniformly at random, and we need to compute the approximate rank of the element. We can think of the problem in the following way.
     We look at the multiset leaving the details of how we do it.
   - Given r $\in$ [m], return element of rank r with high probability with respect to the randomness n the ordering (say $\geq$ 1 - δ)
     (random order selection)[Guho - McGregor '07]

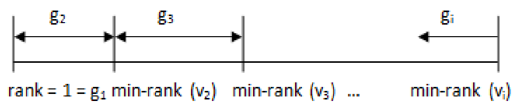


10.2 Greenwald - Khanna Algorithm

The algorithms has some slight similarities with Misra-Gries algoritms for solving frequent element problem. Maintain a collection of s tuples $(v_1, g_1, \Delta_1), ...(v_s, g_s, \Delta_s)$ where
v_i is a value from stream so far (δ)
g_i = min-rank (v_i) - min-rank ($v_i - 1$) // gain: how much more is this rank relative to the previously seen element. Storing g_i we can update easier and faster.
Minimum value seen so far and the maximum value seen so far will always be stored.
Δ_i = max-rank (v_i) - min-rank (v_i) // range
min (δ) = $v_1 < ... < v_s$ = max (δ) // seen so far, increasing order

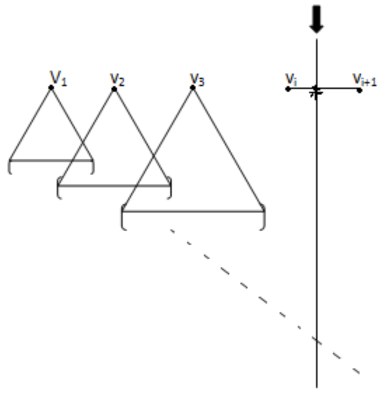
```
Initialize :
1  Starting reading elements from the stream;
2  $m \leftarrow 0$ ;
Process a token $v$:
3  Find $i$ such that $v_i < v < v_{i+1}$;
4  $v$ becomes the "new" $v_{i+1} \longrightarrow$ associated tuple = $(v, 1, \lfloor 2\varepsilon m \rfloor)$;
5  $m \leftarrow m + 1$;
6  if $v$ = new min/new max then
7      $\Delta \leftarrow 0$ // every $\frac{1}{\varepsilon}$ elements;
8  Check if compression is possible (withis some bound) and it it is, do the compression:
9  if $\exists$ $i$ such that $g_i + g_{i+1} + \Delta_{i+1} \leq \lfloor 2\varepsilon m \rfloor$ then
10     Remove $i^t h$ tuple;
11     Set $g_{i+1} \leftarrow g_i + g_{i+1}$;
12  Output : a random element that is filtered
```

Space: O($\frac{1}{\varepsilon}$ log εmlogn)
At any point, G-K algorithm can tell the rank of any element up to $\pm e$ where e = $max_i(\frac{g_i+\Delta_i}{2})$



min-rank (v_i) = $sum^i_{j=1} g_j$
max-rank (v_i) = $sum^i_{j=1} g_j + \Delta_i$

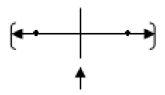
Identify i such that $v_i < v < v_{i+1}$
Then rank(v) $\in$ [min-rank(v_i), max-rank(v_{i+1})]
In this statement min-rank(v_i) = a and max-rank(v_{i+1}) = b. So we do approach to the real rank of the element with a and b filters.
rank (v) $\geq$ min-rank (v_i) = $sum^i_{j=1} g_j$
rank (v) $\leq$ max-rank (v_{i+1}) = $sum^i_{j=1} g_j + \Delta_{i+1}$



— rank range for v — $\leq g_{i+1} + \Delta_{i+1} \leq 2e$ If we announce rank(v) $\approx \frac{a+b}{2}$, then error $\leq \pm(\frac{b-a}{2}) \leq e$
Algorithm maintains the invariant $g_i + \Delta \mathrm{i} \leq 1 + \lfloor 2\varepsilon m \rfloor$. So ranks are known up to error $\frac{1+\lfloor 2\varepsilon m \rfloor}{2}$ = εm + 1/2

10.3 Guha-McGregor Algorithm

- Split stream into pieces: $\sigma =< S_1, E_1, S_2, E_2, ..., S_p, E_p >$
  S: sample
  E: Estimate
  One pass polylog space # passes for polylog space = O($\log\log m$) and m and p both O($\log m$)
- Maintain filters $a_n, b_n (1 \leq h \leq p)$ initially
  $a_i = -\infty, b_i = \infty$
  such that with high probability rel-rank - ϕ element ∈ (a_n, b_n)
- Sample phase: find first token u ∈ (a_n, b_n), $fromS_n$
- Estimate phase: compute r = rel-rank (u, E_n)
- Update filters, setting as in binary search
  True rank of u ∈ $[rm - \sqrt{l}, rm + \sqrt{l}]$ with high probability



Filter update

Lecture 11

Geometric Streams and Coresets

11.1 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.

11.2 Coresets and Their Properties

The idea of a coreset was first formulated in Agarwal, Har-Peled and Varadarajan [**?**]; 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 $C_P : \mathbb{R}^d \to \mathbb{R}_+$. We say that C is *monotone* if

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

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

$$C_\sigma(x) \ := \ \max_{y\in\sigma} \|x-y\|_2 \,, \tag{11.1}$$

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

Definition 11.2.1. 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

$$\forall\, T \subseteq \mathbb{R}^d \ \forall\, x \in \mathbb{R}^d : \ C_{Q\cup T}(x) \ \le \ C_{P\cup T}(x) \ \le \ \alpha C_{Q\cup T}(x)\,. \tag{11.2}$$

Clearly, if C is monotone, the left inequality always holds. The cost function for MEB, given by (11.1), 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.

11.3 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 11.3.1. *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.

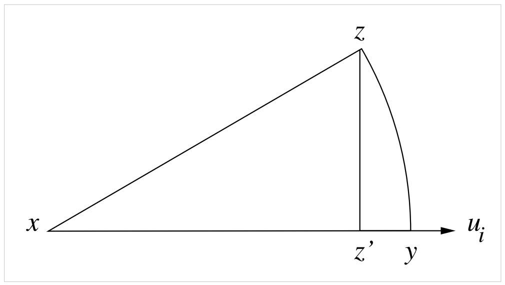
Theorem 11.3.2. *In $d \ge 2$ dimensions, the MEB cost function admits a $(1+\varepsilon)$-coreset of size $O(1/\varepsilon^{(d-1)/2})$.*

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 11.3.1, 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

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

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 (11.2) 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

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

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 (11.2) holds with $\alpha = 1 + \theta^2$. Since we wanted Q to be a $(1+\varepsilon)$-coreset, we may take $\theta = \sqrt{\varepsilon}$.

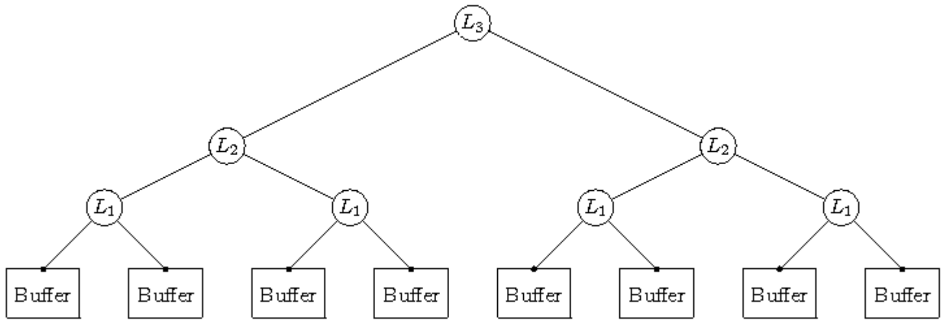
Finally, with this setting of θ, we have $|Q| \leq 2t = O(1/\varepsilon^{(d-1)/2})$, which completes the proof. □

11.4 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 11.4.1. *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 $\mathcal{A}_j$. By design, $\mathcal{A}_j$ creates a virtual stream that is fed into $\mathcal{A}_{j+1}$. Algorithm $\mathcal{A}_0$ 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 $\mathcal{A}_1$, and empties the buffer.

For $j \geq 1$, $\mathcal{A}_j$ receives a coreset at a time from $\mathcal{A}_{j-1}$. 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 $\mathcal{A}_{j+1}$, and discards both P and P'.

Thus, $\mathcal{A}_0$ uses space $O(B)$ and, for each $j \geq 1$, $\mathcal{A}_j$ uses space $O(A(\delta))$. The highest-numbered $\mathcal{A}_j$ 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. $\square$

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 $O(\varepsilon^{-1/2} \log^{3/2} m)$.

Lecture 12

Metric Streams and Clustering

In Lecture 11, 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.

12.1 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 12.1.1. 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 $d(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|\|_p$, 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.

12.2 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 $\Delta(\sigma, R)$. We can build these restrictions on R into the definition of Δ by setting $\Delta(\sigma, R) = \infty$ whenever R does not meet the restrictions. 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 a 4-approximation. The second will improve the approximation ratio from 4 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.

12.3 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_∞. Then we consider the algorithm's workings and establish certain invariants that it maintains.

Lemma 12.3.1. *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 $R \subseteq M$, we have $\Delta_\infty(\sigma, R) \geq t/2$.*

Proof. Suppose there exists $R \subseteq M$ with $\Delta_\infty(\sigma, R) < t/2$. Then $|R| \leq k$, and 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

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

```
Initialize :
1    $S \leftarrow$ first $k+1$ points in stream ;
2    $(x, y) \leftarrow \arg\min_{(u,v)\in S} d(u, v)$ ;
3    $\tau \leftarrow d(x, y)$ ;
4    $R \leftarrow S \setminus \{x\}$ ;
Process $x$:
5    if $\min_{r\in R} d(x, r) > 2\tau$ then
6        $R \leftarrow R \cup \{x\}$ ;
7        while $|R| > k$ do
8            $\tau \leftarrow 2\tau$ ;
9            $R \leftarrow$ maximal $R' \subseteq R$ such that $\forall\, r \neq s \in R' : d(r, s) \geq \tau$ ;
Output : $R$ ;
```

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 $\{x_1, \ldots, x_{k+1}\}$. □

Lemma 12.3.2. *The doubling algorithm maintains the following invariants at the start of each call to the* processing *section.*

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

Proof. At the end of initialization, Invariant 1 holds by definition of x, y, 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.

First, we consider the case that the **if** condition 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, suppose that the **if** 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. And 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 **while** loop at Line 7. Invariant 1 may be broken by Line 8 but is explicitly restored in Line 9. Just 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 $x \in \sigma$ be an arbitrary data point. Then $d(x, R) \leq \Delta_\infty(\sigma, R) \leq \tau$. Let $r' = \arg\min_{r\in R} d(x, r)$. If $r' \in R'$, then $d(x, R') \leq d(x, r') = d(x, R) \leq \tau$. Otherwise, by maximality of R', there exists a representative $s \in R'$ such that $d(r', s) < \tau$. Now

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

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 12.3.3. *The doubling algorithm uses $O(k)$ space and outputs a summary R whose cost is at most* 4 *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 12.3.2 (Invariant 2), we have $\Delta_\infty(\sigma, \hat{R}) \le 2\hat{\tau}$.

By the same lemma (Invariant 1), every pair of distinct points in $\hat{R}$ is at distance at least $\hat{\tau}$. Therefore, by Lemma 12.3.1, we have $\Delta_\infty(\sigma, R^*) \ge \hat{\tau}/2$. Putting these together, we have $\Delta_\infty(\sigma, \hat{R}) \le 4\Delta_\infty(\sigma, R^*)$. □

12.4 Metric Costs and Threshold Algorithms

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

Definition 12.4.1. A summarization cost function Δ is said to be *metric* if for all streams σ, π and summaries S, T, we have

$$\Delta(\sigma[S] \circ \pi, T) - \Delta(\sigma, S) \;\le\; \Delta(\sigma \circ \pi, T) \;\le\; \Delta(\sigma[S] \circ \pi, T) + \Delta(\sigma, S)\,. \tag{12.1}$$

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 (12.1).

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.

$$\underbrace{\circ \otimes \circ \circ \otimes \circ \otimes \circ \circ \circ}_{\sigma} \mid \underbrace{\circ \circ \circ \circ \circ \circ}_{\pi}$$

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 12.4.2. Let Δ be a summarization cost function and let $\alpha \ge 1$ 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 $\Delta(\sigma, S) \le \alpha t$.
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 12.3.1 guarantees that this is a 2-threshold algorithm.

12.5 Guha's 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 $\mathcal{A}$, for some $\alpha \ge 1$.

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

The idea behind Guha's algorithm is to run multiple copies of $\mathcal{A}$ in parallel, with geometrically increasing thresholds. Occasionally a copy of $\mathcal{A}$ will fail; when it does, we start a new copy of $\mathcal{A}$ 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 $\mathcal{A}$ 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 $\mathcal{A}$ 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 $\mathcal{A}$ that has the lowest threshold.

12.5.1 Space Bounds

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

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

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 12.4, we get $s_{\mathcal{A}} = O(k)$ and $\alpha = 2$. Therefore, for k-center, we have an algorithm running in space $O((k/\varepsilon)\log(1/\varepsilon))$.

12.5.2 The Quality of the Summary

Consider a run of Guha's algorithm on an input stream σ. Consider the instance of $\mathcal{A}$ 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 $\mathcal{A}$ 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

$$\Delta(S_{i-1} \circ \sigma_i, S_i) \;\le\; \alpha(\varepsilon/\alpha)^{j-i+1} t , \quad \text{for } 1 \le i \le j+1 , \tag{12.2}$$

where we put $S_0 = \varnothing$. Since Δ is metric, by (12.1), after the simplification $\sigma[S] = S$, we have

$$\Delta(\sigma_1 \circ \cdots \circ \sigma_i, S_i) \;\le\; \Delta(S_{i-1} \circ \sigma_i, S_i) + \Delta(\sigma_1 \circ \cdots \circ \sigma_{i-1}, S_{i-1}) . \tag{12.3}$$

Using (12.3) repeatedly, we can bound the cost of the algorithm's final summary as follows.

$$\begin{aligned}
\Delta(\sigma, S_{j+1}) \;=\; \Delta(\sigma_1 \circ \cdots \circ \sigma_{j+1}, S_{j+1}) \;&\le\; \sum_{i=1}^{j+1} \Delta(S_{i-1} \circ \sigma_i, S_i) \\
&\le\; \sum_{i=1}^{j+1} \alpha(\varepsilon/\alpha)^{j-i+1} t \qquad \text{(by (12.2))} \\
&\le\; \alpha t \sum_{i=0}^{\infty} \left(\frac{\varepsilon}{\alpha}\right)^i \;=\; (\alpha + O(\varepsilon)) t .
\end{aligned}$$

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

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

Strictly speaking, the above reasoning assumed that at least one instance of $\mathcal{A}$ 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 $2 + O(\varepsilon)$.

Lecture 13

Graph Streams: Basic Algorithms

13.1 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*. This motivates the study of graph algorithms that operate in streaming fashion: the input is a stream that describes a graph.

The model we shall work with in this course is that the input stream consists of tokens $(u, v) \in [n] \times [n]$, describing the edges of a simple[1] graph G on vertex set $[n]$. 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. The number n is known beforehand, but m, the length of the stream and the number of edges in G, is not. Though we can consider both directed and undirected graphs in this model, *we shall only be studying problems on undirected graphs*; so we may as well assume that the tokens describe doubleton sets $\{u, v\}$.

Unfortunately, most of the interesting things we may want to compute for a graph provably $\Omega(n)$ space in this model, even allowing multiple passes over the input stream. We shall show such results when we study lower bounds, later in the course. 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. Thus, 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, it is $n(\log n)^{O(1)}$ space. Algorithms achieving such a space bound are sometimes called "semi-streaming" algorithms.[2]

13.2 The Connectedness Problem

Our first problem is: 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 lecture, the algorithm will consist of maintaining a subgraph of G satisfying certain conditions. For connectedness, the idea is to maintain a spanning forest, F, of G. 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 $[n]$.

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

The well known UNION-FIND data structure can be used to do the work in the processing section quickly. To test

[1]A simple graph is one with no loops and no parallel edges.

[2]The term does not really have a formal definition. Some authors would extend it to algorithms running in $O(n^{3/2})$ space, say.

```
Initialize   : $F \leftarrow \varnothing, X \leftarrow 0$ ;
Process $\{u, v\}$:
1 if $\neg X \;\wedge\; (F \cup \{\{u, v\}\}$ does not contain a cycle$)$ then
2     $F \leftarrow F \cup \{\{u, v\}\}$ ;
3     if $|F| = n - 1$ then $X \leftarrow 1$ ;
Output       : $X$ ;
```

acyclicity of $F \cup \{\{u, v\}\}$, we simply check if root(u) and root(v) are distinct in the data structure. Note that this algorithm assumes an*insertion-only* graph stream: edges only arrive and never depart from the graph. All algorithms in this lecture will make this assumption.

13.3 The Bipartiteness Problem

A bipartite graph is one whose vertices can be partitioned into two disjoint sets, S and T say, so that every edge is between a vertex in S and a vertex in T. Equivalently, a bipartite graph is one whose vertices can be properly colored using two colors.[3] Our next problem is to determine whether the input graph G is bipartite.

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.

```
Initialize   : $F \leftarrow \phi, X \leftarrow 1$ ;
Process $\{u, v\}$:
1 if $X$ then
2     if $F \cup \{\{u, v\}\}$ does not contain a cycle then
3         $F \leftarrow F \cup \{\{u, v\}\}$ ;
4     else if $F \cup \{\{u, v\}\}$ contains an odd cycle then
5         $X \leftarrow 0$ ;
Output       : $X$ ;
```

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 13.3.1. *The above algorithm outputs* 1 *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). 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 $\chi(u) \neq \chi(v)$. □

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

13.4 Shortest Paths and Distance Estimation via Spanners

Now consider the problem of estimating the distance in G between two vertices that are revealed after the input stream has been processed. That is, build a small data structure, in streaming fashion, that can be used to answer queries of the form "what is the distance between x and y?"

Define $d_G(x, y)$ to be the distance in G between vertices x and y:

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

where the minimum of an empty set defaults to ∞. 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.

$$\forall\, x, y \in [n] : \; d_G(x, y) \;\le\; d_H(x, y) \;\le\; t \cdot d_G(x, y)\,, \tag{13.1}$$

where $t \ge 1$ is an integer constant. A subgraph H satisfying (13.1) is called a *t-spanner* of G. Note that the left inequality trivially holds for every subgraph H of G.

```
Initialize   : $H \leftarrow \varnothing$ ;
Process $\{u, v\}$:
1  if $d_H(u, v) \ge t + 1$ then
2      $H \leftarrow H \cup \{\{u, v\}\}$ ;
Output       : On query $(x, y)$, report $\hat{d}(x, y) = d_H(x, y)$ ;
```

We now show that the final graph H constructed by the algorithm is a t-spanner of G. This implies that 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 $[d_G(x, y),\, t \cdot d_G(x, y)]$.

Pick any two distinct vertices $x, y \in [n]$. We shall show that (13.1) holds. If $d_G(x, y) = \infty$, then clearly $d_H(x, y) = \infty$ as well, and we are done. Otherwise, let π be the shortest path in G from x to y, 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 $d_{H'}(v_{i-1}, v_i) \le t$, where H' was the value of H at that time. Since H' is a subgraph of the final H, we have $d_H(v_{i-1}, v_i) \le t$. Thus, in both cases, we have $d_H(v_{i-1}, v_i) = t$. By the triangle inequality, it now follows that

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

which completes the proof, and hence implies the quality guarantee for the algorithm that we claimed earlier.

How much space does the algorithm 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.

13.4.1 The Size of a Spanner: High-Girth Graphs

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 $\gamma(H) \ge t + 2$. The next theorem places an upper bound on the size of a graph with high girth (see the paper by Alon, Hoory and Linial [**?**] and the references therein for more precise bounds).

Theorem 13.4.1. *Let n be sufficiently large. Suppose the graph G has n vertices, m edges, and $\gamma(G) \ge k$, for an integer k. Then*

$$m \;\le\; n + n^{1+1/\lfloor \frac{k-1}{2} \rfloor}\,.$$

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 $\ell = \lfloor \frac{k-1}{2} \rfloor$. 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. $\square$

Using $\lfloor \frac{k-1}{2} \rfloor \geq \frac{k-2}{2}$, 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 our algorithm 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 $\widetilde{O}(n^{5/3})$.

Lecture 14

Finding Maximum Matchings and Counting Triangles

Scribe: Ranganath Kondapally

[The rest is unchecked; use at your own risk.]

14.1 The Problems

The number of graph problems that one could try to solve is huge and we will only be able to cover a small fraction. In the previous lecture, we considered some very basic graph problems that we could solve easily in streaming fashion. Now we turn to *maximum matching*, which is a more sophisticated problem. It is well-studied in the classical theory of algorithm design, and Edmonds's polynomial-time algorithm for the problem [**?**] 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 space (in the semi-streaming sense) if we settle for an approximation algorithm.

Before moving on to other topics, we shall consider the problem of counting (or estimating) the number of *triangles* in a graph. This number becomes meaningful if one thinks of the input graph as a social network, for instance. 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.

14.2 Maximum Cardinality Matching

A matching is defined to be a graph where every vertex has degree at most 1. Having fixed a vertex set, one can think of a matching as a set of edges no two of which share a vertex. The maximum cardinality matching (or simply maximum matching) problem, abbreviated as MCM, is to find a largest sized subgraph M of a given graph G such that M is a matching.

Problem: We are given a graph stream (stream of edges). We want to find a maximum matching.

There are two types of maximum matchings that we will consider:

- Maximum cardinality matching(MCM) : We want to maximize the number of edges in the matching.

- Maximum weight matching(MWM) : In this case, edges are assigned weights and we want to maximize the sum of weights of the edges in the matching.

"Semi-Streaming" model : aim for space $\tilde{O}(n)$ or thereabouts [we just want to do better than $O(n^2)$ space usage]

Both the algorithms for MCM and MWM have the following common characteristics:

- Approximation improves with more number of passes
- Maintain a matching (in memory) of the subgraph seen so far

Input: Stream of edges of a n-vertex graph

Without space restrictions: We can compute a MCM, starting with an empty set, and try to increase the size of matching : By finding an *augmenting path.*

Definition 14.2.1. Augmenting Path: Path whose edges are alternately in M and not in M, beginning and ending in unmatched vertices.

Theorem 14.2.2. *Structure theorem: If there is no augmenting path, then the matching is maximum.*

So, when we can no longer find an augmenting path, the matching we have is an MCM by the above theorem.

Streaming Algorithm for MCM: Maintain a *maximal* matching (cannot add any more edges without violating the matching property). In more detail:

- **Initialize:** $M \leftarrow \emptyset$
- **Process** (u, v)**:** If $M \cup \{(u, v)\}$ is a matching, $M \leftarrow M \cup \{(u, v)\}$.
- **Output:** $|M|$

Theorem 14.2.3. *Let $\hat{t}$ denote the output of the above algorithm. If t is the size of MCM of G, then $t/2 \le \hat{t} \le t$.*

Proof. Let M^* be a MCM and $|M^*| = t$. Suppose $|M| < t/2$. Each edge in M "kills" (prevents from being added to M) at most two edges in M^*. Therefore, there exists an unkilled edge in M^* that could have been added to M. So, M is not maximal which is a contradiction. □

State of the art algorithm for MCM: For any ε, can find a matching M such that $(1-\varepsilon)t \le |M| \le t$, using constant (depends on ε) number of passes.

Outline: Find a matching, as above, in the first pass. Passes $2, 3, \ldots$ find a "short" augmenting path (depending on ε) and increase the size of the matching. Generalized version of the above structure theorem for matchings gives us the quality guarantee.

MWM algorithm:

- **Initialize:** $M \leftarrow \emptyset$
- **Process** (u, v)**:** If $M \cup \{(u, v)\}$ is a matching, then $M \leftarrow M \cup \{(u, v)\}$. Else, let $C = \{$edges of M conflicting with $(u, v)\}$ (note $|C| = 1$ or 2). If $wt(u, v) > (1 + \alpha)\, wt(C)$, then $M \leftarrow (M - C) \cup \{(u, v)\}$.
- **Output:** $\hat{w} = wt(M)$

Space usage of the above algorithm is $O(n \log n)$.

Theorem 14.2.4. *Let M^* be a MWM. Then, $k \cdot wt(M^*) \le \hat{w} \le wt(M^*)$ where k is a constant.*

With respect to the above algorithm, we say that edges are "born" when we add them to M. They "die" when they are removed from M. They "survive" if they are present in the final M. Any edge that "dies" has a well defined "killer" (the edge whose inclusion resulted in its removal from M).

We can associate a ("Killing") tree with each survivor where survivor is the root, and the edge(s) that may have been "killed" by the survivor (at most 2), are its child nodes. The subtrees under the child nodes are defined recursively.

Note: The above trees, so defined, are node disjoint (no edge belongs to more than one tree) as every edge has a unique "killer", if any.

Let $S = \{\text{survivors}\}$, $T(S) = \bigcup_{e \in S}[$ edges in the Killing tree(e) not including $e]$ (descendants of e)

Claim 1: $wt(T(S)) \leq wt(S)/\alpha$

Proof. Consider one tree, rooted at $e \in S$.

$$wt(\text{level}_i \text{ descendants}) \leq \frac{wt(\text{level}_{i-1} \text{ descendants})}{1+\alpha}$$

Because, $wt(\text{edge}) \geq (1+\alpha)wt(\{\text{edges killed by it}\})$

$$\begin{aligned}
\Rightarrow wt(\text{level}_i \text{ descendants}) &\leq \frac{wt(e)}{(1+\alpha)^i} \\
wt(\text{descendants}) &\leq wt(e)\Big(\frac{1}{1+\alpha} + \frac{1}{(1+\alpha)^2} + \ldots\infty\Big) \\
&= wt(e)\frac{1}{1+\alpha}\Big(\frac{1}{1-\frac{1}{1+\alpha}}\Big) \\
&= wt(e)\frac{1}{1+\alpha-1} \\
&= \frac{wt(e)}{\alpha}
\end{aligned}$$

$wt(T(S)) = \sum_{e \in S} wt(\text{descendants of } e) \leq \sum_{e \in S} \frac{wt(e)}{\alpha} = \frac{wt(S)}{\alpha}$

□

Claim 2: $wt(M^*) \leq (1+\alpha)(wt(T(S)) + 2 \cdot wt(S))$

Proof. Let $e_1^*, e_2^*, \ldots$ be the edges in M^* in the stream order. We will prove the claim by using the following charging scheme:

- If e_i^* is born, then charge $wt(e_i^*)$ to e_i^* which is in $T(S) \cup S$
- If e_i^* is not born, this is because of 1 or 2 conflicting edges
  - One conflicting edge, e: Note: $e \in S \cup T(S)$. Charge $wt(e_i^*)$ to e. Since e_i^* could not kill e, $wt(e_i^*) \leq (1+\alpha)wt(e)$
  - Two conflicting edges e_1, e_2: Note: $e_1, e_2 \in T(S) \cup S$. Charge $wt(e_i^*)\frac{wt(e_j)}{wt(e_1)+wt(e_2)}$ to e_j for $j = 1, 2$. Since e_i^* could not kill e_1, e_2, $wt(e_i^*) \leq (1+\alpha)(wt(e_1) + wt(e_2))$. As before, we maintain the property that weight charged to an edge $e \leq (1+\alpha)wt(e)$.
- If an edge is killed by e', transfer charge from e to e'. Note: $wt(e) \leq wt(e')$, so e' can indeed absorb this charge.

Edges in S may have to absorb $2(1+\alpha)$ times their weight (conflict two edges in M^*, etc). This proves the claim. □

By claim 1&2,

$$\begin{aligned} wt(M^*) &\le (1+\alpha)\Big(\frac{wt(S)}{\alpha} + 2\cdot wt(S)\Big) \\ &= (1+\alpha)\Big(\frac{1+2\alpha}{\alpha}\Big)wt(S) \\ &= \Big(\frac{1}{\alpha} + 3 + 2\alpha\Big)wt(S) \end{aligned}$$

Best choice for α (which minimizes the above expression) is $1/\sqrt{2}$. This gives

$$\frac{wt(M^*)}{3+2\sqrt{2}} \le \hat{w} \le wt(M^*)$$

Open question: Can we improve the above result (with a better constant)?

14.3 Triangle Counting:

Given a graph stream, we want to estimate the number of triangles in the graph. Some known results about this problem:

- We can't multiplicatively approximate the number of triangles in $o(n^2)$ space.
- We can approximate the number of triangles up to some *additive* error
- If we are given that the number of triangles $\ge t$, then we can multiplicatively approximate it.

Given a input stream of m edges of a graph on n vertices with at least t triangles, we can compute (ε,δ)-approximation using space $O(\frac{1}{\varepsilon^2}\log\frac{1}{\delta}\cdot\frac{mn}{t})$ as follows:

1. Pick an edge (u,v) uniformly at random from the stream
2. Pick a vertex w uniformly at random from $V\setminus\{u,v\}$
3. If (u,w) and (v,w) appear after (u,v) in the stream, then output $m(n-2)$ else output 0.

It can easily be shown that the expectation of the output of the above algorithm is equal to the number of triangles. As before we run copies of the above algorithm in parallel and take the average of their output to be the answer. Using Chebyshev's inequality, from the variance bound, we get the space usage to be $O(\frac{1}{\varepsilon^2}\log\frac{1}{\delta}\cdot\frac{mn}{t})$.

Bar-Yossef, Kumar, Sivakumar [BKS02] algorithm: Uses space $\tilde{O}(\frac{1}{\varepsilon^2}\log\frac{1}{\delta}\cdot(\frac{mn}{t})^2)$. Even though the space usage of this algorithm is more than the previous one, the advantage of this algorithm is it is a "sketching" algorithm (computes not exactly a linear transformation of the stream but we can compose "sketches" computed by the algorithm of any two streams and it can handle edge deletions).

Algorithm: Given actual stream of edges, pretend that we are seeing virtual stream of triples $\{u,v,w\}$ where $u,v,w\in V$. Specifically,

$$\begin{array}{ccc} \text{actual token} & & \text{Virtual token} \\ \{u,v\} & \rightarrow & \{u,v,w_1\},\{u,v,w_2\},\ldots,\{u,v,w_{n-2}\} \end{array}$$

where $\{w_1,w_2,\ldots,w_{n-2}\} = V\setminus\{u,v\}$.

Let F_k be the k^{th} frequency moment of virtual stream and

$$T_i = \Big|\Big\{\{u,v,w\} : u,v,w \text{ are distinct vertices and } \exists \text{ exactly } i \text{ edges amongst } u,v,w\Big\}\Big|$$

Note: $T_0 + T_1 + T_2 + T_3 = \binom{n}{3}$.

$$\begin{aligned} F_2 &= \sum_{u,v,w} (\text{number of occurrences of } \{u, v, w\} \text{ in the virtual stream})^2 \\ &= 1^2 \cdot T_1 + 2^2 \cdot T_2 + 3^2 \cdot T_3 \\ &= T_1 + 4T_2 + 9T_3 \end{aligned}$$

Similarly, $F_1 = T_1 + 2T_2 + 3T_3$ (Note: $F_1 = \text{length of virtual stream} = m(n-2)$) and $F_0 = T_1 + T_2 + T_3$.

If we had estimates for F_0, F_1, F_2, we could compute T_3 by solving the above equations. So, we need to compute two sketches of the virtual stream, one to estimate F_0 and another to estimate F_2.

Lecture 15

Communication Complexity

Scribe: Aarathi Prasad

Recall the Misra-Gries algorithm for MAJORITY problem where we found the majority element in a data stream in 2 passes. We can show that it is not possible to find the element using 1 pass in sublinear space. Lets see how to determine such lower bounds.

15.1 Introduction to Communication Complexity

First let's start with an introduction to an abstract model of communication complexity.
There are two parties to this communication "game," namely Alice and Bob. Alice's input is $x \in X$ and Bob's input is $y \in Y$. X and Y are known before the problem is specified. Now we want to compute $f(x, y)$, $f : X \times Y \to \mathbb{Z}$, when $X = Y = [n]$, $Z = \{0, 1\}$. For example,

$$f(x, y) = (x + y) \mod 2 = x \mod 2 + y \mod 2$$

In this case, Alice doesn't have to send the whole input x, using $\lceil \log n \rceil$ bits; instead she can send $x \mod 2$ to Bob using just 1 bit! Bob calculates y mod2 and uses Alice's message to find *f(x,y)*. However, only Bob knows the answer. Bob can choose to send the result to Alice, but in this model, it is not required that all the players should know the answer. Note that we are not concerned about the memory usage, but we try to minimise the number of communication steps between Alice and Bob.

15.1.1 EQUALITY problem

This problem finds its application in image comparisons.

$$\text{Given } X = Y = \{0,1\}^n \ , \ Z = \{0,1\}^n$$

$$\text{EQ}(x, y) = \begin{cases} 1 & \text{if x = y} \\ 0 & \text{otherwise} \end{cases}$$

So in this case, communication between Alice and Bob requires n bits. For now, we consider a one-way transmission, ie messages are sent only in one direction. For symmetric functions, it doesn't matter who sends the message to whom.

Theorem 15.1.1. *Alice must send Bob n bits, in order to solve EQ in a one-way model.*

$$D^{\rightarrow}(\text{EQ}) \geq n$$

Proof: Suppose Alice sends $< n$ bits to Bob. Then the number of different messages she might send $\leq 2^1 + 2^2 + \ldots.2^{n-1} = 2^n - 2$. But Alice can have upto 2^n inputs.

Let's recall the Pigeonhole principle. There are n pigeons, m holes and $m < n$. This implies that ≥ 2 pigeons have to go into one hole.

Using the pigeonhole principle, there exists two different inputs $x \neq x'$, such that Alice sends the same message α on inputs x and x'.

Let $P(x, y)$ be Bob's output, when input is (x, y). We should have

$$P(x, y) = \mathrm{EQ}(x, y) \tag{15.1}$$

Since $P(x, y)$ is determined fully by Alice's message on x and Bob's input y, using equation 13.1, we have

$$P(x, y) = \mathrm{EQ}(x, x) = 1 \tag{15.2}$$

$$P(x', y) = \mathrm{EQ}(x', x) = 0 \tag{15.3}$$

However since Bob sees the message α from Alice for both inputs x and x', $P(x, y) = P(x', y)$, which is a contradiction.

Hence the solution to the EQUALITY problem is for Alice to send all the 2^n messages using n bits.

Theorem 15.1.2. *Using randomness, we can compute EQ function with an error probability* $\leq 1/3$*, in the one-way model, with message size O(*$\log n$*)*

$$R^{\rightarrow}(\mathrm{EQ}) = O(\log n)$$

Proof: Consider the following protocol.

- Alice picks a random prime $p \in [n^2, 2n^2]$
- Alice sends Bob $(p, x \bmod p)$, using O($\log n$) bits
- Bob checks if $y \bmod p = x \bmod p$, outputs 1 if true and 0 otherwise

If EQ(x, y) = 1, output is correct. If EQ(x, y) = 0, then its an error if and only if *(x-y)* mod p = 0.

Remember that x and y are fixed and p is random. Also we should choose a large prime for p, since if we choose p = 2, there is high error probability.

Let $x - y = p_1 p_2 \ldots p_t$ be the prime factorisation of $x - y$, where $p_1, p_2, \ldots p_t$ need not be distinct. For the output EQ(x, y) to be incorrect, p $\in \{p_1, p_2 .. p_t\}$.

$$Pr[error] \leq \frac{t}{\text{no of primes in } [n^2, 2n^2]} \tag{15.4}$$

$$x - y \leq 2^n, \quad p_1 p_2 \ldots p_t \geq 2^t \Rightarrow t \leq n$$

Using prime number theorem,

$$\text{Number of primes in } [1..N] \approx \frac{N}{\ln N}$$

$$\begin{aligned}
\text{Number of primes in } [n^2, 2n^2] &\approx \frac{2n^2}{\ln\ 2n^2} - \frac{n^2}{\ln\ n^2} \\
&= \frac{2n^2}{\ln\ 2\ +\ 2\ \ln\ n} - \frac{n^2}{2\ \ln\ n} \\
&\geq \frac{1.9\, n^2}{2\ \ln\ n} - \frac{n^2}{2\ \ln\ n} \\
&= \frac{0.9\, n^2}{2\ \ln\ n}
\end{aligned}$$

$$\begin{aligned}
\text{Using estimates in equation 3, } Pr[error] &\leq \frac{n}{0.9\, n^2\ /\ 2\ \ln\ n} \\
&= \frac{2\ \ln\ n}{0.9\, n} \\
&\leq \frac{1}{3}
\end{aligned}$$

15.2 Communication complexity in Streaming Algorithms

Now lets consider communication complexity models in streaming algorithms. Here the stream is cut into two parts, one part is with Alice and the other with Bob. We claim that if we can solve the underlying problem, we can solve the communication problem as well.

Suppose $\exists$ a deterministic or randomised streaming algorithm to compute $f(x, y)$ using s bits of memory, then

$$D^{\rightarrow}(f)\ \leq\ s \quad \text{OR} \quad R^{\rightarrow}(f)\ \leq\ s$$

Alice runs the algorithm on her part of the stream, sends the values in memory (s bits) to Bob, and he uses these values, along with his part of the stream, to compute the output.

Note one-pass streaming algorithm implies a one-way communication. So a

$$\text{p pass streaming algo } \Rightarrow \begin{cases} \text{p messages from A } \rightarrow \text{B} \\ \text{p -1 messages from B } \rightarrow \text{A} \end{cases}$$

Hence we can generalize that a communication lower bound proven for a fairly abstract problem can be *reduced* to a streaming lower bound. ie from a streaming lower bound, we can reach the lower bound for a communication problem.

15.2.1 INDEX problem

Given $X = \{0,1\}^n,\ Y = [n],\ Z = \{0,1\}$,

$$\text{INDEX}(x, j)\ =\ x_j\ =\ j^{th} \text{ bit of } x$$

e.g., $\text{INDEX}(1100101, 3) = 0$

Theorem 15.2.1.

$$D^{\rightarrow}(\text{INDEX})\ \geq\ n$$

Proof Can be proved using Pigeonhole Principle.

Say we have a MAJORITY streaming algorithm using s space. We are given an INDEX instance, Alice's input :

1100101 and Bob's input : 3. The scheme followed is

Given an instance (x, j) of INDEX, we construct streams σ, π of length m each as follows. Let A be the streaming algorithm, then

- ALICE's input x is mapped to $a_1 a_2 ... a_m$, where $a_i = 2(i-1) + x_i$
- BOB's input j is mapped to $bb...b$, b occurs m times, where b = 2(j-1)
- Alice and Bob communicate by running A on $\sigma.\pi$
- If A says "no majority", then output 1, else output 0.

1-pass MAJORITY algorithm requires $\Omega(m)$ space. By theorem 13.2.1, communication uses $\geq m$, $s \geq m$ = 1/2 stream length. We also have $R^{\to}(\text{INDEX}) = \Omega(m)$.

Lecture 16

Reductions

Scribe: Priya Natarajan

Recap and an important point: Last time we saw that 1-pass MAJORITY requires space $\Omega(n)$ by reducing from INDEX. We say $\Omega(n)$, but in our streaming notation, m is the stream length and n is the size of the universe. We produced a stream $\sigma \circ \pi$, where $m = 2N$ and $n = 2N$, and we concluded that the space s must be $\Omega(N)$. Looks like $s = \Omega(m)$ or $s = \Omega(n)$ but we have proven the lower bound only for the worst of the two. So, in fact, we have $s = \Omega(min\{m, n\})$. For MAJORITY, we could say $\Omega(n)$ because m and n were about the same. We won't be explicit about this point later, but most of our lower bounds have $\Omega(min\{m, n\})$ form.

Today, we will see some communication lower bounds. But first, here is a small table of results:

f	$D^{\rightarrow}(f)$	$R^{\rightarrow}_{1/3}(f)$	$D(f)$	$R_{1/3}(f)$
INDEX	$\geq n$	$\Omega(n)$	$\leq \lceil \log n \rceil$	$\leq \lceil \log n \rceil$
EQUALITY	$\geq n$	$O(\log n)$	$\geq n$	$O(\log n)$
DISJOINTNESS	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$

Table 16.1: In this table, we will either prove or have already seen almost all results except $R^{\rightarrow}_{1/3}(EQ)$. Also, we will only prove a special case of DISJ.

We will start by proving D(EQ) $\geq n$. In order to prove this, however, we first have to prove an important property about deterministic communication protocols.

Definition 16.0.2. In two-way communication, suppose Alice first sends message α_1 to Bob to which Bob responds with message β_1. Then, say Alice sends message α_2 to Bob who responds with message β_2, and so on. Then, the sequence of messages $< \alpha_1, \beta_1, \alpha_2, \beta_2, \ldots >$ is known as the *transcript* of the deterministic communication protocol.

Theorem 16.0.3. ***Rectangle property of deterministic communication protocol****: Let P be a deterministic communication protocol, and let X and Y be Alice's and Bob's input spaces. Let $trans_p(x, y)$ be the transcript of P on input $(x, y) \in X \times Y$. Then, if $trans_p(x_1, y_1) = trans_p(x_2, y_2) = \tau$ (say); then $trans_p(x_1, y_2) = trans_p(x_2, y_1) = \tau$.*

Proof: We will prove the result by induction.

Let $\tau =< \alpha_1, \beta_1, \alpha_2, \beta_2, \ldots >$. Let $\tau' =< \alpha'_1, \beta'_1, \alpha'_2, \beta'_2, \ldots >$ be trans$_p(x_2, y_1)$.

Since Alice's input does not change between (x_2, y_1) and (x_2, y_2), we have $\alpha'_1 = \alpha_1$.

Induction hypothesis: τ and τ' agree on the first j messages.

Case 1: j is odd = $2k - 1$ ($k \geq 1$) (say).

So, we have: $\alpha'_1 = \alpha_1, \alpha'_2 = \alpha_2, \ldots, \alpha'_{k-1} = \alpha_{k-1}, \alpha'_k = \alpha_k$ and $\beta'_1 = \beta_1, \beta'_2 = \beta_2, \ldots, \beta'_{k-1} = \beta_{k-1}, \beta'_k =$?.

Now, β'_k depends on Bob's input which does not change between (x_1, y_1) and (x_2, y_1), and it depends on the transcript so far which also does not change. So, $\beta'_k = \beta_k$.

Case 2: j is even = $2k$ ($k \geq 1$) (say).

The proof for this case is similar to that of case 1, except we consider inputs (x_2, y_2) and (x_2, y_1).

By induction, we have the rectangle property.

Now that we have the rectangle property at hand, we can proceed to prove:

Theorem 16.0.4. *D(EQ)* $\geq n$.

Proof: Let P be a deterministic communication protocol solving EQ. We will prove that P has at least 2^n different transcripts which will imply that P sends $\geq n$ bits.

Suppose $\text{trans}_P(x_1, x_1) = \text{trans}_P(x_2, x_2) = \tau$, where $x_1 \neq x_2$.

Then, by rectangle property:
$\text{trans}_P(x_1, x_2) = \tau$
$\Rightarrow$ output on (x_1, x_1) = output on (x_1, x_2)
$\Rightarrow 1 = \text{EQ}(x_1, x_1) = \text{EQ}(x_1, x_2) = 0$. [Contradiction]

This means that the 2^n possible inputs lead to 2^n distinct transcripts.

Note: General communication corresponds to multi-pass streaming.

Theorem 16.0.5. *Any deterministic streaming algorithm for DISTINCT-ELEMENTS, i.e., F_0 must use space $\Omega(n/p)$, where p is the number of passes.*

Before seeing the proof, let us make note of a couple of points.

Why $\Omega(n/p)$? Over p passes, Alice and Bob exchange $2p - 1$ messages; if the size of each message (i.e., space usage) is s bits, then:

total communication cost = $s(2p - 1)$
$\Rightarrow$ total communication cost $\leq 2ps$.

If we can prove that the total communication must be at least n bits, then we have:
$n \leq 2ps$
$\Rightarrow s \geq n/2p$
i.e., $s = \Omega(n/p)$

Proof Idea: We will reduce from EQ. Suppose Alice's input $x \in \{0,1\}^n$ is $\{100110\}$, and Bob's input $y \in \{0,1\}^n$ is $\{110110\}$. Using her input, Alice produces the stream $\sigma = < 1, 2, 4, 7, 9, 10 >$ (i.e., $\sigma_i = 2(i-1) + x_i$). Similarly, Bob produces the stream $\pi = < 1, 3, 4, 7, 9, 10 >$.

Suppose Alice's string disagrees in d places with Bob's string (i.e., Hamming distance$(x, y) = d$). Then, $F_0(\sigma \circ \pi) = n + d$. If σ and π are equal, then $d = 0$, i.e., $\text{EQ}(x, y) = 1 \Leftrightarrow d = 0$.

So, $\text{EQ}(x, y) = 1 \Rightarrow d = 0 \Rightarrow F_0(\sigma \circ \pi) = n$
$\text{EQ}(x, y) = 0 \Rightarrow d \geq 1 \Rightarrow F_0(\sigma \circ \pi) \geq n + 1$

However, note that unless the F_0 algorithm is exact, it is very difficult to differentiate between n and $n + (a\ small\ d)$. As we proved earlier, we can only get an approximate value for F_0, so we would like that if $x \neq y$, then Hamming distance between x and y be noticeably large. In order to get this large Hamming distance, we will first run Alice's and Bob's input through an encoder that guarantees a certain distance. For example, running Alice's n-bit input x through an encoder might lead to a $3n$-bit string $c(x)$ (similarly, y and $c(y)$ for Bob), and the encoder might guarantee that $x \neq y \Rightarrow$ Hamming distance between $c(x)$ and $c(y) \geq 3n/100$ (say).

Proof: Reduction from EQ.

Suppose Alice and Bob have inputs $x, y \in \{0,1\}^n$. Alice computes $\tilde{x} = C(x)$, where C is an error-correcting code with encoded length $3n$, and distance $\geq 3n/100$. Then, she produces a stream $\sigma = < \sigma_1, \sigma_2, \ldots, \sigma_{3n} >$, where $\sigma_i = 2(i-1) + \tilde{x}_i$. Similarly, Bob has y, he computes $\tilde{y} = C(y)$ and produces a stream π where $\pi_i = 2(i-1) + \tilde{y}_i$.

Now, they simulate F_0 streaming algorithm to estimate $F_0(\sigma \circ \pi)$.

If $x = y$, $\mathrm{EQ}(x, y) = 1$, then x̃ = ỹ, so $F_0 = 3n$.

If $x \neq y$, $\mathrm{EQ}(x, y) = 0$, then Hamming(x̃, ỹ) $\geq 3n/100$, so $F_0 \geq (3n + 3n/100)$. That is $F_0 \geq 3n * 1.01$.

If F_0 algorithm gives a $(1 \pm \varepsilon)$-approximation with $\varepsilon < 0.005$, then we can distinguish between the above two situations and hence solve EQ. But $D(\mathrm{EQ}) \geq n$, therefore, $D(F_0) = \Omega(n/p)$ with p passes.

Lecture 17

Set Disjointness and Multi-Pass Lower Bounds

Scribe: Zhenghui Wang

17.1 Communication Complexity of DISJ

Alice has a n-bit string x and Bob has a n-bit string y, where $x, y \in [0,1]^n$. x, y represent subset X, Y of $[n]$ respectively. x, y are named characteristic vector or bitmask, i.e.

$$x_j = \begin{cases} 1 & \text{if } j \in X \\ 0 & \text{otherwise} \end{cases}$$

Compute function

$$\text{DISJ}(x, y) = \begin{cases} 1 & \text{if } X \cap Y \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

Theorem 17.1.1. $R(\text{DISJ}) = \Omega(n)$

Proof. We prove the special case: $R^{\rightarrow}(DISJ) = \Omega(n)$ by reduction from INDEX. Given an instance (x, j) of INDEX, we create an instance (x', y') of DISJ, where $x' = x$, y' is a vector s.t.

$$y'_i = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{otherwise} \end{cases}$$

By construction, $x' \cap y' \neq \emptyset \Leftrightarrow x_j = 1$ i.e. DISJ(x', y') =INDEX(x', y'). Thus, a one way protocol of DISJ implies a one way protocol of INDEX with the same communication cost. But $R^{\rightarrow}(\text{INDEX}) = \Omega(n)$, so we have $R^{\rightarrow}(\text{DISJ}) = \Omega(n)$. □

The general case is proved by [Kalya], [Razborov '90] [Bar Yossof-Jayarom Kauor '02].

17.2 ST-Connectivity

ST-connectivity: Given an input graph stream, are two specific vertices s and t connected?

Theorem 17.2.1. *Solving ST-CONN requires $\Omega(n)$ space. (semi-streaming is necessary.)*

Proof. By reduction from DISJ.
Let (x, y) be an instance of DISJ. Construct an instance of CONN on a graph with vertex set $\{s, t, v_1, v_2, \cdots, v_{n-2}\}$. So Alice gets set of edges $A = \{(s, v) : v \in x\}$ and Bob gets set of edges $B = \{(v, t) : v \in y\}$.

$$\begin{aligned} & \text{Vertex } s \text{ and } t \text{ are connected in the resulting graph} \\ \Leftrightarrow\ & \exists \text{ a length-two path from } s \text{ to } t \\ \Leftrightarrow\ & \exists v \text{ s.t. } (s, v) \in A \text{ and } (v, t) \in B \\ \Leftrightarrow\ & \exists v \in x \cap y \\ \Leftrightarrow\ & \text{DISJ}(x, y) = 1 \end{aligned}$$

Reducing this communication problem to a streaming problem, we have the required space in a p pass algorithm is $\Omega(n/p)$. □

17.3 Perfect Matching Problem

Given a graph on n vertices, does it have a perfect matching i.e. a matching of size $n/2$?

Theorem 17.3.1. *One pass streaming algorithm for this problem requires $\Omega(n^2)$ space.*

Proof. Reduction from INDEX.
Suppose we have an instance (x, k) of INDEX where $x \in \{0,1\}^{N^2}, k \in [N^2]$. Construct graph G, where $V_G = \cup_{i=1}^{N}\{a_i, b_i, u_i, v_i\}$ and $E_G = E_A \cup E_B$. Edges in E_A and E_B are introduced by Alice and Bob respectively.

$$E_A = \{(u_i, v_j : x_{f(i,j)=1})\}, \text{ where } f \text{ is a 1-1 correspondence between } [N] \times [N] \to [N^2]$$

e.g $f(i, j) = (N - 1)i + j$

$$E_B = \{(a_l, u_l : l \neq i)\} \cup \{(b_l, v_l : l \neq j)\} \cup \{(a_i, b_j)\}, \text{ where } i, j \in [N] \text{ are s.t. } f(i, j) = k$$

By construction,

$$\begin{aligned} & G \text{ has a perfect matching} \\ \Leftrightarrow\ & (u_i, v_j) \in E_G \\ \Leftrightarrow\ & x_k = 1 \\ \Leftrightarrow\ & \text{INDEX}(x, k) = 1 \end{aligned}$$

Thus, the space usage of a streaming algorithm for perfect matching must be $\Omega(N^2)$. The number of vertices in instance constructed is $n = 4N$. So the lower bound in term of n is $\Omega((\frac{n}{4})^2) = \Omega(n^2)$. □

17.4 Multiparty Set Disjointness (DISJ$_{n,t}$)

Suppose there are t player and each player A_i has a n-bit string $x_i \in \{0,1\}^n$. Define

$$\text{DISJ}_{n,t} = \begin{cases} 1 & \text{if } \bigcup_{i=1}^{n} x_i \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

Theorem 17.4.1 (C.-Khot-Sun '03/Granmeicr '08). $R(\text{DISJ}_{n,t}) = \Omega(n/t)$

Theorem 17.4.2. *Approximating 1-pass F_k with randomization allowed requires $\Omega(m^{1-2/k})$ space.*

Proof. Given a DISJ instance $(x_1, x_2, \cdots, x_t)$ s.t. every column of matrix $\begin{pmatrix} x_1 \\ x_2 \\ \cdots \\ x_t \end{pmatrix}$ must contain 0,1 or t ones and at most one column can have t ones. Player A_i converts x_i into a stream $\pi_i =< \sigma_1, \sigma_2, \cdots >$ such that $\sigma_j \in \pi_i$ iff. $x_{ij} = 1$. Simulate the F_k algorithm on stream $\pi = \pi_1 \circ \pi_2 \circ \cdots \circ \pi_t$. The frequency vector of this stream is either $0, 1$ only (if $\text{DISJ}_{n,t} = 0$) or $0, 1$ and a single t(If $\text{DISJ}_{n,t} = 1$). In the first case, $F_k \leq n$ while $F_k \geq t^k$ in the second case.
Choose t s.t. $t^k > 2m$, then 2-approximation to F_k can distinguish these two situations. Thus, The memory usage is $\Omega(m/t^2) = \Omega(\frac{m}{(m^{1/k})^2}) = \Omega(m^{1-2/k})$, for stream length $m \leq nt$ and $m = \Omega(1/t)$. □

Bibliography

[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.

[BGKS06] Lakshminath Bhuvanagiri, Sumit Ganguly, Deepanjan Kesh, and Chandan Saha. Simpler algorithm for estimating frequency moments of data streams. In *Proc. 17th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 708–713, 2006.

[BJK$^+$04] 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, 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.

[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.

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

[FM88] Peter Frankl and Hiroshi Maehara. The johnson-lindenstrauss lemma and the sphericity of some graphs. *J. Combin. Theory Ser. B*, 44(3):355–362, 1988.

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

[IW05] Piotr Indyk and David P. Woodruff. Optimal approximations of the frequency moments of data streams. In *Proc. 37th Annual ACM Symposium on the Theory of Computing*, pages 202–208, 2005.

[JL84] W. B. Johnson and J. Lindenstrauss. Extensions of Lipschitz mapping into Hilbert space. *Contemp. Math.*, 26:189–206, 1984.

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

[Nis90] Noam Nisan. Pseudorandom generators for space-bounded computation. In *Proc. 22nd Annual ACM Symposium on the Theory of Computing*, pages 204–212, 1990.