CS85: Data Stream Algorithms

Lecture Notes, Fall 2009

Amit Chakrabarti
Dartmouth College

Latest Update: December 15, 2009

Contents

Lecture 0

Preliminaries: The Data Stream Model

Scribe: Amit Chakrabarti

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

Scribe: Amit Chakrabarti

1.1 The Problem

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

```
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 $\hat{f_a} = 0$ ;
```

1.3 Analysis of the Algorithm

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

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

Scribe: Amit Chakrabarti

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].

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 $\text{zeros}(h(j)) \geq \log d$, and we don't expect any tokens to hit $\text{zeros}(h(j)) \gg \log d$. Thus, the maximum value of $\text{zeros}(h(j))$ 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 $Y_r = \sum_{j\colon f_j>0} X_{r,j}$. 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.

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

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

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

Finding Frequent Items via Sketching

Scribe: Radhika Bhasin

3.1 The Problem

We return to the FREQUENCY-ESTIMATION problem that we saw in Lecture 1. We give two randomized one-pass algorithms for this problem that work in the more general *turnstile model*. Thus, we have an input stream $\sigma = \langle a_1, a_2, \ldots \rangle$, where $a_i = (j_i, c_i) \in [n] \times \{-L, \ldots, L\}$. As usual, this implicitly defines a frequency vector $\mathbf{f} = (f_1, \ldots, f_n)$ where

$$f_j = \sum_{i:\, j_i = j} c_i \,.$$

We want to maintain a "sketch" of the stream, which is a data structure based on which we can, at any time, quickly compute an estimate $\hat{f}_j$ of f_j, for any given $j \in [n]$.

We shall give two different solutions to this problem, with incomparable quality guarantees. Both solutions maintain certain counters who size is bounded not by the length of the stream, but by the maximum possible frequency. Thus, we let m denote the maximum possible value attained by any $|f_j|$ as we process the stream. Whenever we refer to counters in our algorithms, they are assumed to be $O(\log m)$-bits long.

3.2 The Count-Min Sketch Algorithm

This sketch was introduced by Cormode and Muthukrishnan [CM05]. A Count-Min (CM) sketch with parameters (ε, δ) provides guarantees akin to an (ε, δ)-approximation, but not quite! The parameter ε is used in a different way, though δ still represents the probability of error. Read on.

The CM sketch consists of a two-dimensional $t \times k$ array of counters, plus t independent hash functions, each chosen uniformly at random from a 2-universal family. Each time we read a token, resulting in an update of f_j, we update certain counters in the array based on the hash values of j (the "count" portion of the sketch). When we wish to compute an estimate $\hat{f}_j$, we report the minimum (the "min" portion) of these counters. The values of t and k are set, based on ε and δ, as shown below.

```
Initialize :
1   $t \leftarrow \log(1/\delta)$ ;
2   $k \leftarrow 2/\varepsilon$ ;
3   $C[1 \ldots t][1 \ldots k] \leftarrow \vec{0}$ ;
4   Pick $t$ independent hash functions $h_1, h_2, \ldots, h_t : [n] \to [k]$, each from a 2-universal family ;
Process $(j, c)$:
5   for $i = 1$ to $t$ do
6       $C[i][h_i(j)] \leftarrow C[i][h_i(j)] + c$ ;
Output : On query $a$, report $\hat{f}_a = \min_{1 \le i \le t} C[i][h_i(a)]$
```

3.2.1 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_j. Thus, we always have

$$f_j \le \hat{f}_j .$$

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 $Y_{i,j}$ denote the contribution of j to this excess. Notice that j makes a contribution iff it collides with a under the ith hash function, i.e., iff $h_i(j) = h_i(a)$. And when it does contribute, it causes f_j to be added to this counter. Thus,

$$Y_{i,j} = \begin{cases} f_j \,, & \text{if } h_i(j) = h_i(a) \text{, which happens with probability } 1/k \,, \\ 0 \,, & \text{otherwise.} \end{cases}$$

Notice that the probability above is $1/k$ because h_i is 2-universal. Anyway, we now see that $\mathbb{E}[Y_{i,j}] = f_j/k$. By linearity of expectation,

$$\mathbb{E}[X_i] = \mathbb{E}\left[\sum_{\substack{j=1 \\ j \ne a}}^{n} Y_{i,j}\right] = \sum_{\substack{j=1 \\ j \ne a}}^{n} \mathbb{E}[Y_{i,j}] = \sum_{\substack{j=1 \\ j \ne a}}^{n} \frac{f_j}{k} = \frac{\|\mathbf{f}\|_1 - f_a}{k} .$$

Using our choice of k, we have

$$\mathbb{E}[X_i] = \frac{\varepsilon}{2}(\|\mathbf{f}\|_1 - f_a) \le \frac{\varepsilon}{2} \cdot \|\mathbf{f}\|_1 .$$

So, by Markov's inequality,

$$\Pr[X_i \ge \varepsilon \|\mathbf{f}\|_1] \le \frac{1}{2} .$$

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,

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

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

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

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

3.2.2 Space Usage

The Count-Min algorithm needs to store (1) the $k \times t$ array of counters, and (2) representations of the t hash functions. The array needs space $O(kt \log m) = O(\frac{1}{eps} \log \frac{1}{\delta} \cdot logm)$. Assuming, w.l.o.g., that n is a power of 2, we can use a finite-field-based hash function (as in Homework 1), which will require $O(\log n)$ space for each of the t functions. Thus, the total space usage is at most

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

3.3 The Count Sketch Algorithm

Count Sketch Algorithm allows us to the estimate the frequencies of all items in the stream.The algorithm uses two hash functions,along with a two-dimensional $t \times k$ array of counters.The first hash function h_i maps input input items onto k,and secondary hash function g_i maps the input items onto {-1,+1}.Each input item j causes $g_i(j)$ to be added on to the entry $C[i][h_i(j)]$ in row i,for $1 \le i \le t$.For any row i, $g_i(j).C[i][h_i(j)]$ is an unbaised estimator for f_j.The estimate $\hat{f}_j$, is the median of these estimates over the t rows.The values of t and k are set, based on ε and δ, as shown below.

```
Initialize :
1   $t \leftarrow \log(1/\delta)$ ;
2   $k \leftarrow 3/\varepsilon^2$ ;
3   $C[1 \ldots t][1 \ldots k] \leftarrow \vec{0}$ ;
4   Pick $t$ independent hash functions $h_1, h_2, \ldots, h_t : [n] \to [k]$, each from a 2-universal family ;
5   Pick $t$ independent secondary hash functions $g_1, g_2, \ldots, g_t : [n] \to \{-1, 1\}$, each from a 2-universal family ;
Process $(j, c)$:
6   for $i = 1$ to $t$ do
7       $C[i][h_i(j)] \leftarrow C[i][h_i(j)] + c.g_i(j)$ ;
Output : On query $a$, report $\hat{f}_a = \text{median}_{1 \le i \le t} C[i][h_i(a)]$
```

$g_i(j)$ is either $+1$ or -1 and the decision is made at random.For any token a,$g_1(a)$ will either add one or subtract one and is constant among multiple occurences of the token.If another token b collides with token a, descision of $g_i(b)$ will be made independent of $g_i(a)$.

Case1: No collison of token a with other tokens

$$C[i][h_i(a)] += f_a.g_i(a)$$

Case2: In case of Collisons,other tokens will increment or decrement the counter with equal probability.

$$C[i][h_i(a)] += f_a.g_i(a) + \text{Error due to collision}$$

Without loss of generality,assuming $a = 1$ and $i = 1$,then a good estimate for $C[1][h_1(1)]$ should be equal to $f_1.g_1(1)$.

3.4 Analysis of the Algorithm

For a fixed token a, we now analyze the excess in one such counter, say in $C[i][h_i(a)]$. Let the random variable X denote this excess. For $j \in [n] \setminus \{a\}$, let Y_j denote the contribution of j to this excess. Notice that j makes a contribution iff it collides with a under the ith hash function, i.e., iff $h_i(j) = h_i(a)$. And when it does contribute, it causes f_j to be added to this counter. Thus,

$$Y_j = \begin{cases} f_j\,, & \text{which happens with probability } 1/2k\,, \\ -f_j\,, & \text{which happens with probability } 1/2k\,, \\ 0\,, & \text{which happens with probability } 1-1/k\,, \end{cases}$$

Notice that the probability above is $1/2k$ because h_i and g_i both are 2-universal. And Expectation $\mathbb{E}[Y_j] = 0$.

Writing X as the sum of pairwise independent random variables

$$X = (Y_2 + Y_3 + Y_4 + \ldots, Y_n)$$

Random variables $(Y_2 + Y_3 + Y_4 \ldots, Y_n)$ are pairwise independent as the hash functions chosen are 2-universal.

Therefore, by pairwise independence we have

$$\text{Var}[X] = \text{Var}[Y_2] + \text{Var}[Y_3] + \ldots \text{Var}[Y_n]$$

Variance of a random variable Y_j is given by

$$\begin{aligned} \text{Var}[Y_j] &= \mathbb{E}[Y_j^2] - \mathbb{E}[Y_j]^2 \\ \text{Var}[Y_j] &= \mathbb{E}[Y_j^2] - 0 \\ \text{Var}[Y_j] &= f_j^2 \frac{1}{k} \end{aligned}$$

Therefore,

$$\text{Var}[X] = \frac{(f_2^2 + \ldots f_n^2)}{k}.$$

Writing in the norm form,we have

$$\text{Var}[X] = \frac{\|f\|_2^2 - f_1^2}{k}$$

Defining $K = \sqrt{\|f\|_2^2 - f_1^2}$, then

$$\text{Var}[X] = \frac{K^2}{k}$$

So,Standard Deviation, $\sigma = \sqrt{\frac{K^2}{k}}$

Using Chebyshev's Inequality,

$$Pr[|X| \geq \varepsilon K] = Pr[|X| \geq \varepsilon\sqrt{k}\left(\frac{K}{\sqrt{k}}\right)] \leq \left(\frac{1}{\varepsilon^2 k}\right) \leq \frac{1}{3}$$

where $t = \varepsilon\sqrt{k}$ and $\sigma = \frac{K}{\sqrt{k}}$

Thus,the final guarantee for Count Sketch is

$$|\hat{f}_j - f_j| \leq \varepsilon \|f\|_2$$

where the inequality fails with probability at most δ after amplifying the probability of success by taking median of $t = \log \frac{1}{\delta}$ independent instances.(Using chernoff bound)

3.4.1 Space Usage

The Count Sketch algorithm needs to store (1) the $t \times k$ array of counters, and (2) representations of the two hash functions. The array needs space $O(kt \log m) = O(\frac{1}{eps^2} \log \frac{1}{\delta} \cdot logm)$.

3.4.2 Comparison of Count Min and Count Sketch Algorithm

Guarantee

Count Min Sketch: Guarantee is in norm-1.

$$|\hat{f}_a - f_a| \leq \varepsilon K_{cms} \leq \varepsilon \|f\|_1$$

where $K_{cms} = \|f\|_1 - \|f_a\|$

Count Sketch:Guarantee is in norm-2.

$$|\hat{f}_a - f_a| \leq \varepsilon(K_{cs}) \leq \varepsilon \|f\|_2$$

where $K_{cs} = \sqrt{\|f\|_2^2 - f_a^2}$

In general, $\|f\|_2 \leq \|f\|_1$, so CMS gives a better guarantee.It's a lot better guarantee if the frequency's are evenly distributed.

Space Required

Space required is measured by the number of counters as the size of the counters is the same.

CMS:Space $= O(\frac{1}{\varepsilon} \log \frac{1}{\delta})$ counters

CM :Space $= O(\frac{1}{\varepsilon^2} \log \frac{1}{\delta})$ counters

Count sketch algorithm uses more space but gives a better guarantee than CMS.

Lecture 4

Estimating Distinct Items Using Hashing

Scribe: Andrew Cherne

4.1 The Problem

Again we are in the *vanilla streaming model*. We have a stream $\sigma = \langle a_1, a_2, a_3, \ldots, a_n \rangle$, with each $a_i \in [m]$, 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. When d is calculated exactly, its value is equal to the *zeroth-frequency moment*, denoted F_0. This is an important problem with many real world examples. Estimating the number of distinct elements has applications in databases, network statistics and Internet analysis.

4.2 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 3 algorithms, the third of which we are presenting.

4.3 Analysis of the Algorithm

4.3.1 Space Complexity

Since there are 3 distinct components to the structure of this algorithm, the total space required is the sum of these. First is the hash function h, which requires $O(\log m)$ space. The secondary hash function, g, requires $O(\log m + \log(1/\varepsilon))$ space. Finally we need the space of the buffer, which, using this clever implementation with the secondary hash function, g, stores at most $1/\varepsilon^2$ elements, each requiring space of $O(\log(1/\varepsilon)(\log\log m))$. Therefore the overall space complexity of the algorithm is $O(\log m + 1/\varepsilon^2 \cdot (\log(1/\varepsilon) + \log\log m))$.

```
  **Initialize** : choose a 2-universal hash function $h : [m] \to [m]$ ;
  let $h_t$ = last $t$ bits of $h$ $(0 \le t \le \log m)$ ;
  $t \leftarrow 0$ ;
  $B \leftarrow \emptyset$ ;
  choose a secondary 2-universal hash function $g : [m] \to [O(\frac{\log m}{\varepsilon^2})]$ ;
  **Process** $j$:
1 if $h_t(j) = 0^t$ then
2     $B \leftarrow B \cup \{g(j)\}$ ;
3     if $|B| > \frac{c}{\varepsilon^2}$ then
4         $t++$ ;
5         rehash $B$ ;
6
7 **Output** : $|B|2^t$ ;
```

4.3.2 Analysis: How Well Does The BJKST Algorithm Do?

Let $t^\star$ be the final value of t after processing the stream. Then we say that

$$\text{FAILURE} \equiv ||B|2^{t^\star} - d| \ge \varepsilon d$$

If the deviation is $\ge \varepsilon d$ (where d is the number of distinct elements), then it is a failure event (this is just relative error).

Let $X_{t,j}$ be indicator random variable such that

$$X_{t,j} = \begin{cases} 1 & \text{if } h_t(j) = 0^t \text{ (with probability } \frac{1}{2^t}\text{);} \\ 0 & \text{otherwise.} \end{cases}$$

and $X_t = \sum_{j:f_j>0} X_{t,j}$ is a sum of d pairwise independent identically distributed indicator random variables. That is, X_t is the number of elements in the buffer using h_t.

$$\mathbb{E}[X_t] = \frac{d}{2t}$$

$$\begin{aligned} Var[X_t] &= \sum_j Var[X_{t,j}] \\ &\le \sum_j \mathbb{E}[X_{t,j}] \\ &= \mathbb{E}[X_t] \\ &= \frac{d}{2^t} \end{aligned}$$

$$\Longrightarrow Var[X_t] = \mathbb{E}[X_t] = \frac{d}{2^t}$$

$$\Longrightarrow \text{FAILURE} \equiv |X_{t^\star}2^{t^\star} - d| \ge \varepsilon d$$

Continuing the equality from above, and dividing both sides by 2^t,

$$\text{FAILURE} \equiv \bigcup_{t=0}^{\log m} (|X_t - \frac{d}{2^{t^*}}| \geq \frac{\varepsilon d}{2^t} \cap t^* = t)$$

If $t^\star = t$, then the above is just equal to $|X_{t^\star} 2^{t^\star} - d| \geq \varepsilon d$ (union on all possible values of t).

For small values of t, $|X_t - \frac{d}{2^t}| \geq \frac{\varepsilon d}{2^t}$ is unlikely.

Similarly, for large values of t, $t^* = t$ is unlikely.

Let s be an integer such that

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

Where anything greater than s is considered large and anything less than $s-1$ is considered small. We are starting the analysis from $t = 1$ since at $t = 0$ there would be no failure event:

$$\subseteq \bigcup_{t=1}^{s-1} (|X_t - \frac{d}{2^t}| \geq \frac{\varepsilon d}{2^t}) \cup \bigcup_{t=s}^{\log m} (t^* = t)$$

$$\subseteq \bigcup_{t=1}^{s-1} (|X_t - \mathbb{E}[X_t]| \geq \varepsilon \sqrt{\frac{d}{2^t}} \sigma X_t) \cup (t^* \geq s)$$

(in Chebyshev form)

$$\begin{aligned} Pr[t^\star \geq s] &= Pr[x_{s-1} > \frac{c}{\varepsilon^2}] \text{ (Chebyshev)} \\ &\leq \mathbb{E}[x_{s-1}] \frac{\varepsilon^2}{c} \text{ (Markov)} \end{aligned}$$

$$\begin{aligned} Pr[\text{FAILURE}] &\leq \sum_{t=1}^{s-1} \frac{2^t}{\varepsilon^2 d} + \frac{d}{2^{s-1}} \frac{\varepsilon^2}{c} \\ &\leq \frac{2^s}{\varepsilon^2 d} + \frac{d\varepsilon^2}{2^{s-1} c} \\ &\leq \frac{1}{12} + \frac{48}{c} \\ &= \frac{1}{6} \text{ with } c = 576 \end{aligned}$$

4.4 Future Work

In their paper, the authors note that this problem has been shown to have a lower bound of $\Omega(\log m)$ and $\Omega(1/\varepsilon)$ (via reduction of the INDEX problem). Interesting future work would be to devise an algorithm that uses polylog space or a lower bound of $\Omega(1/\varepsilon^2)$.

Lecture 5

Estimating Frequency Moments

Scribe: Robin Chhetri

5.1 Background

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 $f_1 + \cdots + f_n = m$.

We have already seen that the count-min-sketch algorithm with parameters (ε, δ).Please note that, [Pr[answer not with ε relative error]] $\leq \delta$. The Count Min-Sketch gives $\left|\hat{f} - f_i\right| \leq \varepsilon \|\mathbf{f}\|_1$ with high probability with $\tilde{O}(1/\varepsilon)$ space where $\tilde{O}(.)$ means that we ignore factors polynomial in $logm, logn, log 1/\varepsilon$ etc

In Count-Sketch Algorithm,we have $\left|\hat{f} - f_i\right| \leq \varepsilon \|\mathbf{f}\|_1$ with high probability and the space requirement is $\tilde{O}(1/\varepsilon^2)$. The question here is how to estimate $\varepsilon \|\mathbf{f}\|_1$

Please note the $\tilde{O}$ notation. Typically, the space is $O\left(f(\varepsilon).log\frac{1}{\delta}\right)$ but it is more convenient to express it in the form of $\tilde{O}$. Therefore, we will be using the $\tilde{O}$ notation quite a bit.

The k^{th} frequency moment F_k is defined by

$$F_k = \sum_m |f_m|^k$$

Alon, Matias, and Szegedy or AMS[96] gave an ε, δ approximation for computing the moment. This is what we will cover in this lecture.

First some background:The first frequency moment $\|\mathbf{f}\|_1$ is

$$\|\mathbf{f}\|_1 = |f_1| + |f_2| + ... + |f_n|$$

If only positive updates are taken, that is there are no deletions but only insertions, then

$$\|\mathbf{f}\|_1 = f_1 + f_2 + ... + f_n = m$$

Clearly, m is trivial to compute.

In fact,maintaining running sum also works, as frequencies stay positive, which implies strict turnstile model.

How to we find $\|\mathbf{f}\|_2$? Strictly speaking, we know that sum of squares of frequencies is the 2nd frequency norm, that is

$$\|\mathbf{f}\|_2 = \sqrt{f_1^2 + f_2^2 + ... + f_n^2} = \sqrt{F}$$

But unfortunately, we can't use this here. This is because we need to find the individual frequencies, f_1,f_2, .. and so on but we don't have space for this computation.

Therefore, we give a general algorithm. The K^{th} frequency moment of the stream is

$$F_k = F_k(\sigma) = f_1^k + f_2^k + ... + f_n^k$$

We assume that we are only taking positive updates, that is

$$f_j = |f_j|$$

The zero moment of the set of frequencies of F, F_0 is

$$F_0 = f_1^0 + f_2^0 + ... + f_n^0$$

Please note that ,we are taking a convention $0^0 = 0$ here as some of the frequencies could be zero.

This equals to the number of distinct elements (DISTINCT-ELEMENTS).

Similarly, the first moment is simply the sum of frequencies or m, which takes $O(logm)$ space.

$$F_1 = f_1 + f_2 + ... + f_n = m$$

And, the second moment is

$$F_2 = f_1^2 + f_2^2 + ... + f_n^2$$

This takes $\tilde{O}\left(\frac{1}{\varepsilon^2}loglogn + logn\right)$ space

5.2 AMS estimator for F_k

We pick an uniform random token from the stream. We count the number of times, our token appears in the remaining stream. And we output $m(r^k - (r-1)^k)$. Please note that we don't know the length of the stream before we choose the token. Eventually, we will run many copies of this basic estimator to get our final estimation with better guarantees.

The space for the algorithm is $O(logm + logn)$.

```
Initialize : $M \leftarrow 0$;
$R \leftarrow 0$;
$Token \leftarrow 1$
Process $j$:
1 $m \leftarrow m + 1$ ;
2 With Probability $\frac{1}{m}$ do:
3  $token \leftarrow j$ ;
4  $r \leftarrow 0$ ;
5 if $j == token$ then
6     $r \leftarrow r + 1$ ;
7  Output  : $m(r^k - (r-1)^k)$;
```

5.3 Analysis of the Algorithm

How to compute the expectation?

Let N be the position number of token picked by A. Then, $N \in_R [m]$ where $\in_R$ means that they are uniformly random.

We can think of the choice of m as a two step process

1:Pick a random token value from $[n]$

$$Pr[picking\ j] = \frac{f_j}{m}$$

2:Pick one of the P_j of j uniformly at random

$$\begin{aligned}
\mathbb{E}[m(R^k - (R-1)^k] &= m\mathbb{E}[R^k - (R-1)^k] \\
&= m \sum_{j=1}^{n} \frac{f_j}{m} \sum_{i=1}^{f_j} ((f_i - i + 1)^k - (f_j - i)^k) \frac{1}{f_j} \\
&= m \sum_{j=1}^{n} \frac{f_j}{m} \sum_{i=1}^{f_j} (i^k - (i-1)^k) \frac{1}{f_j} \\
&= \sum_{j=1}^{n} \sum_{j=1}^{f_j} (i^k - (i-1)^k) \\
&= \sum_{j=1}^{n} f_j^k \\
&= F_k
\end{aligned}$$

which is the k^{th} norm. Therefore, the estimator has the right expectation value. Even if errors are large in the algorithm, we can bring them down in subsequent runs.

Now, to compute the variance

$$Var[X] = \mathbb{E}[X^2] - \mathbb{E}[X]^2 \le \mathbb{E}[X^2]$$

We pick the token at random and use the occurrence,

$$\begin{aligned} Var[X] &= m\mathbb{E}^2[(R^k - (R-1)^k)^2] \\ &= m^2 \sum_{j=1}^{n} \frac{f_j}{m} \sum_{i=1}^{f_j} \frac{1}{f_j}(i^k - (j-1)^k)^2 \\ &= m \sum_{j=1}^{n} \sum_{i=1}^{f_j} (i^k - (i-1)^k)^2 \end{aligned}$$

We are going to crudely estimate this using upper bounds, by using: $(x^k - (x-1)^k)^2 \leq kx^{k-1}(x^k - (x-1)^k)$ from mean value theorem

Get:

$$\begin{aligned} Var[X] \leq m \sum_{j=1}^{n} \sum_{i=1}^{f_j} ki^{k-1}(i^k - (i-1)^k) &\leq m \sum_{j=1}^{n} kf_j^{k-1} \sum_{i=1}^{f_j} (i^k - (i-1)^k) \\ &= m \sum_{j=1}^{n} kf_j^{k-1}.f_j^k \\ &= mk \sum_{j=1}^{n} f_j^{2k-1} \end{aligned}$$

i.e,

$$Var[X] \leq kF_1 F_{2k-1} \leq kn^{1-\frac{1}{k}} F_k^2$$

Using

$$F_1 F_{2k-1} \leq n^{1-1/k} F_k^2$$

Here, we have the expected value is right, and we have a bound on the variance. We want to compute the average to bring down the ε error.

Now, we have

$$E[X_i] = \mu$$

Let $X_1, X_2, ..., X_n$ be independent and identically distributed random variables

And

$$Var[X_i] \leq \frac{V}{T}$$

Chebyshev inequality gives,

$$\begin{aligned} Pr[Y + \mathbb{E}[Y]] \geq \varepsilon\mathbb{E}[Y] &= Pr[|Y - \mathbb{E}[Y]| \geq \frac{\varepsilon\mathbb{E}(Y)}{\sigma(Y)}.\sigma(Y) \\ &\leq \frac{\sigma^2(Y)}{\varepsilon^2\mathbb{E}[Y]^2} \leq \frac{V/T}{\varepsilon^2\mu^2} \\ &= \frac{V^2}{\varepsilon^2\mu^2}\frac{1}{T^2} \end{aligned}$$

We want (ε,1/3). So, we pick T so that this becomes 1/3. Thus, choice of $T = \dfrac{3V}{\varepsilon^2\mu^2}$

remembered as

$$\frac{3Var[X_i]}{\varepsilon^2[X_i]^2}$$

Basic estimator X has $\mathbb{E}[X] = F_k$

$$Var[X] \le kn^{1-1/k}F_k^2$$

Therefore, for failure probability $1/\delta$, number of repetitions required for failure

$$\begin{aligned} &= \frac{3}{\varepsilon^2}\frac{kn^{1-1/k}F_k^2}{F_k^2} \\ &= O\left(\frac{1}{\varepsilon^2}kn^{1-1/k}\right) \end{aligned}$$

For F_2, space is

$$= O\left(\frac{1}{\varepsilon^2}n^{1/2}(logm + logn)\right)$$

which is not great, but is pretty good. In the next lecture, we will see something specific for F_2 that will bring it down to log space.

Lecture 6

Why does the amazing AMS algorithm work?

Scribe: Joe Cooley

Normal distribution Probability distribution on the reals whose shape looks like the plot drawn in class.

Gaussian distribution A high dimensional analogue of a 1-dimensional normal distribution. More precisely, it's a vector $(z_1, z_2, \ldots, z_n)\ \varepsilon\ \Re^n$ (reals) where each z_i is drawn from a normal distribution. When drawing a random number n from a normal distribution, it will likely equal 0. Probability exponentially decreases as numbers become large.

The following equation desribes a gaussian distribution:

$$y \ = \ \exp^{-x^2}$$

or more generally,

$$y \ = \ \alpha \exp^{-\beta(x-\gamma)^2}$$

One can do linear combinations of gaussian distributions and still get gaussians. For example, $\vec{z}, \vec{w} \rightarrow A\vec{z} + B\vec{w}$.

$$\text{with probability} \begin{cases} p & \text{pick } \vec{z}_1 \sim \text{Gaussian}_1 \\ 1-p & \text{pick } \vec{z}_2 \sim \text{Gaussian}_2 \end{cases}$$

Define $F_2 = \sum_{j=1}^{n} {f_j}^2$.

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

This works because mixtures of Gaussians give gaussians. Gaussian distributions behave ”nicely” under a certain set of conditions.

Consider the $N(0, 1)$ distribution. "0" refers to centering distribution on $\gamma = 0$, and "1" refers to "thickness" of curve, i.e., σ (stdev) of resulting distribution $= 1$.

Recall that in the discrete scenario, each outcome in the set of outcomes has a weight, and the weights sum to 1.

What is the probability distribution on the reals, i.e., what's the probability of getting a number in the area of X?

The probability distribution on $\Re$ can be specified in two ways.

1. probability density function (pdf)

$$F : R \rightarrow [0, 1] \; s.t. \int_{-\infty}^{\infty} f(x)\, dx = 1,$$

   i.e., assign a weight to every real number s.t. their sum is 1.

2. cumulative distribution function (cdf)

$$F \rightarrow [0, 1], \; F(t) = \Pr[X \leq t] = \int_{-\infty}^{\infty} f(x)dx, \; f(x) = F'(x)$$

   Note that X is drawn from a distribution and is at most t.

Back to the N(0,1) distribution. The pdf $= \frac{1}{\sqrt{2\pi}} e^{-x^2}$ and has the following property.

First, let $X_1, X_2, \ldots, X_n \sim N(0, 1)$ i.i.d., and let $a_1, a_2, \ldots, a_n \;\varepsilon\; \Re$

Theorem 6.0.1. $a_1X_1 + a_2X_2 + .. + a_nX_n \sim \sqrt{a_1^2 + a_2^2 + .. + a_n^2}X$*, where* $X \sim N(0, 1)$

(proof...fair amount of calculus)

For the intuition, imagine $a_1X_1 + a_2X_2 + .. + a_nX_n$ are orthogonal unit vectors in n-dimensions, length (the $\|L\|_2$-norm) in euclidean sense of vector would be $\sim \sqrt{a_1^2 + a_2^2 + .. + a_n^2}X$. Adding normal random variables like these is like taking length of orthogonal vectors in high dimensions

Recall that $\|\hat{z}\|_p = (\sum_{i=1}^{n} |z_i|^p)^{\frac{1}{p}}, \; 0 < p \leq 2$

Definition 6.0.2. A distribution μ on $\Re$ is p-stable if $X_1, X_2, .., X_n \sim \mu$ i.i.d. (distributed according to μ) and $\vec{a} = (a_1, \ldots, a_n) \;\varepsilon\; \Re^n$, then $a_1X_1 + .. + a_nX_n \sim \|\vec{a}\|_p X$.

Theorem 6.0.3. $N(0, 1)$ *is* 2*-stable*

Theorem 6.0.4. $\forall \; p \;\varepsilon\; (0, 2], \exists$ *a p-stable distribution (constructive)*

How do we generate reals? Given some precision b, we can generate real numbers in time linear in b (# of bits needed); the underlying distribution should be close to b. (We "just" need to generate a fair coin.)

Theorem 6.0.5. *Cauchy distribution is* 1*-stable where the Cauchy pdf is* $\frac{2}{\pi}\frac{1}{1+x^2}$ *(a fat tail distribution).*

Intuitively (in a plot), at 0, y = 1 and the drop is slow, i.e., it has a fat tail. No well-defined variance (∞ variance), integrating from $-\infty$ to ∞ gives ∞. To compare distributions with infinite variances, we find the highest moment that is still finite.

Back to algorithms...

Stream problem

Estimate $F_p = \|\vec{f}\|_p^p$, or equivalently, estimate $\|\vec{f}\|_p \rightarrow L_p$.

"Algorithm A_1**"** (we'll handle reducing memory later)

```
Initialize :
1   Choose $n$ random reals $X_1, \ldots, X_n \sim \mu$, a $p$-stable distribution ;
2   $x \leftarrow 0$ ;
Process $j$:
3   $x \leftarrow x + X_j$ ;
Output : $|x|$
```

```
Initialize :
1   Choose $n$ random reals $X_1, \ldots, X_n \sim \mu$, a $p$-stable distribution ;
2   $x \leftarrow 0$ ;
Process $j$:
3   $x \leftarrow x + cX_j$ // with turnstile model;
Output : $|x|$
```

Assume frequency is bounded by some constant, which would bound the counter size. Thus, storing x requires $O(\log(m))$ bits provided we guarantee each $|X_i| \leq O(1)$

Problems with real numbers

- X precision not bounded
- X range not bounded

Just truncate large values because the probability of getting a large is small (see plot); we'll just add to the error.

Algorithm operation
Token arrives (j,c) $\rightarrow$ real storage $\rightarrow$ access X_j portion of random string.

Why does this work?
Consider $p = 1$:

Let X be the final x of the algorithm, then by 1-stability $X \equiv \|f\|_1 Z$, where $Z \sim \mu$

$X = f_1 X_1 + .. + f_n X_n$

Where in the 1-stable case, so we're using the Cauchy distribution scaled by the L_1-norm.

Why do we output $|x|$ instead of x?

Because the variance isn't defined, we can't compute expectation. Instead of computing expectation (like all other solutions we've seen except for Misra-Gries).

We're going to take the median of values and argue that this is reasonable. To do this, we need to take the median of a random variable. What does this mean? How do we take the median of a random variable?

Consider the cdf of a Cauchy distribution. What does median mean in this context? Intuitively, when taking a median, arrange values on a line and take middle value. When we have a continuous distribution, we'd like to choose the median where $\frac{1}{2}$ the mass is on one side, and half on the other.

Definition 6.0.6. For a random variable $R \sim \xi$, "median of R" or "median of ξ" $= t,\ s.t.\ Pr[R \leq t] = \frac{1}{2}$, i.e., $t,\ s.t.\ \int_{-\infty}^{t} \xi(x)dx = \frac{1}{2}$, i.e., $t,\ s.t.\ \Xi(t) = \frac{1}{2}$, i.e., $\Xi^{-1}(\frac{1}{2})$.

How do we know that a unique t exists to satisfy the median property? Derivative assumptions must hold (we won't discuss those here).

Algorithm "A_2" (Piotr Indyk, MIT)

Run $k = \frac{c}{\varepsilon^2}\log(\frac{1}{\delta})$ copies of A_1 in, and output the median of the answers.

Theorem 6.0.7. *This gives (ε, δ)-approximation to $L_1 = \|f\|_1 = F_1$.*

Consider that we have 2 streams, each defining a histogram. We'd like to the know the, for example, L_1 distance between the two vectors. If we'd like to know the L_2 distance, use the previous algorithm where X^2 is output.

Proof. Essentially, use a good old Chernoff bound, but first suppose $X \sim$ Cauchy (X is distributed according to Cauchy) □

Claim median$[|X|] = 1$ (corollary, median$[a|X|] = |a|$ by linearity)

Proof. pdf of $|X| = g$ where

$$\begin{cases} g(x) = 0, & x < 0, \\ \frac{2}{\pi}\frac{1}{1+x^2}, & x > 0 \end{cases}$$

The median = t $s.t.$

$$\begin{aligned} \frac{1}{2} &= \int_{-\infty}^{t} g(x)dx \\ &= \frac{2}{\pi}\int_{-\infty}^{t} \frac{dx}{1+x^2} \\ &= \frac{2}{\pi}\arctan(t) \end{aligned}$$

$$\begin{aligned} \Rightarrow \quad \arctan(t) &= \tfrac{\pi}{4} \\ \Rightarrow \quad t &= \tan(\tfrac{\pi}{4}) \\ &= 1 \end{aligned}$$

□

What's the intuition behind what we're saying? If we take a random variable from the Cauchy distribution, the median of the value is close to 1. If we take a bunch of values, the median is even more likely to be closer to 1. We can use the Chernoff bound to show this.

Claim

Let $X_1, .., X_k$ (from definition of A_2) $\sim$ |Cauchy|, then

$$\Pr[median(X_1, \ldots, X_n) \notin [1 - O(\varepsilon), 1 + O(\varepsilon)]] \le \delta$$

How do we prove this? Imagine plotting cdf of |Cauchy| 1 is the median, so curve touches $\frac{1}{2}$ here.

If the median is too small, let's examine one value.

$$\Pr[X_1 \le 1 - O(\varepsilon)] \le \frac{1}{2} - \varepsilon,$$

by continuity and bounded derivative of cdf of |Cauchy|.

So, how much do we need to twiddly $1 - O(\varepsilon)$ to hold the inequality $\frac{1}{2} - \varepsilon$?

$$\begin{aligned} &\Pr[X \le t] \\ &\Pr[X \le 1] = \tfrac{1}{2} \end{aligned}$$

$\Pr[X \le 0.99]$ is close to $\frac{1}{2}$ (we assume we're bounded derivative property we haven't discussed.)

$\mathbb{E}[i \ s.t. \ X_i \le 1 - O(\varepsilon)] \le (\frac{1}{2} - \varepsilon)k$

Plugging into the Chernoff bound, we have

$$\Pr[\text{median} \leq 1 - O(\varepsilon)] \leq \exp^{-\varepsilon^2 k O(1)} \leq \frac{\delta}{2}$$

What a messy algorithm!

- How many bits of precision do we need?
- How do we generate from a distribution?
- etc.

A taste of later...
We can generate a large random string with small random string and a PRNG.

Based on the more general algorithm, A_1, if we chose gaussian variables, we could compute the L_2 norm. Using yesterday's algorithm $(-1, 1)$, behaves like gaussian distribution.

If $Y_1, Y_2, .., Y_n \varepsilon_R -1, 1$ then $\vec{Y} = (Y_1, .., Y_n)$ is "close" to $\vec{Z} = (Z_1, .., Z_n)$ where $Z_i \leftarrow N(0, 1)$ for $n \rightarrow \infty$.

An open question posed by Piotr Indyk
Is there some way to choose values from a simple distribution with small of independence that happens to work, i.e., replace large array of reals w/ $(-1, 1)$, i.e., we don't have to rely on a heavy hammer (i.e., a PRNG).

Lecture 7

Finding the Median

Scribe: Jon Denning

7.1 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 7.1.1. 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 7.1.2. 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:

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

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

7.2 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 7.2.1. *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

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

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

$$L_{i+1,j} = \min_k (L_{i,2k} + L_{i,2(j-k)}) \tag{7.1}$$

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.

$$U_{i+1,j} = \max_k (U_{i,2k} + U_{i,2(j-k+1)-1}) \tag{7.2}$$

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 (7.1) and (7.2) satisfy lemma.) □

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 7.2.2. *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,

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

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 8

Approximate Selection

Scribe: Alina Djamankulova

8.1 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.

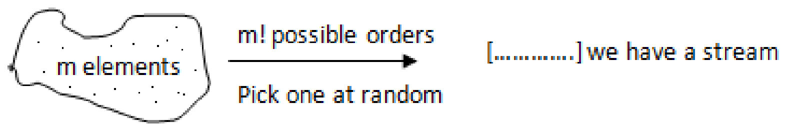
1. Given target rank r, return element of rank $\approx r$. The important note is that we are not looking for the approximate value of r element, which does not make a lot of sense.
   Let us define a notion of relative rank. Relative rank is computed the following way

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

   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]

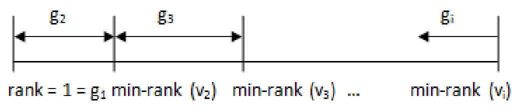


8.2 Greenwald - Khanna Algorithm

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, Δ_1), ...(v_s, g_s, Δ_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

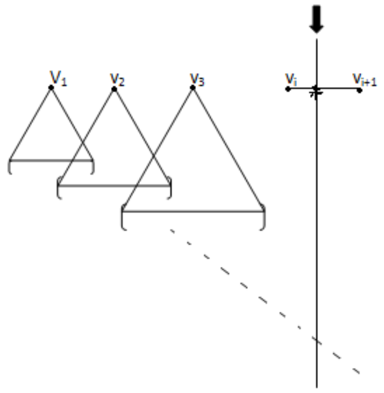
```
   Initialize :
1     Starting reading elements from the stream;
2     $m \leftarrow 0$ ;
   Process a token $v$:
3     Find $i$ such that $v_i < v < v_{i+1}$;
4     $v$ becomes the "new" $v_{i+1} \longrightarrow$ associated tuple = $(v, 1, \lfloor 2\varepsilon m \rfloor)$;
5     $m \leftarrow m + 1$;
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 $i^th$ 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) = $sum^i_{j=1} g_j + \Delta_i$

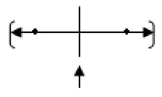
Identify i such that $v_i < v < v_{i+1}$
Then rank(v) $\in$ [min-rank(v_i), max-rank(v_{i+1})]
In this statement min-rank(v_i) = a and max-rank(v_{i+1}) = b. So we do approach to the real rank of the element with a and b filters.



rank (v) ≥ min-rank (v_i) = $sum^i_{j=1} g_j$
rank (v) ≤ 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 $g_i + \Delta$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

8.3 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 9

Graph Stream Algorithms

Scribe: Ryan Kingston

9.1 Graph Streams

In a graph stream, tokens represent lines in a graph, where the vertices are known in advance. Duplicate pairs can be represented in a form that is convenient to the problem, such as weighted lines, parallel lines, etc. Given n vertices $\{1, 2, ..., n\}$, stream = $\langle(u_1, v_1), (u_2, v_2), ..., (u_m, v_m)\rangle$, with each $(u_i, v_i) \in [n]$.

Finding the most interesting properties of the graph requires $\Omega(n)$ space, e.g. "Is there a path of length 2 between vertices 1 and 2?"

The edges arrive as follows: $(1, v)$ for $v \in S$ where $S \subseteq \{3, 4, ..., n\{$, then $(v, 2)$ for $v \in T$ where $T \subseteq \{3, 4, ..., n\{$. The question then boils down to "is $S \cap T \neq \phi$"? This is known as the Disjointess Problem, which is known to require $\Omega(n)$. Graph streams are of the "semi-streaming model" with slightly different goals as the streams we've seen previously. Namely, to make the $(\frac{space}{n})$ as small as we can, and $O(n \cdot poly(log n))$ is our new "holy grail".

9.2 Connectedness Problem

```
Initialize   : $F \leftarrow \phi$, where F is a set of edges ;
Process $(u, v)$:
1 if $F \cup \{(u, v)\}$ does not contain a cycle then
2     $F \leftarrow \{(u, v)\} \cup F$ ;
3
Output       : if $|F|$ (The number of edges) $= n - 1$ then
    Yes
else
    No
```

The UNION-FIND data structure can be used to determine connected components in the graph. For example, if component u and component v are equal, then (u, v) forms a cycle and the edge can be discarded, otherwise the components can be merged. This ensures that the graph will be maintained as a forest. Note that this works only on *insertion-only* graph streams (edges in the stream do not depart from the graph).

9.3 Bipartite Graphs

A bipartite graph is a graph with no odd cycles, also known as a 2-colored graph. If a graph is bipartite, then a depth-first search should traverse back and forth between the bipartitions. To discover whether a graph is bipartite or not can be found by checking that the graph has no odd cycles.

Definition: Monotone properties - Properties that are not destroyed by adding more edges to the graph. For example, once a graph is discovered to be connected, adding more edges cannot cause the graph to become disconnected. The same holds true for bipartite graphs; A graph that is discovered to be non-bipartite, it is always going to be non-bipartite after adding more edges.

```
Initialize : $F \leftarrow \phi$;
$Ans \leftarrow$ "Yes"
Process $(u, v)$:
if $F \cup \{(u, v)\}$ does not contain a cycle then
    $F \leftarrow \{(u, v)\} \cup F$ ;
else
    if cycle created by $(u, v)$ is odd then
        Ans ← "No"

Output : Ans
```

Outline of proof, given an input graph G:

1. Information retained allows us to compute a 2-coloring of G.
2. Information discarded doesn't affect this coloring or its validity.
3. The first edge to cause odd cycle violates the coloring.

9.4 Shortest Paths (Distances)

At the end of the stream, given vertices x, y, estimate $d_G(x, y) = min \left\{ \begin{array}{l} \text{\# edges in an x-y path in } G \\ \infty, \text{ if no x-y path in } G \end{array} \right\}$

```
Initialize : $H$ a subgraph of $G \leftarrow \phi$
Process $(u, v)$:
if $H \cup \{(u, v)\}$ does not contain a cycle of length $< t$ then
    $H \leftarrow H \cup \{(u, v)\}$

Output : Given query $(x, y)$, output $d_H(x, y)$
```

Quality analysis: $d_G(x, y)$ the actual distance $\leq d_H(x, y) \leq t \cdot d_G(x, y)$

We say that H is a *t-spanner* of G if it contains no cycles of length $< t$. A smaller t will give us a more accurate answer but requires more space. The *girth* of a graph is the length of its shortest cycle. Here, we are ensuring that girth$(H) \leq t$.

Theorem 9.4.1. *A graph on n vertices with girth* $\geq t$ *has* $O(n^{1+\frac{2}{t}})$ *edges. This is known as Bollobá's Theorem.*

9.4.1 Algorithm Analysis

Say H has m edges (and without loss of generality, $m > n$). If we delete from H all vertices of degree $< \frac{m}{n}$. The number of deleted edges $\leq (\frac{m}{n} - 1) \cdot n = m - n$.
$\therefore$ The number of edges remaining $\geq m - (m - n) = n > 0$.

Consider a breadth-first search from any root in the remaining subgraph, $\acute{H}$. Up to $\frac{t-1}{2}$ levels, we get $\geq \frac{m}{n}$ children per level (because the branching factor is $\geq \frac{m}{n}$) $\rightarrow (\frac{m}{n})^{\frac{t-1}{2}}$ vertices. So, $n \geq$ number of vertices of $\acute{H} \geq (\frac{m}{n})^{\frac{t-1}{2}} \Rightarrow n^{\frac{2}{t-1}} \geq \frac{m}{n} \Rightarrow m \leq n^{1+\frac{2}{t-1}}$

Lecture 10

Finding Matching, Counting Triangles in graph streams

Scribe: Ranganath Kondapally

10.1 Matchings

Definition 10.1.1. Given a graph $G(V, E)$, Matching is a set of edges $M \subseteq E$, such that no two edges in M share a vertex.

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

Theorem 10.1.3. *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 10.1.4. *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, ... 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 10.1.5. *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: $wt(T(S)) \le wt(S)/\alpha$

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

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

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,

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

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

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

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

10.2 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 $\geq t$, 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 kth frequency moment of virtual stream and

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

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 = 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 11

Geometric Streams

Scribe: Adrian Kostrubiak

11.1 Clustering

11.1.1 Metric Spaces

When talking about clustering, we are working in a metric space. A metric space is an abstract notion in which points have certain distances between each other. This metric space consists of a set of M with a distance function $d\colon M \times M \to \mathbb{R}^+$ and the following properties:

1. $d(x, y) = 0 \Leftrightarrow x = y$ (Identity)
2. $d(x, y) = d(y, x)$ (Symmetry)
3. $\forall\, x, y, z\colon d(x, y) \le d(x, z) + d(z, y)$ (Triangle Inequality)

Note that if properties (2) and (3) hold and property (1) does not hold then we are working in a *semi-metric space*.

One example of a metric space is $\mathbb{R}^n$, under the distance function $d(\vec{x}, \vec{y}) = \|\vec{x} - \vec{y}\|_p$ such that $(p > 0)$.

11.1.2 Working with Graphs

We will be working with weighted graphs that are in metric space. In these weighted graphs, $M =$ the vertex set, $d =$ the shortest weighted path distance, and all weights ≥ 0. Whenever the set of points is finite, a weighted graph is the general situation.

A general idea for a clustering algorithm is as follows:
Input: A set of points from a metric space and some integer $k > 0$.
Goal: Partition the points into $\le k$ clusters in the "best" way, using a cost function.
Possible objective (cost) functions:

- Max diameter $= \max_c$(max distance between two points in c)

- Three other cost functions involving representatives with the following properties:
  clusters $c_1, \ldots, c_k$
  representatives $r_1 \in c_1, \ldots, r_k \in c_k$
  and let $x_i 1, x_i 2, \ldots,$ be points in c_i

  1. k-center:
  $$max_i \left(max_j \left(d(x_{ij}, r_i)\right)\right)$$
  2. k-median:
  $$max_i \left(\sum_j d(x_{ij}, r_i)\right)$$
  3. k-means:
  $$max_i \left(\sqrt{\sum_j d(x_{ij}, r_i)^2}\right)$$

11.2 Doubling Algorithm

The goal of the doubling algorithm is to try to minimize the k-center objective function.
The following is an outline of the algorithm:

Input stream: $x_1, x_2, \ldots, x_m \in M$, where (M, d) is a metric space, and some integer $k > 0$, which is the max number of clusters that we wish to compute.
Desired output: a set $y_1, \ldots, y_k$ of representatives such that the k-center cost of using these representatives is (approximately) minimized.

- When we see $(k+1)^{\text{st}}$ point in the input stream, suppose that the minimum pairwise distance is 1, by rescaling the distances.
- Then at this point, we know that $OPT \geq 1$, where OPT is defined as
  $$OPT = \text{cost of optimum clustering}$$
- The algorithm now proceeds in phases.
  At the start of the i^{th} phase, we know that $OPT \geq d_i/2 (d_1 = 1)$.
- Phase i does the following:
  - Choose each point as the new representative if its distance from all other representatives $> 2d_i$.
  - If we're about to choose the $(k+1)^{\text{st}}$ representative, stop this phase, and let $d_{i+1} = 2d_i$.
- If the algorithm ends with $w < k$ clusters, we are O.K. and we can just add in some arbitrary cluster. This additional cluster could be empty or not.

11.3 Summaries

A summary of a stream $\sigma = < x_1, x_2, \ldots >$ with $x_i \in U$ is any set $S \subseteq U$.
Associated with the summary is a summarization cost, $\Delta(\sigma, S)$.

With the k-center algorithm, we'll use the following function in order to determine the summarization cost:

$$\Delta(\sigma, S) = \max_{x\in\sigma}\left(\min_{y\in S}(d(x,y))\right)$$

The following is an example of a stream, with $\circ$ representing normal elements in the stream and $\otimes$ representing elements that have been chosen as representatives. In this example, σ has been processed, and the rest of the stream, π has not yet been processed.

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

We can say Δ is a <u>metric cost</u> if for any streams σ, π and their respective summaries, S, T, we have:

$$\Delta(S\circ\pi, T) - \Delta(\sigma, S) \le \Delta(\sigma\circ\pi, T) \le \Delta(S\circ\pi, T) + \Delta(\sigma, S) \qquad (1)$$

We can think of this scenario as S summarizing σ, and T summarizing the whole stream, $\sigma\circ\pi$.

11.4 k-center Problem

Problem: Summarize a stream, while minimizing a summarization cost given by the function Δ. In order to do this, the following conditions are required:

1. Δ is a metric cost
2. Δ has an α-approximate threshold algorithm, $\mathcal{A}$

Note that a threshold algorithm is an algorithm that takes as input an estimate t and a stream σ, and

- Either produces a summary S (where $|S| = k$) such that $\Delta(\sigma, S) \le \alpha t$
- Or fails, proving that $\forall\, T$ (where $|T| = k$), $\Delta(\sigma, T) > t$

Looking at the Doubling Algorithm, any one phase of the doubling algorithm is a 2-approximate threshold algorithm. If S^* is an optimum summary for σ then running $\mathcal{A}$ with threshold $t^* = \Delta(\sigma, S^*)$ will surely not fail. Therefore, it will produce a summary S with $\Delta(\sigma, S) \le \alpha \cdot t^* = \alpha \cdot OPT$.

11.4.1 Guha's Algorithm

Guha's algorithm (2009) is just one algorithm designed to solve the k-center problem in the streaming model. The idea behind this algorithm is to run multiple copies in parallel with different thresholds. So we will run p instances of the $\mathcal{A}$ algorithm all in parallel. If any run i with a threshold of t_i fails, then we can know for sure that $\forall j < i$ copies of the algorithm will fail too.

The following is an outline of Guha's Algorithm:

- Keep p (such that $(1+\varepsilon)^p = \frac{\alpha}{\varepsilon}$) instances of the $\mathcal{A}$ algorithm with contiguous geometrically increasing thresholds of the form $(1+\varepsilon)^i$ running in parallel.
- Whenever $q \le p$ of instances fail, start up next q instances of $\mathcal{A}$ using the summaries from the failed instances the "replay" the stream. When an instance fails, kill all the other instances with a lower threshold.
(Note that the threshold goes up from $(1+\varepsilon)^i \to (1+\varepsilon)^{i+p}$).
- At the end of input, output the summary from the lowest threshold alive instances of $\mathcal{A}$.

We can think of this algorithm as raising the threshold of an instance by $(1+\varepsilon)^p$ rather than starting a new instance.

11.4.2 Space Bounds

Let: $\sigma_i =$ the portion of the stream in phase i,
$S_i =$ summary of the stream at the end of phase i,
$\pi =$ the final portion of the stream,
$T =$ the final summary,
and $t =$ the estimate that $\mathcal{A}$ was using when input ended.

Then: $\Delta(S_j \circ \pi, T) \le \alpha t$ (by the α-approximate property).

$$\begin{aligned}\Delta(\sigma_1 \circ \sigma_2 \circ \ldots \circ \sigma_j \circ \pi, T) &\le \Delta(S_j \circ \pi, T) + \Delta(\sigma_1 \circ \sigma_2 \circ \ldots \circ \sigma_j, S_j) \text{ by (1)}\\ &\le \alpha t + \Delta(\sigma_1 \circ \sigma_2 \circ \ldots \circ \sigma_j, S_j)\end{aligned}$$

Since $\mathcal{A}$ was running with threshold $\frac{t}{(1+\varepsilon)^p} = \frac{\varepsilon t}{\alpha}$ at the end of σ_j then by α-approximate property,

$$\Delta(S_{j-1} \circ \sigma_j, S_j) \le \alpha \frac{\varepsilon t}{\alpha} = \varepsilon t$$

and by the metric property,

$$\begin{aligned}\Delta(\sigma_1 \circ \sigma_2 \circ \ldots \circ \sigma_{j-1} \circ \sigma_j, S_j) &\le \Delta(S_{j-1} \circ \sigma_j, S_j) + \Delta(\sigma_1 \circ \ldots \circ \sigma_{j-1}, S_{j-1})\\ &\le \varepsilon t + \Delta(\sigma_1 \circ \ldots \circ \sigma_{j-1}, S_{j-1})\end{aligned}$$

Repeating this argument $(j-1)$ times,

$$\begin{aligned}\Delta(\sigma_1 \circ \sigma_2 \circ \ldots \circ \sigma_j \circ \pi, T) &\le \alpha t + \varepsilon t + \varepsilon^2 t + \ldots\\ &= \alpha t + t\left(\frac{\varepsilon}{1-\varepsilon}\right)\end{aligned}$$

Let S^* be an optimum summary. Then the instance of $\mathcal{A}$ with a threshold of $\frac{t}{1+\varepsilon}$ failed on the input stream, so

$$OPT = \Delta(\sigma_1 \circ \sigma_2 \circ \ldots \circ \sigma_j \pi, S^*) > \frac{t}{1+\varepsilon}$$

So our output T has

$$\begin{aligned}\text{cost } &\le \alpha t + t\left(\frac{\varepsilon}{1-\varepsilon}\right)\\ &= t\left(\alpha + \frac{\varepsilon}{1-\varepsilon}\right)\\ &< \left(\alpha + \frac{\varepsilon}{1-\varepsilon}\right)(1+\varepsilon) \cdot OPT\\ &= (\alpha + O(\varepsilon)) \cdot OPT\end{aligned}$$

So we get an $(\alpha + O(\varepsilon))$-approximate algorithm.
So in using this with k-center, this will leave us with a $(2+\varepsilon')$-approximation. And the space required is

$$\begin{aligned}\tilde{O}(k \cdot p) &= \tilde{O}\left(k \cdot \frac{log(\alpha/\varepsilon)}{log(1+\varepsilon)}\right)\\ &= \tilde{O}\left(\frac{k}{\varepsilon} \cdot log\left(\frac{\alpha}{\varepsilon}\right)\right)\end{aligned}$$

Lecture 12

Core-sets

Scribe: Konstantin Kutzkow

12.1 The Problem

Core-sets

- Arise in computational geometry
- Naturally gives streaming algorithms for several (chiefly geometric) problems

Suppose we have cost function $C_P : \mathbb{R}^d \to \mathbb{R}^+$ parameterized by a set P of points in $\mathbb{R}^d$. C_P is monotonic, i.e. if $Q \subseteq P$, then $\forall x \in \mathbb{R}^d : C_Q(x) \leq C_P(x)$.

Definition 12.1.1. Problem: Min-Enclosing-Ball (**MEB**). In this case $C_P(x) = \max_{y \in P} \|x - y\|_2$, i.e. radius of smallest ball containing all points in $P \subset \mathbb{R}^2$.

Definition 12.1.2. We say $Q \subseteq P$ is an α-core-set for P if $\forall T \subset \mathbb{R}^d, \forall x \in \mathbb{R}^d : C_{Q \cup T}(x) \leq C_{P \cup T}(x) \leq \alpha C_{Q \cup T}(x)$ for $\alpha > 1$.

The first inequality always holds by monotonicity of C_P. We will think of T as disjoint of P. We want $\alpha > 1$ as small as possible.

12.2 Construction

Theorem 12.2.1. *MEB admits a $(1+\delta)$-core-set of size $O(\frac{1}{\delta^{2(d-1)}})$ for $\mathbb{R}^d$. In particular for $d = 2$ we have $O(\frac{1}{\delta^2})$.*

Proof. Let $v_1, v_2, .., v_t \in \mathbb{R}^d$ be t unit vectors, such that given any unit vector $u \in \mathbb{R}^d$ there exists $i \in [t]$ with $\text{ang}(u, v_i) \leq \theta$, where $\text{ang}(u, v) = |\arccos\langle u, v\rangle|$. (In $\mathbb{R}^d$ we need $t = O(\frac{1}{\theta^{d-1}})$, for $\mathbb{R}^2$ we need $t = \frac{\pi}{2\theta}$.) A core-set for P is equal to

$$Q = \bigcup_{i=1}^{t}\{\text{the two extreme points of } P \text{ along direction } v_i\} = \bigcup_{i=1}^{t}\{\arg\max_{x \in P}\langle x, v_i\rangle, \arg\min_{x \in P}\langle x, v_i\rangle\}$$

[1]. Note that the size of the core-set is $\leq 2t = O(\frac{1}{\theta^{d-1}})$.

We need to show that $\forall T \subset \mathbb{R}^d, x \in \mathbb{R}^d : C_{P \cup T}(x) \leq \alpha C_{Q \cup T}(x)$ for some small α (1).

Suppose z is the furthest point from x in $P \cup T$, i.e. $z = \arg\max_{x' \in P \cup T} \|x - x'\|$. $C_{P \cup T}(x) = \|x - z\|$. If $z \in Q \cup T$, then clearly $C_{Q \cup T}(x) = C_{P \cup T}(x)$ and we are done. If not, $z \in P \backslash Q$. Let $u = x - z$ and v_i be a direction such that $\text{ang}(\frac{u}{\|u\|}, v_i) \leq \theta$, exists by construction. W.l.o.g. we assume $\langle v_i, u \rangle > 0$. Let $y = \arg\max_{x' \in P} \langle x', v_i \rangle$. Then, by construction, $y \in Q$. By the constraints on y:

- $\|y - x\| \leq \|z - x\|$
- $\langle y, v_i \rangle \geq \langle z, v_i \rangle$

and we see that

$$\|y - x\| \geq \|z - x\| \cos\theta = C_{P \cup T}(x) \cos\theta$$

i.e.

$$C_{Q \cup T}(x) \geq C_{P \cup T}(x) \cos\theta$$

We have proved (1) with $\alpha = 1/\cos\theta$. Set θ such that $\frac{1}{\cos\theta} = 1 + \delta \Rightarrow \theta = \arccos \frac{1}{1+\delta} = O(\delta^2)$, which completes the proof. □

12.3 Streaming algorithm

Theorem 12.3.1. *Suppose a problem of the form "minimize C_P" where C_P is a cost function that admits a $(1 + O(\varepsilon))$-core-set of size $A(\varepsilon)$. Then the problem has a streaming algorithm using space $O(A(\frac{\varepsilon}{\log m}) \cdot \log m)$, for approximation $(1 + \varepsilon)$.*

Before proving the theorem we present two simple lemmas:

Lemma 12.3.2. *(Merge property) If Q is an α-core-set for P and Q' is an β-core-set for P', then $Q \cup Q'$ is an $(\alpha\beta)$-core-set for $P \cup P'$'*

Proof. The first inequality follows from monotonicity and for the second we have $\forall T \subset \mathbb{R}^d, \forall x \in \mathbb{R}^d$:

$$C_{P \cup P' \cup T}(x) \leq \alpha C_{Q \cup P' \cup T}(x) \leq \alpha\beta C_{Q \cup Q' \cup T}(x)$$

for $\alpha > 1$ since Q and Q' are core-sets for P and P', respectively. □

Lemma 12.3.3. *(Reduce property) If Q is an α-core-set for P and R is a β-core-set for Q then R is an $(\alpha\beta)$-core-set for P.*

Proof. From Q α-core-set for P we have $\forall T \subset \mathbb{R}^d, \forall x \in \mathbb{R}^d$:

$$C_{P \cup T}(x) \leq \alpha C_{Q \cup T}(x)$$

and analogously $\forall T \subset \mathbb{R}^d, \forall x \in \mathbb{R}^d$:

$$C_{Q \cup T}(x) \leq \beta C_{R \cup T}(x)$$

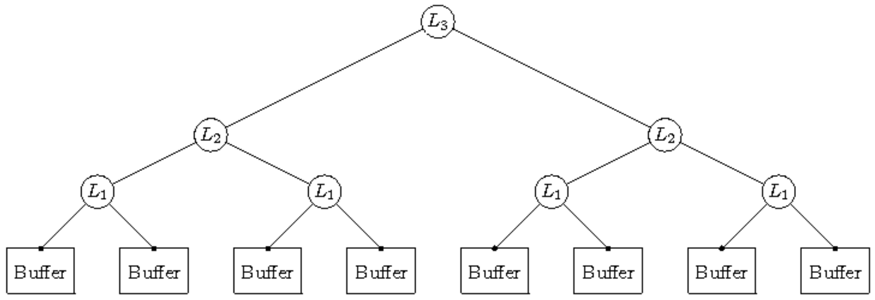
thus $\forall T \subset \mathbb{R}^d, \forall x \in \mathbb{R}^d : C_{P \cup T}(x) \leq \alpha\beta C_{R \cup T}(x)$. □

Now we prove the theorem:

[1]$\arg\min_{x \in P} f(x)$ returns x_0 such that $\forall x \in P : f(x_0) \leq f(x)$ for some real-valued function $f : P \to \mathbb{R}$

Proof. Place incoming points in a buffer of size B. When the buffer is full, summarize using next level core-set in post-order. Maintain at most one core-set per level by merging as needed.

If each core-set is a $(1+\delta)$-core-set, then final (root) core-set is a $((1+\delta)^{2\log\frac{m}{B}})$-core-set for the input stream. So final approximation is $\approx 1 + 2\log\frac{m}{B}\delta$. Set δ so that $2\log\frac{m}{B}\delta = \varepsilon$. The required space is $\log\frac{m}{B}$ (number of core-sets) times $A(\frac{\varepsilon}{\log m})$ (number of each core-set) $+B$ (size of buffer) $= O(A(\frac{\varepsilon}{\log m}) \cdot \log m)$. □



Corollary: MEB admits a $(1+\varepsilon)$-approximation and requires space $O(\frac{\log^3 m}{\varepsilon^2})$.

Lecture 13

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.

13.1 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: $X \times Y \to Z$, 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.

13.1.1 EQUALITY problem

This problem finds its application in image comparisons.

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

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

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

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

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

$$P(x, y) = \mathrm{EQ}(x, y) \tag{13.1}$$

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

$$P(x, y) = \mathrm{EQ}(x, x) = 1 \tag{13.2}$$

$$P(x', y) = \mathrm{EQ}(x', x) = 0 \tag{13.3}$$

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 13.1.2. *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)* $\bmod p = 0$.

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 $\in \{p_1, p_2 .. p_t\}$.

$$Pr[error] \leq \frac{t}{\text{no of primes in } [n^2, 2n^2]} \tag{13.4}$$

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

Using prime number theorem,

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

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

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

13.2 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.

13.2.1 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., INDEX(1100101, 3) = 0

Theorem 13.2.1.

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

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 $R^{\rightarrow}$(INDEX) $= \Omega(m)$.

Lecture 14

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	$D^{\to}(f)$	$R^{\to}_{1/3}(f)$	$D(f)$	$R_{1/3}(f)$
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 14.1: In this table, we will either prove or have already seen almost all results except $R^{\to}_{1/3}(EQ)$. 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 14.0.2. 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 14.0.3. ***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 14.0.4. *$D(EQ) \geq n$.*

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 14.0.5. *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 $x = y$, EQ$(x, y) = 1$, then $\tilde{x} = \tilde{y}$, so $F_0 = 3n$.

If $x \neq y$, EQ$(x, y) = 0$, then Hamming$(\tilde{x}, \tilde{y}) \geq 3n/100$, 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 $D(\text{EQ}) \geq n$, therefore, $D(F_0) = \Omega(n/p)$ with p passes.

Lecture 15

Set Disjointness and Multi-Pass Lower Bounds

Scribe: Zhenghui Wang

15.1 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 15.1.1. $R(\text{DISJ}) = \Omega(n)$

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 $R^{\rightarrow}(\text{DISJ}) = \Omega(n)$. $\square$

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

15.2 ST-Connectivity

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

Theorem 15.2.1. *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 $\Omega(n/p)$. □

15.3 Perfect Matching Problem

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

Theorem 15.3.1. *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 $\Omega((\frac{n}{4})^2) = \Omega(n^2)$. □

15.4 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 15.4.1 (C.-Khot-Sun '03/Granmeicr '08)**.** $R(\text{DISJ}_{n,t}) = \Omega(n/t)$

Theorem 15.4.2. *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

[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 Annu. ACM Symp. Theory Comput.*, pages 20–29, 1996.

[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.

[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.

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