CS-621 Theory Gems November 14, 2012

Lecture 16

Lecturer: Aleksander Madry *Scribes: Chidambaram Annamalai and Yann Barbotin*

1 Introduction

Today, we continue our discussion of streaming algorithms for the distinct elements problem that we started last time.

2 Recap of the Streaming Model

In the last lecture, we introduced the streaming model. The goal of this model is to help us develop algorithms that are useful in the scenarios where we want to process large amounts of data while having only very limited storage (and processing time) at our disposal. The motivating example here is a network router. This device has an extremely small memory compared to the network traffic that flows through it (in particular, it would be hopeless for it to try to story all that traffic in its memory). However, we still would like it to be able to compute some useful statistics of that traffic (e.g., number of distinct IP destinations requested, the most frequently requested IP address). Of course, at first, one might expect this to be an impossible task, but – as we will see in coming lectures – quite often one can obtain pretty satisfactory solutions.

Formally, in the streaming model (cf. Figure 2), we view the data as a long vector – a *stream* – $\boldsymbol{y} = (y_1, \ldots, y_n)$, where the length n of the stream is known in advance (and very large). The elements y_i of this stream belong to a fixed universe $\mathcal{U}$ of size m and (without loss of generality) we will assume that $\mathcal{U} := \{1, 2, \ldots, m\}$ and that $m \leq n$.

Now, the way we can access the data is quite restricted, we are only allowed to have one pass over stream. So, a streaming algorithm during this pass has to accumulate in its (limited) memory, enough information to be able to compute the desired statistic of the data represented by the stream.

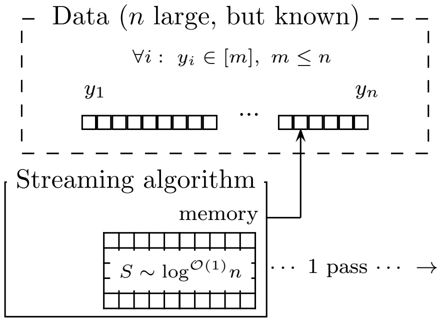
Data (n large, but known)
$\forall i: \ y_i \in [m], \ m \leq n$
y_1
y_n
Streaming algorithm
memory
$S \sim \log^{O(1)} n$
$\cdots$ 1 pass $\cdots$ $\rightarrow$

Figure 1: Illustration of the streaming model.

It is natural to wonder what should be the size of the memory that we allow our algorithm to have and usually the golden standard here will be space complexity that is poly-logarithmic in n, i.e., $\log^{O(1)} n$.

3 Distinct Elements Problem

The particular problem we will focus on today is the *distinct elements* problem in which we want to compute the total number of distinct elements occurring in the stream. Formally, given the stream $\boldsymbol{y}$ we want to compute

$$DE(\boldsymbol{y}) := |\{j \in [m] \mid y_i = j, \text{ for some } 1 \leq i \leq n\}|.$$

Probably the first streaming algorithm for this task that one can think of would be based on maintaining a characteristic vector of the already seen elements. That is, we would start with a bit vector of m bits that are initially all set to 0 and then – as we are making our pass through the date – on updating the corresponding bits to 1 whenever we see a new element. Clearly, once the pass is finished, simply outputting the number of 1s in this bit vector will give us the right answer.

Unfortunately, although this algorithm is correct, its space complexity is $O(m)$, which can be $\Omega(n)$ and thus is unacceptable for us. (Remember, we are shooting for only $\log^{O(1)} n$ space.) Furthermore, as we have seen in the last lecture, this trivial $O(m)$ space algorithm is actually optimal if we insist on an exact solution for the problem.

So, to be able to obtain some satisfactory solution, we need to relax our requirements and allow our answers to be only approximately correct and probabilistic in nature. In particular, we would like to establish the precise trade-off between the space complexity and the accuracy of our answers.

3.1 Approximate Distinct Elements Problem

To this end, let us make the notion of an approximate solution precise. We will allow our algorithms to use randomness (we will see a bit later down the road that this, again, is necessary to get anything interesting). Therefore, in addition to the parameter $\varepsilon > 0$ that will govern our desired approximation, we will also introduce a second parameter, $\delta > 0$, that bounds the probability of failure of the algorithm to provide an (approximately) correct answer.

Formally, we say that a (randomized) streaming algorithm provides an (ε, δ)-approximation to the distinct element problem iff given a data stream $\boldsymbol{y}$ with $DE(\boldsymbol{y})$ distinct elements, it outputs, with probability at least $1 - \delta$, an estimate that is within $(1 + \varepsilon)$ multiplicative error of $DE(\boldsymbol{y})$. (Note that the probability $1 - \delta$ is measured only with respect to the internal randomness of the algorithm – the stream is fixed and deterministic.)

3.2 The Decision Distinct Elements Problem

In the previous lecture, we showed that the task of solving distinct elements problem (ε, δ)-approximately can be reduced to a certain decision version of the problem (this reduction imposes a small space overhead in the process). In this decision distinct element (DDE) problem – in addition to the stream $\boldsymbol{y}$ and the desired parameters ε and δ – we also get a "guess" T on the number of distinct element $DE(\boldsymbol{y})$ in the stream. Then, our goal is to just output (with probability at least $(1 - \delta)$)

- "Yes", if $DE(\boldsymbol{y}) \geq (1 + \epsilon)T$;
- "No", if $DE(\boldsymbol{y}) \leq (1 - \varepsilon)T$;
- either "Yes" or "No", otherwise.

That is, we want to be able to distinguish between the case when $DE(\boldsymbol{y})$ is much larger than T and much smaller than T (and we do not care about the case when T is actually close to $DE(\boldsymbol{y})$).

As we already said, one can reduce the general distinct elements problem to its decision version and, in particular, in the last lecture, we established the following lemma.

Lemma 1 *If a streaming algorithm solves the decision distinct element problem (ε, δ)-approximately using $S'(\varepsilon, \delta)$ space then the (general) distinct elements problem can be solved (ε, δ)-approximately in space*

$$S(\varepsilon, \delta) = O(\varepsilon^{-1} \log n) \cdot S'(\varepsilon, O(\delta\varepsilon / \log n)).$$

In the light of the above, we focus on devising a streaming algorithm for the decision distinct element problem. We will do it in two steps. First, we will state and analyze an algorithm that is very simple but has large error probability. Then, we show how to refine this algorithm to get the desired approximation and error probability bounds.

3.3 Algorithm A

Consider the following algorithm – we will call it Algorithm A:

1. Choose a random subset $U \subseteq [m]$ of the universe of elements such that, for any $j \in [m]$,

$$Pr[j \in U] = \frac{1}{T},$$

   and the events $j \in U$ are all independent.

2. Make a pass over the stream to compute

$$\mathsf{sum}_U(\boldsymbol{y}) := \sum_{j \in U} x_j,$$

   where x_j is the number of occurrences of element j in the stream.

3. At the end, output "No" if $\mathsf{sum}_U(\boldsymbol{y})$ is equal to 0; and output "Yes" otherwise.

Before we proceed to analyzing the correctness of this algorithm, let us note that it is easy to implement it in streaming setting, as long as, we are provided with a membership oracle for the set U (i.e., as long as, during the execution of the algorithm we are always able to answer a question "Is $j \in U$?", for any j). In particular, once we ignore the space complexity needed to be able to answer such membership queries (we will get back to this point later), this algorithm has space complexity of $O(\log n)$. (This space requirement corresponds to just keeping the current value $\sum_{j \in U} x_j$ that gets updated in each step.)

To get some intuition on why this algorithm might work, let E be the set of elements appearing in the stream (and thus $|E| = \mathrm{DE}(\boldsymbol{y})$). Clearly, our algorithm answers "Yes" iff the intersection of E and U is non-empty. However, since each element j is chosen to be in U with probability $1/T$, we have that the expected size of this intersection is

$$\mathbb{E}[|U \cap E|] = \frac{\mathrm{DE}(\boldsymbol{y})}{T}.$$

As a result, if $\mathrm{DE}(\boldsymbol{y})$ is significantly smaller than T, this expectation will be much smaller than 1 and thus we hope that the intersection will be empty making the algorithm answer "No", as it should. On the other hand, if $\mathrm{DE}(\boldsymbol{y})$ is much larger than T, $\mathbb{E}[|U \cap E|] >> 1$ making the probability of non-empty intersection (and thus correctly answering "Yes") sufficiently large, as well.

Now, to substantiate and quantify the above intuition we prove the following lemma.

Lemma 2 *For all $\varepsilon > 0$ that are sufficiently small and $T \geq \varepsilon^{-2}$*

$$\begin{aligned} \Pr[\mathsf{sum}_U(\boldsymbol{y}) = 0] &\leq 1/e - \varepsilon/3 \quad , \textit{ if } \mathrm{DE} \geq (1+\varepsilon)T, \\ \Pr[\mathsf{sum}_U(\boldsymbol{y}) = 0] &\geq 1/e + \varepsilon/3 \quad , \textit{ if } \mathrm{DE} \leq (1-\varepsilon)T, \end{aligned}$$

where $\mathrm{DE} = \mathrm{DE}(\boldsymbol{y})$.

Proof As we already said, the algorithm answers "No" iff the intersection of the set E of elements in the stream and of the set U is empty. Thus, by construction of the set U and the fact that $|E| = \mathrm{DE}(\boldsymbol{y})$ we have that the probability the algorithm answer "No" is

$$\Pr[\mathsf{sum}_U(\boldsymbol{y}) = 0] = (1 - 1/T)^{\mathrm{DE}}.$$

Now, let us note that the RHS of the above expression is a monotone function and thus it is sufficient to prove the desired inequalities when DE takes one of "critical" values of $(1-\epsilon)T$ or $(1+\epsilon)T$.

To this end, let us define $c := \mathrm{DE}/T$ and thus $c \in \{1-\varepsilon, 1+\varepsilon\}$. Note that by employing Taylor expansion we have that

$$\ln(1-\frac{1}{T}) = -\frac{1}{T} + O(\frac{1}{T^2}),$$

whenever ε and thus $\frac{1}{T}$ is small enough.

Using this approximation, we get that

$$\begin{aligned}\Pr[\mathsf{sum}_U(\boldsymbol{y})=0] &= (1-1/T)^{\mathrm{DE}} = (1-1/T)^{cT},\\ &= \exp(cT\ln(1-1/T)),\\ &= \exp(-c+O(1/T)),\\ &= \exp(c+O(\varepsilon^2)),\end{aligned}$$

as $T \geq \varepsilon^{-2}$. The lemma follows by considering the case of $c = 1-\varepsilon$ and $c = 1+\varepsilon$ and simple estimation. ■

Note that the above lemma requires an assumption that $T \geq \varepsilon^{-2}$. However, if $T \leq \epsilon^{-2}$, one can just use a trivial algorithm for the problem that keeps track of all the elements seen in the stream so far. If at any point of time the number of such elements becomes at least $(1+\varepsilon)T$, this algorithms will stop tracking further elements and will answer "Yes", otherwise it will answer "No". It is easy to see that this solves the problem exactly (and in deterministic way) and the resulting space complexity is only $O(T\log n) = O(\varepsilon^{-2}\log n)$. So, the case of $T \geq \varepsilon^{-2}$ is the really interesting one here.

3.4 Algorithm B

From the Lemma 2 above, we see that the estimator used by Algorithm A can indeed differentiate between the two cases that we want to be able to distinguish between. Unfortunately, the "distance" between the "Yes" and "No" cases is relatively small and would result in pretty large error probability.

To cope with that, we will now apply a standard technique that allows one to amplify this distance and obtain an arbitrarily small probability of error. To this end, let $\delta > 0$ be our desired error probability and let us consider the following algorithm – we call it Algorithm B:

1. Run k parallel (and independent) instances of Algorithm A, where $k = C\varepsilon^{-2}\log\frac{1}{\delta}$, for some constant $C > 0$ to be fixed later. Let $U_1, \ldots, U_k$ be the random sets chosen by each of these instances.

2. At the end, let Z be the number of $\mathsf{sum}_{U_i}(\boldsymbol{y})$ that ended up being equal to zero. Answer "No" if $Z > \frac{k}{e}$; output "Yes", otherwise.

The basic idea behind the above algorithm is that, by Lemma 2, we can see that the expected value of Z should be at most $\frac{k}{e} - \frac{k\varepsilon}{3}$, if the answer should be "Yes"; and at least $\frac{k}{e} + \frac{k\varepsilon}{3}$, if the answer should be "No". So, the difference of these two expectations grows as $\frac{2k\varepsilon}{3}$. Therefore, the hope is that it is large enough that the variable Z will be very unlikely to be that far from its expected value (and thus make us confuse the two cases).

To show that this hope is well-funded and the variable Z indeed exhibits high concentration around its expected value, let us define Z_i to be a zero-one variable that is equal to 1 iff $\mathsf{sum}_{U_i}(\boldsymbol{y})$ is equal to zero. Clearly, we have $Z = \sum_i Z_i$.

Now, the crucial thing to notice is that all Z_i are independent. As a result, we can apply Chernoff bound to argue about the concentration of their sum Z. Recall that (one version) of Chernoff bound states the following

Theorem 3 *Let $X_1, X_2, \dots, X_n$ be independent zero-one random variables such that each X_i takes the value 1 with probability p_i. Then, for any $\gamma > 0$,*

$$Pr[|X - \mu| \geq \gamma\mu] \leq 2\exp(-\frac{\gamma^2\mu}{3}),$$

where $X = \sum_i X_i$, and $\mu = \sum_i p_i = E[X]$ is the expectation of X.

To get the desired bound on the probability of Algorithm B failure, consider the case when $E[Z] = \frac{k}{e} - \frac{k\varepsilon}{3}$ and thus the right answer should be "Yes". (The proof for the "No" case is completely analogous.) Applying the above Chernoff bound to variables $Z_1, \dots, Z_k$ with $\gamma = \frac{k\varepsilon}{3E[Z]}$, we get that the probability of Z being larger than $\frac{k}{e}$ (and thus ending up making Algorithm B provide a wrong answer) is at most

$$\Pr[|Z - E[Z]| > \frac{k\varepsilon}{3}] \leq 2\exp(-\frac{k^2\varepsilon^2}{3E[Z]}) \leq 2\exp(-\frac{k\varepsilon^2}{3}).$$

Given our choice of $k = C\varepsilon^{-2}\log\frac{1}{\delta}$, we can make this probability smaller than δ, by just choosing C to be sufficiently large constant.

So, our Algorithm B indeed provides a correct (ε, δ)-approximation to the decision version of distinct elements problem. Also, it is not hard to see that its space complexity is only $O(k\log n) = O(\varepsilon^{-2}\log n\log\frac{1}{\delta})$ needed to maintain k instances of Algorithm A. Together with Lemma 1, this implies an algorithm that computes (ε, δ)-approximation to the distinct elements problem in space

$$O(\varepsilon^{-1}\log n) \cdot \tilde{O}(\varepsilon^{-2}\log n\log\frac{1}{\delta}) = \tilde{O}(\varepsilon^{-3}\log^2 n\log\frac{1}{\delta}),$$

where $\tilde{O}(f)$ notation hides factors that are poly-logarithmic in f.

It turns out that by applying more sophisticated approach, one can obtain a space complexity of only $O((\varepsilon^{-2} + \log n)\log\frac{1}{\delta})$, which can be shown to be optimal (at least, when δ is constant). Also, an implementation of another algorithm (based on the similar ideas that we used above) was able to solve the distinct elements (words) problem on all works of Shakespeare with 10% error using only 128 bits of space.

3.5 Storing the Set U

There is still one point that needs to be addressed. When analyzing the space complexity of Algorithm A, we ignored the space needed to store a membership oracle for the set U. Clearly, as this set is random, a simple entropy argument shows that one requires at least $\Omega(T\log m)$ bits to store it. This is prohibitive even for moderate values of T.

To address this problem, let us first note that instead of storing the set U, we could think about storing a random function $f : [m] \to [T]$ and making $f(j) = 1$ correspond to "j is in U". Of course, this does not seem to help as storing such a random function f requires even more space – $\Omega(m\log T)$.

However, the key observation here is that to make Algorithm A work, we do not really need the function f to be fully random. One can show that by tweaking some parameters a little bit and doing more careful analysis, one can make the Algorithm A give a very similar guarantee to the one offered by Lemma 2 even if f is only "somewhat random". More precisely, it suffices that f is just 2-*wise independent hash function*, i.e., that it only has a guarantee that if for any two different $j, j' \in [m]$ and possible values $t, t' \in [T]$, the probability that $f(j) = t$ and $f(j') = t'$ is exactly $\frac{1}{T^2}$ (which would need to be the case if f was a fully random function).

It turns out that although every fully random function is 2-wise independent hash function, there exist families of 2-wise independent hash functions that are can be stored using only $O(\log m)$ bits (and thus are far from being fully random). We will see one very simple construction of such a family in the next lecture.

So, by applying 2-wise independent hash functions we can obtained the desired $O(\log n)$ space complexity of Algorithm A and our above result for distinct element problem follows.

3.6 Handling Deletions of Stream Elements

One might wonder why in the Algorithm A we chose to maintain the value of $\mathsf{sum}_U(\boldsymbol{y})$. After all, all we really need to know for the analysis of this algorithm to go through is whether there is some element from U in the stream. To compute that, we do not need to know the total number of occurrences of elements from U in the stream - we just need to know whether this number is zero or not.

The main reason for this design choice is that it allows our algorithm to work in an important generalization of the basic streaming model in which in addition to insertion of elements (as was the case so far), we also allow deletions of elements. (This extension is sometimes called the *turnstile model.*)

Note that as our estimator $\mathsf{sum}_U(\boldsymbol{y})$ is linear in the frequencies x_j of elements, we are able to maintain it also when deletions are allowed. In particular, one can show that the algorithm we developed above works in the turnstile model too, provided that the stream is *well-formed*, i.e., that the number of times an element is deleted in the stream is never bigger than the number of times it was inserted.

On more general note, it turns out that such taking linear sketches of the stream is a very powerful technique (we will see it frequently in the coming lectures) and essentially all known streaming algorithms that work in turnstile model employ it.