CS-621 Theory Gems November 21, 2012

Lecture 19

Lecturer: Aleksander Mądry *Scribes: Chidambaram Annamalai and Carsten Moldenhauer*

1 Introduction

We continue our exploration of streaming algorithms. First, we briefly discuss the sparse approximation problem – a problem of fundamental importance in streaming and signal processing – and its connection to the Count-Min algorithm. Next, we start investigation of the lowerbounds on space complexity of streaming algorithms. In particular, we revisit the question of lowerbounds for the distinct elements problem and introduce the framework of communication complexity that (as we will see later) will be instrumental in establishing a host of streaming lowerbound results.

2 Sparse Approximation

Informally, the problem of sparse approximation can be thought of as the problem of finding, for a given input vector x, a vector $\hat{x}$ that has "low complexity" and that approximates x well in some ℓ_p norm. There are many possible notions of "low complexity" of a vector, but here we want it to mean that the vector $\hat{x}$ is k-*sparse*, i.e., that it has only k non-zero coordinates, for some $k \geq 1$. (Thus, one can view $\hat{x}$ as a "short summary" or "sparse representation" of x).

More formally, for a given sparsity parameter $k \geq 1$, a desired accuracy parameter $\varepsilon > 0$, and a desired upperbound on the probability of failure $\delta > 0$, an (ε, δ)-approximation to the k-*sparse* ℓ_p-*approximation* problem is an algorithm that: given a vector x returns a k-sparse vector $\hat{x}$ such that, with probability at least $1 - \delta$, we have

$$\|x - \hat{x}\|_p \leq (1 + \varepsilon)\mathsf{Err}_p^k, \tag{1}$$

where $\mathsf{Err}_p^k := \min_{\|\bar{x}\|_0 = k} \|x - \bar{x}\|_p$ is the ℓ_p-error of the best k-sparse ℓ_p-approximation to x. (Sometimes, one considers variants of the guarantee from (1) that have different ℓ_p norms on the right and left sides of the inequality. In general, the higher p in the ℓ_p norm on the right side we can get, the better is our approximation quality.)

In the streaming setting, the k-sparse ℓ_p-approximation problem corresponds to scenario in which the vector x we want to approximate, is the vector of elements' frequencies in a stream. So, our algorithm should be able to compute a k-sparse $\hat{x}$ that ℓ_p-approximates x while having only one pass over the stream.

It is easy to see that the best k-sparse ℓ_p-approximation to x consists of choosing $\hat{x}$ to be equal to x after zeroing out all but k largest coordinates. So, one can view the k-sparse ℓ_p-approximation problem as a kind of generalization of the most frequent element estimation. (Also, note that when the frequency vector x is itself k-sparse, then $\mathsf{Err}_p^k = 0$ and thus we need to be able to reconstruct the vector x exactly, i.e., we need to have $\hat{x} = x$).

Now, how to develop streaming algorithms for the k-sparse ℓ_p-approximation problem? It turns out (cf. Problem Set 4) that in the case of $p = 1$, a simple modification of the Count-Min algorithm developed in the last lecture, provides an (ε, δ)-approximation to the k-sparse ℓ_1-approximation problem in $O(k\varepsilon^{-1} \log n \log \frac{m}{\delta})$ space.

2.1 Compressed Sensing

We also want to mention in passing that the k-sparse ℓ_p-approximation problem is also related to an important paradigm in signal processing called *compressed sensing*. The general setup here is that we have a certain n-dimensional vector (signal) x that we can access only via performing on it linear measurements $A_i^T x$, described by arbitrary vectors $A_i \in \mathbb{R}^n$. (For notational convenience, we collect these measurement vectors A_i into a measurement matrix A whose rows correspond to transposes of A_is. So, Ax is a vector storing all the results of these linear measurements on x.)

Now, in the compressed sensing setting, such linear measurements on the data are expensive, and hence we would like to minimize their number (in other words, minimize the number of rows of the measurement matrix A) while still being able to recover a good ℓ_p-approximation $\hat{x}$ of the vector x from them.

In general, trying to compute a good approximation of x from the measurements Ax is hopeless if A has significantly fewer rows than columns (i.e., if the number of measurements is significantly smaller than the dimension of x). However, the key insight behind compressed sensing is that most real-life signals have certain structure, in particular, are usually sparse in some known basis (e.g., in Fourier basis). So, it is ok to provide a recovery procedure that produces a vector $\hat{x}$ that approximates the input vector x well only if x was indeed pretty sparse.

This is formalized by requiring the vector $\hat{x}$ to provide a similar guarantee on the quality of its approximation of x as the one in k-sparse ℓ_p-approximation problem (cf. Equation (1)). (Note again that the Err_p^k term measures how close to being k-sparse the vector x is and, in particular, if x is k-sparse itself then $\hat{x}$ has to recover x exactly.)

The question of how many measurements one needs in such setting attracted a lot of attention. It turned out that if we indeed are ok with approximation quality that is dependent on Err_p^k, then number of needed measurements can be substantially smaller than n. For example, for the case of ℓ_1-approximation one can get it to be only $O(k \log \frac{n}{k})$ (for a sparsity parameter k), which is exponentially smaller than n for small k and is within constant of the information-theoretical lowerbound.

Now, there is an interesting connection between compressed sensing and streaming algorithms for the k-sparse ℓ_p-approximation problem. Namely, it is easy to see that any streaming algorithm for the latter problem that is based on linear sketches, i.e., its estimate $\hat{x}$ of the frequency vector x is based on estimators that are linear in x, provides a solution for the former problem (with k being the sparsity parameter and ℓ_p be the error measuring norm) whose number of measurements is proportional to the algorithm's space complexity. As we already mentioned earlier, linear sketches are a very popular technique in streaming algorithms and, in particular, the Count-Min algorithm (that can be used to solve k-sparse ℓ_1-approximation problem) is employing it. So, discovering this connection introduced a host of new ideas to compressed sensing area, resulting in some important advances (especially in providing algorithms with very efficient encoding and decoding).

3 Streaming Lowerbounds

So far in our treatment of streaming algorithms, we were almost exclusively focused on developing algorithms for various versions of L_p-norm estimation problems. Now, we want to proceed to looking at the dual question here: what lowerbounds on space complexity of such streaming algorithms can we establish?

Today, we start with the simplest (but still quite powerful) approach to this question – the "pigeon-hole principle"– and apply it to the L_0-norm estimation (i.e., the distinct elements problem). The general idea behind this approach is based on proving that any streaming algorithm for a given problem that has too low space complexity, can be transformed into an encoding/decoding scheme that can unambiguously compress any vector from some set of size N, to a description that has only $o(\log N)$ bits. As, by "pigeon-hole principle", existence of such scheme is impossible, this establishes the corresponding lowerbound.

We first illustrate this idea on a simple example of $\Omega(m)$ lowerbound for deterministic and exact L_0-norm estimation. (Note that we already proved this lowerbound once in Lecture 15.)

Lemma 1 *Any deterministic and exact algorithm for L_0-estimation requires $\Omega(m)$ space.*

Proof As already mentioned, the proof is by contradiction. Hence, assume there exists a deterministic streaming algorithm that performs exact L_0-norm estimation in $o(m)$ space. We will transform this

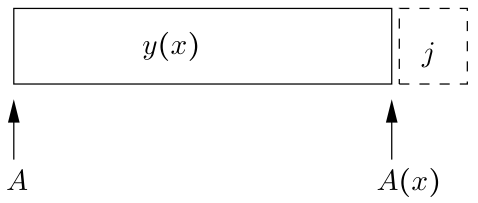

Figure 1: Encoding of x and decoding of the jth bit.

algorithm into an encoding/decoding scheme for $2^{\Theta(m)}$ different strings that uses a description of $o(m)$ bits. This will yield the desired contradiction.

We first develop the encoding scheme. Note that we do not need to care here about the running time of neither the encoding nor the decoding algorithms. All we need is just to ensure that the encoding and decoding mechanisms is unambiguous.

Fix some $x \in \{0,1\}^m$ with $\|x\|_0 = \frac{m}{2}$ – note that there is $2^{\Omega(m)}$ of such vectors. We encode this bitvector in the following way. First, we represent x as a frequency vector of a stream $\boldsymbol{y}(x)$. This can be done in a straightforward manner by just making $\boldsymbol{y}(x)$ consist of inserting one after another all the elements $j \in [m]$ such that $x_j = 1$.

Next, we run our hypothetical algorithm on $\boldsymbol{y}(x)$ and look at the content of the memory of that algorithm after this run. We denote this memory content by $A(x)$ and use it as the encoding of x.

Note that as our hypothetical algorithm has $o(m)$ space complexity, the size of $A(x)$ is $o(m)$ and that the algorithms L_0-norm estimate has to be $\frac{m}{2}$.

It remains to devise a decoding scheme that will uniquely reconstruct x from $A(x)$. We will do this by showing that we can decode from $A(x)$ whether the j-th bit x_j of x is 0 or 1.

To this end, let us fix our attention on some $j \in [m]$. Note that by our construction of the stream $x_j = 1$, iff the element j was inserted in the stream $\boldsymbol{y}(x)$. To recover the value of x_j, we run our hypothetical algorithm starting with the memory content $A(x)$ on the (one-element) stream $\boldsymbol{y}(j)$ that inserts only element j (cf. Figure 1). Clearly, the state of the algorithm after we do so is equivalent to its state after being run (from clean start) on the concatenated stream $\boldsymbol{y}(x), \boldsymbol{y}(j)$.

Now, if the algorithms' estimate of number of distinct elements does not change (from $\frac{m}{2}$ to $\frac{m}{2}+1$) during processing the stream $\boldsymbol{y}(j)$, it means that j has been already encountered in the stream $\boldsymbol{y}(x)$ and thus $x_j = 1$. Otherwise, $x_j = 0$.

Therefore, we see that we are able to reconstruct x uniquely and we indeed have an unambiguous encoding/decoding scheme that compresses any of $2^{\Omega(m)}$ strings into $o(m)$ space. So, we get a contradiction that concludes our proof. ■

In the light of the above result, we cannot hope for any reasonable streaming algorithm that solves the distinct element problem exactly, at least if this algorithm is deterministic. But, how about randomness? Does allowing the algorithm to use randomness (and fail with some small probability) enable us to solve the distinct elements problem exactly?

It turns out that the answer is still "No", but proving that requires a bit more delicate reasoning.

Lemma 2 *Any (randomized) algorithm that, with probability at least* $1 - \frac{1}{16}$*, solves the distinct elements problem exactly requires* $\Omega(m)$ *space.*

Proof

As before, let us assume for the sake of contradiction that there exists a randomized algorithms that on any input stream, with probability at least $1 - \frac{1}{16}$, returns the exact number of distinct elements in that stream. Once again, we will show that we can use such algorithm to devise a way of unambiguously encoding any of $2^{\Omega(m)}$ bitvectors in $o(m)$ space.

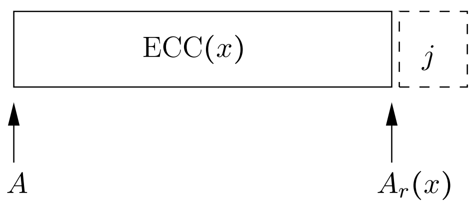

Figure 2: Encoding of ECC(x) and decoding of the jth bit of ECC(x).

To make this encoding robust in the presence of noise introduced by the fact that the algorithm is allowed to fail now, we will need an error correcting code ECC : $\{0,1\}^m \to \{0,1\}^{m'}$. We require from this code that: it has constant rate, i.e., $m' = O(m)$, each codeword has exactly $\frac{m'}{2}$ ones in it, i.e., $|\mathrm{ECC}(x)|_0 = \frac{m'}{2}$, and that the minimum distance between any two different codewords is at least $\frac{m'}{4}$. Note that this means, in particular, that if we get a codeword ECC(x) that has at most $\frac{m'}{8}$ bits corrupted then we can uniquely reconstruct x out of it. (One can devise such an error correcting code by, e.g., a simple modification of Hadamard code.)

We proceed now to defining our encoding and decoding procedure. At first, we will present a *randomized* encoding/decoding scheme. Later we will show that there exists a way of fixing the random bits used by this scheme to make it fully deterministic and unambiguous.

To this end, let us fix some $x \in \{0,1\}^m$. Consider a stream $\boldsymbol{y}(x)$ (over m' elements) that encodes the codeword ECC(x) as its frequency vector. (Again, we can do that by just inserting in the stream all elements $j \in [m']$ such that $\mathrm{ECC}(x)_j = 1$.) Note that by the property of our error correcting code, the number of distinct elements in this stream is always $\frac{m'}{2}$.

Now, let $A_r(x)$ be the memory content of our hypothetical algorithm after processing that stream. Note that $A_r(x)$ takes $o(m') = o(m)$ bits to store and it is parametrized by the bitvector r that corresponds to the values of random bits that that algorithm used during its computations. We take $A_r(x)$ to be our encoding of x.

Next, let us focus on decoding procedure. Given $A_r(x)$ we want to decode the bits of ECC(x) and hope that no more than $\frac{m'}{8}$ of them ends up being corrupt – in this case, we will be able to uniquely decode x. The way we decode the bits of ECC(x) from $A_r(x)$ is completely analogous to the approach we employed in the proof of Lemma 1. That is, to recover the bit $j \in [m']$, we run our hypothetical algorithm with memory content $A_r(x)$ on a (one-element) stream $\boldsymbol{y}(j)$ that just inserts element j (see Figure 2). (So, effectively, this corresponds to running the algorithm from clean start on the concatenated streams $\boldsymbol{y}(x), \boldsymbol{y}(j)$.) If at the end the estimate of the algorithm on the number of distinct elements is $\frac{m'}{2}$ then we assume that $\mathrm{ECC}(x)_j = 0$; otherwise, we take $\mathrm{ECC}(x)_j = 1$. Note that the decoding procedure for each such bit j is also randomized, i.e., it is parametrized by some bitvector r_j describing the value of random bits used by the algorithm when processing the stream $\boldsymbol{y}(j)$.

We want now to lowerbound the probability (over the choice of random bitvectors $r, r_1, \ldots, r_{m'}$) that the above randomized encoding of x and then subsequent decoding it, will indeed recover x back. To this end, note that for any $j \in [m']$, the probability that our encode-decode sequence fails to estimate the j-th bit of ECC(x) correctly, is exactly the probability that the hypothetical algorithm fails to provide a correct answer to the distinct elements problem on the concatenated stream $\boldsymbol{y}(x), \boldsymbol{y}(j)$. So, this probability is at most $\frac{1}{16}$.

In the light of the above, the expected number of bits of the codeword ECC(x) that will not be estimated correctly is at most $\frac{m'}{16}$. By Markov's inequality, this means that with probability at least $\frac{1}{2}$, there will be at most $\frac{m'}{8}$ corrupted bits of ECC(x) and thus we will be able to indeed recover x out of that. So, rephrasing, we just showed that for a fixed $x \in \{0,1\}^m$, at least half of the possible choices of the bitvectors $r, r_1, \ldots, r_{m'}$ makes the encoding-decoding procedure to work correctly.

However, a simple counting argument shows that the above implies that there exists a fixed choice

$r^*, r_1^*, \ldots, r_{m'}^*$ of these bitvectors such that for at least half of the possible choices of $x \in \{0,1\}^m$, encoding x as $A_{r^*}(x)$ (which is a deterministic procedure now!) and then decoding it with randomness fixed according to bitvectors $r_1^*, \ldots, r_{m'}^*$, will indeed result in correct result. So, we proved an existence of an deterministic and unambiguous encoding/decoding scheme that works for at least 2^{m-1} bitvectors and results in encoding of $o(m)$ size. The lemma follows by getting this contradiction. ■

The above lemma shows that one cannot hope for any non-trivial streaming algorithm for exact estimation of the number of distinct elements. This justifies settling for only approximate answers. However, even once we accept necessity of approximation, one might wonder if randomness is still necessary to get something interesting.

As it turns out, the answer is again "Yes" and the use of both approximation and randomness is needed here. (So, our settling for these two choices when designing the distinct elements algorithm in Lecture 16, was indeed justified.)

Lemma 3 *Any deterministic algorithm that solves the distinct elements problem up to a multiplicative error of less than* $3/2$*, requires* $\Omega(m)$ *space.*

The proof of this lemma follows the same pattern as the proofs of Lemmas 1 and 2. We again employ the same error correcting code as in the proof of Lemma 2 to encode an arbitrary bitvector $x \in \{0,1\}^m$ on the stream. One then uses the distance properties of this code to show that encoding of two different xs have to be different enough that even when only being approximately correct is sufficient, the algorithm still has to have different memory content after processing each of these encodings.

4 Communication Complexity

Although the above lowerbounds for the distinct elements problem are pretty satisfactory, such "pigeon-hole principle"-based arguments do not seem to be sufficient for obtaining lowerbound results for many other streaming problems. Therefore, we proceed to introducing a much more general framework for proving such lowerbounds: *communication complexity*. This approach turns out to be very powerful. Essentially all the known streaming lowerbounds are established via it (the "pigeon-hole principle" can be seen as just its special case). In fact, its applicability goes well beyond streaming algorithms. Most notably, it is used to prove unconditional data structure space/time lowerbounds.

In communication complexity, we consider the following setup. Alice and Bob each have an n-bit string x_A and x_B respectively – we think of them as two parts of the input. They wish to compute some predefined function $f(x_A, x_B)$ of that input. The way this computation is being performed via a very simple protocol. First, Alice sends some message to Bob (this message depends only on her part of the input x_A and the function f). Then, Bob looking at his part x_B of the input and Alice's message, responds with a message that should be equal to $f(x_A, x_B)$.[1]

Now, our goal here is to try to understand the *communication complexity* $CC(f)$ of various functions f, being the worst-case minimum of the amount of communication – i.e., the number of bits $CC(f)$ that Alice needs to send to Bob – needed to ensure that Bob is always able to determine the value of the function $f(x_A, x_B)$ correctly. (Note that when estimating this number we place no restriction on the amount of computation that Alice and Bob may need to perform between sending the messages. All we care about is just the number of bits exchanged.)

Trivially, Alice could simply send x_A over to Bob, who could then compute the answer. So, we know that in general $CC(f) = O(n)$. However, for certain functions this could be a highly suboptimal estimate. For example, if f corresponds to just computing the parity (XOR) of all the bits in x_A and x_B then the communication complexity $CC(f)$ of such f is only 1.

[1] In general, in many applications one needs to consider multi-round protocols too, but for our purposes it is sufficient to restrict our attention to single-round ones.

Interestingly though, there are certain interesting choices of functions for which the trivial $O(n)$ upperbound is actually the best possible (e.g., checking if x_A and x_B are equal, or whether there is a bit that is set to 1 both in x_A and x_B). Such “hard” functions are very useful as they are the basis of various lowerbound constructions. In the next lecture we will see examples of this when exploring a simple but powerful connection between communication protocols and streaming lowerbounds.