CSC2420: Algorithm Design, Analysis and Theory
Fall 2023
An introductory (i.e. foundational) level graduate course.

Allan Borodin

November 28, 2023

Week 11

Announcements:

- I posted a final question for Assignment 3. There is also a bonus question but only for those who are interested and have the time.
- Next week is the last week of classes and it would be good to submit grades before the holiday break.

Todays agenda
We will discuss the following topics:

- The CVM streaming algorithm for counting the number of distinct elements in the input stream.
- Sublinear time algorithms.

Estimating the number of distinct elements

We will finish our discussion of streaming algorithm with the Chakraborty, Vinodchandran and Meel (CVM) algorithm for estimating F_0, the 0^{th} frequence moment. Following CVM, we will stick with the AMS notation in which the length of the stream $a_1, a_2, \ldots, a_m$ is m with input items $a_i \in \{1, 2, \ldots, n\}$.

The goal is an (ϵ, δ) approximation in small space; that is,
$Prob[(1-\epsilon)F_0 \leq CVM \leq 1+\epsilon)F_0] \geq 1-\delta$
A substantial history has led to Blasiok [2018] who obtains an (ϵ, δ) approximtion using space complexity $O(\log n + \frac{1}{\epsilon^2} \log \frac{1}{\delta})$, which is optimal in each of the parameters n, ϵ, δ.

The first $O(\log n)$ space algorithm is due to Flajolet and Martin [1985] which relied on the (still unproven) assumption of efficiently computable hash functions with full independence. A small modification of their algorithm is used in practice. Subsequent $O(\log n)$ space algorithms require special properties of bounded independent Hash functions and are viewed as being too difficult to teach and/or not that practical.

Motivating the new algorithm

The new CVM algorithmn is conceptually simple, easy to implement and "can be taught in an undergraduate course".
In its simplest form, the space bound has a $\log m$ term. It is possible to remove this multiplicative $\log m$ dependence on m at a small loss in simplicity. An important aspect of the CVM algorithm is that it does not use any hash functions but rather is based on simple random sampling.

The algorithm is not quite as space efficient as some previous algorithms but has the significant advantage in that it can be modified to efficiently apply to related problems whereas the methods relying on hash function would not be efficient in running times.

I like simplicity!

Motivating the CVM algorithm continued

In particular, the algorithm extends nicely to the setting where input item a_i is replaced by a set $S_i \subseteq \{1, 2, \ldots, n\}$ and the objective is to estimate $|\cup_i S_i|$ assuming the collection of sets is a "Delphic Family".
A Delphic family $\{S_i\}$ is one in which the following can be done in time and space $O(\log n)$ for each set S_i:

- Determine $|S_i|$
- Sample uniformly at random an element from S_i
- For any $x \in \{1, \ldots, n\}$ determine if $x \in S_i$

Futhermnore the new algorithm also leads to a practical estimator for counting the number of satisfying assignments in a DNF formula.

Tha ideas behind the CVM algorithm

Like the simple algorithm for primality testing that works for all numbers excxept Carmichael numbers, the starting point for the CVM algorithm relies on a simple idea: Suppose with some probability p we place each input item in a bin $\mathcal{X}$ and the number of balls in the bin is k, then $\frac{k}{p}$ is a good estimator for the number of items.

However, since we want to estimate the number of distinct elements, input items that occur frequently will dominate the count.

The clever idea is to only put in the last occurence of an input item into the bin $\mathcal{X}$.

The remaining idea of the algorithm is how to choose an appropriate probability p. If p is too large, the space (of the bin) will be too large. If p is too small, $\frac{|\mathcal{X}|}{p}$ will not be a good estimator.

The remaining idea is to choose p adaptively.

The CVM algorithm

Algorithm 1 F_0-Estimator

Input Stream $\mathcal{A} = \langle a_1, a_2, \ldots, a_m \rangle$, ε, δ

1: **Initialize** $p \leftarrow 1$; $\mathcal{X} \leftarrow \emptyset$; thresh $\leftarrow \lceil \frac{12}{\varepsilon^2} \log(\frac{8m}{\delta}) \rceil$
2: **for** $i = 1$ to m **do**
3: $\quad \mathcal{X} \leftarrow \mathcal{X} \setminus \{a_i\}$
4: $\quad$ With probability p, $\mathcal{X} \leftarrow \mathcal{X} \cup \{a_i\}$
5: $\quad$ **if** $|\mathcal{X}| =$ thresh **then**
6: $\quad\quad$ Throw away each element of $\mathcal{X}$ with probability $\frac{1}{2}$
7: $\quad\quad$ $p \leftarrow \frac{p}{2}$
8: $\quad\quad$ **if** $|\mathcal{X}| =$ thresh **then Output** $\perp$
9: **Output** $\frac{|\mathcal{X}|}{p}$

Figure: The CVM algorithm

Sketch of the CVM analysis

Line 8 can only occur with very low probability and in practice it is not needed. Theoretically it can be removed by replacing the IF statement by a While statement leaving a very small probability that the run time performance of the algorithm would suffer.

The threshold *thresh* prevents the space from getting too big.

The adaptive setting of the probability p, keeps the probability from getting too small.

This is the technical part of the analysis and leads to the estaimate (ϵ, δ) estiamte.

As is often the case, even simple algorithms can require some non trivial analysis. However, the analysis is much simpler than in the previous results. The entire analysis is 3 pages.

Sublinear time

We continue to consider contexts in which randomization is provably necessary. In particular, we will now study sublinear time algorithms which, even more than streaming algorithms, almost always require randomization.

- An algorithm is sublinear time if its running time is $o(n)$, where n is the length of the input. As such an algorithm must provide an answer without reading the entire input.
- Thus to achieve non-trivial tasks, we almost always have to use randomness in sublinear time algorithms to sample parts of the inputs.
- The subject of sublinear time algorithms is a big topic and we will only present a very small selection of hopefully representative results.
- The general flavour of results will be a tradeoff between the accuracy of the solution and the time bound.
- Like streaming algorithms, sublinear time often concerns estimates of various properties rather than producing solutions since solutions (of search and optimization problems) almost always require at least linear space.

A deterministic exception: estimating the diameter in a finite metric space

- We first conisder an exception of a "sublinear time" algorithm that does not use randomization. (Comment: "sublinear in a weak sense".)
- Suppose we are given a finite metric space M (with say n points x_i) where the input is given as n^2 distance values $d(x_i, x_j)$. The problem is to compute the diameter D of the metric space, that is, the maximum distance between any two points.
- For this maximum diameter problem, there is a simple $O(n)$ time (and hence sublinear) algorithm; namely, choose an arbitrary point $x \in M$ and compute $D = \max_j d(x, x_j)$. By the triangle inequality, D is a 2-approximation of the diameter.
- I say sublinear time in a weak sense because in an explicitly presented space (such as d dimensional Euclidean space), the points could be explicitly given as inputs and then the input size is n and not n^2.

Sampling the inputs: some examples

- The goal in this area is to minimize execution time while still being able to produce a reasonable answer with sufficiently high probability.
- We will consider the following examples:
  1.
  2. Searching in an (anchored) sorted linked list [Chazelle,Liu,Magen 2003]
  3. Estimating the average degree in a graph [Feige 2006]
  4. Estimating the size of some maximal (and maximum) matching [Nguyen and Onak 2008] in bounded degree graphs.
  5. Examples of property testing, a major topic within the area of sublinear time algorithms. See Dana Ron's DBLP for many results and surveys.

Searching in an (anchored) sorted linked list

- Suppose we have an array $A[i]$ for $1 \leq i \leq n$ where each $A[i]$ is a pair (x_i, p_i) with $x_1 = \min\{x_i\}$ and p_i being a pointer to the next smallest value in the linked list.
- That is, $x_{p_i} = \min\{x_j | x_j > x_i\}$. (For simplicity we are assuming all x_j are distinct.)
- We would like to determine if a given value x occurs in the linked list and if so, output the index j such that $x = x_j$.

A $\sqrt{n}$ algorithm for searching in anchored sorted linked list

Let $R = \{j_i\}$ be $\sqrt{n}$ randomly chosen indices plus the index 1.
Access these $\{A[j_i]\}$ to determine k such that x_k is the largest of the accessed array elements less than or equal to x.
Search forward $2\sqrt{n}$ steps in the linked list to see if and where x exists

Claim:

This is a one-sided error algorithm that (when $x \in \{A[i]\}$) will fail to return j such that $x = A[j]$ with probability at most $1/2$.

Some further comments on searching in a sorted linked list

If the list is doubly linked, we can find the smallest item in $O(\sqrt{n})$ time and do not need the assumption that the list is anchored.

Using the Yao Principle, the $O(\sqrt{n})$ time bound is optimal.

The algorithm can be generalized to provide a $O(\sqrt{n})$ time algorithm to determine if two n vertex polygons intersect (and if so, return a point of intersection.

Estimating average degree in a graph

- Given a large graph $G = (V, E)$ (that can only be accessed by some kind "local queries") with $|V| = n$, we want to estimate the average degree d of the vertices.
- We want to construct an algorithm that approximates the average degree within a factor less than $(2 + \epsilon)$ with probability at least $3/4$ in time $O(\frac{\sqrt{n}}{poly(\epsilon)})$. We will assume that we can access the degree d_i of any vertex v_i in one step.
- Like a number of results in this area, the algorithm is simple but the analysis requires some care.

The Feige algorithm

Sample $8/\epsilon$ random subsets S_i of V each of size (say) $\frac{\sqrt{n}}{\epsilon^{2.5}}$
Compute the average degree a_i of nodes in each S_i.
The output is the minimum of these $\{a_i\}$.

What is the difference between the average degree problem and estimating the average of n numbers?

To estimate the average of a set of numbers $\{x_i\}$ (with $1 \leq x_i \leq n-1$) requires $\Omega(n)$ time. This is a needle in a haystack problem problem if all but one of the $x_i = 1$ and the other one is (say) n^2.

The average degree in a graph seems pretty much like the average of a set of numbers. What is different?

What is the difference between the average degree problem and estimating the average of n numbers?

To estimate the average of a set of numbers $\{x_i\}$ (with $1 \leq x_i \leq n-1$) requires $\Omega(n)$ time. This is a needle in a haystack problem problem if all but one of the $x_i = 1$ and the other one is (say) n^2.

The average degree in a graph seems pretty much like the average of a set of numbers. What is different?

For a connencted graph, the difference is intuitively that while we may not sample a high degree vertex, we are likely to find their neighbours and this will wind up accounting for the high degree edges.

A more precise argument would use the following theorem:

Erdos-Gallai

The sequence $d_1 \geq d_2 \ldots \geq d_n \geq 1$ is a graph degree sequence if and only if $\sum_{i=1}^{n} d_i$ is even and $\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} d_i$.

The precise statement of Feige's approximation

We will just sketch a slightly weaker result but here is precisely the statement in the Feige 2006 SICOMP paper.
For any d_0 (the minimum degree in the graph) and for $\rho = 2 + \epsilon$, the Feige algorithm computes an estimation within a factor of ρ with high probability (e.g., any constant) and uses $O(\frac{1}{\epsilon}\sqrt{n/d_0})$ degree queries.

The analysis of the approximation

Since we are sampling subsets to estimate the average degree, we might have estimates that are too low or too high. But we will show that with high probability these estimates will not be too bad.

The proof will follow from the following two lemmas concerning the average degree a_i of a subset S_i.

1. Lemma 1: $Prob[a_i < \frac{1}{2}(1-\epsilon)\bar{d}] \leq \frac{\epsilon}{64}$
2. Lemma 2: $Prob[a_i > (1+\epsilon)\bar{d}] \leq 1 - \frac{\epsilon}{2}$

How the proof follows from the lemmas

The probability bound in Lemma 2 is amplified to any constant (say $1/8$) by repeated trials; i.e., the $8/\epsilon$ independent trials of random S_i. That is,

$$Prob[Alg > (1+\epsilon)\bar{d}] = Prob[a_i > (1+\epsilon)\bar{d}] \quad \forall i$$
$$< \left(1 - \tfrac{\epsilon}{2}\right)^{8/\epsilon} \leq e^{-4} \leq 1/8.$$

For Lemma 1, we fall outside the desired bound if any of the repeated trials gives a very small estimate of the average degree but by the union bound this is no worse than the sum of the probabilities for each trial. That is, $Prob[ALG < \frac{1}{2}(1-\epsilon)\bar{d}] \leq \sum_{i=1}^{\epsilon/8}(\epsilon/64) = 1/8$.

Sketch of the lemmas in Feige's average degree algorithm

Lemma 2 is relatively easy. Consider any S_i. Letting X_j be the degree of vertex v_j we have $\mathbb{E}[X_j] = \bar{d}$, and
$\mathbb{E}[a_i] = \mathbb{E}\left[\frac{1}{|S_i|}\left(\sum_{j:j\in S_i} X_j\right)\right] = \frac{1}{|S_i|}\sum_{j:j\in S_i}\mathbb{E}[x_j] = \bar{d}$

We can then use Markov's inequality to obtain Lemma 2.

The proof of Lemma 1 is more involved. We cannot simply amplify an error bound for a random subset since each S_i trial gives another chance of finding a low estimate. Instead we will have to use a Chernoff bound for the probability that a sum of independent random variables deviates from its mean. The following is sufficient:

A Chernoff bound

Let $Z_1, \ldots, Z_s$ be a sequence of iindependent r.v.s with $Z_j \in [0, 1]$ and let $\mu = \mathbb{E}[\sum_j Z_j]$. Then $Prob[\sum_j Z_j \leq (1-\epsilon)s\mu] \leq e^{-\epsilon^2 s\frac{\mu}{4}}$

Continuing the proof sketch for Lemma 1

Let H be the $\sqrt{\epsilon' n}$ vertces of the the highest degree. Here ϵ' will be chosen to satisfy $(1-\epsilon') \cdot (1/2 - \epsilon') \leq (1/2 - \epsilon)$

Assune that the random selection of nodes in the algorithm was restricted to just $L = V \setminus H$.

Of course, by removing high degree vertices from the random sampling, the probability of concluding that $\bar{d} \leq \frac{1}{2}(1-\epsilon)d$ increases.

Claim $\sum_{i \in L} d_j \geq (\frac{1}{2} - \epsilon') \sum_{i \in V} d_i - \epsilon' n$
The Claim is a couting algorithm noting that the $\sum_{i \in V}$ counts every edge twice while $\sum_{i \in L}$ might omit $\epsilon' n$ edges within H and only count edges between H and L once.

Continuation of proof of Lemma 1

Thus the average degree in L is at least $\frac{1}{2}(\bar{d} - \epsilon')$. So it remain to obtain a lower bound on the average degree of vertices in L. We use the following observations:

- Let d_H = minimum degree of any vertex in H
- Let X_j = degree of a vertex $v_j \in S_i$ which implies $X_j \in [1, d_H]$
- Let $Z_j = X_j/d_H$
- The Chernoff bound and the bound we have for the average degree of vertices in L, then gives us : $Prob[a_i < (1/2)(1-\epsilon)\bar{d}] \leq e^{-\frac{\epsilon^2 s\mathbb{E}[X_j]}{4d_H}}$ using the Chernoff bound.

If $s = |S_i|$ were sufficiently large (i.e. $s \geq \epsilon^2 \frac{d_H}{\mathbb{E}[X_j]}$), we are done. But we would like the bound to not depend on d_H and $\mathbb{E}[X_j]$.

This is handled by considering two cases; namely for when $d_H < |H|$ and when $d_H \geq |H|$.

Understanding the input query model

- As we initially noted, sublinear time algorithms almost invariably sample (i.e. query) the input in some way. The nature of these queries will clearly influence what kinds of results can be obtained.
- Feige's algorithm for estimating the average degree uses only "degree queries"; that is, "what is the degree of a vertex v".
- Feige shows that in this degree query model, any algorithm that acheives a $(2-\epsilon)$ approximation (for any $\epsilon > 0$) requires time $\Omega(n)$.
- In contrast, Goldreich and Ron [2008] consider the same average degree problem in the "neighbour query" model; that is, upon a query (v, j), the query oracle returns the j^{th} neighbour of v or a special symbol indicating that v has degree less than j. A degree query can be simulated by $\log n$ neighbour queries.
- Goldreich and Ron show that in the neighbour query model, that the average degree $\bar{d}$ can be $(1+\epsilon)$ approximated (with one sided error probability 2/3) in time $O(\sqrt{(}n/\bar{d})poly(\log n, \frac{1}{\epsilon})$
- They also show that this $\sqrt{(}n)$ time bound is essentially optimal.

Approximating the size of a maximum matching in a bounded degree graph

- We recall that the size of any *maximal* matching is within a factor of 2 of the size of a maximum matching.
- Our goal is to compute with high probability a *maximal* matching in time depending only on the maximium degree D.

Nguyen and Onak Algorithm

Choose a random permutation p of the edges $\{e_j\}$
% Note: this will be done "on the fly" as needed
The permutation determines a maximal matching M as given by the greedy algorithm that adds an edge whenever possible.
Choose $r = O(D/\epsilon^2)$ nodes $\{v_i\}$ at random
Using an "oracle" let X_i be the indicator random variable for whether or not vertex v_i is in the maximal matching.
Output $\tilde{m} = \sum_{i=1\ldots r} X_i$

Performance and time for maximal matching

Claims

1. $m \leq \tilde{m} \leq m + \epsilon$ n where $m = |M|$.
2. The algorithm runs in time $2^{O(D)}/\epsilon^2$

- This immediately gives an approximation of the *maximum* matching m^* such that $m^* \leq \tilde{m} \leq 2m^* + \epsilon n$
- A more involved algorithm by Nguyen and Onak yields the following result:

Nguyen and Onak maximum matching result

Let $\delta, \epsilon > 0$ and let $k = \lceil 1/\delta \rceil$. There is a randomized one sided algorithm (with probability 2/3) running in time $\frac{2^{O(D^k)}}{\epsilon^{2^{k+1}}}$ that outputs a maximium matching estimate $\tilde{m}$ such that $m^* \leq \tilde{m} \leq (1+\delta)m^* + \epsilon n$.

Property Testing

- Perhaps the most prevalent and useful aspect of sublinear time algorithms is for the concept of property testing. This is its own area of research with many results.
- Here is the concept: Given an object G (e.g. a function, a graph), test whether or not G has some property P (e.g. G is bipartite).
- The tester determines with sufficiently high probability (say $2/3$) if G has the property or is "ϵ-far" from having the property. The tester can answer either way if G does not have the property but is "ϵ-close" to having the property.
- We will usually have a 1-sided error in that we will always answer YES if G has the property.
- We will see what it means to be "ϵ-far" (or close) from a property by some examples.

Tester for linearity of a function

- Let $f : Z_n -> Z_n$; f is linear if $\forall x, y \ f(x+y) = f(x) + f(y)$.
- Note: this will really be a test for group homomorphism
- f is said to be ϵ-close to linear if its values can be changed in at most a fraction ϵ of the function domain arguments (i.e. at most ϵn elements of Z_n) so as to make it a linear function. Otherwise f is said to be ϵ-far from linear.

The tester

Repeat $4/\epsilon$ times
Choose $x, y \in Z_n$ at random
  If $f(x) + f(y) \neq f(x+y)$
  then Output f is not linear
End Repeat If all these $4/\epsilon$ tests succeed then Output linear

- Clearly if f is linear, the tester says linear.
- If f is ϵ-far from being linear then the probability of detecting this is at least $2/3$.

Testing a matrix multiplication algorithm

Suppose we are given a new fast matrix multiplication algorithm which given three $n \times n$ matrices A, B, C (over some ring or field R) is supposed to output $C = A \cdot B$. Recall there is an algorithm that runs in time $O(n^{2.34})$ but maybe you don't trust it.

A naive way to check by using a known trusted algorithm (perhaps the standard n^3 algorithm) and then verify. But then why use the new algorithm?

Instead we can choose a random small set $S \subset R$ and randomly choose a vertor $\mathbf{v} \in R^n$ and check if $A(B(\mathbf{v})) = \mathbf{C}(\mathbf{v})$. This will take $O(n^2)$ operations and the probability can be made arbitrarily small by choosing a large S or by repeating the test.

This is not a sublinear time algorithm but in practice might be very useful in contrast to trying to prove correctness of the algorithm. The problem of computing a matrix vector product provably requires n^2 operations but not sure if this rules out a tester running in time $o(n^2)$.

Testing if an array is sorted

- Given an array $A[i] = x_i, i = 1 \ldots n$ of distinct elements, determine if A is sorted (i.e. $i < j \Rightarrow A[i] < A[j]$) or is ϵ-far from being monotone in the sense that more than $\epsilon * n$ list values need to be changed in order for A to be monotone.
- The algorithm randomly chooses $2/\epsilon$ random indices i and performs binary search on x_i to determine if x_i in the list. The algorithm reports that the list is monotone if and only if all binary searches succeed.
- Clearly the time bound is $O(\log n/\epsilon)$ and clearly if A is monotone then the tester reports monotone.
- If A is ϵ-far from monotone, then the probability that a random binary search will succeed is at most $(1 - \epsilon)$ and hence the probability of the algorithm failing to detect non-monotonicity is at most $(1 - \epsilon)^{\frac{2}{\epsilon}} \leq \frac{1}{e^2}$

Graph Property testing

- Graph property testing is an area by itself. There are several models for testing graph properties.
- Let $G = (V, E)$ with $n = |V|$ and $m = |E|$.
- Dense model: Graphs represented by adjacency matrix. Say that graph is ϵ-far from having a property P if more than ϵn^2 matrix entries have to be changed so that graph has property P.
- Sparse model, bounded degree model: Graphs represented by vertex adjacency lists. Graph is ϵ-far from property P is at least ϵm edges have to be changed.
- In general there are substantially different results for these two graph models.

The property of being bipartite

- In the dense model, there is a constant time one-sided error tester. The tester is (once again) conceptually what one might expect but the analysis is not at all immediate.

Goldreich, Goldwasser,Ron bipartite tester

Pick a random subset S of vertices of size $r = \Theta(\frac{\log(\frac{1}{\epsilon})}{\epsilon^2})$
Output bipartite iff the induced subgraph is bipartite

- Clearly if G is bipartite then the algorithm will always say that it is bipartite.
- The claim is that if G is ϵ-far from being bipartite then the algorithm will say that it is not bipartite with probability at least $2/3$.
- The algorithm runs in time quadratic in the size of the induced subgraph (i.e. the time needed to create the induced subgraph).

Testing bipartiteness in the bounded degree model

- Even for degree 3 graphs, $\Omega(\sqrt{n})$ queries are required to test for being bipartite or ϵ-far from being being bipartite. Goldreich and Ron [1997]
- There is a nearly matching algorithm that uses $O(\sqrt{n} \cdot poly(\log n/\epsilon))$ queries. The algorithm is based on random walks in a graph and utilizes the fact that a graph is bipartite iff it has no odd length cycles.

Goldreich and Ron [1999] bounded degree algorithm

Repeat $O(1/\epsilon)$ times
  Randomly select a vertex $s \in V$
  If algorithm *OddCycle*(s) returns cylce found then REJECT
End Repeat
If case the algorithm did not already reject, then ACCEPT

- *OddCycle* performs $\sqrt{n} \cdot poly(\log n/\epsilon)$ random walks from s each of length $poly(\log n/\epsilon)$. If some vertex v is reached by both an even length and an odd length prefix of a walk then report cycle found; else report odd cycle not found