CSC2422 Fall 2022: Algorithm Design, Analysis and Theory
An introductory (i.e. foundational) level graduate course.

Allan Borodin

November 30, 2022; Lecture 10

Brief Announcements and todays agenda

- Assignment 3 due Wednesday, December 7 1PM. So far I have posted three questions.
- In last weeks slides, I added a slide regarding a variant of the "7-way-branching" algorithm for 3SAT. Namely I have skectched a "3-way-brancing" algorithm with a better exponential time bound.
- I also added a slide giving the analysis for the claim about the random walk algorithm for 3SAT.

Todays agenda

- Continue sublinear time algorithms
- The streaming model

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 connected 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
$\mathbb{E}[a_i] = \mathbb{E}\Big[\frac{1}{|S_i|}\big(\sum_{j:j\in S_i} X_j\big)\Big] = \frac{1}{|S_i|}\sum_{j:j\in S_i}\mathbb{E}[x_j] = \bar{d}$

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, that 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 $\tilde{m} = \sum_{i=1 \dots r} X_i$
```

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 list for monotonicity

- Given a list $A[i] = x_i, i = 1 \ldots n$ of distinct elements, determine if A is a monotone list (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 also 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 $O(\sqrt{n} poly(\log n/\epsilon))$ 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 $poly(\log n/\epsilon)$ 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

Sublinear space: A slight detour into complexity theory

- Sublinear space has been an important topic in complexity theory since the start of complexity theory. While not as important as the $P = NP$ or $NP = co - NP$ question, there are two fundamental space questions that remain unresolved:
  1. Is $NSPACE(S) = DSPACE(S)$ for $S \geq \log n$?
  2. Is P contained in $DSPACE(\log n)$ or $\cup_k SPACE(\log^k n)$? Equivalently, is P contained in polylogarthmic parallel time.
- Savitch [1969] showed a non deterministic S space bounded TM can be simulated by a deterministic S^2 space bounded machine (for space constructible bounds S).
- Further in what was (and perhaps still is) considered a very surprising result, Immerman [1987] and independently Szelepcsényi [1987] $NSPACE(S) = co - NSPACE(S)$. (Savitch's result was also considered suprising by some researchers when it was announced.)

USTCON vs STCON

We let $USTCON$ (resp. $STCON$) denote the problem of deciding if there is a path from some specified source node s to some specified target node t in an unidrected (resp. directed) graph G.

- As I may have previously mentioned, the Aleliunas' et al [1979] random walk result showed that $USTCON$ is in $RSPACE(\log n)$ and after a sequence of partial results about $USTCON$, Reingold [2008] was eventually able to show that $USTCON$ is in $DSPACE(\log n)$
- It remains open if
  1. $STCON$ (and hence $NSPACE(\log n)$) is in $RSPACE(\log n)$ or even $DSPACE(\log n)$.
  2. $STCON \in RSPACE(S)$ or even $DSAPCE(S)$ for any $S = o(\log^2 n)$
  3. $RSPACE(S) = DSPACE(S)$.

The streaming model

- In the data stream model, the input is a sequence A of input items (or input elements) $a_1, \ldots, a_n$ which is assumed to be too large to store in memory. Let $a_i \in [1, D]$
  Small notational dilemma: The seminal paper in the streaming area is the Alon, Matias and Szegedy (AMS) paper for computing the frequency moment problem. Their notation is that a stream is a sequence $a_1, \ldots a_m\}$ with $a_i \in \{1, 2, \ldots n\}$. To be more consistent with the online literature, I will use n for the length of the sequence and D for the domain of the a_i with the exception of the discussion of the AMS paper where I will use their notation.
- We usually assume that n is not known and one can think of this model as a type of online or dynamic algorithm.
- The space available $S(n, D)$ is some sublinear function. The input items stream by and one can only store information in space S.

The streaming model continued

- In some papers, space is measured in bits (which is what we will do) and sometimes in words, each word being $O(\log n)$ bits.
- It is also desirable that that each input item is processed efficiently, say $\log(n) + \log(m)$ time, and perhaps even in time O(1) (assuming we are counting operations on words as $O(1)$).
- The initial (and primary) work in streaming algorithms is to approximately compute some function (say a statistic) of the data or identify some particular item(s) in the data stream.
- Lately, the model has been extended to consider "semi-streaming" algorithms for optimization problems. For example, for a graph problem such as matching for a graph $G = (V, E)$, the goal is to obtain a good approximation using space $\tilde{O}(|V|)$ rather than $O(|E|)$.
- Most results concern the space required for a one pass algorithm. But there are results concerning multi-pass algorithms and also results concerning the tradeoff between the space and number of passes.

An example of a deterministic streaming algorithms

As in sublinear time, it will turn out that almost all of the results in this area are for randomized algorithms. Here is one exception.

The missing item problem

Suppose we are given a stream $A = a_1, \ldots, a_{n-1}$ and we are promised that the stream A is a permutation of $\{1, \ldots, m\} - \{x\}$ for some integer x in $[1, n]$. (Here $D = n$.) The goal is to compute the missing x.

- Space n is obvious using a bit vector $c_j = 1$ iff j has occured.
- Instead we know that $\sum_{j \in A} = n(n+1)/2 - x$.
  So if $s = \sum_{i \in A} a_i$, then $x = n(n+1)/2 - s$.
  This uses only $2 \log m$ space and constant time/item.

Generalizing to k missing elements

Now suppose we are promised a stream A of length $n - k$ whose input elements consist of a permutation of $n - k$ distinct elements in $\{1, \ldots, n\}$. We want to find the missing k elements.

- Generalizing the one missing element solution, to the case that there are k missing elements we can (for example) maintain the sum of j^{th} powers $(1 \leq j \leq k)$ $s_j = \sum_{i \in A} (a_i)^j = c_j(n) - \sum_{i \notin A} x_i^j$. Here $c_j(m)$ is the closed form expression for $\sum_{i=1}^{n} i^j$. This results in k equations in k unknowns using space $k^2 \log n$ but without an efficient way to compute the solution.
- As far as I know there may not be an efficient small space *deterministic* streaming algorithm for this problem.
- Using randomization, much more efficient methods are known; namely, there is a streaming alg with space and time/item $O(k \log k \log n)$; it can be shown that $\Omega(k \log(n/k))$ space is necessary.

Some well-studied streaming problems

- Computing frequency moments. Let $A = a_1 \dots a_m$ be a data stream with $a_i \in [n] = \{1, 2, \dots n\}$. Let m_i denote the number of occurences of the value i in the stream A. For $k \geq 0$, the k^{th} frequency moment is $F_k = \sum_{i \in [n]} (m_i)^k$. The frequency moments are most often studied for integral k.
    1. $F_1 = m$, the length of the sequence which can be simply computed.
    2. F_0 is the number of distinct elements in the stream
    3. F_2 is a special case of interest called the repeat index (also known as Gini's homogeneity index).
- Finding k-heavy hitters; i.e. those elements appearing more than m/k times in stream A.
- Finding rare or unique elements in A.

The majority element problem

While most streaming algorithms concern one pass over the input stream, there are results that use two or more passes.

One relatively easy (but still very interesting) result is the Misra-Gries algorithm for computing k heavy hitters. As a special case, we have the majority problem (i.e. the k-hitter problem for $k = 2$).

There is a temptation to solve this problem by divide and conquer; divide the sequence in half, find the heavy hitters in each half and then check.

The streaming model fascilitates thinking about a much better solution. In the case of majority, lets just try to maintain one possible candidate in the first pass and then check to see if the candidate is a true more than majority item in the second pass. See the Chakrabarti Lecture notes.

The Misra-Gries algorithm

Maintain a candidate for the majority element and a counter for that candidate.

When the counter is empty, the next element in the stream becomes the candidate.

Every time the next elmennt in the stream is the candidate increase the counter by 1. If the next element is not the candidate decrease the counter by 1.

Claim: If there is a majority element then it has to be the current candidate.

We can use a second pass over the elements to check if the candidate occurs more than $n/2$ times.

The space used is $O(\log n + \log D)$ and the time is $(\log n)$ (or $O(1)$ if counting element comparisons) per input element.