CSC2420: Lecture 10

- Todays agenda :
- Sublinear time algorithms continued
  - The average degree in a graph
  - Maximal matching in a graph
  - Property testing
    - Monotonicity of a list of elements
    - Monotonicity of a function
    - Linearity of a function
    - Graph properties
- Begin streaming model (following a very brief comment on traditional complexity issues regarding small space.)

Estimating average degree in a graph (example of use of Chernoff bounds)

- Given a graph G = (*V, E*) and |*V* | = *n*, we want to estimate the average degree *d* of all vertices of G. We want to construct an algorithm that approximates the average degree within a factor less than (2 + eps) with probability at least 3/4 in time O(n/poly(eps)). 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 a little (or much) care.
- The algorithm samples 8/eps random subsets *S_i* of *V* each of size (say) sqrt *n*/eps^3 computing the average degree *a_i* of nodes in each *S_i*. The output is the minimum of these {*a_i*}.

Analysis of approximation

- Since we are sampling subsets to estimate the average degree, we should expect that the estimates can be too high or too low but we will show that with high probability these estimates will not be too bad. More precisely, we need:
- Lemma 1: $Prob[a_i < \frac{1}{2}(1-\epsilon)d] \leq \frac{\epsilon}{64}$
- Lemma 2: $Prob[a_i > (1+\epsilon)d] \leq 1 - \frac{\epsilon}{2}$

The probability bound in Lemma 2 is amplified as usual by repeated trials. 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.

Proof of Lemma 2

- We only need a weak error bound here (and then can amplify) so we only need the Markov inequality.
- Let s = sqrt *n*/eps^3 (the size of each sample set of vertices) and let the r.v. *X_j* denote the degree of the *j* th vertex sampled.
- Then the r.v. *a_i* = (*1/s*) [*X_1 + …+ X_s*] and has expectation d (the average degree of the graph).
- By Markov's inequality, when r.v. Z >= 0, Prob[*Z >= c* * **E**[*Z*]] <= *1/c* so setting c = $1 + \epsilon$ and noting that $1 + \epsilon < 1 - \frac{\epsilon}{2}$ we get Lemma 2.

Proof of Lemma 1

- We cannot amplify an error bound here since each random subset S of vertices gives another chance of getting a low estimate (since we are taking the minimum over all trials). Instead we will have to use a Chernoff bound for estimating the deviation of a sum of independent r.v.s from its mean.
- Namely, the form of the Chenoff bound we will use is as follows: Let {Z_i: I = 1, …,s} be independent random variables in [0,1]. Then

$$Prob[\sum_{i=1}^{s} Z_i \leq (1-\epsilon)E[\sum Z_i]] \leq e^{-\epsilon^2 \frac{E[\sum Z_i]}{4}}$$

Lemma 2 continued

- Let H be the $\sqrt{\epsilon n}$ vertices of highest degree and let L be V-H. We can assume that we are sampling from L since this will be an underestimate on the distrbution determinng *a_i*.
- Counting only edges with at least one vertex in L, **E**[*a_i*] >= (1/2) (d |*V*| - |*H*|^2)/|*L*|>= (1/2)(1-ϵ)d
- Letting *d_H* be the min degree within *H*, and setting *Z_j = X_j/d_H*, the Chernoff bound shows that

$$Prob[a_i < (1/2)(1-\epsilon)d] \leq e^{-\frac{\epsilon^2 s E[X_j]}{4 d_H}}$$

Lemma 2 continued

- If s is sufficiently large ($s \geq \epsilon^{-2} \frac{d_H}{E[X_j]}$), then we get the desired bound. But we want a bound on s that does not depend on d_H and E[X_j]. This needs two cases depending on whether or not $d_h \geq \frac{1}{\epsilon}|H|$
- Case 1: $d_h \geq \frac{1}{\epsilon}|H|$ In this case we have:

$$E[X_j] = \frac{\sum_{v \in L} d(v)}{|L|} \geq \frac{|H|d_H - |H|^2}{|L|} \geq \frac{|H|(1-\epsilon)d_H}{|V|}$$

- Case 2: $d_H < \frac{1}{\epsilon}|H|$ Here we just need that E[X_*j*] is at least 1.

Approximate size of a maximal matching in a bounded degree graph.

- This may also seem like a contrived problem as there is a simple greedy algorithm to compute a maximal matching but this is an example of a more dramatic time bound and contains a nice proof. And the size of a maximal matching is within a factor of two of a maximum matching. This is also the starting point for an extension of this result which computes the size of a maximum matching.

The algorithm

- Choose a random permutation p of the edges {*e_j*} by choosing a random value in [0,1] for each *e_j* . (Note: this will be done "on the fly" as needed) The permutation determines a maximal matching as given by the greedy algorithm that adds an edge whenever possible.
- Choose *r* = O(*d/eps^2*) nodes {*v_i*} at random
- Using an "oracle" let *X_i* be the indicator r.v for whether or not vertex *v_i* is in the maximal matching.
- Output *sum_i = 1 …r} X_i*
- Using a Chernoff bound, it follows that with sufficiently high probability the output is within (*eps/2*) *n* of the size of the maximal matching. We note that for a connected graph of max degree d, every maximal matching has at least *n*/(2d) edges. Hence the algorithm is giving a PTAS type approximation.

Sketch of analysis

- The interesting aspect of the analysis is the analysis of the time bound. The surprising fact is that the oracle is only 2^{O(d)} time. The oracle will be a recursive oracle determining if an adjacent edge is in the match.
- Consider an edge e adjacent to *v_i*. This edge will be in the maximal matching as long as no previous (wrt to p) edge *e'* is in the matching. If there is a recursive call of depth *k* from *e,* then the recursive path has decreasing values for the "priority" of the edges. This can only happen with probability 1/(*k*!). There are at most *d*^k nodes at distance *k* so that the expected number of nodes visited is at most *d^k*/(*k*!). The *sum_k d^k*/(*k*!) = e^d. Thus to compute *X*_i will take at most d (e^d) = 2^{O(d)}.

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. graph), determine 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 "*eps*-far" from having the property. The tester can answer either way if G does not have the property but is "*eps*-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 "*eps*-far" from a property by some examples.

Testing a list for monotonicity

- Given a list A[i] = *x_i, i = 1 …n* of distinct elements, determine if A is a monotone list (*i<j* implies A[*i*]< A[*j*]) or is eps-far from being monotone in the sense that more than *eps * n* list values need to be changed in order for A[] to be monotone.
- The algorithms in this area are often very simple but sometimes the analysis can be quite involved. The algorithm and analysis here are both easy.
- The algorithm randomly chooses *2/eps* 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.

The analysis for list monotonicity

- Clearly the time bound is O(*log n/eps*) and clearly if A[] is monotone then the tester reports monotone.
- Let S = {*i:* tester succeeds on *x_i*}. We claim that the subsequence {A[*i*]: *i* in S} is monotone. This can be seen by observing that if *i<j* and both searches succeed then at some point in the binary search, there is an *x_k* such that *x_i <= x_k* and *x_k <= x_j*.
- Now if A[] is eps-far from monotone, then the probability that a binary search succeeds is */S|/n*.
- We claim that |S| <= (*1-eps*)*n* for otherwise we only have to change eps*n values to make A[] monotone.
- Hence the probability of saying YES when A[] is eps-far from monotone is at most (*1-eps*)^{*2/eps*} <= e^{-2}

Tester for Boolean function monotonicity

- Let f:{0,1}^*n* -> {0,1}; we say f is monotone if f(*x*)<=f(*y*) for all *x* < *y* where *x*<*y* means *x_i* <=*y_i* for all *i* and *x_j* < *y_j* for at least one coordinate *j*. (The result can be extend to other ordered domains.) We say that f is *eps*-close to a montone function if changing at most an *eps* fraction (i.e. *eps * 2^n*) of the values makes the function monotone.
- The tester for this problem relies on the following fact: f is monotone iff f is monotone on every pair of inputs that differ only on a single coordinate. One direction is immediate and the other follows by “interpolation” (in the sense used in say cryptography).

The algorithm for Boolean function monotonicity

- Repeat O(*n/eps*) times
  - Randomly select i in {*1,…,n*} and *z* in {0,1}^*n*
  - Let *x* (resp. *y*) be *z* with *i* th bit set to 0 (resp. 1)
  - If f(*x*) > f(*y*) then reject
- If all tests succeed then report monotone.

- This is clearly a one-sided error algorithm as it always reports monotone when the function is monotone.

Glimpse of analysis

- Let U be the set of *n* 2^{*n-1*} pairs (*x,y*) in {0,1}^*n* such that x and y differ in just one coordinate with *x_i = 0* and *y_i =1*, and let V(f) be the subset of U such that f(*x*)>f(*y*). This is the set of pairs that can catch non-monotonicity. If we let d-mon(f) be the minimum distance of f to some monotone function, then the main theorem to be proved is that the probability of detecting non-montonicity in one trial p-alg(f) = |V(f)|/|U| >= d-mon(f)/*n*.
- This is a relatively difficult result to prove. (It also turns out that p-alg(f) <= 2 d-mon(f)/*n* with examples where both extremes can be nearly met.)

Glimpse continued

- The idea is to show how to gradually turn f into a monotone function by a series of “switch operators” S_i(h) for a Boolean function h.
- Namely, for every coordinate *i*, S_*i*(h) reverses the value of h on any violating pair of inputs differing only on the *i* th coordinate. That leads to another distance function D_*i*(f) = {x| f(x) not= S_*i*(f)}. It follows that sum_*i* D_*i*(f) = 2 V(f).
- The main and fairly technical lemma is that for all i,j: D_*i* (S_*j*(h)) <= D_*i*(h) and in particular if h is monotone in a set of dimensions T, then S_*j*(h) is monotone is dimensions T+{*j*}. Then it follows that f can be turned into monotone g = S_*n*(S_{*n-1*}(….S_1(f)….) within the desired 2 p-alg(f) *n 2^n* function changes.

Tester for function linearity

- Let *f:Z_n -> Z_n*; f is linear if f(*x+y*) = f(*x*) + f(*y*).
- Note: this is really a test for group homomorphism
- Similar to the previous result, define an alternative distance (from linearity) function p-alg(f) as Prob_{*x,y*} [f(*x+y*) not equal to f(*x*)+f(*y*)]
- The tester: Repeat 4*/eps* times

  Choose *x,*y in *Z_n* at random and check
  if f(*x*) + f(*y*) = f(*x + y*).
  Say linear if all hold with equality; o.w. not linear

Clearly if f is linear, the tester says linear. And if f is *eps* away from being linear then the probability of detecting this in one trial is at least 2/3.

Graph Property Testing

- Graph property testing is almost 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 *eps*-far from having a property P if more than *eps n^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 eps-far from property P is at least *eps m* edges have to be changed.

Substantially different results for the different models

- For the property of being *k*-colourable, in the dense model, for every fixed *k*, there is a constant time tester. The tester is (once again) conceptually what you might expect and the analysis is not at all immediate.
- Pick a random subset S of vertices of size O(*k log k/eps^2*) and query all the induced edges; Say *k*-colourable if induced graph is k-colorable.
- The case of *k=2* is a tester for being bipartite.

Bipartite tester in bounded degree model

- Even for degree 3 graphs, the number of queries required is Omega(sqrt{*n*}) to test for being bipartite (or eps-far from being being bipartite).
- There is a nearly matching algorithm that uses O(sqrt{*n*} poly(log *n/eps*)) queries. The algorithm is based on random walks in a graph and is substantially different than most of the sublinear algorithms previously seen.

Biparitite bounded degree tester

- Repeat O(*1/eps*) time
- 1. Uniformly select *s* in V .
  2. If odd-cycle(*s*) returns found then reject.

In case the algorithm did not reject in any one of the above iterations, it accepts.

- odd-cycle(*s*)
- Let K = poly((*log n*)/ε) ·sqrt{*n*}, and L = poly((*log n*)/ε);
- Perform K random walks starting from *s*, each of length L;
- If some vertex *v* is reached (from *s*) both on a prefix of a random walk corresponding to an even-length path and on a prefix of a walk corresponding to an odd-length path then return found. Otherwise, return not-found.

Sublinear space

- Sublinear space has been an important topic in complexity theory since the start of complexity theory. While not as important as the P vs NP question, there are two fundamental questions that remain unresolved:
  - Is NSPACE(S) = DSPACE(S)? It is known that a non det. S space bounded TM can be simulated by a deterministic S^2 space bounded machine. Further NSPACE(S) = co-NSPACE(S).
  - Is P contained in SPACE(log *n*) or SPACE(log^*k n*)?

The streaming model

- In the data stream model, the input is a sequence A of inputs *a_1, …,a_n* which is assume to be too large to store in memory. The space available S(*n*) is some sublinear function. The input streams by and what can only store information in the sublinear space allotted. (It is also often desirable that that each input is process efficiently, perhaps even in time O(1).)
- The goal is to approximately compute some function or statistic of the data or identify some particular elements of the data stream.
- Most results concern the space required for a one pass algorithm. But there are other results concerning the tradeoff between the space and number of passes.

Some well studied streaming problems

- Computing frequency moments. Let A = *a_1…a_m* be a date stream with a_i in N = {1,2,…*n*}. Let *m_i* denote the number of occurences of value *a_i* in the stream A. The *k* th frequency moment F_*k* = sum_{*i* in N} (*m_i*)^*k* . F_1 = *m*, the length of the sequence. F_0 is the number of distinct elements in the stream and F_2 is a special case of interest called the repeat index (aka Gini's homogeneity index).
- Finding *k*-heavy hitters; i.e. those elements appearing at least *m/k* times in stream A.
- Finding rare or unique elements in A.

What is known about computing F_*k*

- Given an error eps and confidence bound delta, the goal is to compute an estimate F'_*k* such that |F'_*k*-F_*k*| > eps F_*k* with probability at most delta.
- For all *k >1*, there is a (space) lower bound of n^{*1-2/k*}. There is a nearly matching O~() bound.
- For *k = 0* and every *c > 2*, there is an O(log *n*) space alg : that (*1/c*) F_0 <=alg <= *c* F_0 with delta = *2/c*
- For *k =1, log m* is obvious but an estimate can be obtained with space O(log log *m* + 1/eps) space
- For *k =2*, $O(\frac{log(1/\delta}{\epsilon^2}(logn + logm))$ space

Computing F_*k*

- The basic idea behind these randomized approximation algorithms is to define a random variable Y whose expected value is close to F_*k*, variance is sufficiently small such that this r.v. can be calculated under the space constraint.
- The seminal paper on this topic was by Alon, Matias and Szegedy in STOC 1996 where they have a space O~(*n*^{1-1/*k*}) upper bound (and the improved result for F_0,F_1 and F_2). The improved bound of O~(*n*^{1-2/*k*}) is due to Indyk and Woodruff (STOC 05)

The AMS F_*k* algorithm

- Let constants s1 and s2 be defined as follows:
  $s_1 = \frac{8}{\epsilon^2} n^{1-\frac{1}{k}}$ $s_2 = 2log\frac{1}{\delta}$
- The output Y of the algorithm is the median of s_2 random variables Y1 , Y2 , ..., Ys_2 where Y_i is the average of s_1 random variables X_ij , 1 ≤ j ≤ s_1 . All X_ij are independent identically distributed random variables. Each X = X_ij is calculated in the same way using only O(log *n* + log *m*) bits as follows: Choose randomly *p* in [*1,…, m*], then see the value of a_p . Maintain *r* = |{*q*|*q* ≥ *p* and a_q = a_p }| . Define X = m(r^k – (r – 1)^k).
- Note that in order to calculate X , we only require to store a_p (i.e. log n bits) and r (i.e. at most log m bits). We need to show that E(X) = F_k and that the variance is small enough to use the Chebyshev inequality.