0368.416701 Sublinear Time Algorithms November 9, 2009

Lecture 4

Lecturer: Ronitt Rubinfeld *Scribe: Mille Gandelsman*

Sublinear time algorithm for maximal matching

Reminder

Last time we described the implementation of the oracle for the greedy sequential algorithm for finding maximal matching in a graph $G = (V, E)$. Such an oracle uses random numbers $\{r_e\}_{e \in E}$ assigned to each of the edges as follows:

Oracle implementation
Given an edge e, for every e' neighboring e of smaller rank than e, recursively check if $e' \in M$:

- if yes: return "$e \notin M$"
- else: continue.
- if no neighboring e' (of smaller rank) in M, return "$e \in M$"

Concretely, the algorithm itself is as follows:

- assign $r_e \in_r [0, 1]$ to each edge $e \in E$ for the use of the oracle.
- run the oracle reduction algorithm on a set S of $\Theta(\frac{1}{\epsilon^2})$ randomly picked edges, set $X_e = 1$ if oracle says $e \in M$ and $X_e = 0$ otherwise.
- return $\frac{m}{|S|} \sum_{e \in S} X_e + \frac{\epsilon}{2} m$, where m denotes the total number of edges in the graph.

Recall the recursion description (that appears in the scribe notes of the last lecture), and note the number of edges on which we recurse. It specifically means that we recurse only on lower numbered edges.

Correctness
The correctness follows directly from the correctness of the naive greedy algorithm, as the described algorithm is just a variant of the latter.

Running time analysis

Claim 1 *The expected number of queries to graph to handle a single oracle query is* $2^{O(d)}$

There are $O(\frac{1}{\epsilon^2})$ turns to the oracle, so the next claim using the above is immediate:

Claim 2 $E[\textit{total number of queries}] \leq \frac{1}{\epsilon^2} 2^{O(d)}$

Proof of claim 1
$Pr[\textit{given path of length } k \textit{ is explored}] = Pr[\textit{ranks of edges in path are decreasing}] = \frac{1}{k!}$
On the other hand, the number of paths of length $k \leq d^k$, and therefore: $E[\textit{number of vertices explored in the tree}] \leq \sum_{k=0}^{\infty} \frac{d^k}{k!} \leq e^d$, meaning that: $E[\textit{query complexity}] = O(de^d) = 2^{O(d)}$, as required.

Comments

- We don't have to allocate r_e for all edges in advance. Rather, we can hold a data structure (for instance: binary search tree for the edges using their r_e's as keys) and check each time we encounter an edge e whether generated such r_e already exists. We will generate a new value r_e only if it wasn't already generated.
- The algorithm in this sense is very sublinear, meaning that for most of the edges we will not generate the value r_e.

Approximating the average degree of a graph

Given a simple graph $G = (V, E)$ (meaning - no self loops and no parallel edges), we will denote for every vertex $v \in V$: $d(v) = deg(v) = no.\ of\ v\text{'s neighbors}$

Definition 3 (The average degree of a graph) *The average degree of a graph $G = (V, E)$, denoted by $\overline{d}$ is:* $\overline{d} = \frac{1}{|V|}\sum_{v \in V} d(v)$

We will use two types of queries for the algorithm:

- Degree query: given $v \in V$ output $d(v)$.
- Neighbor query: given $v \in V$ and some integer $i \in N$ output the $i - th$ neighbor of v.

The straightforward algorithm will naturally look like this: Pick s nodes $v_1, v_2, ..., v_s \in V$, and output $\frac{1}{s}\sum_{i=1}^{s} d(v_i)$. The following problem rises in the naive analysis of this algorithm: for each vertex $v_i \in V$ it holds that $0 \leq d(v_i) \leq n$ where $n = |V|$, so in order to estimate the average degree using (for instance) Chernoff bound, without assuming anything else about the numbers, we need $\Omega(n)$ samples. However, we'll see that since the d_i's come from the degrees of a graph, the estimates can actually be much better. Note that the algorithm itself is not necessarily so bad, it is just that the analysis described is bad.

The problem is well demonstrated in the example of the following two graphs:

- $G_1 = (V_1, E_1)$ where $V_1 = \{v_1^1, v_1^2, ..., v_1^k\}$ and $E_1 = \{(v_1^1, v_1^2), (v_1^3, v_1^4), ..., (v_1^{k-1}, v_1^k)\}$ - the graph with disjoint $\frac{k}{2}$ edges.
- $G_2 = (V_2, E_2)$ where $V_2 = \{v_2^1, v_2^2, ..., v_2^k\}$ and $E_2 = \{(v_2^1, v_2^2), (v_2^1, v_2^3), ..., (v_2^1, v_2^k)\}$ - the "”star” graph, with $k - 1$ edges.

$\overline{d}(G_1) = 1$ whereas $\overline{d}(G_2) = 2$, something that it will be impossible for the naive algorithm to distinguish without sampling $\Omega(n)$ vertices. There two questions we should ask ourselves are therefore:

- If we can accept a mistake factor 2, is it enough to use degree queries?
- If we would like a better approximation, how and how much can neighbor queries help?

The first idea of the sublinear algorithm is to group vertices into buckets with similar degrees. Note that each bucket will have low variance.

Definitions

- $\beta = \frac{\epsilon}{c}$ for $c > 1$.
- l is a lower bound for $\overline{d}$.
- Bucketing - define B_i as: $B_i = \{v | (1 + \beta)^{i-1} < d(v) \leq (1 + \beta)^i\}$ for $i = 0, ..., t - 1$ where t is an upper bound on $log_{(1+\beta)^n} + 1$; $B_{-1} = \{v | d(v) = 0\}$.
- $n = |V|$.

Algorithm description

Sample some $S \subseteq V$, for each bucket B_i denote: $S_i = S \cap B_i$, and set γ_i as follows:

- if S_i is "large", meaning that $|S_i| \geq \frac{1}{t}\sqrt{\frac{\epsilon}{6}\frac{l}{n}}$: $\gamma_i = \frac{|S_i|}{|S|}$ which is our approximation for $\frac{|B_i|}{n}$.
- otherwise: set $\gamma_i = 0$.

Output: $\sum_i \gamma_i(1+\beta)^{i-1}$.

Recall that $(1+\beta)^{i-1}$ is the lower bound on the degree of some vertex $v \in B_i$. Assuming that for every i: $\frac{|S_i|}{|S|} \leq \frac{|B_i|}{n}(1+\rho)$ for some ρ, we get that:
$\sum_i \gamma_i(1+\beta)^{i-1} \leq \sum_i \frac{|S_i|}{|S|}(1+\beta)^{i-1} \leq \frac{|B_i|}{n}(1+\beta)^{i-1}(1+\rho) \leq \overline{d}(1+\rho)$
where the first inequality is true because some of the γ_i's equal to 0, the second is simply our assumption, and the third uses the definition of B_i.
Let us now try to determine how much could we underestimate the value of $\overline{d}$, there are three pairwise disjoint sets of edges $e = (u, v)$:

- E_1 = edges (u, v) s.t. both u and v are in some large B_i; such edges are counted twice in the algorithm.
- E_2 = edges (u, v) s.t. u is in some large B_i, whereas v is in some small B_i; such edges are counted once in the algorithm.
- E_3 = edges (u, v) s.t. both u and v are in some small B_i; such edges are not counted in the algorithm at all.

It holds that: $E = E_1 \cup E_2 \cup E_3$. By our choice of the cutoff for deciding which B_i's are large we guarantee that $|E_3| \leq O(\epsilon n)$ so: $\sum_i \gamma_i(1+\beta)^{i-1} \geq \frac{d}{2}\frac{1-\rho}{1+\beta} - \epsilon$ (the total number of vertices in small buckets is $O(\sqrt{\epsilon n})$ which implies that the total number of edges in E_3 is about $O(\sqrt{\epsilon n}\ over\ 2) = O(\epsilon n)$).

Claim 4 *Sample size of* $\frac{n}{\sqrt{\epsilon n}} = \sqrt{\frac{n}{\epsilon}}$ *gives* $(2 - O(1))$ *approximation of* $\overline{d}$

The claim is straightforward conclusion from the above discussion.

Full scale algorithm description

The full scale algorithm uses also the random neighbors queries as follows: For large bucket i, denote by α_i the fraction of edges from E_2 hanging off bucket i. The lower bound on the average degree of the edges in B_i is $(1+\alpha_i)\gamma_i(1+\beta)^{i-1}$, where the α_i is added to make up for the edges from E_2 that were not seen in small buckets.

How to approximate α_i

For every vertex in a large S_i, we pick a random neighbor and set X_i as follows: $X_i = 1$ if the neighbor is in some small S_j and $X_i = 0$ otherwise. If all vertices in B_i would have had the same degree then we could choose a vertex uniformly at random from B_i, here two vertices have nearly the same degree (concretely - up to $(1+\beta)$). Denote: $p = Pr[\textit{edge slot of } S_i \textit{ is picked}]$. For an edge (u, v) such that $u \in B_i$ the probability to pick an edge is $\frac{1}{|B_i|}\frac{1}{d(u)}$ so: $\frac{1}{\#S_i's\ edge\ slots}\frac{1}{1+\beta} \leq p \leq \frac{1}{\#S_i's\ edge\ slots}(1+\beta)$ where by edge slot we mean the number of vertices of the edge that are in the bucket.
The full scale algorithm will look like this:

- set $S = \Theta(\sqrt{\frac{n}{l}}\epsilon^{-\frac{4}{5}}log^2(n)log(\frac{1}{\epsilon}))$ sampled vertices.
- set $S_i = S \cap B_i$.
- if S_i is large, i.e. $\frac{|S_i|}{|S|} \geq \frac{1}{t}\sqrt{\frac{\epsilon}{6}\frac{l}{n}}$ use $\frac{|S_i|}{|S|}$ to approximate $\frac{|B_i|}{n}$, setting: $\gamma_i = \frac{|S_i|}{|S|}$.

- for every large bucket i and for every $v \in S_i$: pick random neighbor u of v and set X_v as follows: $X_v = 1$ if u is in some small bucket and $X_v = 0$ otherwise.
- for every large bucket i: set $\overline{\alpha_i} = \frac{|\{v \in S_i | X_v = 1\}|}{|S_i|}$.
- Output $\frac{1}{|S|} \sum_{i \in \{j | j \textit{ is a large bucket index}\}} (1 + \overline{\alpha_i})|S_i|(1+\beta)^i$.