CSE 522: Sublinear (and Streaming) Algorithms **Spring 2014**

Lecture 17: Tolerant Testing, Graph Algorithms/Streaming

May 28, 2014

Lecturer: Paul Beame *Scribe: Paul Beame*

1 Tolerant Testing

So far in property testing we have consider testers of the form, with probability at least 2/3, determine whether

- the input has property P, or
- the input is ε-far from any input having property P.

This gives complete freedom to answer if the input is close to having property P but does not exactly have this property. It is natural to think of property testing algorithms for which the input queries might have a very small probability of error. In this case it would be good to be able to separate inputs that are ε-far from having property P from those that are ε'-close to having property P where $\varepsilon' > 0$. We call this a $(\varepsilon', \varepsilon)$-tolerant tester. Ideally, one would want such a tester to have a running time that is a small inverse polynomial in $\varepsilon - \varepsilon'$.

Though it is not of this desired form, on the second problem set there is a fairly simple problem to show that when the distance is fractional Hamming distance, any property testing algorithm making $q(n, \varepsilon)$ queries is automatically an $(\varepsilon/(3q(n, \varepsilon)), \varepsilon)$-tolerant property tester.

2 Sublinear Mutliplicative-Additive Approximation Algorithms

In lecture 15 we showed how to efficient approximate the MST size within a εn additive term, which yielded a $(1 \pm \varepsilon)$-approximation because the MST size is at least $n - 1$. The εn additive term is essential for many sublinear algorithms for optimization problems because the algorithm will always miss reading most of the input and that unread part could change the optimum by a large amount. To handle these cases naturally, we need a more general definition of approximation.

Definition 2.1. *An algorithm gives an* (α, ε)-approximation *to the optimum value OPT of a minimization problem iff the algorithm produces as output a value $\tilde{OPT}$ such that*

$$OPT \leq \tilde{OPT} \leq \alpha \cdot OPT + \varepsilon\, n.$$

As with the MST algorithm, we consider a query model in which an algorithm can query the degree $deg(v)$ of any vertex v, and can give the query (v, i) which gives the name of the i-th outneighbor of the vertex v. When we described the MST algorithm we assumed that the graph had a maximum degree d. Similar bounds hold even in terms of the average degree $\bar{d}$ but again we assume a maximum degree d for simplicity here.

The problems we will consider are the vertex cover problem, the maxmimal (not maximum) matching problem and the maximal independent set problem. These three problems are related in terms of approximation: Any vertex cover must contain at least one vertex from each edge in any matching. If we take all vertices touched by any maximal matching then we certainly obtain a vertex cover of the graph, but it is also at most twice the minimum size required. Furthermore, it is easy to see that a maximal matching is simply a maximal independent set in the *line graph* $L(G) = (V', E')$ of the original graph $G = (V, E)$, where $V' = E$ and E' contains all pairs of edges of G that share a vertex.

The general idea will be to derive an algorithm that will test for any vertex v, using a small number of queries, whether or not v is touched by a fixed maximal matching and therefore included in the vertex cover given by the 2-approximation above. The algorithm to estimate the size of the vertex cover will randomly choose an independent sample of vertices and test each one using the algorithm. The fraction of vertices in the sample that are in the cover is the estimate of its size which is at most $2 \cdot OPT$ by construction. By Chernoff bounds, since the sampling procedure is a sum of independent coin flips, each of which has probability of heads $2 \cdot OPT/n$ if we take $s = O(1/\varepsilon^2)$ samples, it is within $\pm\varepsilon n/2$ of OPT with probability at least $2/3$. We if add $\varepsilon n/2$ to the value, then the resulting $\tilde{OPT}$ will always be between $2 \cdot OPT$ and $2 \cdot OPT + \varepsilon n$. It remains to show how to determine whether a vertex v is touched by a fixed maxmal matching.

From greedy algorithms to sublinear algorithms We first recall the standard family of greedy algorithms for maximal independent set (or maximal matching). We consider the vertices (edges, for maximal matching) in some fixed order given by a permutation π on $[n]$.

The algorithm is as follows: Suppose that the vertices are $v_1, \ldots, v_n$. Begin with $I = \emptyset$. For $i = 1$ to n, add vertex $v_{\pi(i)}$ to I iff it is not a neighbor of any element of I. In the case of maximal matching, the edges are ordered and we add each edge to the matching M that does not share endpoints with any element of M.

The above algorithm is naturally very sequential. In the 1980's Luby [] described both randomized and deterministic parallel versions of this basic algorithm idea. We consider a simple randomized version: Each vertex independently chooses a random real *rank* $\in [0, 1]$. Since having two identical rank values has probability 0, this implicitly defines a rank order on the vertices (and the same can be done for edges in the case of maximal matching). A vertex (edge) will be included in the maximal independent set (maximal match) if all its neighbors are larger rank or, more generally, if all its neighbors of smaller rank are not in the independent set because of their other neighbors.

In the original parallel algorithm, the algorithm is actually run in $\log n$ rounds: in each round only those vertices (edges) that are smaller than all their neighbors are retained; all their neighboring vertices (edges) are deleted and the algorithm is run for another round. The algorithm in this case is optimized for parallel execution but is not sublinear.

In the sublinear algorithm we consider, which we describe for the case of maximal matching, was originally designed by Nguyen and Onak [] using the same idea of fixing a random ranking at the start. However, we only produce the rank values on demand, storing the ones we have produced, and we explicitly follow the chain of neighbors to determine the presence or absence of a single edge in the maximum matching relative to the induced permutation π. We describe the algorithm involving two oracle functions VO^π which determines whether or not vertex v is contained in any edge of the maximal matching corresponding to a random ranking π. $VO^\pi(v)$ will call function $MO^\pi(e)$ which will determine whether or not edge e is contained the maximal matching corresponding to π. (In both the O stands for "oracle".

$VO^\pi(v)$

Let $e_1, \ldots, e_d$ be the edges touching vertex v in order of increasing rank under π.
for $i \leftarrow 1$ to d **do**
    if $MO^\pi(e_i) = TRUE$ **then**
        Return TRUE
    end if
end for
Return FALSE

$MO^\pi(e)$

Let $e_1, \ldots, e_k$ for $k \leq 2(d-1)$ be the edges touching edge e in order of increasing rank under π.
$i \leftarrow 1$
while $\pi(e_i) < \pi(e)$ **do**
    if $MO^\pi(e_i) = TRUE$ **then**
        Return FALSE
    else
        $i \leftarrow i + 1$
    end if
end while
Return TRUE

Intuitively, the algorithm checks the neighbors of a vertex or edge starting with the smallest rank neighbor since that is the hardest to eliminate because elimination only occurs if there are even smaller numbered edges. The recursion in $MO^\pi(e)$ is tricky to analyze as is the fact that ranks from previous parts of the execution may reappear in later parts. It is also obvious that in the worst case the recursion can do on for an extended number of nodes. However, the randomness involved

in the algorithm will show that these chains are bounded independent of the number of nodes in the graph.

Assume that we are at the start of the algorithm when no prior ranks have been found. Consider a fixed path of edges of length t beginning with a call to MO^{π} from $VO^{\pi}(v)$ including a fixed edge e touching v and assume that the rank of every edge has not been set when it is explored. Each edge is touched by at most $2(d-1)$ other edges since the graph has degree at most d. In order for the algorithm to explore the edges along this path, each edge must have smaller rank than the previous one. The probability that this happens for these edges is precisely $1/t!$. There are at most $(2d-2)^{t-1}$ paths of length t in this sequence of edges Observe also for a path of length t need be counted only once in order to describe its contribution because its subpath of length $t-1$ will count all but its last edge. Therefore we have an expected cost of this call of at most

$$\sum_{t=1}^{\infty}(2d-2)^{t-1}/t! = \frac{1}{2d-2}\sum_{t=1}^{\infty}(2d-2)^{t}/t! = (e^{2d-2}-1)/(2d-2).$$

Nguyen and Onak showed that a similar exponential upper bound in d but independent of n, also applies when the ranking has previously been explored to a small extent, yielding a $2^{O(d)}/\varepsilon^2$ query $(2,\varepsilon)$-approximation algorithm for vertex cover. They were not able to take into account the full ordering properties in which the vertices were considered, or the fact that the original vertex v was randomly sampled. Yoshido, Yamamoto, and Ito later used these properties to show that it is an $O(d^4/\varepsilon^2)$ query algorithm, though the proof is quite involved and too long to present here. Onak, Ron, Rosen, and Rubinfeld then extended these ideas, along with more efficient data structures to store and compare rankings and explore the neighborhoods around each vertex to show that the above algorithm blueprint can actually be made to work in time $O(\bar{d}/\varepsilon^2)$.

3 Graph Semi-streaming Algorithms

We mention a final topic briefly because it was not chosen for the project presentations. Though there are a few algorithmic problems on graphs where streaming algorithms using space sublinear in the number of vertices are possible, they are the exception rather than the rule. In graph semi-streaming algorithms, the goal is to obtain algorithms that always are better than $\Theta(n^2)$ space even if the input graph is dense, and ideally to produce algorithms with space complexity $n\log^{O(1)} n$. These algorithms are still worthwhile when dealing with very large possibly dense graphs when a million vertices in a graph can easily be stored in main memory, but storing all the edges would be prohibitive. This is quite an active area of current research. We give a very brief taste of some results:

Spanners and All-Pairs Shortest Paths approximation The goal is given G to produce a (small) subgraph H of unweighted graph G that allows us to estimate (the lengths of) all pairs

shortest paths in G. H is a k-*spanner* of G iff for every pair of vertices $u, v \in G$, the length of the shortest path in H is at most k times the length in G.

The streaming algorithm to produce a $(2k-1)$-spanner is quite easy: For each edge (u, v), if there is no path between u and v of length at most $2k-1$ in H then add (u, v) to H.

H obviously satifies the $(2k-1)$-spanner property since any edge that is not included can be bypassed using the existing path in H and we simply splice together the paths corresponding to each edge not in H to get a path in G and each edge's costs have been increased by at most a $2k-1$ factor.

It remains to show that H is not too large. We claim that H has size at most $n^{1+1/k}$. Let m be the number of edges in H.

Observe that H does not have any cycles smaller than $2k+1$ since the last edge of a cycle of length at most $2k$ would not have been added. We will show that if H has too many edges then this is not possible. To see this, we first prune H to remove all vertices of degee significantly lower than the average degree $\bar{d} = 2m/n$. That is, we define $d' = \bar{d}/2$ and repeatedly remove from the graph H any vertex of degree $< d'$. At each step this may create additional vertices with degree smaller than d'. The number of edges of H removed by this procedure must be $< d'n = \bar{d}n/2 = m$ so the resulting graph H' (which is called the emphd'-core of H) must be non-empty.

We derive a contradiction from this fact. H' has minimum degree at least d' and no $2k$-cycles or smaller because it is a subgraph of H. Now consider a breadth-first search tree from any vertex of H' of height k. The fan-out of the root is at least d' and of each internal node is at least $d'-1$. Since there are no cycles of length smaller than k, all these vertices must be distinct. Therefore $n \leq d'(d'-1)^{k-1}$ and hence $d'-1 \leq n^{1/k}$. Therefore $\bar{d}$ is $O(n^{1/k})$ and hence m is $O(n^{1+1/k})$.

Lower Bounds Guruswami and Onak has derived lower bounds for a number of problems in the graph semi-streaming model, based on communication complexity of direct sums of pointer-jumping problems. For example they show that using p passes it requires $n^{1+1/p}/p^{O(1)}$ space to determine whether or not two vertices are within distance $2(p+1)$ of each other. However, other problems can be done using $n \log^{O(1)} n$ space.

Vertex Sketches Even nicer ideas are possible to allow one to use sketching for graphs, which allows deletions as well as insertions of edges. A particularly beautiful idea is used by Ahn, Guha, and McGregor is to produce a $\log^{O(1)} n$-size linear sketch of the neigborhood vector for each vertex (for a total of $n \log^{O(1)} n$ space) and then combine them to deduce connectivity properties and even spanning forests in a single pass.

The general idea is to do hooking and linking of connected components as in the Boruvka or Kruskal spanning tree algorithms but without the weight. One imagines the neighborhood vector

of each vertex as being described by a length $\binom{n}{2}$ vector of that has a ± 1 value for each edge touch the vertex depending on whether the neighbor has a smaller or larger vertex number (and is otherwise 0). One can show that the sum of the vectors associated with a connected component yields a vector that has ± 1 values associated with its outedges. Such a vector has all its ℓ_p norms the same and one can use a sketch that allows one to sample components in proportion to their value in order to derive an algorithm that will choose a linking edge for each component. Doing this repeatedly (only $\log n$ levels of linking are enough) eventually connects the graph and from this the spanning trees can be found. Because it is a linear sketch, it is completely generalizable to deletions.

4 Beyond

We have only given a sample of problems and directions in streaming and sublinear algorithms. There is much more to explore and more, including many open problems can be found at http://sublinear.info.