ECS 189A Sublinear Algorithms for Big Data **Fall 2024**

Lecture 5: Lecture Title

Lecturer: Jasper Lee *Scribe: Reina Itakura and Kei Rockwell*

1 Overview & The Bounded Degree Graph Model

In the previous lecture, we considered property testing (with two different distance metrics) on lists. Over the next few lectures, we'll instead be looking at graphs. An important thing to keep in mind with sublinear algorithms for graphs is what the size of the input is, as that is what it's sublinear to. The algorithms discussed today are generally independent of the size of the graph.

The first consideration for graph algorithms is how we represent the graph. There are two main representations: an adjacency list, which is useful for sparse graphs; and an adjacency matrix, typically used for dense graphs. We will be focusing on sparse graphs, so our chosen representation will be based on an adjacency list. However, there are two notions of sparsity that could be considered. A "sparse" graph could have low *average* degree but permit a few high degree vertices. The model we will use today requires a uniformly sparse graph, where each vertex has a low maximum degree.

1.1 Bounded Degree Graph Model

The graph model we utilize has one operation, $\text{Query}(v, i)$ that returns the i-th neighbor of v. Each vertex has maximum degree d, which ensures the graph is uniformly sparse provided d is small relative to n. The only other guarantee that this model makes is if $\text{Query}(v, i)$ is not Null, then v has at least i neighbors. Notably, there is no guarantee of any ordering between different lists: vertex v could be "before" vertex u on one adjacency list but not on another. Binary search allows the degree of a vertex to be found in $\log d$ time and query complexity.

The distance notion we choose is based entirely on the difference between adjacency lists. For this, we can represent the adjacency lists as an $n \times d$ matrix

$$d(G_1, G_2) = \frac{1}{dn}\,(\text{number of entries in the matrix where } G_1 \neq G_2)$$

This distance notion is impacted by the (not sorted) order of the adjacency lists. In addition, given a permutation $\pi : [n] \to [n]$, its possible for $d(G, \pi G) > 0$. While this seems an unintuitive distance metric (as isomorphic graphs might be considered very distant), it allows for the testing of "weird" properties. The connectedness related properties we discuss today do not rely on representation, but some properties are entirely representation dependent. For example, "vertex 1 is connected to vertex 7" is representation dependent (as vertex names are arbitrary) but is still a property that is captured by this distance notion.

1.2 Giving Meaning to Distance

Given the above distance notion, what does it mean for a graph to be ε-far from the set of connected graphs? One answer is that we need to add or redirect at least εdn entries in the adjacency lists to make it connected. But this lacks good insight into the structural properties of the graphs themselves. Since we're trying to reason about connectedness, a reasonable starting point would be trying to quantify how many connected components a ε-far from connected graph has.

Lemma 5.1. *If graph G is ε-far from connected, then there are at least $\frac{\varepsilon dn}{2}$ connected components.*

Proof. It is equivalent to show that if a graph G has less than $\frac{\varepsilon dn}{2}$ connected components, then G is not ε-far from the set of connected graphs.

Given k connected components, a graph can be connected by adding $k-1$ edges. In most cases, adding an edge requires two changes — each involved node can change one entry in the adjacency list — which results in $2k-2$ changes required. However, one additional change is needed if a d-clique lands at the end of the connected component "chain", which can happen at most twice. So G can be connected making at most $2k$ changes.

Consequently, if G has less than $\frac{\varepsilon dn}{2}$ connected components, it can be connected with less than εdn changes and so is not ε-far. □

Aside: the contrapositive is usually a good starting point for showing facts about objects that are ε-far, because it frequently enables the proof to be completed by constructing a fairly simple object that is not ε-far. It seems a lot muddier to prove these results directly.

2 Connectedness Testing In Sparse Graphs

One problem that we might ask about a sparse graph is if it is connected or not. In particular, we will be testing to determine if a graph is connected vs ε-far from being connected, using the distance notion described in Section 1.1.

First, notice that if there are many connected components, most components should be relatively small. If we can sample one such component, the graph is clearly not connected. So, to determine if a graph is ε-far from being connected, we can look for connected components below a certain size based on that ε:

Lemma 5.2. *If G is ε-far from connected, then there are at least $\dfrac{\varepsilon dn}{4}$ components of size less than $\dfrac{4}{\varepsilon d}$.*

Proof. Suppose, seeking contradiction, that there is a graph G that is ε-far from connected but with less than $\frac{\varepsilon dn}{4}$ components smaller than the desired bound. From Lemma 5.1, there must be at least $\frac{\varepsilon dn}{2}$ connected components, so G must have more than $\frac{\varepsilon dn}{2} - \frac{\varepsilon dn}{4} = \frac{\varepsilon dn}{2}$ connected components with size greater than $\frac{4}{\varepsilon d}$. But then the total number of nodes contained in these "large" components *exceeds* $\frac{\varepsilon dn}{4} \cdot \frac{4}{\varepsilon d} = n$.

Based off of our initial intuition, we can run BFS on a random sample of nodes and look for small connected components, specifically those with less than $\frac{4}{\varepsilon d}$ nodes.

Algorithm 5.3 Connectedness Testing In Sparse Graphs

```
1: procedure REPEAT $O(\frac{1}{\varepsilon d})$ times:
2:     Pick a random vertex u
3:     Run BFS from u for at most $\frac{4}{\varepsilon d}$-size neighborhood
4: end procedure
5: Reject if a connected component of size $< n$ is found.
```

What's the query/runtime complexity? Since we run BFS on $O(\frac{1}{\varepsilon d})$ vertices, and per BFS we see at most a $O(\frac{1}{\varepsilon d})$ vertex-neighborhood where each vertex has a max degree of d,

$$\leq O\left(\frac{1}{\varepsilon d}\right) \cdot O\left(\frac{1}{\varepsilon d}\right) \cdot d \leq O\left(\frac{1}{\varepsilon^2 d}\right)$$

This bound is not especially precise because the d factor assumes each vertex reached in the BFS results in d work being done, while much of that ends up being performed in the next stage of the BFS.

Note: Other algorithms can improve this bound to $O(\frac{1}{\varepsilon} \text{ polylog } \frac{1}{\varepsilon^2 d})$.
Note: A non-adaptive algorithm will have an $\Omega(\sqrt{n})$ query complexity lower bound.

3 Approximating Number of Connected Components

A related problem is approximating the number of connected components. We will aim to do this with "additive error," namely that the approximation is within $\pm \varepsilon n$ of the actual value, again with probability $\frac{2}{3}$. The algorithm discussed today accomplishes this with query complexity in $O\left(\frac{d}{\varepsilon^3}\right)$. Improvements to $O\left(\frac{d}{\varepsilon^2} \log \frac{d}{\varepsilon}\right)$ are known, and the query complexity of such an algorithm is lower bounded by $\Omega\left(\frac{d}{\varepsilon^2}\right)$.

The intuition guiding our approach is similar to the connectedness testing case — the average size of connected components should determine how many there are. Guided by this, we will sample random vertices to perform BFS on, which will then return the size of the component. This allows us to estimate the mean connected component size.

Notation: $\mathcal{N}_u$ denotes the size of the connected component containing u.

Proposition 5.4. *The number of connected components in a graph G is*

$$\sum_{u \in G} \frac{1}{\mathcal{N}_u}$$

Proof. For each connected component $\mathcal{C}$ of G:

$$\sum_{u \in \mathcal{C}} \frac{1}{\mathcal{N}_u} = 1 \tag{1}$$

Summing (1) over each connected component $\mathcal{C}$ of G clearly yields the number of connected components, and is equivalent to the original sum. □

Algorithm 5.5 Inefficient Procedure to Approximate # of Connected Components

```
1: for $\Theta(\frac{1}{\varepsilon^2})$ times do
2:     Pick random vertex $u$
3:     use BFS to compute $\mathcal{N}_u$
4: end for
5: return (scaled) sum of $\frac{1}{\mathcal{N}_u}$
```

However, Line 3 could have high cost if there are few connected components in the graph. The BFS will make queries linear in size to the connected component, which could be up to $O(n)$. We sacrifice some sampling accuracy to work around this by upper bounding the BFS to $2/\varepsilon$ vertices, which introduces a bias in the sample mean.

Algorithm 5.6 Approximating Number of Connected Components

```
1: for $\Theta(\frac{1}{\varepsilon^2})$ times do
2:     Pick random vertex $u$
3:     use BFS to compute $\mathcal{N}_u$, stopping after $O(2/\varepsilon)$ vertices
4: end for
5: return (scaled) sum of $\frac{1}{\mathcal{N}_u}$
```

This algorithm has two different sources of error — the bias from truncating the sampling discussed above and the expected error from the sample mean. The bias is bounded by $\frac{\varepsilon n}{2}$. The sample mean error can be easily bounded using Hoeffding's inequality, as $1/\mathcal{N}_u$ is bounded above by 1 for any u. In particular, we choose to bound it by $\varepsilon n/2$ so that the total error is within $\pm \epsilon n$ as desired.

4 Approximating Minimum Spanning Tree (MST) Weight

Finally, we take a look at approximating minimum spanning tree weight. In the Bounded Degree Graph Model described in Section 1.1, the degree of any vertex in the graph is at most degree d. In addition, we will assume each edge has integer edge weights in $\{1 \dots w\}$, which can be achieved by rounding/approximating each edge weight to an integer.

We want to find the minimum weight spanning tree (denoted $w(MST)$) with room for $\pm\varepsilon n$ error. Notice that $w(\text{MST}) \pm \varepsilon n \in [(1 \pm \varepsilon) \cdot w(\text{MST})]$. In other words, the output will be within a multiplicative constant of $w(MST)$. This is because each edge weight is at least 1, which implies $w(MST) \geq n - 1$. The query complexity of the algorithm we will look at will be $O\left(\frac{dw^4 log(w)}{\varepsilon^3}\right)$. In addition, as of 2020, the known query complexity is $O(\frac{dw}{\varepsilon^2} \log \frac{dw}{\varepsilon})$ upperbound, and a $\Omega(\frac{dw}{\varepsilon^2})$ lowerbound.

The idea for the algorithm is to reduce the approximate weight of the MST to the approximate counting number of connected components in a family of subgraphs.

Recall Kruskal's Algorithm for calculating the MST of a graph. It is a greedy algorithm which takes the next lowest weight edge that connect disconnected connected components, until all vertices are connected with one another.

Algorithm Kruskal's Algorithm for Minimum Spanning Tree

```
1: $G \leftarrow$ input graph
2: while $G$ is disconnected do $i = 1, 2, \ldots$
3:     Keep adding edges weight $= i$ if adding that edge connects disconnected components.
4: end while
```

Let β_i = the number of edges in MST with weight $> i$, and C_i = the number of components in the subgraph induced by edges weight $\leq i$. By induction, we can prove that after i iterations of Kruskal's, we have C_i many connected components. Since the number of edges needed to connect C_i connected components is $C_i - 1$, $\beta_i = C_i - 1$. We can then find a closed form for $w(MST)$:

$$\begin{aligned} w(MST) &= \sum_{i=0}^{w-1} (i+1)(\beta_i - \beta_{i+1}) \\ &= \sum_{i=0}^{w-1} \beta_i \\ &= \sum_{i=0}^{w-1} C_i - 1 \\ &= n - w + \sum_{i=1}^{w} C_i \qquad (*) \end{aligned}$$

In Algorithm 5.6, we have a way to estimate the number of connected components in a graph. Repeated application of this algorithm, then, allows us to approximate the MST weight using the above sum.

Algorithm 5.7 Approximate MST Weight

```
for $i = 1 \ldots w-1$ do
    Compute $\widetilde{C_i} \leftarrow$ output of Algorithm 5.6 with error $= \frac{\varepsilon}{w}$ and fail probability $\delta = \frac{1}{3w}$
end for
Return $n - w + \sum_i \widetilde{C_i}$
```

To calculate the total error, note that we are calling Algorithm 5.6 with error $\frac{\varepsilon}{w}$ approximately w times. Thus, our total error will be $\leq \frac{\varepsilon}{w} \cdot w = \varepsilon$. Similarly, we run Algorithm 5.6 with a fail probability $(\delta) = \frac{1}{3w}$ approximately w times, resulting in a fail probability $\leq \frac{1}{3w} \cdot w = \frac{1}{3}$.

In Section 3 we express the number of connected components as $= \sum_u \frac{1}{N_u}$, and in Algorithm 5.6, we estimate the number of components by summing random samples. However, in Algorithm 5.7, we are computing $\widetilde{C_i}$ for all weights, and are not taking random samples. Why don't we subsample $\sum C_i$ instead? There are two reasons:

1. We have no idea how to calculate the variance over uniform $\{C_i\}$.
2. Although we can use Hoeffding's inequality (see lecture 3) to bound $\sum_{i=1\ w} C_i$, but this will give us different guarantees. Hoeffding's gives us a bound of $O(M^2/(\varepsilon n)^2)$, which is $O(1/\varepsilon)$.