CS5234: Combinatorial and Graph Algorithms Lecture 1–3

Sublinear Time Graph Algorithms

Lecturer: Seth Gilbert *August 16–31, 2018*

Abstract

Today, we will focus on sublinear time graph algorithms. The goal is to compute some property of a graph while only looking at a very small fraction of the nodes in the graph. We will look at how to decide if a graph is connected, to compute the number of connecte components, and the (approximate) weight of the minimum spanning tree.

1 Background

Assume we have a graph $G = (V, E)$. We will assume for today that the graph has maximum degree d (and so $m \leq dn$). We will also assume that the graph is stored as an adjacency list. We will assume that $d \geq 3$. (Otherwise all our graphs our lines.)

The basic way in which we can access the graph is via a *query* operation. The *query*(v, i) returns the ith neighbor in the adjacency list of v, or null if there is no such neighbor.

We will measure the distance between two graphs as the fraction of entries in the adjacency list that have to change. That is, given two graphs G_1 and G_2, if we need to add or delete e edges to transform G_1 into G_2, we will say that

$$distance(G_1, G_2) = \frac{2e}{dn} .$$

Notice that here, e is the minimum number of edge changes needed to transform G_1 into G_2. Each addition or deletion of an edge requires changing two entries in the adjacency list representation of the graph (i.e., both endpoints of the edge). Also notice that we divide by dn, which is the maximum possible size of the graph—rather than the actual size.

Throughout these notes, we will assume that ϵ is some fixed constant $< 1/2$ and > 0.

2 Graph Connectivity

Our first goal is to decide whether a graph $G = (V, E)$ is connected, or ϵ-far from being connected. We say that a graph G is ϵ-far from being connected if we need to modify at least ϵdn entries in the adjacency list of G to create a connected graph. That is, for every graph G' where $distance(G, G') \leq \epsilon$, the graph G' is not connected.

Our goal is to build an algorithm where, given a graph G, it produces the following output:

- If the graph G is connected, it returns true.
- If the graph G is ϵ-far from being connected, it returns false with probability at least 2/3.
- Otherwise, it can return either true or false.

We begin with a simple claim that a graph that is ϵ far from being connected has a large number of connected components. This will make it easy to tell whether a graph is connected or ϵ-far from being connected.

Claim 1 *If graph G is ϵ-far from being connected, then it has at least $\epsilon dn/4$ connected components.*

Proof Assume the graph has k connected components. A graph with k connected components can be connected by adding $k-1$ edges (each connecting two connected components), which requires $2(k-1)$ changes to the adjacency list of G.

Assume for the sake of contradiction that $k < \epsilon dn/4$. Then $2(k-1) < \epsilon dn$, which immediately implies that G is not ϵ-far from being connected (since we can construct a G' that is close and connected adding $2(k-1) < \epsilon dn$ edges). This implies a contradiction and hence G must have at least $\epsilon dn/4$ connected components.

There is one problem with this argument. We have to build graph G' from G by adding edges connecting connected components. However, it may be the case that some nodes in G already have d neighbors, i.e., we cannot add another edge without violating the constraint on the maximum size of the adjacency list.

Assume you have a connected component A in which all the nodes have d neighbors. If this component has ℓ nodes, this implies that there are ℓd edges inside the connected component. Find a spanning tree T of the component A. Note that the spanning tree has at most $\ell - 1$ edges. Thus there must be at most one edge (u, v) in A that is not in the spanning tree. Delete that edge. Now we have two nodes u and v with degree at most $d-1$, i.e., we can add an edge from u or v to another connected component.

For each component in which there are not at least two nodes with degree $< d$, run this procedure on that component, deleting at most one edge. After we are done with this procedure, each component has at least two nodes with degree $< d$ that can be used to connect the component to other components.

Each time we run the edge deletion procedure, we have to modify two entries in an adjacency list. Since there are k components, this has cost $2k$.

Therefore, beginning with graph G, we need to perform $2k + 2(k-1) \leq 4k$ changes to the adjacency list to construct a conencted graph G'. If $k < \epsilon n/4$, then the total number of changes to construct a connected graph is $< \epsilon dn$, which contradicts the claim that G is ϵ-far from being connected. We thus conclude that $k \geq \epsilon dn/4$. □

Sometimes, a tighter claim is given, arguing that really the graph has at least $\epsilon dn/2$ connected components if it is ϵ-far from connected, since if the graph has k connected components, we only need to modify $2k$ positions in the adjacency list (since we can modify the adjacency list directly, instead of via adding and deleting edges). However, for today, the above claim is good enough.

We can now argue that if G is ϵ-far from being connected, not only does it have a large number of connected components, then it must also have a large number of *small* connected components:

Claim 2 *If G is ϵ-far from connected, that it has at least $\epsilon dn/8$ connected components of size at most $8/\epsilon d$.*

Proof This claim follows from a simple counting argument, i.e., the basic fact that no more than half the elements in a set can be bigger than twice the average size.

We know (from the previous claim) that the graph G has at least $\epsilon dn/4$ connected components. Assume for the sake of contradiction that $\geq \epsilon dn/8$ of the components are of size $> 8/\epsilon d$. Let S be the set of such connected components. This would imply that the set S contains more than $(\epsilon dn/8)(8/\epsilon d) = n$ nodes. That, however, is impossible since the graph in total only has n nodes.

We conclude, then, that graph G has $< \epsilon dn/8$ components of size $> 8/\epsilon d$. This implies that the remaining components must be of size $\leq 8/\epsilon d$, i.e., there must be at least $\epsilon dn/4 - \epsilon dn/8 = \epsilon dn/8$ connected components of size $\leq 8/\epsilon d$. □

This motivates the following algorithm: we choose nodes at random, and determine if they are in a connected component of size $\leq 8/\epsilon d$. If so, we can safely return false: the graph is not connected. If the graph is connected, it will always return true, as it will never discover a small connected component. If the graph is ϵ-far from being connected, then there must be at least $\epsilon dn/8$ such small connected components, each of which must contain at least one node.

Algorithm 1: Connected$(G = (V, E), n, d, \epsilon)$

repeat $16/\epsilon d$ times
  Choose a random $u \in V$.
  Perform a BFS from u, stopping as soon as $8/\epsilon d$ new nodes are discovered.
  if *u is in a connected component with $\leq 8/\epsilon d$ nodes* **then return false**.
return true.

Hence each iteration has a $\epsilon d/8$ probability of finding a small connected component. By repeating $\Theta(8/\epsilon d)$ times, we can ensure that we find a small connected component with probability at least $2/3$.

We first analyze the cost of this algorithm:

Claim 3 *The algorithm runs in time $O(1/\epsilon^2 d)$.*

Proof The main cost in each iteration is performing the BFS. The BFS explores at most $8/\epsilon d$ nodes, and at each of these nodes it examines the entire adjacency list, i.e., up to d neighbors. (Remember, during the BFS we may find edges that connect to nodes we have already found. So we must explore more than $8/\epsilon d$ edges to explore all the nodes in a connected component of size $8/\epsilon d$.) Thus total number of edges explores is huts at most $8d/\epsilon d = 8/\epsilon$.

The algorithm repeats the sampling step $16/\epsilon d$ times, and hence the total cost is $O((1/\epsilon d)(1/\epsilon)) = O(1/\epsilon^2 d)$. □

We now how that the algorithm is correct. It is easy to see that if the graph is connected, then the algorithm never returns false: it will certainly never find a small connected component in that case. The interesting case is when the graph is ϵ-far from connected.

Claim 4 *If graph G is ϵ-far from connected, then with probability at least $2/3$, the algorithm returns false.*

Proof If the graph G is ϵ-far from connected, then we know (from the above claims) that it has at least $\epsilon dn/8$ connected components of size at most $8/\epsilon d$. Since each connected component contains at least one node, there are at least $\epsilon dn/8$ nodes in the graph wherein if the algorithm selects that node, it will find a small connected component and return false.

Thus, in each iteration, there is a probability of at least $(\epsilon dn/8)/n = \epsilon d/8$ of detecting that the graph is not connected. We can therefore calculate the probability that, even though the graph is ϵ-far from connected, the algorithm never selects a node in a small component:

$$\begin{aligned} \left(1 - \frac{\epsilon d}{8}\right)^{\frac{16}{\epsilon d}} &\leq e^{-2} \\ &\leq 1/3 \end{aligned}$$

Thus we conclude that with probability at least $2/3$, the algorithm correctly detects the fact that the graph is disconnected. □

To conclude, we have given an algorithm for determining, with probability at least $2/3$, whether a graph is connected or ϵ-far from connected.

3 Number of Connected Components

The next question we want to address is the number of connected components in the graph. The previous algorithm only determines whether or not the graph is connected. We would like to extend this algorithm to also determine the (approximate) number of connected components in the graph.

Specifically, given a graph $G = (V, E)$, let C be the number of connected components in the graph. Our goal is to develop an algorithm that will return a value C' such that:

$$C - \epsilon n \leq C' \leq C + \epsilon n \, .$$

Or, to put it differently, the error should be at most ϵn, i.e., $|C' - C| \leq \epsilon n$. Notice that this approximation is useful only if the graph has a large number of connected components, i.e., it is similar to the condition that G is ϵ-far from connected. If the graph only has a small number of connected components, then the estimate is not particularly useful. We will see later, however, that we can use this estimate to come up with a good approximately for the weight of the minimum spanning tree.

First idea. For each node u, let $n(u)$ be the number of nodes in the connected component containing u. For example, if the graph is connected, then for every node u, $n(u) = n$. If the graph has no edges, then for every node u, $n(u) = 1$.

Fix some connected component A. Notice that if we sum $1/n(u)$ for every node in A, then we get exactly 1:

$$\sum_{u \in A} \frac{1}{n(u)} = 1 \, .$$

Let $n' = n(u)$ for some $u \in A$. Every node v in A have the same value $n(v) = n'$, and there are exactly n' such nodes, so the summation reduces to $n'(1/n') = 1$.

Now if we sum across all the connected components, we get exactly the number of connected components:

$$\sum_{u \in V} \frac{1}{n(u)} = C \, .$$

Each connected component contributes 1 to the total count, and so the summation yields C, the number of connected components.

Hypothetical solution. Assume for now that, for each node u, we already knew the value $n(u)$. Our goal is to determine the approximate value of $\sum_{u \in V} \frac{1}{n(u)}$. We can accomplish this by sampling:

Algorithm 2: SimpleSampling$(G = (V, E), n, d, \epsilon)$

```
sum = 0.
repeat s times
    Choose a random u ∈ V.
    sum = sum + 1/n(u).
return n · (sum/s).
```

Here, we select a sample S consisting of s nodes and compute the average value of $n(u)$ for the nodes $u \in S$. We then argue that this is close to the average value of $n(u)$ for all nodes $u \in V$, and hence by multiplying this by n, we get the sum of all the $n(u)$'s.

We now fix $s = 4/\epsilon^2$. We want to show that the value returned C' is close to the correct number of connected components C:

Claim 5 *Let C' be the value returned by the algorithm. Then $|C' - C| \leq \epsilon n/2$ with probability at least $2/3$.*

Proof Let $S = u_1, u_2, \ldots, u_s$. Let Y_i be a random variable equal to $1/n(u_i)$. Notice that when the algorithm completes, $sum = \sum Y_i$. Each Y_i is a random variable in the range $[0, 1]$. Notice that $\mathrm{E}[Y_i] = (1/n)\sum_{u \in V} 1/n(u) = C/n$. Let $Y = \sum(Y_i)$. Then we also know that $\mathrm{E}[Y] = (s/n)C$.

We want to analyze:

$$\begin{aligned}\Pr[|C' - C| > \epsilon n/2] = \Pr[|(n/s)Y - (n/s)\mathrm{E}[Y]| > \epsilon(n/s)s/2] \\ = \Pr[|Y - \mathrm{E}[Y]| > \epsilon s/2]\end{aligned}$$

We can bound this probability using a Hoeffding bound, e.g., for any s independent Y_i in $[0, 1]$ where $Y = \sum Y_i$, we know that:

$$\Pr[|Y - \mathrm{E}[Y]| \geq t] \leq 2e^{-2t^2/s}.$$

In this case, we fix $t = \epsilon s/2$, and we conclude that:

$$\begin{aligned}\Pr[|Y - \mathrm{E}[Y]| > \epsilon s/2] &\leq 2e^{-2(\epsilon^2 s^2/4)/s} \\ &\leq 2e^{-\epsilon^2 s/2}\end{aligned}$$

Fixing $s = 4/\epsilon^2$, we see that this probability is at most $2e^{-2} < 1/3$. □

We have shown that if we knew the precise values $n(u)$ for every node, then we could efficiently in $O(1/\epsilon^2)$ time determine the approximate number of connected components.

Estimating the $n(u)$ values. Instead of using the precise values of $n(u)$ (which would be too expensive to compute), we are going to use approximate values that are good enough. Here is the key observation:

- If the component is large, then $n(u)$ is large and hence $1/n(u)$ is very small. Thus it contributes a negligible amount to the sum computed in the algorithm above.
- If the component is small, then we can afford to compute $n(u)$.

Specifically, we are going to use the same trick we did in determining if the graph G is connected: we are going to do a limited BFS, stopping if the connected component is too large. If the connected component is small, we find an exact value of $n(u)$. If the component is large, we stop and use an approximate value; the error will be negligible because $1/n(u)$ is sufficiently small.

For each node u, define $\tilde{n}(u)$ as follows:

- If $n(u) \leq 2/\epsilon$, then set $\tilde{n}(u) = n(u)$.
- Otherwise, set $\tilde{n}(u) = 2/\epsilon$.

Notice that $\tilde{n}(u) \leq n(u)$. We define:

$$error(u) = \frac{1}{\tilde{n}(u)} - \frac{1}{n(u)}$$

This is the error added to the calculation based on the fact that we did not compute $n(u)$ properly. We we want to find the number of connected components, we approximate $\tilde{C} = \sum_{u \in V} 1/\tilde{n}(u)$ instead of $C = \sum_{u \in V} 1/n(u)$. Thus, the

error involved is:

$$\begin{aligned} |\tilde{C} - C| &= |\sum_{u \in V} \frac{1}{\tilde{n}(u)} - \sum_{u \in V} \frac{1}{n(u)}| \\ &\leq \sum_{u \in V} |\frac{1}{\tilde{n}(u)} - \frac{1}{\tilde{n}(u)} \\ &= \sum_{u \in V} error(u) \end{aligned}$$

We have already noted that $error(u) \geq 0$ since $\tilde{n}(u) \leq n(u)$. We also notice that $error(u) \leq \epsilon/2$, since:

- Case 1: $n(u) \leq 2/\epsilon$.
$$error(u) = \frac{1}{\tilde{n}(u)} - \frac{1}{n(u)} = 0$$
- Case 2: $n(u) > 2/\epsilon$, and so $\tilde{n}(u) = 2/\epsilon$.
$$error(u) = \frac{1}{\tilde{n}(u)} - \frac{1}{n(u)} \leq \frac{1}{\tilde{n}(u)} = \epsilon/2$$

In either case, the maximum error is $\epsilon/2$. We thus conclude that the total error, summing over all the nodes u is:

$$\begin{aligned} |\tilde{C} - C| &= \sum_{u \in V} error(u) \\ &\leq \sum_{u \in V} \epsilon/2 \\ &\leq \epsilon n/2 \end{aligned}$$

That is, if we compute $\tilde{C}$, we will be within $\epsilon/2$ of the correct number of connected components.

Final algorithm. Putting the pieces together, we get the following algorithm:

Algorithm 3: ConnectedComponents$(G = (V, E), n, d, \epsilon)$

$sum = 0$.
repeat s times
    Choose a random $u \in V$.
    Perform a BFS from u, stopping as soon as $2/\epsilon$ new nodes are discovered.
    if *u is in a connected component with $\leq 2/\epsilon$ nodes* **then** $sum = sum + 1/n(u)$.
    if *u is in a connected component with $> 2/\epsilon$ nodes* **then** $sum = sum + \epsilon/2$.
return $n \cdot (sum/s)$.

Fixing $s = 4/\epsilon^2$, we first analyze the running time:

Claim 6 *The connected component algorithm runs in time $O(d/\epsilon^3)$.*

Proof Each iteration of the algorithm does a BFS that explores up to $2/\epsilon$ new nodes. Each node explored requres examining a neighbor list of size d. Thus the cost of each iteration is $2d/\epsilon$. Since there are $4/\epsilon^2$ iterations, we conclude that the total cost is $8d/\epsilon^3$. □

We next argue that it is correct. Let C' be the value returned by the algorithm. We have already shown that $|\tilde{C} - C| \leq \epsilon n/2$. It remains to show that $|C' - \tilde{C}| \leq \epsilon n/2$, with probability at least $2/3$, which will conclude the analysis.

The sampling argument is identical to that above, replacing $n(u)$ with $\tilde{n}(u)$. Let $S = u_1, u_2, \ldots, u_s$. Let Y_i be a random variable equal to $1/\tilde{n}(u_i)$. Notice that when the algorithm completes, $sum = \sum Y_i$. Each Y_i is a random variable in the range $[0, 1]$. Notice that $\mathrm{E}[Y_i] = (1/n)\sum_{u \in V} 1/\tilde{n}(u) = \tilde{C}/n$. Let $Y = \sum(Y_i)$. Then we also know that $\mathrm{E}[Y] = (s/n)\tilde{C}$.

We want to analyze:

$$\begin{aligned}\Pr\left[|C' - \tilde{C}| > \epsilon n/2\right] &= \Pr[|(n/s)Y - (n/s)\mathrm{E}[Y]| > \epsilon(n/s)s/2] \\ &= \Pr[|Y - \mathrm{E}[Y]| > \epsilon s/2]\end{aligned}$$

We can bound this probability using a Hoeffding bound, fixing $t = \epsilon s/2$. We conclude that:

$$\begin{aligned}\Pr[|Y - \mathrm{E}[Y]| > \epsilon s/2] &\leq 2e^{-2(\epsilon^2 s^2/4)/s} \\ &\leq 2e^{-\epsilon^2 s/2}\end{aligned}$$

Fixing $s = 4/\epsilon^2$, we see that this probability is at most $2e^{-2} < 1/3$.

We have thus shown the following claim:

Claim 7 *Fix a graph $G = (V, E)$. Let C' be the value returned by the algorithm, and let C be the number of connected components in G. Then with probability at least $2/3$, $|C' - C| \leq \epsilon n$. The running time is $O(d/\epsilon^3)$.*

Boosting the probability. Assume, instead that we wanted a smaller probability of error, e.g., a specific error probability of $\delta < 1$. Notice that we can readily achieve this smaller error simply be adjusting the sample size s.

Recall that the sampling procedure guarantees a probability of error of $2e^{-\epsilon^2 s/2}$. In this case, we fix $s = 2\ln(2/\delta)/\epsilon^2$. We can then compute the error:

$$\begin{aligned}2e^{-\epsilon^2 s/2} &= 2e^{-\epsilon^2(2\ln(2/\delta)/\epsilon^2)/2} \\ &= 2e^{-\ln(2/\delta)} \\ &= 2(\delta/2) \\ &= \delta\end{aligned}$$

The resulting algorithm runs in time $O(d\log(1/\delta)/\epsilon^3)$. If, for example, you want the (approximately) correct answer with high probability (i.e., with probability at least $1 - 1/n^c$, for some constant c), then you modify the algorithm to run in time $O(d\log n/\epsilon^3)$.

4 Approximate Minimum Spanning Tree

We now turn our attention to the problem of finding a minimum spanning tree. We clearly cannot actually find the entire spanning tree in $o(n)$ time, since it takes $n - 1$ time just to write down a spanning tree (i.e., to produce the output). Instead, our goal is to find the *approximate weight* of the minimum spanning tree.

We are given a graph $G = (V, E)$ where each edge $e \in E$ has a weight $w(e) \in [1, W]$. Notice that we are assuming that weights are integers in the range of 1 to W. We will also assume that the graph has maximum degree d. Finally, we assume that the graph G is connected, i.e. there is a spanning tree.

Special case. To get some intuition, let us think of the special case where every edge has weight 1 or 2. Let G_1 be the subset of G containing only edges of weight 1. Let C_1 be the number of connected components in G_1.

Notice that the MST must have at least $(C_1 - 1)$ edges of weight 2. That is, there must be edges to connect all the connected components in G_1.

Since there are, in total, $n - 1$ edges in an MST, we can conclude that there must be $(n-1) - (C_1 - 1) = (n - C_1)$ edges of weight 1 in the tree.

We conclude that the weight of the MST is $(n - C_1) + 2(C_1 - 1) = (n - 2) + C_1$. Thus if we can calculate the number of connected components in G_1, we can immediately find the weight of the minimum spanning tree.

General case. More generally, let G_i be the graph containing all the edges in G with weight $\leq i$. Let C_i the number of connected components in C_i. We can now relate the number of connected components in graphs C_i to the edges in the MST:

Claim 8 *For all $i \in [1, W]$, the MST for G contains $C_i - 1$ edges of weight $> i$.*

That is, the MST must contain edges to connect the different components in C_i, and all of those edges have weight $> i$.

Using this fact, we can calculate the weight of the MST as a function of the C_i.

Claim 9 *The weight of the MST of G is $n - W + \sum_{i=1}^{W-1} C_i$*

Proof First, we observe that the number of edges of weight 1 is $(n - C_1)$. In general, the number of edges of weight $i + 1$ is $(C_i - 1) - (C_{i+1} - 1) = (C_i - C_{i+1})$. Thus, the weight of the MST is:

$$(n - C_1) + \sum_{i=1}^{W-1} (i + 1)(C_i - C_{i+1})$$

Notice that this sum telescopes, i.e., it is equal to:

$$\begin{aligned}(n - C_1) + (2C_1 - 2C_2) + (3C_2 - 3C_4) + (4C_4 - 4C_5) \ldots \quad &= \\ n + C_1 + C_2 + \ldots + C_{W-1} - WC_W\end{aligned}$$

Notice that $C_W = 1$, i.e., the entire graph is connected. So this reduces to $n - W + \sum_{i=1}^{W-1} C_i$. □

MST approximation algorithm. We now know how to calcualte the weight of an MST, assuming we can compute the number of connected components in a graph. And we have already (in the previous section) developed an algorithm for approximating the number of connected components. This yields the following algorithm:

Algorithm 4: ApproxMST$(G = (V, E), n, d, w, \epsilon)$

```
weight = n − W.
for i = 1 to W − 1
    Let x_i = ConnectedComponents(G, n, d, ε/W).
    weight = weight + x_i.
return weight.
```

The first question to determine is the probability of success. If the ConnectedComponents algorithm succeeds with probability $2/3$, then the probability that all the iterations succeed is $(2/3)^W$, i.e., very small. Instead, we will use the variant of ConnectedComponents that succeeds with probability $\delta = 1 - (1/3W)$. Now, the probability that any iterations fails is $1/3$. With probability at least $2/3$, all the iterations succeed.

We now examine the approximation ratio in the case where all the iterations succeed. Notice that the guarantee of the connected component algorithm is that $|C_i - x_i| \leq \epsilon n/W$. Therefore when we compute the sum of the x_i:

$$\begin{aligned}\sum_{i=1}^{W-1} x_i &\leq \textstyle\sum_{x=1}^{W-1}(C_i + \epsilon n/W) \\ &\leq [\textstyle\sum_{i=1}^{W-1} C_i] + \epsilon n\end{aligned}$$

Similarly, we observe that $\sum_{i=1}^{W-1} x_i \geq [\sum_{i=1}^{W-1} C_i] - \epsilon n$. Hence, since the weight of the MST is $n - W + \sum_{i=1}^{W-1} C_i$, and the algorithm returns $n - W + \sum_{i=1}^{W-1} x_i$, we conclude that:

$$MST - \epsilon n \leq weight \leq MST + \epsilon n$$

Therefore the algorithm returns a weight that is within an additive ϵn of the real MST weight.

Finally, we notice that the weigth of the MST is $\geq n - 1 \geq n/2$. So we can transform the additive approximation into a multiplicative approximate. That is:

$$MST - \epsilon(2MST) \leq MST - \epsilon n$$

That is, $MST(1 - 2\epsilon) \leq weight$. Similarly, on the other side:

$$MST + \epsilon n \leq MST + \epsilon(2MST)$$

And so, again, we conclude that $weight \leq MST(1 + 2\epsilon)$.

From this we conclude that:

Claim 10 *With probability at least $2/3$, the $weight$ returned satisfies the following property:*

$$MST(1 - 2\epsilon) \leq weight \leq MST(1 + 2\epsilon)$$

The final issue to consider is performance. We have already shown that each invocation of the ConnectedComponents algorithm runs in time $O(d \log(W)/(\epsilon/W)^3) = O(dW^3 \log(W)/\epsilon^3$. Thus the final running time is:

Claim 11 *The algorithm runs in time $O(dW^4 \log(W)/\epsilon^3$.*

Notice that there is no dependency on n, but there is a strong dependency on W. You might wonder if we can do better. It turns out there is a lower bound:

Theorem 12 *Any algorithm for finding an approximate MST within a $(1 \pm \epsilon)$ factor requires at least $\Omega(dW/\epsilon^2)$ time.*

The best known algorithm has a running time of $O((dW/\epsilon^2) \log (dW/\epsilon))$. (See "Approximating the minimum spanning tree weight in sublinear time" by Chazelle, Rubinfeld, and Trevisan.)

5 Degree estimation

Given a graph $G = (V, E)$, we want to estimate the average degree via sampling. We will consider the following basic sampling algorithm:

Algorithm 5: AverageDegree($G = (V, E), n, \epsilon$)

1 $min = n$
2 **repeat** k times
3 $\quad total = 0$
4 $\quad$ **repeat** s times
5 $\quad\quad$ Choose a random $u \in V$.
6 $\quad\quad total = total + degree(u)$
7 $\quad$ **if** $total/s < min$ **then** $min = total/s$
8 **return** min.

In each iteration (lines 3–6), the algorithm chooses a sample of s nodes, and calculate the average degree of the nodes in the sample. To find the final answer, we repeat k times and return the minimum value.

It is, in fact, quite surprising that this works. Imagine that I gave you a vector $D = [d_1, d_2, \ldots, d_n]$. Now, you sample from this vector s elements and return their average. If $s = o(n)$, it will not give you a good approximation of the average! Consider, for example, the vector that is $D = [n, n, n, n, 0, \ldots, 0]$. The average value is $4n/n = 4$. However, if you sample $s = o(n)$ elements, you are very likely to return an average of 0. So if you are sampling from an array, you need a very large sample to get a good estimate of the average value

However, we are not sampling from an array. We are sampling from a graph, and a graph has special structure. For example, you can't have the situation above where 4 nodes have degree n and the rest have degree 0. We can leverage this structure of a graph to find a 2-approximation of the degree in $O(\sqrt{n})$ time.

Setup. Fix a graph $G = (V, E)$ with n nodes and m edges. Assume G is connected, and that $m \geq n$. Let d be the average degree of the graph. Define $degree(v)$ to be the degree of node v. As a basic fact, we know that $d = 2m/n$, since $\sum_{v \in V}(degree(v)) = 2m$ (i.e., each edge is counted twice).

We also will fix an ϵ. Our goal is to show that the sampling procedure above, for a proper choice of s, gives a $(1/2 - \epsilon)$-approximate answer.

We focus on analyzing one iteration of the algorithm, i.e., one execution of lines 3–6. Let x_i be the degree of the ith node in the sample. Let $X = \sum_i (x_i)$. We can calculate the expected value of each x_i:

$$\mathrm{E}[x_i] = \sum_{j=1}^{n} \frac{1}{n} degree(i) = 2m/n = d$$

That is, the expected value of each x_i is d. Hence $\mathrm{E}[X] = sd$, and the expected outcome of the algorithm is d.

Can the estimate be too big? We first show that the estimate is unlikely to be too big. This is a simple application of Markov's Inequality. We know that:

$$\Pr[X \geq (1+\epsilon)\mathrm{E}[X]] \leq \frac{\mathrm{E}[X]}{(1+\epsilon)\mathrm{E}[X]} \leq \frac{1}{1+\epsilon}$$

Recall the following two basic facts, when $x < 1$:

$$\begin{aligned} (1+x)^{-1} &\geq 1 - x \\ (1+x)^{-1} &\leq 1 - x/2 \end{aligned}$$

Both of these follow from the Taylor expansion of $(1+x)^{-1} = 1 - x + x^2/2 - \ldots$.

From this, we conclude that the probability that $X \geq (1+\epsilon)sd$ is at most $(1+\epsilon)^{-1} \leq (1 - \epsilon/2)$.

Now, if we choose $k = 8/\epsilon$, we see that the probability that *all* k iterations find a total $\geq (1+\epsilon)sd$ is $\leq (1-\epsilon/2)^{8/\epsilon} \leq e^{-4} \leq 1/8$.

We conclude that with probability at least $7/8$, at least one of the iterations returns a total that is $\leq sd(1+\epsilon)$, and hence after dividing by s, the algorithm returns a value that is $\leq d(1+\epsilon)$.

Claim 13 *For $k = 8/\epsilon$, the probability that the algorithm returns a value larger than $d(1+\epsilon)$ is $\leq 1/8$.*

Notice that so far, we have not used anything about the structure of the graph. This claim depends only on Markov's Inequality.

Can the estimate be too small? The tricky part of the analysis is showing that the estimate cannot be too small. For this, we partition the graph into "high degree" and "low degree" nodes.

Let H be the $\sqrt{\epsilon n}$ nodes with the highest degree, and let L be the remaining nodes in the graph. Let $deg(L) = \sum_{u \in L} degree(L)$ be the sum of the degrees of the nodes in L, and let $deg(H) = \sum_{u \in H} degree(H)$ be the sum of the degrees of the nodes in H.

We now argue that at least half of the edges have an endpoint in L (within a $(1-\epsilon)$ factor):

Claim 14 $deg(L) \geq m(1-\epsilon)$

Proof We show that $deg(H)$ must be small. Notably, let E_H be all the edges that have two endpoints in H. Let E_C be all the edges that have one endpoint in H and one endpoint in C. Notice that $deg(H) = 2|E_H| + |E_C|$.

Since there are only $\sqrt{\epsilon n}$ nodes in H, there can only be $\binom{\sqrt{\epsilon n}}{2}$ edges in E_H. That is, $E_H \leq \sqrt{\epsilon n} \cdot (\sqrt{\epsilon n} - 1)/2 \leq \epsilon n/2$.

Also, we know that $|E_C| \leq m$, since m is an upper bound on the total number of edges in the graph.

Putting these facts together, we conclude that $deg(H) \leq 2(\epsilon n/2) + m \leq \epsilon n + m$. Finally, since $n \leq m$ (by assumption), this implies that $deg(H) \leq \epsilon m + m \leq m(1+\epsilon)$.

Now, we can compute $deg(L)$. Notice that $deg(L) + deg(H) = 2m$, and so we can find $deg(L) = 2m - deg(H) \geq 2m - m(1+\epsilon) \geq m - m\epsilon = m(1-\epsilon)$. □

This is useful: if we can show that the sampling gives us a good estimate of the average degree in L, then we immediately have found a 2-approximation of a good estimate of d.

Ideally, we would like to sample only from L, ignoring samples from H. That would immediately yield an estimate of the degree of L. Unfortunately, we don't know whether a node is in L or H. Instead, we analyze a related random process.

Let M be the maximum degree of any node in L. We define $y_i = \min(x_i, M)$. That is, the maximum value of y_i is M. If x_i is $\leq M$, i.e., if $i \in L$, then we set $y_i = x_i$. Otherwise, if $x_i \in H$, we simply set $y_i = M$. Notice, of course, that this is only for the purpose of analysis, as we do not know the value of M.

We now compute the expected value of each y_i:

$$\begin{aligned}
\mathrm{E}[y_i] &= \sum_{j=1}^{n} \frac{1}{n} \min(degree(i), M) \\
&= \sum_{u \in L} \frac{1}{n} \min(degree(u), M) + \sum_{u \in H} \frac{1}{n} \min(degree(u), M) \\
&\geq \sum_{u \in L} \frac{1}{n} \min(degree(u), M) \\
&\geq \frac{1}{n} \sum_{u \in L} degree(u) \\
&\geq \frac{1}{n} deg(L) \\
&\geq \frac{m}{n}(1-\epsilon) \\
&\geq \frac{d}{2}(1-\epsilon)
\end{aligned}$$

That is, the expected value of each y_i is at least as large as the average degree of the nodes in L, and hence is at least $1/2$ of the degree of the graph. We can also bound the the expected value of each y_i in terms of the average degree of the nodes in H:

$$\begin{aligned}
\mathrm{E}[y_i] &= \sum_{j=1}^{n} \frac{1}{n} \min(degree(i), M) \\
&= \sum_{u \in L} \frac{1}{n} \min(degree(u), M) + \sum_{u \in H} \frac{1}{n} \min(degree(u), M) \\
&\geq \sum_{u \in H} \frac{1}{n} \min(degree(u), M) \\
&\geq \frac{1}{n} |H| M \\
&\geq \frac{M}{\sqrt{n/\epsilon}}
\end{aligned}$$

We want to show that $\mathrm{E}[Y] = \mathrm{E}[\sum_i y_i]$ is at least $(sd/2)(1-\epsilon)(1-\delta)$, for some δ. We will show that

$$\Pr[Y \leq (1-\delta)\mathrm{E}[Y]]$$

is small, and since $\mathrm{E}[Y] \geq (sd/2)(1-\epsilon)$, that also means that:

$$\Pr[Y \leq (1-\delta)(1-\epsilon)(sd/2)]$$

is small. (If a random variable is unlike to be less than some value z, then it is at least as unlikely to be less than some smaller value $z' \leq z$.)

For this, we apply a Chernoff bound:

$$\Pr[Y < (1-\delta)\mathrm{E}[Y]] \leq e^{-\mathrm{E}[Y]\delta^2/(2M)}$$

Notice that since the y_i are not $0/1$ random variables, we have to use the version of a Chernoff bound that works for random variables in the range $[0, M]$. See the exercises on Problem Set 1.

We now simplify to show that this is small:

$$\begin{aligned} e^{-\mathrm{E}[Y]\delta^2/(2M)} &\leq e^{-(sM/\sqrt{n/\epsilon})\delta^2/(2M)} \\ &\leq e^{-(s/\sqrt{n/\epsilon})\delta^2/(2)} \end{aligned}$$

We choose $s = 6k\sqrt{n/\epsilon}/\delta^2$, from which we conclude:

$$e^{-\mathrm{E}[Y]\delta^2/(2M)} \leq e^{-3k} \leq 1/(8k)$$

From this we conclude that, throughout all k iterations (via a union bound), with probability at least $7/8$, $Y \geq (1-\delta)(1-\epsilon)(sd/2)$.

Notice that $X \geq Y$, and so we can also conclude that with probability at least $7/8$, $X \geq (1-\delta)(1-\epsilon)(sd/2)$. This implies that the result of the algorithm approaches a two approximation.

Claim 15 *With probability at least $7/8$, the value output by the algorithm is at least $(1-\delta)(1-\epsilon)(d/2)$.*

The total running time is $ks = (3/\epsilon)(3/\epsilon)(6\sqrt{n/\epsilon}(1/\delta^2)$. Setting $\delta = \epsilon$:

Claim 16 *The algorithm runs in time $O(\sqrt{n}\left(\frac{1}{\epsilon^{9/2}}\right))$.*

Noting that $(1-\epsilon)^2 \geq (1-2\epsilon)$, it guarantees:

Claim 17 *The algorithm outputs a value $\tilde{d}$ such that $d(1/2-\epsilon) \leq \tilde{d} \leq d(1+\epsilon)$ with probability at least $3/4$.*

Feige has shown how to get a better dependence on ϵ. (See Feige, "On sums of independent random variables with unbounded variance and estimating the average degree in a graph.") However, it is impossible to do better than, roughly, $\Omega(\sqrt{n/d})$ queries in a model where you can only ask each node, "How many neighbors do you have?" However, in the model where you can ask for the identity of the ith neighbor (instead of just the number of neighbors), then you can do better, getting a $(1+\epsilon)$ approximation using (again) approximately $O(\sqrt{n})$ queries (modulo polylog factors). (See Goldreich and Ron, "Approximating average parameters of graphs.")

6 Diameter of a Graph

We now consider the problem of deciding whether a graph has small diameter. Given a connected, unweighted graph $G = (V, E)$ with n nodes and m edges, we can certainly perform a BFS from every node to determine the exact diameter of the graph in $O(m^2)$ time. In this section, our goal is to do something faster.

In particular, we want to distinguish between a graph that really does have small diameter, and a graph that is *far* from a graph with small diameter. Here, we say that a graph G is ϵ-far from a graph G' if we need to add or delete $\lceil \epsilon n \rceil$ edges from G to transform it into G'. (Notice that this is a slightly different definition than used above.)

Given a value D, our goal in this problem is as follows:

- If the graph has diameter $\leq D$, return true.
- If the graph is ϵ-far from any graph with diameter $\leq 4D+2$, return false with probability at least $2/3$.

The running time of the algorithm should be $O(1/\epsilon^3)$. Notice we are not going to assume anything about the degree of the graph.

As in the case of testing connectivity, we want to use sampling, along with local exploration, to decide if the graph has diameter $> 4D + 2$. We want to find a property that is true of every graph with small and that is easy to check quickly. Consider the following property:

Property 18 ((k, C)-friendly) *We say that a graph is (k, C)-friendly if for every node $u \in V$, there are at least k other nodes within distance C of u.*

Claim 19 *If graph G is (k, D)-friendly, then it is within distance $1/k$ of a graph with diameter at most $4D + 2$.*

Proof For every node in the graph, we are going to label it as either: (i) a center, (ii) a friend, or (iii) a fringe. We are going to do this greedily, by repeating the following process:

- Choose a node u in the graph that is not labeled. Label it a center.
- Label every node within distance D of u as a friend of u (even if it was previously labeled as a fringe of someone else).
- Label every node within distance $2D$ of u that is not yet labeled as a fringe of u.
- Delete u and all the friends of u from the graph.

When this procedure completes, all the nodes are labeled. Notice that in each iteration of the procedure, we remove at least k nodes from the graph: We know that node u has at least k nodes within distance D in the original graph (by assumption, since the graph is (k, D)-friendly). We also know that node u is not within distance $2D$ of any previously chosen center, since otherwise it would have been labeled a fringe and not selected. Thus, all the nodes within distance D of u are of distance at least D from any previously chosen center, by the triangle inequality. Since the only nodes removed from the graph are within distance D of a center, we conclude that none of the nodes within distance D of u have been removed, and hence at least k of them remain in the graph.

Since at least k nodes are removed in each iteration, the entire procedure completes in at most $\lceil n/k \rceil$ iterations. We will now add at most ϵn edges to the original graph G to produce a graph with diameter at most $4D + 2$.

First, choose a center u arbitrarily. Now, add an edge connecting every center to u. Notice that this requires adding at most $\lceil n/k \rceil$ edges, and hence the original graph is ϵ-close to the new graph. Next, notice that every node is within distance $2D$ of its center, whether it is a friend or a fringe of that center. Thus every node is within distance $2D + 1$ of u. Finally, given a pair of nodes u and v, since each of them are within distance $2D + 1$ of u, then u and v are within distance $4D + 2$ of each other. □

Property 20 ((k, C)-friendly) *We say that a graph is almost (k, c)-friendly if for at least $(1 - 1/k)n$ nodes $u \in V$, there are at least k other nodes within distance C of u.*

Claim 21 *If graph G is almost (k, D)-friendly, then it is within distance $2/k$ of a graph with diameter at most $4D + 2$.*

Proof There are at most n/k nodes that do not have at least k nodes within distance D. The remainder of the graph G' (constructed by removing these at most n/k bad nodes) is (k, C)-friendly, and hence by adding at most $\lceil n/k \rceil$ edges, we can modify G' to have diameter at most $4D + 2$. Recall that our construction identified a special center u that was within distance $2D + 1$ of every node.

Now, we can add another (at most) n/k edges connecting the bad nodes (previously excluded) to the special center u. Each of these nodes is now within distance 1 of u, and hence within distance $2D + 2 \leq 4D + 2$ of any other node.

We have now shown that graph G is within distance $2/k$ of a graph with diameter at most $4D + 2$. □

We can now describe the algorithm. Our goal is to decide whether or not the graph is almost $(2/\epsilon, D)$-friendly. If it is almost $(2/\epsilon, D)$-friendly, we return true. Otherwise, we return false.

We determine whether a graph is almost $(2/\epsilon, D)$-friendly by sampling. We choose a node u at random, and perform a BFS to decide whether there are at least $2/\epsilon$ nodes within distance D. This takes time at most $O(1/\epsilon^2)$: notice that when we are performing the BFS, a node can visit at most $2/\epsilon$ neighbors that have previously been visited, since we are going to visit only $2/\epsilon$ nodes in total! If we find $2/\epsilon$ nodes within distance D, we stop and label this a good node. Otherwise, we label it a bad node.

We repeat this sampling procedure several times. If we find only good nodes, then we return true. If we ever find a bad node, then we return false.

Notice that if the graph has diameter $\leq D$, then every node must be good. Hence, we always return true. Conversely, assume that the graph is not ϵ-close to a graph with diameter $\leq 4D + 2$. This implies that graph G is *not* almost $(2/epsilon, D)$-friendly, i.e., there must be at least $\epsilon n/2$ bad nodes. Thus, each query has a probability of $\epsilon/2$ of finding a bad node. The probability that we perform $4/\epsilon$ iterations and yet do not find a bad node is at most:

$$(1 - \epsilon/2)^{4/\epsilon} \leq e^{-2} \leq 1/3$$

Thus, with probability at least $2/3$ we will find a bad node and report false.

The total running time, then, is $(4/\epsilon)(4/\epsilon^2) = O(1/\epsilon^3)$.

7 Maximal Matching

Given a graph $G = (V, E)$, a matching is a collection of edges $M \subseteq E$ such that no two edges in M share an endpoint. A *maximum* matching is the largest possible matching for a graph. A *maximal* matching is a matching that cannot be made any larger, i.e., for which every edge in $E \setminus M$ shares an endpoint with some edge in M.

We will assume the graph is represented as an adjacency list, and that we know n and m. We will also assume that the graph has maximal degree d.

Our goal is to give an algorithm for finding the approximate size of a maximum matching, or the approximate size of a maximal matching, in time $O(e^d/\epsilon^2)$. A better analysis of the algorithm presented actually shows that the running time is $O(d^4/\epsilon^2)$.

Maximal vs. maximum matchings.

Theorem 22 *A maximal matching M is at least $1/2$ the size of a maximum matching, i.e., it is a 2-approximation.*

Proof Let M be a maximal matching and let M' be a maximum matching. For every edge $e = (u, v) \in M'$, we know that either u or v has an adjacent edge in M. (If not, we could have added the edge e to M and so M would not be maximal.) Let e' be an edge adjacent to either u or v in M', and assign e' to cover e. When we are done, every edge in M' is covered by one edge in M.

Next, observe that each edge in M covers at most two edges in M'. To see this, fix an edge $(w, z) \in M$. It can only cover edges that are adjacent to w and z. And there is at most one edge in M' adjacent to w and at most one edge in M' adjacent to z. Hence we conclude that (w, x) covers at most two edges in M'.

Putting these two facts together, we conclude that $|M| \geq |M'|/2$: otherwise, it would be impossible for the edges in M to cover all the edges in M' given that each edge in M can only cover two edges in M'. □

Simple Algorithm. A maximal matching is, essentially, trivial to find: iterate through all the edges, and add each edge to the matching if you can. A more precise implementation of this idea is as follows:

- Assign each edge a random real number between $[0, 1]$. (This could be done with a hash function.)
- Sort the edges according to their assigned number from smallest to largest.
- Iterate through the edges, in order from smallest to largest, and add each edge $e = (u, v)$ to the matching M if there are no edges already in M with endpoints u or v.

It is easy to see that the resulting matching is maximal, since we have added every edge we can.

Notice that given an ordering of edges (determined by the random numbers), this determines exactly one maximal matching.

Sampling. Assume we have fixed the random numbers assigned to the edges and let M be the maximal matching that results from that ordering. And assume that we can query an edge and ask, "is edge $e \in M$?" We want to design a sampling algorithm that looks at only a few edges and determines the size of M. (Later we will show how to implement the algorithm to answer these queries.) We will see how, using random sampling, we can estimate the size of M, and hence can estimate the size of a maximum matching within a factor of 2.

Assume we choose a sample of s nodes to query. For each node that we query, check all the adjacent edges and see if any of them are in the matching M. Let x_i be the random variable equal to 1 if the ith node sampled is adjacent to an edge in M, and 0 otherwise. Let $X = \sum(x_i)$. The algorithm returns $(1/2)(X/s)n$, where n is the total number of nodes in the graph. Notice that $\mathrm{E}[x_i] = 2|M|/n$. Using:

$$\Pr[|X - 2s|M|/n| > 2\epsilon s] \leq e^{-4\epsilon^2 s^2/s}$$

Thus, by choosing $s = \Theta(1/\epsilon^2)$, we see that the error is at most ϵn, i.e., the estimated size of the maximal matching is $|M| \pm \epsilon n$.

(Why is it hard to solve this problem efficiently by sampling if you want a multiplicative error? Think about the star graph.)

Deciding if a node is matched. We now need a method for deciding if a node has an adjacent edge in the matching. Imagine the sampling algorithm asks us about node u: "is any edge e adjacent to u in the matching M?"

Thus, you can imagine the following algorithm: for each edge e' adjacent to u, determine whether e' is in the matching. To determine whether an edge e' is in the matching, recursively check whether any edge adjacent to edge e' is in the matching. Unfortunately, this procedure will not perform well: the recursion never halts!

However, we do not need to check *all* the edges adjacent to u. And to determine if an edge e is in the matching, we do not need to check *all* the edges adjacent to e. Remember, we have assigned random values (e.g., weights) to each edge, and when constructing the matching M, we considered the edges in order from smallest value to largest. Assume we are trying to check edge e. Let $w(e)$ the value assigned to edge e. Any edge e' with a value larger than $w(e)$ can be safely ignored: edge e' is processed after e in constructing the matching, and hence whether or not e is in the matching does not depend on e' at all!

Therefore, we can suggest a modified (recursive) algorithm for determining whether e is in the matching: for each edge e' adjacent to u or v where $w(e') < w(e)$, recursively determine whether e' is in the matching; add e to the

matching if none of the recursively queried edges are in the matching. Notice that this algorithm now terminates: there is a total ordering of edges, and eventually we will arrive at a node that has no adjacent edges of smaller value.

Analysis. We want to show that the expected cost of querying an edge is at most $O(e^d)$. Assume we are processing a query of edge e. We want to know, in expectation, how many edges at distance k from e do we explore? We can then (via linearity of expectation) sum up over all k to get a bound on the total expected number of edges visited.

First, notice that there are at most d^k paths of distance k that start at node u and terminate at some node v of distance k from e. For each of these paths, we want to calculate the probability that we explore that path. (Notice that many of these paths may overlap; that is okay.) Clearly, we can bound the expected number of edges we explore at distance k by bounding the expected number of paths of length k (since for each explored edge, we must have followed a path to get there).

Consider some path P from node u to node v at distance k from e. What is the probability that we follow path P and explore all the edges along the way?

Notice that we only explore edge e' if the weights of the edges are strictly decreasing on the path from u to v. (Remember, we only recursively continue exploring on edges of smaller weight.) Notice that the weights on the edges in the path P determine a permutation, i.e., an ordering of the edges. There are $k!$ possible orderings, only one of which results in the algorithm exploring path P, i.e., the ordering in which the weights are deceasing. Hence, we conclude that path P is explored with probability at most $1/k!$.

We conclude that the expected number of paths of length k explored is at most $d^k/k!$. Thus, the total expected cost is:

$$\sum_{k=1}^{\infty} \frac{d^k}{k!} = O(e^d)$$

We will not prove this bound on the series, but you can get some sense for it by using Sterling's approximation (i.e., $d^k/k! \approx (ed/k)^k$), and noticing that all the terms fall off rapidly from the largest term in this sequence where $k = d$.

We conclude that the expected cost of each query is $O(e^d)$.

Since the matching algorithm queries $O(1/\epsilon^2)$ queries, the overall expected running times is $O(e^d/\epsilon^2)$.

Improvements. It turns out there is one easy improvement: from each edge, you should explore the neighboring edges in *increasing* order of weight. By looking at smaller weight edges first, you increase the likilihood that you will find an edge that is already in the matching and hence can abort the remainder of the search. It turns out that with this one improvement (and a fair bit of technical analysis), you can improve the running time to $O(d^4/\epsilon^2)$.