CS5234: Combinatorial and Graph Algorithms

Problem Set 5

Due: September 29st, 6:30pm

Instructions. This problem set focuses on sublinear time graph algorithms, particularly the problems of finding an approximate minimum spanning tree and the approximate number of connected components. First, we explore why we need $\Omega(W)$ time to find the weight of the minimum spanning tree. Then, we look at the question of developing a faster algorithm for finding connected components (and hence also for finding the weight of the minimum spanning tree) that depends only on the *average* degree, not the *maximum* degree.

- Start each problem on a separate page.
- Make sure your name is on each sheet of paper (and legible).
- Staple the pages together, and hand it in before class starts, or submit it on IVLE in the workbin. Alternatively, if you submit it late, you can put it in the envelope next to my office door (and send me an e-mail).

Remember, that when a question asks for an algorithm, you should:

- First, give an overview of your answer. Think of this as the executive summary.
- Second, describe your algorithm in English, giving pseudocode if helpful.
- Third, give an example showing how your algorithm works. Draw a picture.

You may then give a proof of correctness, or explanation, of why your algorithm is correct, an analysis of the running time, and/or an analysis of the approximation ratio, depending on what the question is asking for.

Advice. Start the problem set early—some questions take time. Come talk to me about the questions. (Different students have different questions. Some have questions about how to write a good proof. Others need pointers of designing an algorithm. Still others want to understand the material from lecture more deeply before applying it to the problem sets.) I'm here for you to talk to.

Collaboration Policy. The submitted solution must be your own unique work. You may discuss your high-level approach and strategy with others, but you must then: (i) destroy any notes; (ii) spend 30 minutes on facebook or some other non-technical activity; (iii) write up the solution on your own; (iv) list all your collaborators. Similarly, you may use the internet to learn basic material, but do not search for answers to the problem set questions. You may not use any solutions that you find elsewhere, e.g. on the internet. Any similarity to other students' submissions will be treated as cheating.

Exercises and Review (*Do not submit.*)

Exercise 1. Imagine you have an array $A[1\,..\,n]$. Each value in the array is an integer between 1 and M. Consider the following algorithm for finding the approximate sum of the values in the array: Fix $s = 12/\epsilon$. Dante claims that this is a correct algorithm, and gives the following proof:

Algorithm 1: Sum(A, n, s)

```
1 sum = 0
2 repeat s times
3     Choose a random i ∈ [1, n].
4     sum = sum + A[i].
5 return n(sum/s)
```

Let x_i be the value of the ith random sample, and $X = \sum(x_i)$. Let $A = \sum_i A[i]/n$, i.e., A is the average value and nA is the sum of the array (i.e., what we want to find). We know that $\mathrm{E}[X] = sA$. Then:

$$\Pr[|X - \mathrm{E}[X]| \geq \epsilon \mathrm{E}[X]] = \Pr[|X - sA| \geq \epsilon sA] \leq 2e^{-sA\epsilon^2/3} .$$

We know that $A \geq 1$, so by choosing $s \geq 12/\epsilon^2$, we conclude that the probability of error is $\leq 2e^{-4} \leq 1/3$. And, when there is no error, $|X - sA| < \epsilon sA$, which implies that: $|nX/s - nA| < \epsilon nA$, i.e.,

$$nA(1-\epsilon) \leq sum \leq nA(1+\epsilon) .$$

We thus conclude that the algorithm returns a $(1 \pm \epsilon)$ estimate of the sum of the values in the array. What is wrong with this algorithm and proof?

Exercise 2. Consider the following algorithm for estimating the number of edges in a connected graph $G = (V, E)$: Let x_i be the random variable representing the ith pair (u, v) selected, where

Algorithm 2: Edges$(G = (V, E), n, s)$

```
1 sum = 0
2 repeat s times
3     Choose a random u ∈ [1, n], v ∈ [1, n].
4     if there is an edge (u, v) ∈ E then sum = sum + 1
5 return (sum/s)C(n, 2).
```

$x_i = 1$ if $(u, v) \in E$. Let $X = \sum(x_i$. Notice that $\mathrm{E}[x_i] = m/\binom{n}{2}$ (where m is the actual number of edges in the graph), and $\mathrm{E}[X] = sm/\binom{n}{2}$. What happens if you try to apply a Chernoff Bound or a Hoeffding Bound to show that the result is a good estimate of the number of edges in the graph? Think about different types of graphs, i.e., both dense and sparse graphs.

Standard Problems (to be submitted)

Problem 1. Can we go faster?

In class we saw an algorithm for finding the approximate weight of the minimum spanning tree of a graph G in time $O(dW^4 \log W/\epsilon^3)$, assuming the graph has maximum degree d and maximum edge weight W. And there exists an improved algorithm that we did not see in class that runs in time $O((dW/\epsilon^2)\log(dW/\epsilon))$.

This raises a natural question: why does the running time depend on W? Is there any way to find the approximate weight of the MST in $o(W)$ time?[1] In this problem, we explore this problem in more detail, showing why we need at least $\Omega(W)$ time.

Basic assumptions. For the purpose of this question, we will restrict our attention to algorithms with the following properties:

- It runs on a graph $G = (V, E)$ with maximum degree d and maximum edge weight W. Let $|MST|$ be the weight of the minimum spanning tree of G.
- It returns a weight w such that $(|MST| - n/4) < w < (|MST| + n/4)$. That is, it gives an additive approximation with error $< n/4$.

Below, whenever we consider an MST algortihm, it satisfies these properties. Our goal is to show that such an algorithm cannot run in time $o(W)$.

Two graphs. We now define two graphs G_1 and G_2. Each of these graphs is a line consisting of n nodes. Graph G_1 has all edges of weight 1, and so the MST of graph G_1 has weight $n-1$. Graph G_2 has some edges of weight W, and all the other edges are of weight 1. In part (a), below, you will decide more precisely how many edges graph G_2 has of weight W.

The Differentiation Problem. We define a new problem called *The Differentiation Problem*. The input to your new problem is a graph, either G_1 or G_2. Your goal is to design an algorithm for deciding whether it received graph G_1 or graph G_2, i.e., the output should either be "1" or "2."

Problem 1.a. Assume $W < n/4$ and $n > 4$. First, specify more precisely how many edges of weight W graph G_2 should have. Then, prove that if algorithm A can find the approximate MST weight in $o(W)$ time (with the properties specified above) with probability at least $2/3$, then it could also solve the problem of distinguishing graph G_1 from graph G_2 in $o(W)$ time with probability at least $2/3$.

[1] Obviously, if W is sufficiently large, e.g., $W > m \log n$, then we can simply use Prim's or Kruskal's algorithm. Hence we are interested in the case where $W = o(n)$.

Solution: Choose graph G_2 to contain $\lfloor n/W \rfloor$ edges of weight W, with all the remaining edges of weight 1. Notice that graph G_2 has an MST of weight at least $(n-1)+(n-W) > 3n/2$. We have already indicated that graph G_1 has an MST of weight exactly $n-1$.

Assume algorithm A finds an approximate MST. Consider the following algorithm B for solving the differentation problem: run algorithm A, which returns some weight w. If $w < 5n/4$, then return G_1. Otherwise, return G_2.

Notice that on input G_1, the approximation guarantee of algorithm A insures that it returns a weight $< (n-1) + n/4 < 5n/4$ with probability at least $2/3$, and hence algorithm B returns G_1 with probability at least $2/3$. Similarly, on input G_2, the approximation guarantee of algorithm A insures that it returns a weight $> 3n/2 - n/4$ with probability at least $2/3$, and hence algorithm B returns graph G_2 with probability at least $2/3$.

We have therefore reduced the problem of differentiating G_1 and G_2 to the problem of finding an approximate MST weight. Hence if we show that the differentiation problem requires $\Omega(W)$ time, we can also conclude that the approximate MST problem also requires $\Omega(W)$ time.

Problem 1.b. A key implication of Part (a) is that if we can prove that no algorithm can distinguish graph G_1 from G_2 in $o(W)$ time (with probability at least 2/3), then we can conclude that no algorithm can find the approximate MST weight in $o(W)$ time (with probability at least 2/3).

Assume algorithm B is an algorithm for distinguishing graph G_1 and G_2 with probability at least 2/3. Assume algorithm B works as follows:

- Algorithm B first chooses a uniform random sample S of the edges in the input graph. Assume the sample contains s edges, and s is fixed in advance (i.e., does not depend on what the algorithm sees while it runs).
- Then, it processes sample S in some way and returns either "1" or "2." (It does not look at the graph again; only the edges in S affect the outcome.)

Prove that, if algorithm B succeeds with probability at least 2/3, then the sample S has to be of size at least $\Omega(W)$, i.e., B has running time at least $\Omega(W)$.

Solution: We will assume for this solution that n is divisible by W. (The proof generalizes for any value of n sufficiently large.) Recall that graph G_2 has $\lfloor n/W \rfloor = n/W$ edges of weight W, and the remaining edges of weight 1.

Assume that $s < W/4$. We will show that algorithm B fails on either graph G_1 or graph G_2 with probability $> 1/3$.

When algorithm B is run on graph G_1, it sees only edges of weight 1 in the sample, and it returns an output of "1" with probability $p \geq 2/3$. Otherwise, algorithm B does not satisfy the correctness requirement with respect to graph G_1 and we are done.

Now consider algorithm B when it is run on graph G_2. If the sample S contains only edges of weight one, it must also return "1" with probability $p \geq 2/3$, since the output of algorithm B depends only on what it observes in the sample S.

We analyze the sample S and determine the probability that it contains only edges of weight 1. For each item in the sample, the probability of finding an edge of weight W is $(n/W)/n = 1/W$. Thus, the probability that the algorithm does not find any edges of weight W in the sample is at least $(1 - 1/W)^s \geq e^{-2s/W}$. If $s < W/4$, then this is at least $e^{-1/2} > 1/2$.

And, recall, if the sample contains all ones, then algorithm B returns G_1 with probability at least 2/3. Thus, in total, algorithm B returns the wrong answer with probability $> (1/2)(2/3) = 1/3$.

So we conclude that the sample must be of size $s > W/4$, i.e., $s = \Omega(W)$ for the algorithm B to successfully distinguish G_1 from G_2 with probability at least 2/3.

Thus we have proved that no algorithm that simply takes a uniform random sample of size $o(W)$ can find the approximate weight of an MST in time $o(W)$.

Problem 1.c. (**Optional.**) So far, we have showed that one special type of algorithm, i.e., one that takes a uniform random sample, cannot solve the approximate MST problem in $o(W)$ time. Maybe there is some other more clever solution that does not use a uniform random sample and can still succeed in time $o(W)$?

In fact, no! We have one very powerful tool for translating the impossibility of an algorithm that takes random samples into a general claim of impossibility. This is known as Yao's Principle. The idea behind Yao's principle is to relate the performance of a *deterministic* algorithm on a random input to the performance of a *randomized* algorithm on a worst-case input. In the context of this class, one version of Yao's Principle shows the following fact:

Theorem 1 (Yao's Principle) *Assume the following:*

> *There exists a distribution D of the inputs such that: for every deterministic algorithm A of query complexity q, $Pr[A(x)$ is wrong$] > 1/3$.*

Then we can conclude:

> *For any randomized algorithm A of query complexity q there exists an input x such that: $Pr[A(x)$ is wrong$] > 1/3$.*

Here, the query complexity of an algorithm is the number of locations in the input that are examined in the execution of the algorithm. Yao's Principle shows that if every deterministic algorithm need more than q queries to respond to inputs drawn from distribution D, then every randomized algorithm also needs at least q queries to respond to a worst-case input.

To use Yao's Principle to show a lower bound for the Graph Differentiation Problem, you need to chose a distribution D over inputs (i.e., graphs G_1 and G_2), and show that every *deterministic* algorithm that is correct with probability at least $2/3$ requires $\Omega(W)$ queries when run on an input chosen according to distribution D.

Then Yao's principle shows that every randomized algorithm that is correct with probability at least $2/3$, running on a worst-case input, requires at least $\Omega(W)$ queries and hence $\Omega(W)$ time.

Use Yao's principle, along with the reduction from Part (a), to show that every randomized algorithm that finds a sufficiently good additive approimation to the MST weight with probability at least $2/3$ requires at least $\Omega(W)$ time.

Solution: As we showed in Part (a), it is sufficient to show that the Differentiation Game requires $\Omega(W)$ time. Hence we focus on applying Yao's Principle to the Differentiation Game.

For the purpose of Yao's Principle, we fix $q = W/8$. We want to show that if an algorithm B solves the Differentiation Game with probability at least $2/3$, then it must make more than q queries. We fix a distribution of input graphs as follows:

- W.p. $1/2$, input graph G_1.
- W.p. $1/2$, input graph G_2 where each edge has weight W with probability $1/W$.

The deterministic algorithm will access some set S of edges in the graph. (These choices depend on the input.) For each element accessed, it has weight W with probability $1/W$ and weight 1 otherwise. By the same argument as in the previous part, if the input graph is G_2, the probability that any of the edges accessed by the algorithm have weight W is $< 1 - e^{-4} < 1/3$.

Notice that if the algorithm sees all edges of weight one, it must always execute the same steps (as it is deterministic), and hence it always outputs the same answer. We must consider both the possibilities: when it sees edges of weight 1, then it returns "2" or it returns "1." (The first option may seem obviously silly, but we must consider both possibilities in the proof.)

In Case 1, it outputs "2" when it sees all edges of weight 1. In this case, the probability of returning the correct answer is at most $1/2$, since with probability $1/2$ the input is graph G_1 (resulting in the algorithm seeing all ones).

In Case 2, it outputs "1" when it sees all edges of weight 1. In this case, with probability at least $1/2$, the input is graph G_2; with probability at least $1/2$, the algorithm detects only edges of weight 1 and outputs the wrong answer. Hence the algorithm is correct with probability $< 1 - (1/2)(2/3) = 2/3$.

In either case, the probability of returning the right answer is $< 2/3$.

We conclude that if a deterministic algorithm B correctly solves the Differentiation Problem with q queries with probability at least $2/3$, then $q > W/8$. Applying Yao's Principle, we conclude that a randomized algorithm A that solves the Differentiation Problem with probability at least $2/3$ on all inputs must use at least $q > W/8$ queries, i.e., has running times $\Omega(W)$.

Finally, by our reduction from approximate weight MST, we conclude that any algorithm that can find an approximation of the weight of an MST with error at most $n/4$ with probability at least $2/3$ must use at least $\Omega(W)$ time.

Problem 1.d. Humperdink does not believe your lower bound. He believes that he has an algorithm for finding the approximate weight of an MST, as long as all the edges in the graph are either 1 or W. (That is, there are no weights in the range $[2, W-1]$.) Notice that the graphs G_1 and G_2 above are both graphs of this type, i.e., only containing weights 1 and W. Humperdink

also believes he can solve the Differentiation Problem in $o(W)$ time.

Humperdink proposes the following algorithm for finding the approximate weight of an MST containing only edges of weight 1 and W:

1. Let G' be the graph G with all the edges of weight W removed, i.e., only edges of weight 1. (Update: you do not need to explicitly construct graph G'.)

2. Run the sublinear time algorithm for finding the number of connected components on graph G'. (Recall, the algorithm works by performing a BFS on graph G', which can be easily done by ignoring edges of weight W.) Use the algorithm presented in class that returns a correct answer with probability at least 2/3.

3. Assume the algorithm returns the answer k. Then we conclude there must be $(k-1)$ edges of weight W and $(n-1)-(k-1)$ edges of weight 1, and so Humperdink's Algorithm returns an approximate MST weight of $n - W + k(W-1)$.

Humpedink claims that this returns a good approximation of the MST weight in time $o(W)$. (Assume Humperdink wants to find a weight w where $(|MST| - n/4) < w < (|MST| + n/4)$.) He claims that this shows that your lower bound must be wrong.

What is wrong with Humperdink's argument? Be as precise as possible, i.e., do not just say that it cannot work because of the lower bound. Explain to Humperdink clearly where exactly it fails.

Solution: The problem is in the relationship between the error and the time complexity. Recall that the algorithm presented in class for finding the number of connected components with error at most ϵn requires time $\Theta(d/\epsilon^3)$.

Notice that we multiply the output of the connected component algorithm k by $W-1$, i.e., we get an additive error of $\epsilon(W-1)n$. In order to get an estimate that is within $n/4$ of the real MST weight, we need $\epsilon(W-1)n \leq n/4$, i.e., we need $\epsilon = 1/4(W-1)$. However, if $\epsilon = 1/4(W-1)$, then the running time of the algorithm for finding connected components is $\Omega(W)$.

Problem 2. Average is better.

Recall that in class we designed an algorithm for finding the number of connected components in a graph that runs in time $O(d/\epsilon^3)$, where d is the maximum degree of the graph. In this problem, the goal is to design an algorithm that runs in time $O(\bar{d}/\epsilon^3)$ where $\bar{d}$ is the *average* degree of the graph. We will proceed in several steps.

Problem 2.a. Say that node v is of *rank* k if node v is the kth largest node in the graph when sorted by degree. That is, if you sort the nodes in the graph from largest degree to smallest degree, breaking ties arbitrarily (e.g., by node identifier), then node v would be the kth node in the sorted list.

For a constant C (e.g., $C = 512$), give an algorithm for finding a node with rank at least $\epsilon n/C$ and rank at most $\epsilon n/4$. Prove that your algorithm is correct with probability at least $7/8$, and that the running time is $O(d^*/\epsilon)$, where d^* is the degree of the node returned.

Hint: Choose a random sample, and take the largest degree node in the sample. For the running time, recall that it takes time degree(v) to find the degree of v.

Solution: Choose a sample of $S = C/(32\epsilon)$ nodes, and choose the node with the largest degree.

The probability of getting a node in the top $\epsilon n/C$ is ϵ/C. So, the probability of not getting a node in the top ϵ/C is $(1 - \epsilon/C)^{C/(32\epsilon)} \geq e^{-2/32} \geq 15/16$.

On the other hand, the probability of getting a node in the top $\epsilon n/4$ is $\epsilon/4$, and so the probability of not getting a node in the top $\epsilon n/4$ is at most $(1-\epsilon/4)^{-C/(32\epsilon)} \leq e^{-C/128} \leq e^{-512/128} \leq 1/16$.

Hence, with probability at least $7/8$, the set S contains at least one element with rank $\geq \epsilon n/4$, and no items with rank $\geq \epsilon n/C$.

For each node, we have to compute the rank. Since we return the node with the largest degree, and that node has degree d^*, clearly each sampled node requires time d^*, i.e., the total running time is $d^*C/(32\epsilon)$.

Problem 2.b. Let d^* be the degree of the node found in the previous part, and assume that it is the degree of a node with rank at least $\epsilon n/C$ and at most $\epsilon n/4$. Show that $d^* = O(\bar{d}/\epsilon)$. (Here we treat C as a constant.)

Solution: Recall that you cannot have more than half the elements with degree twice the average. Similarly, you cannot have more than $\epsilon n/C$ of the nodes with degree more than C/ϵ times the average.
Assume, for the sake of contradiction that there were more than $\epsilon n/C$ nodes with degree more than $\bar{d}C/\epsilon$. Then the total degree of these nodes would be more than $\bar{d}n$. Hence even if all the remaining nodes had degree 0, the average degree would be $> \bar{d}$, which is a contradiction.
Recall that d^* is the degree of a node with rank at least $\epsilon n/C$. Since we cannot have $\epsilon n/C$ nodes with degree more than $\bar{d}C/\epsilon$, we can conclude that d^* is the rank of a node with degree $\leq \bar{d}C/\epsilon$, i.e., $d^* = O(\bar{d}/\epsilon)$.

Problem 2.c. Let d^* be the degree of the node found in the previous part, and assume that it is the degree of a node with rank at least $\epsilon n/C$ and at most $\epsilon n/4$. In the graph G, how many connected components contain at least one node of degree $> d^*$?

Solution: Since d^* is the degree of a node with rank at most $\epsilon n/4$, we know there are at most $\epsilon n/4$ nodes with degree $> d^*$.

Problem 2.d. Finally, put all the pieces together. Give a modified algorithm for computing the connected components of a graph that runs in times $O(\bar{d}/\epsilon^4)$. (**Note:** the original version of this problem set asked for running time of $O(\bar{d}/\epsilon^3)$, however any bound of $O(\bar{d}/\epsilon^c)$ for some constant c is acceptable.) Argue that your algorithm is correct (based on the previous parts) and explain why it runs in the specified time. (You do not need to repeat the proof from class, but can simply explain carefully where it changes, in sufficient detail so that it is clear you understand how to modify the proof.)

Solution: First, we run the algorithm from Part (a) to identify a degree $d^* = O(\bar{d}/\epsilon)$. Notice that this takes at most time $O(\bar{d}/\epsilon)$. It is correct with probability at least 7/8.

Second, we run the connected components algorithm as specified in class, except whenever we find a node with degree $> d^*$, we abort and skip that sample entirely. Notice that the running time of this is at most $O(d^*/\epsilon^3) = O(\bar{d}/\epsilon^4)$.

In more detail: Recall, previously, we specified that $n(u)$ is the number of nodes in the connected component of u, and $\tilde{n}(u) = \min(n(u), 2/\epsilon)$. We modify this so that if a BFS at node u encounters a node with degree $> d^*$, then $\tilde{n}(u) = 0$. Our goal as before is to approximate $\sum(1/\tilde{n}(u))$; as before, the sampling procedure ensures a good estimate. Finally, we observe that by reduce $\tilde{n}(u)$ to zero in some cases, we have at most decreased $\sum(1/n(u))$ by $\epsilon n/4$, since there are at most $\epsilon n/4$ components containing a node with degree $> d^*$.