CS5234: Combinatorial and Graph Algorithms

Problem Set 3

Due: September 1st, 6:30pm

Instructions. This problem set wraps up the first part of the class on streaming algorithms! The first question looks closely at the L0-sampler and tries to use it solve a problem for which it is not well suited! The second question asks you to develop an algorithm for testing whether a graph is bipartite. For both of these questions, an important aspect is communicating your ideas clearly. Take the time to think about the best way to explain your ideas.

- 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. In the lecture notes for Week 3, we describe a streaming algorithm for deciding if a graph is 2-connected. Give a streaming algorithm for deciding if a graph is k-connected. Prove that your algorithm is correct and that the probability of error is small. How much space is needed?

Exercise 2. In Week 3, we described a streaming algorithm for finding a 2-approximate minimum spanning tree which finds a spanning tree T whose weight is at most $2|T_{MST}|$, where $|T_{MST}$ is the weight of the MST. Describe how to generalize this algorithm to find a $(1+\epsilon)$-approximate minimum spanning tree, i.e., a tree T whose weight is at most $(1+\epsilon)|T_{MST}|$. How much space does this require?

Exercise 3. Imagine you have an algorithm A that estimates the number of whoozits in a graph G. (What is a whoozit? I don't know.) The algorithm guarantees that A gives the correct answer with probability at least $2/3$. Assume you know absolutely nothing else about A.

Can you use algorithm A to construct a new algorithm A' that is correct with probability $9/10$? If so, how? If not, why not?

Imagine instead that algorithm A returns a value X where $\mathrm{E}[X]$ is equal to the number of whoozits in graph G. Can you use algorithm A to construct a new algorithm A' that is correct with probability $9/10$? If so, how? If not, why not?

Standard Problems (to be submitted)

Problem 1. Perilous Proofs

Consider the following algorithm for reconstructing a graph from a stream. The algorithm should reproduce the entire graph after the stream is complete. First, we build a sketch using a single L0-sampler:

- We maintain one L0-sampler of the vector v which is indexed by pairs of nodes (as in the graph sketch).
- Whenever an element in the stream adds an edge (u, v), we increment the vector at position (u, v), updating the L0-sampler.
- Whenever an element in the stream deletes an edge (u, v), we decrement the vector at position (u, v), updating the L0-sampler.

After the stream is complete, we reconstruct the graph as follows. The goal of the reconstruction is to enumerate *all* the edges in the graph. Let E' be an initially empty set.

Repeat until the sampler returns NULL (i.e., no more edges in the graph):

- Let (u, v) be an edge sampled from the L0-sampler.
- Add (u, v) to the set E'.
- Instruct the sampler to decrement the vector at position (u, v), i.e., deleting edge (u, v).

We implement the sampler using the L0-sampler described in class, setting $\epsilon = 1/n^3$. That is, the sampler is correct with probability at least $1 - 1/n^3$, and using space $O(\log^4 n)$. We now give a proof that this algorithm works correctly with probability at least $1 - 1/n$:

Proof Let $G = (V, E)$ be the original graph. We prove the claim by induction. Our inductive hypothesis is that before and after each iteration of the outer loop, the L0-sampler is a valid sketch of the graph $G = (V, E \setminus E')$. Therefore, when the L0-sampler contains no more edges (ie., $E \setminus E' = \emptyset$), we conclude that E' must contain all the edges.

The base case of the induction is true by construction: after the stream completes, E' is empty, and the L0-sampler is a valid sketch of the graph G.

For the inductive step, notice that we find a valid edge (u, v), delete it from the sampler and add it to E'. Thefore the invariant continues to hold.

Finally, we observe that in each iteration, the probability that the L0-sampler fails is at most $1/n^3$. There are at most n^2 edges in the graph and hence at most n^2 iteration. Thus, by a union bound, we conclude that the probability that it fails and returns a bad answer at any point during the algorithm is at most $n^2/n^3 = 1/n$. □

Problem 1.a. Explain why this immediately seems likely to be wrong, without even analyzing the details. A very simple "sanity check" should suggest to you that there must be a mistake. (Always do a sanity check on your algorithms and calculations!)

Even without knowing anything about how the L0-sampler is implemented (and hence for all possible implementations that provide the specified guarantees), it is impossible for this algorithm to work. Why?

Solution: A simple counting argument should make you very suspicious. The graph contains m edges, and specifying those m edges requires at least m space. (That is an informal claim, and would require a more careful proof, but it should be intuitively true.) The L0-sampler uses at most $O(\log^4 n)$ space. It is obviously impossible to store $O(m)$ information in $O(\log^4 n)$ space. Hence this algorithm seems very, very suspicious. You should immediately suspect a problem!

Problem 1.b. Explan in more detail what actually goes wrong. Where, precisely, does the algorithm fail? Please give a specific answer here (i.e., do not just write what goes wrong with the proof). At what point does the algorithm give a bad answer? How does the data structure fail? What is the maximum number of edges that the algorithm can reconstruct before it fails (even if you are very, very lucky)?

Solution: Think about how the L0-sampler really works. The L0-sampler consists, at its base, of $\Theta(\log^3 n)$ 1-samplers. Each of the 1-samplers can return at most one edge.
Consider what happens when the stream is complete. Some 1-samplers store one edge, and some have multiple values mapped to them and hence return an error. As the reconstruction algorithm runs, edges are deleted, and some of the 1-samplers go from "error" to storing a single edge. Even so, during the entire execution of the reconstruction algorithm, each 1-sampler can return at most 1 edge.
Therefore, after $\Theta(\log^3 n)$ iterations (at the latest), ever 1-sampler will either be empty or in an "error" state and the L0-sampler will return no more edges.

Problem 1.c. What is wrong with the proof? Where exactly does the proof make a mistake? Again, be as precise as possible, pointing to the exact location in the proof where something is wrong.

Solution: There are two ways to look at the problem: one is that the proof misinterprets what an L0-sampler promises; the other is to try to understand why the L0-sampler cannot guarantee more.
First, the proof misinterprets the guarantee of the L0-sampler. The proof assumes that the L0 sampler satisfies the following theorem:

> For every access to the L0-sampler, it returns a uniformly random non-zero index with probability at least $1 - \epsilon$.

This is not true. It guarantees that with probability at least $1 - \epsilon$ it is a valid sketch, and it guarantees that it will return *one* non-zero index. There is no guarantee that it will return more than one non-zero index.
Why, however, does the L0-sampler not work for more than one sample? All the analysis assumes that the stream of edge modifications is independent of the random choices made by the algorithm (i.e., independent from the hash functions). As a thought experiment, imagine you knew the hash functions in advance. Then you could cause all sorts of trouble by choosing edges that the hash functions all map to the same place, in the same way.
In effect, that is what you are doing when you use the L0-sampler to find an edge and then delete it. The choice to delete that edge is now not independent of the hash functions used to construct the sampler, because it was a result of the hash functions themselves.
All the analysis treated the hash functions as random maps, independent of the item being mapped. However, if the item being deleted is the edge returned previously by the sampler, it is not independent, and the hash functions do not behave in a random manner—they behave in exactly the manner that they previouly behaved.

Problem 1.d. Assume you are given an L0-sampler, and you know all the hash functions used inside the L0-sampler. Can you design a graph for which the L0-sampler will fail with probability $> 1/2$? Think about this, but you do *NOT* need to submit a detailed answer. Do not write more than one or two sentences.
Solution: Yes! There are many ways to do this. To accomplish this, you can construct a dense graph (e.g., with $\Theta(n^2)$ edges) with (at least) one edge that is not mapped to any of the useful s-sparse samplers. (What makes an s-sparse sampler useful? If the graph has $\Theta(n^2)$ edges, then it is clearly useless if the probability p of assigning an edge to that sampler is too large, e.g., $p > 1/n$.)
Think about all the s-sparse samplers that accept edges with probaiblity 1/4, 1/8, 1/6, etc. Notice that all of those samplers together accept no more than 1/2 the edges. Thus we can find a set of $\binom{n}{2}/2$ edges that are only passed on to the s-sparse samplers with probability $p = 1$ and $p = 1/2$. For these samplers, the probability that an edge is mapped to a unique 1-sparse sampler is very low!
Notice, of course, that you do not need a dense graph to trick an L0-sampler. Since the sampler is only storing $O(\log^4 n)$ bits of information, even a sparse graph is good enough.

Problem 2. Is it bipartite?

In this problem, your job is to develop a streaming algorithm for determining whether a graph is bipartite.[1] Your algorithm should process a stream of edge additions and deletions, and then when the stream is complete, it should determine whether the graph is bipartite or not. It should return the correct answer with probability at least $1 - 1/n$. And it should use as little space as possible.

Describe your algorithm for determining whether a graph is bipartite. Your goal is to give a clear and succinct description of the algorithm, including: (i) a high-level overview, (ii) a precise explanation of how it processes the elements of the stream, and (iii) a precise explanation of how it determines whether a graph is bipartite. You are encouraged to use pictures and pseudocode as necessary to clearly explain the algorithm. (You may use as building blocks any of the algorithms we have studied in this class, but your description should be sufficiently clear that someone who took the class one year ago can understand.[2])

Then, prove that your algorithm is correct, i.e., that it determines whether the graph is bipartite. (State and prove any additional lemmas that help.) Give the probability of error, and show that the algorithm achieves it. Similarly, analyze the space complexity of your algorithm.

Hint: The following construction may be helpful. Given a graph $G = (V, E)$, construct a new graph $D = (V', E')$ as follows:

- For every node $u \in V$, add two nodes u_1 and u_2 to V'.
- For every edge (x, y) in E, add two edges (x_1, y_2) and (x_2, y_1) to E'.

(Perhaps, draw a picture!) You might try to relate the number of connected components in D to whether or not G is bipartite. (Try a few examples, and count the connected components in D.)

[1] Note that the graph does not have to be connected to be bipartite.

[2] For example, referring to "that claim you made last Thursday" may not be useful for someone who took the class last year!

Solution:

Overview. We will show that graph D has exactly twice as many connected components as G if and only if G is bipartite. This immediately yields an algorithm: run two algorithms for counting the connected components in a streaming graph; for one of these algorithms, give it the edges of G, for the other, give it the edges of D. (Notice that for each edge $e = (u, v)$ that is updated in the stream for G, we can derive two edges (u_1, v_2) and (u_2, v_1) that are updated in the stream for D.) We can then determine whether G is bipartite by comparing the number of connected components in G and D. We run each of the two streaming algorithms for counting connected components with error parameter $1/2n$, and so by a union bound, we ensure that the algorithm is correct with probability at least $1 - 1/n$. The total space required is simply that of two streaming algorithms for connected components, i.e., $O(n \log^5 n)$.

We now prove the key claim, i.e., that D has twice as many connected components as G if and only if G is bipartite. The proof below is somewhat verbose, and yet with a picture the basic idea should be relatively clear: connected components containing an odd cycle in G yield a single connected component in D, while connected components with no odd cycles in G yield two connected components in D. The bulk of the proof involves showing this property carefully.

Lemma 1 *Let $G = (V, E)$ be a graph, and let $D = (V', E')$ be the double cover of G as defined above. Then $CC(G) = 2CC(D)$ if and only if G is bipartite.*

Proof Assume that G contains an odd-length cycle, and let W be the nodes in a connected component in G with an odd-length cycle. Let u be a node in W that is on an odd-length cycle C. Then in D, we see that u_1 is connected to u_2, since at every step along the odd-length cycle C in graph D, we alternate between 1 and 2 nodes; hence if we start at u_1, then when we return to u after an odd number of steps we will be at u_2.

Also, since W is connected, there is a path from every node $u \in W$ to every node $v \in W$. If that path is of even length, then there is a path from u_1 to w_1 and from u_2 to w_2 in D. If that path is of odd length, then there is a path from u_1 to w_2 and from u_2 to w_1 in D. Either way, since u_1 and u_2 are connected, then w_1 and w_2 are also connected in D. More generally, we see that the nodes W_1 and W_2 in D form a single connected component.

On the other hand, assume that G contains a connected component with no odd-length cycles, and let W be the nodes in that connected component. Let $u \in W$ be such a component. Since any path from u_1 to u_2 in D would be of odd-length, we conclude that u_1 and u_2 are not connected in D. Thus nodes W_1 and W_2 in D contains at least 2 connected components.

We also observe that nodes W_1 and W_2 have at most 2 connected components in D: let u be some node in W; then every node in W is connected to either u_1 or u_2 in D. To see this, let v_1 be an arbitrary node in W. (The same holds in reverse for v_2.) Then there is a path in G from v to u that is either of even or odd length. If the path is of even length, then there is a path in D from v_1 to u_1. If the path is of odd length, then there is a path in D from v_1 to u_2. Either way, v_1 is connected to either u_1 or u_2. Since this is true for all nodes in W_1 and W_2, we conclude that it has at most 2 connected components.

Finally, we wrap up the proof. Assume G is bipartite. In this case, all the connected components of G have no odd cycles. Then, as we have just argued, each of these connected components in G yields 2 connected components in D, and hence D has exactly twice as many connected components as G.

Conversely, assume G is not bipartite. In this case, at least one connected component of G has an odd cycle. Hence that connected components in G yields exactly one connected component in D. All the remaining connected components in G yield either one or two connected components in D. Thus D must have strictly less than twice the number of connected components as G. □