15-451/651: Design & Analysis of Algorithms
Lecture #25: Graph Sketching

In the last lecture we saw how to find a non-zero item of an arbitrary non-zero vector x presented as a stream of insertions and deletions to its coordinates, using $O(\log^2 n)$ bits of space. More precisely, we assumed all the entries of x were integers in the range $\{-n^C, -n^C + 1, -n^C + 2, \ldots, n^C - 1, n^C\}$ at all times in the stream, and thus representable with $O(\log n)$ bits, and in this case we came up with an algorithm which had the following guarantees.

With probability $O(\log n)/n$, the algorithm is incorrect and may output something arbitrary. With the remaining $1 - O(\log n)/n$ probability,

- if $x = 0^n$, we output FAIL,
- otherwise if $x \neq 0^n$, with probability at least $4/5$, we output an entry i for which $x_i \neq 0$, and with the remaining probability we output FAIL.

Recall from the previous lecture that the $O(\log n)/n$ probability of being incorrect was somewhat arbitrary, and could be adjusted just by replacing the prime p in the 1-Sparse-Finder algorithm with a larger prime. That is, we chose p to be a prime between n^2 and $2n^2$, but if we instead choose a prime p between n^3 and $2n^3$, then the probability of being incorrect is just $O(\log n)/n^2$. We will need this smaller probability for today's lecture. Let us refer to the above algorithm as Non-Zero Finder.

1 Graph Sketching

Our main motivation will be to process a graph stream. We now would like to find a distribution on "wide-fat" matrices $S \in \mathbb{R}^{k \times n}$ for a small number k of rows, so that we can compress a graph, represented as a matrix. For example, an unweighted undirected graph G on n nodes may be represented as an $n \times n$ adjacency matrix A_G where $(A_G)_{i,j} = 1$ if $\{i, j\}$ is an edge, and $(A_G)_{i,j} = 0$ otherwise. Note that A_G is a symmetric matrix. For example, one could ask the question - is there a distribution on matrices S with $k = \text{poly}(\log n)$ for which given $S \cdot A_G$, one can output a spanning forest of G? We will consider this question again later in the lecture.

Before getting into the motivation for graph sketching, we first consider a stream of insertions of edges $e_1, \ldots, e_m$ of a graph G. Suppose we would like to output a spanning tree of G. We could trivially store all the edges of G, but we may have $m = \Theta(n^2)$, and this would require $\Omega(n^2)$ bits of memory. Instead, can we accomplish certain tasks with only $n \cdot \text{poly}(\log n)$ bits of memory, which would be much more efficient?

For example, consider the task of ouputting a spanning tree of a given connected, unweighted graph G. Here is a natural algorithm SpanningTree: for each edge e you see in the stream, store it if and only if it does not form a cycle with the edges you have already stored. By definition of the algorithm, the set E' of edges that you store in the stream does not form a cycle, and hence, you store at most $n - 1$ edges, and so $O(n \log n)$ bits of space, since for each edge you need to store the identities of its endpoints.

Lemma 1 *Algorithm* SpanningTree *returns a spanning tree given that G is connected.*

Proof: Suppose for the sake of contradiction, that the graph with edgeset E' were disconnected, and consider a cut $(S, V \setminus S)$ where S and $V \setminus S$ are non-empty, and there are no edges in E' from S to $V \setminus S$. Since G is connected, there is an edge $e = \{u, v\}$ from S to $V \setminus S$ in G. When processing edge e in the stream, there cannot already be a path from u to v in E', since such a path would contain an edge from S to $V \setminus S$, and by assumption E' has no edges from S to $V \setminus S$. Consequently, e does not form a cycle with the edges E' that have already been stored, and so e would have been included in E' by the streaming algorithm. ■

While the above lemma works on streams with insertions to its edges, it does not work if the stream is allowed to have both insertions and deletions to its edges. Outputting a spanning tree in this case is now much more involved, and we will use sketching to solve this problem.

The usual sketching motivation for vectors also applies to graphs. Given a stream of insertions or deletions to the edges of a graph, we can, for example, store a sketch $S \cdot A_G$ of the adjacency matrix A_G, where S is our sketching matrix. The memory required of the streaming algorithm is (the number of rows of S) $\cdot n \log n$ bits to store $S \cdot A_G$, and one also needs to store S, which is (the number of rows of S) $\cdot n \log n$. Here we assume the entries of S can be represented with $O(\log n)$ bits. So SA_G and S are both (the number of rows of S) $\times n$ matrices, and we store $O(\log n)$ bits to represent each of their entries.

So if we wanted to find a spanning tree of a connected unweighted graph in a stream of insertions and deletions to its edges, we could store $S \cdot A_G$ and S. The goal would be to find a distribution on matrices S with a small number of rows. As we will see, we will be able to find such a distribution on matrices S with $O(\log^2 n)$ rows!

1.1 Parallel Computing

Before doing so, we mention an application to parallel computing. Suppose there are a constant number of servers, each holding a disjoint subset of the edges of a graph. More precisely, suppose there are $k = O(1)$ servers, $S^1, \ldots, S^k$, holding $E^1, \ldots, E^k$, respectively, and $E = \cup_i E^i$ on a common vertex set V. That is, the underlying graph is $G = (V, E)$. Let $G^i = (V, E^i)$. Suppose we would like to compute a spanning tree of G with low communication. Then the i-th server could compute $S \cdot A_{G^i}$, and transmit this to a centralized coordinator, for each $i = 1, 2, \ldots, k$. The centralized server could then use linearity to compute $\sum_{i=1}^k S \cdot A_{G^i} = S \cdot \sum_{i=1}^k A_{G^i} = SA_G$. This is just the sketch of the graph A_G! Notice that it was important that the sketching matrix S be *linear*, so that the sum of the sketches is the sketch of the sum. If from $S \cdot A_G$ one can output a spanning tree of G, then this gives a *distributed* algorithm for computing a spanning tree.

Notice that the communication is just k times the size of any individual sketch, which is (the number of rows of S) $\cdot n \log n$, and thus the total communication, summed up over all servers is $O(k)\cdot$ (the number of rows of S) $\cdot n \log n$. Thus, if we have a sketch with a small

number of rows, then we also reduce the amount of communication in parallel computing applications.

2 Computing a Spanning Tree with Insertions and Deletions

To present the algorithm, we first need to recall a classic algorithm for computing a spanning tree, namely, Boruvka's algorithm. We will need to slightly modify the algorithm for our setting, so present a self-contained analysis here.

Assume we have an undirected input graph which is unweighted and connected.

Modified Boruvka's Spanning Tree Algorithm

1. Initialize edgeset E' to $\emptyset$
2. Create a list L of n groups of vertices, each initialized to a single vertex
3. While L has more than 1 group,
   - For each group G, include in E' an arbitrary edge e from a vertex in G to a vertex not in G
   - Merge groups connected by an edge in the previous step
4. Find an output a spanning tree whose edges are within the set E'.

The slides give a good illustration of running Modified Boruvka's Spanning Tree Algorithm on an example graph, and so we refer you to the slides.

We now move on to the analysis.

Lemma 2 *If there are at least* 2 *groups in an iteration, then each group has an outgoing edge.*

Proof: A group is just a list of vertices. If there were a group with no outgoing edge, since there is at least one other group containing at least one vertex, the graph would be disconnected, a contradiction. ■

Lemma 3 *If there are t groups at the start of an iteration, then there are at most $t/2$ groups at the end of the iteration.*

Proof: Consider a graph whose vertices are the groups $G_1, \ldots, G_t$ at the start of the iteration, and where the edges correspond to the groups that we connect. From each group we choose an arbitrary outgoing edge connecting it to another group; this is possible by Lemma 2. By definition of the algorithm, the number of groups at the end of the iteration is the number of connected components in this graph. Since each connected component

corresponds to merging at least two groups, the number of groups at the end of the iteration is at most $t/2$. ■

By Lemma 3, after at most $\log_2 n$ iterations, there is only one group left. Notice that the total number of edges included in the edgeset E' in Modified Boruvka's Spanning Tree Algorithm is at most $n + n/2 + n/4 + \cdots + 2 + 1 \leq 2n$.

Lemma 4 *If V is the set of vertices of G, and E' is the edgeset output by* Modified Boruvka's Spanning Tree Algorithm*, then $H = (V, E')$ is a connected graph.*

Proof: The invariant that we maintain is that each group G_i in each iteration is connected by the edges in our set E'. Suppose this is true at the beginning of an iteration, and we connect two groups G_i and G_j by an outgoing edge. Since each of G_i and G_j are individually connected, and now connected by an edge, it follows that $G_i \cup G_j$ is connected by our edgeset E', showing the invariant continues to hold at the end of the iteration. Since in the final iteration we have a single group, it follows that this group is connected by our edgeset E'. ■

By Lemma 4, the graph H is connected, and thus, it contains a spanning tree. A spanning tree on the vertex set V using a subset of edges of E' is also a spanning tree on the vertex set V using a subset of edges of E, and thus we output a spanning tree of G, as desired.

We now show how to use the Modified Boruvka's Spanning Tree Algorithm in the context of graph sketching. Let the n nodes be $\{1, 2, 3, \ldots, n\}$. For each node a_i, we introduce a vector a_i of length $\binom{n}{2}$ which is indexed by node pairs:

- if $\{i, j\}$ is an edge, $a_i[i, j] = 1$ if $j > i$ and $a_i[i, j] = -1$ if $j < i$.
- if $\{i, j\}$ is not an edge, then $a_i[i, j] = 0$.

The key reason for representing the graph this way is the following lemma:

Lemma 5 *For a subset S of nodes in G, Support$(\sum_{i \in S} a_i) = E(S, V \setminus S)$. Here Support$(x)$ denotes the set of non-zero elements of x. Also, $E(S, V \setminus S)$ denotes the set of edges in G from S to $V \setminus S$.*

Proof: For an edge $\{i, j\}$, if $i, j \in S$ the sum of the entries along the $\{i, j\}$-th column of $\sum_{i \in S} a_i$ is 0, since we will have a 1 plus a -1, plus a bunch of 0s. If neither i nor j is in S, the sum along the $\{i, j\}$-th column of $\sum_{i \in S} a_i$ will be 0. Finally, if exactly one of i or j is in S, the sum along the $\{i, j\}$-th column will be 1 or -1, proving the lemma. ■

Note that one way to store $a_1, \ldots, a_n$ in the stream is that every time we see an insertion or deletion to an edge $\{i, j\}$, we can simple update the $\{i, j\}$-th column of a_i and a_j accordingly - if it is an edge insertion we put a 1 in the $\{i, j\}$-th column of a_i if $j > i$, otherwise we put a -1 in the $\{i, j\}$-th column of a_i. Similarly, we put a 1 in the $\{i, j\}$-th column of a_j if $i > j$, otherwise we put a -1 in the $\{i, j\}$-th column of a_j. Similarly, if it an edge deletion we put a 0 in both the $\{i, j\}$-th column of a_i and the $\{i, j\}$-th column of a_j.

The main problem with this approach is that each a_i is $\Theta(n^2)$-dimensional! So we have n different a_i, and this very naïve representation requires $\Theta(n^3)$ bits of space. The natural way around this is to sketch each a_i!

Indeed, instead of storing each a_i, let us store a sketch $C \cdot a_i$, for $i = 1, \ldots, n$. Here a_i is thought of as an $\binom{n}{2}$-dimensional column vector, and C is $k \times \binom{n}{2}$ for a small value of k, i.e., C is a wide-fat matrix. Then, we will only need to store $O(n \cdot k \cdot \log n)$ bits, which if k is small, will be a significant parameter reduction! In fact, C will exactly correspond to our sketch for outputting a non-zero item of a non-zero vector, which has $O(\log n)$ rows and thus $C \cdot a_i$ for a given i can be stored using $O(\log^2 n)$ bits of space.

It turns out, to run Modified Boruvka's Spanning Tree Algorithm, we will need to store not one sketch of each a_i, but $O(\log n)$ sketches $C_1 a_i, \ldots, C_{O(\log n)} a_i$ of each a_i, where $C_1, \ldots, C_{O(\log n)}$ are independent Non-Zero Finder sketches. For each $i = 1, 2, \ldots, n$, and for each $j = 1, 2, \ldots, O(\log n)$, we store $C_j a_i$. Notice that we use *the same* C_j for each a_i. Sinc each $C_j a_i$ can be represented with $O(\log^2 n)$ bits of space, in total this gives us $O(n \log^3 n)$ bits of space, as we range over all $i = 1, 2, \ldots, n$ and $j = 1, 2, \ldots, O(\log n)$.

A key observation is that the non-zero items of $C_j a_i$ are exactly the edges adjacent to i! Moreover, for a subset S of vertices, $\sum_{j \in S} C_j a_i = C_j(\sum_{i \in S} a_i)$, and by Lemma 5, the non-zero entries of $\sum_{i \in S} a_i$ are exactly the outgoing edges of the group S of vertices! By the guarantee of Non-Zero Finder, C_j will return such an outgoing edge. These are exactly the ingredients needed by Modified Boruvka's Spanning Tree Algorithm!

Here is the algorithm:

1. For each node j, use $C_i \cdot a_j$ to get an edge incident on j.
2. For $i = 2, 3, \ldots, O(\log n)$,
   (a) To get an outgoing edge from a group G of vertices in iteration i, compute $\sum_{j \in G} C_i a_j = C_i(\sum_{j \in G} a_j)$ and run the procedure associated with C_j to find an outgoing edge e from G to $V \setminus G$. Include edge $e \in E'$, where E' is the edgeset being maintained in Modified Boruvka's Spanning Tree Algorithm
3. As in Modified Boruvka's Spanning Tree Algorithm, output a spanning tree among the edges in E'.

As mentioned above, the total space is $O(n \log^3 n)$ bits, and the correctness follows almost identically to the analysis we gave for Modified Boruvka's Spanning Tree Algorithm. There is one minor change though, which is that Non-Zero Finder does not always return a non-zero element of the support of the vector which it is applied to. We said it could be incorrect with probability $O(\log n)/n^2$, but since we only run it $O(n \log n)$ times, we can assume every invocation is correct by a union bound. Even though it is correct, as mentioned with probability up to $1/5$, in some iteration i for some group G, $C_i(\sum_{j \in G} a_j)$ may just output FAIL. We handle this as follows.

The expected number of groups G for which $C_i(\sum_{j \in G} a_j)$ outputs FAIL in any given iteration is at most $(1/5)t$, where t is the number of groups G at the beginning of iteration i.

By a Markov bound, with probability at least $2/3$, at most $3t/5$ groups will not be connected to another group. If this happens, then there are at least $2t/5$ groups connected to another group. To maximize the number of connected components remaining, these $2t/5$ groups could all pair up with each other, in which case they will form $2t/10$ connected components. Thus, after the iteration, there will be at most $3t/5 + 2t/10 = 4t/5$ connected components with probability at least $2/3$. Since the sketches are independent across iterations, this means that with probability $11/\text{poly}(n)$, after $O(\log n)$ iterations, there will be only one connected component left. That is why we chose $O(\log n)$ sketches of each a_i.

All in all, we get an $O(n \log^3 n)$ space algorithm for outputting a spanning tree in a stream with insertions and deletions to the edges! This was shown in a breakthrough paper of Ahn, Guha, and McGregor in SODA 2012, "Analyzing Graph Structure via Linear Measurements"; see the paper for more details.