Analyzing Graph Structure via Linear Measurements.

Reviewed by Nikhil Desai

Introduction

Graphs are an integral data structure for many parts of computation. They are highly effective at modeling many varied and flexible domains, and are excellent for representing the way humans themselves conceive of the world. Nowadays, there is lots of interest in working with large graphs, including social network graphs, "knowledge" graphs, and large bipartite graphs (for example, the Netflix movie matching graph).

Many of the papers we have covered so far in CS167 have worked with graphs. Karloff et al. (**citation**) discussed counting triangles with social networks in a MapReduce environment, while Gupta et al. (**citation**) proposed a new way of discovering "triangle-dense" clusters representing potential communities in

There are a few similarities in the types of graphs examined in such papers.

1. First, beyond a doubt, all of them are **large.** The Facebook graph, for example, contains on the order of a billion vertices and nearly one trillion edges. Even the excellent graph primitives we learned in CS161, such as depth-first and breadth- first search, would require significant time and space to run on such graphs.
2. Second, they are all highly **dynamic.** New structures appear in social network graphs at every moment. In particular, one of the defining features of such graphs is that new edges and vertices can be constantly added. The structural changes these induce are of some interest.
3. Third, there is significant interest in the **global structure** of these graphs, and in particular how that structure changes over time.
4. Fourth, while we would like an infinite amount of space and time to solve these questions, we don't. Compared to the size of the graphs, **our computing power is very limited.**

These conditions, roughly, define the so-called "streaming graph problem," in which the user must analyze a graph presented in the form of incremental updates, using limited space and time.

What does "limited space and time" mean? To determine this, we need to review the various techniques available for graph storage. The standard representation we learned in CS161 was the adjacency matrix, which takes up size $\Theta(V^2)$. In practice, however, the standard is the *adjacency list,* which takes up space $\Theta(V + E)$, in the worst-case $O(V^2)$. Another uncommon form is the incidence matrix, which takes up space $\Theta(VE) = O(V^3)$. In general, the space available to us will be smaller than the worst-case storage necessary for such representations - in particular, it will be something like $O(V \log^k V)$. With regards to time, streaming models tend to ask that we not look at any part of the graph more than once - in other words, we must make a *single pass* through the graph and then provide an answer to a given query. Technically, this establishes a "runtime" bounded above by $O(V + E)$ or $O(V^2)$; in practice, we want to perform exactly one action involving the *actual* graph, and thereafter run in time linear in the size of the underlying sketch.

This may seem like a needlessly constrained model; why should we put arbitrary restrictions on our runtime or the auxiliary space used like this? To get some intuition, consider a common problem - getting information about the connectivity of a graph. This means not just simple questions like finding the smallest connected component, but also the more general question - for any given $u, v \in G$, are u and v connected?

Let's use connectivity as a central "conceit" for our examination of dynamic graphs, and see what we can do to fix these problems.

Connectivity

For a general graph, the problem of connectivity is well-understood and easy to solve. In particular, to find out if u is connected to v, we need only run a depth-first search from u to expose all the nodes in a graph to which u is connected. This depth-first search takes time $\Theta(V+E)$ (with very small constant factors), and is the best we can do on a static graph.

Consider what happens when we generalize this problem to dynamic graphs, however. In particular, make two further generalizations: assume that k edges or vertices are added, one by one, to the graph, and assume that after every edge addition we will query for the connectivity of two arbitrary nodes in the graph. This moves the graph into the realm of the *streaming* model as well - observe that we can build up a graph by simply starting with an empty graph, then adding its $V + E$ edges,. making $V + E$ queries.

With no additional storage allowed, the straightforward algorithm to solve this problem will have to simply rerun DFS over the graph after *every single query,* leading to an effective runtime of $O(kV + kE)$. In the case where $k = V + E$, this will degenerate to $O(E^2) = O(V^4)$ in the worst case. Clearly, this performance is poor; we would like to know how to do better. It turns out the solution to this involves something called "sketching," which we shall see more of below.

Sketching

When solving a problem on a static input, there's no advantage to holding on to information after the algorithm is completed; in fact, it would be an obvious waste to do so. However, when working with *dynamic* input, especially one which changes "incrementally," it becomes advantageous to retain at least *some* information about it. Usually, a dynamic dataset is so big that one cannot simply store it in memory; instead, we must store some smaller representation of it, which gives us an "approximation" of the data it contains. This, vaguely, is the notion of sketching.

More specifically, given a dynamic data structure T, a **sketch** $\mathcal{S}(T)$ is a substantially sublinear dynamic data structure that "approximates" the contents of T. We will break this down further.

- By **sublinear,** we mean that if $|D|$ denotes the memory necessary to store data structure D, then $|\mathcal{S}(T)| = o(|T|)$. In practice, we will want to achieve a "win" of roughly a linear factor; thus, if $|T| = \Theta(n)$, we will seek for $|\mathcal{S}(T)| = \Theta(\log^k n)$ for some k, and likewise a data structure of size $|T| = O(n^2)$ (such as a general graph) will get space $O(n \log^k n)$.
- By **dynamic,** we mean that $\mathcal{S}(T)$ should handle updates. To be fair, we will allow $\mathcal{S}(T)$ to *choose* which types of updates it supports, and refuse to support any others. More on this below.
- By **"approximation,"** we mean a few things. As a prelude, observe that a sketch $\mathcal{S}(T)$ *cannot* perfectly preserve the contents of T, because $|\mathcal{S}(T)|$ must be strictly *smaller* than $|T|$, and information theory tells us perfect compression in this form is impossible. There are two ways we can deal with this problem. First, we could decide to preserve only certain types information about T, letting queries relying on other information fail. Second, we could decide to relax the requirement of accuracy, and instead let the sketch only *approximate* the answers to specific questions about T. In practice, we shall do both.

This may seem a little obscure, so we will give a few examples of sketches.

1. Consider the **Bloom filter.** As covered in CS 161, the Bloom filter is a way to approximate the notion of a set, supporting queries such as IsElement and AddElement (but no others). It has space strictly smaller than that necessary to store an actual set (in fact, depending on the error parameters, it can be as small or as large as the user needs!), and because it supports the AddElement operation, it supports the dynamic updating of the set. This sounds an awful lot like a sketch!

2. Recall the paper from two weeks ago in which we discussed locality-sensitive hashing. For a set T, define $\mathcal{S}(T) = \{h(x) \mid x \in T\}$, where h is drawn from some locality-sensitive family $\mathcal{H}$ of hash functions. In this case, $\mathcal{S}(T)$ supports $\textsc{AreNeighbors}(x, y)$ for any $x, y \in T$, and supports any queries that rely solely upon discerning the "families" of its member.

At this point, sketches may seem like an unnecessarily limited form of computation - why on earth would we ever want to use them? In practice, however, forcing these restrictions allows sketches to have immense general-purpose applications.

- If we have a large dataset T, very often we will be unable to hold all of T in memory at once. Every time we want to obtain information about T, we will need to rerun an at-best-linear computation over T; if we have many such queries, this will quickly become large. Sketches provide a way to "cache" the information we know about T.
- Many times, we will receive information as a "stream" of updates rather than as a single monolithic input. In the Facebook social graph, for example, new members and new friendships occur all the time. We want our cache to be able to incorporate these updates and record them over time.

To re-emphasize a few points, recall that sketches will *not* preserve all information about a dataset; they can be thought of as "lossy" compressions. As a consequence, only certain questions can be asked of sketches, and even those may not be accurately answered. For many algorithms, though, these restrictions lead to powerful gains in space and time.

Solving connectivity with sketches

To recap: we want a $O(V \log^k V)$ data structure that answers questions about connectivity. In particular: for $u, v \in G$, is there a path from u to v? And what happens after we add or delete an arbitrary edge (x, y)?

We already tried one naive approach, namely rerunning the depth-first search algorithm after each addition of an edge, but saw quickly that it was too slow.

We propose a simple "sketch" that will solve the problem, as long as only edges and vertices are added to the graph. In particular, run Kruskal's algorithm over the graph to generate a *minimum spanning forest* of the graph. Then each connected component of the graph will be represented by *a unique* MST, and u and v will be connected if and only if they are part of the same MST. Since the maximum number of edges in a minimum spanning forest is $V - 1$, the total size of this data structure is $O(V)$. Furthermore, we store the connected components using the union-find disjoint-set data structure covered in CS 161, which takes up space $O(V)$. The union-find structure supports testing for set membership and “merging” subsets with high effectiveness, but does not support “splitting” subsets up.

Obviously, this sketch supports the query $\textsc{Is-Connected}(x, y)$. It also supports $\textsc{Add-Edge}(x, y)$ - to connect the nodes u and v, we check if they are already in the same connected component. If they are, we ignore the addition; if they are not, we connect them and record of u and v draw an edge connecting the two vertices in the graph.

Here’s a problem, though: *what happens if we delete edges?*

Key problem

We’ve now finally arrived at the problem the McGregor et al. paper aims to solve. In particular, we want to find a sketch $\mathcal{S}(G)$ of size $O(V \log^k V)$ for a graph G that solves the problem of dynamic graph connectivity, supporting the high-level queries of whether two vertices $u, v \in G$ are connected, and adding or deleting an edge (u, v) from G.

Intuition for the sketch

It turns out that the most effective way to sketch graphs will rely on being able to sketch some kind of *representation* of them. Observe that most of the ways we represent graphs rely on regular data structures - in particular, matrices. The problem of sketching matrices is well-understood, and our graphs enforce a constraint that makes sketching them even easier. In particular, objects such as the adjacency matrix representing a graph have a **key constraint**: their elements are binary! This means that we only need the relatively coarse constraint of looking for nonzero elements in the matrix in order to generate an efficient sketch of a graph.

Conveniently, the problem of sketching vectors of numbers is well-understood in many different applications. In particular, the authors of the paper observed that the problem of ℓ_p sampling a vector is easily done,

Jowhari et al: “comb” sketching Input: vector $\mathbf{v}$ with some zero and nonzero elements Output: index i such that $\mathbf{v}[i]$ is nonzero with high probability Input like a snaggle-toothed comb; output returns a “tooth” of the comb Runtime: about $O(\log^2 n)$

Comb sketch

In particular, the following theorem holds from previous work in the field.

Theorem (Jowhari et al): For a vector x of length n, there exists a $O(\log^2 n)$-sized sketch $\mathcal{S}(x)$ such that: 1. $\mathcal{S}$ is *linear* - that is, there exists some operation $\oplus$ such that $\mathcal{S}(x) \oplus \mathcal{S}(y) = \mathcal{S}(x+y)$ for all y of length n, and 2. We can "easily" (in roughly constant time) obtain $i \leftarrow \mathcal{S}(x)$ such that $x_i \neq 0$

These sketches are called **comb sketches,** and there exist

We've almost created a sketch for graphs that will satisfy the properties above. Namely, consider the adjacency matrix A of a graph, and consider what happens when we comb-sketch each row. Since each row is of length V, the comb sketch of each row is of size $O(\log^2 V)$. We call the collection of rows a "comb sketch" of the matrix, and note that its total size is $O(V \log^2 V)$.

It turns out that when applied to the "vector of neighbors" $\mathbf{a}^v$ of a vertex v in a graph G, this second property has a useful graph-theoretic meaning - it allows us to easily find a neighbor v of a given vertex u in a graph. Our intuition will thus be as follows: to approximate connectivity in a graph, we must comb-sketch a matrix representation of the graph.

With some modification, the linearity property of the comb sketch becomes useful as well. In particular, if we can find a vector that somehow represents the neighbors of a graph, but also has the property that when the vectors of two neighboring vertices u and v are added together, the entry corresponding to their connection cancels out, but their outgoing edges add, we will be able to simulate the process of edge contraction.

It turns out that this is easy with a slight tweak to the standard adjacency matrix. Instead of the adjacency matrix, we will construct a larger matrix A_G, defined as follows. Define A_G to be an $n \times \binom{n}{2}$ matrix, with row set $[n]$ and column set $\binom{[n]}{2}$. Let A_G's columns thus be indexed by pairs of the form (i, j), where i is less than j. Then, define $A_G[i, (j, k)] = 1$ if $i = j$ and $(v_i, v_k) \in E$; -1 if $i = k$ and $(v_j, v_i) \in E$; 0 otherwise.

We claim that a key lemma holds when the matrix A_G is defined in this way. In particular,

Lemma (3.1 from paper). Let S be a subset of nodes in G, and let E_S denotes the set of edges crossing the cut $(S, V\S)$. Then the number of edges crossing this cut, $|E_S|$, is simply $\ell_0(\mathbf{x})$ where $\mathbf{x} = \sum_{v \in S} \mathbf{a}^v$.

Proof. Proving this is simple. In particular, note that any edge (u, v) in G can fall into one of four categories - both vertices u and v are in the set S; u is in S but v is not; u is not in S but v is; or both u and v lie outside the set S. Observe also that by definition, $\mathbf{a}[w, (i, j)] = 0$ if $w \neq i$ or $w \neq j$. Thus, every column corresponding to (i, j) has exactly two nonzero entries - a 1 in the ith row and a -1 in the jth row.

1. In the first case, both i and j are in S, so that (i, j) does not cross the cut.

Then,

$$\mathbf{x}[(i,j)] = \sum_{v \in S} \mathbf{a}^v[(i,j)] = a^i[(i,j)] + a^j[(i,j)] = (-1) + 1 = 0.$$

2. In the second case, i is in S but j is not, so that (i,j) crosses the cut. Then,

$$\mathbf{x}[(i,j)] = \sum_{v \in S} \mathbf{a}^v[(i,j)] = a^i[(i,j)] = 1.$$

3. In the third case, j is in S but i is not, so that (i,j) crosses the cut. Then,

$$\mathbf{x}[(i,j)] = \sum_{v \in S} \mathbf{a}^v[(i,j)] = a^j[(i,j)] = -1.$$

4. In the fourth case, neither i nor j is in S, and thus (i,j) does not cross the cut. Then,

$$\mathbf{x}[(i,j)] = \sum_{v \in S} \mathbf{a}^v[(i,j)] = 0.$$

Observe then that $\mathbf{x}[(i,j)] \neq 0$ iff (i,j) crosses the cut $(S, V\S)$. Thus, the nonzero elements of $\mathbf{x}$ correspond exactly with edges that connect a node in S with a node outside S.

Using this lemma, we will be able to simulate **edge contraction** in a graph.

One requirement of the comb sketches we have been using is that because they are probabilistic data structures, we cannot “use” them multiple times. To see why this would be contradictory, consider a sketch $\mathcal{S}(\mathbf{v})$ of a vector $\mathbf{v}$ containing nothing but nonzero elements. As long as there are nonzero elements in the vector, repeat the following algorithm - obtain an index i containing a nonzero element from the sketch $\mathcal{S}(\mathbf{v})$, then update the sketch by setting $\mathbf{v}[i]$ to zero. At the end of this process, the guarantees we made with regards to comb sketching will require the sketch to return all n original elements of the vector, and thus allow us to “retrieve” a vector of size $\Theta(n)$ from a sketch of size $O(\log^2 n)$. This is obviously impossible; in practice, the vector will start failing to return existing nonzero elements, and thus will no longer fulfill its requirements.

Thus, in practice, we will require queries to sketches to be **non-adaptive** when building algorithms that use them. Even more so, we will require that we not make repeated queries on a given sketch - since each sketch is probabilistic, making multiple draws from the sketch will affect the probabilities in a way that will be suboptimal. We thus require that for each “query” we make of a sketch, we draw a new copy of the sketch from the graph.

Contraction-based connectivity algorithm

In order to solve our problem of connectivity, we will need to find an algorithm that can partition the graph into connected components in our extremely limited setting - namely, finding an arbitrary neighbor v of a vertex u, and contracting a subset $S \subseteq G$. Fortunately, it turns out such an algorithm exists. Namely:

1. For each $u \in G$, pick a adjacent edge $(u, v) \in G$.
2. Using a disjoint-set data structure, register that u and v are in the same connected component.
3. Contract each of these edge, turning their adjacent nodes into "supernodes."
4. Repeat 1-3 until no more edges in G remain.

A formal correctness proof is beyond the scope of this paper, but it should be clear that contracting an edge does not change the nature of the connected components of a graph. Observe also that each "round" of the algorithm contracts at least $V/2$ nodes - thus, the discrepancy in the number of nodes $\hat{V} - cc(G)$ will at least halve with each round, and thus after $O(\log n)$ rounds, all edges will be removed and the connected components obtained.

To turn this into a sketch-based algorithm, we must observe that we cannot use just one sketch for the entire algorithm, as we will end up repeatedly resampling from it in order to find adjacent edges to our "supernodes." As observed earlier, this is unworkable. As a consequence, we end up creating one sketch for each round of the algorithm. The revised algorithm looks as follows:

- Construct $t = O(\log V)$ sketches $\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_t$ of M, of size $O(n \log^2 n)$ each.
- Initialize a disjoint-set data structure $\hat{V}$ with each node as a different connected components.
- For $i \in [t]$,
  - For each "supernode," or connected component, given by $n \in \hat{V}$, draw an edge bordering v using the sketch $\sum_{v \in n} \mathbf{a}_v$.
  - Register that the two nodes bordering the edge are in the same connected component.
  - Contract each of the edges drawn using the sketch.
- Simulate contraction of edge (i, j) from sketch $\mathcal{S}_k$ by computing $\mathcal{S}_{k+1}(\mathbf{a}_i) + \mathcal{S}_{k+1}(\mathbf{a}_j)$

Recap

This paper covered the *intersection* of several interesting and well-understood problems. In particular, linear sketching algorithms were well-understood (we used one scheme in our own sketch!); unbounded incremental graph algorithms had been proposed by Thorup and others; and very simple bounded, but *addition-only,* graph algorithms already existed. However, the problem of solving graph sketching in a fully-incremental context with bounded memory had not been solved, and this turned out to be the most interesting use case. Furthermore, the “tech transfer” of using the comb sketch in a graph context was extremely useful, as was the existence of a simple algorithms with a new “angle of attack” - namely that it used only contraction.