Lecture 13

Graph Streams: Basic Algorithms

13.1 Streams that Describe Graphs

We have been considering streams that describe data with some kind of structure, such as geometric structure, or more generally, metric structure. Another very important class of structured large data sets is *large graphs*. This motivates the study of graph algorithms that operate in streaming fashion: the input is a stream that describes a graph.

The model we shall work with in this course is that the input stream consists of tokens $(u, v) \in [n] \times [n]$, describing the edges of a simple[1] graph G on vertex set $[n]$. We assume that each edge of G appears exactly once in the stream. There is no easy way to check that this holds, so we have to take this as a promise. The number n is known beforehand, but m, the length of the stream and the number of edges in G, is not. Though we can consider both directed and undirected graphs in this model, *we shall only be studying problems on undirected graphs*; so we may as well assume that the tokens describe doubleton sets $\{u, v\}$.

Unfortunately, most of the interesting things we may want to compute for a graph provably $\Omega(n)$ space in this model, even allowing multiple passes over the input stream. We shall show such results when we study lower bounds, later in the course. These include such basic questions as "Is G connected?" and even "Is there a path from u to v in G?" where the vertices u and v are known beforehand. Thus, we have to reset our goal. Where $(\log n)^{O(1)}$ space used to be the holy grail for basic data stream algorithms, for several graph problems, it is $n(\log n)^{O(1)}$ space. Algorithms achieving such a space bound are sometimes called "semi-streaming" algorithms.[2]

13.2 The Connectedness Problem

Our first problem is: decide whether or not the input graph G, which is given by a stream of edges, is connected. This is a Boolean problem — the answer is either 0 (meaning "no") or 1 (meaning "yes") — and so we require an exact answer. We *could* consider randomized algorithms, but we won't need to.

For this problem, as well as all others in this lecture, the algorithm will consist of maintaining a subgraph of G satisfying certain conditions. For connectedness, the idea is to maintain a spanning forest, F, of G. As G gets updated, F might or might not become a tree at some point. Clearly G is connected iff it does.

The algorithm below maintains F as a set of edges. The vertex set is always $[n]$.

We have already argued the algorithm's correctness. Its space usage is easily seen to be $O(n \log n)$, since we always have $|F| \leq n - 1$, and each of F requires $O(\log n)$ bits to describe.

The well known UNION-FIND data structure can be used to do the work in the processing section quickly. To test

[1] A simple graph is one with no loops and no parallel edges.

[2] The term does not really have a formal definition. Some authors would extend it to algorithms running in $O(n^{3/2})$ space, say.

Initialize : $F \leftarrow \varnothing$, $X \leftarrow 0$;

Process $\{u, v\}$:

if $\neg X \ \wedge$ ($F \cup \{\{u, v\}\}$ *does not contain a cycle*) **then**
  $F \leftarrow F \cup \{\{u, v\}\}$;
  if $|F| = n - 1$ **then** $X \leftarrow 1$;

Output : X ;

acyclicity of $F \cup \{\{u, v\}\}$, we simply check if root(u) and root(v) are distinct in the data structure. Note that this algorithm assumes an*insertion-only* graph stream: edges only arrive and never depart from the graph. All algorithms in this lecture will make this assumption.

13.3 The Bipartiteness Problem

A bipartite graph is one whose vertices can be partitioned into two disjoint sets, S and T say, so that every edge is between a vertex in S and a vertex in T. Equivalently, a bipartite graph is one whose vertices can be properly colored using two colors.[3] Our next problem is to determine whether the input graph G is bipartite.

Note that being bipartite is a *monotone* property (just as connectedness is): that is, given a non-bipartite graph, adding edges to it cannot make it bipartite. Therefore, once a streaming algorithm detects that the edges seen so far make the graph non-bipartite, it can stop doing more work. Here is our proposed algorithm.

Initialize : $F \leftarrow \phi$, $X \leftarrow 1$;

Process $\{u, v\}$:

if X **then**
  if $F \cup \{\{u, v\}\}$ *does not contain a cycle* **then**
    $F \leftarrow F \cup \{\{u, v\}\}$;
  else if $F \cup \{\{u, v\}\}$ *contains an odd cycle* **then**
    $X \leftarrow 0$;

Output : X ;

Just like our connectedness algorithm before, this one also maintains the invariant that F is a subgraph of G and is a forest. Therefore it uses $O(n \log n)$ space. Its correctness is guaranteed by the following theorem.

Theorem 13.3.1. *The above algorithm outputs* 1 *iff the input graph G is bipartite.*

Proof. Suppose the algorithm outputs 0. Then G must contain an odd cycle. This odd cycle does not have a proper 2-coloring, so neither does G. Therefore G is not bipartite.

Next, suppose the algorithm outputs 1. Let $\chi : [n] \to \{0, 1\}$ be a proper 2-coloring of the final forest F (such a χ clearly exists). We claim that χ is also a proper 2-coloring of G, which would imply that G is bipartite and complete the proof.

To prove the claim, consider an edge $e = \{u, v\}$ of G. If $e \in F$, then we already have $\chi(u) \neq \chi(v)$. Otherwise, $F \cup \{e\}$ must contain an even cycle. Let π be the path in F obtained by deleting e from this cycle. Then π runs between u and v and has odd length. Since every edge on π is properly colored by χ, we again get $\chi(u) \neq \chi(v)$. □

[3] A coloring is proper if, for every edge e, the endpoints of e receive distinct colors.

13.4 Shortest Paths and Distance Estimation via Spanners

Now consider the problem of estimating the distance in G between two vertices that are revealed after the input stream has been processed. That is, build a small data structure, in streaming fashion, that can be used to answer queries of the form "what is the distance between x and y?"

Define $d_G(x, y)$ to be the distance in G between vertices x and y:

$$d_G(x, y) \;=\; \min\{\text{length}(\pi) : \pi \text{ is a path in } G \text{ from } x \text{ to } y\},$$

where the minimum of an empty set defaults to ∞. The following algorithm computes an estimate $\hat{d}(x, y)$ for the distance $d_G(x, y)$. It maintains a suitable subgraph H of G which, as we shall see, satisfies the following property.

$$\forall\, x, y \in [n] :\; d_G(x, y) \;\le\; d_H(x, y) \;\le\; t \cdot d_G(x, y)\,, \tag{13.1}$$

where $t \ge 1$ is an integer constant. A subgraph H satisfying (13.1) is called a *t-spanner* of G. Note that the left inequality trivially holds for every subgraph H of G.

Initialize : $H \leftarrow \varnothing$;

Process $\{u, v\}$:
1 **if** $d_H(u, v) \ge t + 1$ **then**
2 $\quad H \leftarrow H \cup \{\{u, v\}\}$;

Output : On query (x, y), report $\hat{d}(x, y) = d_H(x, y)$;

We now show that the final graph H constructed by the algorithm is a t-spanner of G. This implies that the estimate $\hat{d}(x, y)$ is a t-approximation to the actual distance $d_G(x, y)$: more precisely, it lies in the interval $[d_G(x, y),\, t \cdot d_G(x, y)]$.

Pick any two distinct vertices $x, y \in [n]$. We shall show that (13.1) holds. If $d_G(x, y) = \infty$, then clearly $d_H(x, y) = \infty$ as well, and we are done. Otherwise, let π be the shortest path in G from x to y, and let $x = v_0, v_1, v_2, \ldots, v_k = y$ be the vertices on π, in order. Then $d_G(x, y) = k$.

Pick an arbitrary $i \in [k]$, and let $e = \{v_{i-1}, v_i\}$. If $e \in H$, then $d_H(v_{i-1}, v_i) = 1$. Otherwise, $e \notin H$, which means that at the time when e appeared in the input stream, we had $d_{H'}(v_{i-1}, v_i) \le t$, where H' was the value of H at that time. Since H' is a subgraph of the final H, we have $d_H(v_{i-1}, v_i) \le t$. Thus, in both cases, we have $d_H(v_{i-1}, v_i) = t$. By the triangle inequality, it now follows that

$$d_H(x, y) \;\le\; \sum_{i=1}^{k} d_H(v_{i-1}, v_i) \;\le\; tk \;=\; t \cdot d_G(x, y)\,,$$

which completes the proof, and hence implies the quality guarantee for the algorithm that we claimed earlier.

How much space does the algorithm use? Clearly, the answer is $O(|H| \log n)$, for the final graph H constructed by it. To estimate $|H|$, we note that, by construction, the shortest cycle in H has length at least $t + 2$. We can then appeal to a result in extremal graph theory to upper bound $|H|$, the number of edges in H.

13.4.1 The Size of a Spanner: High-Girth Graphs

The *girth* $\gamma(G)$ of a graph G is defined to be the length of its shortest cycle; we set $\gamma(G) = \infty$ if G is acyclic. As noted above, the graph H constructed by our algorithm has $\gamma(H) \ge t + 2$. The next theorem places an upper bound on the size of a graph with high girth (see the paper by Alon, Hoory and Linial [AHL02] and the references therein for more precise bounds).

Theorem 13.4.1. *Let n be sufficiently large. Suppose the graph G has n vertices, m edges, and $\gamma(G) \ge k$, for an integer k. Then*

$$m \;\le\; n + n^{1+1/\lfloor \frac{k-1}{2} \rfloor}\,.$$

Proof. Let $d = 2m/n$ be the average degree of G. If $d \leq 3$, then $m \leq 3n/2$ and we are done. Otherwise, let F be the subgraph of G obtained by repeatedly deleting from G all vertices of degree less than $d/2$. Then F has minimum degree at least $d/2$, and F is nonempty, because the total number of edges deleted is less than $n \cdot d/2 = m$.

Put $\ell = \lfloor \frac{k-1}{2} \rfloor$. Clearly, $\gamma(F) \geq \gamma(G) \geq k$. Therefore, for any vertex v of F, the ball in F centered at v and of radius ℓ is a tree (if not, F would contain a cycle of length at most $2\ell \leq k-1$). By the minimum degree property of F, when we root this tree at v, its branching factor is at least $d/2 - 1 \geq 1$. Therefore, the tree has at least $(d/2-1)^\ell$ vertices. It follows that

$$n \;\geq\; \left(\frac{d}{2} - 1\right)^\ell \;=\; \left(\frac{m}{n} - 1\right)^\ell ,$$

which implies $m \leq n + n^{1+1/\ell}$, as required. □

Using $\lfloor \frac{k-1}{2} \rfloor \geq \frac{k-2}{2}$, we can weaken the above bound to

$$m \;=\; O\left(n^{1+2/(k-2)}\right) .$$

Plugging in $k = t + 2$, we see that the t-spanner H constructed by our algorithm has $|H| = O(n^{1+2/t})$. Therefore, the space used by the algorithm is $O(n^{1+2/t} \log n)$. In particular, we can 3-approximate all distances in a graph by a streaming algorithm in space $\widetilde{O}(n^{5/3})$.