Graph Connectivity Sketches
(Lecture 13 and Part of Lecture 14)

DS-563 / CD-543 @ Boston University
Instructor: Krzysztof Onak

Spring 2024

1 Problem

Input: a stream describing a graph G on $V = [n]$

Question: Is G connected?

We consider two versions of the problem: insertion only (the input stream is a sequence of edges that are never deleted) and insertion–deletion (the input stream is a sequence of updates of the form "`insert` (u, v)" and "`delete` (u, v)" with no edge deleted before it is inserted).

2 Insertion–only streams

We keep a subset F of edges that is a spanning forest of the graph we have seen so far. Initially, $F = \emptyset$. For every edge (u, v) that we see, if u and v are already connected by F, we do nothing. Otherwise, we add this edge to F. At the end of the stream, G is connected if and only if all vertices are connected by F.

Space usage: $O(n)$ words of space, because F consists of at most $n - 1$ edges.

Note: For fast processing, instead of storing explicitly F, you can use the union–find data structure.

3 Insertion–deletions streams

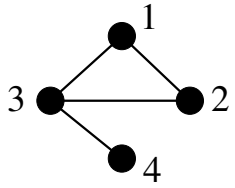
3.1 First attempts

- Sample edges and see what has not been deleted: the graph might become very dense and then have lots of deletions, so this won't work.
- Is a graph that is very sparse at the end of the stream the worst case then? Not really. We have not covered this topic in detail, but if the final graph has k edges, then it can be fully recovered, using $O(k \operatorname{polylog}(n))$ words of space. This can be achieved using a set of techniques known as *sparse recovery* or *compressed sensing*.

3.2 Encoding of the graph

We encode adjacency lists of every vertex as a vector of length $\binom{n}{2}$. Every entry corresponds to a single pair of vertices in $V = [n]$ and is indexed by (unordered) pairs $\{j, j'\}$, where $j, j' \in V$ and are different. For a given vertex $i \in V$, we create a vector x_i, such that $(x_i)_{\{j,j'\}}$, the entry indexed by $\{j, j'\}$, is non-zero if and only if $i \in \{j, j'\}$ and $\{j, j'\}$ is present in the graph. In other words, the entry corresponding to a specific edge is non-zero if this edge is incident on i and present in the graph. If this entry is non-zero, it is either -1 or 1. More specifically, $(x_i)_{\{i,j\}} = -1$ if $j < i$ and $(x_i)_{\{i,j\}} = 1$ if $j > i$.

Example:



$$
\begin{array}{cccccccccc}
 & & & \{1,2\} & \{1,3\} & \{1,4\} & \{2,3\} & \{2,4\} & \{3,4\} & \\
x_1 & = & (& 1 & 1 & 0 & 0 & 0 & 0 &) \\
x_2 & = & (& -1 & 0 & 0 & 1 & 0 & 0 &) \\
x_3 & = & (& 0 & -1 & 0 & -1 & 0 & 1 &) \\
x_4 & = & (& 0 & 0 & 0 & 0 & 0 & -1 &)
\end{array}
$$

Note that most entries are 0. If a specific edge is not present in the graph, all entries in the column corresponding to this edge are zero. Otherwise, only two of them are non-zero, i.e., those corresponding to the endpoints of the edge. Moreover, one of them is -1 and the other one is 1.

The last property has very useful consequences. Namely, these vectors can be combined to represent the connectivity of a subset of vertices. In the rest of this note, we write x_S for any subset S of the vertices to denote $\sum_{i \in S} x_i$.

Claim: For any subset S of vertices and any pair $\{j, j'\}$ of vertices, the entry corresponding to $\{j, j'\}$ in x_S is non-zero if and only if $\{j, j'\}$ is present in the graph and connects S with $V \setminus S$.

Proof sketch: Let $e = \{j, j'\}$. First, if e is not present in the graph, the entries corresponding to e are 0 in all vectors x_i, and hence the corresponding entry in x_S is 0 as well, as desired. It remains to show that the claim holds if e is present in the graph. Consider three cases. If e connects a vertex in S to a vertex not in S, then exactly one of the non-zero entries corresponding to e is included in the summation and the corresponding entry in x_S is non-zero as well, which is what what we hoped for. Otherwise, if e connects vertices in $V \setminus S$, no non-zero entry corresponding to e ends up in the summation, and the corresponding entry in x_S is zero, as desired. Finally, if both endpoints of e belong to S, the only non-zero entries corresponding to e are included in the summation and they cancel each other out, which finishes the proof. □

3.3 Borůvka's algorithm

We build on ideas from Borůvka's algorithm. In particular, consider Algorithm 1, which is a parallel algorithm for connectivity. It has the following useful property.

Claim: Before the i-th iteration of the **repeat** loop, each component is either a maximal connected component or its size is at least 2^{i-1}.

Proof sketch: Before we start the first iteration, the size of each component is $1 = 2^0$ and therefore each component is trivially connected in the underlying graph. Suppose now that the claim holds before iteration i. Consider a component C that exists after the iteration. If it exists before the iteration and does not change during it, then C has no edges connecting it to any vertex outside of C and is already a maximal connected component. If C is a new component after iteration i, it is a result of merging two or more components that existed before the iteration. Each of them is connected in the underlying graph and since

Algorithm 1: An algorithm for discovering connected components

```
foreach vertex v do
    create a component of size 1 that contains only v
repeat
    foreach component C do
        select an arbitrary edge connecting C to the rest of the graph (if there is such an edge)
    merge components that are connected via selected edges
```

they are connected with graph edges, their union is connected as well. Moreover, since they are not maximal connected components, the size of each of them is at least 2^{i-1} and the size of their union has to be at least twice as much, i.e., $2 \cdot 2^{i-1} = 2^i$. □

Corollary: The algorithm can be stopped after $\lceil \log n \rceil$ iterations of the loop, since the components will not change after that. Components constructed by the algorithm will at this point be exactly the connected components of the graph.

3.4 ℓ_0–sampling

We use the following tool that allows for extracting a non-zero coordinate of a vector.

> There is a linear sketching algorithm that takes a vector in $\{-n, \ldots, n\}^n$ and turns it into $\mathrm{polylog}(n)$ bits. For any v in the allowed range, the algorithm can correctly report a non-zero coordinate of v if v is non-zero—or report that v is an all–zero vector—with probability at least $1 - n^{-3}$.

These types of algorithms are usually constructed so they do not only report a non-zero coordinate, but they report a *uniformly random* non-zero coordinate, which may be important in some applications. This is why we refer to this algorithm as ℓ_0-*sampling*.

We won't show how to construct it here, but one could for instance sample coordinates with probabilities 2^{-i} for a logarithmic number of different settings of i. For some setting of i, we are likely to sample just one coordinate, and then we can verify that this is just a single coordinate. If this is the case, we can restore both the index of the coordinate and its magnitude, by adding these values for all coordinates that were sampled.

3.5 Putting it all together

In order to find out whether our graph is connected, we simulate Borůvka's algorithm. For each iteration of Borůvka's algorithm—and recall we need only $O(\log n)$ of them—we run an independent instance of the ℓ_0–sampling algorithm and apply the linear transformation it yields to all vectors x_i. Overall, this requires $O(\log n) \cdot n \cdot \mathrm{polylog}(n) = O(n \,\mathrm{polylog}(n))$ space.

Let us now describe how we use these sketches. In the first iteration, we start with a separate component for each vertex. We have sketches for each x_i, $1 \leq i \leq n$, provided by the first ℓ_0–sampling algorithm. We apply this ℓ_0–sampling algorithm to discover one edge connecting each single–vertex component to the rest of the graph. We then use these edges to merge components. What is the probability that we fail to correctly discover an edge for any of the components? By the union bound, it is at most $n \cdot \frac{1}{n^3} = \frac{1}{n^2}$.

To simulate the second, or any later, iteration, we need to discover edges connecting each of the components to the rest of the graph (if there are such edges). We use the fact that the ℓ_0–sampling algorithm is a linear sketching algorithm. For a given component C, the sketch for x_C is the sum of sketches for each $i \in C$, which we already have. Once we compute the sketch for x_C for each component C, we apply the ℓ_0 sampling procedure to each of them to discover an edge connecting C to the rest of the graph if there is such an edge. Then we can merge components connected with the discovered edges. The probability that the ℓ_0 procedure fails to correctly identify edges connecting any of the components C with the rest of the graph—or to correctly detect that C has no such connections—is at most (number of components)$\cdot \frac{1}{n^3} \leq \frac{1}{n^2}$.

We repeat this approach over all iterations of our simulation of Borůvka's algorithm. Overall, the probability that any query to any instance of the ℓ_0–sampling algorithm fails is at most $\lceil \log n \rceil \cdot \frac{1}{n^2} = o\left(\frac{1}{n}\right)$.

Note that we need $O(\log n)$ copies of the ℓ_0–sampling algorithm and corresponding sketches for x_i's, because each of them is guaranteed to work properly only for non-adaptive queries, and we have $O(\log n)$ rounds/iterations, each consisting of a set of non-adaptive queries about all current components.[1]

3.6 An additional distributed application

Note that this algorithm can applied in the following communication setting. Suppose that there are n players, each knowing the adjacency list of a single vertex. Perhaps it's the local connectivity of this player. Then each of them can send a $\mathrm{polylog}(n)$–bit message to a single player, who can then find out whether all the players are connected. The only requirement is that all the players have shared randomness needed to initialize the instances of the ℓ_0–sampling algorithm.

It is interesting that, for connectivity, the entire graph (i.e., potentially $\Theta(n^2)$ bits of information) can be reduced to $O(n \,\mathrm{polylog}(n))$ information. In particular, the adjacency list of each neighborhood of size $n-1$ can be reduced to $O(\mathrm{polylog}(n))$ bits, independently of other neighborhoods, as long as all players share a common source of randomness.

4 Extensions

This approach can be extended to the following problems (which may or may not be covered in the discussion section):

- outputting a spanning forest
- reconstructing a minimum spanning tree
- k–connectivity[2]

5 A lower bound for adversarially robust streaming algorithms

In our earlier discussion of adversarially robust streaming algorithms, we focused on the problem of estimating the number of distinct elements in the stream. It is natural to ask whether all streaming algorithms have

[1] See our discussion of adversarially robust streaming algorithms in earlier lectures for more information on what could go wrong otherwise.

[2] For instance, finding all the bridges in the graph, where an edge is a bridge if removing it increases the number of connected components.

an adversarially robust version that does not use much more space (perhaps only a factor of $\mathrm{polylog}(N)$ more, where N is the maximum of the range of numbers and the length of the stream).

It turns out that this is not possible for connectivity sketches. Suppose you had connectivity sketches of size $n\,\mathrm{polylog}(n)$ that can handle n^2 updates while answering n^2 adaptive queries in the meantime. We are not making this very formal here, but this would imply that one could recover the entire graph from such a connectivity sketch, which would mean that one could compress $\Theta(n^2)$ bits of information into $O(n\,\mathrm{polylog}(n))$ bits, which is not possible. This could be achieved by discovering edges adjacent to each vertex and removing them one by one.