Graph Connectivity Sketches

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

Spring 2025

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, 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* (aka. *compressed sensing*).

3.2 Overview: Three ingredients

We obtain our solution by combining three ingredients, which we introduce one by one:

1. **An encoding of a graph:** We express the graph as a matrix with $\Theta(n^3)$ $\{-1, 0, 1\}$–entries. Each row of the matrix expresses the connectivity of a single vertex. While this encoding may seem wasteful, it has properties which allow for expressing the connectivity of any set of vertices to its complement by a linear combination of rows of the matrix.

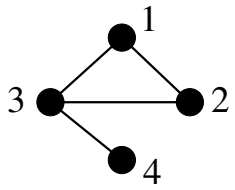
2. **Borůvka's algorithm:** Two popular algorithms for finding the minimum spanning trees are Prim's and Kruskal's algorithms. Borůvka's algorithm is another algorithm for this problem, which guarantees more "parallelism." We will see that it is a crucial property, which will allows for constructing a spanning tree or forest for the input graph in a small number of non-adaptive rounds.

3. **ℓ_0–sampling:** We won't cover exact implementation details of this technique here, but it allows for compressing high–dimensional vectors into a low–dimensional linear sketch that is sufficient for recovering at least one non-zero entry with high probability. We apply this to rows of our graph's encoding to significantly reduce the amount of information we need to store.

3.3 Ingredient 1: The graph's encoding and its properties

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 to 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. □

3.4 Ingredient 2: Borůvka's algorithm

We build on ideas from Borůvka's algorithm. In particular, consider Algorithm 1, which is a parallel algorithm for connectivity.

Algorithm 1: An algorithm for discovering connected components

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

This algorithm starts by creating a separate component for each vertex. Then, in the infinite loop, it keeps merging these components using selected edges. We say that a subset S of vertices is a *maximal connected component* if vertices that belong to it are all connected and there is no other vertex that is connected to S via an edge.

The main loop in our pseudocode is infinite, but we want to show that after a relatively small number of iterations, all the components are maximal connected components.

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 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 do not change after that. All components constructed by the algorithm are at this point maximal components, i.e., they are the connected components of the graph.

3.5 Ingredient 3: 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 $1 - n^{-3}$.
>
> Additionally, storing the algorithm's randomness requires $\mathrm{polylog}(n)$ bits and all the computation, both the sketch computation and reporting of a non-zero coordinate, can be performed in $O(\mathrm{polylog}(n))$ space.

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.

3.6 Putting it all together

In order to find out whether the input graph is connected, we want to simulate Borůvka's algorithm after seeing the entire stream. If we could explicitly compute the input graph's encoding—i.e., if we had the final vectors x_i for all vertices i—then we would be able to determine an edge connecting any set subset of vertices S to its complement $V \setminus S$ (if such an edge exists) by computing x_S and looking for a non-zero coordinate in x_S. However, we do not want to explicitly store and maintain the graph's encoding since this would require $n^{\Omega(1)}$ space, and in particular, doing it directly would take $\Theta(n^3)$ space due to the length of vectors x_i. To address this, we do not store x_i's, but instead maintain their linear sketches provided by the ℓ_0–sampling algorithm. This allows for both getting a sketch of x_S by summing the sketches of x_i for $i \in S$ as well as determining a non-zero coordinate of x_S, for any fixed S, with high probability.

We can now break the computation up into two phases:

- **Computing ℓ_0–sketches of x_i's during a pass over the stream:** For each iteration of the **repeat** loop in Borůvka's algorithm—recall that only $O(\log n)$ of them are needed—we maintain an independent instance of the ℓ_0–sketching algorithm with a sketch of each of x_i's. (The need for separate sketches for each iteration becomes more clear later, but the main reason is that they only provide good guarantees with high probability for a fixed vector and we want an independent instance for each iteration that is likely to work with components determined by the previous iterations.)

  Initially, all x_i's are zero vectors, and so are their sketches. When an edge (u, v) is inserted or removed, only two coordinates in all of x_i's change: $(x_u)_{\{u,v\}}$ and $(x_v)_{\{u,v\}}$, which means that only sketches for x_u and x_v have to be updated. This is achieved by taking advantage of the linearity of the sketches. We compute the linear sketches of the updates to x_u and x_v, which are both zero vectors except one coordinate, and add them to the previous sketches for x_u and x_v, respectively. This way we maintain the ℓ_0–sketches for all x_i's in $O(n \operatorname{polylog}(n))$ space.

- **Simulating Borůvka's algorithm, using the sketches:** We now describe how we simulate the required $\lceil \log n \rceil$ iterations of Borůvka's algorithm. 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 existing 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 (whenever such edges exist). For the j-th iteration, we use the sketches computed by the j-th ℓ_0–sampling algorithm. Since this algorithm is a linear sketching algorithm, we can compute the sketch of x_C for any component C by summing sketches x_i for all $i \in C$. Once we compute the sketch for x_C for each component C created in the previous iteration, we

apply the ℓ_0 sampling procedure to each of them to discover a non-zero coordinate, which corresponds to an edge connecting C to the rest of the graph, or to detect that the vector is an all zero vector.

Now, what is the probability that the j-th ℓ_0–sampling algorithm fails to correctly identify an edge connecting some component C created in the first $j-1$ iteration with the rest of the graph or to correctly detect that this component has no such connection (i.e., it is maximal)? By its properties, the algorithm computes a correct answer with probability at least $1 - n^{-3}$ for any *fixed* vector. By using an independent instance of the ℓ_0–sampling algorithm for each iteration, we ensure that the set of vectors x_C, which are determined by the current components C, that we use with a given instance is determined (i.e., fixed) before we start using this instance of ℓ_0–sampling. Hence by the union bound, the probability that the algorithm fails on any of them is at most

$$(\text{number of components}) \cdot \frac{1}{n^3} \leq \frac{1}{n^2}.$$

By applying the union bound over all simulated iterations of Borůvka's algorithm, we bound the probability that any of the instances of the ℓ_0–sampling algorithm fails at some point by

$$\lceil \log n \rceil \cdot \frac{1}{n^2} = o\left(\frac{1}{n}\right).$$

An additional note on adaptive use of randomized algorithms: A somewhat subtle but important issue here was the fact that the guarantee of the algorithm that we use only holds with high probability for any fixed vector. The problem is that if we kept using the same ℓ_0–sampling sketches throughout the simulation of Borůvka's algorithm, vectors x_C would not be fixed in advance, but would correspond to components C that are a function of internal randomness of the sketching algorithm. Hence, these vectors would not be fixed in advance. We get around this issue by using Borůvka's algorithm, which allows us to reduce the connectivity question to $O(\log n)$ rounds of adaptivity, in which all vectors that we consider are a function of only previous rounds. This is a property that DFS or BFS would not provide as their level of adaptivity can be as large as $\Omega(n)$.

This is a topic that we will discuss next in this course. More specifically, we will explore the framework of *adversarially robust streaming algorithms*, which can be used on adaptive data streams.

3.7 [Bonus, not discussed in class] 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 $\text{polylog}(n)$–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 $O(n \,\text{polylog}(n))$ information. In particular, the adjacency list of each neighborhood of size $n-1$ can be reduced to $O(\text{polylog}(n))$ bits, independently of other neighborhoods, as long as all players share a common source of randomness.

4 [Bonus, not discussed in class] Extensions

This approach can be extended to many connectivity related questions, including the following:

- outputting a spanning forest,
- reconstructing a minimum spanning tree,
- k–connectivity.[1]

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