Chapter 3

Connected Components and Minimum Spanning Tree

In this chapter, we look at the problem of computing connected components and minimum spanning trees. It is relatively easy to find solve them in strongly superlinear and strongly sublinear memory regimes. In the first two sections of the chapter, we focus our efforts on solving MST in near linear memory regime. Next, we discuss a connectivity algorithm due to Andoni et al. [ASS+18] that yields a round complexity depending on the diameter of the graph, under the strongly sublinear memory regime. Finally, we discuss a constant round algorithm due to Andoni et al. [ANOY14] to compute geometric minimum spanning trees under the strongly sublinear memory regime.

Problem definitions Given an undirected graph $G = (V, E)$, a connected component is a subgraph in which any two vertices are connected by a path. There are two typical ways to represent connected components on a graph:

1. Every vertex v stores a label $l(v)$ such that $l(u) = l(v)$ if and only if vertices u and v are in the same connected component.

2. Explicitly build a tree for each connected component, yielding a maximal forest of the graph.

A related graph problem is computing the minimum spanning tree (MST) on connected weighted graphs. The goal of MST is to find a subset T of edges such that the sum of edge weights in T are minimized. In general, where the graph may not be connected, the MST problem is also

called the minimum spanning forest (MSF) problem. On graphs with identical edge weights, we see that computing a minimum spanning forest is equivalent to computing connected components.

The three memory regimes One can compute MST in $O(1)$ rounds under the strongly superlinear memory regime using the *filtering* technique of Lattanzi et al. [LMSV11]. As the idea is similar to Section 2.1, we leave the details as an exercise.

Exercise 3.1. MST with strongly superlinear memory in $O(1)$ rounds *Devise an algorithm that computes the minimum spanning tree of a given graph* $G = (V, E)$ *with* $|V| = n$ *in* $O(\frac{1}{\epsilon})$ *rounds using* $O(n^{1+\epsilon})$ *memory per machine, for some constant* $\epsilon > 0$.

On the other extreme, MST can be computed in $O(\log n)$ rounds with strongly sublinear memory by carefully simulating Borůvka[1]. Note that a long chain of proposed edges may occur while simulating Borůvka, where merging them could take many rounds. This issue can be resolved by using randomness to drop a constant fraction of proposed edges, in a manner that remaining edges do not form long chains. We leave the details as an exercise.

Exercise 3.2. MST with strongly sublinear memory in $O(\log n)$ rounds *Devise an algorithm that computes the minimum spanning tree of a given graph* $G = (V, E)$ *with* $|V| = n$ *in* $O(\frac{1}{\epsilon})$ *rounds using* $O(n^{\alpha})$ *memory per machine, for some constant* $\alpha \in (0, 1)$. *You may assume that edge weights are unique.*

In Section 3.1, we see an algorithm for computing minimum spanning trees in $O(1)$ rounds in the near linear memory regime, where each machine has $S = \widetilde{O}(n)$ memory. It assumes the existence of an algorithm `connected-components` which computes connected components in $O(1)$ rounds in the near linear memory regime. In Section 3.2, we construct such an algorithm `connected-components` using *graph sketching*.

Known lower bounds It remains a major open question whether there exist connectivity MPC algorithms with sub-logarithmic time. There are strong indications that such algorithms do not exist: Beame et al. [BKS13] show logarithmic lower bounds for restricted algorithms, and no known algorithm can distinguish a n-node cycle from two $\frac{n}{2}$-node cycles

[1] See https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm

in $o(\log n)$ rounds. Furthermore, Roughgarden et al. [RVW18, Theorem 6.1] showed that an unconditional lower bound would imply stronger circuit lower bounds.

Recently, there has been some work using parameters of the input graph such as diameter D of the graph: Andoni et al. [ASS+18] gave an algorithm in the strongly sublinear memory regime that uses $\Theta(m)$ total memory to compute connected components in $\mathcal{O}(\log D \cdot \log\log_{\frac{m}{n}} n)$ rounds.

3.1 MST using near linear memory

Recall from Exercise 2.1 that sorting can be done in $\mathcal{O}(1)$ rounds with strongly sublinear memory. Hence, we can sort edges in ascending weight ordering in $\mathcal{O}(1)$ rounds and label the edges $e_1, e_2, \ldots, e_m$ such that $w(e_1) \leqslant w(e_2) \leqslant \cdots \leqslant w(e_m)$. In the sequential setting, Kruskal's algorithm[2] iterates through such a sorted edge list to decide whether an edge is in the MST. This gives the following observation, which we shall exploit.

Observation Edge e_i is in the MST if and only if its endpoints are not in the same connected components in the graph with edge set $\{e_1, \ldots, e_{i-1}\}$.

Using the observation directly to determine whether all edges are in the MST would require $\mathcal{O}(m)$ calls to `connected-components`. To speed things up in MPC, we group edges into chunks of n edges. In the near linear memory regime, each chunk can fit into a single machine and all chunks can be processed simultaneously. For $i = \{1, \ldots, \frac{m}{n}\}$, denote

- $E_i = \{e_{(i-1)n+1}, \ldots, e_{in}\}$ as the i^{th} chunk of n edges
- $E'_i = \cup_{j=1}^{i} E_j$ as the union of edge sets E_1 to E_i
- F'_i as the maximal forest computed from each set of edges E'_i

Denote F'_0 as the graph without edges (i.e. all vertices are isolated). By the above observation, we know that any edge in $\{u, v\} \in E_i$ is in the MST only if the components of u and v in F'_{i-1} differ. Since both $|E_i| \in \mathcal{O}(n)$ and $|F'_{i-1}| \in \mathcal{O}(n)$, a single machine can simultaneously hold E_i and F'_{i-1}.

Assuming that there exists an algorithm `connected-components` which computes connected components in $\mathcal{O}(1)$ rounds in near linear memory

[2] See https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

regime, then all maximal forests F'_i can be computed in $O(1)$ rounds in parallel. Then, by putting E_i and F'_{i-1} into machine i, all MST edges can be determined in a one further MPC round. In Section 3.2, we construct such an algorithm `connected-components` using *graph sketching*.

3.2 Connectivity using near linear memory

We first describe the technique of *graph sketching* then explain how to use it in MPC to compute connected components in $O(1)$ rounds with near linear memory.

3.2.1 Graph sketching

Graph sketching is a technique first developed in the context of streaming algorithms. In streaming problems, updates appear one-by-one, and one has maintain/compute certain properties or data structures under limited memory. That is, one cannot store entire stream and compute offline. The length of the stream may also be unknown. For the connected components problem, edges can be added or deleted in the stream. Below, we show a randomized algorithm for computing a maximal forest with high success probability. For the streaming setting, it uses a data structure with $O(n \log^4 n)$ memory. In the near linear memory regime, this will fit into a single machine.

Coordinator model For a change in perspective[3], consider the following computation model where each vertex acts independently from each other. Then, upon request of connected components, each vertex sends some information to a centralized coordinator to perform computation and outputs the maximal forest.

The coordinator model will be helpful in our analysis of the algorithm later as each vertex will send $O(\log^4 n)$ amount of data (a local sketch of the graph) to the coordinator, totalling $O(n \log^4 n)$ memory as required. This conceptual model also suggests how to perform graph sketch in MPC.

Two warm ups Before we give the full construction, we first look at two warm up problems. Fix a subset $A \subseteq V$ and look at the cut C between A

[3]In reality, the algorithm simulates all the vertices' actions so it is not a real multi-party computation setup.

and $V \setminus A$. In the first warm up, we assume there is only one cut edge across C, and wish to find it using $\mathcal{O}(\log n)$ bits of memory. Building upon the previous warm up, the second warm up wishes to find *any* cut edge across the cut C when there are $k > 1$ cut edges.

Warm up 1: Finding the single cut edge

Definition 3.1 (The single cut problem)**.** *Fix an arbitrary subset $A \subseteq V$. Suppose there is exactly 1 cut edge $\{u, v\}$ between A and $V \setminus A$. How do we output the cut edge $\{u, v\}$ using $\mathcal{O}(\log n)$ bits of memory?*

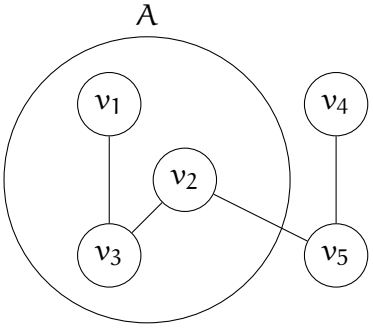
Without loss of generality, assume $u \in A$ and $v \in V \setminus A$. Note that this is not a trivial problem on first glance since it already takes $\mathcal{O}(n)$ bits for any vertex to enumerate all adjacent edges. To solve the problem, we use a *bit trick* which exploits the fact that any edge $\{a, b\} \in A$ will be considered twice by vertices in A. Since one can uniquely identify each vertex with $\mathcal{O}(\log n)$ bits, consider the following:

- Identify an edge by the concatenation the identifiers of its endpoints. Say, $\mathrm{id}(\{u, v\}) = u \circ v$ if $\mathrm{id}(u) < \mathrm{id}(v)$.
- Locally, each vertex u computes $\mathrm{XOR}_u = \oplus_{e \in E, e \ni u} \mathrm{id}(e)$.
- Vertices send the coordinator their XOR sum.
- Coordinator computes $\mathrm{XOR}_A = \oplus_{u \in A} \mathrm{XOR}_u$.

Example Suppose $V = \{v_1, v_2, v_3, v_4, v_5\}$ where $\mathrm{id}(v_1) = 000$, $\mathrm{id}(v_2) = 001$, $\mathrm{id}(v_3) = 010$, $\mathrm{id}(v_4) = 011$, and $\mathrm{id}(v_5) = 100$. Then,

$$\mathrm{id}(\{v_1, v_3\}) = \mathrm{id}(v_1) \circ \mathrm{id}(v_3) = 000010$$

and so on. Suppose we query for the cut edge $\{v_2, v_5\}$ with $A = \{v_1, v_2, v_3\}$.



Then,

$$\begin{aligned}
\mathrm{XOR}_1 &= 000010 = \mathrm{id}(\{v_1, v_3\}) && = 000010\\
\mathrm{XOR}_2 &= 000110 = \mathrm{id}(\{v_2, v_3\}) \oplus \mathrm{id}(\{v_2, v_5\}) && = 001010 \oplus 001100\\
\mathrm{XOR}_3 &= 001000 = \mathrm{id}(\{v_2, v_3\}) \oplus \mathrm{id}(\{v_1, v_3\}) && = 001010 \oplus 000010\\
\mathrm{XOR}_4 &= 011100 = \mathrm{id}(\{v_4, v_5\}) && = 011100\\
\mathrm{XOR}_5 &= 010000 = \mathrm{id}(\{v_4, v_5\}) \oplus \mathrm{id}(\{v_2, v_5\}) && = 011100 \oplus 001100
\end{aligned}$$

Thus, $\mathrm{XOR}_A = \mathrm{XOR}_1 \oplus \mathrm{XOR}_2 \oplus \mathrm{XOR}_3 = 000010 \oplus 000110 \oplus 001000 = 001100 = \mathrm{id}(\{v_2, v_5\})$ as expected. Notice that every edge in A contributes an even number of times to the coordinator's XOR sum.

Claim 3.2. $\mathrm{XOR}_A = \oplus_{u \in A} \mathrm{XOR}_u$ *is the identifier of the cut edge.*

Proof. For any edge $\{a, b\}$ such that $a, b \in A$, $\mathrm{id}(\{a, b\})$ is in both XOR_a and XOR_b. So, $\mathrm{XOR}_a \oplus \mathrm{XOR}_b$ will cancel out the contribution of $\mathrm{id}(\{a, b\})$. Hence, the only remaining value in $\mathrm{XOR}_A = \oplus_{u \in A} \mathrm{XOR}_u$ will be the cut edge since only one endpoint lies in A. □

Remark Similar ideas are often used in the random linear network coding literature (e.g. Ho et al. [HMK+06]).

Warm up 2: Finding one out of $k > 1$ cut edges

Definition 3.3 (The k-cut problem)**.** *Fix an arbitrary subset $A \subseteq V$. Suppose there are exactly k cut edges (u, v) between A and $V \setminus A$, and we are given an estimate $\widehat{k}$ such that $\frac{\widehat{k}}{2} \leqslant k \leqslant \widehat{k}$. How do we output a cut edge $\{u, v\}$ using $\mathcal{O}(\log n)$ bits of memory, with high probability?*

A straight-forward idea is to independently mark each edge, each with probability $1/\widehat{k}$. In expectation, we expect one edge to be marked. Since edges are marked independently with probability $1/\widehat{k}$, the probability that a fixed cut edge $\{u, v\}$ is marked while no other edges are marked is $(1/\widehat{k})(1 - (1/\widehat{k}))^{k-1}$. Denote the marked cut edges by E', then

$$\begin{aligned}
\Pr[|E'| = 1] &= k \cdot (1/\widehat{k})(1 - (1/\widehat{k}))^{k-1} && \text{There are } k \text{ cut edges}\\
&\geqslant (\widehat{k}/2)(1/\widehat{k})(1 - (1/\widehat{k}))^{\widehat{k}} && \text{Since } \frac{\widehat{k}}{2} \leqslant k \leqslant \widehat{k}\\
&\geqslant (1/2) \cdot 4^{-1} && \text{Since } 1 - x \geqslant 4^{-x} \text{ for } x \leqslant 1/2\\
&\geqslant \frac{1}{10}
\end{aligned}$$

Remark The above analysis assumes that vertices can locally mark the edges in a consistent manner (i.e. both endpoints of any edge make the same decision whether to mark the edge or not). This can be done with a sufficiently large string of *shared randomness*. We discuss this later.

From above, we know that $\Pr[|E'| = 1] \geqslant 1/10$. If $|E'| = 1$, we can re-use the idea from the case when $k = 1$. However, if $|E'| \neq 1$, then XOR_A may correspond erroneously to another edge in the graph. In the above example, $\text{id}(\{v_1, v_2\}) \oplus \text{id}(\{v_2, v_4\}) = 000001 \oplus 001011 = 001010 = \text{id}(\{v_2, v_3\})$.

To fix this, we use *random bits as edge IDs* instead of simply concatenating vertex IDs: Randomly assign (in a consistent manner) each edge with a random ID of $k = 20 \log n$ bits. Since the XOR of random bits is random, for any edge e, $\Pr[\text{XOR}_A = \text{id}(e) \mid |E'| \neq 1] = (\frac{1}{2})^k = (\frac{1}{2})^{20 \log n}$. Hence,

$$
\begin{aligned}
&\Pr[\text{XOR}_A = \text{id}(e) \text{ for some edge } e \mid |E'| \neq 1] \\
&\leqslant \sum_{e \in \binom{V}{2}} \Pr[\text{XOR}_A = \text{id}(e) \mid |E'| \neq 1] && \text{Union bound over all edges} \\
&= \binom{n}{2} \cdot (\frac{1}{2})^{20 \log n} && \text{There are } \binom{n}{2} \text{ possible edges} \\
&= 2^{-18 \log n} && \text{Since } \binom{n}{2} \leqslant n^2 = 2^{2 \log n} \\
&= \frac{1}{n^{18}} && \text{Rewriting}
\end{aligned}
$$

Thus, with high probability, we can correctly distinguish $|E'| = 1$ from $|E'| \neq 1$. Furthermore, $\Pr[|E'| = 1] \geqslant \frac{1}{10}$. For any given $\epsilon > 0$, there exists a constant $C(\epsilon)$ such that if we repeat $t = C(\epsilon) \log n$ times, the probability that *all* t tries fail to extract a single cut is $(1 - \frac{1}{10})^t \leqslant \frac{1}{\text{poly}(n)}$.

Maximal forest with $\mathcal{O}(n \log^4 n)$ memory

Recall that Borůvka's algorithm builds a minimum spanning tree by iteratively finding the cheapest edge leaving connected components and adding them into the MST. The number of connected components decreases by at least half per iteration, so it converges in $\mathcal{O}(\log n)$ iterations.

Any arbitrary cut has cut size $k \in [0, n]$. Using $\mathcal{O}(\log n)$ guesses for $\widehat{k} = 2^0, 2^1, \ldots, 2^{\lceil \log n \rceil}$, we can use the approach to the k-cut problem to find a single cut edge:

- If $\widehat{k} \ll k$, the marking probability will select nothing (in expectation).
- If $\widehat{k} \approx k$, then we expect to find a valid edge ID.

- If $\widehat{k} \gg k$, more than one edge will get marked, which we will then detect (and ignore) since XOR_A will likely not be a valid edge ID.

Using shared randomness, vertices compute $\mathcal{O}(\log n)$ sketches. Each sketch has size $\mathcal{O}(\log^3 n)$ and is computed using random (but consistent) edge IDs and marking probabilities:

- For each vertex v, XOR_v takes $\mathcal{O}(\log n)$ bits to represent
- Make $\mathcal{O}(\log n)$ guesses of cut size k
- Repeat $\mathcal{O}(\log n)$ times independently to amplify success probability

One round of Borůvka can be simulated by using a unused sketch:

- Find an out-going edge from each connected component using the approach to the k-cut problem
- Join connected components by adding edges to graph

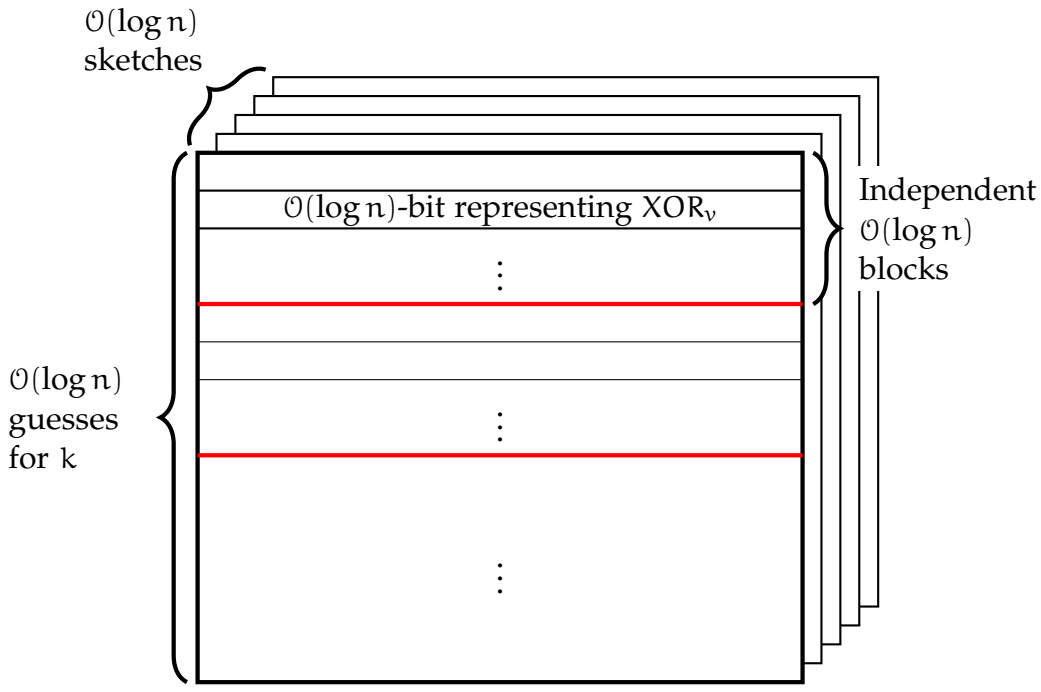


Figure 3.1: A pictorial representation of the $\mathcal{O}(\log n)$ sketches for vertex v that is sent to the coordinator. A sketch takes $\mathcal{O}(\log^3 n)$ bits. Each row is a XOR of adjacent edge IDs, subject to edge sampling probability $\widehat{k}$.

Each sketch uses $\mathcal{O}(\log^3 n)$ memory and $\mathcal{O}(\log n)$ sketches can simulate Borůvka, so a total of $\mathcal{O}(n \log^4 n)$ memory suffices. With $t = C(\epsilon) \log n$ tries at each step, we fail to find one cut edge leaving a connected component with probability $\leqslant (1 - \frac{1}{10})^t$. For a sufficiently large constant C, an application of union bound tells us that we succeed with high probability.

Remark One can drop the memory requirement per vertex from $\mathcal{O}(\log^4 n)$ to $\mathcal{O}(\log^3 n)$ by using a constant t instead of $t \in \mathcal{O}(\log n)$ such that the success probability is a constant larger than $1/2$. Then, simulate Borůvka for $\lceil 2 \log n \rceil$ steps. See Ahn, Guha and McGregor [AGM12] for details. Note that they use a slightly different sketch.

Theorem 3.4. *Any randomized distributed sketching protocol for computing spanning forest with success probability $\epsilon > 0$ must have expected average sketch size $\Omega(\log^3 n)$, for any constant $\epsilon > 0$.*

Proof. See Nelson and Yu [NY18] for details. □

Constructing random edge IDs using ϵ-bias sample spaces

We can use ϵ-bias sample spaces to obtain the random edge IDs discussed earlier. Since each bit of ID's operates independently, we focus on constructing 1-bit ID's first. Let $N = \binom{n}{2}$.

Claim 3.5. *There exists a collection B of functions $b : [N] \to \{0, 1\}$ such that:*

1. *For any $E' \subseteq [N]$ such that $|E'| \geqslant 1$,*

$$\Pr_{b \in B}[(\bigoplus_{e \in E'} b(e)) = 0] \in [\frac{1}{2} - \epsilon, \frac{1}{2} + \epsilon]$$

2. $|B| = \mathcal{O}(\frac{N}{\epsilon^2})$

Proof. Fix any arbitrary subset E' of $[N]$. Observe that for $b \in_{u.a.r.} \{0,1\}^N$, we have $\Pr[\bigoplus_{e \in E'} b(e) = 0] = \frac{1}{2}$. Fix $k = \frac{cN}{\epsilon^2}$ for some large constant c. Pick B of size k uniformly from $\{0,1\}^N$. In expectation, half of the $b \in B$ satisfy the condition $\bigoplus_{e \in E'} b(e) = 0$. By Chernoff bound, the probability that number of such b is *not* in $[(\frac{1}{2} - \epsilon)k, (\frac{1}{2} + \epsilon)k]$ is at most $2e^{-\frac{\epsilon^2 k}{2}} \ll 2^{-N}$. By taking union bound over all 2^N choices of E', the probabilistic method tells us that such a collection exists. □

To store edge IDs, it suffices to store which functions from B are used. Suppose we use IDs of length $l = C \cdot \log n$ for some constant C. Then, only $l \cdot \log |B| \in \mathcal{O}(\log^2 n)$ bits of information need to be stored. Additionally, since each bit of ID is selected independently, the probability of a collision from XOR's is upper bounded by $(\frac{1}{2} + \epsilon)^l$. By selecting C to be large enough constant, the collision probabilities can be made to be exponentially small. Fast constructions of such B's exist but are out of the scope of the course.

3.2.2 Simulating in MPC

In the near linear memory regime, each machine has $S = \widetilde{\mathcal{O}}(n)$ memory. To compute a sketch for a vertex, we first need to put all edges incident to a vertex onto a single machine. This can be done in $\mathcal{O}(1)$ rounds (See Exercise 3.3). After distribution edges, sketches can be computed independently in a single MPC round. Since each vertex produces sketches of size $\mathcal{O}(\log^4 n)$, the sketches for all n vertices fit into the memory of a single machine. Thus, the connected component can be successfully computed in one further MPC round, with high probability. As we can see, this entire process takes $\mathcal{O}(1)$ rounds in near linear memory regime.

Recall in Section 3.1 that we use connected components as a subroutine to solve MST. To be precise, we solve $\frac{m}{n}$ copies of connected component problem in parallel, in a single MPC round. This is doable because the required sketches fit into a single machine and we have $\mathcal{O}(\frac{m}{n})$ machines.

Exercises

Exercise 3.3. Edge distribution

Given an arbitrary initial distribution of edges, devise an algorithm that puts edges incident to the same node into a single machine in $\mathcal{O}(1)$ rounds with $\widetilde{\mathcal{O}}(n)$ memory per machine.

For example, suppose $E = \{\{a, b\}, \{a, c\}, \{b, d\}\}$. We wish to put $\{a, b\}$ and $\{a, c\}$ into the same machine so that all endpoints of a are in the same machine. Similarly, we wish to put $\{a, b\}$ and $\{b, d\}$ onto the same machine, possibly in a different machine as before. Note that the edge $\{a, b\}$ is duplicated and stored twice, once for a and once for b.

Hint *Sort the edges.*