Lecture 5: Graph algorithms

In the previous chapter, we used MAX-CUT as a running example to illustrate some derandomisation techniques. In this chapter, we will again look at graph algorithms, but focusing on problems for which we *know* efficient deterministic algorithms. The key message here is that randomisation does allow us to do things *even more efficiently* – and, sometimes, to also extract theorems about graphs from our algorithms!

Karger's Min-Cut algorithm

We will start with a beautiful algorithm, due to Karger [16] (and improved by Karger and Stein [17]), for the *minimum* cut question:

> MIN-CUT: Given an (undirected) connected graph $G = (V, E)$ on n vertices and m edges, output a cut (A, B) (partition of V) *minimising* the number $c(A, B)$ of edges between A and B.

[16] David R. Karger. Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm. In *SODA*. ACM/SIAM, 1993

[17] David R. Karger and Clifford Stein. An $\tilde{O}(n^2)$ algorithm for minimum cuts. In *STOC*. ACM, 1993

If G is not connected, we can detect this in $O(m + n)$ time, and then a "minimum cut" is... easy to find.

Of course, we want an efficient algorithm for that. As a baseline, one could try to "just" find a good deterministic algorithm to solve the problem. Fortunately, we have some:

Fact 25.1. MIN-CUT *can be solved by computing* $n - 1$ *instances of the* MAX-FLOW *problem. This can be done is polynomial time in* n *and* m *(and, actually, in time* [18] $O\left(mn \log \frac{n^2}{m}\right)$*).*

Recall the MAX-FLOW problem: given a directed weighted graph and two vertices s and t, find a maximum feasible flow from s to t.

[18] Jianxiu Hao and James B. Orlin. A faster algorithm for finding the minimum cut in a directed graph. *J. Algorithms*, 17(3):424–446, 1994

This is annoying, as this strongly hints that, well, we're done here. However, the above algorithm is quite involved: can we do as well, or even better, with a *simple* randomised algorithm?

As it turns out, *yes*. Here is the gist of the algorithm: (1) Pick an edge of the graph uniformly at random. (2) "Merge" its two endpoints. (3) Repeat.

That's all! Of course, to formally describe and analyse this mind-blowingly simple algorithm, we first need to define what we mean by "merging" two vertices. This is an operation called *contraction*:

Definition 25.1. Let $G = (V, E)$ be a multigraph[19] and $e = (u, v) \in E$ one of its edges. The *contraction of* G *with respect to* e, denoted G/e, is the multigraph on $|V| - 1$ vertices defined from G as follows:

[19] We allow parallel edges, but no self-loops. So there could be several edges between two distinct vertices u, v (but none from u to itself).

1. Replace u and v by a single vertex, uv;
2. Replace all edges of E of the form (u, w) or (v, w) by an edge (uv, w);
3. Remove all self-loops (uv, uv) the second step may have created.

The process is illustrated in Fig. 10. Note that a contraction can be performed in time $O(n)$ given either the adjacency list or adjacency matrix representation of the multigraph.

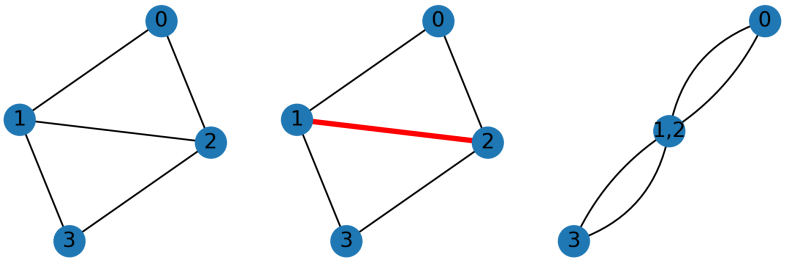

Figure 10: A contraction of the edge $e = (1, 2)$: the original (multi)graph is on the left, and the resulting (multi)graph G/e on the right.

Another way to interpret the contraction is that, after contracting some edges to get a multigraph $G' = G/(e_1, \ldots, e_k)$, each vertex u in G' corresponds to a subset of vertices $S_u \subseteq V$ from the original graph $G = (V, E)$: the subset of all vertices that were contracted together to become u. And any two distinct u, v from G' correspond to *disjoint* subsets $S_v, S_v \subseteq V$ (during a contraction, a vertex cannot be merged to two separate new vertices!).

So if after a sequence of contractions we end up with a multigraph G' which has only *two* vertices u, v, we get a cut in our original graph G: the cut (S_u, S_v). And the value $c(S_u, S_v)$ of this cut is then exactly the number of parallel edges between u and v.

This is the basis for our algorithm, which we are now able to state:

Require: multigraph $G = (V, E)$
1: **while** $|V| > 2$ **do**
2: Pick an edge $e \in E$ uniformly at random
3: Contract it, and let $G \leftarrow G/e$
4: **return** the cut defined by the remaining two vertices.

Algorithm 10: Karger's MIN-CUT algorithm.

This is all. Each iteration of the loop takes time $O(n)$; each contraction reduces the number of vertices by one, and we started with n vertices: so we have $n - 2$ iterations. Overall, running Algorithm 10 takes $O(n^2)$ time. *But is the cut it returns any good?* And importantly, *why* would we expect to be any good?

The contraction operation does, and we will see in the tutorial that sampling an edge uniformly can be done in time $O(n)$ as well.

Some intuition. As a thought experiment, consider any (fixed) cut $C = (A, B)$ of the graph. The only way C will survive until the end of the algorithm (and be returned in Line 3) is if we never

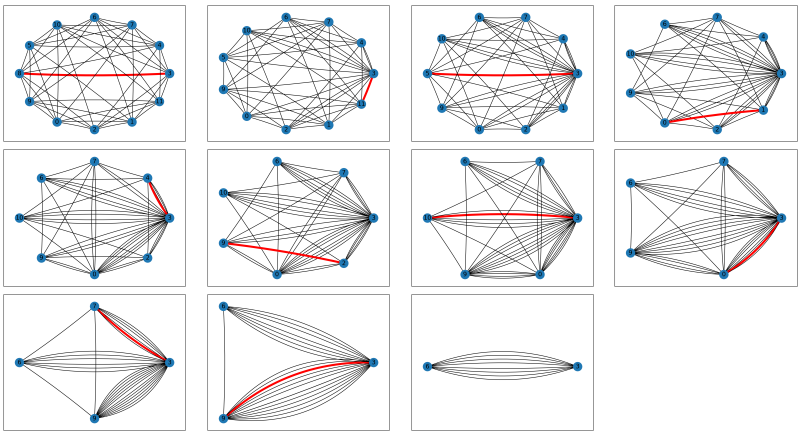

Figure 11: The sequence of steps for one run of Karger's algorithm (Algorithm 10) on a (regular) graph with $n = 12$ vertices and $m = 48$ edges. The cut returned has 8 edges.

contract any edge going from a vertex in A to a vertex in B: that is, any vertex of the cut itself. Because as soon as we contract such an edge, some vertex in A is merged with some vertex in B, and the cut (A, B) does no longer exist in our new contracted multigraph. Put differently, the more edges there are between A and B, the less likely the cut C should be to make it to the end of the algorithm, and as a result, we expect "small cuts" (those with fewer edges crossing) to have a better probability to be returned. But that's exactly what we want: by definition, minimum cuts are the smallest cuts possible! So they should be the ones being the most likely to be returned by our algorithm...

Consider a different strategy for Line 2 of the algorithm, which would sample a pair of distinct vertices (u, v) uniformly at random (not necessarily an edge). Would that work?

To make it formal, fix any *minimum* cut $C = (A, B)$ of G, and let $k = c(A, B)$ be its value. For $1 \leq i \leq n - 2$, let $\mathcal{E}_i$ be the event that the edge e picked in the i-th step of the algorithm does *not* belong to our cut C. By the above discussion,

$$\begin{aligned}\Pr[\, C \text{ is returned} \,] &= \Pr[\mathcal{E}_1 \cap \mathcal{E}_2 \cap \cdots \cap \mathcal{E}_{n-2}] \\ &\qquad \text{(No edge from } C \text{ is ever contracted)} \\ &= \Pr[\mathcal{E}_1] \Pr[\, \mathcal{E}_2 \mid \mathcal{E}_1 \,] \cdots \Pr[\, \mathcal{E}_{n-2} \mid \mathcal{E}_1 \cap \mathcal{E}_2 \cap \cdots \cap \mathcal{E}_{n-1} \,]\end{aligned}$$

Based on this, what we need to conclude is to get a good lower bound on the probability

$$\Pr[\, \mathcal{E}_{i+1} \mid \mathcal{E}_1 \cap \mathcal{E}_2 \cap \cdots \cap \mathcal{E}_i \,]$$

for all $1 \leq i \leq n - 3$: then, we will multiply all of them, and hope for the best. Write $G_i = (V_i, E_i)$ for the multigraph at the end of step i: so $G_0 = G$, and G_{n-2} is the 2-vertex multigraph obtained at the end. The probability that an edge of C is chosen in step $i + 1$ to be contracted (if C has survived until then, which is the event $\mathcal{E}_1 \cap \mathcal{E}_2 \cap \cdots \cap \mathcal{E}_i$) is then equal to

$$\frac{k}{|E_i|} \tag{41}$$

We need to (upper) bound this probability, and all we know is that:

- the number of vertices is $|V_i| = n - i$;

- the value of any minimum cut of G_i is k (there is one cut of size k, our cut C which survived so far; and there cannot be smaller cuts, as they would imply a smaller-than-minimum cut in the original graph G as well).

The key observation is that the *minimum degree* of G_i must then be at least k. Otherwise, there would exist some vertex $u \in V_i$ with less than k neighbours: choosing the cut $\{u\}, V_i \setminus \{u\}$ would give a cut in G_i of size less than k. Using the Handshaking Lemma,[20] we have

$$|E_i| = \frac{1}{2} \sum_{v \in V_i} \deg v \geq \frac{1}{2} |V_i| \cdot k \tag{42}$$

[20] Which, importantly, also holds in (simple) multigraphs.

or, equivalently,

$$\frac{k}{|E_i|} \leq \frac{2}{|V_i|} = \frac{2}{n-i}.$$

This shows that

$$\Pr[\, \mathcal{E}_{i+1} \mid \mathcal{E}_1 \cap \mathcal{E}_2 \cap \cdots \cap \mathcal{E}_i \,] = 1 - \frac{k}{|E_i|} \geq 1 - \frac{2}{n-i} \tag{43}$$

and as a result, "multiplying all the conditional probabilities and hoping for the best" gives

$$\begin{aligned}
\Pr[\, C \text{ is returned} \,] &= \prod_{i=0}^{n-3} \Pr[\, \mathcal{E}_{i+1} \mid \mathcal{E}_1 \cap \cdots \cap \mathcal{E}_i \,] \\
&\geq \prod_{i=0}^{n-3} \left(1 - \frac{2}{n-i}\right) \\
&= \prod_{i=0}^{n-3} \frac{n-i-2}{n-i} \\
&= \prod_{j=3}^{n} \frac{j-2}{j} \\
&= \frac{1 \cdot 2 \cdot 3 \cdot 4 \cdots (n-2)}{3 \cdot 4 \cdot 5 \cdots (n-2)(n-1)n} \\
&= \frac{2}{(n-1)n}.
\end{aligned} \tag{44}$$

What we showed is that, with probability at least $\frac{2}{(n-1)n}$, Karger's algorithm (Algorithm 10) returns this specific minimum cut C. There may be more than one possible minimum cut, so the probability it returns *some* minimum cut is at least $\frac{2}{(n-1)n}$:

Theorem 26. *Karger's algorithm (Algorithm 10) returns a minimum cut with probability at least* $\frac{2}{(n-1)n} = \Omega(1/n^2)$.

On the one hand, this is great: the algorithm works! On the other hand, this is somewhat problematic: the probability of success we can guarantee is *very* small. Fortunately, similarly to what we saw in Chapter 2, we can increase our probability of success by repetition, using Algorithm 10 as a blackbox. That is:

Algorithm 11: Amplifying the probability of Karger's MIN-CUT algorithm via repetition.

Require: multigraph $G = (V, E)$, integer T
1: **for** $1 \le t \le T$ **do** ▷ Use fresh (independent) random bits for each
2: Run Algorithm 10 on G, let C_t be the output
3: **return** the smallest cut among all cuts $C_1, \dots, C_T$ obtained

From Theorem 26, we know that each of the T independent repetitions of the algorithm has probability $p \ge \frac{2}{n(n-1)}$ of returning a minimum cut. And since it is returning the best cut among them, Algorithm 11 will return a minimum cut unless *none* of these T cuts is a minimum cut. So

$$\Pr\left[\begin{array}{c}\text{Algorithm 11 fails to}\\ \text{return a minimum cut}\end{array}\right] = (1-p)^T \le \left(1 - \frac{2}{n(n-1)}\right)^T \le e^{-\frac{2T}{n(n-1)}}$$

where we used the inequality $1 - x \le e^{-x}$ in the end. To achieve probability of success $1 - \delta$, it suffices to choose T so that the RHS is at most δ: one can check that setting

$$T = \left\lceil n^2 \ln(1/\delta) \right\rceil$$

suffices. Overall, the running time is $O(Tn^2)$, showing the following:

Theorem 27. *For any $\delta > 0$, the "Best-of-T" version of Karger's algorithm (Algorithm 11) returns a minimum cut with probability at least $1 - \delta$, and runs in time $O(n^4 \log(1/\delta))$.*

Given how simple the algorithm is, this is quite remarkable! However, given that the (much more involved) best deterministic algorithm can find a minimum cut in time $O(mn \log \frac{n^2}{m}) = O(n^3)$, it is natural to wonder if we can do even better.

Improving Karger's algorithm: the Karger–Stein algorithm

The starting point is to note that Karger's algorithm does *very* well in the first few iterations, but the guarantees degrade quickly towards the end. Again, let's look at a fixed minimum cut C of size k: the probability to "kill" C with the first contraction is very small, k/m. At the i-th step, when i is not too big, this is still very small: in Eq. (43), we bounded it by

$$\frac{2}{n-i} \approx \frac{2}{n}$$

All good! But at the *end* of the algorithm, the last few steps, this becomes really, really bad: at the last step ($i = n - 3$), for instance, the probability that C is "killed" is only bounded by

$$\frac{2}{n-i} = \frac{2}{3}$$

This tells us that after surviving almost until the end, we can only guarantee that our minimum cut C has a 33% chance of surviving

the very last step! And the first few contractions before that are not much better: each of them has a constant probability of killing C.

Based on this, it makes sense to only run Karger's algorithm for a while, and then do "something else" once we have contracted sufficiently many edges. This leaves two questions: (1) When should we stop? and (2) What should we do afterwards?

To answer the first question, we can look back at our analysis of the success probability. If we stop after $n - s$ steps, we are left with s vertices, and similarly to what we did in Eq. (44) we can guarantee that any fixed minimum cut survives with probability at least

$$\prod_{i=0}^{n-s-1} \frac{n-i-2}{n-i} = \prod_{j=s+1}^{n} \frac{j-2}{j} = \frac{s(s-1)}{n(n-1)}$$

If we choose $s = \frac{n}{\sqrt{2}} + 1$, we get

$$\Pr[\, C \text{ survives these } n - s \text{ steps} \,] \geq \frac{1}{2}\,.$$

So this answers (1): we should stop once only $\frac{n}{\sqrt{2}} + 1$ vertices remain. Then, even if there was only a single minimum cut C in the original graph, it will have survived with probability at least $1/2$. But turning to question (2): what to do afterwards?

We reduced the size of the problem by a constant factor, from n vertices to $\approx \frac{n}{\sqrt{2}}$. In the absence of a better idea, this seems to call for a recursive approach.

"When in doubt, recurse"

First, the base case: we can only recurse if we make progress at each call, and that can only happen if

$$\left\lceil \frac{n}{\sqrt{2}} + 1 \right\rceil > n$$

and that is only true for $n \geq 7$. This gives us our base case: if $n \leq 6$, we will just compute a minimum cut by brute force (in constant time).

Second, $1/2$ probability is much better than $\approx 1/n^2$, but it is still small: for a recursive approach, dropping our probability of success by such a constant factor at each recursive step could be bad. But if we have a probability of success at least $1/2$, repeating the first stage *twice* might not be a bad idea: we would have two different multigraphs G_1, G_2 on $s \approx \frac{n}{\sqrt{2}}$ vertices, each of them (independently) still containing a minimum cut with probability at least $1/2$. "In expectation", at least $2 \cdot (1/2) = 1$ still will have a minimum cut. This gives us our algorithm, given in Algorithm 12. To analyse this KargerStein algorithm, we need to establish its running time $T(n)$ and its probability of success $p(n)$. The running time turns out to be the simplest: we have two calls to ModifiedKarger, which (as in Algorithm 10) each take time $O(n^2)$; following by two recursive calls on instances of size $s \approx n/\sqrt{2}$. Ignoring the ceiling for simplicity, this gives the recurrence relation

$$T(n) = 2T(n/\sqrt{2}) + O(n^2) \tag{45}$$

Algorithm 12: The Improved Karger–Stein MIN-CUT algorithm.

```
1:  procedure ModifiedKarger(G = (V, E), s)
2:      while |V| > s do
3:          Pick an edge e ∈ E uniformly at random
4:          Contract it, and let G ← G/e
5:      return G
6:  procedure KargerStein(G = (V, E))
7:      if |V| ≤ 6 then
8:          return a minimum cut                ▷ Brute-force computation
9:      Set s ← ⌈n/√2 + 1⌉
10:     ▷ Contraction
11:         G_1 ← ModifiedKarger(G, s)
12:         G_2 ← ModifiedKarger(G, s)
13:     ▷ Recursion
14:         C_1 ← KargerStein(G_1)
15:         C_2 ← KargerStein(G_2)
16:     return the smallest cut among C_1, C_2
```

which solves to $\boxed{T(n) = O(n^2 \log n)}$ *via* the standard techniques.

Verify it, *e.g.*, with the Master Theorem; or, even better, without it.

The probability of success $p(n)$ is trickier. From our setting of s, we know that G_1 still contains a minimum cut with probability at least $1/2$: whenever that happens, C_1 will be a minimum cut if the recursive call to KargerStein is successful, which itself happens with probability at least $p(n/\sqrt{2})$. That is,

$$\Pr[\, C_1 \text{ is a minimum cut} \,] \geq \frac{1}{2} \cdot p\left(\frac{n}{\sqrt{2}}\right)$$

Similarly, looking at G_2 we have $\Pr[\, C_2 \text{ is a minimum cut} \,] \geq \frac{1}{2} \cdot p(n/\sqrt{2})$. Since we are taking the best of C_1, C_2 on Line 16, the algorithm succeeds unless *neither* of C_1, C_2 is a minimum cut:

$$\begin{aligned} p(n) &= 1 - (1 - \Pr[\, C_1 \text{ is a minimum cut} \,])(1 - \Pr[\, C_2 \text{ is a minimum cut} \,]) \\ &\geq 1 - \left(1 - \frac{1}{2} p\left(\frac{n}{\sqrt{2}}\right)\right)^2 \end{aligned}$$

We are left with the task of solving this recurrence relation on $p(n)$, with the base cases $p(n) = 1$ for $n \leq 6$.

Claim 27.1 ((⋆⋆)). *The recurrence relation*

$$p(n) \geq 1 - \left(1 - \frac{1}{2} p\left(\frac{n}{\sqrt{2}}\right)\right)^2$$

has solution $p(n) = \Omega(1/\log n)$.

Proof. Write $n = \sqrt{2}^t$ for $t \geq 1$. Expanding the square, this boils down to analysing the recurrence relation

$$p(\sqrt{2}^t) \geq p(\sqrt{2}^{t-1}) - \frac{1}{4} p(\sqrt{2}^{t-1})^2$$

or, equivalently (reparameterizing by setting $f(t) = p(\sqrt{2}^{-t}) \in [0,1]$),

$$f(t) \geq f(t-1) - \frac{1}{4} f(t-1)^2 \,. \tag{46}$$

Note that the function $x \mapsto x - \frac{1}{4}x^2$ is increasing on $[0,1]$: this will come handy later. We will show by induction on t that $f(t) \geq \frac{1}{t+2}$.

(⋆⋆) Another "rabbit-out-of-the-hat" proof: set $g(t) = \frac{4}{f(t)} - 1$, and substitute in the inequality. Solve the resulting inequality.

- This is true for $t = 0$, since $f(0) = p(1) = 1$.
- Assuming it is true for $t-1$, we have

$$\begin{aligned} f(t) &\geq f(t-1) - \frac{1}{4} f(t-1)^2 \\ &\geq \frac{1}{t+1} - \frac{1}{4}\left(\frac{1}{t+1}\right)^2 \\ &\qquad \text{(induction hypothesis and } x \mapsto x - \tfrac{1}{4}x^2 \text{ increasing)} \\ &= \frac{4t+3}{4(t+1)^2} = \frac{1}{t+2} + \frac{3t+2}{4(t+1)^2(t+2)} \\ &\geq \frac{1}{t+2} \end{aligned}$$

  concluding the induction proof.

Recalling that $n = \sqrt{2}^t$, we have $t = 2\log n$, and the above shows that $p(n) \geq \frac{1}{2\log n + 2} = \Omega\left(\frac{1}{\log n}\right)$, as claimed. □

What we have shown can be summarised as follows:

Theorem 28. *The Karger–Stein algorithm (Algorithm 12) runs in time $O(n^2 \log n)$, and returns a minimum cut with probability at least $\Omega(1/\log n)$.*

Moreover, with exactly the same approach as for Theorem 27 (using Theorem 28 and setting $T = O(\log n \cdot \log(1/\delta))$), we get

Exercise: prove it!

Corollary 28.1. *For any $\delta > 0$, the "Best-of-T" version of the Karger–Stein algorithm returns a minimum cut with probability at least $1 - \delta$, and runs in time $O\left(n^2 \log^2 n \log(1/\delta)\right)$.*

This is now typically *much* faster than the $O\left(mn \log \frac{n^2}{m}\right)$ running time of the deterministic algorithm!

Specifically, as long as the graph is even mildly dense, *i.e.*, $m \gg n \log n$.

Remark 28.1. There is a different deterministic algorithm, due to Stoer and Wagner [21] and not based on computing maximum flows, with the running time $O(mn + n^2 \log n)$ (slightly better than $O\left(mn \log \frac{n^2}{m}\right)$, but still worse than Theorem 27). Interestingly, this algorithm also works by performing some type of contraction (merging two carefully selected vertices at each step).

[21] Mechthild Stoer and Frank Wagner. A simple min-cut algorithm. *J. ACM*, 44(4):585–591, 1997

How many minimum cuts are there?

The MIN-CUT question we have considered so far asks to *find* a minimum cut in a graph G: *any* minimum cut. There is always at least *one* minimum cut, but could there be more? How many, at most?

- $\Theta(n)$?
- $\Theta(n^2)$?
- $\Theta(2^n)$?
- Something else?

And *how to prove it*?

Fortunately, we already have the answer, *and* done the proof. We just did not realise it at the time! This is a beautiful example where analysing an algorithm establishes a structural result, almost "as a side effect."

Taking a step back: in order to prove Theorem 26, we have shown that if C is a minimum cut of G, then Algorithm 10 outputs C with probability at least

$$\frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}}$$

This means that if there exists M distinct minimum cuts in the graph G, the probability to output one of them is at least

$$\Pr\left[\begin{smallmatrix}\textsc{Karger}(G) \text{ outputs} \\ \text{one of } C_1,\dots,C_M\end{smallmatrix}\right] = \sum_{i=1}^{M} \Pr[\,\textsc{Karger}(G) \text{ outputs } C_i\,] \geq \frac{M}{\binom{n}{2}}$$

But probabilities are at most one, so $\Pr\left[\begin{smallmatrix}\textsc{Karger}(G) \text{ outputs} \\ \text{one of } C_1,\dots,C_M\end{smallmatrix}\right] \leq 1$. Which means that

$$M \leq \binom{n}{2}$$

and we get the following "for free":

Theorem 29. *An undirected graph $G = (V, E)$ on $|V| = n$ vertices has at most $\binom{n}{2}$ minimum cuts.*

This is quite surprising, since every graph on n vertices has exactly $2^{n-1} - 1$ distinct (not necessarily minimum) cuts. As usual, we can ask whether this $\binom{n}{2}$ bound is tight: and the answer is *yes*, as there exist some n-vertex graphs with that many minimum cuts. A simple example is a cycle on n vertices, where choosing any 2 edges out of n defines a distinct minimum cut: see Fig. 12.

Do you see why?

Minimum Spanning Tree in Expected Linear Time

Another classic and fundamental graph problem is the *minimum spanning tree* one, which, given a connected weighted graph G, asks to find a spanning tree with minimum total weight:

> MINIMUM SPANNING TREE (MST): Given an (undirected) connected graph $G = (V, E)$ on n vertices and m edges with positive weights $w\colon E \to \mathbb{R}_+$, output a spanning tree T minimising $w(T) = \sum_{e \in T} w(e)$.

A *spanning tree* of a graph G is a connected subgraph of G with no cycle. A *spanning forest* is the same thing without the requirement to be connected.
A *minimum spanning forest* (MSF) is the equivalent of an MST when the graph is not connected: it asks for a collection of MSTs, one for each connected component of the graph.

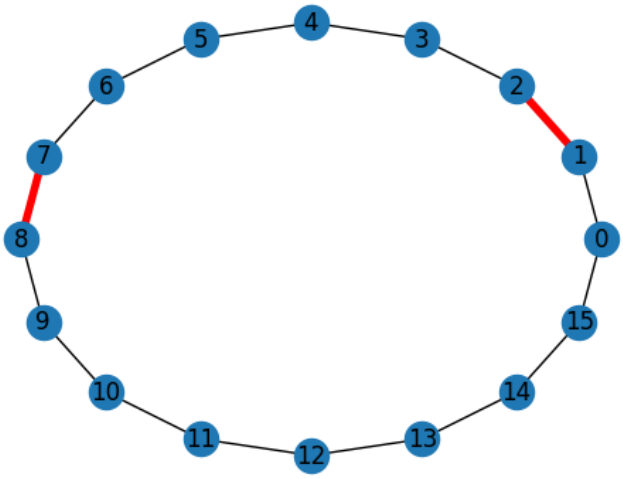

Figure 12: The cycle graph C_{16} on 16 vertices, along with a specific (mininum) cut (defined by the two red edges). Any choice of two edges creates a different minimum cut of the graph: there are $\binom{16}{2}$ such choices.

As in the previous section, you most likely remember from previous algorithms class that *we have deterministic algorithms to solve this efficiently:*

- Kruskal's algorithm solves it in time $O(m \log n)$
- Prim's algorithm solves it in time $O(m \log n)$ when implemented with a heap, or, better, $O(m + n \log n)$ using a Fibonacci heap
- Borůvka's algorithm solves it in time $O(m \log n)$
- the Fredman–Tarjan algorithm solves it in time $O(m \log^* n)$
- Chazelle's algorithm solves it in time $O(m\alpha(m, n))$

$\log^*$ is the iterated logarithm, an incredibly slow-growing function defined as "the number of times one must apply the logarithm to reach a value at most 1:"

$$\log^* x = \begin{cases} 0 & \text{if } x \leq 1 \\ 1 + \log^* \log x & \text{otherwise.} \end{cases}$$

$\log^* n$ still goes to infinity as n grows, but *very* slowly.

α is the *inverse Ackermann function*, which grows *even slower*.

The key point is that these algorithms (or their analysis) get more and more involved as we go down the list, and that *no deterministic algorithm running in linear time (that is, $O(m)$) is known.*

There is, however, a *randomised* algorithm for MST running in *expected* linear time, due to Karger, Klein, and Tarjan [22]. We will not go through its description and analysis in detail, but will only provide the key building blocks.

[22] David R. Karger, Philip N. Klein, and Robert Endre Tarjan. A randomized linear-time algorithm to find minimum spanning trees. *J. ACM*, 42(2):321–328, 1995

From now on, we will assume for convenience that all the weights $\{w(e)\}_{e \in E}$ are distinct. This is to make sure we can break ties consistently, and is without loss of generality. One nice consequence of this assumption is that the MST is now *unique*: there can be only one!

A standard way to implement consistent tie-breaking when some weights are equal is to do so using the lexicographic order of the edges.

Can you see why?

The main idea behind the algorithm can be summarized like this:

> If we had a way to remove most edges from G *without affecting its MST*, then we could recurse on a much sparser graph G'.

The question here is *how* to efficiently remove "most edges" (in expectation) without killing the MST in the process.

The first building block we need to answer this are the *cut* and *cycle* properties, which underly the proof (and ideas) behind Prim's and Borůvka's algorithm (cut property), and Kruskal's algorithm (cycle property):

> **Cut property.** Let $S \subseteq V$ be any subset of vertices, and let e be the minimum-weight edge with exactly one endpoint in S. Then the MST of G contains e.

> **Cycle property.** Let $C \subseteq E$ be any cycle, and let e be the maximum-weight edge belonging to C. Then the MST of G does not contain e.

We will require the definition of an edge being "heavy" with respect to a forest:

Definition 29.1. For any weighted graph $G = (V, E)$ and forest $F \subseteq E$ of G, we say that an edge $e \in E \setminus F$ is *F-heavy* if (1) adding e to F creates a cycle, and (2) e is the maximum-weight edge of that cycle.

From the cycle property, we readily get the following fact:

> **F-heaviness property.** Let $F \subseteq E$ be a forest of G, and let $e \in E \setminus F$ be an F-heavy edge. Then the MST of G does not contain e.

This looks promising: what this says is that if we have a forest (any forest!) F of G, we can safely remove all F-heavy edges from G without killing the MST. This sounds exactly like what we are hoping for! Provided, of course, that we can (1) efficiently find all F-heavy edges, and (2) that there are *many* of them.

The second building block ensures that, at least, we can do (1):

Fact 29.1. *There exists a deterministic algorithm* MSTVerification [23,24] *which, on input a graph $G = (V, E)$ with weights w and a forest F of G, outputs the set of F-heavy edges of G in time $O(m + n)$.*

To third and last building blocks will take care of (2). The idea here is that we *already* had the MST T (which we can see as a forest), then by the cycle property *every* edge $e \in G \setminus T$ is T-heavy: and so we could use Fact 29.1 on T to find (and remove) $m - n + 1$ edges from G in time $O(m + n)$. That would be amazing progress – but of course, *we do not have T*, that's the thing we are trying to compute!

What we *could* do, however, is computing the MST T' of a small random subgraph G' of G, and use that as our "guiding forest" to find which heavy edges to remove from G. If G' has sufficiently few edges and vertices (for instance, $m/2$ and $n/2$) then we can compute its MST T' recursively. So to do that, we need to "sparsify" G: both in terms of vertices and edges.

[23] Brandon Dixon, Monika Rauch, and Robert Endre Tarjan. Verification and sensitivity analysis of minimum spanning trees in linear time. *SIAM J. Comput.*, 21(6):1184–1192, 1992

[24] Valerie King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18(2):263–270, 1997

This type of algorithms is typically used to check whether a given tree is really an MST, hence the name "MST Verification."

For the edges, this is easy: given a graph $G = (V, E)$, we can build a new graph G' with (in expectation) much fewer edges by keeping each edge of E independently with probability $p \in [0, 1]$: this gives us $G' = (V, E')$ with $\mathbb{E}[|E'|] = p|E|$. Our third building block tells us what happens to the MST[25] when we do that: is the MSF of G' still "good"?

[25] Or, rather, maximum spanning forest (MSF), since randomly removing some edges might have disconnected G'.

Lemma 29.1 (Random Subsampling Lemma). *Let $G' = (V, E')$ be a subgraph of $G = (V, E)$ obtained by subsampling each edge $e \in E$ independently with probability $p \in [0, 1]$, and $F \subseteq E'$ be the MSF of G'. Then the expected number of edges* in *G that are not F-heavy is at most $\frac{|V|}{p}$.*

We leave the proof of this lemma as an exercise. The crucial part of the statement is that we compute F as the MSF of the *sparser* graph G' but get a guarantee on the number of F-heavy edges with respect to the *original* graph G.

(⋆⋆) Exercise!

To sparsify G in terms of vertices, the last building block we need is Borůvka's algorithm, or, rather, what happens when we run it only for a couple iterations: Algorithm 13.

Algorithm 13: t-step version of Borůvka's algorithm

```
1:  procedure BoruvkaStep(G = (V, E), t)
2:      F ← ∅                                   ▷ F stands for "Forest"
3:      for 1 ≤ i ≤ t do
4:          for all v ∈ V do
5:              Find the lightest edge e ∈ E incident to v:
                        e ← argmin{w(v, u) : (v, u) ∈ E}
6:              Contract it, and let G ← G/e
7:              F ← F ∪ {e}                     ▷ Add it to F
8:          for all u, v ∈ V do                 ▷ In the new graph
9:              if u, v are connected by more than one edge then
10:                 only keep one with the smallest weight
11:     return (G, F)      ▷ New graph and forest of contracted edges
```

Lemma 29.2. *The t-step version of Borůvka's algorithm, on input a connected graph $G = (V, E)$, returns a new graph $G' = (V', E')$ such that $|V'| \leq |V|/2^t$, and runs in time $O(t \cdot m)$.*

Proof sketch. Each step can be implemented to run in time $O(m)$; and at each step, each vertex is contracted with at least one other, so the total number of vertices decreases by at least a factor 2. □

Here $t \geq 1$, $p \in [0, 1]$ are parameters we will get to choose for things to work out.

At this point, we finally have all we need for the algorithm. To summarise our strategy:

1. Sparsify G in terms of vertices, to go from n to $n' = n/2^t$: this gives a graph G_1 on n' vertices (and a leftover forest F_1 of contracted edges)

One can check that adding F_1 to an MSF of G_1 gives the MST of G, so it remains to find an MSF of G_1.

2. Sparsify G_1 in terms of edges, to go from m to $m' \leq pm$ (in expectation: this is the only random step): this gives a graph G_2 on n' vertices and m' edges
3. Recursively find the MSF F_2 of G_2: this should be less expensive, as G_2 is smaller than G
4. Find all the F_2-heavy edges in G_1, and remove them from G_1 to get a graph G_3: there should be many by Lemma 29.1, and can be done efficiently by Fact 29.1
5. Recursively find the MSF F_3 of G_3: this should be less expensive, as G_3 is smaller than G: and this is also the MSF of G_1
6. return $T = F_1 \cup F_3$ as the MST of G

If we are lucky, we remove many edges and get $m' \ll m$ and also end up with many F_3-heavy edges, so the recursive calls will be faster. If we are unlucky, the recursive calls will be on bigger graphs, and so will be slower.

It is worth pointing out that the only random step in the above strategy is Step 2, and it does not affect *correctness*: it only affects the running time.

We can finally state the algorithm itself: To analyse it, we need

Algorithm 14: The Karger–Klein–Tarjan (KKT) Algorithm: MST in expected linear time

```
1:  procedure SUBSAMPLE(G = (V, E), p)
2:      E' ← ∅
3:      for all e ∈ E do                    ▷ Independently for each edge
4:          Add e to E' with probability p
5:      return G' = (V, E')
6:  procedure LINEARTIMEMST(G = (V, E), t, p)
7:      if |V| ≤ 2 then return G                                  ▷ Base case
8:      G_1 ← BORUVKASTEP(G, t)
9:      G_2 ← SUBSAMPLE(G_1, p)
10:     F_2 ← LINEARTIMEMST(G_2, t, p)                            ▷ Recursive call
11:     H ← MSTVERIFICATION(F_2, G_1)                             ▷ Find F_2-heavy edges
12:     G_3 ← (V_1, E_1 \ H)                                      ▷ Remove them from G_1
13:     F_3 ← LINEARTIMEMST(G_3, t, p)                            ▷ Recursive call
14:     return F_1 ∪ F_3
```

to establish its correctness and expected running time. We will only do the second, as correctness follows from the discussion and lemmas above.[26]

[26] Please check and establish it, this is a good exercise.

The expected running time $T(m, n)$ can be decomposed into the time taken by Lines 8, 11 and 12, which by Fact 29.1 and Lemma 29.2 take total time $O(m + n) + O(tm) = O(t(m + n))$; and the time of the two recursive calls, which take time $T(|E_2|, |V_2|) + T(|E_3|, |V_3|)$. Now, $|V_1| = |V_2| = |V_3| \leq n/2^t$ by Lemma 29.2, and this is deterministic (comes from the Borůvka step); but $|E_2|$ and $|E_3|$ are random. All we know is that

$|E_1| \leq m$ but is not necessarily equal to m, as the Borůvka step might have (deterministically) removed a few edges during the contractions.

$$\mathbb{E}[|E_2|] = p|E_1| \leq pm \tag{47}$$

because of subsampling, and that by Lemmas 29.1 and 29.2

$$\mathbb{E}[|E_3|] \leq \frac{|V_1|}{p} \leq \frac{n}{p2^t}. \tag{48}$$

So we have, for some absolute constant $C > 0$, that

$$T(m,n) \le C \cdot t(m+n) + \mathbb{E}[T(|E_2|,|V_2|)] + \mathbb{E}[T(|E_3|,|V_3|)] \qquad (49)$$

where the expectation is over the randomness of the subsampling of the edges (Line 9). Let us show by induction that

$$T(m,n) \le C' \cdot (m+n)$$

for some suitable constant $C' > 0$. The base case is easy, since Algorithm 14 runs in constant time for $n \le 2$ (our recursive base case). Now, for the induction, assume this holds for all (m,n) with $m' < m$ and $n' < n$: Eq. (49) gives

$$\begin{aligned} T(m,n) &\le Ct(m+n) + \mathbb{E}\left[C'(|E_2|+|V_2|)\right] + \mathbb{E}\left[C'(|E_3|+|V_3|)\right] \\ &\le Ct(m+n) + C'\left(\mathbb{E}[|E_2|] + \frac{n}{2^t} + \mathbb{E}[|E_3|] + \frac{n}{2^t}\right) \\ &\qquad\qquad \text{(Bound on } |V_2|,|V_3|\text{)} \\ &\le Ct(m+n) + C'\left(pm + \frac{n}{p2^t} + \frac{2n}{2^t}\right) \quad \text{(Eqs. (47) and (48))} \\ &= (Ct + C'p)m + \left(Ct + C'\frac{2+1/p}{2^t}\right)n \qquad (50) \end{aligned}$$

One can check that setting $t = 3$, $p = 1/2$, Eq. (50) becomes

$$T(m,n) \le (3C + C'/2)(m+n)$$

which gives $T(m,n) \le C'(m+n)$ as long as $C' \ge 6C$. This concludes the proof by induction, and the proof of the theorem:[27]

Theorem 30. *The Karger–Klein–Tarjan algorithm (Algorithm 14) computes a minimum spanning tree (MST) of an n-vertex, m-edge undirected connected graph in* expected $O(m)$ *time.*

To conclude, one last thing left as an exercise: can you convert this Las Vegas algorithm into a high-probability Monte-Carlo one?

Check what happens for other choices of t and p: for instance, can you choose $t = 2$? (This corresponds to only doing two rounds of the Borůvka step.) What about $t = 1$?

[27] Modulo the parts we left as an exercise.