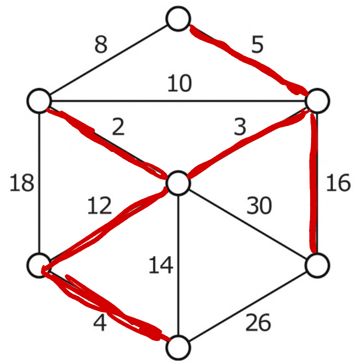


Minimum Spanning Tree (MST)

Given connected <u>undirected</u> graph $G=(V,E)$.

Weights $w: E \to \mathbb{R}$ ↙ could be negative!

Goal: Find the minimum spanning tree, a spanning tree T minimizing $w(T) = \sum_{e \in T} w(e)$

Assumption: $w(e) \neq w(e')$ when $e \neq e'$.

$\Rightarrow$ guarantees MST is unique

o.w., could be multiple MST's

Ex. $w(e) = 1 \; \forall e \Rightarrow$ every spanning tree has weight $|V| - 1$

The One Algorithm:

T: the MST we want to find

want to select edges bit-by-bit for T

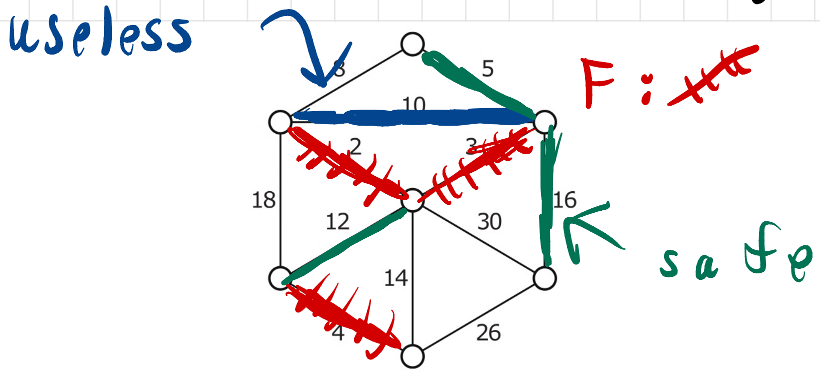
partway through algorithm

$F \subseteq T$: the edges we chose so far

- acyclic (a forest)
- call it the <u>intermediate spanning forest</u>
- intially the set of $|V|$ one-vertex trees

- will add edges to make
  $F' \supseteq F$ s.t. $F' \subseteq T$.
- then recursively find
  MST $T \supseteq F'$.
- stop if F is connected

Given F there are two special subsets of edges



- useless edges: outside F but both endpoints in same component of F
  - T has no useless edges
    F + useless e has a cycle?
- each component of F has a safe edge: the lightest one leaving the component

If $F \neq T$, there is at least one safe edge.
If $F = T$, all $e \notin T$ are useless.

Claim: MST T contains <u>every</u> safe edge. In fact, for all $S \subset V$, tree T has lightest edge with one endpoint in S.

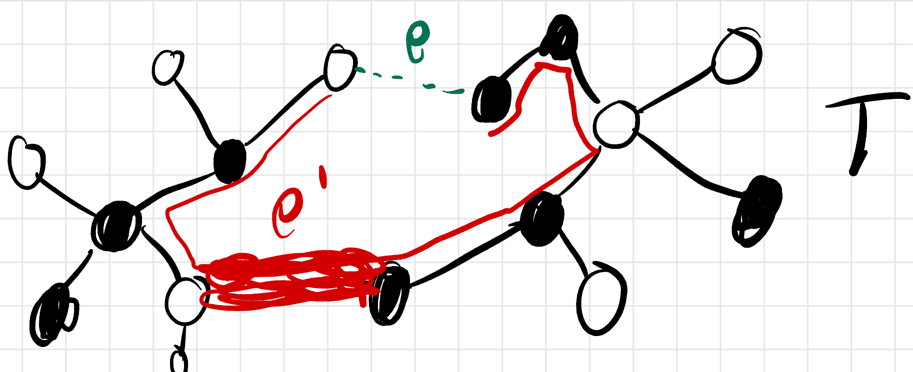
e: lightest edge leaving S

If MST T contains e ✓

o.w.

● $\in S$

o $\notin S$



exists a path in T between
e's endpoints
path has an edge e' that
goes from in S to not in S
T-e' has two components
Both components have one
endpoint of e.
$\Rightarrow$ T-e'+e is a spanning
tree.

not the
MST!

But $w(e) < w(e')$

so $w(T-e'+e) < w(T)$

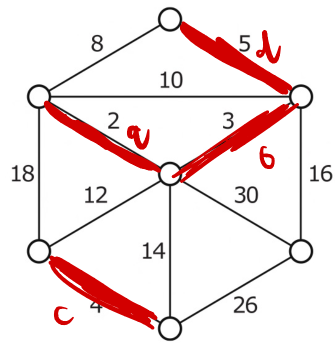
so $e \in T$ after all! $\perp$

So ... add one or more safe edges to F and recurse!

... but which ones?

Kruskal ['56]:

Scan edges in increasing weight order; add each safe edge you see.



Claim: When we scan e, all e' s.t. $w(e') \leq w(e)$ are in F or useless.

- if e not useless, it is lightest for both endpoint's components $\Rightarrow$ safe

Disjoint Sets: Maintains disjoint subsets over a collection of objects.

MakeSet(v): creates set $\{v\}$

Find(v): returns an "ID" for v's set. Find(u) = Find(v) iff u & v in same set

Union(u, v): replaces sets for u + v with union of the sets

$\underline{\text{Kruskal}(V, E)}$:

```
sort E by increasing weight
F ← (V, ∅)
for each vertex v ∈ V
    MakeSet(v)
for i ← 1 to |E|
    uv ← ith lightest edge in E
    if Find(u) ≠ Find(v)
        Union(u, v)
        add uv to F
return F
```

$\leftarrow O(E \log E)$

$\leftarrow O(\log V)$ per operation

$$O(E \log E) = O(E \log V^2)$$
$$= O(E \log V)$$

Total time $O(E \log V) + O(E \log V)$
$$= O(E \log V)$$

dominated by sorting

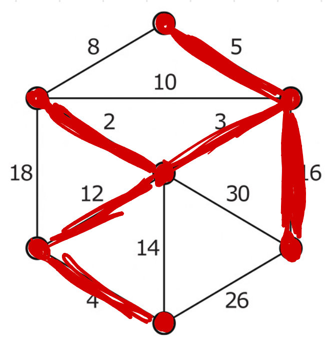
even with faster disjoint sets

Prim-Jarník:

Jarník ['29], Prim ['57].

F always has <u>one</u> non-trivial component T'. Others are isolated vertices.

Jarník: Repeatedly add safe edge of T' to T'.



(T' starts as any one vertex)

To implement:

keep a <u>priority queue</u> of edges incident to T'

each time you add an edge to T', add new incident edges to T'.

at least one endpoint in T'

when you Extract Min, check if edge leaves T' + ignore it if not

$O(\log E) = O(\log V)$ time per heap operation, so

$O(E \log V)$

But really, use Borůvka ['26].

See Erickson 7.3.