Algorithms (2020)

Minimum Spanning Trees

Recap

- HW on greedy - due Wed.
- Office hours today:
  3pm (not 1pm)

  (email if you ever need a non-office hour time)

Minimum Spanning Trees

edge : $w(e)$ = value in $\mathbb{R}$

Goal: Given a weighted graph G, w, find a spanning tree of G, T, that minimizes:

$$w(T) = \sum_{e \in T} w(e)$$

edge in T

S

tree ✓

minimum?

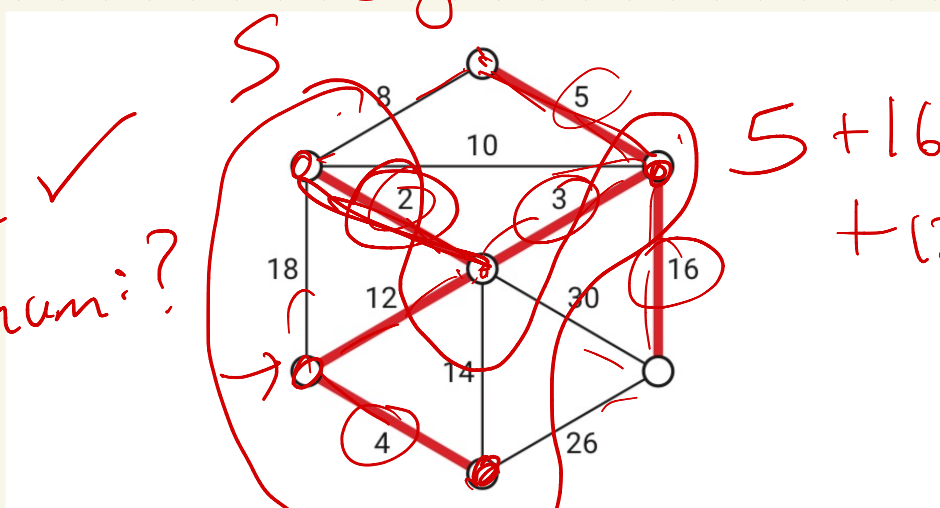


Figure 7.1. A weighted graph and its minimum spanning tree.

$5 + 16 + 3 + 2$
$+ 12 + 4$

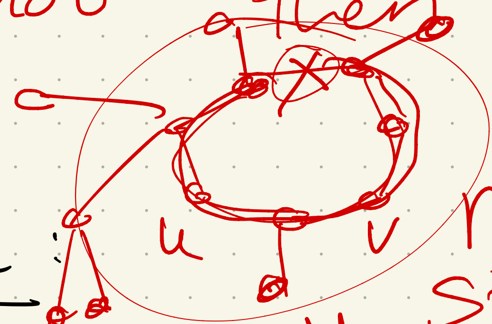
Motivation: roads
networks
etc.

First:

Does it have to be a tree?

Yes! Why?

Spps not — then $\exists$ a cycle



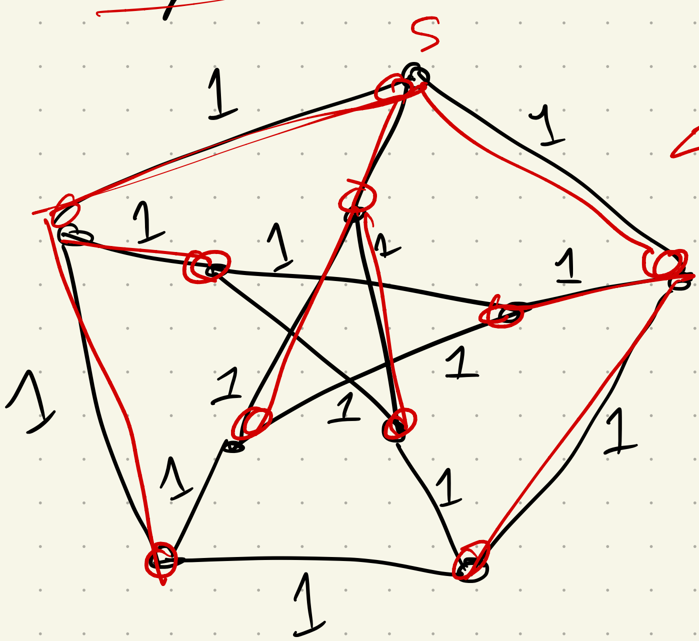
goal: connect every pair of vertices

remove any edge — still connected

Second:

These are "obviously" not unique!

Ex:



unweighted!

tree? any spanning tree works — use DFS or BFS

Things will be cleaner if we have unique trees. So:

Lemma: Assuming all edge weights are distinct, then MST is unique.

Pf: By contradiction:

Suppose T & T' are both MSTs, with $T \neq T'$.

↳ at least one edge different

- $T \cup T'$ contains a cycle C

↳ every u & v connected in T, T' has another edge e, e' that path = cycle

- That cycle must have 2 edges of equal weight

MST would lose one of those edges.

$\Rightarrow$ Contradiction!

$T \cup T'$, C

Could be more than 1 edge!

Now, what if weights aren't unique?

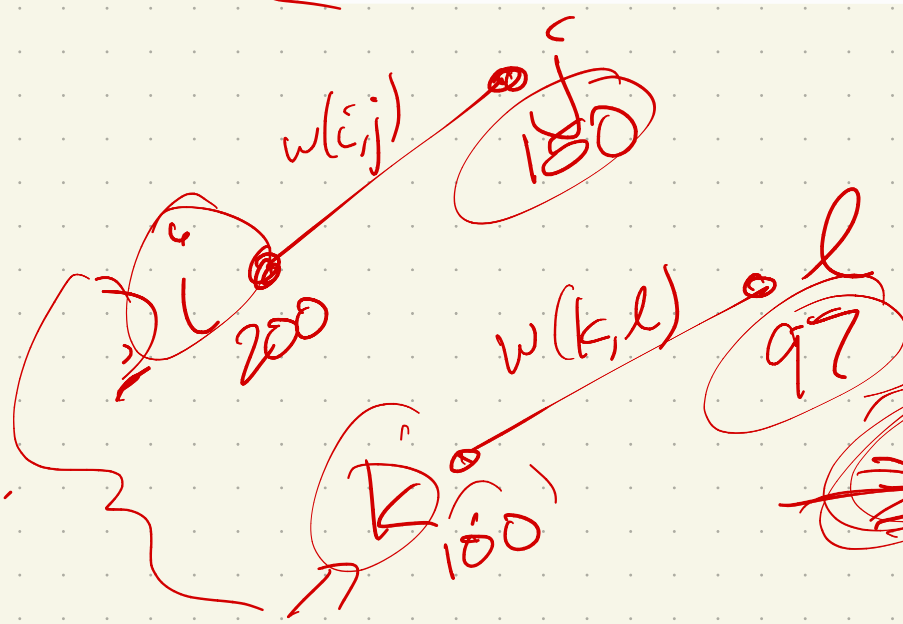
Just need a way to consistently break ties.

← if $w(e) = w(e')$, one will always win!

```
SHORTEREDGE(i, j, k, l)
    if w(i, j) < w(k, l)          then return (i, j)
    if w(i, j) > w(k, l)          then return (k, l)
    if min(i, j) < min(k, l)      then return (i, j)
    if min(i, j) > min(k, l)      then return (k, l)
    if max(i, j) < max(k, l)      then return (i, j)
    ⟨⟨if max(i,j) > max(k,l)⟩⟩    return (k, l)
```

they aren't equal weight

break ties.



assuming vertices are stored in some array

V: [1 - - - n]

adj list.

So, takeaway:

Can assume unique MST.

Next: an algorithm.

The magic truth of MSTs:

You can be SUPER greedy.

Almost any natural idea will work!

This is highly unusual, &

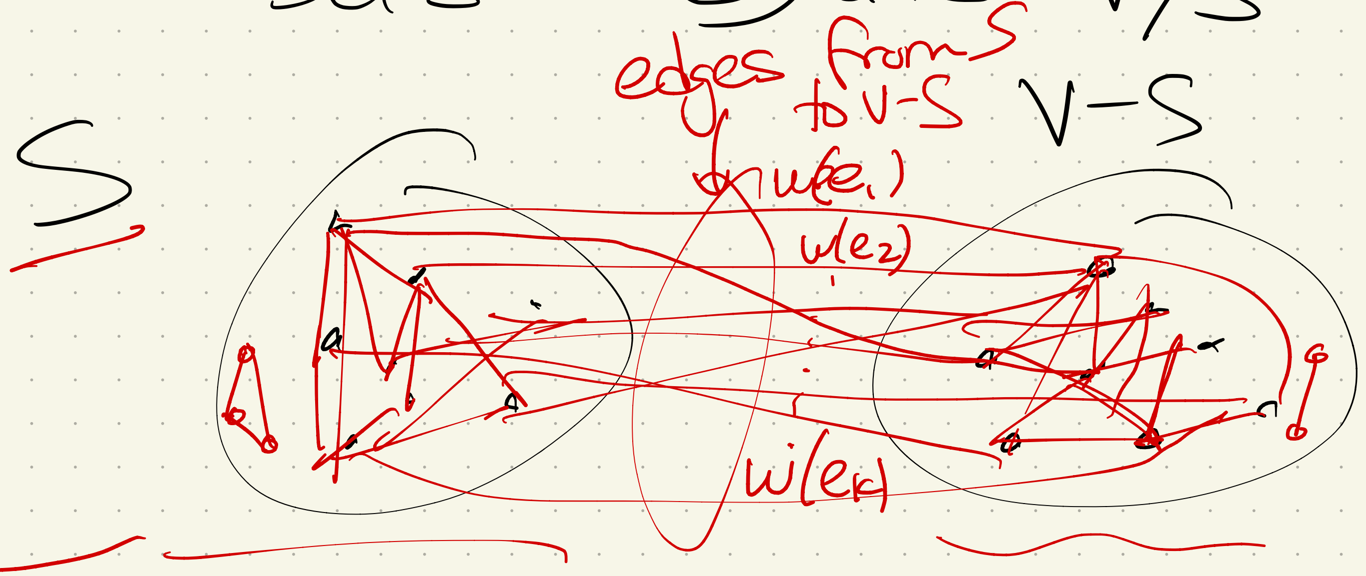
there's a reason for it:

these are a (rare) example of something called a matroid.

(Way beyond this class...)

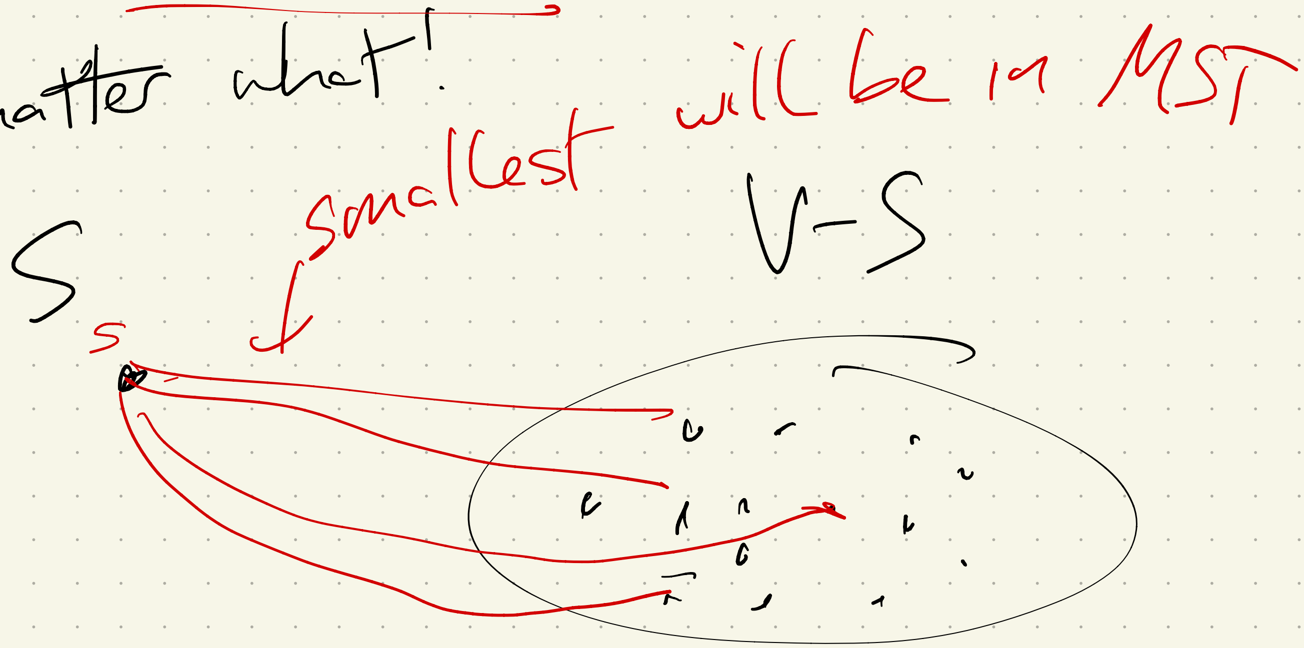
Key property:

Consider breaking G into two sets: S and V/S

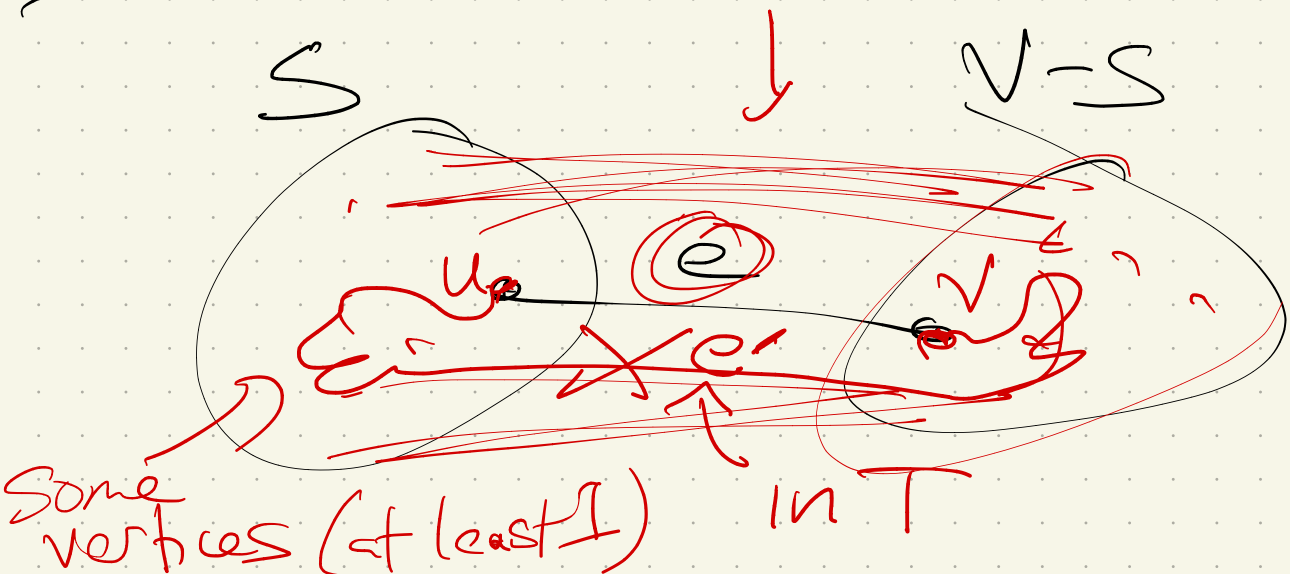


The MST will always contain the lowest edge connecting the two sides.

No matter what!



Proof: consider minimum edge e



Suppose MST does not contain e.

T

Then MST must have some other S to V-S edge

Take $T \cup e$ (uv): tree + an edge has at least one cycle.

why? since $e = uv$ & u + v have a path in T. Take cycle & remove a different S to V-S edge. $w(e') > w(e)$, since e was smallest. new tree $< T$.

Generic Algorithm: → apply that lemma w/ S & V−S

Build a forest: an acyclic subgraph.

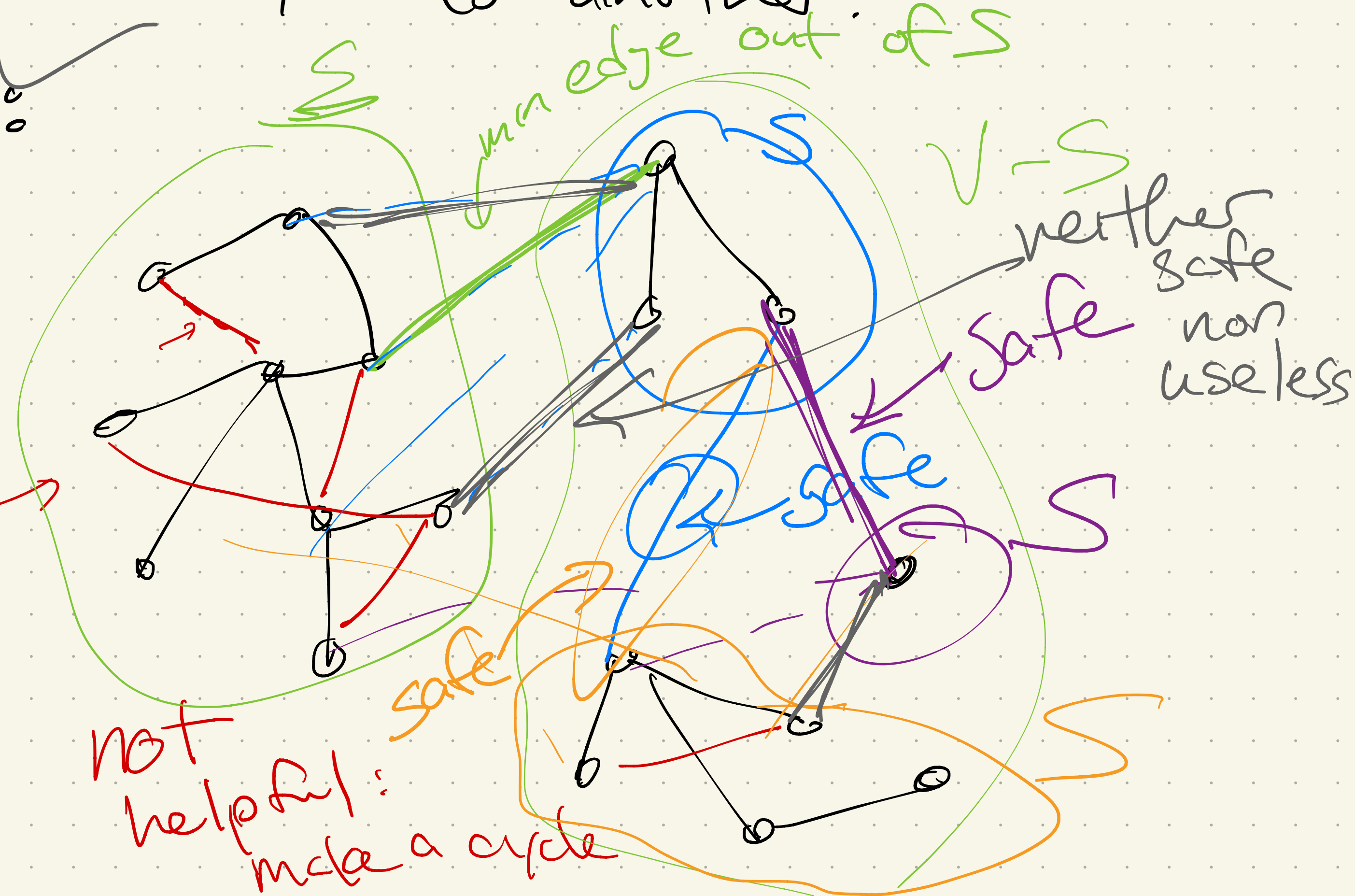
less than a tree, no cycles.

Dfn: An edge is useless if it connects 2 endpts in same component of F

also edges that are neither.

An edge is safe if it is minimum edge from some component of F to another.

F:



So idea:

Add safe edges until you get a tree

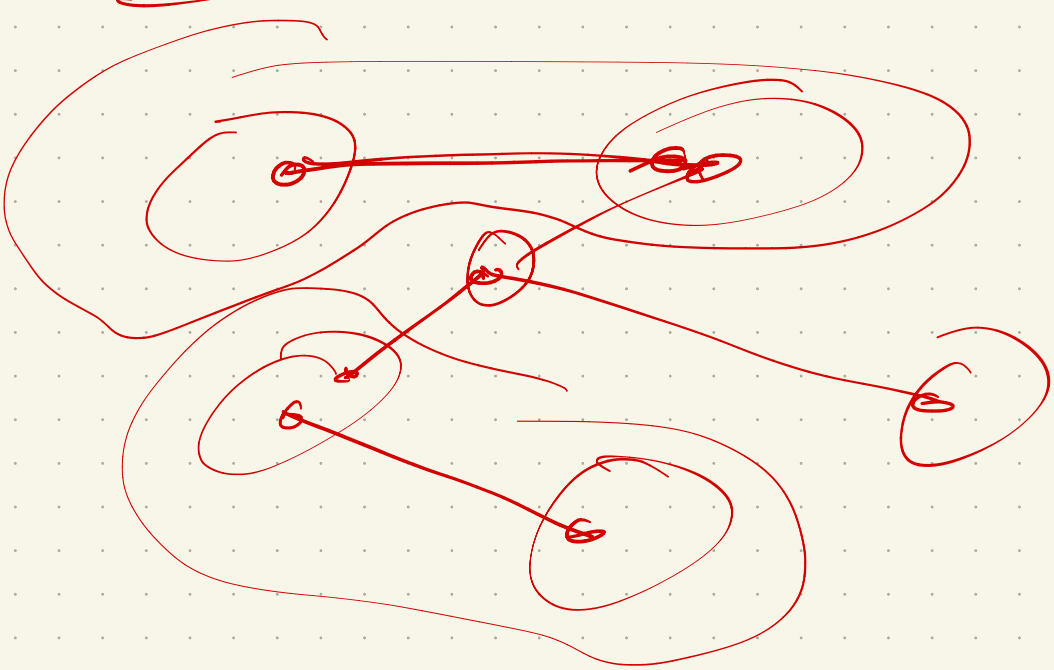
If everything isn't connected, must have some safe edge.

not a tree

Why?

Pick a component of F. Make it S + rest V − S, + apply that lemma.

Add it + recurse.



We'll see 3 ways:

Start w/ n components

1. Find all safe edges. Add them + recurse.

Boruvka

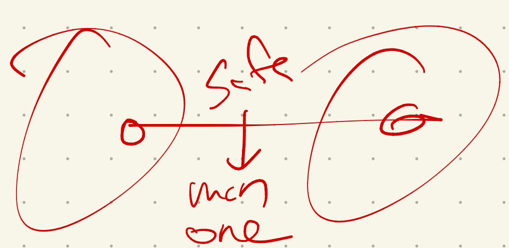
2. Keep a single connected component

At each iteration, add 1 safe edge.

3. Sort edges + loop through them.

If edge is safe, add it.

all 3 use this S, V−S lemma



First one: (1926-ish)

Borůvka: Add *ALL* the safe edges and recurse.

each vertex in its own component

2 components

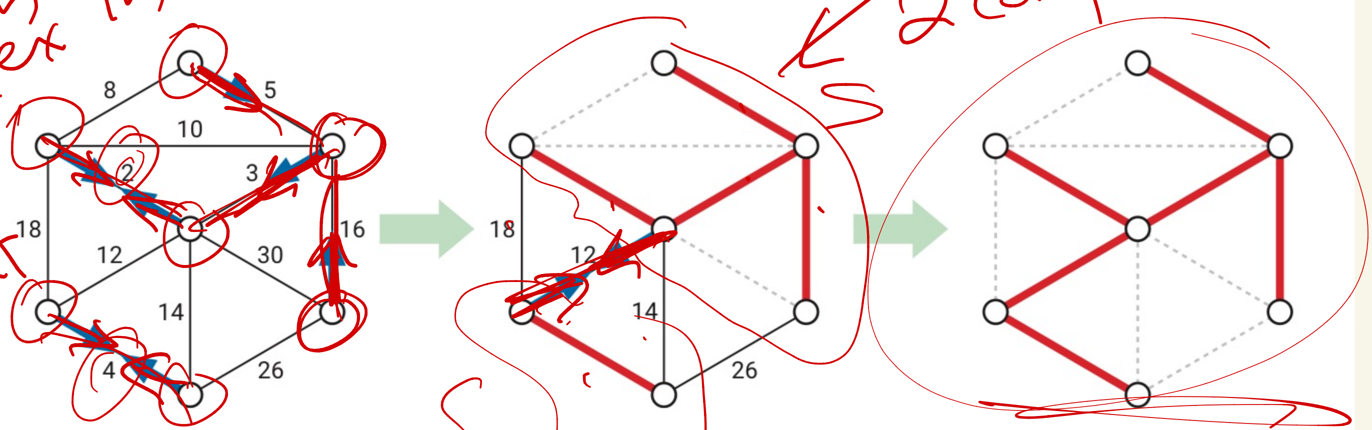


Figure 7.3. Borůvka's algorithm run on the example graph. Thick red edges are in F; dashed edges are useless. Arrows point along each component's safe edge. The algorithm ends after just two iterations.

So we need to:

loop: While more than 1 component: (not a tree)

- Track components
- Find all safe edges
- Add them

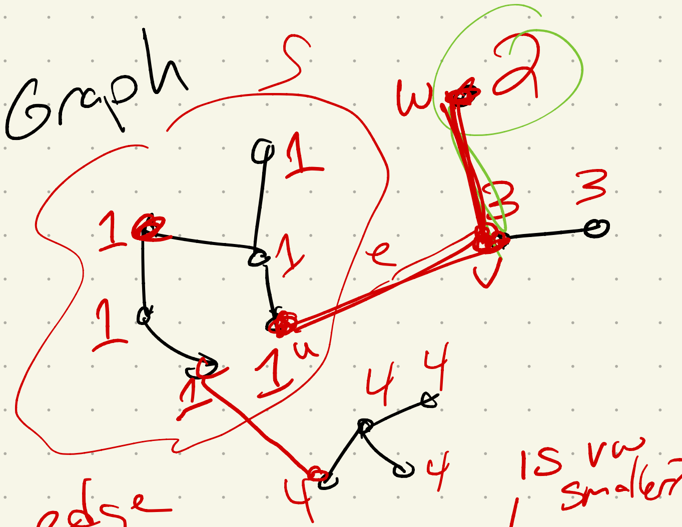
More formally:

```
BORŮVKA(V, E):
  F = (V, ∅)
  count ← COUNTANDLABEL(F)
  while count > 1
    ADDALLSAFEEDGES(E, F, count)
    count ← COUNTANDLABEL(F)
  return F
```

no edges in F

key for runtime

Graph



min edge

Safe:

is vw smaller?

uv	wv	uv	
1	2	3	4

each comp gets 1 safe edge

```
ADDALLSAFEEDGES(E, F, count):
  for i ← 1 to count
    safe[i] ← NULL
  for each edge uv ∈ E
    if comp(u) ≠ comp(v)
      if safe[comp(u)] = NULL  or  w(uv) < w(safe[comp(u)])
        safe[comp(u)] ← uv
      if safe[comp(v)] = NULL  or  w(uv) < w(safe[comp(v)])
        safe[comp(v)] ← uv
  for i ← 1 to count
    add safe[i] to F
```

O(V)

O(E)

Uses WFS-variant from Ch 5:

```
COUNTANDLABEL(G):
  count ← 0
  for all vertices v
    unmark v
  for all vertices v
    if v is unmarked
      count ← count + 1
      LABELONE(v, count)
  return count
```

```
⟨⟨Label one component⟩⟩
LABELONE(v, count):
  while the bag is not empty
    take v from the bag
    if v is unmarked
      mark v
      comp(v) ← count
      for each edge vw
        put w into the bag
```

DFS

Correctness:

- MST must have any safe edge
- We keep computing safe edges & adding
- Stop when # connected components = 1

$\Longrightarrow$ Have the MST!

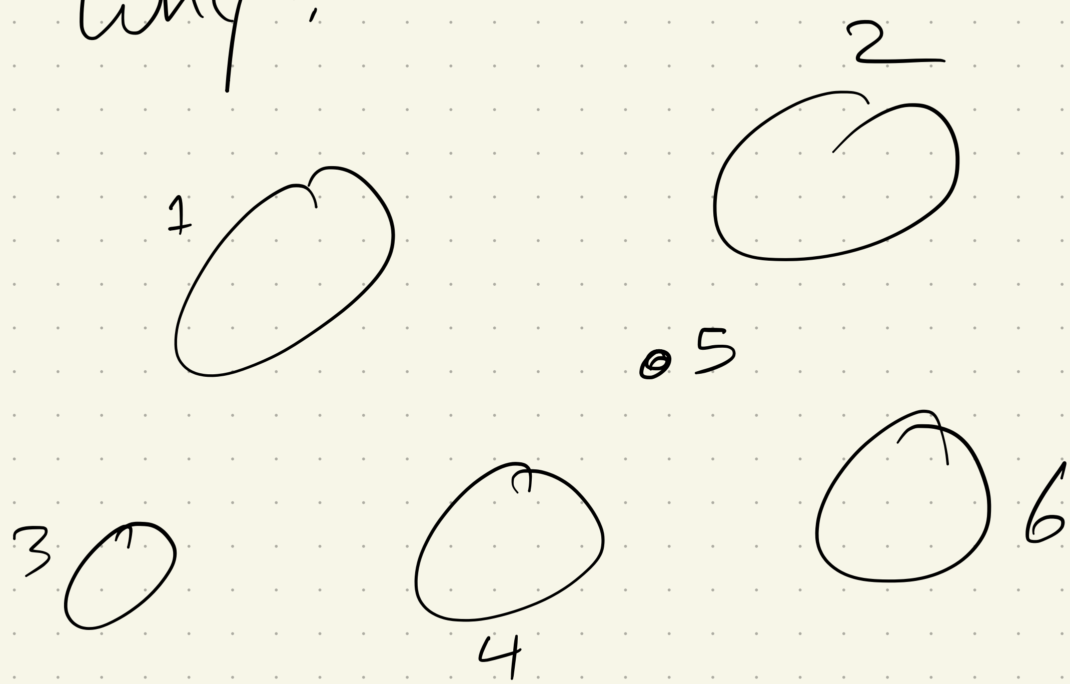
Run time:

A bit trickier!

Depends on how many safe edges we get.

Claim: There are at least #components/2 safe edges each time.

Why?



So: runtime:

```
ADDALLSAFEEDGES(E, F, count):
  for i ← 1 to count
      safe[i] ← NULL
  for each edge uv ∈ E
      if comp(u) ≠ comp(v)
          if safe[comp(u)] = NULL  or  w(uv) < w(safe[comp(u)])
              safe[comp(u)] ← uv
          if safe[comp(v)] = NULL  or  w(uv) < w(safe[comp(v)])
              safe[comp(v)] ← uv
  for i ← 1 to count
      add safe[i] to F
```

Looks at each vertex & edge in worst case:

```
BORŮVKA(V, E):
  F = (V, ∅)
  count ← COUNTANDLABEL(F)
  while count > 1
      ADDALLSAFEEDGES(E, F, count)
      count ← COUNTANDLABEL(F)
  return F
```

BFS/DFS on tree:

How many iterations?