Algorithms – Spring '25

Strongly & weakly
connected comps.
Intro to MST

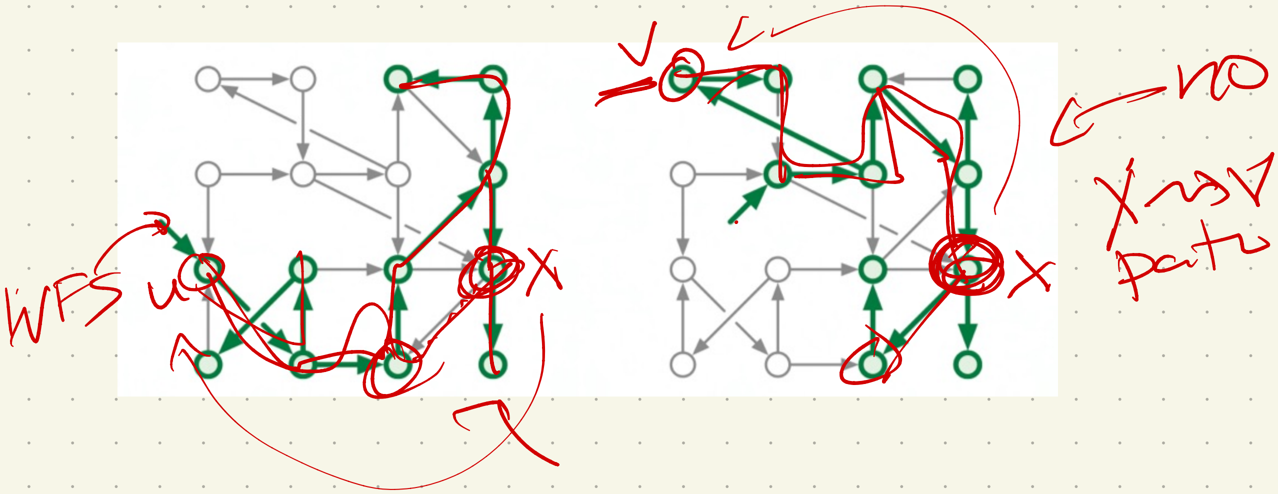
Recap

- No class next week - happy break!
- HW + readings for after break are posted
- Instructor feedback form should be posted (check email)
- Tuesday after break
  - ↳ out of town, no office hours
  - ↳ makeup on Wed/Thurs

Strong connectivity

In an undirected graph, if $u \rightsquigarrow v$, then $v \rightsquigarrow u$.

Not true in directed case:



So 2 notions:

weak connectivity:
u,v are weakly connected if either $u \rightsquigarrow v$ or $v \rightsquigarrow u$

strong connectivity:
both $u \rightsquigarrow v$ and $v \rightsquigarrow u$

related: SCCs strongly conn comps.

Can actually order the strongly connected pieces of a graph:

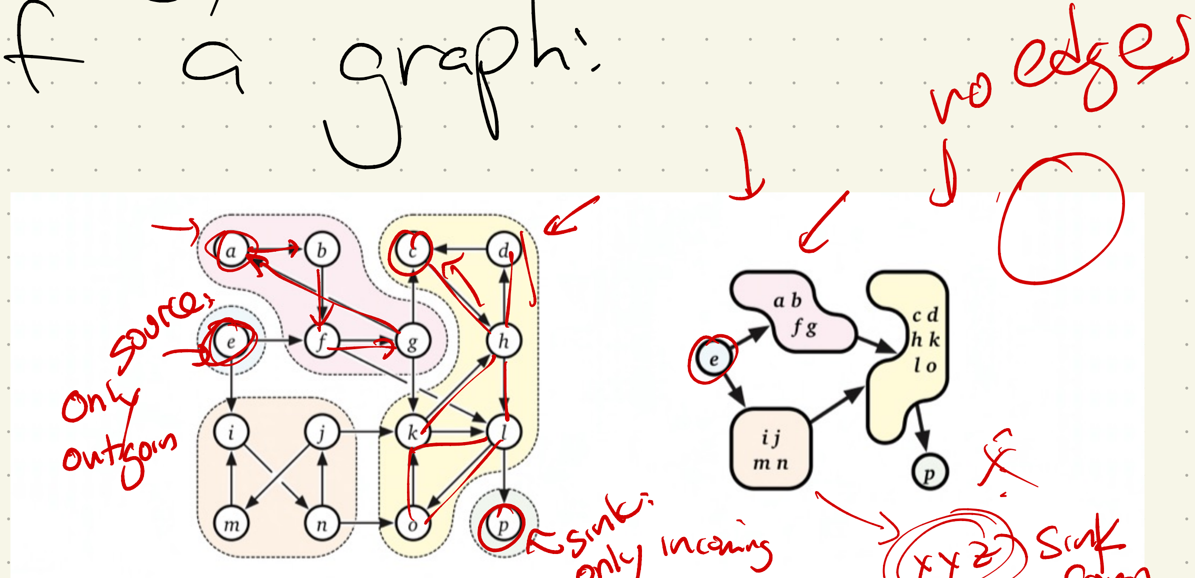
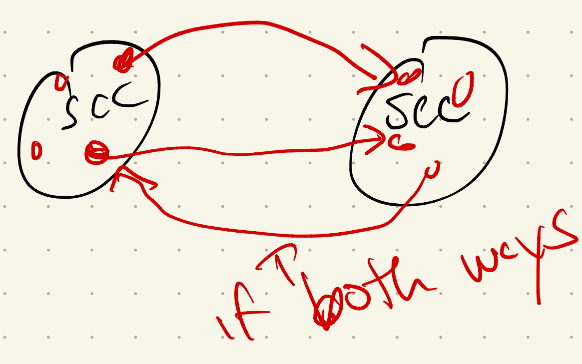


Figure 6.13. The strong components of a graph G and the strong component graph $scc(G)$.

How?

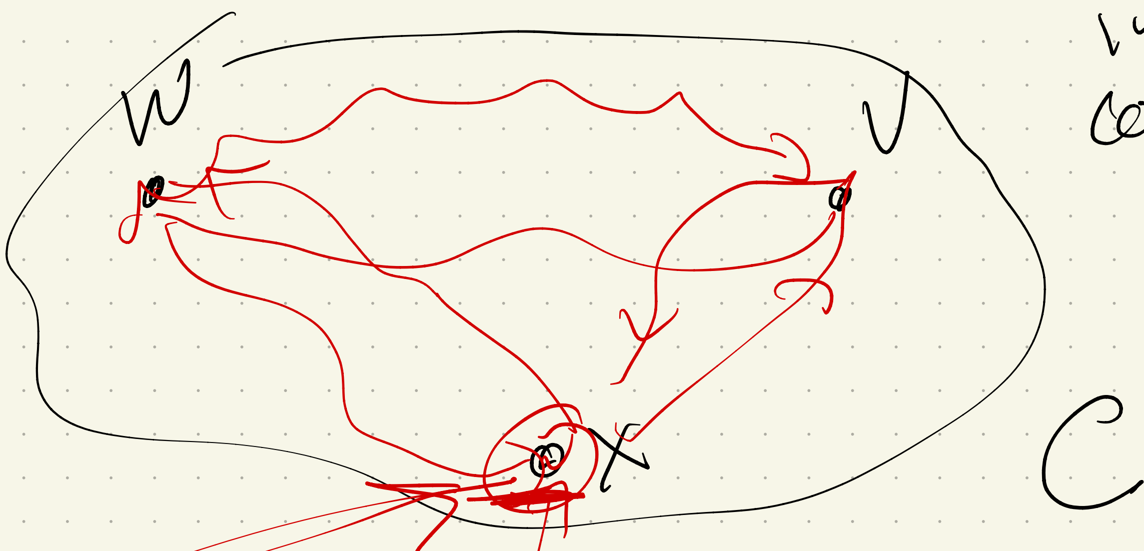
- Well, each component either isn't connected, or only has 1-way edges. Why?



More formally:

Every strong cc must have at least one vertex with no parent. (or parent outside comp)

Proof: Consider two vertices in component



Let x be first vertex in clock-order in scc:

Possible to compute SCCs in $O(V+E)$ time.

Need good sinks!

DFS (rev(G))

↳ find sinks

Then, reverse back to G & run DFS from them.

(See book for details)

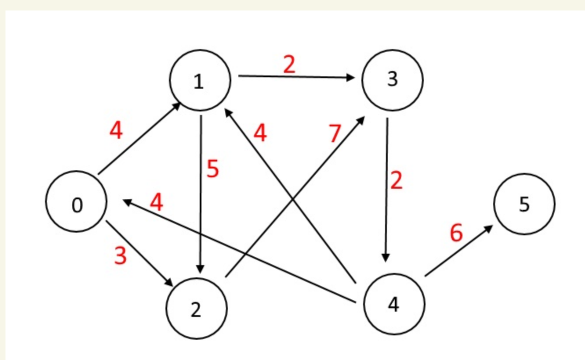
Next module:

Minimum Spanning trees

& shortest paths.

Both are on weighted graphs - so $G=(V,E)$, plus $w: E \to \mathbb{R}$ (or $\mathbb{R}^+$)

picture:



Weighted Graph

vertex 0

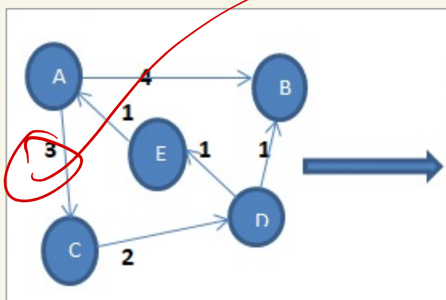
$\boxed{1}$: 4

↓

$\boxed{2}$: 3

↑

weight of edge



Weighted Graph

	A	B	C	D	E
A	0	4	3	0	0
B	0	0	0	0	0
C	0	0	0	2	0
D	0	1	0	0	1
E	1	0	0	0	0

Adjacency matrix

Minimum Spanning Trees

Goal: Given a weighted undirected Graph G, $w: E \to \mathbb{R}^+$ the weight function, find a spanning tree T of G that minimizes:

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

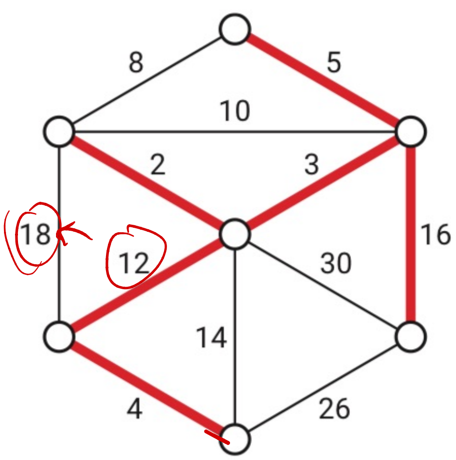


Figure 7.1. A weighted graph and its minimum spanning tree.

Motivation:

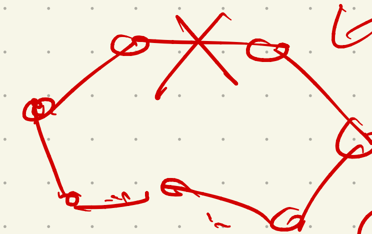
Connectivity: tree is minimally connected subgraph.

First:

Does it have to be a tree?

Yes. Why?

think contradiction: if not: cycle



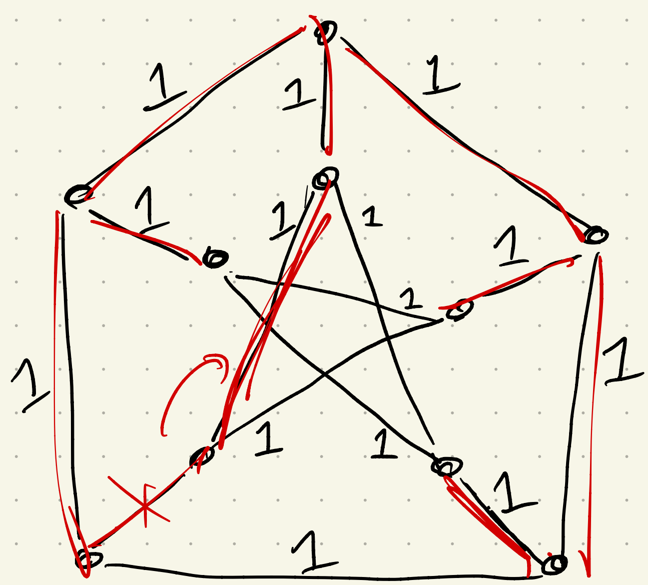
sum of all edges

if all positive, can delete one and do better

Second:

These are obviously not unique!

Ex:



tree?

any subtree works

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

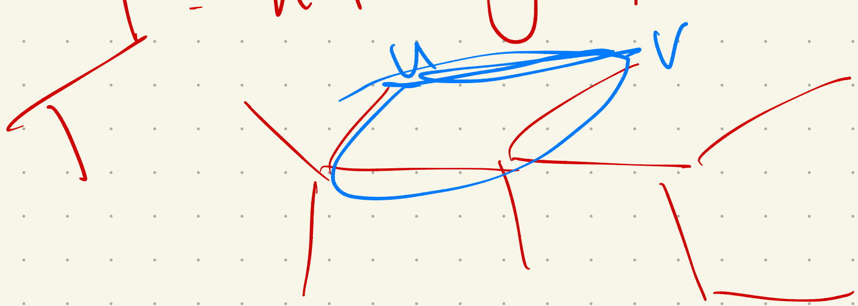
<u>Lemma</u>: Assuming all edge weights are distinct, then MST is unique.

<u>Pf</u>: By contradiction:
Suppose T & T' are both MSTs, with $T \neq T'$

- $T \cup T'$ contains a cycle → T' has at least one edge <u>not</u> in T
- That cycle must have 2 edges of equal weight

$\Rightarrow$ Contradiction!

T: $n-1$ edges, no cycles



$u v$ path in $T + \{u,v\}$

uv

Now, what if weights aren't unique?

Just need a way to consistently break ties.

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)

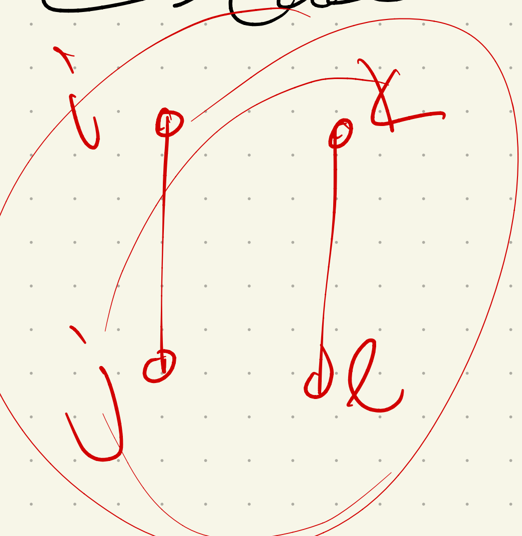
5 11 1 8

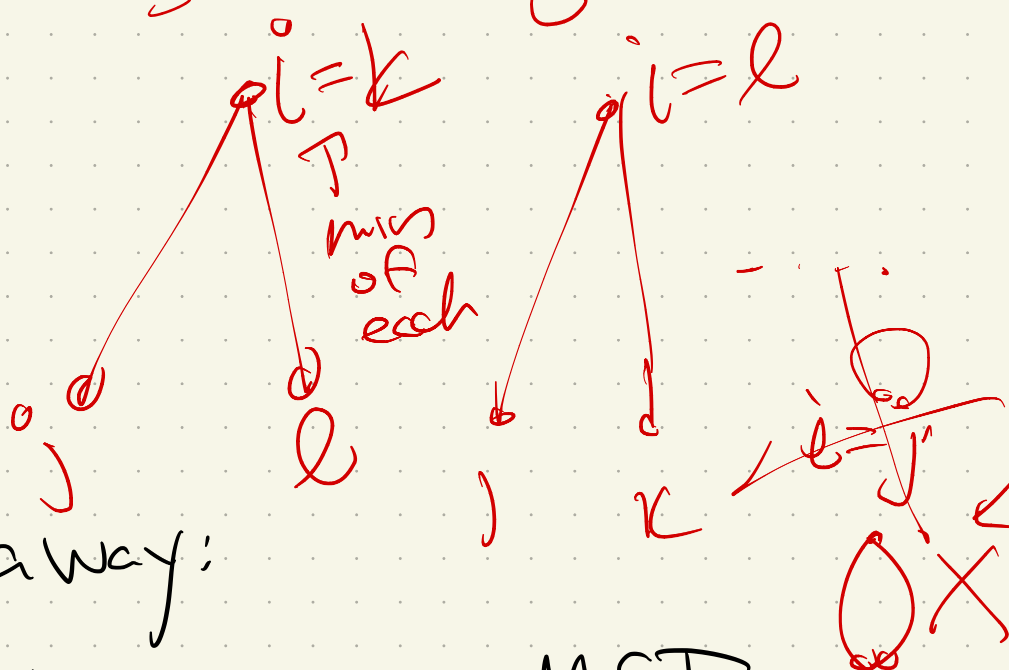
not tied

if min index vertex is smaller return that edge

cases!

we know edges have same weight





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!

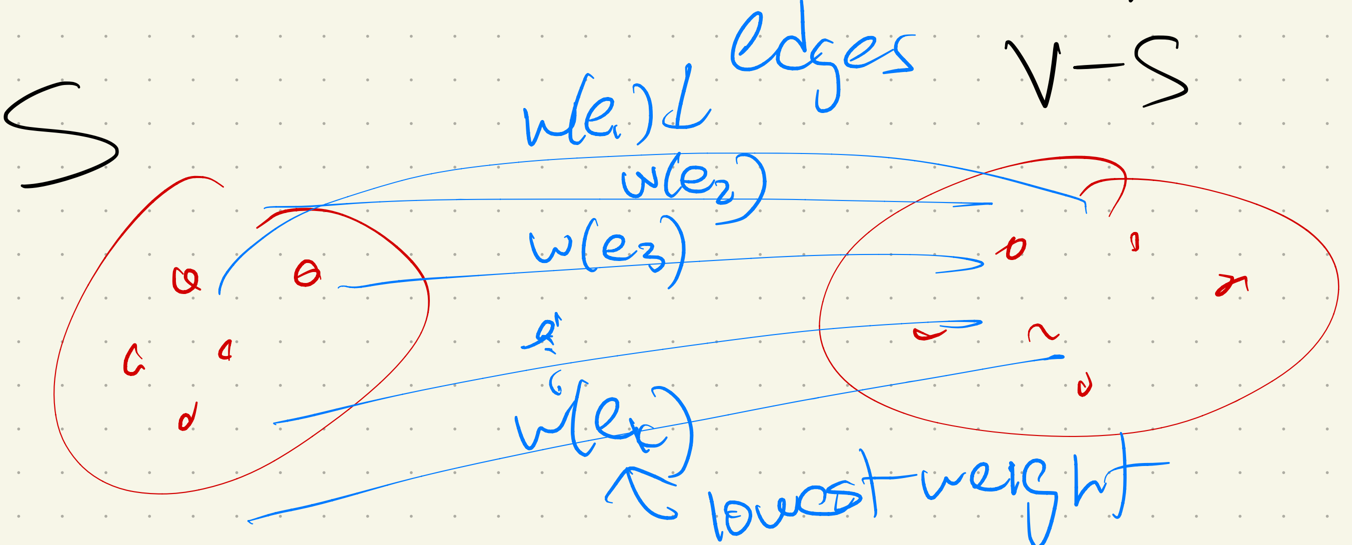
This is **highly** unusual, & there's a reason for it:

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

(Way beyond this class...)

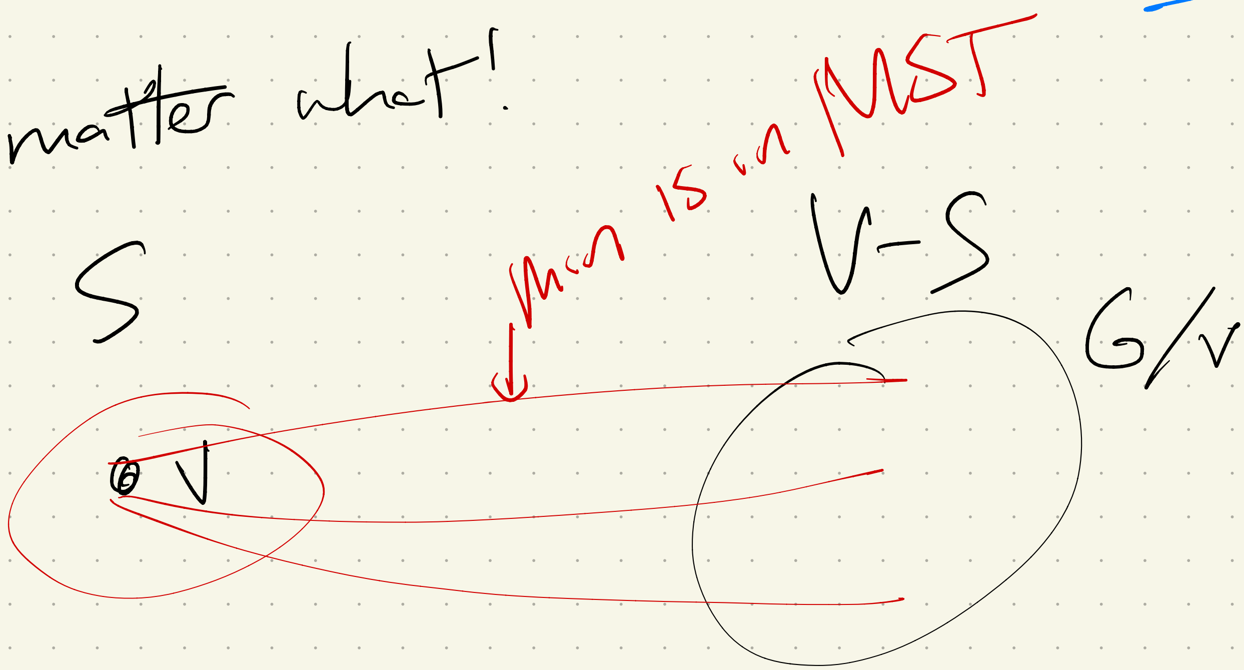
Key property:

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

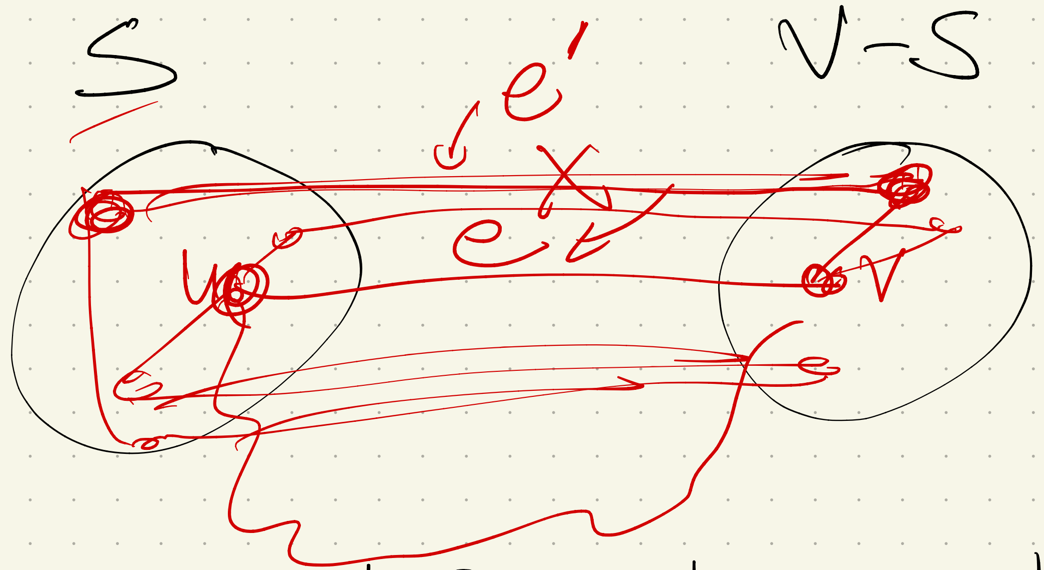


The MST will always contain the lowest edge connecting the two sides.

No matter what!



Proof: consider minimum edge $e = \{u, v\}$



Suppose MST does not contain e.

$\Rightarrow$ MST must have some u−v path not including e → call this p

p + e is a cycle.

for any S, & V−S with $u \in S$ and $v \in V-S$, p must use some edge from S to V−S.

Consider those edges → all larger than e, Generic Algorithm:

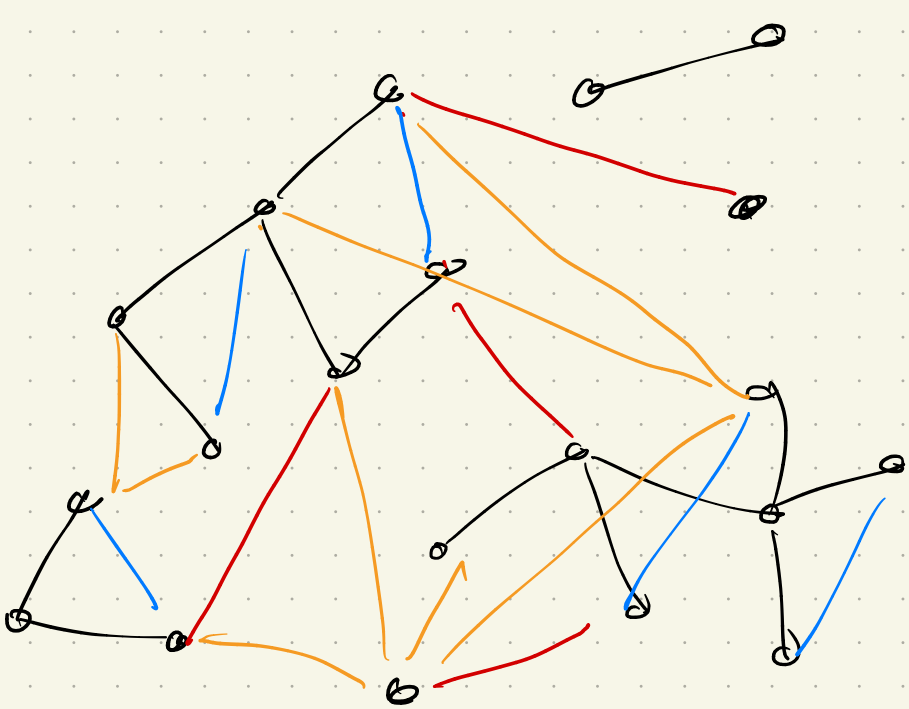
Build a forest: an acyclic subgraph.

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

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

also edges that are neither.

F:



So idea:

Add safe edges until you get a tree

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

Why?

Add it & recurse.

We'll see 3 ways:

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

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.

First one: (1926-ish)

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

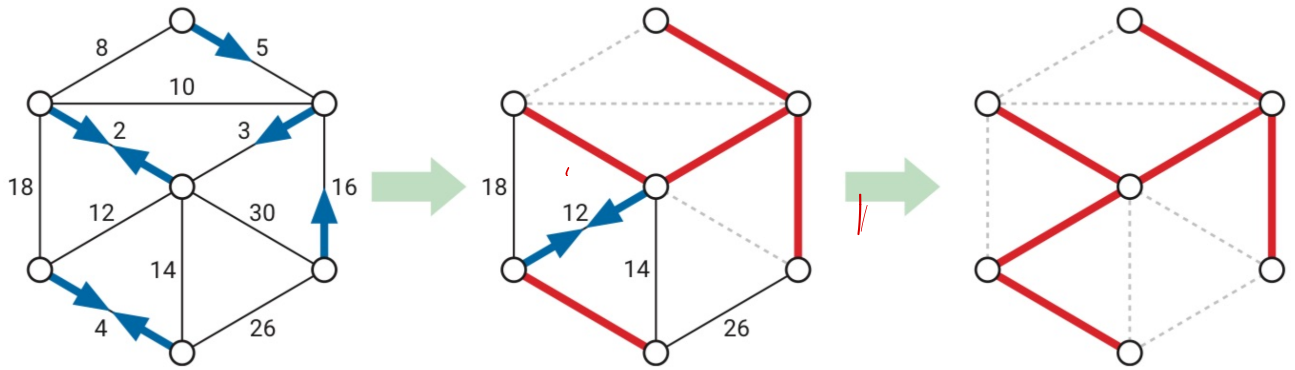


Figure 7.3. Borůvka's algorithm run on the example graph. Thick red edges are in F; dashed edges are useless. Arrows point along each component's safe edge. The algorithm ends after just two iterations.

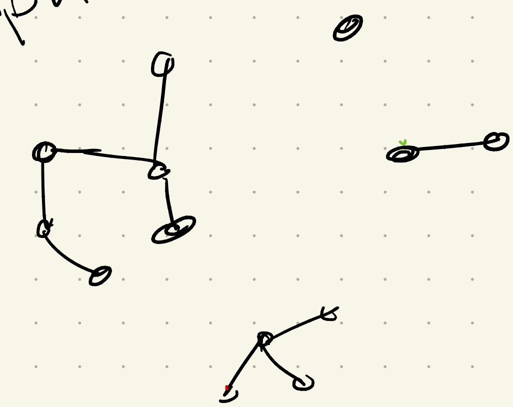
So we need to:

While more than 1 component:

- Track components
- Find all safe edges
- Add them

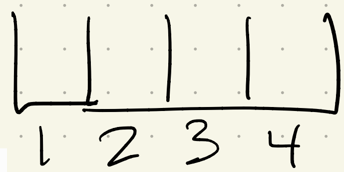
More formally:

Graph



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

Safe:




```
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
```

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
```

Correctness:

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

$\Longrightarrow$ Have the MST!

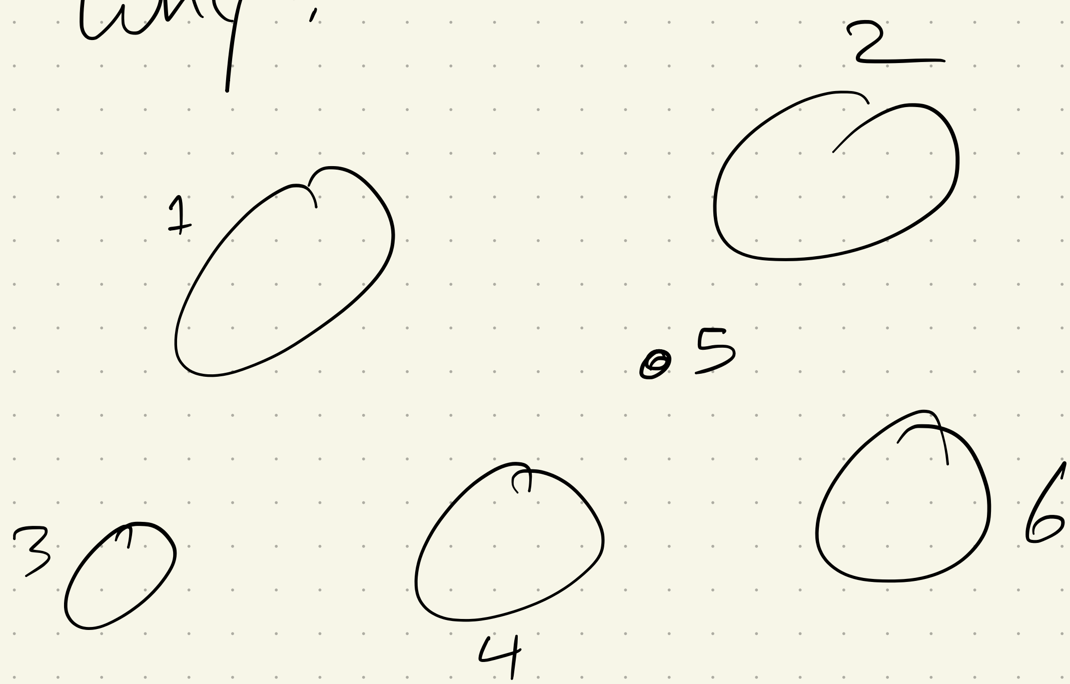
Run time:

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?

<u>So</u>: 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?