Algorithms - Spring '25

Strongly & weakly connected comps.
Intro to MST

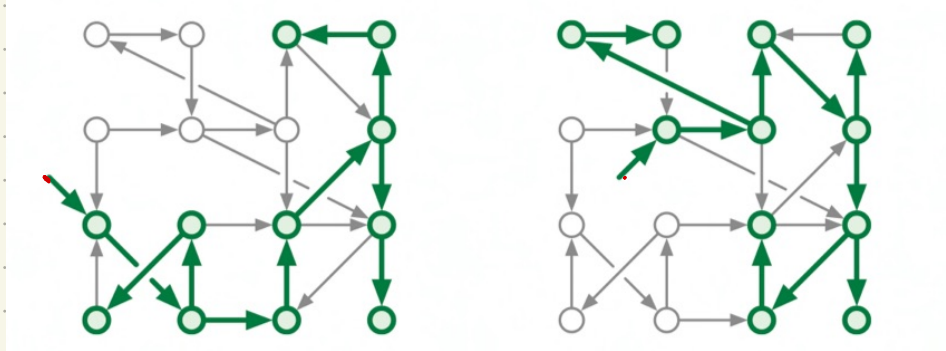
Recap

- No class next week - happy break!
- HW + readings for after break are posted
- Instructor feedback form should be posted (check email)

Strong connectivity

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

Not true in directed case:



So 2 notions:

weak connectivity:

strong connectivity:

related: SCCs

Can actually order the strongly connected pieces of a graph:

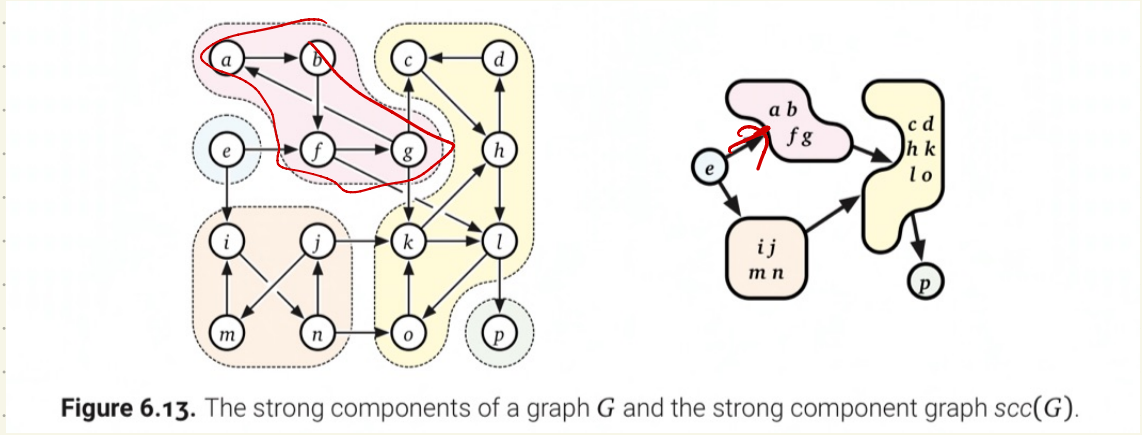


Figure 6.13. The strong components of a graph G and the strong component graph $scc(G)$.

How?

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

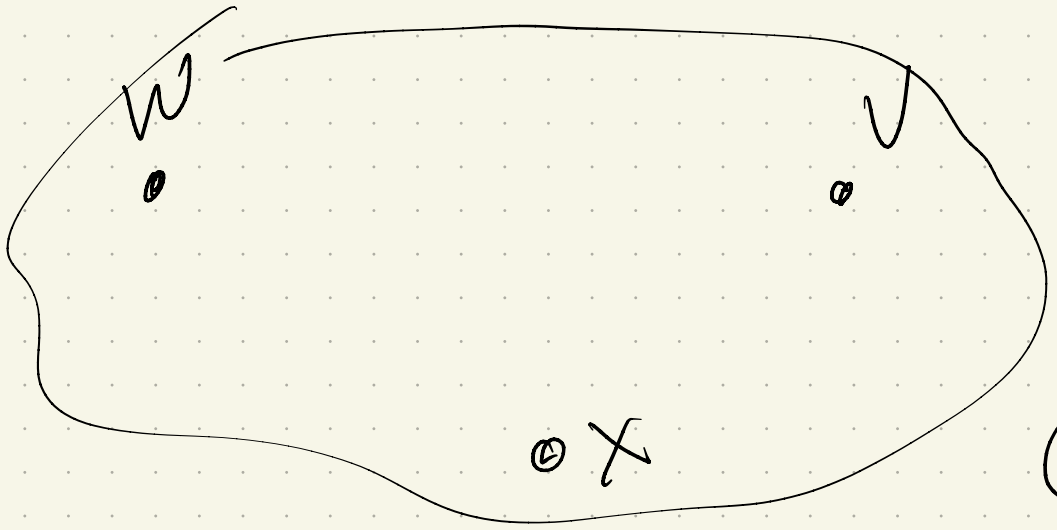
scc

scc

More formally:

Every strong cc must have at least one vertex with no parent.

Proof: Consider two vertices in component



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

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)

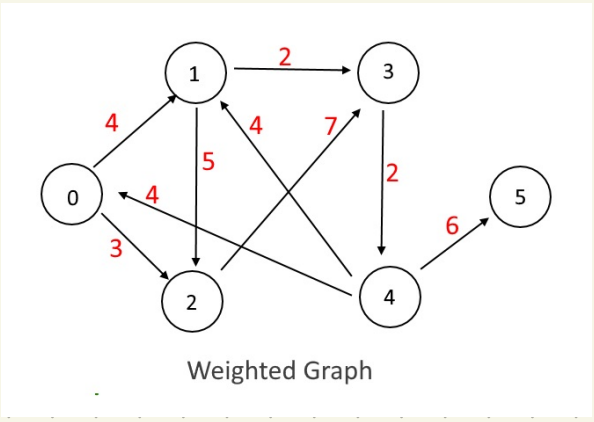
Next module:

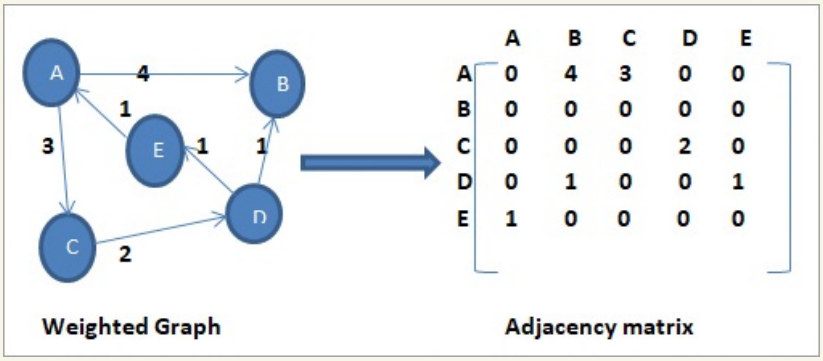
Minimum Spanning trees

& shortest paths.

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

picture:





Minimum Spanning Trees

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

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

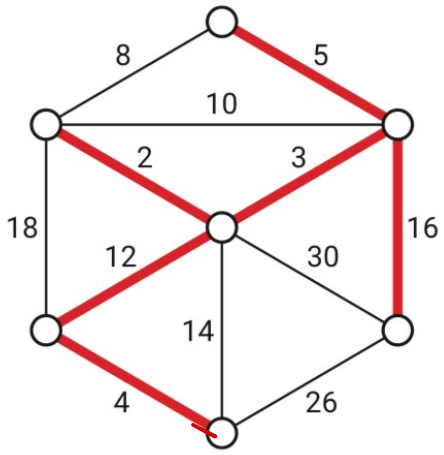


Figure 7.1. A weighted graph and its minimum spanning tree.

Motivation:

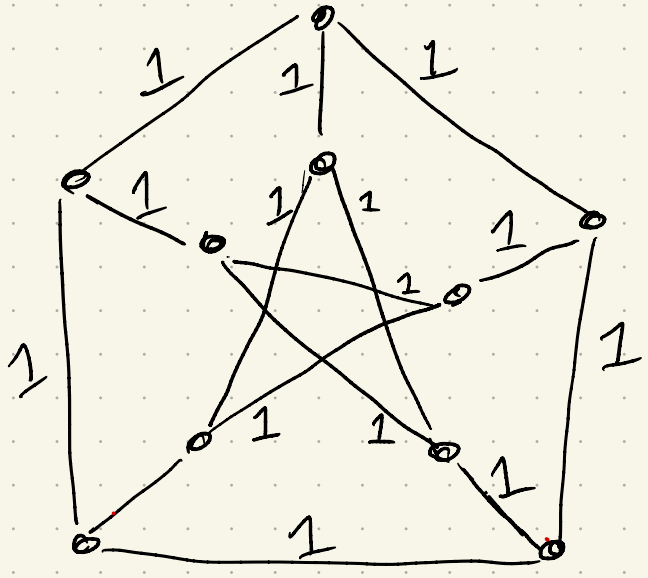
First:

Does it have to be a tree?

Second:

These are obviously not unique!

Ex:



tree?

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 $T \neq T'$

- $T \cup T'$ contains a cycle
- That cycle must have 2 edges of equal weight

$\Rightarrow$ Contradiction!

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)
```

not tied

→ cases!

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

V-S

S

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

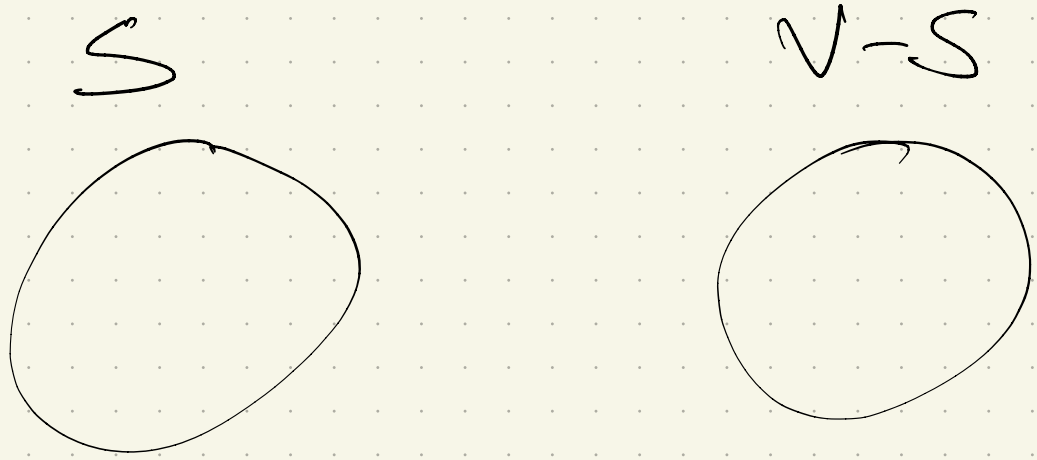
No matter what!

S

V-S

G/V

Proof: consider minimum edge e



Suppose MST does not contain e.

Generic Algorithm:

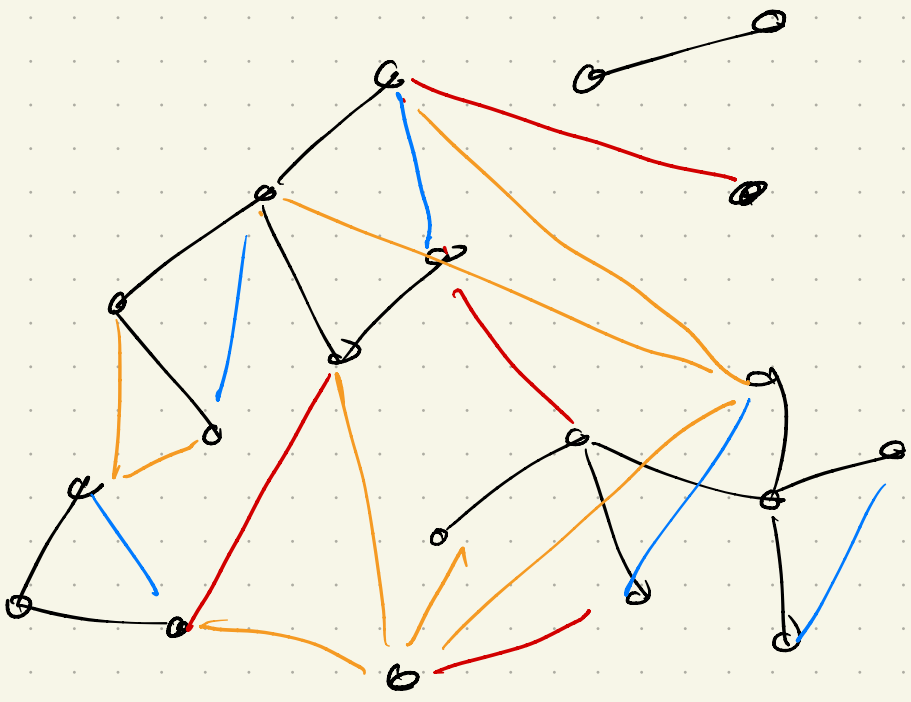
Build a forest: an acyclic subgraph.

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.

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.

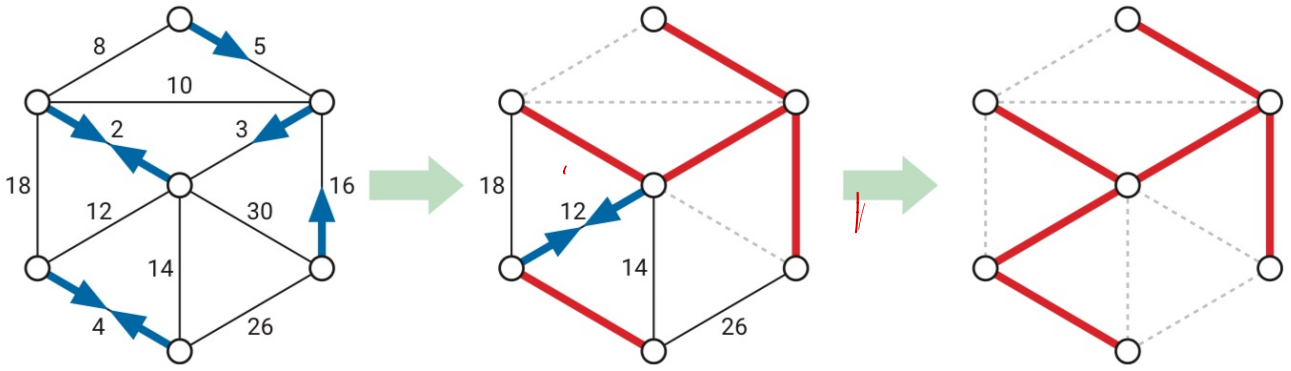


Figure 7.3. Borůvka's algorithm run on the example graph. Thick red edges are in F; dashed edges are useless. Arrows point along each component's safe edge. The algorithm ends after just two iterations.

So we need to:

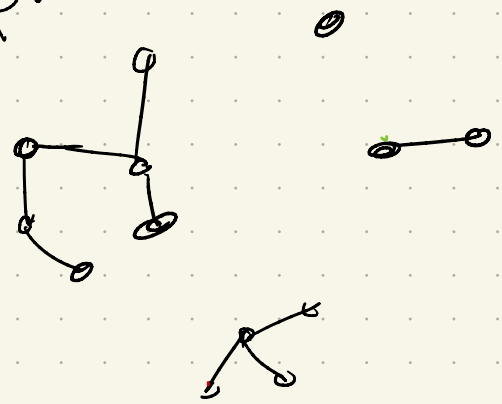
While more than 1 component:

- Track components
- Find all safe edges
- Add them

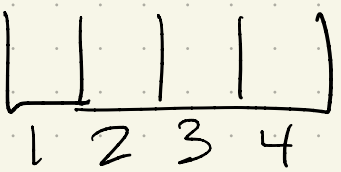
More formally:

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

Graph



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

$\Rightarrow$ Have the MST!

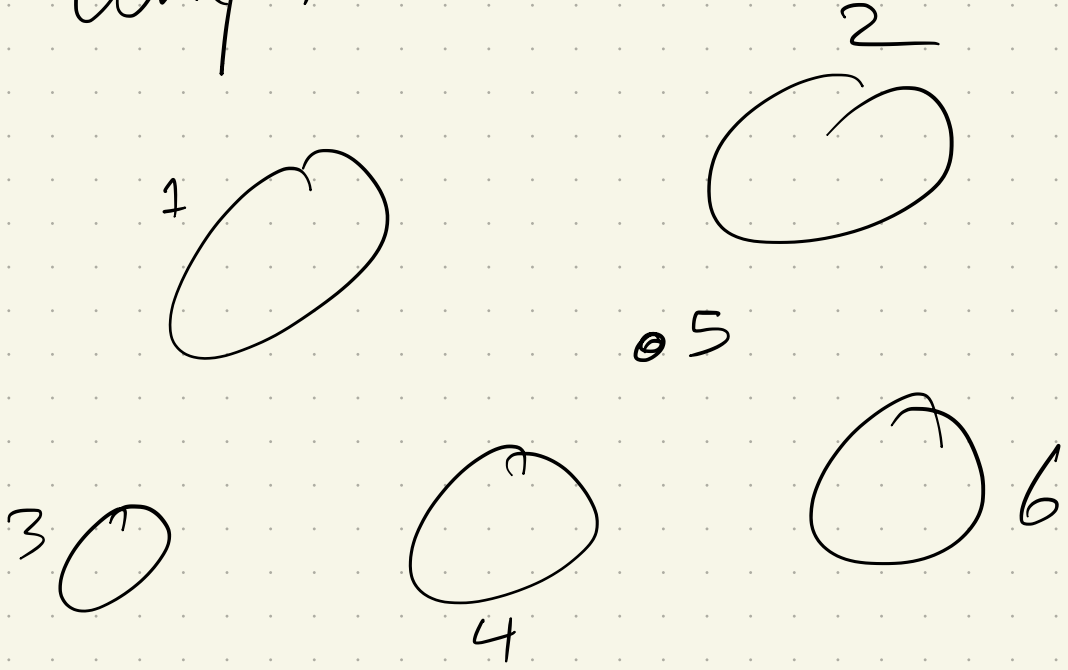
Run time:

A bit trickier!

Depends on how many safe edges we get.

Claim: There are at least $\#\text{components}/2$ 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?

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?