Algorithms

Minimum Spanning Trees

Recap

- HW5 due next ~~Wed~~ Thurs.
- Have a good break!
- Midterm recap:

  Average: 64

- Midterm letter grades in banner.

  Calculation:

$$\left(\left(\text{HW average} * .2\right) + \left(\text{MT} * .2\right) + \left(.05 * \text{Per.}\right)\right) \div .45) = \% + 5\%$$

- No reading due next Wed.

Next: Minimum Spanning Trees!

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

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

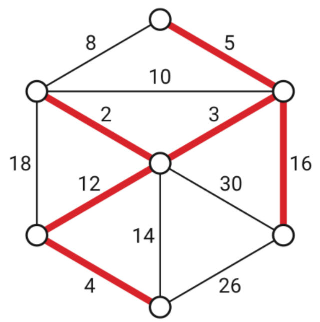


Figure 7.1. A weighted graph and its minimum spanning tree.

Motivation: Everywhere

First:

Does it have to be a tree?

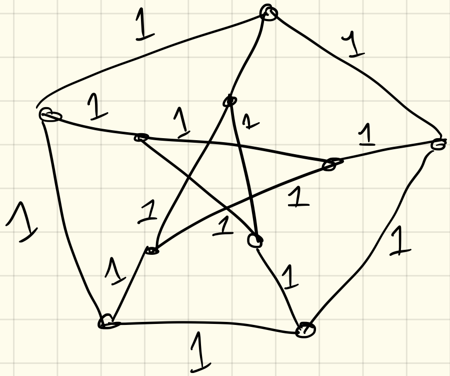
min way to connect every vertex:

if cycle, remove an edge (assuming positive weights)

Second:

These are "obviously" not unique!

Ex:



tree? any spanning tree has weight $n-1$

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
- That cycle must have 2 edges of equal weight

$\Rightarrow$ Contradiction!

Can argue $w(e') \leq w(e)$

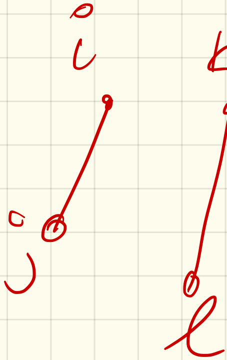
& $w(e) \leq w(e')$

so $w(e) = w(e')$

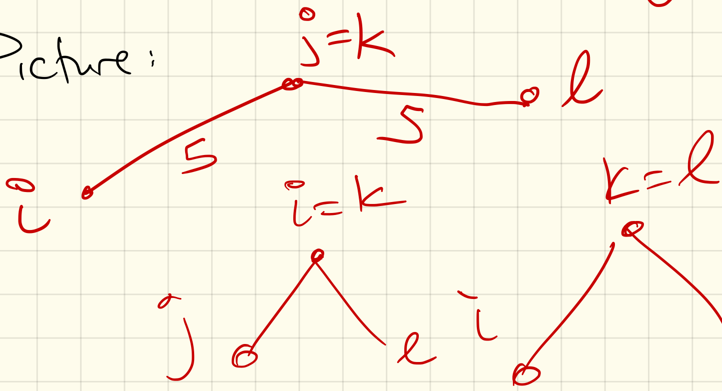
Now, what if weights <u>aren't</u> unique?

Just need a way to consistently break ties.

$\underline{\textsc{ShorterEdge}(i,j,k,l)}$	
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)



Picture:



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!

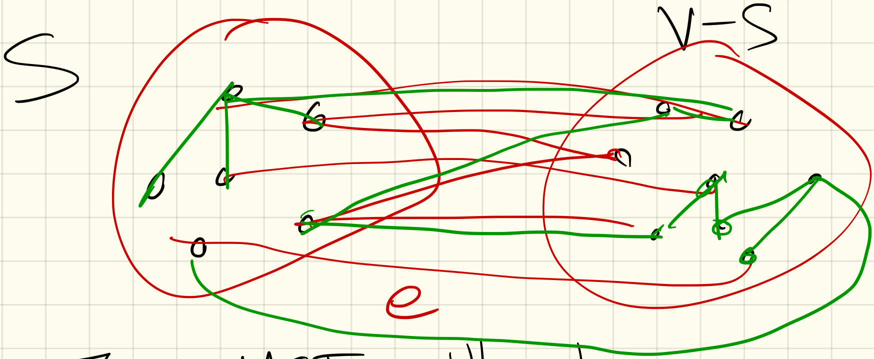
This is *highly* unusual, & there's a reason for it:

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.

$w(e) <$ any other edge from S to V-S

Pf

Consider the MST & suppose it doesn't contain e.

MST + e has a cycle, which has another edge e' from S to V-S.

T - e' + e is a s.t., & is better.

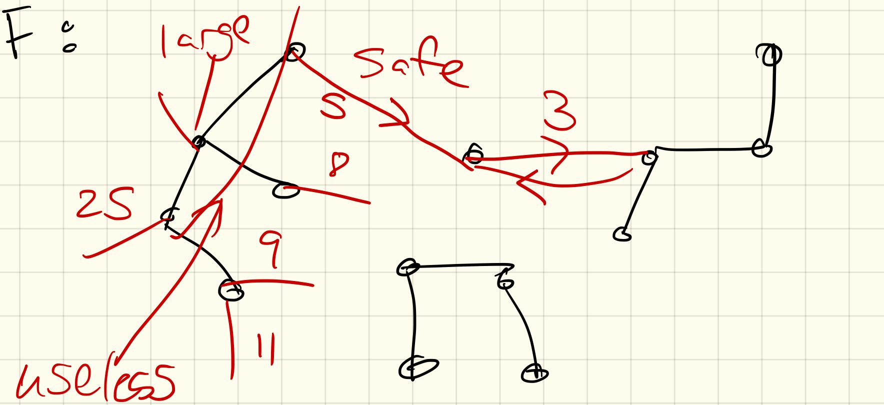
Generic Algorithm:

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.

F:

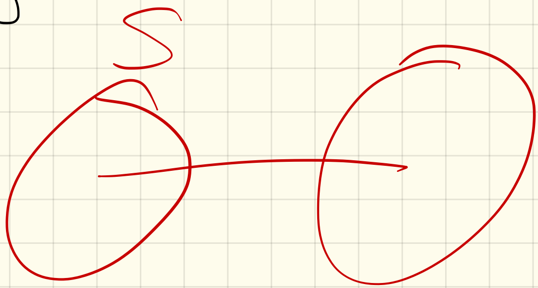


So idea:

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.

differ: runtime

First one: (1926-ish)

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

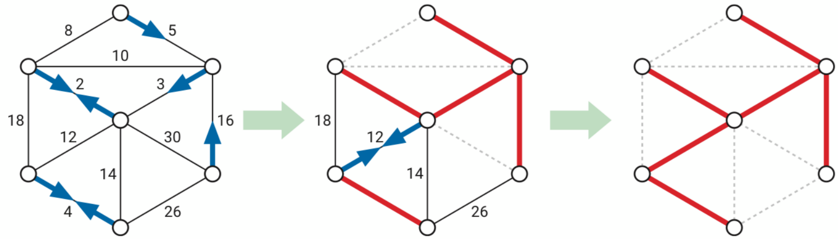


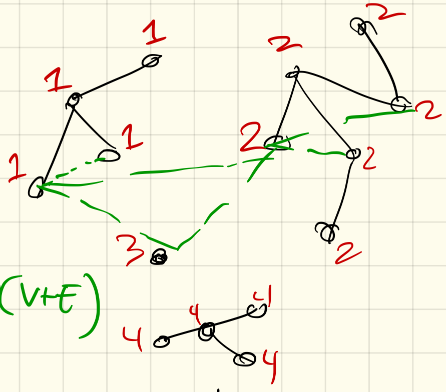
Figure 7.3. Borůvka's algorithm run on the example graph. Thick red edges are in F; dashed edges are useless. Arrows point along each component's safe edge. The algorithm ends after just two iterations.

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
```

$O(V+E)$?

$O(V+E)$

$O(V+E)$

repeats ?

?

Safe: | | | | |
1 2 3 4

track min each in each component

```
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
```

$O(E)$

if not useless

if e in min out of u or v's comp., save it

$O(V+E)$

Uses WFS-variant from Monday:

```
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
```

$O(V+E)$

```
⟨⟨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!

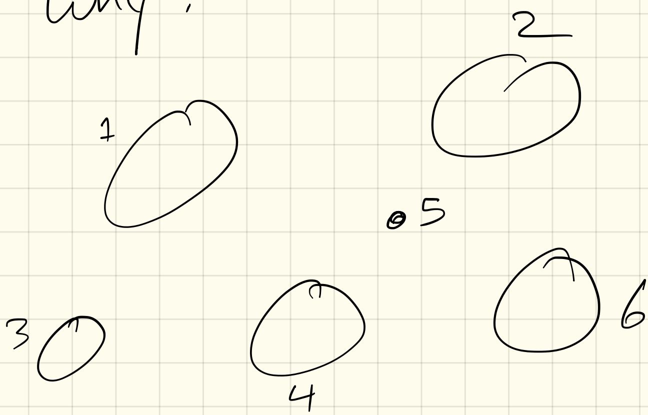
Run time:

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?

After break:

2 more: Prim
Kruskal

(These are greedy also, but make a different choice of what F is & which safe edge(s) to add.)