# Algorithms

## MST (part 2)

# Recap

- Reading due Friday
- HW due tomorrow by 5pm
  (in main office or to me)
- Next week: sub on Wed. & Fri.
  (In class work day one of the days
  ↳ useful for HW!!)

- Next HW:
  Oral grading on Monday, 11/4 & Tuesday, 11/5
  Sign-up in class, next Monday!

Next: Minimum Spanning Trees!

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

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

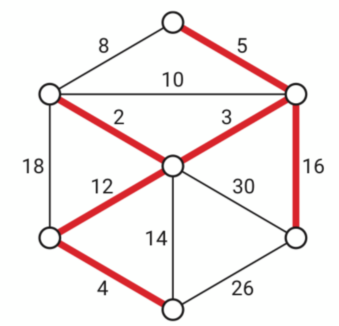



**Figure 7.1.** A weighted graph and its minimum spanning tree.

Assumption: - These are unique.

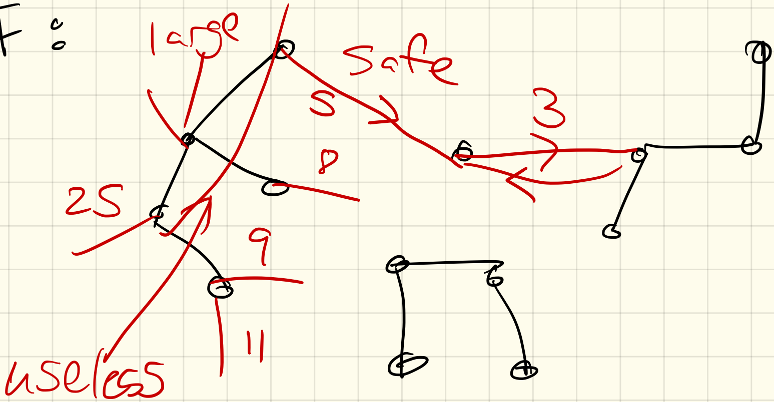
Generic Algorithm:

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?

*Lemma*: For *any* split of G into 2 sets $S$ & $V-S$, the minimum edge from $S$ to $V-S$ will be in MST.

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.

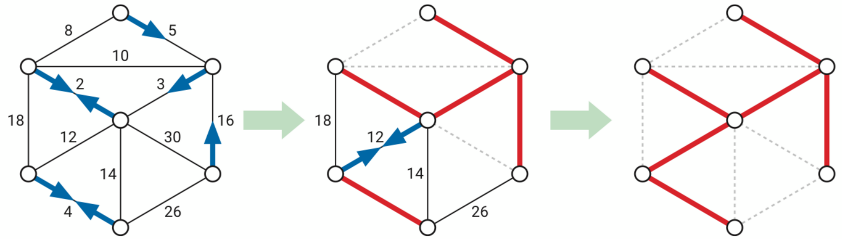


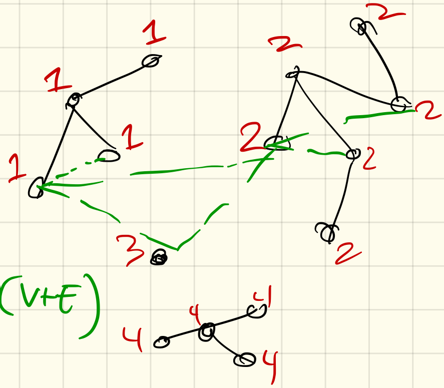

**Figure 7.3.** Borůvka's algorithm run on the example graph. Thick red edges are in $F$; dashed edges are useless. Arrows point along each component's safe edge. The algorithm ends after just two iterations.

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

$\Rightarrow$ Have the MST!

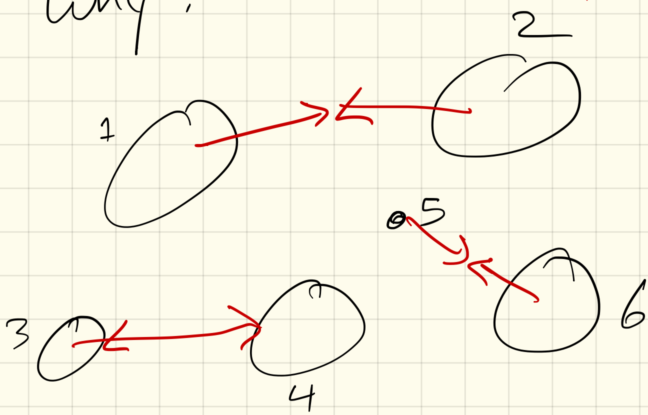
## Run time:

A bit trickier! $O(V+E) + O\left((\#\text{loop repeats}) \cdot (V+E)\right)$

Depends on how many safe edges we get.

**Claim:** There are at least #components/2 safe edges each time. (could be #comp)

Why?




while (#comp > 1)
$O(V+E)$

→ since reduce by $\frac{1}{2}$ each time, & starts $= n$, $\leq \log_2 V$ times

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:

$O(V+E)$

```
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?

$O(\log_2 V)$

$\Rightarrow O((V+E)\cdot\log_2 V) = \boxed{O(E\log V)}$

Prim's algorithm:

(really Jarník, we think)

Keep one spanning subtree.

O(V) times

While $|T| \neq n$

add next safe edge

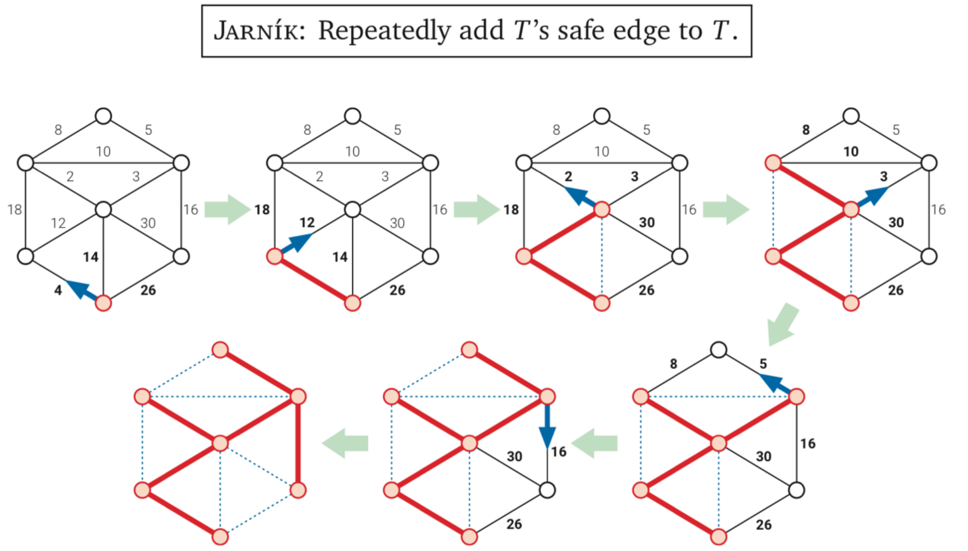



**Figure 7.4.** Jarník's algorithm run on the example graph, starting with the bottom vertex. At each stage, thick red edges are in $T$, an arrow points along $T$'s safe edge; and dashed edges are useless.

Implementation:

From all edges going from $V(T)$ to $V(G)-V(T)$, add <u>safe</u> one.

??

min weight edge

<u>Q:</u> Which data structure?

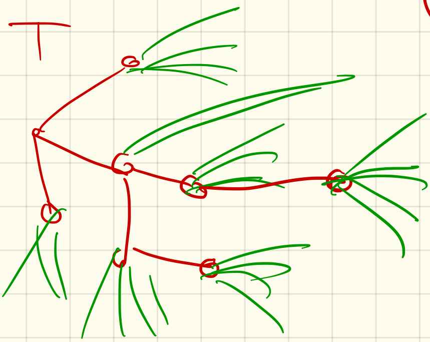
heap (or priority queue)

- ExtractMin: $O(1)$
- Insert or deleteMin: $O(\log E)$

Runtime:

$O(V)$ { While $|T| < n-1$
pick min, & delete it
add new V's edges
to PQ

$d(v) \cdot \log E$



$\leq O((V+E)\log E)$

Can improve if use a better heap: Fib. heap.

(Book goes over alternative — don't worry if that's a bit unclear.)

Comparison to Boruvka:

Faster, unless $E = O(V)$

# Kruskal's Algorithm:

KRUSKAL: Scan all edges by increasing weight; if an edge is safe, add it to $F$.

If not, it must be useless.

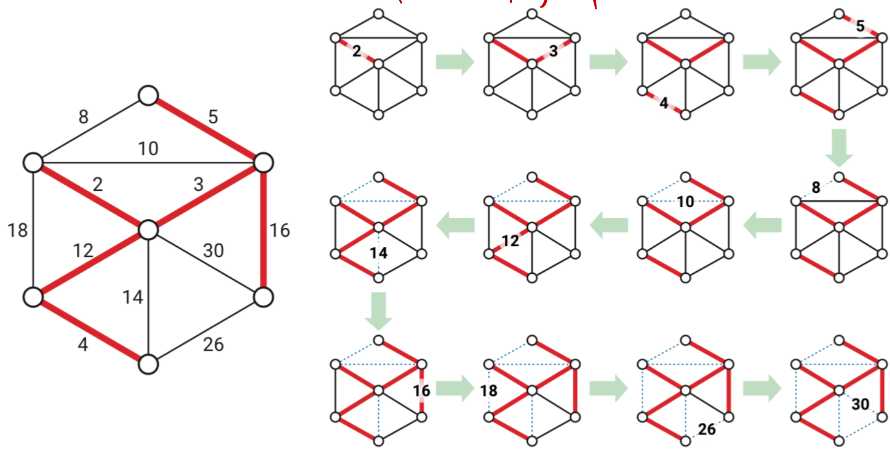



**Figure 7.6.** Kruskal's algorithm run on the example graph. Thick red edges are in $F$; thin dashed edges are useless.

How to implement?

Sort: $O(E \log E)$

Need to test if endpts of an edge are in different components

# Algorithm :

```
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
```

read in book:

$O(\log n)$ amortized per operation

Data structure:

Union find

- MAKESET($v$) — Create a set containing only the vertex $v$.
- FIND($v$) — Return an identifier unique to the set containing $v$.
- UNION($u, v$) — Replace the sets containing $u$ and $v$ with their union. (This operation decreases the number of sets.)

How fast?

amortized running time