Minimum Spanning Trees

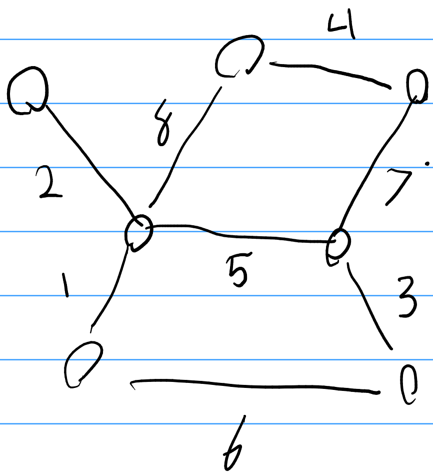
Given an undirected, weighted graph

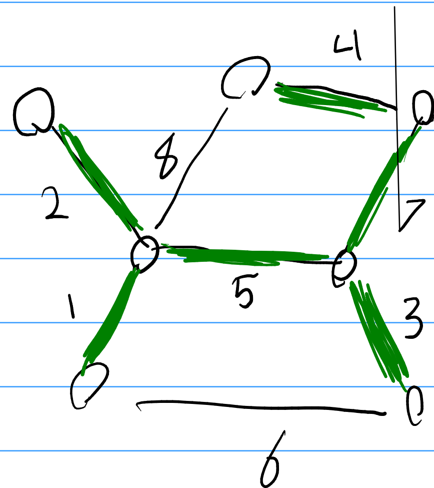
$G = (V, E)$ w/ weights $w(e)$

Goal: find a spanning tree $T \subseteq E$ with minimal weight

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

Example: Connect houses into electrical network w/ least amount of wire





Three algorithms:

Prim's	[1957?]	
Boruvka's	[1926]	$E \log V$
Kruskal's	[1956	

$E + V \log V$ w/ Fibonacci Heap (for Prim's)

In class: assume edge weights all distinct.

[We will see $\Rightarrow$ MST is unique]

All have the same (greedy) approach:

Let T be the true MST

will maintain a subset $F \subseteq T$ of edges found so far

F is "intermediate spanning forest"

F partitions the vertices into its connected components: $C_1, C_2, \ldots \subseteq V$

$F = \{\} \Rightarrow C_i = \{i\}$

$F = T \Rightarrow C_1 = V$

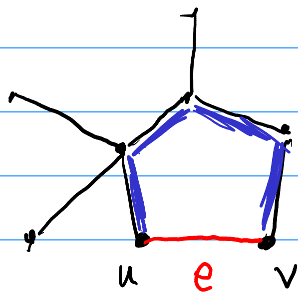
Given F, we say an edge $e = (u, v)$ is

- Useless if u, v in same component
- Safe if, for some component C_i, e is min-weight edge w/ exactly 1 vertex in C_i

Lemma: MST contains every safe edge.

Pf Suppose otherwise, and $e = (u, v)$
is safe for some F
but not in MST T.

T is spanning $\Rightarrow \exists$ $u \rightsquigarrow v$ path P in T



$u \in C_i$ but $v \notin C_i$ for component C_i of F

$\Rightarrow$ Some edge $e' \in P$ has one vertex
in C_i & one vertex not in C_i.

$\Rightarrow$ $w(e') > w(e)$ [e is minimum-weight leaving C_i]

$\Rightarrow$ $T' = T + e - e'$ has $w(T') < w(T)$
T' is a spanning tree [n-1 edges & connected]
$\Rightarrow$ T is not MST, $\Rightarrow\Leftarrow$ ∎

MST algorithms

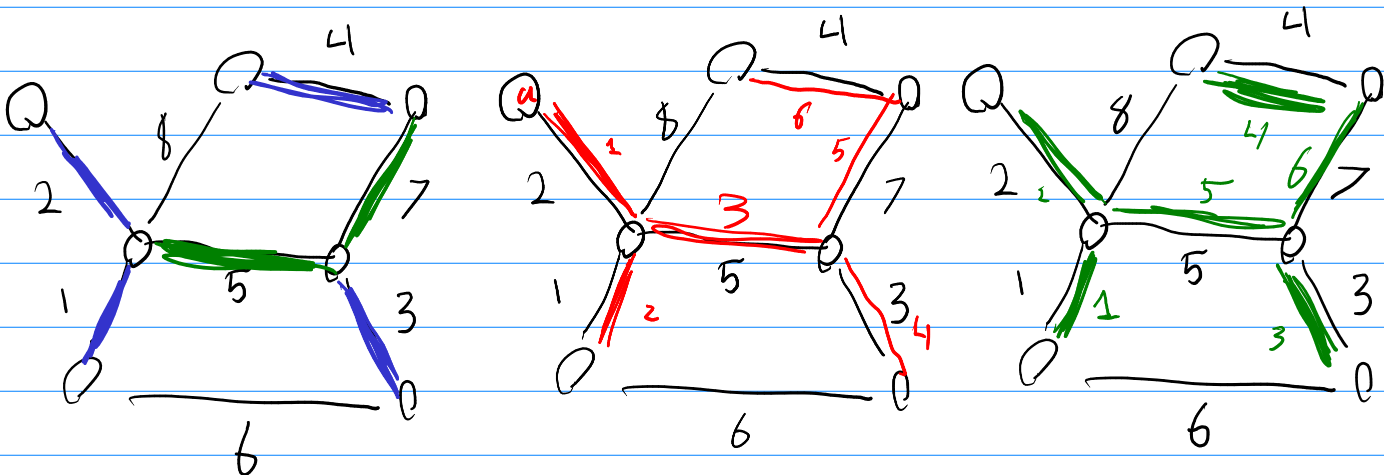
(1) Start with $F = \{\}$
(2) while $|F| < n-1$
How? → (3) Find 1 or more safe edges & add them to F.
(4) return F

> **Safe** if, for some component C_1, e is min-weight edge w/ exactly 1 vertex in C_1

Boruvka: choose all safe edges

Prim: Pick a vertex s, & $C_1 :=$ connected to s in F; always choose min-weight edge from C_1

Kruskal: choose the min-weight edge in entire graph that crosses components



Boruvka: round 1, 2 | Prim: order | Kruskal: order

Q: how fast?

Boruvka: $O(E)$ time per round. [search]

After first round, all component size ≥ 2
$\Rightarrow$ $\leq V/2$ components. # components divides by 2 each round
$\Rightarrow$ $\leq \log_2 V$ rounds.

$\Rightarrow O(E \log V)$ time.

Prim: Naively $O(E)$ per round & $V-1$ rounds $\Rightarrow O(E \cdot V)$.

Better: whatever-first-search w/ a heap containing edges touching current connected component.

```
def Prim():
    s = V[0]          # arbitrary choice of root
    Parent = {}
                          # weight  vertex  parent
    Q = PriorityQueue([(None, s, None)])

    while Q:
        _, u, p = Q.popmin()
        if u in parent: continue
        Parent[u] = p
        for v in u.adj:
            Q.push((w(u→v), v, u))
    return [(u, Parent[u]) for u in parent
            if Parent[u] is not None]
```

Time: E push/pop operations $\Rightarrow O(E \log E) = O(E \log V)$.

[Reformulate: $O(V)$ push/pop + $O(E)$ decrease-key
+ Fibonacci Heap $\Rightarrow O(E + V \log V)$]

Kruskal: Sort $(O(E \log E) = O(E \log V))$
+ scan through list
+ check if each edge already connected in F

Naively: $O(V)$ to search $F \Rightarrow O(E \cdot V)$.
Better: use a data structure.

Method 1: each vertex stores its component Id
& track each component's size.

When edge is added, relabel smaller component
[by BFS/DFS].
When a vertex is relabeled, its component size $\geq$ doubles
$\Rightarrow$ each vertex relabeled $\leq \log V$ times
$\Rightarrow O(V \log V)$ total time maintaining this

$\Rightarrow O(E \log V)$ sort + $O(V \log V)$ after

In general: Union Find data structure

MakeSet(u) $\leftarrow$ create a new set w/ u
Find(u) $\leftarrow$ return unique ID for u's set
Union(u, v) $\leftarrow$ merge u & v's set

V makeSet, $2E$ find, $V-1$ Union.

Kruskal (V, E):

```
for u in V:    MakeSet(u)
sort E by increasing weight
F = {}

for (u,v) in E:
    if Find(u) = Find(v): continue
    F.add((u,v))
    union(u, v)
return F
```

Fancy union-find: $O(E\,\alpha(V)) + O(E \log V)$ sort

α = inverse Ackermann