Minimum Spanning Trees

Given an undirected, weighted graph

\[ G = (V, E) \text{ w/ weights } w(e) \]

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?]: $E + V \log V$

Boruvka’s [1926]: $3E \log V$

Kruskal’s [1956]: $E \log V$

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.

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

$F$ is an “intermediate spanning forest”

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

$F \subseteq E_3 \Rightarrow C_1 = E_3$

$F = T \Rightarrow C_1 = V$

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

- useless if $u, v$ in same component
- safe if, for some component $C_j$, $e$ is min-weight edge w/ exactly 1 vertex in $C_j$
Lemma: MST contains every safe edge.

\[ \text{Prove: Suppose otherwise, and } e = (u, v) \text{ is safe for some } F \text{ but not in MST } T. \]

\[ T \text{ is spanning } \Rightarrow \exists u \sim v \text{ path } P \text{ in } T \]

\[ u \in C_i \text{ but } v \notin C_i \text{ for component } C_i \text{ of } F \]

\[ \Rightarrow \text{ some edge } e' \in P \text{ has one vertex in } C_i \text{ and one vertex not in } C_i. \]

\[ \Rightarrow w(e') > w(e) \] [e is minimum weight leading C_i]

\[ \Rightarrow T' = T + e - e' \text{ has } w(T') < w(T) \]

\[ T' \text{ is a spanning tree [n-1 edges & connected]} \]

\[ \Rightarrow T \text{ is not MST}, \Rightarrow \]
MST algorithms

1. Start with \( F = \emptyset \)
2. While \( |F| < n - 1 \)
   - Find 1 or more safe edges
   - Add them to \( F \)
3. Return \( F \)

Safe if, for some component \( C_i \), e is min-weight edge w/ exactly 1 vertex in \( C_i \)

Boruvka: choose all safe edges

Prim: Pick a vertex \( s \), a \( C_i \) connected to \( s \) in \( F \)
   - Always choose min-weight edge from \( C_i \)

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

Boruvka: found 1/2
Prim: order
Kruskal: order
Q: how fast?

Baruva: \( O(V) \) time per round. [search]

After first round, all components size \( \geq 2 \)

\( \Rightarrow \) \( \leq V/2 \) 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 \& \( V-1 \) round) \( \Rightarrow \) \( O(E \cdot V) \).

Better: whatever-first-search w/ a heap containing
eaves touching

current connected component.

```python
def Prim():
    s = V[0]
    Parent = E[3]
    Q = Priority Queue([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 \( \text{Id} \)
+ track each component's size

when edge is added, relabel smaller component
- \( \text{by BFS/DFS} \).

when a vertex is relabeled, its component size 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

\( \text{make set} \ (u) \ \Leftrightarrow \text{create a new set } \ u \)
\( \text{find} \ (u) \ \Leftrightarrow \text{return unique Id } \text{ for } u \text{'s set} \)
\( \text{union} \ (u, v) \ \Leftrightarrow \text{merge } u \text{'s set} \)

\( V \text{ make set}, \ 2E \text{ find}, \ V-1 \text{ union} \).
Kruskal (V,E):

for u in V: Make Set(u)

sort E by increasing weight

F = \emptyset

for (u,v) in E:

if Find(u) = Find(v): continue

F.add((u,v))

Union(u,v)

return F

Fancy union-find: \(O(|E| \times \alpha(|V|)) + o(|E| \log |V|)\) sort as in work Ackerman