Talk Announcment

- WHO: Ariadna Quattoni, Technical University of Catalonia (and MHC CS alum!)
- WHEN: 12:15pm, Monday, Mar. 4
- WHERE: Kendade 307
- TITLE: Methods for Automatic Image Tagging and Retrieval

Plan for Today: MST

- Proofs of Prim and Kruskal
- Remove distinctness assumption
- Implementation
  - Prim - O(m log n)
  - Kruskal - O(m log n), with Union-Find data structure
- Network design: beyond MST

Prim Implementation

```
T = {}
S = {s}
While S != V {
   Let e = (u, v) be the minimum cost edge from
         S to V-S
   T = T ∪ {e}
   S = S ∪ {v}
}
```

Prim Implementation

```
T = {}, S = {s}

for all edges e = (s, v) incident to s
   a[v] = c_e          // maintain attachment cost for v
   edgeTo[v] = e;  // edge with smallest attachment cost
end

while S != T {
     let v be node that minimizes a[v], and let e = edgeTo[v]
     T = T ∪ {e}
     S = S ∪ {v}
     for all edges e = (v, w) incident to v
          if c_e < a[w] then
               a[w] = c_e
               edgeTo[w] = e
          end
     end
}
```

n x extractMin → n log n

m x changeKey → m log n

Analogous to Dijsktra: O(m log n) using heap-based priority queue

Kruskal Implementation?

```
Sort edges by weight: c1 ≤ c2 ≤ … ≤ cm
T = {}
for e = 1 to m {
   if T ∪ {e} does not contain a cycle {
      T = T ∪ {e}
   }
}
```

Loop executes m times. How much time does it take to check if $T \cup \{e\}$ has a cycle?

Cycles?

- Let $e = (u, v)$. When does $T \cup \{e\}$ have a cycle?
- When there is already a path from u to v
- u and v are in the same connected component in $G' = (V, T)$
- How do we check this?

First Cut

- Let $e = (u, v)$. When does $T \cup \{e\}$ have a cycle?
- Run BFS from u in $G' = (V, T)$ to see if v is currently reachable from u (time: $O(n)$)
- Total time: $O(mn)$
- (We can do better)

Better Approach

Explicitly maintain connected components

Sort edges by cost: $c_1 \leq c_2 \leq \ldots \leq c_m$.

$T \leftarrow \{\}$

for each $u \in V$ make a singleton set $\{u\}$

for each edge $e_i = (u, v)$

    if (u and v are in different sets) {

        $T \leftarrow T \cup \{e_i\}$

        merge the sets containing u and v

    }

Goal: O(log n) for all operations → O(m log n) overall

Union-Find

- Data structure to maintain disjoint sets
- Operations:
  - Find(v) - determine which set a node is in
  - Union(S_1, S_2) - merge two sets
- Useful for Kruskal and other algorithms!

Union-Find: First Try

- Array-based implementation: for each node, store the name of the component it belongs to
- Work through this on board
- Worst-case running time:
  - Find: $O(1)$
  - Union: $O(n)$
- Can be improved so that any sequence of n Union operations takes $O(n \log n)$, but we'll abandon in favor of better apparoch

Pointer-Based Union-Find

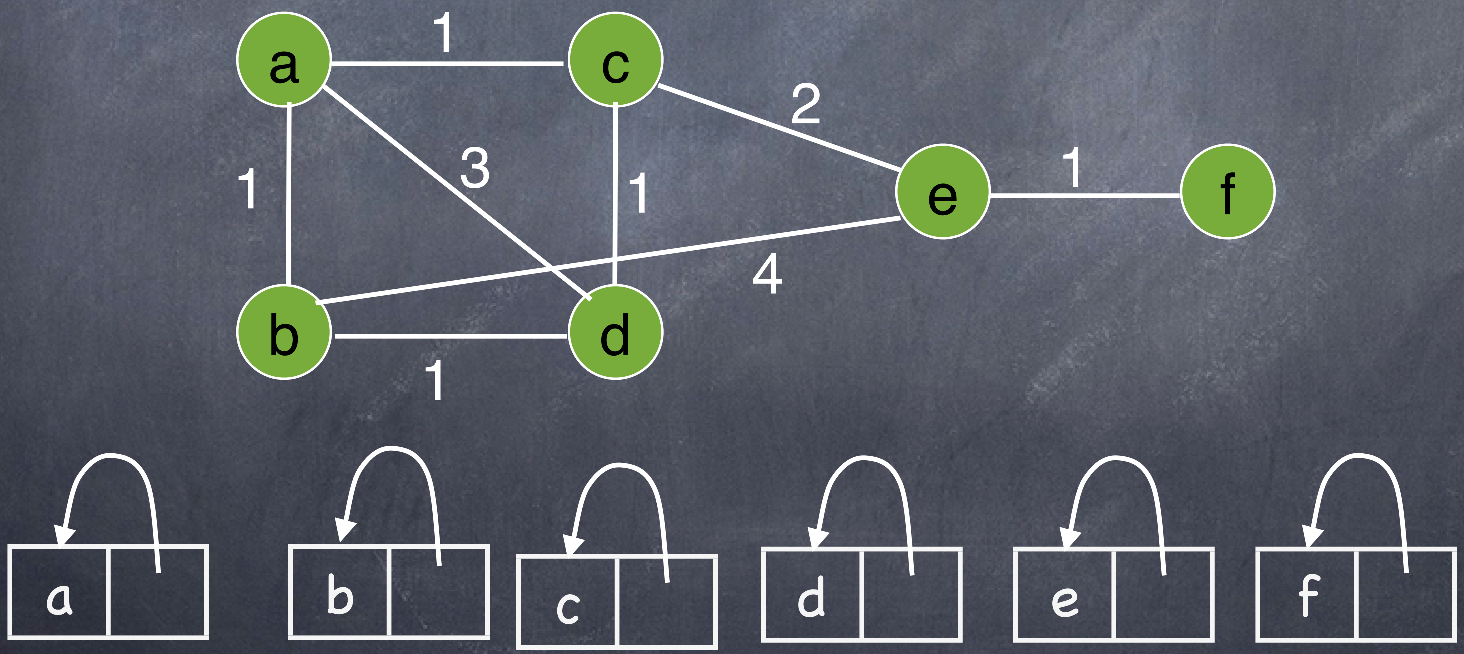
- Idea: elect a node to represent each set, so

  name of set = name of representative node

- Each node maintains a pointer to its representative

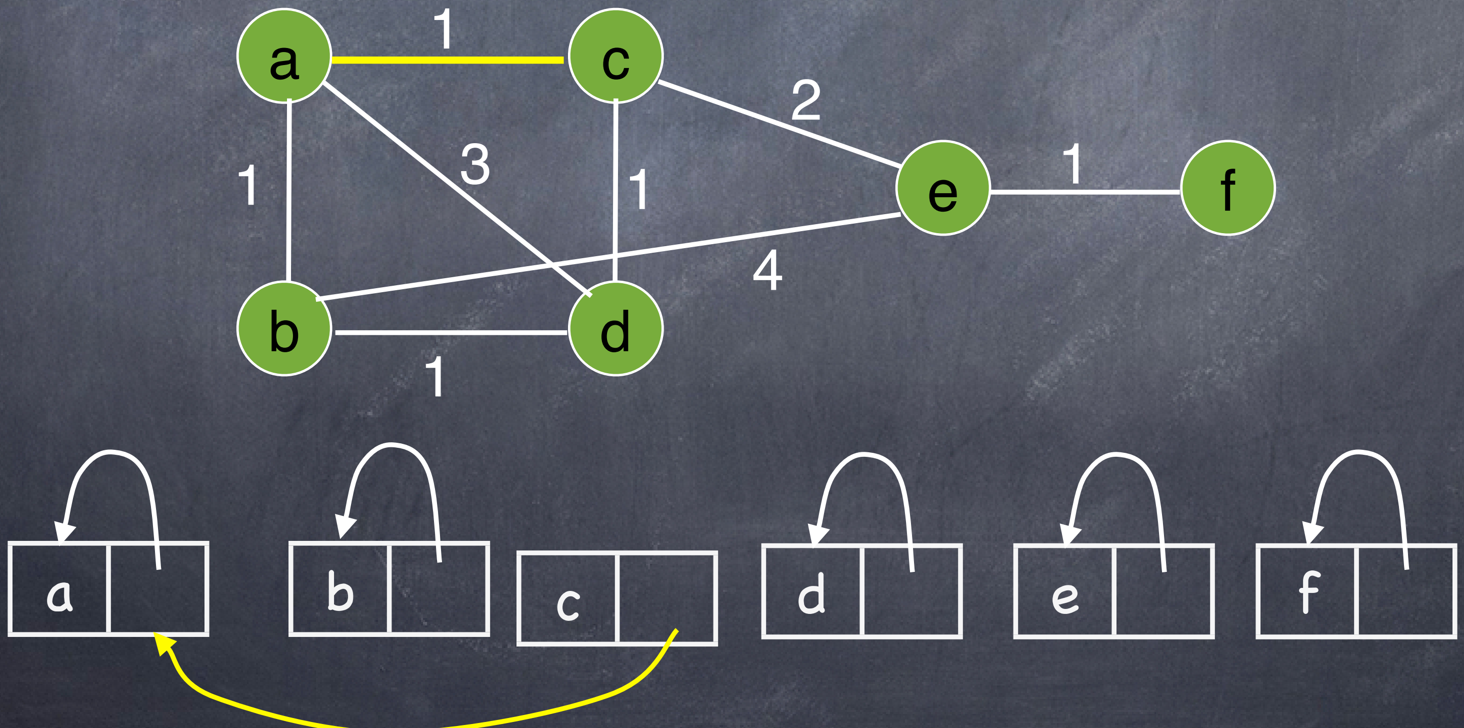
A Faster Union

- Associate with each node a pointer to its name.

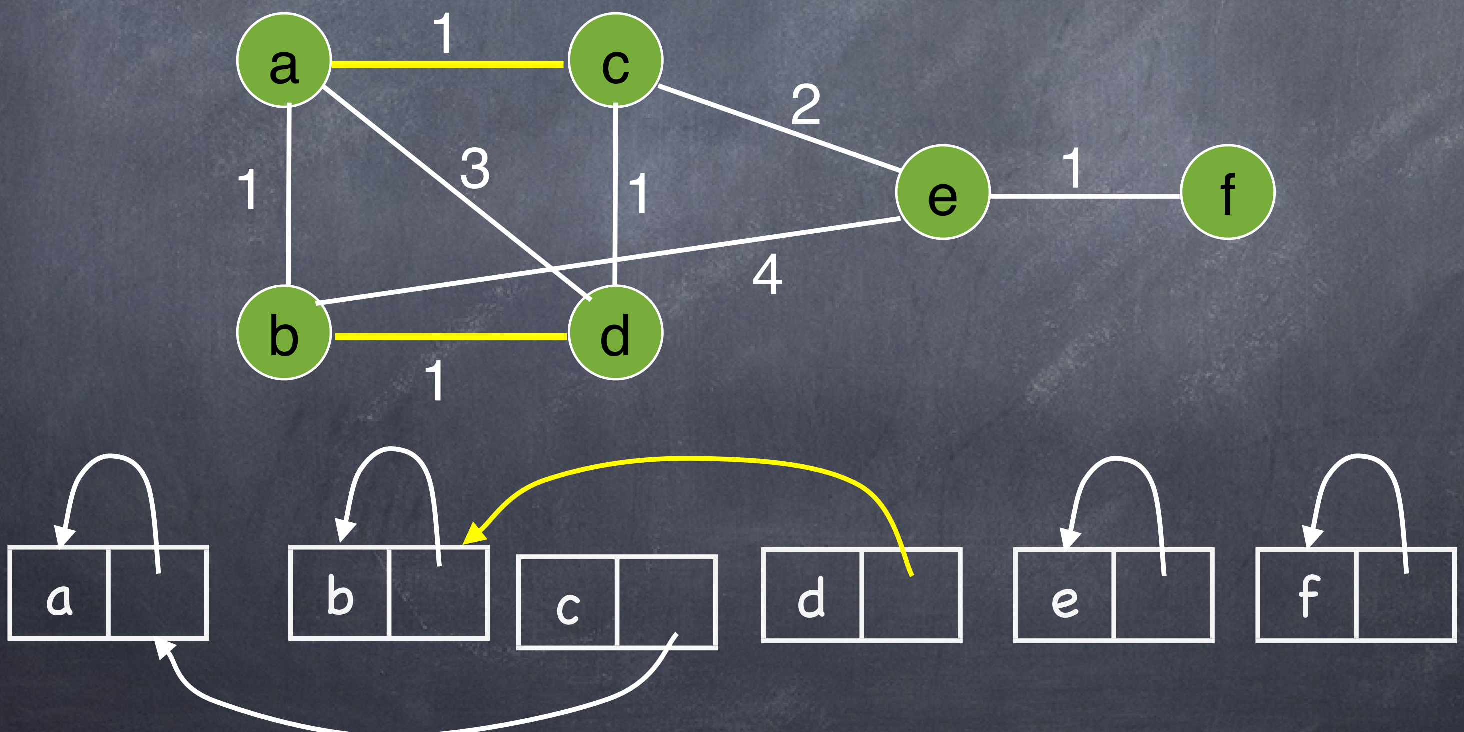


A Faster Union

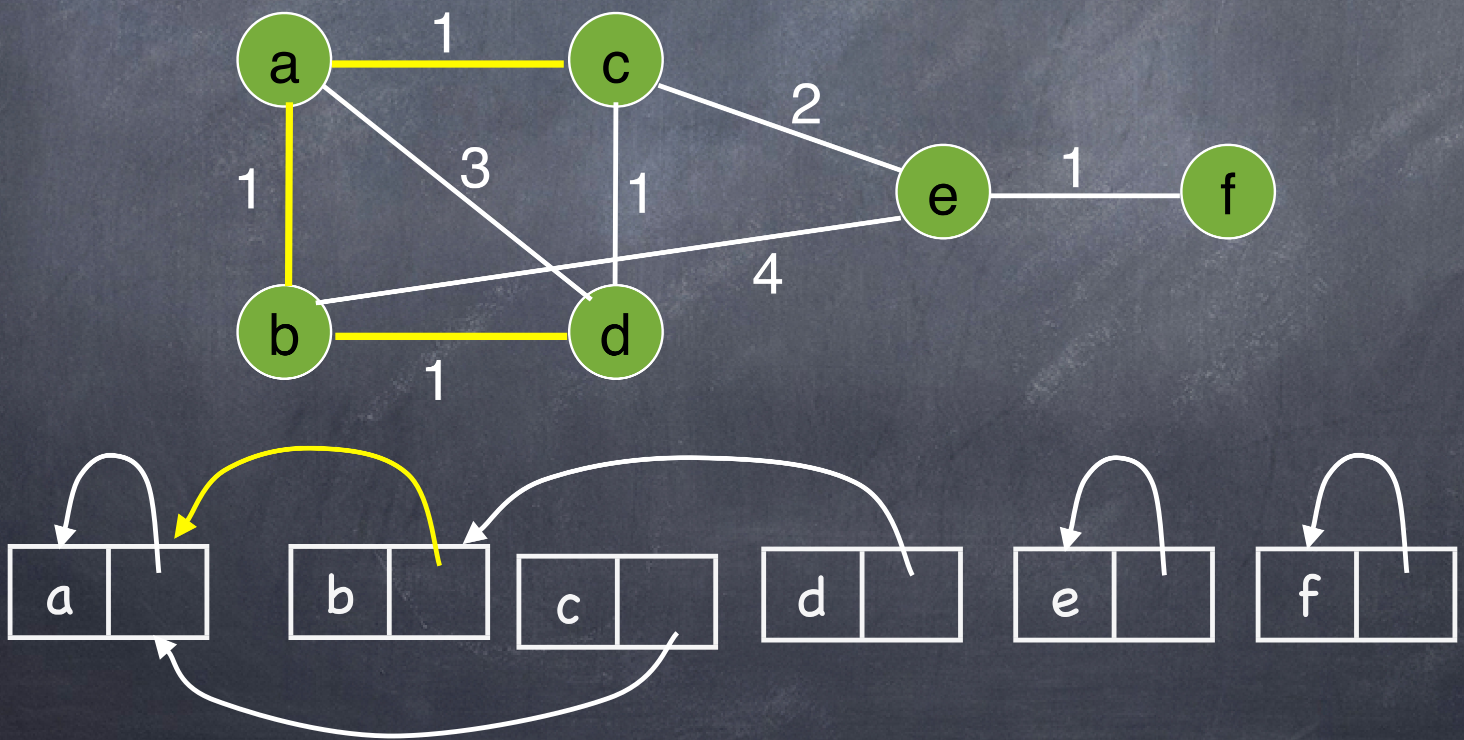
- On Union, update the head pointer of the smaller set.



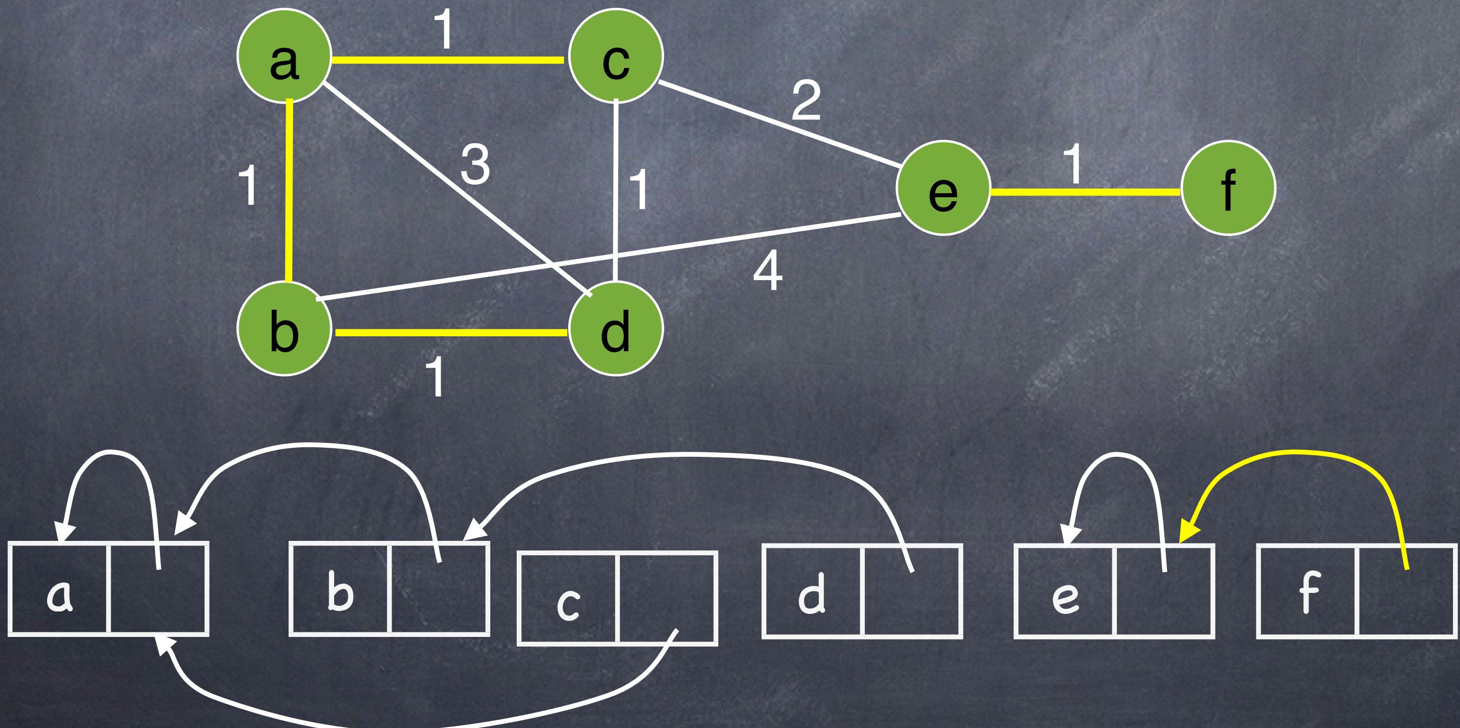
A Faster Union



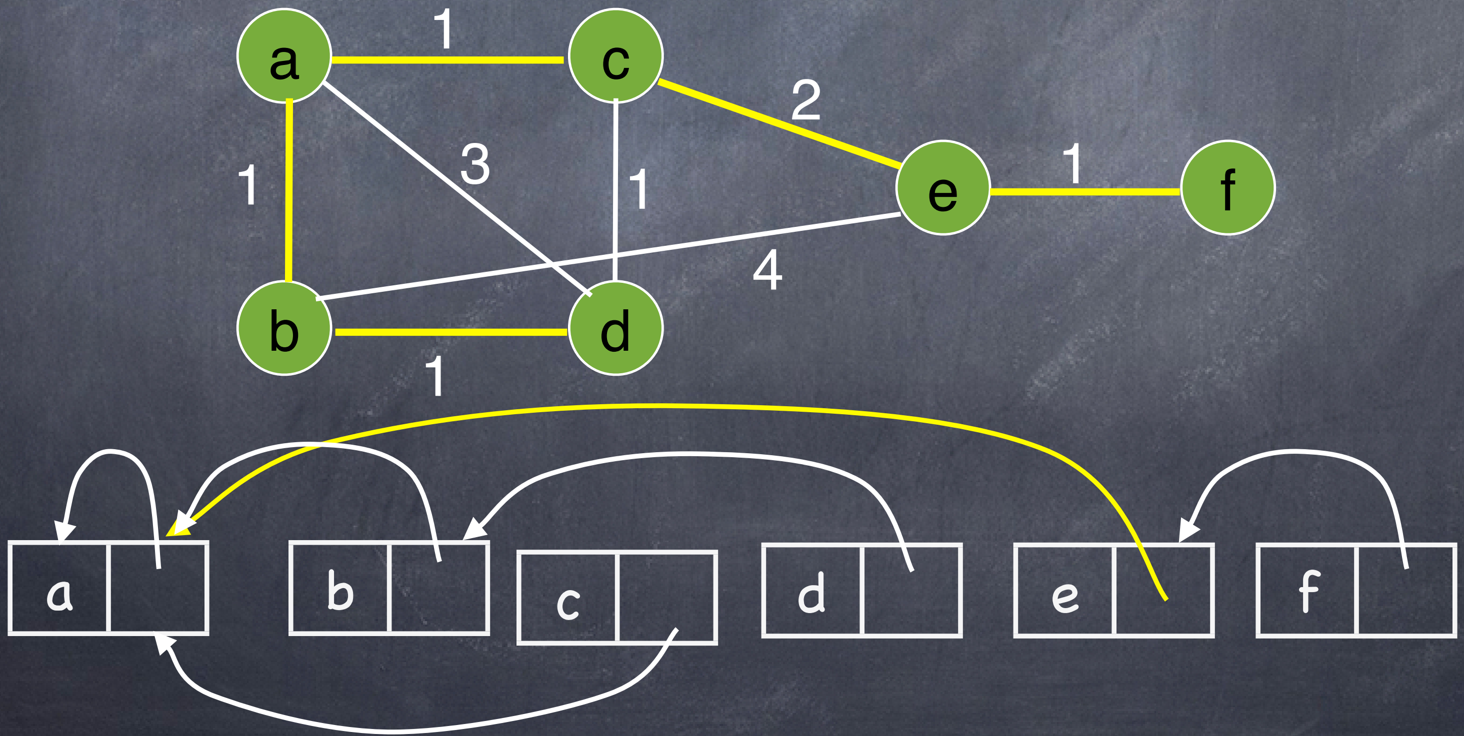
A Faster Union



A Faster Union



A Faster Union



Pointer-Based Union Find

- Union(S_1, S_2) - O(1) (update pointer)
- Find(v) - ??? (follow pointers to representative)

- Claim: if we follow convention of updating the pointer of the smaller set, then Find(v) is O(log n)

- Example and proof on board

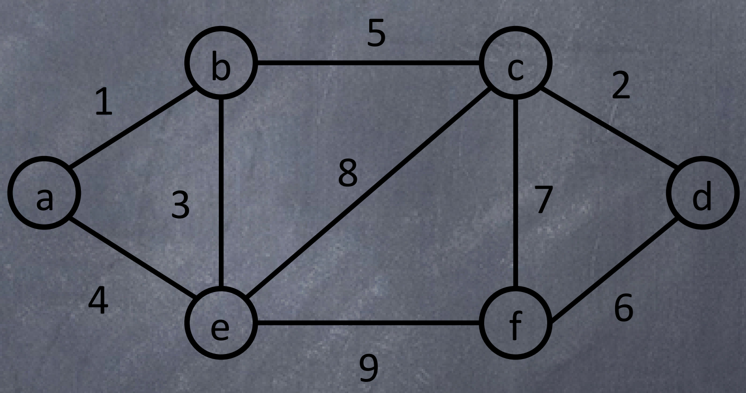
Summary

- Kruskal is O(m log n) with appropriate Union-Find data structure
- Possible to improve Union-Find even more so Kruskal becomes O(m α(n)), where α(n) is inverse Ackerman's function
  - Grows incredibly slowly (essentially constant)

Network Design: Steiner Tree Problem

Given: undirected graph $G = (V, E)$ with edge costs $c_e > 0$ and terminals $X \subseteq V$

Find: edge subset $T \subseteq E$ such that (V, T) has a path between each pair of terminals and the total cost $\sum_{e \in T} c_e$ is as small as possible



Easier? Harder?