

**Note: this set of notes (and those for the next several lectures) are just my own notes that I used in class. We didn't necessarily cover everything on the notes on a particular day, so some material is repeated. Also, all the material is available in the Dasgupta book, chapters 3, 4, and 5, I believe. -jfh**

Last time we ended with a min weight spanning tree algorithm:

Input: a connected, weighted graph  $G = (V, E)$

Output: a spanning tree  $T = (V, E')$  of minimum total weight.

[Observe:  $T$  is a tree,  $E' \subseteq E$ ]

Algorithm:

Sort  $E$  by weight

$E' =$  empty set of edges

foreach edge  $e$  in  $E$  in increasing order of weight

  if adding  $e$  to  $E'$  doesn't create a cycle :

    add  $e$  to  $E'$

  if  $\text{size}(E') = |V| - 1$ : DONE

Slightly improved version

Sort  $E$  by weight

$E' =$  empty set of edges

$C =$  set of components of the graph defined by  $(V, E')$

// initially,  $C$  has one component for each node of  $G$ !

foreach edge  $e$  in  $E$  in increasing order of weight

  if endpoints of  $e$  are in two different components

    add  $e$  to  $E'$

    merge the components at the endpoints of  $e$

  if  $\text{size}(E') = |V| - 1$ : DONE

Problems:

\* how do we tell if two things are in the same component?

\* how do we know that the algorithm works, even if we CAN do the component thing?

Let's pause for a little analysis.

Warmup facts:

Tree: a connected acyclic graph.

Observations on trees:

1. Removing any edge disconnects

2. If  $T$  has  $n$  nodes, it has  $n-1$  edges

3. If  $G = (V, E)$  is connected and undirected, and  $|E| = |V|-1$ , then  $G$  is a tree

4. An undirected graph is a tree iff there's a unique path between any two nodes.

Observation about MWST: removing a cycle edge cannot disconnect a graph.

Our algorithm (Kruskal's, actually) is **greedy**: add next lightest edge that doesn't make a cycle

Let's quickly prove it works.

Recall that if weights were all different, we had an argument that the min weight edge HAS to be in the tree. (if not, add it; that makes a cycle. break the cycle by removing some other edge. You lowered the weight!)

More general idea: if we *partition*  $V$  into  $V_1$  and  $V_2$  ( $V_1 \cup V_2 = V$ ;  $V_1 \cap V_2 = \emptyset$ ), then we induce a partition on edges:  $E_{11}$  = edges with both ends in  $V_1$ ;  $E_{22}$  = Edges with both ends in  $V_2$ .  $E_{12}$ : edges with one end in  $V_1$  and the other in  $V_2$ .

Observation: if  $e_1, e_2, \dots, e_k$  are the min weight edges in  $E_{12}$ , then ANY MWST must contain one of  $e_1, e_2, \dots, e_k$ . (Typical: there's a UNIQUE min weight edge, so you pick that one).

**Application:** when we add an edge, it goes from some component  $C_1$  to some component  $C_2 \neq C_1$ . Let  $V_1 = C_1$ , and  $V_2 = V - C_1$ . Then apply cut principle.

Problem: need to know whether endpoints are in same component.

Solution: Union/Find. Blindingly efficient when many operations are done. Operations are "makeset", "union", and "find".

```
forall u in V
  makeset(u)
```

```
X = {} // the MST
sort edges in E by weight
foreach edge (u,v) in E, in increasing order of weight
  if (find(u) != find(v):
    add edge (u,v) to X
    union(u, v)
```

```
return X
```

```
proc makeset(x)
  parent(x) = x
  rank(x) = 0
```

```
proc find(x)
```

```

while x != parent(x) : x = parent(x)
return x

```

```

proc union(x, y)
  rx = find(x); ry = find(y); // find parent.
  if rx == ry: return
  if rank(rx) > rank(ry):
    parent(ry) = rx;
  else
    parent(rx) = ry
  if rank(rx) == rank(ry) : // only time rank increases!
    rank(ry) = rank(ry) + 1

```

[Improved find: during traversal, re-point each node to the root! Makes amazing efficiency!]

Analysis:

Sort E by weight  $O(|E| \log |E|)$

$E'$  = empty set of edges

$C$  = set of components of the graph defined by  $(V, E')$

// initially,  $C$  has one component for each node of  $G$ !

foreach edge $e$ in $E$ in increasing order of weight	$ E $ times
if endpoints of $e$ are in two different components	execute "find"
add $e$ to $E'$	execute union (at most $ V  - 1$ times)
merge the components at the endpoints of $E$	
if $\text{size}(E') =  V  - 1$ : DONE	

Return to analysis of union-find:

Prop1: for any  $x$ ,  $\text{rank}(x) < \text{rank}(\text{parent}(x))$

Prop 2: Any root node of rank  $k$  has at least  $2^k$  nodes in its tree

Observation: every internal node was once a root, and never loses kids. So any INTERNAL node of rank  $k$  has at least  $2^k$  nodes in its tree.

Prop 3: Consider all rank- $k$  nodes and their descendants. Total is at most  $n$  descendants. There are at least  $2^k$  in each cluster. So at most  $n/2^k$  clusters. Conclusion: there are at most  $n/2^k$  nodes of rank  $k$ .

Application: let  $k = \log n$ . Then there's at most one node of rank  $\log n$ . Hence every tree is of height  $\leq \log n$ . So "find" is  $O(\log n)$  !

So Kruskal is  $O(|V| \log |V| + |E| \log |E|)$  [latter term for sorting costs]

Since  $|E| \leq |V|^2$ ,  $\log |E|$  is  $\leq 2 \log |V|$ . So

$O(|E| \log |V|)$ .

---

Alternative: build up MWST by building a tree on  $k$  nodes ( $k = 0, 1, 2, \dots, |V|$ )

At each stage, add the edge from the current set to the untouched set that has min weight.

Jarnik: 1930

Prim: 1957

Dijkstra: 1959

```
proc prim (G, w, u0)
forall u in V
    cost(u) = infty
    prev(u) = nil
cose(u0) = 0
H = makequeue(V, cost()) // priority queue based on costs of edges
while H not empty
    v = deletemin(H)
    foreach (v, z) in E
        if (cost(z) > w(v, z))
            cost(z) = w(v, z)
            prev(z) = v
```

result: the “prev” array defines the tree.

Looks surprisingly like Dijkstra’s algorithm!

---

Now return to shortest paths:

All pairs shortest path problem. Solve it for intermediate nodes in the set  $1 \dots k$ ; increase  $k$  until done.