Announcements

- HW due Friday
- Next week's homework won't be due until after break (but don't leave it all for after break!!)
Minimum Spanning Tree

Idea: Have a set of nodes, I want to build communications network on them.
Have distances (or costs) for each possible connection.

Goal: Build cheapest network which connects each pair.

Called MST - minimum spanning tree.
What we showed:

Lemma: Let $T$ be a min cost set of edges connecting the vertices. Then $(V, T)$ is a tree.

Cut Prop: Let $S$ be any subset of $V$ and let $e$ be min-cost edge from $S$ to $V-S$. Every MST must contain $e$.

Cycle Prop: Let $C$ be a cycle in $G$ and let $e$ be the most expensive edge on $C$. Then $e$ is not in any MST.
Algorithms:

- Kruskal's: Sort edges, min to max. Add edge if it doesn't create a cycle.
- Backwards Kruskal's: Sort edges, max to min. Delete edge if it doesn't disconnect $G$.
- Prim's: Take vertex & expand via cheapest edge.
Running times:

Prim's: Maintain a min-heap which holds every edge leaving $S$.

How many times will I extractMin? $n-1$ because $\log m = \log n^2 = O(\log n)$.

How many times might I insert into my heap? (ChangeKey) at most once per edge $\sum_{u \in V} d(u) = 2 |E| = O(m)$

$\Rightarrow O(m \log n + n \log n) = O(m \log n)$
Kruskals:

\[
\frac{m \text{ things}}{m \leq n^2} \Rightarrow \log m = \Theta(\log n)
\]

Need to sort edges \( \leq m \) in increasing order:

\[O(m \log m) = O(m \log n)\]

→ Add edge to \( T \) if it doesn’t create a cycle.

From HW, \( O(m+n) \) using BFS/DFS

\[O(m \log n + m(m+n)) = O(m^2)\]
Data Structure: Union-Find

Suppose we want to maintain connected components in a graph as edges are added.

(Want to avoid O(n) search each time!)
Operators:

- **Find (u)**: returns name of connected component that vertex u is in.
- **Make UnionFind (S)**: make UF data structure where each element in S is its own component.
- **Union (u, v)**: merge the component containing u with the component containing v.
Array based implementation. Keep an array entry for each element which stores the set it is in.

- \( \text{Find}(u) \): \( O(1) \) array lookup \( A[i] \)
- \( \text{Union}(u, v) \): \( O(n) \)-like
- \( \text{MakeUF}(S) \): \( O(n) \)

\[ \begin{align*}
1 & \quad 1 \quad 1 \\
\overline{1} & \overline{1} & \overline{1} & \overline{1} & \overline{1} \\
v_1, v_2, v_3, v_4 \text{ & } S & \text{ & } 6 & \ldots & n
\end{align*} \]
Pointer-Based

Each node is initially alone: MakeUF(S)

\[ \text{Find}(v_1) = v_1, \]
\[ \text{Union}(v_4, v_6) : \text{repoint at } v_6, \]
\[ \text{Union}(v_3, v_2) \]
\[ \text{Find}(v_4) = v_6, \]
\[ \text{Union}(v_6, v_5) \]
\[ \text{Find}(v_4) = v_5 \]
When doing union $(u, v)$ first find $(u)$ and find $(v)$.

$\text{Union}(V_3, V_5)$

$\text{Union}((\text{Find}(V_3), \text{Find}(V_5)))$

Can I add edge $V_3 - V_5$?

$\text{Find}(V_3) = \text{Find}(V_5)$?

Runtime: $O(n)$ chains of pointers $\Rightarrow O(n)$ Find
To improve:

When $\text{Union}(v_i, v_j)$ is called:

- $\text{Find}(v_i)$
- $\text{Find}(v_j)$

Then point smaller set to larger set.

(so store size)
Runtime:
Time to find(v) is now the number of times that the leader of v's component changes.

Every time leader changes, the set must have at least doubled in size, since v was in the smaller part of the union.

If there are n elements in v's set, how many times could it have doubled? $O(\log n)$
So:

- \( \text{Find}(u) : O(\log n) \)
- \( \text{Union}(u,v) : O(\log n) \) \( \rightarrow \) does 2 finds
- \( \text{MakeUF}(S) : O(n) \)

\[ \Rightarrow \text{Kruskal's:} \]

- Sort edges \( O(m \log n) \)
- Create EDF \( O(n) \)
  - For each edge:
    - \( \text{Union} \) \( O(\log n) \)
    - \( \text{Find} \) \( O(\log n) \)
  - \( O(m \log n) + O(m \log n) \)
An improvement (not necessary for Kruskal’s, but shhhhhhhhh...):

What is worst case for find?
(i.e., when does it happen?)

\[ \log n \]

Find (v) \( \log n \)

Find (v) \( \log n \)
Path-compression: repoint every one at representative

\[ O(\log n) \]

Find(v)
By shortening path from \( v \) to \( x \), we make latter find calls quicker.

We won't do detailed analysis, but \( \mathcal{O}(n \alpha(n)) \) time, \( \Rightarrow \) almost \( \mathcal{O}(1) \) per operation.

\[ \alpha(n) = \text{inverse Ackermann function} \]

(\text{SMALL})

essentially, this is \( \mathcal{O}(n) \) time