CS314 - Union-Find

Note Title 2/22/2010

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

<u>Idea</u>: • Have a set of nodes + want to build communications network on them.

• Have distances (or costs) for each possible connection

<u>Goal</u>: 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? $n-1$

How many times might I insert into my heap? $\log m \le \log n^2 = O(\log n)$

(Changekey) at most once per edge

$$\underbrace{\sum_{v \in V} d(v) = 2|E|}_{O(m)}$$

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

$n \le m$

m things

$m \le n^2$

$\log m \le \log n^2 = 2\log n$

$\Rightarrow \log m = O(\log n)$

Kruskals:

Need to sort edges & go 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

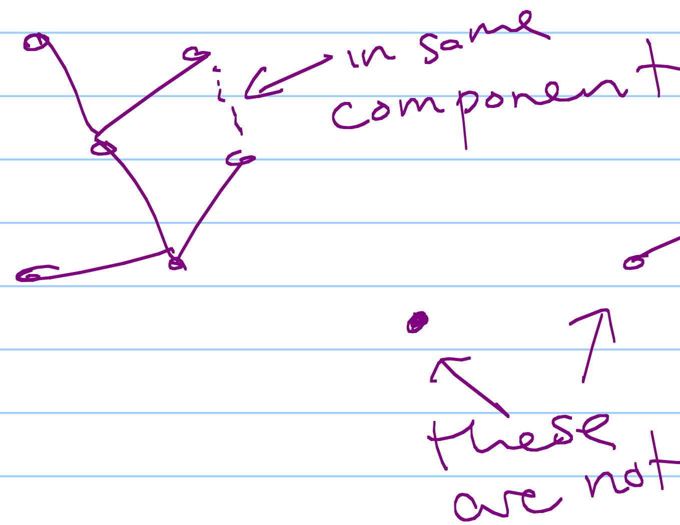
$$\Rightarrow O(m \log n + m(m+n)) = O(m^2)$$

$\boxed{\text{ick}}$

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!)

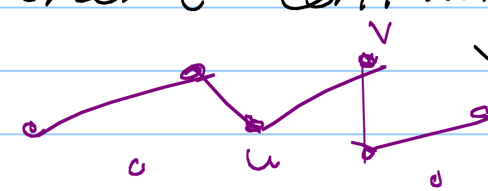


Operations :

Find(u) : returns name of connected component that vertex u is in

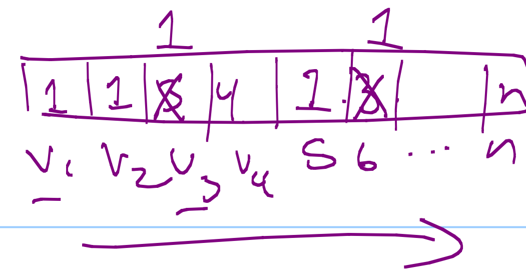
→ MakeUnionFind(S) : make U-F datastructure where each element in S is its own component

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.

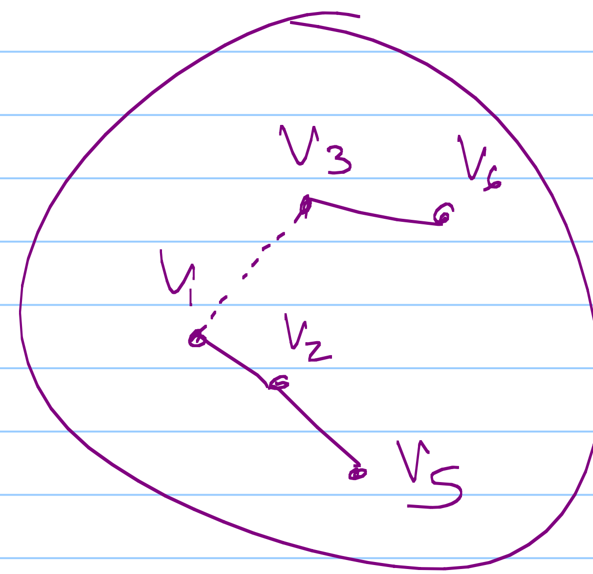


Find(u): $O(1)$ array lookup $A[5]$

Union(u,v): $O(n)$ = idk

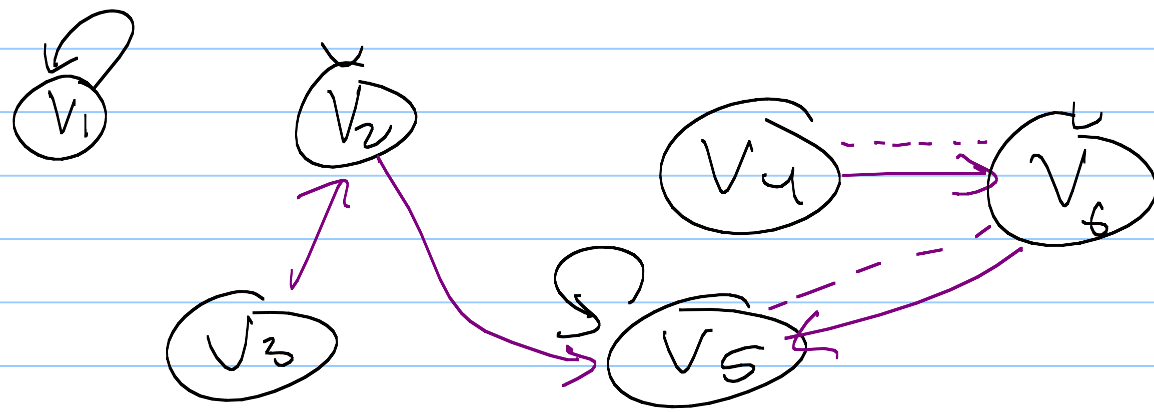
MakeUF(S): $O(n)$

Union(v_1, v_2)
Union(v_2, v_5)
Union(v_3, v_6)
Union(v_1, v_3)



Pointer Based

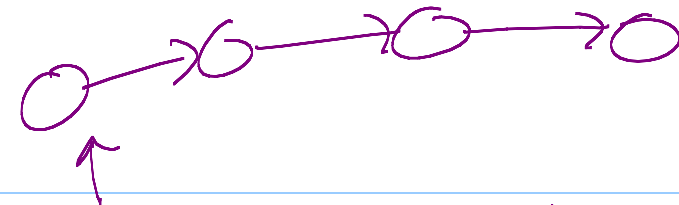
Each node is initially alone: MakeUF(S)

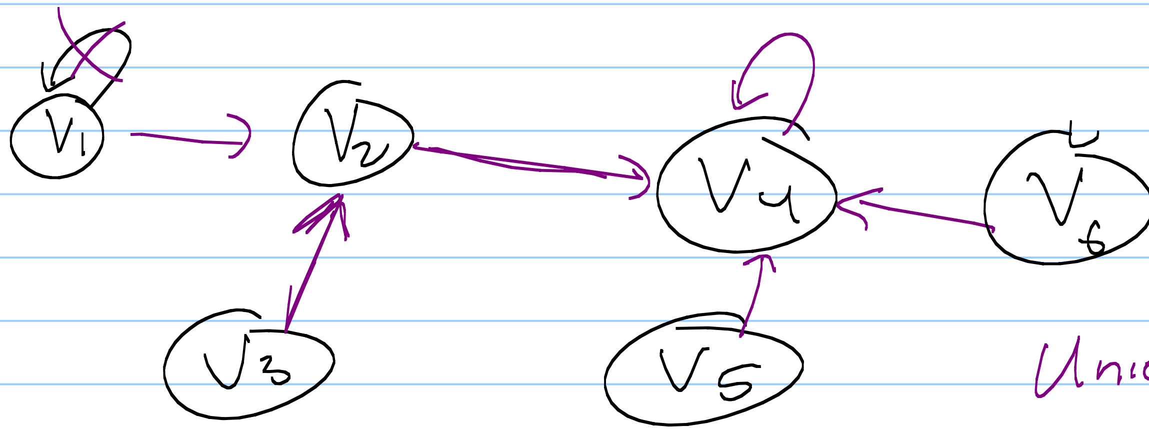


Find(v_1) = v_1

Union(v_4, v_6) : repoint at v_6
Union(v_3, v_2)
Find(v_4) = v_6

Union(v_6, v_5)
Find(v_4) = v_5





When doing union(u, v), first find(u) & find(v).

Union(v_3, v_5) ← my code

Union(Find(v_3), Find(v_5)) ← intended

Can I add edge $v_3 - v_5$?
Find(v_3) = Find(v_5)?

Runtime: O(n) chains of pointers ⇒ O(n) Find

To improve:

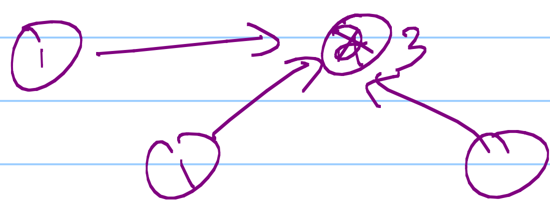
When Union(v_i, v_j) is called:

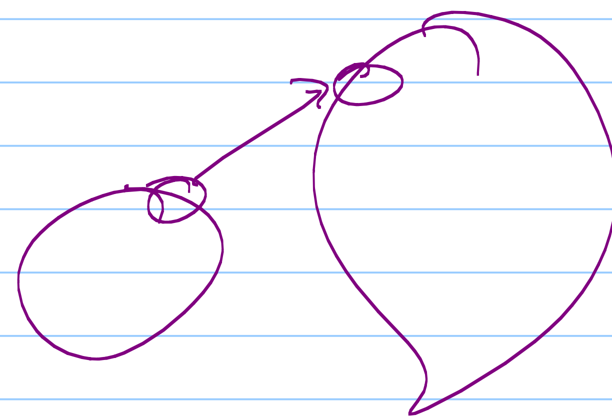
Find(v_i)

Find(v_j)

Then point smaller set to larger set.

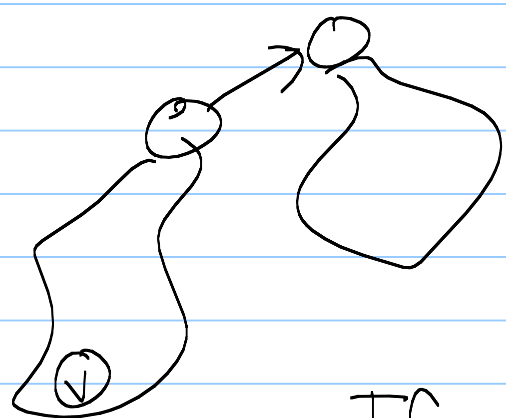
(so store size)





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:

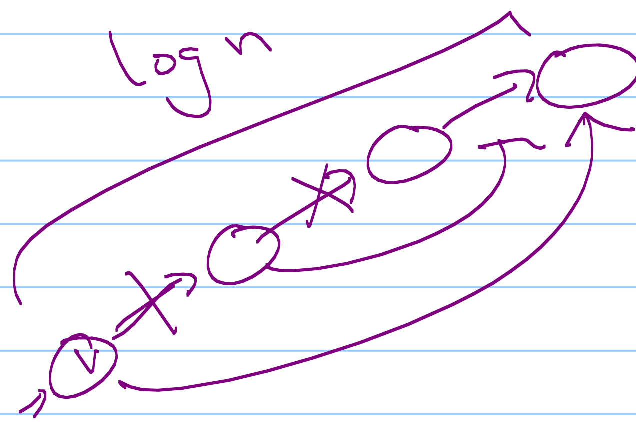
- Find(u) : $O(\log n)$
- Union(u,v) : $O(\log n)$ → does 2 finds
- MakeUF(S) : $O(n)$

⇒ Kruskal's:
- Sort edges $O(m \log n)$
- Create UF $O(n)$
- For each edge:
  - Test if new edge will create a cycle.
    - 2 Finds : $O(\log n)$
    - Union $O(\log n)$

$O(m \log n) + O(m \log n)$

An improvement (not necessary for Kruskal's, but still nice):

What is worst case for find? (ie when does it happen?)



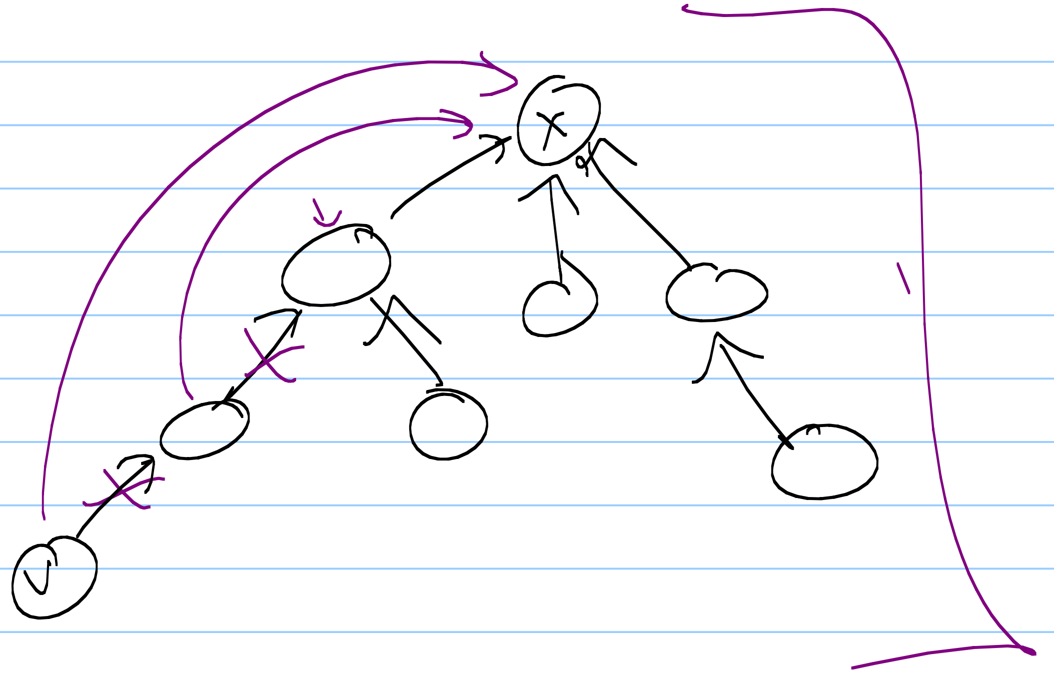
Find(v) log n

⋮

Find(v) log n

Path-Compression:

repoint everyone on path from v to root at representative



$O(\log n)$

Find(v)

By shortening path from v to x, we make later Find calls quicker.

We won't do detailed analysis, but n Find/Union operations gives $O(n\,\alpha(n))$ time. $\Rightarrow$ almost $O(1)$ per operation

$\alpha(n)$ = inverse Ackermann function (SMALL)

essentially, this is $O(n)$ time