Lecture 20: Array Disjoint Sets, Prim's, DFS

CSE 373: Data Structures And Algorithms

Example

Using the union-by-rank and path-compression optimized implementations of disjoint-sets draw the resulting forest caused by these calls:

If tied, make alphabetically earlier node the root.

1. makeSet(a)
2. makeSet(b)
3. makeSet(c)
4. makeSet(d)
5. makeSet(e)
6. makeSet(f)
7. makeSet(g)
8. makeSet(h)
9. union(c, e)
10. union(d, e)
11. union(a, c)
12. union(g, h)
13. union(b, f)
14. union(g, f)
15. union(h, a)

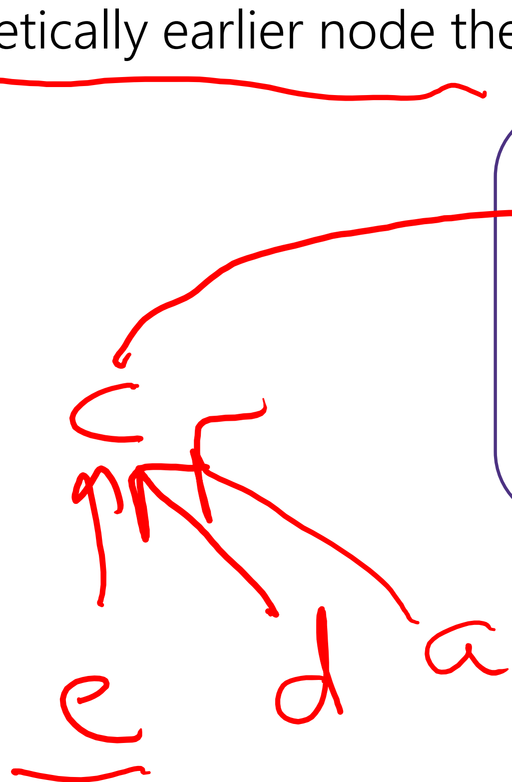
Pollev.com/cse373su19

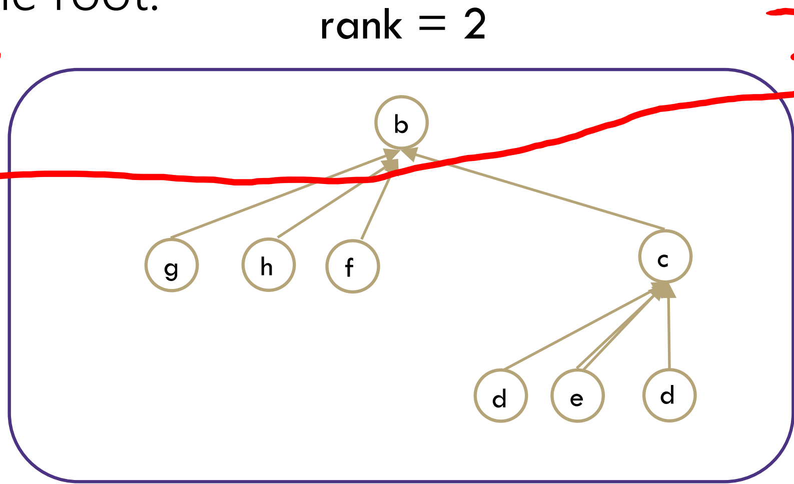
Example

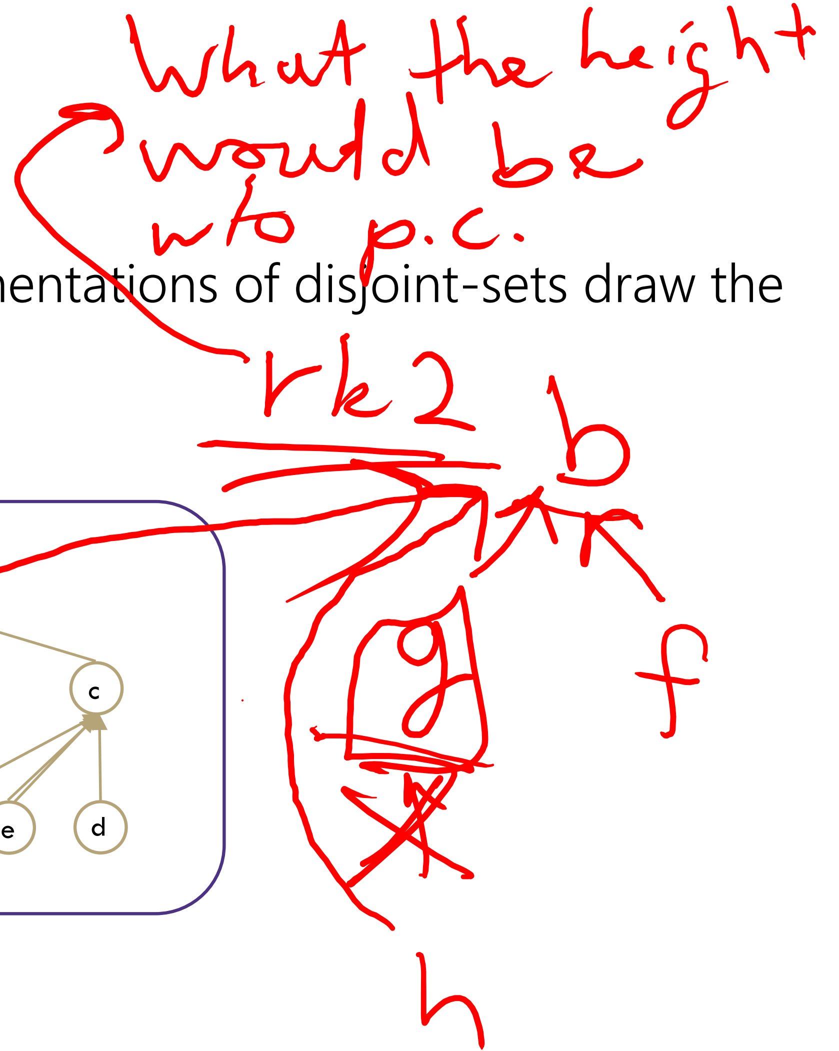
Using the union-by-rank and path-compression optimized implementations of disjoint-sets draw the resulting forest caused by these calls:

If tied, make alphabetically earlier node the root.

1. makeSet(a)
2. makeSet(b)
3. makeSet(c)
4. makeSet(d)
5. makeSet(e)
6. makeSet(f)
7. makeSet(g)
8. makeSet(h)
9. union(c, e)
10. union(d, e)
11. union(a, c)
12. union(g, h)
13. union(b, f)
14. union(g, f)
15. union(h, a)







An optimization

We can even get rid of all the pointers!

- With a similar trick to how we stored heaps.

What information do we need?

- If you're not the root, who is your parent?
- If you are the root, what is the rank of your tree?

Assign every value an index between 0 and $n - 1$.

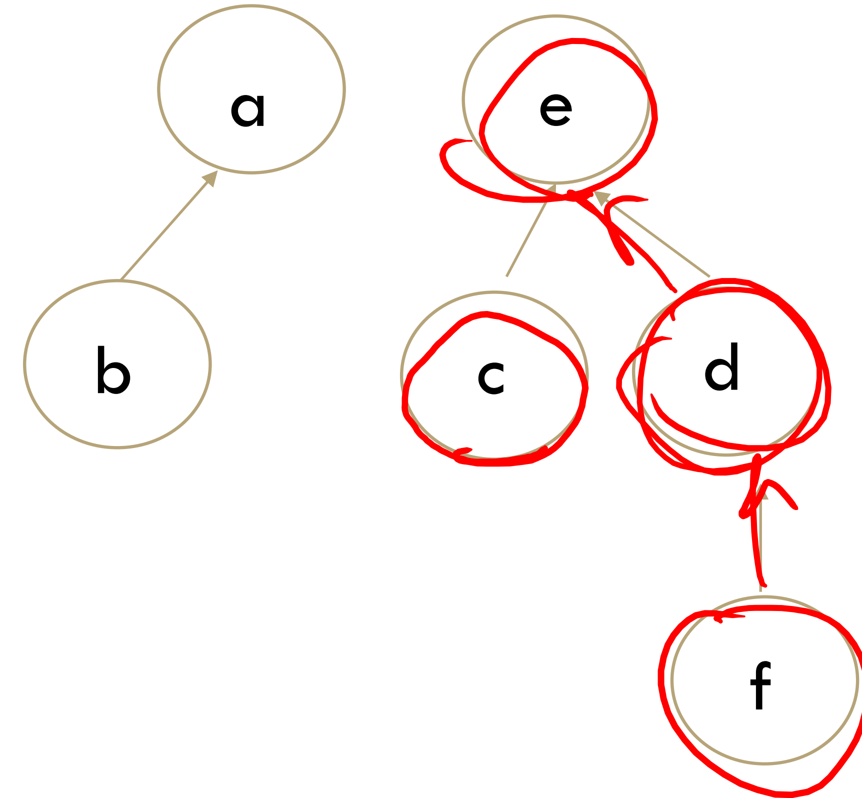
The dictionary should map our set elements to their index.

In the array, store the **index** of your parent.

Make the representative the index of the root (rather than its value).

Array disjoint sets

	a	e	d	c	b	f
index	0	1	2	3	4	5
value	-	-	1	1	0	2

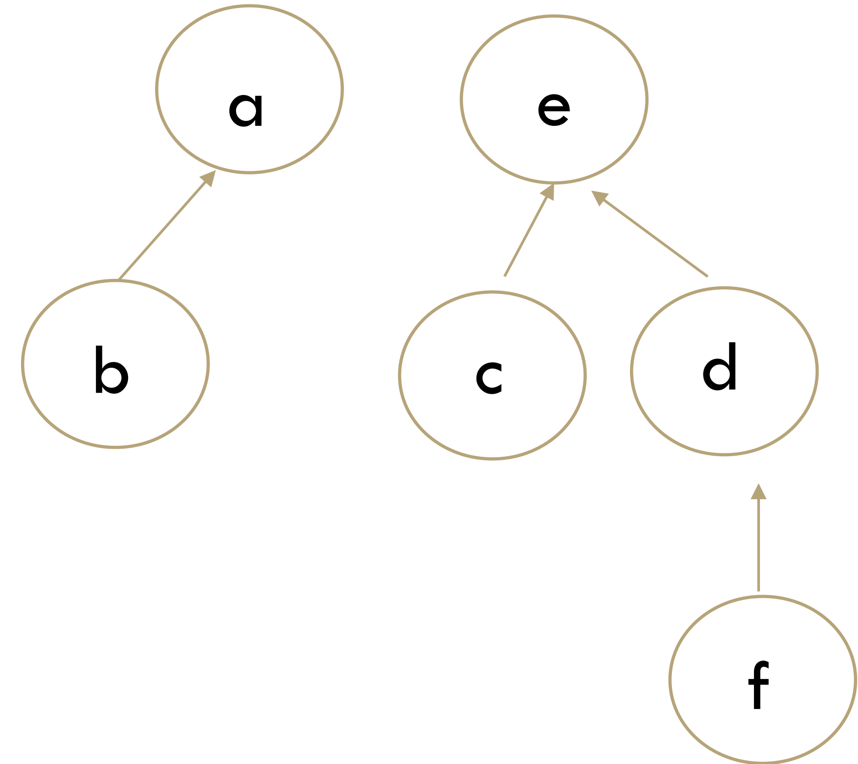


Array disjoint sets

What about the ranks?
We'd like to store them in the roots

How do we tell the difference from rank 2 and index 2?

	a	e	d	c	b	f
index	0	1	2	3	4	5
value	1	2	1	1	0	2

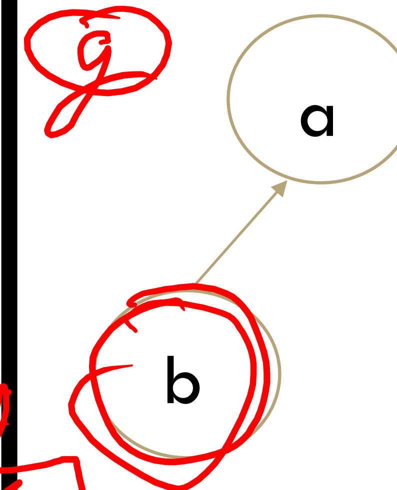


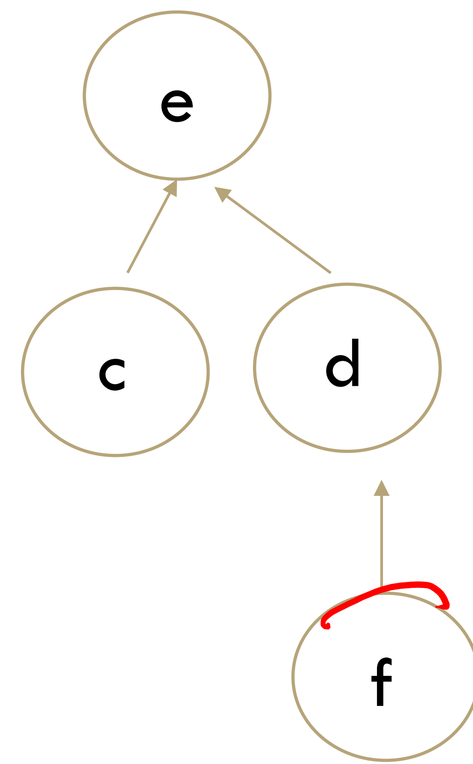
Array disjoint sets

What about the ranks?
We'd like to store them in the roots

Store a negative number instead!
Specifically –(rank) - 1

	a	e	d	c	b	f
index	0	1	2	3	4	5
value	-2	-3	1	1	0	2





Now we can always tell the difference between an index and a rank!
This is the version you're implementing in Project 4.

Kruskal’s Algorithm

```
KruskalMST(Graph G)
    initialize each vertex to be its own component
    sort the edges by weight
    foreach(edge (u, v) in sorted order){
        if(u and v are in different components){
            add (u,v) to the MST
            Update u and v to be in the same component
        }
    }
```

What's the running time of Kruskal's?

For MST algorithms, assume that m dominates n
(if it doesn't, there is no spanning tree to find)

```
KruskalMST(Graph G)
    initialize new DisjointSets DS
    for(v : G.vertices) { DS.makeSet(v) }
    sort the edges by weight
    foreach(edge (u, v) in sorted order){
        if(DS.findSet(u) != DS.findSet(v)){
            add (u,v) to the MST
            DS.union(u,v)
        }
    }
```

What's the running time of Kruskal's?

For MST algorithms, assume that m dominates n
(if it doesn't, there is no spanning tree to find)

```
KruskalMST(Graph G)
    initialize new DisjointSets DS
    for(v : G.vertices) { DS.makeSet(v) }
    sort the edges by weight
    foreach(edge (u, v) in sorted order){
        if(DS.findSet(u) != DS.findSet(v)){
            add (u,v) to the MST
            DS.union(u,v)
        }
}
```

$\Theta(1)$

$\Theta(n)$

$\Theta(m \log m)$

m calls, do we have to worry about the $\log n$ worst case?

n calls, do we have to worry about the $\log n$ worst case?

Intuition: We could make the $\log n$ running time happen once…but not much more than that.
Since we're counting total operations, we're actually going to see the "in-practice" behavior

Whether we hit worst-case or not: $\Theta(m \log m)$ is dominating term.

Running Time Notes

Intuition: We could make the bad case happen once…but not really more than that.

Since we're counting total operations, we're actually going to see "in-practice" behavior

This kind of statement is "amortized analysis"

- It's also the math behind why we always double the size of array-based data structures.

Some people write the running time as $\Theta(m \log n)$ instead of $\Theta(m \log m)$

They're assuming the graph doesn't have any **multi-edges.**

- I.e. there's at most one edge between any pair of vertices.

And they just think $\Theta(m \log n)$ looks better (even though it's just a constant factor)

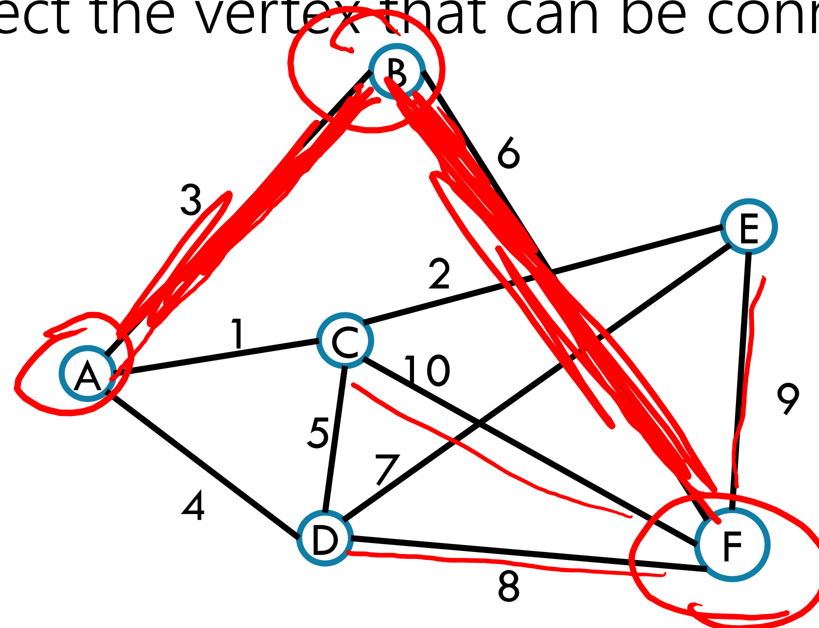
Another MST algorithm

Last Wednesday we said there were two ways to find an MST.

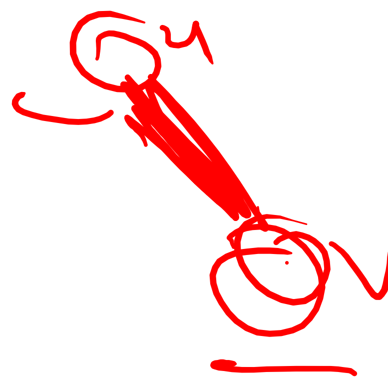
Let's talk about the other way.

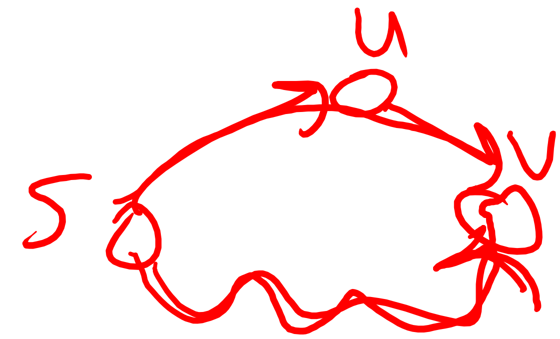
Think vertex-by-vertex

- Maintain a tree over a set of vertices
- Have each vertex remember the cheapest edge that could connect it to that set.
- At every step, connect the vertex that can be connected the cheapest.



Code

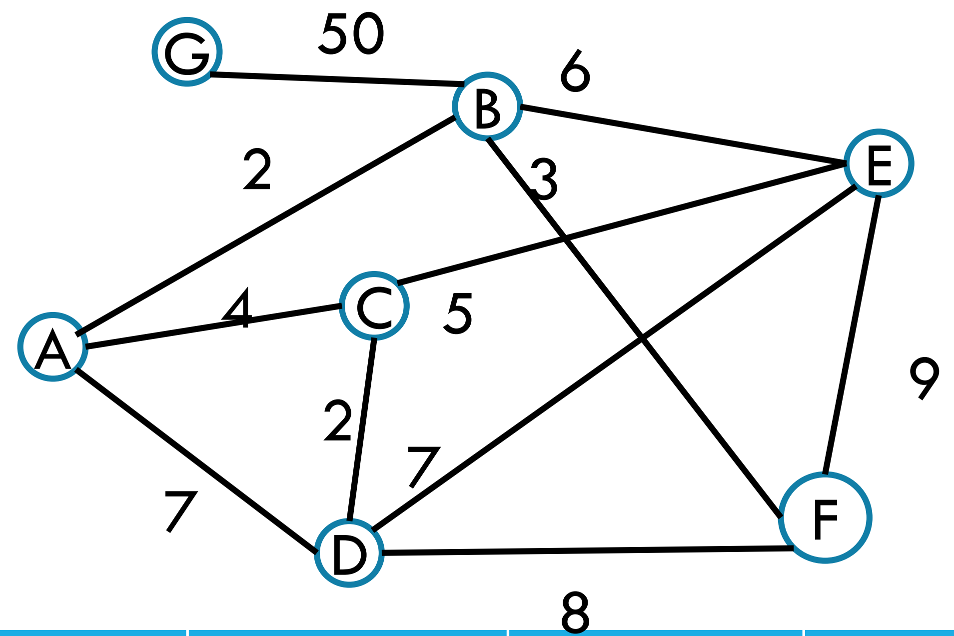





```
PrimMST(Graph G)
        initialize costToAdd to ∞
        mark source as costToAdd 0
        mark all vertices unprocessed, mark source as processed
        foreach(edge (source, v) ) {
                v.costToAdd = weight(source,v)
                v.bestEdge = (source,v)
        }
        while(there are unprocessed vertices){
                let u be the cheapest to add unprocessed vertex
                add u.bestEdge to spanning tree
                foreach(edge (u,v) leaving u){
                        if(weight(u,v) < v.costToAdd AND v not processed){
                                v.costToAdd = weight(u,v)
                                v.bestEdge = (u,v)
                        }
                }
        mark u as processed
        }
```

Try it Out

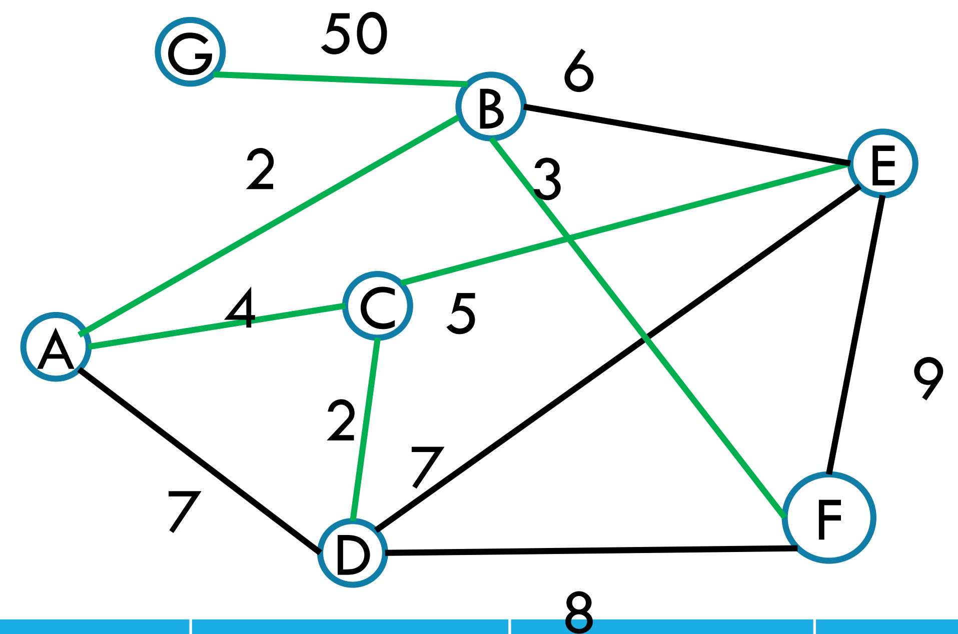
```
PrimMST(Graph G)
   initialize costToAdd to ∞
   mark source as costToAdd 0
   mark all vertices unprocessed
   mark source as processed
   foreach(edge (source, v) ) {
         v.costToAdd = weight(source,v)
         v.bestEdge = (source,v)
   }
   while(there are unprocessed vertices) {
            let u be the cheapest unprocessed vertex
            add u.bestEdge to spanning tree
            foreach(edge (u,v) leaving u){
               if(weight(u,v) < v.costToAdd
                  AND v not processed){
                     v.costToAdd = weight(u,v)
                     v.bestEdge = (u,v)
                 }
            }
            mark u as processed
         }
```



Vertex	costToAdd	Best Edge	Processed
A			
B			
C			
D			
E			
F			
G			

Try it Out

```
PrimMST(Graph G)
   initialize costToAdd to ∞
   mark source as costToAdd 0
   mark all vertices unprocessed
   mark source as processed
   foreach(edge (source, v) ) {
         v.costToAdd = weight(source,v)
         v.bestEdge = (source,v)
   }
   while(there are unprocessed vertices) {
            let u be the cheapest unprocessed vertex
            add u.bestEdge to spanning tree
            foreach(edge (u,v) leaving u){
               if(weight(u,v) < v.costToAdd
                  AND v not processed){
                     v.costToAdd = weight(u,v)
                     v.bestEdge = (u,v)
                 }
            }
            mark u as processed
         }
```



Vertex	costToAdd	Best Edge	Processed
A	--	--	Yes
B	2	(A,B)	Yes
C	4	(A,C)	Yes
D	~~7~~ 2	~~(A,D)~~(C,D)	Yes
E	~~6~~ 5	~~(B,E)~~(C,E)	Yes
F	3	(B,F)	Yes
G	50	(B,G)	Yes

Does This Algorithm Always Work?

Prim's and Kruskals' Algorithms are **greedy** algorithms. Once we decide to include an edge in the MST we never reconsider the decision.

Greedy algorithms rarely work.

There are special properties of MSTs that allow greedy algorithms to find them.

MSTs are so *magical* that there's more than one greedy algorithm that works.

The takeaway from MST lectures is NOT "greedy algorithms are the best"

The takeaway is "MSTs are the coolest.

Some Extra Comments

Prim was the employee at Bell Labs in the 1950's

The mathematician in the 1920's was Boruvka

- He had a different *also greedy* algorithm for MSTs.
- Boruvka's algorithm is trickier to implement, but is useful in some cases.
- In particular it's the basis for fast **parallel** MST algorithms.

"Prim's Algorithm" is a very bad name.

- It was discovered by Jarnik 20 years before Prim.
- And rediscovered by Kruskal before Prim.

Some Extra Comments

There's at least a fourth greedy algorithm for MSTs…

- Called "Reverse-delete"
- Starting from heaviest edge, delete if would not disconnect graph.

If all the edge weights are distinct, then the MST is unique.

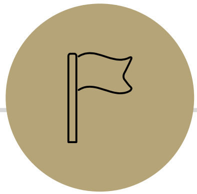
If some edge weights are equal, there may be multiple spanning trees. Prim's/Kruskal's are only guaranteed to find you one of them.

Aside: A Graph of Trees

A tree is an undirected, connected, and acyclic graph.

A forest is any undirected and acyclic graph.

EVERY TREE IS A FOREST.



Optional content: MST correctness

Why do all of these MST Algorithms Work?

MSTs satisfy two very useful properties:

Cycle Property: The heaviest edge along a cycle is NEVER part of an MST.

Cut Property: Split the vertices of the graph any way you want into two sets A and B. The lightest edge with one endpoint in A and the other in B is ALWAYS part of an MST.

Whenever you add an edge to a tree you create exactly one cycle, you can then remove any edge from that cycle and get another tree out.

This observation, combined with the cycle and cut properties form the basis of all of the greedy algorithms for MSTs.