Minimum Spanning Trees

Data Structures and Algorithms

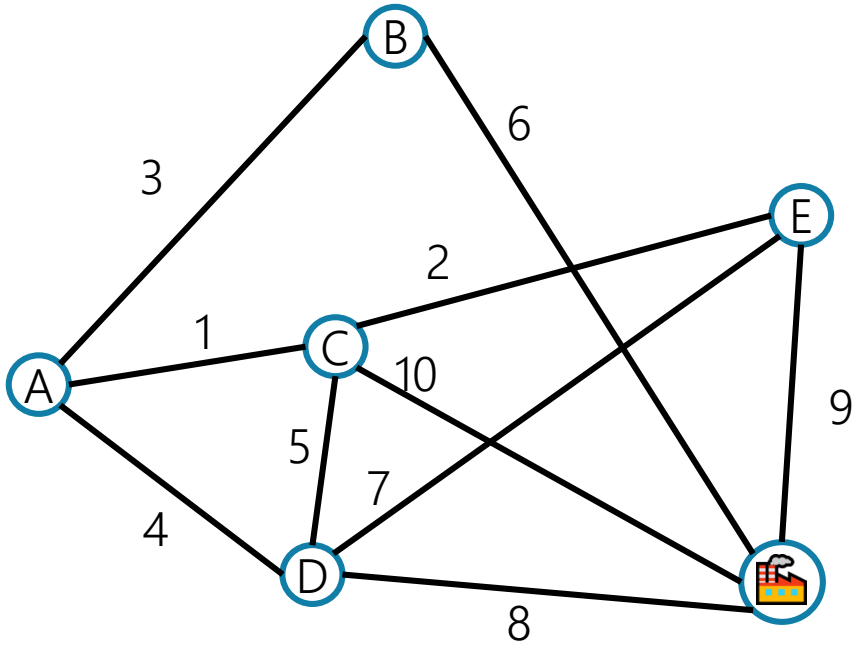
Announcements

Talk on technical interviews
- This Friday 1:10 PM
- Gugenheim 220

On “optimization experiments” section of P3 writeup, if you find the experiments are taking WAY too long, decrease the depth (and tell us you’ve done that)
- I don’t think depth 5 will be a problem for most laptops/the lab machines, but we’d rather you get some numbers than none.

Minimum Spanning Trees

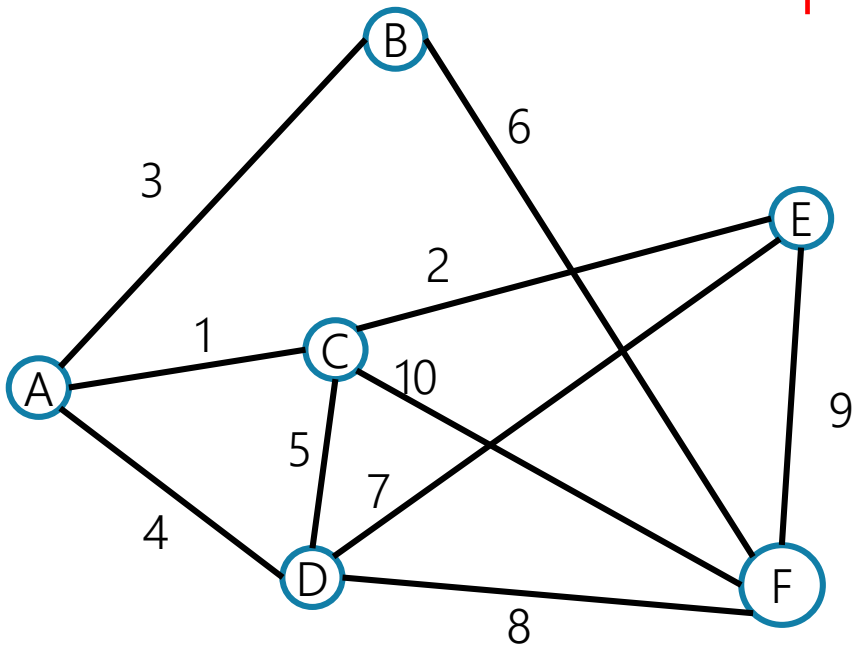
It's the 1920's. Your friend at the electric company needs to choose where to build wires to connect all these cities to the plant.



She knows how much it would cost to lay electric wires between any pair of locations, and wants the cheapest way to make sure electricity from the plant to every city.

Minimum Spanning Trees

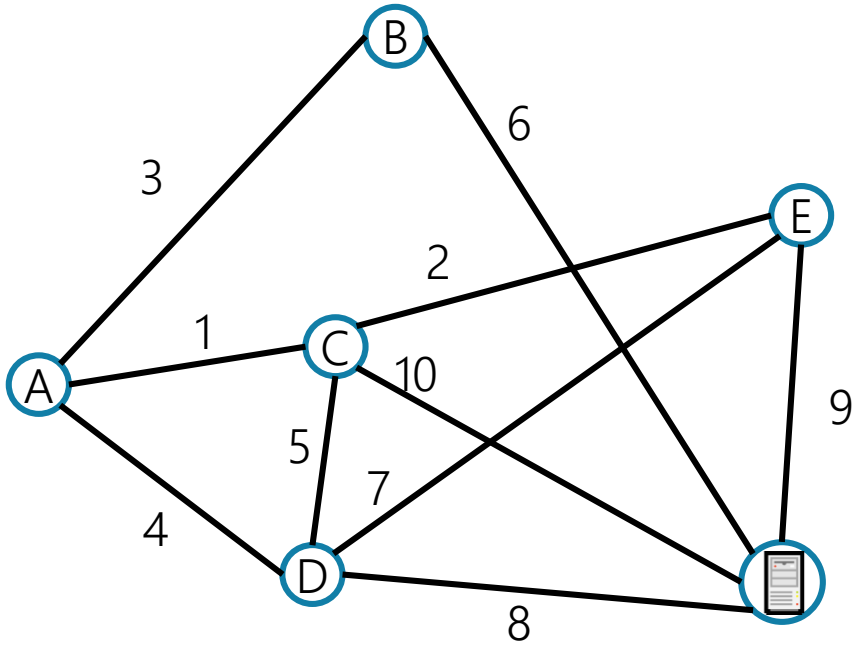
It's the 1950's Your boss at the phone company needs to choose where to build wires to connect all these phones to each other.



She knows how much it would cost to lay phone wires between any pair of locations, and wants the cheapest way to make sure Everyone can call everyone else.

Minimum Spanning Trees

It's today. Your friend at the ISP needs to choose where to build wires to connect all these cities to the Internet.



She knows how much it would cost to lay cable between any pair of locations, and wants the cheapest way to make sure Everyone can reach the server

Minimum Spanning Trees

What do we need? A set of edges such that:

- Every vertex touches at least one of the edges. (the edges **span** the graph)
- The graph on just those edges is **connected**.
  - i.e. the edges are all in the same **connected component.**
  - A connected component is a vertex and everything you can reach from it.
- The minimum weight set of edges that meet those conditions

Claim: The set of edges we pick never has a cycle. Why?

Aside: Trees

On graphs our tees:

- Don't need a root (the vertices aren't ordered, and we can start BFS from anywhere)
- Varying numbers of ~~children~~ neighbors
- Connected and no cycles

Tree (when talking about undirected graphs)

An undirected, connected acyclic graph.

MST Problem

What do we need? A set of edges such that:

- Every vertex touches at least one of the edges. (the edges **span** the graph)
- The graph on just those edges is **connected**.
- The minimum weight set of edges that meet those conditions.

Our goal is a tree!

Minimum Spanning Tree Problem

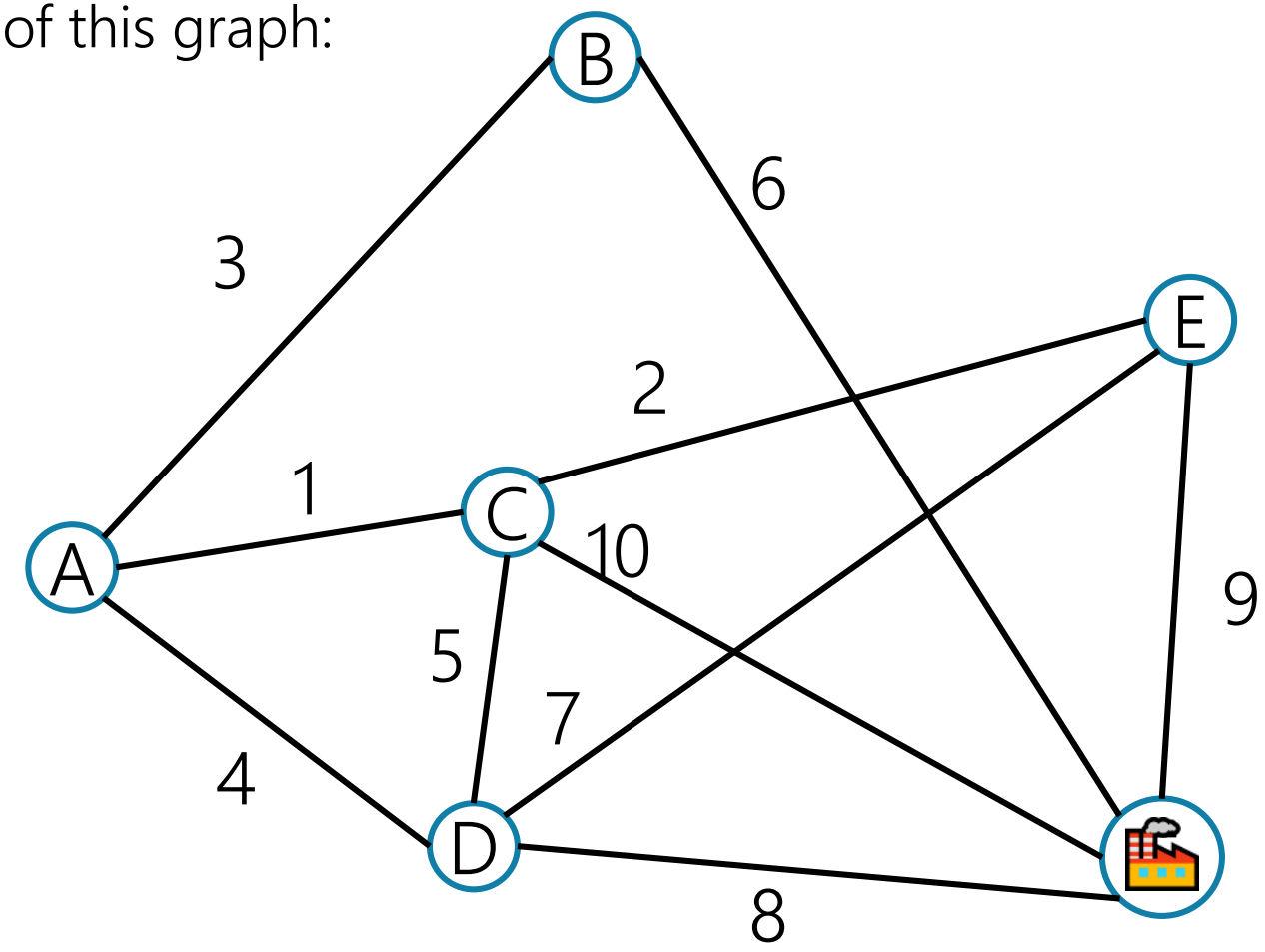
Given: an undirected, weighted graph G

Find: A minimum-weight set of edges such that you can get from any vertex of G to any other on only those edges.

We'll go through two different algorithms for this problem today.

Example

Try to find an MST of this graph:



Prim's Algorithm

Algorithm idea: choose an arbitrary starting point. Add a new edge that:

- Will let you reach more vertices.
- Is as light as possible

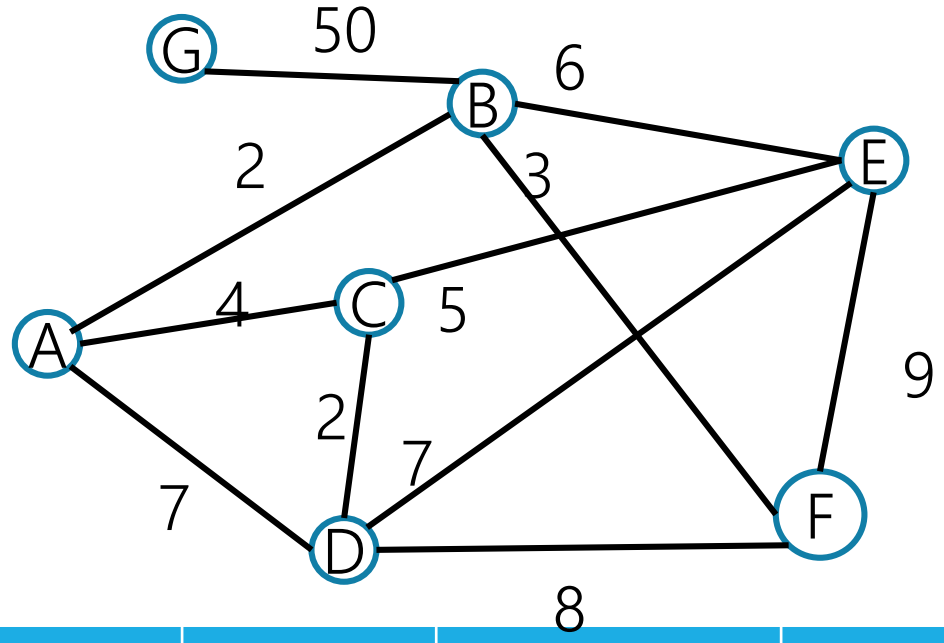
We'd like each not-yet-connected vertex to be able to tell us the lightest edge we could add to connect it.

Code

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

Try it Out

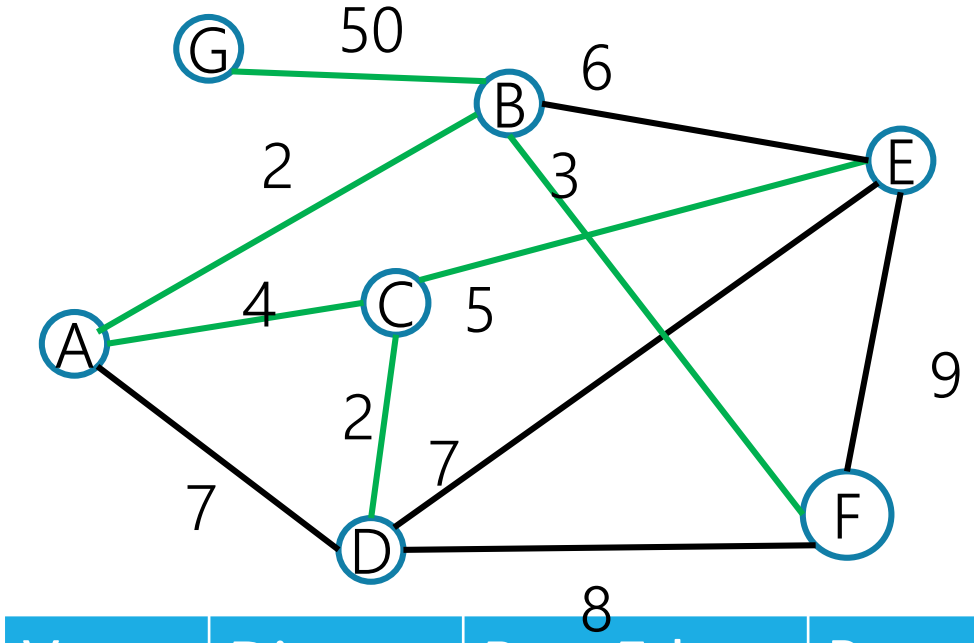
```
PrimMST(Graph G)
    initialize distances to ∞
    mark source as distance 0
    mark all vertices unprocessed
    foreach(edge (source, v) )
        v.dist = weight(source,v)
        v.bestEdge = (source,v)
    while(there are unprocessed vertices){
        let u be the closest unprocessed vertex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



Vertex	Dist.	Best Edge	Processed
A			
B			
C			
D			
E			
F			
G			

Try it Out

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



Vertex	Dist.	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 Algorithm is a **greedy** algorithm. Once it decides to include an edge in the MST it never reconsiders its decision.

Greedy algorithms rarely work.

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

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

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.

Does This Algorithm Always Work?

Prim's Algorithm is a **greedy** algorithm. Once it decides to include an edge in the MST it never reconsiders its decision.

Greedy algorithms rarely work.

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

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

A different Approach

Prim's Algorithm started from a single vertex and reached more and more other vertices.

Prim's thinks vertex by vertex (add the closest vertex to the currently reachable set).

What if you think edge by edge instead?

Start from the lightest edge; add it if it connects new things to each other (don't add it if it would create a cycle)

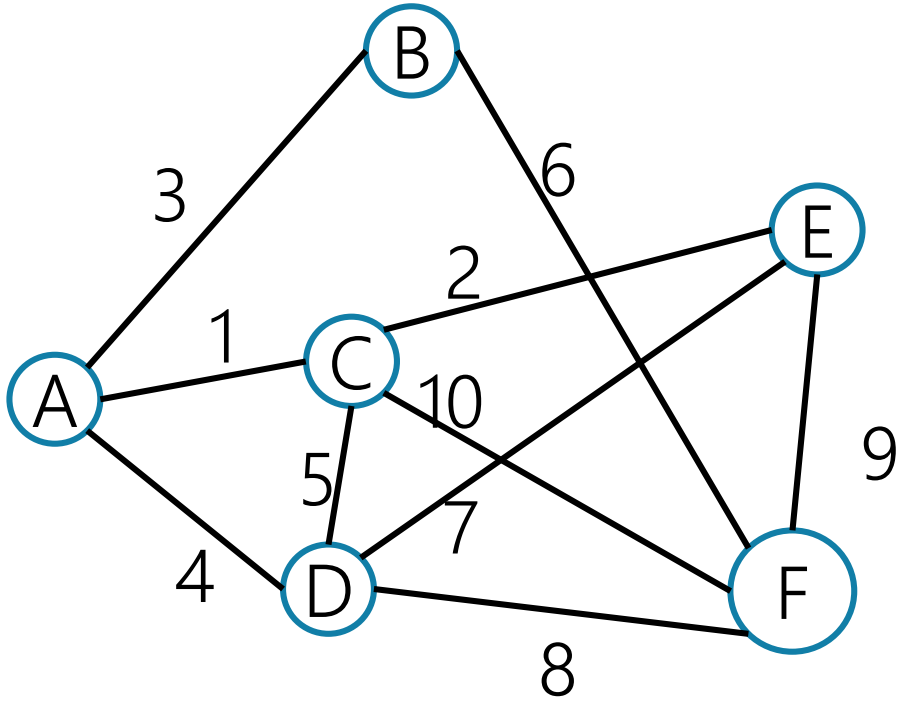
This is Kruskal's Algorithm.

Kruskal's Algorithm

```
KruskalMST(Graph G)
   initialize each vertex to be a connected
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
    }
  }
```

Try It Out

```
KruskalMST(Graph G)
    initialize each vertex to be a connected 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
        }
    }
```

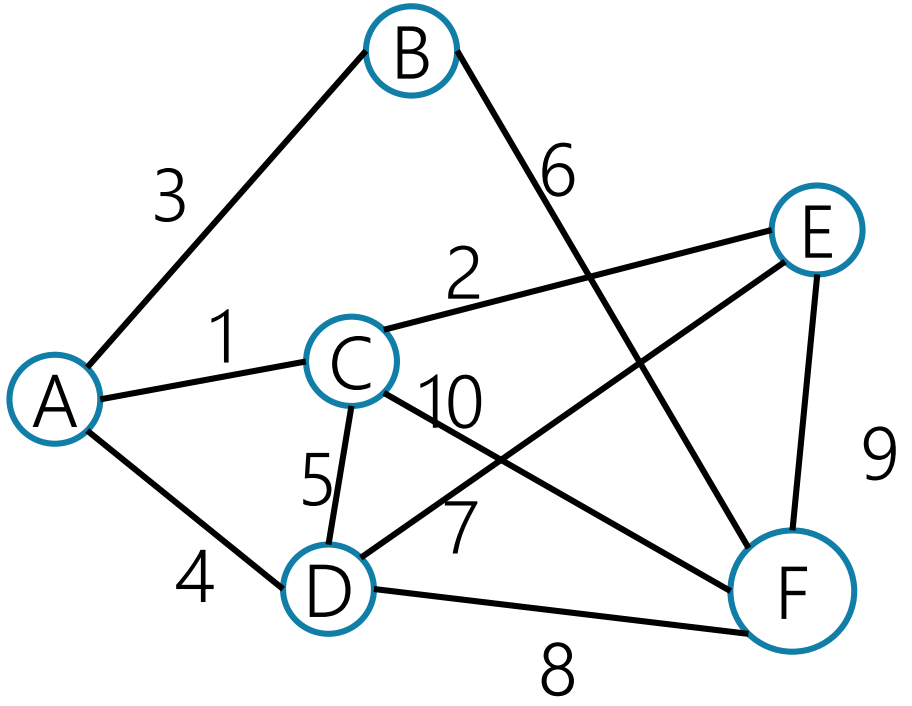


Edge	Include?	Reason
(A,C)		
(C,E)		
(A,B)		
(A,D)		
(C,D)		

Edge (cont.)	Inc?	Reason
(B,F)		
(D,E)		
(D,F)		
(E,F)		
(C,F)		

Try It Out

```
KruskalMST(Graph G)
    initialize each vertex to be a connected 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
        }
    }
```



Edge	Include?	Reason
(A,C)	Yes	
(C,E)	Yes	
(A,B)	Yes	
(A,D)	Yes	
(C,D)	No	Cycle A,C,D,A

Edge (cont.)	Inc?	Reason
(B,F)	Yes	
(D,E)	No	Cycle A,C,E,D,A
(D,F)	No	Cycle A,D,F,B,A
(E,F)	No	Cycle A,C,E,F,D,A
(C,F)	No	Cycle C,A,B,F,C

Kruskal's Algorithm: Running Time

```
KruskalMST(Graph G)
    initialize each vertex to be a connected 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
     }
   }
```

Kruskal's Algorithm: Running Time

Running a new BFS in the partial MST, at every step seems inefficient.

Do we have an ADT that will work here?

Not yet…

Union-Find Crash Course

aka Disjoint Sets

Represents…well…disjoint sets.

Union-Find ADT

state

Set of Sets

- **Disjoint:** No element appears in multiple sets
- No required order
- Each set has representative

behavior

makeSet(x) – creates a new set where the only member (and the representative) is x.

findSet(x) – looks up the set containing element x, returns representative of that set

union(x, y) – combines set containing x and set containing y. Picks new representative.

Union-Find Running Time

We don't have time to talk about an implementation.

Here's the important thing:

Can do these operations in:

Operation	Amortized	Non-amortized
MakeSet()	$\Theta(1)$	$\Theta(1)$
Union()	$O(\log^* n)$	$O(\log n)$
Find()	$O(\log^* n)$	$O(\log n)$

$\log^* n$

$\log^* n$

the number of times you need to apply $\log()$ to get a number at most 1.

E.g. $\log^*(16) = 3$

$\log(16) = 4 \qquad \log(4) = 2 \qquad \log(2) = 1.$

$\log^* n$ grows ridiculously slowly.

$\log^*(10^{80}) = 5.$

For all practical purposes these operations are constant time.

Using Union-Find

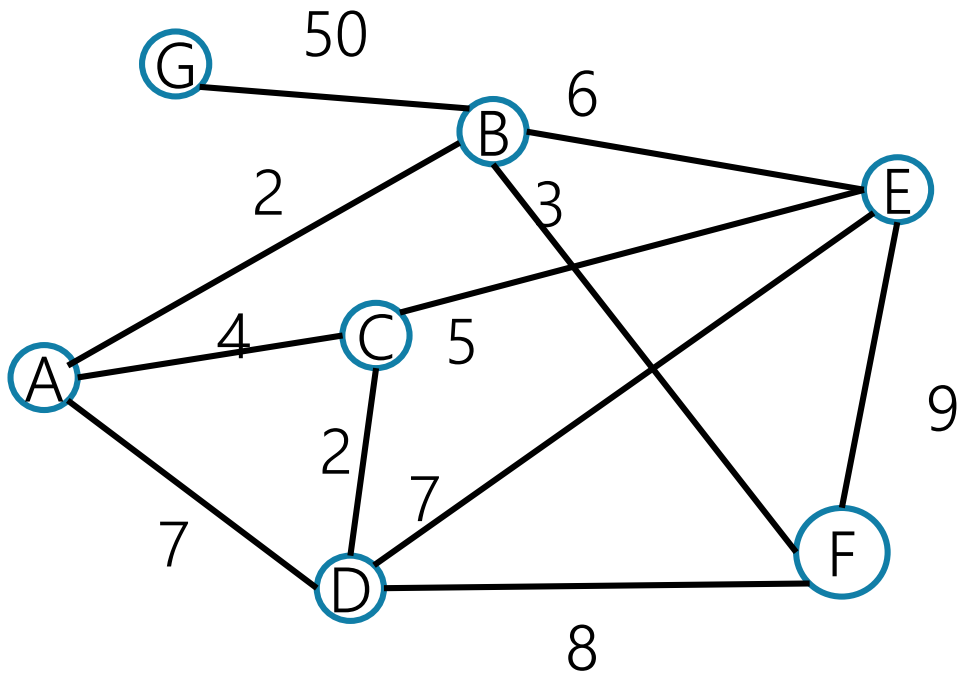
Have each disjoint set represent a connected component

- A connected component is a “piece” of a (disconnected) undirected graph
- i.e. a vertex, and everything you can reach from that vertex.

When you add an edge, you **union** those connected components.

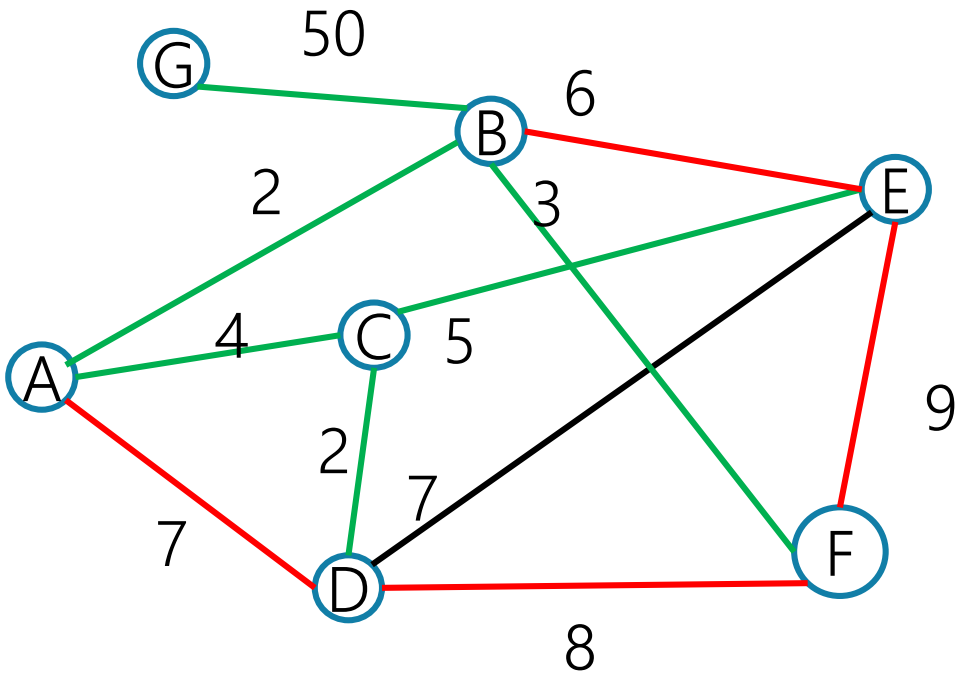
Try it Out

```
KruskalMST(Graph G)
   initialize each vertex to be a connected component
  sort the edges by weight
  foreach(edge (u, v) in sorted order){
    if(find(u) != find(v)){
      add (u,v) to the MST
      Union(u,v)      }
  }
```



Try it Out

Edge	Include?	Reason
(A,B)	Yes	
(C,D)	Yes	
(B,F)	Yes	
(A,C)	Yes	
(C,E)	Yes	
(B,E)	No	Cycle A,C,D,B,A
(A,D)	No	Cycle A,D,C
(D,E)	No	Cycle C,D,E
(D,F)	No	Cycle A,B,F,D,C,A
(E,F)	No	Cycle E,F,B,A,C,E
(B,G)	Yes	



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.

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

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.

How would we describe the graph Kruskal's builds.

It's not a tree until the end.

It's a forest!

A forest is any undirected and acyclic graph

EVERY TREE IS A FOREST.