Minimum Spanning Trees

Data Structures and Algorithms

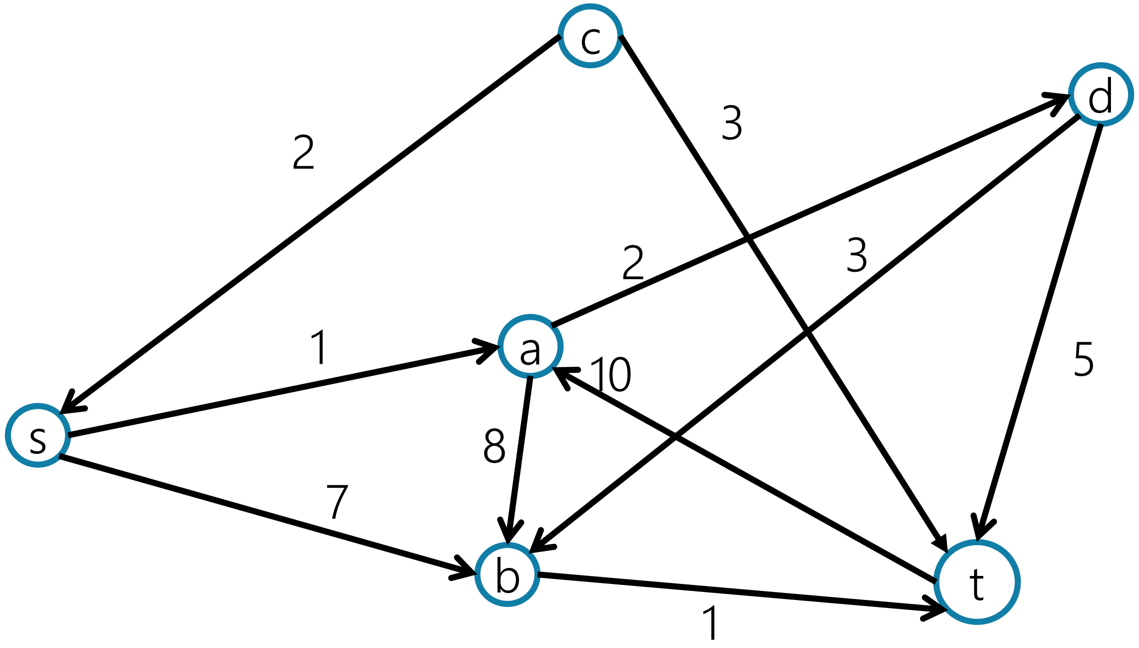
Announcements

Project 3 Part 1 grades are out (only sent to one partner)
Project 3 Part 2 due tonight.

Next written homework out soon.

Warm Up

Run Dijkstra's Algorithm on this graph to find the shortest paths from s to t.



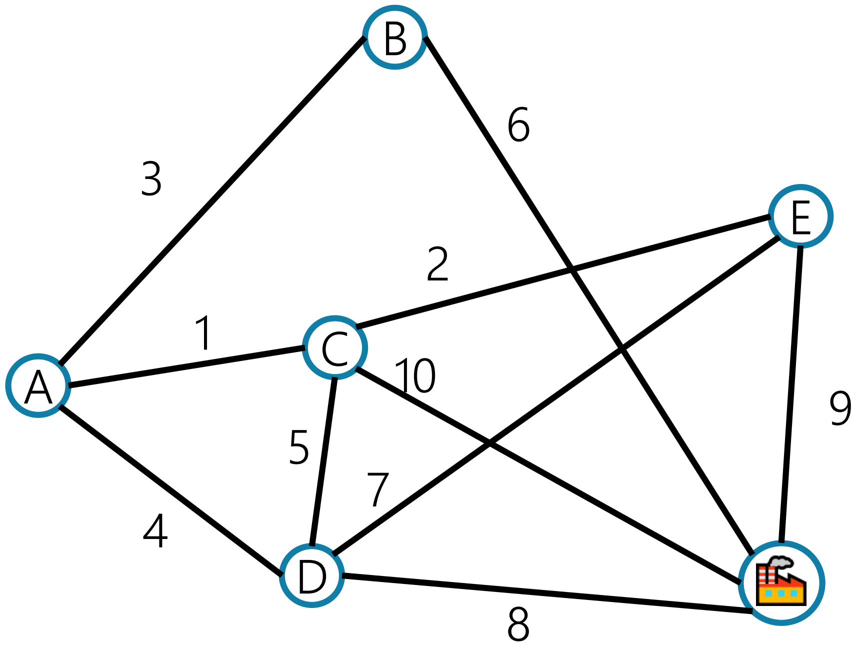
Today

Last Time: Topological Sorting and Strongly Connected Components

Today: One more graph algorithm. Something completely different: Minimum Spanning Trees

Minimum Spanning Trees

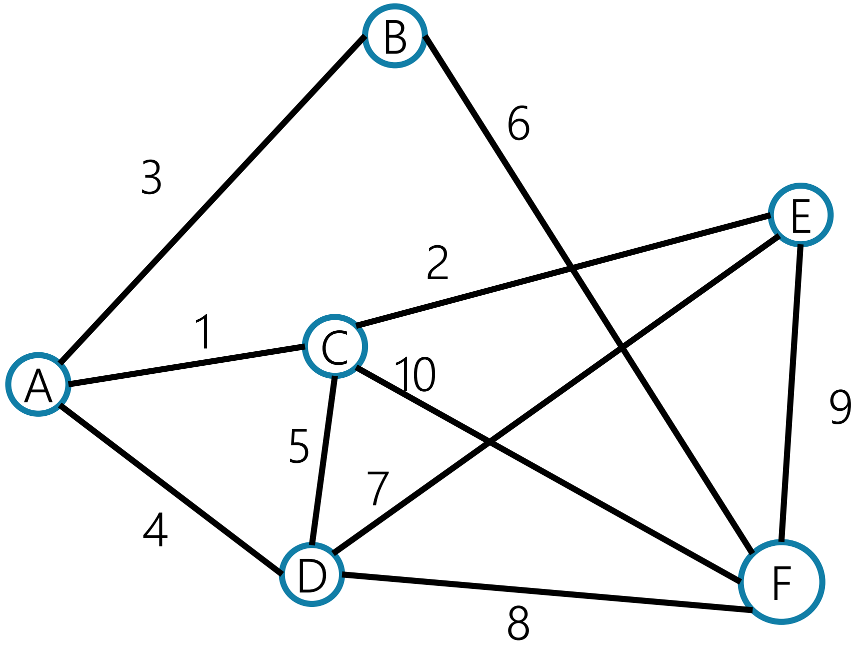
It's the 1920's. Your friend at the electric company needs to choose where to build wires to connect all these cities to the plant.



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

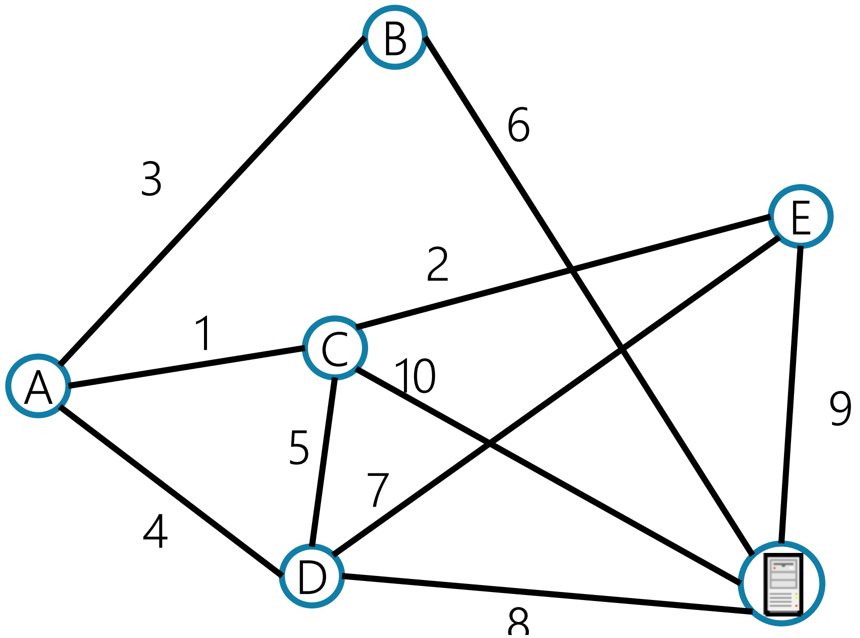
It's the 1950's Your boss at the phone company needs to choose where to build wires to connect all these cities to each other.



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 with fiber optic cable



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**.
- The minimum weight set of edges that meet those conditions.

Assume all edge weights are positive.

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

Aside: Trees

Our BSTs had:

- A root
- Left and/or right children
- Connected and no cycles

Our heaps had:

- A root
- Varying numbers of children
- Connected and no cycles

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
- Connected and no cycles

Tree (when talking about 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!

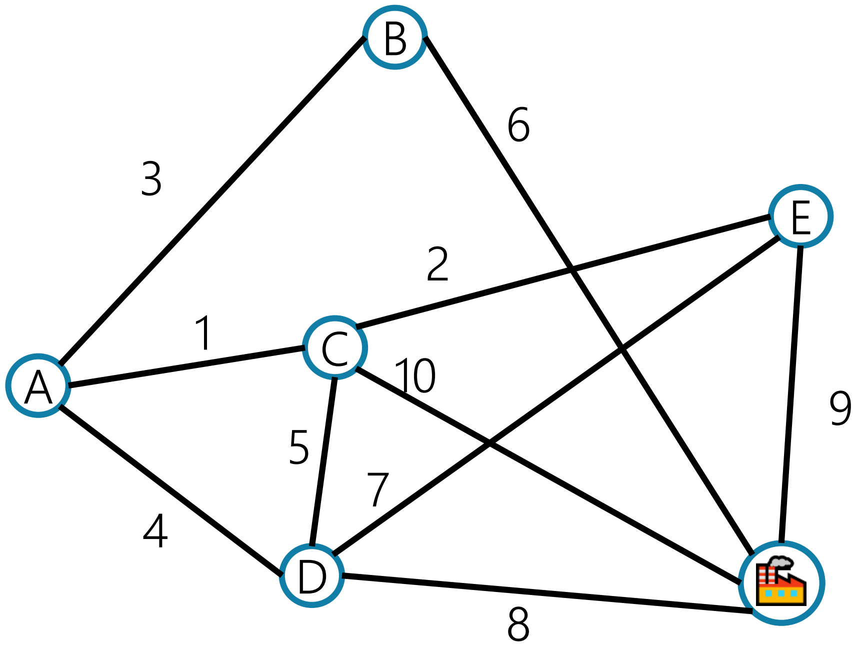
Minimum Spanning Tree Problem

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 = w(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(w(u,v) < v.dist){
                v.dist = w(u,v)
                v.bestEdge = (u,v)
            }
        }
        mark u as processed
    }
```

Running time analysis:
Code structure and data structure choices are the same as for Dijkstra's shortest path algorithm.

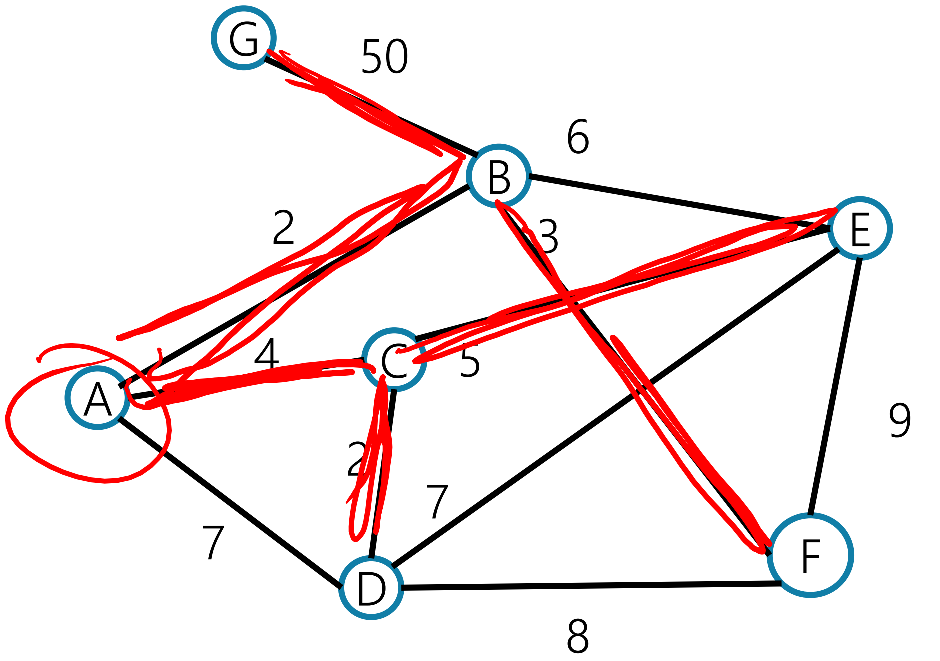
The analysis is very similar.

Running Time:

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

Try it Out

```
PrimMST(Graph G)
   initialize distances to ∞
   mark source as distance 0
   mark all vertices unprocessed
   foreach(edge (source, v) )
      v.dist = w(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(w(u,v) < v.dist){
            v.dist = w(u,v)
            v.bestEdge = (u,v)
         }
      }
     mark u as processed
   }
```



Vertex	Distance	Best Edge	Processed
A	-	⅄	✓
B	2	(A,B)	✓
C	4	(A,C)	✓
D	2	(C,D)	✓
E	5	(C,E)	✓
F	3	(B,F)	✓
G	50	(B,G)	✓

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.

- Robbie can tell you more offline.

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)

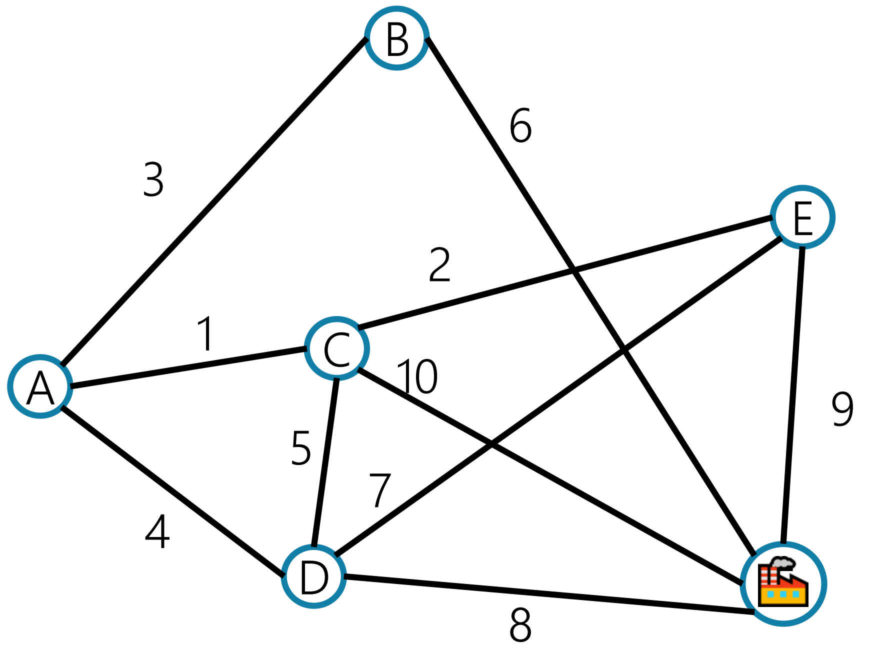
This is Kruskal's Algorithm.

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
      }
   }
```



Fact: A tree on n vertices has exactly n-1 edges.

Once we added the edge (B, Power plant), we could have realized we had the full tree with that condition and saved some time.

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
        }
    }
```

Should run an $O(n \log n)$ sort (e.g. merge sort). Leads to $O(|E| \log |E|)$. Usually written as $O(|E| \log |V|)$. See 5b on the homework for context on why these are equivalent.

Only way we know now is to run BFS in the partial spanning tree. A tree has at most $|V|$ edges, so BFS in a partial tree takes $O(|V|)$ time.

For each loop run $|E|$ times $\Rightarrow O(|E||V| + |E| \log |V|) = \boxed{O(|E||V|)}$ right now

Kruskal's Algorithm: Running Time

Running a new [B/D]FS in the partial MST, at every step seems inefficient.

Tree isn't changing much in a single step. Shouldn't need a whole new BFS each time.

Do we have an ADT that will work here?

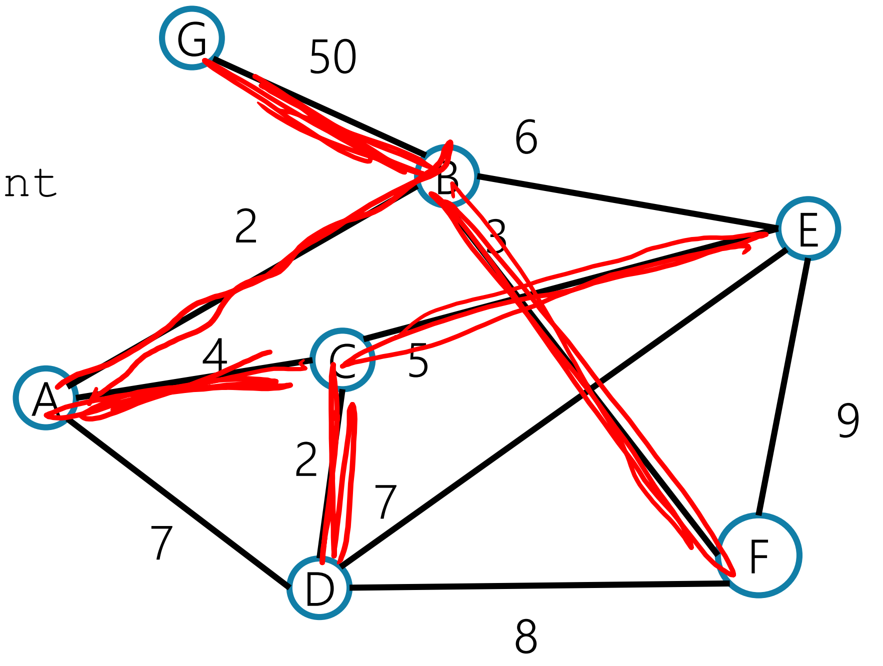
Not yet…

Kasey will tell you about the "Union-Find" ~~data structure~~ ADT next week.

Will give us a better running time than $O(|V||E|)$.

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
      }
   }
```



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.

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

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

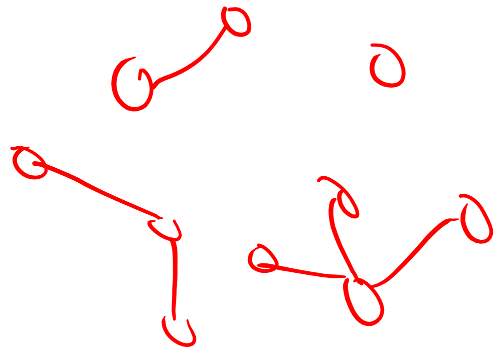
If some edge weights are equal, there may be multiple spanning trees. Prim's/Dijkstra's are only guaranteed to find you one of them.

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

↳ A tree is also a forest (by definition).

EVERY TREE IS A FOREST.

Appendix: MST Properties, Another MST Application

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.

One More MST application

Let's say you're building a new building.

There are very important building donors coming to visit TOMORROW,

- and the hallways are not finished.

You have n rooms you need to show them, connected by the unfinished hallways.

Thanks to your generous donors you have n-1 construction crews, so you can assign one to each of that many hallways.

- Sadly the hallways are narrow and you can't have multiple crews working on the same hallway.

Can you finish enough hallways in time to give them a tour?

Minimum Bottleneck Spanning Tree Problem

Given: an undirected, weighted graph G
Find: A spanning tree such that the weight of the maximum edge is minimized.

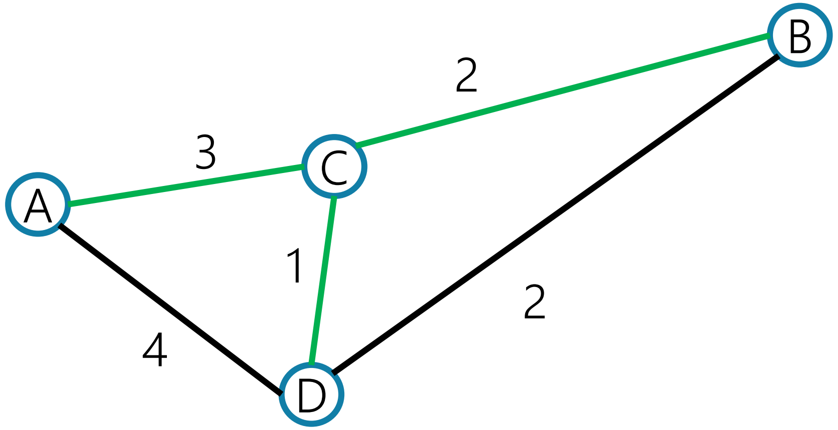
MSTs and MBSTs

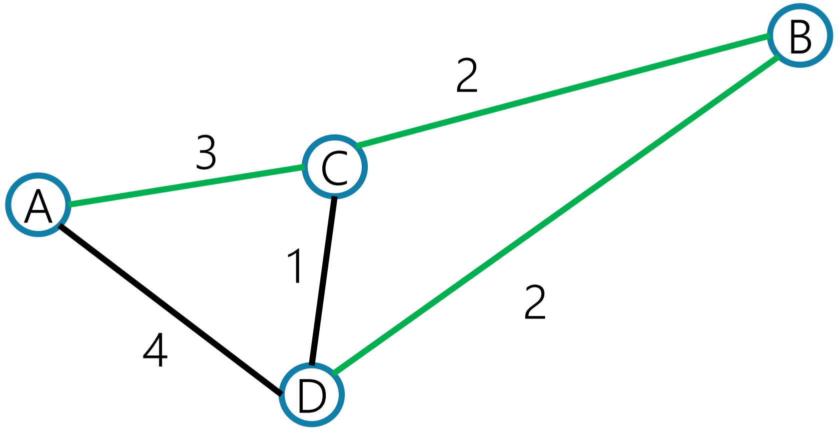
Minimum Spanning Tree Problem

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.

Minimum Bottleneck Spanning Tree Problem

Given: an undirected, weighted graph G
Find: A spanning tree such that the weight of the maximum edge is minimized.





Graph on the right is a minimum bottleneck spanning tree, but not a minimum spanning tree.

Finding MBSTs

Algorithm Idea: want to use smallest edges. Just start with the smallest edge and add it if it connects previously unrelated things (and don't if it makes a cycle).

Hey wait…that's Kruskal's Algorithm!

Every MST is an MBST (because Kruskal's can find any MST when looking for MBSTs)

but not vice versa (see the example on the last slide).

If you need an MBST, any MST algorithm will work.

There are also some specially designed MBST algorithms that are *faster* (see Wikipedia)

Takeaway: When you're modeling a problem, be careful to really understand what you're looking for. There may be a better algorithm out there.