Lecture 19: Minimum Spanning Trees

CSE 373: Data Structures and Algorithms

Administriva

Homework Home Stretch!

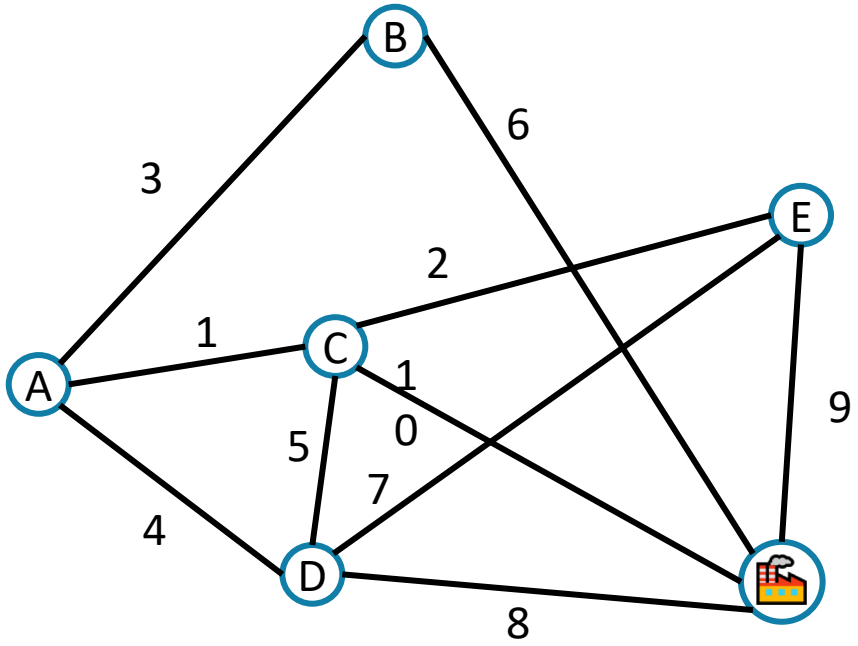
- HW 5 Part 2 due today
- HW 3 regrade due today to HW 3 repo
- HW 6 (individual) out today
- HW 7 Partner Form out today, due 12pm Wednesday

Review Sessions! 6-8pm SIEG 134

- Monday March 4th – Graph Review
- Monday March 11th – Final Review

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

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.

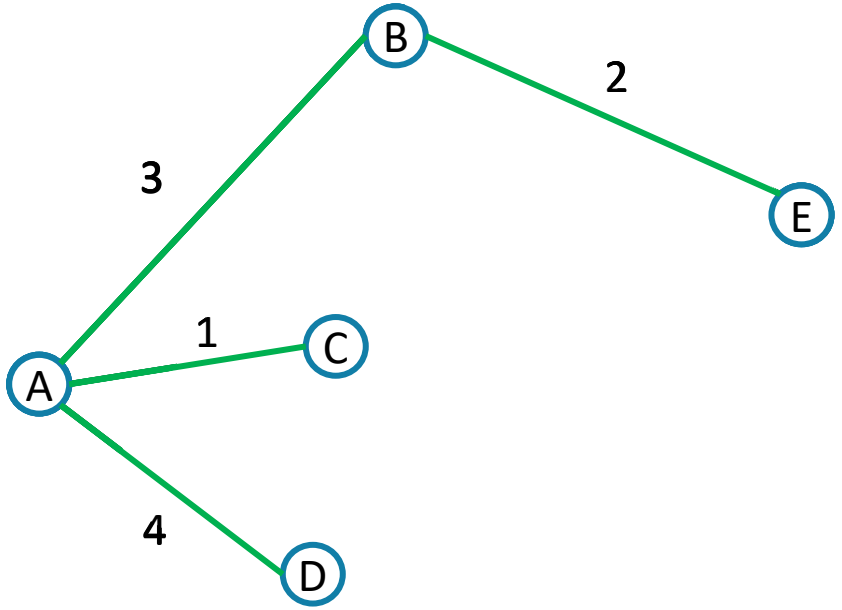
Notice we do not need a directed graph!

Assume all edge weights are positive.

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

MST is the exact number of edges to connect all vertices

- taking away 1 edge breaks connectiveness
- adding 1 edge makes a cycle
- contains exactly V – 1 edges



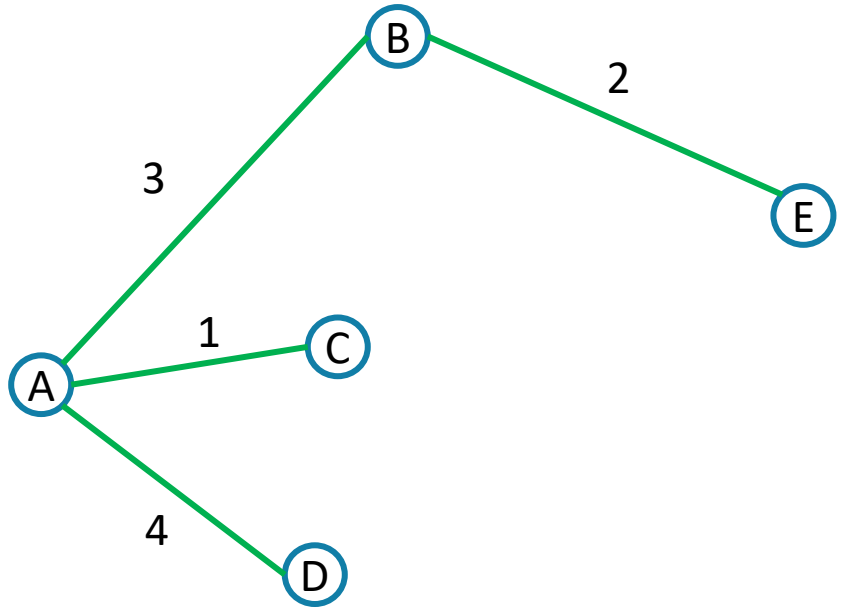
Aside: Trees

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:

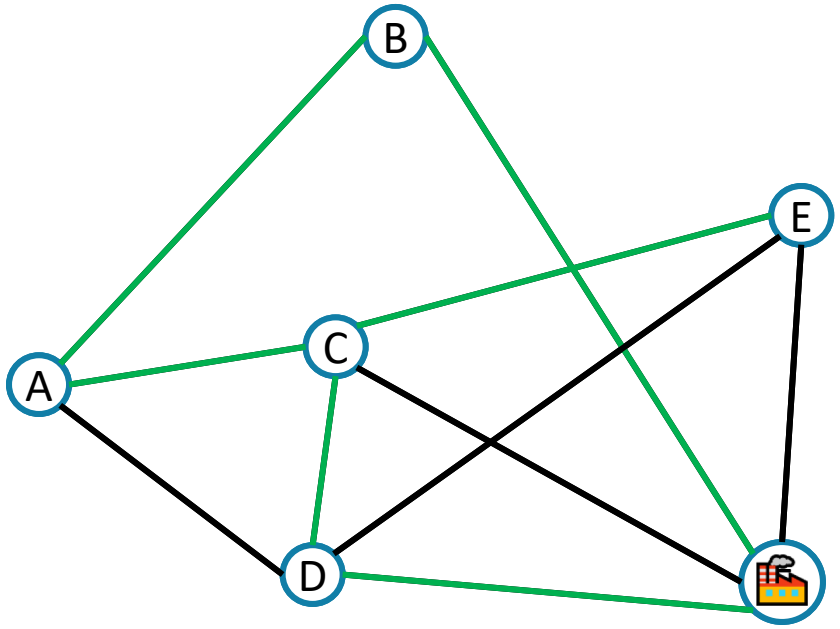
Graph Algorithm Toolbox

BFS

1. Pick an arbitrary starting point
2. Queue up unprocessed neighbors
3. Process next neighbor in queue
4. Repeat until all vertices in queue have been processed

Dijkstra's

1. Start at source
2. Update distance from current to unprocessed neighbors
3. Process optimal neighbor
4. Repeat until all vertices have been marked processed



Example

Try to find an MST of this graph:

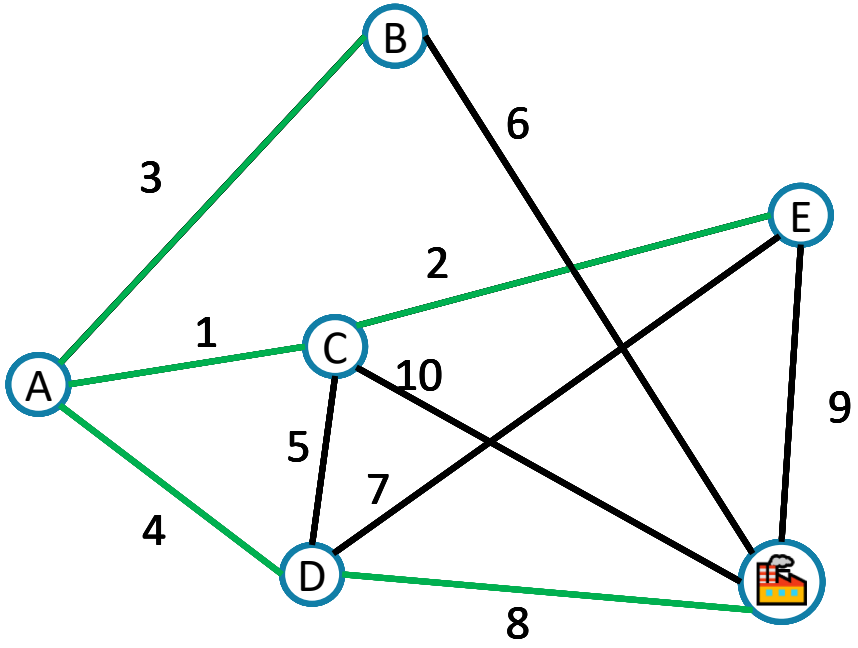
Graph Algorithm Toolbox

BFS

1. Pick an arbitrary starting point
2. Queue up unprocessed neighbors
3. Process next neighbor in queue
4. Repeat until all vertices in queue have been processed

Dijkstra's

1. Start at source
2. Update distance from current to unprocessed neighbors
3. Process optimal neighbor
4. Repeat until all vertices have been marked processed



Prim's Algorithm

Dijkstra's

1. Start at source
2. Update distance from current to unprocessed neighbors
3. Process optimal neighbor
4. Repeat until all vertices have been marked processed

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark source as distance 0
   mark all vertices unprocessed
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      foreach(edge (u,v) leaving u){
         if(u.dist+weight(u,v) < v.dist){
            v.dist = u.dist+weight(u,v)
            v.predecessor = u
         }
      }
      mark u as processed
   }
```

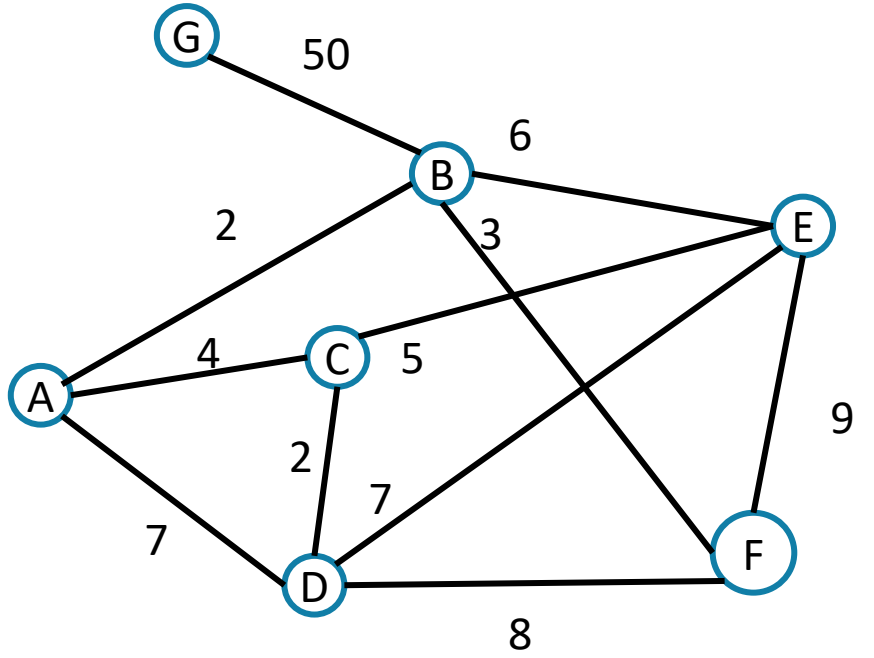
Algorithm idea:

1. choose an arbitrary starting point
2. Investigate edges that connect unprocessed vertices
3. Add the lightest edge to solution (be greedy)
4. Repeat until solution connects all vertices

```
Prims(Graph G, Vertex source)
   initialize distances to ∞
   mark source as distance 0
   mark all vertices unprocessed
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      foreach(edge (u,v) leaving u){
         if(weight(u,v) < v.dist){
            v.dist = u.dist+weight(u,v)
            v.predecessor = u
         }
      }
      mark u as processed
   }
```

Try it Out

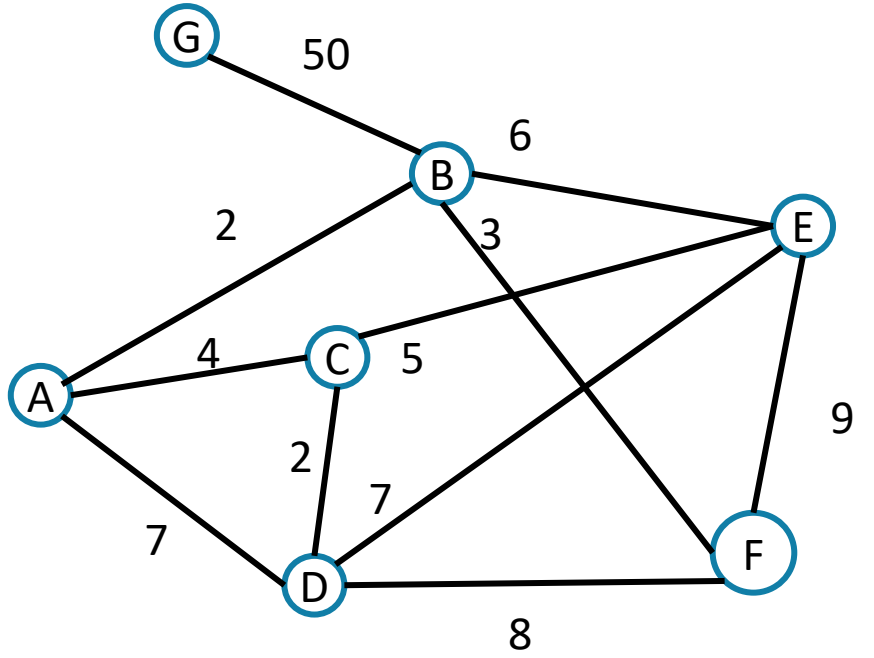
```
PrimMST(Graph G)
  initialize distances to ∞
  mark source as distance 0
  mark all vertices unprocessed
  foreach(edge (source, v) ) {
    v.dist = weight(source,v)
    v.bestEdge = (source,v)
  }
  while(there are unprocessed vertices){
    let u be the closest unprocessed vertex
    add u.bestEdge to spanning tree
    foreach(edge (u,v) leaving u){
      if(weight(u,v) < v.dist && v unprocessed ){
        v.dist = weight(u,v)
        v.bestEdge = (u,v)
      }
    }
    mark u as processed
  }
```



Vertex	Distance	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 && v unprocessed ){
        v.dist = weight(u,v)
        v.bestEdge = (u,v)
      }
    }
    mark u as processed
  }
```



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

Prim's Runtime

Runtime = VlogV + ElogV

```
Prims(Graph G, Vertex source)
   initialize distances to ∞
   mark source as distance 0
   mark all vertices unprocessed
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      foreach(edge (u,v) leaving u){
         if(weight(u,v) < v.dist){
            v.dist = u.dist+weight(u,v)
            v.predecessor = u
         }
      }
      mark u as processed
   }
```

Runtime = VlogV + ElogV

```
Dijkstra(Graph G, Vertex source)
   initialize distances to ∞
   mark source as distance 0
   mark all vertices unprocessed
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      foreach(edge (u,v) leaving u){
         if(u.dist+weight(u,v) < v.dist){
            v.dist = u.dist+weight(u,v)
            v.predecessor = u
         }
      }
      mark u as processed
   }
```

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).

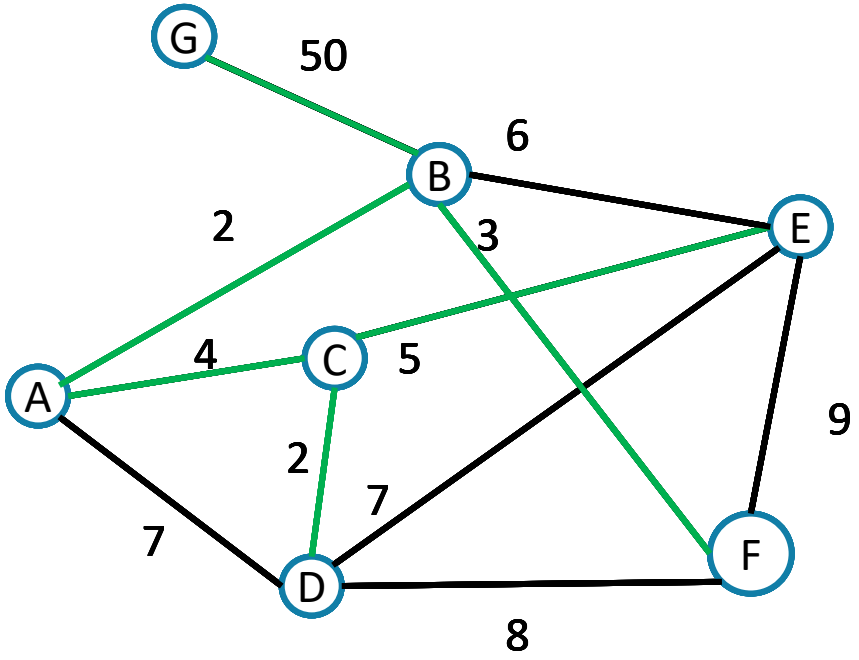
What if you think edge by edge instead?

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.

Example

Try to find an MST of this graph by adding edges in sorted order

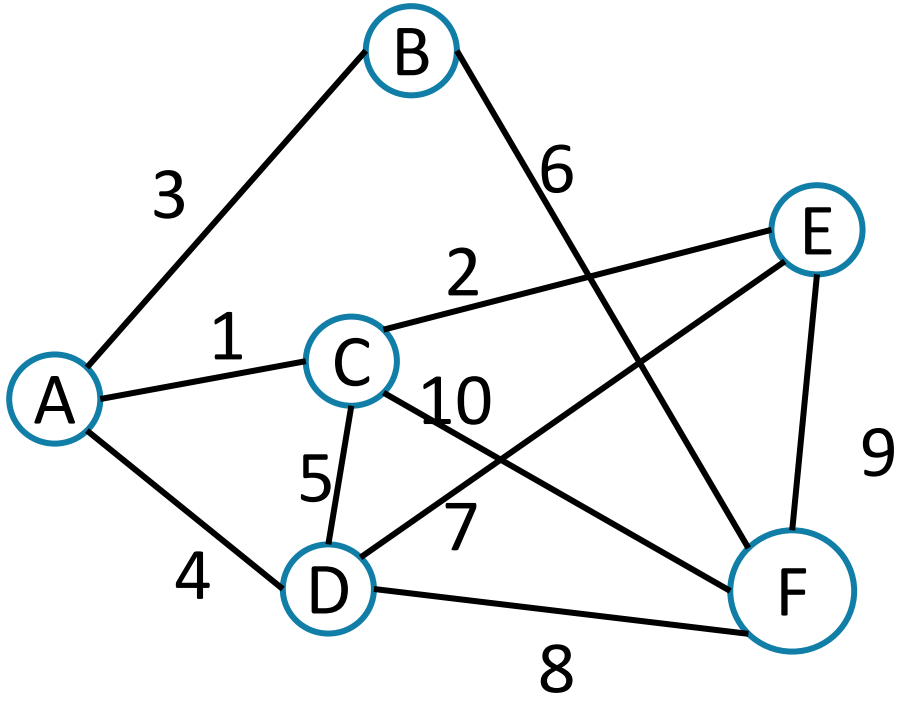


Kruskal's Algorithm

```
KruskalMST(Graph G)
    initialize each vertex to be an independent 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 an independent 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
         }
     }
```

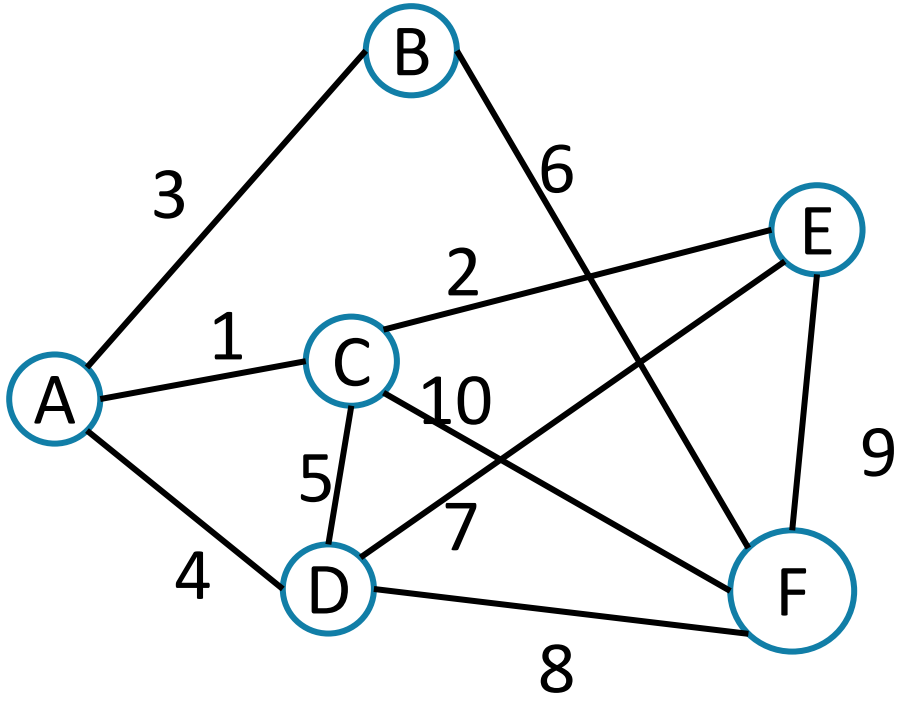


Edge	Include?	Reason
(A,C)		
(C,E)		
(A,B)		
(A,D)		
(C,D)		

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 an independent 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

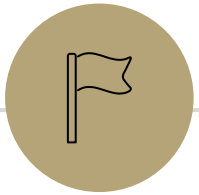
Edge (cont.)	Inc?	Reason
(B,F)	Yes	
(D,E)	No	Cycle A,C,E,D,A
(D,F)	No	Cycle A,D,F,B,A
(E,F)	No	Cycle A,C,E,F,D,A
(C,F)	No	Cycle C,A,B,F,C

Kruskal's Algorithm Implementation

```
KruskalMST(Graph G)
   initialize each vertex to be an independent 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
      }
   }
```

```
KruskalMST(Graph G)
   foreach (V : vertices) {
      makeMST(v); +?
   }                                  +V(makeMST)
   sort edges in ascending order by weight +ElogE
   foreach(edge (u, v)){
      if(findMST(v) is not in findMST(u)){+?
         union(u, v) +?
      }                               +E(2findMST + union)
   }
```

How many times will we call union?
V – 1
-> **+Vunion + EfindMST**



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.

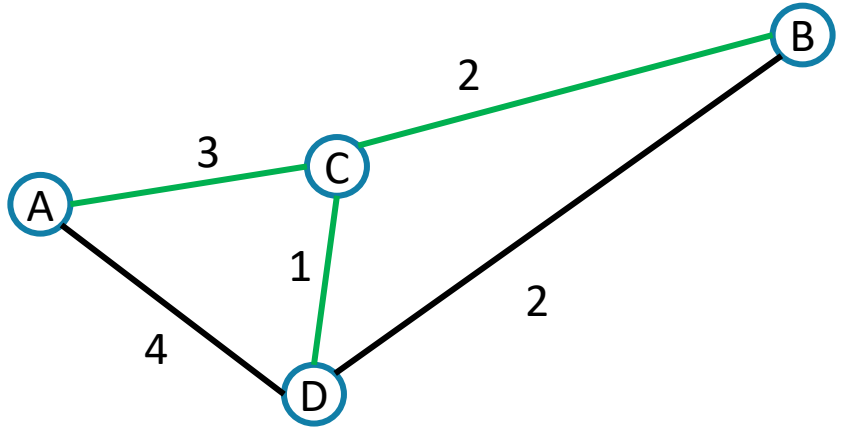
MSTs and MBSTs

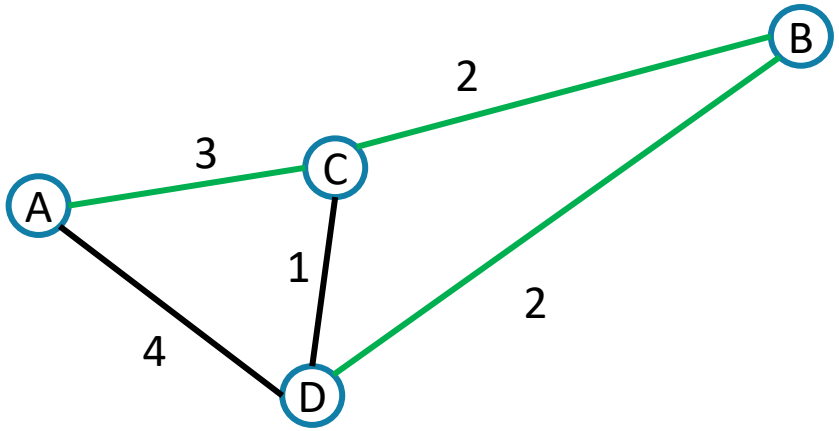
Minimum Spanning Tree Problem

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.