Minimum Spanning Trees

CSE 332 Sp25
Lecture 25

Announcements

	Monday	Tuesday	Wed	Thursday	Friday
This Week	Veteran's Day	Ex 11 (prefix prog)due	Ex 13 (MST,prog) out		TODAY Ex 12 (concurrency, GS) due final conflict form due
Next Week	Ex 14 (P/NP, GS) out		Ex 13 due		Ex 14 due

Final Exam information page is up.

If you are requesting a conflict exam, please fill out form today!
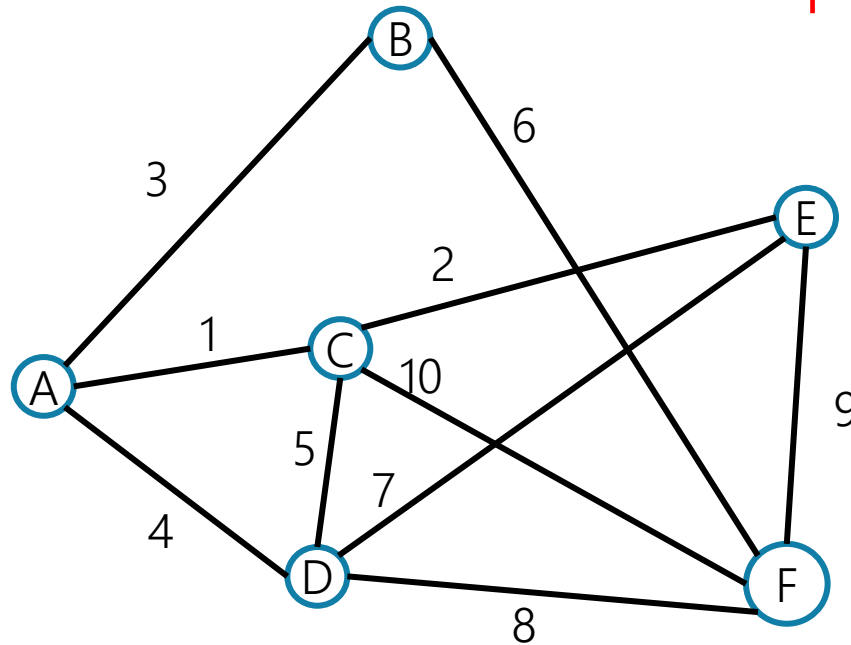
Minimum Spanning Trees

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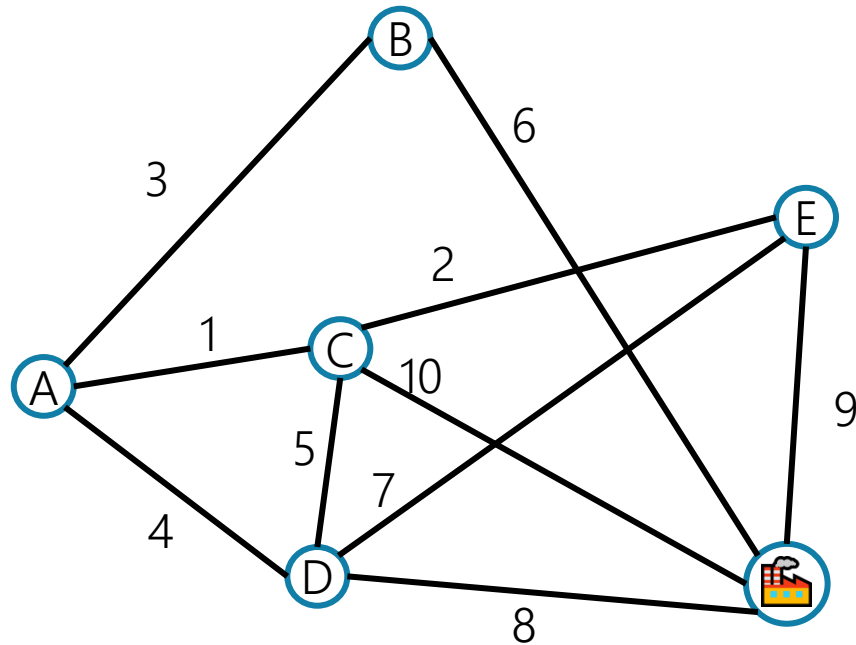
She knows how much it would cost to lay electric wires between any pair of locations, and wants the cheapest way to make sure electricity from the plant to every city.
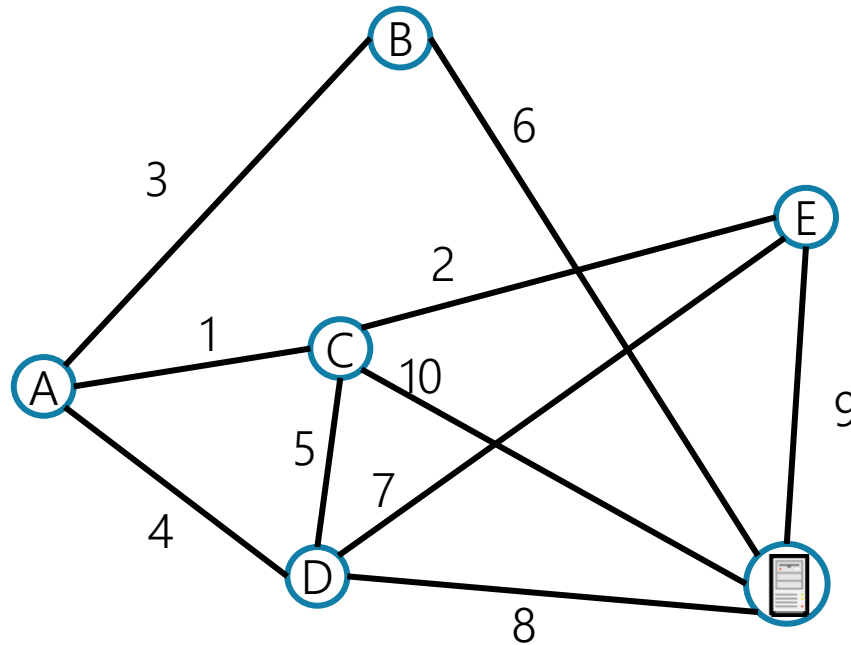
Minimum Spanning Trees

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
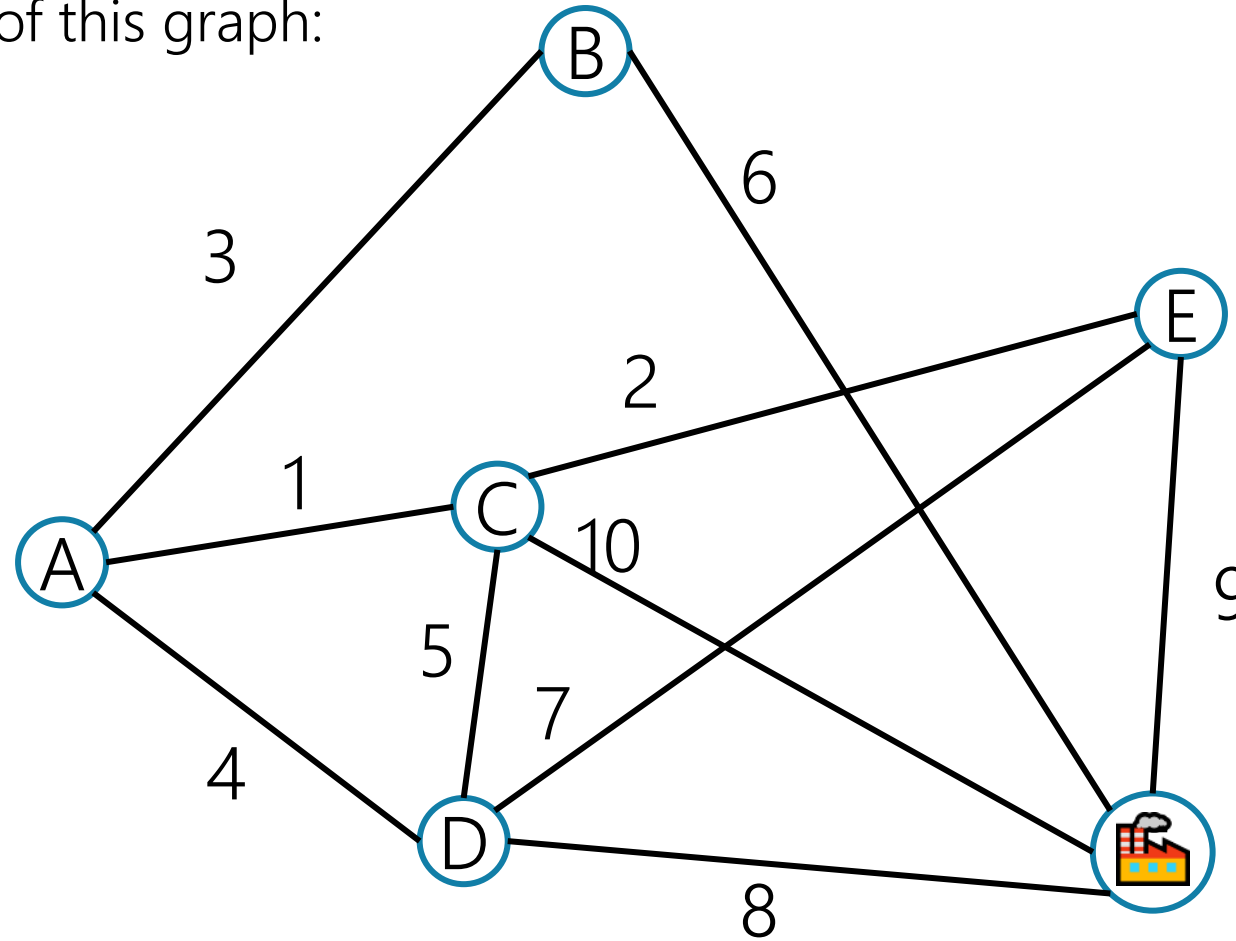
Given: an undirected, weighted graph G

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.

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 costs to ∞
    mark source as cost 0
    mark all vertices unprocessed
    foreach(edge (source, v) ) {
        v.cost = weight(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.cost){
                v.cost = weight(u,v)
                v.bestEdge = (u,v)
            }
        }
        mark u as processed
    }
}
```

Try it Out

```
PrimMST(Graph G)
    initialize costs to ∞
    mark source as cost 0
    mark all vertices unprocessed
    foreach(edge (source, v) ) {
        v.cost = weight(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.cost){
                v.cost = weight(u,v)
                v.bestEdge = (u,v)
            }
        }
        mark u as processed
    }
}
```
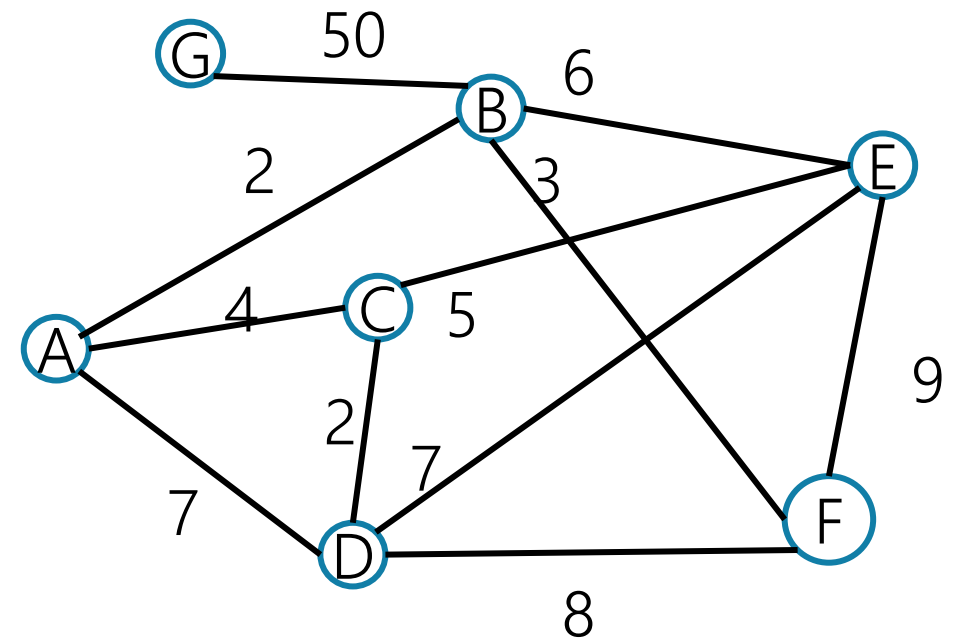

Vertex	Cost	Best Edge	Processed
A			
B			
C			
D			
E			
F			
G			

Try it Out

```
PrimMST(Graph G)
    initialize costs to ∞
    mark source as cost 0
    mark all vertices unprocessed
    foreach(edge (source, v) ) {
        v.cost = weight(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.cost){
                v.cost = weight(u,v)
                v.bestEdge = (u,v)
            }
        }
        mark u as processed
    }
}
```
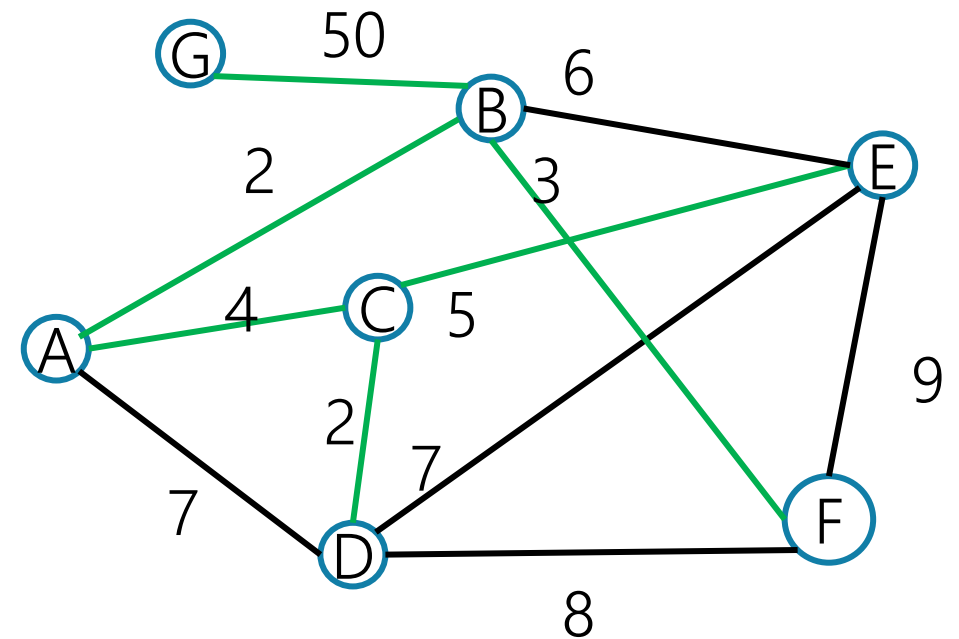

Vertex	Cost	Best Edge	Processed
A	--	--	Yes
B	2	(A,B)	Yes
C	4	(A,C)	Yes
D	~~7~~ 2	~~(A,D)~~(C,D)	Yes
E	~~6~~ 5	~~(B,E)~~(C,E)	Yes
F	3	(B,F)	Yes
G	50	(B,G)	Yes

```
PrimMST(Graph G)
    initialize costs to ∞
    mark source as cost 0
    mark all vertices unprocessed //and add to priority queue
    foreach(edge (source, v) ) {
        v.cost = weight(source,v) }
    while(there are unprocessed vertices){
        let u be the closest unprocessed vertex //removeMin
        add u.bestEdge to spanning tree
        foreach(edge (u,v) leaving u){
            if(weight(u,v) < v.cost){
                v.cost = weight(u,v) //updatePriority!!
                v.bestEdge = (u,v)
            }
        }
        mark u as processed
    }
}
```

Running time: $\Theta(E \log V)$
Analysis same as Dijkstra, but can assume $E \geq V - 1$

Some Exercise Notes

We'll ask you to implement Prim's in Exercise 13.

You have a few options for the priority queue:

1. Use a Java library priority queue---but it doesn't have updatePriority() so you'll need a workaround:
A. Add edges instead of vertices to the priority queue OR
B. Allow multiple copies of each vertex into the queue (instead of decreasing priority, put in a second copy at the new priority OR

2. Use your (Exercise 2) priority queue instead---call updatePriority!

Will these change the running time? No! $\log(E) = \Theta(\log(V))$ for simple graphs.

Read the paragraph in the spec about this before you get too far. Also see alternate version of pseudocode in section slides tomorrow.

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)
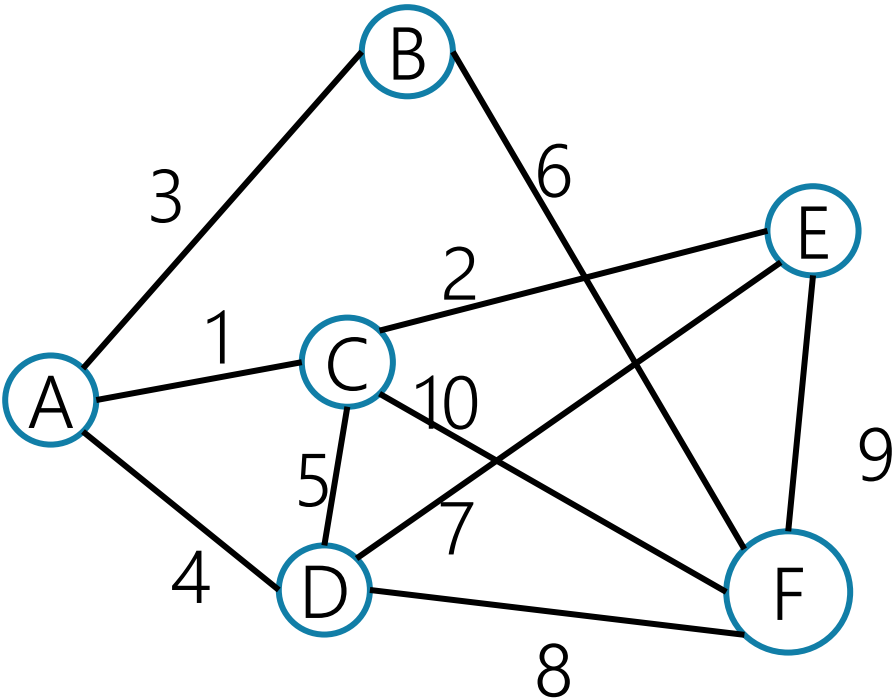
This is Kruskal's Algorithm.

Kruskal's Algorithm

```
KruskalMST(Graph G)
   initialize each vertex to be a connected component
   sort the edges by weight (increasing)
   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
        }
    }
}
```
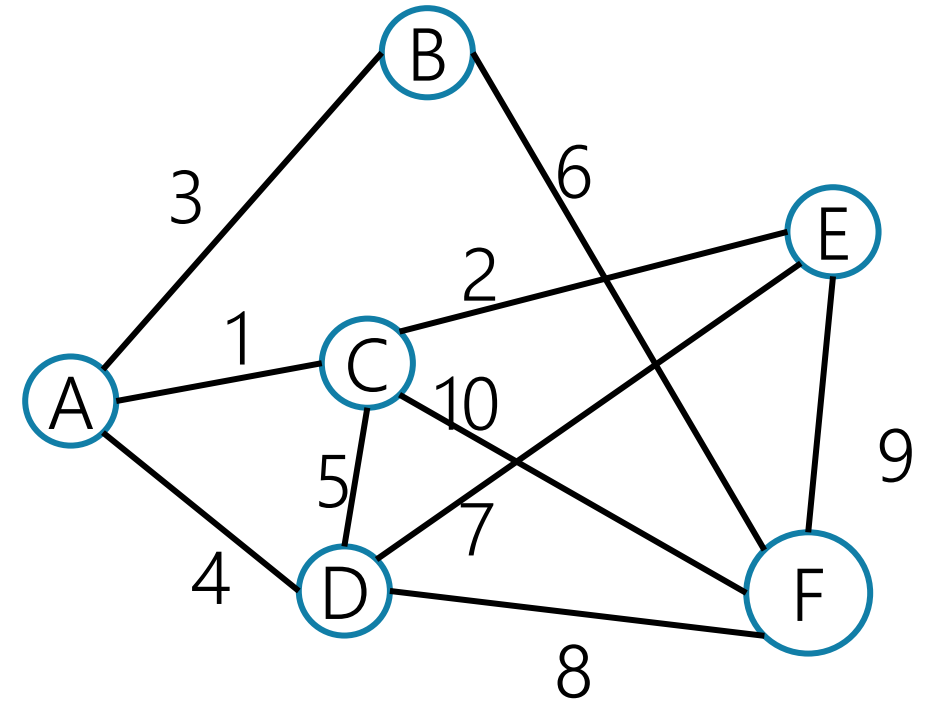
Edge	Include?	Reason
(A,C)	Yes	
(C,E)	Yes	
(A,B)	Yes	
(A,D)	Yes	
(C,D)	No	Cycle A,C,D,A

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

How do we find connected components? Well BFS is our existing tool to do that, but...

Running a new BFS in the partial MST, at every step seems inefficient. The answer changes little by little, so we'll recompute work frequently.

Do we have an ADT that will work here?

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 name of that set

union(x, y) – combines sets containing x and y. Picks new name for combined set.

Union-Find Running Time

What's important for us?

Amortized running times! We care about the total time across the entire set of unions and finds, not the running time of just one.

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

Uses "forest of up-trees" implementation.

$\log^* n$

$\log^* n$

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

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

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

$\log^* n$ grows <u>ridiculously</u> slowly.

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

For all practical purposes these operations are constant time.
They're not constants (don't delete them from big-O notation), but you will never worry about these in figuring out how many seconds a piece of code takes.

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(find(u),find(v))
    }
  }
```
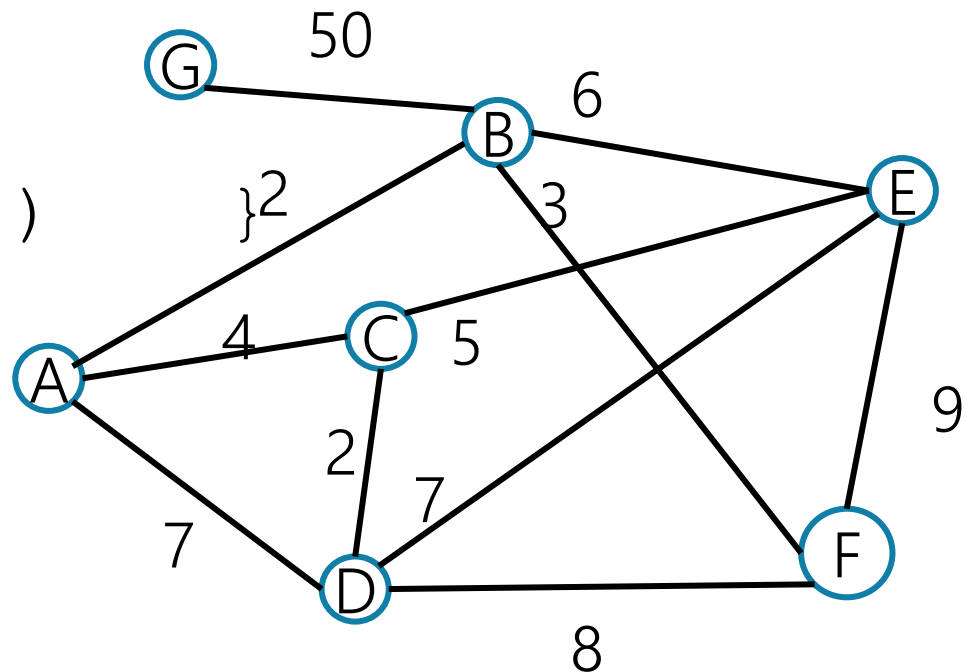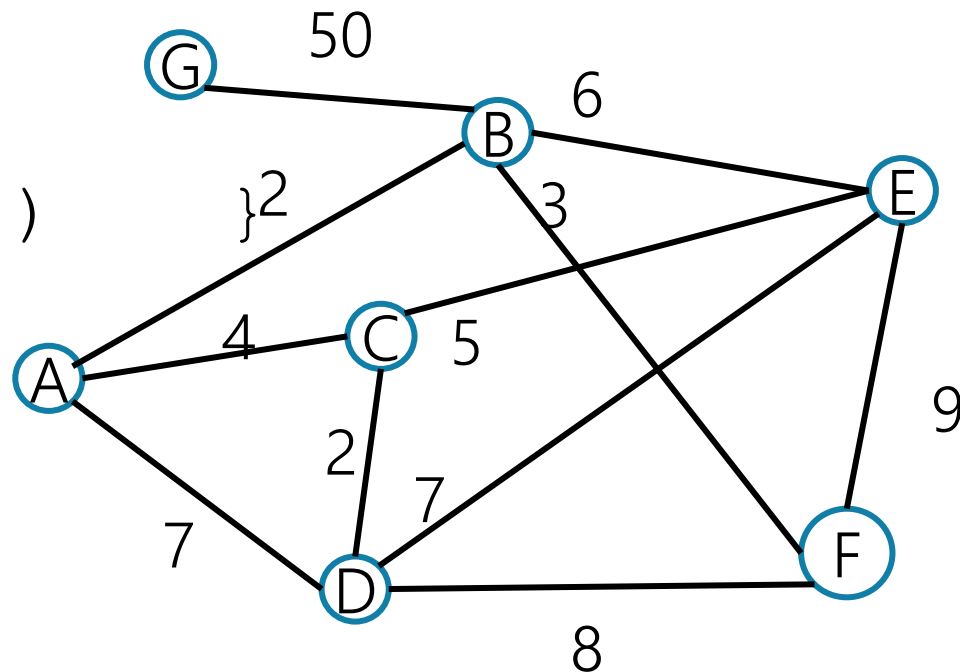
Running Time?

```
KruskalMST(Graph G)
    initialize each vertex to be a connected component
    sort the edges by weight                $E \log E$
    foreach(edge (u, v) in sorted order){
$E \log^* V$  if(find(u) != find(v)){
    $E$        add (u,v) to the MST
$E \log^* V$      union(find(u),find(v))
    }
```

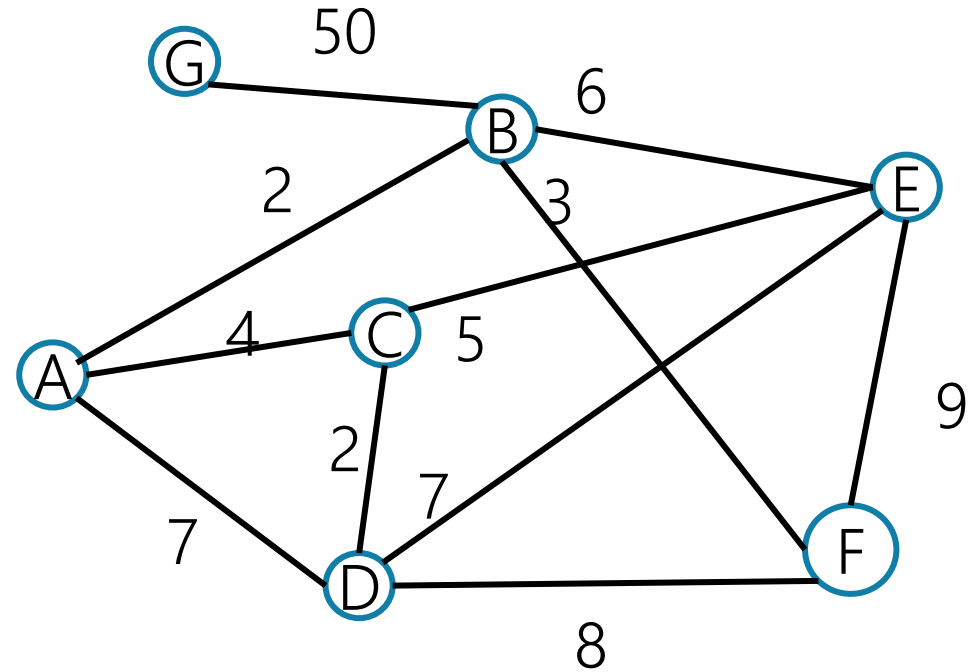$E \log E$ is the dominant term, but
you'll usually see this written
$$\Theta(E \log V)$$
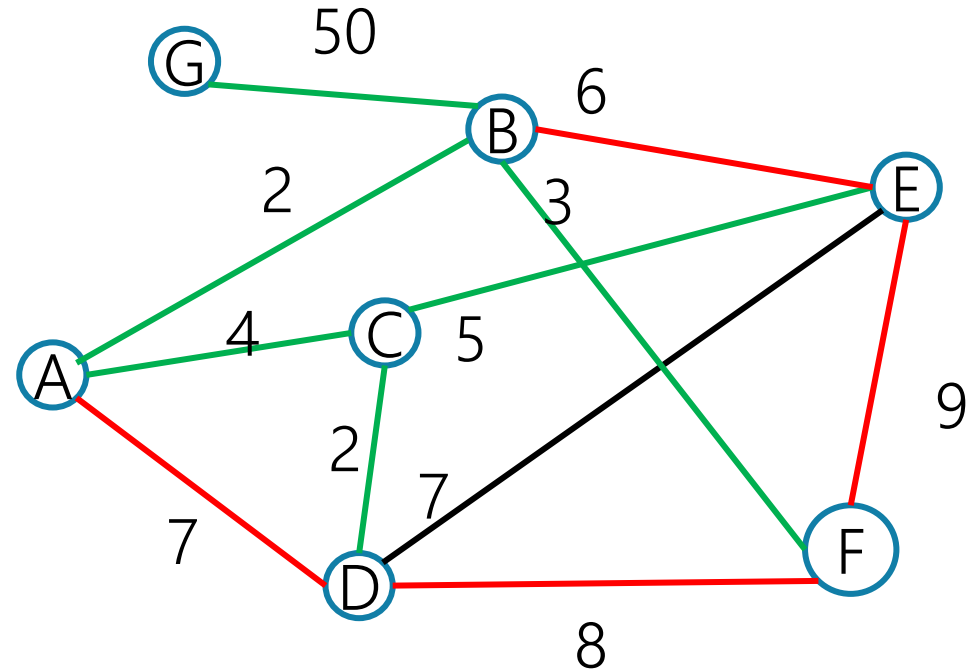Since $E \leq V^2$, those are equivalent.

Try it Out

Edge	Include?	Reason

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.

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!

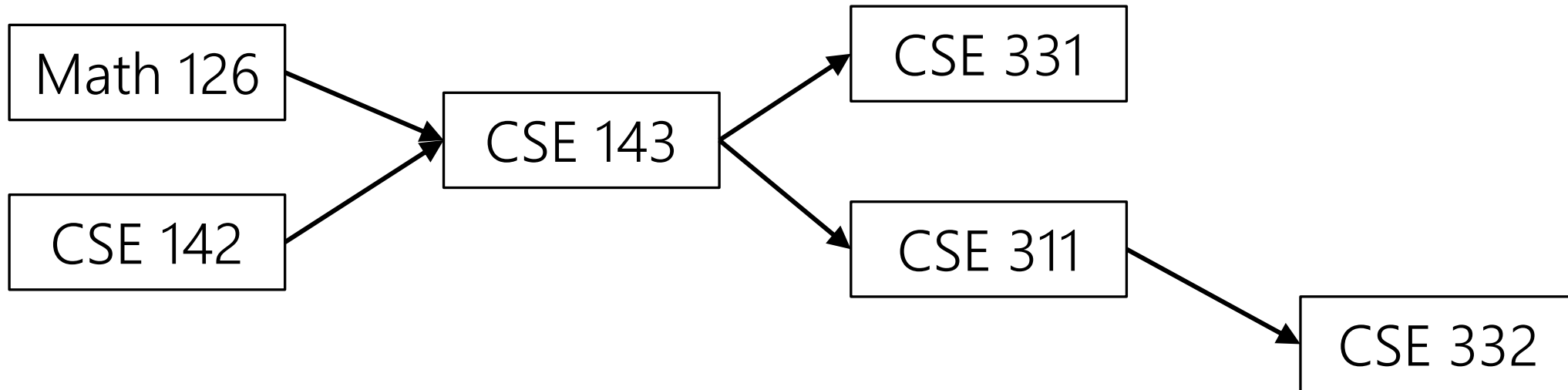A forest is any undirected and acyclic graph

EVERY TREE IS A FOREST.

Two More Simple Graph Algorithms

Ordering Dependencies

Today's next problem: Given a bunch of courses with prerequisites, find an order to take the courses in.

Ordering Dependencies

Given a directed graph G, where we have an edge from u to v if u must happen before v.

Can we find an order that **respects dependencies**?

Topological Sort (aka Topological Ordering)
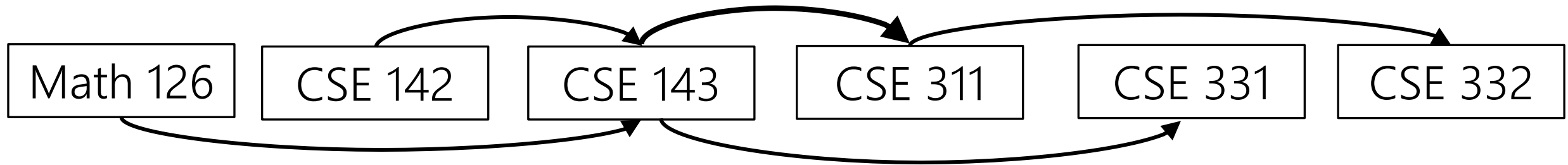**Given:** a directed graph G
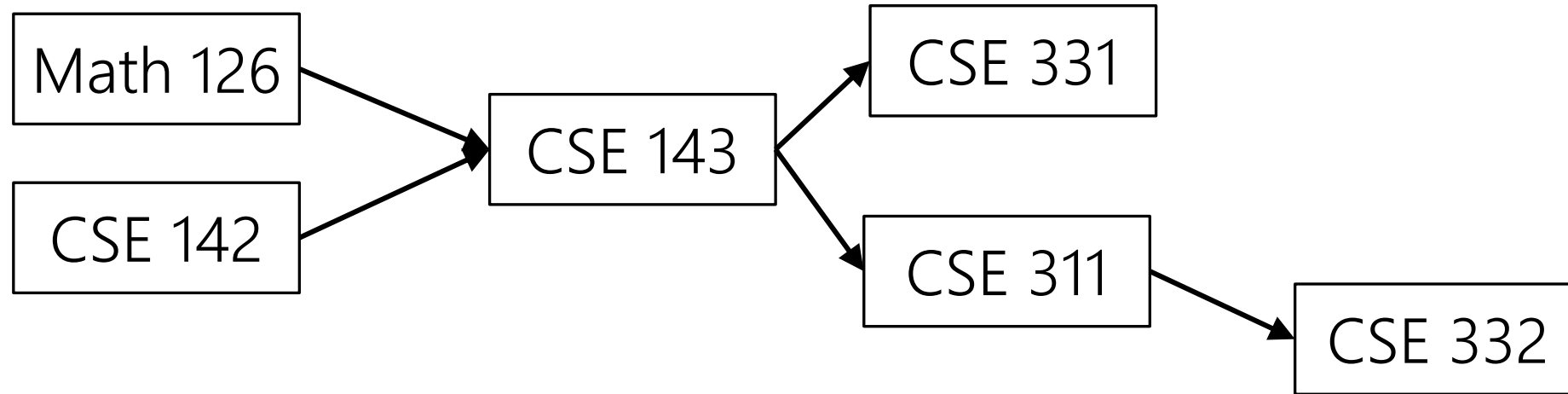**Find:** an ordering of the vertices so all edges go from left to right.

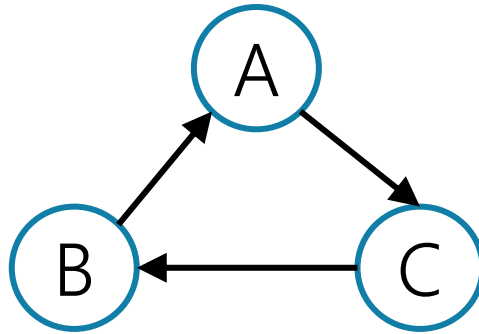Uses:
Compiling multiple files
Graduating.

Topological Ordering

A course prerequisite chart and a possible topological ordering.

Can we always order a graph?

Can you topologically order this graph?

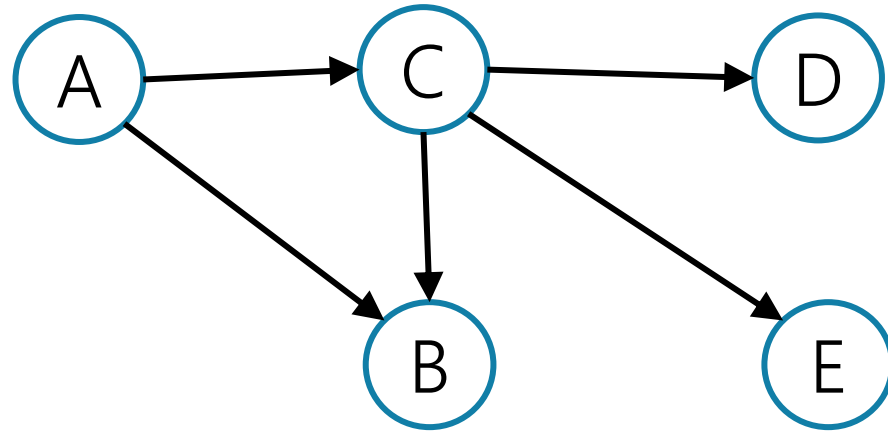**Directed Acyclic Graph (DAG)**

A directed graph without any cycles.

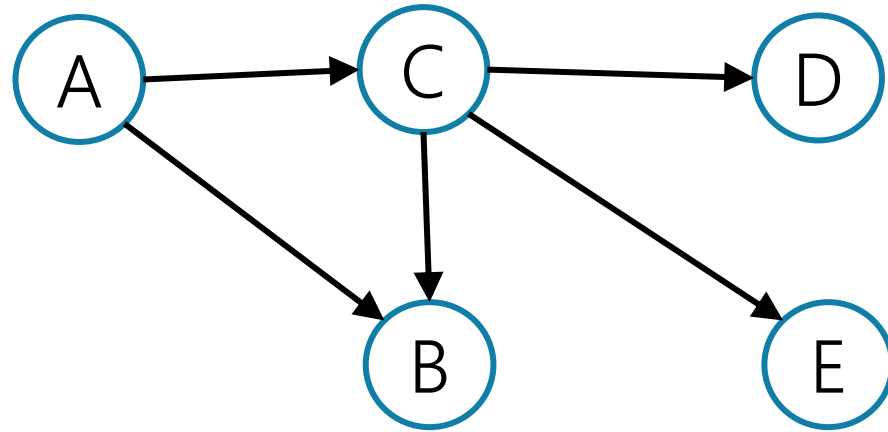A graph has a topological ordering if and only if it is a DAG.

Ordering a DAG

Does this graph have a topological ordering? If so find one.

Ordering a DAG

Does this graph have a topological ordering? If so find one.

If a vertex doesn't have any edges going into it, we can add it to the ordering.
More generally, if the only incoming edges are from vertices already in the ordering, it's safe to add.

How Do We Find a Topological Ordering?

```
TopologicalSort(Graph G, Vertex source)
    count how many incoming edges each vertex has
    Collection toProcess = new Collection()
    foreach(Vertex v in G){
        if(v.edgesRemaining == 0)
            toProcess.insert(v)
    }
    topOrder = new List()
    while(toProcess is not empty){
        u = toProcess.remove()
        topOrder.insert(u)
        foreach(edge (u,v) leaving u){
            v.edgesRemaining--
            if(v.edgesRemaining == 0)
                toProcess.insert(v)
        }
    }
```

What's the running time?

```
TopologicalSort(Graph G, Vertex source)
    count how many incoming edges each vertex has
    Collection toProcess = new Collection()
    foreach(Vertex v in G){
        if(v.edgesRemaining == 0)
            toProcess.insert(v)
    }
    topOrder = new List()
    while(toProcess is not empty){
        u = toProcess.remove()
        topOrder.insert(u)
        foreach(edge (u,v) leaving u){
            v.edgesRemaining--
            if(v.edgesRemaining == 0)
                toProcess.insert(v)
        }
    }
```
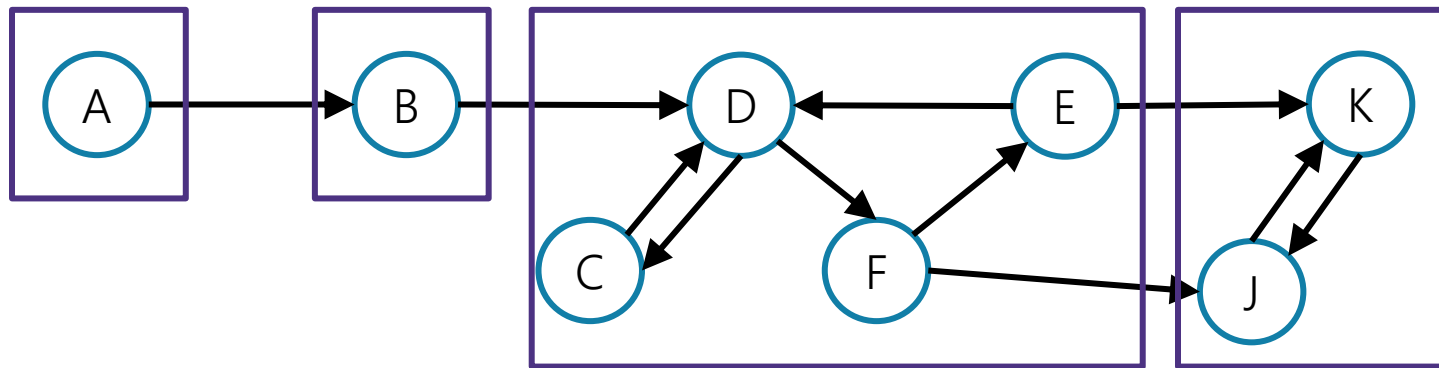
Running Time: $O(|V| + |E|)$

Finding a Topological Ordering

Instead of counting incoming edges, you can actually modify DFS to find you one (think about why).

But the "count incoming edges" is a bit easier to understand (for me ☺)

Problem 2: Find Strongly Connected Components

{A}, {B}, {C,D,E,F}, {J,K}

Strongly Connected Component

A subgraph C such that every pair of vertices in C is connected via some path **in both directions,** and there is no other vertex which is connected to every vertex of C in both directions.

Connectedness Definitions

In an <u>undirected</u> graph, a connected component is a "piece" of the graph: a vertex and everything its connected to via a path.

Equivalently, a subgraph C such that every pair of vertices in C is connected via some path and there is no other vertex which is connected to every vertex of C in both directions.

In a <u>directed</u> graph, you might care about

Weakly connected components (ignore the directions on the edges, if it were undirected, would it be connected?)

Strongly connected (can you get in both directions)

Can you find Strongly Connected Components?

A couple of different ways to use DFS to find strongly connected components.

Wikipedia has the details.

High level: need to keep track of "highest point" in DFS tree you can reach back up to. Similar idea on undirected graphs on HW2.

What do you need to know?
On a small graph, find the SCC by hand

Know that you can modify DFS to find SCCs in $\Theta(V + E)$ time.

Optional: Graph practice

Designing New Algorithms

When you need to design a new algorithm on graphs, whatever you do is probably going to take at least $\Omega(m + n)$ time.

So you can run any $O(m + n)$ algorithm as "preprocessing"

Finding connected components (undirected graphs)

Finding SCCs (directed graphs)

Do a topological sort (DAGs)

Designing New Algorithms

Finding SCCs and topological sort go well together:

From a graph G you can define the "meta-graph" G^{SCC}
(aka "condensation", aka "graph of SCCs")

G^{SCC} has a vertex for every SCC of G

There's an edge from u to v in G^{SCC} if and only if there's an edge in G from a vertex in u to a vertex in v.

Why Find SCCs?

Let's build a new graph out of them! Call it G^{SCC}

- Have a vertex for each of the strongly connected components
- Add an edge from component 1 to component 2 if there is an edge from a vertex inside 1 to one inside 2.
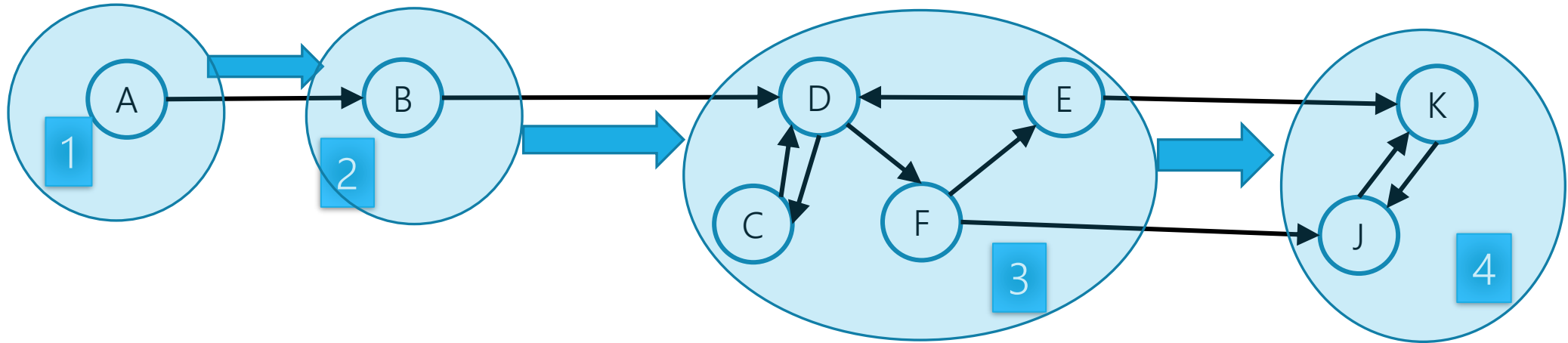
Designing New Graph Algorithms

Not a common task – most graph problems have been asked before.

When you need to do it, Robbie recommends:

Start with a simpler case (topo-sorted DAG, or [strongly] connected graph).

A common pattern:

1. Figuring out what you'd do if the graph is strongly connected

2. Figuring out what you'd do if the graph is a topologically ordered DAG

3. Stitching together those two ideas (using G^{SCC}).

Graph Modeling

But…Most of the time you don't need a new graph algorithm.

What you need is to figure out what graph to make and which graph algorithm to run.

"Graph modeling"

Going from word problem to graph algorithm.

Often finding a clever way to turn your requirements into graph features.

Mix of "standard bag of tricks" and new creativity.

Graph Modeling Process

1. What are your fundamental objects?
-Those will probably become your vertices.

2. How are those objects related?
-Represent those relationships with edges.

3. How is what I'm looking for encoded in the graph?
-Do I need a path from s to t? The shortest path from s to t? A minimum spanning tree? Something else?

4. Do I know how to find what I'm looking for?
-Then run that algorithm/combination of algorithms
-Otherwise go back to step 1 and try again.

Scenario #1

You've made a new social networking app, Convrs. Users on Convrs can have "asymmetric" following (I can follow you, without you following me). You decide to allow people to form multi-user direct messages, but only if people are probably in similar social circles (to avoid spamming).

You'll allow a messaging channel to form only if for every pair of users a,b in the channel: a must follow b or follow someone who follows b or follow someone who follows someone who follows b, or ...
And the same for b to a.

You'd like to be able to quickly check for any new proposed channel whether it meets this condition.

What are the vertices?

What are the edges?

What are we looking for?

What do we run?

Scenario #1

You've made a new social networking app, Convrs. Users on Convrs can have "asymmetric" following (I can follow you, without you following me). You decide to allow people to form multi-user direct messages, but only if people are probably in similar social circles (to avoid spamming).

You'll allow a messaging channel to form only if for every pair of users a,b in the channel: a must follow b or follow someone who follows b or follow someone who follows someone who follows b, or …
And the same for b to a.

You'd like to be able to quickly check for any new proposed channel whether it meets this condition.

What are the vertices?

Users

What are the edges?

Directed – from u to v if u follows v

What are we looking for?

If everyone in the channel is in the same SCC.

What do we run?

Find SCCs, to test a new channel, make sure all are in same component.

Scenario #2

Sports fans often use the "transitive law" to predict sports outcomes.

In general, if you think A is better than B, and B is also better than C, then you expect that A is better than C.

Teams don't all play each other – from data of games that have been played, determine if the "transitive law" is realistic, or misleading about at least one outcome.

What are the vertices?

What are the edges?

What are we looking for?

What do we run?

Scenario #2

Sports fans often use the "transitive law" to predict sports outcomes -- .
In general, if you think A is better than B, and B is also better than C, then you expect that A is better than C.

Teams don't all play each other – from data of games that have been played, determine if the "transitive law" is realistic, or misleading about at least one outcome.

What are the vertices?

Teams

What are the edges?

Directed – Edge from u to v if u beat v.

What are we looking for?

A cycle would say it's not realistic.
OR a topological sort would say it is.

What do we run?

Cycle-detection DFS.
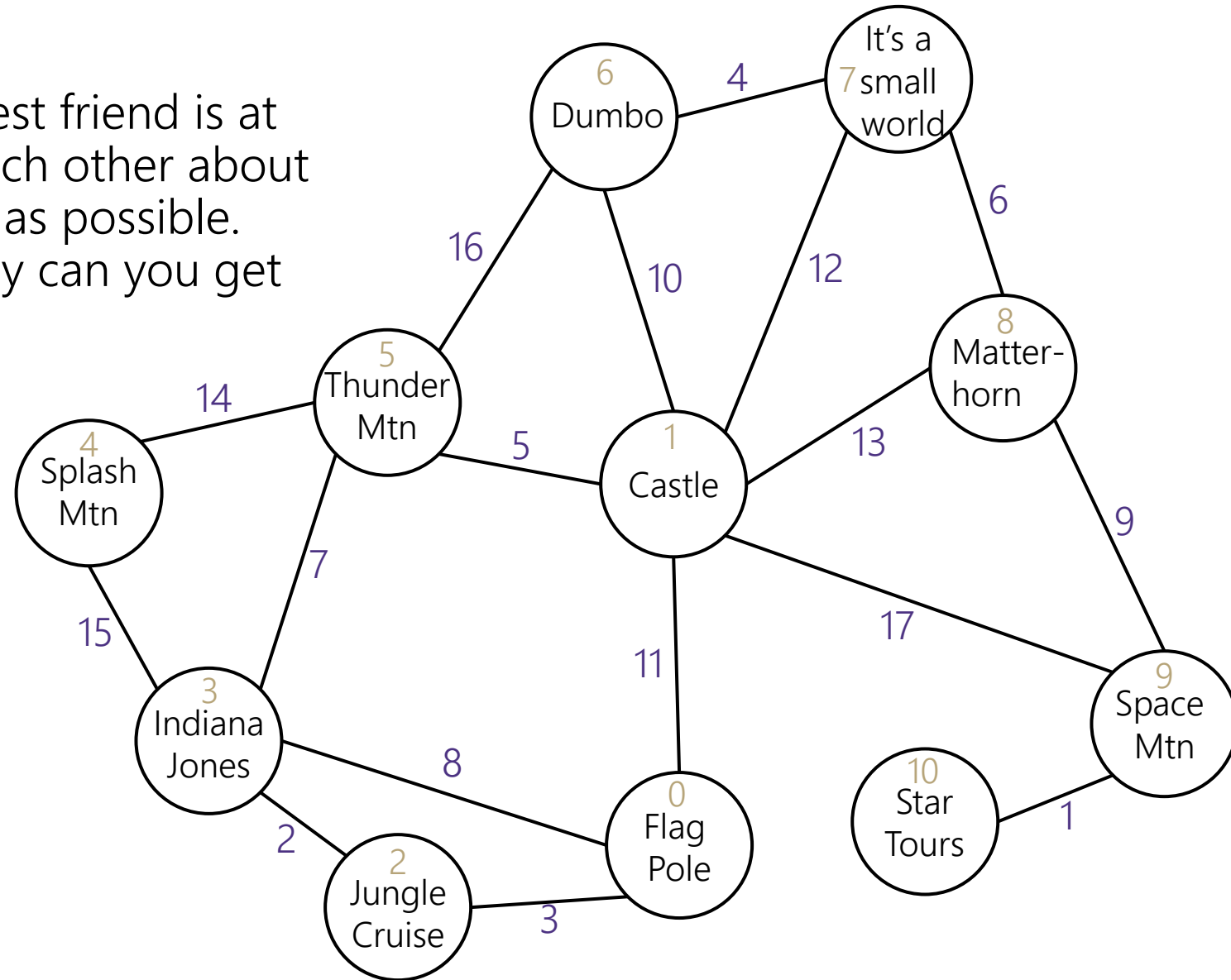a topological sort algorithm (with error detection)

Scenario #3

You are at Splash Mountain. Your best friend is at Space Mountain. You have to tell each other about your experiences in person as soon as possible. Where do you meet and how quickly can you get there?

What are the vertices?

What are the edges?

What are we looking for?

What do we run?

Scenario #3

You are at Splash Mountain. Your best friend is at Space Mountain. You have to tell each other about your experiences in person as soon as possible. Where do you meet and how quickly can you get there?

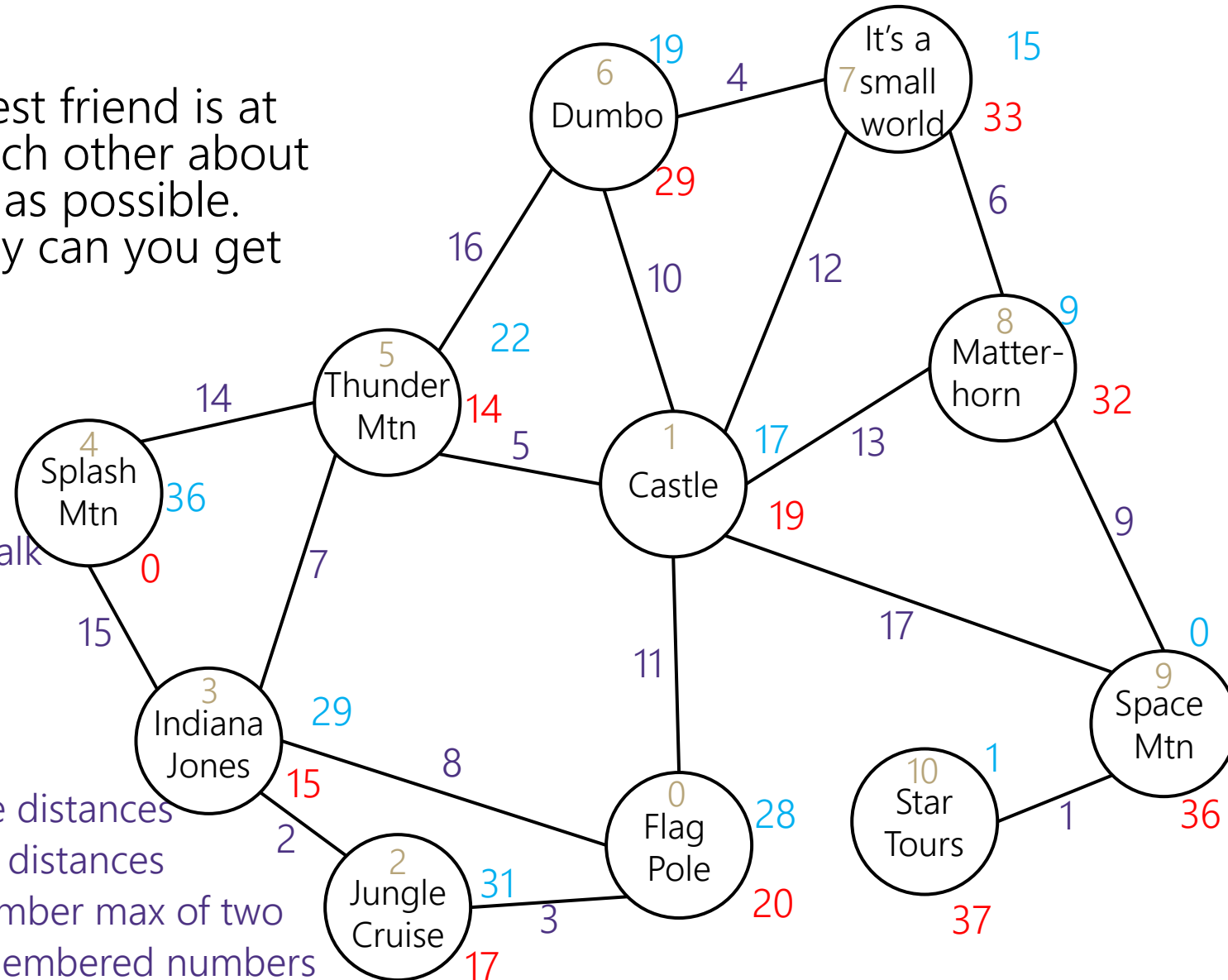**What are the vertices?**
Rides

What are the edges?
Walkways with how long it would take to walk

What are we looking for?
- The "midpoint"

What do we run?
- Run Dijkstra's from Splash Mountain, store distances
- Run Dijkstra's from Space Mountain, store distances
- Iterate over vertices, for each vertex remember max of two
- Iterate over vertices, find minimum of remembered numbers

Scenario #4

You're a Disneyland employee, working the front of the Splash Mountain line. Suddenly, the crowd-control gates fall over and the line degrades into an unordered mass of people.

Sometimes you can tell who was in line before who; for other groups you aren't quite sure. You need to restore the line, while ensuring if you **knew** A came before B before the incident, they will still be in the right order afterward.

What are the vertices?
People

What are the edges?
Edges are directed, have an edge from X to Y if you know X came before Y.

What are we looking for?
- A total ordering consistent with all the ordering we do know.

What do we run?
- Topological Sort!