Graph Modeling and start of Greedy

CSE 421 Winter 23
Lecture 5

Announcements

Monday is a holiday!

Office hours will be shifted around a bit (see the calendar)

No lecture Monday

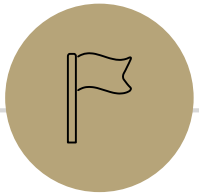
When proving things on the homework:

This isn't 311. Two big differences:

In 311 we want to see you know how proofs work; here we want to know you know how the problem works.

In 311 the statement was always true. Here, you're proving your own submission correct. You might be trying to prove something false!

The proof is a chance for you to check your own work!



Graph Modeling

Problem 1: Ordering Dependencies

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

We can only do things one at a time, 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.

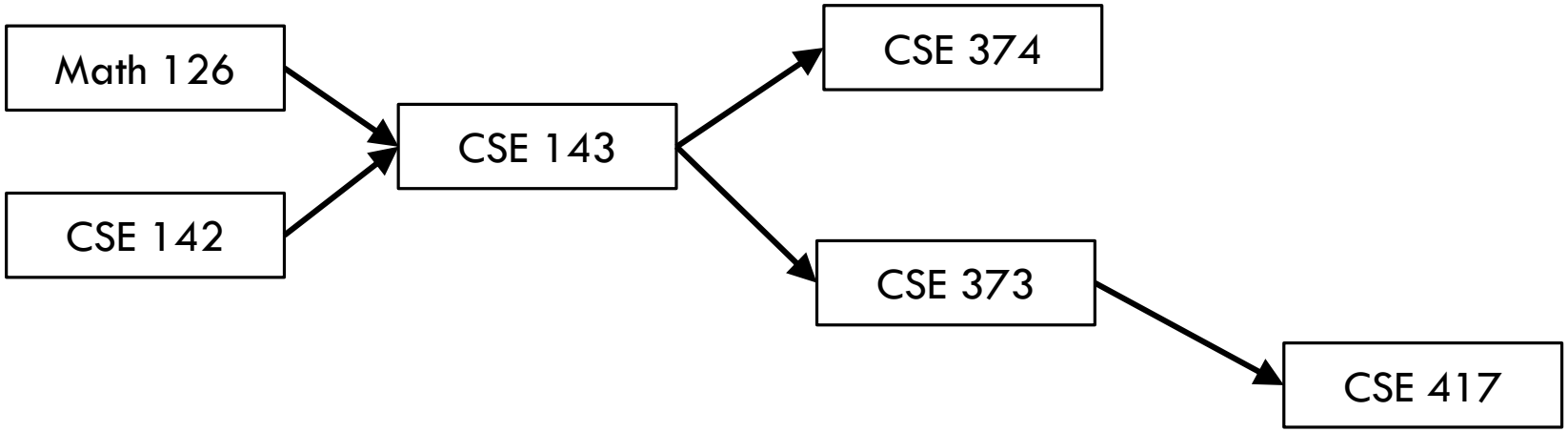
Uses:

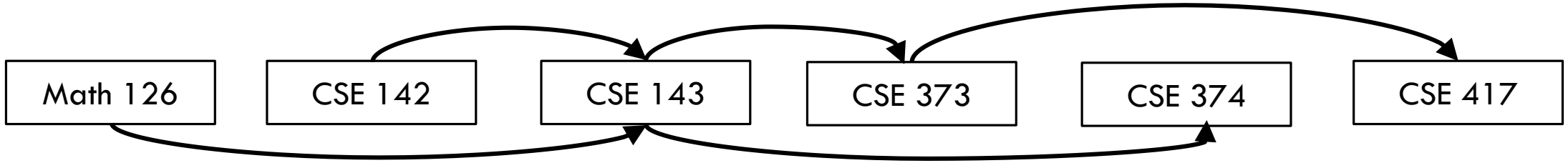
Compiling multiple files

Graduating

Topological Ordering

A course prerequisite chart and a possible topological ordering.



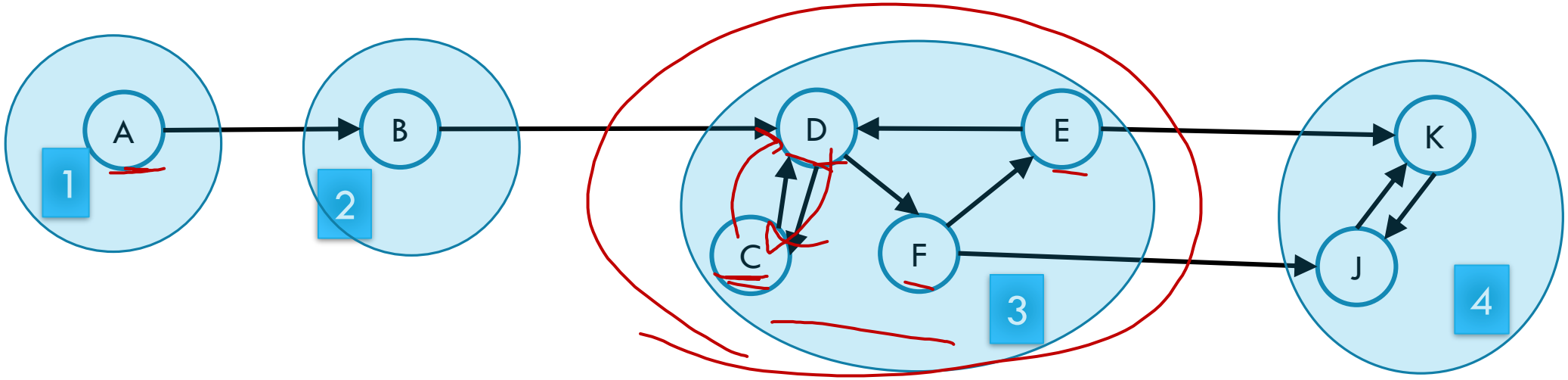


Problem 2

Given a graph, find its strongly connected components

Strongly Connected Component

A set of vertices 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.



How do these work?

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.

You can use this algorithm as a library function!

We also listed all the ones from 332 on this webpage.

- Topological sort

  You saw an algorithm in 332

  Important thing: both run in $\Theta(m+n)$ time.

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

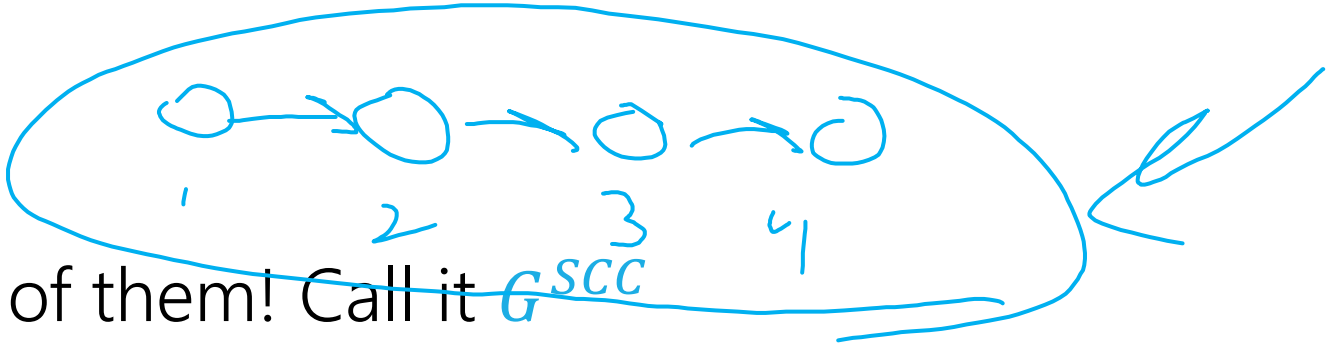
Finding SCCs and topological sort go well together:

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.

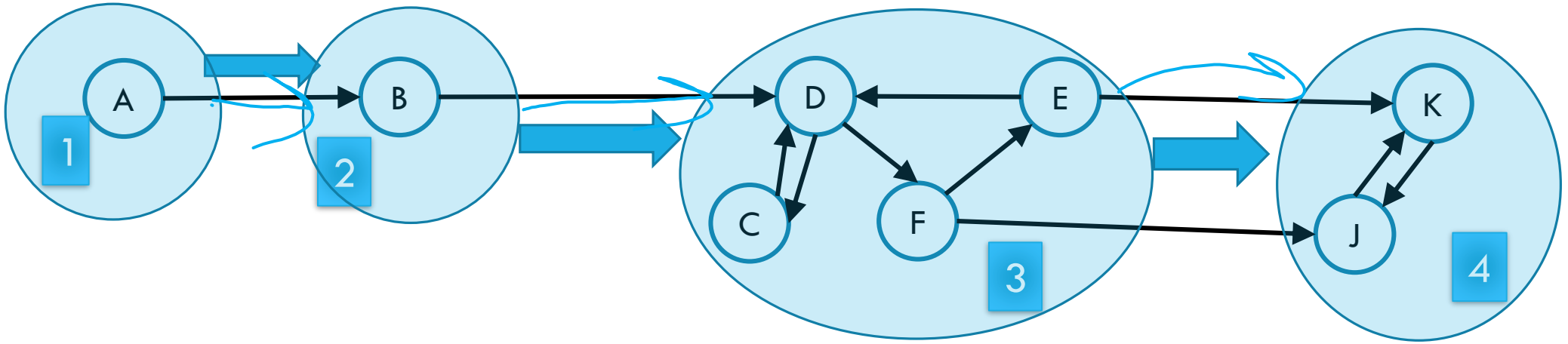
Why Find SCCs?



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).

One way to think about the 2-SAT problem on HW2

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}).

Problem Solving Suggestions

- Read the problem carefully.
  - Are there any technical terms in the question? Any formulas?
  - What kind of object will you get as input? What type is your output?
- Do you understand it? Write sample inputs and outputs
  - We'll often give you samples, but it helps to add your own.
- Now start thinking about solutions
  - On those examples, how would you get the solution?
  - Does this remind you of any algorithms from class?
  - Can you think of a new idea?
  - It's ok to start with slow solutions and try to speed them up!
  - Try the graph modeling process.

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.

— What are the possible 'states'

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.

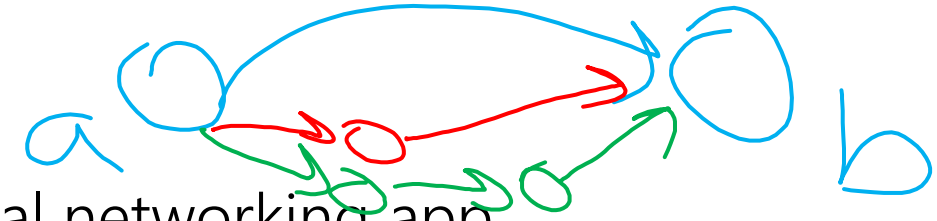
What are the vertices?

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

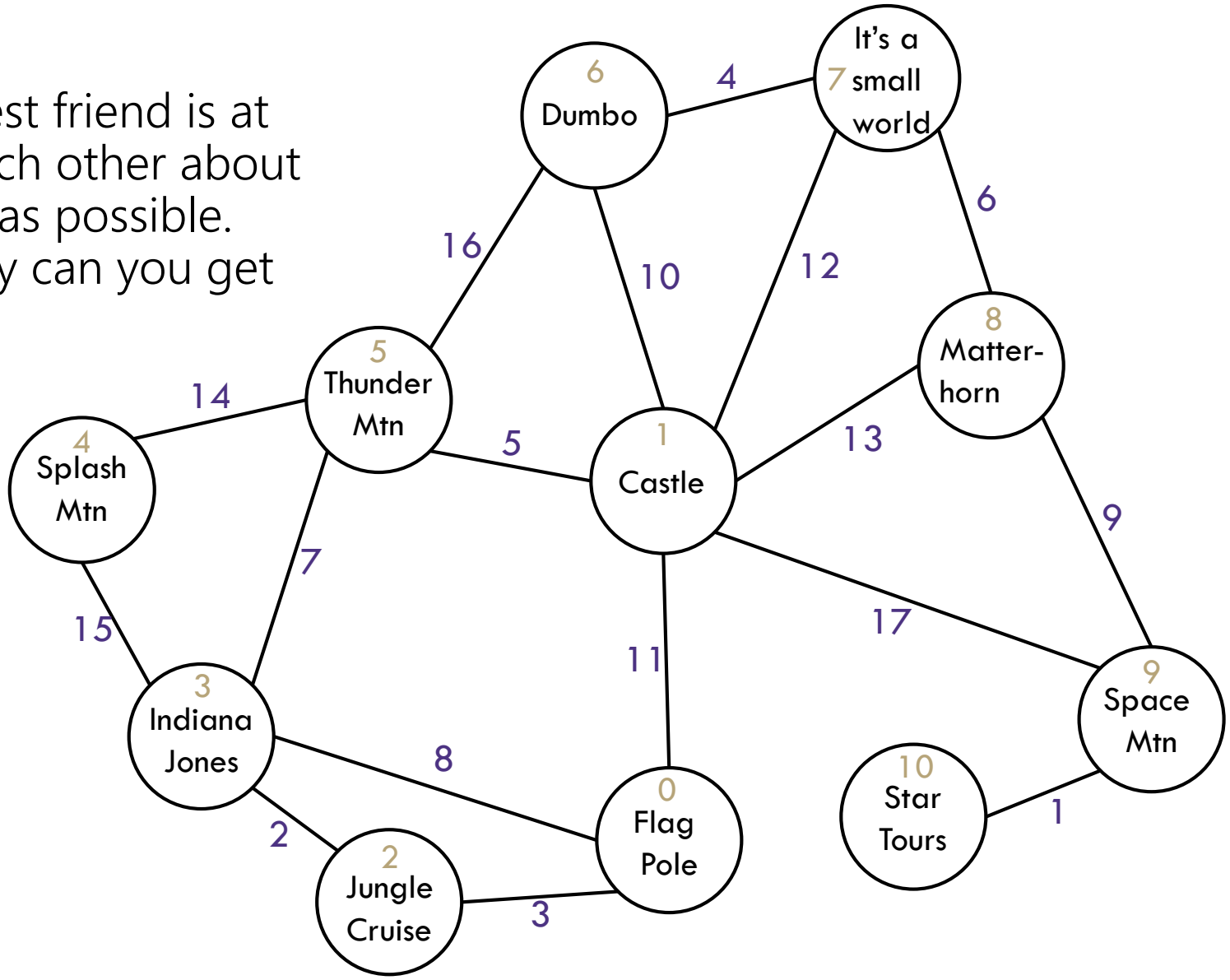
You are at Splash Mountain. Your best friend is at Space Mountain. You have to tell each other about your experiences in person as soon as possible. Where do you meet and how quickly can you get there?

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?

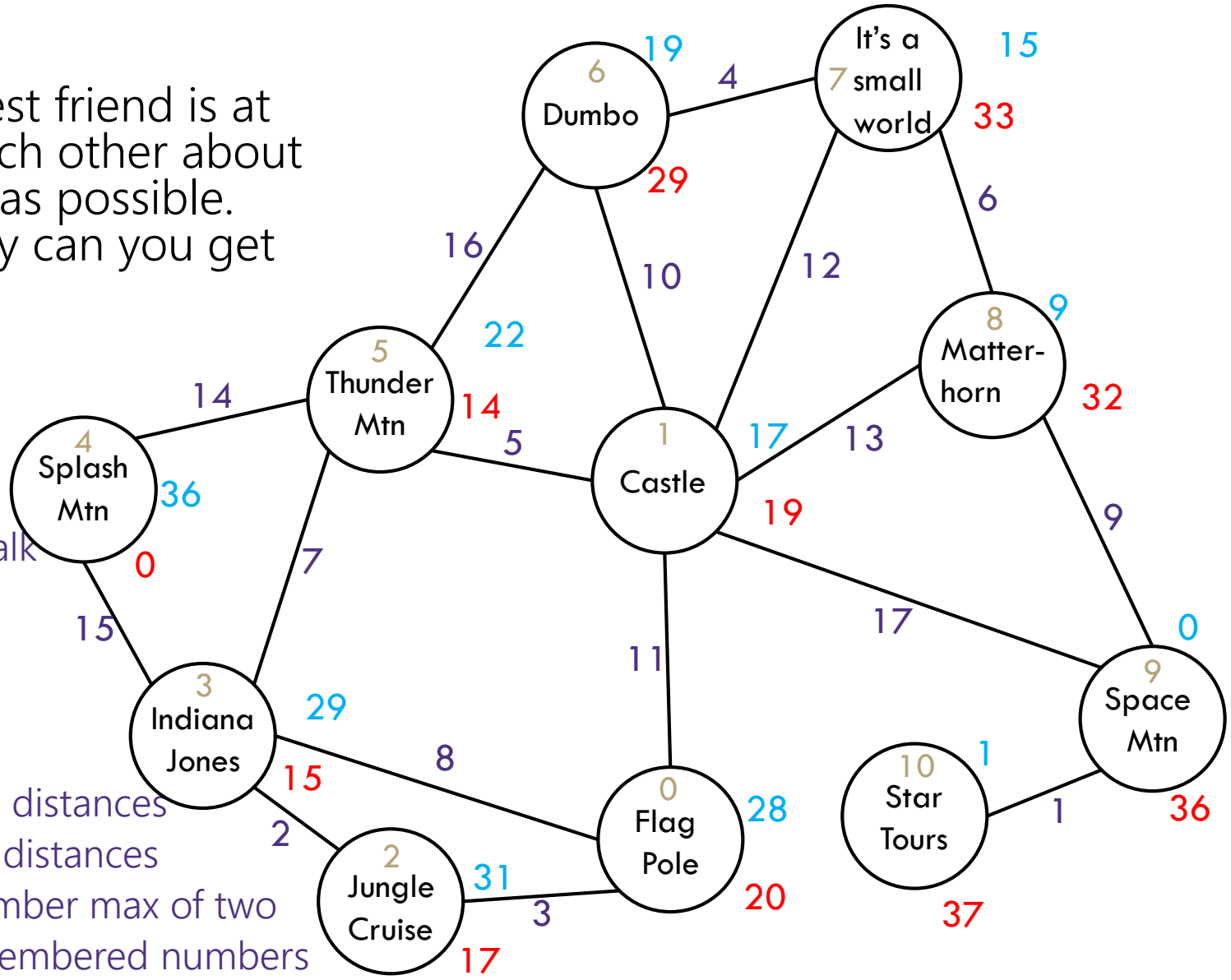
Walkways with how long it would take to walk

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?

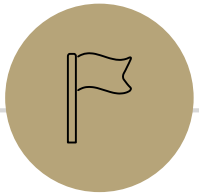
Edges are directed, have an edge from X to Y if you know X came before Y.

What are we looking for?

- A total ordering consistent with all the ordering we do know.

What do we run?

- Topological Sort!



Greedy Algorithms

Our Next Topic

Another abrupt change of topic.

In fact, the whole course is a sequence of seemingly-abrupt topic changes…

We're giving you a list of tools – a list of common ways of thinking when approaching a new problem.

Think of each week as a new tool in your toolbox.

Greedy Algorithms

What's a greedy algorithm?

An algorithm that builds a solution by:

Considering objects one at a time, in some **order**.

Using a **simple rule** to decide on each object.

Never goes back and changes its mind.

Greedily do what looks best for you right here, right now.

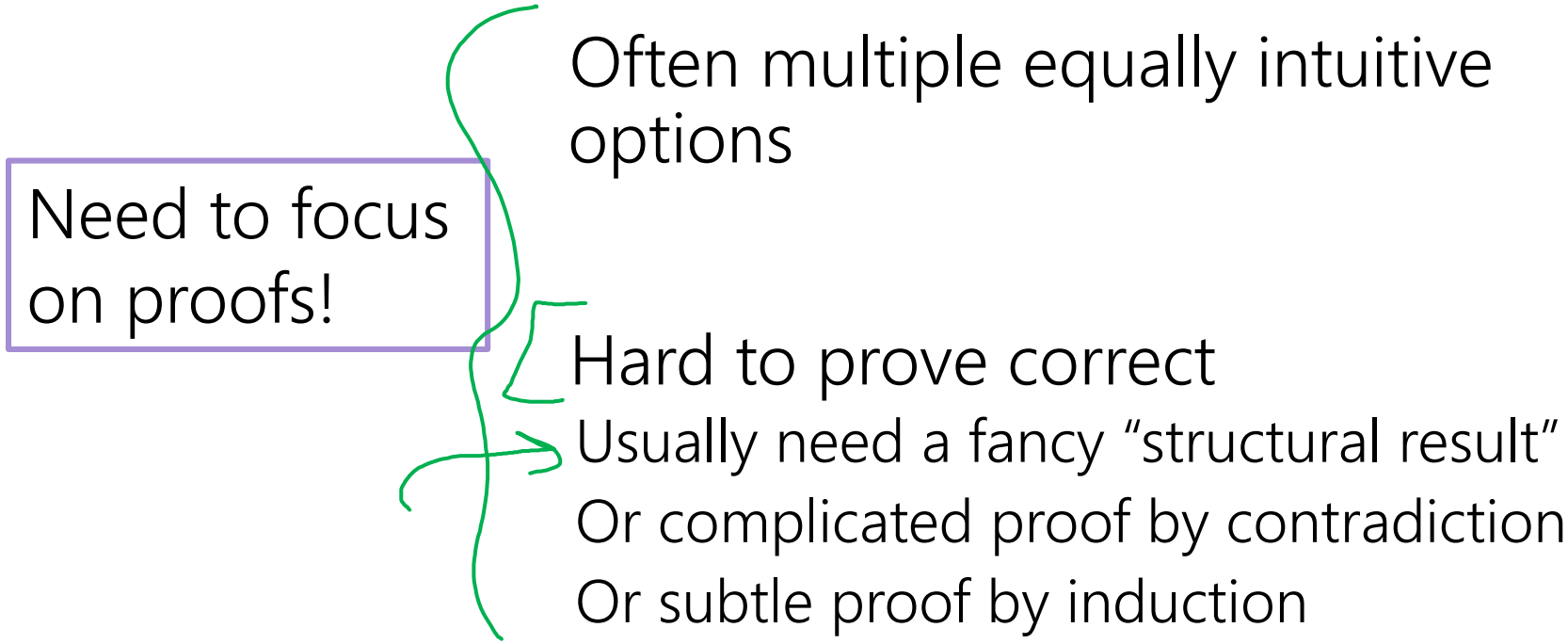
Greedy Algorithms

PROS

Simple

CONS

Rarely correct



Your Takeaways

Greedy algorithms are great *when they work*.

But it's hard to tell when they work – the proofs are subtle.

And you can often invent 2-3 different greedy algorithms; it's rare that 1 gives you the best answer, extremely rare that all would.

So you have to be EXTREMELY careful.

But they are very often useful when you need an answer that is very good, but not optimal (more on Friday).

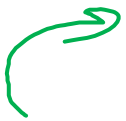
Three Common Proof Styles

"Structural result" – the best solution **must** look like this, and the algorithm produces something that looks like this.

Greedy stays ahead – at every step of the algorithm, the greedy algorithm is at least as good as anything else could be.

Exchange – Contradiction proof, suppose we swapped in an element from the (hypothetical) "better" solution.

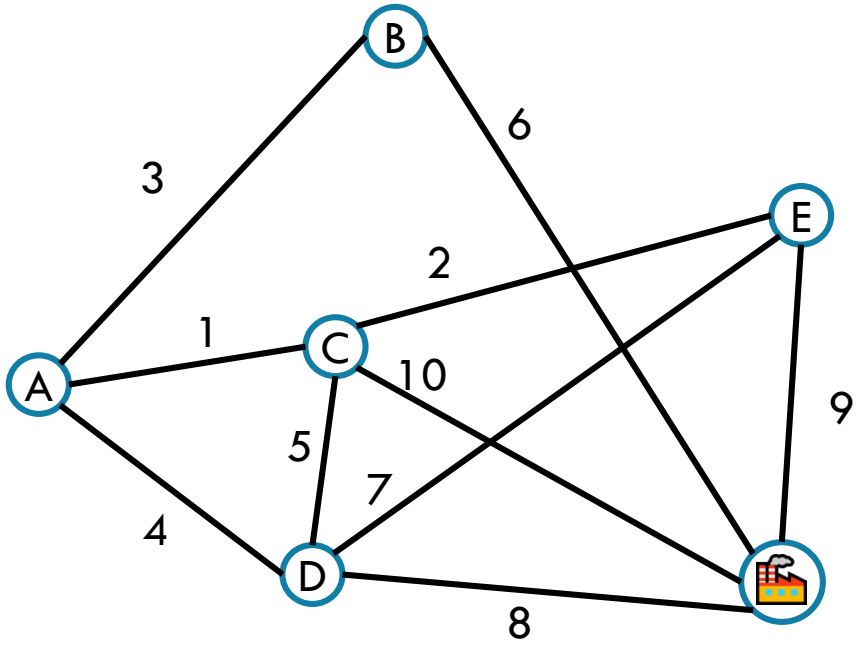
Where to start? With some greedy algorithms you've already seen.



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 cities, 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**.

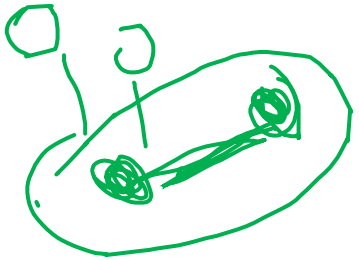
The minimum weight set of edges that meet those conditions.

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.

Greedy MST algorithms



You've seen two algorithms for MSTs

Kruskal's Algorithm:

Order: Sort the edges in increasing weight order

Rule: If connect new vertices (doesn't form a cycle), add the edge.

Prim's Algorithm:

Order: lightest weight edge that adds a new vertex to our current component

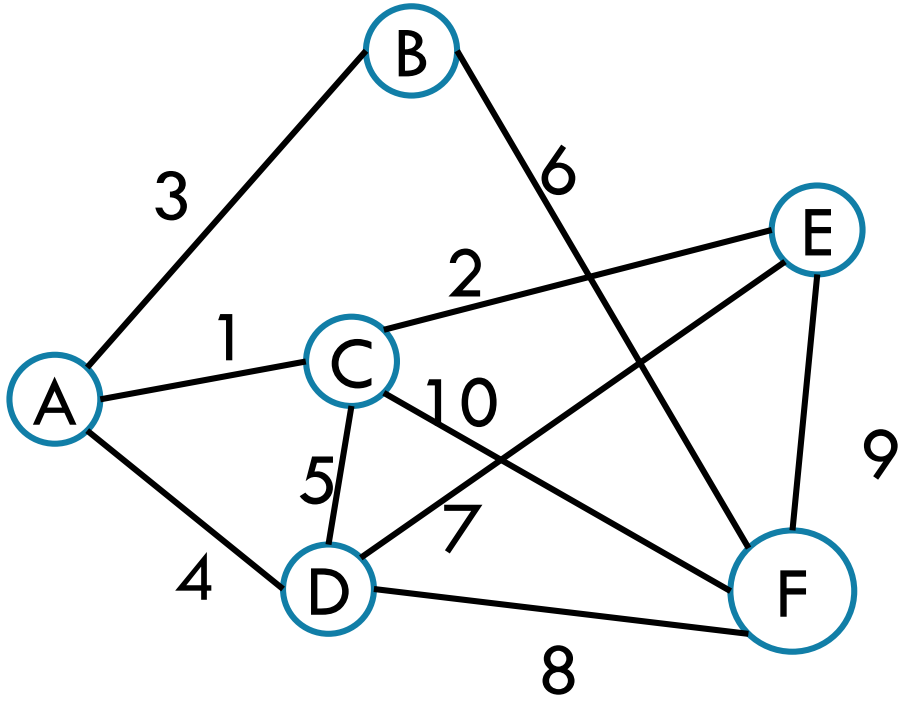
Rule: Just add it!

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
        }
    }
```

Try It Out

```
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
                }
        }
```

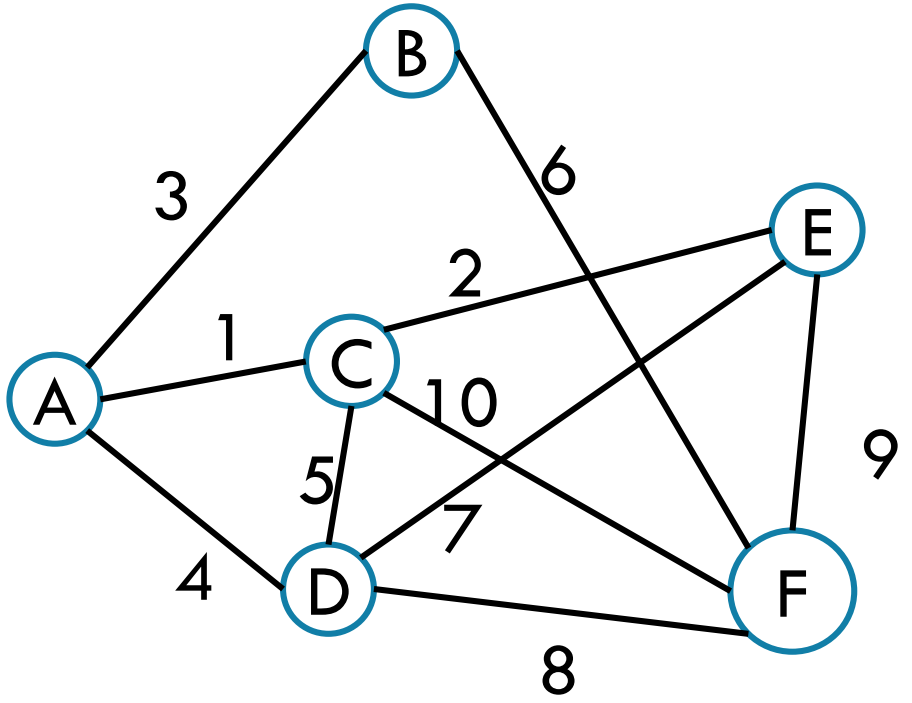


Edge	Include?	Reason
(A,C)		
(C,E)		
(A,B)		
(A,D)		
(C,D)		

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 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
                }
        }
```



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

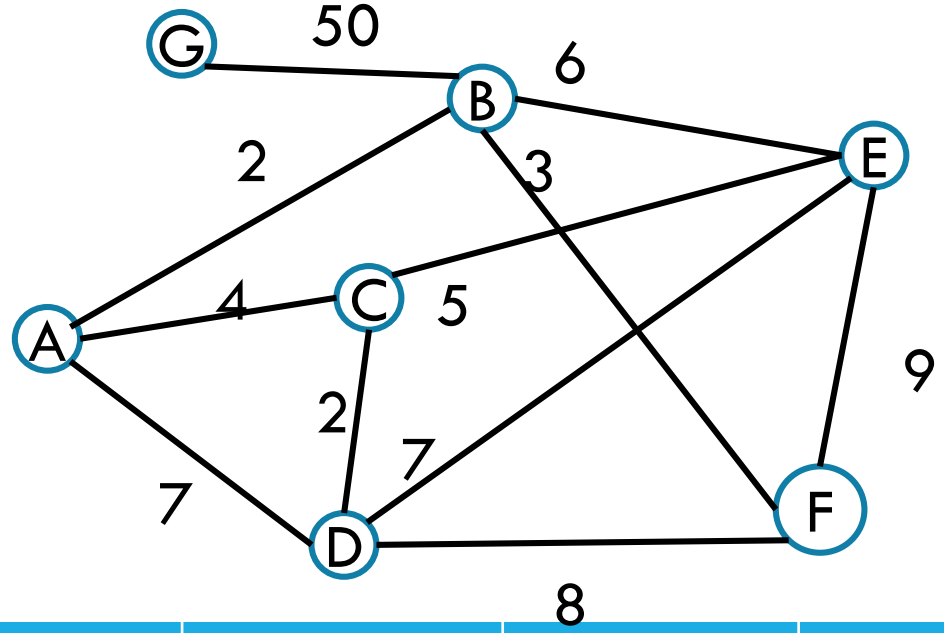
Edge (cont.)	Inc?	Reason
(B,F)	Yes	
(D,E)	No	Cycle A,C,E,D,A
(D,F)	No	Cycle A,D,F,B,A
(E,F)	No	Cycle A,C,E,F,D,A
(C,F)	No	Cycle C,A,B,F,C

Prim's Algorithm

```
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

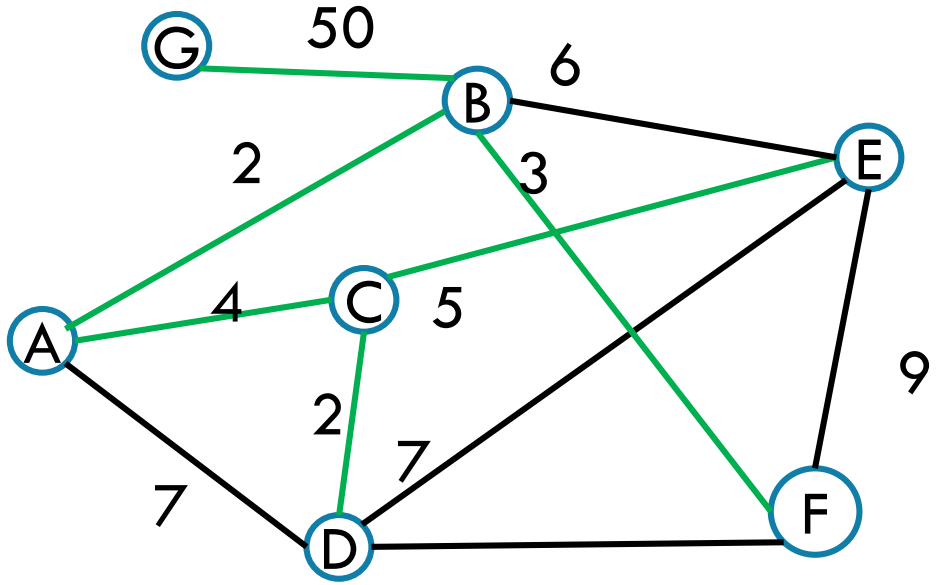
```
PrimMST(Graph G)
   initialize costToAdd to ∞
   mark source as costToAdd 0
   mark all vertices unprocessed
   mark source as processed
   foreach(edge (source, v) ) {
         v.costToAdd = weight(source,v)
         v.bestEdge = (source,v)
   }
   while(there are unprocessed vertices) {
            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

Correctness

You're already familiar with the algorithms.

We'll use this problem to practice the proof techniques.

We'll do both **structural** and **exchange**

Structural Proof

For simplicity – assume all edge weights are distinct and that there is only one minimum spanning tree.

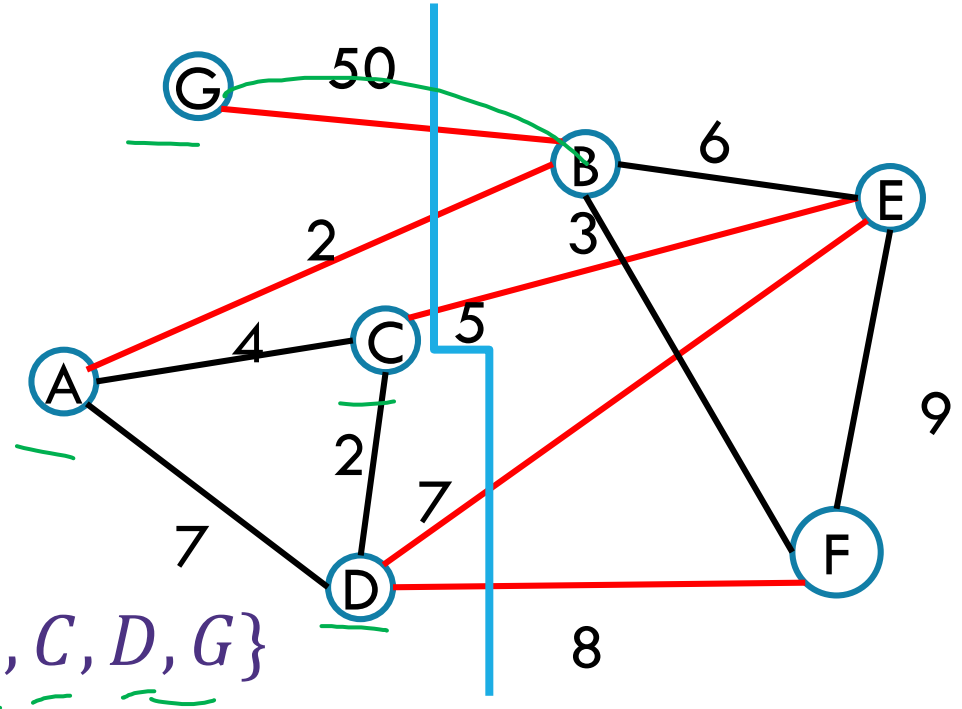
"Structural result" – the best solution **must** look like this, and the algorithm produces something that looks like this.

Example: every spanning tree has $n - 1$ edges.
So we better have our algorithm produce $n - 1$ edges.

Is that enough? No! Lots of different trees (including non minimum ones) have $n - 1$ edges. Need to say which edges are in the tree.

Safe Edge

A "cut" $(S, V \setminus S)$ is a split of the vertices into a subset S and the remaining vertices $V \setminus S$.

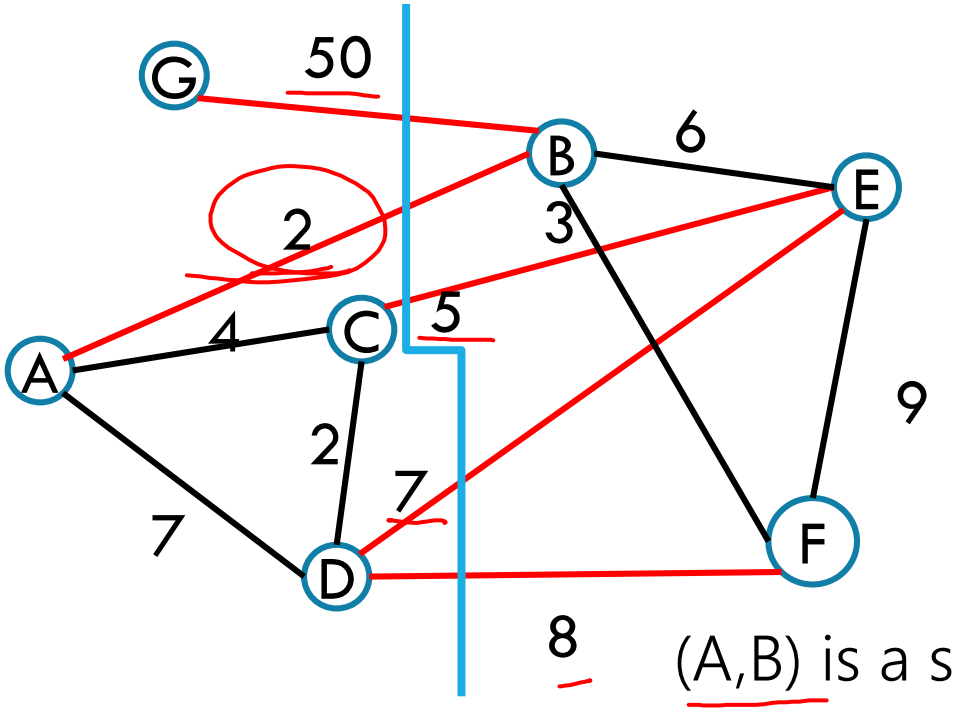


$S = \{A, C, D, G\}$

Edges in red "span" or "cross" the cut (go from S to $V \setminus S$).

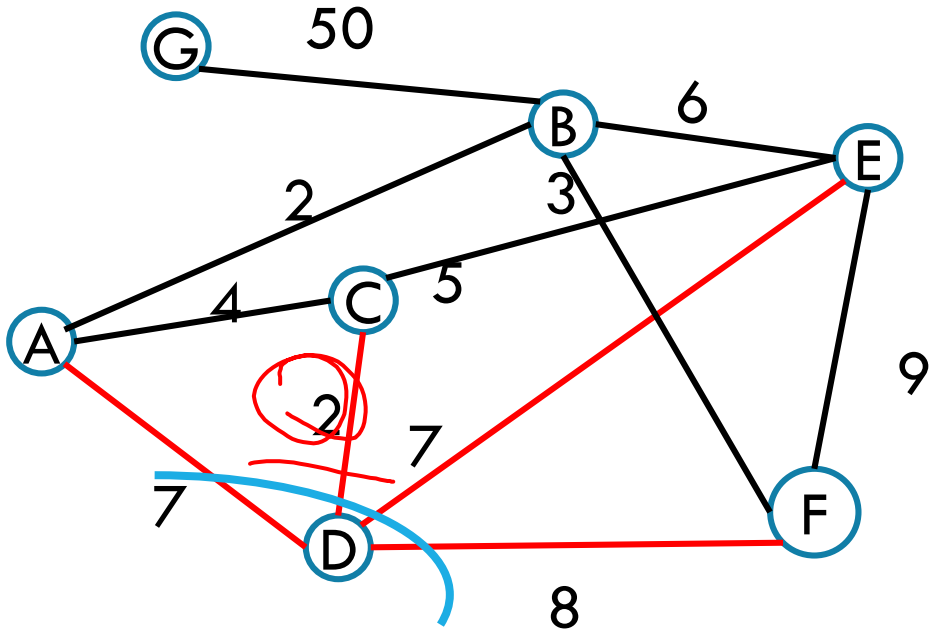
Safe Edge

Call an edge, e, a "**safe edge**" if there is some cut $(S, V \setminus S)$ where e is the minimum edge spanning that cut



(A,B) is a safe edge

(C,D) is a safe edge

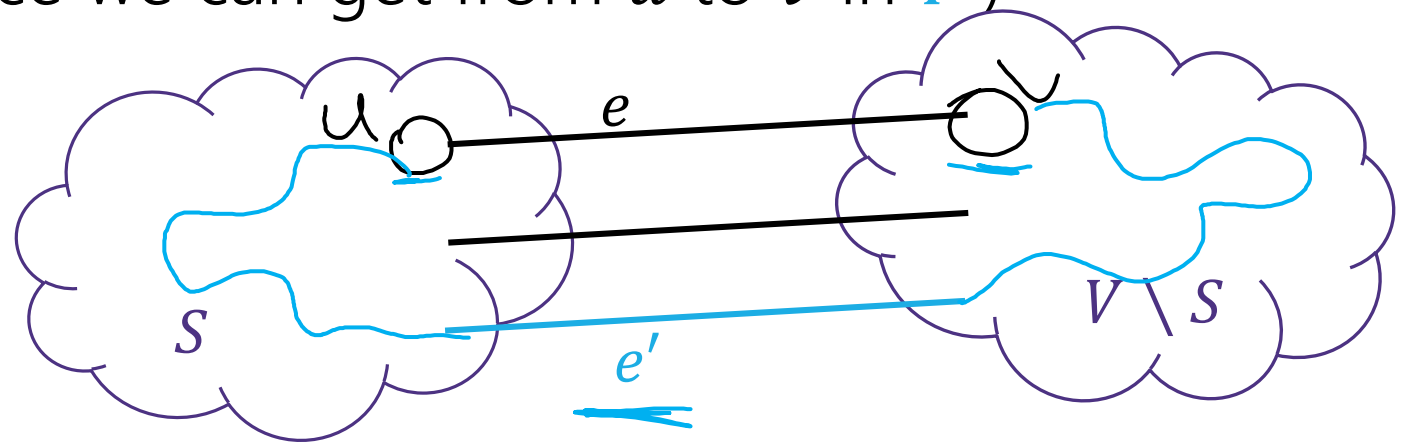


MSTs and Safe Edges

Claim: Every safe edge is in the MST.

Proof: Suppose, for the sake of contradiction, that $e = (u, v)$ is a safe edge, but not in the MST.

Let $(S, V \setminus S)$ be a cut where e is the minimum edge spanning $(S, V \setminus S)$. Let T' be the MST. The MST has (at least one) an edge e' that crosses the cut (since we can get from u to v in T')



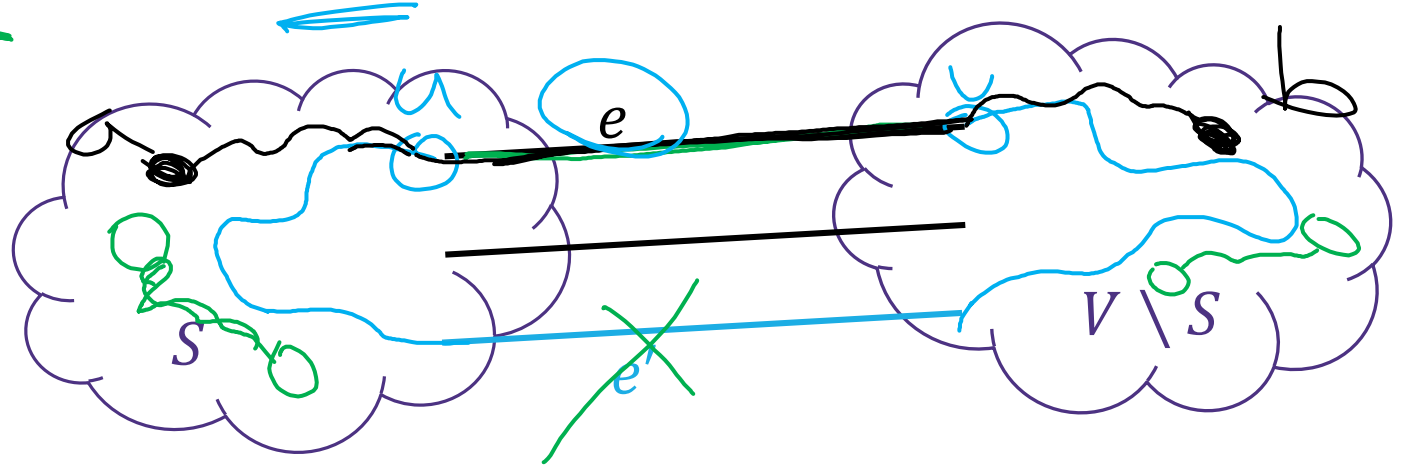
Claim: Every safe edge is in the MST.

MSTs and Safe Edges

Add $e=(u,v)$ to T'.

The new graph has a cycle including both e and e', The cycle exists because u and v were connected to each other in T' (since it was a spanning tree).

Consider T'', which is T' with e added and e' removed.



Claim: Every safe edge is in the MST.

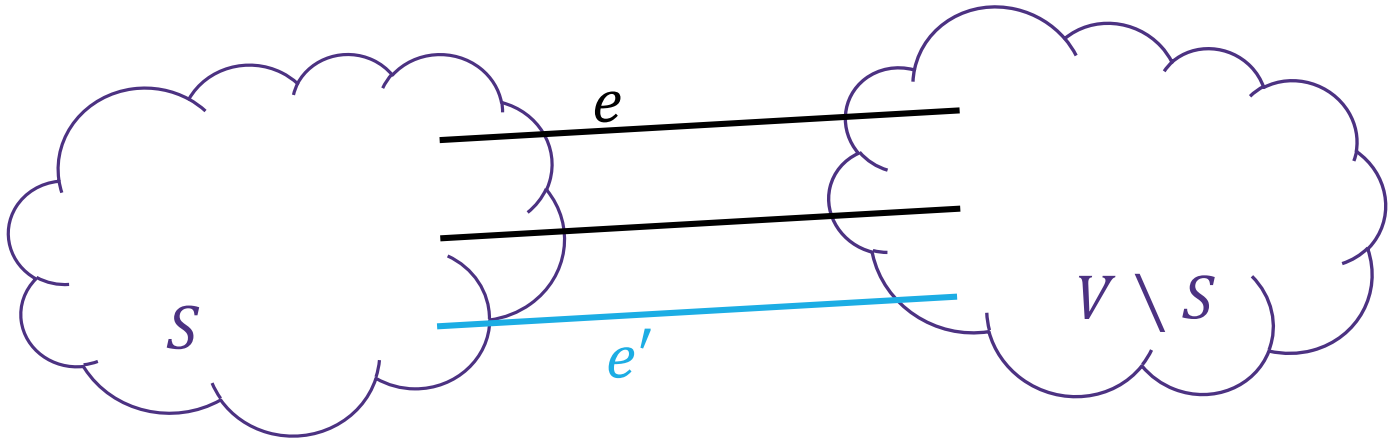
MSTs and Safe Edges

Consider T'', which is T' with e added and e' removed.

T'' spans: if the path from x to y in T' didn't use e' it still exists. If it did use e', follow along the path to e', along the cycle through e to the other side.

And it's a tree (it has $n - 1$ edges).

What's its weight? Less than T'; e was the lightest edge spanning $(S, V \setminus S)$. That's a contradiction! T' was the minimum spanning tree.

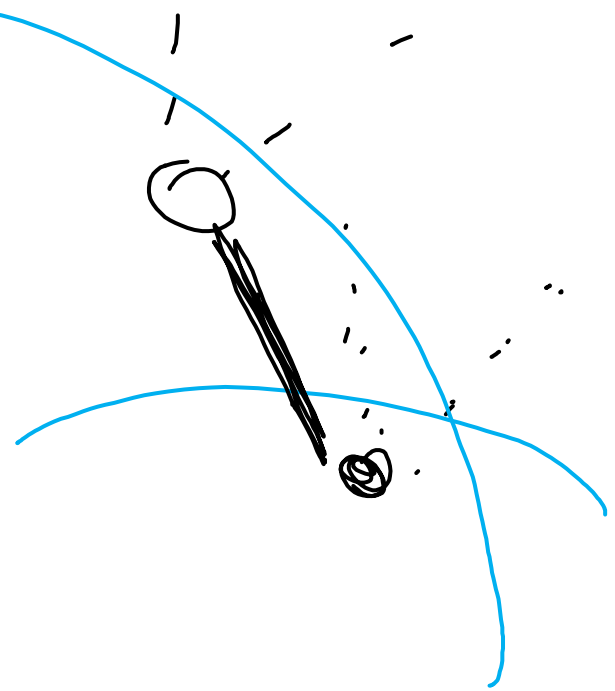


Structural Result

That's the structural result.

e is a "**safe edge**" if there is some cut $(S, V \setminus S)$ where e is the minimum edge spanning that cut.

Theorem: Every safe edge is in the MST.



So what? The goal is to analyze an algorithm!

Let's start with Prim's!

Prim's only adds safe edges

Claim: Prim's only adds safe edges.

When we add an edge, we add the minimum weight one among those that span from the already connected vertices to the not-yet-connected ones.

That's a cut! And that cut shows the edge we added is safe!

Are we done? Do we know Prim's Algorithm is correct now?

Not yet! What if there are still other edges to add!

Why Aren't We Done?

Imagine we define an "ultra-safe" edge as an edge that is lighter than every other edge in the graph.

With a similar proof to our last one, you can show every "ultra-safe" edge is in the MST.

Now imagine Brim's Algorithm: sort the edges by increasing weight, add the first edge in the sorted list.

Brim's algorithm only adds ultra-safe edges!

But that's not a correct MST algorithm!!!

Prim's only adds safe edges

Claim: Prim's only adds safe edges.

When we add an edge, we add the minimum weight one among those that span from the already connected vertices to the not-yet-connected ones.

That's a cut! And that cut shows the edge we added is safe!

So we only add safe edges…

…and we produce an acyclic, connected, spanning graph (since each edge must connect new vertices, we can't create a cycle; the loop ends only when the graph is connected). So we have a (full) spanning tree.