More Graph Algorithms

Data Structures and Algorithms

Announcements

Talk on technical interviews **today!**
Gugenheim 220 at 1:10 PM.

Para Exercise feedback soon.

P2 Feedback (hopefully) Saturday.

Announcements

Please fill out course evaluations.

They'll be helpful for me.

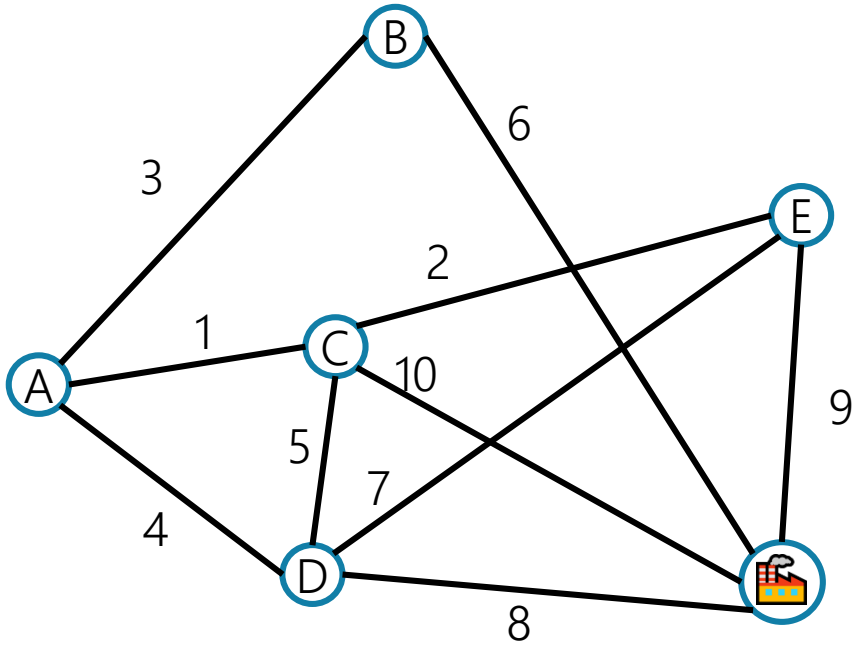
They'll also be helpful for CSE/future students.

Balancing preparing you for future courses and not overworking you is hard.

Tell us the parts of the quarter that were particular pain points.

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.

Prim's Algorithm

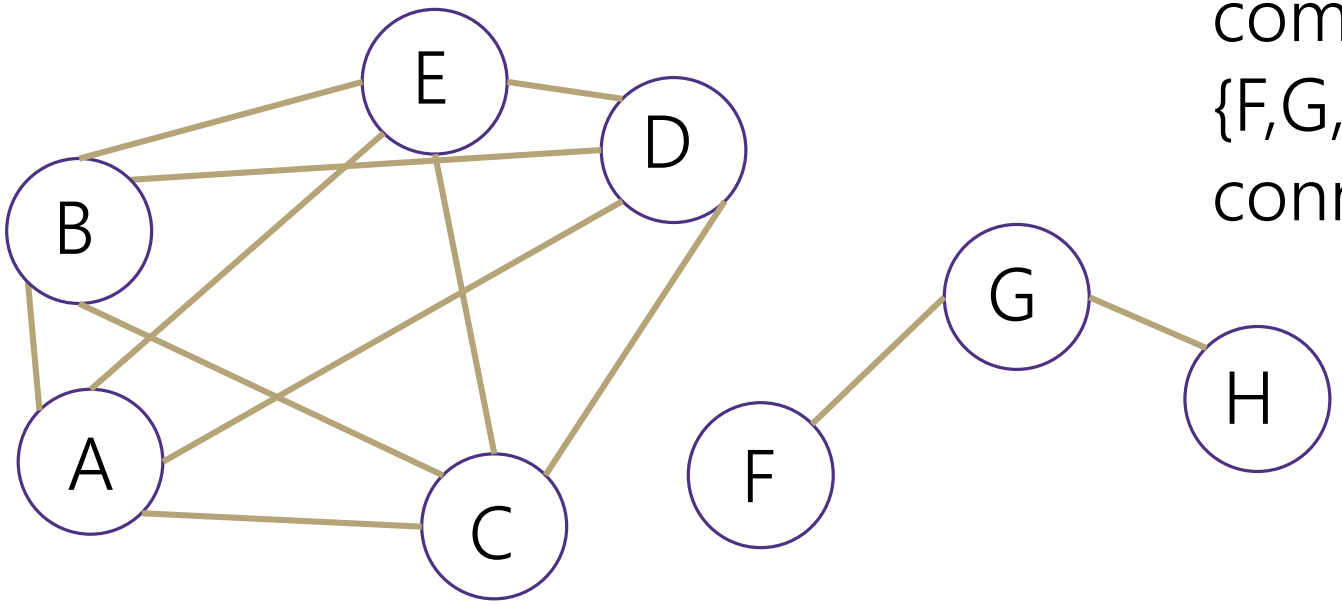
```
PrimMST(Graph G)
   initialize distances to ∞
   mark source as distance 0
   mark all vertices unprocessed
   foreach(edge (source, v) )
      v.dist = weight(source,v)
      v.bestEdge = (source,v)
   while(there are unprocessed vertices){
      let u be the closest unprocessed vertex
      add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

Running time:
$O(|V| \log |V| + |E| \log |V|)$

Connected Component

Connected Component – A set of vertices such that

- There is a path between every pair of vertices
- If you added any other vertex to the set, there would not be a path between every pair of vertices.



{A,B,C,D} is not a connected component (could add E)
{F,G,H} and {A,B,C,D,E} are connected components.

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
        }
    }
```

How do we check the if statement?

BFS would lead to an overall running time of $O(|E|(|V| + |E|))$

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()	$O(\log^* n)$	$\Theta(\log n)$
Find()	$O(\log^* n)$	$\Theta(\log n)$

$\log^* n$

$\log^* n$

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.

How do you see if two vertices are in the same component?

What happens when you add an edge?

Using Union-Find

Have each disjoint set represent a connected component.

How do you see if two vertices are in the same component?

- `find()` on both
- The representatives are the same if and only if the components are the same.

What happens when you add an edge?

- Union the two components

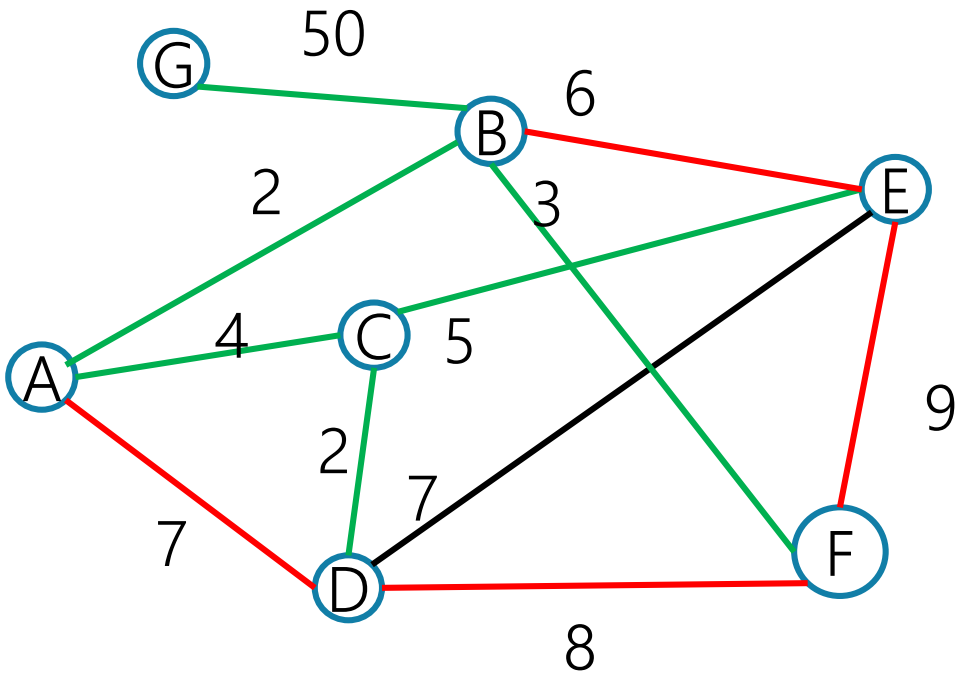
Now With Disjoint Sets

```
KruskalMST(Graph G)
    foreach(Vertex v){ makeSet(v) }
   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)
     }
   }
```

Running Time?
Dominated by sorting.
$O(|E|\log|\mathrm{E}|)$.

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.

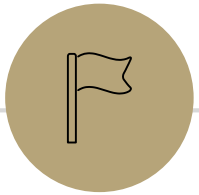
How would we describe the graph Kruskal's builds?

It's not a tree until the end.

It's a forest!

A forest is any undirected and acyclic graph

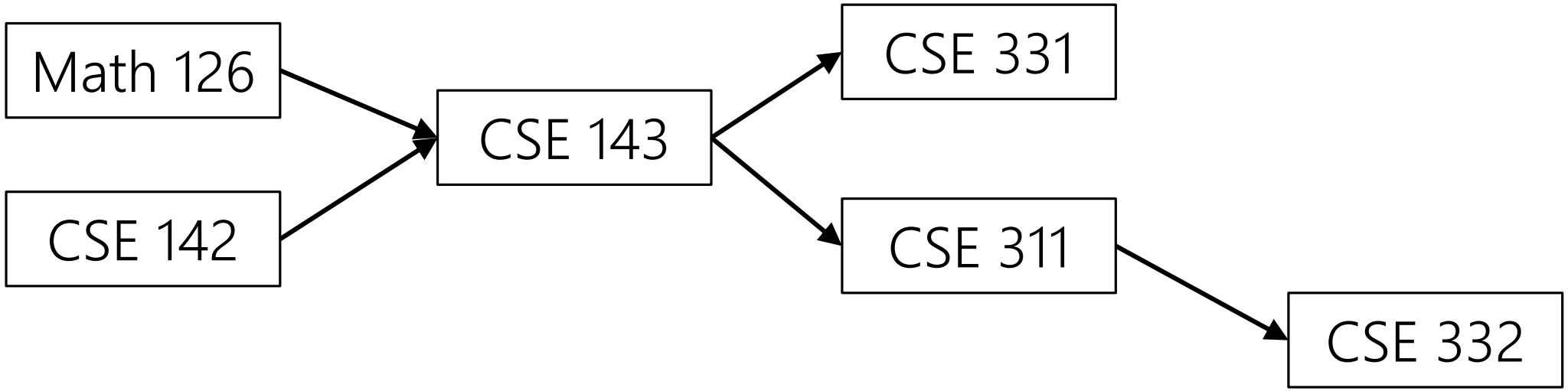
EVERY TREE IS A FOREST.



Topological Sort

Problem 1: Ordering Dependencies

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



Problem 1: 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.

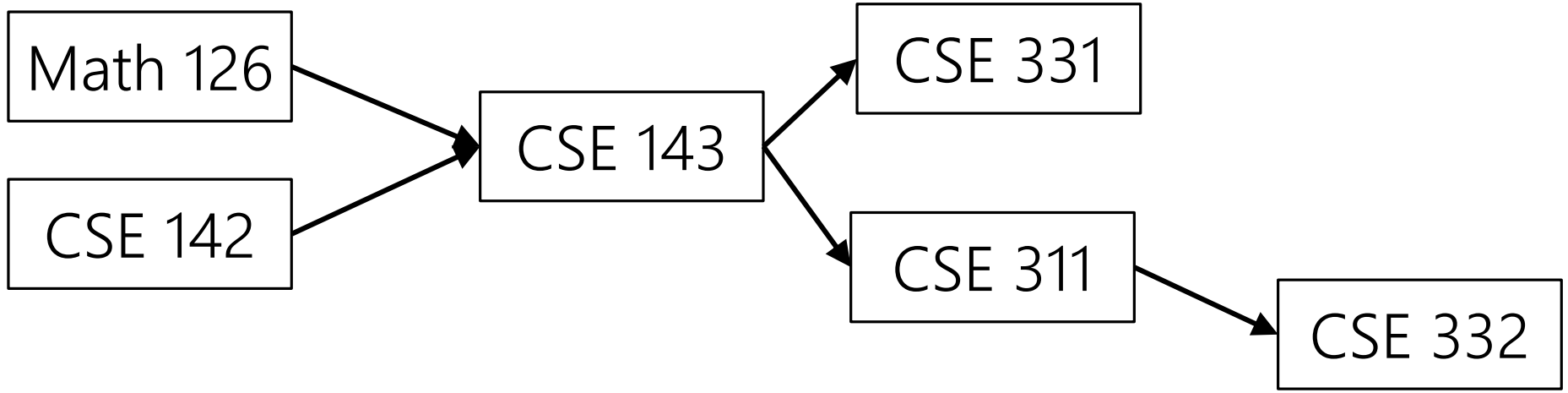
Uses:

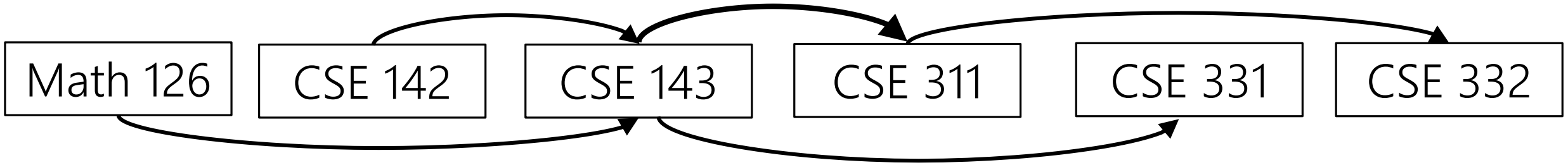
Compiling multiple files

Graduating.

Topological Ordering

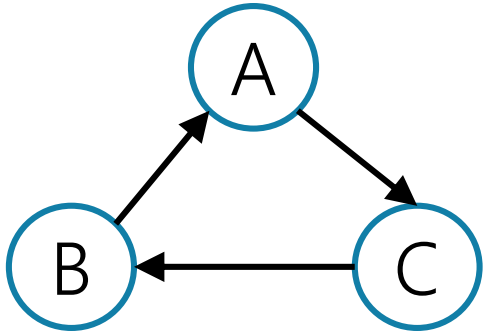
A course prerequisite chart and a possible topological ordering.





Can we always order a graph?

Can you topologically order this graph?



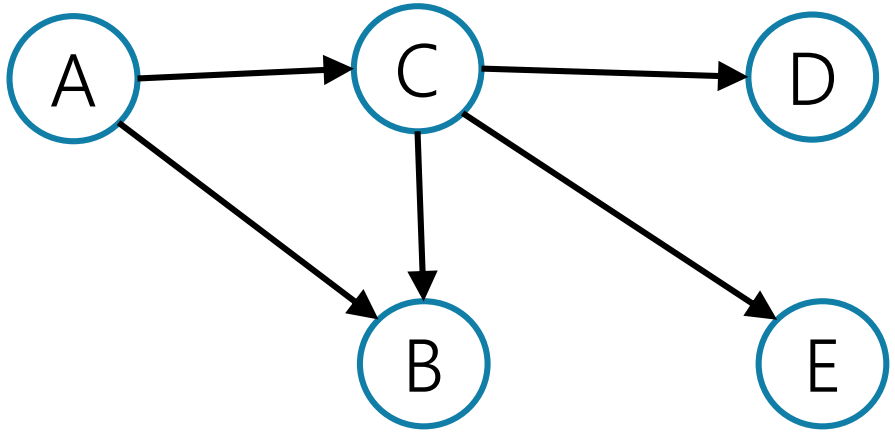
Directed Acyclic Graph (DAG)

A directed graph without any cycles.

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

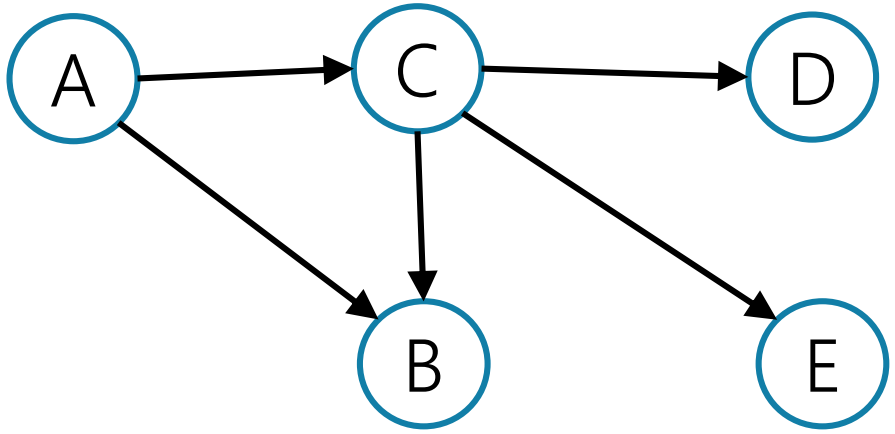
Ordering a DAG

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.

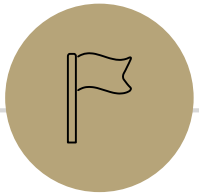
How Do We Find a Topological Ordering?

```
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?

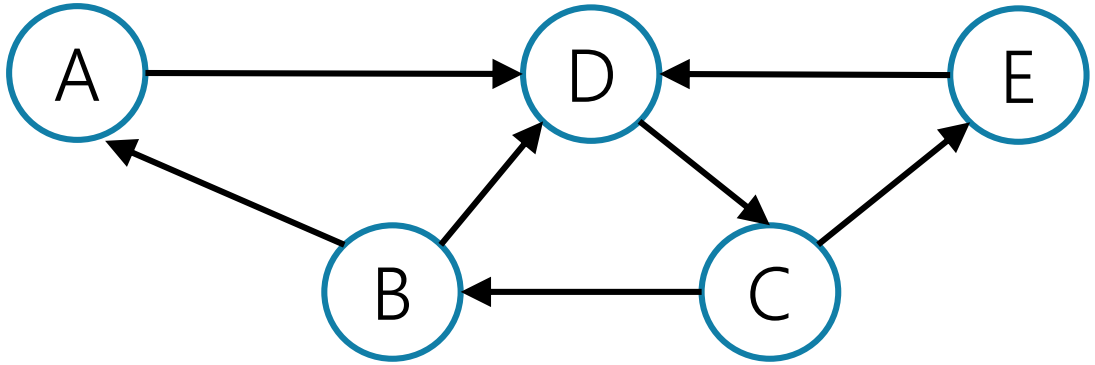
```
TopologicalSort(Graph G, Vertex source)
    count how many incoming edges each vertex has
    Collection toProcess = new Collection()
    foreach(Vertex v in G){
        if(v.edgesRemaining == 0)
            toProcess.insert(v)
    }
    topOrder = new List()
    while(toProcess is not empty){
 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

Running Time: $O(|V| + |E|)$

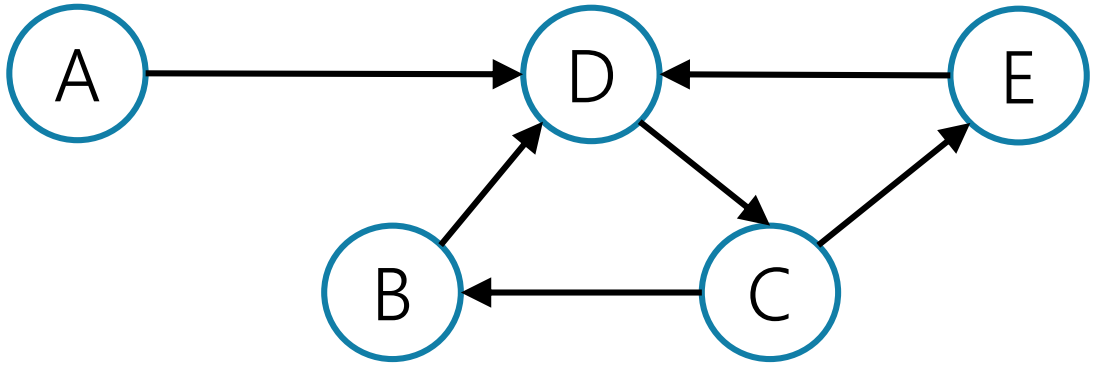


Strongly Connected Components

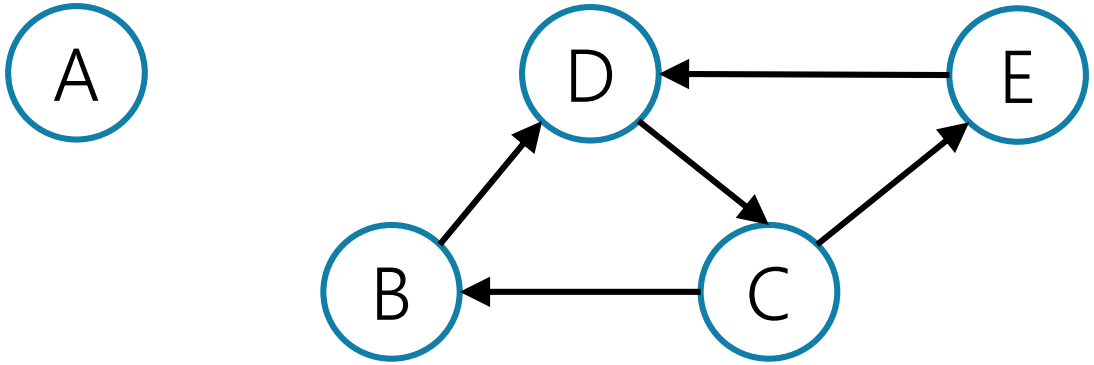
Directed Graph Connectedness



Strongly Connected: Can get from every vertex to every other and back!



Weakly Connected: Could get from every vertex to every other and back, if you ignore the direction on edges.

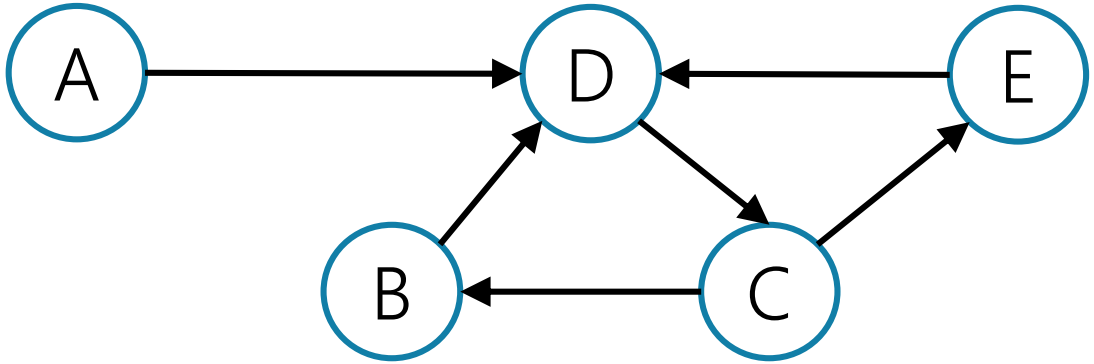


Disconnected: Can't get from every vertex to every other and back, even if you ignore the direction on edges.

Strongly Connected Components

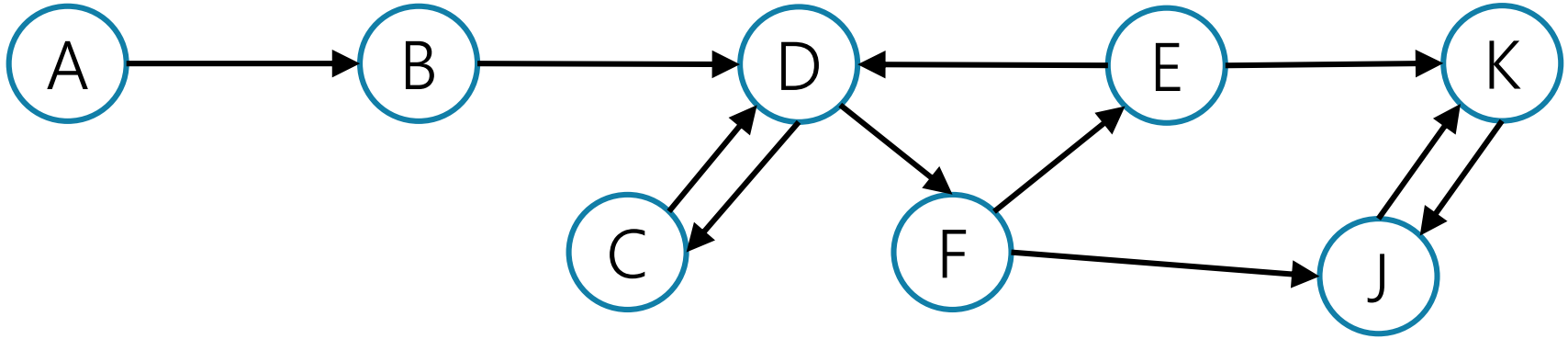
Strongly Connected Component

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.

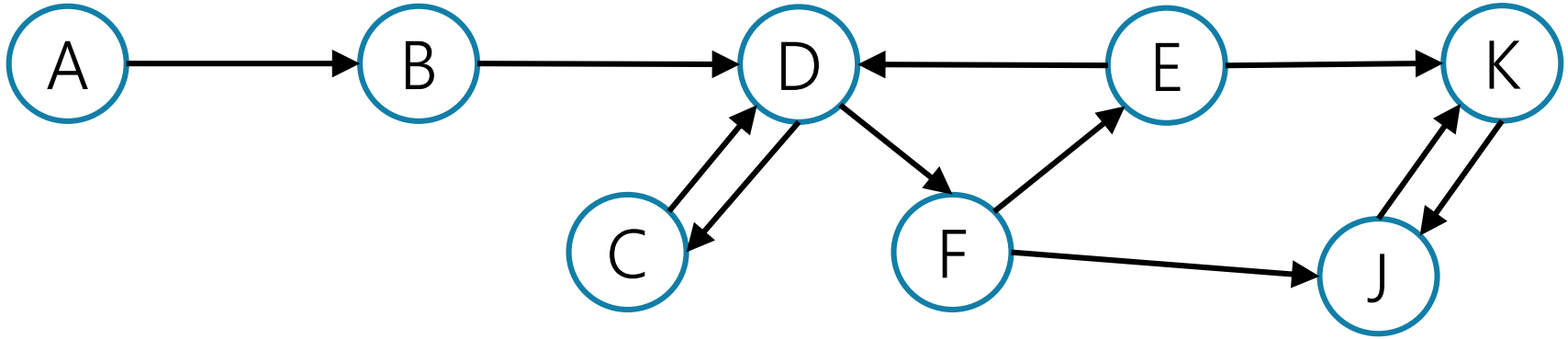


SCCs: {A}, {B,C,D,E}

Strongly Connected Components Problem



Strongly Connected Components Problem



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

Strongly Connected Components Problem

Given: A directed graph G
Find: The strongly connected components of G

SCC Algorithm

Ok. How do we make a computer do this?

You could:

- run a BFS from every vertex,
- For each vertex record what other vertices it can get to
- and figure it out from there.

But you can do better. There's actually an O(|V|+|E|) algorithm!

An Important Subroutine

There's a second way to traverse a graph.

Depth First Search

- Won't find you shortest paths
- But does produce interesting information about what vertices you can reach.

Depth First Search (DFS)

BFS uses a queue to order which vertex we move to next

Gives us a growing “frontier” movement across graph

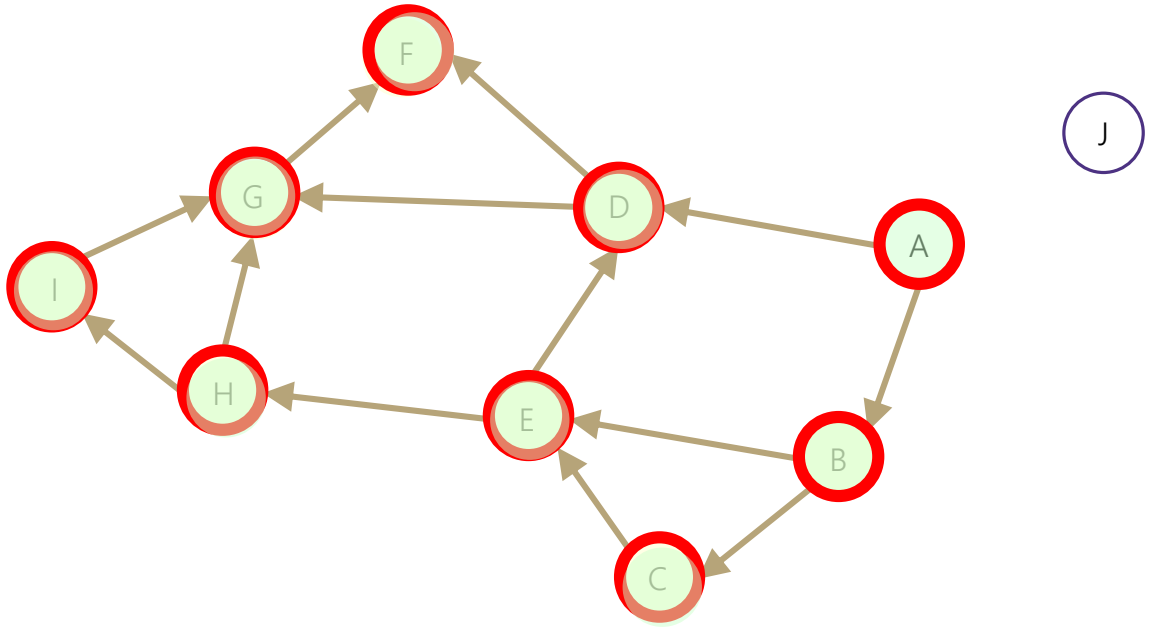
Can you move in a different pattern? What if you used a stack instead?

```
bfs(graph)
    toVisit.enqueue(first vertex)
   mark first vertex as visited
    while(toVisit is not empty)
       current = toVisit.dequeue()
       for (V : current.neighbors())
          if (v is not visited)
             toVisit.enqueue(v)
           mark v as visited
       finished.add(current)
```

```
dfs(graph)
    toVisit.push(first vertex)
   mark first vertex as visited
    while(toVisit is not empty)
       current = toVisit.pop()
       for (V : current.neighbors())
          if (V is not visited)
             toVisit.push(v)
           mark v as visited
       finished.add(current)
```

Depth First Search

```
dfs(graph)
    toVisit.push(first vertex)
    mark first vertex as visited
    while(toVisit is not empty)
        current = toVisit.pop()
        for (V : current.neighbors())
            if (V is not visited)
                toVisit.push(v)
                mark v as visited
        finished.add(current)
```



Current node: D

Stack: D G F H E B C

Finished: A B E H G F I C D

DFS

Running time?

- Same as BFS: $O(|V| + |E|)$

You can rewrite DFS to be a recursive method.

Use the call stack as your stack.

No easy trick to do the same with BFS.

SCC Algorithm

Ok. How do we make a computer do this?

You could:

- run a BFS from every vertex,
- For each vertex record what other vertices it can get to
- and figure it out from there.

But you can do better. There's actually an O(|V|+|E|) algorithm!

SCC Algorithm

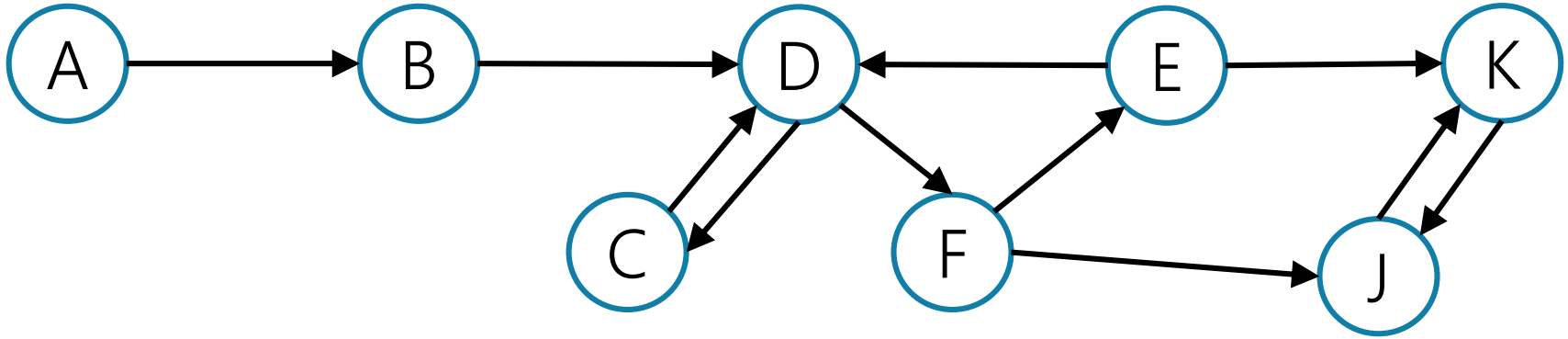
I only want you to remember two things about the algorithm:

- It is an application of depth first search.
- It runs in linear time

The problem with running a [B/D]FS from every vertex is you recompute a lot of information.

The time a vertex is popped off the stack in (recursive) DFS contains a "smart" ordering to do a second DFS where you don't need to recompute that information.

SCC Algorithm



If we run a DFS from A and another one from B, we'll go through almost the entire graph twice.

Starting at J or K and moving from "right to left" will let us avoid recomputation.

Details at end of this slide deck.

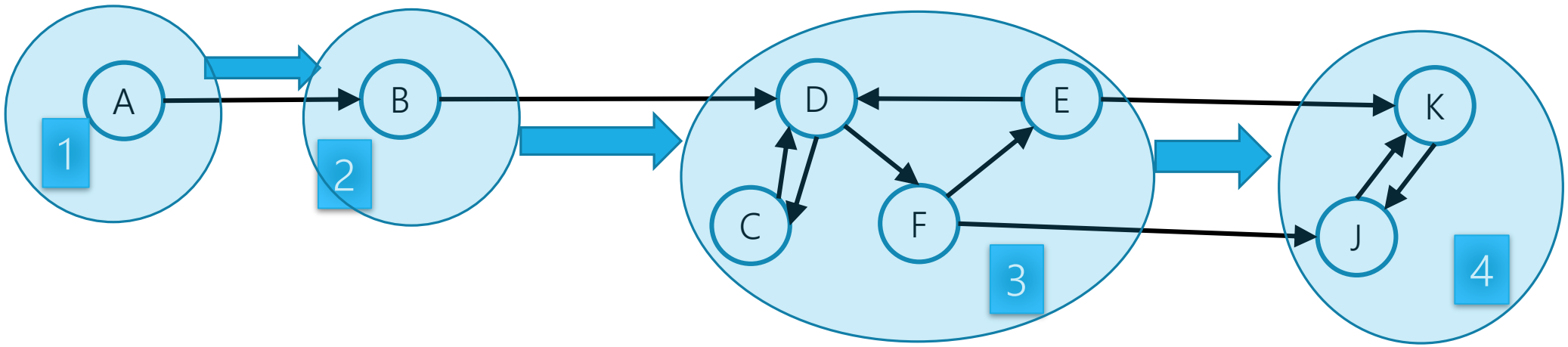
For the final, we might ask you to find Strongly Connected Components, but won't require you use the algorithm.

Why Find SCCs?

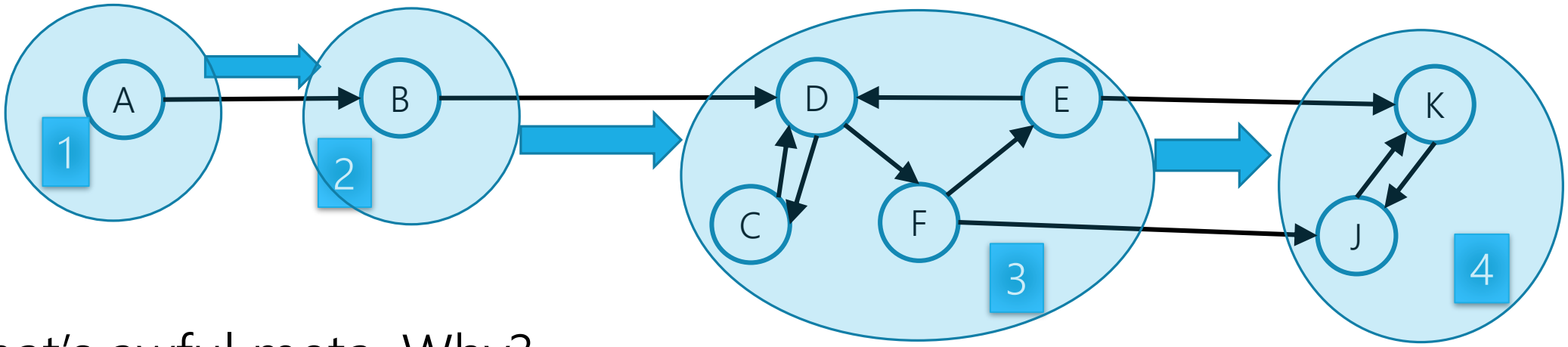
Graphs are useful because they encode relationships between arbitrary objects.

Let's build a new graph out of the SCCs! Call it H

- 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.



Why Find SCCs?



That's awful meta. Why?

This new graph summarizes reachability information of the original graph.

- I can get from A (of G) in 1 to F (of G) in 3 if and only if I can get from 1 to 3 in H.

Why Must H Be a DAG?

H is always a DAG (do you see why?).

Takeaways

Finding SCCs lets you **collapse** your graph to the meta-structure.
If (and only if) your graph is a DAG, you can find a topological sort of your graph.

Both of these algorithms run in linear time.

Just about everything you could want to do with your graph will take at least as long.

You should think of these as **"almost free" preprocessing** of your graph.

- Your other graph algorithms only need to work on
  - topologically sorted graphs and
  - strongly connected graphs.

A Longer Example

The best way to really see why this is useful is to do a bunch of examples.

You'd need to take 421 first.
The second best way is to see one example right now…

This problem doesn't *look like* it has anything to do with graphs

- no maps
- no roads
- no social media friendships

Nonetheless, a graph representation is the best one.

Example Problem: Final Creation

We have a list of types of problems we might want to put on the final.

- ForkJoin code, Hash tables, B-Trees, Graphs,…

To try to make you all happy, we might ask for your preferences. Each of you gives us two preferences of the form "I [do/don't] want a [] problem on the final" *

We'll assume you'll be happy if you get at least one of your two requests.

Final Creation Problem

Given: A list of 2 preferences per student.
Find: A set of questions so every student gets at least one of their preferences (or accurately report no such question set exists).

*This is NOT how I'm making the final.

Final Creation: Take 1

We have Q kinds of questions and S students.

What if we try every possible combination of questions.

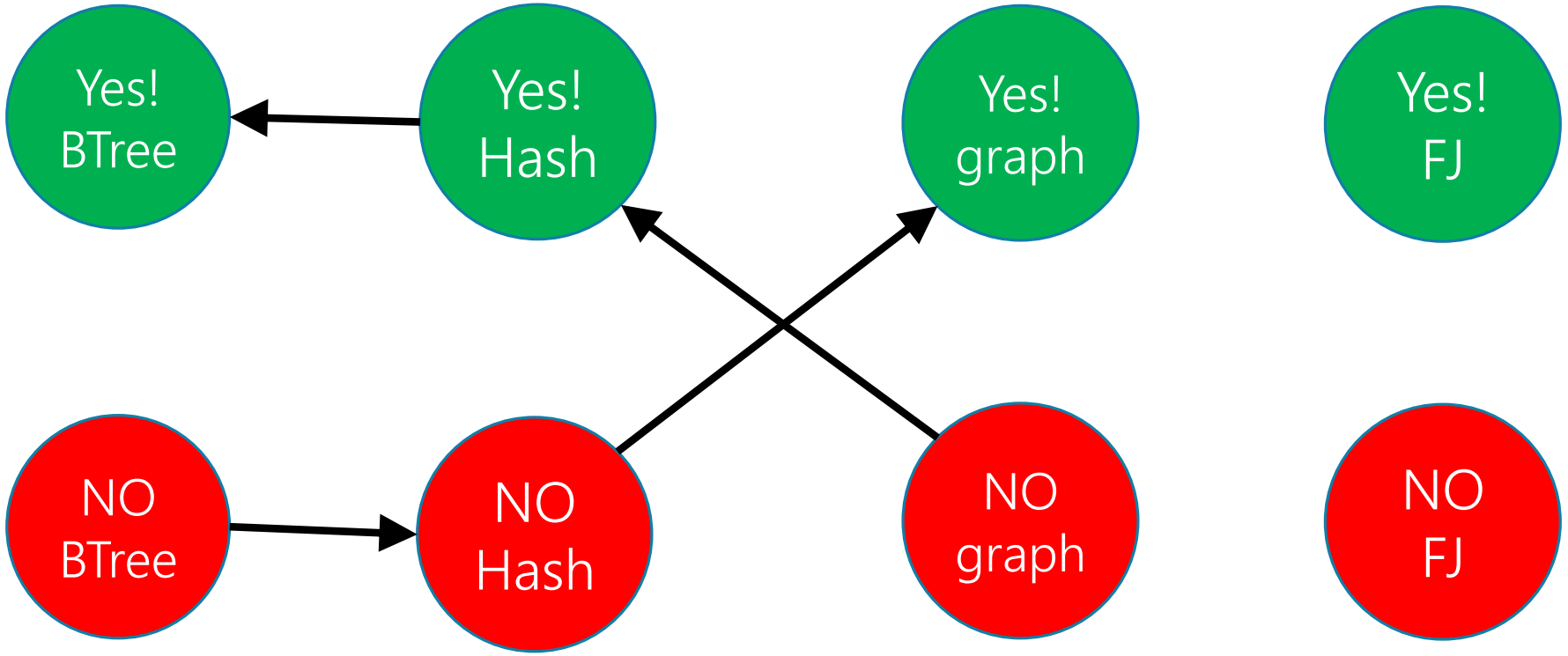
How long does this take? $O(2^Q S)$

If we have a lot of questions, that's **really** slow.

Final Creation: Take 2

Each student introduces new relationships for data:

Let's say your preferences are represented by this table:



Problem	YES	NO
B-Tree	X	
Hash Table		X
Graph		
Fork Join		

Problem	YES	NO
B-Tree		
Hash Table	X	
Graph	X	
Fork Join		

If we don't include a B-Trees, can you still be happy?

If we do include a hash tables can you still be happy?

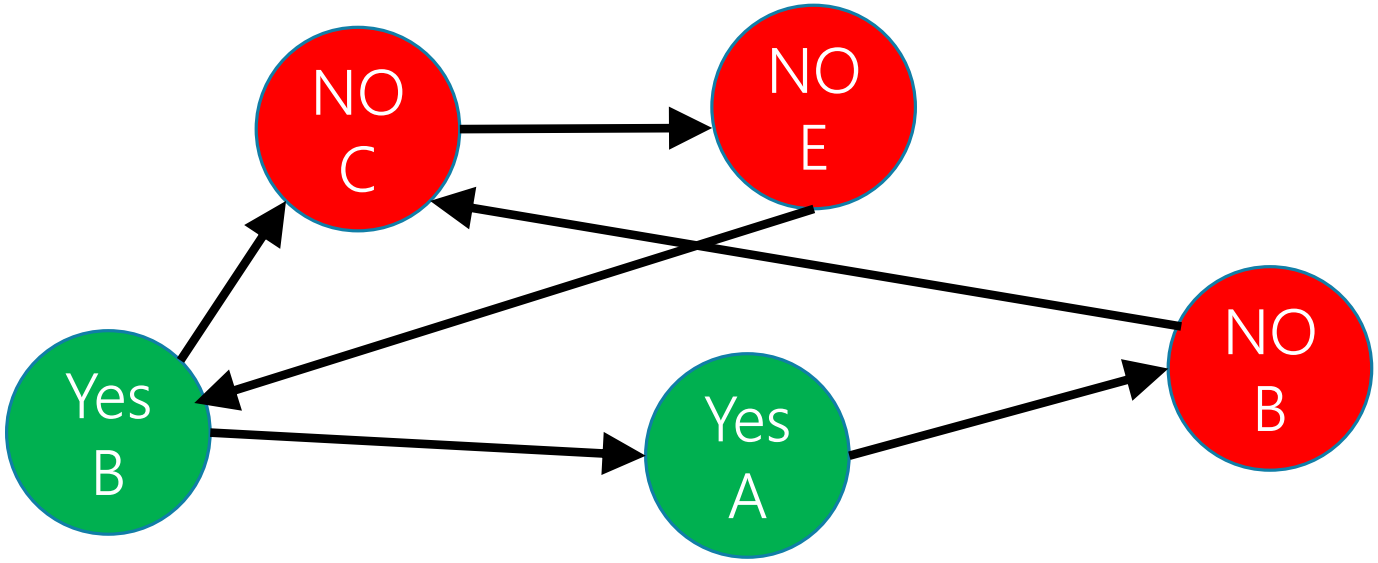
Final Creation: Take 2

Hey we made a graph!

What do the edges mean?

- We need to avoid an edge that goes TRUE THING → FALSE THING

Let's think about a single SCC of the graph.



Final Creation: SCCs

The vertices of a given SCC must either be all true or all false.

Algorithm Step 1: Run SCC on the graph. Check that each question-type-pair are in different SCC.

Now what? Every SCC gets the same value.

- Treat it as a single object!

We want to avoid edges from true things to false things.

- "Trues" seem more useful for us at the end.

Is there some way to start from the end?

YES! Topological Sort

Making the Final

Algorithm:
Make the requirements graph.

Find the SCCs.

If an SCC has including and not including a problem, no final is possible.

Run topological sort on the graph of SCC.

Starting from the end:

- if everything in a component is unassigned, set them to true, and set their opposites to false.
- Else If one thing in a component is assigned, assign the same value to the rest of the nodes in the component and the opposite value to their opposites.

Making The Final

This works!!

The proof is a bit more involved. Just trust me.

How fast is it?

O(Q + S). That's a HUGE improvement.

Some More Context

The Final Making Problem was a type of “Satisfiability” problem.

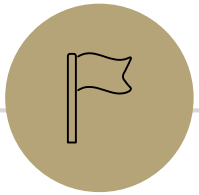
We had a bunch of variables (include/exclude this question), and needed to satisfy everything in a list of requirements.

SAT is a general way to encode lots of hard problems.

Because every requirement was “do at least one of these 2” this was a 2-SAT instance.

What happens if we change the 2 to a 3?

The problem is very different. We’ll pick up that idea Monday.



Optional Content: Strongly Connected Components Algorithm

Efficient SCC

We'd like to find all the vertices in our strongly connected component in time corresponding to the size of the component, not for the whole graph.

We can do that with a DFS (or BFS) as long as we don't leave our connected component.

If we're a "sink" component, that's guaranteed. I.e. a component whose vertex in the meta-graph has no outgoing edges.

How do we find a sink component? We don't have a meta-graph yet (we need to find the components first)

DFS can find a vertex in a source component, i.e. a component whose vertex in the meta-graph has no incoming edges.

- That vertex is the last one to be popped off the stack in the recursive version of DFS.

So if we run DFS in the *reversed* graph (where each edge points the opposite direction) we can find a sink component.

Efficient SCC

So from a DFS in the reversed graph, we can use the order vertices are popped off the stack to find a sink component (in the original graph).

Run a DFS from that vertex to find the vertices in that component *in size of that component time.*

Now we can delete the edges coming into that component.

The last remaining vertex popped off the stack is a sink of the remaining graph, and now a DFS from them won't leave the component.

Iterate this process (grab a sink, start DFS, delete edges entering the component).

In total we've run two DFSs. (since we never leave our component in the second DFS).

More information, and pseudocode:

https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm