Lecture 21: More Graph Algorithms

Data Structures and Algorithms

Administrivia

We clarified Exercise 5 Problem [] to explicitly mention “worst-case”

Administrivia: Final Information

Remember the final happens in two parts:

Logistics:
Part 1 takes place in section next week.

- No note sheet
- Go to your officially registered section!

Part 2 in lecture next Friday

- Yes note sheet (8.5 x 11 inches, both sides)
- Like the midterm, we'll mark off certain rows not to sit in

Logistically, these are two separate exams (can't work on day 1 stuff on day 2 or vice versa).

Adminsitrivia

Exams tab on webpage has topics list

- Divided into “day 1,” “day 2,” “both days”

Focus of exam will be on applying the concepts you’ve learned.

- Design decisions
- Graph modeling
- Code modeling

Wrap up of MSTs

We didn't explicitly calculate the big-Θ for Prim's.

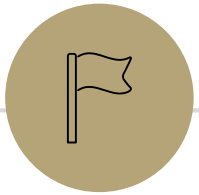
The worst-case is the same as Dijkstra's

- The code is virtually identical, other than switching some constant time operations for other.

That gives $\Theta(m \log n + n \log n)$

If we assume m dominates n (like we did for Kruskal's), we can simplify:

$\Theta(m \log n)$.



Depth First Search

Depth First Search (DFS)

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

Gives us a growing "frontier" movement across graph

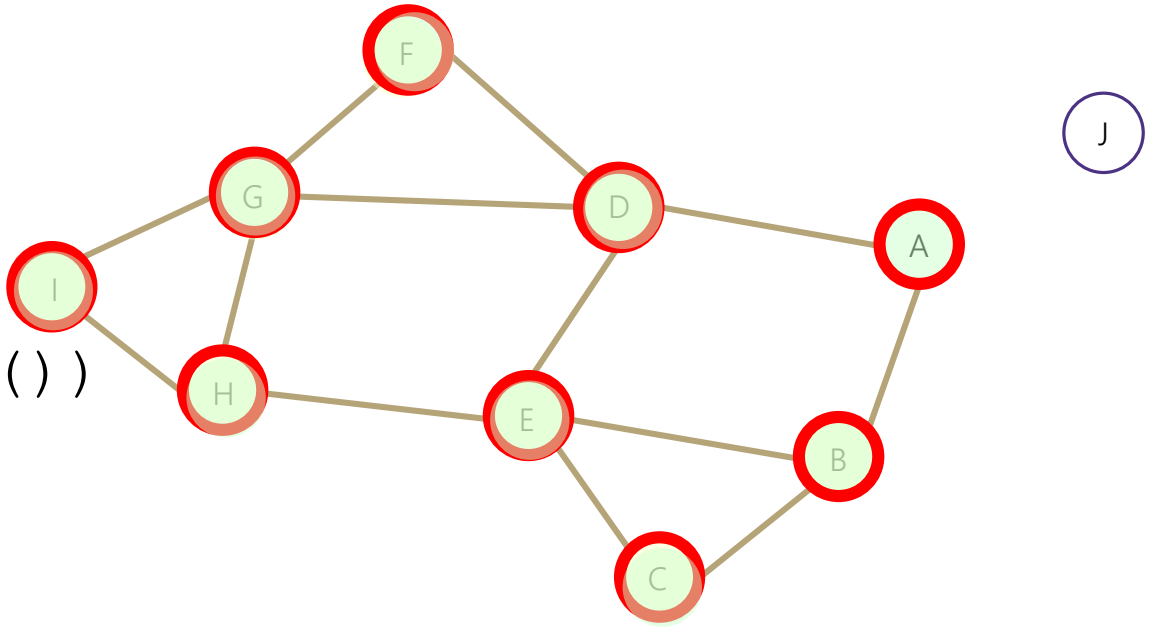
Can you move in a different pattern? What if you used a stack instead?

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

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

Depth First Search

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



Current node: D

Stack: D G B E H G

Finished: A B E H G F I C D

DFS

Worst-case running time?

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

Indicentally, 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.

BFS vs. DFS

So why have two different traversals?

For the same reason we had pre/post/in –order traversals for trees!

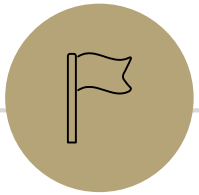
BFS and DFS will find vertices in a different order, so they can let you calculate different things.

Sometimes the order doesn't matter

- like if you want to find components in an undirected graph

We saw BFS is useful for shortest paths in unweighted graphs.

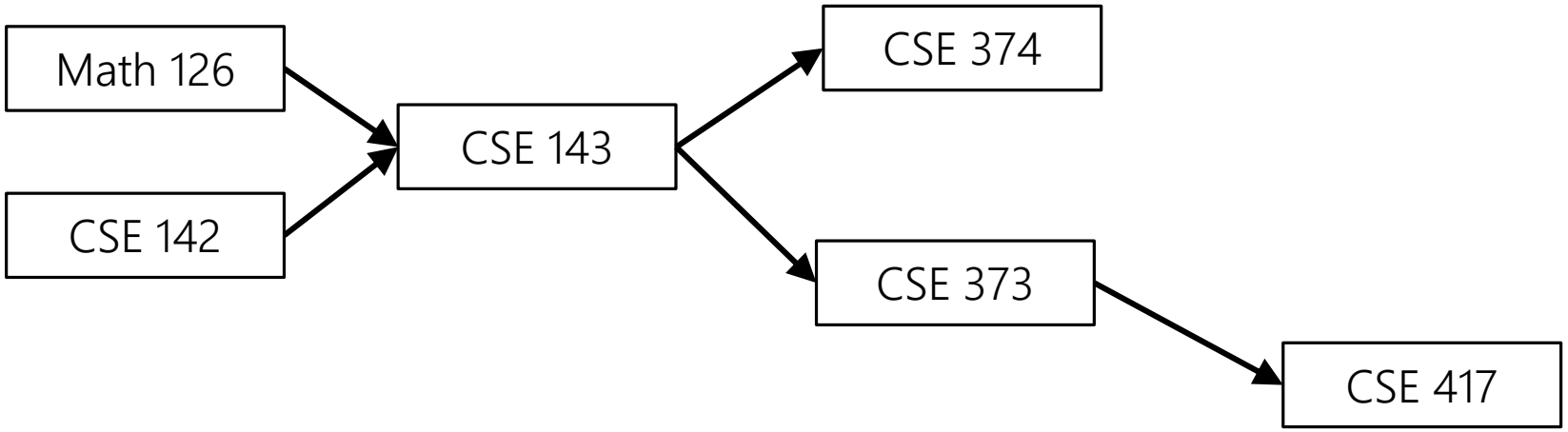
We'll see an application of DFS later today.



Topological Sort

Problem 1: Ordering Dependencies

Today's (first) 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.

We can only do things one at a time, can we find an order that **respects dependencies**?

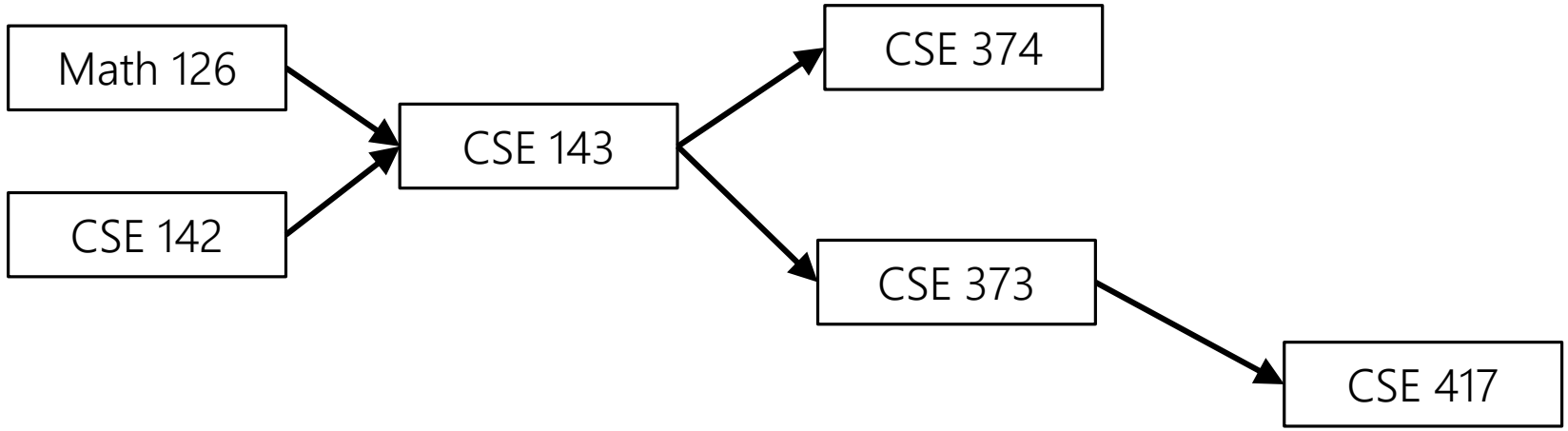
Topological Sort (aka Topological Ordering)

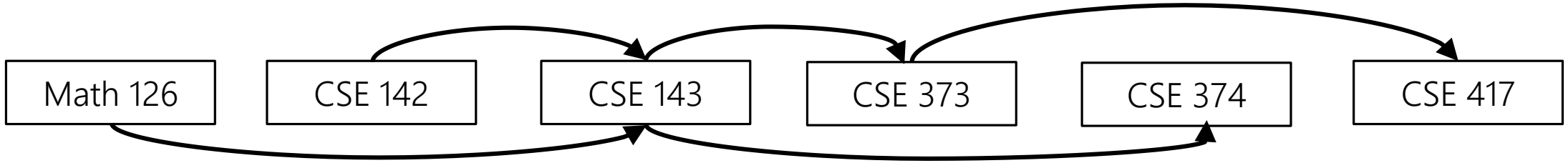
Given: a directed graph G
Find: an ordering of the vertices so all edges go from left to right.

Uses:
Compiling multiple files
Graduating

Topological Ordering

A course prerequisite chart and a possible topological ordering.

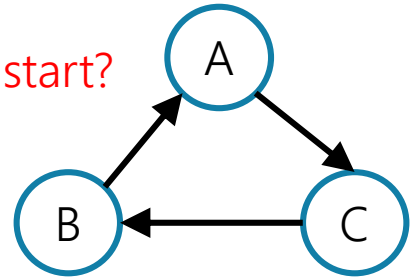




Can we always order a graph?

Can you topologically order this graph? No

Where do I start?



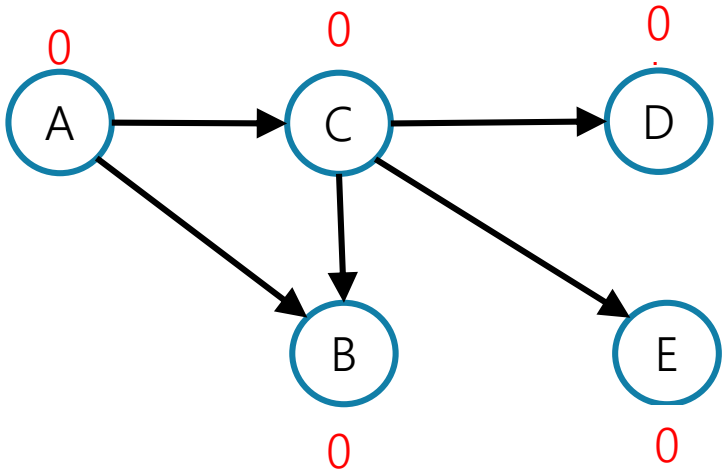
Directed Acyclic Graph (DAG)

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.



A C B D E

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)
        }
    }
```

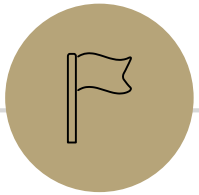
[B/D]FS
Graph linear
+ V + E

Pick something with $O(1)$ insert / removal

$+V$

Runs as most once per edge
$+E$

$$O(V + E)$$



Strongly Connected Components

Connected Components (undirected graphs)

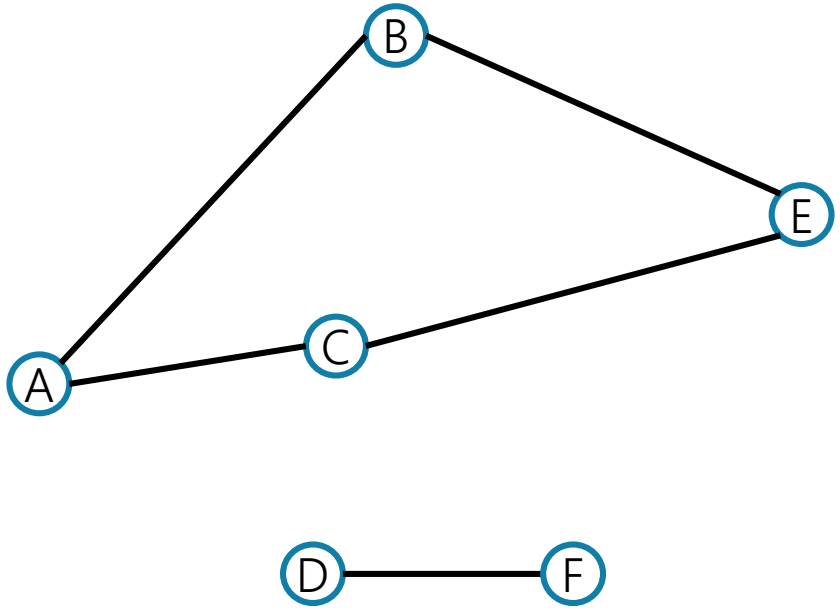
A **connected component** (or just **"component"**) is a "piece" of an undirected graph.

Connected component [undirected graphs]

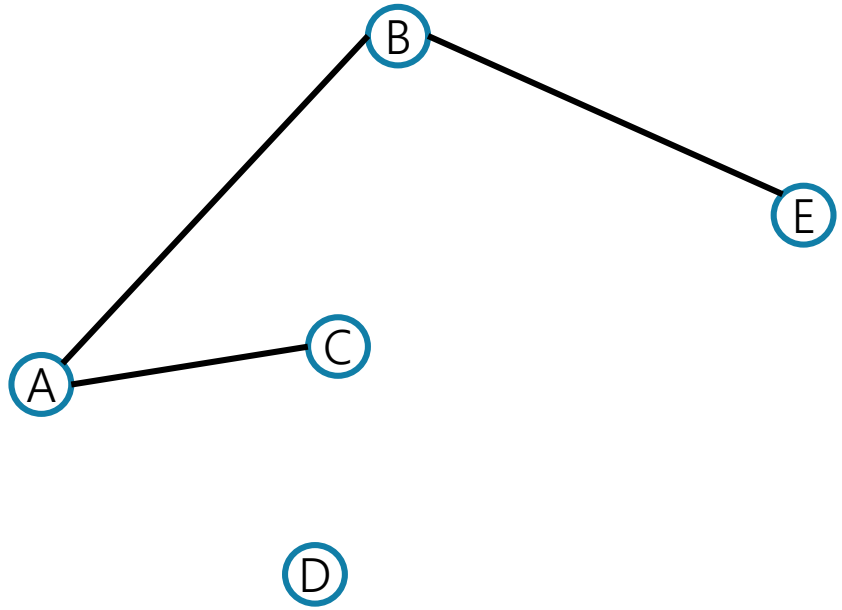
A set S of vertices is a connected component (of an undirected graph) if:

1. It is connected, i.e. for all vertices u, v in S: there is a walk from u to v
2. It is maximal:
   - Either it's the entire set of vertices, or
   - For **every** vertex u that's not in S, $S \cup \{u\}$ is not connected.

Find the connected components



{A,B,C,E}, {D,F} are the two components



{A,B,C,E}, {D} are the two components

Directed Graphs

In directed graphs we have two different notions of "connected"

One is "I can get there from here OR here from there"
The other is "I can get there from here AND here from there"

Weakly Connected/Weakly Connected Components:

- Pretend the graph is undirected (ignore the direction of the arrows)
- Find the components of the undirected graph.

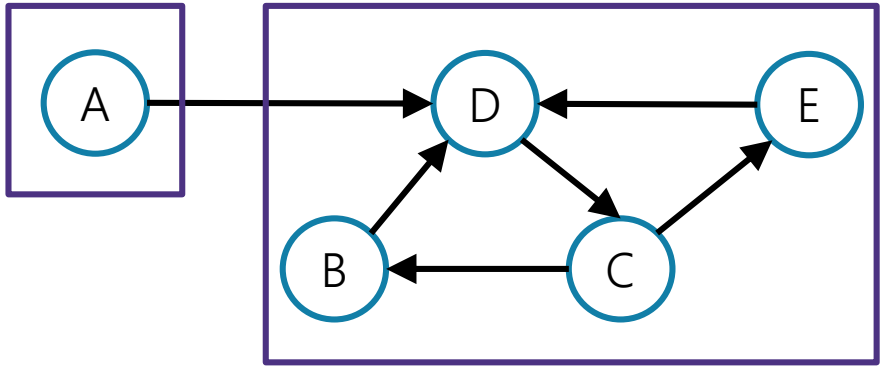
Strongly connected components

- Want to get both directions

Strongly Connected Components

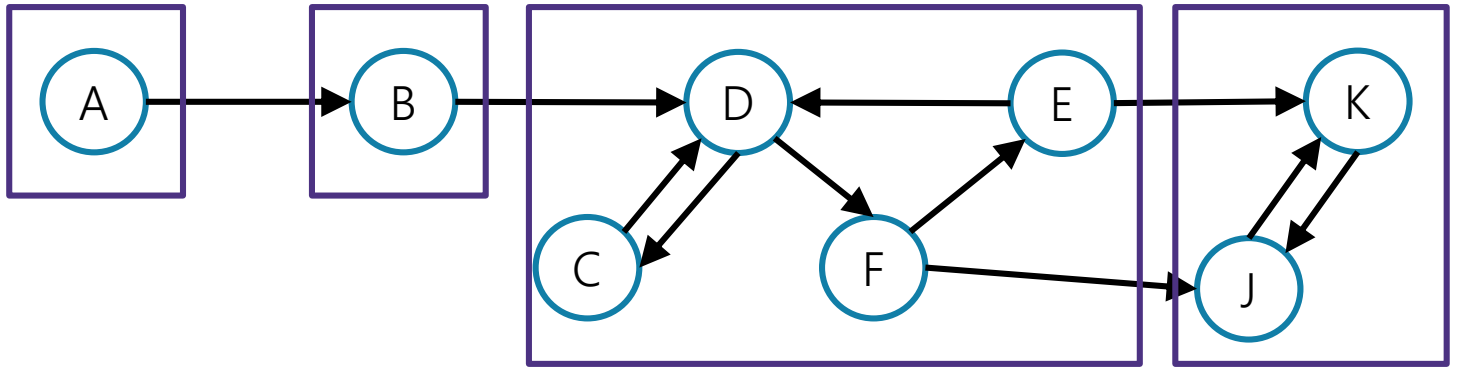
Strongly Connected Component

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.



Note: the direction of the edges matters!

Your turn: Find Strongly Connected Components



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

Strongly Connected Component

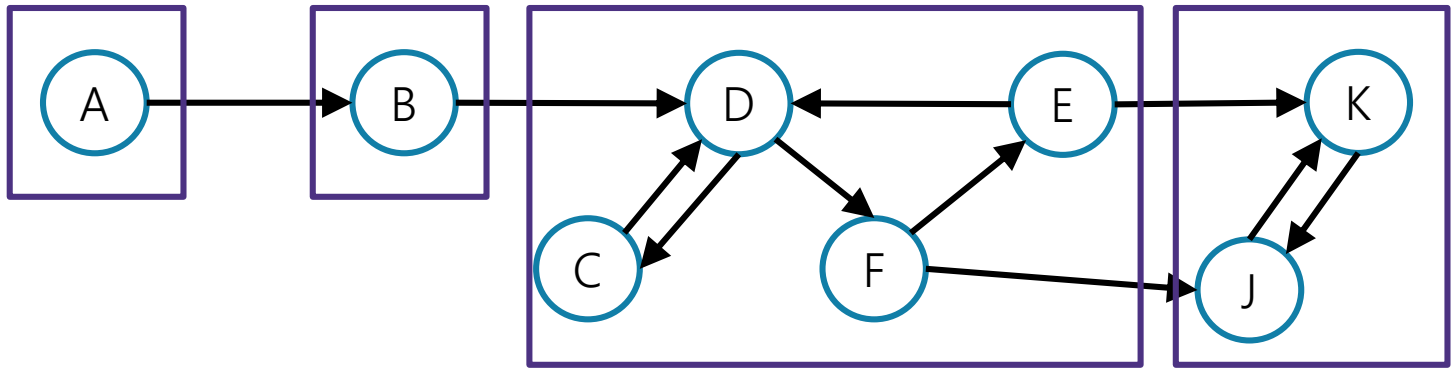
A subgraph C such that every pair of vertices in C is connected via some path **in both directions,** and there is no other vertex which is connected to every vertex of C in both directions.

Finding 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



Finding 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

But you can do better!

We're recomputing a bunch of information, going from back to front skips recomputation.

- Run a DFS first to do initial processing
- While running DFS, run a second DFS to find the components based on the ordering you pull from the stack
- Just two DFSs!
- (see appendix for more details)

Know two things about the algorithm:

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

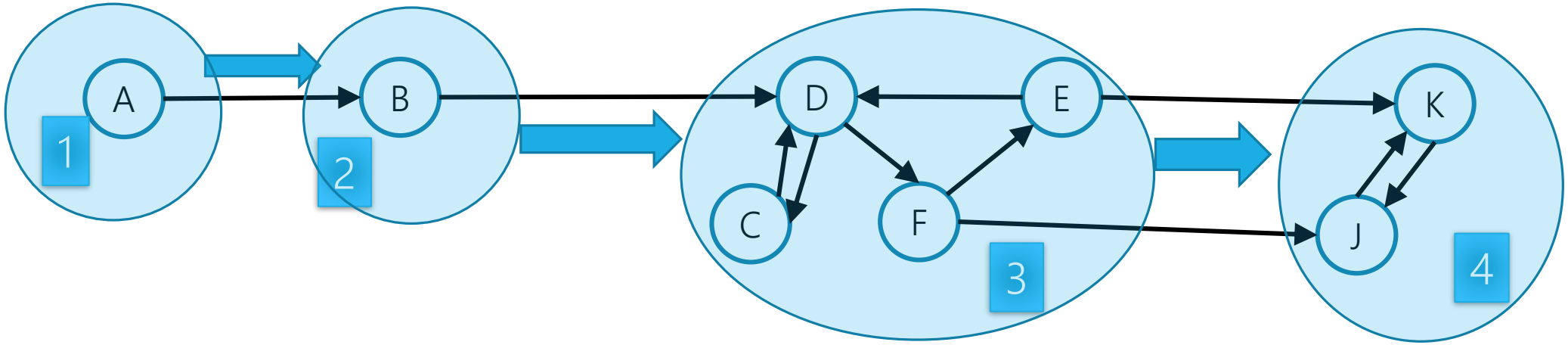
Why Find SCCs?

Graphs are useful because they encode relationships between arbitrary objects.

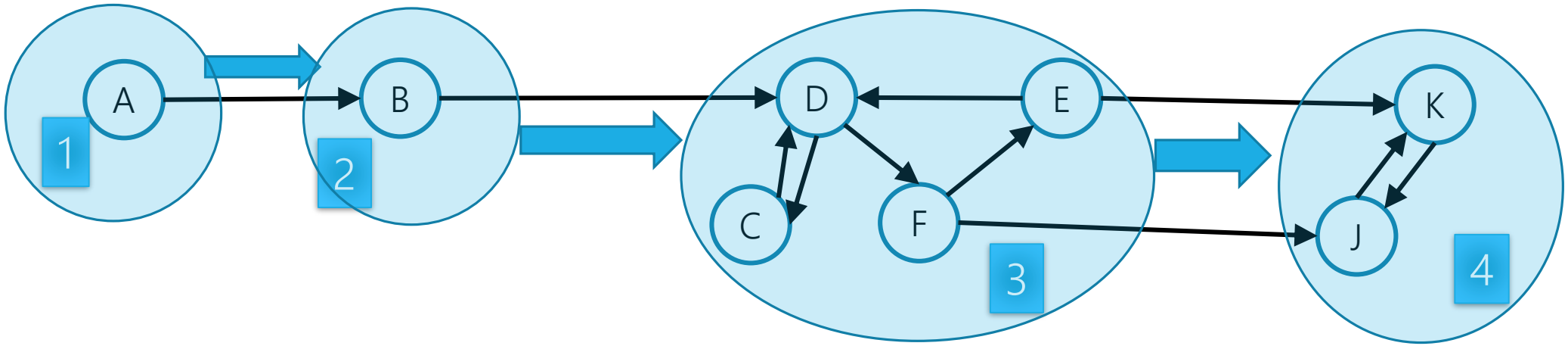
We've found the strongly connected components of G.

Let's build a new graph out of them! 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

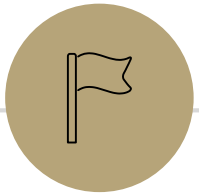
Finding SCCs lets you **collapse** your graph to the meta-structure.
If (and only if) your graph is a DAG, you can find a topological sort of your graph.

Both of these algorithms run in $O(m + n)$ 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.



Graph Modeling Practice

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 are a Disneyland employee and you need to rope off as many miles of walkways as you can for the fireworks while still allowing guests to access to all the rides.

What are the vertices?

Rides

What are the edges?

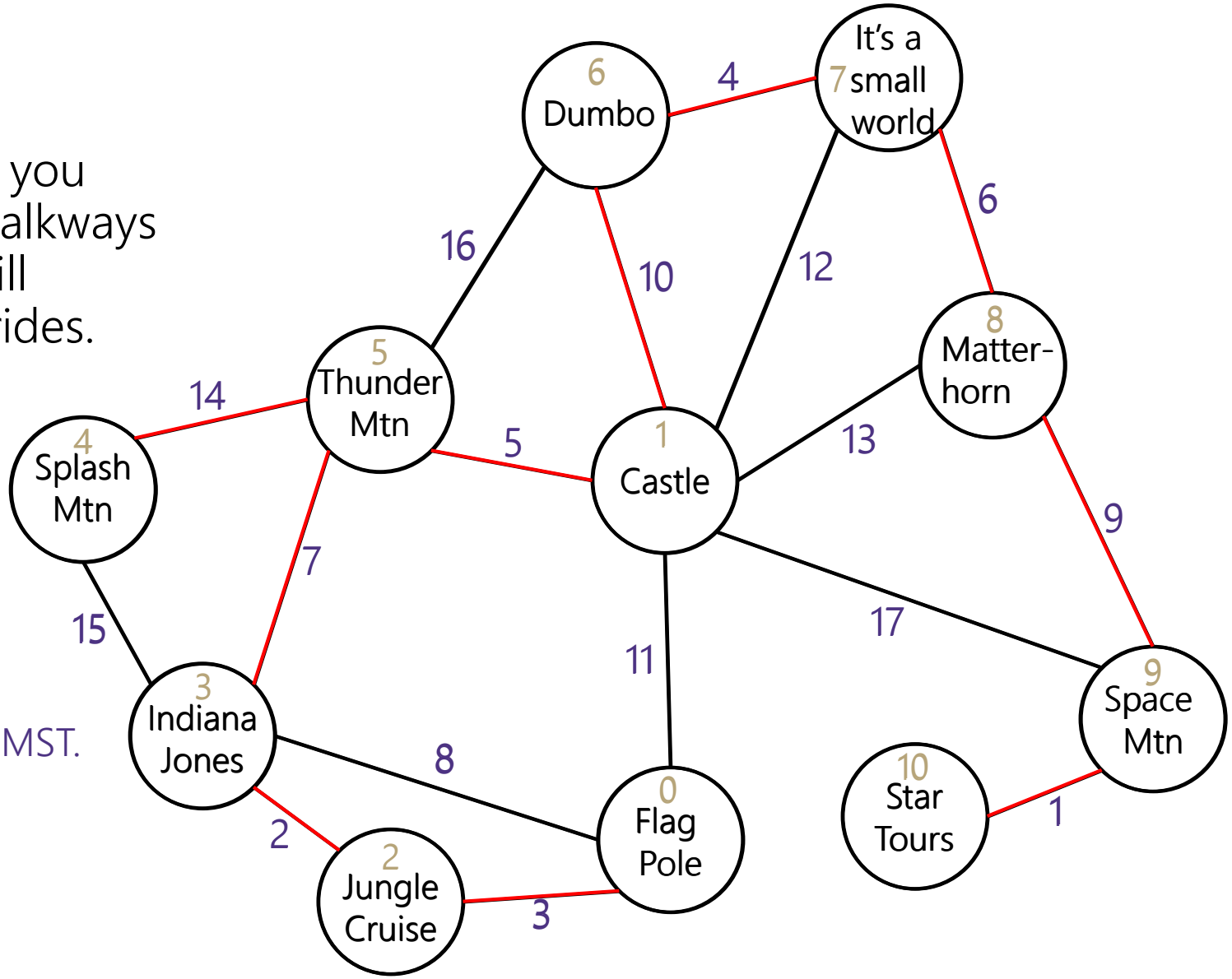
Walkways with distances

What are we looking for?

- We want to rope off everything except an MST.

What do we run?

- Kruskal's or Prim's



Scenario #2

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?

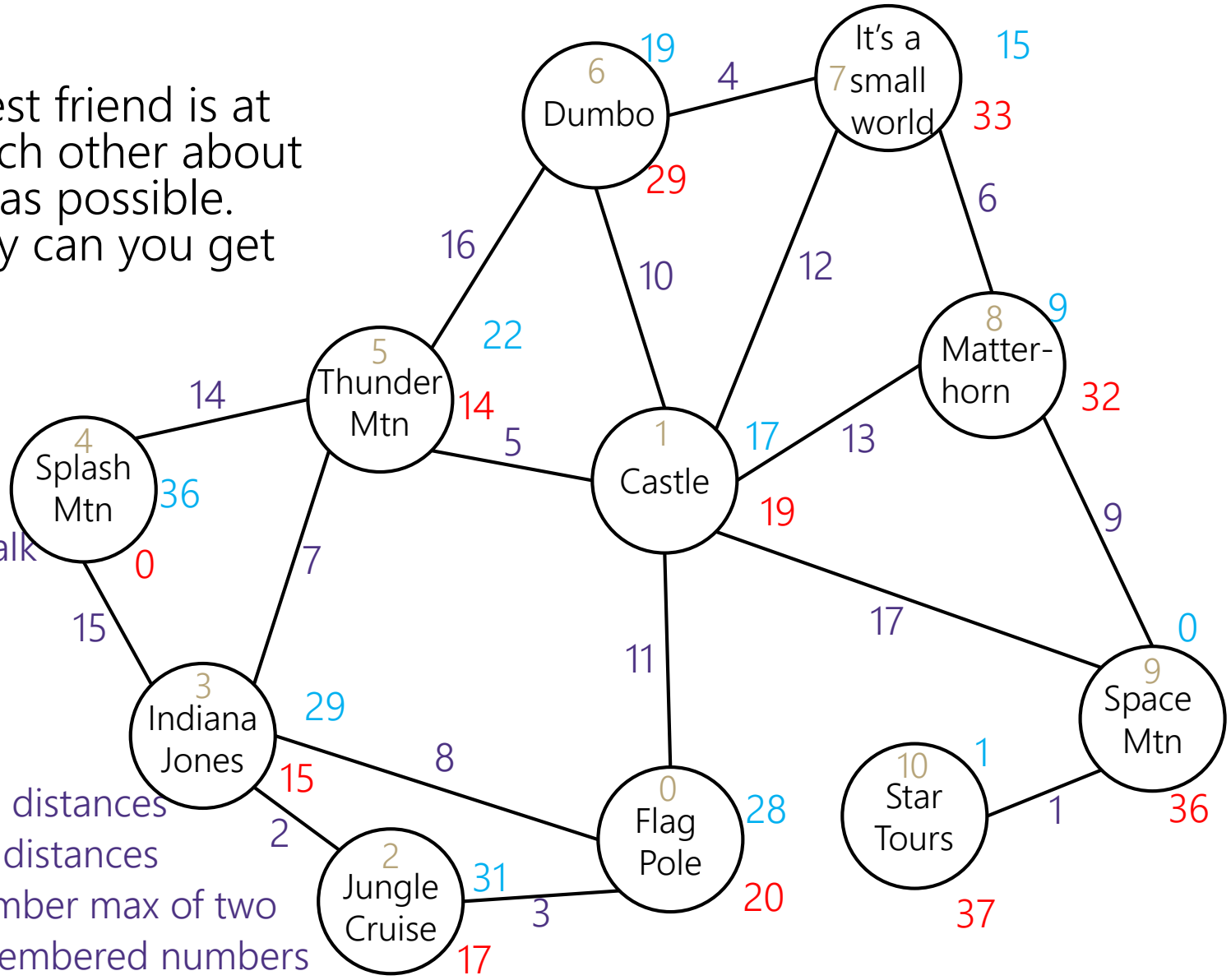
Walkways with how long it would take to walk

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 #3

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?

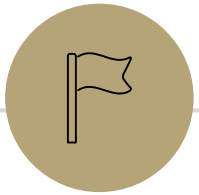
Edges are directed, have an edge from X to Y if you know X came before Y.

What are we looking for?

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

What do we run?

- Topological Sort!



Optional Content: Strongly Connected Components Algorithm Details

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.

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

http://jeffe.cs.illinois.edu/teaching/algorithms/notes/19-dfs.pdf (mathier)