DFS, Graph Modeling

CSE 417 Winter 21
Lecture 6

Edge Classification (for DFS on directed graphs)

Edge type	Definition	When is (u, v) that edge type?
Tree	Edges forming the DFS tree (or forest).	v was not seen before we processed (u, v).
Forward	From ancestor to descendant in tree.	u and v have been seen, and `u.start < v.start < v.end < u.end`
Back	From descendant to ancestor in tree.	u and v have been seen, and `v.start < u.start < u.end < v.end`
Cross	Edges going between vertices without an ancestor relationship.	u and v have not been seen, and `v.start < v.end < u.start < u.end`

The third column doesn't look like it encompasses all possibilities.

It does – the fact that we're using a stack limits the possibilities:

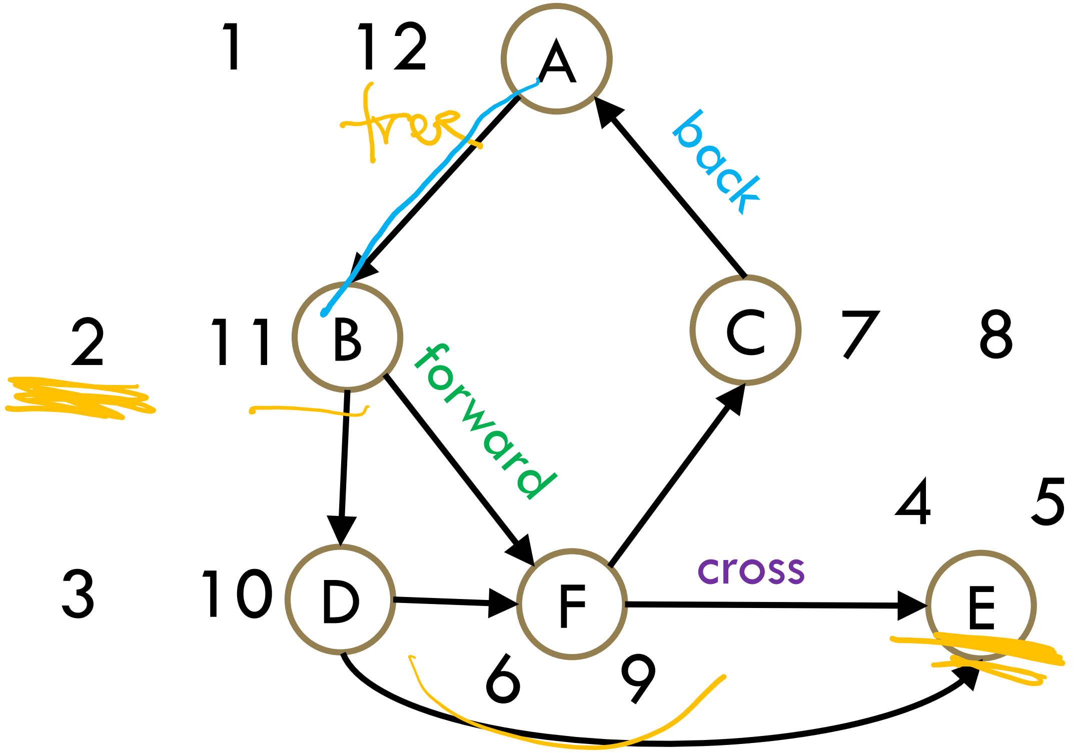
e.g. `u.start < v.start < u.end < v.end` is impossible.

And the rules of the algorithm eliminate some other possibilities.

Try it Yourselves!

```
DFSWrapper(G)
    counter = 0 1
    For each vertex u of G
        If u is not "seen"
            DFS(u)
        End If
    End For

DFS(u)
    Mark u as "seen"
    u.start = counter++
    For each edge (u,v) //leaving u
        If v is not "seen"
            DFS(v)
        End If
    End For
    u.end = counter++
```

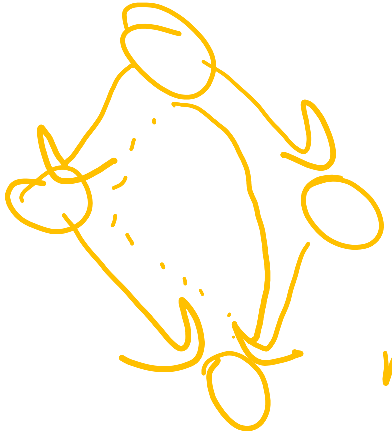


Actually Using DFS

Here's a claim that will let us use DFS for something!

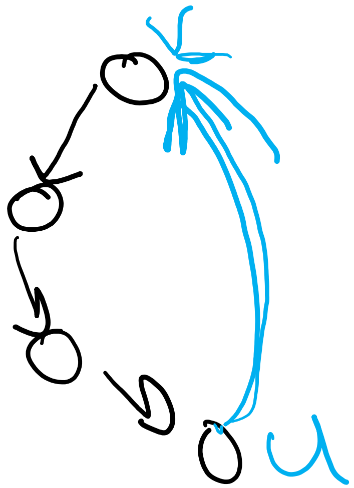
Back Edge Characterization
DFS run on a directed graph has a back edge if and only if it has a cycle.

cycle



not a cycle

Forward Direction



If DFS on a graph has a back edge then it has a cycle.

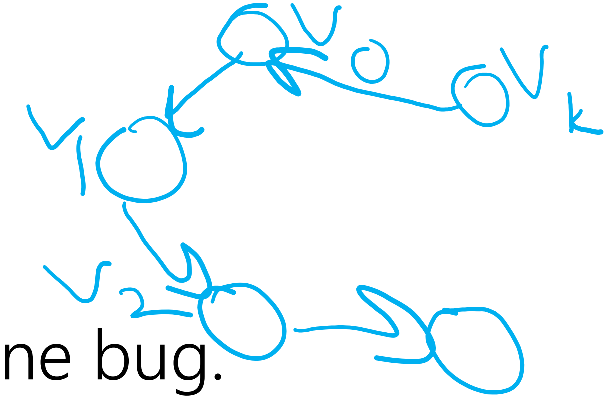
Suppose the back edge is (u, v).

A back edge is going from a descendant to an ancestor.

So we can go from v back to u on the tree edges.

That sounds like a cycle!

Backward direction



This direction is trickier.
Here's a "proof" – it has the right intuition, but (at least) one bug.

Suppose G has a cycle $v_0, v_1, \dots, v_k$.

Without loss of generality, let v_0 be the first node on the cycle DFS marks as seen.

For each i there is an edge from v_i to v_{i+1}.

We discovered v_0 first, so those will be tree edges.

When we get to v_k, it has an edge to v_0 but v_0 is seen, so it must be a back edge.

Talk to your neighbors to find a bug –then try to fix it.

Fixing the Backward Direction

We might not just walk along the cycle in order. Are we going to visit v_k "in time" or might (v_k, v_0) be a cross edge?

DFS discovery
`DFS(v)` finds exactly the (unseen) vertices reachable from v.

Fixing the Backward Direction

We might not just walk along the cycle in order. Are we going to visit v_k "in time" or might (v_k, v_0) be a cross edge?

Suppose G has a cycle $v_0, v_1, \ldots, v_k$.

Without loss of generality, let v_0 be the first node on the cycle DFS marks as seen.

For each i there is an edge from v_i to v_{i+1}

v_k is reachable from v_0 so we must reach v_k before v_0 comes off the stack.

When we get to v_k, it has an edge to v_0 but v_0 is seen, so it must be a back edge.

Summary

DFS discovery
`DFS(v)` finds exactly the (unseen) vertices reachable from v.

Back Edge Characterization
A directed graph has a back edge if and only if it has a cycle.

BFS/DFS caveats and cautions

Edge classifications are different for directed graphs and undirected graphs.

DFS in undirected graphs don’t have cross edges.

BFS in directed graphs can have edges skipping levels (only as back edges, skipping levels up though!)

Summary – Graph Search Applications

BFS

Shortest Paths (unweighted) graphs)

DFS

Cycle detection (directed graphs)

Topological sort

Strongly connected components

Cut edges (on homework)

EITHER

2-coloring

Connected components (undirected)

Usually use BFS – easier to understand.

Designing New Algorithms on Directed Graphs

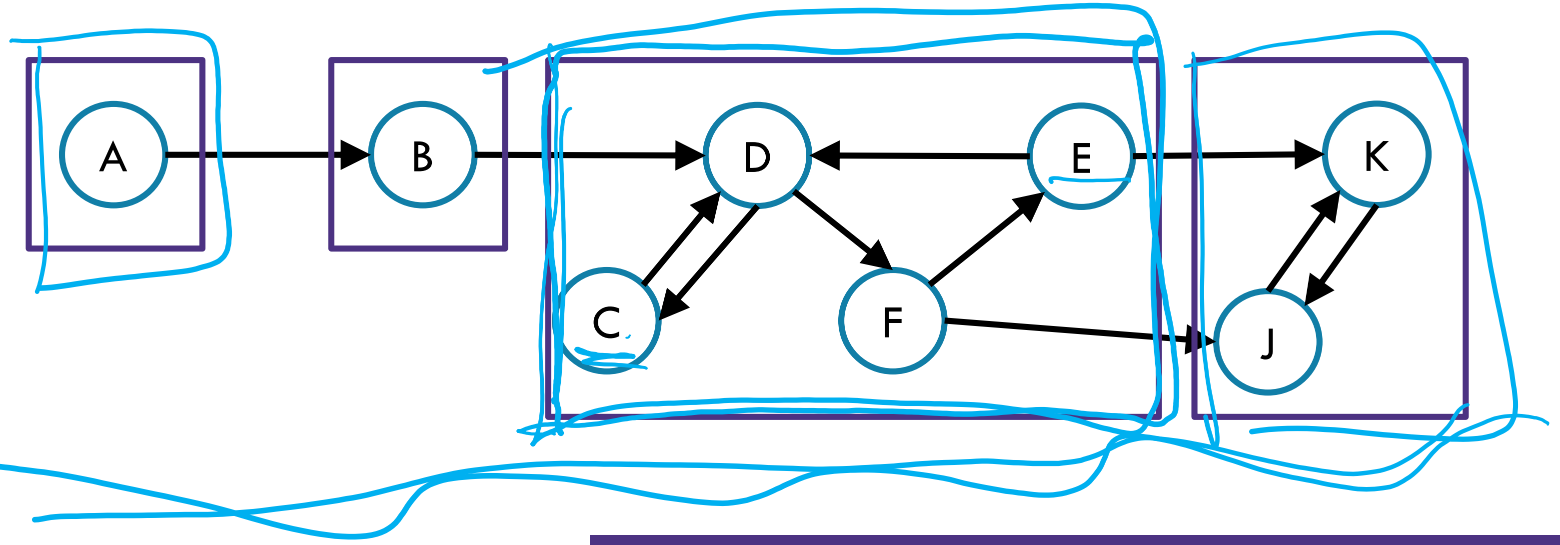
In 373 you learned some applications of depth first search:

Finding Strongly Connected Components

Finding a Topological Sort of a DAG

We'll briefly review what these do, but won't go into details.

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.

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.

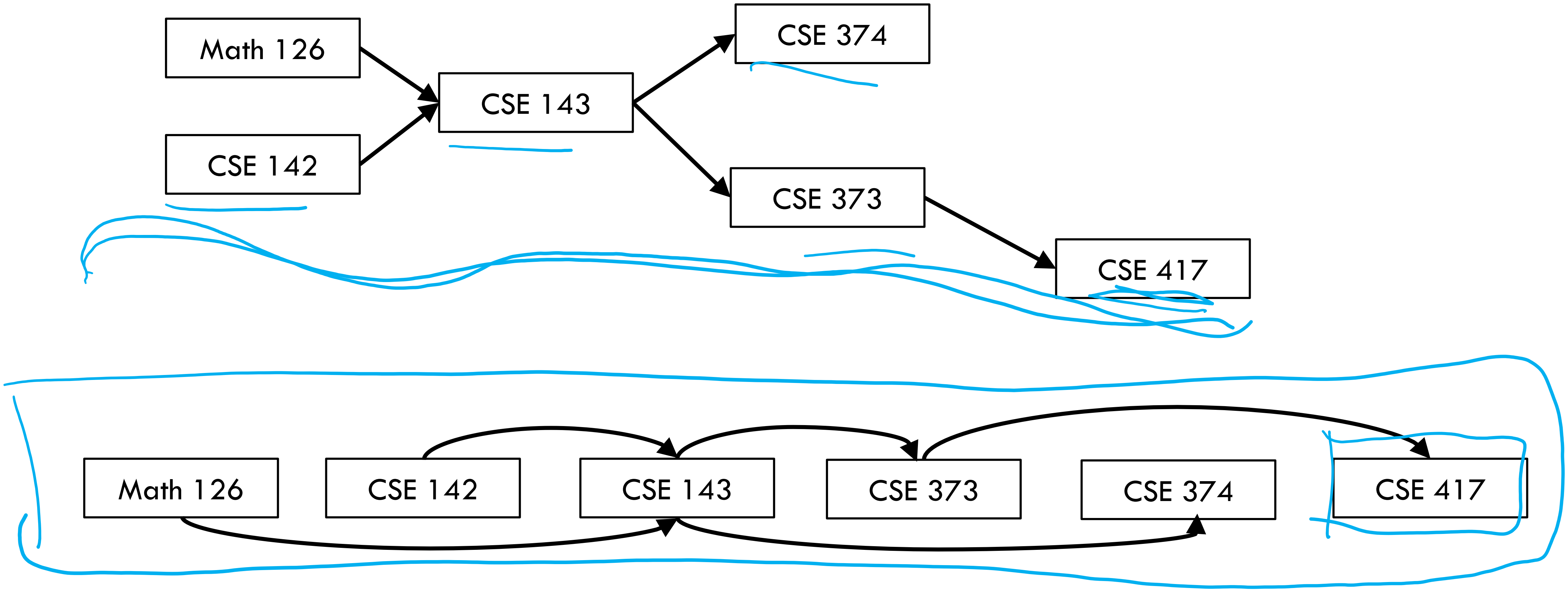
Uses:

Compiling multiple files

Graduating

Topological Ordering

A course prerequisite chart and a possible topological ordering.



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. Similar idea on undirected graphs on HW2.

Topological sort

You saw an algorithm in 373

Important thing: runs 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")

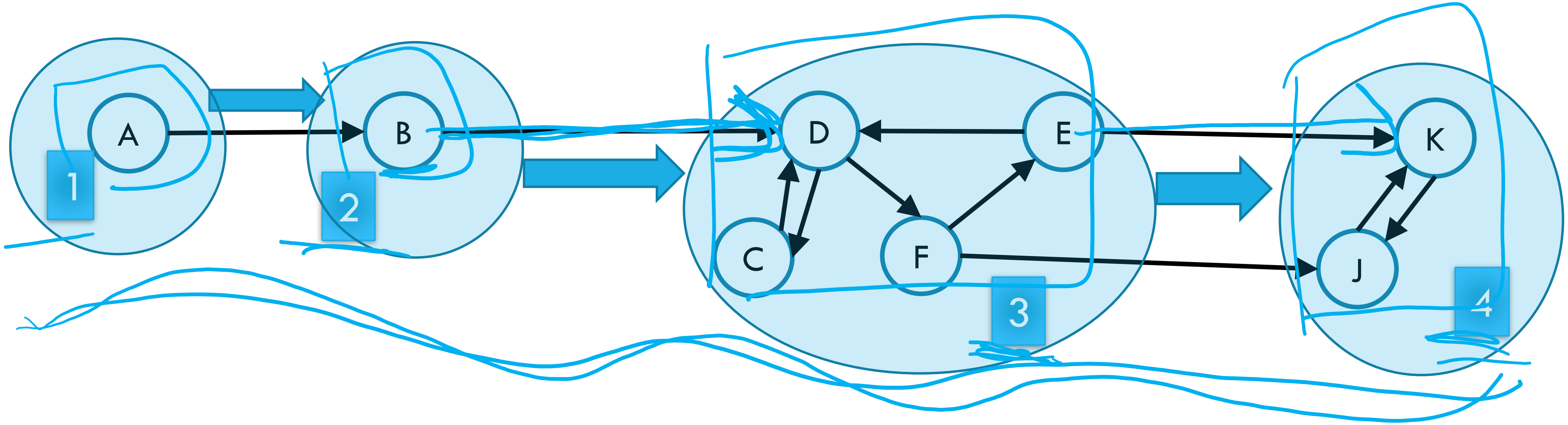
G^{SCC} has a vertex for every SCC of G

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 of the problems on HW2 does this – it walks you through the process of designing an algorithm by:

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

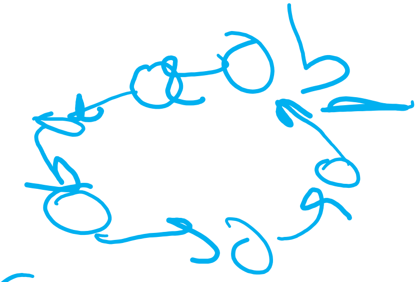
Otherwise go back to step 1 and try again.

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?

users

What are the edges?

directed edge

$u \to v$ if u follows v

What are we looking for?

cycle

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?

Rides

What are the edges?

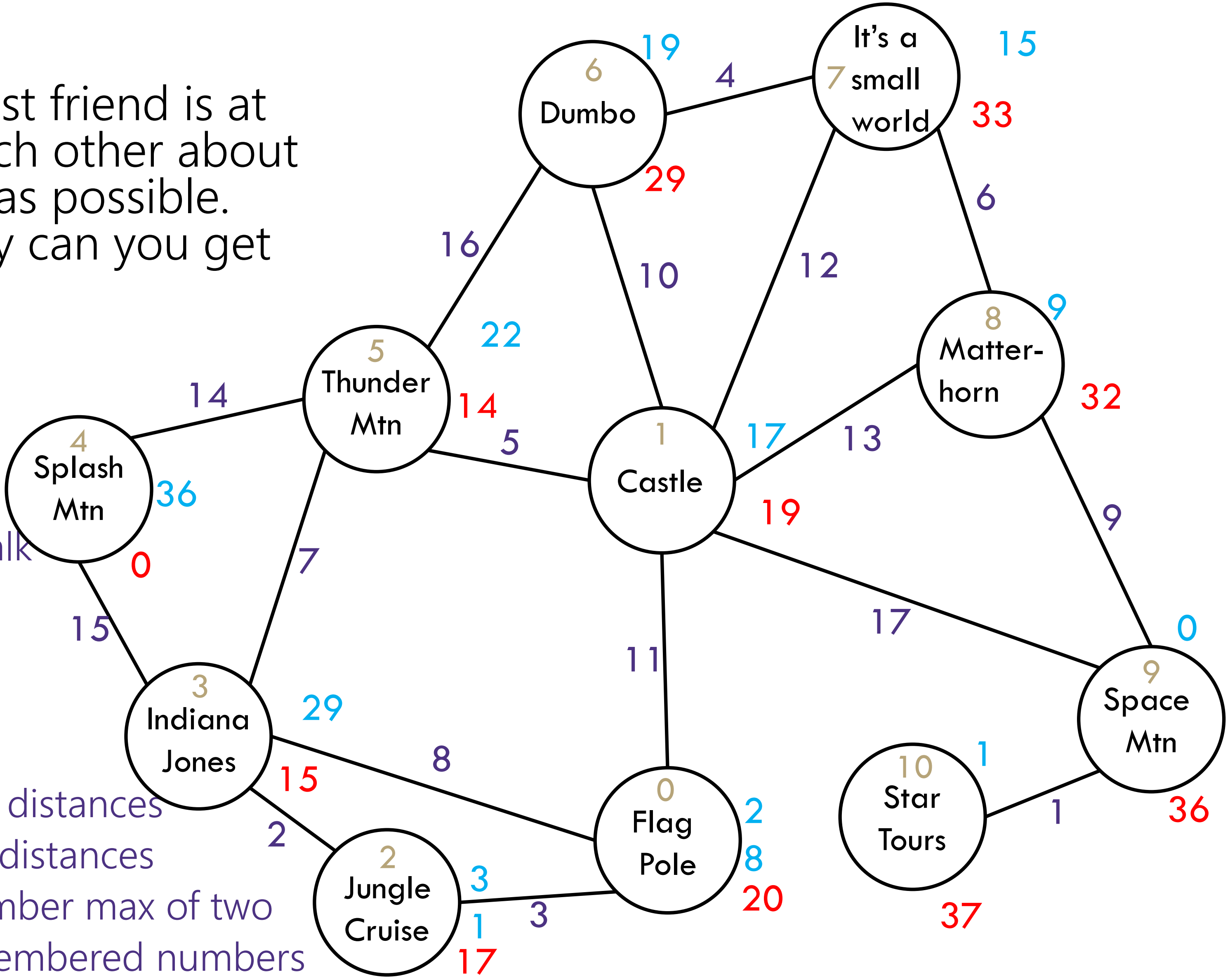
Walkways with how long it would take to walk

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!