

# DFS and Graph Modeling

CSE 421 Spring 26  
Lecture 5

# The Big Result

## Bipartite (also called "2-colorable")

A graph is bipartite if and only if it has no odd cycles.

Proof:

Lemma 1 says if a graph has an odd cycle, then it's not bipartite (or in contrapositive form, if a graph is bipartite, then it has no odd cycles)

Lemma 3 says if a graph has no odd cycles then it is bipartite.

**Lemma 1: If a graph contains an odd cycle, then it is not bipartite.**

**Lemma 3: If a graph has no odd-length cycles, then it is bipartite.**

# Algorithm $\rightarrow$ proof

The final theorem statement doesn't know about the algorithm – we used the algorithm to prove a graph theory fact!

# Wrapping it up

```
BipartiteCheck(graph) //assumes graph is connected!  
  toVisit.enqueue(first vertex)  
  mark first vertex as seen  
  toVisit.enqueue(end-of-layer-marker)  
  l=1  
  while(toVisit is not empty)  
    current = toVisit.dequeue()  
    if(current == end-of-layer-marker)  
      l++  
      toVisit.enqueue(end-of-layer-marker)  
  current.layer = l  
  for (v : current.neighbors())  
    if (v is not seen)  
      mark v as seen  
      toVisit.enqueue(v)  
    else //v is seen  
      if(v.layer == current.layer)  
        return "not bipartite" //intra-level edge  
  return "bipartite" //no intra-level edges
```

On homework, you can tell us "assume BipartiteCheck was modified to handle disconnected graphs" if you want those handled automatically.  
You just add a wrapper, like you've seen in 332.

# Testing Bipartiteness (two statements)

Our algorithm should answer "yes" or "no"

"yes  $G$  is bipartite" or "no  $G$  isn't bipartite"

Whenever this happens, you'll have two parts to the proof:

If the right answer is yes, then the algorithm says yes.

If the right answer is no, then the algorithm says no.

OR

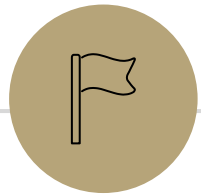
If the right answer is yes, then the algorithm says yes.

If the algorithm says yes, then the right answer is yes.

# Proving Algorithm Correct

If the graph is bipartite, then by Lemma 1 there is no odd cycle. So by the contrapositive of lemma 2, we get no intra-level edges when we run BFS, thus the algorithm (correctly) returns the graph is bipartite.

If the algorithm returns that the graph is bipartite, then we cannot have any intra-level edges (since we check every edge in the course of the algorithm). We proved earlier that there are no edges skipping more than one level. So if we assign odd levels to "red" and even levels to "blue" the algorithm has verified that there are no edges between vertices of the same color. So the graph is bipartite by definition.



**DFS**



# Only Discussing High-Level Ideas

All the details are in the end of the slide deck

# DFS vs. BFS

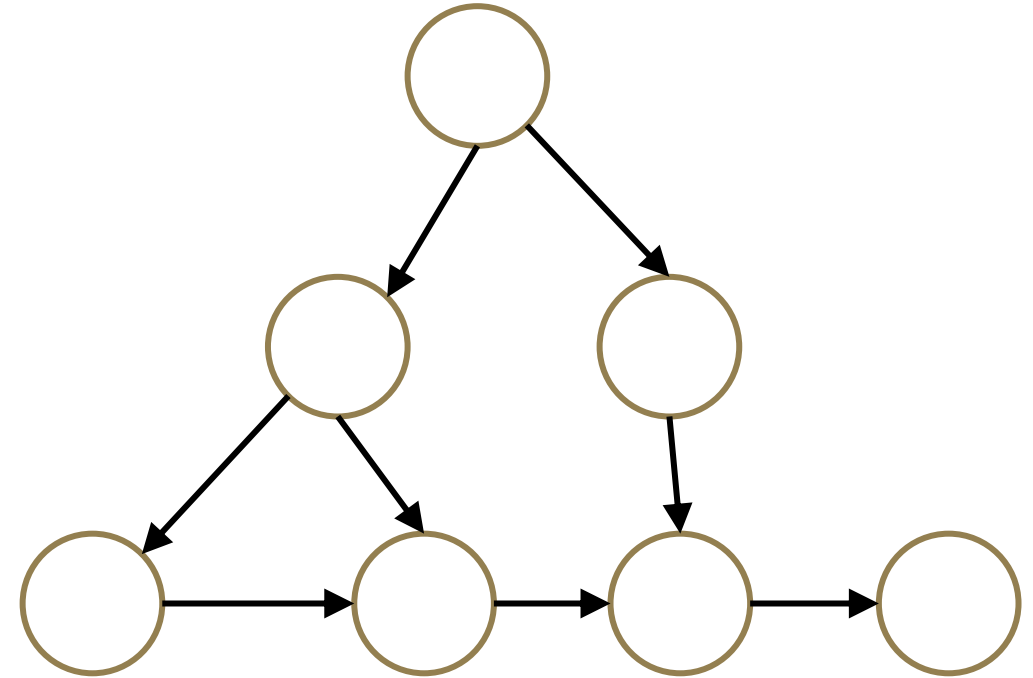
In BFS, we explored a graph  
"level-wise"

We explored everything  
next to the starting vertex.

Then we explored  
everything one step further  
away.

Then everything one step  
further

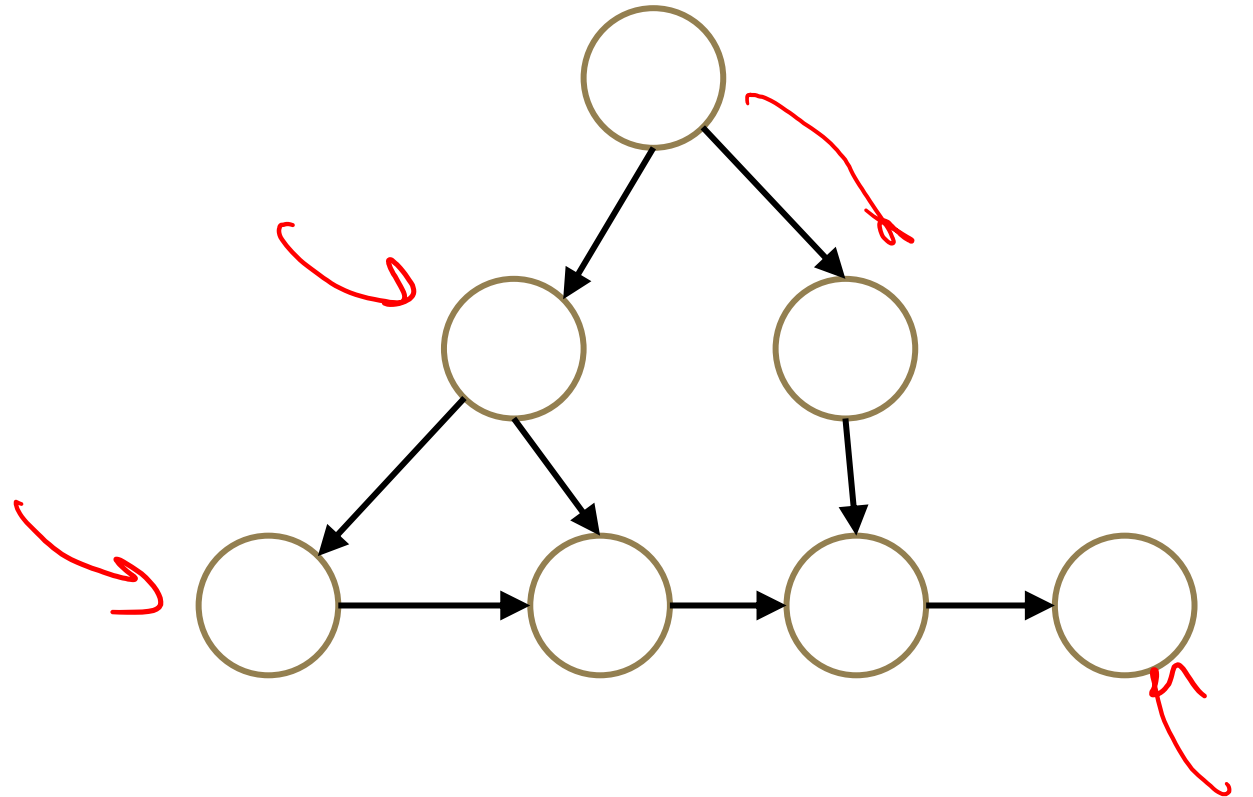
...



# DFS vs. BFS (DFS example)

In DFS, we explore deep into the graph.

We try to find new (undiscovered) nodes, then "backtrack" when we're out of new ones.



# DFS – pseudocode

In 332, you might have seen two versions: an iterative version (like BFS, but replace the queue with a stack) and a recursive version (use the call stack to manage the search).

The iterative version is a really nice object lesson in data structures.

No one actually writes DFS that way (except in data structures courses).

You'll basically always see the recursive version instead. (using the call stack instead of a user-created stack data structure)

# DFS – pseudocode

Instead of using an explicit stack, we're going to use recursion  
The call stack is going to be our stack.

We want to explore as deeply as possible from each of our outgoing edges

DFS (u)

↳ Mark u as "seen"

↳ For each edge (u,v) //leaving u

    If v is not "seen"

        ↳ DFS (v)

    End If

End For

# DFS – iterative vs. recursive

Both the explicit stack version and the recursive version “are” DFS.

For example, they can both traverse through the graph in the same fundamental way. You can use them for similar applications.

But they’re not identical – they actually use the stack in different ways. If you’re trying to convert from one to the other, you’ll have to think carefully to do it.

# Reaching Everything

One possible use of DFS is visiting every vertex

How can we make sure that happens?

What did you do for BFS when you had this problem?

Add a while loop, and call DFS from each vertex.

```
DFSWrapper (G)
```

```
  For each vertex u of G
```

```
    If u is not "seen"
```

```
      DFS (u)
```

```
    End If
```

```
  End For
```

```
DFS (u)
```

```
  Mark u as "seen"
```

```
  For each edge (u,v) //leaving u
```

```
    If v is not "seen"
```

```
      DFS (v)
```

```
    End If
```

```
  End For
```

# Bells and Whistles

Depending on your application, you may add a few extra lines to the DFS code to compute the thing you want.

Usually just an extra variable or two per vertex.

For today's application, we need to know what order vertices come onto and off of the stack.

```
DFS (u)                                DFSWrapper (G)
  Mark u as "seen"                      ↪ counter = 1
  ↪ u.start = counter++                  For each vertex u of G
  For each edge (u,v) //leaving u        If u is not "seen"
    If v is not "seen"                   DFS (u)
      DFS (v)                             End If
    End If                                End For
  End If
  End For
  ↪ u.end = counter++
```

# Edge Classification

When we use DFS to search through a graph, we'll have different "kinds" of edges.

Like when we did BFS, we had:

- Edges that went from level  $i$  to level  $i + 1$
- Intra-level edges.

DFS is a bit more complicated.

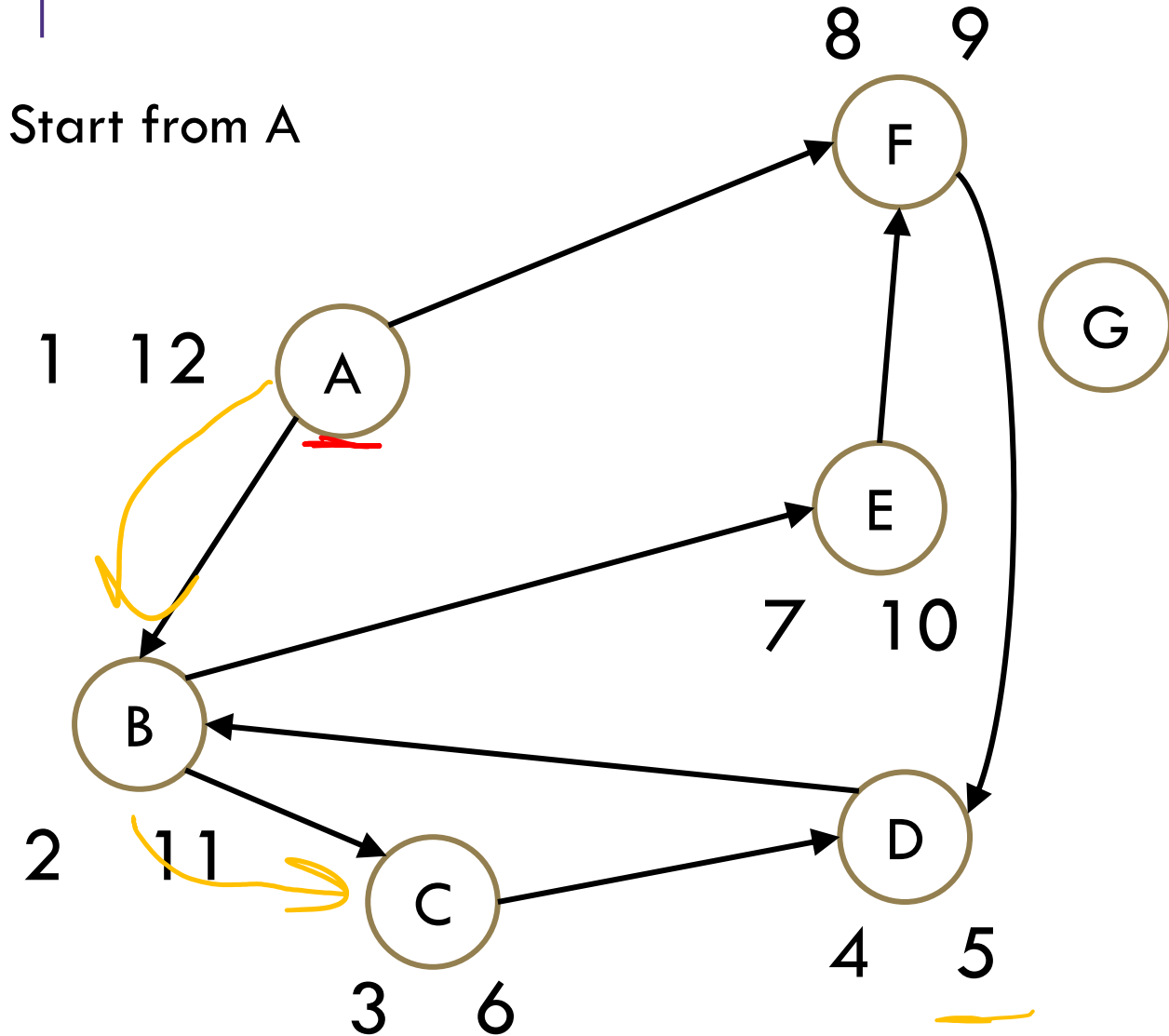
You don't have to memorize the details.

What we want you to know:

"pre" and "post" (aka "stop"/"end") times when you come onto and off of the stack

Will tell you what "type" of edge you are

# DFS: start/end



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

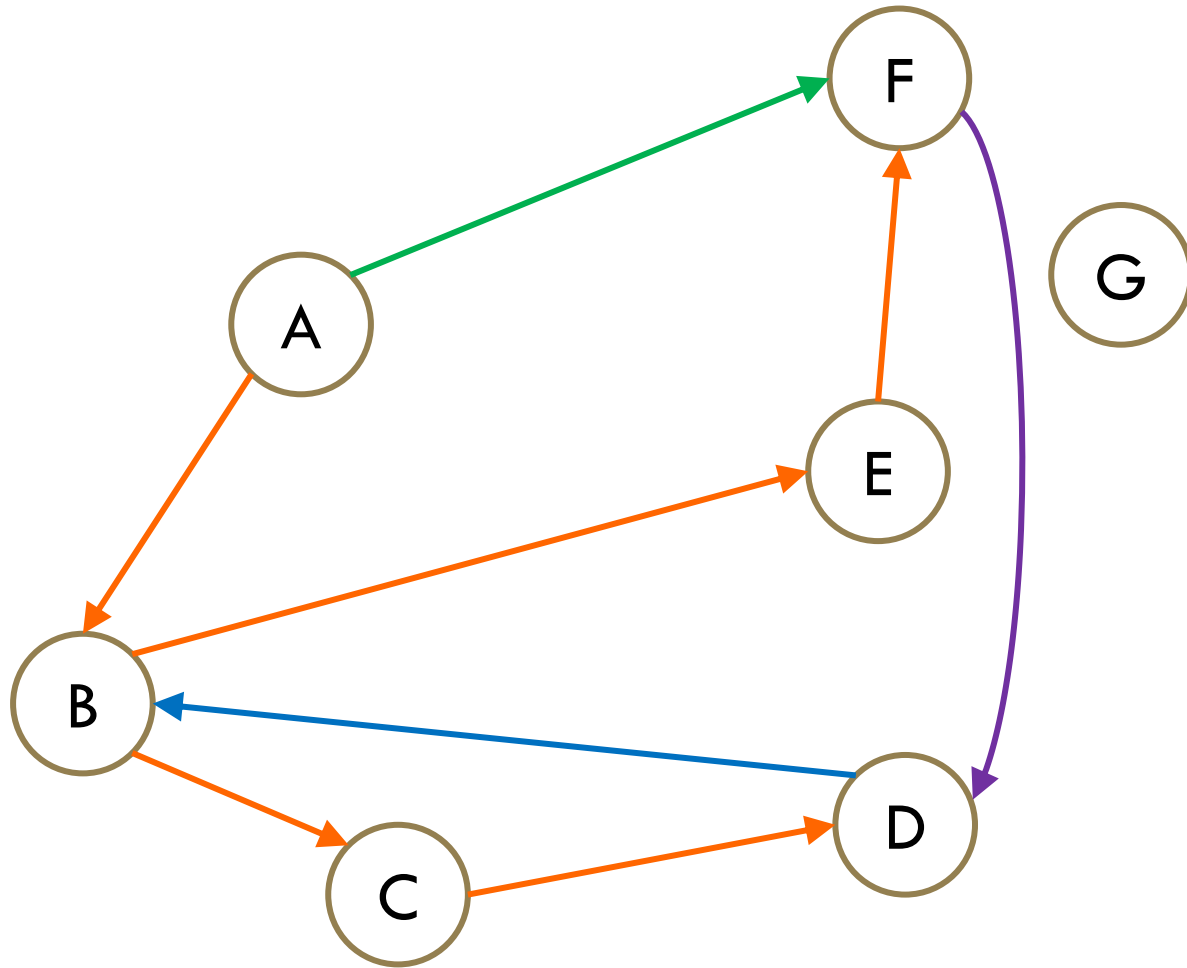
`u.end = counter++`

Vertex: F  
Last edge used: (F,D)

Vertex: E  
Last edge used: (E,F)

Vertex: B  
Last edge used: (B,E)

Vertex: A  
Last edge used: (A,F)



DFS (u)

Mark u as "seen"

`u.start = counter++`

For each edge (u,v) //leaving u

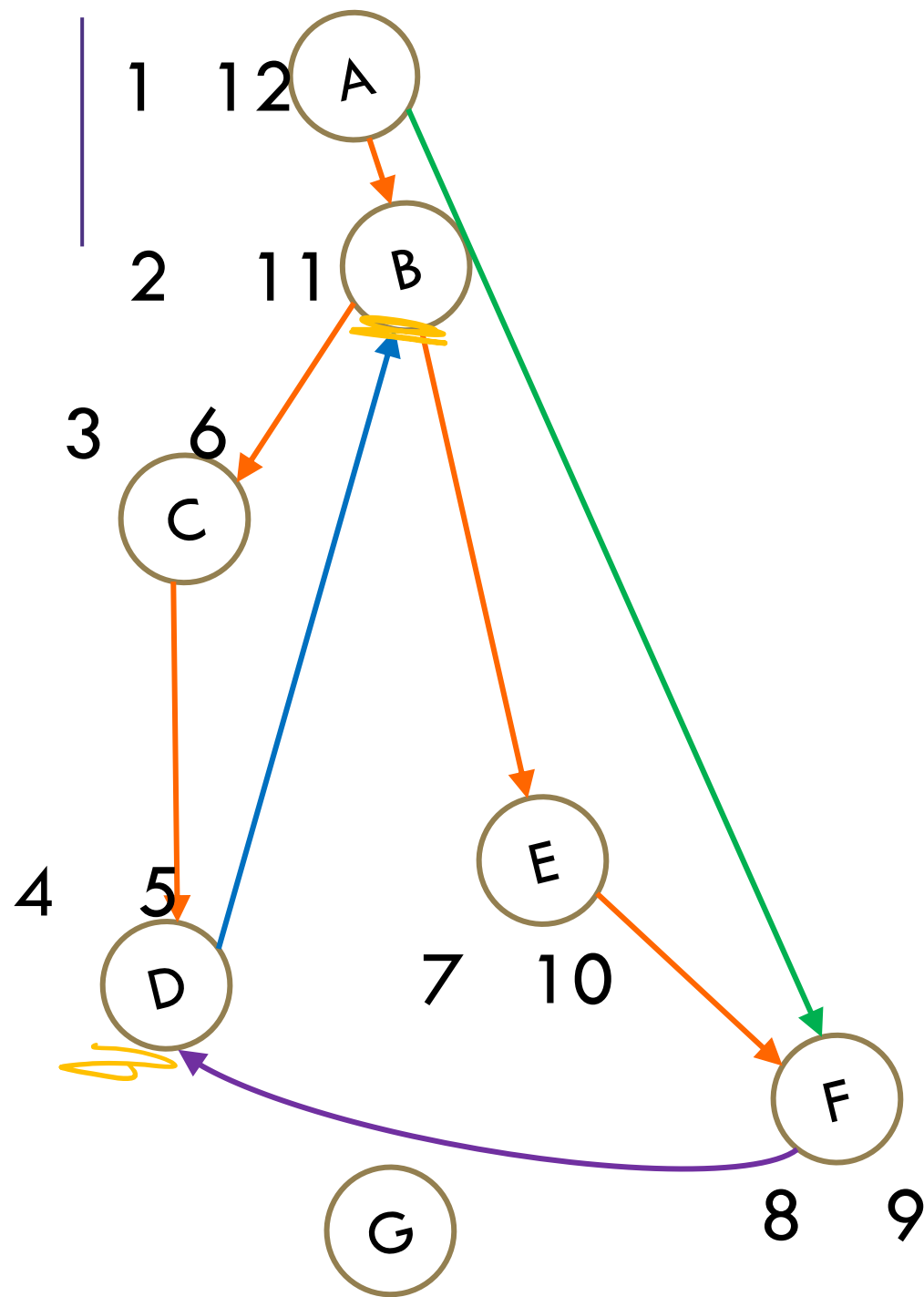
    If v is not "seen"

        DFS (v)

    End If

End For

`u.end = counter++`



The orange edges (the ones where we discovered a new vertex) form a tree!\*

We call them **tree edges**.

That blue edge went from a descendent to an ancestor B was still on the stack when we found (B,D).

We call them **back edges**.

The green edge went from an ancestor to a descendant F was put on and come off the stack between putting A on the stack and finding (A,F)

We call them **forward edges**.

The purple edge went...some other way.

D had been on and come off the stack before we found F or (F,D)

We call those **cross edges**.

\*Conditions apply. Sometimes the graph is a forest. But we call them tree edges no matter what.

# 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 $v.start < v.end < u.start < u.end$

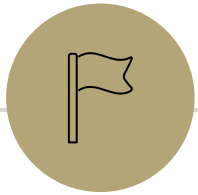
The third column doesn't look like it encompasses all possibilities.

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.

We will give you this table on exams!

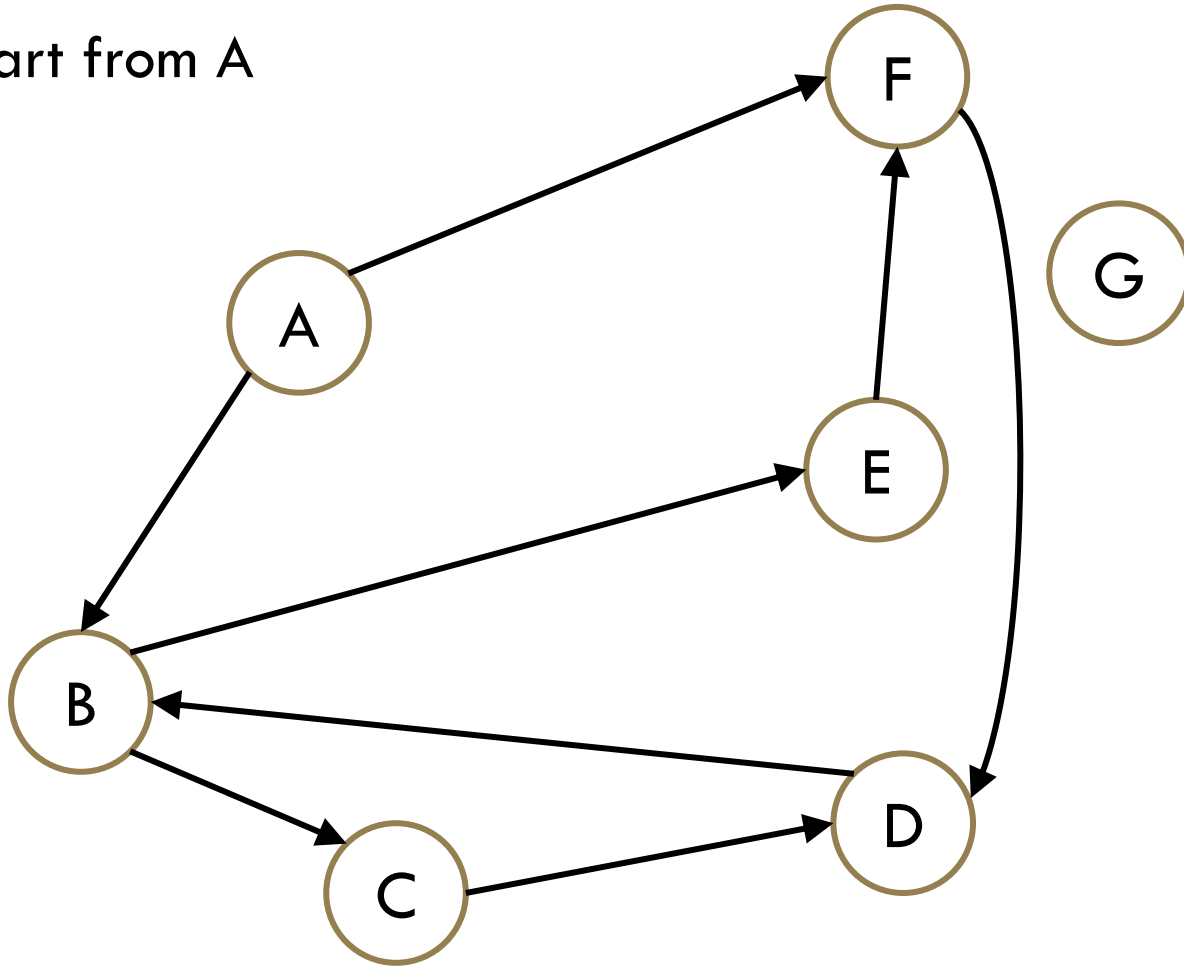


**A lot of Details: DFS**

---

# Running DFS

Start from A



DFS (u)

Mark u as "seen"

For each edge (u,v) //leaving u

If v is not "seen"

DFS (v)

End If

End For

Vertex: F  
Last edge used: (F,D)

Vertex: E  
Last edge used: (E,F)

Vertex: B  
Last edge used: (B,E)

Vertex: A  
Last edge used: (A,F)

# DFS Discovery

DFS (u)

Mark u as "seen"

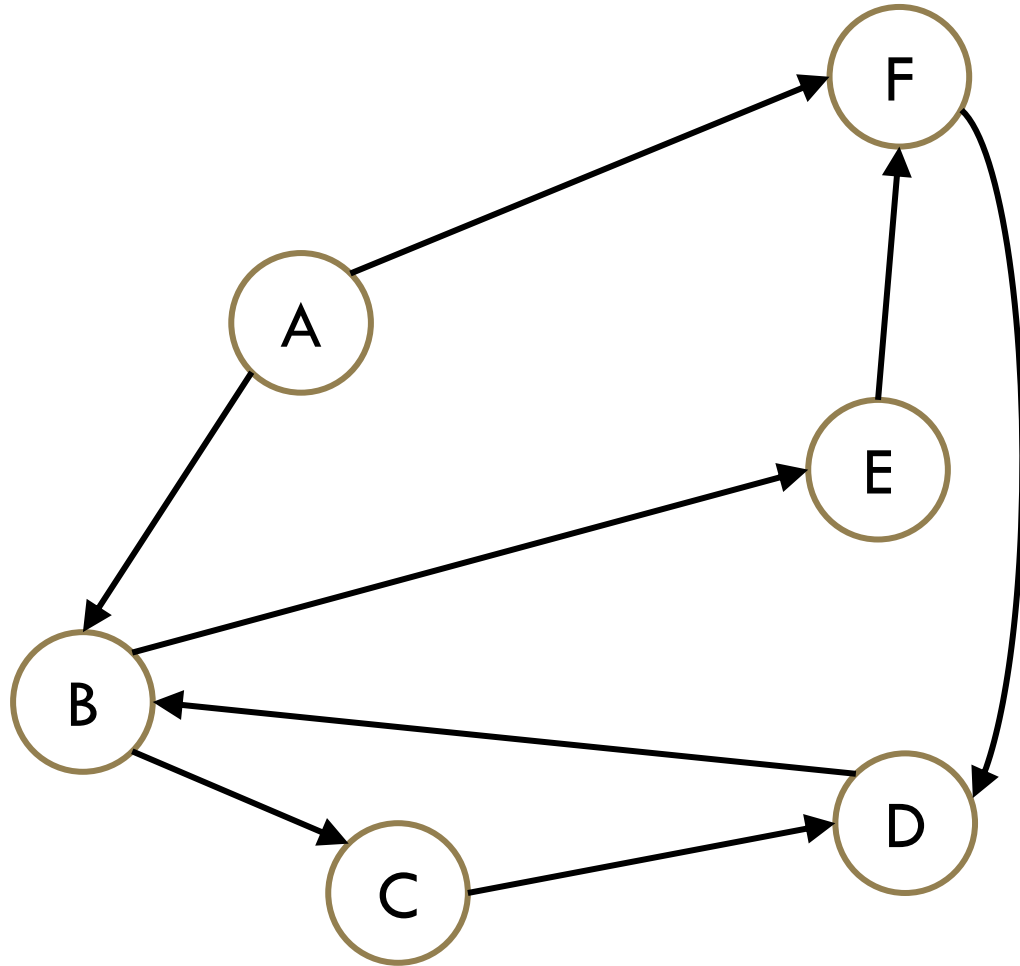
For each edge (u,v) //leaving u

If v is not "seen"

DFS (v)

End If

End For



**G** HEY!

We missed something!

DFS discovery

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

# Try it Yourself!

DFSWrapper (G)

`counter = 0`

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  $u$

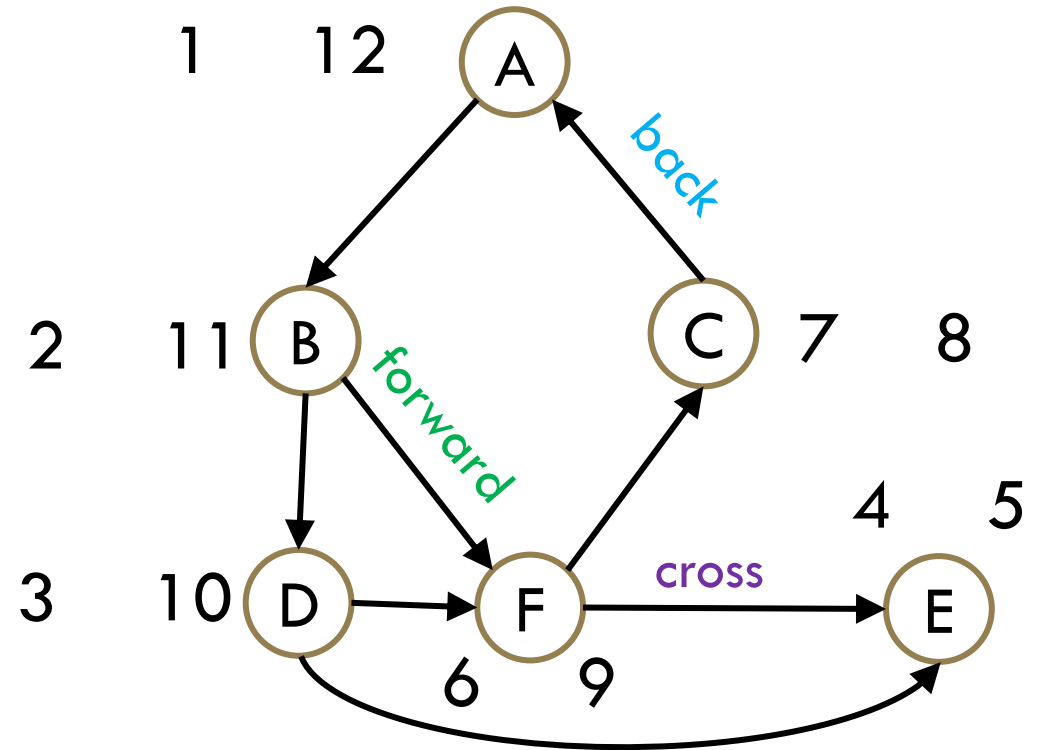
    If  $v$  is not "seen"

        DFS( $v$ )

    End If

End For

`u.end = counter++`



# Actually Using DFS

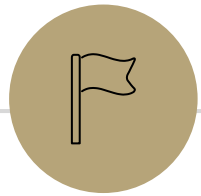
Here's a claim that will let us use DFS for something!

## Back Edge Characterization

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

Not responsible for the details, just notice the similarity to the type of proof we did before (differentiate types of edges, use them to find information)

More examples in any of the books on the resources tab. (also this proof in last quarter's slides).



# Graph Search Takeaways

---

# BFS/DFS caveats and cautions

Edge classifications are different for directed graphs and undirected graphs.

Be careful with details

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!)

We focused on connected graphs--you'll only find everything that you can reach from the start vertex.

# Your Takeaways

When searching through a graph, order matters!

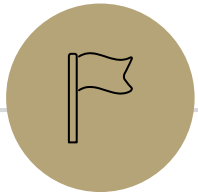
BFS and DFS do different things!

BFS/DFS algorithms usually keep track of extra information/calculate something at each vertex/use edge classification to solve the problem. A few extra bells and whistles in the code, but usually little more.

DFS can solve a wide-variety of problems, but the algorithms tend to be subtle.

BFS a lot more intuitive, start there if you can.

BE CAREFUL with directed/undirected graphs. The algorithms aren't always easy to convert.



# Solving new problems: Graph Modeling

---

# Summary – Graph Search Applications

## BFS

Shortest Paths (unweighted graphs)

## DFS

Cycle detection (directed graphs)

Topological sort

Strongly connected components

## EITHER

2-coloring

Connected components (undirected)

Usually use BFS –  
easier to understand.

# 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.

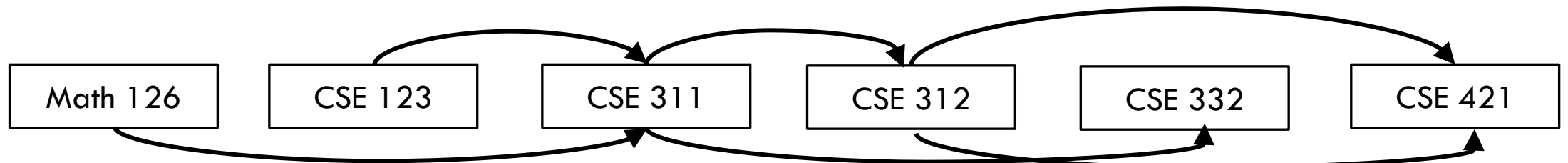
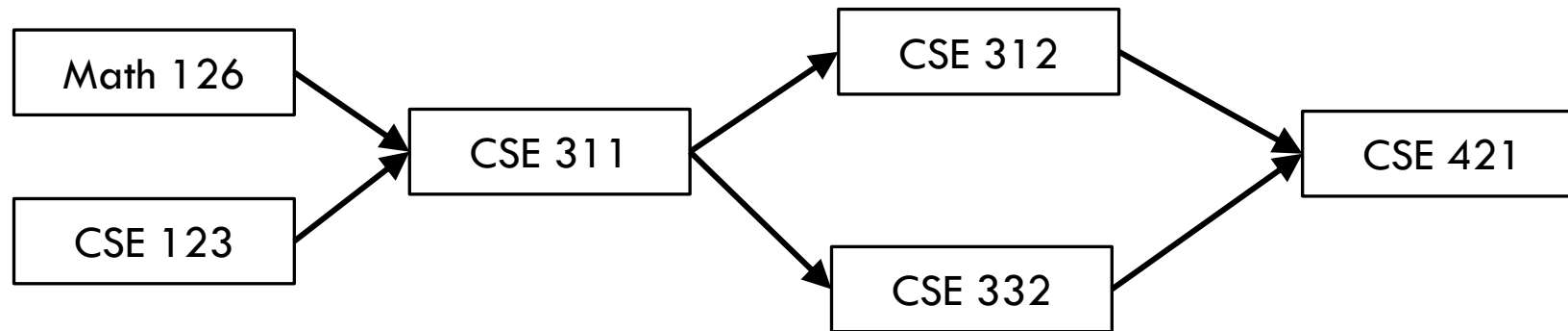
Uses:

Compiling multiple files

Graduating

# Topological Ordering

A course prerequisite chart and a possible topological ordering.



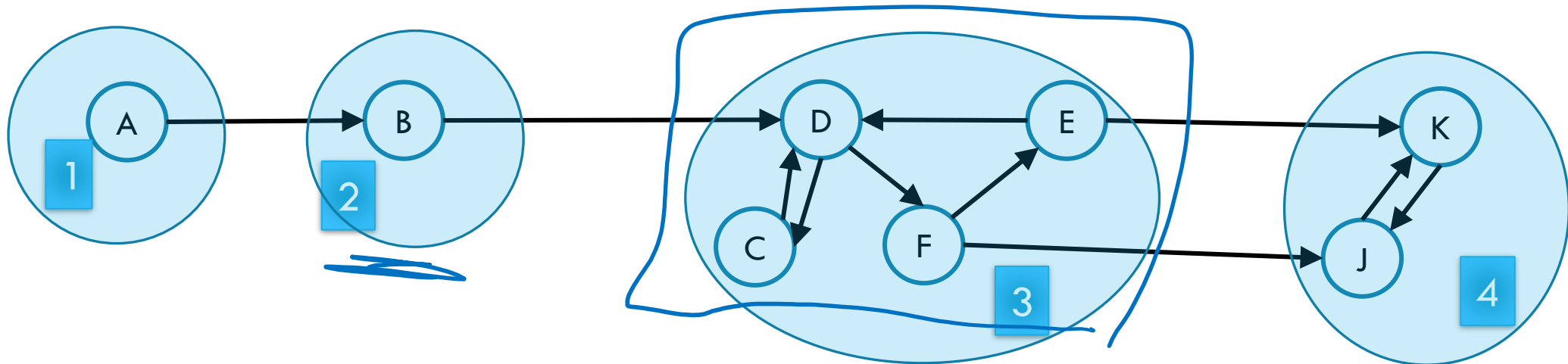
# Problem 2

$\{C, D, E\}$

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 probably saw an algorithm in 332; if not, just believe that we have a library for it.

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 (combining)

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  $G$

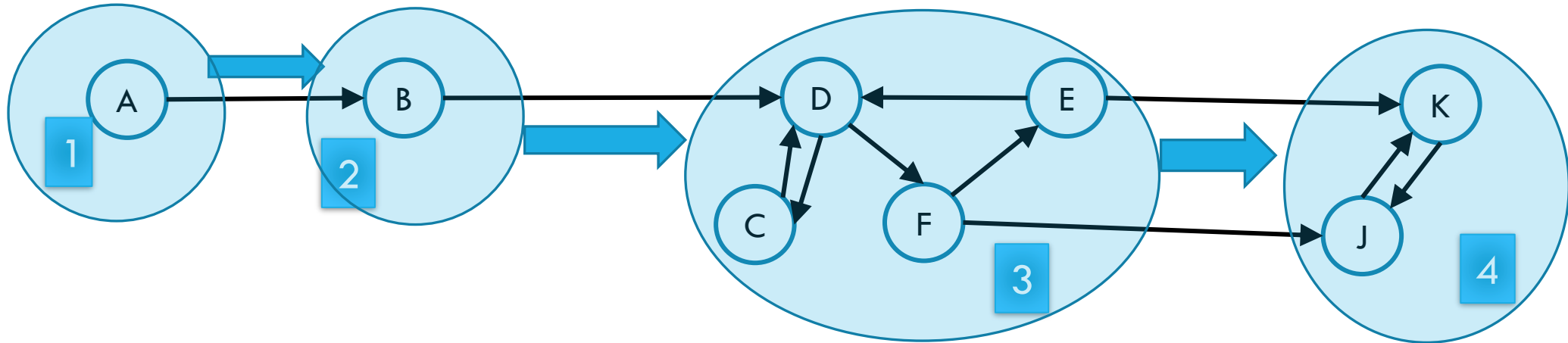
There's an edge from  $u$  to  $v$  in  $G^{SCC}$  if and only if there's an edge in  $G$  from a vertex in  $u$  to a vertex in  $v$ .

# 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

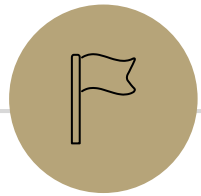
Not a common task – most graph problems have been asked before.

When you need to do it, Robbie recommends:

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

HW problem walks you through 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  $G^{SCC}$ ).



# Modeling



# 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.

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, Convr. Users on Convr 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 (answer)

You've made a new social networking app, Convr. Users on Convr 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 (answer)

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

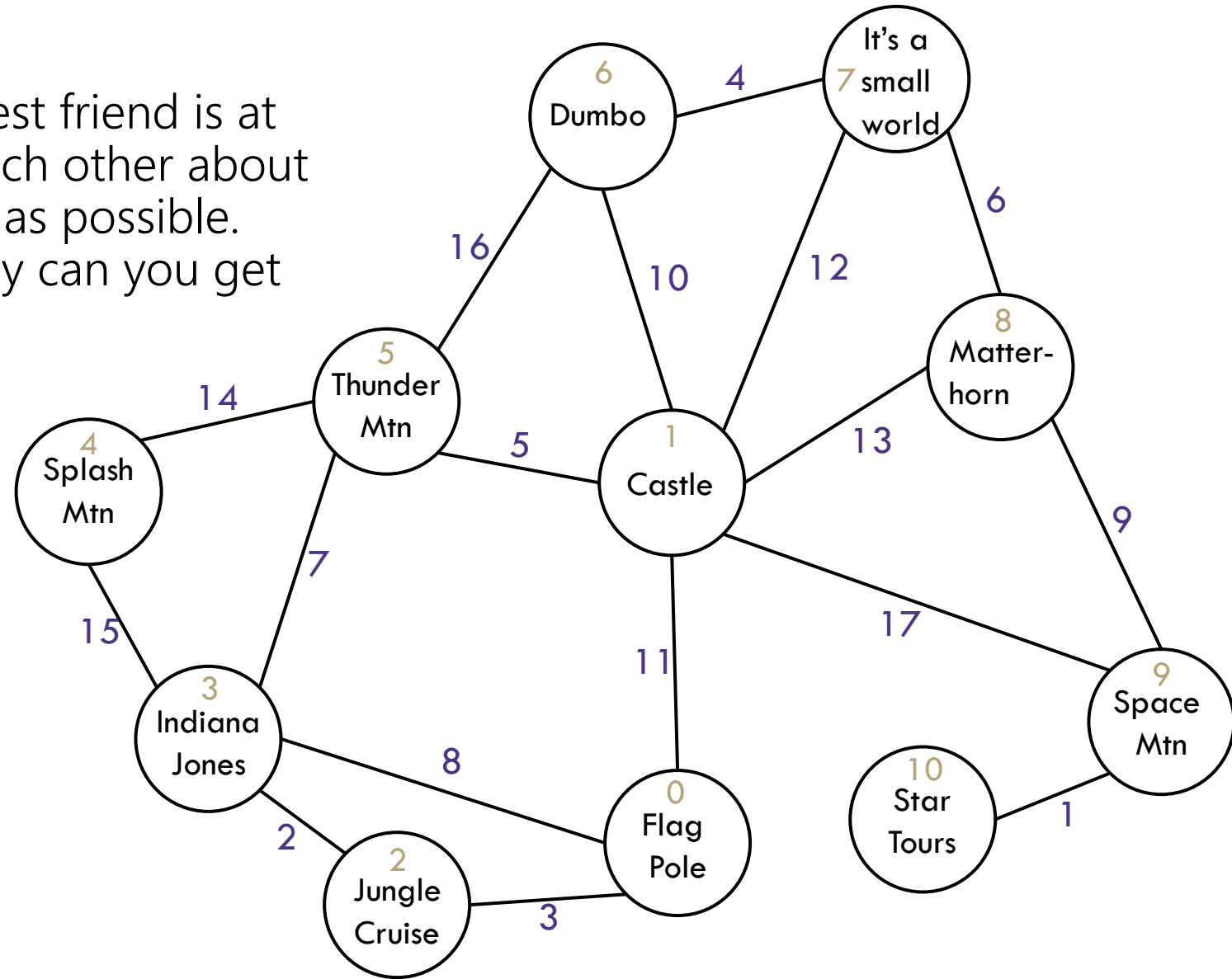
You are at Splash Mountain. Your best friend is at Space Mountain. You have to tell each other about your experiences in person as soon as possible. Where do you meet and how quickly can you get there?

What are the vertices?

What are the edges?

What are we looking for?

What do we run?



# Scenario #3 (answer)

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?

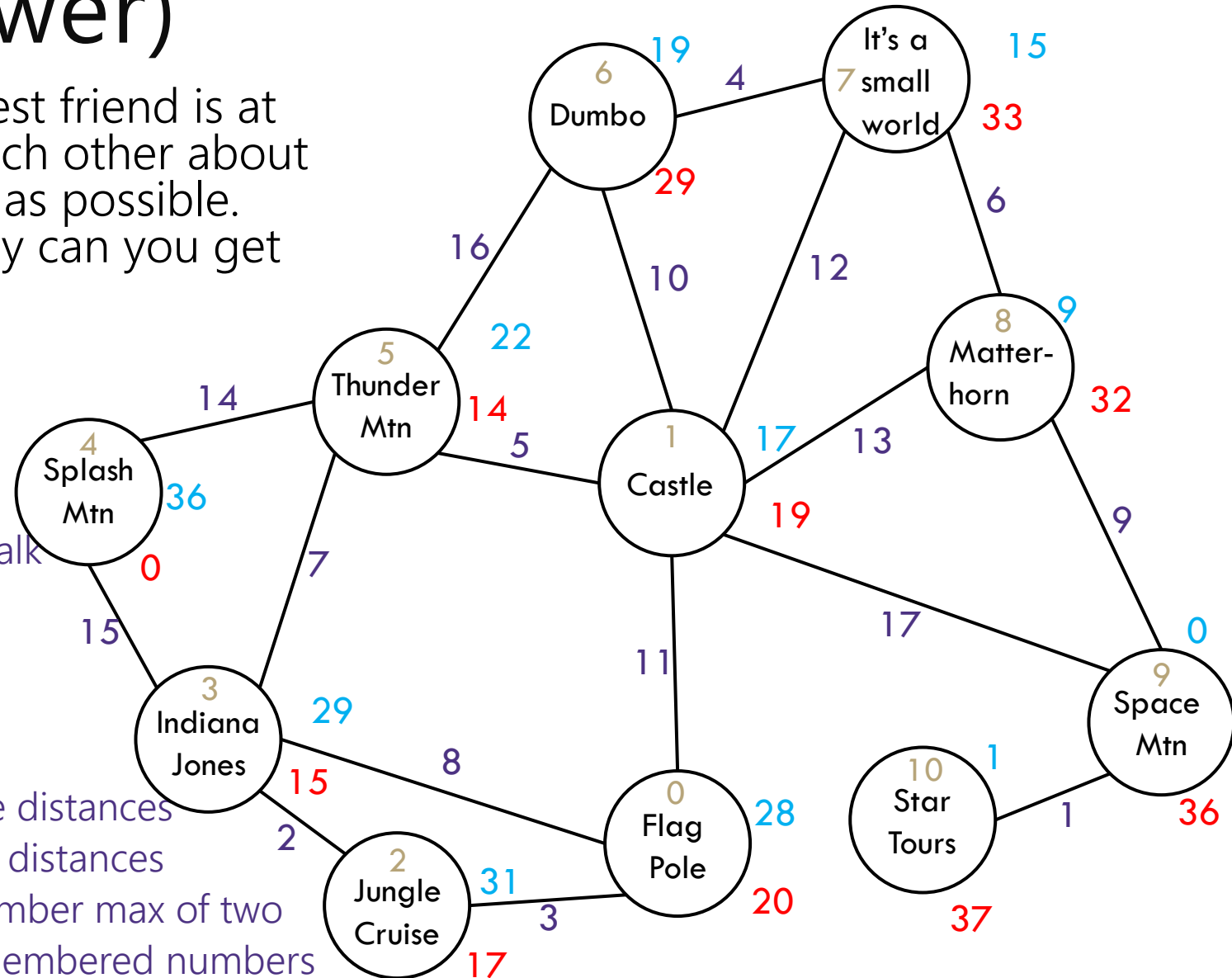
Walkways with how long it would take to walk

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!