New version of slides went up a few minutes ago.

Depth First Search

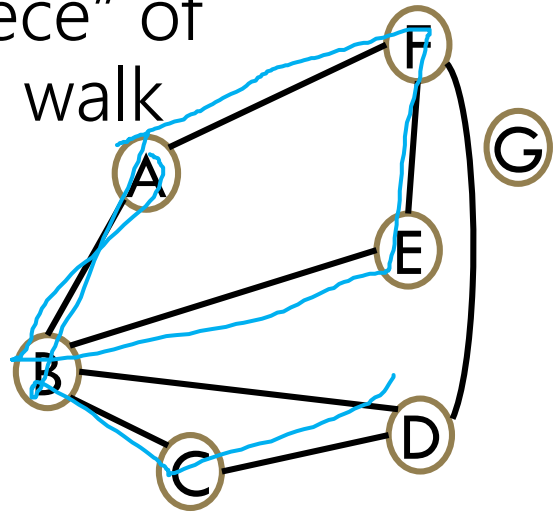
CSE 417 22AU
Lecture 7

Some Graph Vocabulary

A **walk** is a list of vertices where you can get from each vertex to the next along an edge. (repeats allowed).

A **path** is a list of vertices where you can get from each vertex to the next, and there are no repeats.

In an *undirected graph* a **connected component** is a "piece" of the graph, it's a vertex and anything you can reach on a walk starting at that vertex.



DFS vs. BFS

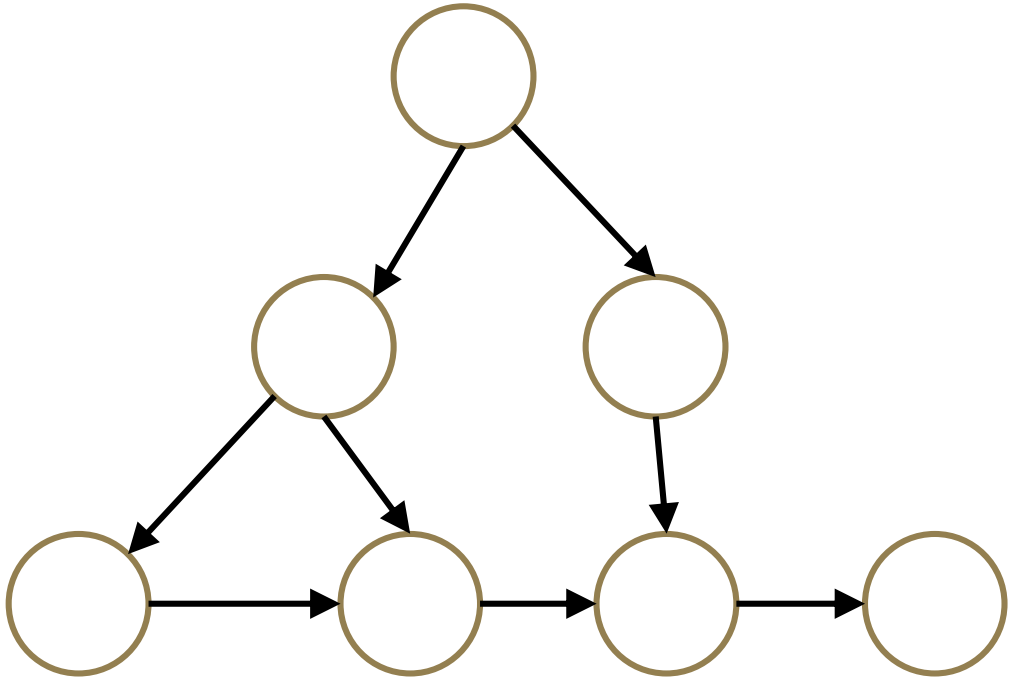
In BFS, we explored a graph "level-wise"

We explored everything next to the starting vertex.

Then we explored everything one step further away.

Then everything one step further

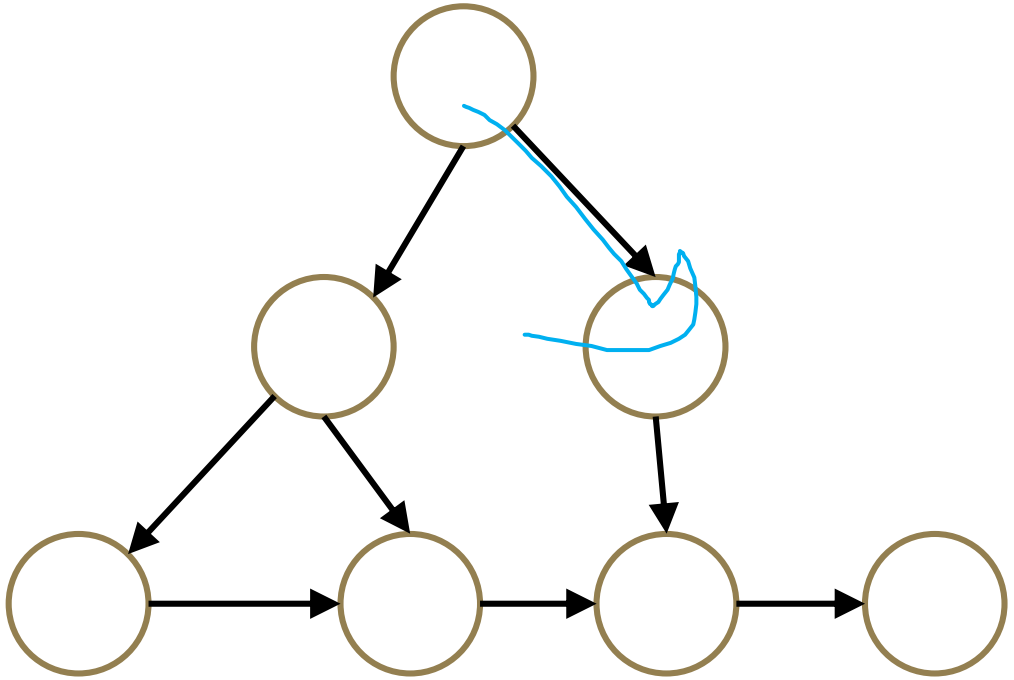
…



DFS vs. BFS

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 373, you probably took your BFS code, replaced the queue with a stack and said “that’s the pseudocode.”

That’s a really nice object lesson in stacks.

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 the data structure stack)

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 – pseudocode

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

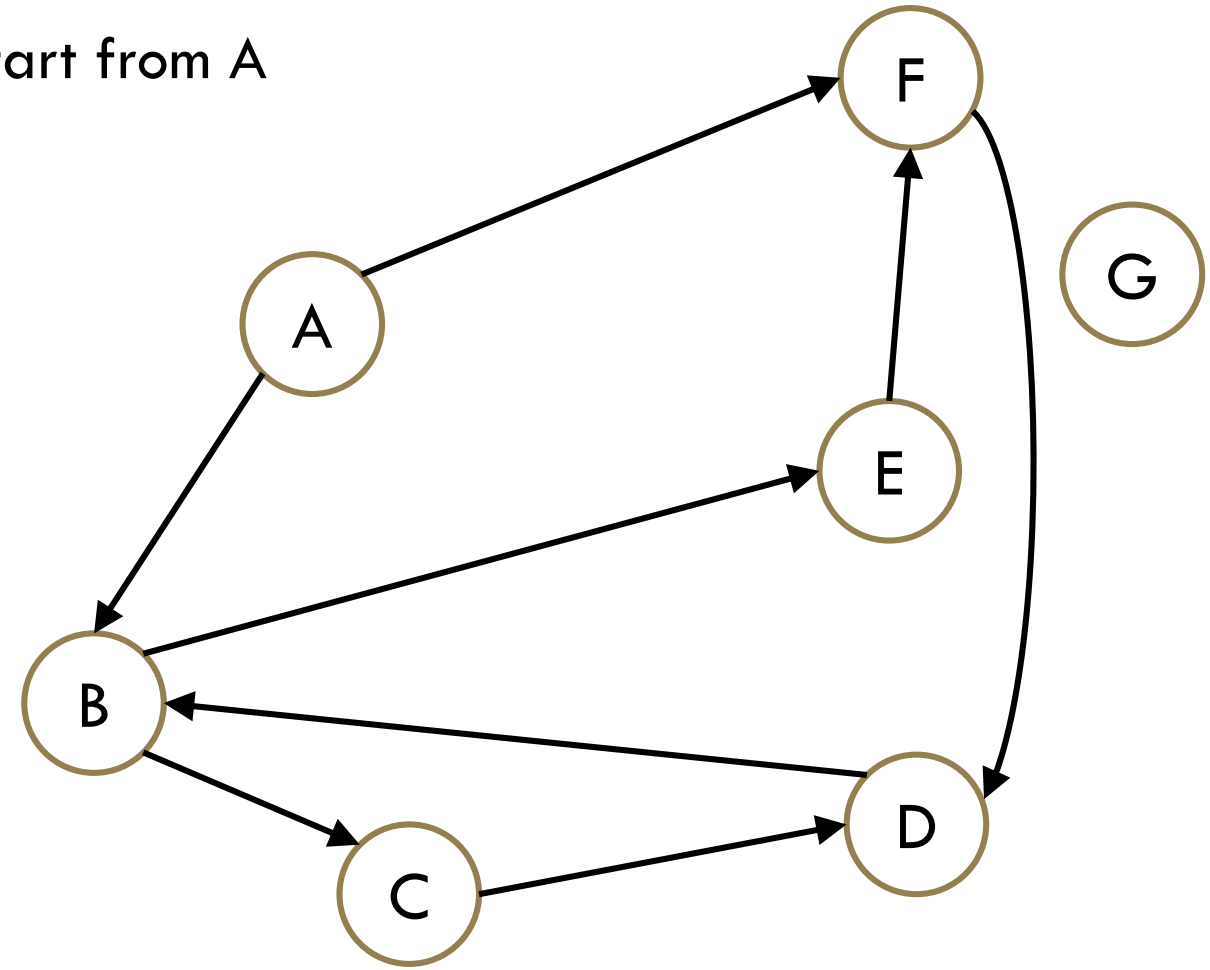
For example, they can both traverse through the graph in the same fundamental way. You can use them for similar applications.

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.

Running DFS

```
DFS(u)
      Mark u as "seen"
      For each edge (u,v) //leaving u
            If v is not "seen"
                  DFS(v)
            End If
      End For
```

Start from A



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

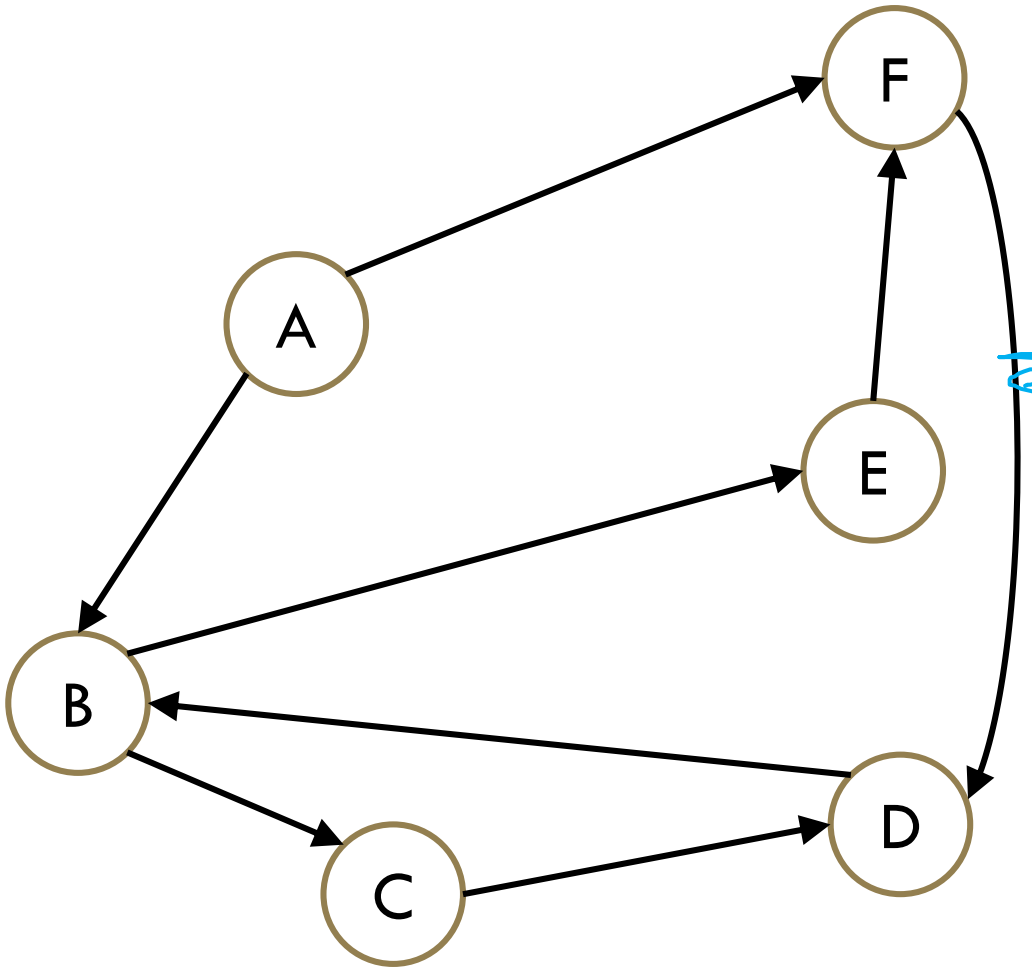
Vertex: E
Last edge used: (E,F)

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

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

Running DFS

```
DFS(u)
      Mark u as "seen"
      For each edge (u,v) //leaving u
            If v is not "seen"
                  DFS(v)
            End If
      End For
```



HEY!

We missed something!

DFS discovery

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

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)
    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++
```

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

Edge Classification

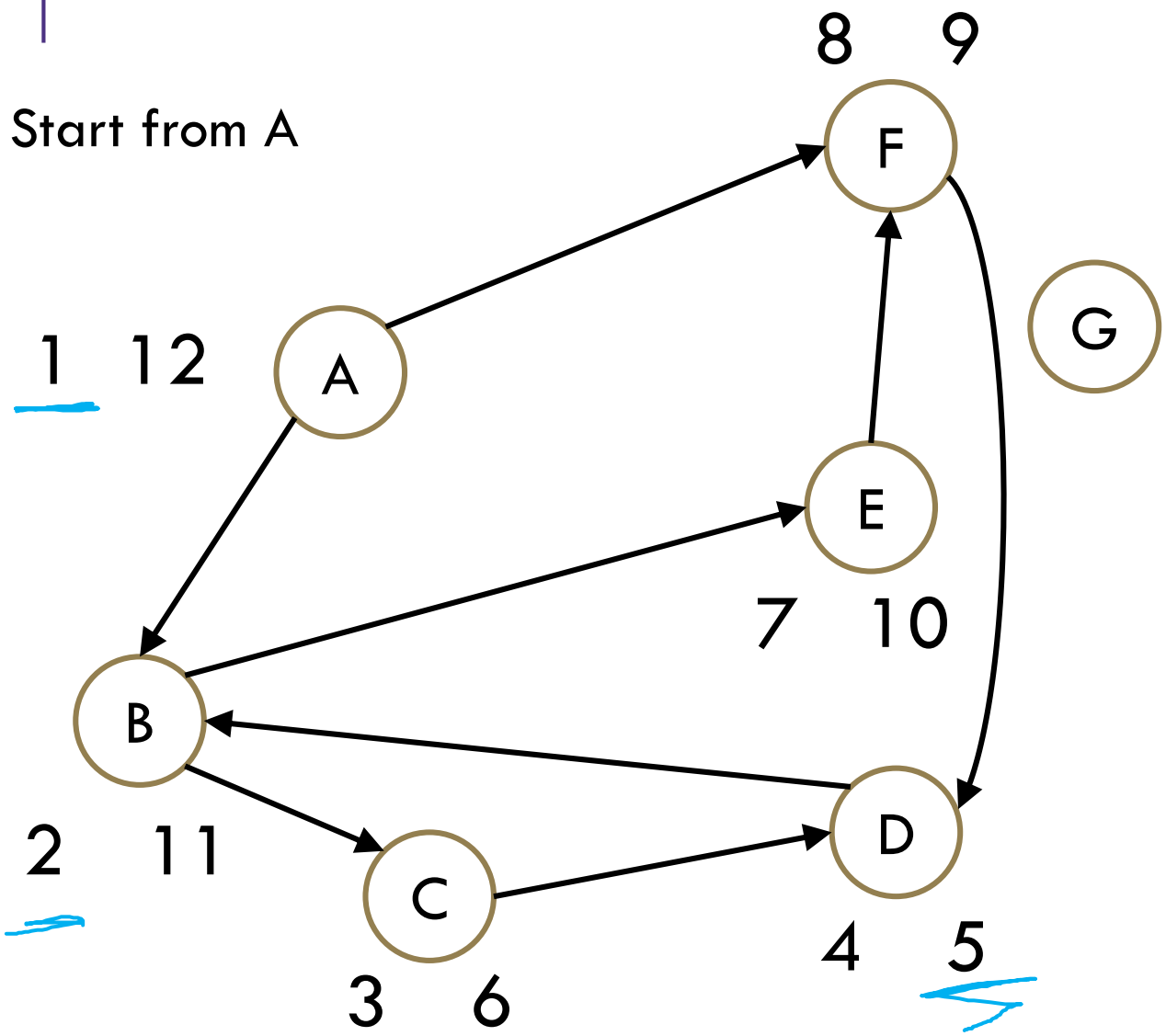
When we use DFS to search through a graph, we'll have different "kinds" of edges.
Like when we did BFS, we had:
Edges that went from level i to level $i + 1$
Intra-level edges.

We'll do a few examples to help classify the edges.
Then do an application of the classification.

Our goal: find a cycle in a directed graph.

Running DFS

Start from A



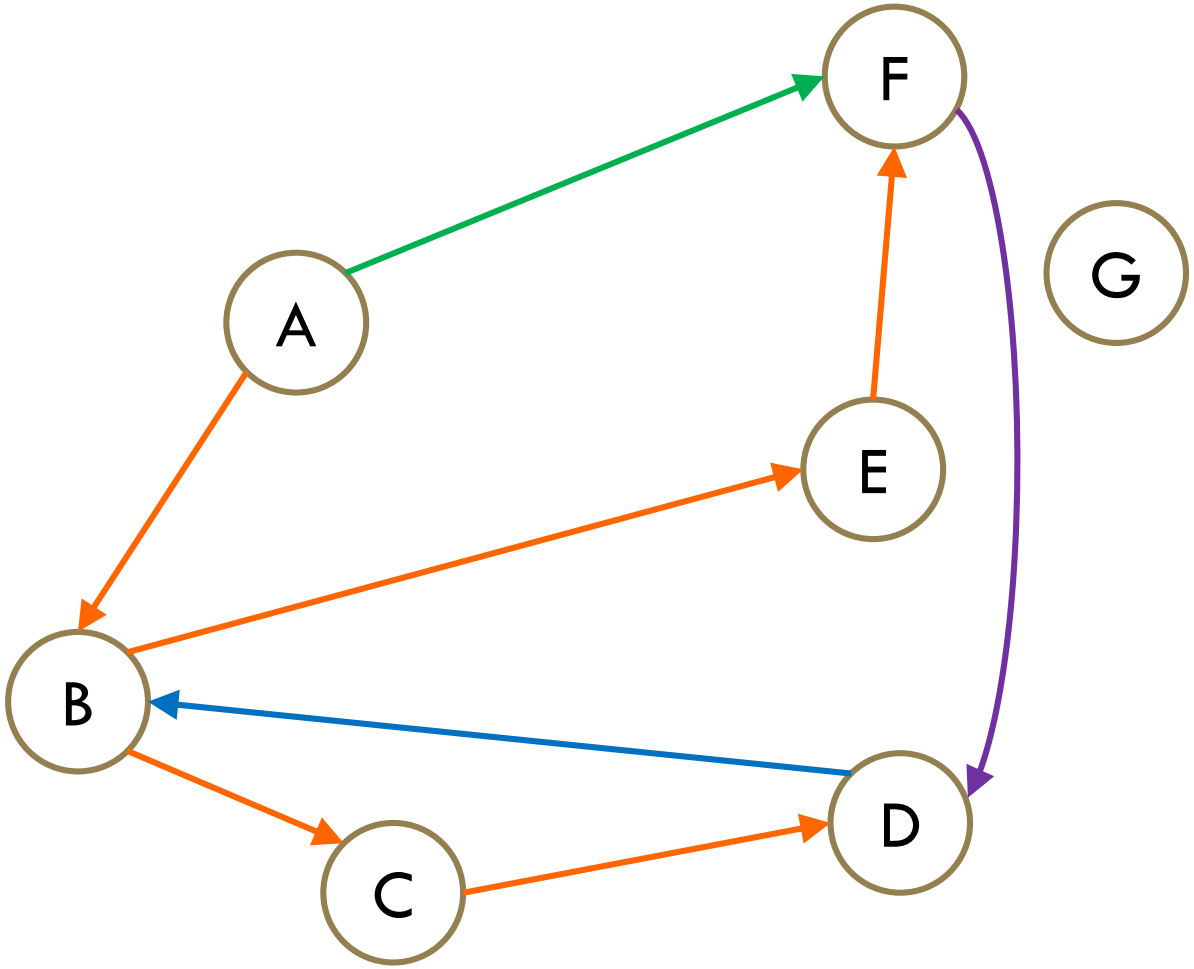

```
DFS(u)
        Mark u as "seen"
        u.start = counter++
        For each edge (u,v) //leaving u
              If v is not "seen"
                    DFS(v)
              End If
        End For
        u.end = counter++
```

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

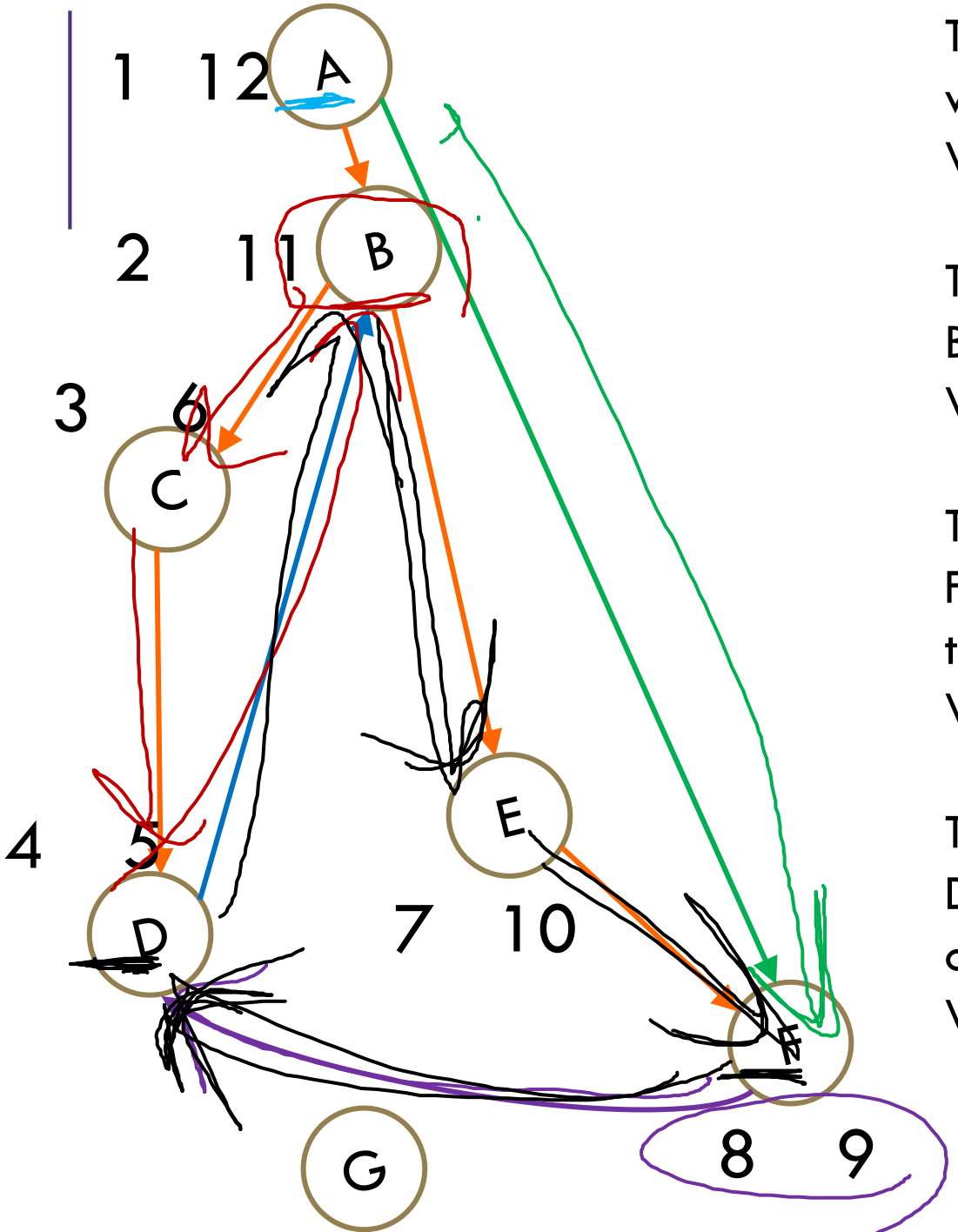
Vertex: E
Last edge used: (E,F)

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 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:

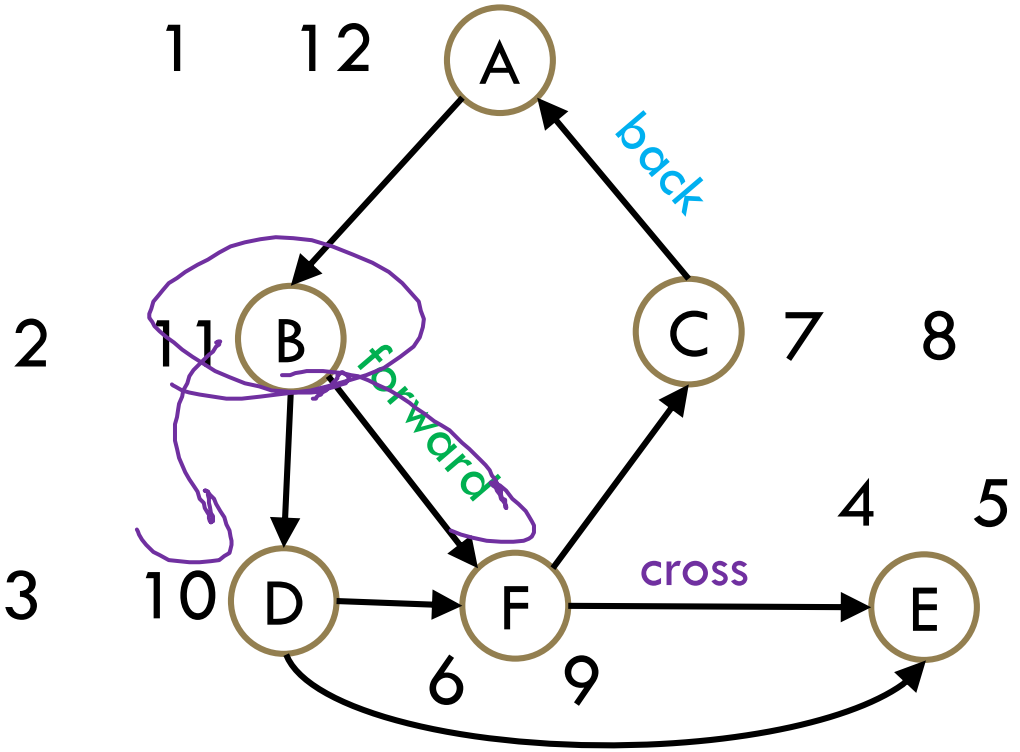
e.g. `u.start < v.start < u.end < v.end` is impossible.

And the rules of the algorithm eliminate some other possibilities.

Try it Yourselves!

```
DFSWrapper(G)
     counter = 0
     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.

Forward Direction

If DFS on a graph has a back edge then it has 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!

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.

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.

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 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.

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)

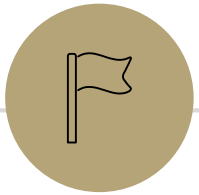
DFS

- Cycle detection (directed graphs)
- Topological sort
- Strongly connected components

EITHER

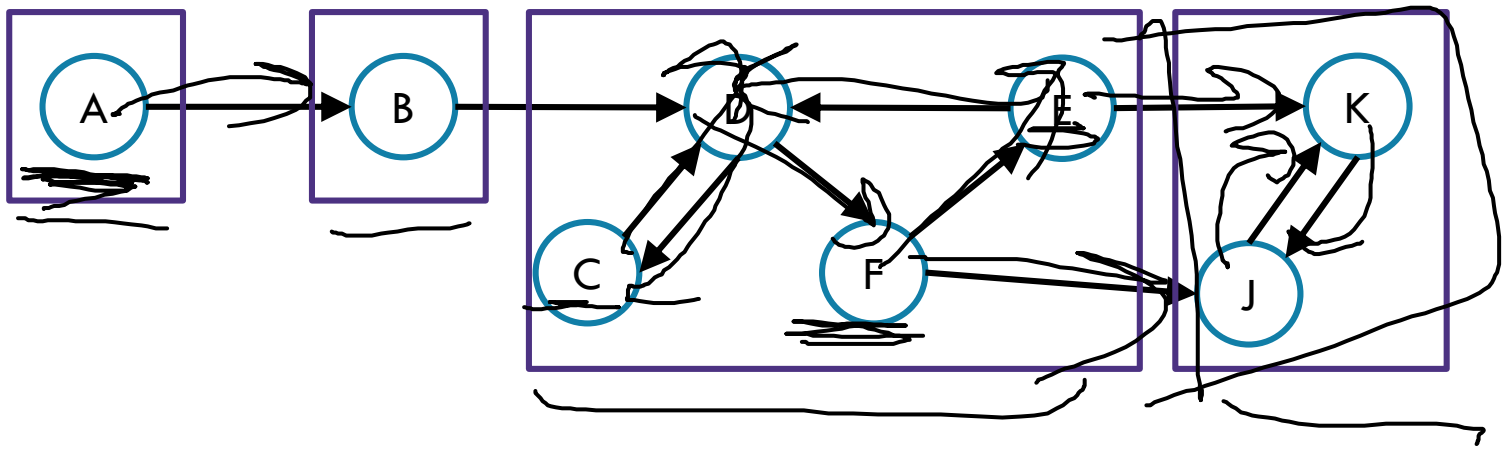
- 2-coloring
- Connected components (undirected)

Usually use BFS – easier to understand.



Graph Modeling

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 walk **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.

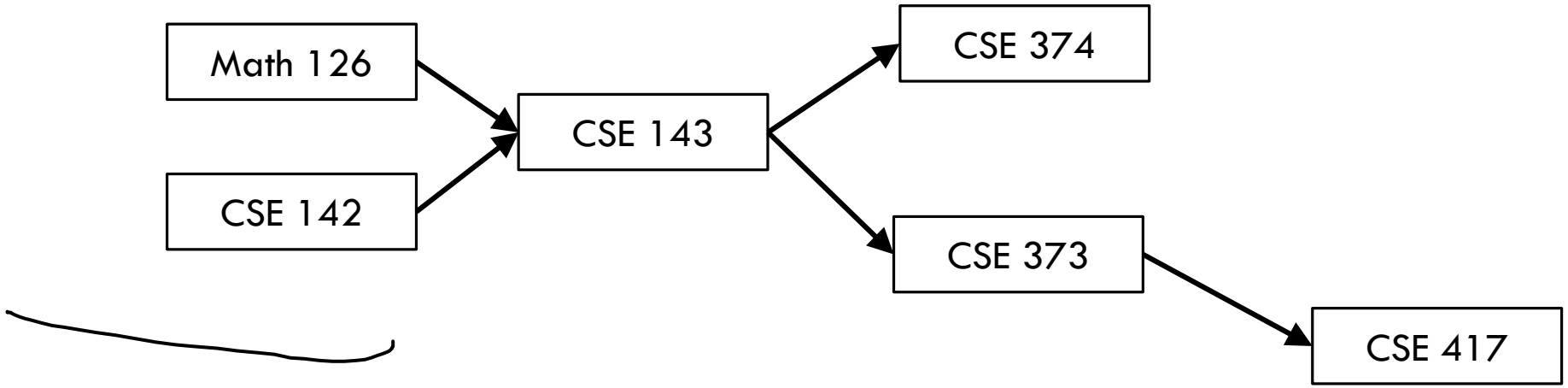
Uses:

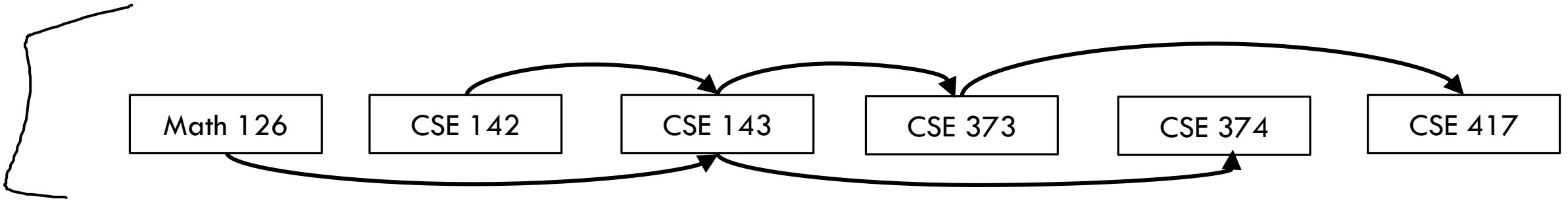
Compiling multiple files

Graduating

Topological Ordering

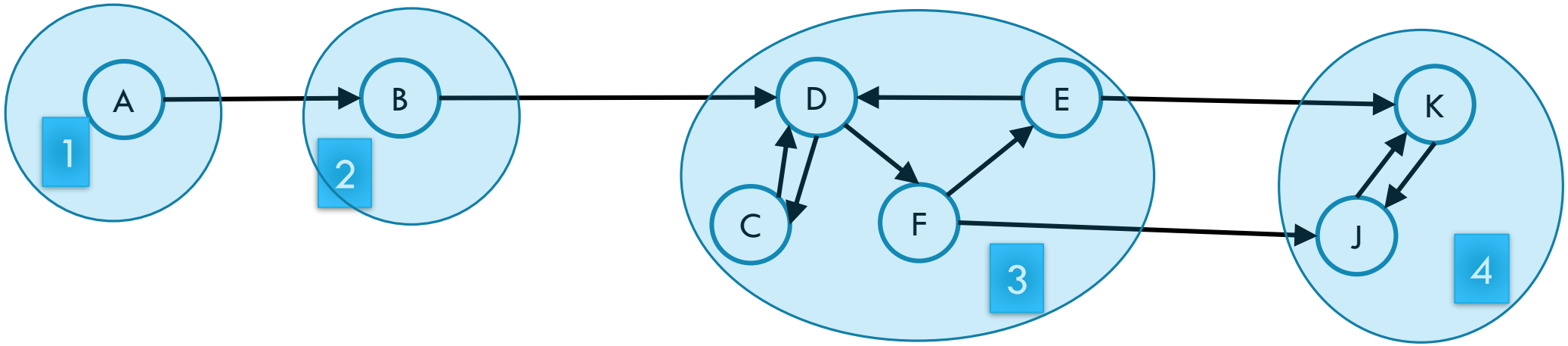
A course prerequisite chart and a possible topological ordering.





Problem 2

Given a graph, find its strongly connected components



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.

A homework problem will have a simple version.

Topological sort

You saw an algorithm in 373

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

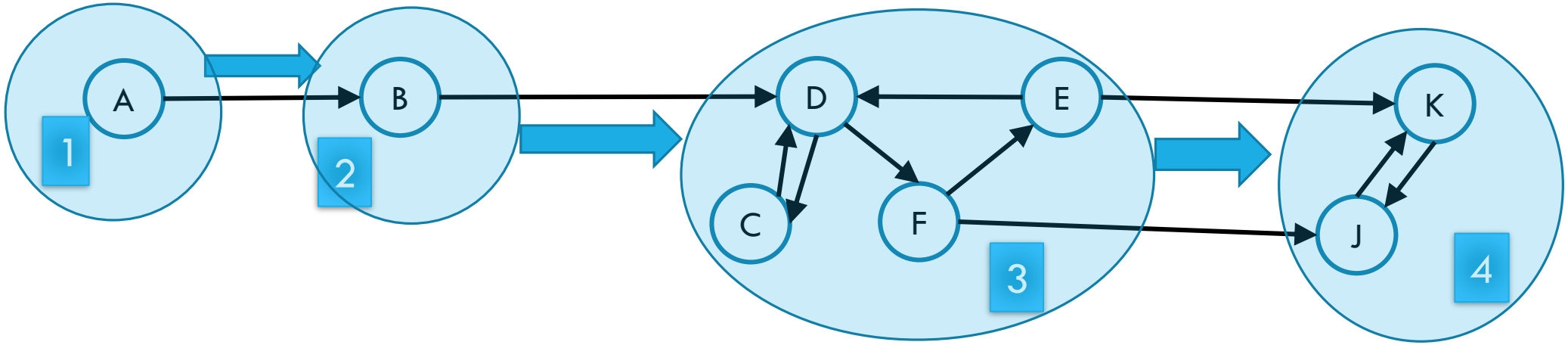
There's an edge from u to v in G^{SCC} if and only if there's an edge in G from a vertex in u to a vertex in v.

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).

A HW2 problem 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}).