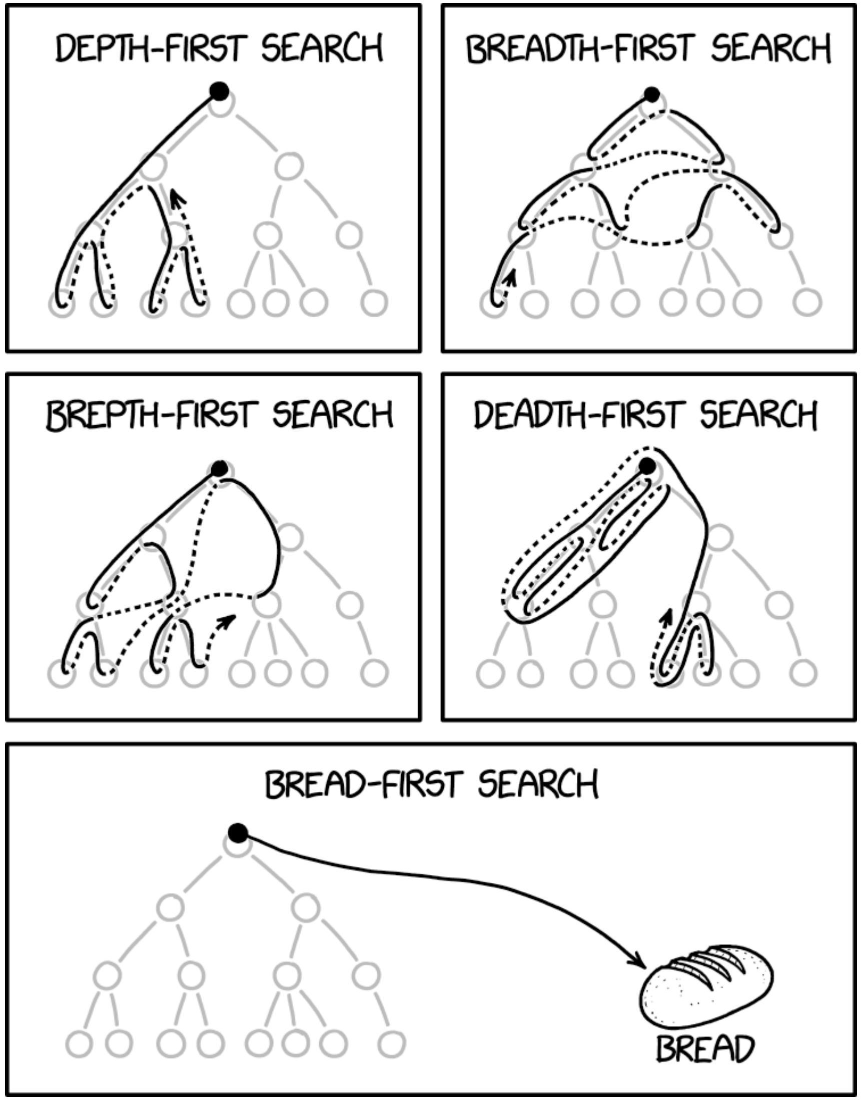
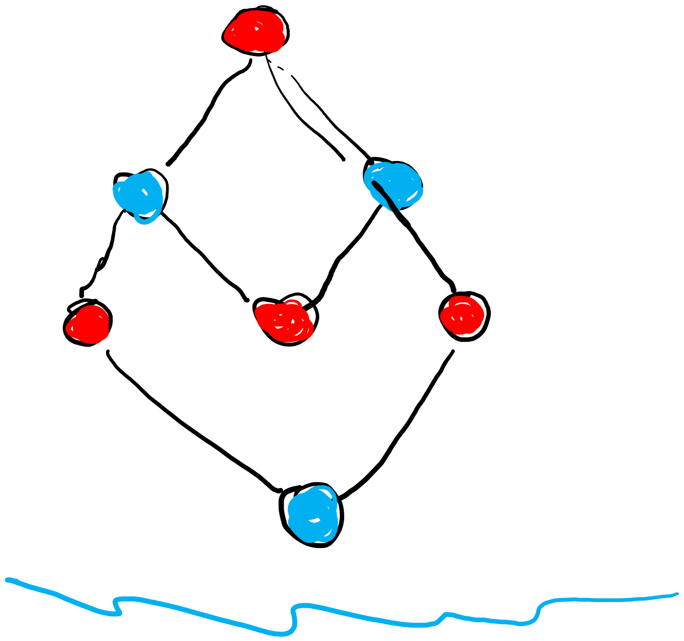


xkcd.com/2407



BFS and DFS

CSE 417 Winter 21
Lecture 5

Announcements

Fill out your breakout room preferences on Canvas.

If you found the proof by contradiction in lecture 3 really hard, you're not alone! That was possibly the hardest proof by contradiction we'll see this quarter. If you feel ok doing the proof by contradiction on the homework, you're ready to keep going in the course.

Hints/Common questions on HW1:

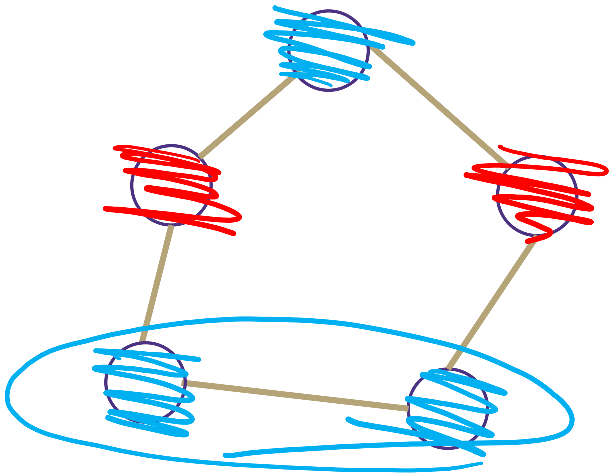
Problem 5: To prove "if p then q" by contradiction, you would start with "Suppose p and not q" – an implication is false when the promise is broken.

Problem 7: The family we showed you in lecture 3 is a good example of how to format a description of a family. It's not good inspiration for building a family with many SMs. A better idea is to combine many small "sub-instances", not build a big, intricate, interrelated one.

Problem 8: in part (a) you should imagine you have access to an executable file for Gale-Shapley. You cannot alter Gale=Shapley, you have to use it as a "black box" You can decide what to feed into it, and modify the output it give you.

Lemma 1

If a graph contains an odd cycle, then it is not bipartite.



Start from any vertex, and give it either color.
Its neighbors **must** be the other color.
Their neighbors must be the first color
…
The last two vertices (which are adjacent) must be the same color.
Uh-oh.

Layers

Can we have an edge that goes from layer i to layer $i+2$ (or lower)?

No! If u is in layer i, then we processed its edge while building layer $i+1$, so the neighbor is no lower than layer i.

Can you have an edge within a layer?

Yes! If u and v are neighbors and both have a neighbor in layer i, they both end up in layer $i+1$ (from their other neighbor) before the edge between them can be processed.

BFS With Layers

```
search(graph)
    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)
```

It’s just BFS!

With some extra bells and whistles.

Testing Bipartiteness

How can we use BFS with layers to check if a graph is 2-colorable?

Well, neighbors have to be “the other color”

Where are your neighbors?

Hopefully in the next layer or previous layer…

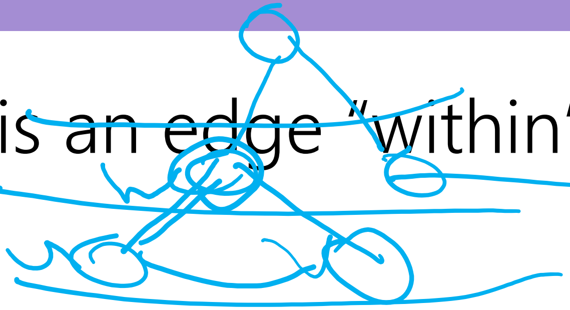
Color all the odd layers red and even layers blue.

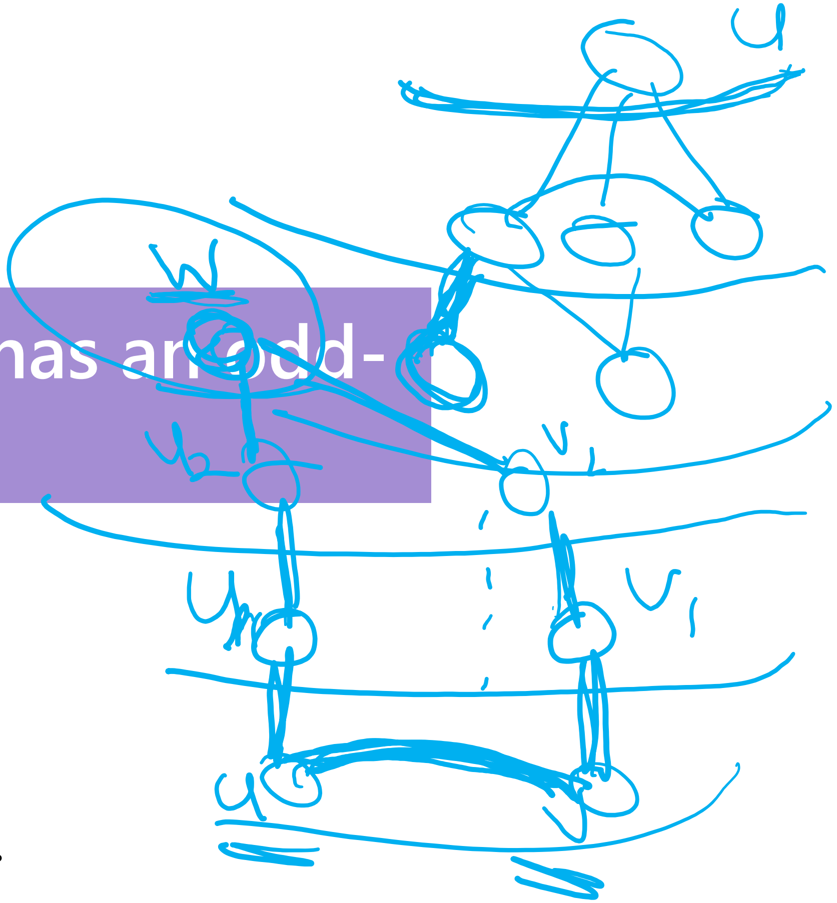
Does this work?

Lemma 2

If BFS has an intra-layer edge, then the graph has an odd-length cycle.

An “intra-layer” edge is an edge “within” a layer.





Follow the “predecessors” back up, layer by layer.

Eventually we end up with the two vertices having the same predecessor in some level (when you hit layer 1, there’s only one vertex)

Since we had two vertices per layer until we found the common vertex, we have $2k + 1$ vertices – that’s an odd number!

Lemma 3

'if p then q'

'if not q then not p'

If a graph has no odd-length cycles, then it is bipartite.

Prove it by **contrapositive**

The contrapositive implication is "the same one" prove that instead!

We want to show "if a graph is not bipartite, then it has an odd-length cycle.

Suppose G is not bipartite. Then the coloring attempt by BFS-coloring must fail.

Edges between layers can't cause failure – there must be an intra-level edge causing failure. By Lemma 2, we have an odd cycle.

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.

The Big Result

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
```

Testing Bipartiteness

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.

On Graph G
return
"yes, bipartite"

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 vs. BFS

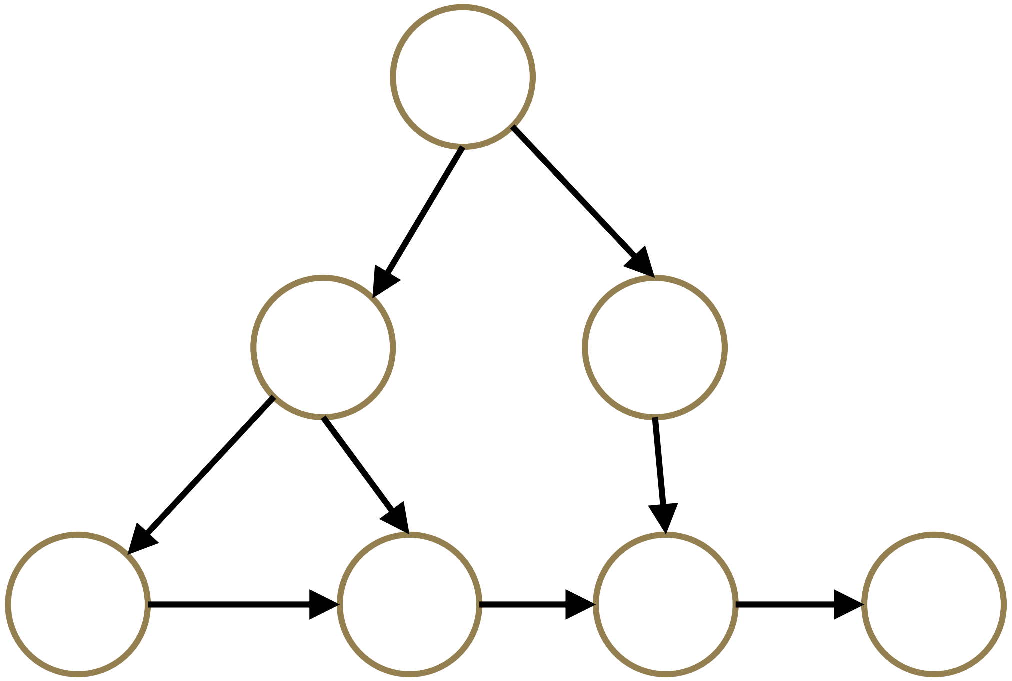
In BFS, we explored a graph "level-wise"

We explored everything next to the starting vertex.

Then we explored everything one step further away.

Then everything one step further

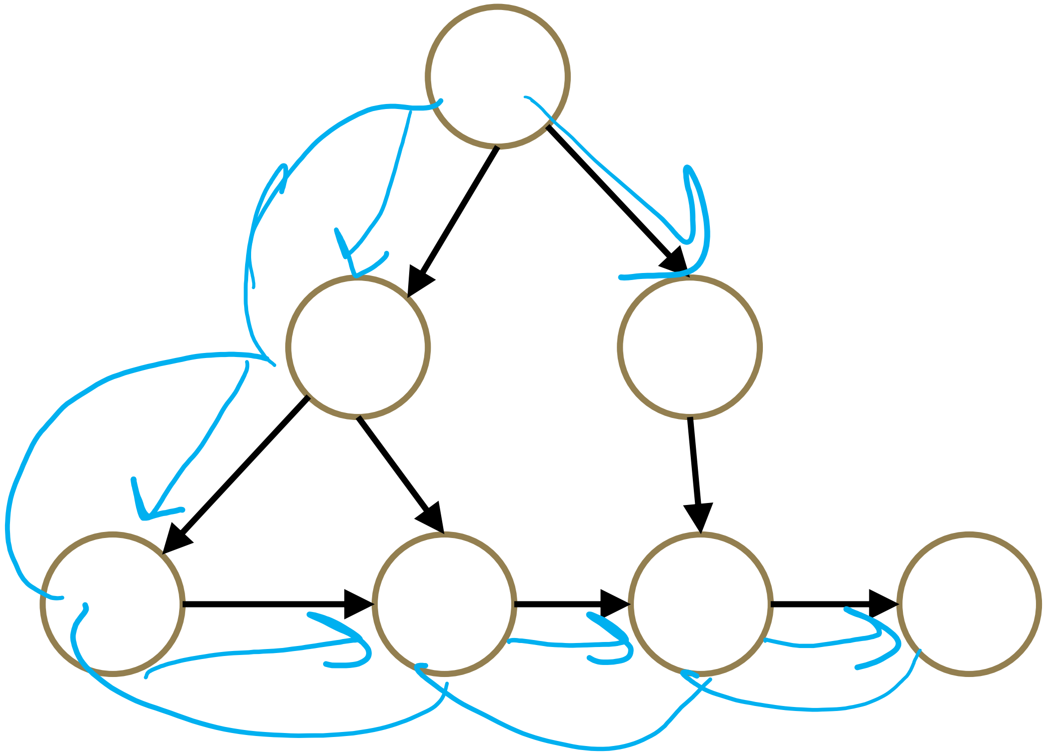
…



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

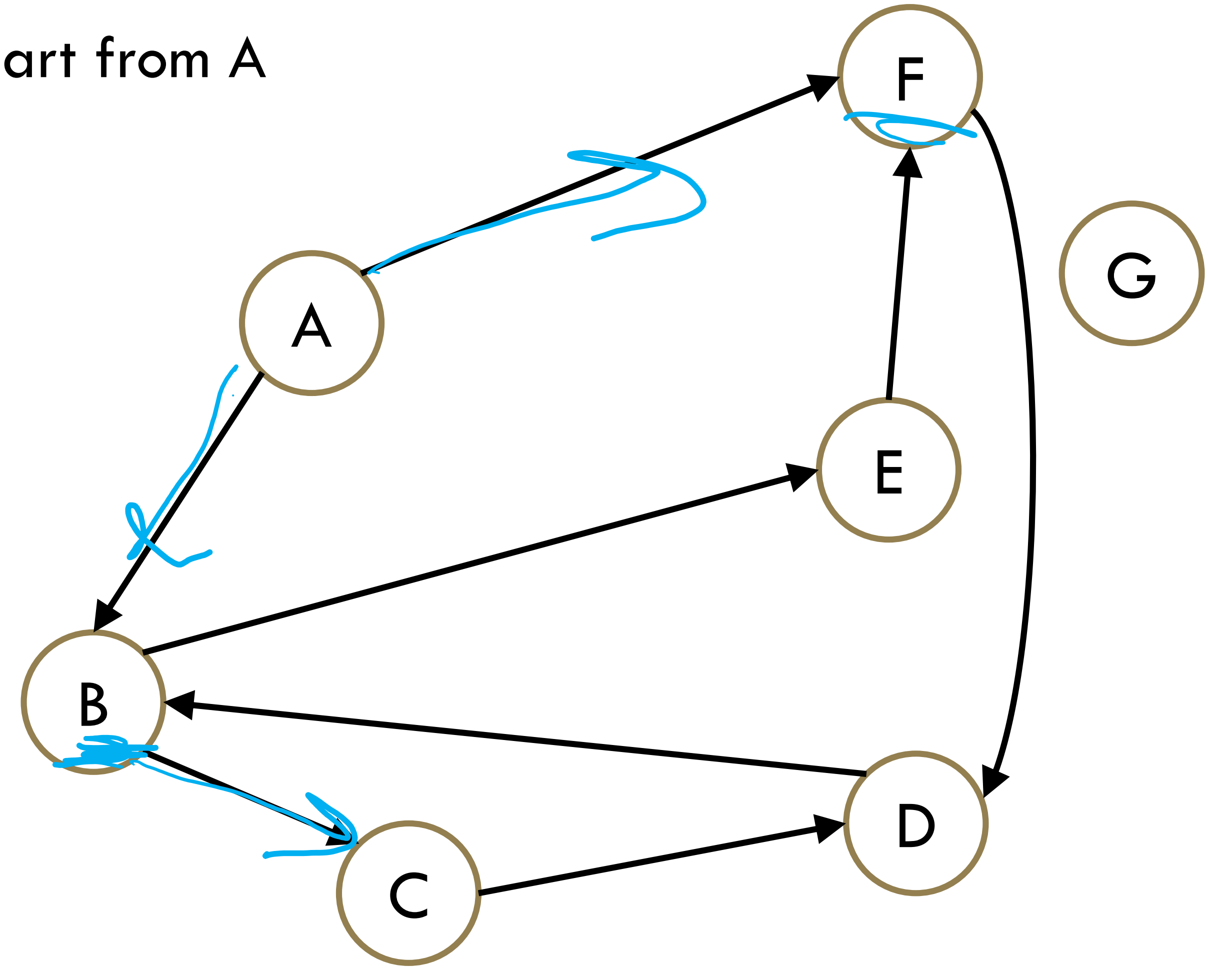
Both the explicit stack version and the recursive version "are" DFS.

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

Start from A



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

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

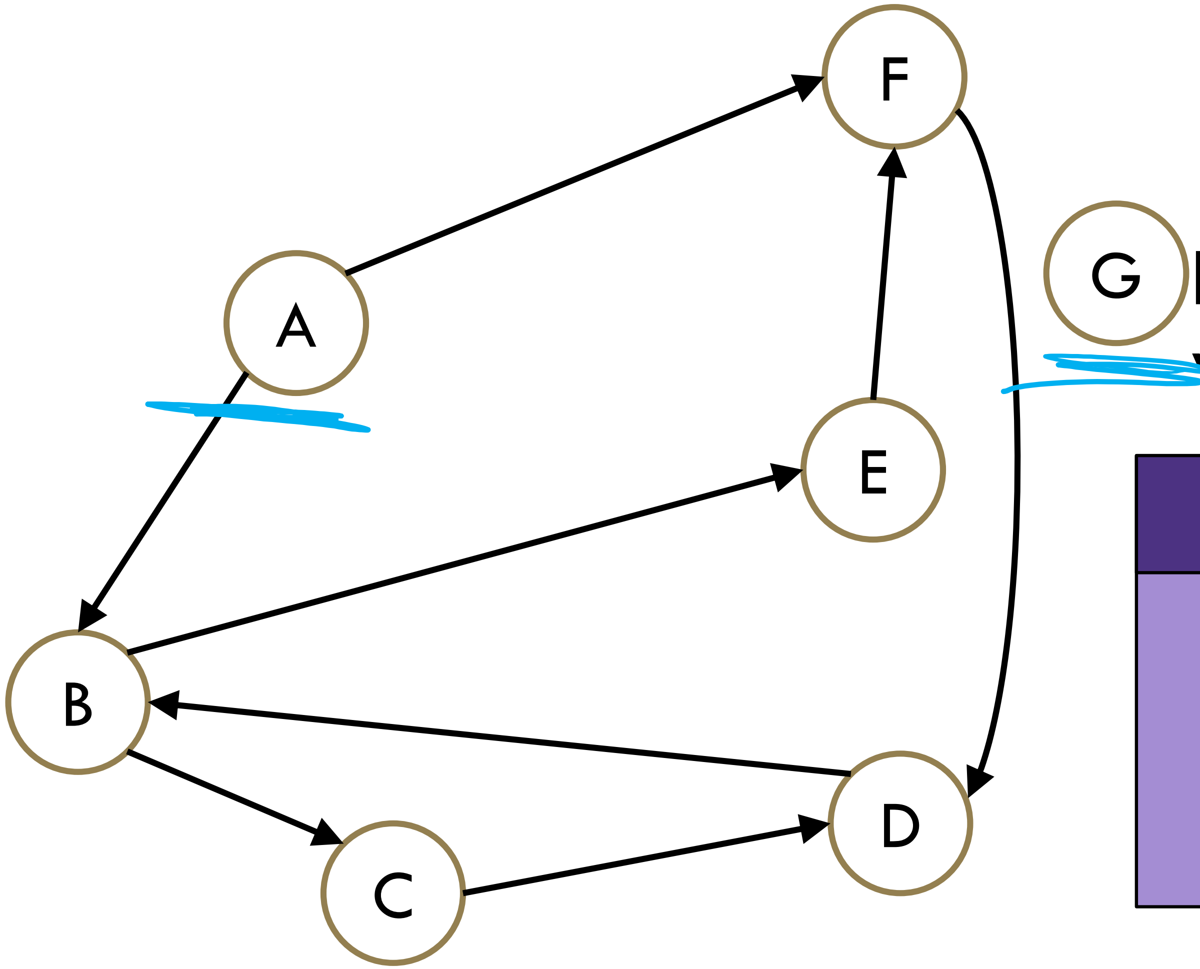
Vertex: E
Last edge used: (E,F)

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

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

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.

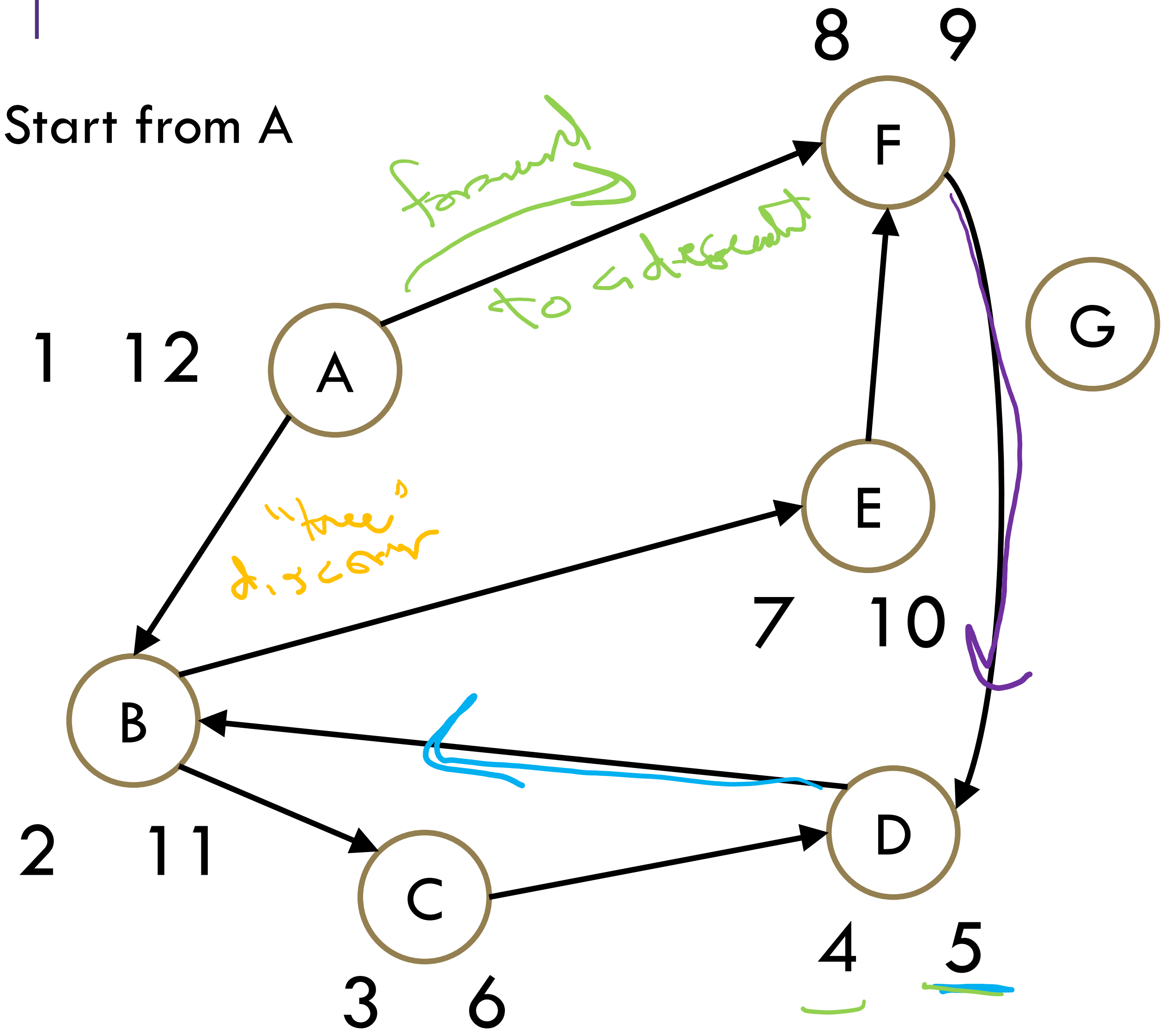
Then do an application of the classification.

Our goal: find a cycle in a directed graph.

Running DFS

```
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++
```

Start from A

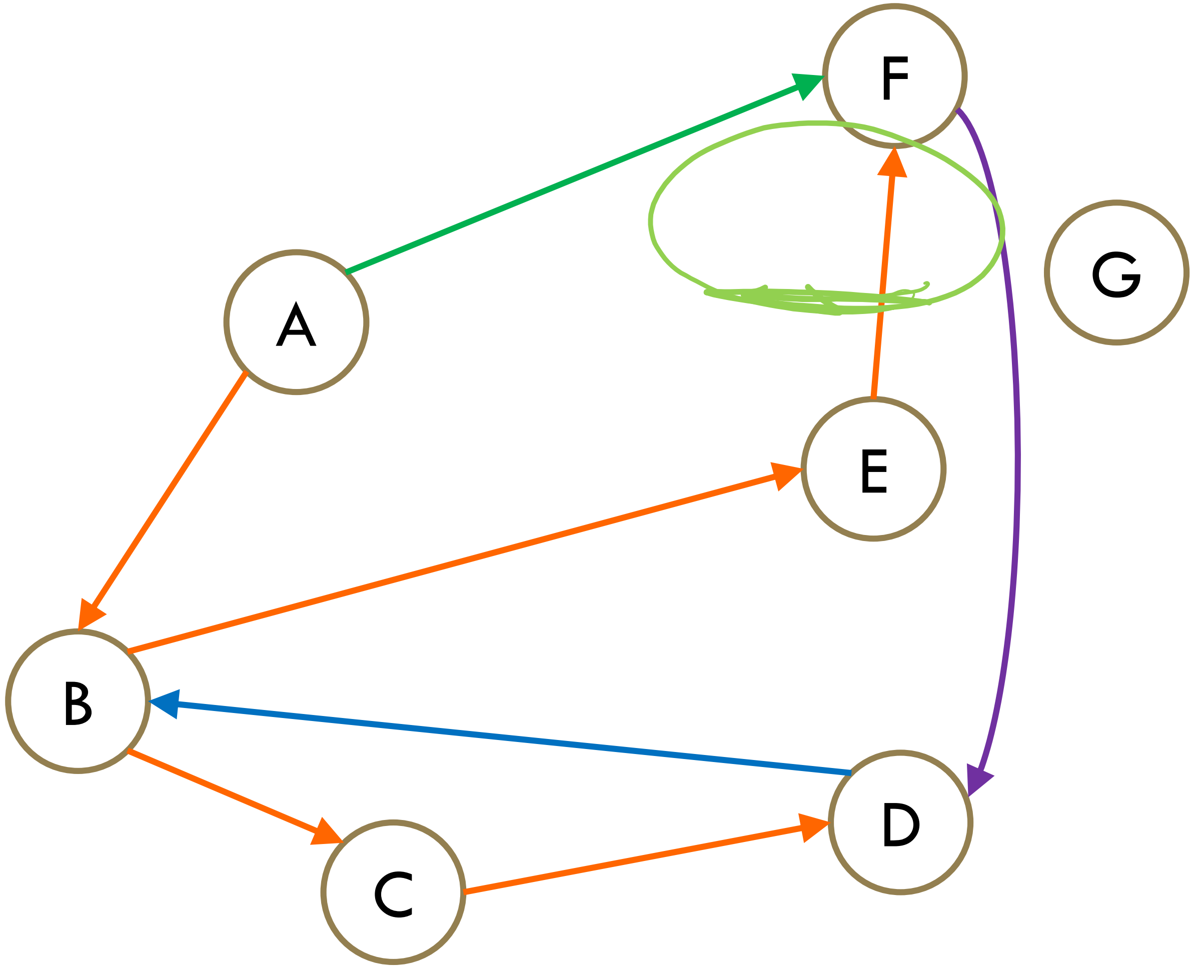


Vertex: F
Last edge used: (F,D)

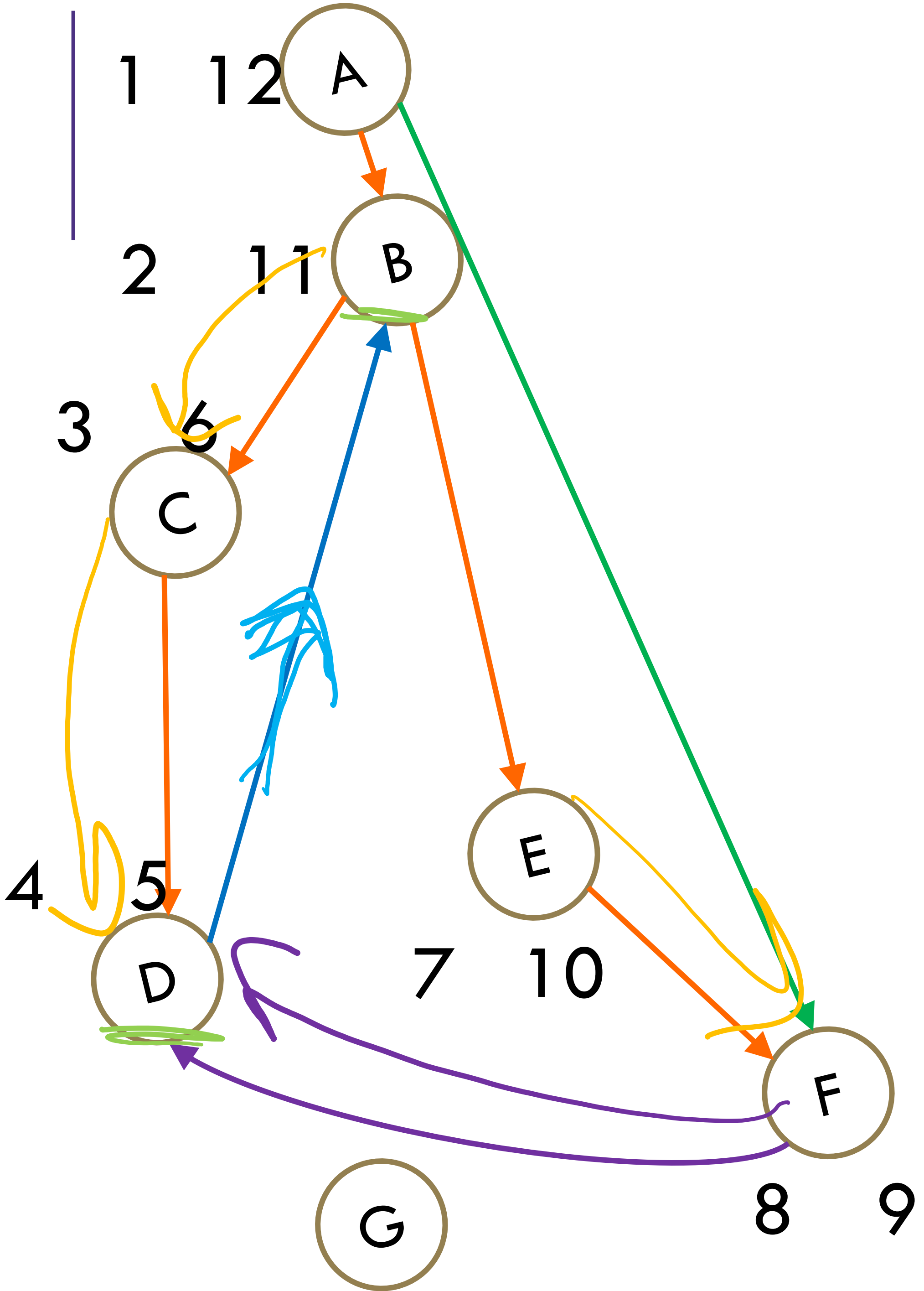
Vertex: E
Last edge used: (E,F)

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

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



```
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 $u \rightarrow v$	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:

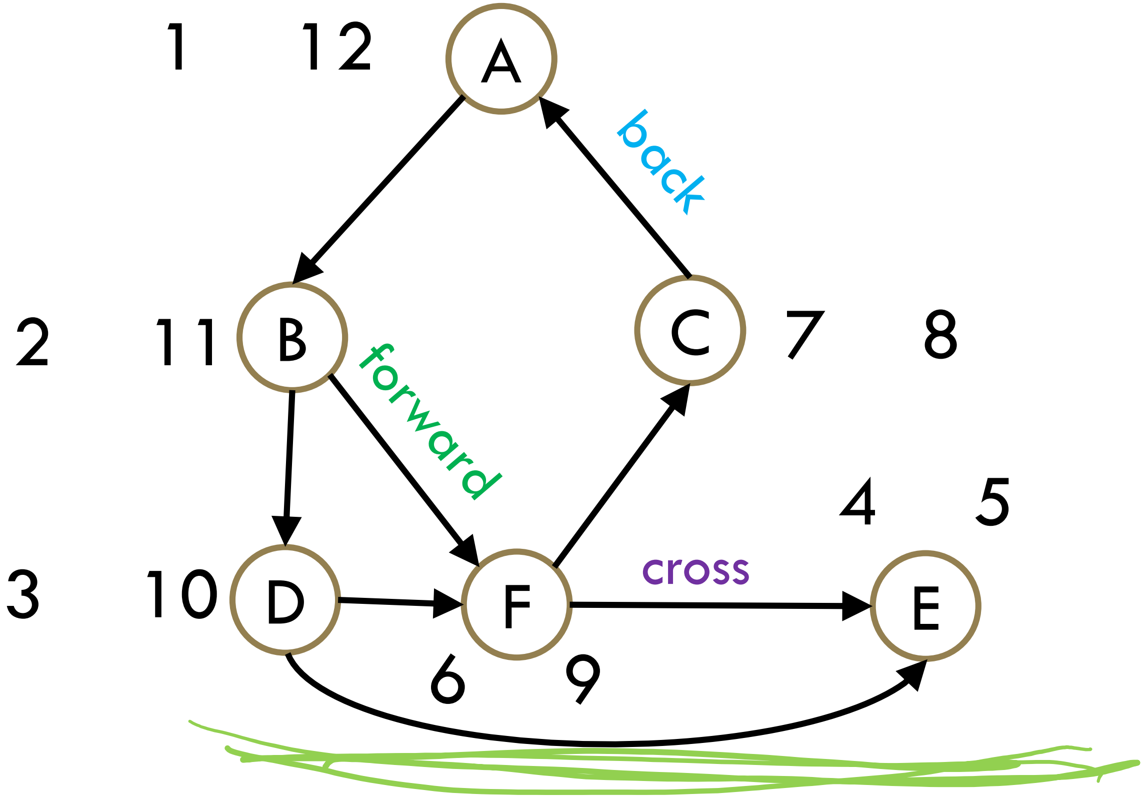
e.g. `u.start < v.start < u.end < v.end` is impossible.

And the rules of the algorithm eliminate some other possibilities.

Try it Yourselves!

```
DFSWrapper(G)
    counter = 0
    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.

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 to v

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 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.

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.